МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
Кафедра телекоммуникаций и информационных технологий

АЛМАКОВ
Никита Андреевич

РАЗРАБОТКА ВЫСОКОПРОИЗВОДИТЕЛЬНОЙ СИСТЕМЫ ОБРАБОТКИ И ВИЗУАЛИЗАЦИИ СТАТИСТИКИ В РЕАЛЬНОМ ВРЕМЕНИ

Дипломная работа

Научный руководитель:
старший преподаватель
Н.А. Зенькевич

Допущена к защите
«___» _____ 201_ г.
Зав. кафедрой телекоммуникаций и информационных технологий, доцент,
канд. физико-математических наук Ю. И. Воротницкий
_____

Минск, 2015

РЕФЕРАТ ДИПЛОМНОЙ РАБОТЫ

Алмакова Никиты Андреевича

Разработка высокопроизводительной системы обработки и визуализации статистики в реальном времени

Дипломная работа выполнена на 68 страницах А4, без приложений справочного или информационного характера. Включает 3 главы, 22 рисунка, 4 таблицы и 19 литературных источников.

Целью дипломной работы является разработка высокопроизводительной системы обработки и визуализации статистики в реальном времени.

Для достижения цели дипломного проекта была спроектирована архитектура серверного кластера, позволяющего горизонтально масштабировать каждый элемент приложения (сервера приложений, сервера баз данных, сервера рабочих процессов) независимо друг от друга. Разработано web-приложение, предоставляющее JSON RESTful API и WebSocket API для клиентских приложений и обеспечивающее обновление данных в режиме реального времени.

DIPLOMA THESIS ABSTRACT

Nikita Almakov

Development of a high-performance system of statistics processing and visualisation in real time.

The diploma thesis was accomplished on 68 pages of A4 format, without enclosures of informational purposes. It includes 3 chapters, 22 figures, 4 tables and 19 references.

The goal of the diploma thesis is the development of a high-performance system of statistics processing and visualisation in real time.

For diploma thesis goal attainment, server cluster architecture was designed to make possible to scale an each unit of an application horizontally and independently of one another (whether it is an application server, a database server or a server with working processes). The developed web-application provides JSON RESTful API and WebSocket API for client-based applications and ensures realtime data updates.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Изначально World Wide Web (WWW) представлялась ее создателям как «Пространство для обмена информацией, в котором люди и компьютеры смогут общаться между собой» [1]. Поэтому первые web-приложения представляли собой примитивные файл-серверы, возвращавшие статические HTML-страницы.

Однако на сегодняшний день web-приложения ушли достаточно далеко от простых статических страниц и накладывают серьезные ограничения на инфраструктуру, необходимую для их работы. Большинство современных web-приложений, вне зависимости от их направленности, динамичны и позволяют пользователю взаимодействовать с сервером без перезагрузки страницы. Более того, некоторые web-приложения позволяют получать обновления данных в режиме реального времени. Однако данные возможности требуют не только широкий и надежный канал связи, но высокую скорость работы самого приложения, что можно обеспечить только высокой эффективностью хранения и обработки данных.

Размеры современного интернет-пространства также накладывают серьезные ограничения на web-приложения — они должны легко масштабироваться и оставаться надежными под изменяющейся нагрузкой. Редко можно встретить приложение, имеющее заметную коммерческую ценность и менее ста тысяч пользователей. Такие масштабы требуют особых подходов к проектированию архитектуры приложения и использования специальных технологий.

Выбранная предметная область — высокопроизводительные системы обработки и визуализации статистики в реальном времени, весьма актуальна и бурно развивается. Такая сфера, как онлайн симуляторы менеджера спортивных команд стремительно набирает популярность: у ESPN, CBS и Yahoo! Sports, крупнейших американских игроков спортивного интернета, существуют целые подразделения, занимающиеся разработкой и развитием web-приложений, предоставляющих пользователям возможность собирать свои спортивные

команды и соревноваться с другими пользователями в системе виртуальных лиг и турниров, следить за ходом турниров в реальном времени и получать подробнейшую статистику по каждому аспекту данной предметной области. Число пользователей данных приложений крайне велико — по данным «Fantasy Sport Trade Association» за 2013 год в США и Канаде количество игроков превысило 41 миллион человек. Размер этого рынка оценивается в 3-4 миллиарда долларов.

Целью данного дипломного проекта является разработка высокопроизводительной системы обработки и визуализации статистики в реальном времени в виде серверного кластера масштабируемого web-приложения реального времени, реализующего симулятор менеджера спортивных команд для нескольких видов спорта.

Для достижения этой цели необходимо выполнить следующие задачи:

- проанализировать основные подходы к проектированию высоконагруженных масштабируемых web-приложений реального времени.
- опираясь на результаты предыдущего этапа, спроектировать архитектуру серверного кластера для разрабатываемого web-приложения, основные компоненты данного кластера и механизм их взаимодействия.
- реализовать и развернуть web-приложение симулятора менеджера спортивных команд.
- провести тестирование приложения и доработку по мере необходимости.

СПИСОК УПОТРЕБЛЯЕМЫХ СОКРАЩЕНИЙ И ТЕРМИНОВ

API (Application Programming Interface, «Интерфейс программирования приложений» либо «Интерфейс прикладного программирования») — Набор готовых констант, структур и функций, используемых при программировании пользовательских приложений и обеспечивающих правильное взаимодействие между пользовательским приложением и системой.

REST (Representation State Transfer, «Передача состояния представления» либо «Передача репрезентативного состояния») — Стиль построения архитектуры распределённого приложения.

JSON (JavaScript Object Notation) — Текстовый формат обмена данными, основанный на JavaScript.

BSON (Binary JavaScript Object Notation) — Бинарное представление формата JSON.

HTTP (HyperText Transfer Protocol, «Протокол передачи гипертекста») — Протокол прикладного уровня передачи данных.

SOAP (Simple Object Access Protocol, «Простой протокол доступа к объектам») — Протокол обмена структурированными сообщениями в распределённой вычислительной среде.

YAML (YAML Ain't Markup Language, «YAML — не язык разметки») — Человекочитаемый формат сериализации данных, ориентированный на удобство ввода-вывода типичных структур данных различных языков программирования.

1. АНАЛИЗ МЕТОДОВ И СРЕДСТВ РАЗРАБОТКИ МАСШТАБИРУЕМЫХ WEB-ПРИЛОЖЕНИЙ РЕАЛЬНОГО ВРЕМЕНИ

1.1 Обзор возможных вариантов архитектуры web-приложений

Web-приложение можно определить, как программную систему клиент/сервер, в состав которой, как минимум, входят следующие архитектурные компоненты: HTML/XML-браузер на одном или более клиентских компьютерах, взаимодействующих с web-сервером по протоколу HTTP, и сервер приложений, который управляет бизнес-логикой. Из определения не следует, что в web-приложении нельзя использовать распределенные объекты, а также то, что Web-сервер и сервер приложений не могут размешаться на одном и том же компьютере.

На достаточно высоком уровне абстракции можно выделить существующие в настоящее время архитектурные шаблоны web-приложений. Архитектурный шаблон отражает фундаментальную структурную и организационную схему программных систем. Он предоставляет набор предопределенных подсистем, описывает спектр их обязанностей, а также представляет правила и рекомендации для организации взаимодействия между ними. Рассмотрим два наиболее используемых в настоящее время архитектурных шаблона web-приложений. [2]

1.1.1 Странично-ориентированная архитектура

Наиболее распространенный на данный момент вариант архитектуры web-приложений. Странично-ориентированная архитектура включает в себя следующие компоненты:

- Клиентский браузер (англ. Client browser) — любой стандартный браузер, поддерживающий язык разметки HTML. Браузер функционирует как

обобщенное устройство с интерфейсом пользователя. Пользователь приложения посредством браузера запрашивает с сервера статические либо динамически web-страницы в формате HTML. На возвращаемой странице содержится полностью отформатированный пользовательский интерфейс — текст и управляющие элементы, которые отображаются браузером на экране клиентского компьютера.

• Сервер приложений (англ. Application server) — программная платформа, предназначенная для эффективного исполнения процедур, программ, скриптов, реализующих основную бизнес-логику приложения. Серверное приложения как правило реализуется с применением шаблона проектирования «модель-представление-контроллер» (англ. «Model-View-Controller», MVC), концепция которого впервые была описана Т. Реенскаугом [3]. Основная цель применения данного шаблона состоит в отделении бизнес-логики (контроллера) от данных (модели) и их представления.

1. Модель предоставляет остальным компонентам приложения объектное представление данных с помощью технологии объектно-реляционного отображения (англ. Object-relational mapping, ORM). Объекты модели осуществляют загрузку и сохранение данных в реляционную базу данных, а также реализуют бизнес-логику.

2. Представление создает пользовательский интерфейс для отображения полученных контроллером данных, а также передает запросы пользователя на манипуляцию данными в контроллер. Представление как правило является шаблонами — файлами HTML с дополнительным включением фрагментов кода на одном из используемых языков программирования. Вывод, сформированный данными фрагментами кода, включается в текст шаблона, после чего получившаяся страница HTML возвращается пользователю.

3. Контроллер — основной компонент, отвечающий за взаимодействие с пользователем. Контроллер считывает необходимые данные из модели и подготавливает их для представления, а также сохраняет полученные от представления данные в модели.

• Страница HTML — web-страница с интерфейсом пользователя и некоторой содержательной информацией, динамически формируемой на стороне сервера в зависимости от параметров запроса.

Общая схема странично-ориентированной архитектуры представлена на рисунке 1.1

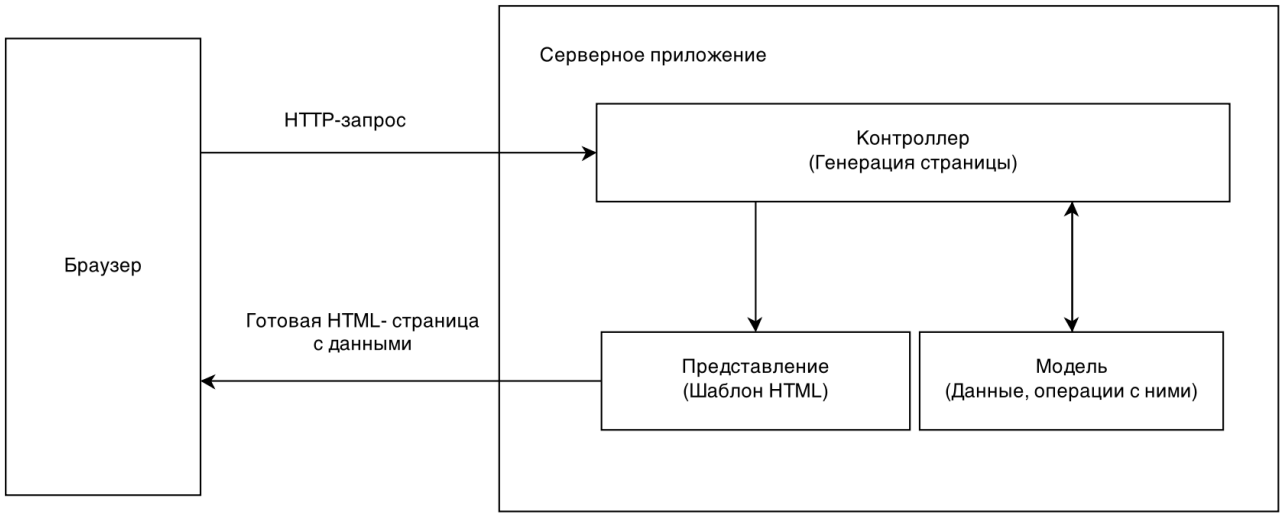

Рисунок 1.1 — Странично-ориентированная архитектура

В целом странично-ориентированная архитектура проста в реализации, поддерживается подавляющим большинством инструментов для создания web-приложения и достаточна для большинства приложений, в работе с которыми пользователям не требуется особая интерактивность.

1.1.2 Сервис-ориентированная архитектура

Сервис-ориентированная архитектура представляет собой модульный подход к разработке приложений, основанный на использовании распределенных, слабо связанных, заменяемых компонентов, оснащенных стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам.

В контексте разработки web-приложений, сервис-ориентированная архитектура означает, что сервер приложений занимается не формированием HTML-страниц, а предоставляет некоторый стандартизированный API, посредством которого с ним взаимодействуют пользовательские приложения

(одностраничные приложения, выполняемые в браузере; мобильные приложения) и другие сервисы.

В разработке сервис-ориентированных web-приложений существует два основных подхода: подход на основе удаленного вызова процедур (англ. Remote Procedure Call, RPC) и подход на основе передачи состояния представления (англ. Representational State Transfer, REST). В таблице 1.1 дано краткое сравнение данных подходов.

Таблица 1.1 — Сравнение RPC и REST подходов

Критерий сравнения	RPC	REST
Используемый протокол	XML-RPC, SOAP, WSDL	HTTP
Основная концепция	Объекты и методы	Ресурсы
Модель работы	Рабочий процесс с сохранением клиентского состояния	Передача состояния без сохранения
Доступ	Имя удаленной процедуры и методы кодируются в теле запроса	URL в качестве URI ресурса
Используемый формат данных	XML	Любой (как правило JSON)
Типизация	Строгая	Отсутствует
Ошибки	Кодируются в соответствии со специальным стандартом в теле ответа	Коды состояния HTTP
Интроспекция	Полная с использованием WSDL	Не используется
Кэширование	Отсутствует на уровне протоколов	Стандартное для HTTP

Подход на основе удаленного вызова процедур базируется на использовании протокола SOAP (англ. Simple Object Access Protocol, простой протокол доступа к объектам), являющегося расширением протокола XML-RPC [4]. В основе интерфейсов сервисов SOAP лежит концепция, основанная на объектах и их методах. SOAP активно использует XML для кодирования запросов и ответов, а также строгую типизацию данных, гарантирующую их целостность при передачи между клиентом и сервером. Детали сервисного запроса, такие как имя удаленной процедуры и входные аргументы, кодируются в теле запроса. Роль протокола HTTP сводится к передаче запросов SOAP от клиента к серверу с помощью метода POST. Модель SOAP поддерживает определенную степень интроспекции, позволяя разработчикам сервиса описывать его API в файле формата Web Service Description Language (WSDL, язык описания web-сервисов). Клиенты SOAP могут автоматически получать из этих файлов подробную информацию об именах и сигнатурах методов, типах входных и выходных данных и возвращаемых значениях.

Подход на основе REST был описан и популяризован Р. Филдингом, одним из создателей протокола HTTP [5]. В основе данного подхода лежит концепция ресурса, взаимооднозначно определяемого его URI (англ. Uniform Resource Identifier, унифицированный идентификатор ресурса), роль которого в web-сервисах, построенных по архитектуре REST, играет URL-адрес. Набор методов ресурса ограничен четырьмя базовыми методами HTTP --- POST, GET, PUT, DELETE, соответствующими операциям создания, получения, модификации и удаления ресурса. Обработка ошибок основана на использовании кодов состояния HTTP. В качестве формата данных как правило используется JSON. Сетевой ресурс не должен сохранять состояние клиента между парами «запрос-ответ». Общая схема RESTful JSON web-сервиса представлена на рисунке 1.2.

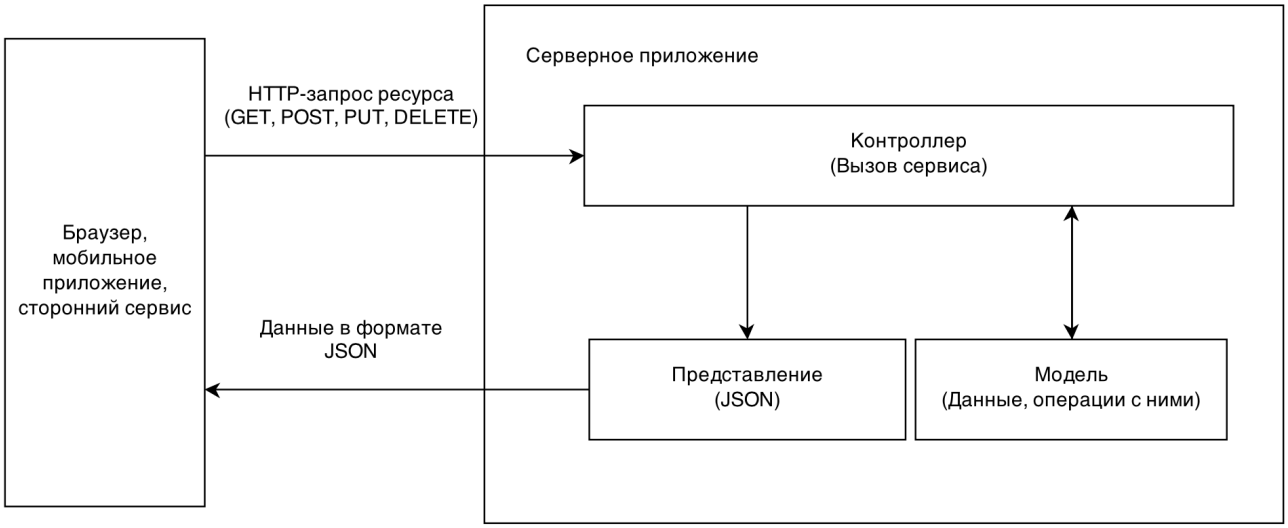

Рисунок 1.2 — Архитектура RESTful JSON веб-сервиса

В целом можно сказать, что подход на основе RPC стоит применять при разработке web-сервисов со сложной, требовательной к надежности, логикой взаимодействия, выходящей за рамки базовых операций создания, получения, модификации и удаления, а также при разработке приложения на платформах, хорошо поддерживающих соответствующие стандарты, и языках со статической типизацией, таких как Java и .NET/C#.

