Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ
Зав.кафедрой,
к.ф.-м.н.
_____ С. В. Миронов

ОТЧЕТ О ПРАКТИКЕ

Студента 4 курса 411 группы факультета КНиИТ
Низамутдинов Артура Салаватовича

вид практики: преддипломная
кафедра: математической кибернетики и компьютерных наук
курс: 4
семестр: 8
продолжительность: 4 нед., с 06.05.2016 г. по 02.06.2016 г.

Руководитель практики от университета,

зав. кафедрой, к. ф.-м. н. _____ С. В. Миронов

Руководитель практики от организации (учреждения, предприятия),

зав. кафедрой, к. ф.-м. н. _____ С. В. Миронов

Тема практики:«Разработка высоконагруженных приложений на языке JavaScript.»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В 2009 году Райан Дал после двух лет экспериментирования над созданием серверных веб-компонентов разработал NodeJS.

NodeJS является программной платформой, основанной на разработанном компанией Google JavaScript–движке V8 (транслирующем JavaScript в машинный код), преобразующий JavaScript из узкоспециализированного в язык общего назначения. NodeJS используется преимущественно на сервере, выполняя роль веб-сервера, но кроме этого есть возможность разрабатывать на NodeJS и десктопные приложения (с использованием NW.js, Electron или AppJS для Windows, Linux и Mac OS) и даже программировать микроконтроллеры (например, espruino и tessel).

В основе Node.js лежит событийно-ориентированное и асинхронное (или реактивное) программирование с неблокирующим вводом/выводом.

Поставленные задачи:

- изучить архитектуру NodeJS;
- рассмотреть способы асинхронного обмена данными с сервером с использованием websocket, comet, iframe и jsonp;
- сравнить websocket и comet;
- реализовать веб-сервер на websocket и comet;
- реализовать утилиту для нагрузочного тестирования веб-сервера на websocket и comet;
- провести нагрузочное тестирование и сделать выводы.

1 Архитектура NodeJS

В основе NodeJS лежит выполнение приложения в одном программном потоке, а также асинхронная обработка всех событий. При запуске NodeJS-приложения создается единственный программный поток. NodeJS-приложение выполняется в этом потоке в ожидании, что некое приложение сделает запрос. Когда NodeJS-приложение получает запрос, то никакие другие запросы не обрабатываются до тех пор, пока не завершится обработка текущего запроса.

На первый взгляд, все это кажется не очень эффективным, если бы не то обстоятельство, что NodeJS работает в асинхронном режиме, используя цикл обработки событий и функции обратного вызова. Цикл обработки событий просто опрашивает конкретные события и в нужное время вызывает обработчики событий. В NodeJS таким обработчиком событий является функция обратного вызова.

В отличие от других однопоточных приложений, когда к NodeJS - приложению делается запрос, оно должно, в свою очередь, запросить какие-то ресурсы (например, получить доступ к файлу или обратиться к базе данных). В этом случае NodeJS инициирует запрос но не ожидает ответа на этот запрос. Вместо этого запросу назначается некая функция обратного вызова.Когда запрошенное значение будет готово (или завершено) генерируется событие, активизирующее соответствующую функцию обратного вызова, призванную что-то сделать либо с результатами запрошенного действия, либо с запрошенными ресурсами.

Если несколько человек обращаются к NodeJS-приложению в одно и то же время и приложению нужно обратиться к ресурсам из файла, для каждого запроса NodeJS назначает свою функцию обратного вызова событию ответа. Когда для каждого из них ресурс становится доступен, вызывается нужная функция обратного вызова, и запрос удовлетворяется. В промежутке NodeJS-приложение может обрабатывать другие запросы либо для того же приложения, либо для какого-нибудь другого.

Хотя приложение не обрабатывает запросы в параллельном режиме, в зависимости от своей загруженности и конструкции можно даже не заметить задержки в ответе. А что лучше всего, приложение очень экономно относится к памяти и к другим ограниченным ресурсам. [1]

2 Архитектура NGINX

NGINX (сокращение от engine x) — это HTTP-сервер и обратный прокси-сервер, почтовый, а также TCP/UDP прокси-сервер общего назначения, изначально написанный Игорем Сысоевым.

Для лучшего представления устройства, сперва необходимо понять как NGINX запускается. У NGINX есть один мастер-процесс (который от имени суперпользователя выполняет такие операции, как открытие портов и чтение конфигурации), а также некоторое количество рабочих и вспомогательных процессов. Например, на 4-х ядерном сервере мастер-процесс NGINX создает 4 рабочих процесса и пару вспомогательных кэш-процессов, которые в свою очередь управляют содержимым кэша на жестком диске.

Говоря о многопоточности важно отметить, что любой процесс или поток — это набор самодостаточных инструкций, который операционная система может запланировать для выполнения на ядре процессора. Большинство сложных приложений параллельно запускают множество процессов или потоков по двум причинам:

1. Чтобы одновременно задействовать больше вычислительных ядер;
2. Процессы и потоки позволяют проще выполнять параллельные операции (например работать с множеством соединений одновременно).

2.1 Почему архитектура так важна?

Процессы и потоки сами по себе расходуют дополнительные ресурсы. Каждый из процессов или потоков потребляет некоторое количество памяти, и кроме того постоянно подменяют друг друга на процессоре (так называемое переключение контекста). Современные серверы могут справляться с сотнями активных процессов и потоков, но производительность сильно парает, как только заканчивается память или огромное количество операций ввода-вывода приводит к слишком частой смене контекста.

Наиболее типичный подход к построению сетевого приложения — это выделять для каждого соединения отдельный процесс или поток. Такая архитектура действительно проста для понимания и легка в реализации, но при этом плохо масштабируется, когда приложению приходится работать с тысячами соединений одновременно.

2.2 Как же работает NGINX?

