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

**Платформа .NET**

.NET Framework — программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая 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 общеязыковой среды выполнения (среды CLR) и библиотеки классов .NET Framework. Основой платформы .NET Framework является среда CLR. Среду 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.

**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 методаBeginOperationName 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, будет содержать один или несколько методовMethodNameAsync. 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. Из-за длинных цепочек асинхронных 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). Эта библиотека усовершенствует 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.

Библиотека TPL 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 фреймворка 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.

**Node.js**

Node.js — 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, РНР, 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.

**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.

**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:

* 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(User Datagram Protocol) - 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".

**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.

**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.

**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 (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.

**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.

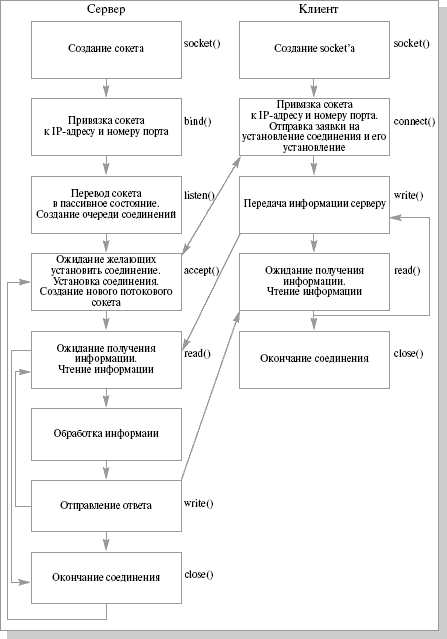


Рисунок 1

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

**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.

**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.

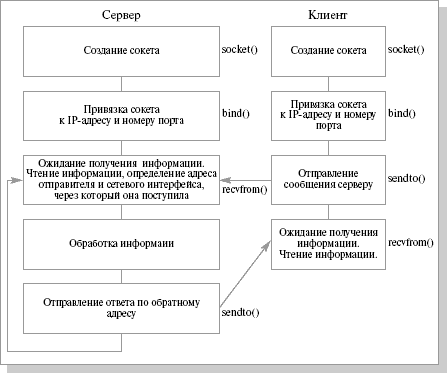


Рисунок 1

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

**Сравнение производительности сокетов среды .NET и Node.js 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. 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 платформе 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 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 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

**Сравнение Node.js и .NET HTTP-серверов запущенных в многопоточных режимах**

Не так давно на платформе .NET появилась технология вызова асинхронных методов с помощью ключевых слов "async/await". Её цель - упростить асинхронные вызовы функций и процедур и заменить старую методику, использующую функции обратного вызова.

Ключевое слово "async" указывает, что метод или лямбда-выражение может выполняться асинхронно. А оператор "await" позволяет остановить текущий метод, пока не завершится работа метода, помеченного как "async", не останавливая выполнение потока. Эти операторы используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора "async" и содержащий одно или несколько выражений "await", называется асинхронным методом.

В данном эксперименте мы будем осуществлять запросы клиента к сервера, загружая сервер сортировкой массива из 30000 чисел. На рисунке 6 представлен график работы Node.js и .NET HTTP-сервера. Для Node.js мы выбрали многопоточный подход с использованием кластера. В .NET мы слушаем заранее оговорённый порт и любой запрос обрабатывается асинхронным способом.

Исходя из результатов данного эксперимента мы видим, что Node.js значительно опережает .NET. Это связанно 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.

**Рисунок 6**

**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 и заключается развертывание 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 котроллером 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 6.

Из результатов эксперимента мы можем убедиться, что Node.js справляется с нагрузкой гораздо лучше, чем .NET.

**Рисунок 6**

**Заключение**

# 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 платформе 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 платформе 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 платформе 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 платформе 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 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, 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 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()

{

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 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];

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

program.Start().Wait();

}

}

**Листинг 8**

TCP-сервер на платформе 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 платформе .NET

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());

}

}