МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра технологий программирования

МИХАЙЛОВ
Антон Артурович

РАЗРАБОТКА И РАЗВЁРТЫВАНИЕ СЕРВЕРНОЙ ЧАСТИ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ НА МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ

Дипломная работа

Научный руководитель:
старший преподаватель,
Н.А. Карпович

Допущена к защите

«____»_____2021 г.

Зав. кафедрой технологий программирования
доктор технических наук, профессор,
Заслуженный деятель науки
Республики Беларусь А.Н. Курбацкий

Минск, 2021

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

SOA	сервис-ориентированная архитектура (Service-oriented Architecture),
SOAP	простой протокол доступа к объектам (Simple Object Access Protocol),
REST	передача состояния представления (Representational State Transfer),
CI	непрерывная интеграция (Continuous Integration),
CD	непрерывное развёртывание (Continuous Deployment),
DDD	предметно-ориентированное программирование (Domain-driven Design),
ESB	сервисная шина предприятия (Enterprise Service Bus),
JWT	JSON Web Token,
RPC	удалённый вызов процедуры (Remote Protocol Call),
REST	передача состояния представления (Representational State Transfer).
CNCF	Cloud Native Computing Foundation,
UML	унифицированный язык моделирования (Unified Modeling Language).
AWS	Amazon Web Services,
GCP	Google Cloud Platform,
MVP	минимально жизнеспособный продукт (Minimum Valuable Product),
ORM	объектно-реляционное отображение (Object-Relational Mapping),
HTTP	протокол передачи гипертекста (HyperText Transfer Protocol),
JSON	JavaScript Object Notation,
URL	унифицированный указатель ресурса (Uniform Resource Locator),
IDL	язык описания интерфейсов (Interface Definition Language).
SDK	набор средств разработки (Software Development Kit),

GKE	Google Kubernetes Engine,
IaaS	инфраструктура как услуга (Infrastructure as a Service),
PaaS	платформа как услуга (Platform as a Service),
SaaS	программное обеспечение как услуга (Software as a Service),

Реферат

Дипломная работа, 76 с., 52 рис., 5 формул, 30 источников, 15 приложений.

Ключевые слова: МИКРОСЕРВИСЫ, GOLANG, KUBERNETES, DOCKER, gRPC, МНОГОПОЛЬЗОВАТЕЛЬСКАЯ ИГРА, AGONES, OPENMATCH, GORM, ОБНАРУЖЕНИЕ СТОЛКНОВЕНИЙ.

Предмет исследования — микросервисная архитектура, серверы многопользовательских игр.

Объект исследования — объектом исследования являются разработка и развёртывание серверной части многопользовательской игры, платформы Agones и OpenMatch, контейнеризация при помощи Docker и развёртывание микросервисных приложений на Kubernetes.

Цели работы — разработать серверную часть игры проекта Medieval.IO на микросервисной архитектуре, используя технологии, изученные в ходе прохождения производственной и преддипломной практик, и развёрнуть её на облачном сервисе для интеграции с мобильным и настольным клиентами.

Методы исследования — а) теоретические: изучение литературы, посвященной проектированию микросервисных приложений, проектированию, разработке и развёртыванию серверов многопользовательских игр; б) практические: проектирование архитектуры серверной части игры, разработка микросервисов на языке программирования Golang с использованием фреймворка gRPC, развёртывание серверной части при помощи услуги Google Kubernetes Engine облачной платформы Google Cloud Platform.

Результатами являются — динамически масштабируемый в зависимости от нагрузки сервер игры, развёрнутый в глобальной сети и доступный для интеграции с клиентом игры.

Область применения — разработка динамически масштабируемых серверов игр/веб-приложений.

Рэферат

Дыпломная праца, 76 ст., 52 мал., 5 формул, 30 крыніц, 15 дадаткаў.

Ключавыя словы: МІКРАСЭРВІСЫ, GOLANG, KUBERNETES, DOCKER, gRPC, ШМАТКАРЫСТАЛЬНІЦКАЯ ГУЛЬНЯ, AGONES, OPENMATCH, GORM, ВЫЯЎЛЕННЕ СУТЫКНЕННЯЎ.

Прадмет даследавання — мікрасэрвісная архітэктура, серверы шматкарыстальніцкіх гульняў.

Аб'ект даследавання — аб'ектам даследавання з'яўляюцца распрацоўка і разгортванне сервернай часткі шматкарыстальніцкай гульні, платформы Agones і OpenMatch, контейнеризация пры дапамозе Docker і разгортванне мікрасэрвісных прыкладанняў на Kubernetes.

Мэты працы — распрацаваць серверную частку гульні праекта Medieval.IO на мікрасэрвіснай архітэктуры, выкарыстоўваючы тэхналогіі, вывучаныя ў ходзе праходжання вытворчай і пераддыпломнай практык, і разгарнуць яе на воблачным сэрвісе для інтэграцыі з мабільным і настольным кліентамі.

Метады даследавання — а) тэарэтычныя: вывучэнне літаратуры, прысвечанай праектаванню мікрасэрвісных прыкладанняў, праектаванню, распрацоўцы і разгортванню сервераў шматкарыстальніцкіх гульняў; б) практычныя: праектаванне архітэктуры сервернай часткі гульні, распрацоўка мікрасэрвісаў на мове праграммавання Golang з выкарыстаннем фрэймворка gRPC, разгортванне сервернай часткі пры дапамозе паслугі Google Kubernetes Engine воблачнай платформы Google Cloud Platform.

Вынікамі з'яўляюцца — дынамічна маштабуемы ў залежнасці ад нагрузкі сервер гульні, разгорнуты ў глабальнай сетцы і даступны для інтэграцыі з кліентам гульні.

Вобласць ужывання — распрацоўка дынамічна маштабуемых сервераў гульняў / вэб-прыкладанняў.

Abstract

Diploma, 76 p., 52 illustrations, 5 formulas, 30 sources, 15 appendixes.

Keywords: MICROSERVICES, GOLANG, KUBERNETES, DOCKER, gRPC, MULTIPLAYER, AGONES, OPENMATCH, GORM, COLLISION DETECTION.

Research subjects are microservice architecture, multiplayer game servers.

The objects of research are the development and deployment of the server side of a multiplayer game, the Agones and OpenMatch platforms, containerization using Docker and the deployment of microservice applications on Kubernetes.

The purpose is to develop the server part of the game of the Medieval.IO project on a microservice architecture, using the technologies studied during the production and pre-graduation practices, and deploy it on a cloud service for integration with mobile and desktop clients.

Methods of research are a) theoretical methods: a study of literature on the design of microservice applications, design, development and deployment of servers for multiplayer games; b) practical methods: designing the architecture of the server side of the game, developing microservices in the Golang language using the gRPC framework, deploying the server side using the Google Kubernetes Engine service of the Google Cloud Platform.

The result is a dynamically scalable depending on the load game server deployed in the global network and available for integration with the game client.

Scope is development of dynamically scalable game / web application servers.

ВВЕДЕНИЕ

У многих современных многопользовательских и онлайн игр достаточно часто встречаются одни и те же проблемы: проблемы с реагированием на возрастающий поток игроков, приводящий к нехватке или количества доступных серверов, или их мощностей и продолжительные обслуживания при обновлении версии игр или исправлении каких-то проблем и багов, при которых игроки не имеют доступа к игре.

Особенно критическими данные проблемы являются для новых компаний, у которых либо не было проектов до этого, либо их предыдущие проекты не принесли им стабильной базы игроков до выпуска нового проекта. Невозможно точно просчитать сколько игроков появится в первые моменты запуска игры, какой будет пик активности, как будет изменяться количество игроков в течение первой недели или месяца. Поэтому возможность динамического масштабирования серверов без продолжительного обслуживания критически необходима для предоставления как можно лучшего опыта для игроков даже во время пика активности. Без данной возможности будут появляться проблемы как с организацией игр, так и с самим процессом игры, вследствие этого начнётся отток игроков и проект может уйти в небытие практически сразу.

Явными примерами последних двух лет являются такие игры как Among Us и Outriders. Among Us был выпущен ещё в 2018 году и основывается на известной игре «Мафия». Однако бум популярности этой игры произошёл именно в конце 2020 года. Как только в игру нахлынули миллионы игроков, серверы испытывали большие сложности с обработкой такого количества игроков. Иногда для участия в игре или создания новой сессии игрокам приходилось сидеть по 20 минут, нажимая на одну и ту же кнопку с надеждой не получить сообщение об ошибке в ответ. Outriders в свою очередь была выпущена в апреле 2021 года и в течении первой недели также испытывала множество проблем, вплоть до полной недоступности серверов в течении несколько часов.

Хоть данные проблемы могли быть вызваны и ошибками в логике сервера или в невозможности выделить новые сервера в силу отсутствия дополнительных аппаратных ресурсов, следует попробовать исключить любые другие возможные проблемы, которые могут произойти. Одной из таких проблем является создание достаточного количества серверов в зависимости от нагрузки на игру и количества пользователей, находящихся в игре одновременно.

Данная работа представляет собой попытку решить данную проблему при помощи технологий, изученных в ходе прохождения производственной и преддипломной практик, а также знаний, приобретённых в ходе обучения в

университете.

При анализе возможных решений и поиске существующих технологий был использован один важный критерий – все платформы должны иметь в своей основе платформу Kubernetes, в силу того, что данная технология имеет открытый исходный код и входит в CNCF – проект, который занимается разработкой и поддержкой основных компонент современной облачной инфраструктуры.

В главе 1 описана микросервисная архитектура, её отличия от монолитной архитектуры и SOA, основные подходы и шаблоны, применимые к поставленной задаче, а также фреймворк gRPC, широко распространённый в облачных приложениях.

В главе 2 описаны принципы работы серверной части и её общения с клиентом, рассмотрены подходы для решения проблем, связанных с задержкой при передаче информации между клиентом и сервером, а также жульничеством при помощи модификации клиента.

В главе 3 проведено проектирование и разработка всех компонентов серверной части, а также описаны алгоритмы обнаружения коллизий (или столкновений) для обработки состояния сессии игры полностью на стороне сервера.

В главе 4 проведена контейнеризация всех микросервисов серверной части игры и их развёртывание, используя платформы Agones, OpenMatch и Kubernetes, на облачной платформе Google Cloud Platform.

ГЛАВА 1. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА

1.1. Общий обзор микросервисной архитектуры

Микросервисная архитектура (рис. 1.1) предполагает набор слабо зацепленных сервисов. В данной архитектуре сервисы имеют мелкую гранулярность (fine-grained, то есть каждый сервис предназначен для выполнения какой-то конкретной узкой задачи) и протоколы общения между сервисами являются легковесными (например, HTTP).

Впервые данная архитектура была упомянута в 2005 году Питером Роджерсом на конференции Web Services Edge. В то время стандартом являлся SOAP SOA для подобных применений, но Питер предложил использовать REST-сервисы, которые он назвал «микро-веб-сервисами». Вместо использования достаточно сложного и тяжеловесного SOAP он предложил сервисам вызывать другие сервисы по REST протоколу, то есть абстрагироваться от логики сервисов, скрывая её под простыми URI интерфейсами. Официально название «микросервис» впервые прозвучало на ежегодном семинаре архитекторов программного обеспечения, который проходил в 2011 году в Венеции. Чуть позже, в 2012 году, в Кракове появились публикации про «гранулярный SOA», которые рассматривали основы микросервисной архитектуры.

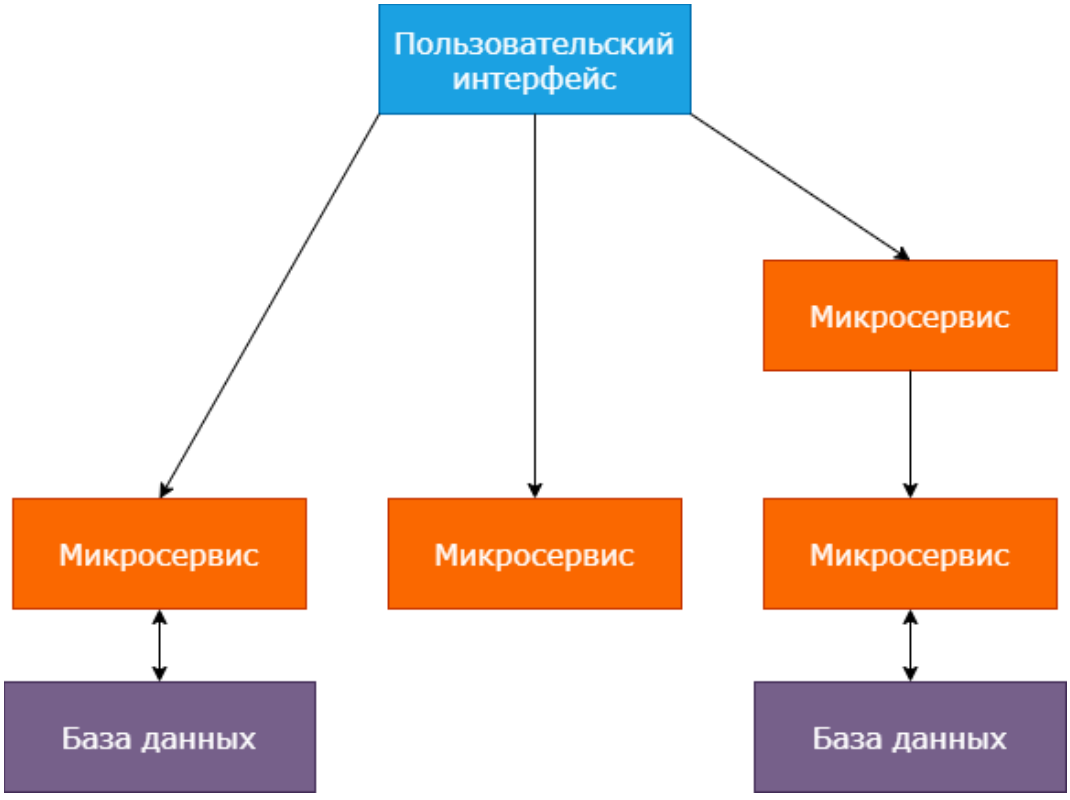


Рисунок 1.1 – Схематический пример микросервисной архитектуры

Данная архитектура широко используется для облачных приложений и бессерверных вычислений.

1.1.1. Что такое микросервис?

На самом деле определение понятия «микросервис» является достаточно размытым. Определение, данное выше, про сервисы, имеющие мелкую гранулярность и протоколы общения между которыми являются легковесными, также не отображает полной картины функционала и особенностей таких сервисов. Наиболее часто упоминаемыми характеристиками данного типа сервисов являются:

- микросервисы – это процессы, которые взаимодействуют по сети, используя легковесные протоколы такие как HTTP;
- развёртываются независимо друг от друга;
- проектируются и разрабатываются в соответствии с определёнными бизнес-требованиями и бизнес-возможностями;
- могут быть разработаны, используя различные языки программирования, базы данных и могут быть развёрнуты в различных средах окружения;
- разрабатываются относительно небольшими командами из 5-12 человек.