Подход на основе REST целесообразно применять при разработке web-сервисов, логика которых хорошо укладывается в четыре базовых операции протокола HTTP, web-сервисов, основными клиентами которых будут приложения, написанные на языке Javascript, а также при высоких требованиях к производительности и масштабируемости.

1.2 Основные подходы к масштабированию web-приложений

1.2.1 Общие понятия о масштабируемости web-приложений

С ростом популярности web-приложения растет и нагрузка. Для обеспечения непрерывной работы web-приложения в течении всего цикла его жизнедеятельности необходимо еще на этапе проектирования обеспечить масштабируемость данного приложения. Под масштабируемостью понимается способность системы, сети или процесса справляться с увеличением рабочей

нагрузки (увеличивать свою производительность) при добавлении ресурсов (обычно аппаратных). Масштабируемость — важный аспект любого web-приложения, для которого требуется обеспечивать отказоустойчивость и работу под большими нагрузками. Система называется масштабируемой, если она способна увеличивать производительность пропорционально дополнительным ресурсам.

Масштабируемость можно оценить через отношение прироста производительности системы к приросту используемых ресурсов. Чем ближе это отношение к единице, тем лучше. Также под масштабируемостью понимается возможность наращивания дополнительных ресурсов без структурных изменений центрального узла системы.

В системе с плохой масштабируемостью добавление ресурсов приводит лишь к незначительному повышению производительности, а с некоторого «порогового» момента добавление ресурсов не приносит положительного эффекта.

Масштабирование бывает двух типов:

• Вертикальное масштабирование — увеличение производительности каждого компонента системы с целью повышения общей производительности. Масштабируемость в этом контексте означает возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий. Это самый простой способ масштабирования, так как не требует никаких изменений в прикладных программах, работающих на таких системах. Вертикальное масштабирование, как правило, ограничено текущем уровнем развития аппаратного обеспечения сервера и способностью приложений задействовать все предоставляемые им вычислительные ресурсы.

• Горизонтальное масштабирование — разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам (или их группам), и (или) увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Масштабируемость в этом

контексте означает возможность добавлять к системе новые узлы, серверы, процессоры для увеличения общей производительности. Этот способ масштабирования может требовать внесения изменений в программы, чтобы программы могли в полной мере пользоваться возросшим количеством ресурсов.

Ниже рассмотрены основные техники, обеспечивающие масштабируемость web-приложений.

1.2.2 Кластеризация

Кластер — группа серверов, объединенных логически, способных обрабатывать идентичные запросы и использующаяся как единый ресурс. Группа серверов обладает большей производительностью и большей надежностью (за счет избыточности), чем один сервер. Объединение серверов в один ресурс происходит на уровне программных протоколов.

В архитектуре высоконагруженных web-проектов обычно осуществляется кластеризация серверов приложений и баз данных. Для того, чтобы осуществить кластеризацию приложений, необходимо увеличить количество серверов, продублировать приложение на них и организовать балансировку нагрузки, т.е. распределение запросов между серверами приложений. Возможны два способа балансировки нагрузки:

- Балансирующий узел — в этом случае клиент шлет запрос на один фиксированный, известный ему сервер, а тот уже перенаправляет запрос на один из рабочих серверов. Преимущества данного подхода в том, что клиенту ничего не надо знать о внутреннем устройстве системы — о количестве серверов, об их адресах и особенностях --- всю эту информацию знает только балансировщик нагрузки. Однако недостаток данного подхода в том, что балансирующий узел является единой точкой отказа системы — если он выйдет из строя, вся система окажется неработоспособна. Кроме того, при большой нагрузке балансировщик может просто перестать справляться со своей работой, поэтому такой подход применим не всегда.

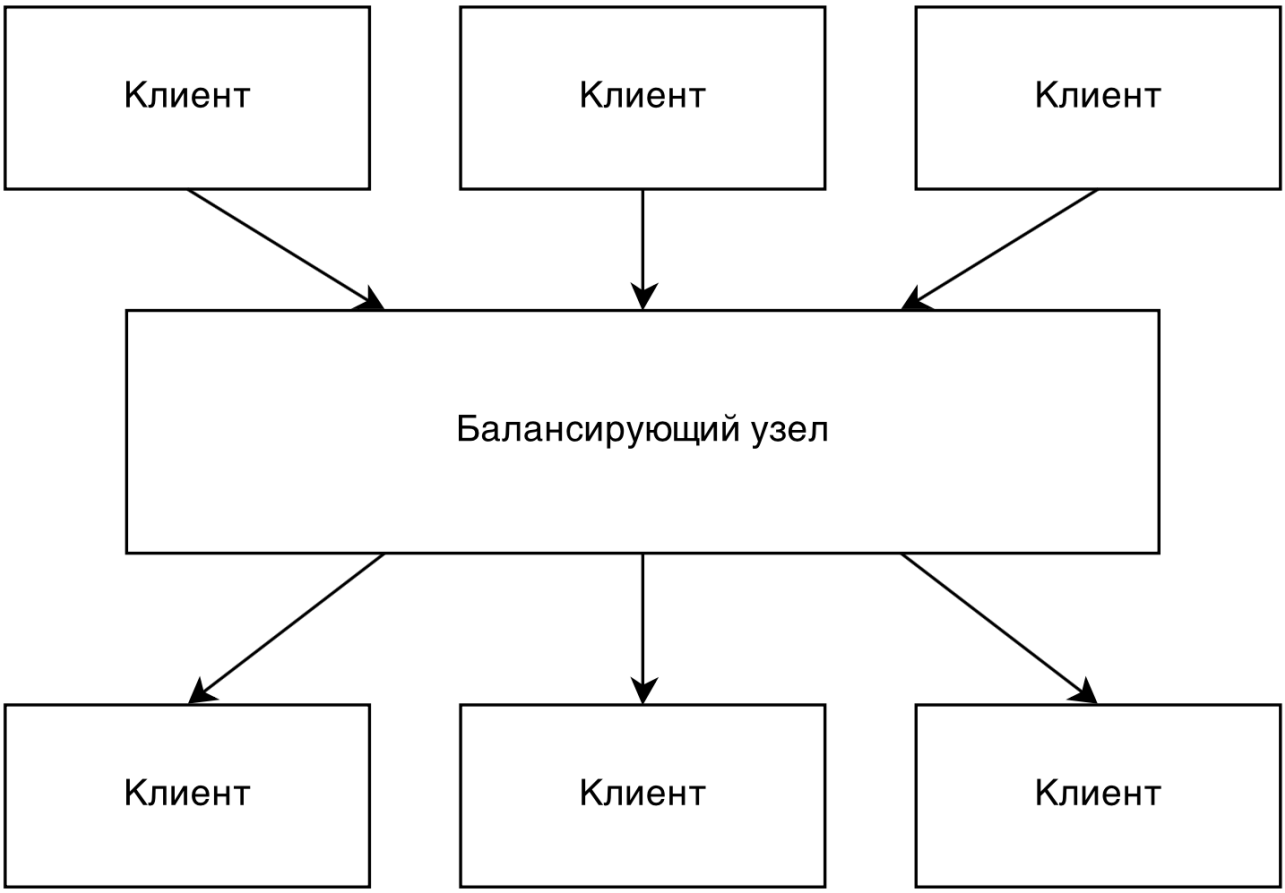

Рисунок 1.3 — Распределение нагрузки с использованием балансирующего узла

• Балансировка на стороне клиента — выбор используемого сервера осуществляется самим клиентом. В этом случае клиент должен знать о внутреннем устройстве системы, чтобы уметь правильно выбирать, к какому серверу обращаться. Плюсом данного подхода является отсутствие единой точки отказа. Минусом — усложнение логики клиента и меньшая гибкость балансировки.

Стоит отметить, что приведенные выше методы балансировки нагрузки не являются взаимоисключающими.

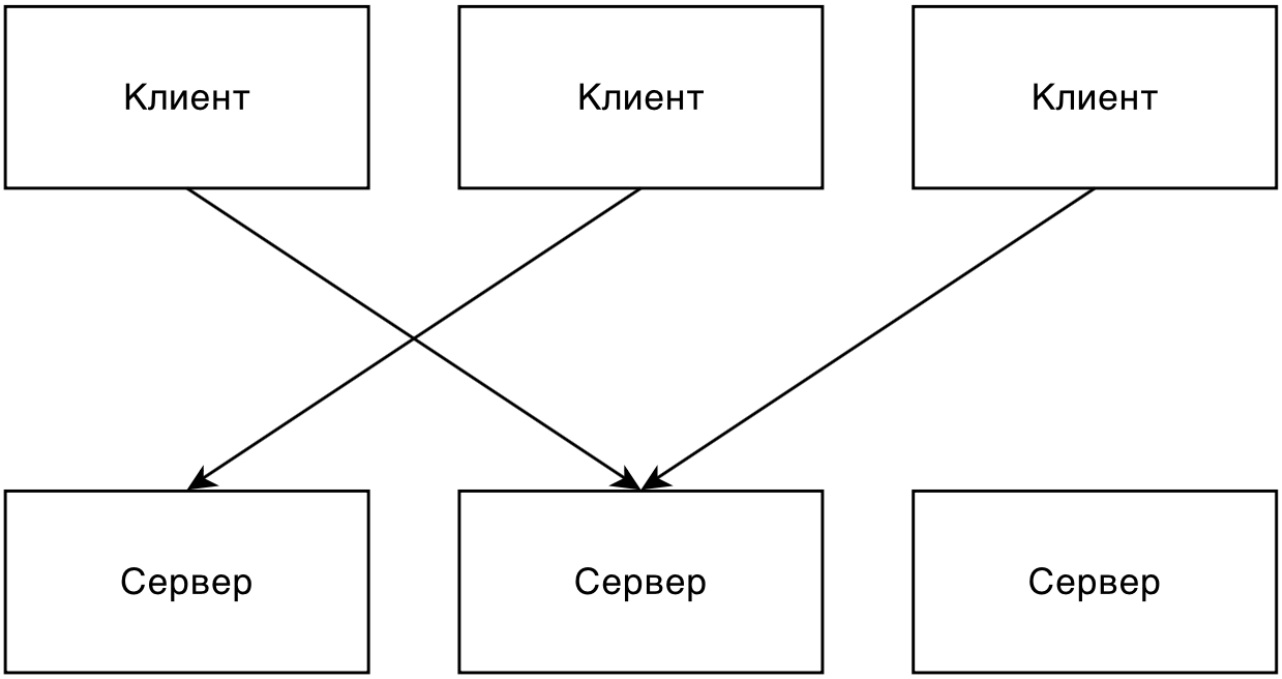

Рисунок 1.4 — Распределение нагрузки с использованием балансировки на стороне клиента

Кластеризация баз данных основана на применении двух подходов:

• Метод репликации — основан на синхронизации содержимого базы данных между несколькими серверами. Возможны две схемы репликации: схема «ведущий-ведомый» (master-slave), и схема «ведущий-ведущий» (master-master).

1. «ведущий-ведомый» (master-slave) — запросы, связанные с изменением данных, принимаются только ведущим сервером, а запросы, связанные с получением данных распределяются между ведомыми серверами. Ведомые сервера обновляют свои данные в соответствии с текущим состоянием ведущего сервера.

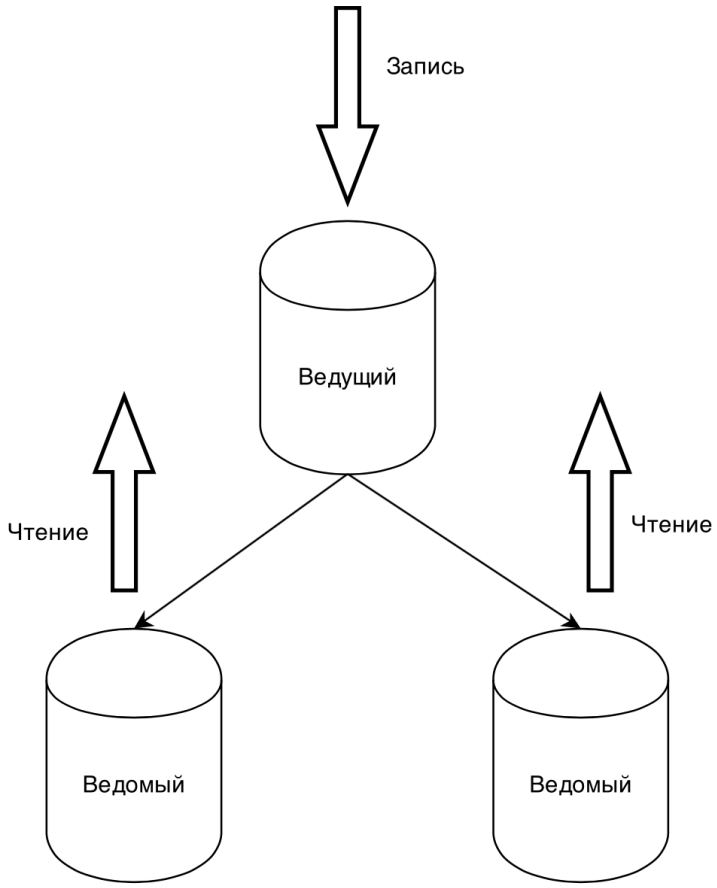

Рисунок 1.5 — Репликация по схеме «ведущий-ведомый»

2. «ведущий-ведущий» (master-master) — все запросы, как на чтение, так и на запись, распределяются между всеми серверами кластера базы данных.

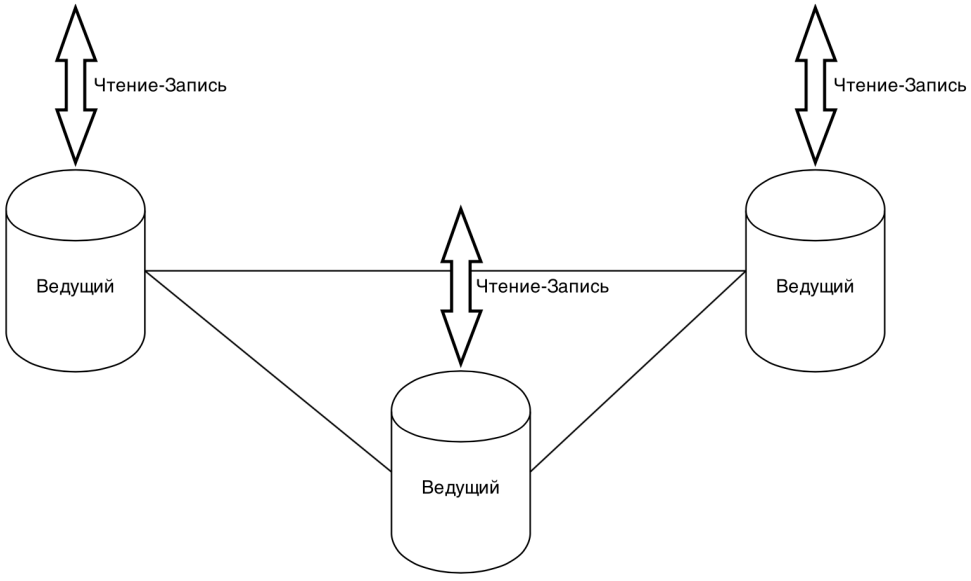

Рисунок 1.6 — Репликация по схеме «ведущий-ведущий»

• Механизм распределения — применяется в случае необходимости распределения нагрузки на операции записи данных, а также если базу данных невозможно разместить в пределах одного сервера. Возможны две схемы распределения:

1. Вертикальное распределение — в простейшем случае представляет собой вынесение отдельных таблиц базы данных на другой сервер. При этом потребуется изменить запросы, чтобы обращаться к разным серверам за разными данными. В пределе возможно хранить каждую таблицу на отдельном сервере (хотя на практике это вряд ли будет выгодно). При таком распределении теряется возможность делать SQL-запросы, объединяющие данные из таблиц, расположенных на разных серверах.

2. Горизонтальное распределение — заключается в распределении данных одной таблицы по нескольким серверам. Фактически, на каждом сервере создается таблица такой же структуры, и в ней хранится определенная порция данных. Распределять данные по серверам можно по разным критериям:

- по диапазону: записи с id меньше заданного хранятся на сервере А, остальные — на сервере Б
- по списку значений: записи типа «ЗАО» и «ОАО» сохраняем на сервер А, остальные — на сервер Б
- по значению хэш-функции от некоторого поля записи.

Горизонтальное разбиение данных позволяет хранить неограниченное количество записей, однако усложняет выборку. Наиболее эффективно можно выбирать записи только когда известно, на каком сервере они хранятся.

