

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования

«Ярославский государственный университет им. П.Г.Демидова»
(ЯрГУ)

Кафедра компьютерной безопасности и
математических методов обработки информации

Курсовая работа

Сравнение производительности web-серверов, написанных на платформах Node.js и .NET

Научный руководитель
д. ф.-м. н., старший преподаватель

_____ О.В.Власова

«__» _____ 2016 г.

Студент группы КБ-41

_____ С.А.Попов

«__» _____ 2016 г.

Ярославль 2016

Содержание

Введение	3
1. Описание платформ .NET и Node.js	4
1.1. Платформа .NET	4
1.2 Асинхронность в .NET	4
1.3 Node.js	6
1.4 Асинхронность в Node.js	6
2. Протоколы передачи данных	8
2.1 TCP.....	8
2.2 UDP.....	9
2.3 HTTP	10
3. Сокеты	10
3.1 Потокные сокеты	11
3.2 Алгоритм работы потокового сокета	12
3.3 Дейтаграммные сокеты	14
3.4 Алгоритм работы дейтаграммного сокета	14
4. Сравнение производительности	15
4.1 Сравнение производительности .NET и Node.js сокетов для протокола TCP	15
4.2. Сравнение производительности .NET и Node.js сокетов для протокола UDP	19
4.3 Сравнение многопоточного и однопоточного HTTP-сервера на Node.js	22
4.4 Сравнение Node.js и .NET HTTP-серверов в многопоточном режиме	25
4.5 Сравнение Node.js и .NET HTTP-серверов запущенных под IIS	28
Список литературы	32
Приложение А.....	33
Приложение Б.....	34
Приложение В.....	35
Приложение Г	36
Приложение Д	37
Приложение Е	39
Приложение Ж.....	41
Приложение И	44
Приложение К.....	47
Приложение Л.....	50

Введение

В последнее время всё больше и больше разработчиков смотрят в сторону недавно появившейся программной платформы Node.js, главной особенностью которой является событийно-асинхронная архитектура и возможность писать на javascript. Она вызвала небывалый ажиотаж в среде разработчиков, что даже многие стали отказываться от разработки сервисных приложений на тяжеловесной платформе .NET, в пользу Node.js.

Node.js часто хвалят за то, что она отвечает запросам корпораций и позволяет собирать приложения с API, которое может обращаться к серверной части и большим объемам данных в лёгкой и эффективной манере. Действительно, концентрация на переиспользуемом RESTful API как более гибком способе построения архитектуры масштабных программных систем позволила Node.js найти своё место. Некоторые крупные компании вроде PayPal, Yahoo, IBM уже начали использовать возможности Node.js в своих проектах. Даже компания Samsung не так давно опубликовала информацию относительно того, что Node.js и JavaScript работают на низкопроизводительных системах лучше, чем любые другие платформы.

Node.js способен значительно сократить время разработки приложения, сохраняя при этом тот же функционал. Джон Оустерхаут, помогавший в разработке значимого скриптового языка и набора инструментов Tcl/Tk, ещё в 1990-х привёл аргумент о том, что скриптовые языки программирования по своей сути более продуктивны, нежели более тяжеловесные, такие, как C или C++.

По сравнению с более тяжёлыми стеками, такими как .NET, разработка приложений с помощью Node.js происходит быстрее, и с развитием экосистемы Node только ускоряется.

Цели данной работы:

- 1) Изучить принцип работы протоколов транспортного и прикладного уровней и особенности их реализации на платформах .NET и Node.js
- 2) Ознакомиться с асинхронными моделями платформ .NET и Node.js
- 3) Реализовать серверную часть программы для протоколов TCP, UDP, HTTP как на платформе .NET(на языке C#), так и на платформе Node.js(на языке javascript)
- 4) Написать клиентскую часть программы для отправки запросов на сервера на платформах .NET и Node.js
- 5) Выполнить сравнительный анализ производительности с разными настройками серверов на платформах .NET и на .Node.js

1. Описание платформ .NET и Node.js

1.1. Платформа .NET

.NET Framework — программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общезыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

Платформа .NET Framework состоит из общезыковой среды выполнения (среды CLR) и библиотеки классов .NET Framework. Основой платформы .NET Framework является среда CLR. Среда выполнения можно считать агентом, который управляет кодом во время выполнения и предоставляет основные службы, такие как управление памятью, управление потоками и удаленное взаимодействие. При этом накладываются условия строгой типизации и другие виды проверки точности кода, обеспечивающие безопасность и надежность. Фактически основной задачей среды выполнения является управление кодом. Код, который обращается к среде выполнения, называют управляемым кодом, а код, который не обращается к среде выполнения, называют неуправляемым кодом. Библиотека классов является комплексной объектно-ориентированной коллекцией допускающих повторное использование типов, которые применяются для разработки приложений — начиная с обычных приложений, запускаемых из командной строки, и приложений с графическим интерфейсом пользователя (GUI), и заканчивая приложениями, использующими последние технологические возможности ASP.NET, такие как Web Forms и веб-службы XML.

Платформа .NET Framework может размещаться неуправляемыми компонентами, которые загружают среду CLR в собственные процессы и запускают выполнение управляемого кода, создавая таким образом программную среду, позволяющую использовать средства как управляемого, так и неуправляемого выполнения. Платформа .NET Framework не только предоставляет несколько базовых сред выполнения, но также поддерживает разработку базовых сред выполнения независимыми производителями.

Например, ASP.NET размещает среду выполнения и обеспечивает масштабируемую среду для управляемого кода на стороне сервера. ASP.NET работает непосредственно со средой выполнения, чтобы обеспечить выполнение приложений ASP.NET и веб-служб XML.

1.2 Асинхронность в .NET

По мере развития .NET Framework было много нововведений и подходов для запуска асинхронных операций. Первым решением для асинхронных задач стал подход под названием APM (Asynchronous Programming Model). Он основан на асинхронных делегатах, которые используют пару методов с именами BeginOperationName и EndOperationName, которые соответственно начинают и завершают асинхронную операцию OperationName. После вызова метода BeginOperationName приложение может продолжить выполнение инструкций в вызывающем потоке, пока асинхронная операция выполняется в другом. Для каждого вызова метода BeginOperationName в приложении также должен присутствовать вызов метода EndOperationName, чтобы получить результаты операции.

Данный подход можно встретить во множестве технологий и классов, но он чреват усложнением и избыточностью кода.

В версии 2.0 была введена новая модель под названием EAP (Event-based Asynchronous Pattern). Класс, поддерживающий асинхронную модель, основанную на событиях, может содержать один или несколько методов `MethodNameAsync`. Он может отражать синхронные версии, которые выполняют то же действие с текущим потоком. Также в этом классе может содержаться событие `MethodNameCompleted` и метод `MethodNameAsyncCancel` (или просто `CancelAsync`) для отмены операции. Данный подход распространен при работе с сервисами. Но из-за длинных цепочек таких методов, очень сложно обрабатывать исключения, так как стек исключения, передаётся от одной функции к другой.

Вероятно, самым главным среди новых средств, внедренных в версию 4.0 среды .NET Framework, является библиотека распараллеливания задач -TPL(Task Parallel Library). Эта библиотека усовершенствовала многопоточное программирование двумя основными способами. Во-первых, она упрощает создание и применение многих потоков. И во-вторых, она позволяет автоматически использовать несколько процессоров. Иными словами, TPL открывает возможности для автоматического масштабирования приложений с целью эффективного использования ряда доступных процессоров. Благодаря этим двум особенностям библиотеки TPL она рекомендуется в большинстве случаев к применению для организации многопоточной обработки.

Применяя TPL, параллелизм в программу можно ввести двумя основными способами. Первый из них называется параллелизмом данных. При таком подходе одна операция над совокупностью данных разбивается на два параллельно выполняемых потока или больше, в каждом из которых обрабатывается часть данных. Так, если изменяется каждый элемент массива, то, применяя параллелизм данных, можно организовать параллельную обработку разных областей массива в двух или больше потоках. Нетрудно догадаться, что такие параллельно выполняющиеся действия могут привести к значительному ускорению обработки данных по сравнению с последовательным подходом.

Несмотря на то что параллелизм данных был всегда возможен и с помощью класса `Thread`, построение масштабируемых решений средствами этого класса требовало немало усилий и времени. Это положение изменилось с появлением библиотеки TPL, с помощью которой масштабируемый параллелизм данных без особого труда вводится в программу.

Второй способ ввода параллелизма называется параллелизмом задач. При таком подходе две операции или больше выполняются параллельно. Следовательно, параллелизм задач представляет собой разновидность параллелизма, который достигался в прошлом средствами класса `Thread`. А к преимуществам, которые сулит применение TPL, относится простота применения и возможность автоматически масштабировать исполнение кода на несколько процессоров.

Также эта библиотека позволяет автоматически распределять нагрузку приложений между доступными процессорами в динамическом режиме, используя пул потоков CLR. Библиотека TPL занимается распределением работы, планированием потоков, управлением состоянием и прочими низкоуровневыми деталями. В результате появляется возможность максимизировать производительность приложений .NET, не имея дела со сложностями прямой работы с потоками.

В последних версиях платформы .NET появились новые возможности на основе все тех же задач, которые упрощают написание асинхронного кода и делают его более

читабельным и понятным. Для этого введены новые ключевые слова "async" и "await", которыми помечаются асинхронные методы и их вызовы. Асинхронный код становится очень похожим на синхронный: мы просто вызываем нужную операцию и весь код, который следует за ее вызовом, автоматически будет завернут в функцию обратного вызова, которая вызовется после завершения асинхронной операции. Также данный подход позволяет обрабатывать исключения в синхронной манере; явно дожидаться завершения операции; определять действия, которые должны быть выполнены, и соответствующие условия.

1.3 Node.js

Node.js - это программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API (написанный на C++), подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль веб-сервера, но есть возможность разрабатывать на Node.js и десктопные оконные приложения и даже программировать микроконтроллеры. Основной упор в Node.js делается на создании высокопроизводительных, хорошо масштабируемых клиентских и серверных приложений для «веб реального времени».

Эту платформу разработал Райан Дал (RyanDahl) в 2009 году, после двух лет экспериментирования с созданием серверных веб-компонентов на Ruby и других языках. В ходе своих исследований он пришел к выводу, что вместо традиционной модели параллелизма на основе потоков следует обратиться к событийно-ориентированным системам. Данная модель принципиально отличается от распространенных платформ для построения серверов приложений, в которых масштабируемость достигается за счет многопоточности. Утверждается, что благодаря событийно-ориентированной архитектуре снижается потребление памяти, повышается пропускная способность и упрощается модель программирования. Сейчас платформа Node быстро развивается, и многие считают ее привлекательной альтернативой традиционному подходу к разработке веб-приложений - на базе Apache, PHP, Python и т. п.

В основе Node.js лежит автономная виртуальная машина JavaScript с расширениями, делающими ее пригодной для программирования общего назначения с упором на разработку серверов приложений. В основе реализации лежит цикл обработки событий неблокирующего ввода/вывода и библиотеки файлового и сетевого ввода/вывода, причем все это построено поверх движка V8 JavaScript (заимствованного из веб-браузера Chrome). Библиотека ввода/вывода обладает достаточной общностью для реализации любого протокола на базе TCP или UDP: DNS, HTTP, IRC, FTP и др. Но хотя она поддерживает разработку серверов и клиентов произвольного протокола, чаще всего применяется для создания обычных веб-сайтов.

1.4 Асинхронность в Node.js

В Node.js принята однопоточная событийно-ориентированная архитектура. Именно благодаря такой архитектуре Node.js демонстрирует столь высокую производительность. Так и есть, только к этому надо добавить еще стремительность движка V8 JavaScript. В традиционной модели сервера приложений параллелизм обеспечивается за счет

использования блокирующего ввода/вывода и нескольких потоков. Каждый поток должен дожидаться завершения ввода/вывода, перед тем как приступить к обработке следующего запроса.

В Node.js имеется единственный поток выполнения, без какого-либо контекстного переключения или ожидания ввода/вывода. При любом запросе ввода/вывода задаются функции обработки, которые впоследствии вызываются из цикла обработки событий, когда станут доступны данные или произойдет еще что-то значимое. Модель цикла обработки событий и обработчика событий – вещь распространённая, именно так исполняются написанные на JavaScript скрипты в браузере. Ожидается, что программа быстро вернет управление циклу обработки, чтобы можно было вызвать следующее стоящее в очереди задание.

Чтобы понять основную концепцию Node.js, можно рассмотреть следующий пример:

```
result = query('SELECT * from db');
```

Ожидается что при исполнении этой строчки кода, программа в этой точке приостанавливается на время, пока слой доступа к базе данных отправляет запрос базе, которая вычисляет результат и возвращает данные. В зависимости от сложности запроса его выполнение может занять весьма заметное время. Это расточительно, потому что пока поток простаивает, может прийти другой запрос, а если заняты все потоки, то запрос будет просто отброшен. Да и контекстное переключение обходится не бесплатно; чем больше запущено потоков, тем больше времени процессор тратит на сохранение и восстановление их состояния. Кроме того, стек каждого потока занимает место в памяти. И просто за счет асинхронного событийно-ориентированного ввода/вывода Node устраняет большую часть этих накладных расходов, привнося совсем немного собственных.

Часто рассказ о реализации параллелизма с помощью потоков сопровождается предостережениями типа «дорого и чревато ошибками», или «проектирование параллельных программ может оказаться сложным, и не исключены ошибки» (фразы взяты из результатов, выданных поисковой системой). Причиной этой сложности являются доступ к разделяемым переменным и различные стратегии предотвращения взаимоблокировок и состязаний между потоками. Node призывает подходить к параллелизму по-другому. Обратные вызовы из цикла обработки событий - гораздо более простая модель параллелизма, как для понимания, так и для реализации.

В Node.js вышеупомянутый запрос следовало бы записать так:

```
query( 'SELECT * from db', function (result) {  
    // произвести какие-то операции с результатом  
}):
```

Разница в том, что теперь результат запроса не возвращается в качестве значения функции, а передается функции обратного вызова, которая будет вызвана позже. Таким образом, возврат в цикл обработки событий происходит почти сразу, и сервер может перейти к обслуживанию других запросов. Одним из таких запросов будет ответ на запрос, отправленный базе данных, и тогда будет вызвана функция обратного вызова. Такая модель быстрого возврата в цикл обработки событий повышает степень использования ресурсов серверов.

2. Протоколы передачи данных

URI - унифицированный (единообразный) идентификатор ресурса, символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д.

Пакет - это определённым образом оформленный блок данных, передаваемый по сети в пакетном режиме.

Заголовок - набор инструкций, касающийся данных, которые содержатся в пакете. В состав таких инструкций могут входить:

- Длина пакета (в некоторых сетях используют пакеты фиксированной длины, тогда, как в других, информация о длине содержится в заголовке);
- Данные синхронизации (несколько битов, обеспечивающих согласование пакета с сетью);
- Номер пакета (порядковый номер пакета в последовательности пакетов);
- Протокол (в сетях, поддерживающих разные типы информации, протокол указывает на то, пакет какого типа передается: электронное письмо, веб-страница или потоковое видео);
- Адрес назначения (обозначение места, куда пакет должен попасть);
- Адрес отправителя (обозначение места, из которого пакет был отправлен);

2.1 TCP

TCP (Transmission Control Protocol) - это ориентированный на соединение протокол, что означает необходимость «рукопожатия» для установки соединения между двумя хостами. Как только соединение установлено, пользователи могут отправлять данные в обоих направлениях.

