МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра « Прикладная математика и информатика»
(наименование кафедры)
01.03.02. Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

(направленность (профиль)/специализация)

БАКАЛАВРСКАЯ РАБОТА

на тему Проектирование и разработка алгоритма тестирования peer-to-peer сетей

Студент	Д.В. Косых		
	(И.О. Фамилия)		(личная подпись)
Руководитель	А.В. Шляпкин		
	(И.О. Фамилия)		(личная подпись)
Консультанты	М.А. Четаева		
	(И.О. Фамилия)		(личная подпись)

Допустить к защите
Заведующий кафедрой к.т.н., доцент, А.В. Очеповский _____
(ученая степень, звание, И.О. Фамилия) (личная подпись)

«_____» _____ 20___ г.

Тольятти 2018

Аннотация

Темой данной выпускной квалификационной работы является «Разработка и реализация алгоритма тестирования peer-to-peer сетей».

Актуальность темы «Разработка и реализация алгоритма тестирования peer-to-peer сетей» обуславливается построением peer-to-peer сети и тестированием её на отказоустойчивость.

Цель работы – разработка алгоритма системы тестирования, который позволит протестировать сеть на отказоустойчивость.

Объект исследования – алгоритм системы тестирования.

Первая часть выпускной работы описывает методы разработки автоматизированной системы тестирования. Рассматриваются особенности каждого метода. Проводится анализ архитектурных систем. Описываются протоколы передачи данных.

Вторая глава выпускной бакалаврской работы описывает выбор и обоснование использования языка программирования для разработки программы. Поясняется функциональный код программы.

Третья глава сосредоточена на разработке математической модели тестирования и применения моделирования в разработке.

В заключении подводятся основные итоги, формируются окончательные выводы по рассматриваемой теме.

Выпускная квалификационная работа содержит пояснительную записку объёмом 56 страницы, включая 20 рисунков, 1 таблицу и 15 формул, одно приложение и список литературы из 20 источников, включая 9 на иностранном языке.

ABSTRACT

The title of the graduation work is “Development and implementation peer-to-peer network testing algorithm”.

The relevance of the topic « Development and implementation peer-to-peer network testing algorithm » is determined by the construction of peer-to-peer networks and by testing it for fault tolerance.

The purpose of the work is the development of an algorithm of the testing system, which will allow testing the network for fault tolerance.

The object of investigation - is the algorithm of the testing system.

The first part of the graduation work describes the methods of the automated testing system. The features of each method are considered. The analysis of architectural systems is carried out. Data transmission protocols are described.

The second part of the graduation work on the selection and substantiation of the use of programming languages for program development. The functional code of the program is explained.

The third part of the de scribes on developing a mathematical model for testing and applying modeling in development.

In conclusion, the main results, the final conclusions on the topic under consideration, are summarized.

The graduation project consist of an explanatory note on 42 pages, including 12 figures, 3 tables, 24 formulas, the list of 20 references including 23 foreign sources.

Оглавление

ВВЕДЕНИЕ

Компьютерные информационно-вычислительные сети – стремительно развиваются с каждым годом. Тем самым являясь востребованным ресурсом в современном обществе. Они позволяют обмениваться информацией с окружающими нас людьми. Необходимость иметь доступ к большому количеству информации, лежащей на других компьютерах, породила локальные сети. Это помогло ненадолго, так как объединить, таким образом, большое количество компьютеров оказалось невозможно. Поэтому были предложены архитектурные системы, позволяющие принимать и передавать файлы с каждого компьютера, с которым есть связь и на котором имеется определенное программное обеспечение. Но есть ряд проблем, связанных с передачей и приемом файлов. Одна из них - низкая скорость интернета. И чтобы сеть работала без сбоев и потерей пакетов, нужно её протестировать.

Актуальность данной работы обуславливается тем, что при построение интернет соединения необходимо тестирование на отказоустойчивость, имитируя потерю пакетов. Прибегнув к такому подходу, можно избежать огромное количество проблем до использования сети пользователем, тем самым снизив будущие затраты.

Объект исследования ВКР - система тестирования ПО peer-to-peer сетей.

Предмет исследования ВКР - методы разработки системы тестирования.

Цель работы - заключается в разработке системы тестирования, для понимания того, как будут вести себя передаваемые данные при плохой связи.

Для достижения поставленной цели потребуется:

- Проанализировать методы разработки системы тестирования.
- Выбрать архитектуру сети.
- Найти технологию, позволяющая имитировать «плохой интернет».
- Протестировать систему тестирования.

В процессе работы изучены современные методы, средства и способы тестирования.

В заключении подводятся основные итоги и выводы по работе.

В списке используемой литературы перечисляются учебные пособия, периодические издания, электронные ресурсы и литература на иностранном языке.

Глава 1. Основные методы создания системы тестирования и выбор средств для разработки ПО

1.1 Сетевые архитектуры

Термин «сетевая архитектура» включает в себя общую структуру сети, то есть все компоненты, через которые функционируют сети, включая аппаратное и системное программное обеспечение. Сетевая архитектура представляет собой комбинацию стандартов, топологий и протоколов, необходимых для создания работоспособной сети. Каждый из сетевых элементов решает одну общую проблему. Взаимодействие между всеми элементами сети осуществляется через формализованный набор правил, называемый протоколом[2].

Архитектура сети определяет основные элементы сети, программное обеспечение, техническое обеспечение, характеризует её общую логическую организацию и описывает методы кодирования.

Архитектура сети стала логическим результатом эволюции компьютерных и телекоммуникационных технологий. С одной стороны, сети могут рассматриваться как средство передачи информации на большие расстояния, для чего в них применяются методы кодирования и мультиплексирования данных, получившие развитие в различных телекоммуникационных системах, а с другой стороны, они являются частным случаем распределенных компьютерных систем[1].

Существует три вида сетевых архитектур:

- сети типа «клиент-сервер»;
- одноранговая архитектура (пиринговые сети);
- гибридные сети.

1.1.1 Архитектура «клиент-сервер»

Архитектура «клиент – сервер» – одна из концепций информационной сети, где основная часть ее ресурсов сосредоточена на серверах, обслуживающих своих клиентов. Представляет собой распределенную

структуру приложения, которая разделяет задачи или рабочие нагрузки между серверами и клиентами.

Клиенты и серверы обмениваются сообщениями в шаблоне обмена сообщениями запрос-ответ . Клиент отправляет запрос, и сервер возвращает ответ. Этот обмен сообщениями является примером межпроцессного общения. Чтобы общаться, компьютеры должны иметь общий язык, и они должны следовать правилам, чтобы как клиент, так и сервер знали, чего ожидать. Язык и правила связи определены в протоколе связи . Все клиент-серверные протоколы работают на прикладном уровне . Протокол уровня приложения определяет основные шаблоны диалога. Чтобы формализовать обмен данными еще дальше, сервер может реализовать интерфейс прикладного программирования.

Сервер может получать запросы от многих отдельных клиентов за короткий промежуток времени. Выполняет только ограниченное количество задач в любой момент и полагается на систему планирования , чтобы приоритезировать входящие запросы от клиентов. Чтобы предотвратить злоупотребление и максимизировать доступность , сервер может ограничить доступность для клиентов.

Включает в себя два типа компонентов.

Сервера – это объект, представляющий и реализующие процедуры организации, хранения и выдачи клиентам нужной информации. Работает по заданиям клиентов и управляет выполнением их. После выполнения запроса сервер посылает полученные результаты клиенту, пославшему задание. Клиенты – это рабочие станции, которые используют ресурсы сервера. Клиенты, запрашивают у сервера данные, путем отправки запроса и последующего ожидания ответа от сервера. Традиционные потоковые системы на основе клиент-сервер могут быть легко созданы и обеспечены хотя обычно он налагает на владельца значительные затраты на развертывание и обслуживание.

На данный момент является наиболее распространенной архитектурой в мире. Прекрасным примером архитектуры могут служить веб-сайты. При просмотре сайта происходит отправка запроса веб-серверу, который возвращает требуемую информацию клиенту. На рисунке 1 показан простой вариант архитектуры типа «клиент-сервер»:

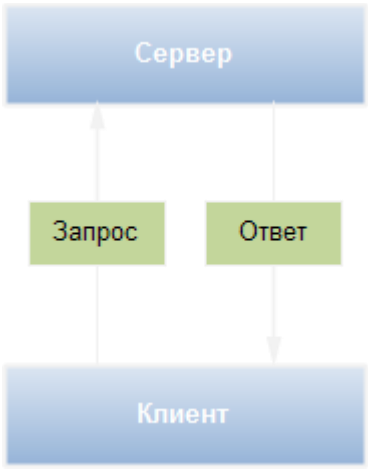

Рисунок 1 – Архитектура «клиент-сервер»

Достоинства:

- высокая производительность;
- программный код клиентского приложения и серверного разделен;
- возможность организации эффективной защиты данных;
- эффектная организация резервного копирования данных;
- все вычисления происходят на сервере;
- большая защищенность информации от несанкционированного доступа.

Недостатки:

- при отказе сервера прекратиться работа всей сети;
- для поддержки системы требуется высококвалифицированный специалист;
- высокая стоимость оборудования.