1.2.3 Кэширование

Техника кэширования основана на повышении производительности за счет сохранения результата повторяющихся операций, и использования его

вместо повторного выполнения операций, а также благодаря приближению данных к месту их использования. В типичной архитектуре web-приложения существует несколько мест для потенциального применения кэша:

- Кэширование запросов к базе данных — например, стоит задача постраничной навигации по списку некоторой выборки: определенной категории товаров или объектов. Классическое решение - это использование в операторе SELECT (на примере MySQL) ключевых слов LIMIT/OFFSET. Однако, если выбрать весь массив данных и сохранить его в кэше, а затем брать результаты выборки из кэша и делать операции выборки по страницам или сортировки по каким-то критериям уже на стороне клиента, то мы значительно сократим нагрузку на БД.
- Кэширование опкодов приложения — если приложение написано на интерпретируемом языке программирования, то процедура разбора, компиляции и выполнения сценария выполняется каждый раз для нового запроса. Разобранный сценарий представляет собой последовательность опкодов, которую можно поместить в кэш, устранив тем самым этапы разбора и компиляции сценария для каждого запроса.
- Кэширование динамических страниц — кэшируется результирующий HTML-файл (или его часть), формируемый сервером приложений и отдается непосредственно web-сервером.

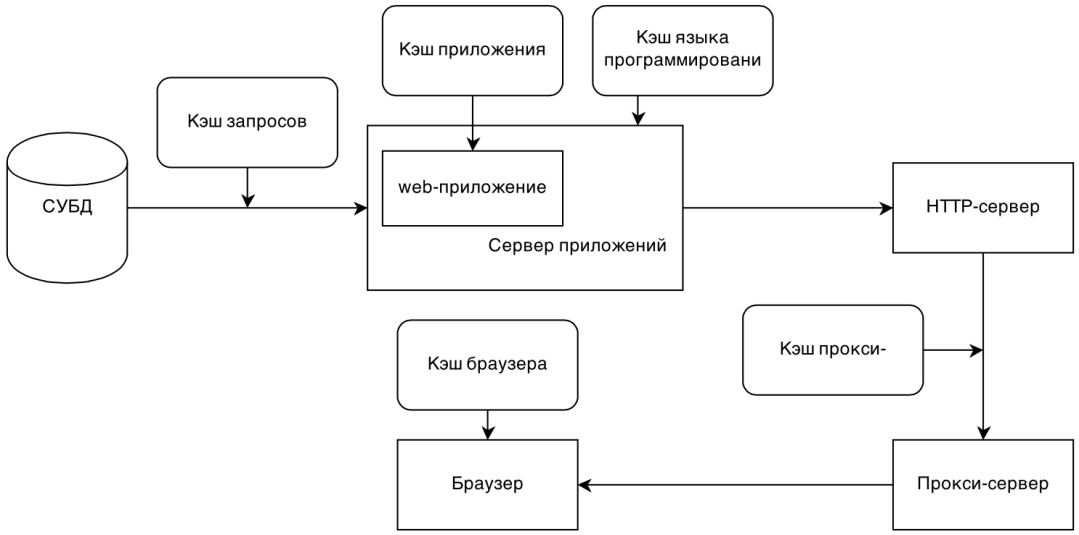

Рисунок 1.7 — Потенциальные места применения кэширования в web-приложении

Существуют три типа кэширования с различной механикой работы:

- Ленивый кэш (англ. Lazy cache) — Кэш сохраняет данные и отдает их, пока не истечет время устаревания. Данный тип кэширования целесообразно применять для данных, которые практически никогда не изменяются, иначе возможно получение устаревших данных.
- Синхронизированный кэш (англ. Synchronized cache) — Клиент вместе с данными получает метку последнего изменения и может спросить у поставщика не изменились ли данные, чтобы повторно их не запрашивать. Данный тип кэширования реализован в протоколе HTTP.
- Кэш сквозной записи (англ. Write-trough cache) — Любое изменение данных выполняется сразу и в хранилище и в кэше. Данные при таком типе кэширования не устаревают никогда, однако возможно возникновение так называемой когерентности кэшей. Например, одни и те же данные используются для формирования разных страниц и кешируются страницы. Страницы, сформированные позже, будут содержать обновленные данные, а страницы, закэшированные раньше, будут содержать устаревшие данные.

1.2.4 Асинхронное выполнение операций

Когда пользователь совершает запрос на сайт, он ожидает получить ответ, но для этого необходимо проделать соответствующую работу. Пусть, человек обновил запись в своем блоге, необходимо осуществить последовательность трудоемких операций помимо создания новой записи в базе данных: обновление счетчиков, оповещение «друзей», рассылка электронных уведомлений.

Однако, если все эти действия будут выполняться синхронно, система может превысить максимальный интервал ожидания ответа пользователем. В таком случае применяется следующий архитектурный шаблон — данные сохраняются в промежуточное хранилище и далее обрабатываются с помощью отдельного асинхронного рабочего процесса. Термин «асинхронность» означает в общем случае разнесенность операций во времени. То есть данные

собираются сейчас, а обрабатываются тогда, когда будет возможно. Зачастую требуется, чтобы промежуточное хранилище обладало определенными свойствами: сохранение порядка и очередности задач.

Для подобных целей используют инструмент, называемый очередями сообщений, — особый вид хранилища, поддерживающий логику FIFO (первый вошел — первый вышел). Образуются очереди сообщений и очереди задач, которые необходимо выполнить. Например, вместо того чтобы отправлять e-mail, можно поместить в очередь задачу: «Отправить e-mail». А далее фоновый рабочий процесс сможет получить данное сообщение и выполнить отправку e-mail-a.

В крупных web-системах могут использоваться одновременно десятки очередей: очередь на отправку электронной почты, очередь для обновления счетчиков, очередь для обновления новостных лент пользователей и т. д. В целом подход на основе асинхронного выполнения операций и использования очередей сообщений дает следующие преимущества:

- Разделение логики работы приложения на независимые друг от друга части, взаимодействующие через общий интерфейс, предоставляемый очередью сообщений.
- Обеспечение сохранности сообщений — в случае, если обработчик сообщений дал сбой, сообщения не будут утеряны, а будут накапливаться в очереди до тех пор, пока обработчик снова не будет исправен.
- Масштабируемость — не составляет труда увеличить количество процессов-обработчиков сообщений, в целях увеличения производительности.
- Эластичность — в случае резкого, аномального роста нагрузки система продолжит функционировать т.к. сообщения будут накапливаться в очереди и ждать обработки.
- Отказоустойчивость — если процесс, обрабатывающий сообщение из очереди аварийно завершит свое выполнение, данное сообщение может быть перенаправлено другому обработчику.

1.3 Основные подходы к реализации web-приложений реального времени

Обычно, когда пользователь посещает страницу web-приложения с помощью браузера, веб-серверу, на котором размещена эта страница, направляется HTTP-запрос. Веб-сервер подтверждает получение запроса и посылает браузеру ответ. Однако во многих случая — например, если речь идет о биржевых котировках, новостях, расписаниях движения транспорта и т.п.,— к тому времени, когда браузер обновит страницу, сведения, поступившие вместе с ответом, могут оказаться устаревшими.

Протокол HTTP — полудуплексный протокол передачи данных, инициатором соединения в котором выступает клиент. Для отображения изменений, произошедших на сервере, с помощью протокола HTTP существует несколько подходов [6]:

- Метод опроса (англ. polling) — при использовании данного метода браузер регулярно отправляет серверу запросы через определенные промежутки времени и получает на них немедленные ответы. Исторически, это был первый вариант доставки информации в браузер в режиме реального времени. Данное решение хорошо подходит для ситуаций, в которых периодичность появления сообщений, подлежащих отправке браузеру, точно известна, поскольку это позволяет синхронизировать клиентские запросы и отправлять их только тогда, когда на сервере имеется новая информация.
- Метод продленного запроса (англ. long-polling) — при использовании данного метода, браузер направляет серверу запрос, и тот удерживает соединение открытым в течении некоторого времени. Если в это время сервер получает уведомление, он отправляет клиенту ответ, содержащий сообщение. В противном случае сервер отправляет ответ, закрывающий соединение. В случае интенсивного обмена небольшими порциями данных метод продленного запроса не дает никакого преимущества по сравнению с методом опроса.

• Метод стриминга (англ. streaming) — в данном методе браузер посылает завершенный запрос, но сервер отправляет ответ, не приводящий к закрытию соединения. Как только появляется сообщение, готовое к отправке, ответ обновляется, но сигнал о завершении ответа не посылается. В результате этого клиент получает новые сообщения практически без задержки. Вместе с тем, поскольку стриминг остается инкапсулированным в HTTP, оказавшиеся на пути брэндмауэры и прокси-серверы могут создавать дополнительные проблемы.

Каждый из приведенных выше методов предоставления информации в реальном времени использует HTTP-заголовки запросов и ответов, в которых содержится много лишних дополнительных заголовочных данных, приводящих к увеличению времени задержки. Пытаясь имитировать полнодуплексное соединение поверх полудуплексного протокола HTTP, многие реализации данных подходов используют два соединения: одно для передачи данных от клиента к серверу, а второе — от сервера к клиенту. Необходимость обслуживания этих соединений и обеспечения их согласованной работы влечет за собой дополнительные накладные расходы в виде возрастания нагрузки на сервер, усложняет организацию всего процесса и ухудшает масштабируемость приложения.

Для решения данных проблем в спецификации HTML5 был предложен протокол WebSocket — полнодуплексный протокол связи, предназначенный для обмена сообщениями между браузером и сервером в режиме реального времени.

Соединение WebSocket устанавливается путем расширения протокола HTTP до протокола WebSocket в процессе первоначального обмена специальными подтверждениями (квитирование) между клиентом и сервером через одно и то же базовое TCP/IP соединение. Как только соединение установлено, кадры данных могут пересылаться в обоих направлениях между клиентом и сервером в полностью дуплексном режиме. Доступ к самому соединению осуществляется посредством события message и метода send,

определенных интерфейсом WebSocket. С 11 декабря 2011 года протокол имеет статус RFC [7] и на текущий момент поддерживается в следующих браузерах:

- Google Chrome (начиная с версии 4.0.249.0)
- Apple Safari (начиная с версии 5.0.7533.16)
- Mozilla Firefox (начиная с версии 4)
- Opera (начиная с версии 10.70 9067)
- Internet Explorer (начиная с версии 10)

Спецификация HTML5 WebSocket является огромным прогрессом в отношении масштабируемости web-приложений реального времени. Данный протокол позволяет уменьшить объем служебного трафика, связанного с передачей HTTP-заголовков, в 500, а в некоторых случаях — даже в 1000 раз, а время запаздывания — в 3 раза. Таким образом, использование протокола WebSocket является наиболее предпочтительным вариантом для реализации web-приложений реального времени.

2. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ВЫСОКОПРОИЗВОДИТЕЛЬНОЙ СИСТЕМЫ ОБРАБОТКИ И ВИЗУАЛИЗАЦИИ СТАТИСТИКИ В РЕАЛЬНОМ ВРЕМЕНИ

Проектирование системы реального времени включает в себя определение требований, разработку архитектуры системы, описание статических элементов системы и их взаимодействия. В качестве графического языка документации используется нотация UML 2.0 (Unified Modeling Language).

В соответствии с введением данной работы, в качестве системы обработки и визуализации статистики в реальном времени будет разработан симулятор менеджера спортивных команд.

2.1 Анализ сущностей и отношений предметной области симулятора менеджера спортивных команд

Симулятор менеджера (симулятор владельца, «фэнтези-спорт») — игра, в которой участники выступают в качестве владельцев виртуальных спортивных команд, конкурирующих с командами других игроков на основе актуальной статистики выступлений реальных игроков профессионального спорта.

Каждый спортсмен, доступный для набора в команду игрока, имеет прототип в реальности. Как правило, спортсмены должны играть в одном дивизионе или лиге определенной страны, но существует множество других вариантов. Каждый виртуальный спортсмен имеет стоимость и позицию, которые учитываются при формировании команды игрока. Стоимость спортсмена зависит от успешности выступления его реального прототипа. Позиция спортсмена соответствует позиции его прототипа.

Участникам предлагается набрать команду фиксированного размера, состоящую из различных позиций. Так, в типовой состав для футбола включается 1 вратарь, 4 защитника, 3-4 полузащитника и 2-3 форварда. При

этом позиция реального игрока и его позиция в виртуальной команде должны соответствовать друг другу. Для набора команды каждому игроку отводится фиксированный бюджет. Как правило, на виртуальную команду налагаются ограничения на число спортсменов-одноклубников (прототипы которых играют в одной реальной команде).

В небольших турнирах с малым числом участников команды формируются не на основе бюджета, а путем аукциона между игроками, либо же выбором по очереди. Это означает, что определенный спортсмен может быть включен только в одну команду и все набранные им баллы пойдут в зачет только его владельцу. Зачетные баллы начисляются спортсменам в зависимости от успешности выступления их реальных прототипов. Критерии набора баллов варьируются для различных видов спорта и для различных позиций спортсменов. Например, в футболе могут поощряться следующие достижения:

- футболист провел на поле полный матч;
- футболист забил гол;
- футболист отдал голевой пас;
- команда не пропустила ни одного мяча (для вратаря и защитников).

При этом величина поощрения за каждое достижения различна. За некоторые действия спортсмены могут быть оштрафованы, например:

- футболист забил автогол;
- футболист получил желтую карточку;
- вратарь пропустил гол;

Зачетные баллы команды определяются как сумма зачетных баллов всех ее участников. Сумма балов служит критерием определения победителя турнира.

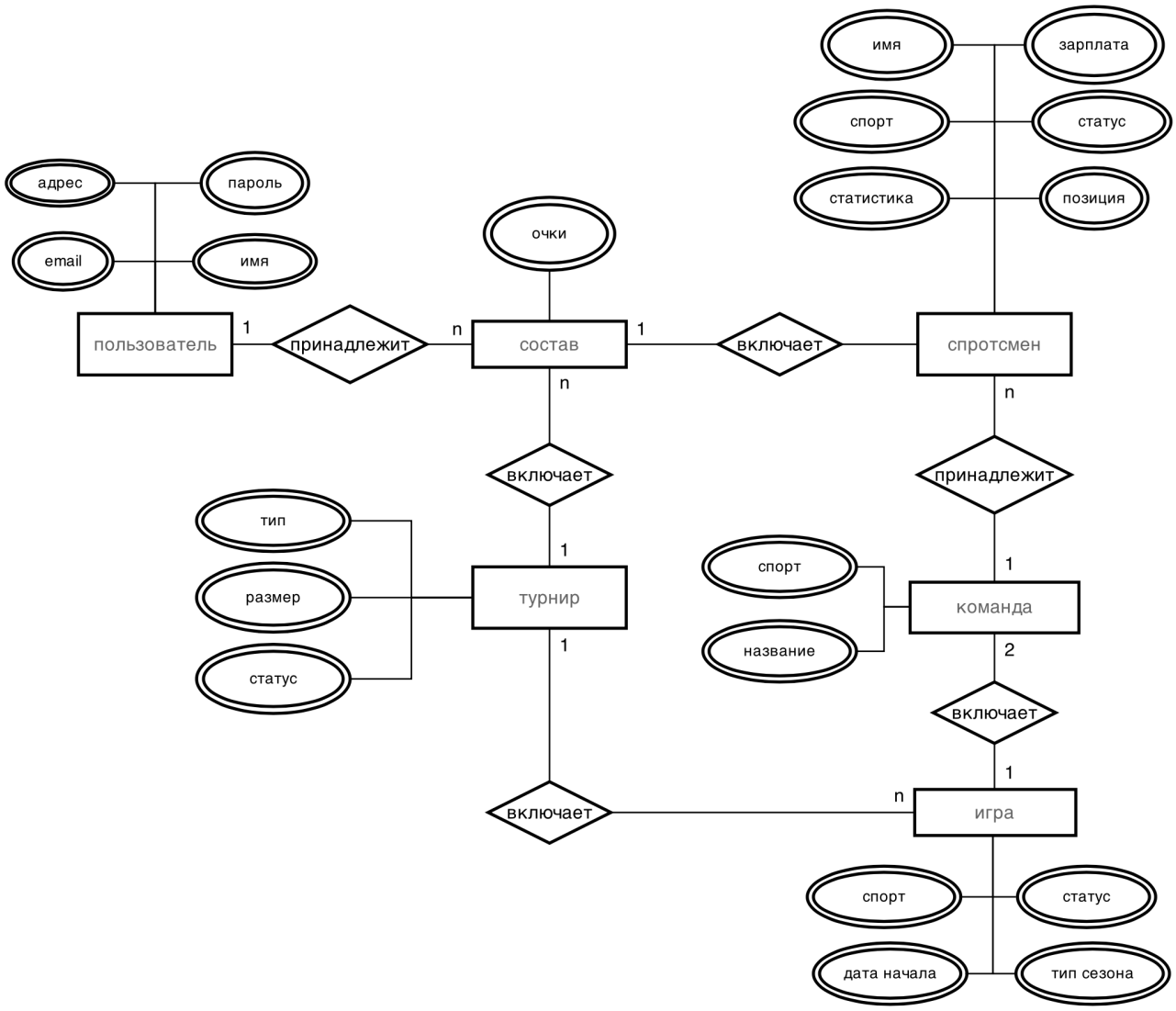

Рисунок 2.1 — Диаграмма сущность-связь

Турнир соответствует серии спортивных матчей в реальном мире, и продолжается от начала первого матча, до конца последнего. По продолжительности турниры, как правило, бывают однодневными, недельными и сезонными. Победитель турнира определяется по его окончании на основе величины зачетных баллов команд игроков. Игрок не может менять состав своей команды во время проведения реальных игр, но может менять состав между играми. Турниры различаются между собой различными параметрами: способом набора команд, доступным для этого бюджетом, количеством участвующих игроков, способом распределения призов и т.д.