Обладая такими характеристиками, микросервисы идеально подходят под принципы CD и CI, так как изменения малой части кодовой базы не приводят к полному переразвёртыванию приложения, а только его небольшой части. Также микросервисы соответсвуют философии Unix «Do one thing and do it well» (делай одну вещь и делай её хорошо). В этом плане микросервисы делают возможным DDD подход к разработке, который предполагает разделение предметной области задачи на небольшие контексты, построение моделей для данных контекстов и реализацию программного обеспечения по готовым моделям в небольших командах.

1.2. Сравнение с другими архитектурами программного обеспечения

Космические корабли, бороздящие просторы Вселенной, это конечно хорошо, но всё познаётся в сравнении. Для того чтобы сполна понять возможности, плюсы и минусы микросервисной архитектуры требуется её сравнение с существующими альтернативами.

1.2.1. Микросервисная архитектура vs монолитная архитектура

Монолитная архитектура (рис. 1.2) считается традиционной при создании каких-либо новых проектов. Всё приложение является неразделимым, является одним целым. Такая архитектура предполагает одну большую кодовую базу.

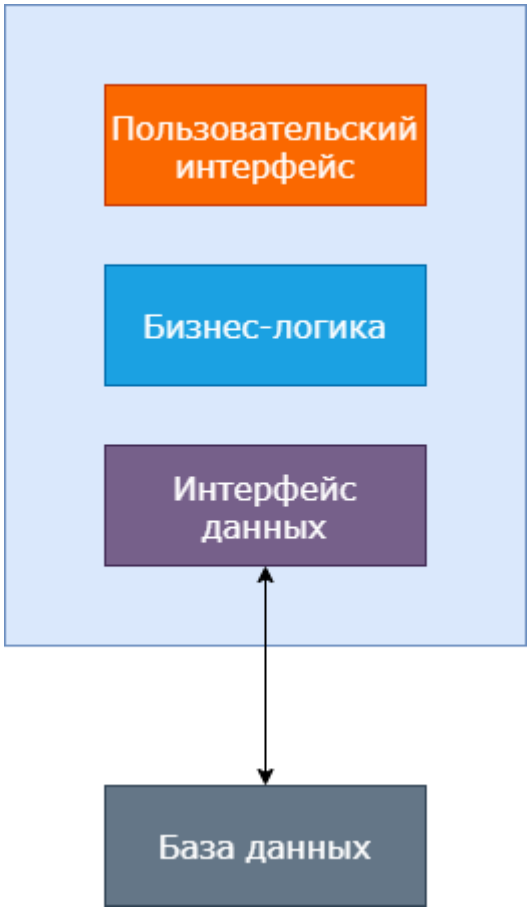


Рисунок 1.2 – Схематический пример монолитной архитектуры

Плюсы монолитной архитектуры:

- легче организовать логгирование, кэширование и мониторинг производительности, так как приложение всего одно и не имеет внешних взаимодействий с её частями (как например у микросервисной архитектуры);
- выявление проблем и тестирование значительно проще за счёт того, что сквозные (end-to-end) тесты пишутся и прогоняются намного быстрее, чем в микросервисной архитектуре;
- развёртывание приложения с монолитной архитектурой намного проще, нужно позаботиться только об одном приложении и не нужно постоянно держать в голове возможные взаимодействия данного приложения, так как оно скорее всего будет иметь один большой интерфейс для взаимодействия с ним;

- простая для понимания и использования. Любой начинающий программист сможет написать небольшое монолитное приложение, но далеко не каждый сможет понять и написать приложение с микросервисной архитектурой.

Минусы монолитной архитектуры:

- из-за постоянного увеличения кодовой базы с ростом проекта полное понимание проекта его разработчиками становится почти невозможным. Любое изменение кода должно быть тщательно просмотрено на возможность влияния на другие части приложения, из-за чего процесс разработки становится очень медленным;
- масштабирование для данной архитектуры возможно только в плане всего приложения сразу, что является действительно большой проблемой с точки зрения производительности и возможных «бутылочных горлышек»;
- переход на новые технологии или их добавление является невероятно сложным и трудоёмким, так как в таких ситуациях приложение должно будет быть переписано почти полностью.

Как уже было описано в предыдущей части, микросервисная архитектура разбивает необходимый функционал на части и разрабатывает каждую из них как самостоятельный сервис.

Плюсы микросервисной архитектуры:

- компоненты приложения являются слабо зацепленными, они являются при идеальном разделении бизнес-логики на части независимыми друг от друга. Проблемы с одним из микросервисов не выльется в полную недоступность функционала, как это возможно в случае с монолитом;
- так как кодовая база разделена на части и над каждой из них работает отдельная команда, члены данной команды хорошо разбираются в своей части приложения. Намного проще знакомить с проектом новых разработчиков, они способны работать с кодом практически с первого дня;
- масштабирование каждого из компонентов можно производить отдельно от всех остальных, что позволяет легко избавиться от проблемы «бутылочного горлышка» с оптимальным использованием дополнительных ресурсов;
- технологию, которую выбрали для использования на старте, намного проще заменить на что-то новое. Каждая команда к тому же может экспериментировать в области своего компонента приложения без необходимости менять всё приложение целиком.

Как можно видеть, у каждой из архитектур есть свои сильные и слабые

стороны. Но, как можно заметить на примере таких компаний как Netflix, Google и Amazon, микросервисная архитектура в каком-то смысле является продолжением монолитной в том смысле, что большая часть любых проектов стартует с монолитной архитектурой, но в ходе роста проекта постепенно мигрирует на микросервисную архитектуру [3].

Признаки того, что монолитная архитектура больше не подходит проекту и необходима миграция на микросервисную архитектуру (рис. 1.3):

- члены команды больше не могут свободно ориентироваться по проекту и его кодовой базе;
- новых членов команды сложно ввести в проект, необходимо некоторое время для того, чтобы новичок разобрался перед тем, как он сможет добавлять новый функционал;
- рефакторинг кода превращается в очень большую головную боль;
- самый важный критерий – стоимость добавления нового функционала превышает предполагаемую прибыль от него.

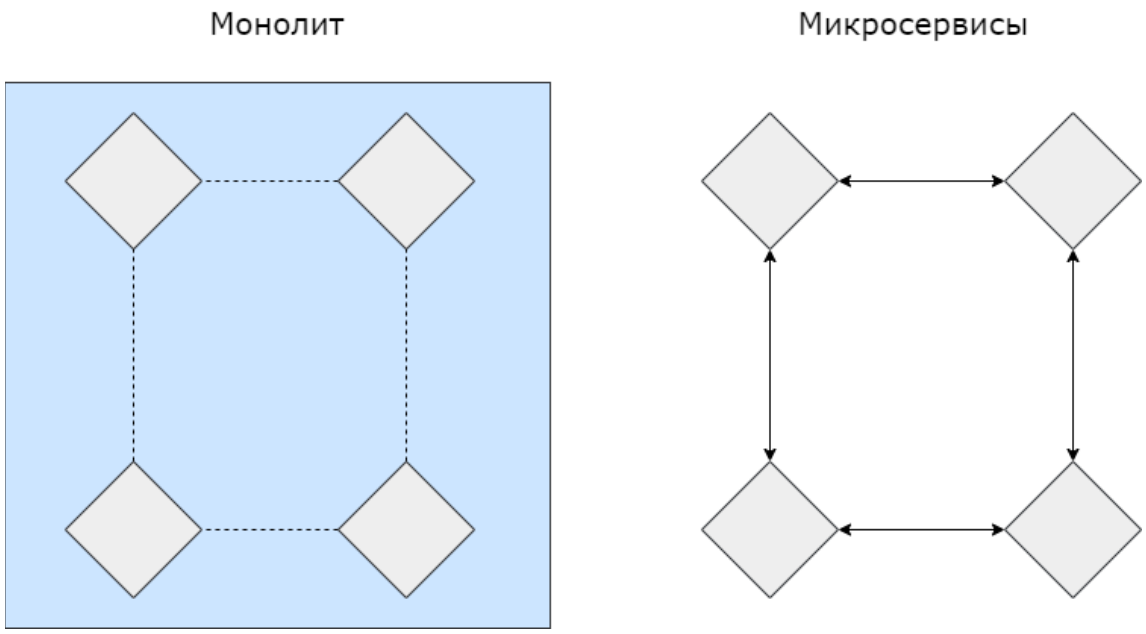


Рисунок 1.3 – Схематический переход от монолитной архитектуры к микросервисной

1.2.2. Микросервисная архитектура vs SOA

С первого взгляда может показаться, что микросервисная архитектура и SOA одно и то же: и там, и там сервисы, предназначенные для выполнения конкретной бизнес-функции, эти сервисы являются слабо зацепленными. Однако всё же между ними есть достаточно тонкая разница. Тонкая, но в то же время достаточно важная, так как эти две архитектуры нельзя воспринимать как

всегда взаимоисключающие, в некоторых случаях они могут даже дополнять друг друга (рис 1.4).

Первая и самая большая разница – это сфера применения. Когда идёт речь про SOA идёт речь про уровень предприятия, так как в SOA одним из важных принципов является повторное использования уже готовых компонент в нескольких приложениях, которое данное предприятие разрабатывает, в то время как микросервисная архитектура подразумевает уровень приложения, где достаточно часто происходит дупликация кода для уменьшения зависимостей и взаимодействий между микросервисами. К тому же микросервисы могут не иметь ничего общего в их разработке, так как для каждого из микросервисов могут применяться абсолютно разные как технологии, так и языки программирования.

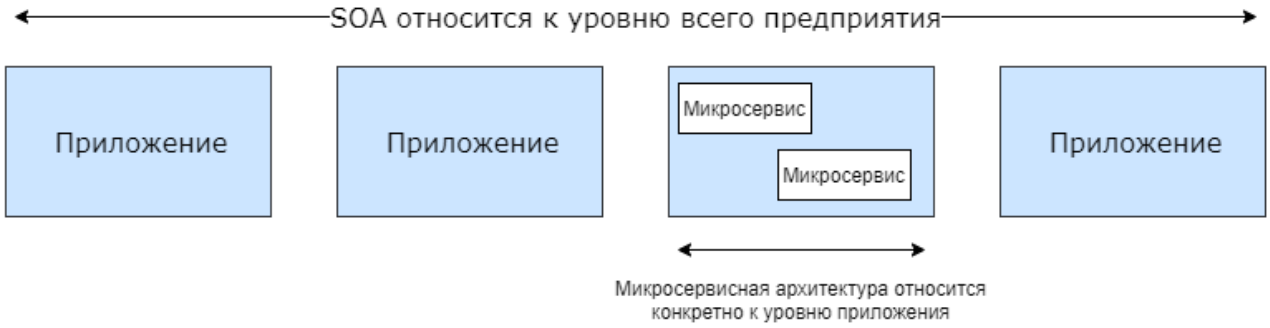


Рисунок 1.4 – Различия в сфере применения между SOA и микросервисной архитектурой

Такая же ситуация видна и на уровне данных. В большинстве примеров архитектур у каждого микросервиса есть своя база данных для уменьшения информационных зависимостей между ними. Но что если нескольким микросервисам нужны одни и те же данные? Такие данные либо дублируются для каждого микросервиса, либо вся ответственность на выдачу этих данных ложится на какой-то конкретный микросервис (данный подход является самым неоптимальным), либо создаётся новый микросервис, предназначенный конкретно для обработки этих данных. При правильном подходе такая зависимость должна быть отловлена на стадии проектирования приложения для оптимального решения такой проблемы с точки зрения как человеческих ресурсов, так и денежных.

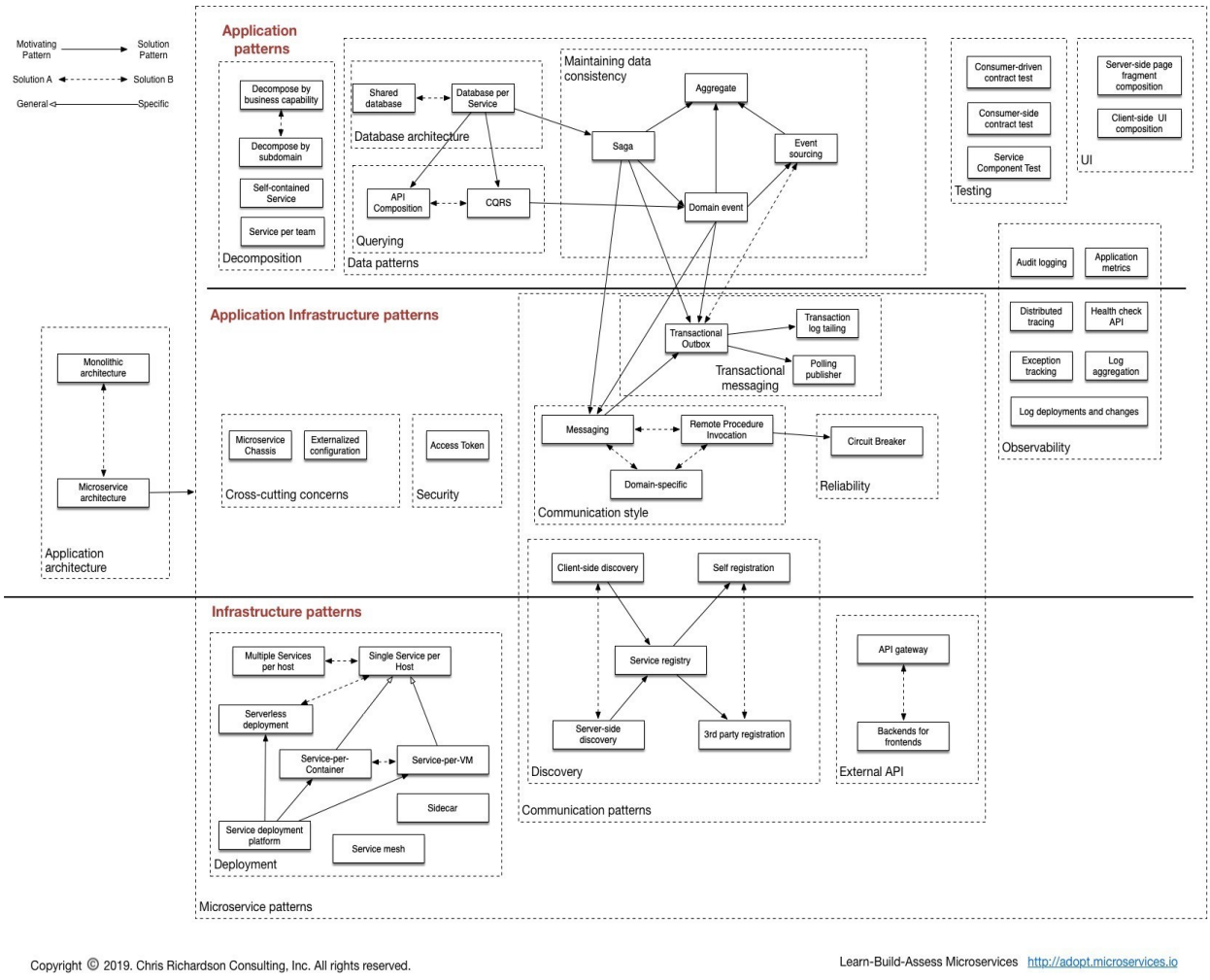
Среди других важных различий можно назвать такие как:

- каждый микросервис разрабатывается независимо от других со своим легковесным протоколом общения, как например HTTP/REST, в то время как в SOA каждый сервис должен использовать общий механизм общения ESB, который может стать единой точкой отказа для всего предприятия;

- микросервисы чаще всего являются очень узкоспециализированными, они спроектированы для качественного выполнения одной задачи, в то время как в SOA нет какого-то конкретного принципа для проектирования сервисов;
- повторное использование одних и тех же компонентов сильно упрощает тестирование и разработку, но в то же время это делает сервисы в SOA медлительнее микросервисов, каждый из которых чаще всего разрабатывается под конкретную задачу или с нуля, или при помощи дубликации кода и его последующего изменения под эту задачу.