1.1.2 Одноранговая архитектура

Одноранговая архитектура (p2p) - концепция информационной сети, в которой все ресурсы рассредоточены по всем системам. Характеризуется тем, что в ней все системы равноправны. Пиринговая сеть предоставляет

компьютерам в сети равноправные возможности обмена различными ресурсами, в том числе и вычислительными. Системы peer-to-peer широко исследовались за последнее десятилетие, что привело к высокомасштабируемым реализациям для широкого спектра распределенных сервисов и приложений. Система P2P назначает симметричные роли машинам, которые могут действовать как клиент и сервер. Это облегчает необходимость того, чтобы какой-либо центральный компонент поддерживал глобальные знания системы. Вместо этого каждый партнер принимает индивидуальные решения на основе местных знаний о остальной системе, обеспечивая масштабируемость по дизайну. Хотя системы P2P были успешно применены к широкому спектру распределенных приложений с некоторыми очень заметными успехами, они, как правило, вышли из моды в пользу гораздо более облачного взгляда на текущий Интернет. Это считается парадоксальным, поскольку облачные системы сами по себе являются крупномасштабными высоко распределенными инфраструктурами. Они находятся в массивных, плотно взаимосвязанных центрах обработки данных и должны эффективно выполнять все большее число машин, имея дело с растущими объемами данных. На данный момент широкомасштабные системы требуют масштабируемых проектов для предоставления эффективных услуг. Утверждается, что локальный система P2P является ключевым фактором масштабируемости, независимо от того, развертывается ли система на одном многоядерном компьютере, распределяется внутри центра обработки данных или полностью децентрализована на нескольких автономных хостах. Также подтверждается замечанием о том, что некоторые из наиболее масштабируемых сервисов, используемых сегодня, сильно зависят от абстракций и обоснований, введенных в контексте систем P2P. Каждый участник сети вносит свой вклад. Это значит, что некоторые файлы, которые могут быть интересны другим пользователям, разрешено у вас скопировать, а взамен получаете возможность получить файлы, необходимые вам файлы от других пользователей. Для обмена файлами используется специальное

программное обеспечение, которое не имеет четкого разделения на серверные и клиентские модули. В таких сетях отсутствуют централизованное управление разделением ресурсов, вследствие чего, каждый узел одновременно выступает как в роли сервера, так и в роли клиента. Применяются в таких областях направления, как:

- обмен файлами;
- распределенные вычисления;
- обмен сообщениями;
- P2P – телефония.

На рисунке 2 показан простой вариант архитектуры типа «одноранговые сети»:

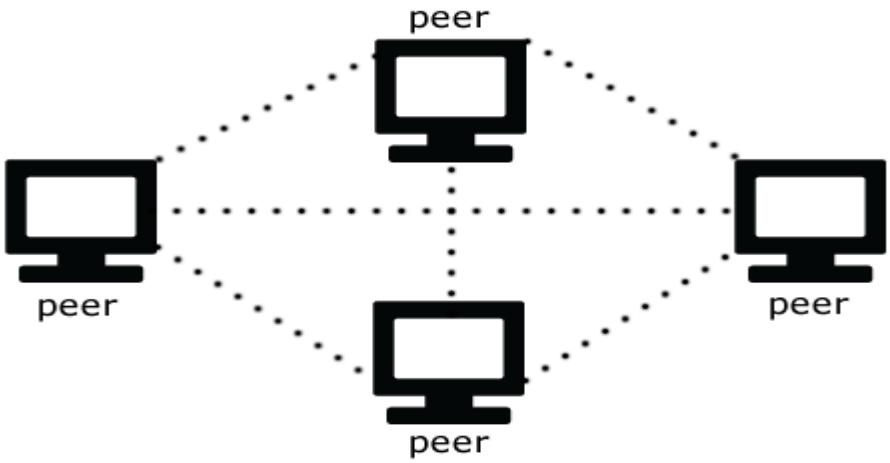

Рисунок 2 - Архитектура типа p2p

Достоинства:

- большая скорость обмена информации;
- устойчивость сети к различным сбоям;
- расширяемость;
- масштабируемость;

Недостатки:

- проблемы безопасности;
- неуправляемость;
- информационная недостоверность информации.

1.1.3 Гибридная архитектура

В дополнение к чистым P2P-сетям, существуют так называемые гибридные сети, которые используют сервера не для хранения информации, а для распределения сетевых операций и объединение клиентов. Гибридные сети объединяют в себе скорость централизованных сетей и надёжность децентрализованных. Если один или более серверов выходят из строя, сеть будет продолжать функционировать. К частично децентрализованным файлообменным сетям относятся EDonkey, BitTorrent. На рисунке 3 показан простой вариант архитектуры типа «гибридная»:

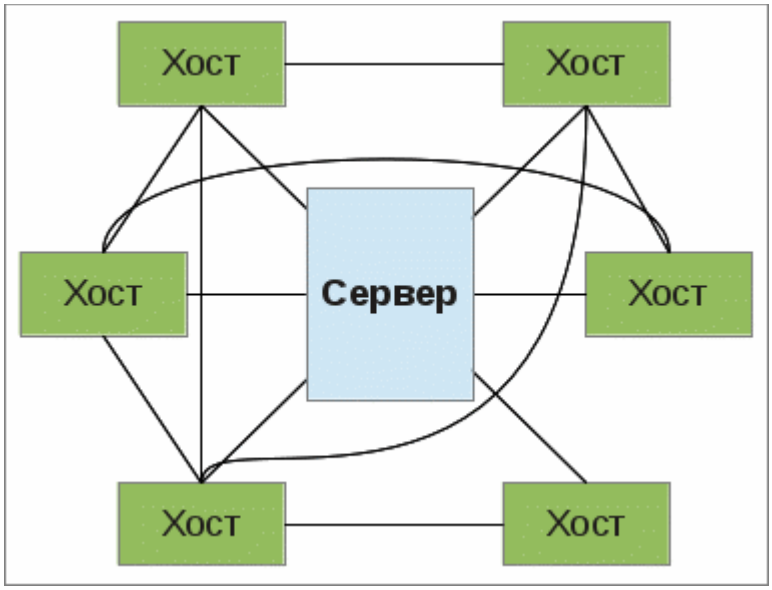

Рисунок 3 - Гибридная архитектура

Одной из областей применения технологии гибридных сетей является обмен файлами и вычислительными ресурсами. Технология гибридных сетей также используется для распределённых вычислений. Они позволяют в очень короткие сроки выполнять поистине огромный объём вычислений, который на суперкомпьютерах потребовал бы, многих лет и даже столетий работы. Такая производительность достигается за счёт того, что некоторая глобальная задача разделяется на большое количество блоков, которые одновременно выполняются сотнями тысяч компьютеров, принимающими участие в проекте.

Достоинства: преимущество серверной модели и одноранговой модели.

Недостатки: Имеют проблемы, характерные для серверных частей.

1.2 Виды тестирования программного обеспечения

На сегодняшний день мало кто сомневается в целесообразности тестирования разрабатываемых программных продуктов, но, к сожалению, не каждый ясно понимает, как правильно внедрять и применять тестирование. Тестирование является одним из наиболее важных этапов разработки программного обеспечения, направленного на обнаружение неисправностей, чтобы повысить качество продукта. По статистике, около 40% от всего времени разработки программного обеспечения затрачивается на тестирование.

Под тестированием принято понимать деятельность, выполняемую для оценки и улучшения качества программного обеспечения. В целом, тестирование основано на обнаружении дефектов и проблем в программных системах.

Процесс тестирования направлен на поиск и устранение ошибок в разработке программного обеспечения, чтобы проверить продукт на соответствие функциональным возможностям и требованиям по качеству, предъявляемым к нему.

Процесс тестирования связан с некоторыми проблемами:

- тестирование требует затрат большого количества времени и человеческих ресурсов в связи с множеством требований, предъявляемых к надежности и качеству продукта;
- поскольку программный продукт постоянно развивается, расширяет функционал, необходимо постоянное тестирование этого функционала.
- жесткие временные рамки;
- процесс тестирования монотонен и однообразен, что способствует низкой мотивации инженеров по тестированию.

Внедрение какого-либо вида тестирования в процесс разработки способствует решению данных проблем, помогая сократить время тестирования и снизить затраты.

На данный момент существует несколько видов тестирования:

- регрессионное тестирование;

- функциональное тестирование;
- установочное тестирование;
- конфигурационное тестирование;
- автоматизированное тестирование программного обеспечения.

Данные тесты могут выполняться вручную, но эти процедуры очень утомительны, занимают много времени и часто могут привести к случайным ошибкам. Самый простой способ запустить этот набор тестов — это использовать полностью автоматизированную тестовую систему, позволяющую пользователю просто вводить необходимые данные для проверки и запустить. После этого система выполняет все необходимые тесты и отображает результаты на экране монитора[7].

Регрессионное тестирование - проверка ПО на правильность функциональной части приложения. Выполняется с регулярной частотой, установленной в зависимости от условий, при внесение изменений в существующие функциональные возможности программного обеспечения. Включает в себя проверку исправления найденной ошибки, проверку на работоспособность работающей ранее функциональности. Является неотъемленной частью экстремального программирования. Может быть использовано не только для проверки корректности программы, но и для оценки качества получаемого результата.

Функциональное тестирование – тестирование программного обеспечения в целях проверки функциональных требований. Часто без автоматизации никак не обойтись. Даже если нужно выполнить тестирование только один раз. Включает в себя: точность, соответствие стандартам и правилам, защищённость, полностью имитирует фактическое использование системы, позволяет своевременно выявить системные ошибки, позволяет экономить за счет исправления ошибок на раннем этапе.

