электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 04, апрель 2016

УДК 004.4'2

Разработка высокопроизводительных сетевых приложений на основе Netty

***Ларионов В.С.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю.Е., к.т.н., доцент,
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
gapyu@bmstu.ru

Введение

Довольно часто разработчики программного обеспечения сталкиваются с потребностью написания высокопроизводительных сетевых приложений, что является нетривиальной задачей. Для работы с сетью существуют различные библиотеки, для увеличения производительности используются несколько потоков выполнения, так называемая многопоточность, запуск разных процессов (многопроцессорность), различные вариации данных методов и другое.

Всё это может реализовать сам разработчик. Но будет ли данная реализация корректно работать, показывать высокую производительность, оптимально использовать все ресурсы системы и сколько это займёт времени?

Все вышеописанные потребности удовлетворяет *Netty*. *Netty* – это клиент-серверный асинхронный неблокирующий событийно-ориентированный фреймворк для работы с сетью, который предоставляет разработчику простые, безопасные элементы управления.

В статье описываются возможности и особенности фреймворка, который возьмёт на себя непосредственную работы с сокетами и реализацию многопоточной обработки событий. В качестве демонстрации преимуществ был написан простой *HTTP*-сервер для отдачи статического контента.

Для разработчика *Netty* представляет собой множество абстракций, инкапсулируя внутреннюю реализацию своих библиотек. Рассмотрим составляющие компоненты более подробно.

Channel

Channel на уровне абстракций представляет собой транспортный канал, по которому передаются данные. Если же говорить о реализации, то это интерфейс языка программирования *Java* (далее интерфейс), который предоставляет связь с сетевым сокетом, файлом либо каким-то аппаратным устройством с возможностями асинхронного ввода/вывода. *Channel* определяет семантику для взаимодействия с сокетом богатым набором методов. Основными из них являются *bind(), close(), connect(), disconnect(), flush(), isActive(), isOpen(), isWritable(), pipeline(), read()* и *write().*

Асинхронность через *Callbacks* и *Futures*

Так как все операции ввода/вывода в *Netty* являются асинхронными, то результат вызова этих операций возвращается сразу, без гарантии того, что запрошенная операция выполнена. Возврат результатов вычислений (обработки событий) в *Netty* происходит двумя способами: через функции обратного вызова, так называемые *callback*-и, либо через специальный объект *future*.

Использование функций обратного вызова является простейшим механизмом для оповещения о каком-либо событии. Например, можно настроить обработчик событий, который будет оповещать о новых подключениях к серверу либо о входящих запросах. Таким образом отпадает необходимость в опросе сетевого сокета на наличие данных.

Объект *future* представляет более улучшенный механизм асинхронного получения данных. Стоит отметить, что *future* внутри себя реализует идею функций обратного вызова, расширяя их возможности. Расширение заключается в том, что в любой момент можно узнать, завершён ли процесс обработки, либо получить промежуточный результат. Причём всё это происходит в неблокирующем режиме. С точки зрения реализации, *future* – это интерфейс с множеством базовых методов. Основными из них являются *addListener(), await(), cause(), getNow(), isDone, isSuccess(), removeListener(), sync().*

От интерфейса *Future* наследуются такие интерфейсы, как *ChannelFuture*, *ChannelProgressiveFuture*, *ChannelPromise* и т.д. Каждый из них, является результатом асинхронной операции ввода/вывода при работе с каналом.

Обработчики *ChannelHandlers*

Обработчик представляет собой абстракцию, позволяющую реализовать какую-либо логическую часть программы. Рассмотрим это на примере веб-сервера. Пусть один обработчик будет производить расшифровку данных, переданных по протоколу *HTTPS*,

другой – разбор *HTTP*-заголовков, третий – формирование ответа от сервера к клиенту. Последний из обработчиков будет шифровать данные и возвращать ответ клиенту по протоколу *HTTPS* через сетевой сокет. Таким образом, каждый обработчик является логически и функционально обособленной частью декомпозиции сетевого приложения.