NGINX использует модель с фиксированным числом процессов, которая наиболее эффективно задействует доступные ресурсы системы:

- Единственный мастер-процесс выполняет операции, которые требуют повышенных прав, такие, как чтение конфигурации и открытие портов, а затем порождает небольшое число дочерних процессов (следующие три типа).
- Загрузчик кэша запускается на старте чтобы загрузить данные кэша, расположенные на диске, в оперативную память, а затем завершается. Его работа спланирована так, чтобы не потреблять много ресурсов.
- Кэш-менеджер просыпается периодически и удаляет объекты кэша с жесткого диска, чтобы поддерживать его объем в рамках заданного ограничения.
- Рабочие процессы выполняют всю работу. Они обрабатывают сетевые соединения, читают данные с диска и пишут на диск, общаются с бэкенд-серверами.

2.3 Внутри рабочего процесса

Каждый рабочий процесс NGINX инициализируется с заданной конфигурацией и набором слушающих сокетов, унаследованных от мастер-процесса.

Рабочие процессы начинают с ожидания событий на слушающих сокетах. События извещают о новых соединениях. Эти соединения попадают в конечный автомат — наиболее часто используемый предназначен для обработки HTTP, но NGINX также содержит конечные автоматы для обработки потоков TCP трафика (модуль stream) и целого ряда протоколов электронной почты (SMTP, IMAP и POP3).

Конечный автомат в NGINX по своей сути является набором инструкций для обработки запроса. Большинство веб-серверов выполняют такую же функцию, но разница кроется в реализации.

2.4 Устройство конечного автомата

Конечный автомат можно представить себе в виде правил для игры в шахматы. Каждая HTTP транзакция — это шахматная партия. С одной стороны шахматной доски веб-сервер — гроссмейстер, который принимает решения

очень быстро. На другой стороне — удаленный клиент, браузер, который запрашивает сайт или приложение по относительно медленной сети.

Как бы то ни было, правила игры могут быть очень сложными. Например, веб-серверу может потребоваться взаимодействовать с другими ресурсами (проксировать запросы на бэкенд) или обращаться к серверу аутентификации. Сторонние модули способны ещё сильнее усложнить обработку.

2.5 Блокирующийся конечный автомат

Вспомните наше определение процесса или потока, как самодостаточного набора инструкций, выполнение которых операционная система может назначать на конкретное ядро процессора. Большинство веб-серверов и веб-приложений используют модель, в которой для «игры в шахматы» приходится по одному процессу или потоку на соединение. Каждый процесс или поток содержит инструкции, чтобы сыграть одну партию до конца. Все это время процесс, выполняясь на сервере, проводит большую часть времени заблокированным в ожидании следующего хода от клиента.

1. Процесс веб-сервера ожидает новых соединений (новых партий инициированных клиентами) на слушающих сокетах.
2. Получив новое соединение, он играет партию, блокируясь после каждого хода в ожидании ответа от клиента.
3. Когда партия сыграна, процесс веб-сервера может находиться в ожидании желания клиента начать следующую партию (это соответствует долгоживущим keepalive-соединениям). Если соединение закрыто (клиент ушел или наступил таймаут), процесс возвращается к встрече новых клиентов на слушающих сокетах.

Важный момент, который стоит отметить, заключается в том, что каждое активное HTTP-соединение (каждая партия) требует отдельного процесса или потока (гроссмейстера). Такая архитектура проста и легко расширяема с помощью сторонних модулей (новых «правил»). Однако, в ней существует огромный дисбаланс: достаточно легкое HTTP-соединение, представленное в виде файлового дескриптора и небольшого объема памяти, соотносится с отдельным процессом или потоком, достаточно тяжелым объектом в операционной системе. Это удобно для программирования, но весьма расточительно.

3 Способы асинхронного обмена данными с сервером

В современном Web асинхронный обмен данными с сервером является практически его неотъемлемой частью. Подобный обмен данными позволяет без перезагрузки страницы клиентского приложения обмениваться различной информацией с сервером, что в свою очередь позволяет как повысить интерактивность web-приложений, так и дает возможность обмена данными в режиме реального времени с сервером.

В далее качестве примеров рассмотрим способы обмена данными с использованием websocket, comet, iframe и jsonp.

3.1 WebSocket

WebSocket протокол был утвержден в качестве стандарта RFC 6455 в декабре 2011 года. Данный тип соединения предоставляет двунаправленное полнодуплексное соединение. С помощью WebSocket можно создавать интерактивные браузерные веб-приложения, которые постоянно обмениваются данными с сервером, но при этом не нуждаются в открытии нескольких HTTP-соединений.

Хоть и WebSocket использует HTTP как основной механизм для передачи данных, однако канал связи не закрывается после получения данных клиентом. Используя WebSocket API вы полностью свободны от ограничений типичного цикла HTTP (request/responce). Это также означает, что до тех пор пока соединение остается открытым, клиент и сервер могут свободно отправлять данные в асинхронном режиме без опроса для чего-нибудь нового.

Цикл работы WebSocket соединения состоит из следующих этапов:

— установления соединения (opening handshake);
— оформления и отправки данных;
— закрытие соединения (closing handshake).

3.1.1 Установление WebSocket соединения

Поскольку протокол WebSocket работает поверх HTTP, то это означает, что при подключении браузер отправляет следующие специальные заголовки:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
```

```
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Описание заголовков:

`GET, Host` — стандартные HTTP—заголовки из URL запроса.

`Upgrade, Connection` — указывают, что браузер хочет перейти на websocket.

`Origin` — протокол, домен и порт, откуда отправлен запрос.

`Sec-WebSocket-Key` — случайный ключ, который генерируется браузером: 16 байт в кодировке Base64.

`Sec-WebSocket-Version` — версия протокола. Текущая версия: 13.

Все заголовки, кроме `GET` и `Host`, браузер генерирует сам, без возможности вмешательства JavaScript.

Далее сервер, проанализировав эти заголовки, решает, разрешает ли он соединение WebSocket с данного домена `Origin`.

В случае, если сервер разрешает WebSocket подключение, то он возвращает клиенту ответ следующего вида:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

Здесь заголовок `Sec-WebSocket-Accept` представляет собой перекодированный с использование специального алгоритма ключ `Sec-WebSocket-Key`. Браузер в свою очередь использует ее для проверки, что ответ предназначается именно ему.

Затем данные передаются по специальному протоколу, структура которого («фреймы») изложена далее. И это уже совсем не HTTP.