Установочное тестирование, выполняется для проверки условий инсталляции продукта с учётом тех или иных требований к системе от

заказчика. Также проводит проверки поведения программного продукта в контексте недостаточных системных требований.

Конфигурационное тестирование - выполнение одних тестов в разных условиях и при различных конфигураций системы. То есть когда один или несколько компонентов архитектуры системы требуется проверить в разном окружении, обычно заявленном в изначальных требованиях. Имеет две цели: определить оптимальную конфигурацию оборудования и проверить объект тестирования на совместимость. Перед началом тестирования рекомендуется: создать матрицу покрытия, проводить приоритезацию конфигурация и проверять каждую конфигурацию шаг за шагом с расставленными приоритетами.

В данной дипломной работе я буду использовать автоматизированное тестирование. Преимуществом автоматизации является:

- исключение «человеческого фактора»;
- большая скорость выполнения;
- снижение затрат на поддержку;
- автоматически сохраняемые отчеты;
- автономность выполнения.

Также имеются недостатки:

- большие затраты на поддержку;
- стоимость инструмента автоматизации;
- пропуск незапрограммированных ошибок;
- ограничения тестирования.

Основная цель автоматизации – оптимизировать время и человеческие ресурсы. Поэтому автоматизированное тестирование эффектно не во всех случаях. Инструменты автоматизированного тестирования обычно используются в сочетании с ручным тестированием. Автоматизированное тестирование может быть рентабельной в долгосрочной перспективе, особенно при повторном использовании регрессионного тестирования. Данное тестирования уменьшает усилия, связанные с ручным тестированием.

Тестировать программу можно на разных уровня:

- Юнит-тестирование;
- интеграционные тесты;
- API;
- GUI - тесты.

Юнит-тесты - это процесс в программирование, позволяющий проверить на корректность отдельные модули исходного кода программы. Это позволяет достаточно быстро проверить код на ошибки.

Интеграционные тесты тестируют какой-то компонент системы, обычно состоящий из многих модулей. Для того чтобы избежать ошибок и не зависеть от внешних условий, интеграционное тестирование производится в контролируемом окружении. Например, перед каждым тестом создается временная база данных с заранее подготовленными записями, очищаются папки для хранения временных файлов, а вместо запросов к внешним сервисам используется заглушка, возвращающая заранее подготовленные ответы. Если это не сделать, то мы можем получать ошибки из-за отсутствия какого-нибудь файла или из-за ошибки внешнего сервиса. Также, тестовый сайт часто разворачивается на отдельном сервере или виртуальном хосте.

API – это набор функций, процедур, классов, которые можно вызывать, чтобы получить данные. Включает прямое тестирование в рамках интеграционного тестирования , чтобы определить, соответствуют ли они ожиданиям функциональности, надежности, производительности и безопасности. Поскольку API-интерфейсы не имеют графического интерфейса , на уровне сообщений выполняется тестирование API . Для тестирования используются такие техники, как:

- анализ граничных значений. В API запросах в явном виде могут передаваться значения параметров. Позволяющие выделить границы входных и выходных значений и проверить их;
- разбиение на классы эквивалентности.

GUI тесты – это тестирование графического интерфейса. Является самым сложным для тестированием. Обычно тестируется с помощью скриптов, которые описывают последовательность действий

1.2.1. Анализ существующих программ для имитации сетевых проблем

Сейчас на просторах интернета представлено большое количество программного обеспечения для имитации сетевых проблем. Чтобы обосновать необходимость моей системы тестирования, нужно провести анализ существующих программ с подобными функциями.

В качестве аналогов были выбраны данные программы:

1. NetLemiter Pro;
2. WANem;
3. Sloppy;
4. Wireshark;
5. Essential NetTools.

NetLemiter Pro – программа для лимитирования и контроля сетевого трафика. Следит за деятельностью каждого приложения, использующего интернет, а также активно управляет трафиком, контролируя скорость потока данных. Имеет возможность устанавливать ограничение скорости для отдельных соединений, графически показывать применение канала, отражает статистику по трафику, возможность задать приоритеты полосы пропускания, ведение статистики соединений, позволяет разделить трафик, как внешний, так и локальный. Также имеет интегрированный планировщик и фаерфол.

WANem – является эмулятором глобальной сети. Также дистрибутив на основе knoppix, который позволяет эмулировать различные условия сетевого подключения. Тем самым доказав свою полезность в тестирование и отладки приложений. Позволяет создать проблемы для сети в ручную, используя удобный web-интерфейс. Эмулирует большое количество характеристик глобальной сети. Выполняет такие функции, как потерю пакетов, изменение порядка следования пакетов и повреждение пакетов.

Sloppy – это прокси-сервер, который эмулирует доступ к сайту через канал с задаваемой полосой пропускания. Из доступных настроек он имеет: адрес сайта, который будем тестировать, выбор скорости и порт, через который будет идти подключение. Также его можно использовать как для тестирования локального проекта, так и для любого проекта в сети.

Wireshark – является широко используемым анализатором сетей. Доступные настройки: профессионально настраиваемые отчёты, автономный анализ, многоплатформенный, фильтрация потока трафика, установка ограничения скорости. Достаточно эффективно находит, диагностирует и решает разнообразные проблемы, возникающие в сети.

Essential NetTools – предназначена для диагностики сети и выявление её проблем. Способна производить поиск активных TCP портов, выводит все доступные сведения об установленных входящих и исходящих соединений с сетью, сканирует заданный диапазон сетевых адресов, тестировать работоспособность запущенных на нем серверов HTTP и FTP.

Также созданы программы под определенные платформы, такие как Network Conditioner для OS X. В MacOS используется программа ipfw. А в линуксподобных системах утилита tc. Но данные программы не рассматривались, так как они имеют ограничения по платформе. Но данные программы имеют один большой минус. Они не умеют работать с p2p-протоколом, обеспечивающий возможность создания и функционирования сети равноправных узлов.

1.3 Протоколы связи

Все мы иногда сталкиваемся с плохой работой сети: пакеты пропадают, не проходит ping в сети и т.п. При разработке p2p сети мне нужно протестировать ее поведение в условиях плохой связи. Для этого воспользуемся протоколами сети интернет.

Протоколы – набор соглашений, определяющие обмен данными, процедуры их интерпретации, правила совместной работы оборудования в

сетях. Протоколы создаются главным образом для облегчения масштабирования различных компьютерных сетей. Основные требования:

Наиболее известные протоколы, используемые в сети Интернет:

- FTP
- HTTP
- WebSocket

Протокол FTP позволяет пересылать информацию, хранящейся в файлах.

Протокол HTTP – символьно-ориентированный протокол прикладного уровня. Лежит в основе обмена данными. Служит для передачи гипертекстовых документов. Позволяет указать способ представления одного и того же ресурса по различным параметрам[15].

Протокол HTTP является клиент-серверным протоколом надстройкой над протоколом TCP. В отличие от TCP, который не учитывает структуру передаваемых пакетов, HTTP обновляет в данных метаинформацию, позволяя получателю правильно интерпретировать полученные данные. На основе HTTP функционирует глобальная сеть Интернет.

Протокол HTTP первоначально является синхронным, то есть построенным по модели «запрос — ответ, при этом соединение полностью одностороннее. Объект XMLHttpRequest помогает частично решить данную проблему, но делает это не совсем хорошо. Он не может связать один вызов с другим, поэтому при каждом новом запросе обязан вычислять с самого начала, кому принадлежит запрос. Таким образом, уровень сложности кода для обработки отдельных запросов от веб-страницы может очень быстро вырасти до практически неосуществимой.

Для данной проблемы уже есть решение. Этим решением является технология WebSocket, которая позволяющая браузеру поддерживать открытое подключение с сервером и обмениваться данными.

WebSocket — протокол полнодуплексной связи (может передавать и принимать одновременно) с сервером поверх TCP-соединения, совместимый с

HTTP, который позволяет создавать интерактивное соединение между клиентом и сервером для обмена сообщениями в режиме реального времени, который работает через один сокет в интернете. Обеспечивает огромное сокращение ненужного сетевого трафика. Благодаря ему не станет клиента и сервера с фиксированными ролями, а появятся два равноправных участника обмена данными. Каждый при этом работает сам по себе, и когда надо отправляет данные другому. Вторая сторона отвечает по мере необходимости, и только если эта необходимость возникнет[20].

В отличие от HTTP запросов, WebSocket устанавливает только одно TCP-соединение и подтверждает, что сервер может связываться с WebSocket, выполняя специальные проверки, после чего сервер и клиент могут отправлять данные через установленное соединение. Соединение всегда остаётся открытым, но не передаёт ненужных HTTP заголовков. При этом в WebSocket нет ограничений на количество соединений.

На сегодняшний день лучшим решением для написания приложений в режиме реального времени является технология WebSocket. Она позволяет создавать устойчивое соединение между клиентом и сервером, уменьшая нагрузку на сервер по сравнению с остальными методами. Важность этой технологии подчеркивается также ее внедрением в наиболее популярные инструменты программирования как в JavaScript.

Глава 2. Разработка математической модели алгоритма тестирования

2.1 Математическое моделирование алгоритма тестирования

Математическая модель – это система математических соотношений, отражающих некоторые свойства явления или объекта. Эффективную математическую модель можно реализовать на компьютере в виде алгоритмической модели. Создавая математическую модель для решения задачи, нужно:

1) выделить предположения, на которых основывается математическая модель;

2) определить, что считать исходными данными, а что — результатами;

3) записать математические соотношения, связывая результаты с исходными данными.