На рисунке 2.1 изображена диаграмма сущность-связь для данной предметной области.

Всего было выделено 6 сущностей. Три из них: «спортсмен», «команда» и «игра» относятся к реальному спорту, остальные - к симулятору менеджера. Между сущностями выделено два типа отношений: отношение принадлежности и отношение включения. Данная диаграмма послужит основой при проектировании схемы базы данных.

2.2 Анализ требований, предъявляемых к проекту

Прежде чем приступать к проектированию, необходимо определиться с требованиями, предъявляемыми к разрабатываемому приложению. Основные требования, предъявляемые к разрабатываемому симулятору менеджера спортивных турниров:

- Приложение должно использовать API стороннего сервиса, поставляющего различную информацию об интересующих видах спорта в формате XML.
- Клиентское приложение должно поддерживать как персональные компьютеры, так и различные мобильные устройства.
- Приложение должно быть спроектировано с учетом необходимости распределения различных функций (реализация игровой логики, реализация API и т.д.) между разными серверами.
- Каждый компонент разрабатываемого приложения должен иметь возможность горизонтального масштабирования.
- Приложение будет развиваться непрерывно.
- Задержка между появлением новых данных в стороннем сервисе до момента отображения этих данных на клиенте должна быть минимальна (приложение должно функционировать в реальном времени).
- Приложение должно быть отказоустойчивым, функционировать под высокими нагрузками.

2.3 Варианты использования симулятора менеджера спортивных команд

Диаграммы вариантов использования описывают взаимоотношения и зависимости между группами вариантов использования и действующих лиц, участвующими в процессе. Они предназначены для упрощения взаимодействия с будущими пользователями системы, с клиентами, и особенно пригодятся для определения необходимых характеристик системы.

Другими словами, диаграммы вариантов использования говорят о том, какие действия будет выполнять система, не указывая при этом применяемые методы.

Разрабатываемое приложение имеет двух актантов: «Пользователь» «Администратор». На рисунке 2.2 представлена диаграмма вариантов использования для актанта «Пользователь».

Пользователь может осуществлять следующие действия:

- просмотреть список турниров, осуществляя его фильтрацию по различным параметрам (спорту, типа, дате начала и т.д.);
- набрать состав;
- просмотреть список атлетов для данного спорта, осуществляя его фильтрацию по различным параметрам (позиции, зарплате, команде и т.д.);
- просматривать результаты турнира в реальном времени.

Основная задача администратора — создание наборов игр и шаблонов турниров, на основании которых будут автоматически создаваться турниры, доступные для участия пользователям. На рисунке 2.3 представлена диаграмма вариантов использования для актанта «Администратор».

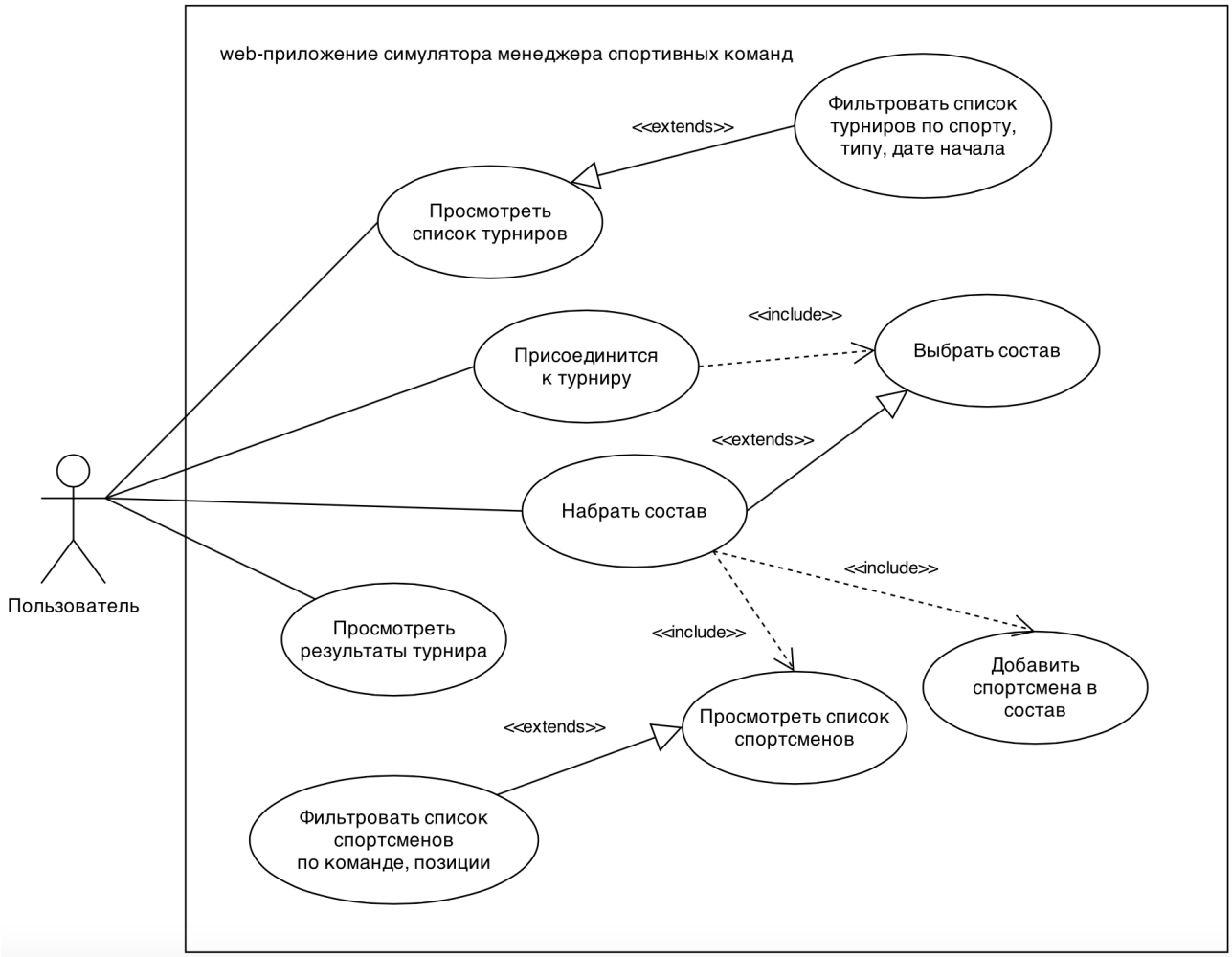

Рисунок 2.2 — Диаграмма вариантов использования для пользователя

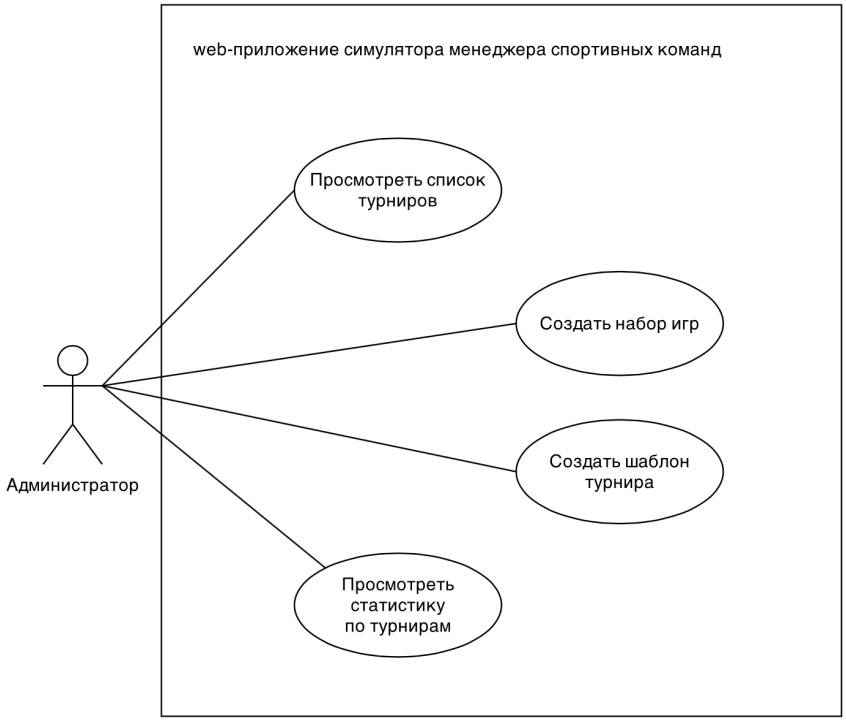

Рисунок 2.3 — Диаграмма вариантов использования для администратора

2.4 Проектирование общей архитектуры серверного кластера приложения

Исходя из анализа предметной области, а также требований к проекту и возможных вариантов использования, с учетом достоинств и недостатков различных подходов, изложенных в первом разделе, были приняты следующие основные архитектурные решения:

- Исходя из требований к масштабируемости, а также необходимости поддержки различных клиентских приложений (как одностраничных веб-приложений, так и мобильных приложений), серверная часть приложения будет реализована в виде сервиса, предоставляющего JSON RESTful API для клиентских приложений;
- Часть приложения («Игровой движок»), взаимодействующая со сторонним сервисом, предоставляющим различную информацию о ходе матчей в реальном времени, и реализующая основную бизнес-логику (слежение за обновление данных, расчет очков, заработанных спортсменом, расчет зарплат спортсменов, определение победителей турнира и т.д.) будет реализована в виде компоненты, функционирующей независимо от веб-приложения, реализующего RESTful API.
- Для обеспечения обновлений в режиме реального времени на стороне клиента будет использоваться протокол WebSocket.
- В качестве средства, обеспечивающего взаимодействие в режиме реального времени между «Игровым движком» и веб-частью приложения, будет использоваться очередь сообщений.
- Для обеспечения возможности масштабирования и необходимой отказоустойчивости будет использоваться кластер серверов баз данных, функционирующий на основе репликации по схеме «ведущий-ведомый».

Данные архитектурные решения должны обеспечить отказоустойчивость приложения за счет введения избыточности и, тем самым, отсутствия единой точки отказа, а также распределение нагрузки между различными компонентами системы и возможность масштабирования каждого компонента

независимо от других. Общая схема серверного кластера приложения показана на рисунке 2.4.

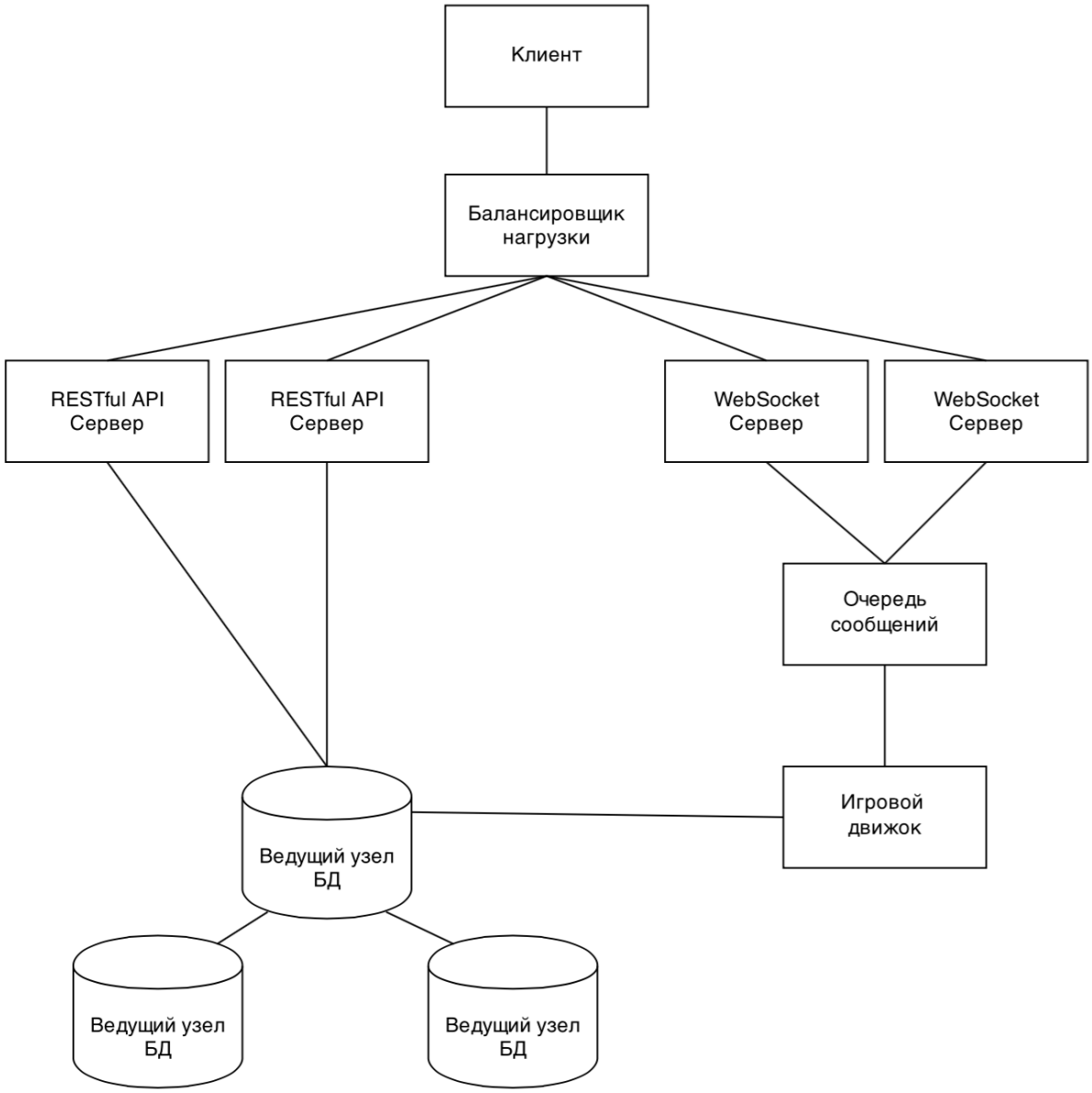

Рисунок 2.4 — Общая архитектура серверного кластера

2.5 Проектирование игрового движка

Игровой движок — часть разрабатываемого симулятора менеджера спортивных турниров, реализующая основную бизнес-логику, которая не зависит от пользователей. Игровой движок должен обеспечивать следующие основные функции:

- Обеспечивать периодический импорт, обработку и хранение полученных со стороннего сервиса данных о составах команд и расписания реальных игр для каждого вида спорта.
- Обеспечивать импорт статистических данных о проходящих в текущее время играх в режиме реального времени. Данный процесс должен запускаться за несколько минут до начала игры и продолжаться до ее окончания.
- Рассчитывать на основании этих данных зачетные баллы, набранные каждым спортсменом, обновлять эти данные в БД и посылать обновления на WebSocket-сервер.
- Определять начало и конец турнира, победителя турнира.
- Автоматически создавать по мере необходимости турниры на основании шаблонов и наборов игр, определенных администратором.

Для обеспечения масштабирования и отказоустойчивости целесообразно разделить игровой движок на несколько частей (рабочих процессов), функционирующих в качестве независимых приложений расположенных на отдельных серверах и взаимодействующих посредством очереди сообщений:

- Основной рабочий процесс выполняет функции: импорта данных о составах команд и расписаниях игр для всех видов спорта; создания турниров; а также генерирования команд начала импорта статистических данных рабочим процессам в соответствии с расписаниями игр для всех видов спорта.
- Для каждого вида спорта создается отдельный рабочий процесс, реализующий функциональность периодического обновления статистических данных для текущих игр текущего вида спорта. В задачи процесса также входит расчет зачетных баллов в соответствии со спецификой данного спорта.

На рисунке 2.5 представлена диаграмма последовательности, отражающая взаимодействие основного рабочего процесса с побочными рабочими процессами.