3.1.2 Оформление и передача данных с использованием WebSocket протокола

В протоколе WebSocket предусмотрены несколько видов пакетов («фреймов»).

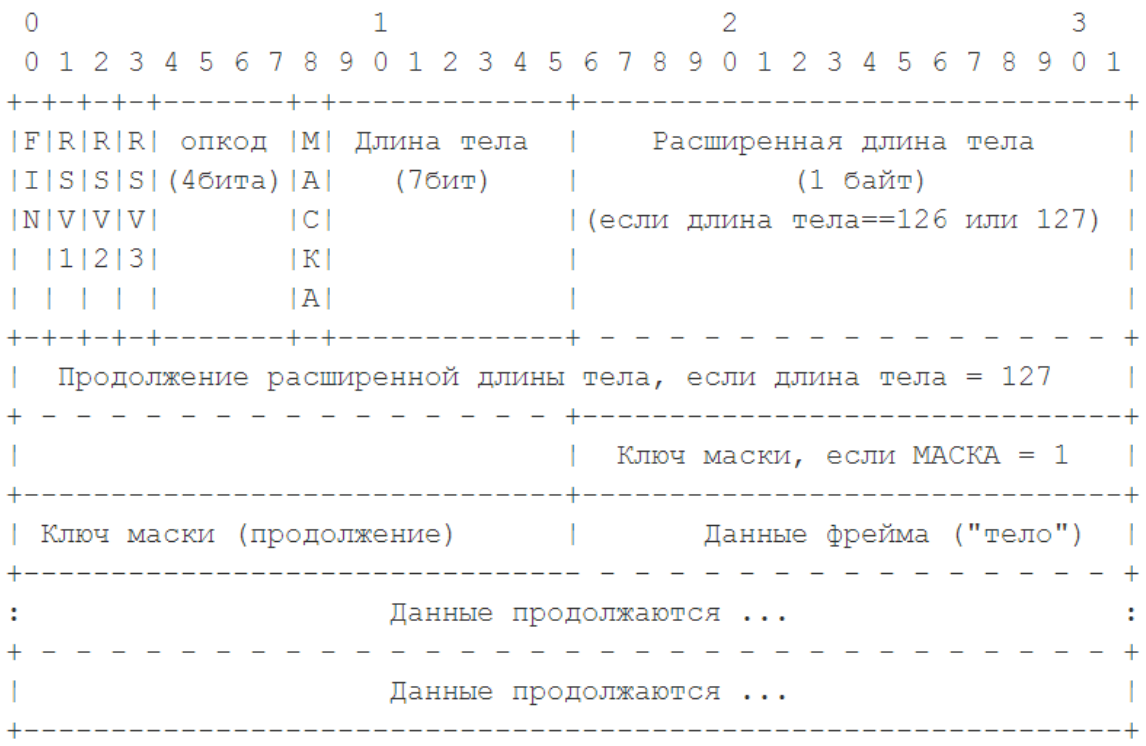

Рисунок 1 – Схема WebSocket фрейма

Они делятся на два больших типа: фреймы с данными («data frames») и управляющие («control frames»), предназначенные для проверки связи (PING) и закрытия соединения.

Фрейм, согласно стандарту, выглядит следующим образом:

3.1.3 Закрытие WebSocket соединения

Процесс закрытия WebSocket соединения гораздо проще, чем процесс открытия.

Любая из сторон WebSocket соединения может отправить управляющий фрейм с данными содержащими специальную управляющую последовательность для начала процесса закрытия. После отправки управляющего фрейма, указывающего, что соединение должно быть закрыто, другая сторона в дальнейшем отбрасывает любые данные, которые были отправлены. В добавок ко всему выше сказанному, процесс закрытия WebSocket соединения безопасен для одновременной инициализации обеими сторонами.

3.2 Comet

Другое название метода — «Очередь ожидающих запросов».

Основа работы данного метода состоит из следующей последовательности шагов:

— Отправляется запрос на сервер
— Соединение не закрывается сервером пока не появится событие;
— Событие отправляется в ответ на запрос;
— Клиент тут же отправляет новый ожидающий запрос.

Ситуация, когда браузер отправляет запрос и держит соединение с сервером, ожидая ответа, является стандартной и прерывается только доставкой сообщений.

При этом в случае, когда соединение рвется само, к примеру, из-за ошибки в сети, то браузер тут же отсылает новый запрос.

3.3 IFrame

По сути IFrame представляет собой окно браузера, вложенное в основное окно.

3.3.1 Общая схема работы

1. На клиентской стороне создается невидимый IFrame на специальный URL;
2. При наступлении событий на сервере в IFrame тут же поступает тег `<script>` — пакет с данными вида:

```
<script>
parent.handleMessage({txt:"Hello",time:123456789})
</script>
```

3. Соединение закрывается в случаях:
   — при возникновении ошибки;
   — каждые 20—30 секунд;
   — когда требуется очистка памяти от старых сообщений (время от времени создаем новый IFrame и удаляем старый).
4.

3.4 JSONP

Если попробовать создать тег `<script src>`, то при добавлении его в документ запустится процесс загрузки с данного `src`. В ответ на запрос сервер