1.3. Шаблоны проектирования приложений с микросервисной архитектурой

Шаблонов за относительно небольшое время существования данной архитектуры было придумано настолько много, что им посвятили целую книгу (рис. 1.5).



Learn-Build-Assess Microservices http://adopt.microservices.io

Рисунок 1.5 – Множество шаблонов микросервисной архитектуры [6]

Даже несмотря на это далеко не все из них можно и нужно использовать в каждом проекте на микросервисной архитектуре. В связи с этим были выделены следующие шаблоны, которые будут использованы в данной работе.

1.3.1. Шлюз API

Данный шаблон позволяет решить множество проблем, которые могут возникнуть в микросервисном приложении. Среди них:

- разным клиентам нужны разные данные, например в случае использовании настольной и мобильной версии сайта/приложения;
- клиент не должен видеть разделение на микросервисы, для него приложение/сайт должно выглядеть одним целым;
- количество экземпляров каждого микросервиса может меняться в течение времени, как и их хосты и порты.

Суть данного шаблона заключается в создании единой точки входа для всех клиентов, которая в свою очередь выступает в роли прокси/роутера для всех микросервисов (рис. 1.6). Данная единая точка входа также может осуществлять верификацию запросов. В качестве шлюза API часто используют контейнер nginx.

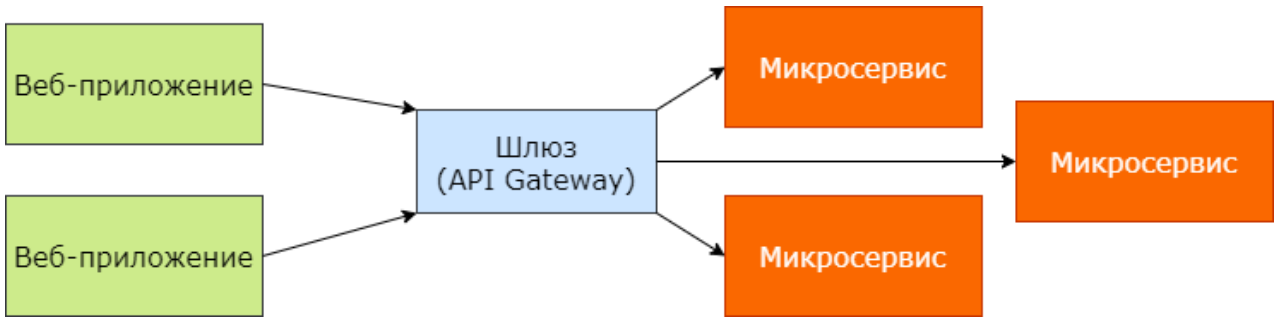


Рисунок 1.6 – Шаблон «шлюз API»

1.3.2. Токен доступа

После применения такого шаблона как «шлюз API», который является единой точкой входа для пользователей приложения, появляется проблема: как узнать имеет ли данный пользователь доступ к ресурсу, который он пытается получить?

Для решения этой проблемы был создан шаблон «токен доступа». Его идея заключается в том, чтобы каждому пользователю выдавать токен, в котором содержится полная информация о пользователе и его группа/разрешениях. Примером такого токена является JWT (рис. 1.7).

Но что делать в случае, если требуется межсервисное взаимодействие? Для этого обычно создаётся специальный токен для каждого из сервисов и, при необходимости взаимодействия между сервисами, каждый из них добавляет себе данный токен в список разрешённых, что позволяет им свободно использовать API друг друга.

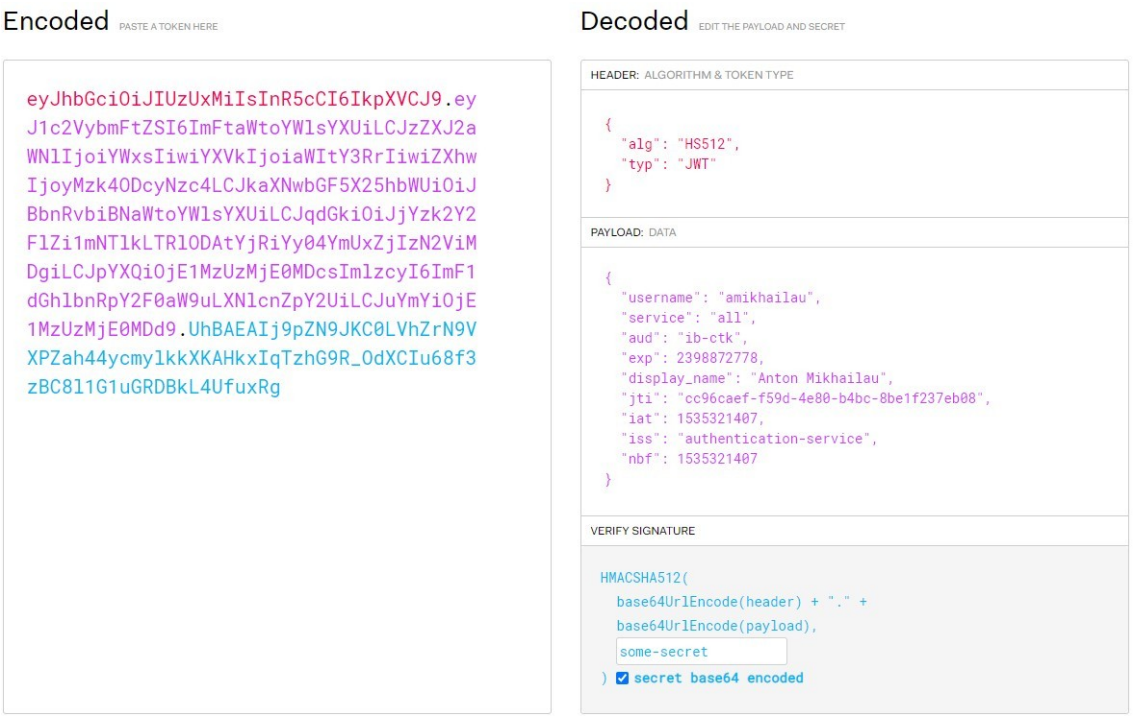


Рисунок 1.7 – JSON Web Token

1.3.3. Отдельная база данных для каждого сервиса

При проектировании приложения с микросервисной архитектурой всегда необходимо решить, как будет выглядеть хранение данных. В случае микросервисов на возможные варианты решения данной задачи накладывается несколько ограничений:

- микросервисы должны разрабатываться, масштабироваться и развёртываться независимо друг от друга;
- структуры хранения данных должны поддерживать инварианты данных, которые могут принадлежать разным микросервисам;
- некоторые бизнес-транзакции могут потребовать данные, которые могут принадлежать разным микросервисам;
- разные микросервисы могут нуждаться в абсолютно разных структурах хранения данных. Например, в то время как под модель одного микросервиса идеально подходит реляционная база данных, для другого микросервиса может понадобиться NoSQL база данных.

Самым очевидным и простым решением является отведение каждому микросервису своей базы данных. Каждый микросервис предоставляет доступ к своим данным только через API. Но как быть теперь с транзациями, которые затрагивают сразу несколько микросервисов? Для решения этой проблемы был создан следующий шаблон – Saga.

1.3.4. Шаблон Saga

Для реализации транзакций, проходящих через несколько микросервисов, используют цепочки локальных транзакций.

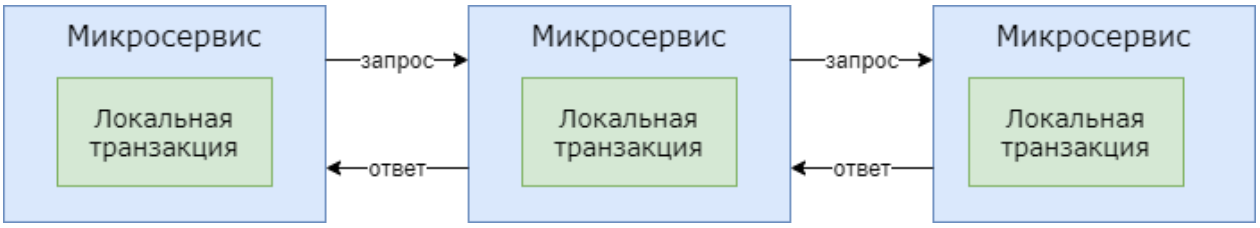


Рисунок 1.8 – Шаблон Saga

По сути, каждый из микросервисов делает обновление своей базы данных и посылает сообщение или запрос в следующий микросервис и так вниз по цепочке. В случае ошибки в одном из микросервисов при совершении обновления данных, в предыдущих по цепочке микросервисах обновления баз данных откатываются.

Для общения между микросервисами в ходе транзакции могут применяться 2 подхода:

- хореография – микросервисы общаются напрямую между собой, то есть сообщения для начала транзакции в следующем микросервисы и обратные сообщения об успешном её завершении/ошибке микросервисы посылают друг другу;
- оркестрация – в данном подходе создаётся специальный микросервис-посредник, который является центром общения для всех микросервисов. Сообщения о начале транзакции и успешном завершении/ошибке сперва отправляются данному микросервису и он в свою очередь перенаправляет данное сообщение получателю.

С оркестрацией есть одна проблема – появляется единая точка отказа, из-за которой вся система может перестать работать. С другой стороны, хореография накладывает дополнительную ответственность на разработчиков каждого из микросервисов для обработки взаимодействий с другими микросервисами.

1.3.5. Отдельный экземпляр микросервиса в каждом контейнере

Как уже было замечено в работе, микросервисы должны разрабатываться независимо друг от друга вне зависимости от применяемых языков программирования или технологий, легко и максимально эффективно с точки зрения аппаратных ресурсов масштабироваться и иметь возможность быстро развёртываться в разных окружениях. Кроме того, в микросервисной архитектуре крайне важно иметь хорошую систему мониторинга каждого экземпляра микросервиса для быстрого реагирования в случае ошибок.

Здесь на помощь приходит контейнеризация. Оно позволяет изолировать каждый микросервис как друг от друга, так и от среды окружения. В частности, очень известным является Docker. Для оркестрации таких контейнеризированных приложений используется специальное программное обеспечение, например Kubernetes.

1.3.6. Внешняя конфигурация

Очень часто для тестирования приложения и разных его версий может понадобиться несколько вариантов данного приложения в разных средах окружения. Отсюда вытекает проблема: как можно развёртывать микросервисы в нескольких средах окружения одновременно без их модификации?

Здесь на помощь приходит вынесение конфигурации микросервисов за их пределы. Для этого такие параметры как конфигурация базы данных, информация для получения доступа к другим микросервисам выносятся из микросервиса и задаётся например как переменные среды окружения. В частности, для Kubernetes существует инструментарий Helm, который позволяет создавать файлы конфигураций, переменные из которых Kubernetes будет использовать для создания экземпляров микросервисов.

1.4. Фреймворк gRPC

Для написания приложений на микросервисной архитектуре чаще всего используется протокол HTTP с сообщениями в формате XML/JSON для общения как с пользователем, так и между сервисами. Но в последнее время, хоть он и появился уже достаточно давно, начал набирать популярность gRPC, который позволяет без зависимости от конкретного языка программирования организовать удалённый вызов процедур.

gRPC – это высокопроизводительный универсальный RPC фреймворк с открытым исходным кодом. Используя его, клиентское приложение может вызывать метод на серверном приложении на другой машине, как если бы это

был локальный объект, что очень сильно облегчает создание распределённых приложений. Для сервера указываются методы, которые можно вызывать удалённо с их параметрами и типами возврата. На стороне сервера этот интерфейс реализуется и запускается gRPC-сервер для обработки вызовов методов. На стороне клиента есть заглушка, которая предоставляет те же методы, что и сервер.

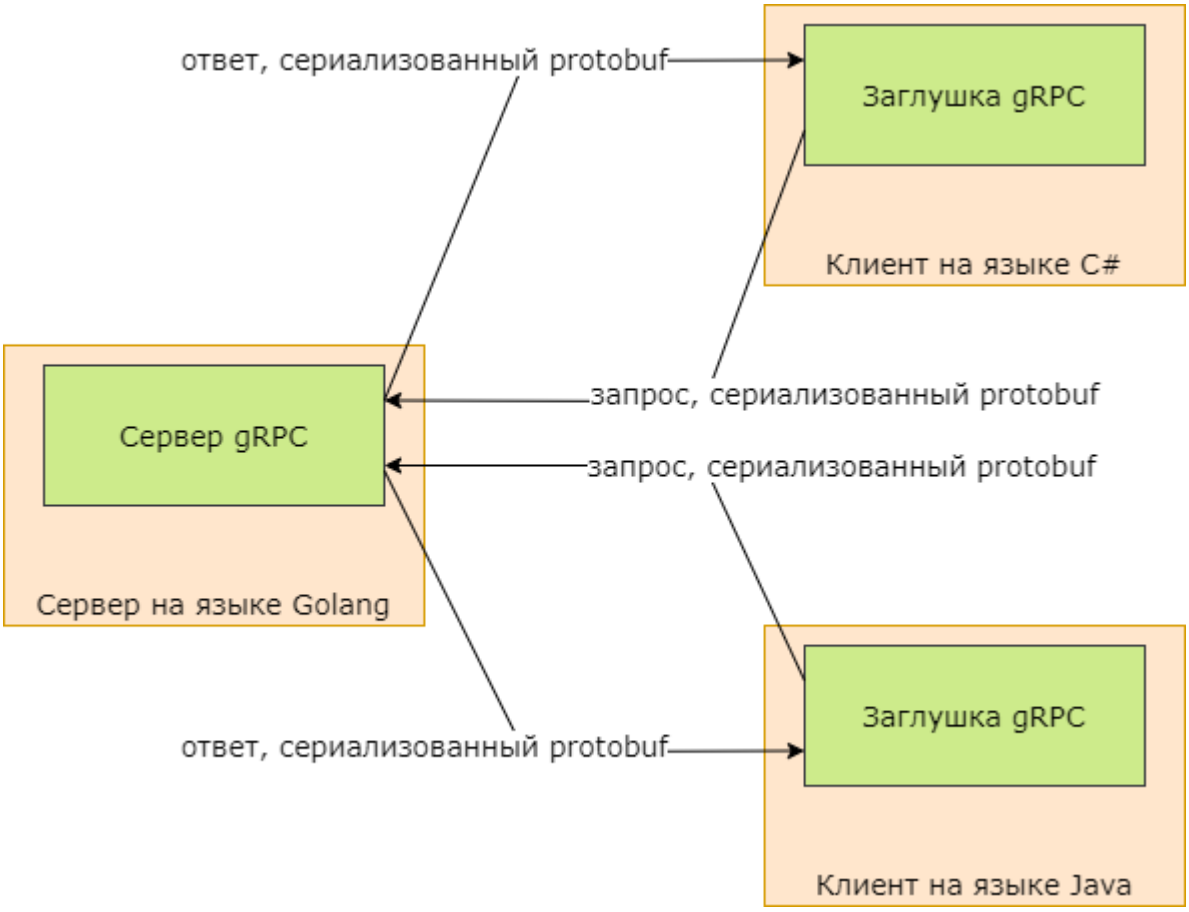


Рисунок 1.9 – gRPC

Для определения интерфейса, реализуемого сервисом на gRPC, и структур данных, используемых в сообщениях, в основном используется такой IDL как protobuf. В качестве транспорта вызовов и сообщений используется HTTP/2. Сами сообщения сериализуются при помощи protobuf в поток байтов.