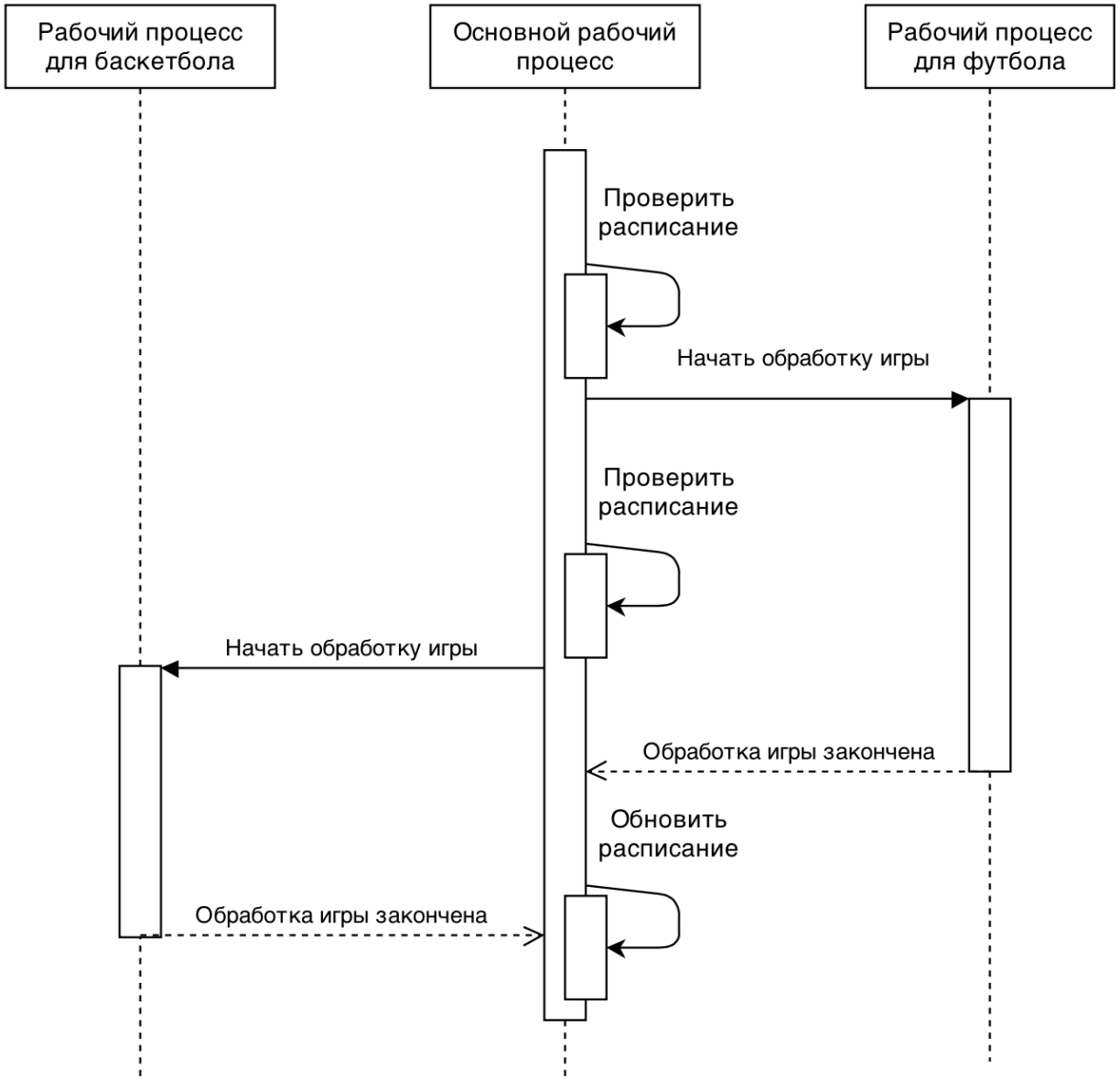

Рисунок 2.5 — Диаграмма последовательности взаимодействия рабочих процессов

Обеспечение обновления данных в режиме реального времени осуществляется в несколько этапов: рабочий процесс, получив данные от сервиса и выполнив их обработку, отправляет сообщение с обновлением данных посредством очереди сообщений WebSocket-серверу. Далее WebSocket-сервер отправляет сообщение клиентскому приложению. Таким образом WebSocket-сервер фактически играет роль шлюза между очередью сообщений и

клиентским WebSocket-приложением. На рисунке 2.6 представлена диаграмма последовательности, показывающая передачу сообщений от рабочего процесса к клиентскому приложению.

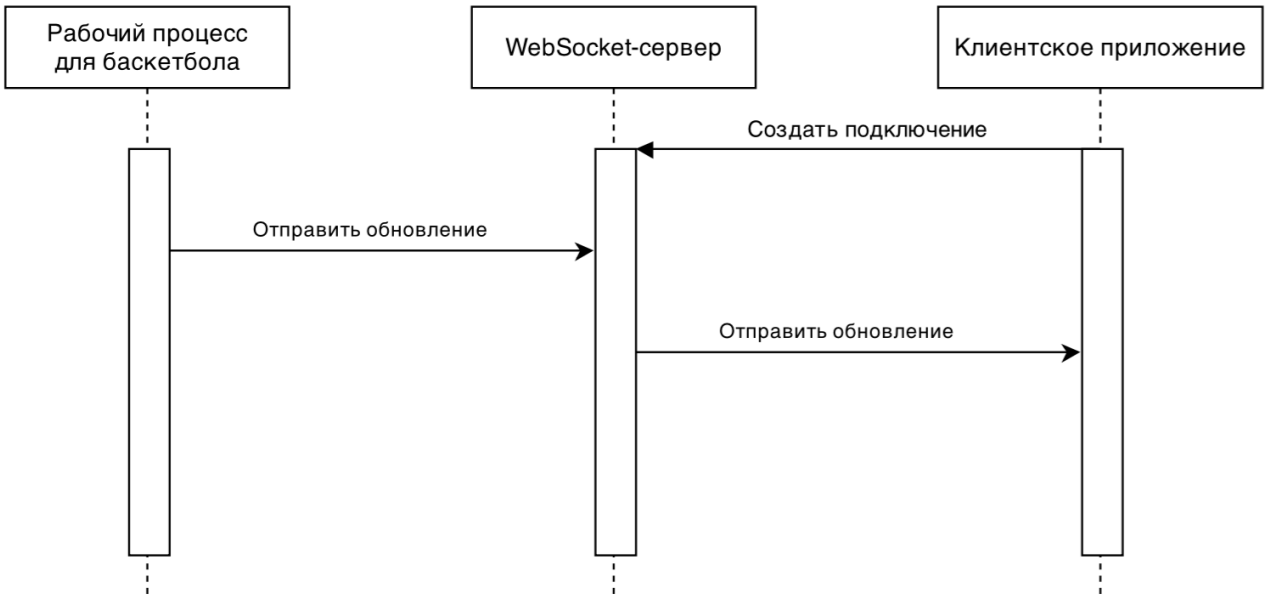

Рисунок 2.6 — Диаграмма последовательности передачи обновлений

2.6 Проектирование алгоритма рабочего процесса для игры

Алгоритм работы рабочего процесса, выполняющего разбор и обновление данных об игре, проходящей в реальности в текущий момент, целесообразно разделить на два: алгоритм, реализующий общую логику для всех видов спорта; алгоритм, реализующий логику, уникальную для каждого вида спорта.

Общими для всех видов спорта являются следующие действия:

- обновление статуса игры в турнире в начале и конце игры;
- выполнение периодических запросов данных со стороннего сервиса;
- обновление зачетных очков игроков в составах пользователей;
- определение победителя турнира.

На рисунке 2.7 представлена блок-схема алгоритма, общего для всех видов спорта.

Алгоритм начинает работу после того, как рабочий процесс получит сообщение от основного рабочего процесса, содержащее идентификатор игры, для которой необходимо начать разбор данных. Далее алгоритм устанавливает статус данной игры в «активна» во всех турнирах, которые включают в себя данную игру. Затем каждых 10 секунд алгоритм запрашивает данные с сервиса и определяет, есть ли в них изменения. Если изменения есть — данные направляются в анализатор. Процесс повторяется до конца игры. После окончания игры рабочий процесс устанавливает для данной игры статус «окончена» во всех турнирах, и, если у какого-либо турнира все игры находятся в статусе «окончена», определяет победителя (победителей) данного турнира.

Предоставляемые сторонним сервисом данные представляют собой файл формата XML, содержащий последовательность игровых событий, а также список спортсменов, участвовавших в этих событиях, их действия и ряд статистических параметров, характеризующих эти действия. На основе данных параметров осуществляется расчет зачетных очков спортсменов по набору правил, характерных для данного спорта. Схема XML файла уникальна для каждого вида спорта.

Рисунок 2.7 — Схема алгоритма рабочего процесса

3. РАЗРАБОТКА СИМУЛЯТОРА МЕНЕДЖЕРА СПОРТИВНЫХ КОМАНД

3.1 Выбор инструментария и технологии разработки

Выбор технологий является важным предварительным этапом разработки сложных информационных систем. Платформа и язык программирования, на котором будет реализована система, заслуживает большого внимания, так как исследования показали, что выбор языка программирования влияет на производительность труда программистов и качество создаваемого ими кода[8].

Перечислим ключевые факторы, повлиявшие на выбор стека технологий:

- приложение должно работать в облаке Amazon;
- приложение должно быть основано на открытых технологиях и программном обеспечении с открытым исходным кодом;
- приложение должно быть масштабируемым и отказоустойчивым;
- обновление данных на стороне сервера должны отображаться на стороне клиента в режиме реального времени;
- специфика предметной области (событийно-ориентированность, отсутствие фиксированной схемы данных).

В качестве языка программирования был выбран язык Python, как простой в поддержке, хорошо приспособленный для web-разработки, знакомый имеющимся разработчикам и удовлетворяющий всем требованиям язык.

Для отображения изменений данных в реальном времени выбрана технология WebSocket. Использование протокола WebSocket означает, что сервер должен в течении длительного времени поддерживать большое количество малоактивных соединений. Традиционные web-сервера и сервера приложений, например, Apache, используют prefork-модель, когда на каждое соединение создается или выделяется из пула процесс-обработчик, который занимает системные ресурсы даже если соединение фактически не активно. При использовании prefork-модели потребление ресурсов системы растет

линейно в соответствии с количеством соединений, независимо от того, передаются по ним данные, или нет. Таким образом, это может привести к тому, что все ресурсы системы будут исчерпаны, хотя фактически, никакой полезной работы происходить не будет. Данная проблема решается использованием асинхронных web-серверов, которые обрабатывают все запросы в одном рабочем процессе, выполняющем цикл обработки событий от дескрипторов ввода-вывода операционной системы.

С учетом вышесказанного, а также используемого языка программирования, был выбран web-сервер/web-фреймворк Tornado, созданный для эффективной работы с push-технологиями и WebSocket.

Предметная область плохо укладывается в традиционную реляционную модель данных, т.к. сущности предметной области не имеют четкой, фиксированной схемы. Использование шаблона проектирования «сущность-атрибут-значение» сильно усложняет схему данных и построение запросов к ним. Второй проблемой реляционных баз данных является сложность горизонтального масштабирования. Таким образом, целесообразно использовать документо-ориентированную базу данных с нефиксированной схемой данных. В качестве такой базы была выбрана MongoDB.

В качестве связующего звена всех рабочих процессов системы была выбрана платформа обмена сообщениями RabbitMQ.

3.1.1 Облачная платформа Amazon Web Services

Amazon Web Services (AWS) [9] — облачная платформа от компании Amazon, занимающая лидирующее положение на рынке облачных сервисов. AWS предлагает полный набор функций и сервисов, необходимых для запуска в облаке приложения любого уровня сложности.

AWS расположен в 8 географических регионах: Восток США, Запад США, северная Калифорния, Бразилия, Европа, Южная Азия, Восточная Азия и Австралия. Ниже приведён обзор сервисов AWS, используемых в данном проекте.

3.1.2 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (Amazon EC2) [10] — облачный web-сервис, предоставляющий виртуальные сервера (Amazon Instance), хранилище данных (Elastic Block Storage), а также балансировщик нагрузки (Elastic Load Balancer). EC2 позволяет запускать заранее сконфигурированные сервера с предустановленными ОС: Red Hat Enterprise Linux, Ubuntu Server, Windows 2008 и другими. Существует возможность создавать свои образы (AMI - Amazon Machine Image) на основе любого дистрибутива Linux а также делать мгновенные слепки (snapshot) с работающих серверов, для последующего разворачивания или использования в качестве резервной копии.

Elastic Load Balancer предоставляет услуги балансировки и автоматического масштабирования. ELB позволяет создавать правила, при которых станет возможно автоматическое увеличение количества используемых серверов, например, если один или несколько серверов не справляются с нагрузкой.

Elastic Block Storage позволяет добавлять к любому виртуальному серверу практически неограниченное количество диском с любым объемом хранения. Диски, созданные по этой технологии, независимы от виртуальных серверов и расположены на специальных серверах для хранения данных.

Управление сервисами осуществляется через web-интерфейс.

3.1.3 Amazon Simple Storage Service

Amazon Simple Storage Service (Amazon S3) [11] — облачный web-сервис, предоставляющий услуги хранения и получения любого объема данных в любое время из любой точки сети. Максимальный поддерживаемый размер одного файла - 5 ТБ. Файлы хранятся в отдельных бакетах (bucket), в которых можно создавать директории и поддиректории. Название бакетов должно быть уникально в рамках всего сервиса. К бакетам можно применять различные политики безопасности: делать их публичными, приватными, разделять права

доступа для различных пользователей и т. д. Манипуляции с бакетами и файлами осуществляются посредством REST или SOAP.

Amazon Simple Storage Service зачастую используется для хранения статического содержимого web-приложений.

3.1.4 Amazon CloudFront

Amazon CloudFront [12] — web-сервис, предоставляющий услуги доставки содержимого (Content Delivery Network, CDN). CloudFront интегрируется с Amazon S3 и позволяет существенно повысить скорость доступа к данным и их доступность за счет репликации данных на множество географически распределенных серверов, и осуществления загрузки данных из максимально близкого к конечному пользователю сервера.

3.1.5 Язык программирования Python

Python [13] — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости исходного кода.

Разработка языка была начата в конце 1980-х годов сотрудником голландского института CWI Гвидо ван Россумом. Python разрабатывался как скриптовый язык для распределённой ОС Amoeba, на основе наработок языка ABC. Первая версия исходного текста интерпретатора была опубликована в 1991г.[14]. С самого начала Python проектировался как объектно-ориентированный язык.

Развитие языка происходит согласно четко регламентированному процессу создания, обсуждения, отбора и реализации документов PEP

(Python Enhancement Proposal) — предложений по развитию Python.

В 2008 г. вышла первая версия Python 3.0, в которой были устранены многие недостатки архитектуры с максимально возможным сохранением совместимости со старыми версиями Python. На сегодняшний день поддерживаются обе ветви развития (Python 2.x и Python 3.x).

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-оринтированное, функциональное, императивное и аспектно-ориентированное. Основные архитектурные черты языка — динамическая строгая утиная типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений, удобные высокоуровневые структуры данных.

Синтаксис языка минималистичен. Интересной особенностью синтаксиса является выделение блоков кода с помощью отступов (пробелов или табуляций), поэтому в Python отсутствуют операторные скобки begin/end, как в языке Pascal, или фигурные скобки, как в языке C. Данная особенность синтаксиса позволяет сократить количество строк и символов в программе и повышает читаемость исходного кода.

Одной из интересных возможностей языка являются генераторы — функции, сохраняющие внутреннее состояние: значения локальных переменных и текущую инструкцию. Наличие генераторов позволяет писать асинхронные программы в синхронном стиле, без использования функций обратного вызова.

Код в Python организовывается в функции и классы, которые могут объединяться в модули. Модули в свою очередь могут объединяться в пакеты.

Эталонной реализацией языка Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Интерпретатор распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях.

Помимо CPython, существует большое количество интерпретаторов языка: PyPy — интерпретатор Python, написанный на Python и поддерживающий JIT-компиляцию; Jython — компилятор Python в байткод виртуальной машины Java; IronPython — компилятор Python в байткод виртуальной машины CLR платформы Microsoft .NET и др.

Python — стабильный и распространенный язык. Он используется во многих проектах и в различных качествах: как основной язык программирования или для создания расширений и интеграции приложений. На

Python реализовано большое количество проектов, также он активно используется для создания прототипов будущих программ. Python используется во многих крупных компаниях.

Python с пакетами NumPy, SciPy и MatPlotLib активно используется как универсальная среда для научных расчетов в качестве замены распространенным специализированным коммерческим пакетам Matlab, IDL и др.

3.1.6 База данных MongoDB

MongoDB [15] — документо-ориетированная система управления базами данных (СУБД) с открытым исходным кодом. Написана на языке C++.

При разработке авторы исходили из необходимости специализации баз данных, благодаря чему им удалось отойти от принципа «один размер подо все». За счет минимизации семантики для работы с транзакциями появляется возможность решения целого ряда проблем, связанных с недостатком производительности, причем горизонтальное масштабирование, по мнению авторов, становится проще. Используемая модель документов хранения данных (JSON/BSON) проще кодируется, проще управляется (в том числе за счет применения так называемого «бессхемного стиля» (англ. schemaless style), а внутренняя группировка релевантных данных обеспечивает дополнительный выигрыш в быстродействии.

MongoDB, по мнению разработчиков, должна заполнить разрыв между простейшими NoSQL-СУБД, хранящими данные в виде «ключ — значение» (простыми и легко масштабируемыми, но обладающими минимальными функциональными возможностями) и большими реляционными СУБД (со структурными схемами и мощными запросами).

