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 языках node js и C#

Научный руководитель

\_\_\_\_\_\_\_\_\_\_\_ О.В.Власова

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2016 г.

Студент группы КБ-41

\_\_\_\_\_\_\_\_\_\_\_\_ С.А.Попов

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2016 г.

Ярославль 2016

Содержание

# Введение

**TCP**

TCP - 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:

* отслеживание переданных сегментов данных
* подтверждение полученных данных
* 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 АСК, 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.

**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 — 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.

В средах .net и node js взаимодействия по протоколам tcp и udp происходит 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".

**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.

**Типы сокетов**

**Потоковый сокет.**

Потоковый 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.

Потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, сдублированных данных. Потоковый 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. Потоковые сокеты достигают этого уровня качества за счет использования протокола ***Transmission Control Protocol (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 TCP 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 TCP определенное число байтов и другая сторона получает эти байты. В свою очередь TCP 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. Потоковый сокет рассчитывает, что TCP обеспечит доставку сообщений по их назначениям.

**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 ***User Datagram Protocol (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.

**Порты**

Порт определен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. По существу с его помощью расширяется понятие IP-адреса. Компьютер, на котором в одно время выполняется несколько приложений, получая пакет из сети, может идентифицировать целевой процесс, пользуясь уникальным номером порта, определенным при установлении соединения.

Сокет состоит из IP-адреса машины и номера порта, используемого приложением TCP. 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.

**Сравнение производительности сокетов среды .net и node js для протокола tcp**

Для реализации tcp-сервера на языке C# 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 языке node js работает по такому же принципу. Мы также читаем 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 языках программирования. Обе реализации сокетов являются асинхронными, что не останавливает поток выполнения программы. Замерять будет клиент, написанный на языке node js. Он посылает n-количество раз сообщение длинной 32768 и 16784 байт. Клиент не отправляет следующее сообщение, пока не получен ответное сообщение от сервера в таком же объёме. Варьируя количество запросов мы получаем функция, зависящую от размера сообщения и количества запросов. На рисунке 1 и 2 приведены графики для размеров сообщения длинной 32768 и 16784 байт и количества запросов от 64 до 4096.

Результаты показывают что время ответа для платформы node js значительно больше, по сравнению с .net. Это вызвано тем, что реализация потоков ещё находится в разработке и каждый день разработчики node js улучшают процесс работы с памятью. Также это вызвано тем, что при проверки скорости ответа сервера node js пришлось отойти от стандартной модели записи и чтения из потока, как это сделано для .net.

Из результатов можно сделать вывод, что node js уступает в скорости обработки запроса, из-за несовершенной реализации потоков. Запись байтов в память в .net быстрее, чем в node js.

Рисунок 1

Рисунок 2

**Сравнение производительности сокетов среды .net и node js для протокола udp**

Реализация сокетов для проткола 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 A, и слушает порт B. 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 3 и 4 показаны результаты работы протоколов udp для node js и .net.

Из результатов видно, что скорость ответа практически не отличается, за исключением что node js при увеличении количества запросов начинает отставать от .net. Это объясняется тем, что реализация работы с потокоми для udp сервера на node js отличается от реализации tcp. В данном случае в node js в обработчике события для присланное сообщение мы получаем уже готовый буфер, а не считываем его в цикле из потока, как это сделано для tcp.

В итоге скорость ответа серверов node js и .net для протоколов транспортного уровня udp практически не отличается.

Рисунок 3

Рисунок 4

**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 инициируют 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.

**Сравнение многопоточного и однопоточного режимов 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 может уступать многим другим языкам с поддержкой асинхронных вызовов. Но к счастью есть выход из данной ситуации. В языке 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 чистоты 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 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 от 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 компьютере. Каждый запрос к сервере будет попадать в один из созданных потоков. Без кластера сервер будет обрабатывать только запросы только в однопоточном режиме. На рисунке 5 представлен график работы сервера node js в многопоточном и однопоточном режимах.

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.

Рисунок 5

**Рисунок 6**

**Рисунок 7**

# 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 T., 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/. -URL: https://nodejs.org/dist/latest-v6.x/docs/api/
6. http://stackoverflow.com/. -URL: http://stackoverflow.com/questions/9290160/node-js-vs-net-performance

# Приложение

**Листинг 1**

udp-клиент на языке node js

'use strict'

const dgram = require("dgram")

const client = dgram.createSocket("udp4")

const options = {

portForReceiving: 3333,

portForSending: 3334,

iterations: 2048,

sizeOfMessage: 32768,

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(`total: ${totalLength}`)

console.log(`work time - ${(begin \* 1e9 + start) / 1000000} ms`)

console.log('Connection was closed for client')

process.exit()

})

client.bind(options.portForReceiving, options.host)

time = process.hrtime()

sendMessage(new Buffer('hello'))