От интерфейса *ChannelHandler* наследуются два широко применяемых интерфейса: *ChannelInboundHandler* и *ChannelOutboundHandler*. Класс, который реализует интерфейс *ChannelInboundHandler* будет обрабатывать только входящие события, *ChannelOutboundHandler* – только исходящие. События между обработчиками передаются с помощью конвейера (*pipeline*).

Конвейер

Конвейер (*Pipeline*) в *Netty* – это абстракция, позволяющая организовать взаимодействие между обработчиками. *ChannelPipeline* представляет собой контейнер для *ChannelHandlers* с *API* для доступа к обработчикам. Каждый канал имеет свой собственный конвейер и создаётся автоматически при создании нового канала. Рассмотрим реализацию конвейера, изображённую на рис. 1.

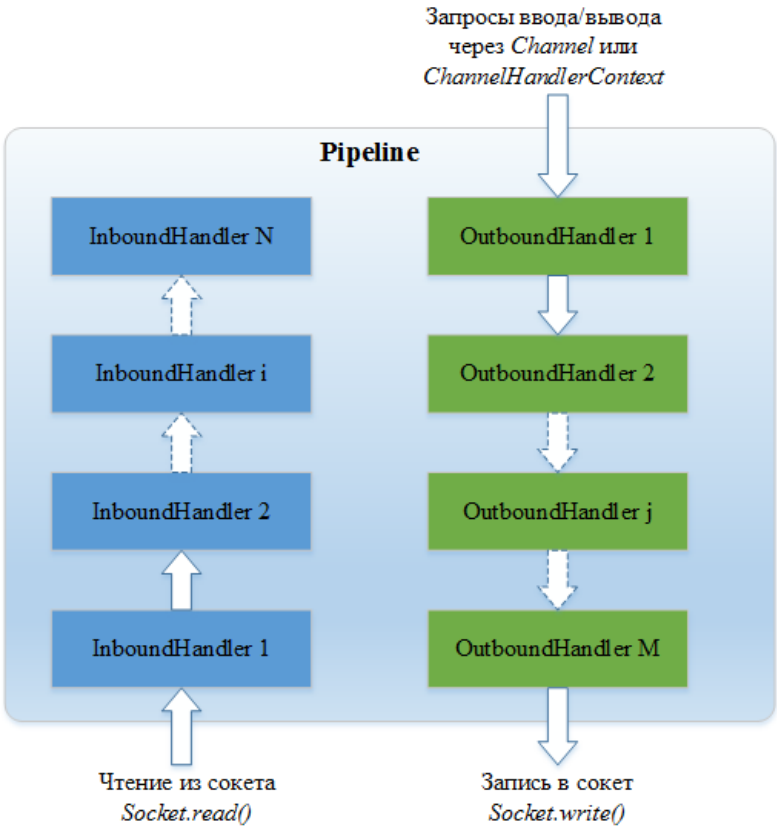

Рис. 1. Схема реализации конвейера в Netty

Как описывалось ранее, обработчики делятся на обработчиков входящих событий (*InboundHandlers*), исходящих (*OutboundHandlers*), а также существуют дуплексные обработчики (*ChannelDuplexHandler*). Обработчики соединены между собой в двусвязный список. Запрос, поступающий от клиента, последовательно обрабатывается обработчиками входящих сообщений, ответ от сервера передаётся в цепь обработчиков исходящих сообщений. Двусвязный список позволяет не только передавать данные между обработчиками в установленном порядке, но и пробрасывать возникающие исключения на предыдущие уровни. Поскольку *ChannelPipeline* является потокобезопасным, обработчики могут быть добавлены в конвейер или удалены из него в любое время.

Таким образом, по мнению автора статьи, разработчику предоставляется возможность декомпозиции бизнес-логики сетевого приложения с возможностью реализации удобного взаимодействия между составными частями.