TCP считается надёжным транспортным протоколом, а это значит, что он использует процессы, которые обеспечивают надёжную передачу данных между приложениями с помощью подтверждения доставки. Передача с использованием TCP аналогична отправке пакетов, которые отслеживаются от источника к получателю.

TCP использует следующие три основные операции для обеспечения надёжности:

- 1) отслеживание переданных сегментов данных
- 2) подтверждение полученных данных
- 3) повторная отправка всех неподтвержденных данных

Также данный протокол 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.

Далее сервер должен отправить клиенту подтверждение (ACK) вместе с порядковым номером (SYN) сервера. В свою очередь клиент также отвечает ACK, и соединение устанавливается. После этого может начаться процесс отправки и получения сообщений.

TCP разбивает сообщение на фрагменты меньшего размера, которые называются сегментами. Этим сегментам присваиваются порядковые номера, после чего они передаются IP-протоколу, который собирает их в пакеты. TCP отслеживает количество сегментов, отправленных на тот или иной узел тем или иным приложением. Если отправитель не получает подтверждения в течение определённого периода времени, то TCP рассматривает эти сегменты как утраченные и повторяет их отправку. Повторно отправляется только утраченная часть сообщения, а не все сообщение целиком. Протокол TCP на принимающем узле отвечает за повторную сборку сегментов сообщений и их передачу соответствующему приложению. Протокол передачи файлов (FTP) и протокол передачи гипертекста (HTTP) — это примеры приложений, которые используют TCP для доставки данных.

2.2 UDP

В то время как функции надёжности TCP обеспечивают более стабильное взаимодействие между приложениями, они также потребляют больше ресурсов и могут стать причиной задержек при передаче данных. Существует некий компромисс между надёжностью и той нагрузкой, которую она представляет для сетевых ресурсов. Дополнительная нагрузка для обеспечения надёжности некоторых приложений может снизить полезность самого приложения и даже отрицательно сказаться на его производительности. В таких случаях использование протокола UDP более предпочтительно.

UDP (User Datagram Protocol) - это более простой, основанный на сообщениях протокол без установления соединения. Он обеспечивает только основные функции для отправки сегментов данных между соответствующими приложениями, при этом незначительно используя ресурсы и проверку данных. Протоколы такого типа не устанавливают выделенного соединения между двумя хостами. Связь достигается путём передачи информации в одном направлении от источника к получателю без проверки готовности или состояния получателя.

Поэтому принцип его работы основан на негарантированной доставке и такая передача данных считается ненадёжной, поскольку при этом отсутствует подтверждение о получении отправленных данных на узле назначения. Соответственно протокол также не задействует процессы транспортного уровня, которые сообщают отправителю об успешной доставке данных.

Протокол UDP подобен тому, как если бы по почте отправляли обычное незарегистрированное письмо. Отправитель не знает, сможет ли адресат получить письмо, а почтовое отделение не несёт ответственности за отслеживание письма или информирование отправителя о том, доставлено ли письмо по адресу.

На платформах .NET и Node.js взаимодействия по протоколам TCP и UDP происходят с помощью сокетов. В среде .NET классы для работы данным протоколам находятся в пространстве имён "System.NET.Sockets". Что касается платформы Node.js, то для работы с TCP соединением используется пространство имён "net".

2.3 HTTP

HTTP - протокол прикладного уровня передачи данных (изначально — в виде гипертекстовых документов в формате HTML, в настоящий момент используется для передачи произвольных данных). Основой HTTP является технология «клиент-сервер», то есть предполагается существование потребителей (клиентов), которые иницируют соединение и посылают запрос, и поставщиков (серверов), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.

HTTP в настоящее время повсеместно используется во Всемирной паутине для получения информации с веб-сайтов. В 2006 году в Северной Америке доля HTTP-трафика превысила долю P2P-сетей и составила 46 %, из которых почти половина — это передача потокового видео и звука.

Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т. д. (В частности для этого используется HTTP-заголовок.) Именно благодаря возможности указания способа кодирования сообщения клиент и сервер могут обмениваться двоичными данными, хотя данный протокол является текстовым.

Протокол HTTP работает поверх транспортного протокола TCP. Таким образом для передачи сообщения между клиентом и сервером, также необходимо сначала установить соединение между ними. Если проводить эксперимент с передачей данных от клиента к серверу и обратно, то результат будет аналогичен эксперименту с протоколом TCP. Разница будет заключаться лишь в том, что и клиент и сервер должны будут сначала раскодировать набор байтов, полученный от протокола TCP. Поэтому имеет смысл проверить Node.js и .NET другим, менее тривиальным способом.

3. Сокеты

Сокет — это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами (локальными или удаленными). Реализация сокетов обеспечивает инкапсуляцию протоколов сетевого и транспортного уровней.

Первоначально сокеты были разработаны для UNIX в Калифорнийском университете в Беркли. В UNIX обеспечивающий связь метод ввода-вывода следует алгоритму open/read/write/close. Прежде чем ресурс использовать, его нужно открыть, задав соответствующие разрешения и другие параметры. Как только ресурс открыт, из него можно считывать или в него записывать данные. После использования ресурса пользователь должен вызывать метод Close(), чтобы подать сигнал операционной системе о завершении его работы с этим ресурсом.

Когда в операционную систему UNIX были добавлены средства межпроцессного взаимодействия (Inter-Process Communication, IPC) и сетевого обмена, был заимствован привычный шаблон ввода-вывода. Все ресурсы, открытые для связи, в UNIX и Windows идентифицируются дескрипторами. Эти дескрипторы, или описатели (handles), могут

указывать на файл, память или какой-либо другой канал связи, а фактически указывают на внутреннюю структуру данных, используемую операционной системой. Сокет, будучи таким же ресурсом, тоже представляется дескриптором. Следовательно, для сокетов жизнь дескриптора можно разделить на три фазы: открыть (создать) сокет, получить из сокета или отправить сокету и в конце концов закрыть сокет.

Интерфейс IPC для взаимодействия между разными процессами построен поверх методов ввода-вывода. Они облегчают для сокетов отправку и получение данных. Каждый целевой объект задается адресом сокета, следовательно, этот адрес можно указать в клиенте, чтобы установить соединение с целью.

Сокет состоит из IP-адреса машины и номера порта, используемого приложением TSP. Поскольку IP-адрес уникален в Интернете, а номера портов уникальны на отдельной машине, номера сокетов также уникальны во всем Интернете. Эта характеристика позволяет процессу общаться через сеть с другим процессом исключительно на основании номера сокета.

За определенными службами номера портов зарезервированы — это широко известные номера портов, например порт 21, использующийся в FTP. Ваше приложение может пользоваться любым номером порта, который не был зарезервирован и пока не занят. Агентство Internet Assigned Numbers Authority (IANA) ведет перечень широко известных номеров портов.

Обычно приложение клиент-сервер, использующее сокеты, состоит из двух разных приложений - клиента, инициирующего соединение с целью (сервером), и сервера, ожидающего соединения от клиента.

Например, на стороне клиента, приложение должно знать адрес цели и номер порта. Отправляя запрос на соединение, клиент пытается установить соединение с сервером. Если события развиваются удачно, при условии что сервер запущен прежде, чем клиент попытался с ним соединиться, сервер соглашается на соединение. Дав согласие, серверное приложение создает новый сокет для взаимодействия именно с установившим соединение клиентом. Теперь клиент и сервер могут взаимодействовать между собой, считывая сообщения каждый из своего сокета и, соответственно, записывая сообщения.

3.1 Потокосые сокеты

Потокосый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т. е. через эту конечную точку приложение может и передавать, и получать данные.

Потокосый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, дублированных данных. Потокосый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Потокосые сокеты достигают этого уровня качества за счет использования протокола (TCP). TCP обеспечивает поступление данных на другую сторону в нужной последовательности и без ошибок.

Для этого типа сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают. Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности.

Однако, отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании ТСП этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на уровень ТСП определенное число байтов и другая сторона получает эти байты. В свою очередь ТСП эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе.

Потоки базируются на явных соединениях: сокет А запрашивает соединение с сокетом В, а сокет В либо соглашается с запросом на установление соединения, либо отвергает его.

Если данные должны гарантированно доставляться другой стороне или размер их велик, потоковые сокеты предпочтительнее дейтаграммных. Следовательно, если надежность связи между двумя приложениями имеет первостепенное значение, выбирайте потоковые сокеты.

Сервер электронной почты представляет пример приложения, которое должно доставлять содержание в правильном порядке, без дублирования и пропусков. Поточковый сокет рассчитывает, что ТСП обеспечит доставку сообщений по их назначениям.

3.2 Алгоритм работы потокового сокета

Вне зависимости от платформы и языка программирования на котором разрабатывается клиент-серверное приложения, основанное на сокетах, алгоритм взаимодействия между клиентом и сервером остаётся неизменным. На рисунке 1 представлен наглядный алгоритм работы потокового сокета.

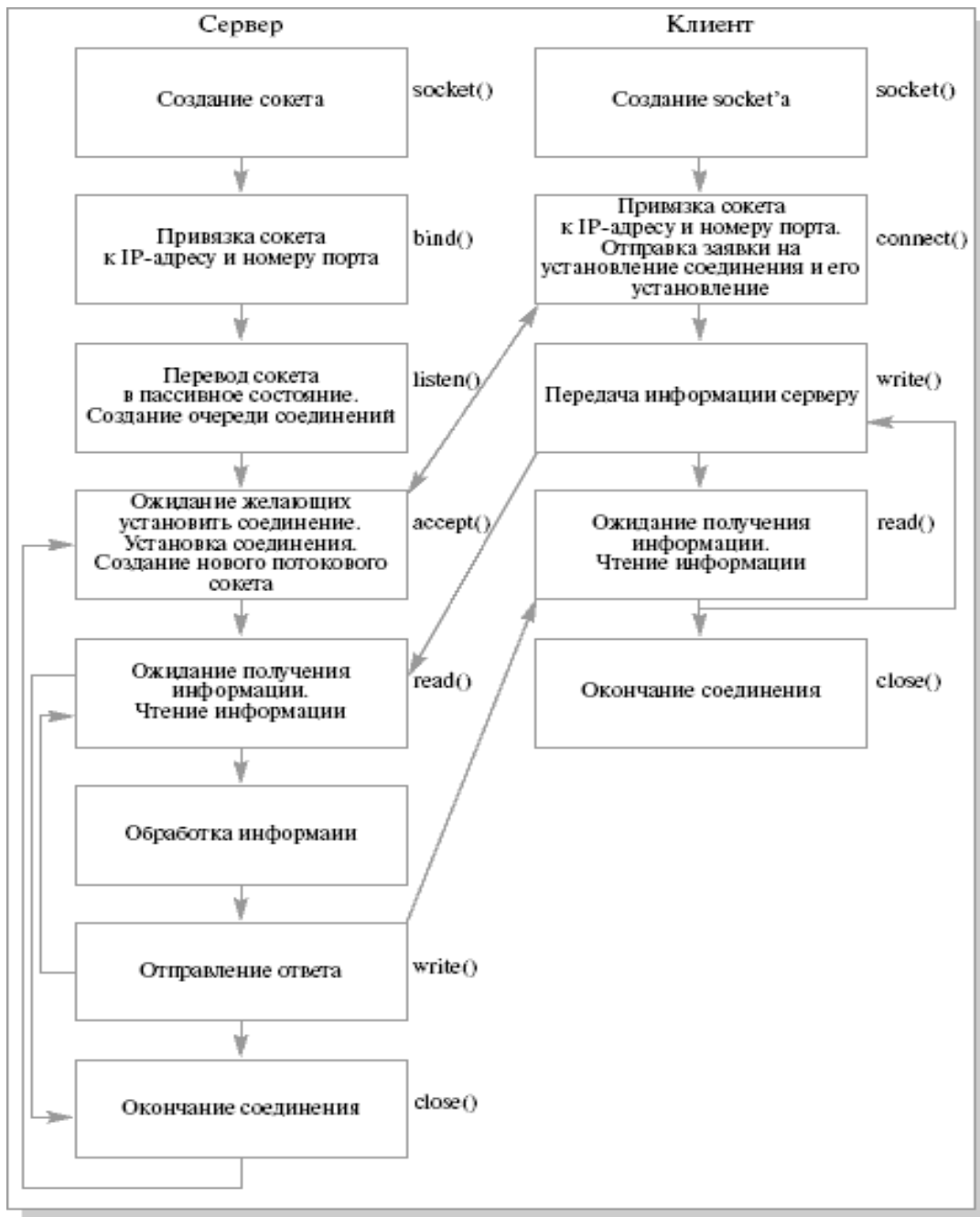


Рис. 1

Клиент :

- 1) создает сокет
- 2) подсоединяется к серверу, предоставляя адрес удаленного сокета (адрес Internet сервера и номер сервисного порта). Это соединение автоматически присваивает клиенту номер порта
- 3) осуществляет считывание или запись на сокет
- 4) закрывает сокет

Сервер:

- 1) создает сокет
- 2) привязывает сокет-адрес (ip адрес и номер порта)

- 3) переводит себя в состояние "прослушивания" входящих соединений для каждого входящего соединения
- 4) принимает соединение (создается новый сокет с теми же характеристиками, что и исходный)
 считывает и записывает на новый сокет
- 5) закрывает новый сокет

3.3 Дейтаграммные сокеты

Дейтаграммные сокеты иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается — сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Потоковые сокеты по сравнению с дейтаграммными действительно дают более надежный метод, но для некоторых приложений накладные расходы, связанные с установкой явного соединения, неприемлемы (например, сервер времени суток, обеспечивающий синхронизацию времени для своих клиентов). В конце концов на установление надежного соединения с сервером требуется время, которое просто вносит задержки в обслуживание, и задача серверного приложения не выполняется. Для сокращения накладных расходов нужно использовать дейтаграммные сокеты.

Использование дейтаграммных сокетов требует, чтобы передачей данных от клиента к серверу занимался UDP. В этом протоколе на размер сообщений налагаются некоторые ограничения, и в отличие от потоковых сокетов, умеющих надежно отправлять сообщения серверу-адресату, дейтаграммные сокеты надежность не обеспечивают. Если данные затерялись где-то в сети, сервер не сообщит об ошибках.

3.4 Алгоритм работы дейтаграммного сокета

Алгоритм работы дейтаграммного сокета отличается от алгоритма потокового сокета, лишь тем, что отсутствует начальное соединение между клиентом и сервером. На рисунке 2 представлен наглядный алгоритм работы дейтаграммного сокета.



Рис. 2

Клиент

- 1) создает сокет
- 2) связывает сокет-адрес с сервисной программой: "binding" (операция, являющаяся необходимой только в случае, если процесс должен получить данные)
- 3) считывает или осуществляет запись на сокет

Сервер

- 1) создает сокет
- 2) связывает сокет-адрес с сервисной программой: "binding" (операция необходима только в случае, если процесс должен получить данные)
- 3) считывает или осуществляет запись на сокет

4. Сравнение производительности

4.1 Сравнение производительности .NET и Node.js сокетов для протокола TCP

Для реализации TCP-сервера на платформе .NET необходимо привязать слушателя на указанный порт и адрес. Для простоты эксперимента адрес был выбран локальным - "127.0.0.1". Также для чистоты эксперимента все обработчики вызываются асинхронно. Задачей сервера является принимать байты, отправленные клиентом и как можно скорее отправлять их обратно. Клиент и сервер знают сколько байт должно быть получено и

отправлено. Как только количество оправленных байт становится равным заранее оговоренному количеству, сервер разрывает соединение с клиентом.