**Листинг 2**

udp-сервер на языке node js

'use strict'

const dgram = require('dgram');

const server = dgram.createSocket('udp4');

const options = {

portForReceiving: 3334,

host: "127.0.0.1",

iterations: 2048

}

const sendMessage = (message, info) => server.send(message, 0, message.length, info.port, info.address);

let lengthContent = 0, iterationNumber = 0

server.on('close', () => console.log('Connection was closed for server'))

server.on('message', (message, info) => {

lengthContent += message.length

iterationNumber++

sendMessage(message,info)

if (iterationNumber > options.iterations) {

console.log(`total: ${lengthContent}`)

lengthContent = 0

iterationNumber = 0

}

})

server.on('error', (err) => {

server.close()

throw err;

})

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

**Листинг 3**

tcp-клиент на языке node js

'use strict'

const net = require('net');

const client = new net.Socket();

const options = {

sizeOfMessage: 32768,

iterations: 2048,

port: 1337,

host: '127.0.0.1'

}

const message = Buffer.alloc(options.sizeOfMessage, '1', 'utf-8');

let contentLength = 0,

iterationNumber = 0,

time

client.on('readable', () => {

let chunk;

while (null !== (chunk = client.read())) {

contentLength += chunk.length

}

iterationNumber++

iterationNumber <= options.iterations && client.write(message)

})

client.on('end', () => {

let [begin,end] = process.hrtime(time)

console.log(`total client: ${contentLength}`)

console.log(`work time - ${(begin \* 1e9 + end) / 1000000} ms`)

console.log('Connection was closed')

})

client.connect(options.port, options.host, () => {

console.log('client connected');

time = process.hrtime()

client.write('hello')

})

**Листинг 4**

tcp-сервер на языке node js

'use strict'

const net = require('net');

const options = {

maxContentLength: 32768 \* 2048 + 5

}

const client = new net.Socket();

let contentLength = 0

const server = net.createServer((client) => {

console.log('client connected')

client.on('close', () => {

console.log(`total server: ${contentLength}`)

console.log('Connection was closed')

contentLength = 0

})

client.on('readable', function () {

let chunk,

buffer = null

while (null !== (chunk = client.read())) {

buffer = buffer === null

? Uint8Array.from(chunk)

: Uint8Array.concat(buffer, chunk)

}

buffer !== null && (contentLength += buffer.length)

if (contentLength < options.maxContentLength) {

return client.write(new Buffer(buffer))

}

buffer !== null && client.end(new Buffer(buffer))

})

})

server.on('error', (err) => {

server.close()

throw err;

})

server.listen(1337, () => console.log('"Waiting for a connection...'))

**Листинг 5**

http сервер на языке 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, fileNumber, 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(`Server running at http://127.0.0.1:${port}/`)

}

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()

}

**Листинг 6**

http-клиент на языке C#

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()

{

Console.WriteLine();

Console.WriteLine("Start sending");

var timer = new Stopwatch();

timer.Start();

var tasks = new Task[\_tasks];

for (int i = 0; i < \_tasks; ++i)

{

tasks[i] = Perform(i);

}

Task.WaitAll(tasks, -1);

timer.Stop();

//\_result.ToList().ForEach(Console.WriteLine);

Console.WriteLine();

Console.WriteLine("tasks: {0} time: {1}", \_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();

timer.Start();

var stringResult = await client.GetStringAsync(url);

timer.Stop();

\_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];

for (var i = 100; i <= 500; i+=100)

{

var client = new Client(address, i);

client.Start();

}

Console.ReadLine();

}

}

**Листинг 7**

http-сервер на языке C#

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];

var program = new Server(address, directory, maxFileNumber, isLoadingIncluded);

program.Start().Wait();

}

}

**Листинг 8**

tcp-сервер на языке С#

public class SocketListener

{

private readonly ManualResetEvent \_threadManager = new ManualResetEvent(false);

private int \_port;

private IPAddress \_ipAdress;

public SocketListener(int port, IPAddress ipAdress)

{

\_port = port;

\_ipAdress = ipAdress;

}

public void StartListening()

{

var ipEndPoint = new IPEndPoint(\_ipAdress, \_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();

Console.WriteLine("Waiting for a connection...");

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("Client connected");

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 == StateObject.MaxContentlength)

{

Console.WriteLine("Connection was closed");

if (stateObject.ByteReceived != stateObject.ByteSent)

{

throw new Exception();

}

Console.WriteLine("total : {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());

}

}

}

**Листинг 9**

udp-сервер на языке C#

public class UdpSocketListener

{

private readonly ManualResetEvent \_threadManager = new ManualResetEvent(false);

private EndPoint \_receiveEndPoint;

private EndPoint \_sendEndPoint;

public UdpSocketListener(int portForSending, int portForReceiving, IPAddress ipAdress)

{

\_sendEndPoint = new IPEndPoint(ipAdress, portForSending);

\_receiveEndPoint = new IPEndPoint(ipAdress, portForReceiving);

}

public void StartListening()

{

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("Waiting for a connection...");

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 == StateObject.MaxContentlength)

{

Console.WriteLine("Connection was closed");

if (stateObject.ByteReceived != stateObject.ByteSent)

{

throw new Exception();

}

Console.WriteLine("total : {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 const int MaxContentlength = 2048 \* 32768 + 5;

public long ByteReceived = 0;

public long ByteSent = 0;

}

**Листинг 10**

Класс создания текстовых файлов для нагрузки 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());

}

}