Буферы для передачи данных

Для передачи данных по конвейеру в *Netty* используется класс *ByteBuf*. *ByteBuf* – это контейнер для байт данных. Для эффективной работы с данными в нём используется два индекса: для чтения – *readerIndex* и для записи – *writerIndex*. При аллокации участка памяти изначально эти два индекса установлены в нулевое положение. После записи N байт в данный участок памяти индекс *writerIndex* инкрементируется на N единиц, после чтения M байт индекс readerIndex увеличивается на M единиц, причём $M \leq N$.

В *Netty* существует 3 вида буферов: *HeapBuf*, *DirectBuf* и *CompositeByteBuf*. В *HeapBuf* память под элементы аллоцируется на куче пространства *JVM*. Плюсами данного вида являются быстрое выделение или перевыделение памяти, а также возможность получения ссылки на байтовый массив, инкапсулируемый внутри класса. *DirectBuf* аллоцирует память на стеке *JVM*, является удобным для работы с сокетами, так как при использовании *HeapBuf* перед отправкой данных в сокет будет происходить их копирование в *DirectBuf*. Недостатками относительно предыдущего вида буферов являются дорогое выделение (перевыделение) памяти, невозможность получить ссылку на массив с данными в виду внутренней реализации. *CompositeByteBuf* способен комбинировать в себе оба перечисленных вида.

Помимо классификации по области выделения памяти, выделяют буферы по способу выделения памяти: *pooled* и *unpooled*. *Unpooled* буфер при вызове *ByteBufAllocator.buffer* каждый раз выделяет новый участок памяти, при использовании

pooled буфера *Netty* будет стараться объединить уже существующие буферы, тем самым сводя к минимуму накладные расходы на выделение и освобождение памяти.

Несомненным преимуществом *ByteBuf* является поддержка технологии *zero copy*, суть которой заключается в исключении промежуточного копирования данных из одной области памяти в другую. В примитивной реализации веб-сервера файл, отправляемый клиенту, сначала копируется с диска (*HDD*, *SSD*) в буфер приложения, а затем уже отправляется в буфер сетевого сокета, откуда уже данные направляются непосредственно в буфер сетевой карты. Недостатком данного подхода является не только использование дополнительной памяти, но и переключение контекста с режима ядра в режим пользователя и обратно. При использовании технологии *zero copy* отпадает необходимость в копировании данных в пользовательский буфер. Реализацией данной технологии в *Java* является метод *transferTo()*, который передаёт данные из файлового канала в записываемый байтовый канал. В *UNIX*-подобных системах вызов метода *transferTo()* обращается к системному вызову *sendfile()*, который передаёт данные из одного файлового дескриптора, в другой. Таким образом, всё копирование производится в контексте ядра.

Класс *Bootstrap*

Для простой организации взаимодействия между обработчиками, каналами, конвейерами в *Netty* реализованы специальные классы *Bootstrap* и *ServerBootstrap*. Класс *Bootstrap* применяется для клиентский приложений, *ServerBootstrap* – для серверных. Благодаря унификации методов, в следствии наследования абстрактного класса *AbstractBootstrap*, разработчику предоставляется возможность исключения дублирования кода. Классы содержат множество методов для индивидуальной настройки каждого из компонентов.

Модель взаимодействия потоков

Архитектура *Netty* основана на шаблоне проектирования “реактор”. Как описывает автор в статье [3], данный паттерн позволяет разделить логику работы приложения на модули, каждый из которых вызывается на определённое, связанное с ним, событие, в отличии от классического варианта написания последовательно исполняемого огромного куска кода. Таким образом, код представляет собой множество блоков, задачей которых является выполнение каких-либо событий.