Если же есть необходимость реализации сервиса, который будет использовать протокол HTTP с сообщениями в формате XML/JSON, то для grpc существует специальный отдельный плагин grpc-gateway для инструмента protoc, который генерирует на основе .proto файла обратный прокси-сервер, который транслирует RESTful HTTP в gRPC. Для использованного данного плагина необходимо использовать аннотацию google.api.http в описании сервисов в .proto файле. Пример можно увидеть в Приложении А, например в сервисе Users.

gRPC поддерживает 4 типа методов:

- унарный rpc – данный метод принимает один запрос и отправляет обратно один ответ.
  Пример такого метода в protobuf: `rpc GimmeFood(GiveFoodRequest) returns (GiveFoodResponse);`
- серверный rpc-поток – метод принимает один запрос и отправляет обратно клиенту поток для чтения последовательности ответов. Клиент читает из потока до тех пор, пока сообщения не закончатся. Порядок сообщений внутри каждого RPC вызова соблюдается.
  Пример такого метода в protobuf: `rpc GimmeLotsOfFood(GiveFoodRequest) returns (stream GiveFoodResponse);`
- клиентский rpc-поток – метод, который принимает поток записанных сообщений от клиента и отвечает на него одним сообщением. Снова же порядок сообщений соблюдается.
  Пример такого метода в protobuf: `rpc GIMMEFOOD(stream GiveFoodRequest) returns (GiveFoodResponse);`
- двунаправленный rpc-поток – метод как принимает поток сообщений, так и отправляет обратно поток сообщений. Эти два потока работают независимо друг от друга, сервер может как и отправлять ответ на каждое полученное сообщение, так и сперва дождаться всех сообщений от клиента и после этого отправить поток сообщений в ответ.
  Пример такого метода в protobuf: `rpc GimmeFoodEveryMinute(stream GiveFoodRequest) returns (stream GiveFoodResponse);`

ВЫВОДЫ

1. Проведён общий обзор микросервисной архитектуры. Поянено понятие «микросервис».
2. Проведено сравнение микросервисной архитектуры с монолитной архитектурой и SOA.
3. Описаны используемые в данной работе шаблоны проектирования приложений с микросервисной архитектурой.
4. Проведён общий обзор фреймворка gRPC.

ГЛАВА 2. СЕРВЕР МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ

2.1. Подходы к взаимодействию между клиентом и сервером

Перед тем как начать разговор об архитектуре серверной части многопользовательской игры, стоит отметить, какие есть возможности решения проблем, которые были перечислены в начале проекта:

- задержка при общении между клиентом и сервером;
- жульничество игроков.

Сперва стоит рассмотреть простейший случай – один клиент и сервер.

2.1.1. Авторитарный сервер

Итак, предположим, что игрок действительно пытается обмануть сервер и прыгнуть в другой конец карты. Как предотвратить такое действие? Можно поставить какую-то проверку на движение в клиенте, но так как игрок имеет доступ к файлам клиента, он может его модифицировать таким образом, что проверка проходить не будет.

Для этого применяется концепт «авторитарный сервер». Клиент всегда посылает информацию на сервер о действии, совершённом пользователем, а сервер в свою очередь отправляет новое состояние аватара пользователя (и в принципе всей сессии). Даже если клиент был модифицирован и/или шлёт на сервер сообщения с неправильными данными, проверки на сервере всегда недоступны пользователю, поэтому они не могут быть модифицированы и, как следствие, всегда откинут те сообщения клиентов, которые будут содержать в себе некорректные данные. Пример такого взаимодействия можно увидеть на рисунке 2.1.

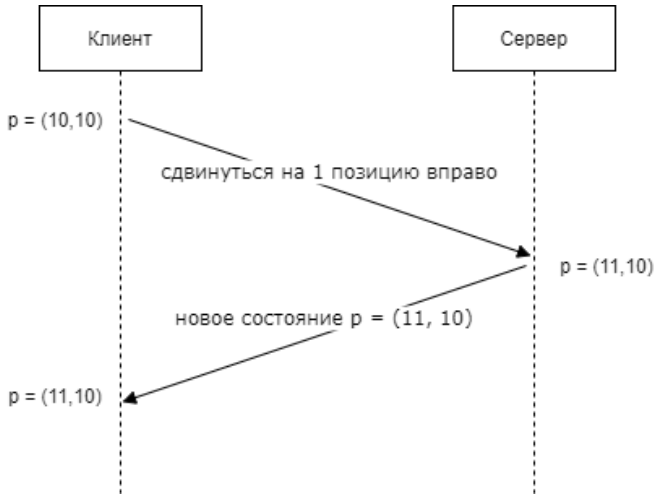


Рисунок 2.1 – Простейшее клиент-серверное взаимодействие

Если подводить краткий итог – клиент лишь посылает на сервер данные о своём действии, а сервер периодически шлёт на клиент обновленный статус сессии игры, который клиент в свою очередь показывает на экране пользователю.

2.1.2. Предсказание на стороне клиента

Итак, проблема жульничества в какой-то степени решена. Однако решение в виде авторитарного сервера лишь усугубляет проблему с задержкой. В качестве примера такой проблемы приведём следующий пример.

Представим, что у нас есть авторитарный сервер и клиент, который шлёт ему действия пользователя. Пусть пользователь совершит движение/действие, у которого анимация занимает некоторое время. Если провести данное взаимодействие в самом простом виде, получим то, что можно увидеть рисунке 2.2.

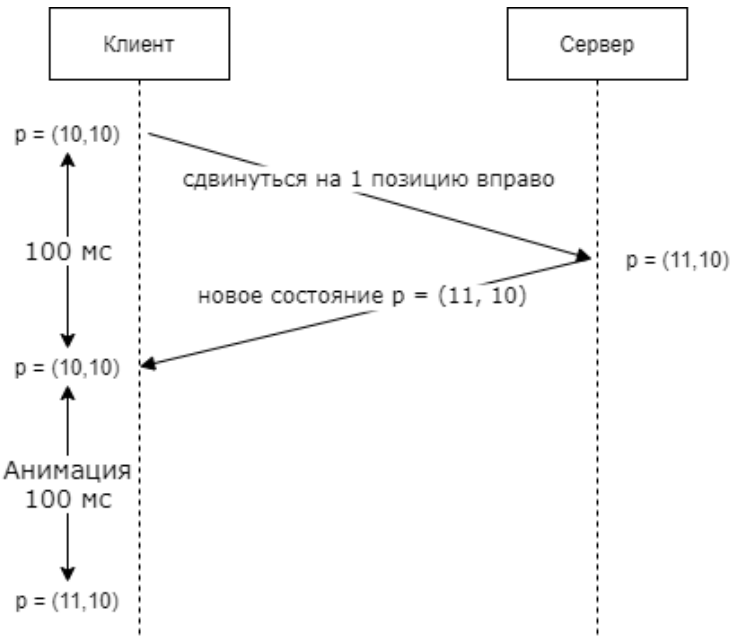


Рисунок 2.2 – Действие с анимацией

Что же можно увидеть? Между тем как пользователь совершил действие и увидел его результат прошло время задержки + время анимации. Получается это действительно долго, клиент игры будет казаться пользователю неотзывчивым. Есть ли возможность как-то сократить это время?

Для этого применяется такой приём как «предсказание на стороне клиента». Вместо того чтобы отрисовывать анимацию после получения обновленного состояния, она отрисовывается сразу же после совершения пользователем действия. Достичь этого можно, если игра является детерминистической и каждое действие пользователя имеет предсказуемый

результат, как например движение вправо может лишь сдвинуть аватар пользователя вправо, но никак не прыгнуть или сдвинуть его назад.

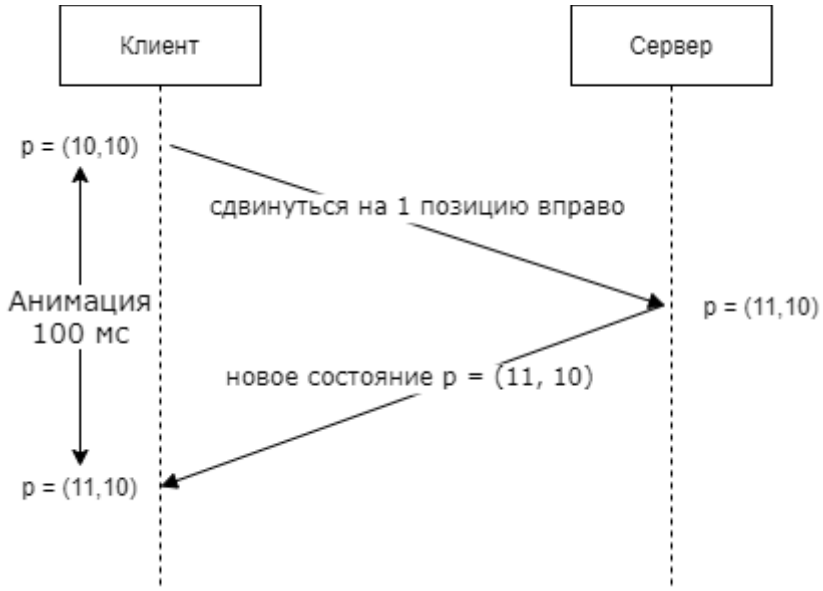


Рисунок 2.3 – Действие с предсказанием на стороне клиента

На рисунке 2.3 можно увидеть пример. Хоть на ней и показан идеальный случай, когда время анимации оказалось равным времени задержки, но в общем случае нам удастся убрать из задержки между совершением пользователем действия и получением результата на экране наименьшее из двух: время анимации или время задержки.

Казалось бы, проблема решена. Но данное решение само по себе имеет одну большую проблему, связанную с синхронизацией состояния клиента с состоянием сервера. Представим себе такой частный случай: пользователь совершил 2 движения друг за другом, которые выполняются с анимацией, а время задержки больше времени анимации. Увидим картинку, изображённую на рисунке 2.4.

Сначала аватар сдвинулся раз, потом сдвинулся ещё раз, потом, при получении первого обновлённого состояния сервера, скакнул назад без анимации, а затем, при получении второго обновлённого состояния, скакнул вперёд без анимации.

Данная проблема решается относительно легко при помощи нумерации сообщений клиента или подписывания их серверным временем. При помощи такого приёма клиент сможет понять, что состояние, которое пришло с сервера, ещё не включает в себя состояние с учётом последнего действия пользователя и прыжок назад не совершается. Пример такой подписи можно увидеть на рисунке 2.5.

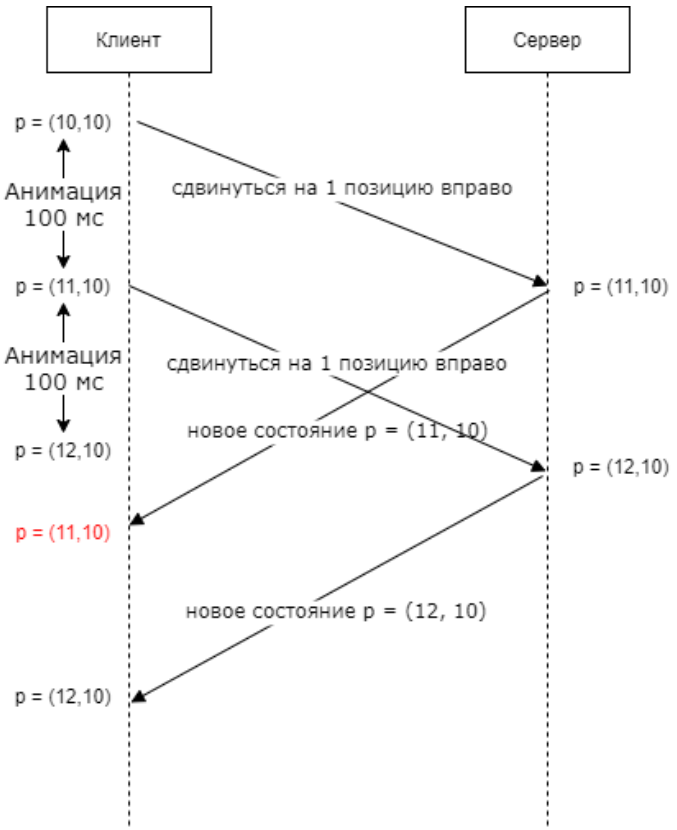


Рисунок 2.4 – Скачки аватара пользователя

Таким образом, аватар пользователя больше не будет скакать туда-обратно при получении обновлённых состояний сессии игры от сервера.

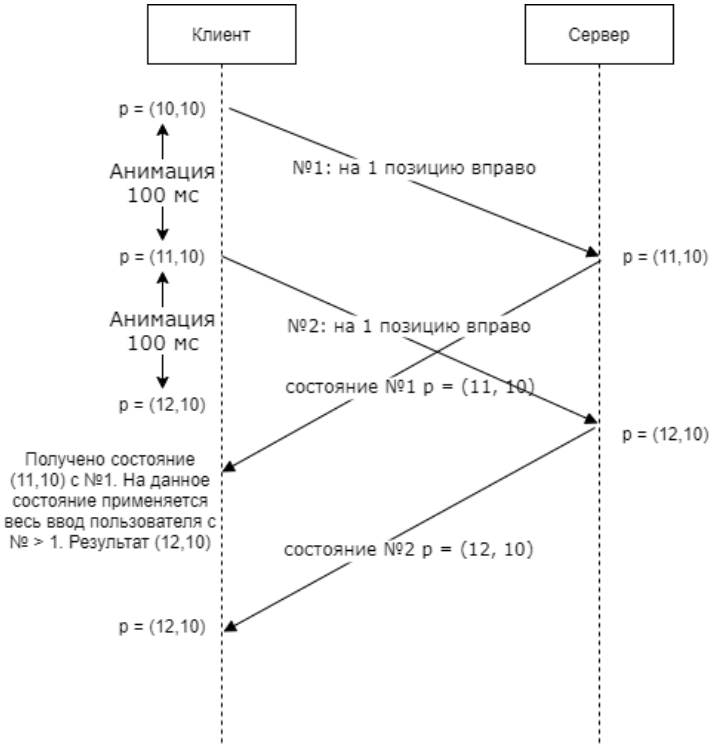


Рисунок 2.5 – Подпись сообщений клиента

2.2. Подходы к взаимодействию между несколькими клиентами и сервером

В предыдущей части можно заметить, что все интеракции приводились на примере лишь одного клиента. При помощи техник, перечисленных выше можно добиться того, чтобы один пользователь не ощутил разницы между однопользовательской игрой и многопользовательской игрой за исключением небольшой задержки. Однако это далеко не решает проблем, возникающих при попытке организовать взаимодействие между несколькими клиентами и сервером.

