СОДЕРЖАНИЕ

[СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ 4](#_Toc197809184)

[ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 5](#_Toc197809185)

[ВВЕДЕНИЕ 6](#_Toc197809186)

[1. РАСПРЕДЕЛЁННЫЕ ВЫСОКОНАГРУЖЕННЫЕ ПРИЛОЖЕНИЯ 10](#_Toc197809187)

[1.1 Общее представление о распределенных приложениях 10](#_Toc197809188)

[1.2 Основные цели проектирования распределенных приложений 12](#_Toc197809189)

[1.2.1 Легкодоступность ресурсов 13](#_Toc197809190)

[1.2.2 Прозрачность распределений 14](#_Toc197809191)

[1.2.3 Открытость 16](#_Toc197809192)

[1.2.4 Масштабируемость 17](#_Toc197809193)

[1.2.5 Быстрое время отклика 19](#_Toc197809194)

[1.2.6 Дублирование 19](#_Toc197809195)

[1.3 Выводы к главе 20](#_Toc197809196)

[2. ОСНОВЫНЕ КОМПОНЕНТЫ АРХИТЕКТУРЫ 21](#_Toc197809197)

[2.1 Основные виды компонентов 21](#_Toc197809198)

[2.2 Пользователи и веб-сервисы 23](#_Toc197809199)

[2.3 Хранилища данных 25](#_Toc197809200)

[2.3.1 Реляционные базы данных 26](#_Toc197809201)

[2.3.2 Нереляционные базы данных 26](#_Toc197809202)

[2.3.3 Кэш 27](#_Toc197809203)

[2.3.4 CDN 28](#_Toc197809204)

[2.4 Сервисы бизнес-логики 29](#_Toc197809205)

[2.5 Элементы управления потоком 32](#_Toc197809206)

[2.5.1 Системы обнаружения сервисов 33](#_Toc197809207)

[2.5.2 Балансировщики нагрузки 34](#_Toc197809208)

[2.5.3 Очереди сообщений 35](#_Toc197809209)

[2.6 Внешние сервисы 36](#_Toc197809210)

[2.7 Выводы к главе 37](#_Toc197809211)

[3. ТРЕБОВАНИЯ К РАСПРЕДЕЛЕННЫМ ВЫСОКОНАГРУЖЕННЫМ ПРИЛОЖЕНИЯМ 38](#_Toc197809212)

[3.1 Нефункциональные требования 38](#_Toc197809213)

[3.1.1 Масштабируемость 39](#_Toc197809214)

[3.1.2 Латентность 40](#_Toc197809215)

[3.1.3 Доступность 41](#_Toc197809216)

[3.1.4 Производительность 42](#_Toc197809217)

[3.1.5 Общее количество пользователей 44](#_Toc197809218)

[3.1.6 Безопасность 44](#_Toc197809219)

[3.1.7 Отказоустойчивость 45](#_Toc197809220)

[3.2 Функциональные требования 46](#_Toc197809221)

[3.3 Выводы к главе 46](#_Toc197809222)

[4. ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ 47](#_Toc197809223)

[4.1 Общие сведения о DSL 47](#_Toc197809224)

[4.3 Разработка компилятора 47](#_Toc197809225)

[4.8 Анализ реализаций при помощи формул 48](#_Toc197809226)

[4.20 Анализ опыта разработки схожих продуктов 49](#_Toc197809227)

[5. РАЗРАБОТКА СЕМАНТИКИ И СИНАКСИСА 50](#_Toc197809228)

[5.1 Подзаголовок 50](#_Toc197809229)

[5.2 Подзаголовок 50](#_Toc197809230)

[6. РАЗРАБОТКА ОСНОВНОЙ БИБЛИОТЕКИ 51](#_Toc197809231)

[6.1 Подзаголовок 51](#_Toc197809232)

[6.2 Подзаголовок 51](#_Toc197809233)

[7. РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ 52](#_Toc197809234)

[7.1 Фронтенд 52](#_Toc197809235)

[7.1.1 Выбор технологий 52](#_Toc197809236)

[7.1.2 Делали реализации 53](#_Toc197809237)

[7.2 Бэкенд 53](#_Toc197809238)

[7.2.1 Выбор технологий 54](#_Toc197809239)

[7.2.2 Делали реализации 54](#_Toc197809240)

[8. ТЕСТИРОВАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ 55](#_Toc197809241)

[8.1 Подзаголовок 55](#_Toc197809242)

[8.2 Подзаголовок 55](#_Toc197809243)

[ЗАКЛЮЧЕНИЕ 56](#_Toc197809244)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 57](#_Toc197809245)

[ПРИЛОЖЕНИЕ А 61](#_Toc197809246)

# СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

DSL – domain-specific language, предметно ориентированный язык

UML – unified model language, язык описания для объектного моделирования

IoT – internet of things, интернет вещей

GPL – general-purpose language, язык общего назначения

СУБД – система управления базами данных

СУРБД – система управления реляционными базами данных

CDN – content delivery network (сеть доставки контента)

SQL – structured query language (структурированный язык запросов)

ACID – atomicity, consistency, isolation, durability (атомарность, согласованность, изолированность, стойкость).

SOA – service-oriented architecture (сервисно-ориентированная архитектура)

DNS – domain name system (система имён доменов)

APN – Apple push notification (служба уведомлений Apple)

HTTP – hyper-text transfer protocol

REST – Representational State Transfer

# ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Предметно ориентированный язык программирования – это язык программирования, специально разработанный для решения задач в определенной области или домене, обеспечивая более высокую выразительность и удобство по сравнению с общими языками программирования.

Синтаксис

# ВВЕДЕНИЕ

В современном мире информационные технологии развиваются в стремительном темпе и это значительным образом сказывается на приложениях с большим количеством пользователей. В последнее время люди всё чаще используют информационные системы для различных целей, таких как коммуникация, хранение и поиск данных, управление бизнесом, совершение покупок и т.д.

Распределенные приложения – это информационные системы, состоящие из нескольких взаимодействующих компонентов, каждый из которых находится и функционирует на отдельной вычислительной машине, коммуницирует с другими при помощи сети и выполняет свою определенную задачу в системе. Ввиду же большого количества пользователей и высокой стоимости вычислительных мощностей, такие системы практически всегда являются высоконагруженными.

Во время зарождения информационных технологий надобности в их распределении не было, программы проектировали и писали с расчётом на то, что они будут исполняться на одном компьютере. В наши дни же практически каждое приложение является распределенным и подразумевает доступ к нему из разных географических точек [1]. Рассмотрим в качестве абстрактного примера сайт небольшой компании, которая занимается продажей каких-то товаров или услуг. Как правило он будет предоставлять возможность создания аккаунтов пользователей, что влечёт за собой необходимость реализации авторизации и аутентификации, а также хранения их данных. Помимо этого, необходимым будет и хранение данных о товарах, возможность онлайн оплаты и прочие дополнительные функции, как, например, почтовая рассылка об обновлениях ассортимента.

Проектирование архитектуры высоконагруженных распределенных систем – крайне сложная задача, которая под силу далеко не каждому разработчику [1]. В том числе это связано и с тем, что ошибка в архитектуре, которую не удалось обнаружить на этапе проектирования, может крайне негативно сказаться на работоспособности и стоимости системы, так как для того, чтобы её исправить, скорее всего придётся переписать или даже разработать заново большое количество компонентов. Предметно-ориентированный язык для генерации этой архитектуры может значительно снизить количество таких ошибок, так как он разрабатывается для выдачи схемы архитектуры, соответствующей основным принципам дизайна систем. Самостоятельно же человек может сразу не учесть необходимые компоненты или добавить лишние.

Задача проектирования высоконагруженных распределенных приложений также требует множество навыков от разработчика. Он должен хорошо разбираться в видах компонентов распределённых систем, таких как базы данных, очереди сообщений, веб-сервера и т.д., иметь представление о системном дизайне, уметь формулировать и оценивать функциональные и нефункциональные требования к продукту [2]. DSL же позволит ограничиться только описанием требований к приложению для получения архитектурной схемы, которую разработчик сможет при желании изменить или дополнить. Такой подход может позволить спроектировать распределённое приложение менее опытному разработчику и за меньший промежуток времени.

Таким образом, разработка DSL для генерации архитектуры высоконагруженных распределенных приложений **актуальна**, так как текущие потребности в проектировании таких систем высоки, а язык сможет помочь избежать ошибок, упростить и ускорить процесс системного дизайна.

Касаемо **степени разработанности** данной темы ВКР, за последние десятилетия было опубликовано множество материалов, посвящённых проектированию распределенной архитектуры, например книги таких известных в сфере информационных технологий авторов, как Брендан Бёрнс, Эндрю Таненбаум, Мартин Клеппман. Эти работы сфокусированы на изучении особенностей распределенных систем и подходов к их самостоятельному построению. Также стоит отметить существующие исследования на тему методов распределения задач по узлам сетевой инфраструктуры, один из которых рассмотрен в [3]. Они решают более конкретные задачи проектирования, однако близки к данной теме. С моей точки зрения, недостаточно изучены возможности автоматизации проектирования архитектуры информационных систем и применения языков программирования для этой задачи. В научных работах при этом хорошо исследованы способы описания такой архитектуры при помощи нотации UML, один из подробных примеров этого рассмотрен в [4].