Основные возможности MongoDB:

- документо-ориентированное хранение (JSON-подобная схема данных);
- достаточно гибкий язык для формирования запросов;
- динамические запросы;

- поддержка индексов;
- журналирование;
- поддержка отказоустойчивости и масштабируемости.
- MapReduce;
- полнотекстовый поиск с учетом морфологии.

СУБД управляет наборами JSON-подобных документов, хранимых в двоичном виде в формате BSON. Хранение и поиск файлов в MongoDB происходит благодаря вызовам протокола GridFS. Подобно другим документо-ориентированным СУБД (CouchDB и др.), MongoDB не является реляционной СУБД. Среди других отличий от традиционных реляционных СУБД:

- Отсутствует оператор «join». Обычно данные могут быть организованы более денормализованным способом, но на разработчиков ложится дополнительная нагрузка по обеспечению непротиворечивости данных.
- Нет такого понятия, как «транзакция». Атомарность гарантируется только на уровне целого документа, то есть частичного обновления документа произойти не может.
- Отсутствует понятие «изоляции». Любые данные, которые считываются одним клиентом, могут параллельно изменяться другим клиентом.

MongoDB поддерживает горизонтальное масштабирование, благодаря наличию механизмов репликации данных между узлами в конфигурации «ведущий — ведомый», и шардинга — разделения одной или нескольких коллекций данных между узлами.

3.1.7 Веб-сервер Nginx

Nginx (англ. engine x) [16] — веб-сервер с открытым исходным кодом, работающий на Unix-подобных операционных системах.

Основные возможности Nginx:

- обслуживание статических запросов, индексных файлов, автоматическое создание списка файлов, кеш дескрипторов открытых файлов;
- ускоренное проксирование без кеширования, простое распределение нагрузки и отказоустойчивость;
- поддержка кэширования при ускоренном проксировании и FastCGI;
- ускоренная поддержка FastCGI и memcached серверов, простое распределение нагрузки и отказоустойчивость;
- модульность, фильтры, в том числе сжатие (gzip), byte-ranges (докачка), chunked ответы, HTTP-аутентификация, SSI-фильтр;
- несколько подзапросов на одной странице, обрабатываемые в SSI-фильтре через прокси или FastCGI, выполняются параллельно;
- поддержка SSL;
- поддержка PSGI, WSGI.

В Nginx рабочие процессы обслуживают одновременно множество соединений, мультиплексируя их вызовами операционной системы select, epoll (Linux) и kqueue (FreeBSD). Рабочие процессы выполняют цикл обработки событий от дескрипторов соединений. Полученные от клиента данные разбираются с помощью конечного автомата. Разобранный запрос последовательно обрабатывается цепочкой модулей, задаваемой конфигурацией. Ответ клиенту формируется в буферах, которые хранят данные либо в памяти, либо указывают на отрезок файла. Буферы объединяются в цепочки, определяющие последовательность, в которой данные будут переданы клиенту. Если операционная система поддерживает эффективные операции ввода-вывода, такие как writev и sendfile, то Nginx применяет их по возможности.

Для эффективного управления памятью Nginx использует пулы. Пул — это последовательность предварительно выделенных блоков динамической памяти. Длина блока варьируется от 1 до 16 килобайт. Изначально под пул выделяется только один блок. Блок разделяется на занятую область и незанятую. Выделение мелких объектов выполняется путем продвижения указателя на незанятую область с учетом выравнивания. Если незанятой

области во всех блоках не хватает для выделения нового объекта, то выделяется новый блок. Если размер выделяемого объекта превышает длину блока, то он полностью выделяется из кучи.

Nginx широко применяется для быстрой отдачи статического содержимого и в качестве реверс-прокси/балансировщика нагрузки перед серверами приложений.

По данным Netcraft на май 2014 года, число сайтов, обслуживаемых Nginx, превышает 146 миллионов, что делает его третим по популярности веб-сервером в мире. При этом, процент активных сайтов, использующих Nginx, составляет 14,4% от общего количества активных сайтов, что делает Nginx вторым в мире по популярности веб-сервером среди активных сайтов, уступая лишь web-серверу Apache.

По данным W3Techs, Nginx наиболее часто используется на высоконагруженных сайтах, занимая первое место по частоте использования среди 1000 самых посещаемых сайтов в мире — больше трети таких сайтов работает на Nginx.

3.1.8 Веб-сервер Tornado

Tornado [17] — масштабируемый, неблокирующий веб-сервер и веб-фреймворк с открытым исходным кодом. Написан на языке Python.

Tornado был разработан для использования в сервисе FriendFeed. После того, как компания была приобретена Facebook в 2009 г., исходные коды сервера были открыты. Tornado был создан для обработки нескольких тысяч одновременных соединений (так называемая проблема c10k, или проблема 10000 соединений).

Данный веб-сервер идеально подходит для веб-сервисов реального времени, в которых каждый пользователь поддерживает активное соединение с серверами в течении длительного времени.

3.1.9. Платформа RabbitMQ

RabbitMQ [18] — платформа с открытым исходным кодом, реализующая систему обмена сообщениями между компонентами программной системы на основе стандарта AMQP (Advanced Message Queuing Protocol). RabbitMQ создан на основе Open Telecom Platform, обеспечивающей высокую надежность и производительность промышленного уровня. Написан на языке Erlang. В качестве хранилища сообщений используется распределенная СУБД реального времени Mnesia.

RabbitMQ состоит из:

- сервера RabbitMQ;
- поддержки протоколов HTTP, XMPP и STOMP;
- клиентских библиотек AMQP для различных языков программирования, в тот числе и Python;
- расширений, реализующих различную дополнительную функциональность (мониторинг, управление через HTTP или веб-интерфейс, передача сообщений между брокерами).

В основе RabbitMQ лежит протокол AMQP, который вводит три основных понятия:

- Сообщение (message) — единица информации, которая передается от отправителя к получателю(ям); состоит из набора заголовков и содержания, которое брокером никак не интерпретируются.
- Точка обмена (exchange) — распределяет отправленные сообщения между одной или несколькими очередями в соответствии с их заголовками.
- Очередь (queue) — место, где хранятся сообщения до тех пор, пока их не заберет получатель.

Начиная с версии 3.2.0 RabbitMQ поддерживает горизонтальное масштабирование для построения кластерной архитектуры.

3.1.10 Очередь заданий Celery

Celery — ПО с открытым исходным кодом, реализованное на языке программирования Python, предоставляющее асинхронную очередь заданий,

основанную на распределенном обмене сообщениями. Задания (представляющие собой по сути функции языка Python) выполняются конкурентно на одном или более рабочем узле. Рабочие узлы могут быть распределены по различным серверам.

Задания могут выполняться как синхронно, так и асинхронно, в реальном времени или по расписанию.

Рекомендуемой платформой для работы Celery является брокер RabbitMQ, однако поддерживаются также Redis, MongoDB, Amazon SQS, IronMQ и реляционные базы данных.

3.2 Разработка схемы базы данных

Проектирование документо-ориентрованной схемы данных достаточно существенно отличается от проектирования реляционной схемы данных. При проектировании реляционных баз данных, как правило, стремятся обеспечить нормализацию данных, т.е. уменьшить избыточность за счет декомпозиции отношений таким образом, чтобы в каждом отношении хранились только первичные факты (факты, не выводимые из других отношений). При проектировании документо-ориентированной схемы данных денормализация и избыточность являются вполне штатным механизмом, заменяющим сложные запросы с объединениями и транзакциями в реляционных базах данных[19]. Зачастую, имеет смысл продублировать некоторые данные из одного документа во втором документе для того, чтобы уменьшить количество запросов, необходимых для их получения.

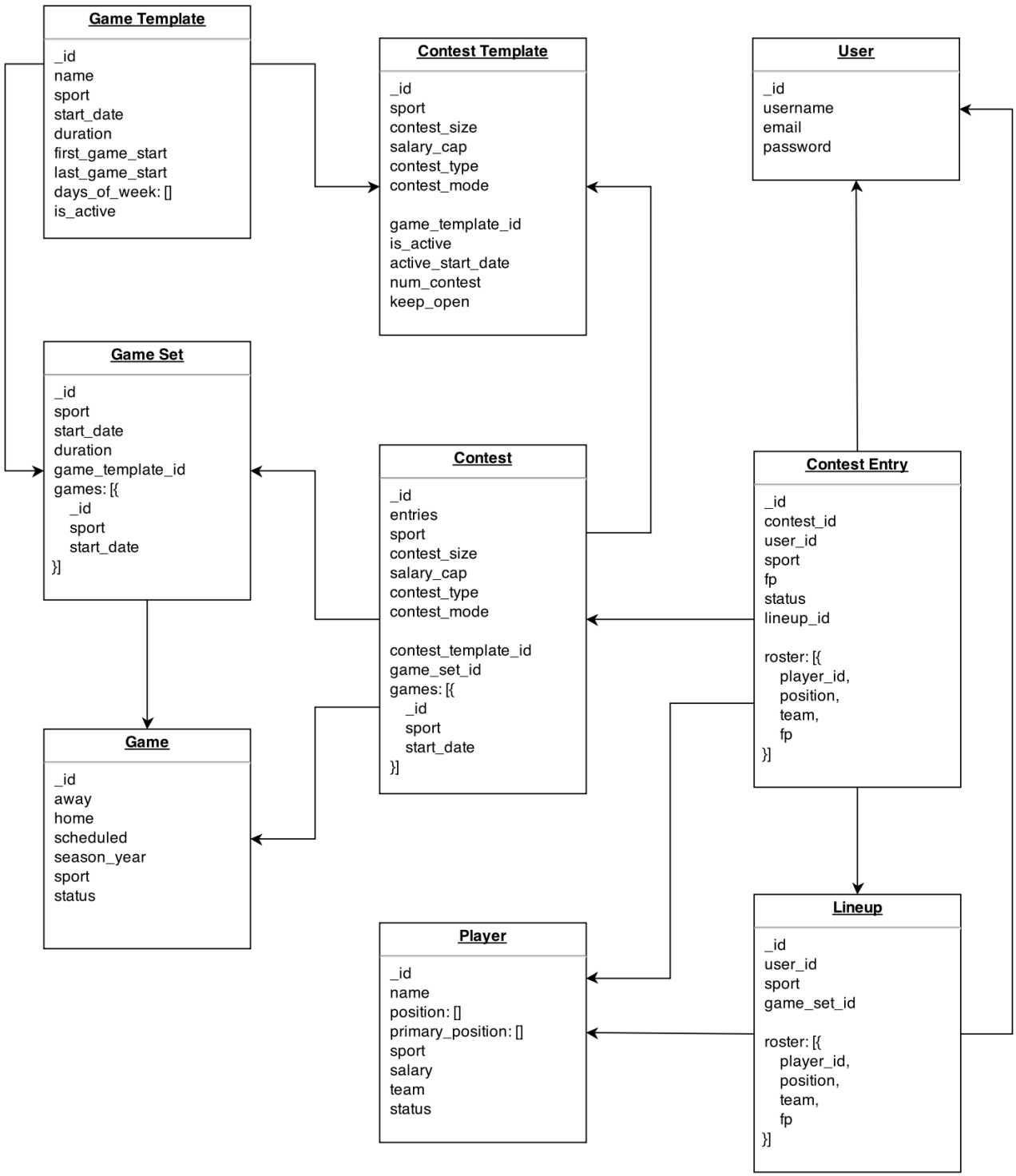

Рисунок 3.1 — Схема базы данных

На рисунке 3.1 представлена разработанная схема данных. Как видно из схемы, широко использовалось дублирование одного документа в качестве вложенного документа в другом. Например, документ Contest, представляющий собой турнир, содержит в себе список вложенных документов, полностью дублирующих документы Game.

Наличие практически во всех документах поля Sport потенциально позволяют использовать его в качестве ключа для партицирования данных, если объем данных или нагрузки станут чрезвычайно велики, а также увеличить масштабирование в будущем.

3.3 Разработка RESTful JSON API

Разработка RESTful JSON API симулятора менеджера спортивных команд не представляет особой сложности. По сути, задача данной части приложения - предоставить интерфейс для создания, чтения и изменения сущностей в базе данных, а также обеспечить проверку корректности пользовательских данных и прав доступа.

С этой целью было реализовано несколько базовых классов, реализующих основную общую функциональность. На рисунке 3.2 представлена диаграмма базовых классов и иерархия их наследования.

Основной класс BaseHandler является обработчиком HTTP-запроса, наследованным от класса RequestHandler, предоставляемого фреймворком tornado и класса-примеси AuthMixin, реализующего функциональность, связанную с авторизацией пользователей. Данный класс содержит методы, реализующие базовую функциональность по формированию JSON-ответа, установки соответствующих HTTP-заголовков и т.д.

Класс-примесь SchematicsWrapperMixin реализует функциональность, связанную с проверкой корректности пользовательских данных и соответствия их схеме данных.

Класс BaseListHandler является базовым обработчиком запросов на получение списка сущностей. Класс FilteredListHandler является расширением класса BaseListHandler, добавляя поддержку фильтрации списка сущностей по различным значениям соответствующих атрибутов сущности (например, спорту).

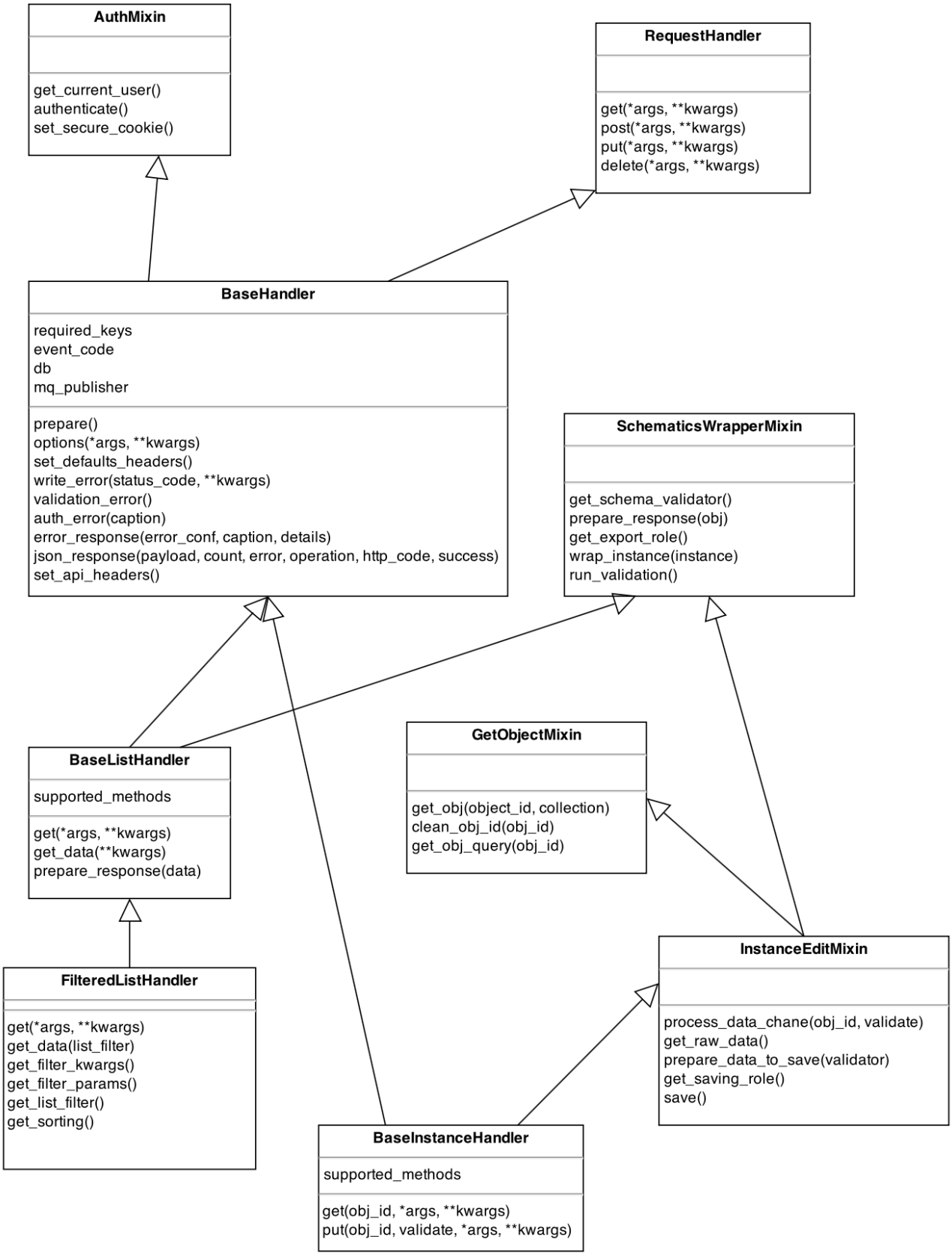

Рисунок 3.2 — Диаграмма базовых классов RESTful API

Класс BaseInstanceHandler является базовым обработчиком запросов для работы с одной сущностью и реализует методы создания, получения и модификации сущности.

Классы-обработчики для каждой конкретной сущности создаются путем наследования от данных базовых классов и классов-примисей и незначительной модификацией их методов.