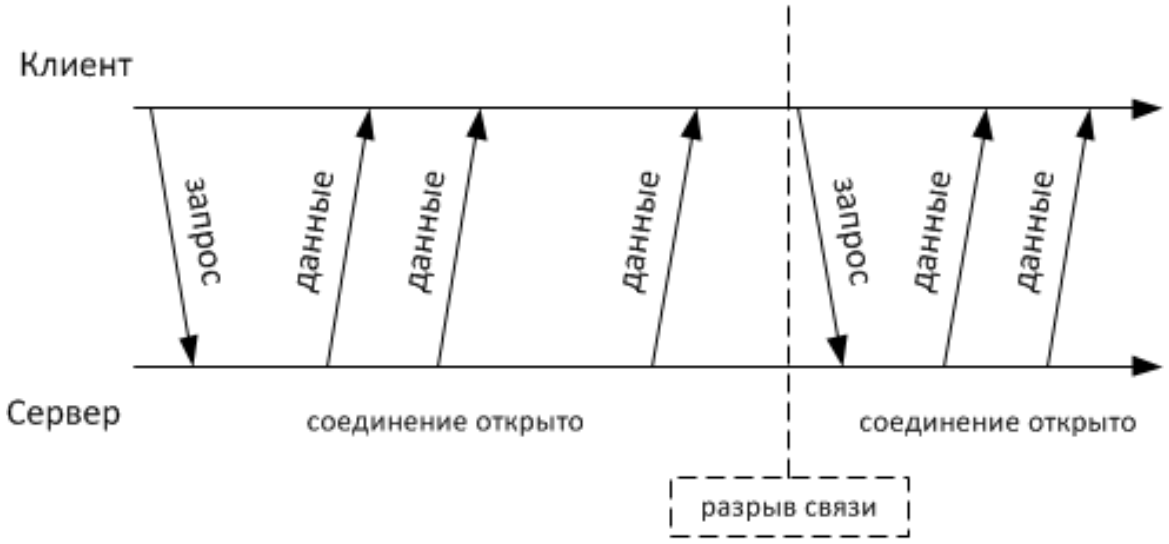

Рисунок 2 – Общая схема работы IFrame транспорта

может прислать скрипт, который будет содержать нужные данные.

С помощью данного способа можно запрашивать данные с любого сервера, в любом браузере, без каких-либо разрешений и дополнительных проверок.

Протокол JSONP — это своего рода надстройка над таким способом коммуникации.

3.4.1 Общая схема работы

1. На клиентской стороне создается тег `<script>` на специальный URL, в котором в качестве параметра передается имя функции обратного вызова, которая будет вызываться при получении данных;
2. В свою очередь сервер формирует ответ в виде вызова этой функции с данными переданными в ней в качестве параметров, и отправляет ответ клиенту;
3. Сразу после того как клиентская сторона получает ответ от сервера, полученный скрипт начинает немедленно выполняться, таким образом вызывая функцию обратного вызова, которая располагается на стороне клиента.

При использовании данного метода необходимо помнить про аспект безопасности, поскольку клиентский код должен доверять серверу при таком способе запроса данных. Ведь серверу ничего не стоит добавить в скрипт любые вредоносные команды.

COMET через протокол JSONP реализуется с использованием длинных запросов, то есть, создается тег `<script>`, браузер запрашивает скрипт у сер-

вера и сервер оставляет соединение висеть, пока не появятся данные, которые необходимо передать клиенту. Когда сервер хочет отправить сообщение — он формирует ответ с использованием формата JSONP, и тут же клиент отправляет новый запрос.

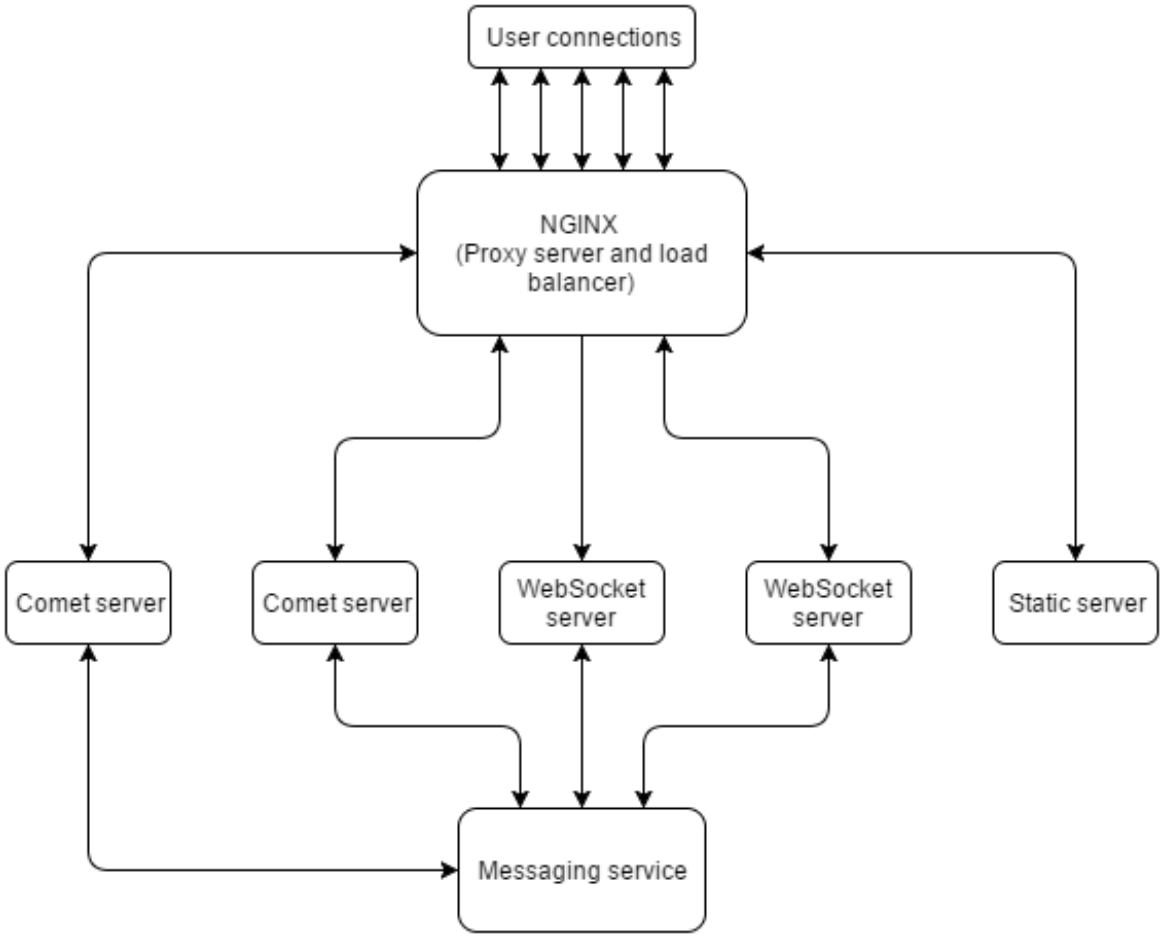

Рисунок 3 – Архитектура приложения

4 Описание архитектуры приложения

Архитектура приложения состоит из следующих компонентов:

— NGINX (прокси-сервер и балансировщик нагрузки)
— 2 WebSocket сервера
— 2 Comet сервера
— Messaging сервис

5 Реализация алгоритмов

=== ============

=== ==================

=== ===== В данном разделе приведены описания следующих алгоритмов:

1. Античный алгоритм Евклида (через разности);
2. Алгоритм Евклида (через остатки);
3. Бинарный алгоритм Стейна.