**Решаемая проблема**: существенная сложность, значительные временные и интеллектуальные затраты на проектирование архитектуры высоконагруженного распределенного предложения без использования средств автоматизации.

**Цель**: автоматизация проектирования архитектуры высоконагруженного приложения.

**Задачи**:

1. Проанализировать особенности проектирования архитектуры распределенных высоконагруженных приложений и существующих реализаций DSL, выявить задачи, которые должен решить DSL.
2. Спроектировать синтаксис и семантику языка.
3. Создать программные средства: фронтенд для работы с языком и бэкенд для исполнения программ на языке, формирования результатов.
4. Провести тестирование на корректность работы и выполнения поставленной цели, доработать при необходимости на основе его результатов.

**Практическая значимость работы** заключается в том, что разработанный в ходе её выполнения предметно-ориентированный язык может быть использован в качестве вспомогательного средства для проектирования и оценки архитектуры распределенных высоконагруженных приложений. Пользователи языка смогут ввести желаемые функциональные и нефункциональные требования и быстро получить UML-диаграмму желаемой архитектуры, после чего доработать её по желанию, либо сразу использовать для дальнейшего создания приложения.

# 1. РАСПРЕДЕЛЁННЫЕ ВЫСОКОНАГРУЖЕННЫЕ ПРИЛОЖЕНИЯ

Для того, чтобы говорить о разработке DSL для генерации архитектуры распределенных высоконагруженных приложений, необходимо сначала разобраться, что такие приложения из себя представляют, как характеризуются и в каких случаях используются. В этой главе рассматриваются основные сведения о распределенных высоконагруженные приложениях.

## 1.1 Общее представление о распределенных приложениях

Распределенное приложение по сути является системой, состоящей из совокупности вычислительных компонентов, каждый из которых выполняет свою задачу и может работать независимо от других. У распределенных приложений есть две основные характерные особенности: они всегда являются совокупностями вычислительных компонентов и при этом выглядят для пользователей как единые монолитные системы [5].

Компоненты современных распределённых приложений как правило могут сильно различаться между собой. Они могут быть как программными, запущенными на вычислительных машинах, так и полностью аппаратными. Технические характеристики у разных компонентов могут сильно варьироваться. Это могут быть как мощные сервера с большим количеством оперативной памяти и внушительной пропускной способностью, так и небольшие компьютеры – всё зависит от конкретной задачи и требований к системе. Количество вычислительных компонентов в распределенном приложении может варьироваться в пределах от нескольких единиц до нескольких миллионов, оно также зависит от требований.

Как правило распределенные приложения являются многопользовательскими, так как зачастую как раз большое количество пользователей в разных точках мира и побудило делать их таковыми. Такие приложения намного легче быстро адаптировать под растущее количество пользователей путём добавления дополнительных компонентов, а значит и наращивания общей вычислительной мощности всей системы. Такой процесс называется горизонтальным масштабированием. Масштабировать приложение можно и вертикально, то есть улучшать текущее оборудование или заменять его на более совершенное. Однако такой подход значительно менее эффективен в большинстве случаев, так как зачастую покупка большого количества стандартных вычислительных машин выгоднее чем нескольких продвинутых с аналогичными суммарными характеристиками.

В массе своей распределенные приложения являются распределенными не просто между большим количеством компьютеров, но и географически – то есть эти компьютеры с вычислительными компонентами располагаются в разных точках мира в зависимости от географии целевых пользователей. Это также помогает обеспечить пользователям из разных мест равный доступ к системе на столько, на сколько это возможно. Достигается такой эффект благодаря грамотному расположению компонентов и настройке их взаимодействия. Например, запросы пользователя некого веб-приложения будут попадать на ближайший к нему доступный сервер-обработчик запросов.

Говоря в общих чертах о распределенных приложениях, стоит уточнить, что или кого мы понимаем под их пользователями. В глобальном плане пользователей можно разделить на две категории: люди и другие приложения [5]. Наиболее распатроненный сценарий взаимодействия человека с распределенным приложением это использование веб-клиента – отправка запросов и получение ответов согласно одному из общепринятых протоколов при помощи браузера, мобильного или десктопного клиентского приложения. Случаи, когда в роли клиента выступает программа имеют большее разнообразие. Чаще всего это происходит, когда необходимо взаимодействие двух разных информационных систем для выполнения какой-либо общей задачи. К примеру, интернет-магазин может взаимодействовать с системой банка для реализации удобной онлайн оплаты прямиком из корзины покупателя. Исключением в данном случае не являются и IoT-системы. К ним можно отнести устройства умного дома, которые отправляют данные и принимают указания из облачного распределенного приложения, умные сельскохозяйственные системы и многое другое. Тут также стоит упомянуть и планировщики задач. Хотя их и трудно назвать полноценными клиентами, планировщики инициируют выполнения какого-либо процесса в приложении в зависимости от какого-либо условия. Планировщики как правило являются частью самой распределенной системы, то есть можно сказать, что приложение в коком-то смысле самостоятельно решает выполнить те или иные действия (разумеется, их условия заранее прописывает человек).

## 1.2 Основные цели проектирования распределенных приложений

Проведя анализ доступных научных работ на тему распределенных систем и личного пользовательского опыта, я пришёл к выводу, что существует ряд целей, которыми как правило задаются разработчики при проектировании современных высоконагруженных распределенных приложений. Их выявление крайне важно для достижения поставленной цели данной выпускной квалификационной работы, так как программные средства автоматизация проектирования таких приложений должны их учитывать для достижения более качественного результата. Согласно [5, 6], я выделил следующие цели дизайна высоконагруженных распределенных приложений: легкодоступность ресурсов для пользователей, прозрачность распределений, открытость, масштабируемость, быстрое время отклика и дублирование вычислительных мощностей. Этот список таже является ответом на вопрос о том, чего мы хотим добиться, делая наше высоконагруженное приложение распределённым. Далее следует более подробное описание каждой из них.

### 1.2.1 Легкодоступность ресурсов

Для начала следует определиться, что подразумевается под ресурсами в контексте распределенных приложений. Это могут быть как файлы самых разных типов, так и хранилища данных, внешние устройства, с которыми взаимодействуют компоненты приложения, внешние сервисы или даже другие распределенные приложения, компьютерные сети. Стоит отметить, что представленный список далеко не полный, так как различных видов ресурсов ещё очень много. [5]

В данном случае важно понимать и причины, по которым возникает необходимость организации доступа для пользователей или компонентов самого приложения к этим ресурсам. Как правило все эти причины вытекают из потребностей конечных пользователей – людей. Это может быть распространение различного контента, сбор статистики для анализа, долгосрочное хранение данных, измерение физических показателей чего-либо и многое другое.

Организация лёгкого доступа к необходимым ресурсам в соответствии с потребностями имеет особую значимость в случае распределенных приложений, в том числе и с высокой нагрузкой. Как было сказано в пункте 1.1, такие приложения делятся на множество компонентов и нередко возникают ситуации, когда необходима организация совместного использования ресурсов между этими компонентами. К примеру, один компонент может обновлять ресурс, а другой читать. Кроме того, в таких приложениях постоянно возникают случаи замены, удаления и добавления новых компонентов, как в рамках развития системы путём добавления новых функций, так и в рамках её масштабирования для поддержки возрастающего числа пользователей.

Таким образом, работа с ресурсами в распределенных высоконагруженных приложениях должна быть организована таким образом, чтобы доступ к ним настраивался легко и быстро для компонентов приложений и пользователей.

### 1.2.2 Прозрачность распределений

Ещё одной важной целью проектирования распределенных систем является достижение необходимой степени прозрачного распределения ресурсов и процессов между компонентами и пользователями этих приложений. Основная задача прозрачности распределений заключается в создании иллюзии для пользователя, что они имеют дело с монолитной системой, функционирующей на одном устройстве, в то время как в реальности множественные вычислительные узлы и используемые ими ресурсы расположены в удалении друг от друга [5, 8.] Во время работы клиента с приложением отдельные компоненты могут выходить из строя или наоборот восстанавливать свою работу, ресурсы могут перемещаться между компонентами [8].

Стоит учитывать и тот факт, что в современном мире огромной популярностью пользуются технологии беспроводной связи. Множество людей использует смартфоны и другие устройства с возможностью выхода в интернет без наличия кабельного подключения, а значит можно сделать вывод, что пользователи также могу перемещаться во время использования приложения. Например, крайне желательно, чтобы пользователь не замечал переключения на оптимальный сервер в связи с изменением его географического положения.

Однако, прозрачность распределений не ограничивается только географической. Существуют разные формы прозрачности в распределенных приложениях. При этом не обязательно реализовывать все эти формы при проектировании распределенного приложения, так как во многих случаях реализация той или иной формы прозрачности распределений может противоречить требованиям к этом приложению. В таблице 1.1 приведён список существующих форм с пояснениями [5].