TCP-сервер на платформе Node.js работает по такому же принципу. Он читает данные из потока, с помощью многочисленных синхронных чтений из буфера, и записываем накопленный массив байт в поток ответа.

Суть эксперимента заключается в проверке скорости ответа серверов написанных на разных платформах. Обе реализации сокетов являются асинхронными, что не останавливает поток выполнения программы. Замерять скорость ответа будет клиент, написанный на платформе Node.js. Он посылает n-количество раз сообщение длиной 32768 и 16784 байт. Клиент не отправляет следующее сообщение, пока не получит ответное сообщение от сервера в таком же объёме. Варьируя количество запросов мы получаем функцию, зависящую от размера сообщения и количества запросов. На рисунке 3 и 4 приведены графики для размеров сообщения длиной 32768 и 16784 байт и количества запросов от 64 до 8192. Код TCP-клиента на Node.js приведён в приложении А. Код TCP-сервера для Node.js приведён в приложении Б. Код TCP-сервера для .NET приведён в приложении И.

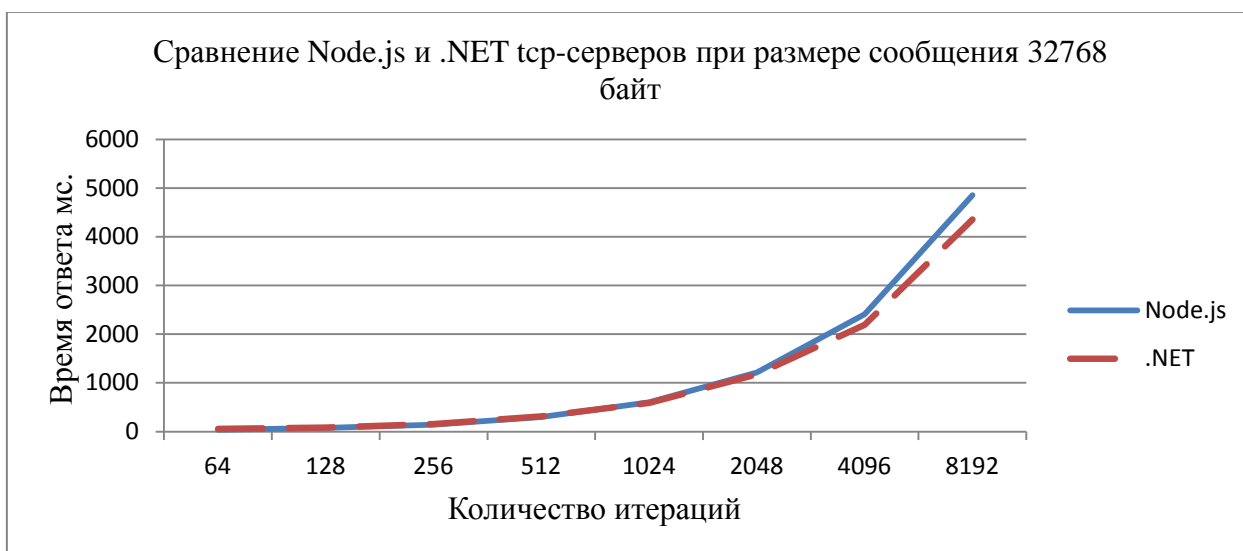


Рис. 3

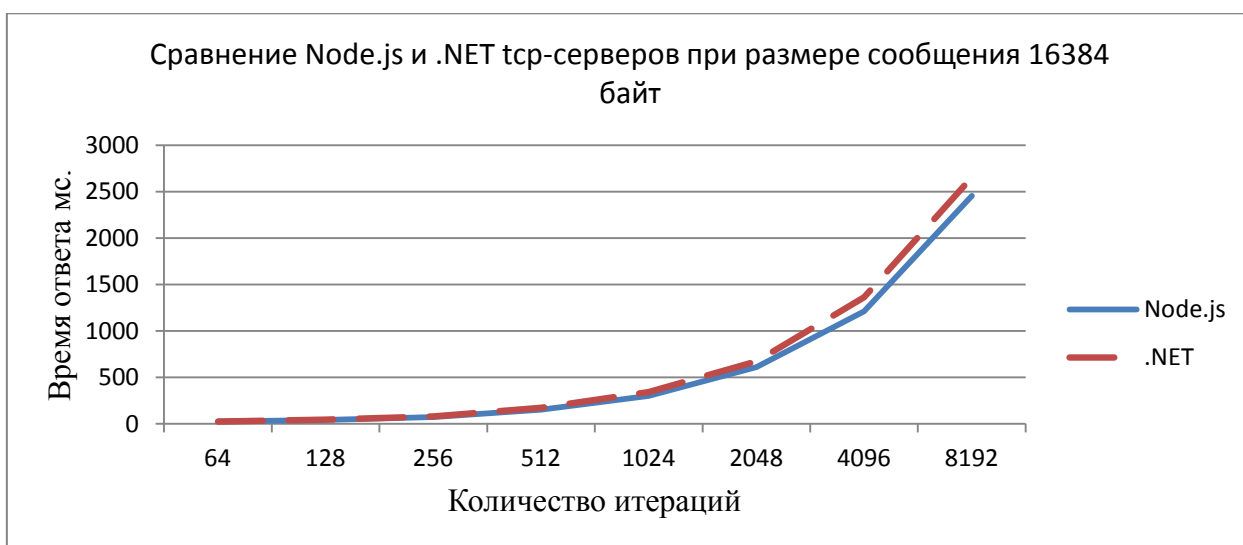
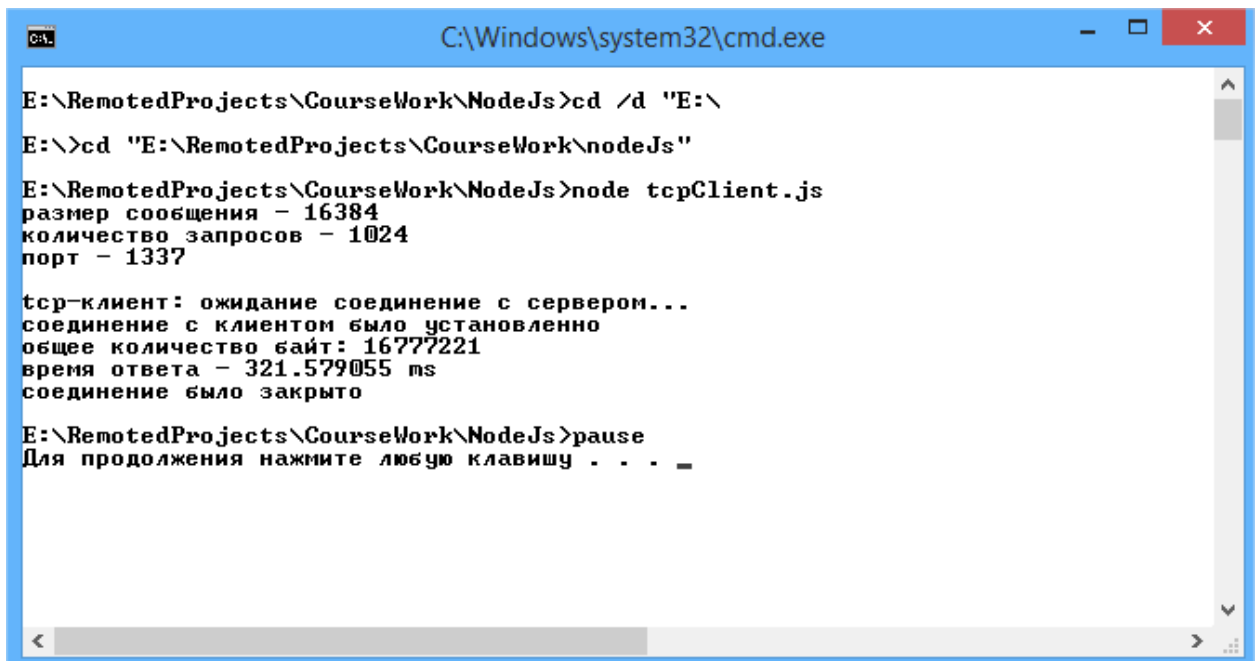


Рис. 4



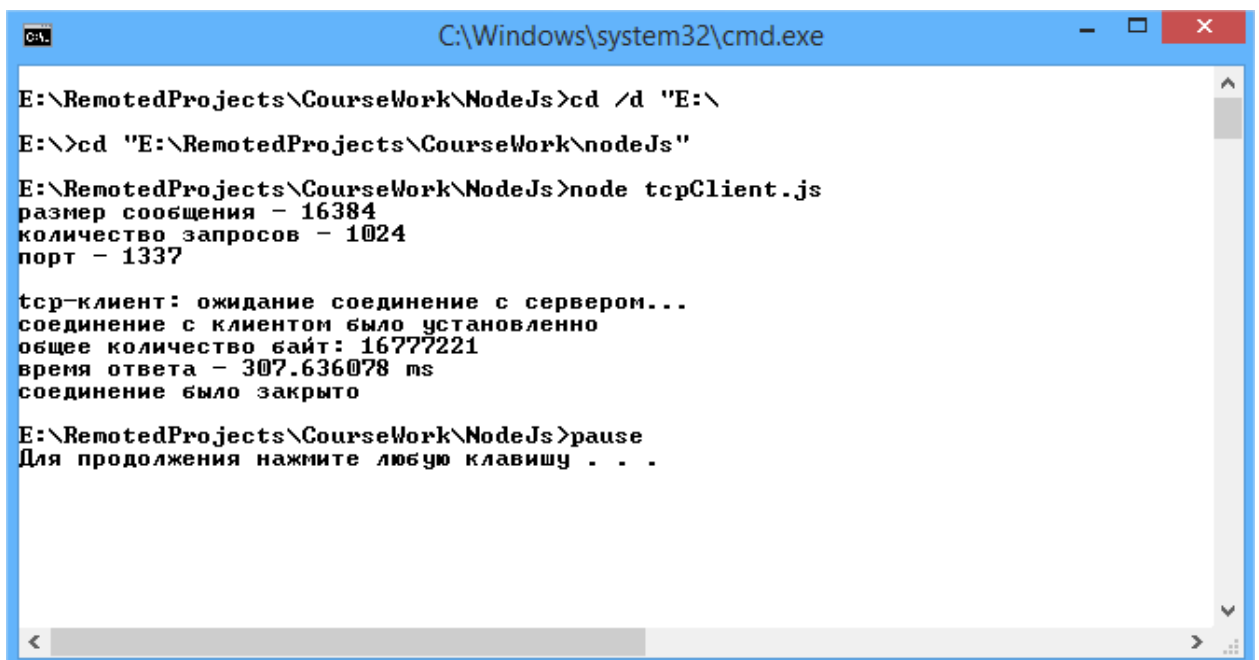
```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node tcpClient.js
размер сообщения - 16384
количество запросов - 1024
порт - 1337

tcp-клиент: ожидание соединение с сервером...
соединение с клиентом было установлено
общее количество байт: 16777221
время ответа - 321.579055 ms
соединение было закрыто

E:\RemotedProjects\CourseWork\NodeJs>pause
Для продолжения нажмите любую клавишу . . .
```

Рис. 5 Результат работы Node.js TCP-сервера



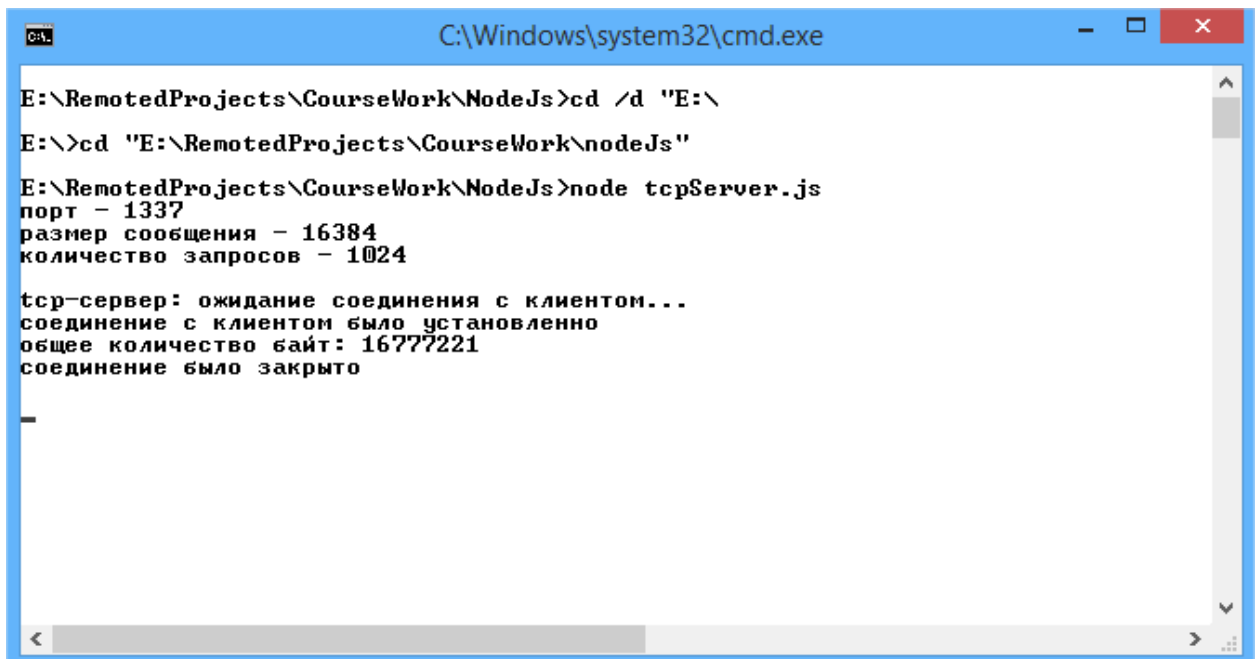
```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node tcpClient.js
размер сообщения - 16384
количество запросов - 1024
порт - 1337

tcp-клиент: ожидание соединение с сервером...
соединение с клиентом было установлено
общее количество байт: 16777221
время ответа - 307.636078 ms
соединение было закрыто

E:\RemotedProjects\CourseWork\NodeJs>pause
Для продолжения нажмите любую клавишу . . .
```