Листинг программ, реализующих эти алгоритмы можно увидеть в приложениях А—Г. Стоит заметить, что алгоритмы представленные в этих приложениях были написаны на языке C++ в среде Visual Studio 2010, и их тестирование на скорость вычисления поставленной задачи проводилось в операционной системе Windows 8, на компьютере HP Envy m6 1154er с процессором Intel Core i5 3210M, тактовая частота которого 2.5 ГГц.

5.1 Подробное описание алгоритмов

Далее приведены краткие рекомендации по работе с представленными программами, блок-схемы и скриншоты работы программ.

5.1.1 Античный алгоритм Евклида

На вход программе должно быть подано два целых положительных числа (те самые числа a и b) и нажата клавиша Enter. миллисекундах соответственно. Общий вид реализованного алгоритма представлен на рисунке 4. Код программы можно увидеть в приложении А.

На рисунке 5 можно увидеть скриншот работы программы.

5.1.2 Алгоритм Евклида

Здесь также, как и в предыдущем описании, на вход программе должно быть подано два целых положительных числа (те самые числа a и b) и нажата клавиша Enter. миллисекундах соответственно. Код программы можно увидеть в приложении Б. Общий вид реализованного алгоритма представлен на рисунке 6.

На рисунке 7 можно увидеть скриншот работы программы.

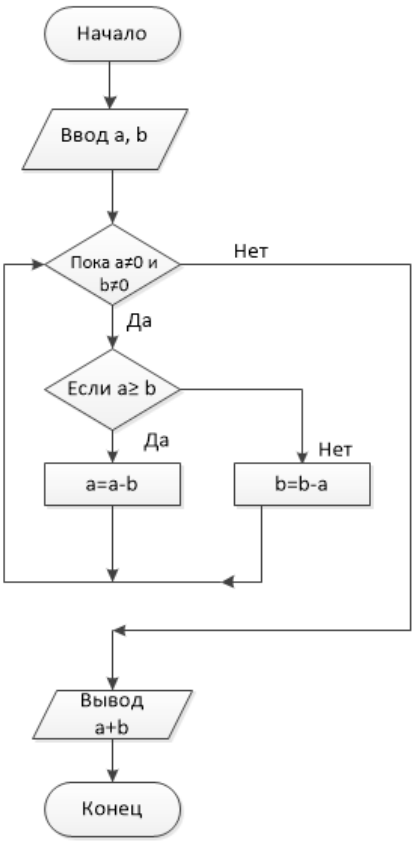

Рисунок 4 – Блок-схема античного алгоритма Евклида

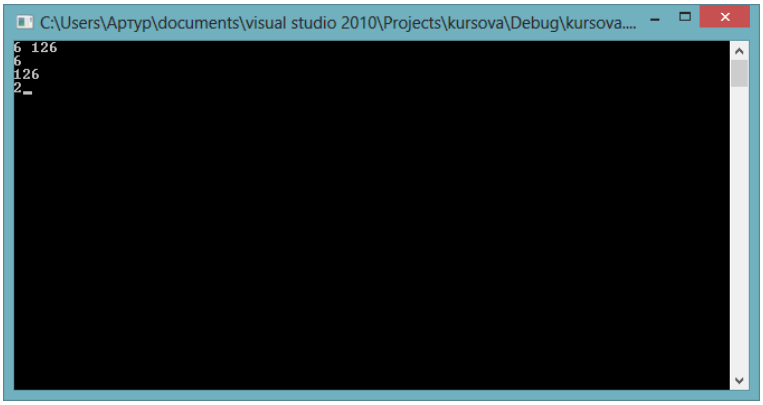

Рисунок 5 – Работа античного алгоритма Евклида

5.1.3 Алгоритм нахождения НОД методом перебора

Здесь также, как и в двух предыдущих описаниях, на вход программе должно быть подано два целых положительных числа (те самые числа a и b) и нажата клавиша Enter. миллисекундах соответственно. Код программы можно увидеть в приложении В. На рисунке 8 представлена блок-схема реализованного алгоритма.

На рисунке 9 можно увидеть скриншот работы программы.