Таблица 1.1 – Формы прозрачности распределений

|  |  |
| --- | --- |
| **Форма** | **Пояснение** |
| Доступ | Сокрытие различий в представлении одних и тех же данных разными компонентами приложения и различий средств, предоставляющих эти данные. Типичный пример – использование стандартных протоколов обмена данными в сети. |
| Местоположение | Сокрытие реального местоположения компонентов и ресурсов от пользователя или других компонентов. Распространённым примером является использование логических имён или уникальных идентификаторов. |
| Перемещение | Сокрытие факта перемещения ресурса в другое место от пользователя или другого компонента во время использования. |
| Миграция | Сокрытие факта перемещения части распределенного приложения. В качестве примера можно привести два перемещающихся устройства коммуникации во время их взаимодействия. |
| Репликация | Сокрытие факта наличия нескольких реплик ресурса или компонента и факта замены одного на другой в случае возникновения проблем с одним из экземпляров. |
| Параллельность | Сокрытие факта параллельного доступа к одному ресурсу или компоненту приложения. |
| Отказ | Сокрытие факта возникновения сбоев и процесса восстановления |

Таим образом, при проектировании распределенных приложений важно достигать соблюдения форм прозрачности распределений, опираясь на требования к приложению. Если эти требования не противоречат достижению какой-либо формы, её стоит соблюдать, так как это упростит развитие приложения и его использование.

### 1.2.3 Открытость

При проектировании распределенного приложения важно ставить цель сделать его открытым. Реализация данного принципа по большей части осуществляется во время реализации компонентов приложения, но при проектировании архитектуры его также стоит учитывать для того, чтобы правильно построить связи между этими компонентами.

Чтобы распределенное приложение называлось открытым, каждый из его компонентов должен предоставлять удобный интерфейс взаимодействия с ним. Интерфейс должен иметь понятную документацию с описанием всех необходимых функций, а также должен реализовывать один из или ряд общеизвестных протоколов взаимодействия. Помимо этого, компоненты открытого распределенного приложения должны быть портативными, то есть должны иметь возможность легко без изменений встраиваться в другое открытое распределенное приложение. Также такие приложения должны быть расширяемыми, обеспечивая простоту интеграции в себя новых компонентов, и легко конфигурируемыми, в том числе и с компонентами других приложений или от других разработчиков. [5, 9]

Если ещё на этапе проектирования архитектуры ставить перед собой цель сделать распределенное приложение расширяемым, то можно будет с большей вероятностью избежать ошибок в ней. Кроме того, такой подход может заставить задуматься о своевременном включении в архитектуру специальных компонентов для организации взаимодействия с компонентами и целыми приложениями от других разработчиков. Интеграция сторонних компонентов в приложения в современных распределенных системах встречается крайне часто, так как намного проще, быстрее и выгоднее взять существующие проверенные реализации распространённых функций. Более подробные примеры этого рассмотрены в главе 2.

### 1.2.4 Масштабируемость

Масштабируемость является одной из основных целей проектирования распределенных высоконагруженных приложений, так как подобным приложениям свойственно расти во время их эксплуатации. Под ростом здесь подразумевается такие факторы, как увеличение количества пользователей, числа и частоты запросов, объема хранимых данных и т.д. У вычислительного оборудования и хранилищ есть конструктивные ограничения на количество операций в секунду и объема хранимых данных соответственно. По этой причине, чтобы система продолжала отвечать заданным требованиям в условиях постоянно увеличивающейся нагрузки, необходимо заменять оборудование на более совершенное или увеличивать его количество.

Распределённое приложение можно масштабировать не только относительно его размера. Существует три измерения, при помощи которых можно определить масштабируемость, они приведены в таблице 1.2 [5].

Таблица 1.2 – Измерения масштабируемости

|  |  |
| --- | --- |
| **Измерение** | **Пояснение** |
| Размер | Если распределённое приложение масштабируемо относительно своих размеров, оно позволяет добавлять в себя дополнительные ресурсы и пользователей, не неся при этом значительных потерь в производительности. К примеру, мессенджер, возросший от 10000 до 10000000 пользователей, должен работать без ощутимых изменений для всех пользователей. |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| География | Если распределенное приложение масштабируемо географически, это значит, что оно позволяет добавлять в себя удалённые компоненты и ресурсы, а также поддерживать работу с удалёнными пользователями. Например, если приложение становится доступно для пользователей в другой стране, то его производительность для них не должна ощутимо отличаться в сравнении с другими пользователями. Эта задача часто решается путём создания центров обработки данных (ЦОД) для каждого региона [2]. ЦОД описаны в главе 2. |
| Администрирование | Если приложение масштабируемо административно, то простота его администрирования практически не снижается при добавлении в него компонентов, управляемых другими организациями. |

Рассмотрим немного подробнее масштабируемость относительно размеров, так как как правило именно её чаще имеют в виду, говоря о масштабируемости распределенных приложений. Она делится на два типа: вертикальную и горизонтальную. Вертикальная подразумевает под собой модификацию существующих вычислительных машин с целью повышения производительности до соразмерных возрастающей нагрузке величин. Горизонтальная же реализуется при помощи увеличения количества этих машин с целью параллельного исполнения одних и тех же задач [10].

На этапе проектирования архитектуры распределенных приложений крайне важно учитывать планируемые возможности горизонтального масштабирования размеров, так как они значительно влияют на общую архитектуру ввиду необходимости управления разделением вычислений между распараллеленными компонентами.

### 1.2.5 Быстрое время отклика

Так как данная выпускная квалификационная работа сфокусирована именно на распределенных приложениях, которые являются высоконагруженными, то времени отклика тоже стоит уделить внимание внимания. Если пользователь ждёт результат достаточно продолжительное время, то у него может сложиться негативное впечатление о приложении и он даже может перестать им пользоваться [11].

Во время проектирования архитектуры распределенного приложения необходимо связывать компоненты таким образом, чтобы время отклика для пользователей было быстрым. Быстрота в данном случае определяется индивидуальными требованиями к приложению. Стоит задуматься об организации балансировки нагрузки, чтобы пользователи имели примерно равную скорость доступа к одним и тем же функциям системы. Также доступ к данным, которые часто используются, должен быть высокоскоростным [2]. Здесь упомянуты далеко не все способы оптимизировать время отклика, для этого можно использовать и специальные компоненты (см. глава 2).

### 1.2.6 Дублирование

В любом распределённом приложении есть критически важные компоненты, при выходе из строя которых оно перестаёт выполнять его основные задачи. Такое может произойти по разным причинам: как из-за технического сбоя или поломки оборудования, так и из-за потери соединения с другими частями приложения, отключения электричества или иной нештатной ситуации. Чтобы избежать проблем функционирования приложения во время подобных сбоев, стоит спроектировать архитектуру с учётом дублирования его важных компонентов [6].

В случае с компонентами, выполняющими вычислительные операции можно организовать замену недоступного другим его исправным экземпляром прямо во время работы приложения. Предполагается, что экземпляров каждого такого компонента должно быть несколько изначально. В случае с хранилищами данных можно использовать различные механизмы репликации. Репликация предполагает дублирование хранилищ данных и постоянную синхронизацию содержимого между этими репликами [2, 5].

## 1.3 Выводы к главе

В текущей главе были изучены и собраны общие сведения о распределённых высоконагруженных системах, а также цели их проектирования. Всё это необходимо для выполнения пунктов практической части, связанных с описанием и генерацией распределенно й архитектуры.

# 2. ОСНОВЫНЕ КОМПОНЕНТЫ АРХИТЕКТУРЫ

После того, как были определены основные характеристики и особенности распределенных высоконагруженных приложений, настало время поговорить об архитектуре. В данной главе исследуются её основные элементы на уровне отдельных компонентов распределенных приложений. Основной задачей главы является перечислить основные их виды, исследовать логику их работы и определить, с какими целями применяется каждый из них.

## 2.1 Основные виды компонентов

Стоит уточнить, что в данной работе под архитектурным компонентом распределенного приложения понимается независимое программное или аппаратное обеспечение или элемент, логически объединяющий несколько других компонентов [2, 12]. Отдельные программные модули, на которые разделяются компоненты, имеющие программный код, в этой работе рассматриваться не будут, так как она сфокусирована на общей архитектуре, а не на конкретных реализациях.

Чтобы начать исследовать функции и случаи применения тех или иных компонентов, для начала нужно определиться, какие они бывают и на какие категории их можно поделить.

Рассмотрим в качестве примера задачу проектирования распределенной архитектуры приложения публикации статей и формирования новостной ленты для его пользователей. Разумеется, в реальности подобное приложение не будет ограничиваться только этими функциями. На рисунке 2.1 изображена схема архитектуры для публикации постов [2]. При помощи этой схемы выясним, какие компоненты и для чего выделяются при проектировании архитектуры.

Изображение выглядит как текст, диаграмма, План, Технический чертеж

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.1 – Схема архитектуры процесса публикации постов

* Для представления пользователя на схеме указан специальный компонент, в котором перечислены возможные типы клиента для конкретного приложения.
* Внешние информационные системы обозначаются как один компонент (на данной схеме примером внешней системы является DNS).
* Для хранения данных существуют разные компоненты в зависимости от типа хранилища.
* Действия бизнес-логики, включающие работу с данными, выполняют отдельные сервисы, каждый из которых выполняет свою определенную задачу.
* Компоненты коммуницируют либо напрямую друг с другом, либо через специальные компоненты, которыми в данном примере является балансировщик нагрузки и очередь сообщений.