При построении математической модели не всегда можно найти формулы, явно выражающие исконные величины через исходные данные. Графические формы часто используются для представления алгоритма.

Представим алгоритм на базе полного графа. Это даст представление о том, как получать и преобразовывать информацию при её передаче, что особенно важно для оптимизации алгоритма.

Граф – это конечное множество v точек, называемых вершинами, и конечный набор x линий, называемых ребрами, соединяющих некоторые пары вершин.

Полный граф – простой ориентированный граф, в котором каждая пара вершин соединена между собой.

Граф представляет структурную математическую модель и отображает ее топологию. Если ввести обозначения ветвей графа, то он будет содержать ту же информацию, что и эквивалентная схема.

При моделировании сетевых систем одной из задач является моделирование сетевого канала данных. Данная трудность обусловлена следующими факторами:

- передача данных по каналу связи носит вероятностный характер, то есть присутствует случайная временная задержка;
- во время передачи по каналу связи возможна потеря данных с определенной вероятностью.

Попытки удовлетворить все эти требования одновременно приводят к большой математической модели, что существенно снижает её ценность.

Эффектным способом представления процессов комплекса работ является сетевой график. Сетевой график – графическое представление сетевой модели объекта управления. Сетевая модель имеет виды:

- матричная форма;
- табличная форма;
- сетевой график.

Сетевая модель – модель, отражающая структура процесса, явления или объекта.

Сетевой график – представляет собой граф без контуров, дугам и вершинам приписаны значения.

На рисунке 4 представлен граф, с помощью которого предлагается моделировать процесс передачи данных по каналу связи.

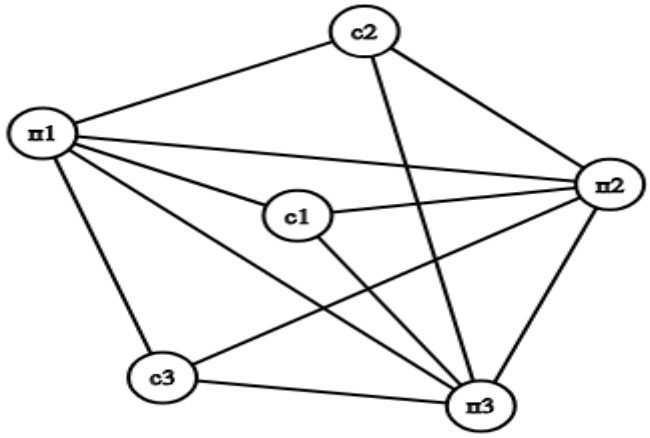

Рисунок 4 – Сетевой граф

Данная модель представляет собой полносвязную топологию. Главным плюсом полносвязной сети является обеспечение максимально возможного количества путей между узлами. И в случаи потери соединения между двумя вершинами, автоматически будет выбран другой путь для связи. Таким образом построить сеть более надежную, чем полносвязная, почти нереально. С вершинами графа связаны клиенты и сервера, с ребрами – время задержки.

Разберем вероятность отказа ребер

Представив сеть в виде графа, можно описать и задачу оптимизации в терминах теории графов.

Существует два наиболее распространенных метода передачи данных, рассмотрим второй способ.

Он выполняет передачу данных в виде блоков информации, в результате чего передача данных может быть сведена к передаче пакетов. Время передачи данных определяется выражением:

$$t_{пд} = t_н + mt_к + t_c l, \quad (1)$$

где $t_{пд}$ – время передачи данных;

m – сообщение размером m байт;

l – длина маршрута;

$t_н$ – длительность подготовки сообщения для передачи, поиска маршрута сети и т.д.;

$t_к$ – время передачи одной единицы данных по одному каналу связи, длительность подобной передачи определяется пропускной способностью;

t_c – время передачи служебных данных при обрыве связи.

Данный подход существенно снижает потребность в памяти для хранения пересылаемых данных, а для передачи пакетов могут использоваться одновременно разные каналы.

2.2 Сетевое моделирование

Сетевое моделирование используется как метод управления и планирования. Весьма эффективно на всех этапах разработки решений. Объектами могут быть любые комплексы работ

При моделировании беспроводных сетей и анализе их характеристик, наибольший интерес представляют характеристики пропускной способности сети и отказоустойчивость. Существующие методы оценки данных характеристик беспроводных сетей, построены на основе стандартов семейства IEEE 802.11.

Проекты с крупной сетью и со сложной топологией в настоящее время не обходится без моделирования будущей сети. Цель моделирования - определить оптимальную топологию, адекватный выбор сетевого оборудования, определение производительности сети и возможные этапы будущего развития. На модели можно опробовать эффект всплесков или реализовать режим коллапса, что нельзя будет себе позволить в работающей сети. Во время моделирования определяются следующие параметры:

- ограниченные пропускные способности различных фрагментов сети;
- время отклика основных серверов в различных режимах, включая те, которые крайне нежелательны в реальной сети;
- влияние установки новых серверов на перераспределение информационных потоков;
- своевременная оптимизации топологии при возникновении узких мест в сети;
- выбор определенного типа сетевого оборудования или режима его работы;
- выбор протокола маршрутизации и его параметров;
- определение максимально допустимого количества пользователей определенного сервера;
- оценка воздействия мультимедийного трафика на работу локальной сети;

Результаты моделирования должны иметь точность 10-20%, так как этого достаточно для большинства целей и не требует слишком много вычислительного времени. Чем точнее воспроизводится поведение сети, тем

больше времени будет нужно. Кроме того, необходимо сделать некоторые предположения относительно распределения нагрузки для конкретных сетей и других сетевых элементов, задержек в переключателях, мостах, времени обработки запросов в серверах. Здесь нужно учитывать и характер решаемых на ЭВМ задач. В случае моделирования реальной сети можно сделать соответствующие измерения, которые иногда также не слишком просты. Исключением является моделирование внешнего трафика, но в этом случае весь трафик должен рассматриваться как фоновый.

2.3 Аналитическое моделирование

Аналитическая модель сети представляет собой совокупность математических соотношений, которые соединяют между собой входные и выходные характеристики сети. При выводе таких соотношений необходимо пренебрегать незначительными деталями или обстоятельствами.

Пиринговая сеть при некотором упрощении может быть представлена как совокупность процессоров. Сообщение, поступающее на узел, ждет некоторое время до его обработки. При этом может образоваться очередь подобных сообщений. Время передачи или общее время задержки сообщения D равно:

$$D = T_p + S + W, \quad (2)$$

где T_p - время распространения;

S – время обслуживания;

W – время ожидания.

Одной из задач аналитического моделирование является определение среднего значения D. При больших загрузках основной вклад дает ожидание обслуживания W.

M/M/2 является одной из схем работы буфера и означает очередь, для которой время прибытия и обслуживания имеет экспоненциальное распределение. Среднее значение длины очереди Q при заданной средней входной частоте сообщений и среднем времени ожидания W определяется на основе теоремы Литла:

$$Q = \alpha W \quad (3)$$

где W – среднее время ожидания;

α – средняя входная частота.

Для варианта очереди M/G/1 входной процесс характеризуется распределением Пуассона со скоростью поступления сообщений.

$$p\ k\ = \frac{(\alpha t)^k}{k!} e^{-\alpha t}, k = 0,1,2,\ldots \quad (4)$$

где k – вероятность поступления сообщения;

t – время.

Пусть n – число клиентов в системе, Q – число клиентов в очереди, j – входящий клиент, тогда:

$$\Pi_j = P\ n = j\ , j = 0,1,2,\ldots \quad \sum_{j=0}^{\infty} \Pi_j = 1 \quad (5)$$

Тогда среднее время ожидания w:

$$W = \frac{Q}{\alpha} = \frac{\rho\tau}{2\ 1-\rho}(1 + \frac{\sigma^2}{\tau^2}) \quad \text{- формула Поллажека-Хинчина} \quad (6)$$

где σ - среднеквадратичное отклонение для распределения времени обслуживания.

Для варианта очереди M/G/1

$$H\ t\ = P\ xJt\ = 1 - e^{-mt} \quad (7)$$

где H - функция распределения времени обслуживания. Откуда следует $\sigma^2 = \tau^2$.

$$W = \frac{\rho\tau}{1-\rho} \quad (8)$$

Для варианта очереди M/D/1 время обслуживания постоянно, а среднее время ожидания составляет:

$$W = \frac{\rho\tau}{2\ 1-\rho} \quad (9)$$

Среднее значение времени ожидания для таких сетей составляет:

$$D = \frac{\alpha[S^2 + 4e + 2\ \tau S + 5\tau^2 + 4e\ 2e - 1\ \tau^2]}{2(1-\alpha\ S + \tau + 2e\tau\ )} - \frac{1 - e^{-2\alpha\tau}\ y + \alpha\tau - 2\alpha\tau e}{\alpha e\ F\ \alpha\ e^{-}\ 1 + \alpha\tau + e^{-2\alpha\tau} - 1} \quad (10)$$

где S – первый момент распределения передачи;

S^2 – второй момент распределения передачи;

t - задержка распространения сигнала в сети;

e - основание натурального логарифма.