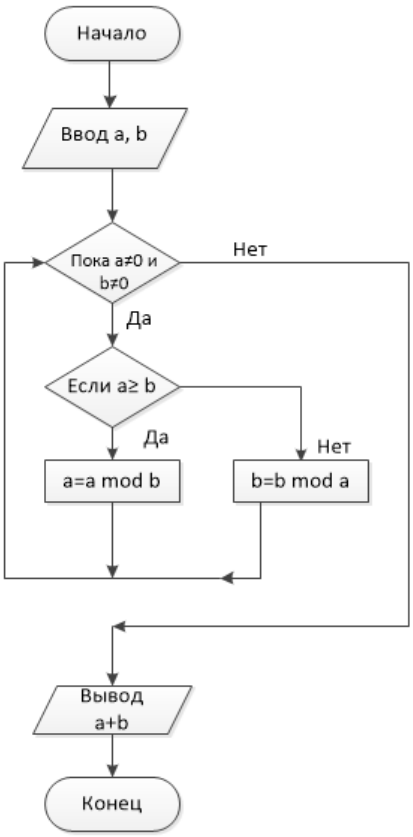

Рисунок 6 – Блок-схема алгоритма Евклида

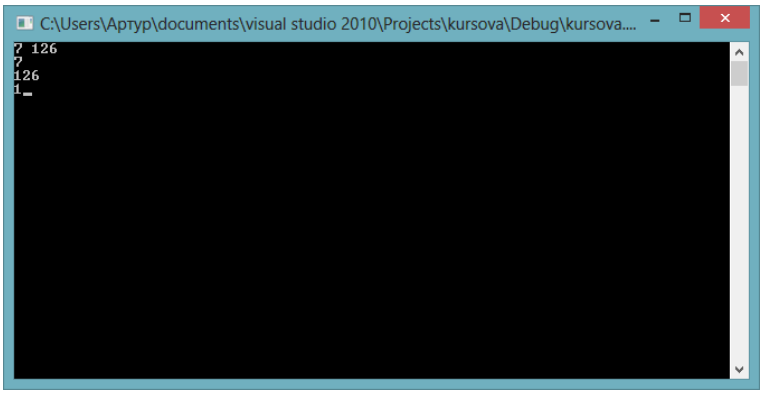

Рисунок 7 – Работа алгоритма Евклида

5.1.4 Бинарный алгоритм Стейна

Здесь также, как и в трех предыдущих описаниях, на вход программе должно быть подано два целых положительных числа a и b и нажата клавиша Enter. миллисекундах соответственно. Код программы можно увидеть в приложении Г. Общий вид реализованного алгоритма представлен в виде блок-схемы на рисунке 10.

На рисунке 11 можно увидеть скриншот работы программы.

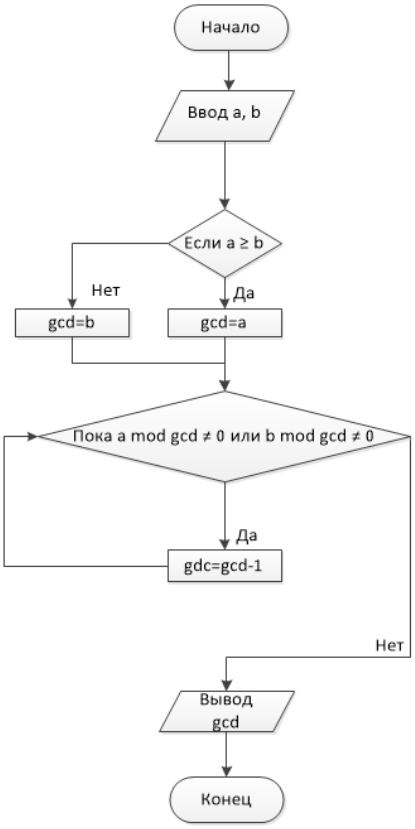

Рисунок 8 – Блок-схема алгоритма для нахождения НОД методом перебора

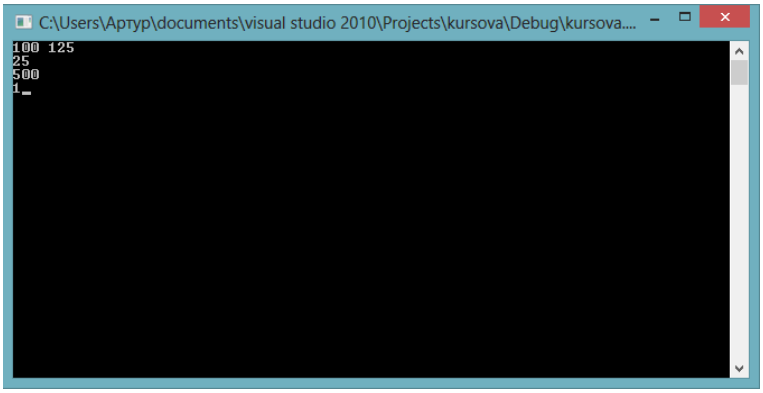

Рисунок 9 – Работа алгоритма для нахождения НОД методом перебора

5.2 Экспериментальные данные

Время работы алгоритма Евклида оценивается теоремой Ламе, которая устанавливает поразительную связь с последовательностью Фибоначчи:

Если $a > b \geqslant 1$ и $b < F_n$ для некоторого n, то алгоритм Евклида выполнит не более $n - 2$ рекурсивных вызовов.

Более того, можно показать, что верхняя граница этой теоремы — оптимальная. При $a = F_n$, $b = F_{n-1}$ будет выполнено именно $n - 2$ рекурсивных вызова. Иными словами, последовательные числа Фибоначчи — наихудшие