Рис. 6 Результат работы .NET TCP-сервера

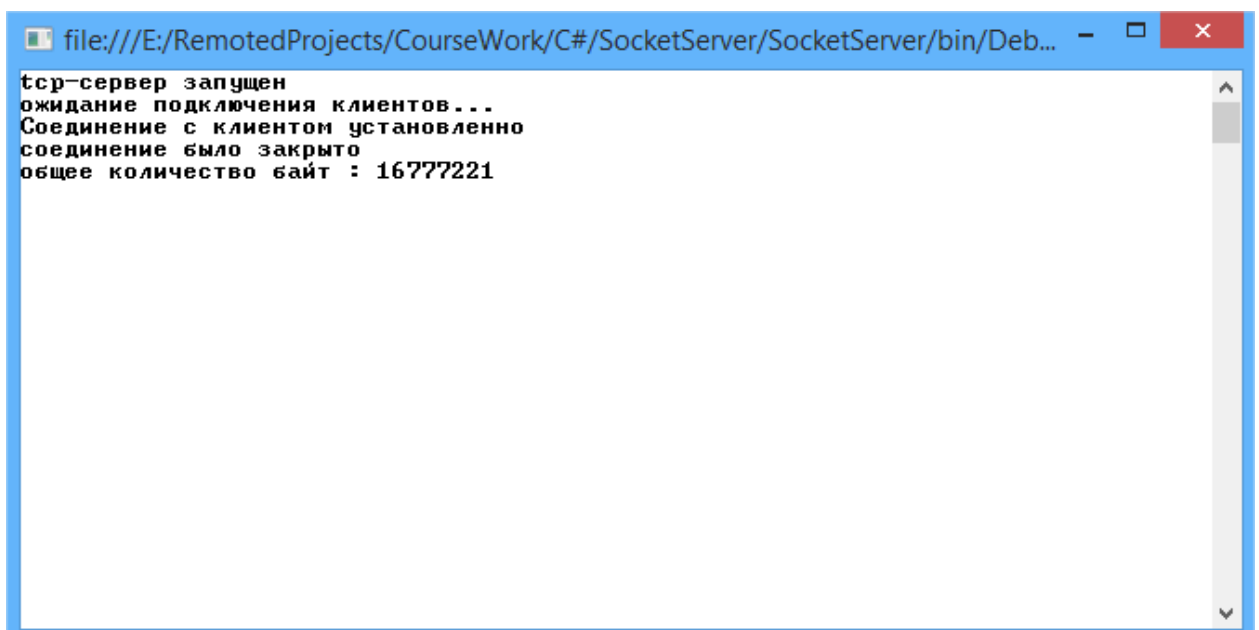


```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node tcpServer.js
порт - 1337
размер сообщения - 16384
количество запросов - 1024

tcp-сервер: ожидание соединения с клиентом...
соединение с клиентом было установлено
общее количество байт: 16777221
соединение было закрыто
```

Рис. 7 TCP-сервер на Node.js



```
file:///E:/RemotedProjects/CourseWork/C#/SocketServer/SocketServer/bin/Deb...

tcp-сервер запущен
ожидание подключения клиентов...
Соединение с клиентом установлено
соединение было закрыто
общее количество байт : 16777221
```

Рис. 8 TCP-сервер на .NET

На основании полученных результатов, мы видим что время ответа для обеих платформ практически идентично. Время ответа tcp-сервера не зависит от платформы.

Из этого следует, что любые расхождения в производительности серверов, работающих по протоколам, основанным на протоколе транспортного уровня TCP, зависят только от самой реализации сервера.

4.2. Сравнение производительности .NET и Node.js сокетов для протокола UDP

Реализация сокетов для протокола UDP выглядит почти также, как и для TCP. Но так как протокол UDP не требует заранее установленного соединения, то нам не нужно перед отправкой сообщения устанавливать соединение между клиентом и сервером. Мы также привязываемся к определённому порту и адресу. Но если для протокола TCP мы использовали один порт, то для UDP необходимо знать два порта - один порт для отправки сообщений, другой для получения. Таким образом если клиент отправляет сообщения через порт А и слушает порт В. То сервер должен сделать всё наоборот.

Для эксперимента выбран тот же самый алгоритм, как и для TCP. Клиент отправляет данные серверу, сервер в свою очередь отправляет их обратно клиенту. Как только количество отправленных данных достигает заранее установленного значения сервер разрывает соединение. На рисунках 9 и 10 показаны результаты работы протоколов UDP для Node.js и .NET. Код UDP-клиента приведён в приложении В. Код UDP-сервера для Node.js приведён в приложении Г. Код UDP-сервера для .NET приведён в приложении К.

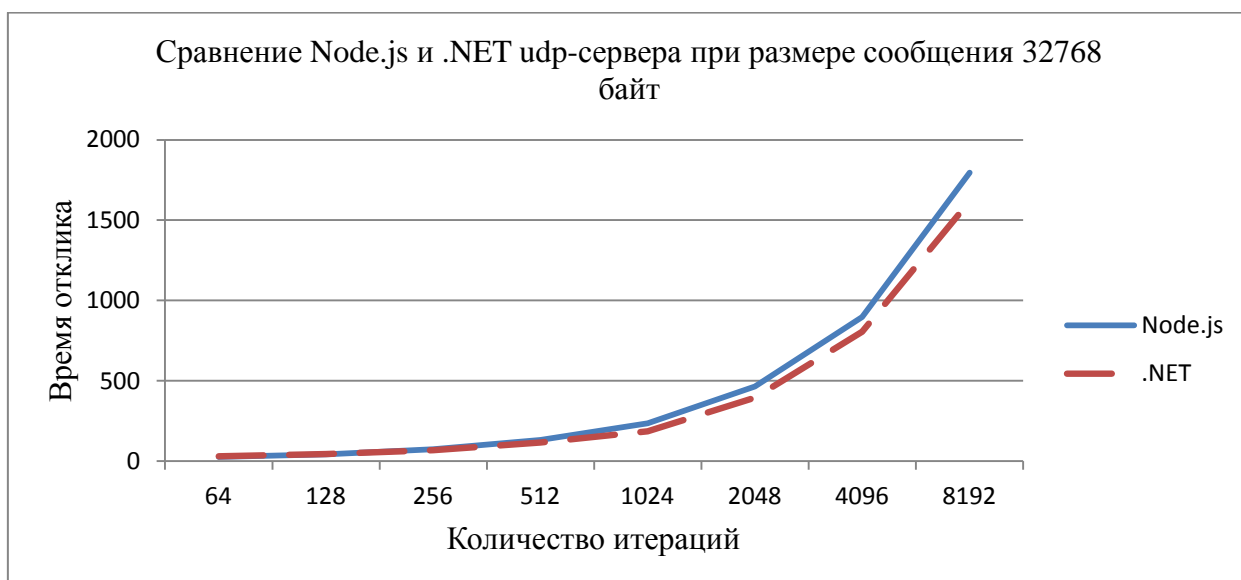


Рис. 9

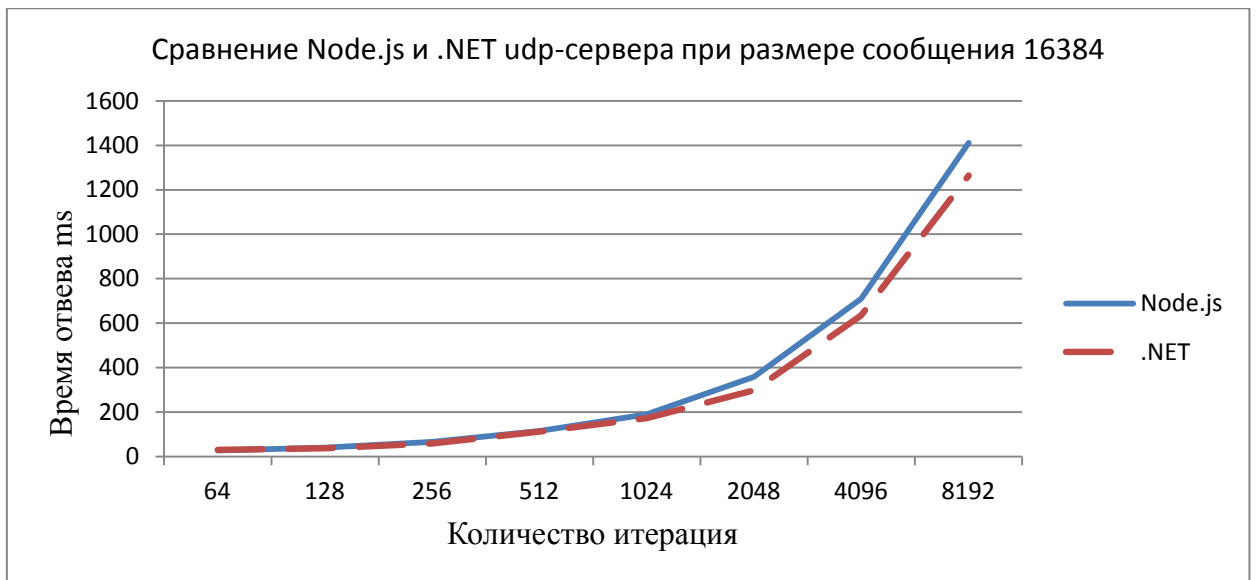


Рис. 10

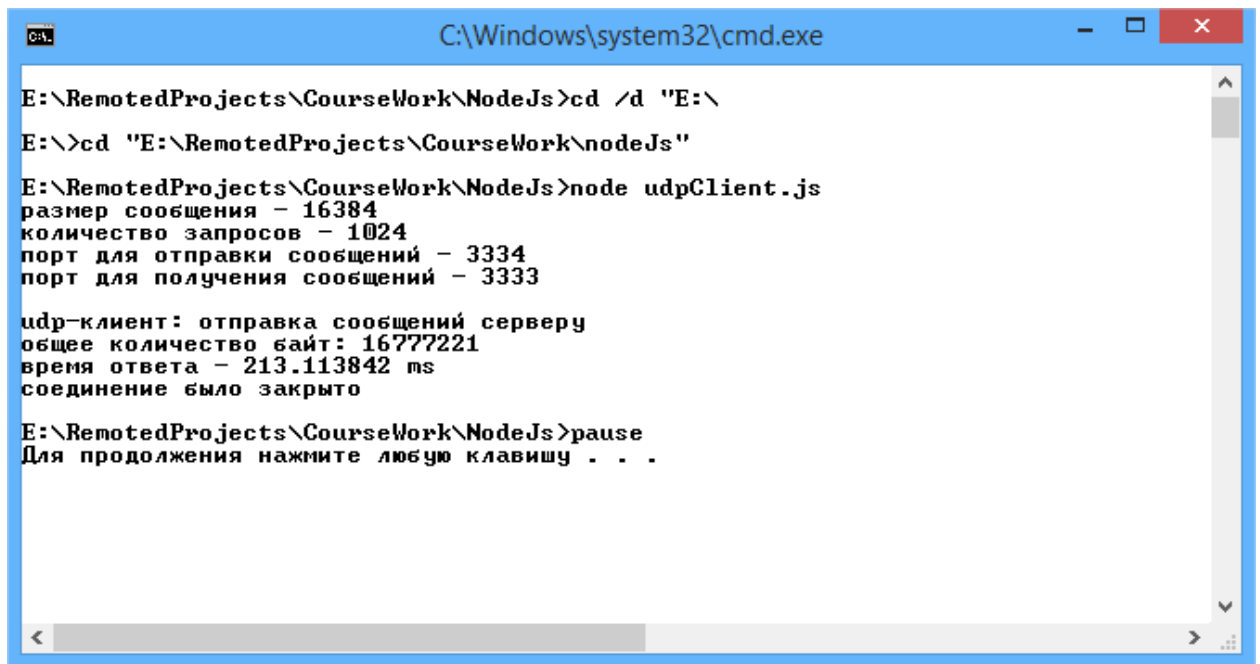
```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node udpClient.js
размер сообщения - 16384
количество запросов - 1024
порт для отправки сообщений - 3334
порт для получения сообщений - 3333

udp-клиент: отправка сообщений серверу
общее количество байт: 16777221
время ответа - 180.233858 ms
соединение было закрыто

E:\RemotedProjects\CourseWork\NodeJs>pause
Для продолжения нажмите любую клавишу . . . _
```

Рис. 11 Результат работы .NET UDP-сервера



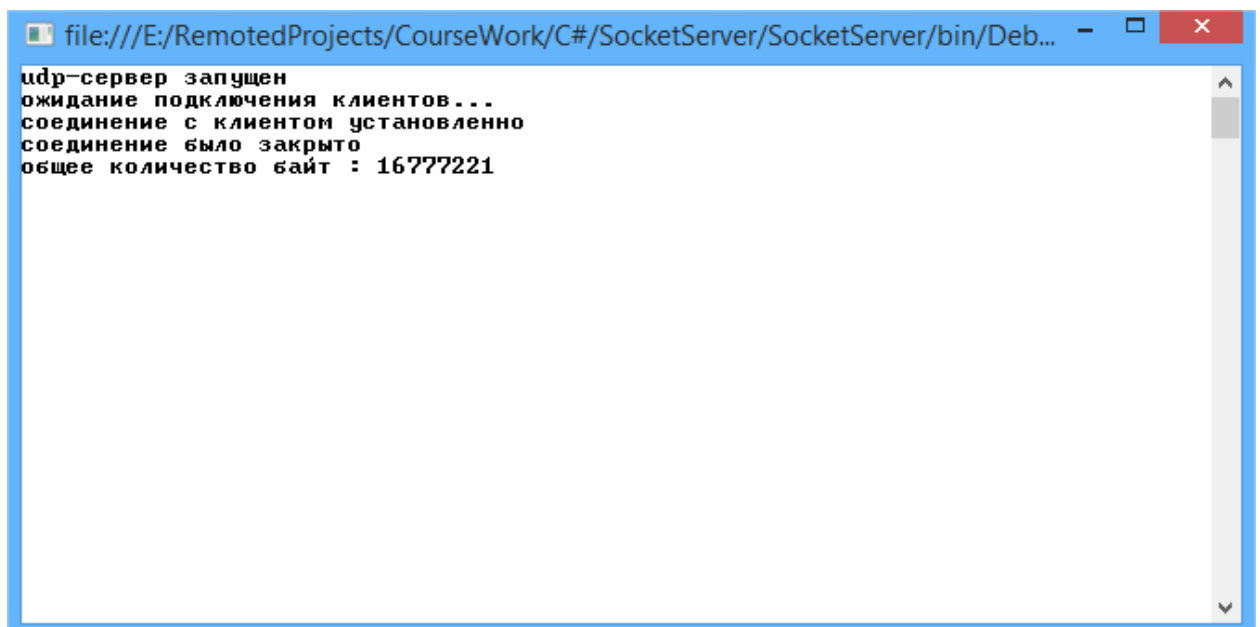
```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node udpClient.js
размер сообщения - 16384
количество запросов - 1024
порт для отправки сообщений - 3334
порт для получения сообщений - 3333

udp-клиент: отправка сообщений серверу
общее количество байт: 16777221
время ответа - 213.113842 ms
соединение было закрыто

E:\RemotedProjects\CourseWork\NodeJs>pause
Для продолжения нажмите любую клавишу . . .
```