Реализация паттерна реактор производится через интерфейсы *EventLoop* и *EventLoopGroup*. Обобщённо можно сказать, что реализация *EventLoop* представляет собой цикл обработки событий, который при возникновении определённых событий выполняет поставленные перед ним задачи, т.е. является потоком выполнения. Задачами в случае реализации веб-сервера являются обработка *HTTP*-запроса и отдача ответа клиенту.

Когда поступают данные (это могут быть как входящие события от клиента, так и исходящие от сервера), поток выполнения “просыпается”, выполняет функции для конкретного типа события, зарегистрированные через *callback*, и возвращается в состояние ожидания до тех пор, пока не произойдёт новое событие.

Преимущество выполнения задач в цикле событий заключается в том, что не нужно заботиться о какой-либо синхронизации. Каждое событие выполняется лишь в единственном потоке, независимом от других, которые также могут обрабатывать соответствующий канал.

EventLoopGroup объединяет в себе несколько реализаций *EventLoop* и позволяет регистрировать каналы (*Channel*), которые в последующем будут обработаны в цикле обработки событий. Количество *EventLoop*-ов (потоков) в *EventLoopGroup* зависит от конфигурации и доступных ядер процессора. Например, в реализации *NioEventLoopGroup*, которая далее используется для написания веб-сервера, количество потоков определяется как удвоенное количество ядер.

С точки зрения реализации серверной части приложения в *Netty* различают 2 вида потоков: **родительские** и **дочерние**. Каждый родительский поток обрабатывает один канал. Например, если сервер слушает 80 порт для приёма запросов по протоколу *HTTP*, а 443 для *HTTPS*-запросов, то *EventLoopGroup* создаст 2 реализации *EventLoop*. Оба этих потока будут являться родительскими.

Входящее соединение передаётся на обработку дочерним потокам с помощью потокобезопасной очереди, благодаря которой производится последовательное выполнение входящих событий. Дочерние потоки используются для исполнения логики приложения с одним или несколькими каналами в неблокирующем режиме. Чтобы не блокировать цикл обработки событий, все блокирующие операции (например, доступ к базе данных) должны выполняться в отдельном потоке.

При реализации клиентской части приложения отпадает необходимость в родительских потоках и используются только дочерние.

Дополнительные возможности

Помимо мощнейших компонентов, описанных ранее, для разработки сетевых приложений в *Netty* существует множество уже готовых реализаций. К ним можно отнести поддержку широко используемых протоколов передачи данных транспортного и прикладного уровней модели *OSI*. К ним относятся *TCP*, *UDP*, *SCTP* – на транспортном уровне; *HTTP* (1.0, 1.1, 2.0), *SPDY*, *DNS*, *UDT* – на прикладном уровне. Помимо этого существуют обработчики для работы с защищёнными соединениями по протоколам *SSL/TLS*, поддержка сжатия данных с использованием *zlib/gzip*, веб-сокеты, а также кодеки для работы с *JBoss Marshaling*, сериализацией *Java*, кэшированием данных в оперативной памяти с использованием *Memcached*, текстоориентированным протоколом *Stomp*, языком описания данных *Protobuf* и т.д.

Пример реализации

Для представления возможностей фреймворка автором статьи был написан элементарный веб-сервер для отдачи статического контента [6]. Для тестирования корректной работы сервера использовался программный продукт [7], в документации которого сформированы требования к веб-серверу. Работа с *HTTP*-протоколом реализована автором без использования встроенных возможностей *Netty* (кодеков) для демонстрации гибкости конфигурации и обработки различных данных.

Для обработки *HTTP*-запросов от клиентов было создано два обработчика событий: *HttpRequestDecoder* и *ServerHandler*. Рассмотрим более подробно каждый из них.

Класс *HttpRequestDecoder* наследует класс *ChannelInboundHandlerAdapter*, который имеет уже базовую реализацию многих методов, описанных в интерфейсе *ChannelInboundHandler*. Для обработки входящих сообщений был переопределён метод *channelRead*().