2.2.1. Синхронизация аватаров пользователей между клиентами

В случае нескольких клиентов у нас нет возможности отсылать обновлённое состояние каждому из клиентов после действия одного из клиентов, такой подход потребует слишком большие производительности сервера. Поэтому вводится такое понятие как «шаг сервера».

Сервер с какой-то периодичностью, которая называется «тик сервера», отправляет всем клиентам новое обновлённое состояние сессии игры, которое было рассчитано, исходя из всех сообщений клиентов, присланных в течении шага сервера. Такой подход как позволяет уменьшить нагрузку на сервер и сеть, так и позволяет упорядочить действия всех пользователей, позволяя тем самым создать своеобразную последовательность состояний сессии игры.

Однако не всё настолько хорошо, как может показаться. Представим себе предыдущий пример (рис. 2.5), но теперь пусть будет также и второй клиент, получающий информацию о движениях аватара пользователя на первом клиенте. В результате получим ситуацию, представленную на рисунке 2.6.

Решением прыжков является следующий подход – отображение состояний других клиентов на один шаг назад. Клиент сможет за дополнительный шаг интерполировать, что же произошло с аватаром другого клиента, на основе дополнительных данных, которые сервер будет передавать клиенту. Интерполяционные данные представляют собой описание того, какое действие совершал другой клиент (рис. 2.7).

Но что если нам нужны точные актуальные данные об аватаре другого клиента, например, если пользователь пытается выстрелить по нему? Пользователь видит только себя в настоящем времени, всё остальное в прошлом.

В данном случае сервер обращается к последовательности состояний, о которой шла речь чуть раньше, и смотрит на состояние сессии, когда был совершён выстрел. Если аватар другого клиента действительно оказался в тот

момент по итогу на пути выстрела пользователя, то снаряд попадёт в него.

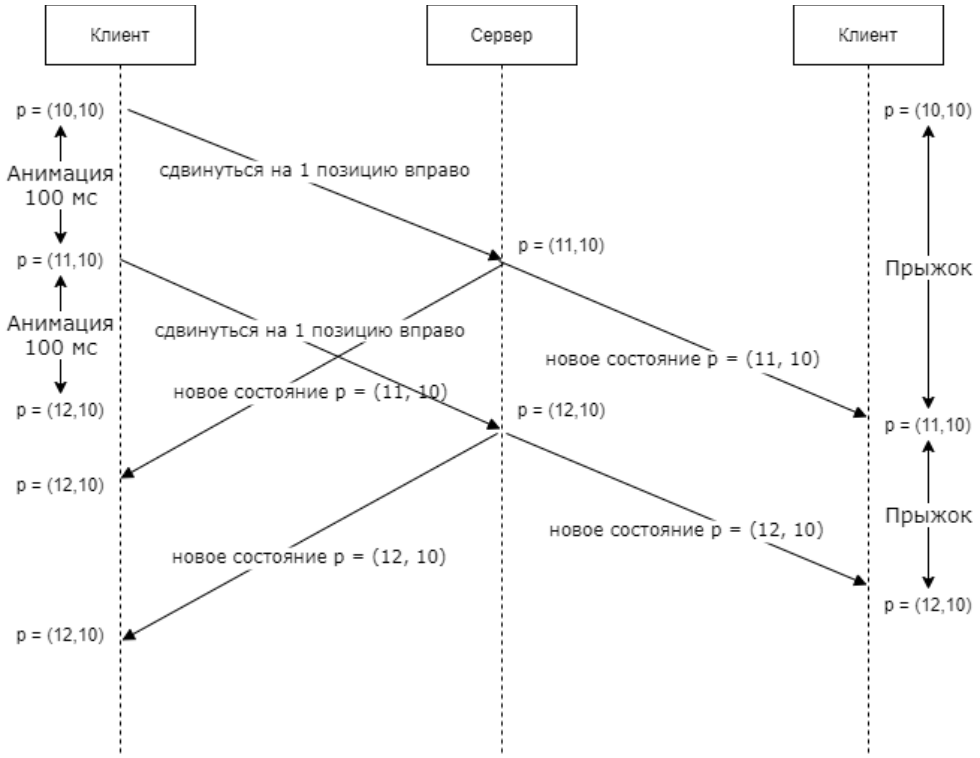


Рисунок 2.6 – Пример получения состояния аватара другого пользователя клиентом

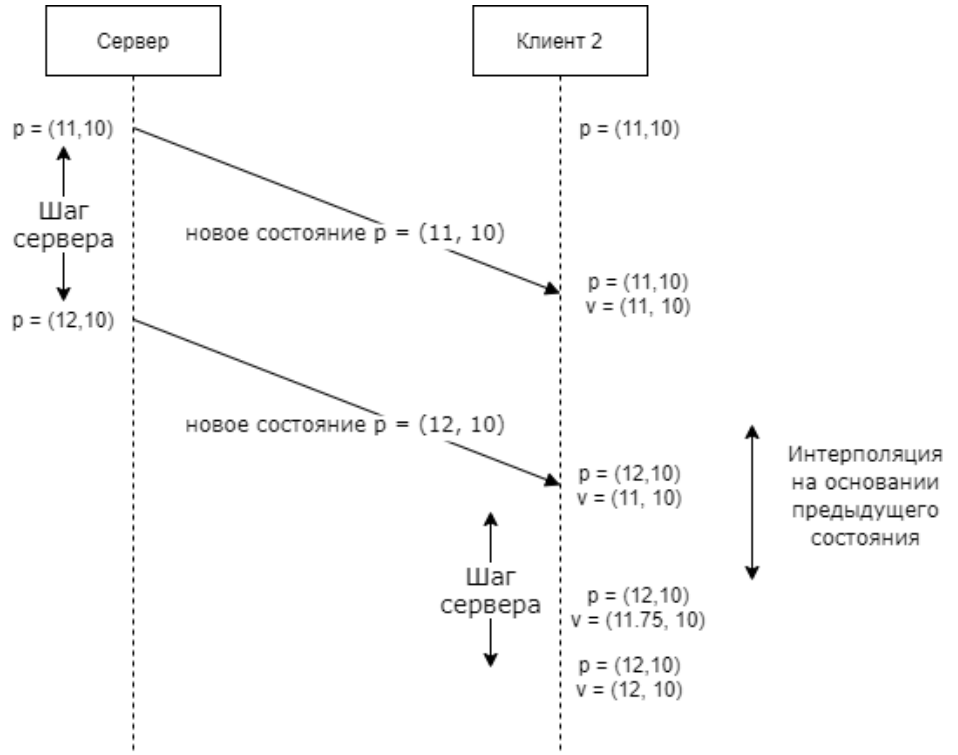


Рисунок 2.7 – Интерполяция позиции аватара другого пользователя по состоянию в прошлом

Здесь есть лишь одна небольшая проблема, которая возникает из-за такого

приёма. Например, если пользователь забежал за стену и через несколько миллисекунд умер. В таком случае пользователь будет негодовать, ведь ему казалось, что он наконец добежал до заветного укрытия, но тем не менее погиб. Проблема здесь заключается в том, что, хоть выстрел и был совершён в тот же момент глобального времени, как и момент, когда пользователь уже находился за укрытием, сервер посмотрел на состояние на 1 шаг назад и определил, что в тот момент пользователь ещё не был за стеной, а, следовательно, он мог быть поражён выстрелом. Клиент же пользователя ещё знать этого не знал и продолжал рисовать на экране картинку на 1 шаг вперёд.

Таким образом, клиент и сервер находят компромисс – хоть появляется проблема, описанная выше, но, по крайней мере, это лучше, чем то, что каждый игрок должен целиться не в аватар противника (или ему на ход, если у снаряда есть время полёта), а куда-то мимо него, в непредсказуемую точку, зависящую от задержки, чтобы попасть.

ВЫВОДЫ

1. Рассмотрена проблема жульничества пользователя.
2. Рассмотрены подходы и приёмы, применяемые для ликвидации проблем синхронизации состояний клиента и сервера.
3. Рассмотрены подходы и приёмы, применяемые для синхронизации состояний сессии игры между несколькими клиентами.

ГЛАВА 3. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ НА МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ

Перед тем как приступить непосредственно к разработке серверной части, необходимо проанализировать функционал игры и составить список требований к серверной части, после чего провести анализ составленного списка и спроектировать как архитектуру сервера, так и каждой его отдельной части.

3.1. Проектирование серверной части

В качестве основных технологий при развёртывании и оркестрации микросервисов будут использоваться Docker и Kubernetes, являющиеся стандартом отрасли и входящие в CNCF. С ними мне довелось достаточно много поработать в ходе прохождения как производственной, так и преддипломной практик. Отталкиваясь от этого, можно приступить к проектированию архитектуры кластера сервера.

3.1.1. Общий обзор требований

При проектировании необходимо было учитывать следующие минимальные требования к логике сервера:

- необходимо хранить и предоставлять информацию об пользователях, их статистике игр, игровых валютах;
- необходимо хранить и предоставлять информацию и её сортировку по некоторым показателям о косметических предметах, продающихся в внутриигровом магазине;
- необходимо позволять пользователю купить предмет из внутриигрового магазина;
- необходимо позволять пользователю создать учётную запись, зайти в учётную запись, изменить пароль и удалить свою учётную запись;
- необходимо позволять администратору создавать, изменять и удалять предметы внутриигрового магазина;
- необходимо позволять игроку купить один из типов валюты за реальные деньги за счёт внутриигровых транзакций;
- необходимо хранить состояние сессии игры на время её существования и синхронизировать данное состояние с клиентами всех пользователей, участвующих в данной сессии;
- необходимо изменять статистику игроков после окончания сессии

игры для всех игроков, которые в ней участвовали;
- необходимо хранить и предоставлять информацию о новостях, связанных с игрой, а также последних обновлениях;
- необходимо позволять администратору создавать и изменять последние новости о игре и её обновлениях.

3.1.2. Макет архитектуры серверной части

На основании данных требований было решено сделать разбиение логики сервера на 3 контекста (части):
- обработка сессии игры;
- обработка пользователей, внутриигрового магазина и новостей игры;
- обработка взаимодействия с платёжными системами для покупки игровых валют.

Данное разбиение на контексты обосновывается следующими аргументами:
- необходимо гибко масштабировать логику, обрабатывающую состояние сессий игры, так как данная логика является наиболее вычислительно сложной. При вынесении её как отдельного контекста в отдельный микросервис это позволит нам легко добиться необходимого контроля над масштабированием за счёт контейнеризации при помощи Docker и оркестрации экземпляров данного микросервиса при помощи Kubernetes;
- желание вынести все интеракции с платежными системами в отдельный контекст для обеспечения лучшей безопасности;
- сильная связанность между объектами «пользователь» и «предмет внутриигрового магазина», желание уменьшить количество межсервисных взаимодействий.

На рис. 3.1 можно увидеть макет архитектуры серверной части. Стоит отметить несколько деталей:
- шлюз сервера – это элемент шаблона API Gateway, указанного в 1-ой главе данной работы. Он будет отвечать за приём и перенаправление запросов на микросервисы;
- Pod – это своеобразный контейнер Kubernetes, в котором можно развернуть несколько Docker-контейнеров. В случае данного проекта все поды будут содержать по одному контейнеру соответствующего микросервиса;
- Kubernetes Ingress отвечает за балансировку запросов между подами и доступ к портам данных подов извне (взаимозаменим с Kubernetes LoadBalancer, если нет необходимости в изменении поведения в

зависимости от адреса). В случае UsersService и PaymentService не важно, какой из экземпляров микросервиса будет обрабатывать запрос, в отличие от GameService;

- данные о пользователях, предметах внутриигрового магазина и новостях будут храниться в базе данных PostgreSQL. База данных будет находиться вне кластера (например, на AWS или GCP);

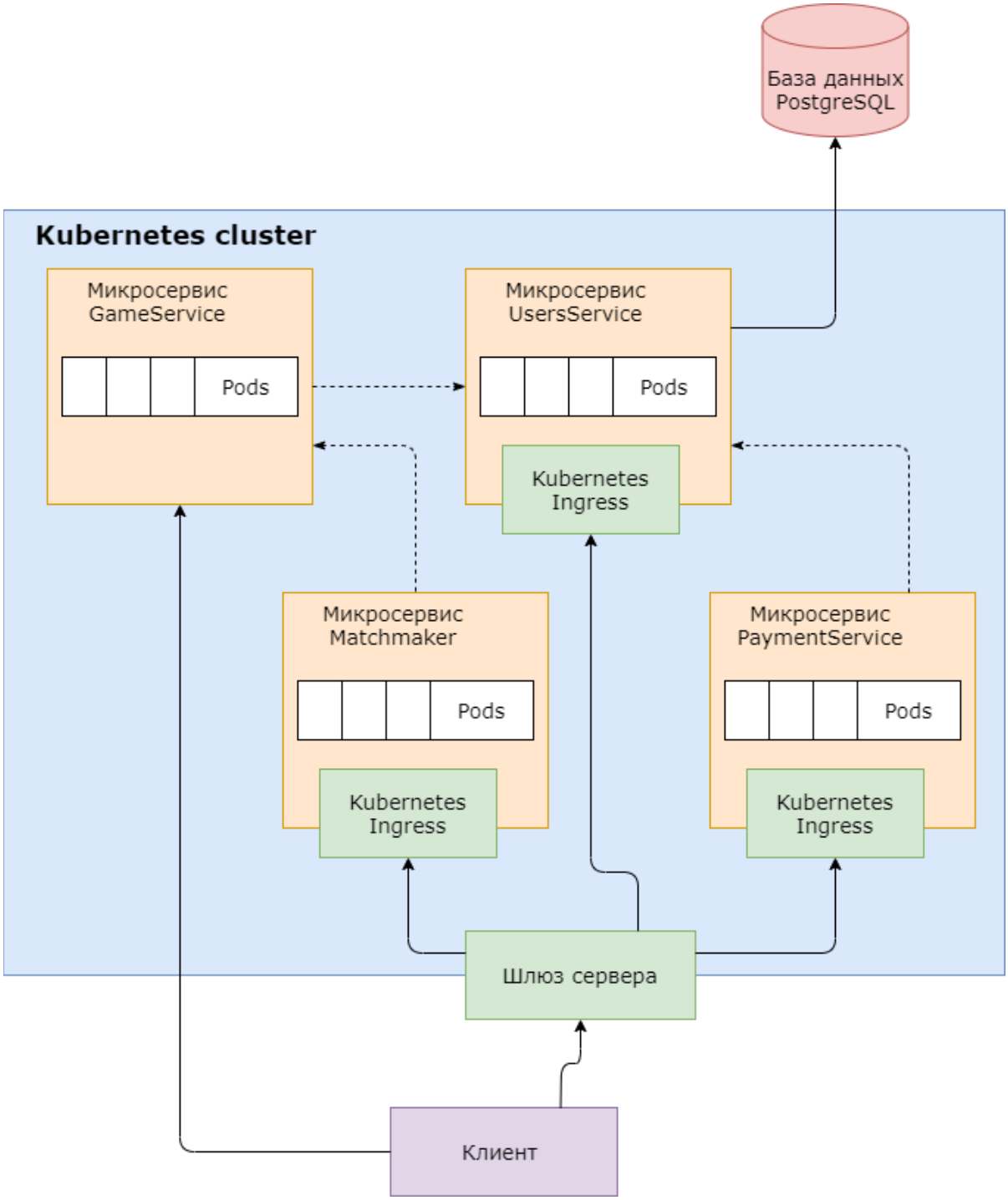


Рисунок 3.1 – Макет архитектуры серверной части

- для обновления статистики пользователей и начисления игровых валют микросервисы GameService и PaymentService будут совершать