Теперь преобразуем (10) с помощью Лапласа для распределения времени передачи сообщения.

$$F\ \alpha\ =\ \int_0^{\infty} f\ t\ e^{-\alpha t} dt \qquad (11)$$

Для варианта очереди M/M/1 входной процесс характеризуется по экспоненциальному закону.

$$P_{loss} = \frac{(1-\rho)}{1-\rho^{W+1}} * p^{W}, \qquad (12)$$

где $\rho = \frac{\alpha}{\mu} < 1$ – коэффициент загрузки канала;

$\alpha -$ интенсивность входного потока;

μ – интенсивность обслуживания выходного потока;

W – емкость запоминающего устройства, измеряемая в пакетах.

В системе G/G/1 общее выражение для вероятности потери пакетов вычисляется по формуле:

$$P_{loss} = \frac{(1-\rho)}{1-\rho^{(W+1)\frac{2}{C_{\alpha}^2+C_{\mu}^2}}} * \rho^{W*\frac{2}{C_{\alpha}^2+C_{\mu}^2}} \qquad (13)$$

где $C_{\alpha}^2 = \frac{(\delta[\alpha])^2}{M[\alpha]}$ – квадратный коэффициент девиации входного потока;

$C_{\mu}^2 = \frac{(\delta[\mu])^2}{M[\mu]}$ – квадратичный коэффициент девиации выходного потока.

Вероятность потери пакета при помощи параметра Херста.

$$P_{loss} = \frac{(1-\rho)}{1-\rho^{(W+1)\frac{2}{C_{\alpha}^2+C_{\mu}^2}}} * \rho^{W^2(1-H)} \qquad (13)$$

2.4 Анализ сетей с простейшим входным потоком

В реальных системах входной буфер имеет ограничение. Поэтому при поступление большого числа пакетов, система может дать сбой. Очевидно, что при проектирование систем связи необходимо уметь вычислять

вероятность возникновения такой ситуации. На рисунке 5 показана система с ограниченной длиной очереди.

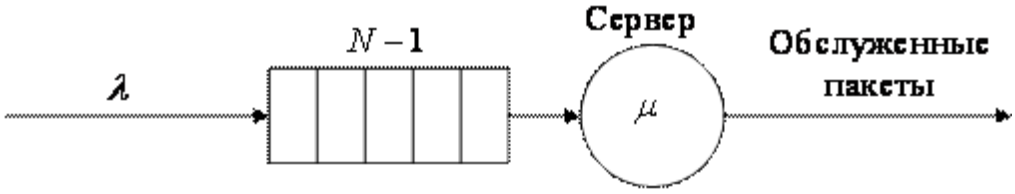

Рисунок 5 – система с ограниченной длиной очереди

Предположим, что максимальное число заявок равно N. Поступивший N+1 пакет получит отказ. Также предположим, что интервалы между поступлениями пакетов распределены по пуассоновскому закону с параметром α, а время обслуживания подчинено показательному закону с параметром μ. Тогда используя форму Эрланга вычислим вероятность нахождения k пакетов в системе:

$$p_{k=1+s} = \frac{\frac{z^n}{n!}(\frac{z}{n!})^s}{\sum_{j=0}^{n}\frac{z^j}{j!}+\frac{z^n}{n!}}. \quad (14)$$

И теперь определим пропускную способность системы как число пакетов, обслуживаемых системой за единицу времени. Заметим, что при потери части пакетов, интенсивность γ будет отличаться от интенсивности входного α. Однако, зная интенсивность обслуживания одного пакета μ и вероятность занятости сервера $(1-p_0)$, средняя интенсивность выходного потока может быть определена по формуле:

$$\gamma = \mu\ 1-p_0\ = \mu(1-\frac{1-Z}{1-Z^{N+1}}) \quad (15)$$

2.5 Выбор языка программирования

JavaScript – кросс-платформенный, легковесный, интерпретируемый, мультипарадигменный язык программирования, предназначенный для создания небольших клиентских и серверных приложений. Является объектно-ориентированным языком, но использует в языке прототипирование. Наиболее широкое применения нашёл как язык веб-программирования. Кроме того, он

обладает рядом свойств, присущих функциональным языкам, что придаёт языку дополнительную гибкость[14].

Ключевые особенности языка:

- автоматическое приведение типов;
- функции выступают объектами базового класса;
- автоматическая сборка мусора;
- анонимные функции.

Сфера применения javascript очень обширна.

- разработка веб-приложений;
- активное участие в ajax;
- Comet;
- серверные приложения;
- мобильные приложения.

Node js - серверная реализация языка программирования JavaScript, основанная на движке V8. Является средой выполнения JavaScript и добавляет возможность взаимодействовать с устройствами ввода-вывода через свой API. Применяется преимущественно на сервере. В состав входит установщик пакетов npm. Node обладает богатой функциональностью и имеет широкое распространение. Основан на событиях и выполняется асинхронно. Обрабатывает входящие запросы в стеке постоянных событий, отправляет небольшие запросы, не дожидаясь ответа. Он предназначен для создания масштабируемых распределенных приложений, например веб-сервер.

Одним из основных преимуществ Node js является то, что он не блокирует ввод/вывод. Является однопоточным и использует модель параллелизма, основанную на цикле событий. Это означает, что платформа может обрабатывать параллельные операции без нескольких потоков выполнения и эффективно масштабируется.

Тем самым, к основным преимуществам JavaScript можно отнести то, что данный язык программирования подходит для создания приложений в сфере веб-программирования, вобрав в себя возможности объектно-

ориентированных и функциональных языков, позволяет работать с протоколами интернета и взаимодействовать с серверной стороной сервиса[8].

2.7 Реализация соединения между сервером и клиентом

Для написания программного обеспечения peer-to-peer сетей понадобится наладить связь между клиентами и серверами с помощью WebSocket. Программное обеспечение написано на языке программирования Javascript.

Чтобы присоединиться к участнику сети, нужно создать сокет. Сокет – программный интерфейс, используется для коммуникации в сети. Для создания сокета необходимо: номера портов TCP участников, IP-адреса участников. Существует два вида сокетов: синхронные и асинхронные. Синхронные отвечают за задержку управления во время выполнения подключения, а асинхронные возвращают его немедленно, продолжая работу в фоновом режиме. Также делятся на два вида, потоковые и дейтаграмные. Потоковые сокеты работают с установкой соединения и гарантирует целостность и успешность доставки данных. Дейтаграмные сокеты работают без установки соединения и не обеспечивают контроля успешности доставки данных, при этом они намного быстрее потоковых. На рисунке 6 изображена функция для создания сокетов.

```
var connection = new WebSocket('ws://127.0.0.1:8080');
socket.create('tcp', '127.0.0.1', 8080, function(socketInfo) {
socket.connect(socketInfo.socketId, function (result) {
socket.write(socketInfo.socketId, "Hello, world!");
    });
});

connection.onopen = function () {
  connection.send('Ping'); // Send the message 'Ping' to the server
};

tcpPermission.requestPermission({remoteAddress:"127.0.0.1", remotePort:6789}).then(
  () => {
    var mySocket = new TCPSocket("127.0.0.1", 8080);

    mySocket.writeable..then(
        () => {
            mySocket.readable.getReader().read().then(
                ({ value, done }) => {
                    if (!done) {
                            console.log(value);
                    }
                    mySocket.close();
```

```
                }
            );
        },
        e => console.error(e);
    );
```

Рисунок 6 – Функция для создания сокета

WebSocket – это мощный способ реализации двунаправленной связи между сервером и клиентом. При этом не создаётся огромное количество соединений. Имеет простой обработчик ошибок. Поддерживается многими современными браузерами. Данное соединение является постоянным и с помощью него можно записывать и получать данные посредством использования стандартного TCP[6].

На рисунке 6 иллюстрируется связь с помощью WebSocket.

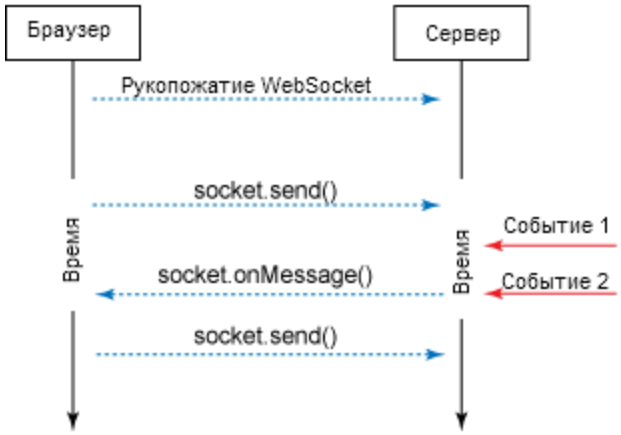

Рисунок 6 - Связь с использованием WebSocket

WebSocket можно рассматривать как TCP сокеты, тем самым решение о том, какой тип данных использовать, остается за клиентом и сервером. При создании объекта WebSocket в HTTP-запросах, происходит обмен заголовками. Клиент отправляет заголовок подобного содержания, как на рисунке 7.

```
GET /chat HTTP/1.1
HOST: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin:
Sec-WebSocket-Key: JHdadlkfhakj/dasfd/24r438rj==
Sec-WebSocket-Version: 13
```

Рисунок 7 – Основные функции для проверки соединения

Описание заголовков:

- GET, HOST – стандартные HTTP-заголовки из URL адреса;

- upgrade, Connection – указывает браузеру, что происходит переход на WebSocket;
- origin – протокол, домен и порт и информация откуда отправлен запрос;
- sec-WebSocket-Key – случайный ключ, который генерируется браузером(16 байт в кодировке Base64);
- sec-WebSocket-Version – версия протокола.