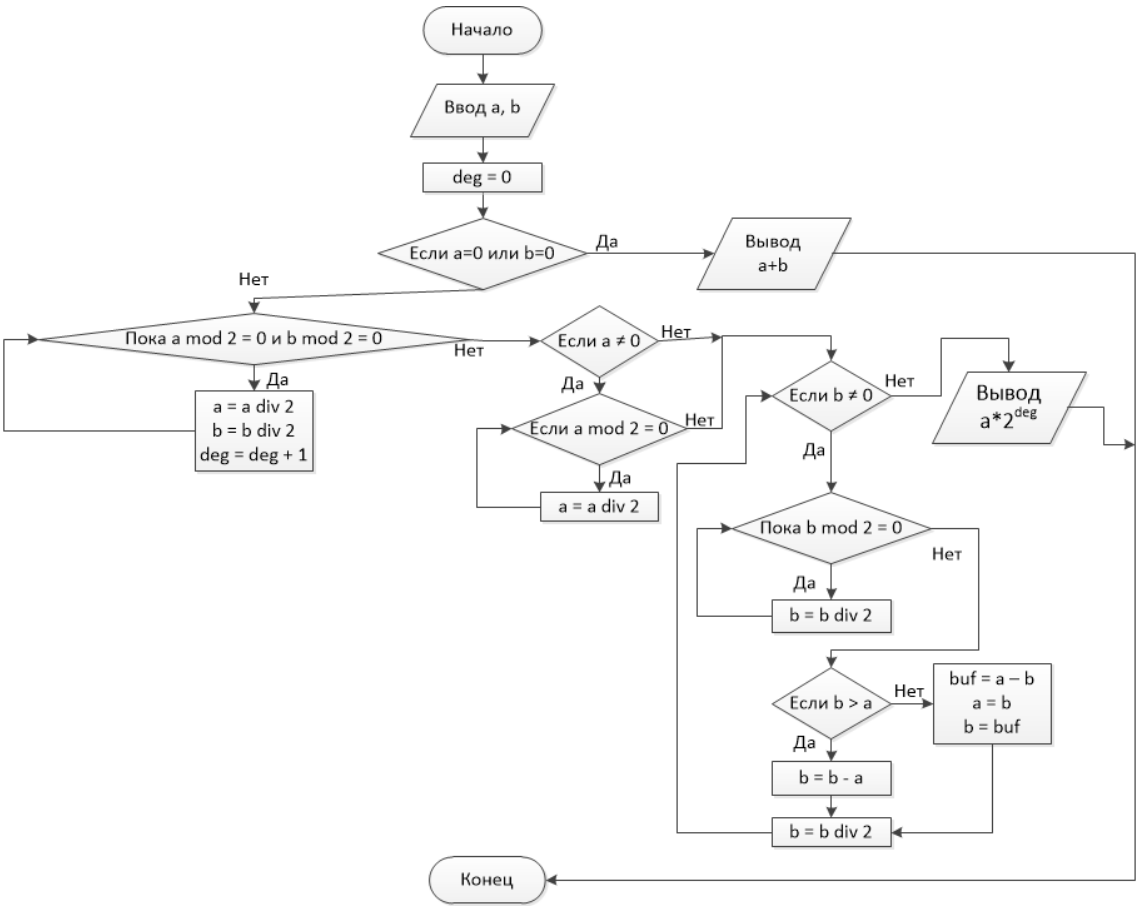

Рисунок 10 – Блок-схема бинарного алгоритма Стейна

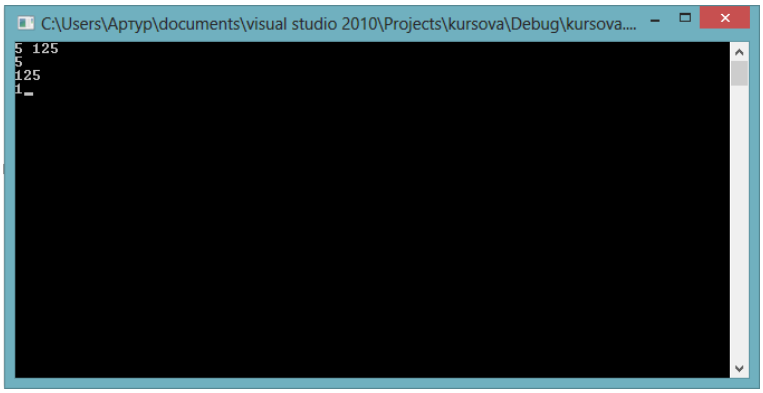

Рисунок 11 – Работа бинарного алгоритма Стейна

входные данные для алгоритма Евклида.

Учитывая, что числа Фибоначчи растут экспоненциально (как константа в степени n), получаем, что алгоритм Евклида выполняется за $O(\log \min(a, b))$ операций умножения.

На языке C++ в среде Microsoft Visual Studio рассмотренные алгоритмы были реализованы. Для проведения сравнительного анализа была написана вспомогательная программа, реализующая алгоритм нахождения n-го числа

Фибоначчи. Проведенные эксперименты показали, что алгоритм Евклида является наиболее эффективным из всех рассмотренных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Кнут, Д. Э.* Искусство программирования Том 2 / Д. Э. Кнут. — М.: Вильямс, 2002.

ПРИЛОЖЕНИЕ А

Листинг античного алгоритма Евклида, написанного на Microsoft Visual Studio C++ 2010

Код приложения `obalg.cpp`.

```
// обычный алгоритм Евклида через вычитания
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
#include "windows.h"

using namespace std;

unsigned long long int gcd(unsigned long long int a, unsigned long long int b)
{
    while (a && b) //пока оба числа не равны нулю
        if (a >= b)//если а больше или равно b
           a -= b; //вычитаем из а число b
        else       //иначе
           b -= a; //вычитаем из b число a
    return a | b;  //возвращаем a + b
}

int main()
{
        //объявление переменных, предназначенных
        //для вычисления времени работы программы
        LARGE_INTEGER start, finish, freq;

        unsigned long long int a, b, nod;
        //a,b - исходные числа
        //nod - наибольший общий делитель
        cin >> a >> b;

        // запоминаем частоту операций

        QueryPerformanceFrequency( &freq );

        // засекаем начало работы алгоритма

```

```
37          QueryPerformanceCounter( &start );
38
39          nod = gcd(a,b);
40
41          cout << nod << endl;//наибольший общий делитель
42
43          cout << a*b/nod << endl;//наименьшее общее кратное
44
45          // time - время в секундах
46
47          double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
48
49          cout << time << endl;
50
51          getch();
52             return 0;
53  }
```

ПРИЛОЖЕНИЕ Б

Листинг алгоритма Евклида, написанного на Microsoft Visual Studio C++ 2010

Код приложения `ostalg.cpp`.

```
// обычный алгоритм Евклида через остатки
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
#include "windows.h"

using namespace std;

unsigned long long int gcd(unsigned long long int a, unsigned long long int b)
{
    while (a && b) //пока оба числа не равны нулю
        if (a >= b)//если а больше или равно b
           a %= b; //a присваиваем остаток от деления его на b
        else            //иначе
           b %= a; //b присваиваем остаток от деления его на a
    return a | b;  //возвращаем a + b
}

int main()
{
        //объявление переменных, предназначенных
        //для вычисления времени работы программы
        LARGE_INTEGER start, finish, freq;

        unsigned long long int a, b, nod;
        //a,b - исходные числа
        //nod - наибольший общий делитель
        cin >> a >> b;

        // запоминаем частоту операций

        QueryPerformanceFrequency( &freq );

        // засекаем начало работы алгоритма

```

```
37          QueryPerformanceCounter( &start );
38
39          nod = gcd(a,b);
40
41          cout << nod << endl;//наибольший общий делитель
42
43          cout << a*b/nod << endl;//наименьшее общее кратное
44
45          // time - время в секундах
46
47          double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
48
49          cout << time << endl;
50
51          getch();
52             return 0;
53  }
```