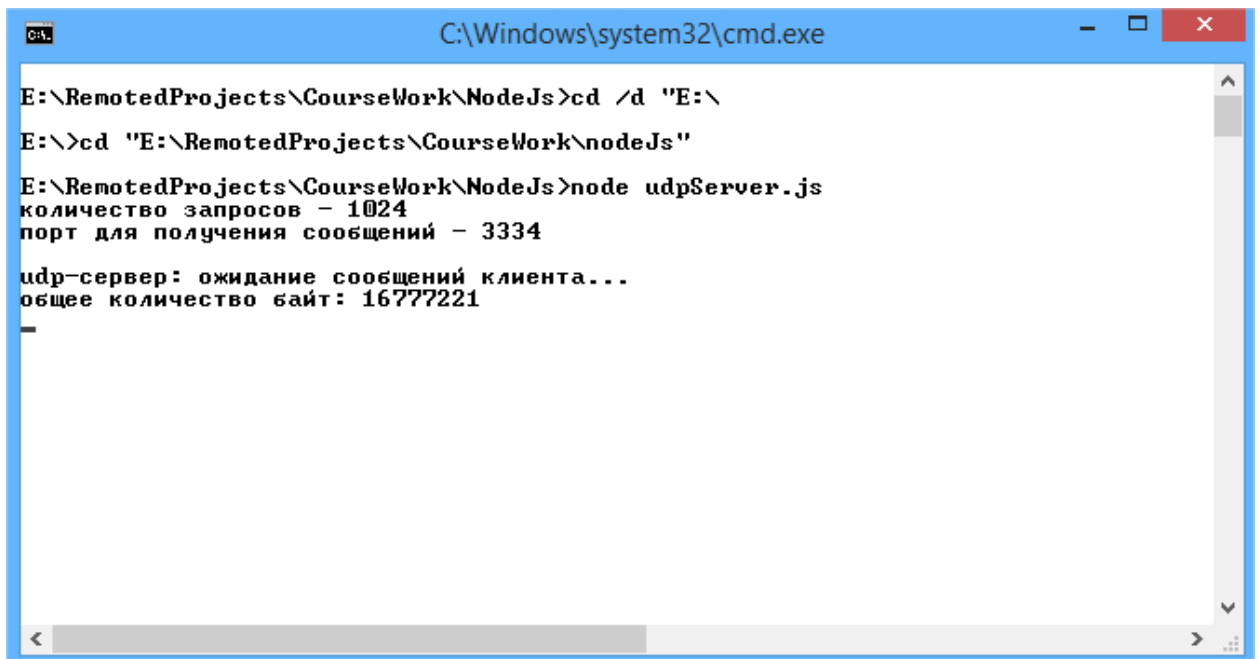
Рис. 12 Результат работы Node.js UDP-сервера



```
file:///E:/RemotedProjects/CourseWork/C#/SocketServer/SocketServer/bin/Deb...

udp-сервер запущен
ожидание подключения клиентов...
соединение с клиентом установлено
соединение было закрыто
общее количество байт : 16777221
```

Рис. 13 .NET UDP-сервер



```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node udpServer.js
количество запросов - 1024
порт для получения сообщений - 3334

udp-сервер: ожидание сообщений клиента...
общее количество байт: 16777221
```

Рис. 14 Node.js UDP-сервер

На основании полученных результатов, мы видим что время ответа для обеих платформ практически идентично. Время ответа udp-сервера также не зависит от платформы.

Из этого следует, что любые расхождения в производительности серверов, работающих по протоколам, основанным на протоколе транспортного уровня UDP, зависят только от самой реализации сервера.

4.3 Сравнение многопоточного и однопоточного HTTP-сервера на Node.js

Как известно Node.js спроектирован таким образом, что в любой момент времени загружено только одно ядро и один поток. Создатель языка Node.js сделал его однопоточным. Хотя цикл работы является асинхронным, на больших нагрузках, когда необходимо производить сложные вычисления, нагружающие центральный процессор, Node.js уступает многим другим языкам с поддержкой асинхронных вызовов. Но к счастью есть выход из данной ситуации. В платформе Node.js существует пространство имён "cluster", с помощью которого можно распараллелить обработку запросов.

В данном тесте мы сравниваем производительность Node.js с использованием кластера и без его применения. Для данного эксперимента мы создали 500 текстовых файлов на диске, где каждый файл состоит из 30000 чисел от 0 до 1 с 7 знаками после запятой.

Алгоритм следующий:

- 1) Клиент связывается с указанным адресом и портом и посылает n-количество запросов на сервер, где каждый запрос - это GET-запрос с номером файла в URL.
- 2) Сервер начинает прослушивать заранее оговорённый адрес и порт(в нашем случае для простоты был выбран локальный адрес - "127.0.0.1")
- 3) При получении запроса от клиента, сервер читает номер файла из URL, открывает файл и записывает все числа из файла в массив.

- 4) Сервер производит сортировку созданного на 3 шаге массива по алгоритму быстрой сортировки и по завершении отправляет сигнал клиенту о завершении работы с файлом
- 5) При накоплении n-количества ответов от сервера клиент замеряет время

С использованием кластера мы создаём столько потоков, сколько ядер на рабочем процессоре. Каждый запрос к серверу будет попадать в один из созданных потоков. Без кластера сервер сможет обрабатывать запросы только в однопоточном режиме. На рисунке 15 представлен график работы сервера на Node.js в многопоточном и однопоточном режимах. Код HTTP-клиента на .NET приведён в приложении Е. Код HTTP-сервера на Node.js приведён в приложении Д. Код программы, создающей текстовые файлы для нагрузки сервера приведён в приложении Л.

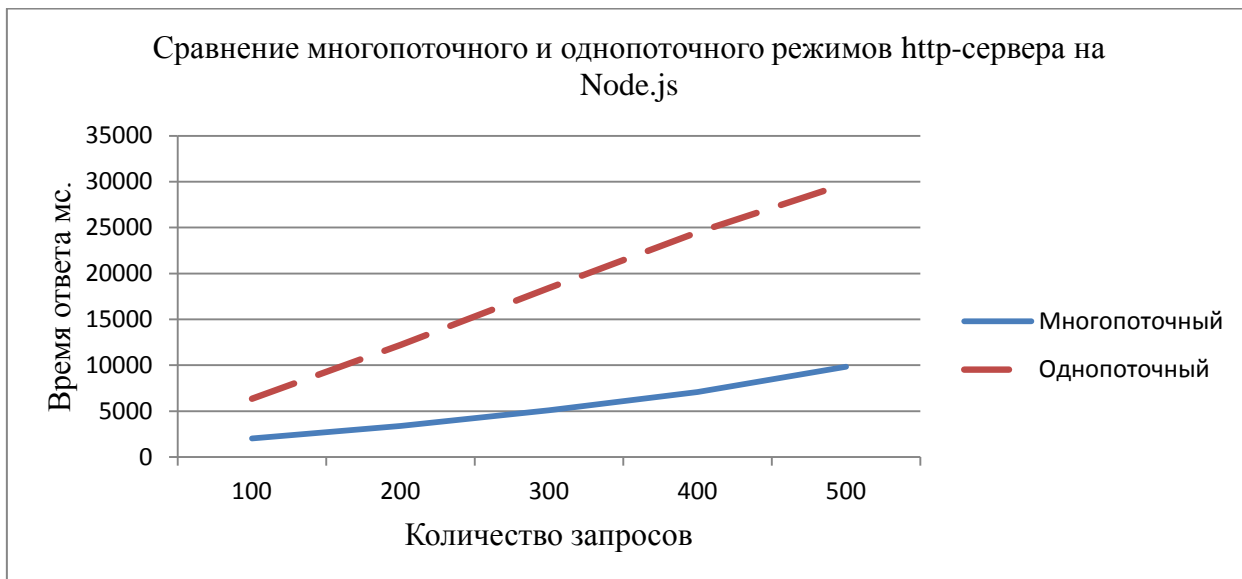


Рис. 15

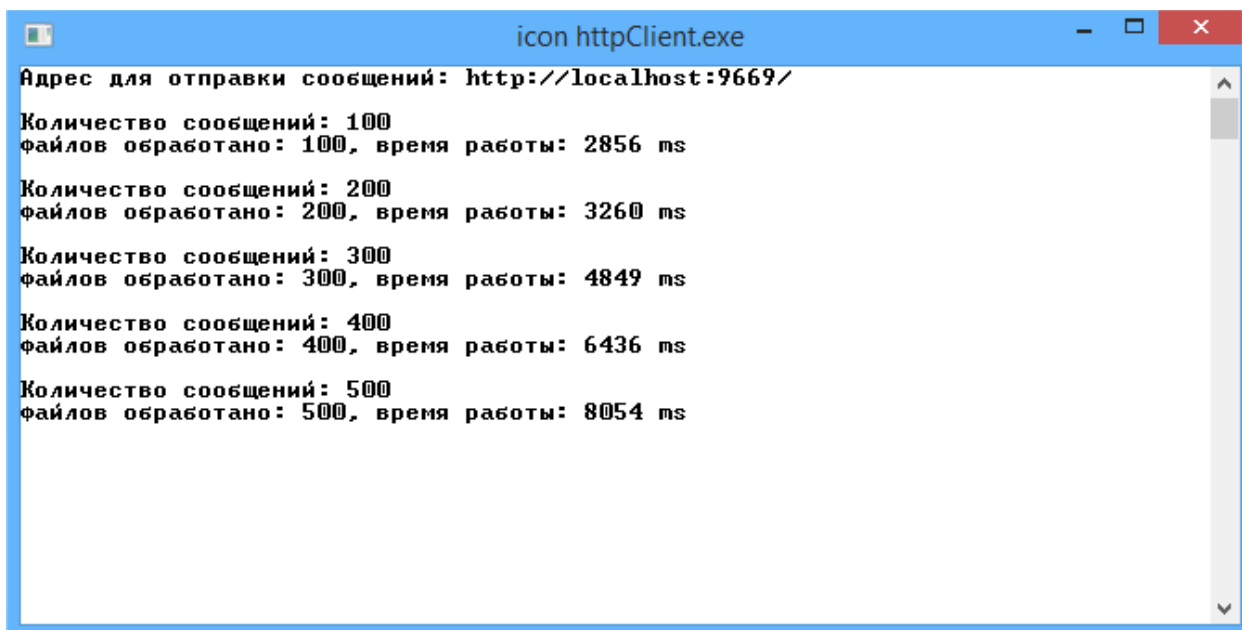


Рис. 16 Результат работы многопоточного Node.js HTTP-сервера

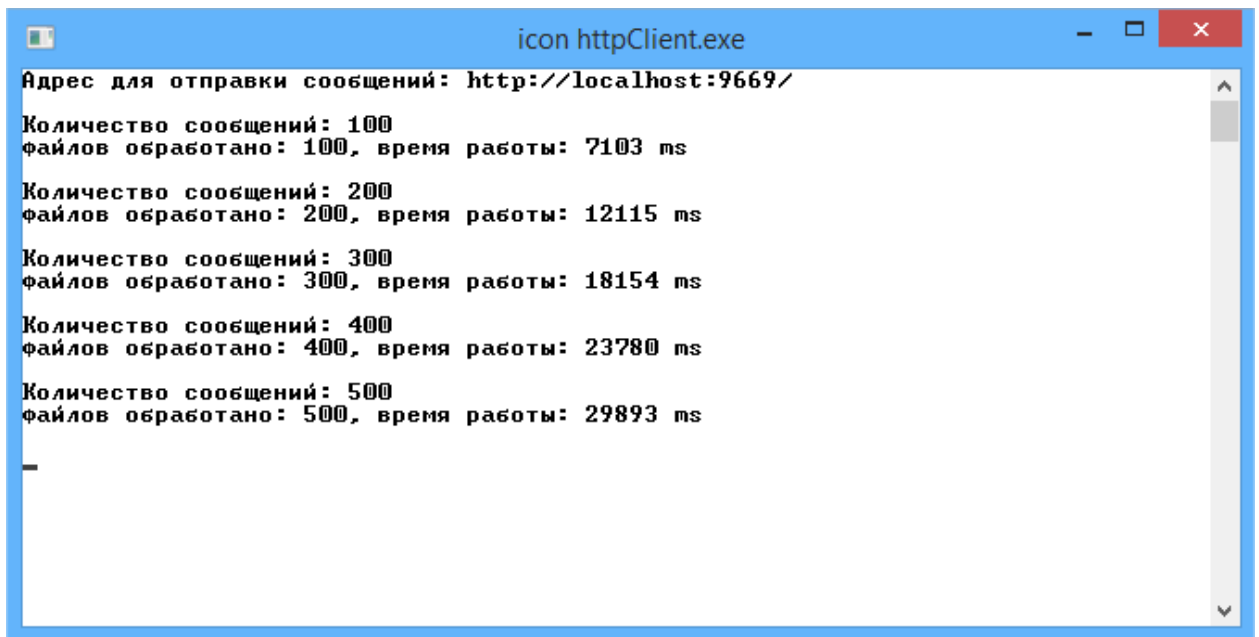


Рис. 17 Результат работы однопоточного Node.js HTTP-сервера

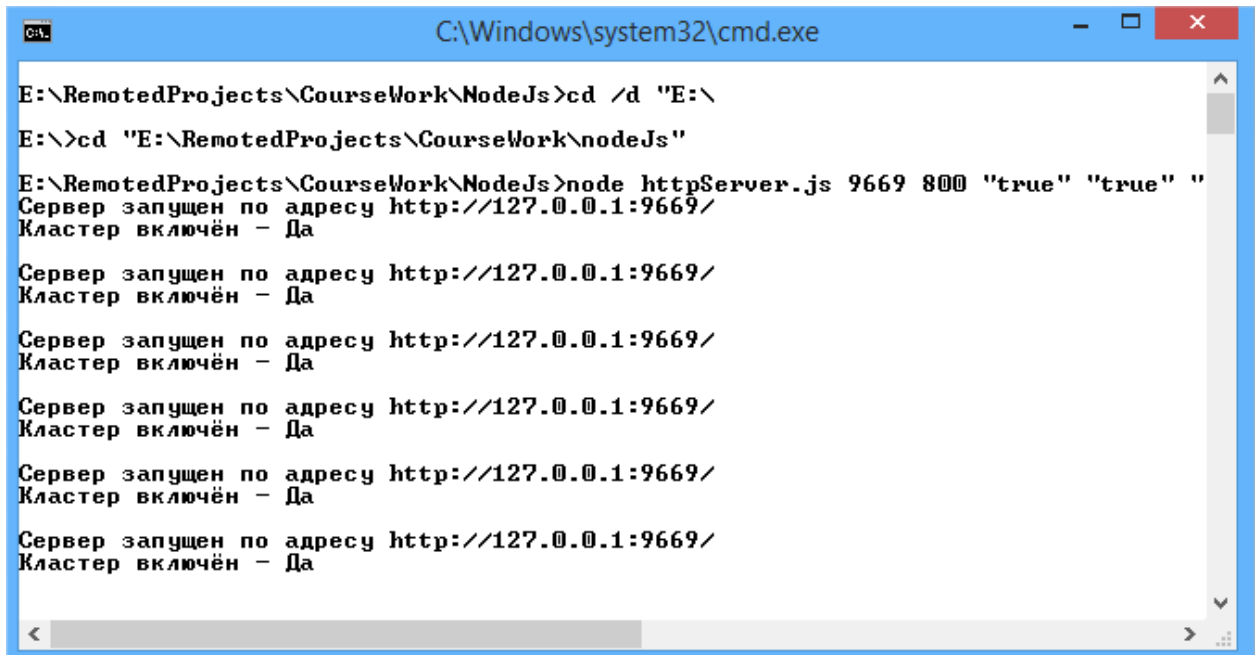
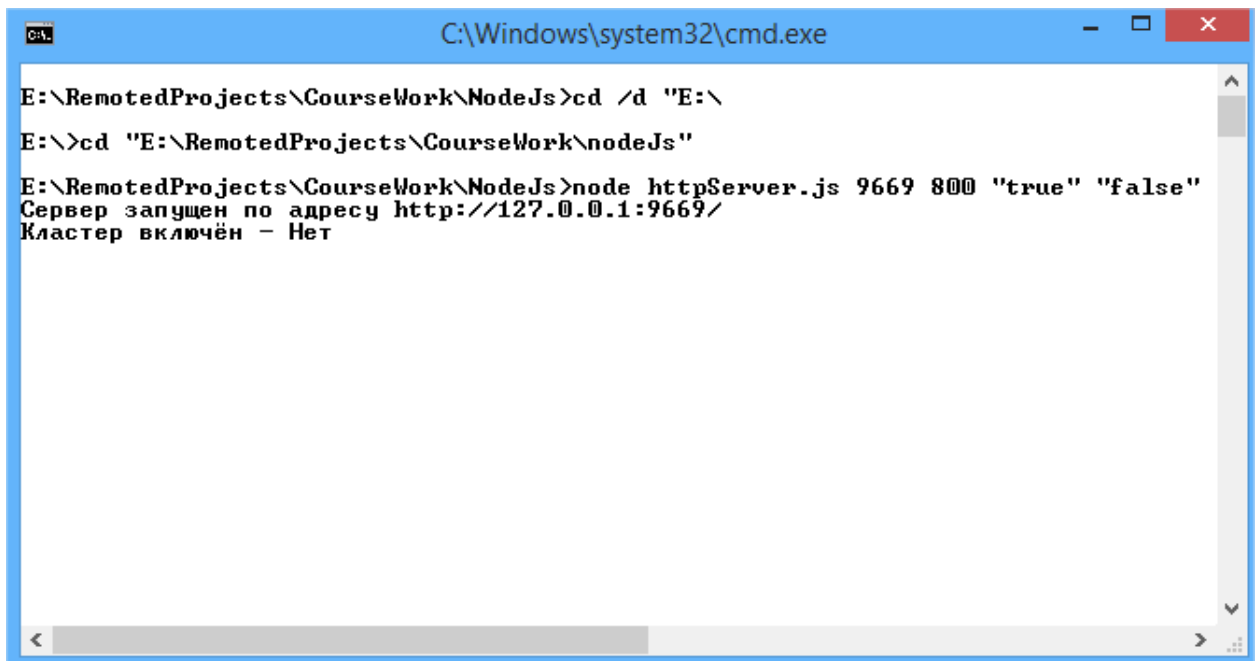