В таблице 3.1 представлены все реализованные ресурсы, поддерживаемые ими методы и приведено краткое описание каждого из них.

Таблица 3.1 — Реализованное RESTful API приложения

URI Ресурса	HTTP методы	Описание
/auth/login	POST	Аутентифицировать пользователя
/auth/logout	GET	Прервать сессию текущего пользователя
/teams	GET	Получить список команд
/players	GET	Получить список спортсменов
/player/:player-id	GET	Получить спортсмена по его идентификатору
/schedule	GET	Получить расписание игр
/user/account	GET	Получить информацию о текущем пользователе
/user/account	POST	Обновить информацию текущего пользователя
/user	GET	Получить текущего пользователя
/user	POST	Создать пользователя
/user	PUT	Обновить пользователя
/users	GET	Получить список пользователя
/contest-entry	POST	Присоединиться к турниру
/contest-entries/my	GET	Получить список турниров, в которых участвует текущий пользователь
/contest/:contest-id	GET	Получить турнир по его идентификатору
/contest	POST	Создать турнир
/contests	GET	Получить список турниров
/lineups	GET	Получить список составов текущего пользователя
/lineup/:lineup-id	GET	Получить состав текущего пользователя по его идентификатору
/lineup/:lineup-id	PUT	Изменить состав текущего пользователя
/lineup	POST	Создать состав

3.4 Разработка системы обновления в реальном времени

Обновления сообщений происходит следующим образом. Рабочий процесс передает сообщение посредством протокола AMQP в очередь сообщений, далее это сообщение получает обработчик WebSocket-соединений и передает его клиентскому приложению.

Протокол AMQP основан на трех базовых понятиях:

- Сообщение (message) — единица передаваемых данных, основная его часть (содержание) никак не интерпретируется сервером. Сообщения могут иметь ключи маршрутизации (routing key).
- Точка обмена (exchange) — в неё отправляются сообщения. Точка обмена распределяет сообщения в одну или несколько очередей. При этом в точке обмена сообщения не хранятся. Точки обмена бывают трёх типов: fanout — сообщение передаётся во все прикрепленные к ней очереди; direct — сообщение передаётся в очередь с именем, совпадающим с ключом маршрутизации (routing key) (ключ маршрутизации указывается при отправке сообщения); topic — нечто среднее между fanout и direct, сообщение передаётся в очереди, для которых совпадает маска на ключ маршрутизации, например, app.notification.sms.* — в очередь будут доставлены все сообщения, отправленные с ключами, начинающимися на app.notification.sms.
- Очередь (queue) — здесь хранятся сообщения до тех пор, пока не будет получены клиентом. Клиент всегда получает сообщения из одной или нескольких очередей.

Совокупность данных понятий позволяет создавать топологию произвольного уровня сложности. Как правило, очереди сообщений используются для реализации двух шаблонов проектирования: «Очередь заданий» и «Отправитель-Подписчик».

Шаблон «Очередь заданий» используется для распределения заданий по получению данных об играх между несколькими рабочими процессами. В роли отправителя сообщений выступает основной рабочий процесс, производящий проверку расписания игр. В случае, если необходимо запустить процесс

получения данных об игре, он отправляет сообщение с идентификатором игры и ключем маршрутизации, в роли которого выступает название спорта. Точка обмена «celery» имеет тип direct и направляет это сообщение в соответствующую очередь. На рисунке 3.3 представлена схема передачи сообщений для рабочих процессов, реализованная по данному шаблону.

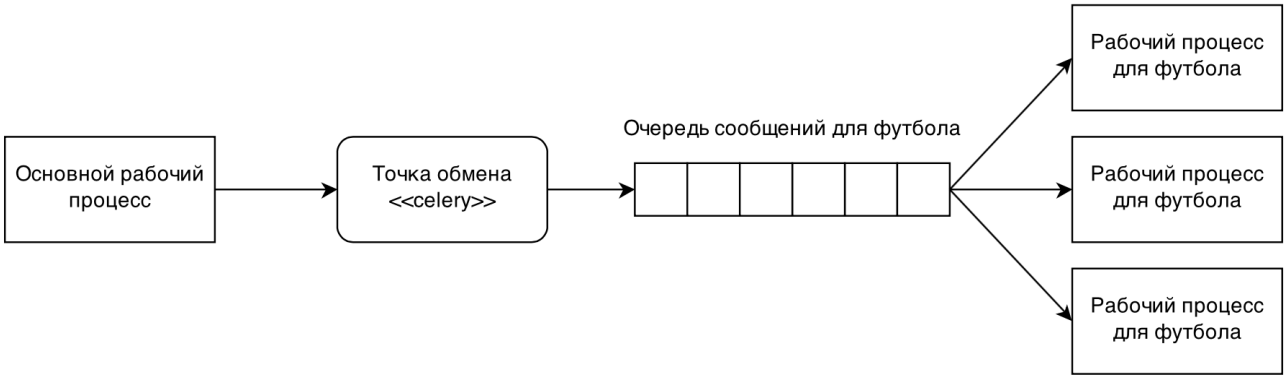

Рисунок 3.3 — Схема передачи сообщения для рабочих процессов

Шаблон «Отправитель-Подписчик» используется для обмена сообщениями между рабочими процессами и обработчиками WebSocket соединений. Сообщениями являются сериализованные документы базы данных, а их идентификаторы выступают в роли ключей маршрутизации. Рабочий процесс, выполняющий разбор игры, отправляет сообщение в точку обмена «tornado». Данная точка обмена имеет типа topic и направляет сообщение во все очереди, для которых подходит ключ маршрутизации.

На рисунке 3.4 представлена схема передачи сообщений для обработчиков WebSocket-соединений, реализованная по данному шаблону.

Как упоминалось в разделе 2, обработчик WebSocket-соединений представляет по сути шлюз между очередью сообщений и клиентом, обеспечивая доставку сообщений от рабочего процесса клиентскому приложению.

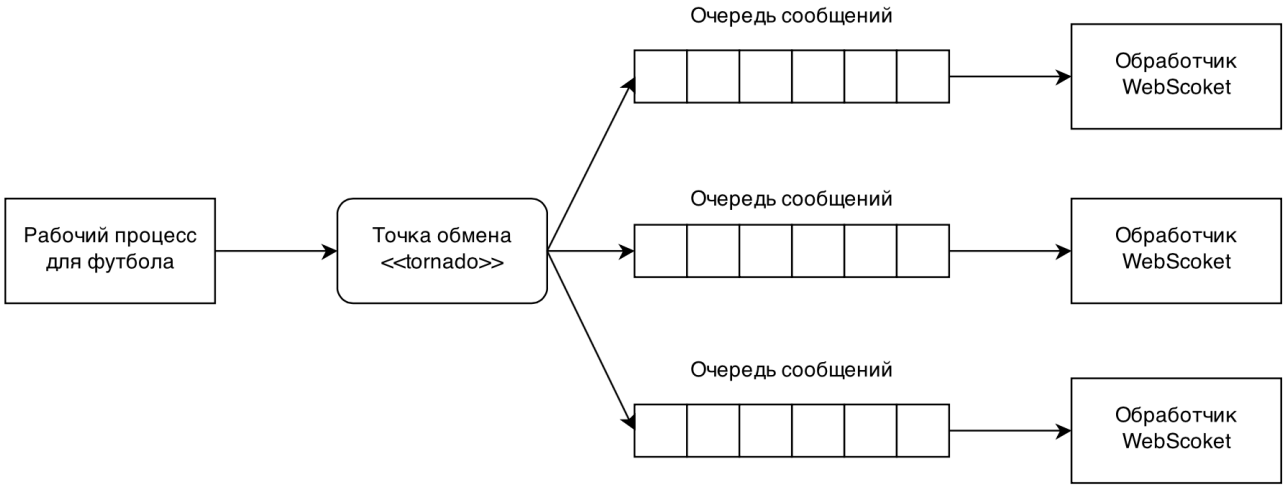

Рисунок 3.4 — Схема передачи сообщения для обработчиков WebSocket-соединений

На рисунке 3.5 представлена диаграмма классов для реализованных базовых классов обработчиков WebSocket-соединений.

Класс BaseMqWebSocketHandler расширяет класс фреймворка tornado WebSocketHandler для работы с очередью сообщений, и передачи сообщений из очереди клиенту. В конкретном обработчике достаточно указать ключ маршрутизации, по которому будет осуществляться выборка сообщений. Реализованные WebSocket-ресурсы, используемые ими ключи маршрутизации и краткое описание приведены в таблице 3.2.

Таблица 3.2 — Описание реализованных WebSocket-ресурсов

URI ресурса	Ключ маршрутизации	Описание
/live/contest/:contest-id/contest-entries	*.:contest-id.*	Обновление всей информации, связанной с конкретным турниром
/live/contests	*.*.*	Получение обновлений всех турниров
/live/contest-entries/my	*.*.:user-id	Получение обновлений о всех турнирах, в которых участвует текущий пользователь

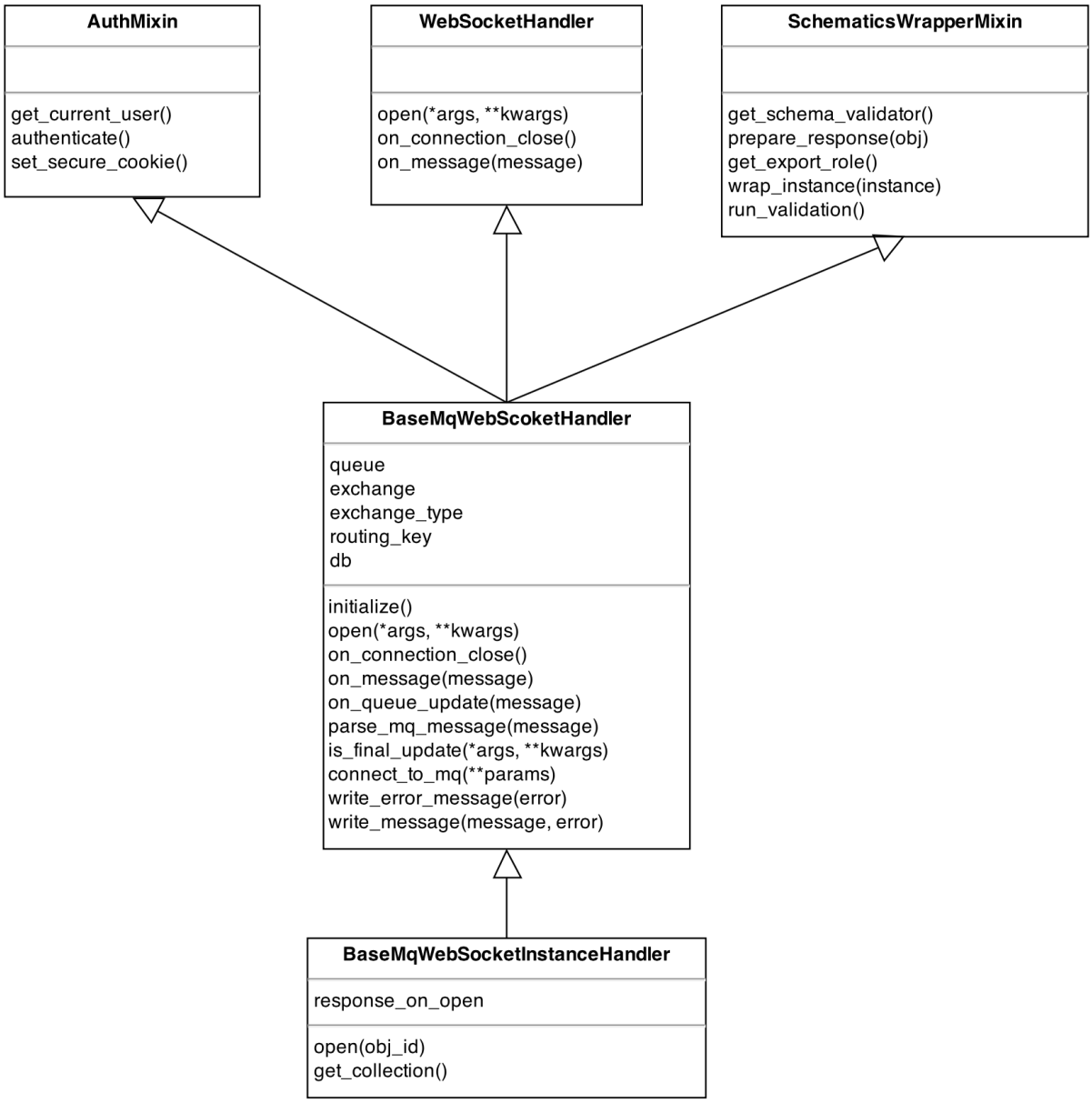

Рисунок 3.5 — Диаграмма классов для базовых обработчиков WebSocket-соединений

3.5 Развертывание кластера приложения

Процесс развертывания серверного кластера приложения состоит из трёх этапов:

- запуск виртуальных серверов в облаке Amazon
- конфигурирование окружения серверов и установка программного обеспечения
- развертывание приложения в готовых окружениях

Запуск виртуальных серверов — наиболее редко выполняющаяся операция.

Типы используемых виртуальных серверов а также их характеристики в зависимости от исполняемой роли представлены в таблице 3.3.

Таблица 3.3 — Характеристики используемых виртуальных серверов

Роль сервера	Тип виртуального сервера	vCPU	Memory(Gb)	Storage(Gb)
Сервер балансировки нагрузки	c3.4xlarge	8	15	2x80
Сервер приложения	c3.xlarge	4	7.5	2x40
Сервер очереди сообщений	r3.large	2	30.5	1x80
Сервер рабочего процесса	c3.large	2	3.75	2x16
Сервер базы данных	i2.xlarge	4	30.5	1x800 SSD

Этап конфигурирование серверов заключается в установке основного прикладного программного обеспечения, такого как базы данных и веб-сервера, а также изменения их настроек и настроек операционной системы.

Этап разворачивания приложения включает в себя подготовку виртуального окружения языка Python, установку всех зависимостей

приложения, копирование приложения из системы контроля версий, а также минимизацию и загрузку статических файлов приложения в сеть доставки содержимого.

Все этапы разворачивания кластера полностью автоматизированы с помощью Ansible — открытого ПО для удаленного управления конфигурациями. Ansible берет на себя всю работу по приведению удаленных серверов в необходимое состояние. Администратору необходимо лишь описать, как достичь этого состояния с помощью так называемых сценариев.

Преимущества Ansible по сравнению с другими аналогичными решениями (здесь в первую очередь следует назвать такие продукты, как Puppet, Chef и Salt) заключаются в следующем:

- на управляемые узлы не нужно устанавливать никакого дополнительного ПО, всё работает через SSH;
- код программы, написанный на Python, очень прост; при необходимости написание дополнительных модулей не составляет особого труда;
- язык, на котором пишутся сценарии, также предельно прост;
- низкий порог вхождения: обучиться работе с Ansible можно за очень короткое время;
- Ansible работает не только в режиме push, но и pull, как это делают большинство систем управления (Puppet, Chef);

Полное развертывание кластера занимает порядка 40 минут. Диаграмма развертывания представлена на рисунке 3.6.