Из приведённого выше примера и литературы по данной теме выделим следующие виды компонентов распределенной архитектуры: пользователи, различные хранилища данных, внешние сервисы, элементы управления потоком информации и сервисы для бизнес-логики. Из сервисов бизнес-логики можно также отдельно выделить веб-серверы, так как им характерна именно обработка пользовательских запросов [1, 2, 4, 5, 12]. Далле в этой главе рассмотрены подробнее каждый из этих видов компонентов.

## 2.2 Пользователи и веб-сервисы

Компонент-пользователь в распределенном приложения, по сути, всегда является программным обеспечением. В случае, когда реальным пользователем является человек, он всё равно взаимодействует с распределенным приложением через клиентское программное обеспечение, к примеру через мобильное приложение, сайт, приложение на компьютере и т.д. В распределенных приложениях нередко бывает и несколько типов пользователей одновременно [2]. На архитектурных схемах пользователь может обозначаться по-разному, один из примеров приведён на рисунке 2.1. Если же использовать формат UML, то пользователя обычно обозначают как на рисунке 2.2 [4]. Слово «Актёр» на рисунке 2.2 в данном контексте обозначает действующее лицо.

Изображение выглядит как текст

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.2 – Пример обозначения пользователя в UML

Хорошей практикой при проектировании распределенных высоконагруженных систем является отделение веб-уровня от остальной бизнес-логики [2, 13]. Именно по этой причине компонент пользователя почти всегда связан с компонентами веб-серверов. Веб-сервер является точкой доступа пользователей к распределенному приложению, а в случае большого количества пользователей и запросов быстро становится узким местом. По этой причине стоит иметь несколько реплик веб-сервера для параллельной обработки запросов и замены вышедших из строя экземпляров [5, 14].

Стоит заметить, что при проектировании архитектуры распределённого приложения важно не предполагать хранение состояние между запросами пользователя на веб-серверах. Хранение состояния затруднит балансирование нагрузки между экземплярами веб-сервера и автоматическое масштабирование [2, 5]. Нужную информацию можно сохранять на время надобности в отдельное хранилище данных. Пример схемы архитектуры без хранения состояния в веб-серверах показан на рисунке 2.3 [2].

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.3 – Пример архитектуры без состояния на веб-серверах

## 2.3 Хранилища данных

Любое распределенное приложение работает с теми или иными данными и практически всегда требуется реализация их хранения. Чтобы выбрать необходимое хранилище необходимо ответить на несколько вопросов: какие данные нужно хранить, как долго их хранить, как часто они читаются/изменяются, каков их объем. Если приложение высоконагруженное, то нелишним будет ещё и задаться вопросом, как быстро хранилище должно предоставить те или иные данные. В данной работе рассмотрены несколько основных типов хранилищ в распределенных системах: реляционные базы данных (СУРБД), нереляционные базы данных, кэш, CDN [2, 5, 15].

### 2.3.1 Реляционные базы данных

Реляционные базы данных позволяют хранить данные в виде таблиц. Управлять этими таблицами и хранящимися в них данными можно при помощи языка SQL, что делает этот доступ к данным унифицированным и удобным [2, 15].

Такие базы данных основываются на принципах ACID, а именно атомарности, согласованности, изолированности и стойкости. Это значит, что СУРБД позволяют выполнять транзакции, которые выполняются либо целиком, либо не выполняются вообще, сохраняют и изменяют данные в соответствии с заданными в базе данных правилами и ограничениями, изолированы друг от друга во время выполнения и гарантированно сохраняют результат при успешном выполнении [15].

Из этого можно сделать вывод, что реляционные базы данных хорошо подходят для задач, в которых важны выполнения принципов ACID. Из распространённых к таким относятся, к примеру, операции с валютой, которые требуют высокой надёжности и согласованности, так как ошибки могут дорого обойтись как компании, так и её клиентам. Реляционные базы данных также хорошо подходят для приложений, данные в которых структурированы, имеют не очень большой объем, их надёжность важнее скорости доступа, их предполагается хранить среднее или продолжительное время [2].

### 2.3.2 Нереляционные базы данных

Нереляционные базы отличаются большим разнообразием по способу хранения данных. Это могут быть хранилища вида «ключ-значение», графовые, объектно-ориентированные, иерархические и т.д. [2, 16]. Они не имеют унифицированного способа управления как в случае с SQL у реляционных баз данных, а значит разработчику нужно изучать структуру и способы управления каждой такой базой данных отдельно в зависимости от конкретной реализации.

Как правило такие нереляционные базы данных отвечают принципам BASE, а именно базовой доступности, гибкого состояния и согласованности в конечном итоге [15]. Это говорит о том, что согласованность в данном случае значительно менее строгая, чем в случае с ACID. Однако это позволяет значительно облегчить репликацию и масштабируемость таких хранилищ данных. Данные становится значительно проще делить на части и хранить в разных местах, что актуально при наличии огромных объёмов этих данных. Скорость доступа у нереляционных баз данных также выше, так как не нужно тратить много времени на согласование и можно лучше подобрать реализацию под конкретный тип данных.

Таким образом, нереляционные базы данных хороши если нужно хранить очень большие объемы данных, данные плохо структурированы, нужен быстрый доступ к данным и не так важна постоянная согласованность [2, 15].

### 2.3.3 Кэш

Кэш – еще один тип хранилищ, который часто применяется в распределенных системах, однако он сильно отличается от баз данных. Кэш основан на энергозависимой памяти, которая при этом намного меньше обычной, но доступ к такой памяти намного быстрее чем к базам данных.

Его часто используют вместе с базами данных, сохраняя туда самые часто запрашиваемые данные, а затем запрашивая данные из базы данных только в случае, если их нет в кэше. Это позволяет значительно снизить задержки доступа к данным, а значит и ответа приложения на запросы пользователя, связанные с данными. Нагрузка на базу данных при этом также снижается, так как удаётся избежать лишних обращений [17].

Лучше всего использовать этот тип памяти вместе с базой данных в тех случаях, если данные изменяются редко, но при этом часто читаются. В то же время кэш может быть использован и как основное хранилище каких-то данных, но только в случае, если данные необходимо хранить очень короткое время, их потеря не будет иметь какого-то большого значения и при этом необходим быстрый доступ к этим данным. Если же данные необходимо хранить долгосрочно, и важно, чтобы они не пропали, то следует использовать постоянное основное хранилище [2, 17].

Существуют разные стратегии размещения кэша для достижения большей эффективности его работы. К примеру, наиболее распространённые это локальное кэширование, когда каждый компонент имеет свой собственный кэш и управляет им, и глобальное – когда существует общий кэш для всех компонентов распределённого приложения [17]. В данном случае стоит найти баланс между этими вариантами.

### 2.3.4 CDN

CDN представляют из себя сети доставки контента из исходного источника до потребителя и основываются на энергозависимой кэш-памяти. В отличии от обычного использования кэширования CND сфокусированы именно на уменьшении времени получения данных при наличии значительного расстояния между исходным источником и потребителем [18].

Основной идеей при использовании этих компонентов (кэшей CDN практически всегда несколько) является располагать кэши CDN как можно ближе к потребителю. В этих кэшах сохраняется наиболее часто запрашиваемый контент и при обращении сначала ищется в них и только при неудаче поиска в кэшах CDN ищется непосредственно и сходном источнике [18].

Чтобы нагляднее продемонстрировать принцип работы CDN обратимся к рисунку 2.4. Предположим, что исходный источник находится далеко от пользователя и обращение к нему будет занимать в среднем 300 мс. Тем временем кэш CDN, который находится значительно ближе, позволяет получить эти же данные за 50 мс.

Изображение выглядит как текст, снимок экрана, диаграмма, круг

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.4 – Принцип работы CDN

CDN чаще всего используются для доставки статического контента, такого как веб-страницы, изображения, видео и т.д., хотя в современном мире всё чаще встречается использование для доставки динамического [2, 18]. В данной работе CDN рассматривается для хранения и доставки именно статических данных.

Таким образом, CDN стоит использовать в проектировании архитектуры в случаях, если приложение предполагает широкую географию пользования и частую доставку пользователям больного объёма редко изменяемых данных, имеет высокие требования к низкой латентности. Подробнее про латентность в главе 3.

## 2.4 Сервисы бизнес-логики

При проектировании архитектуры распределенного высоконагруженного приложения неизбежно встаёт вопрос о том, как организовать вычисления, отвечающие за бизнес-логику. Существует 3 распространённых типа построения архитектуры: монолит, SOA (сервис-ориентированная архитектура) и микросервисы [19]. Монолитная архитектура явно не подходит для распределенных систем, так как она подразумевает наличие только одного компонента, что противоречит особенностям и целям построения распределённых приложений, рассмотренных в главе 1. Остаётся SOA и микросервисная архитектура, которая является наиболее распространённым подтипом SOA [19].