внутрикластерные запросы, помеченные на макете штриховыми линиями, к UsersService;

- в случае GameService критически важно, чтобы запросы одного и того же пользователя были обработаны тем экземпляром микросервиса GameService, который обрабатывает и синхронизирует состояние сессии игры, в которой участвует данный пользователь;
- для подключения к экземпляру GameService клиент сначала должен быть привязан к нему. Процесс привязки производится во время создания новой сессии игры. За него будет отвечать другой микросервис – Matchmaker. Данный микросервис будет собирать данные клиентов, желающих принять участие в игре, при наличии необходимого количества игроков для создания одной или нескольких сессий игр будет запрашивать выделение или самостоятельно выделять экземпляр GameService для обработки данной сессии игры и передачу информации о выделенном экземпляре игры соответствующим клиентам;
- авторизация и аутентификация будут осуществляться при помощи JWT, подписанного алгоритмом RSA512. Приватный ключ для подписи будет храниться как Kubernetes Secret на самом кластере;
- при запросах к GameService клиенты будут использовать GRPC для уменьшения задержки, в то время как при запросах к UsersService и PaymentService будут использоваться HTTP+JSON.

В MVP данного проекта рассматривается разработка UsersService, Matchmaker и GameService. Проектирование и разработка PaymentService будет необходима в случае успешного прохождения данного проекта альфа- и бета-тестирования игроками.

3.1.3. Проектирование UsersService

Самым первым элементом UsersService, который необходимо спроектировать, является база данных.

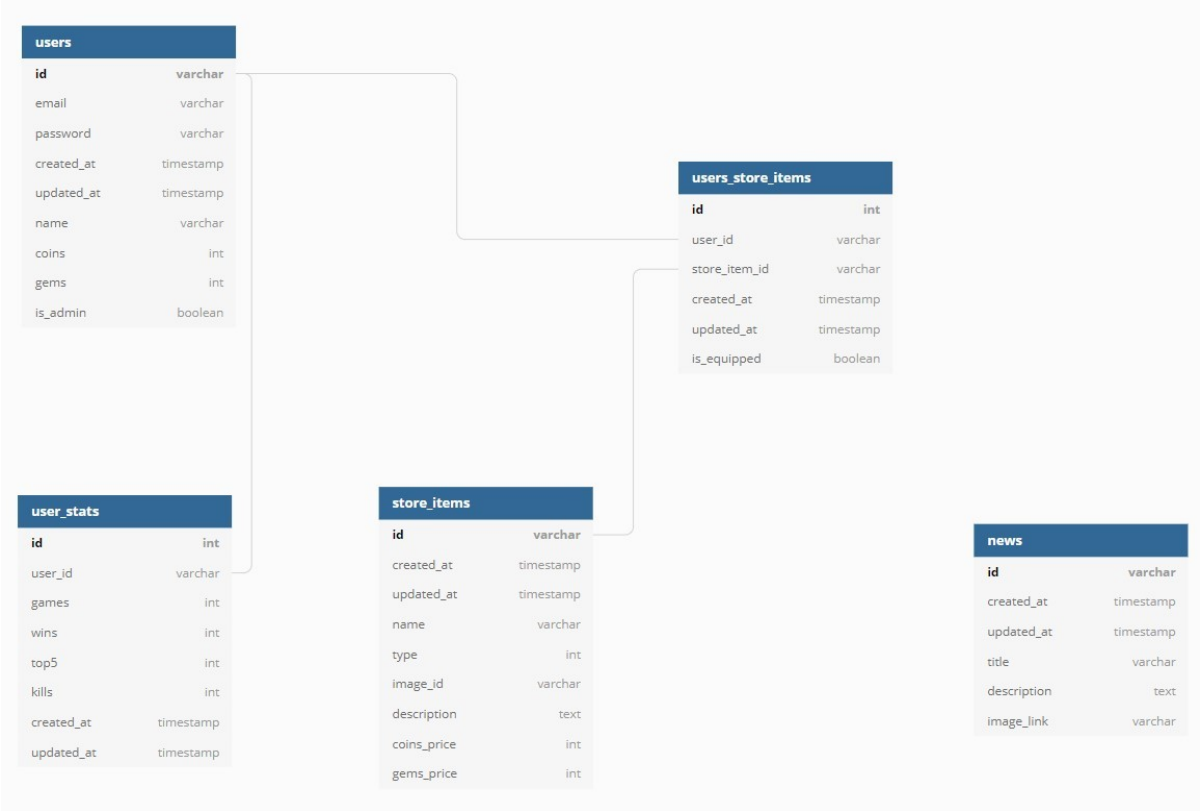
Для начала стоит выделить из требований сущности, которые необходимо будет хранить в базе данных. Всего их 4:

- пользователь;
- статистика пользователя;
- предмет внутриигрового магазина;
- новости.

Для данных сущностей была спроектирована схема базы данных, которую можно увидеть на рисунке 3.2.

Для предоставления возможности купить и надеть косметический предмет

внутриигрового магазина была добавлена дополнительная таблица



users_store_items, которая реализует связь «многие ко многим» между такими сущностями как «пользователь» и «предмет внутриигрового магазина».

Рисунок 3.2 – Схема базы данных

Для хранения пароля будет использован алгоритм SHA512 с динамической солью, которая будет храниться вместе с паролем.

Под каждую из сущностей была создана отдельная компонента, представляемая GRPC-сервисом (рисунок 3.3).

Описание структур данных и интерфейсов UsersService на языке protobuf можно найти в Приложении А.

3.1.4. Проектирование Matchmaker и GameService

Проектирование микросервисов Matchmaker и GameService должно ответить на следующие вопросы:

- как будет масштабироваться GameService при увеличении количества игроков и нагрузки?
- при использовании масштабирования каким образом клиент будет получать возможность подключаться к определённому экземпляру GameService, который отвечает за сессию игры, в которой находится пользователь данного клиента?

- каким образом будут организовываться сессии игр?

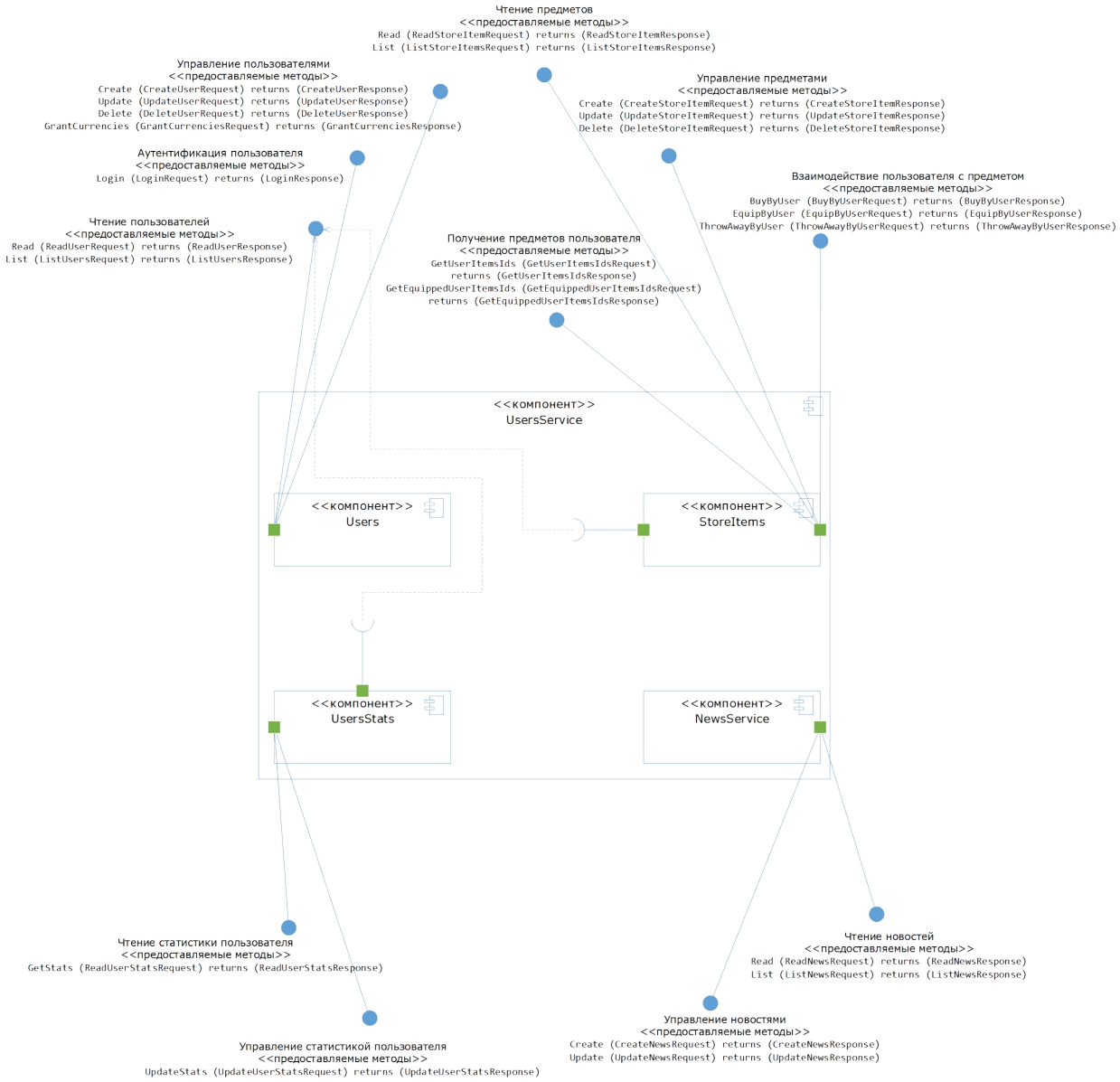
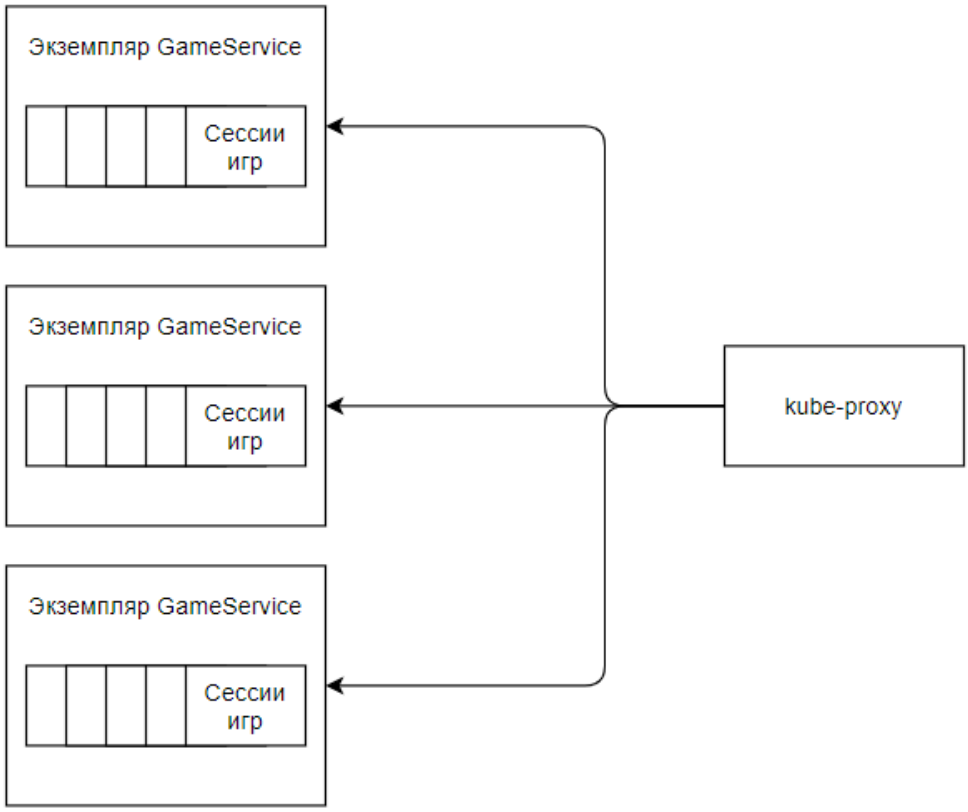


Рисунок 3.3 – Диаграмма компонент UsersService

В качестве решения проблемы масштабирования есть возможность использовать Kubernetes, где в качестве ответственного за масштабирование будет выступать такой ресурс как Deployment, в котором есть возможность указать количество экземпляров GameService. Каждый экземпляр будет хранить определённое количество одновременно существующих сессий игр (рис. 3.4).

Главная проблема, которая возникает при применении данного решения заключается в том, что запрос в каждый экземпляр перенаправляется таким компонентом Kubernetes, как kube-proxy. Kube-proxy не является прокси в обычном смысле данного слова, вместо этого он представляет собой сервис, который предоставляет виртуальный IP для каждого экземпляра. И большой изъян данного сервиса заключается в том, что балансировка нагрузки

осуществляется исключительно через «round-robin» алгоритм, то есть по



круговой.