Рис. 18 многопоточный Node.js HTTP-сервер



```
C:\Windows\system32\cmd.exe

E:\RemotedProjects\CourseWork\NodeJs>cd /d "E:\
E:\>cd "E:\RemotedProjects\CourseWork\nodeJs"
E:\RemotedProjects\CourseWork\NodeJs>node httpServer.js 9669 800 "true" "false"
Сервер запущен по адресу http://127.0.0.1:9669/
Кластер включён - Нет
```

Рис. 19 однопоточный Node.js HTTP-сервер

Мы видим, что многопоточный режим сервера с использованием кластера сильно опережает однопоточный режим, что очевидно, так как распараллеливание всегда даёт лучший результат на большом количестве запросов.

4.4 Сравнение Node.js и .NET HTTP-серверов в многопоточном режиме

Многопоточность - одна из самых сложных вещей в программировании. Но с помощью неё можно достичь высоких показателей производительности. В этом эксперименте для платформы .NET мы будем использовать библиотеку параллельных задач (TPL), а для платформы Node.js многопоточность будет реализована также, как в предыдущем эксперименте - с помощью кластера.

Мы также будем осуществлять запросы клиента к серверу, загружая сервер сортировкой массива из 30000 чисел. На рисунке 20 представлен график работы Node.js и .NET HTTP-сервера. Код HTTP-сервера на Node.js приведён в приложении Д. Код HTTP-сервера на .NET приведён в приложении Ж. Код HTTP-клиента на .NET приведён в приложении Е.

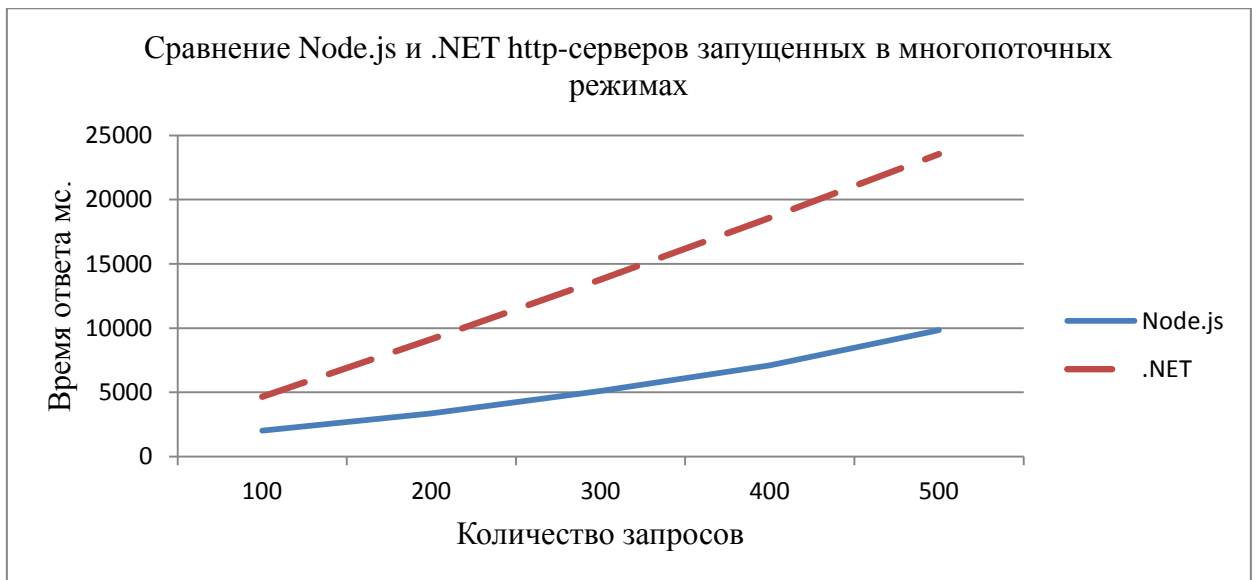


Рис. 20

```
icon httpClient.exe
Адрес для отправки сообщений: http://localhost:9669/
Количество сообщений: 100
файлов обработано: 100, время работы: 2856 ms
Количество сообщений: 200
файлов обработано: 200, время работы: 3260 ms
Количество сообщений: 300
файлов обработано: 300, время работы: 4849 ms
Количество сообщений: 400
файлов обработано: 400, время работы: 6436 ms
Количество сообщений: 500
файлов обработано: 500, время работы: 8054 ms
```

Рис. 21 Результат работы Node.js HTTP-сервера

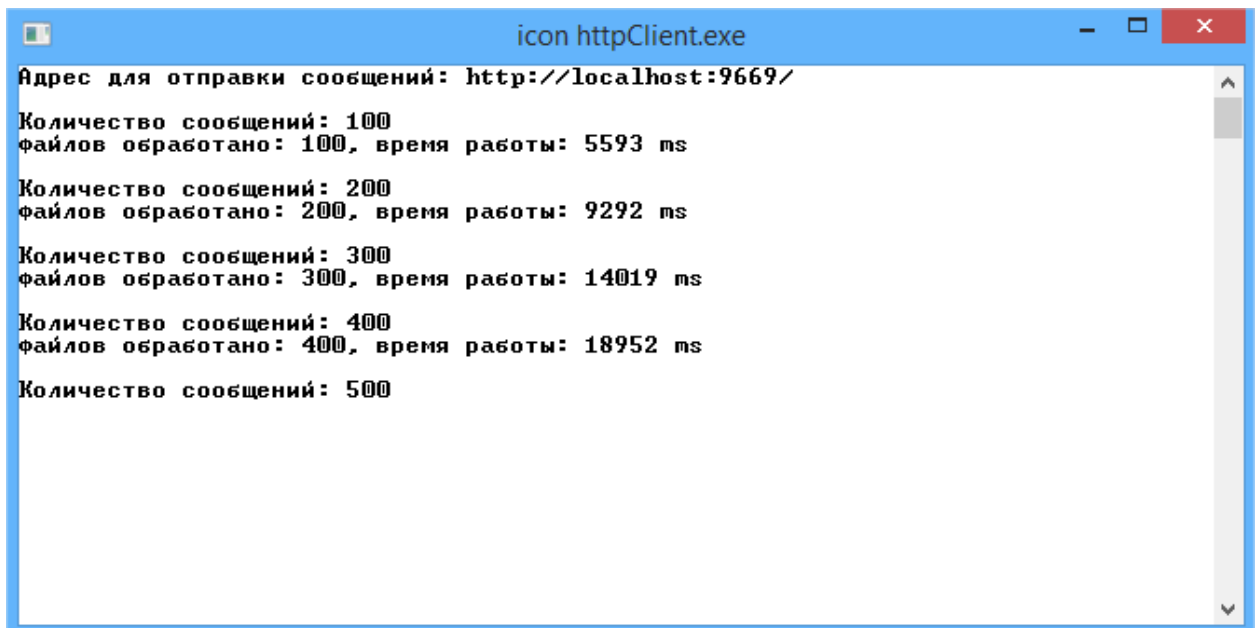


Рис. 22 Результат работы .NET HTTP-сервера

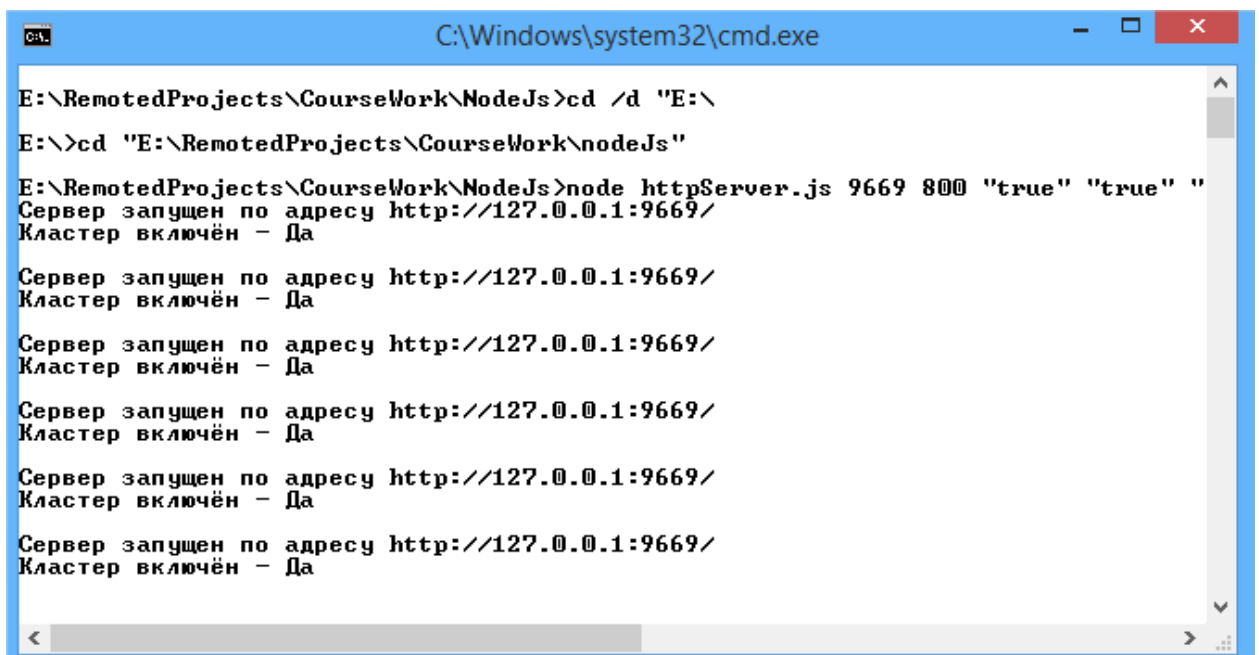


Рис. 23 Node.js HTTP-сервер

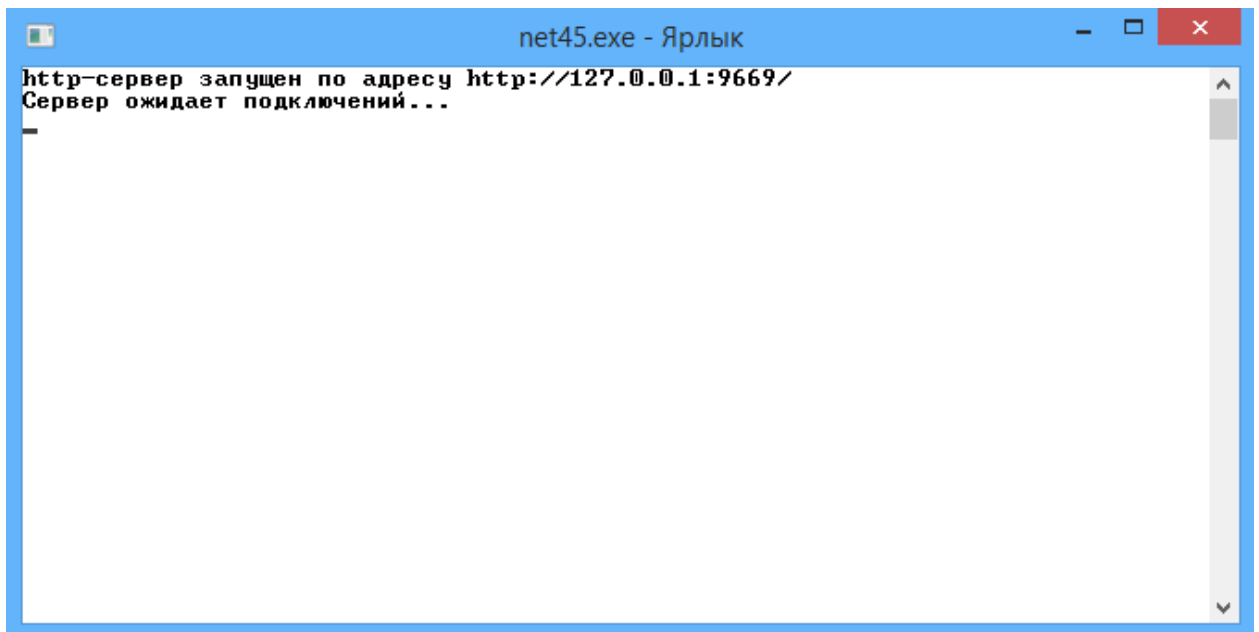


Рис. 24 .NET HTTP-сервер

Исходя из результатов данного эксперимента мы видим, что Node.js значительно опережает .NET. Это связано с тем, что язык Node.js был изначально создан как асинхронная модель обработки запросов. Платформа .NET появилась гораздо раньше, но её асинхронная модель долгое время была сложна и построить правильную архитектуру асинхронным вызовов было не так тривиально. Новый подход с использованием TPL упрощает разработку асинхронных приложений, но даже с ним .NET не может обогнать Node.js с использованием кластера.

4.5 Сравнение Node.js и .NET HTTP-серверов запущенных под IIS

В большинстве случаев в реальных производственных проектах один или несколько серверов будут использоваться для обслуживания клиентских запросов веб-сайта. Эти серверы могут принадлежать и управляться непосредственно вами, специализированной командой или же сторонней компанией, предоставляющей услуги хостинга. В любом случае рано или поздно наступает момент, когда написание кода и его тестирование завершено, и работа должна быть представлена широкой публике - в этом заключается развёртывание веб-сайта. Для этого и необходим IIS(Internet Information Services).