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    /* Обработка входящих данный
    * и формирование объекта класса HttpRequest */
    ctx.fireChannelRead(httpRequest);                // 1
}
```

1 – передача объекта *httpRequest* по конвейеру следующему обработчику событий.

Сформированный объект *httpRequest* передаётся по конвейеру обработчику ServerHandler, задачей которого является формирование и отправка ответа от сервера.

```
public class ServerHandler
               extends SimpleChannelInboundHandler<HttpRequest> {    // 1

    @Override
    public void channelRead0(ChannelHandlerContext ctx, HttpRequest
httpRequest) {                                                       // 2
```

```
        /* Обработка объекта httpRequest
        * и формирование кода HTTP-ответа от сервера
        */
        if (responseCode == ResponseCodes.OK) {
        /* Формирование объекта класса HttpResponse */

            FileInputStream in = new FileInputStream(pathToFile);    // 3
            FileRegion region = new DefaultFileRegion(               // 4
                    in.getChannel(), 0, pathToFile.toFile().length()
            );
            /* Часть кода опущена */
            sendResponse(ctx, response, region);                     // 5
        }

        else {
                /* Формирование HTML-страницы с ошибкой
                * (например, 404) и отправка ответа с помощью
                * метода sendResponse
                */
        }
    }

    private void sendResponse(ChannelHandlerContext ctx, HttpResponse
response, FileRegion region) {
        Channel channel = ctx.channel();
        channel.eventLoop().execute(() -> {                          // 6
            channel.write(response.toByteBuf());                     // 7
            channel.writeAndFlush(region)                            // 8
                .addListener(new ChannelFutureListener() {           // 9
                    @Override
                    public void operationComplete(ChannelFuture future)
                     throws Exception {
                      future.addListener(ChannelFutureListener.CLOSE);
                 }
            });
        });
    }
}
```

1 – наследование обобщённого класса *SimpleChannelInboundHandler<HttpRequest>* позволяет данному обработчику принимать из конвейера только объекты типа *HttpRequest*. Помимо этого, благодаря наследованию в данном классе уже реализовано множество методов для работы с конвейером;

2 – метод *channelRead0()* вызывается, когда сообщение полностью получено данным обработчиком из конвейера. В версии Netty 5.0 данный метод переименован в *messageReceived* с этой же сигнатурой;

3 – создание файлового потока для отправляемого клиенту файла;

4 – для передачи файла используется класс *FileRegion*, который осуществляет отправку данных с помощью технологии *zero copy*;

5 – отдельный метод для отправки данных, в текущем случае, в буфер сетевого сокета;

6 – по рекомендации Нормана Маурера, одного из разработчиков *Netty*, [5], комбинирование операций позволяет сократить накладные расходы на пробуждение и создание потоков;

7 – отправка ответа через конвейер;

8 – отправка ответа через конвейер с последующей записью информации в буфер сетевого сокета;

9 – добавление обработчика, который закроет сетевое соединение с клиентом после завершения отправки файла.

Для организации взаимодействия между обработчиками, конвейером, каналом был использован класс *ServerBootstrap*.

```
public void run() throws Exception {                                    // 1
    EventLoopGroup group = new NioEventLoopGroup();                     // 2
    try {
        ServerBootstrap b = new ServerBootstrap();                      // 3
        b.group(group, group)                                           // 4
         .channel(NioServerSocketChannel.class)                         // 5
         .childHandler(new ChannelInitializer<SocketChannel>() {
             @Override
             public void initChannel(SocketChannel channel) throws Exception
{
                 channel.pipeline()
                         .addLast(new HttpRequestDecoder())             // 6
                         .addLast(new ServerHandler());
             }
         })
        .childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
        .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);// 7

        // Старт сервера
        ChannelFuture f = b.bind(inetHost, port).sync();