SOA и микросервисная архитектура ориентированы на разбиение общей бизнес-логики приложения на отдельные сервисы, каждый из которых становится отдельным компонентом распределенного приложения. Однако есть различия между этими стилями архитектуры. Сервис в SOA как правило решает одну целую бизнес-задачу и может быть довольно большим в зависимости от сложности этой бизнес-задачи. Микросервисы же как правило создаются для каждой логически выделяемой функции, как например авторизация, отправка сообщений или добавление нового элемента пользователем. Это значит, что в отличие от SOA одна бизнес-задача может быть представлена несколькими микросервисами. Также микросервисы как правило имеют свои отдельные хранилища данных, а сервисы в SOA делят хранилища между собой [19, 20]. Визуальное сравнение SOA и микросервисной архитектуры показано на рисунке 2.5.

Изображение выглядит как снимок экрана, текст, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.5 – Визуальное сравнение SOA и микросервисной архитектуры

Небольшие сервисы микросервисной архитектуры можно более гибко и просто масштабировать независимо друг от друга [19]. Так как каждый из этих сервисов выполняет одну определенную функцию, для них проще сделать спецификацию и удобный интерфейс взаимодействия с ними, а значит и сделать всю распределенную систему открытой (см. пункт 1.2.3). Репликацию хранилищ, которые не разделяются между множеством сервисов, также легче организовать.

Таким образом, разделение бизнес-логики на отдельные компоненты при проектировании архитекторы распределенного приложения предпочтительнее осуществлять, основываясь на концепции микросервисной архитектуры. Спроектированной таким образом системе будет проще достигать целей дизайна распределенных приложений. Однако, в первую очередь стоит обращать внимание на требования к конкретному приложению, так как в зависимости от ситуации для некоторых компонентов стиль SOA может подойти куда лучше. К примеру, некоторые бизнес-требования могут требовать крайне большой объем различных вычислений и одновременно быть неудобными для разделения на более мелкие части.

## 2.5 Элементы управления потоком

Под потоком в данном случае понимается информация, передающаяся между компонентами распределенного приложения по сети. Иными словами, в этом разделе речь пойдёт о коммуникации между компонентами.

Рассмотрим в качестве примера простое приложение, состоящее из четырёх компонентов: пользовательское веб-приложение, веб-сервер, сервис с бизнес-логикой и база данных. Также предположим, что оно имеет небольшое количество пользователей, не предполагает горизонтального масштабирования и репликации данных, а также, что веб-сервер, сервис с бизнес-логикой и база данных существуют в единственном экземпляре без возможности миграции на другие вычислительные устройства. Лишь в подобных случаях возможна ситуация, когда компоненты знают адреса друг друга для осуществления коммуникации (указаны вручную в конфигурации каждого из них). В реальных же распределенных высоконагруженных приложениях компоненты постоянно добавляются, их экземпляры выходят из строя и заменяются новыми, метаположение компонентов также может меняться [5]. Вместе с этим возникает потребность и в дополнительных компонентах, которые будут налаживать коммуникацию внутри таких приложений. В данном разделе рассматриваются несколько основных из них: системы обнаружения сервисов, балансировщики нагрузки и очереди сообщений [2].

### 2.5.1 Системы обнаружения сервисов

Системы обнаружения сервисов (service discovery) используются в распределенных приложениях для того, чтоб каждый компонент, которому это нужно, мог при помощи них изучить своё окружение и получить необходимую информацию о других компонентах для дальнейшего взаимодействия с ними. Под сервисом в данном случае понимается любой программный или аппаратный компонент приложения [1]. К примеру, если веб-серверу необходимо обратиться к масштабируемому сервису с какой-то бизнес-логикой, он может использовать механизм обнаружения сервисов для получения информации о доступных экземплярах искомого сервиса и затем уже выбрать из них подходящий. Общая концепция изображена на рисунке 2.6.

Изображение выглядит как текст, снимок экрана, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.6 – Общая концепция обнаружения сервисов

На данный момент есть множество разных готовых реализаций систем для обнаружения сервисов. Распространёнными примерами являются прикладные программы, такие как Consul или Zookeeper, части фреймворков, например Spring Cloud. Также можно реализовать самостоятельно при помощи средств платформы, к примеру Jakarta EE.

Рассмотрим этот механизм немного подробнее на примере Consul. Данное программное обеспечение предполагает использование в распределенной среде и должно быть включено в каждый компонент, использующий его, то есть имеются в виду ищущие и искомые компоненты. Consul основывается на DNS, а значит способен предоставлять адреса обнаруженных сервисов. Он также может самостоятельно обнаруживать факт неисправности экземпляра искомого сервиса, что повышает вероятность предоставления компоненту-клиенту исправного экземпляра [21].

В данной работе не рассматриваются подробно детали реализаций систем обнаружения сервисов, а лишь приводится общая концепция их использования. Учитывать возможность интеграции этих систем при проектировании архитектуры распределенных высоконагруженных приложений стоит, так как они помогают достичь таких рассмотренных в главе 1 целей, как масштабирование (в данном случае имеется в виду горизонтальное), легкодоступность ресурсов (в данном случае ресурсы это связываемые компоненты) и дублирование.

### 2.5.2 Балансировщики нагрузки

Так как в распределённых высоконагруженных приложениях компоненты могут иметь несколько экземпляров, появляется необходимость организации выбора одного из них для выполнения каждого отдельного запроса. Для этого существуют специальные алгоритмы, которые как правило учитывают такие критерии, как загрузка экземпляров на момент выбора, их пропускная способность и т.д. Не исключён и выбор по географическому признаку, если экземпляры находятся на разном удалении от вызывающего компонента [22]. Балансировка нагрузки может выполняться вместе с обнаружением сервисов, чтобы сначала получить информацию о доступных экземплярах.

При проектировании распределенной архитектуры для решения задачи балансировки нагрузки обычно выделяется отдельный компонент – балансировщик нагрузки [2]. Использовать его можно в тех случаях, когда один компонент вызывает другой, имеющий несколько экземпляров. Это даёт множество преимуществ, таких как повышение общей производительности распределённого приложения, снижение времени отклика, более эффективная утилизация вычислительных ресурсов [23].

Балансировщик можно реализовать самостоятельно или выбрать одну из существующих реализаций, к примеру Ribbon в Spring Cloud, балансировщик в Kubernetes и другие.

### 2.5.3 Очереди сообщений

Некоторые задачи в распределенных приложениях стоит выполнять асинхронно. Это может быть как сохранение каких-либо данных, подтверждение о результате которого не нужно сообщать пользователю, так и отправка разного рода оповещений/уведомлений или обработка изображений или видео. Случаев, когда это может пригодиться, крайне много [2, 5]. Для организации асинхронного обмена информацией между компонентами часто используют очереди сообщений, принцип которых заключается в том, что отправитель публикует своё сообщение в соответствующей очереди, после чего компоненты-потребители данной очереди получают это сообщение. Фактически очереди сообщений осуществляют маршрутизацию сообщений между компонентами [2, 24].

Очереди сообщений как правило являются частью брокеров сообщений. Брокеры позволяют создавать очереди и управлять доступом к ним других компонентов. Наиболее известные реализации таких брокеров это RabbitMQ, Memphis, Apache Kafka, Apache ActiveMQ. Выбирать брокер стоит, основываясь прежде всего на конкретные требования к проектируемому приложению. Стоит отдельно отметить брокер Apache Kafka, так как он имеет крайне высокую пропускную способность, что не помешает высоконагруженному приложению. Помимо этого, Apache Kafka ориентирован на использование в распределённой среде, а именно поддерживает горизонтальное масштабирование, имеет встроенную репликацию своих хранилищ и систему отказоустойчивости [24].

Использование очереди сообщений также может повысить общую отказоустойчивость приложения, так как компонент-отправитель может поместить сообщение в очередь, даже если компонент-получатель недоступен в этот момент. Сообщения будет находиться в очереди пока получатель снова не станет доступным и не получит их [2].

Существуют задачи, в рамках которых за короткий промежуток времени приложению может прийти много запросов на выполнение ресурсоёмких операций, которые при этом нельзя распараллелить между разными компонентами. Примером такой задачи является обработка медиаконтента – когда пользователь быстро даёт несколько последовательных команд по изменению одного такого файла. Ждать выполнения предыдущей команды для того, чтобы дать следующую – не лучший вариант, так как заметно увеличится время отклика для пользователя. Хорошим решением будет как раз класть команды в очередь сообщений. В таком случае пользователю не придётся ждать, прежде чем дать следующую команду, а компонент-обработчик медиафайлов выполнит их все из очереди в нужном порядке [2, 24].

## 2.6 Внешние сервисы

Современные распределенные высоконагруженные приложения обычно активно взаимодействуют с другими программными продуктами по сети, а не работают изолированно. Причиной этому чаще всего становится нецелесообразность или невозможность самостоятельной реализации внутри своего приложения функций, которые уже предлагают другие приложения на рынке. Это отражено в примере с взаимодействием приложений интернет-магазина и банка из 1.1. Ещё более распространённым примером является использование DNS (Domain Name System) для того, чтобы пользователи могли обращаться к приложению по удобному доменному имени. В данном случае DNS выступает в роли внешнего сервиса, который необходимо использовать для повышения удобства пользования приложением [2].