Все заголовки, кроме GET и HOST, браузер генерирует сам.

Мы создадим подключение WebSocket, используя соответствующий конструктор. Первый аргумент является ссылкой для соединения. WebSocket использует новую протокол URL - ws. Для создания безопасных соединений веб-сокетов используется протокол wss. Также кроме большей безопасности, wss имеет большое преимущество перед обычным ws – большая вероятность соединения, так как wss шифрует весь трафик и не даёт прокси-серверам оборвать передачу.

Второй аргумент конструктора позволяет использовать дополнительные субпротоколы. Это может быть массив строк или строка. Обязательно каждая из них должна соответствовать названию субпротокола, а сервер принимает только один из переданных параметров. Названия субпротоколов должны соответствовать реестру IANA.

На рисунке 8 изображено подключение по WebSocket.

```
var connection = new WebSocket('ws://localhost',['soap', 'xmpp']);
var P2P = require('socket.io-p2p');
var io = require('socket.io-client');
var socket = io();
var server = require('http').createServer();
var io = require('socket.io')(server);
var p2p = require('socket.io-p2p-server').Server;
io.use(p2p);
server.listen(3030);

var p2p = new P2P(socket);

p2p.on('ready', function(){
  p2p.usePeerConnection = true;
  p2p.emit('peer-obj', { peerId: peerId });
})
```

```
// this event will be triggered over the socket transport
// until `usePeerConnection` is set to `true`
p2p.on('peer-msg', function(data){
  console.log(data);
});
```

Рисунок 8 – Подключение с помощью WebSocket

Используем обработчики событий для получения сведений о новых подключениях, ошибках и входящих сообщений. С помощью события onopen, мы проверяем, открыто ли соединение между сервером и клиентом. Если же оно открыто, то отправляем и регистрируем сообщение на сервер. За регистрацию сообщения отвечает событие onmessage. Далее создается событие onerror для проверки соединения на ошибки. Программный код с описанными функциями изображен на рисунке 9.

```
connections.onopen = function (){
        connection.send('Good');
};
connections.onmessage = function (e){
        console.log('Server: ' + e.data);
};
connections.onerror = function (error){
        console.log('WebSocket Error: ' + error);
};
```

Рисунок 9 – Основные функции для проверки соединения

После установки соединения с сервером, ему можно отправлять данные с помощью метода send. Данный метод поддерживает не только строки, но и двоичные сообщения. Для двоичных сообщений используются объекты Blob и ArrayBuffer. В данном случаи я воспользуюсь объектом ArrayBuffer. Создадим изображение и закодируем его в двоичный код. Затем перекодируем в беззнаковый целочисленный массив данных и отправим с помощью объекта ArrayBuffer. На рисунке 10 изображено отправление данных с помощью объектов.

```
connection.send('message');

var img = canvas_context.getImageData(0,0,400,320);
var binary = new Uint8Array(img.data.length);

for(var i = 0; I < img.data.length; i++){
        binary[i] = img.data[i];
}
```

```
Connection.send(binary.buffer);
```

Рисунок 10 – Отправка данных с помощью метода send и объекта ArrayBuffer

Сервер также может отправить нам сообщение. И при отправке каждый раз срабатывает обратный вызов onmessage. Вызов получает объект события. Если получаемое сообщение является двоичным, то необходимо задать свойству binaryType объекта WebSocket значение arraybuffer. Далее мы считаем длину двоичного кода и записываем в отдельную переменную. В дальнейшем переменная пригодится для функций автоматизированного тестирования.

На рисунке 11 изображена установка свойства binaryType и создание переменной.

```
connection.binaryType = ‘arraybuffer’;
connection.onmessage = function(e){
        console.log(e.data.bytelenght);
        var lenBuf = e.data.bytelenght;
        return lenBuf;
};
```

Рисунок 11 – Установка свойства binaryType

2.8 Реализация p2p сети

Реализация p2p сети происходит следующим образом: мы создаём массив объектов. Затем класс, описывающий сервер, который принимает значение: имя сервера, порт и ip-адрес. Далее записан обработчик нашего основного события – connection. Данное событие срабатывает каждый раз, когда пользователь подключается к нашему серверу (открывает новый сокет). При срабатывание события, в переменную socket попадает объект с открытым сокетом. Поэтому мы имеем возможность обрабатывать одни и те же события для каждого сокета индивидуально, записав обработчик событий внутри события connection. На рисунке 12 изображен обработчик connection и основной класс реализации сервера.

```
class Server{
        constructor(name, port, host){
                this.name = name;
                this.port = port;
                this.host = host;
```

```
        }
}
io.on('connection', function(socket){}
```

Рисунок 12 – Обработчик connection

На рисунке 13 создаём узел сети и прослушаваем входящие сообщения. Обработчик вызывается каждый раз, когда новое соединение готово.

```
var peer = new Peer(
  {
    binaryType: 'arraybuffer',
    video: false,
    audio: false
  }
);
peer.listen();

var connections = {};

peer.onconnection = function(connection) {

  connections[connection.id] = connection;

  connection.ondisconnect = function(reason) {
    delete connections[connection.id];
  };

  connection.onerror = function(error) {
    console.error(error);
  };
```

Рисунок 13 – Создание узла

Также добавлена функция usersCountToLog, которая выводит в лог сервера количество подключенных клиентов и используется каждый раз, когда входит или выходит новый участник сети. На рисунке 14 изображена функция usersCountToLog.

```
function usersCountToLog(){
        logger.info('User count: ' + io.engine.clientsCount);
}
```

Рисунок 14 – Функция usersCountToLog

При подключение к сети, сервер отправляет запрос этому клиенту и ждёт ответ. Если ответ удовлетворяет, то сервер добавляет его в список подключенных пользователей и рассылает всем остальным клиентам информацию о том, что этот пользователь успешно подключился к сети, а самому пользователю о том, что он успешно был подключен. Затем отправляется список всех подключенных пользователей. Если ответ пустой, то

сервер снова направляет запрос, пока не получит подходящий ответ. Данная логика помещена в отдельную функцию setName, внутри обработчика connection. На рисунке 15 изображена функция setName.

```
function setName(name){
        if(name != undefined && name != ''){
                socket.session = {};
                socket.session.userName = name;
                socket.session.address = socket.handshake.address;
                socket.session.id = socket.id;

                socket.broadcast.emit('newUser', socket.session);
                socket.emit('userName', socket.session);

                socket.emit('userList', io.length);

                logger.info('User ' + socket.session.userName + 'join from IP: ' +
socket.session.address);

                usersCountToLog();
                var clients = io.sockets.connected;

                var clientsList = {};
                for(var key in clients){
                        if(clients[key].session)
                                clientsList[key] = clients[key].session;
                }

                socket.emit('clientsList', clientsList);
        }else {// если имя пустое
                socket.emit('setName');
        }
}
setName(null);
```

Рисунок 15 – Функция setName

При вызове функции setName с пустым параметром, сервер отправит клиенту запрос этого имени, которые мы обрабатываем в файле main.js. В этом файле мы задаем несколько переменных, в которые записаны порт и адрес для соединения, затем открываем сокет по этим данным. На рисунке 16 изображены стандартные параметры сервера.

```
var port = 3000;
var server = '127.0.0.1';
var socket = io.connect(server + ':' + port);
var userLogin = new String;
```

Рисунок 16 – Стандартные параметры сервера

Сервер при первом подключении отправляет запрос на подключение, инициализируя событие setName.

2.9 Реализация автоматизированной системы тестирования

Реализация автоматизированной системы тестирования происходит следующим образом: создаём функцию, которая ограничивает количество передаваемых данных и функцию для обрыва связи между пользователями. Далее после установки связи между клиентами, мы запускаем данную функцию. Потом через определенное время срабатывает функция обрыва связи и проверяется какое количество данных было передано и как сильно повлияло на целостность отправленных данных.

Создана сеть серверов, находящиеся в массиве объектов, которые при получении данных проверяют их на целостность. С помощью функций регулярно обрывается связь между ними. После обрыва сети проверяются данные на целостность. Сервера имеют функции приема и отдачи данных. Если данные корректны, то они отправляют их клиентам. Клиенты отправляют и получают данные. Сервера получают и отправляют не только клиентам, но и другим серверам. Связь между серверами и клиентами осуществляется с помощью WebSocket. Полученные алгоритмы были протестированы и рассчитаны на работоспособность при большом размере сети.

Система тестирования реализована на программной платформе node js и на языке программирования JavaScript. Используя фрейморк express, API Socket и Socket.IO. С помощью этих инструментов мы будем передавать данные между пользователями и используя свойство bufferdAmount технологии WebSocket, сможем имитировать канал связи.

Библиотека Socket.IO осуществляет взаимодействие, тем способом, который подходит для его участников. Список доступных методов:

- WebSocket;
- Adobe Flash Player;
- Server-Sent Socket;
- XHR long polling;
- XHR multipart streaming;
- Forever iframe;

- JSONP Polling;
- ActiveX HTMLFile.