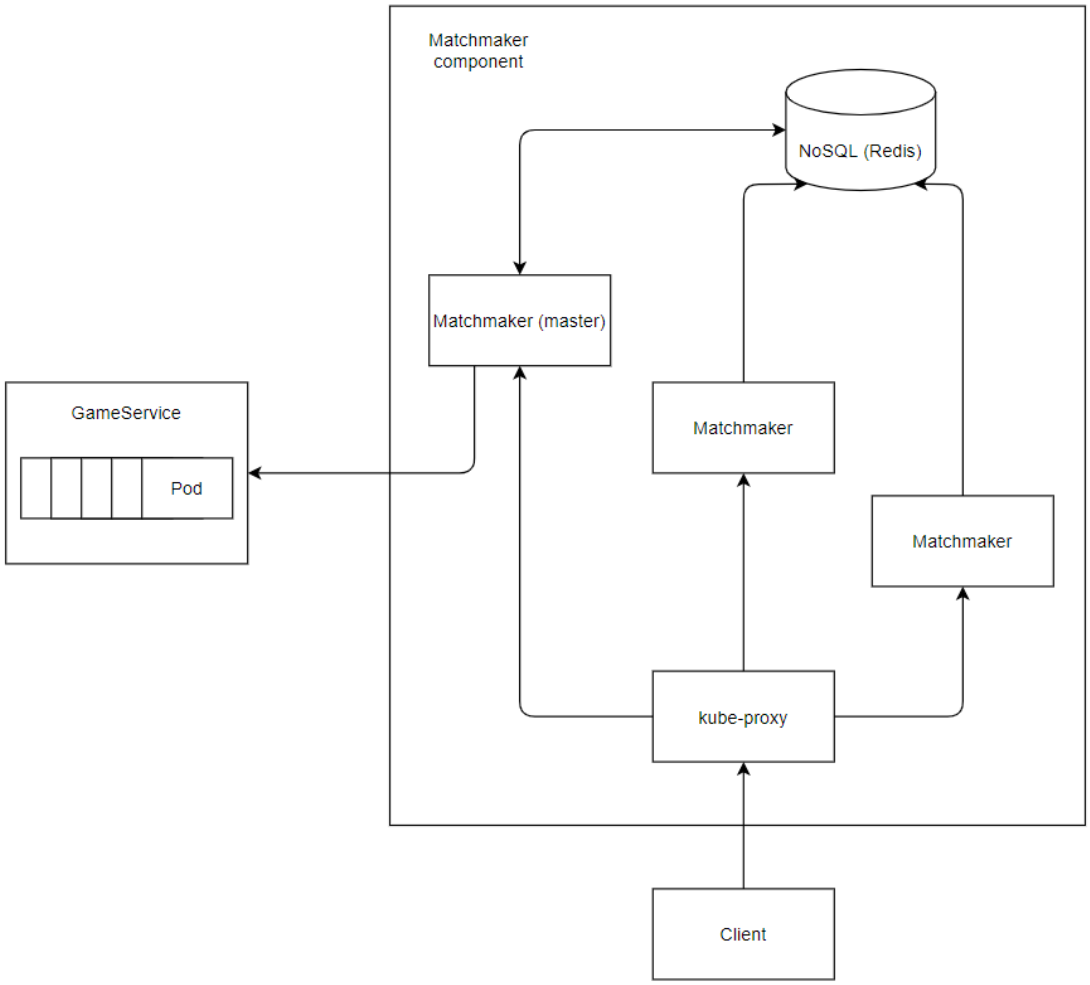
Рисунок 3.4 – Макет решения первого вопроса при помощи стандартных ресурсов Kubernetes

Данное ограничение не позволяет ответить на второй вопрос, поставленный в начале данного пункта «в лоб» через возможности самого Kubernetes и поэтому необходимо использовать надстройки над ним, как например Ambassador, который позволяет при помощи определения специальных заголовков запроса или cookies или за счёт фиксирования IP-адреса, от которого пришёл запрос, которые позволяют создавать так называемый «sticky session» – данное определение как раз подразумевает под собой метод балансирования нагрузки, при котором запросы клиента передаются на один и тот же экземпляр сервера из группы экземпляров.

Но не существует плюсов без минусов. Тот же Ambassador хоть и идёт в обход kube-proxy, но для получения такой возможности требует объявления таких ресурсов как Kubernetes Endpoint, которые указывают доступные для вызова методы микросервиса, и не предлагает в исходном варианте автоматического масштабирования микросервиса. Поэтому в ходе исследования темы масштабирования серверов игр на основе платформы Kubernetes и поиска технологий, предоставляющих похожий функционал, была

выделена платформа с открытым исходным кодом Agones, специально разработанная для решения проблемы развёртывания серверов игр и их динамического масштабирования, которая описана в Главе 4 данной работы.

Для решения последней проблемы – организации сессий игр – была проведена попытка провести анализ требований к организатору и



спроектирован макет решения (рис. 3.5).

Рисунок 3.5 – Макет решения третьего вопроса

Основными требованиями к организатору являются следующие: возможность масштабирования для поддержки увеличенного потока игроков и возможность быстрого восстановления в случае отказа одного из экземпляров организатора.

В данном решении при желании присоединиться к игре клиент посылает запрос организатору. Один из экземпляров организатора принимает данный запрос и помещает игрока в очередь в доступном всем экземплярам хранилище – Redis. Один из экземпляров организатора является главным, который проверяет, набралось ли необходимое количество игроков для создания одной сессии игры. В случае, если данное условие было выполнено, главный

экземпляр забирает данные об игроках из хранилища и отправляет запрос в GameService, чтобы получить информацию о следующем:

- сколько всего есть доступных экземпляров сервера и какой IP-адрес присвоен каждому из них;
- насколько загружен каждый из экземпляров сервер.

На основании данной информации организатор принимает решение какой из экземпляров будет отвечать за новую сессию игры и оповещает его об этом, отправляя вместе с оповещением список игроков. В свою очередь клиентам он возвращает информацию, на какой IP-адрес им нужно послать запрос, чтобы принять участие в сессии игры.

В случае организатора игр нет никакой необходимости обеспечивать «sticky session», поэтому стандартный сервис kube-proxy вместе с ресурсом Ingress/LoadBalancer обеспечат необходимую балансировку нагрузки.

В случае отказа главного организатора задача создания сессии игры при удовлетворении условия количества игроков переходит к другому экземпляру организатора. Для выполнения этого условия все экземпляры в основе являются идентичными, поэтому любой из них может выполнять задачу главного.

В ходе исследования данной проблемы после проектирования возможного решения также были предприняты попытки найти технологию, предоставляющую похожий функционал и имеющую похожую структуру, для использования в данной работе. В ходе поиска была обнаружена платформа с открытым исходным кодом OpenMatch, которая удовлетворяла всем требованиям. Данная технология более подробно описана в Главе 4.

Имея ответы на вопросы более высокого уровня, можно приступить к проектированию деталей внешних интерфейсов интерфейсов, которые каждый из микросервисов должен предоставлять. В случае GameService всё достаточно просто: он должен предлагать простейший интерфейс для подключения клиента и для обмена сообщениями с ним (рис. 3.6). Описание структур данных и интерфейса на языке protobuf можно найти в Приложении Б.

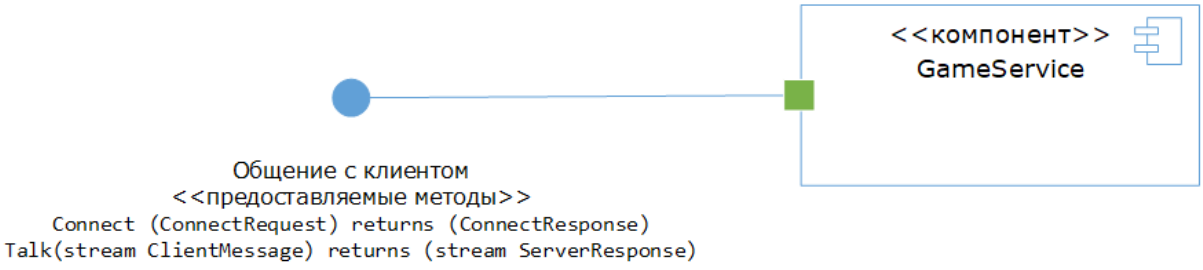
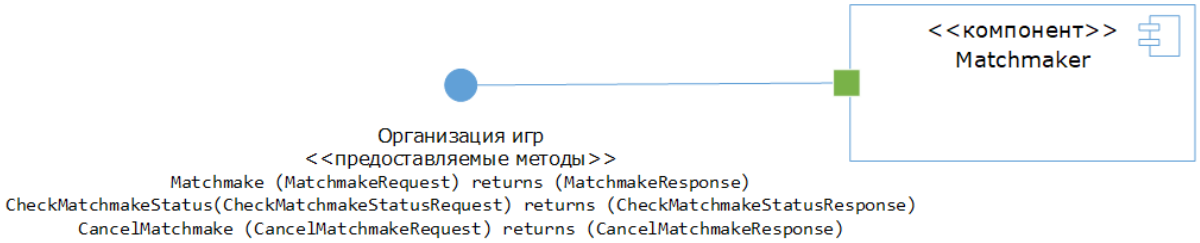


Рисунок 3.6 – Диаграмма компонент GameService

Для Matchmaker была учтена возможность использования OpenMatch для организации сессий игр. Поэтому Matchmaker имеет 2 опции: использовать стандартный организатор, предоставляемый разработчиком в логике данного

микросервиса, или выполнять лишь функцию выделения сервера игры по запросу и являться посредником между клиентом и OpenMatch, который в свою очередь будет выступать организатором сессий игр. Внешний интерфейс должен предоставлять возможности клиенту послать заявку на участие в игре, проверить, готова ли сессия игры и отозвать заявку на участие в игре (рис. 3.7).



Описание структур данных и интерфейса на языке protobuf можно найти в Приложении В.

Рисунок 3.7 – Диаграмма компонент Matchmaker

3.2. Реализация серверной части

Все микросервисы разрабатываются на языке Golang на основе кода, сгенерированного из файлов на языке protobuf при помощи инструмента protoc. Кроме того, для обработки соединений и запросов к данным микросервисам используется библиотека GRPC.

3.2.1. Реализация UsersService

Микросервис UsersService использует базу данных PostgreSQL для хранения всей информации, связанной с пользователями, предметами внутриигрового магазина и новостями игры. Для общения с базой данных микросервис использует как библиотеку GORM – ORM-библиотека для Golang, так и обычные подготовленные выражения. Пример одного из методов можно увидеть в Приложении Г. Большинство методов данного сервиса представляют собой валидацию данных и последующие либо получение информации из базы данных, либо управление (запись, изменение или удаление) информацией в базе данных.

Достаточно важно уделить внимание такой детали как авторизация. Как уже было указано чуть ранее, для авторизации используется JWT, подписанный при помощи алгоритма RSA512 приватным ключом сервера. Для реализации разграничений доступа между администратором и обычным пользователем используется такой шаблон как «интерцептор» (рис. 3.8).

Код интерцептора можно найти в Приложении Д. Полный код микросервиса UsersService можно найти в репозитории [11].

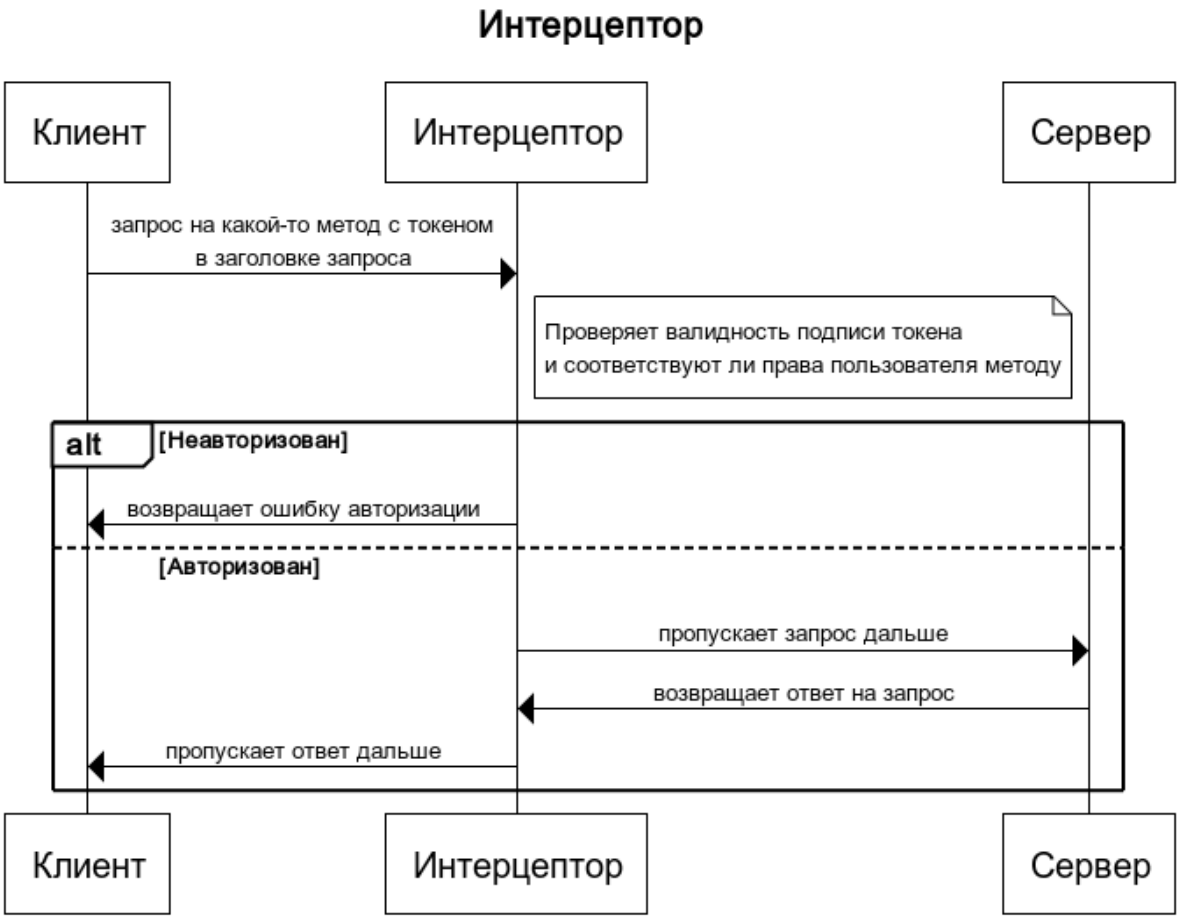


Рисунок 3.8 – Диаграмма последовательности для шаблона «Интерцептор»

3.2.2. Разработка Matchmaker

Так как Matchmaker тесно связан с GameService и будет использовать некоторые общие функции, для избежания ненужной дупликации кода их исходный код будут храниться в одном репозитории. В частности, одна из таких функций, которая используется в тестовом клиенте для GameService, позволяет выделить один из экземпляров сервера для обработки сессии игры. Исходный код можно найти в Приложении Е.

В стандартном организаторе, который используется при отсутствии OpenMatch компоненты, для хранения результатов – соотношения между клиентами, желающих принять участие в игре, и выделенных экземпляров сервера игры – используется потокобезопасное ключ-значение хранилище с ограниченным временем хранения [12], что позволяет очищать устаревшие записи. Для хранения игроков в очереди, которым ещё не была организована сессия игры, используется структура данных языка Golang «срез» в сочетании с мьютексом чтения-записи для синхронизации.