IIS представляют собой группу интернет-серверов, в том числе веб-сервер и FTP-сервер. IIS включает приложения для построения и управления веб-сайтами, машину поиска и поддержку разработчиков веб-приложений, имеющих доступ к базам данных. С помощью данного сервиса можно развернуть приложения как на ASP.NET, так и на Node.js.

В данном эксперименте мы создали приложения MVC с одним контроллером и всего одним методом, который при поступлении запроса, достаёт из URL номер файла, записывает в массив все числа из считываемого файла и производит сортировку. Для чистоты эксперимента, метод контроллера использует технологию "async/await", что позволяет обрабатывать запросы клиента асинхронно. Точно по такому же алгоритму работает и HTTP-сервер на Node.js. В реализации сервера Node.js мы не используем кластер. Вместо этого для IIS мы используем его надстройку - iisnode. После запуска этой

утилиты мы можем развёртывать приложения на Node.js в сервисе IIS. График производительности серверов можно увидеть на рисунке 25. Код HTTP-сервера для .NET приведён в приложении Д.

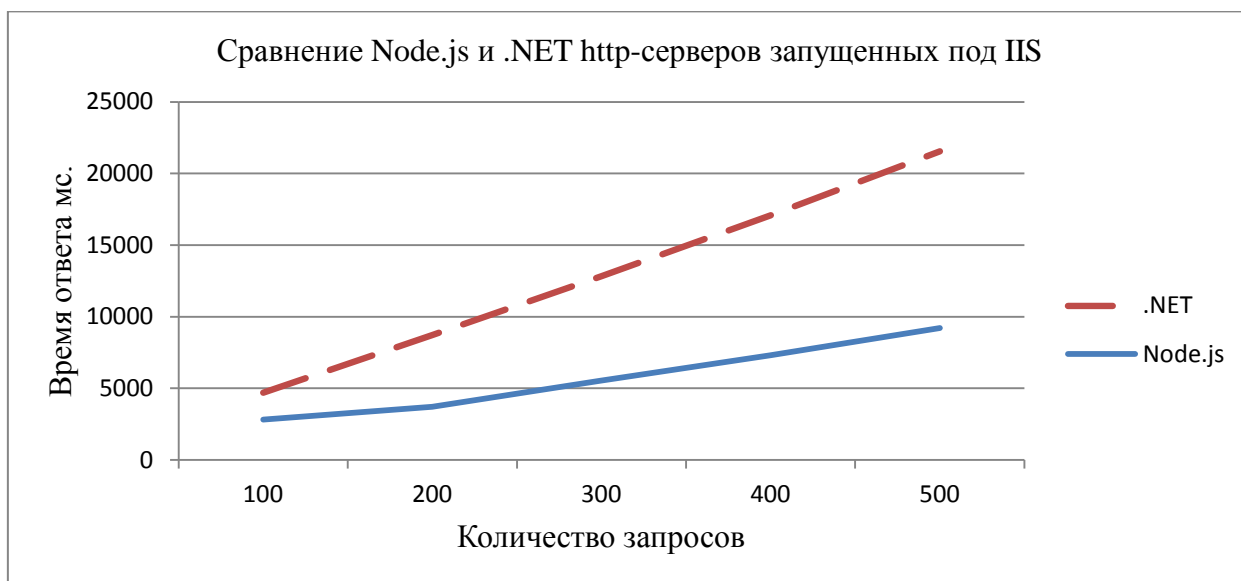


Рис. 25

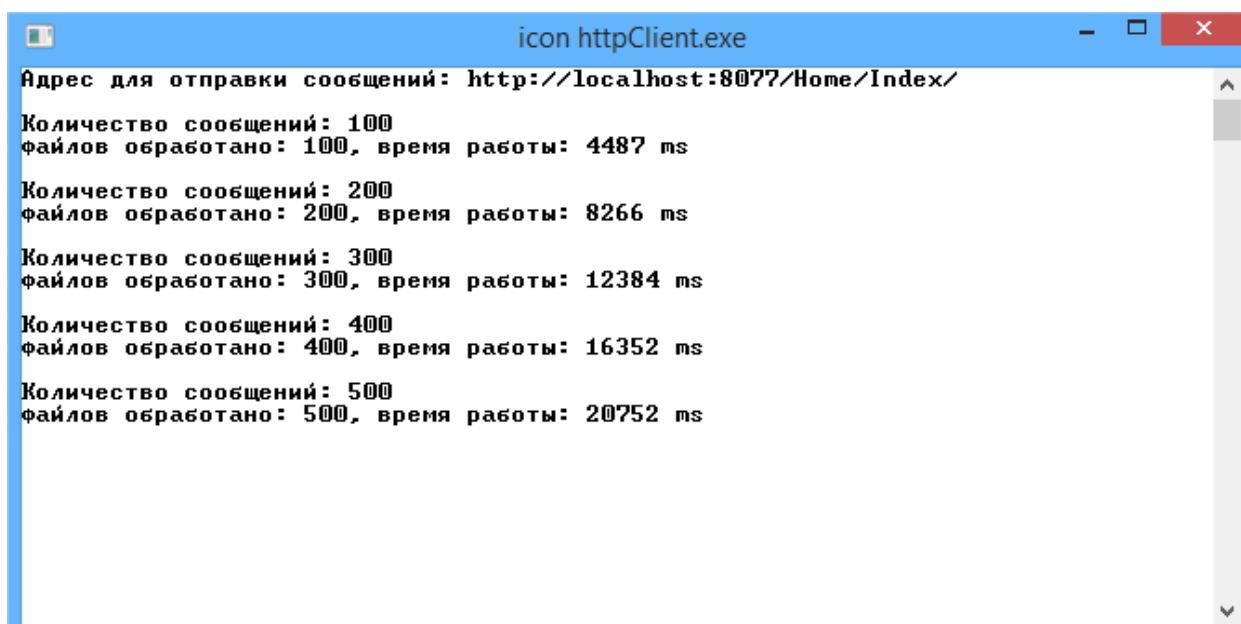


Рис. 26

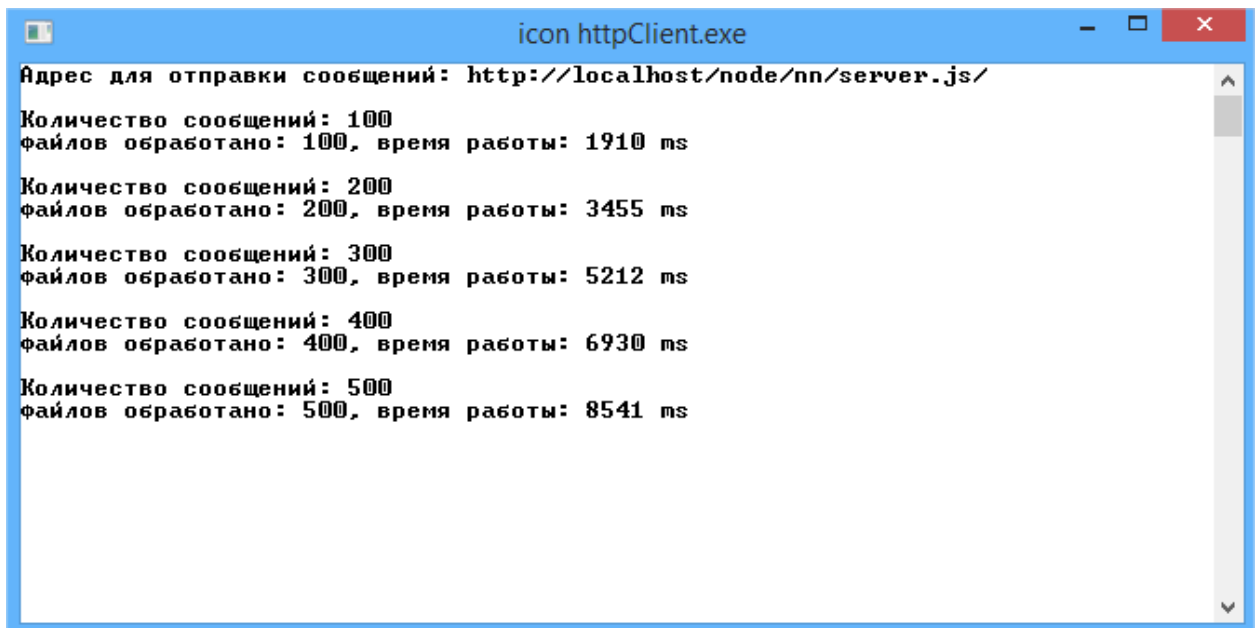


Рис. 27

IIS обслуживает HTTP-сервер таким образом, что под каждый входящий запрос он создаёт отдельный поток, в котором и происходит обработка входящего запроса. Так как Node.js по сравнению с .NET, справляется лучше с обработкой запроса при равных условиях, то из этого следует что и при размещении сервера на IIS, Node.js будет обрабатывать запросы пользователей быстрее, чем .NET.

Заключение

В данной работе были проведены исследования в области реализации веб-серверов, работающих по протоколам транспортного и прикладного уровней на платформах Node.js и .NET. Были также изучены и различия в асинхронных моделях платформ Node.js и .NET.

Нами исследовалось поведение серверов на платформах Node.js и .NET при трех различных сценариях:

Работа веб-серверов по протоколам транспортного уровня TCP и UDP. Результаты оказались практически идентичными, что свидетельствует о независимости работы реализации протокола от платформы, на котором она написана.

Работа веб-серверов по протоколу прикладного уровня HTTP в многопоточном и однопоточном режимах. В этом эксперименте Node.js значительно опередил .NET, что говорит о том что его асинхронная модель гораздо лучше спроектирована для работы сервисных приложений.

Список литературы

1. Хэррон Д. "Node.js. Разработка серверных веб-приложений в JavaScript" - М.: ДМК Пресс, 2012. – 144с.: ил.
2. Сухов К.К. "Node.js. Путеводитель по технологии" - М.: ДМК Пресс, 2015. – 416с.: ил.
3. Кантелон М., Хартер М., Головайчук Т., Райлих Н. "Node.js в действии". — СПб.: Питер, 2014. - 548с.: ил
4. Пауэрс Ш. "Изучаем Node.js" - СПб.: Питер, 2014 - 400с.: ил.
5. Documentation : [HTTPS://nodejs.org/](https://nodejs.org/). -URL: [HTTPS://nodejs.org/dist/latest-v6.x/docs/api/](https://nodejs.org/dist/latest-v6.x/docs/api/)
6. [HTTP://stackoverflow.com/](http://stackoverflow.com/). -URL: [HTTP://stackoverflow.com/questions/9290160/node-js-vs-net-performance](http://stackoverflow.com/questions/9290160/node-js-vs-net-performance)

TCP-клиент на платформе Node.js

```
"use strict"

const net = require('net');
const client = new net.Socket();
const options = {
  sizeOfMessage: 16384,
  iterations: 1024,
  port: 1337,
  host: '127.0.0.1'
}
const state = {
  contentLength: 0,
  iterationNumber: 0,
  time: null
}
const message = Buffer.alloc(options.sizeOfMessage, '1', 'utf-8');

client.on('readable', () => {
  let chunk;
  while (null !== (chunk = client.read())) {
    state.contentLength += chunk.length
  }
  state.iterationNumber++

  state.iterationNumber <= options.iterations && client.write(message)
})
client.on('end', () => {
  let [begin, end] = process.hrtime(state.time)
  console.log(`общее количество байт: ${state.contentLength}`)
  console.log(`время ответа - ${(begin * 1e9 + end) / 1000000} ms`)
  console.log('соединение было закрыто')
})
client.connect(options.port, options.host, () => {
  console.log('соединение с клиентом было установлено');
  state.time = process.hrtime()

  client.write('hello')
})

console.log(`размер сообщения - ${options.sizeOfMessage}`)
console.log(`количество запросов - ${options.iterations}`)
console.log(`порт - ${options.port}`)
console.log()
console.log('tcp-клиент: ожидание соединения с сервером...')
```

TCP-сервер на платформе Node.js

```
'use strict'
const net = require('net');

const options = {
  port: 1337,
  greetingLengthMessage: 5,
  iterations: 1024,
  sizeOfMessage: 16384
}
options.maxContentLength = options.sizeOfMessage * options.iterations +
options.greetingLengthMessage
let contentLength = 0

const server = net.createServer((client) => {
  console.log('соединение с клиентом было установлено')

  client.on('close', () => {
    console.log(`общее количество байт: ${contentLength}`)
    console.log('соединение было закрыто')
    console.log()
    contentLength = 0
  })
  client.on('readable', () => {
    const buffer = client.read()
    if (buffer !== null)
      contentLength += buffer.length

    if (contentLength < options.maxContentLength) {
      return client.write(buffer)
    }

    buffer !== null && client.end(buffer)
  })
})
server.on('error', (err) => {
  server.close()
  throw err;
})

server.listen(options.port, () => {
  console.log(`порт - ${options.port}`)
  console.log(`размер сообщения - ${options.sizeOfMessage}`)
  console.log(`количество запросов - ${options.iterations}`)
  console.log()
  console.log('tcp-сервер: ожидание соединения с клиентом...')
})
```

UDP-клиент на платформе Node.js

```
'use strict'

const dgram = require("dgram")
const client = dgram.createSocket("udp4")

const options = {
  portForReceiving: 3333,
  portForSending: 3334,
  iterations: 1024,
  sizeOfMessage: 16384,
  host: "127.0.0.1"
}
const message = Buffer.alloc(options.sizeOfMessage, '1', 'utf-8')
const sendMessage = (message) => client.send(message, 0, message.length,
options.portForSending, options.host)
let time,
    i = 0,
    totalLength = 0

client.on("message", (msg) => {
  totalLength += msg.length
  i++
  sendMessage(message)
  i > options.iterations && client.close()
})

client.on('close', () => {
  var [begin, start] = process.hrtime(time)
  console.log(`общее количество байт: ${totalLength}`)
  console.log(`время ответа - ${(begin * 1e9 + start) / 1000000} ms`)
  console.log('соединение было закрыто')
  process.exit()
})

console.log(`размер сообщения - ${options.sizeOfMessage}`)
console.log(`количество запросов - ${options.iterations}`)
console.log(`порт для отправки сообщений - ${options.portForSending}`)
console.log(`порт для получения сообщений - ${options.portForReceiving}`)
console.log()
console.log('udp-клиент: отправка сообщений серверу')

client.bind(options.portForReceiving, options.host)
time = process.hrtime()
sendMessage(new Buffer('hello'))
```

UDP-сервер на платформе Node.js

```
'use strict'
const dgram = require('dgram')
const server = dgram.createSocket('udp4')

const options = {
  portForReceiving: 3334,
  host: "127.0.0.1",
  iterations: 1024
}
const sendMessage = (message, info) => server.send(message, 0, message.length, info.port,
info.address)
let lengthContent = 0,
    iterationNumber = 0

server.on('close', () => console.log('соединение было закрыто'))

server.on('message', (message, info) => {
  lengthContent += message.length
  iterationNumber++
  sendMessage(message, info)
  if (iterationNumber > options.iterations) {
    console.log(`общее количество байт: ${lengthContent}`)
    lengthContent = 0
    iterationNumber = 0
  }
})

server.on('error', (err) => {
  server.close()
  throw err;
})

console.log(` количество запросов - ${options.iterations}`)
console.log(` порт для получения сообщений - ${options.portForReceiving}`)
console.log()
console.log('udp-сервер: ожидание сообщений клиента...')

server.bind(options.portForReceiving, options.host)
```