Принимать решения об интеграции в распределённое приложение сторонних сервисов необходимо основываясь на анализе стоящих перед ним задач и опыта решения аналогичных задач авторитетными командами разработчиков. Такой выбор может основываться также и на технологическом стеке приложения, как например для отправки push-уведомлений на устройства с iOS необходимо использовать именно сервис APN.

## 2.7 Выводы к главе

Итак, в данной главе были рассмотрены основные виды архитектурных компонентов распределенных высоконагруженных приложений. Для каждого из них были даны пояснения, сделаны выводы о влиянии на архитектуру, рассмотрены условия и рекомендации по использованию, изучена общая логика функционирования. Теперь можно приступить к анализу требований к распределенным высоконагруженным приложениям и их влиянию на использование рассмотренных в этой главе компонентов.

# 3. ТРЕБОВАНИЯ К РАСПРЕДЕЛЕННЫМ ВЫСОКОНАГРУЖЕННЫМ ПРИЛОЖЕНИЯМ

В данной главе была поставлена задача сфокусироваться на конкретных требованиях заказчиков к разработчикам распределённых приложений и их влиянии на наличие тех или иных архитектурных компонентов, описанных в предыдущей главе. Как правило именно эти требования определяют, будет ли приложение высоконагруженным и имеет ли смысл делать его распределенным. Однако, чтобы не отходить от темы исследования, большее внимание уделяется именно случаям, когда речь идёт о приложениях с такими характеристиками.

Уточнить все желаемые требования к приложению необходимо на этапе подготовки к проектированию её архитектуры. Они делятся на функциональные и нефункциональные. [2]

## 3.1 Нефункциональные требования

Нефункциональные требования определяют, какими свойствами должно обладать приложение. На основании информации, собранной в главах 1 и 2, были выделены основные возможные нефункциональные требования к высоконагруженным приложениям, наличие и значения которых могут значительно сказаться на архитектуре.

К основным возможным нефункциональным требованиям к высоконагруженным распределенным приложениям относятся: масштабируемость, латентность, доступность, производительность (одновременное количество обрабатываемых данных, обслуживаемых пользователей и т.д.), общее количество пользователей, безопасность, отказоустойчивость. Также к ним можно отнести характеристики данных, с которыми приложение будет работать.

Далее рассмотрим каждое из этих требований по отдельности, включая его описание, способы выполнения и влияние на архитектуру.

Важно уточнить, что перечислены не все возможные нефункциональные требования. Данные требования были выбраны на основе информации, собранной в главах 1 и 2. Например, если имеется требование сделать систему масштабируемой, то некоторые компоненты должны будут поддерживать дублирование, а для их управления придётся добавить балансировщик нагрузки.

### 3.1.1 Масштабируемость

Масштабируемость уже обсуждалась в 1.2.4 в контексте целей проектирования распределённых приложений. Сейчас же рассмотрим её как требование заказчика и проанализируем, при помощи каких компонентов из главы 2 его можно достичь.

Вертикальную масштабируемость в рамках данной работы подробно рассматривать не стоит, так как под ней понимается замена текущих вычислительных мощностей на более производительные, а это не должно сильно повлиять на общую архитектуру приложения [10].

Чтобы достичь упростить горизонтальную масштабируемость, стоит разделять бизнес-логику таким образом, чтобы минимизировать зависимость между ними. Такой подход позволит легко добавлять новые экземпляры каждого из них, давая возможность обрабатывать большее число запросов одновременно [25]. Для распределения нагрузки между экземпляры компонентов следует использовать балансировщики нагрузки [10]. Чтобы не конфигурировать вручную добавляющиеся в рамках горизонтального масштабирования экземпляр компонентов, разумно использовать системы обнаружения сервисов, описанные в 2.5.1.

### 3.1.2 Латентность

Латентность – крайне важная характеристика для высоконагруженного распределенного приложения. Говоря о латентности, мы обычно имеем в виду задержку – время между отправкой запроса и получением ответа. Речь не всегда может идти о запросе, который отправляется на сервер, а в принципе о любом действия пользователя, после которого он ожидает увидеть результат.

Обычно требуется достичь как можно более низкого значения латентности, так как долгое ожидание сказывается на желании пользователей пользоваться приложением [11]. Однако, прежде стоит определить, какие требования являются более приоритетными, так некоторые из них взаимосвязаны. К примеру, снижение латентности приложения может негативно сказаться на его надёжности и согласованности, так как быстро переданные данные могут быть не проверены на корректность или быть устаревшими [26].

Рассмотрим значения латентности в наносекундах наиболее типичных компьютерных операций, составленный на основе [2] в таблице 3.1. Как указывает автор источника [2], данные уже довольно устаревшие, но по-прежнему хорошо показывают скорость тех или иных операций относительно друг друга.

Таблица 3.1 – Латентность типичных компьютерных операций

|  |  |
| --- | --- |
| **Операция** | **Латентность, нс** |
| Обращение к основной памяти | 100 |
| Сжатие 1 Кб при помощи Zippy | 10 000 |
| Отправка 2 Кб по сети 1 Гбит/с | 20 000 |
| Последовательное чтение из оперативной памяти 1 Мб | 250 000 |
| Время поиска по диску | 10 000 000 |
| Последовательное чтение 1 Мб из сети | 10 000 000 |

Продолжение таблицы 3.1

|  |  |
| --- | --- |
| Последовательное чтение 1 Мб с диска | 30 000 000 |
| Передача пакета из Калифорнии в Нидерланды и обратно | 150 000 000 |
| Перемещения пакета туда и обратно внутри одного центра обработки данных | 500 000 |

Из представленных данных можно сделать следующие выводы:

* Поиск и чтение данных значительно быстрее при использовании оперативной памяти чем при использовании дисковой.
* Уменьшение расстояния передачи данных значительно снижает латентность.
* Чтение данных из сети осуществляется быстрее, чем чтение из дисковой памяти.

Таким образом, исходя из этих выводов и пунктов 2.3.3, 2.3.4, можно утверждать, что для снижения латентности можно использовать кэши вместе с хранилищами, а также CDN для сокращения расстояния между пользователями и исходными данными. Стоит отметить, что скорость доступа к данным сильно влияет на общую латентность, но важным фактором также является наличие оптимальной с точки зрения скорости бизнес-логики.

### 3.1.3 Доступность

Повысить доступность распределенного приложения означает минимизировать нарушение его работы в случае возникновения сбоев или повышенных нагрузок [27]. Доступность обычно представляют как процент времени исправной работы от общего времени работ. Таким образом, доступность можно рассчитать формуле

где Р – время работы приложения, а П – время его простоя.

Требования доступности как правило начинаются от 99% и растут с количеством девяток после запятой. Это обусловлено тем, что доступность ниже 99% будет означать слишком большое время простоя для современного высоконагруженного распределенного приложения, а 100% достичь невозможно [2]. Соотношение времени простоя и количества девяток после запятой показано в таблице 3.2.

Таблица 3.2 – Соотношение времени простоя и доступности в %

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Доступность, %** | **Суточный простой** | **Недельный простой** | **Месячный простой** | **Годичный простой** |
| 99 | 14,40 м | 1,68 ч | 7,31 ч | 3,65 д |
| 99,9 | 1,44 м | 10,08 м | 43,83 м | 8,77 ч |
| 99,99 | 8,64 с | 1,01 м | 4,38 м | 52,60 м |
| 99,999 | 864,00 мс | 6,05 с | 26,30 с | 5,26 м |
| 99,9999 | 86,40 мс | 604,80 мс | 2,63 с | 31,56 с |

Увеличить доступность распределенного высоконагруженного приложения на этапе проектирования архитектуры можно при помощи репликации (дублирования) важных или часто подвергающихся сбоям компонентов [5]. Важно достичь такой ситуации, чтобы при сбое одного экземпляра компонента его тут же заменял другой. Компоненты используются те же, что и при достижении горизонтальной масштабируемости, но с повышенным вниманием к важным компонентам.

### 3.1.4 Производительность

Когда мы говорим о производительности отдельных вычислительных устройств, мы как правило имеем в виду конкретную величину. Например, в случае с процессорами часто используется количество операций с плавающей запятой или целочисленных в секунду. В случае же с распределенными высоконагруженными приложениями чаще всего имеется в виду количество запросов за момент времени или количество одновременно работающих с приложением пользователей [2, 5].

Если мы знаем конкретные требования к количеству запросов в секунду и числу одновременных пользователей, которые должно поддерживать приложение, необходимо определить (или узнать у заказчика) всю дополнительную информацию, которая позволит нам более-менее точно рассчитать нагрузку разного вида на систему. Чтобы иметь представление о требуемой нагрузке нужно ответы на следующие вопросы:

* Сколько времени и вычислений понадобится для выполнения запросов, которые будет поддерживать приложение?
* С какими данными приложение будет работать?
* Как часто изменяются, добавляются, читаются и удаляются эти данные?
* Как долго следует хранить те или иные данные?
* Каков объем этих данных?