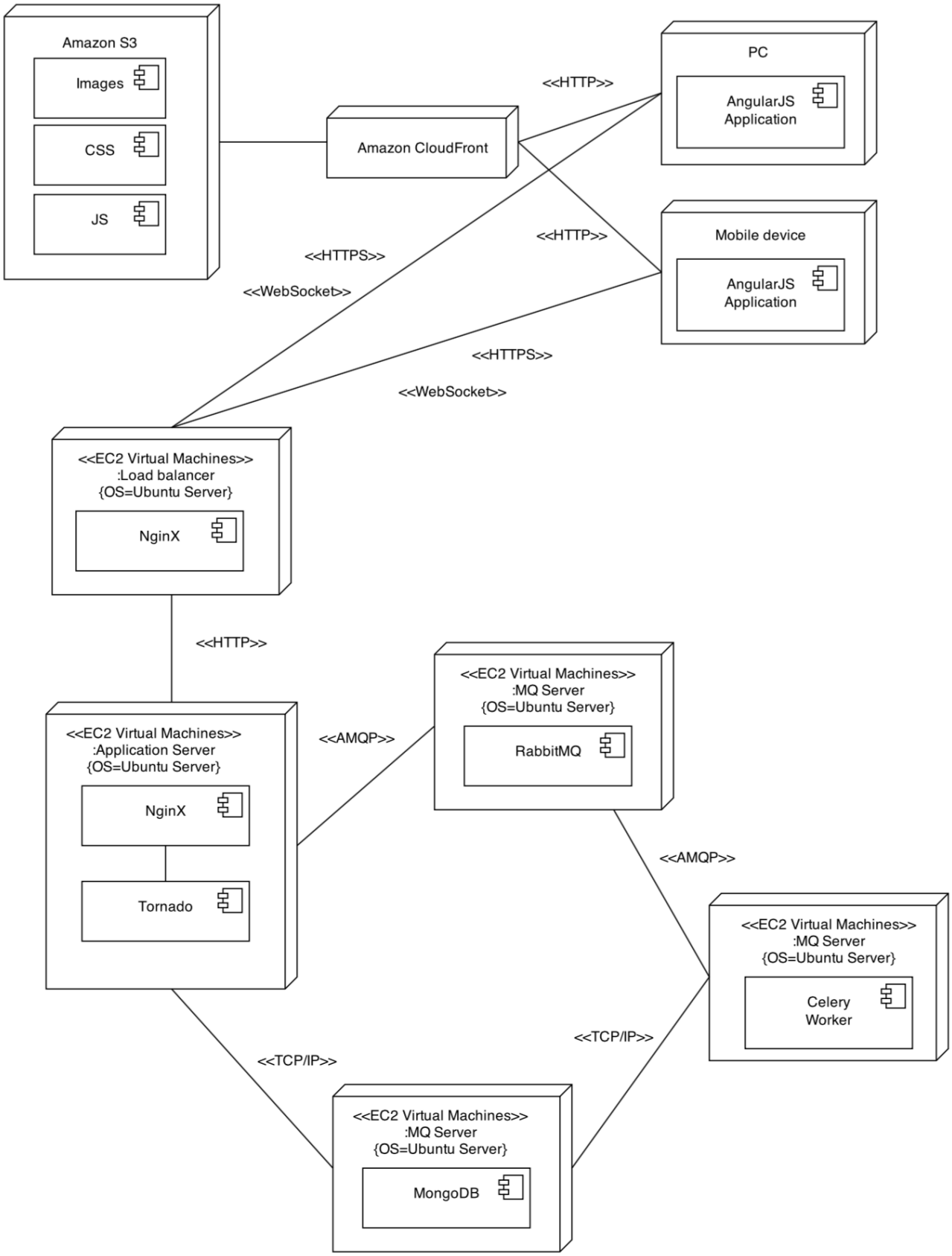

Рисунок 3.6 — Диаграмма развертывания приложения

3.6 Тестирование качества исходного кода

Так как разрабатываемое приложение должно непрерывно эволюционировать и дорабатываться различными разработчиками, одним из важнейших аспектов является требование по обеспечению хорошей читаемости, документированности и корректности исходного кода. С этой целью с самых ранних этапов разработки применялось несколько инструментов для статического анализа корректности кода и соответствия кода принятым нормам стилистического оформления для языка Python:

- pep8.py — это инструмент для проверки соответствия исходного кода принятому в сообществе Python-разработчиков стандарту оформления исходного кода PEP8, регламентирующему правила наименования классов, методов и функций, правило расстановки отступов и пробелов, максимальную допустимую длину строки, порядок следования импортов библиотек и т.д. В случае, если какая либо часть исходного кода не соответствует стандарту, анализатор выдаст предупреждение с указанием имени файла, номера строки и пояснением.
- pyflakes — статический анализатор python-кода, позволяющий выявить распространенные логические ошибки, такие как циклический импорт модулей, неиспользуемые переменные и модули, использование закрытых методов класса, и т.д. Pyflakes также может генерировать предупреждение, если какой-либо модуль, класс или функция не содержат документации. Также возможно определить, какие типы ошибок игнорировать, а какие нет.
- pylint — статический анализатор python-кода, позволяющий определить явные ошибки в коде, такие как использование недекларированой переменной, вызов несуществующего метода класса, синтаксические ошибки и т.д. Также, как и pyflakes, pylint гибко настраиваем.

Данные анализаторы были интегрированы в используемую систему контроля версий исходного кода таким образом, что разработчик не мог внести в нее изменения, если его код содержал ошибки, выявляемые данными программами.

3.7 Нагрузочное тестирование приложения

Непосредственно перед внедрением любой высоконагруженный веб-проект должен пройти нагрузочное тестирование. Нагрузочное тестирование — определение или сбор показателей производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству). В общем случае он означает практику моделирования ожидаемого использования приложения с помощью эмуляции работы нескольких пользователей одновременно.

Нагрузочное тестирование выполнялось с помощью инструмента Apache JMeter — программного обеспечения с открытым исходным кодом, предназначенного для тестирования веб-приложений. Apache JMeter способен производить нагрузочные тесты для JDBC-соединений, FTP, LDAP, SOAP, JMS, POP3, IMAP, HTTP и TCP. В программе реализованы механизмы авторизации виртуальных пользователей, поддерживаются пользовательские сеансы. Организовано логирование результатов теста и разнообразная визуализация результатов в виде диаграмм, таблиц и т. п. Apache JMeter позволяет создавать большое количество запросов с помощью нескольких компьютеров при управлении этим процессом с одного из них.

Сценарий тестирования включает в себя типичную последовательность действий пользователя: авторизация, получение списка турниров, получение списка игроков, создание собственного состава, регистрация этого состава в турнире и т. д.

Тестовый кластер включает в себя следующие серверы:

- 1 сервер балансировки нагрузки;
- 2 сервера веб-приложения;
- 3 сервера базы данных в конфигурации «ведущий-ведомый»;
- 1 сервер очереди сообщений;
- 1 сервер рабочих процессов.

Все сервера имеют тип «m1.small Amazon EC2 Instance» и обладают следующими характеристиками: 1 виртуальный процессор, 1.7 GB оперативной памяти, 1 жесткий диск объемом 160 GB. Поскольку такая конфигурация сервера не позволяет адекватно оценить максимальную производительность приложения, основной целью нагрузочного тестирования являлась оценка возможности масштабирования приложения.

Целевой метрикой нагрузочного тестирования является среднее время получения ответа от сервера, а также зависимость среднего времени ответа от количества виртуальных пользователей.

На рисунке 3.8 представлены результаты нагрузочного тестирования при 100 виртуальных пользователях для одного сервера веб-приложения, на рисунке 3.7 — для двух.

Исходя из графиков, можно сказать, что среднее время ответа при добавлении одного дополнительного веб-сервера сокращается в два раза, таким образом архитектура разработанного приложения обеспечивает линейное масштабирование.

Также стоит отметить, что в процессе проведения нагрузочного тестирования производилось профилирование базы данных. Ни одного медленного (время выполнения больше 100 миллисекунд) запроса выявлено не было.

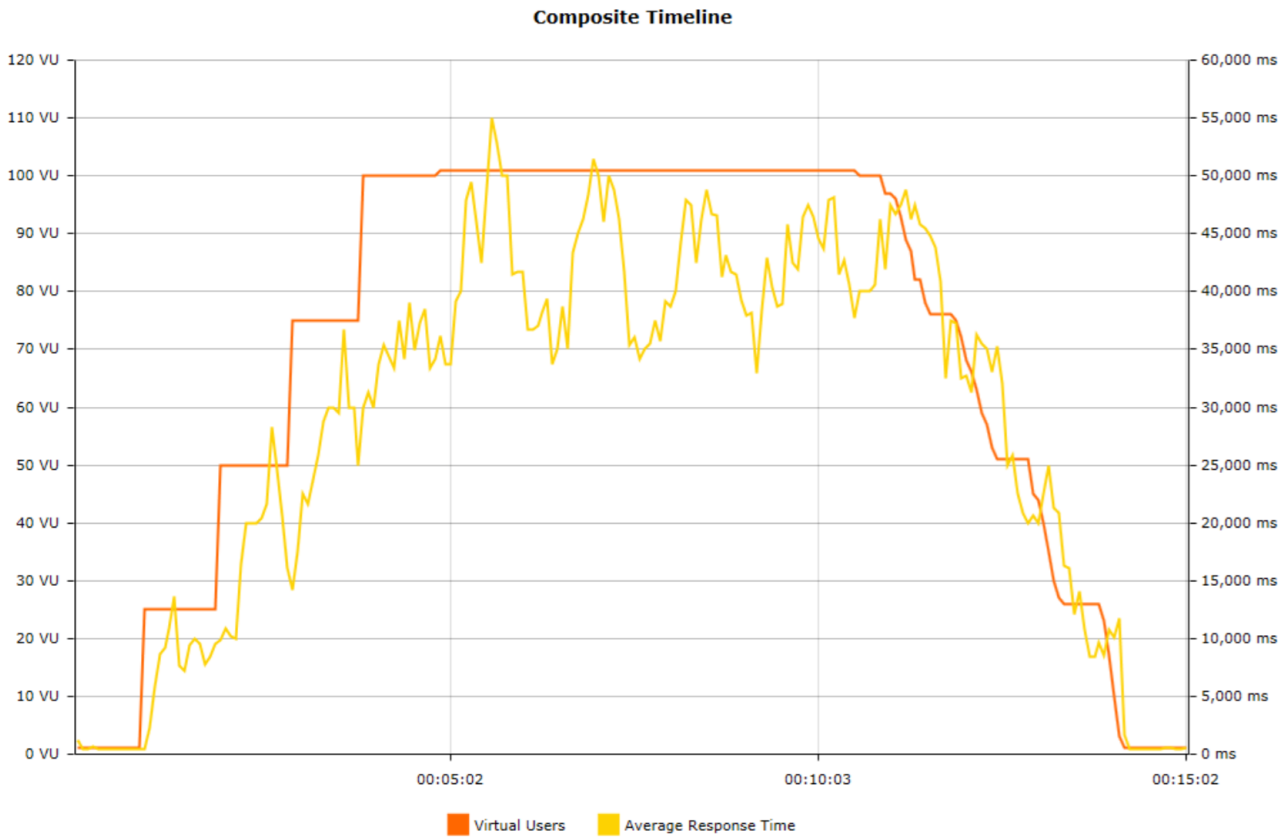

Рисунок 3.7 — График нагрузочного тестирования для одного веб-сервера

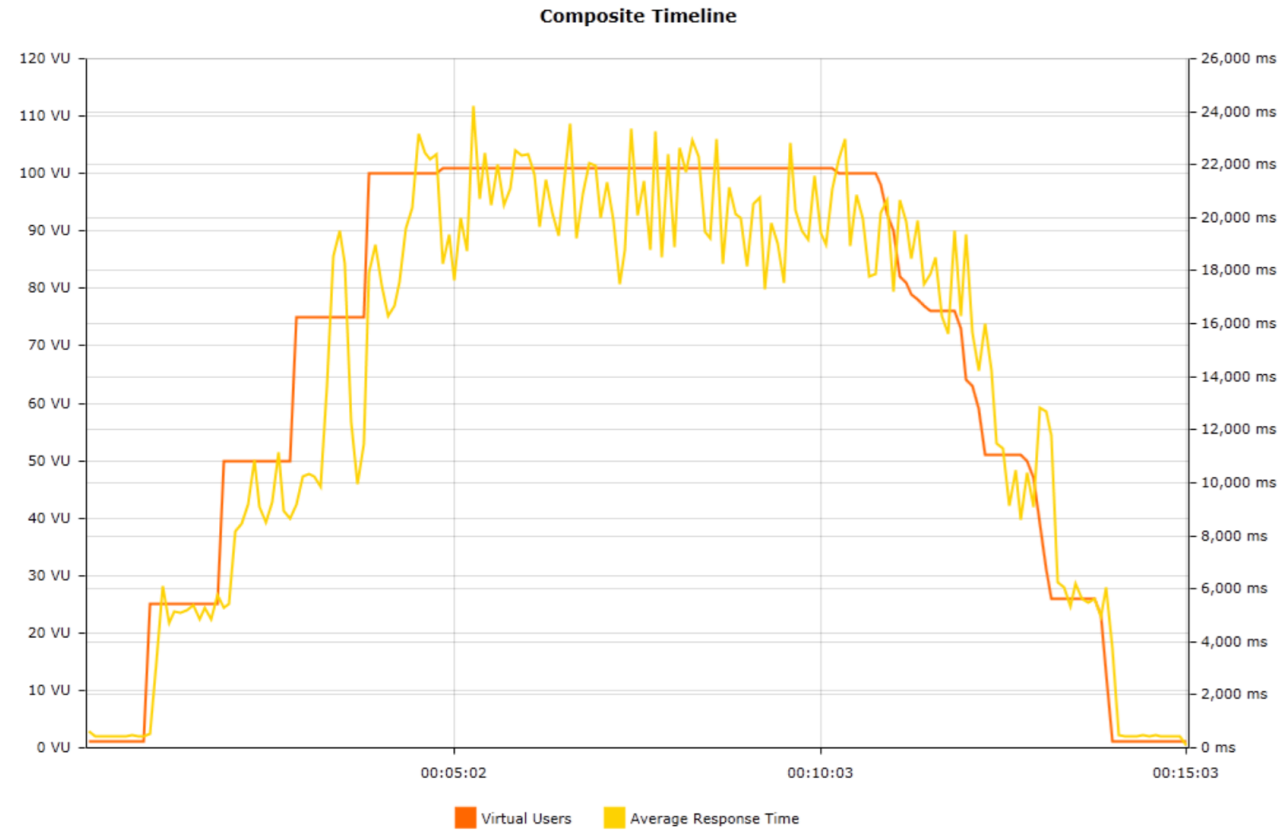

Рисунок 3.8 - График нагрузочного тестирования для двух веб-серверов

ЗАКЛЮЧЕНИЕ

Целью данной дипломной работы была разработка высокопроизводительной системы обработки и визуализации статистики в реальном времени в виде серверного кластера масштабируемого web-приложения реального времени, реализующего симулятор менеджера спортивных команд для нескольких видов спорта. В ходе работы над проектом были пройдены следующие этапы:

- Были проанализированы существующие методы и подходы к реализации масштабируемых web-приложений реального времени. Произведен сравнительный анализ данных методов.
- При проектировании была проанализирована предметная область симулятора менеджера спортивных команд и определены входящие в нее сущности. Разработана общая архитектура масштабируемого и отказоустойчивого серверного кластера, а также схема функционирования данного кластера в режиме реального времени.
- На основе результатов проектирования реализовано web-приложение симулятора менеджера спортивных команд. Произведена полная автоматизация разворачивания серверного кластера для данного приложения
- Произведено нагрузочное тестирование, доказавшее возможность горизонтального масштабирования разработанной архитектуры приложения.

Разработанная система была выполнена с использованием передовых архитектурных решений и технологий, существующих на данный момент.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Berners-Lee, Tim. The World Wide Web - Past, Present and Future / Tim Berners-Lee // J. Digit. Inf. — 1997. — Vol. 1, no. 1.
2. Шкляр, Л. Архитектура веб-приложений: принципы, протоколы, практика: / Пер. с англ. / Л. Шкляр, Р. Розен. Высший уровень. — Эксмо, 2011.
3. M. H. Reenskaug, Trygve. Thing-Model-View-Editor — an Example from a planning system. — 1979.
4. Ньюкомер, Эрик. Веб-сервисы / Эрик Ньюкомер. Для профессионалов. — Питер, 2003.
5. Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures: Ph.D. thesis. — University of California, Irvine, 2000. — AAI9980887.
6. Лабберс, Питер. HTML5 для профессионалов: мощные инструменты для разработки современных веб-приложений: / Пер. с англ. / Питер Лабберс, Брайан Олбери, Фрэнк Салим. Expert’s Voice. — Вильямс, 2011.
7. RFC 6455. The WebSocket Protocol. — Электронные данные. — Режим доступа: https://tools.ietf.org/html/rfc6466. — Дата доступа: 08.03.2015.
8. Макконнел, С. Совершенный код. Мастер-класс / Пер. с англ. / С. Макконнелл. — СПб. : Издательско-торговый дом «Русская редакция», 2005. — 896 с.
9. Amazon Web Services [Электронный ресурс]. — Электронный данные. — Режим доступа https://aws.amazon.com/. — Дата доступа: 08.03.2015.
10. Amazon Elastic Compute Cloud [Электронный ресурс]. — Электронные данные. — Режим доступа: https://aws.amazon.com/ec2/. - Дата доступа: 08.03.2015.
11. Amazon Simple Storage Service [Электронный ресурс]. — Электронные данные. — Режим доступа: https://aws.amazon.com/s3/. — Дата доступа: 08.03.2015.

12. Amazon Cloudfront [Электронный ресурс]. — Электронные данные. — Режим доступа: https://aws.amazon.com/cloudfront/. — Дата доступа: 08.03.2015.
13. Python [Электронные ресурс]. — Электронные данные. — Режим доступа https://www.python.org/. — Дата доступа: 08.03.2015.
14. van Rossum, Guido. Python History [Электронный ресурс]. — Электронные данные. — Режим доступа: http://svn.python.org/view/*checkout*/python/trunk/Misc/HISTORY — Дата доступа: 08.03.2015.
15. MongoDB [Электронный ресурс]. — Электронные данные. — Режим доступа: http://mongodb.com/. Дата доступа: 08.03.2015.
16. Nginx [Электронный ресурс]. — Электронные данные. — Режим доступа: http://nginx.org/ru/. — Дата доступа: 08.03.2015.
17. Tornado web server [Электронный ресурс]. — Электронные данные. — Режим доступа: http://tornadoweb.com/. — Дата доступа: 08.03.2015.
18. RabbitMQ [Электронный ресурс]. — Электронные данные. — Режим доступа: https://www.rabbitmq.com/.
19. Бункер, Кайл. MongoDB в действии: / Пер. с англ. / Карл Бэнкер. — ДМК Пресс, 2012.