Стандартным методом является – WebSocket. Так как он имеет событие bufferdAmount, которое позволит ограничивать количество передаваемого двоичного кода в очереди, тем самым имитируя «плохой интернет» между клиентами и серверами. Очередь может возникать в результате вызова метода send().

Свойство socket.bufferdAmount определяет количество байтов, ожидающих отправки клиенту, а для клиента – сколько байтов ожидает отправки серверу. Количество данных для передачи функция принимает из массива. Для начала создадим массив с определенной скоростью передачи данных. На рисунке 17 изображен массив скорости передачи данных.

```
var speed = [1.2, 9.6, 57.6, 171.2, 326]; // в кбит/сек
```

Рисунок 17 – Массив скорости передачи данных

На стороне сервера создана функция sendArray для ограничения соединения. SendArray принимает два параметра, передаваемый двоичный код и количество данных, которое следует пропустить. На рисунке 18 изображена функция sendArray.

```
port = 11111;
host = '127.0.0.1';
buf_size = 1024;

function sendArrray(arrayBuffer, speed){
        listening_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
        listening_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1);
        listening_socket.bind((host, port));
        listening_socket.listen(backlog);
        if(arrayBuffer > speed){
                time.sleep(5);
        }
        console.log('socket.fufferedAmount: ' + socket.bufferedAmount);
}
```

Рисунок 18 – Функция sendArray

На сегодняшний день лучшим решением для написания приложений в режиме реального времени является технология WebSocket. Она позволяет создавать устойчивое соединение между клиентом и сервером, уменьшая

нагрузку на сервер по сравнению с остальными методами. Важность этой технологии подчеркивается также ее внедрением в наиболее популярные инструменты программирования как в JavaScript.

Глава 3. Тестирование математической модели, реализация математического эксперимента.

5.1 Анализ результатов моделирования

Тестирование приложения является важной частью реализацией алгоритма. В реальных системах входной буфер имеет ограничения на объем хранимых данных. И получается, что при поступлении большого числа пакетов на вход может случиться отказ системы, который в результате приведет к потере данных.

Для проведения математического эксперимента протестируем алгоритм со следующими предопределенными параметрами. Воспользуемся формулами (9, 10, 11) с учётом различных значений w = {5, 10, 15, 20}, показателя Херста H = {0.7, 0.8, 0.95} и для различных коэффициентов девиации C = 8, 16. На рисунке 19 даны графики зависимости вероятности потери от загрузки канала.

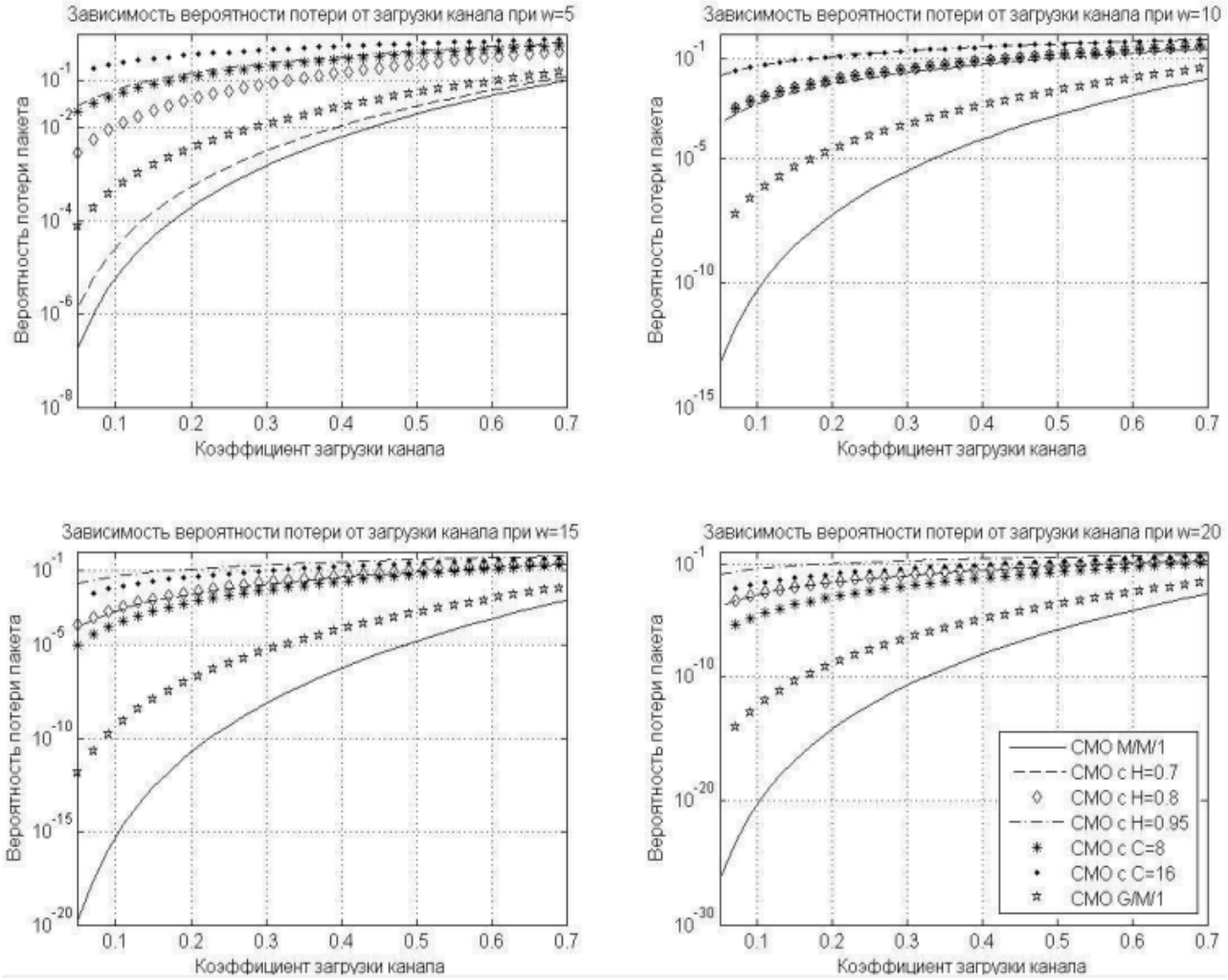

Рисунок 19 – Диапазон изменения вероятностей при различных w

Наиболее оптимистичную оценку даёт формула (11) модель M/M/1[2]. Данную оценку используем как нижнюю границу вероятности потери пакетов для заданного объему БЗУ и известном коэффициенте загрузки канала ρ. Наибольшая вероятность потери пакетов данных наблюдается при использовании модели M/M/1 с индексом Херста 0.95. С ростом трафика и с увеличением коэффициента девиации вероятность потери пакета возрастает. На рисунке 20 показано, что оценка вероятности потери пакетов данных растёт при учёте характера трафика.

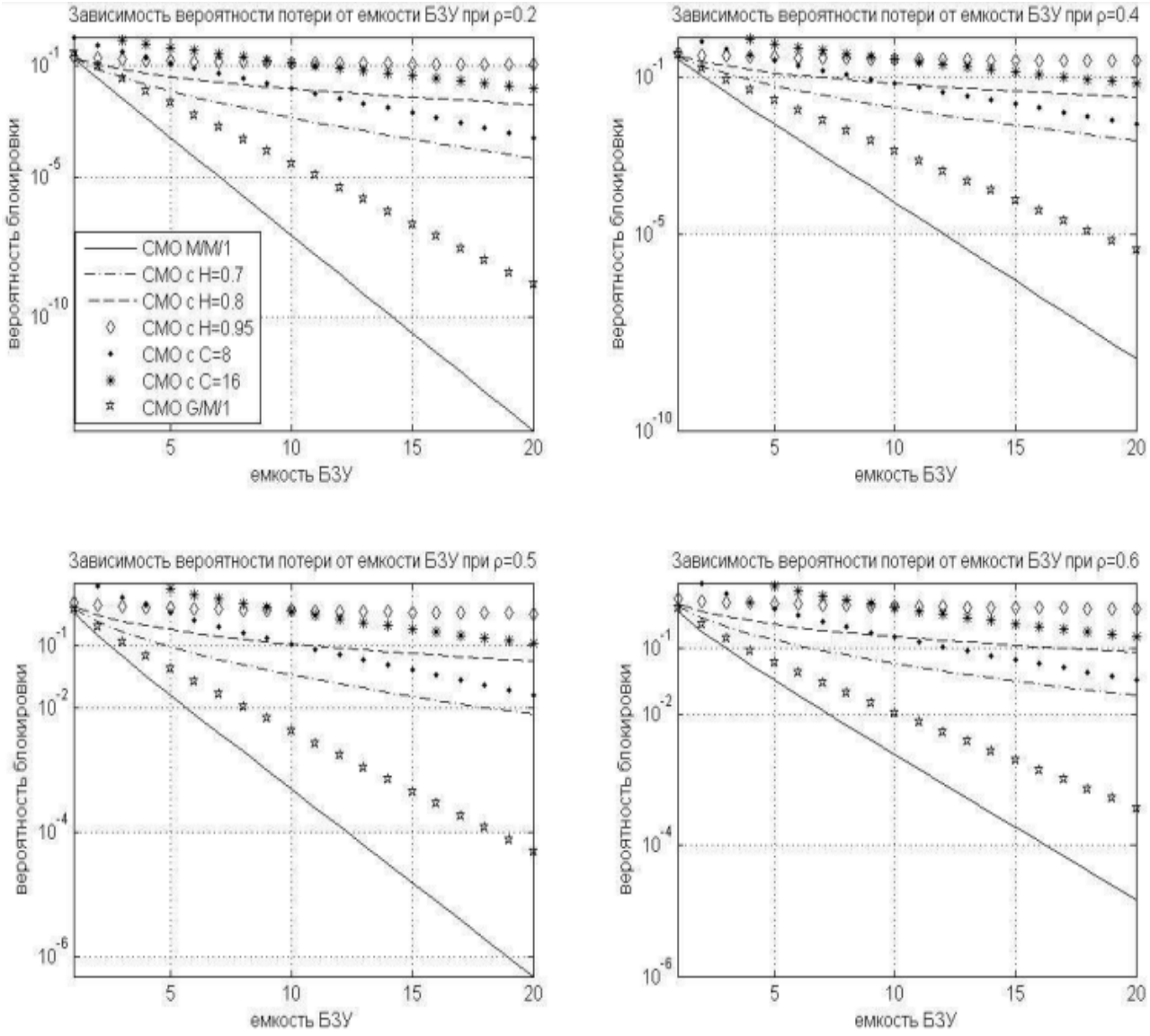

Рисунок 20 – Зависимость вероятности потери от емкости БЗУ при различных ρ

Если известен тип трафика, можно определить диапазон изменения вероятностей потери пакетов данных