Вопросов может быть и больше, следует опираться прежде всего на функциональные требования [2].

При проектировании архитектуры следует определить, от каких ресурсов приложения зависит производительность, и обеспечить их избыточность. К примеру, если требуется поддержка большого количества веб-клиентов одновременно, то стоит предусмотреть репликацию веб-серверов и добавить для них балансировщик нагрузки [30].

В случае с данными сначала важно сделать выводы о том, как они будут использоваться. В случае частого чтения следует спроектировать хранилище данных с кэшем, в котором будут продублированы наиболее востребованные записи [29]. От использования зависит и выбор типа хранилища (см. 2.3).

### 3.1.5 Общее количество пользователей

Важно не только количество одномоментных пользователей, которое, как выяснилось, связано с производительностью, но и общее количество пользователей приложения. В данном случае имеется в виду, что на моменте проектирования архитектуры важно иметь представление о максимальном количестве пользователей либо количестве пользователей через некое продолжительное время эксплуатации [2].

Далее можно сделать предположение об объеме данных, который должно будет хранить приложение. При этом важно определить, какие данные, как часто и на какое время смогут сохранять их пользователи. На основе полученной информации уже следует сделать выводы о количестве и типе хранилищ.

### 3.1.6 Безопасность

Требования к безопасности в распределенных системах часто высоки. Как правило это связано с тем, что система может хранить или передавать чувствительные данные, которые не должны попасть к злоумышленникам или исчезнуть. При этом уровень обеспечения безопасности приложения сказывается и на показателях его доступности. Даже если распределенное приложение не содержит чувствительных данных, оно может быть подвержено атакам на инфраструктуру, которые могут частично или полностью вывести его из строя на продолжительное время.

Не вдеваясь в подробности реализации безопасности внутри каждого компонента приложения, выделим основные способы её обеспечения для распределённых высоконагруженных приложений в целом:

* Так как в распределенных системах компоненты постоянно обмениваются между собой данными, важно использовать средства установления безопасных каналов передачи этих данных [5].
* Должно быть реализовано логическое разграничение доступа между компонентами приложения [28]. Это позволит чётко определить, каким компонентам доступны те или иные данные.
* Стоит задуматься об организации общей схемы безопасности для всех компонентов приложения [28]. Это упросит его настройку, конфигурацию и изменение.
* Данные между компонентами хорошо передавать в зашифрованном виде. Для этого можно использовать подходящие криптографические алгоритмы [5, 28]. Скрывать стоит и физическое местоположение компонентов. Это поможет обеспечить прозрачность распределений (она описана в 1.2.2).
* Важно обеспечить безопасность хранилищ данных, причём как от так злоумышленников, так и от технических сбоев. Для этого можно использовать контроль доступа к данным, шифрование и репликацию.
* При надобности организации авторизации и аутентификации пользователей в приложении, полагаясь на стиль микросервисной архитектуры, можно вынести эту функцию в отдельный сервис [2, 19, 20].

### 3.1.7 Отказоустойчивость

Отказоустойчивость приложения предполагает выполнение заявленных функций даже при возникновении сбоев [5]. Такая система также должна корректно обрабатывать возникающие ошибки.

Как и в случае с доступностью и масштабируемостью, отказоустойчивость можно повысить при помощи репликации (дублирования) ключевых компонентов. Кроме того, чтобы повысить отказоустойчивость компонента, стоит спроектировать его не имеющим состояния. Если экземпляр компонента, хранящего состояние, выйдет из строя, данные в этом состоянии с большой вероятностью потеряются [2, 5].

## 3.2 Функциональные требования

Функциональные требования чётко определяют, какие конкретные задачи должно решать приложение. Они обычно включают в себя пользовательские и бизнес-требования.

Под бизнес-требованиями понимаются задачи, которые ставит перед собой бизнес, частью которого будет данное приложение. Пользовательские функциональные требования же как правило включают в себя описание сценариев, согласно которым пользователи будут использовать приложение [31].

Без чётко сформулированных функциональных требований невозможно спроектировать распределённое приложение, так как именно с их помощью определяется, какие основные компоненты должны быть в нём и в какой последовательности они должны взаимодействовать [2, 4, 5, 31].

К примеру, функциональное требование может быть сформулировано следующим образом: приложение должно принимать текстовое сообщение в запросе пользователя, сохранять его и отправлять другому пользователю. Из этого примера можно понять, что нам как минимум потребуется веб-сервер для обработки запросов пользователя, база данных для хранения сообщений, сервис бизнес-логики для взаимодействия с базой данных и сервис отправки сообщений другим пользователям.

## 3.3 Выводы к главе

Итак, в данной главе были рассмотрены функциональные и нефункциональные требования к распределенным высоконагруженным приложениям. Из нефункциональных требований рассматривались несколько наиболее значимых на этапе проектирования, для каждого из них было дано пояснение и предложены варианты выполнения. Для функциональных требований была указана их роль в архитектуре и приведён пример.

# 4. ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ

В главах 1, 2 и 3 были рассмотрены теоретические основы проектирования архитектуры распределённых высоконагруженных приложений, а поскольку одной из основных задач данной работы является разработка предметно-ориентированного языка (DSL) для автоматизации этого процесса, в данной главе было решено исследовать общие принципы и подходы к реализации таких языков программирования. Также задачей данной главы является рассмотрение существующих реализаций DSL и их достижений в решении поставленных перед ними задач.

## Общие сведения о DSL

Т

## 4.8 Анализ реализаций при помощи формул

Докажем теорему о производных элементарных функций. Пускай имеется некая элементарная функция

и её производная

Тогда производная (2) функции (1) вычисляется как

Заметим, что функция (2) со знаком предела зависит от двух переменных:

Сама элементарная функция (1) и её производная (2) зависит только от одной переменной. Функция, определяемая по формуле (4), зависит от двух переменных. Но при вычислении производной, мы считаем, что переменная *x* принимает произвольное фиксированное значение, принадлежащее области определения функции. То есть мы считаем, что *x* является постоянной. Таким образом мы вычисляем предел от функции одной переменной:

где *x* – постоянная величина.

## 4.20 Анализ опыта разработки схожих продуктов

Общий текст про изучение реализаций и опыта проектирования и реализации предметно ориентированных языков. Общий текст про изучение реализаций и опыта проектирования и реализации предметно ориентированных языков. Общий текст про изучение реализаций и опыта проектирования и реализации предметно ориентированных языков. Общий текст про изучение реализаций и опыта проектирования и реализации предметно ориентированных языков.

# 5. РАЗРАБОТКА СЕМАНТИКИ И СИНАКСИСА

## 5.1 Подзаголовок

Т

## 5.2 Подзаголовок

Т

# 6. РАЗРАБОТКА ОСНОВНОЙ БИБЛИОТЕКИ

## 6.1 Подзаголовок

Т

## 6.2 Подзаголовок

Т

# 7. РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ

В данной главе рассматриваются подробности реализации клиентского веб-приложения. В начале разработки это части общей системы для генерации распределенной архитектуры ставились следующие цели:

* Дать пользователям возможность использовать спроектированный DSL напрямую из браузера без необходимости скачивать какое-либо ПО.
* Организовать написание программ на языке при помощи визуального интерфейса, а не только про помощи стокового ввода программы.

Разработка веб-приложения была разделена на 2 отдельных этапа: разработку фронтенда для пользовательского интерфейса и бэкенда для выполнения основной логики. По итогу оно должно соответствовать клиент-серверной архитектуре.

## 7.1 Фронтенд

Исходя из поставленных целей, было принято решение реализовать фронтенд в виде клиентского приложения, которое будет позволять составлять программы на спроектированном DSL при помощи элементов графического интерфейса, отправлять полученные программы на бэкенд, после чего визуализировать результат.

### 7.1.1 Выбор технологий

Для реализации фронтенда был выбран фреймворк React TypeScript. Он хорошо подходит для решения задач разработки сложных визуальных интерфейсов, позволяет писать структурированный код, так как предлагает создание интерфейса из отдельных компонентов, которые при этом можно использовать многократно. Также существует большое количество библиотек этих компонентов для React TypeScript, что позволяет создавать хорошо выглядящие интерфейсы за короткий срок.

На этапе выбора технологий стоял выбор между React JavaScript и React TypeScript, но был выбран последний, так как он основан на языке TypeScript, который имеет преимущества перед JavaScript. Причиной его выбора стала более строгая типизация переменных, которая при этом имеет статический характер. Это позволяет легче избегать ошибок, связанные с типами данных, при написании кода. JavaScript же является слабо типизированным языком с динамической типизацией.

Чтобы ускорить разработку фронтенда было принято решение использовать одну из библиотек компонентов, которой стала Material UI. Она предлагает большое количество готовых компонентов, таких как различные поля для ввода, контейнеры элементов, стили, иконки и т.д. В данной библиотеке при это довольно большие возможности по кастомизации готовых компонентов. Также стоит отметить, что она на данный момент хорошо поддерживается и имеет удобную документацию с примерами конфигурации и использования.