ПРИЛОЖЕНИЕ В

Листинг алгоритма для нахождения НОД методом перебора, написанного на Microsoft Visual Studio C++ 2010

Код приложения pralg.cpp.

```
// Алгоритм для нахождения НОД методом перебора
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
#include "windows.h"

using namespace std;

int main()
{
        //объявление переменных, предназначенных
        //для вычисления времени работы программы
        LARGE_INTEGER start, finish, freq;

        unsigned long long int a, b, gcd;
        //a,b - исходные числа
        //gcd - наибольший общий делитель

        cin >> a >> b;

        // запоминаем частоту операций

        QueryPerformanceFrequency( &freq );

        // засекаем начало работы алгоритма

        QueryPerformanceCounter( &start );

        if (a >= b) gcd = a; //выбираем max(a,b)
        else gcd = b;


        //пока a и b не кратны gcd, уменьшаем значение
        //gcd на единицу
        while (a % gcd != 0 || b % gcd != 0) gcd--;
```

```
37
38          // засекаем окончание работы алгоритма
39          QueryPerformanceCounter( &finish );
40
41
42          cout << gcd << endl;//наибольший общий делитель a и b
43
44          cout << a*b/gcd << endl;//наименьшее общее кратное a и b
45          // time - время в секундах
46
47          double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
48
49          cout << time << endl;
50
51          getch();
52          return 0;
53  }
```

ПРИЛОЖЕНИЕ Г

Листинг бинарного алгоритма Стейна, написанного на Microsoft Visual Studio C++ 2010.

Код приложения `binalg.cpp`.

```
// бинарный алгоритм Стейна
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
#include "windows.h"

using namespace std;

unsigned long long int gcd(unsigned long long int a, unsigned long long int b)
{
    unsigned long long int buf, deg = 0;

    if (a == 0 || b == 0)//если a или b равно нулю
        return a | b;    //возвращаем a + b

    while (((a | b) & 1) == 0)//пока a и b нечётны
    {
        deg++;  //увеличиваем степень двойки на единицу
        a >>= 1;//делим a и b нацело на 2
        b >>= 1;
    }

    if (a)                      //если a не равно нулю
        while ((a & 1) == 0)//пока a чётно
            a >>= 1;        //делим его на 2

    while (b)//пока b не равно нулю
    {
        while ((b & 1) == 0)//пока b чётно
            b >>= 1;        //делим его на 2

        if (a < b)         //если a < b
            b -= a;        //вычитаем из b число a
        else               //иначе
        {
```

```
37              buf = a - b;//сохраняем a - b
38              a = b;          //присваиваем a число b
39              b = buf;          //присваиваем b сохранённую разность
40          }
41          b >>= 1;          //делим b нацело на 2
42      }
43
44      //возвращаем a умноженное на 2 в степени deg
45      return (a << deg);
46  }
47
48  int main()
49  {
50          //объявление переменных, предназначенных
51          //для вычисления времени работы программы
52          LARGE_INTEGER start, finish, freq;
53
54          unsigned long long int a, b, nod;
55          //a,b - исходные числа
56          //nod - наибольший общий делитель
57
58          cin >> a >> b;
59
60
61          // запоминаем частоту операций
62
63          QueryPerformanceFrequency( &freq );
64
65          // засекаем начало работы алгоритма
66
67          QueryPerformanceCounter( &start );
68
69          nod = gcd(a,b);
70
71          cout << nod << endl;//наибольший общий делитель
72
73          cout << a*b/nod << endl;//наименьшее общее кратное
74
75
76          // time - время в секундах
77
```

```
78          double time = (finish.QuadPart - start.QuadPart) / (double)freq.QuadPart;
79
80          cout << time << endl;
81
82          getch();
83              return 0;
84  }
```

ПРИЛОЖЕНИЕ Д

Листинг программы для нахождения чисел Фибоначчи, написанного на Delphi 7.0

Код приложения `fib.dpr`.

```
program fib;//алгоритм для нахождения чисел Фибоначчи
            //использующий длинную арифметику
{$APPTYPE CONSOLE}
uses
  SysUtils;

  Const MaxDig = 1000;//максимальная длина числа
        Osn = 10000;  //основание системы счисления
  Type TLong = Array[0..MaxDig] Of LongInt;
  Var i, j, n : LongInt;
      a, b, c : TLong;

  //процедура для сложения a и b
  Procedure SumLongTwo(A, B : TLong; Var C : TLong);
   Var i, k : LongInt;
   Begin
     FillChar(C, SizeOf (C), 0);//заполняем массив C нулями

     //присваиваем k наибольшее из длин чисел a и b
     If A[0] > B[0] Then k := A[0]
     Else k := B[0];

     For i := 1 To k Do // C присваиваем A + B
                Begin
                  C[i+1] := (C[i] + A[i] + B[i]) Div Osn;
                  C[i] := (C[i] + A[i] + B[i]) Mod Osn;
                End;

     //уточняем длину числа C
     If C[k+1] = 0 Then C[0] := k
     Else C[0] := k + 1
   End;

  //процедура для вывода на экран длинного числа
  Procedure WriteLong(Const A : TLong);
   Var ls, s : String;
```

```
37        i : LongInt;
38    Begin
39      //преобразуем число Osn Div 10
40      //в строку ls
41      Str(Osn Div 10, ls);
42
43      Write(A[A[0]]); {выводим старшие цифры числа}
44
45      For i := A[0] - 1 DownTo 1 Do
46       Begin
47         Str(A[i], s); {преобразуем чило A[i] в строку s}
48
49         {дополняем незначащими нулями}
50         While Length(s) < Length(ls) Do s := '0' + s;
51
52         Write(s)
53       End;
54    End;
55
56 Begin
57   ReadLn(n);
58
59   //присваиваем a, b, c значение 1
60   a[0] := 1;  a[1] := 1;
61   b := a;  c := a;
62
63   //вычисляем n-ое число Фибоначчи
64   For i := 3 To n Do
65    Begin
66      SumLongTwo(a,b,c);
67      a := b;
68      b := c;
69    End;
70
71   //вывод n-го числа Фибоначчи
72   WriteLong(c);
73
74   ReadLn;
75 End.
```