Каждый тип трафика имеет свою степень фрактальности. Смотрите на таблицу 1.

Тип трафика	Фрактальность трафика (H)
Ethernet	0.9
HTTP	0.75 – 0.92
Видео	0.6 – 0.9
Аудио	0.6 – 0.9

P2P	0.6 – 0.9

Таблица 1 – Типы трафиков

Как видно из таблицы основные типы сетевых приложений имеют коэффициент Херста больше 0.6.

Доказано, что для уменьшения потери пакетов данных при передаче необходимо увеличивать БЗУ. В результате проведенных исследований можно сделать вывод, что фрактальность трафика в значительной мере увеличивает потерю данных при передаче

ЗАКЛЮЧЕНИЕ

В данной работе был спроектирован и разработан алгоритм системы тестирования для программного обеспечения p2p сетей. Для этого были рассмотрены и проанализированы ведущие технологии в области веб-программирования.

Система тестирования реализована на языке программирования JavaScript, используя свойство bufferdAmount. Так же была применена технология WebSocket, позволяющая имитировать канал связи. Создана гибридная p2p сеть. Была выбрана архитектура сети. Также решено было использовать WebSocket для эмуляции «плохого интернета». И выбран принцип тестирования.

Создана сеть серверов, находящиеся в массиве объектов, которые при получении данных проверяют их на целостность. С помощью функций регулярно обрывается связь между ними. После обрыва сети проверяются данные на целостность. Сервера имеют функции приема и отдачи данных. Если данные корректны, то они отправляют их клиентам. Клиенты отправляют и получают данные. Сервера получают и отправляют не только клиентам, но и другим серверам. Связь между серверами и клиентами осуществляется с помощью WebSocket. Полученные алгоритмы были протестированы и рассчитаны на работоспособность при большом размере сети.

Список используемой литературы

1. Создание сетевых приложений. Руководство разработчика, 2014. – 133 с.
2. Грекул, В. И. Проектирование информационных систем / Г.Н. Денищенко, Н.Л. Коровкина. – 2-е изд. – М.: Интернет-Ун-т Информ. Технологий: БИНОМ. Лаб. знаний, 2008. – 299 с.: ил.
3. Гагарина, Л.Г. Технология разработки программного обеспечения / Е.В. Кокорева, Б.Д. Виснадул. – Гриф УМО. – М.: ФОРУМ - ИНФРА-М, 2009. – 399 с.: ил.
4. Емельянова, Н.З. Проектирование информационных систем: учебное пособие / Н.З. Емельянова, Т.Л. Партыка, И.И. Попов. - М.: Форум, 2014. - 432 с.
5. Клочко И. А. Информационные технологии в профессиональной деятельности : учеб. пособие / И. А. Клочко. - Саратов : Вузовское образование, 2014. - 236 с.
6. Клименко, Р. Веб-мастеринг на 100% / Р. Клименко - Спб.: Питер, 2013. – 455 с.
7. Моделирование информационных систем. / Под ред. О.И. Шелухина.- М.: 2015.-368 с.
8. Нидерст-Роббинс, Д. HTML5, CSS3 и JavaScript. Исчерпывающее руководство / Д. Нидерст-Роббинсон - М.: Эксмо, 2014. – 528 с.
9. Роббинс, Д. HTML5. Карманный справочник / Д. Роббинс - М.: Вильямс, 2015. – 192 с.
10. Сухов, К. HTML 5. Путеводитель по технологии / К. Сухов - М.: ДМК Пресс, 2013. – 312 с.
11. Хоган Б., HTML5 и CSS3. Веб-разработка по стандартам нового поколения / Б. Хоган - Спб.: Питер, 2014. – 272 с.
12. Cederholm D., CSS3 for Web Designers. - A Book Apart, 2014.
13. Goldstein, A., HTML5 & CSS3 For The Real World. - SitePoint, 2015.

14. JavaScript. Подробное руководство. 7 изд. – 2015. - 992 с.

15. HTML5 AND WEB SOCKETS - For dummies Quick reference, 2015.- 224 с.

16. Socket.io – Real-time Web application development, 2013. - 140 с.

17. The definitive guide to HTML5 WebSocket – Build real-time applications with HTML5 – 2016. – 234 с.

18. High performance browser networking: What every web developer should know about networking and web performance – 1st edition – 2016. – 420 с.

19. Sawyer McFarland. D., CSS: The Missing Manual. - O'Reilly Media, 2015.

20. WebSocket – Lightweight Client-Server Communications, 2015. - 144 с.

ПРИЛОЖЕНИЕ А

Код файла server.js

```
GET /chat HTTP/1.1
HOST: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin:
Sec-WebSocket-Key: JHdadlkfhakj/dasfd/24r438rj==
Sec-WebSocket-Version: 13
var connection = new WebSocket(‘ws://localhost’,[‘soap’, ‘xmpp’]);
var P2P = require('socket.io-p2p');
var io = require('socket.io-client');
var socket = io();
var server = require('http').createServer();
var io = require('socket.io')(server);
var p2p = require('socket.io-p2p-server').Server;
io.use(p2p);
server.listen(3030);

var p2p = new P2P(socket);

var speed = [1.2, 9.6, 57.6, 171.2, 326]; // в кбит/сек

p2p.on('ready', function(){
  p2p.usePeerConnection = true;
  p2p.emit('peer-obj', { peerId: peerId });
})

// this event will be triggered over the socket transport
// until `usePeerConnection` is set to `true`
```

```
p2p.on('peer-msg', function(data){
  console.log(data);
});
for(var i = 0; I < img.data.length; i++){
        binary[i] = img.data[i];
}
Connection.send(binary.buffer);
connections.onopen = function (){
        connection.send('Good');
};
connections.onmessage = function (e){
        console.log('Server: ' + e.data);
};
connections.onerror = function (error){
        console.log('WebSocket Error: ' + error);
};
connection.binaryType = 'arraybuffer';
connection.onmessage = function(e){
        console.log(e.data.bytelenght);
        var lenBuf = e.data.bytelenght;
        return lenBuf;
};
class Server{
        constructor(name, port, host){
                this.name = name;
                this.port = port;
                this.host = host;
        }
}
io.on('connection', function(socket){}
```

```
connection.send(‘message’);

var img = canvas_context.getImageData(0,0,400,320);
var binary = new Uint8Array(img.data.length);
var peer = new Peer(
  {
    binaryType: 'arraybuffer',
    video: false,
    audio: false
  }
);
peer.listen();
var connections = {};
peer.onconnection = function(connection) {

  connections[connection.id] = connection;

  connection.ondisconnect = function(reason) {
    delete connections[connection.id];
  };

  connection.onerror = function(error) {
    console.error(error);
  };
function usersCountToLog(){
        logger.info(‘User count: ’ + io.engine.clientsCount);
}
function setName(name){
        if(name != undefined && name != ‘’){
                socket.session = {};
```

```
            socket.session.userName = name;
            socket.session.address = socket.handshake.address;
            socket.session.id = socket.id;

            socket.broadcast.emit('newUser', socket.session);
            socket.emit('userName', socket.session);

            socket.emit('userList', io.length);

            logger.info('User ' + socket.session.userName + 'join from IP: ' +
socket.session.address);

            usersCountToLog();
            var clients = io.sockets.connected;

            var clientsList = {};
            for(var key in clients){
                  if(clients[key].session)
                        clientsList[key]  = clients[key].session;
            }

            socket.emit('clientsList', clientsList);
      }else {// если имя пустое
            socket.emit('setName');
      }
}
setName(null);
var port = 3000;
var server = '127.0.0.1';
var socket = io.connect(server + ':' + port);
```

```
var userLogin = new String;
port = 11111;
host = '127.0.0.1';
buf_size = 1024;

function sendArrray(arrayBuffer,  speed){
        listening_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
        listening_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
1);
        listening_socket.bind((host, port));
        listening_socket.listen(backlog);
        if(arrayBuffer  > speed){
                time.sleep(5);
        }
        console.log('socket.fufferedAmount: ' + socket.bufferedAmount);
}
```