        // Ожидать, пока сокета сервера не будет закрыт
        f.channel().closeFuture().sync();
    }
    finally {
        group.shutdownGracefully();                                     // 8
    }
}
```

1 – метод *run()* вызывается у объекта класса сервера, в данном случае *Main*, для запуска веб-сервера;

2 – в качестве *EventLoopGroup* автор статьи использует класс *NioEventLoopGroup*, который реализован в *Netty* на основе *Java NIO*;

3 – экземпляр класса *SeverBootstrap* обеспечивает быструю настройку и взаимодействие различных компонент *Netty*;

4 – поскольку веб-сервер работает лишь на одном порте, то целесообразней использовать одну группу для родительского и дочерних потоков;

5 – установка класса, который используется для создания каналов;

6 – добавление обработчиков в конвейер. Стоит отметить, что обработчики будут вызываться в том порядке, в котором они добавлены в конвейер, таким образом важен порядок их добавления;

7 – для большей производительности используется буфер типа *pooled*;

8 – корректное освобождение ресурсов, занимаемых сервером.

Результаты

Целью написания данного веб-сервера является демонстрация быстрой и гибкой разработки многопоточных сетевых приложений. Поскольку данное приложение является специфичным, то в нём отсутствует возможность различных настроек веб-сервера, которые есть в современных серверах. Тем не менее, ниже приведено сравнение производительности с такими серверами, как *Nginx*, *Vert.x* на *Java*, *Node.js* в их стандартной конфигурации. Стоит подчеркнуть, что тестирование производительности производилось лишь при отдаче статического контента. Выбор экземпляров производился следующим образом: веб-сервер *Nginx* выбран как популярный, высокопроизводительный веб-сервер; фреймворк *Vert.x* построен на основе *Netty* с собственной реализацией веб-сервера; очевидно, что *Node.js* не разрабатывался как веб-сервер для отдачи статического контента, но довольно часто производится сравнение его производительности с Vert.x, как в статье [4].

Тестирование производительности производилось утилитой *Apache Benchmark* [9] на двух различных файлах: в первом случае на локально сохранённой *html*-странице [8], во втором – на текстовом файле, содержащем сообщение “*Hello world*!”. Для каждого теста было выполнено 100 000 запросов при 100 одновременных соединениях. На тестируемом компьютере установлен процессор Intel Core i3 2350M тактовой частоты 2,30 GHz, объём оперативной памяти составляет 8096 МБ, операционная система – Ubuntu 14.04 LTS.