Для отправки запросов на сервер в разработанном приложении используется протокол HTTP – стандартный протокол для передачи гипертекста. Запросы формируются и отправляются в соответствии с архитектурным стилем REST (Representational State Transfer), который предлагает ряд правил и ограничений для создания запросов и интерфейсов прикладного программирования (API).

### 7.1.2 Делали реализации

Т

## 7.2 Бэкенд

Бэкенд был разработан для обработки пользовательских запросов, в которых содержится код на языке. Основная логика обработки запросов крайне проста: программа принимает от клиента код на DSL, вызывает необходимые функции из основной библиотеки DSL и возвращает пользователю результат. Результатом может быть либо текстовое описание UML, либо сообщение об ошибке.

### 7.2.1 Выбор технологий

Для практической реализации был выбран фреймворк Kotlin Spring Boot. В качестве языка, на котором будет написано приложение, был выбран именно Kotlin, а не Java, так как его синтаксис более лаконичный и выразительный. Также Kotlin имеет систему null-безопасности типов, что помогает обнаружить возможные случаи возникновения NullPointerException во время написания программы.

Сам фреймворк Spring Boot хорошо подходит для задачи создания

### 7.2.2 Делали реализации

Т

# 8. ТЕСТИРОВАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ

## 8.1 Подзаголовок

Т

## 8.2 Подзаголовок

Т

# ЗАКЛЮЧЕНИЕ

Ваш текст заключения к дипломной работе. Ваш текст заключения к дипломной работе. Ваш текст заключения к дипломной работе. Ваш текст заключения к дипломной работе. Ваш текст заключения к дипломной работе. Ваш текст заключения к дипломной работе. Ваш текст заключения к дипломной работе.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Распределенные системы. Паттерны проектирования. / Брендан Бёрнс. – Санкт-Петербург : Питер, 2019. – 224 с.
2. System Design. Подготовка к сложному интервью. / Сюй Алекс. – Санкт-Петербург : Питер, 2025. – 304 с.d
3. Белов Владимир Федорович, Гаврюшин Сергей Сергеевич, Занкин Александр Иванович, Исайкин Виталий Юрьевич. Математическая модель архитектуры комплекса средств распределенного проектирования //Математическое моделирование и численные методы. – 2024. – №. 1 (41). – С. 110-123
4. Гома Х. UML. Проектирование систем реального времени, распределенных и параллельных приложений. / Гома Хассан – Москва: Litres, 2022. – 700 с.
5. Распределенные системы. / Мартен ван Стин, Эндрю С. Таненбаум. – Москва : ДМК Пресс, 2021. – 584 с.
6. Чахкиев М. Т. ОСНОВНЫЕ ОСОБЕННОСТИ АРХИТЕКТУРЫ ВЫСОКОНАГРУЖЕННЫХ СИСТЕМ. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ //Точная наука. – 2021. – №. 98. – С. 19-21.
7. Сухов А. В. DSL-обзорная статья //ББК 1 Н 34. – С. 6085.
8. Лещев С. А., Крюкова В. В. ПОДХОДЫ К РЕШЕНИЮ ПРОБЛЕМ ПРОЗРАЧНОСТИ РАСПРЕДЕЛЕННЫХ ИНФОРМАЦИОННЫХ СИСТЕМ //РОССИЯ МОЛОДАЯ. – 2020. – С. 21171.1-21171.5.
9. Ильичов Д. Э., Лысцов Н. А., Карлыганов А. Д. ОБЗОР ИЗВЕСТНЫХ РАСПРЕДЕЛЕННЫХ СИСТЕМ ХРАНЕНИЯ ДАННЫХ //Science and education: problems and innovations. – 2020. – С. 17-19.
10. Федорий Д. А. ОБЕСПЕЧЕНИЕ МАСШТАБИРУЕМОСТИ И ОТКАЗОУСТОЙЧИВОCТЬ РАЗРАБАТЫВАЕМОЙ СИСТЕМЫ //ФУНДАМЕНТАЛЬНЫЕ И ПРИКЛАДНЫЕ ИССЛЕДОВАНИЯ В НАУКЕ И ОБРАЗОВАНИИ. – 2021. – С. 51-55.
11. Yu M. et al. Unravelling the relationship between response time and user experience in mobile applications //Internet Research. – 2020. – Т. 30. – №. 5. – С. 1353-1382.
12. Дронов Р. О. Особенности выбора архитектуры и компонентов для построения распределенных приложений //Информатика: проблемы, методы, технологии. – 2020. – С. 1246-1250.
13. Бахмутов А. Д. ВНУТРЕННЕЕ УСТРОЙСТВО СОВРЕМЕННЫХ ВЕБ-СЕРВЕРОВ //Международная научно-техническая конференция молодых ученых БГТУ им. ВГ Шухова, посвященная 170-летию со дня рождения ВГ Шухова. – 2023. – С. 67-71.
14. Дзюбин И. А., Малышев А. В. СПОСОБЫ ОПТИМИЗАЦИИ ВЫСОКОНАГРУЖЕННЫХ ПРИЛОЖЕНИЙ //Программная инженерия: современные тенденции развития и применения (ПИ-2021). – 2021. – С. 12-20.
15. Ушанов А. Р. Реляционные и нереляционные базы данных //Молодой ученый. – 2021. – №. 23. – С. 111-114.
16. Раджабов А. Р. ТИПЫ БАЗ ДАННЫХ //Introduction of new innovative technologies in education of pedagogy and psychology. – 2024. – Т. 1. – №. 3. – С. 204-210.
17. Шкрябин Г. Д. СТРАТЕГИИ КЭШИРОВАНИЯ ДЛЯ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ //Вестник науки. – 2024. – Т. 1. – №. 12 (81). – С. 1220-1231.
18. Zolfaghari B. et al. Content delivery networks: State of the art, trends, and future roadmap //ACM Computing Surveys (CSUR). – 2020. – Т. 53. – №. 2. – С. 1-34.
19. Караханова А. А. Анализ микросервисной архитектуры, монолитных приложений, архитектуры soa //Синергия наук. – 2020. – №. 46. – С. 255-262.
20. Кравченко Д. А. Микросервисная архитектура //Интерактивная наука. – 2022. – Т. 4. – №. 69. – С. 43.
21. Kysow L. Consul: Up and Running. – " O'Reilly Media, Inc.", 2022.
22. Нехаев А. П., Громова Г. М. ПРОБЛЕМА БАЛАНСИРОВКИ НАГРУЗКИ В РАСПРЕДЕЛЕННОЙ ИНФОРМАЦИОННОЙ СИСТЕМЕ //РОССИЯ МОЛОДАЯ. – 2020. – С. 21152.1-21152.5.
23. КХАИНГ М., Лупин С. А., Тху А. Оценка эффективности методов балансировки нагрузки в распределенных вычислительных системах //International Journal of Open Information Technologies. – 2021. – Т. 9. – №. 11. – С. 30-36.
24. Якунович, А. В. Брокеры сообщений / А. В. Якунович, А. С. Север, Ю. А. Чистякова // Информационные технологии. Физика и математика : материалы 87-й научно-технической конференции профессорско-преподавательского состава, научных сотрудников и аспирантов, Минск, 31 января - 17 февраля 2023 г. - Минск : БГТУ, 2023. – С. 79-82.
25. Кузнецов И. А. Оптимизация распределенных систем для мобильных приложений: улучшение производительности и масштабируемости //Инновационная наука. – 2024. – №. 5-1. – С. 52-57.
26. Yu D. et al. Low reliable and low latency communications for mission critical distributed industrial internet of things //IEEE Communications Letters. – 2020. – Т. 25. – №. 1. – С. 313-317.
27. Злыгостев А. И., Пищик Б. Н. О количественных метриках согласованности и доступности в распределённых базах данных //Вестник Новосибирского государственного университета. Серия: Информационные технологии. – 2022. – Т. 20. – №. 2. – С. 18-26.
28. Кругликов С. В. и др. Информационная безопасность информационных систем с элементами централизации и децентрализации //Вопросы кибербезопасности. – 2020. – №. 1 (35). – С. 2-7.
29. Невмержицкий А. А. Использование локального кэширования для увеличения производительности распределенной системы //Вестник науки и образования. – 2020. – №. 11-3 (89). – С. 14-17.
30. Гридин В. Н., Анисимов В. И., Васильев С. А. Методы повышения производительности современных веб-приложений //Известия Южного федерального университета. Технические науки. – 2020. – №. 2 (212). – С. 193-200.
31. Гумерова Г. Р., Мансурова Т. Г. Моделирование требований к программному обеспечению //Вестник Алтайской академии экономики и права. – 2023. – №. 12-1. – С. 42-52.
32. Индексирование фундаментальных научных направлений кодами информационных классификаций УДК / О.А. Антошкова, Т.С. Астахова, В.Н. Белоозеров и др.; под ред. акад. Ю.М. Арского. – М., 2010. – 322 с.

# ПРИЛОЖЕНИЕ А

Листинг 1 – код компилятора

|  |
| --- |
| import java.util.Scanner;  public class HelloWorld {  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  System.out.print("Введите ваше имя: ");  String name = scanner.nextLine();  System.out.println("Hello, " + name + "!");  scanner.close();  }  } |