HTTP сервер на платформе Node.js

```

'use strict'

const http = require('http')
const fs = require('fs')
const cluster = require('cluster')
const os = require('os')

const processorNumber = os.cpus().length
let [port, fileName, isLoadingIncluded, isClusterIncluded, directory] = process.argv.slice(2)

const quickSort = (array, left, right) => {
  let temp;
  let x = array[left + (right - left) / 2];
  let i = left;
  let j = right;

  while (i <= j) {
    while (array[i] < x) {
      i++;
    }
    while (array[j] > x) {
      j--;
    }
    if (i <= j) {
      temp = array[i];
      array[i] = array[j];
      array[j] = temp;
      i++;
      j--;
    }
  }
  if (i < right)
    quickSort(array, i, right);

  if (left < j)
    quickSort(array, left, j);
}

const createServer = () => {
  http.createServer((request, response) => {
    const file = parseInt(request.url.substring(1))
    let fileName = `000${file % fileNumber}`.slice(-3)
    fileName = `file${fileName}.txt`

    fs.readFile(`${directory}\\${fileName}`, 'ascii', (err, data) => {
      if (err) {
        response.writeHead(400, {'Content-Type': 'text/plain'})

```

```

        response.end()
    }
    else {
        if (isLoadingIncluded === 'true') {
            const array = data.toString().split("\r\n")
            const time = process.hrtime()
            quickSort(array, 0, array.length - 1)
            const [begin,end] = process.hrtime(time)
            const interval = (begin * 1e9 + end) / 1000000

            response.writeHead(200, {'Content-Type': 'text/plain'})
            response.end(`file: ${fileName}, length: ${array.length}, time: ${interval}`)
        } else {
            response.writeHead(200, {'Content-Type': 'text/plain'})
            response.end(`file: ${fileName}; without loading`)
        }
    }
    })
    }).listen(port, '127.0.0.1')
    console.log(`Сервер запущен по адресу http://127.0.0.1:${port}/`)
    console.log(`Кластер включён - ${isLoadingIncluded === 'true' ? 'Да' : 'Нет'}`)
    console.log()
}

if (isClusterIncluded === 'true') {
    if (cluster.isMaster) {
        for (let i = 0; i < processorNumber; i++) {
            cluster.fork()
        }

        cluster.on('exit', (worker, code, signal) => {
            console.log('worker ' + worker.process.pid + ' died')
        })
    }
    else {
        createServer()
    }
} else {
    createServer()
}

```

HTTP-клиент на платформе .NET

```

public class Client
{
    private readonly string _baseUrl;
    private readonly int _tasks;
    private readonly ConcurrentQueue<string> _result;

    public Client(string baseUrl, int tasks)
    {
        _baseUrl = baseUrl;
        _tasks = tasks;
        _result = new ConcurrentQueue<string>();
    }

    public void Start()
    {
        var timer = new Stopwatch();
        timer.Start();
        var tasks = new Task[_tasks];
        for (int i = 0; i < _tasks; ++i)
        {
            tasks[i] = Perform(i);
        }
        try
        {
            Task.WaitAll(tasks, -1);
        }
        catch (AggregateException exception)
        {
            throw exception;
        }
        timer.Stop();

        Console.WriteLine("файлов обработано: {0}, время работы: {1} ms", _tasks,
timer.ElapsedMilliseconds);
    }

    private async Task Perform(int state)
    {
        string url = String.Format("{0}{1}", _baseUrl, state.ToString().PadLeft(3, '0'));
        var client = new HttpClient();
        var timer = new Stopwatch();
        var shouldReconnect = true;

        string stringResult = "";
        while (shouldReconnect)
        {
            try

```

```

        {
            timer.Start();
            stringResult = await client.GetStringAsync(url);
            timer.Stop();
            shouldReconnect = false;
        }
        catch (Exception ex)
        {
            shouldReconnect = true;
        }
    }
}

_result.Enqueue(string.Format("{0,4}\t{1,5}\t{2}", url, timer.ElapsedMilliseconds,
stringResult));
    }
}

static class Program
{
    public static void Main(string[] args)
    {
        var address = args[0];
        Console.WriteLine("Адрес для отправки сообщений: {0}", address);
        Console.WriteLine();

        for (var i = 100; i <= 500; i+=100)
        {
            Console.WriteLine("Количество сообщений: {0}", i);
            var client = new Client(address, i);
            client.Start();
            Console.WriteLine();
        }

        Console.ReadLine();
    }
}

```


HTTP-сервер на платформе .NET

```
public class Server
{
    private readonly HttpListener _listener = new HttpListener();
    private readonly ASCIIEncoding _encoding = new ASCIIEncoding();
    private readonly string _directory;
    private readonly int _maxFileNumber;
    private readonly bool _isLoadingIncluded;

    public Server(string address, string directory, int maxFileNumber, bool isLoadingIncluded)
    {
        _directory = directory;
        _maxFileNumber = maxFileNumber;
        _isLoadingIncluded = isLoadingIncluded;

        _listener.Prefixes.Add(address);
        _listener.Start();
    }

    public async Task Start()
    {
        while (true)
        {
            var context = await _listener.GetContextAsync();
            ProcessRequest(context);
        }
    }

    private string GetFileName(string url)
    {
        var file = string.IsNullOrEmpty(url) ? 1 : int.Parse(url) % _maxFileNumber;

        return string.Format("file{0}.txt", file.ToString().PadLeft(3, '0'));
    }

    private async void ProcessRequest(HttpListenerContext context)
    {
        try
        {
            var filename = GetFileName(context.Request.Url.PathAndQuery.Substring(1));
            string receivedData;

            var path = Path.Combine(_directory, filename);
            using (var reader = new StreamReader(path))
            {
                receivedData = await reader.ReadToEndAsync();
            }
        }
    }
}
```

```

        byte[] response;
        if (_isLoadingIncluded)
        {
            var tuple = await SortAsync(receivedData);
            response =
                _encoding.GetBytes(string.Format("file: {0}, length: {1}, time: {2}", filename,
                    tuple.Item1.Length, tuple.Item2));
        }
        else
        {
            response = _encoding.GetBytes(string.Format("file: {0}; without loading",
filename));
        }

        await context.Response.OutputStream.WriteAsync(response, 0, response.Length);
        context.Response.StatusCode = (int)HttpStatusCode.OK;
    }
    catch (Exception e)
    {
        context.Response.StatusCode = (int)HttpStatusCode.BadRequest;
        Console.WriteLine(e.Message);
    }
    finally
    {
        context.Response.Close();
    }
}

private async Task<Tuple<string[], long>> SortAsync(string rawData)
{
    return await Task.Factory.StartNew(() =>
    {
        var array = rawData.Split(new[] { "\r\n" }, StringSplitOptions.RemoveEmptyEntries);
        var timer = new Stopwatch();
        timer.Start();
        QuickSort(array, 0, array.Length - 1);
        timer.Stop();

        return new Tuple<string[], long>(array, timer.ElapsedMilliseconds);
    });
}

static void QuickSort(string[] array, int left, int right)
{
    string temp;
    string x = array[left + (right - left) / 2];
    int i = left;
    int j = right;
    while (i <= j)
    {
        while (array[i].CompareTo(x) == -1) i++;
        while (array[j].CompareTo(x) == 1) j--;
    }

```

```

        if (i <= j)
        {
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
            i++;
            j--;
        }
    }
    if (i < right)
        QuickSort(array, i, right);

    if (left < j)
        QuickSort(array, left, j);
}
}
static class Program
{
    public static void Main(string[] args)
    {
        string address = args[0];
        int maxFileNumber = int.Parse(args[1]);
        bool isLoadingIncluded = bool.Parse(args[2]);
        string directory = args[3];
        Console.WriteLine("http-сервер запущен по адресу {0}", address);
        Console.WriteLine("Сервер ожидает подключений...");
        var program = new Server(address, directory, maxFileNumber, isLoadingIncluded);

        program.Start().Wait();
    }
}

```

TCP-сервер на платформе .NET

```
public class TcpSocketListener
{
    private readonly ManualResetEvent _threadManager = new ManualResetEvent(false);
    private int _port;
    private IPAddress _ipAddress;
    private readonly ContextInfo _contextInfo;

    public TcpSocketListener(int port, IPAddress ipAddress, ContextInfo contextInfo)
    {
        _port = port;
        _ipAddress = ipAddress;
        _contextInfo = contextInfo;
    }

    public void StartListening()
    {
        Console.WriteLine("tcp-сервер запущен");
        Console.WriteLine("ожидание подключения клиентов...");

        var ipEndPoint = new IPEndPoint(_ipAddress, _port);
        var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp)
        {
            ReceiveBufferSize = StateObject.BufferSize,
            SendBufferSize = StateObject.BufferSize
        };

        try
        {
            socket.Bind(ipEndPoint);
            socket.Listen(100);

            while (true)
            {
                _threadManager.Reset();

                socket.BeginAccept(AcceptCallback, socket);
                _threadManager.WaitOne();
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}
```

```

        Console.WriteLine("\nPress ENTER to continue...");
        Console.Read();
    }

    private void AcceptCallback(IAsyncResult ar)
    {
        _threadManager.Set();
        var socket = ((Socket)ar.AsyncState).EndAccept(ar);
        Console.WriteLine("Соединение с клиентом установлено");

        var stateObject = new StateObject { WorkSocket = socket };
        socket.BeginReceive(stateObject.Buffer, 0, StateObject.BufferSize, SocketFlags.None,
        ReceiveCallback, stateObject);
    }

    private void ReceiveCallback(IAsyncResult ar)
    {
        var stateObject = (StateObject)ar.AsyncState;

        var socket = stateObject.WorkSocket;

        int bytesRead = socket.EndReceive(ar);
        if (bytesRead > 0)
        {
            stateObject.ByteReceived += bytesRead;
            socket.BeginSend(stateObject.Buffer.ToArray(), 0, bytesRead, 0, SendCallback,
            stateObject);
        }
    }

    private void SendCallback(IAsyncResult ar)
    {
        try
        {
            var stateObject = (StateObject)ar.AsyncState;
            var sentBytes = stateObject.WorkSocket.EndSend(ar);
            stateObject.ByteSent += sentBytes;

            if (stateObject.ByteSent == _contextInfo.MaxContentlength)
            {
                Console.WriteLine("соединение было закрыто");
                if (stateObject.ByteReceived != stateObject.ByteSent)
                {
                    throw new Exception();
                }
                Console.WriteLine("общее количество байт : {0}", stateObject.ByteReceived);
                stateObject.WorkSocket.Shutdown(SocketShutdown.Both);
                stateObject.WorkSocket.Close();
            }
            else
            {

```

```
        stateObject.WorkSocket.BeginReceive(stateObject.Buffer, 0,
StateObject.BufferSize, 0,
        ReceiveCallback, stateObject);
    }

    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }
}
}
```

UDP-сервер на платформе .NET

```

public class UdpSocketListener
{
    private readonly ContextInfo _contextInfo;
    private readonly ManualResetEvent _threadManager = new ManualResetEvent(false);
    private EndPoint _receiveEndPoint;
    private EndPoint _sendEndPoint;

    public UdpSocketListener(int portForSending, int portForReceiving, IPAddress ipAddress,
        ContextInfo contextInfo)
    {
        _contextInfo = contextInfo;
        _sendEndPoint = new IPEndPoint(ipAddress, portForSending);
        _receiveEndPoint = new IPEndPoint(ipAddress, portForReceiving);
    }

    public void StartListening()
    {
        Console.WriteLine("udp-сервер запущен");
        Console.WriteLine("ожидание подключения клиентов...");

        var socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
            ProtocolType.Udp)
        {
            ReceiveBufferSize = StateObject.BufferSize,
            SendBufferSize = StateObject.BufferSize
        };

        try
        {
            socket.Bind(_receiveEndPoint);

            while (true)
            {
                _threadManager.Reset();
                Console.WriteLine("соединение с клиентом установлено");
                var stateObject = new StateObject { WorkSocket = socket };
                socket.BeginReceiveFrom(stateObject.Buffer, 0, StateObject.BufferSize,
                    SocketFlags.None,
                    ref _receiveEndPoint, ReceiveCallback, stateObject);
                _threadManager.WaitOne();
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}

```

```

        Console.WriteLine("\nPress ENTER to continue...");
        Console.Read();
    }

    private void ReceiveCallback(IAsyncResult ar)
    {
        var stateObject = (StateObject)ar.AsyncState;
        var socket = stateObject.WorkSocket;

        int bytesRead = socket.EndReceiveFrom(ar, ref _receiveEndPoint);
        if (bytesRead > 0)
        {
            stateObject.ByteReceived += bytesRead;
            socket.BeginSendTo(stateObject.Buffer.ToArray(), 0, bytesRead, 0, _sendEndPoint,
SendCallback, stateObject);
        }
    }

    private void SendCallback(IAsyncResult ar)
    {
        try
        {
            var stateObject = (StateObject)ar.AsyncState;
            var sentBytes = stateObject.WorkSocket.EndSendTo(ar);
            stateObject.ByteSent += sentBytes;

            if (stateObject.ByteSent == _contextInfo.MaxContentlength)
            {
                Console.WriteLine("соединение было закрыто");
                if (stateObject.ByteReceived != stateObject.ByteSent)
                {
                    throw new Exception();
                }
                Console.WriteLine("общее количество байт : {0}", stateObject.ByteReceived);
                stateObject.WorkSocket.Shutdown(SocketShutdown.Both);
                stateObject.WorkSocket.Close();
            }
            else
            {
                stateObject.WorkSocket.BeginReceiveFrom(stateObject.Buffer, 0,
StateObject.BufferSize, 0, ref _receiveEndPoint,
                ReceiveCallback, stateObject);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}

```



```

public class StateObject
{
    public Socket WorkSocket;
    public readonly byte[] Buffer = new byte[BufferSize];
    public const int BufferSize = 32768;
    public long ByteReceived = 0;
    public long ByteSent = 0;
}

public class ContextInfo
{
    public int MaxContentlength
    {
        get { return Iterations * SizeOfMessage + GreetingMessagelength; }
    }
    public int GreetingMessagelength { get; set; }
    public int SizeOfMessage { get; set; }
    public int Iterations { get; set; }
}

```

Класс создания текстовых файлов для загрузки сервера

```
public class DataCreator
{
    public void CreateFiles(int fileNumber, string directory)
    {
        for (int i = 0; i < fileNumber; ++i)
        {
            byte[] data = Generate();
            var fileName = string.Format("file{0}.txt", i.ToString().PadLeft(3, '0'));
            var path = Path.Combine(directory, fileName);
            using (var stream = File.Open(path, FileMode.OpenOrCreate))
            {
                stream.Write(data, 0, data.Length);
            }
        }
    }

    private byte[] Generate()
    {
        var encoding = new ASCIIEncoding();
        var random = new Random((int)(DateTime.UtcNow.Ticks % Int32.MaxValue) + 1);
        var data = new StringBuilder();

        for (long i = 0; i < 30000; ++i)
        {
            data.AppendLine(random.NextDouble().ToString("F7"));
        }
        data.Append(random.NextDouble().ToString("F7"));

        return encoding.GetBytes(data.ToString());
    }
}
```