На рис. 2 и 3 представлена производительность каждого из экземпляров, измеряемая в количестве ответов от сервера за секунду (англ. *rps – request per second*). В качестве значения выбран наилучший из трёх последовательных запусков.

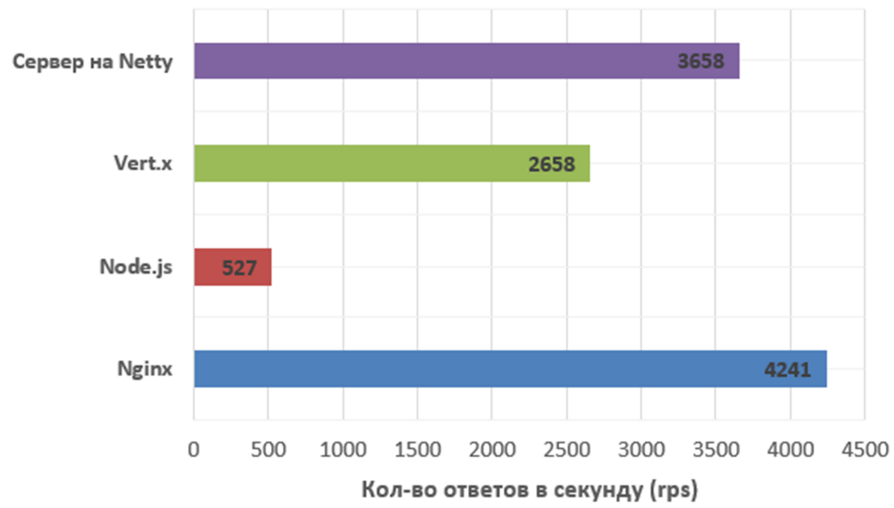

Рис. 2. Количество ответов от тестируемых веб-серверов на html-странице [8]

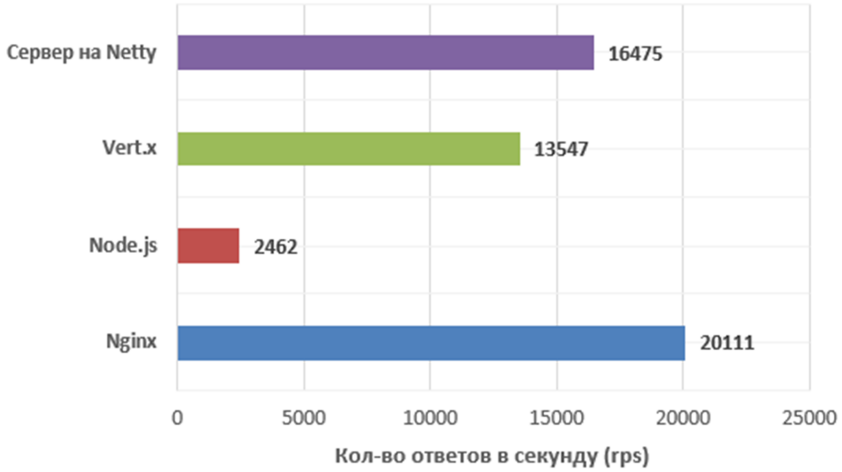

Рис. 3. Количество ответов от тестируемых веб-серверов на текстовом файле с содержимым “Hello world!”

Как можно заметить, реализация веб-сервера для отдачи статического контента на основе фреймворка Netty показывает довольно неплохие результаты. Таким образом, Netty позволяет создавать высокопроизводительные сетевые приложения, реализуя всю сетевую и многопоточную работу.

Заключение

В статье рассмотрены основные составляющие элементы фреймворка для написания многопоточных сетевых приложений на языке программирования *Java – Netty*. Также реализован элементарный веб-сервер для отдачи статического контента, результаты работы которого показали хорошую производительность *Netty*. Благодаря простой конфигурации и высоким показателям данный фреймворк применяется в таких крупных

компаниях, как *Google*, *Yahoo*, *RedHat*, *Facebook*, *Cisco*, *Instagram*, *Twitter* и многих других.

Список литературы

[1]. Official site of the project Netty. Available at: http://netty.io/, accessed 13.09.2015.

[2]. Norman Maurer. Netty in action. Manning Early Access Program, version 8, 2014, 302p.

[3]. Что такое «асинхронная событийная модель», и почему сейчас она «в моде». Режим доступа: http://habrahabr.ru/post/128772/ (дата обращения 12.09.2015).

[4]. Woo Seongmin. Inside Vert.x. Comparison with Node.js. Available at: http://www.cubrid.org/blog/dev-platform/inside-vertx-comparison-with-nodejs, accessed: 13.09.2015.

[5]. Palo Alto. Netty best practices. Available at: http://normanmaurer.me/presentations/2014-facebook-eng-netty/slides.html#28.0, accessed 13.09.2015.

[6]. Larionov Vadim. HttpServer. Available at: https://github.com/Larionov-Vadim/HttpServer, accessed 13.09.2015.

[7]. Bykov Alexander. Http-test-suite. Available at: https://github.com/init/http-test-suite, accessed 13.09.2015.

[8]. Свободная энциклопедия Википедия, статья «Россия». Режим доступа: https://ru.wikipedia.org/wiki/Россия (дата обращения 13.09.2015).

[9]. Apache HTTP server benchmarking tool. Available at: http://httpd.apache.org/docs/2.2/programs/ab.html, accessed 13.09.2015.