В случае присутствия OpenMatch, микросервис Matchmaker будет исполнять роль компонент Director и GameFrontend, которая будет описана как

часть процесса организации игр при помощи OpenMatch в Главе 4.

Диаграмму последовательности взаимодействия микросервиса Matchmaker без OpenMatch с сервером игры можно увидеть на рисунке 3.9.

Рисунок 3.9 – Диаграмма последовательности взаимодействия Matchmaker

Полный код микросервиса Matchmaker можно найти в репозитории [13].

3.2.3. Разработка GameService

Логика GameService разделена на 2 компоненты: GameManager и GameSession. GameSession отвечает за следующее:

- проведение всех расчётов изменений состояния сессии игры вследствие действий пользователей;
- расчёт шагов сервера;
- генерация сообщений и извещений для рассылки клиентам.

GameManager в свою очередь отвечает за следующее:

- инициализация GameSession;

- получение сообщений и извещений от клиентов;
- конвертация в необходимый для GameSession части формат сообщений и извещений от клиентов;
- передача конвертированных сообщений и извещений в GameSession;
- получение сгенерированных сообщений и извещений от GameSession и их последующая рассылка клиентам.

В отличие от Matchmaker и UsersService вместо использования привычного HTTP+JSON для общения с клиентом сервер использует GRPC. Для общения с сервером клиенту также придётся воспользоваться .proto файлом для генерации кода на нужном языке в отличие от HTTP-запросов, для которых достаточно задать URL и тело запроса в JSON-формате.

Общение между компонентами происходит при помощи примитива синхронизации «канал» языка Golang. Каналы являются основным механизмом общения между горутинами, которые представляют собой легковесные потоки. В частности, для передачи извещений об атаке или убийстве GameManager и GameSession используют каналы AttackNotifications и KillNotifications. Пример общения через канал KillNotification можно найти в Приложении Ж.

Как уже обсуждалось в Главе 2, сервер будет рассчитывать все изменения состояния сессии игры, что включает в себя попадание атак, коллизии с непроходимыми объектами и взаимодействие с предметами – любое взаимодействие с любыми другими объектами игры, на основе сообщений от клиентов, применённых к состоянию игры со смещением в 1-2 шага назад.

Вычисления состояния сессии игры требуют чуть более пристального внимания – так как сервер является авторитарным, все изменения в состоянии игроков и предметов будет происходить именно на сервере, поэтому серверу также придётся осуществлять анализ коллизий аватаров и непроходимых объектов и попадание атак, а для этого требуется применение специальных алгоритмов.

В проекте, для которого разрабатывается серверная часть, описанная в данной работе, были сделаны следующие допущения: есть непроходимые для игроков области, которые предопределены ещё до начала игры, аватары игроков могут проходить через друг друга и их модель коллизии представляет собой круг.

3.3. Алгоритмы нахождения и разрешения коллизий объектов

Коллизия объектов – это ситуация, при которой два или более твёрдых тела имеют общие точки (пересекаются) или дистанция между двумя или более твёрдыми телами меньше допустимого.

В общем случае, если у нас есть n объектов, то попарное выполнение

проверки на коллизию приведёт нас к сложности $O(n^2 * T(m))$, где *T(m)* – это сложность алгоритма нахождения коллизии между 2-мя объектами (обычно она зависит от того, как представлены объекты, а также сколько граней имеет каждый). Казалось бы – сложность полиномиальная, но в играх очень важно быстродействие, даже небольшая оптимизация всегда приветствуется, поэтому процесс нахождения коллизий разбивается на 2 фазы: широкая фаза и узкая фаза.

В широкой фазе алгоритм не обязан сразу же найти все коллизии, он должен лишь указать, какие из объектов потенциально находятся в коллизии. При этом конечно же данный алгоритм должен иметь сложность меньшую, чем $O(n^2)$, иначе смысл оптимизации немного теряется.

В узкой фазе уточняются детали потенциальных коллизий, найденных в широкой фазе. Кроме нахождения самой коллизии и точек пересечения объектов коллизии также необходимо каким-то способом разрешить.

Так как объекты очень часто могут иметь весьма непростую форму, сложнее чем обычный выпуклый многоугольник, для ускорения работы алгоритма для всех объектов вычисляется более простая форма, которая полностью покрывает объект, например прямоугольник в двумерном пространстве или прямоугольный параллелепипед в трёхмерном. Данная простая форма называется «ограничивающим объёмом» (рис. 3.10).

Рисунок 3.10 – Используемые ограничивающие объёмы для задач в двумерном пространстве

В случае движущихся объектов вместо ограничивающего объёма для самого объекта часто используют ограничивающий объём, который получается путём покрытия объектом объёма при его перемещении из начала в конец пути, что позволяет избежать объектов с высокой скоростью, пролетающих через препятствия.

Так как игра, разрабатываемая в данной работе, является двумерной, для

ограничивающих объёмов будут использоваться прямоугольник, ограниченный координатной сеткой – один из самых простых вариантов.

3.3.1. Алгоритмы широкой фазы

Сами по себе ограничивающие объёмы не уменьшают сложность поиска коллизий, ведь проблема попарного анализа объектов так и не была решена. Для решения данной задачи существует несколько популярных алгоритмов: Sweep and Prune, динамические деревья ограничивающих объёмов и иерархические сетки.

Sweep and Prune. Суть данного алгоритма достаточно проста. Используя ограничивающие объёмы всех твёрдых объектов, для которых необходимо проверять коллизии, строятся проекции на одну из координатных осей. В результате получается набор интервалов $[x_{min}, x_{max}]$. Если ограничивающие объёмы 2-х твёрдых объектов пересекаются, то они будут иметь как минимум одну общую точку, из чего следует, что их проекции будут накладываться друг на друга. Поэтому для нахождения потенциальных коллизий достаточно проверить пересечения проекций всех ограничивающих объёмов на одну из координатных осей.

В данном алгоритме все концы проекций помещаются в один массив и сортируются по величине координаты. Затем необходимо пройти по этому массиву. При встрече x_{min}, то есть начала какого-то интервала, интервал помещается в множество активных интервалов, при встрече x_{max}, то есть конца какого-то интервала, данный интервал удаляется из множества активных интервалов. Присутствие в множестве более чем одного интервала означает, что данные интервалы пересекаются (рис. 3.11).

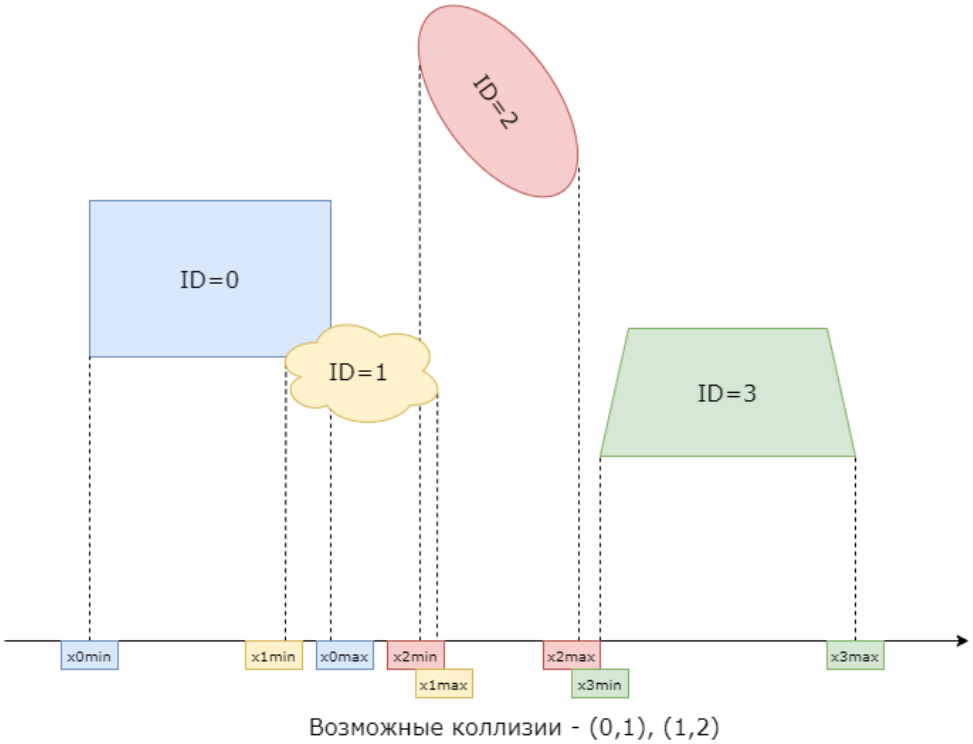


Рисунок 3.11 – Пример использования Sweep and Prune

Казалось бы, сортировка, использование множества, прохождению по массиву являются не самыми дешёвыми операциями, но за счёт того, что в игре за один шаг симуляции объекты почти наверное не переместятся на большую дистанцию, сортированный массив интервалов можно переиспользовать и на следующем шаге. При изменении положения какого-то из объектов для пересортировки массива используется сортировка вставками, которая очень быстро справляется с почти отсортированными массивами.

Единственный вопрос, который возникает при использовании данного алгоритма заключается в координатной оси, которую выбирают для построения проекций. Ведь если все объекты будут выстроены в один ряд, параллельный одной из координатных осей, выбор данной оси будет ошибкой, приводящей к анализу на коллизию всех пар объектов более тяжёлым алгоритмом узкой фазы. Для данного алгоритма существует его более сложная и медленная версия, которая решает данную проблему. В этой версии алгоритма вместо сортировки вставками используется быстрая сортировк, а на каждом из шагов симуляции вычисляется дисперсия центров ограничивающих объёмов по каждой из координат, и координата с наибольшей дисперсией выбирается для анализа на потенциальные коллизии [15, с. 336].

Преимущества Sweep and Prune:

- хорошо подходит под объекты, которые движутся по законам физики;
- использует пространственную и временную согласованность – если 2 объекта находятся рядом, то с очень большой вероятностью через короткий промежуток времени они также будут находится рядом – свойственную обьектам в играх;
- на практике средняя временная сложность простой версии этого алгоритма равна O(n) в отличии от остальных алгоритмов, у которых она равна O(n*log(n)) [18, стр.31].

Недостатки Sweep and Prune:

- использует ограничивающие объёмы, ограниченные координатной сеткой, которые обычно захватывают большую площадь, чем сонаправленные с объектом;
- для объектов, движущихся с большой скоростью, часто находится большое количество возможных коллизий.

Динамические деревья ограничивающих объёмов. Данный алгоритм является более сложным для понимания и реализации вариантом для нахождения потенциальных коллизий в широкой фазе (используется для трассировки лучей). Для всего пространства игры строится бинарное поисковое

дерево – чаще всего АВЛ-дерево для балансировки – в котором каждая вершина представляет собой какой-то ограничивающий объём. Сыновья вершины представляют собой непересекающиеся ограничивающие объёмы, которые покрываются ограничивающим объёмом вершины-родителя (рис. 3.12).

При необходимости найти возможные коллизии для конкретного объекта сравниваются ограниченные объёмы объекта и вершины, и если они пересекаются, то сравнение спускается на уровень ниже к сыновьям данной вершины.

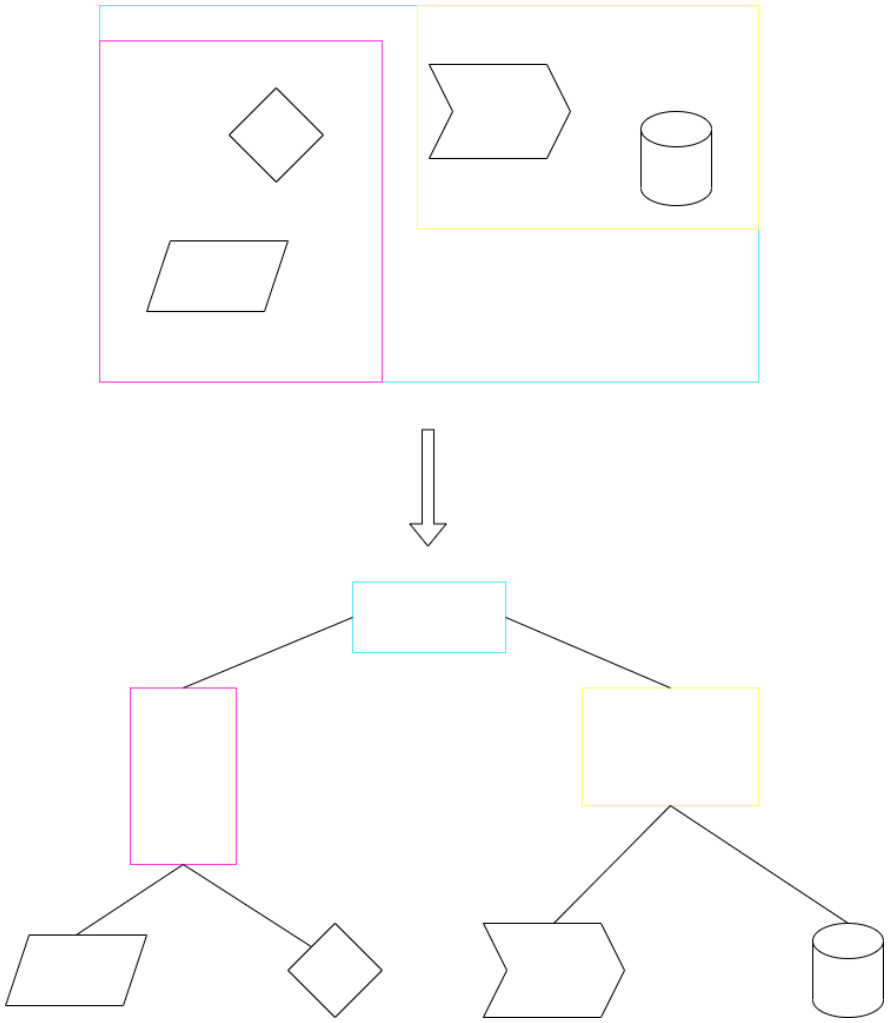


Рисунок 3.12 – Пример динамического дерева ограничивающих объёмов

Преимущества динамических деревьев ограничивающих объёмов:

- хорошее разрешение для нахождения потенциальных коллизий, что позволяет уменьшить количество объектов, которые придётся рассматривать в следующей фазе;

Недостатки динамических деревьев ограничивающих объёмов:

- требует создания промежуточных ограничивающих объёмов, из-за чего деревья могут очень сильно вырастать;
- балансировка деревьев при перемещении объектов требует

дополнительного времени;

- не подходит для объектов, которые могут менять свою геометрию;
- относительно сложны в реализации.

Иерархические сетки. Данный алгоритм предполагает построение сетки на пространстве всей карты и хранения для каждой ячейки идентификаторов всех ограничивающих объёмов (а точнее объектов, которые описываются данными ограничивающими объёмами), которые имеют хотя бы точку в данной ячейке. Если для какой-то ячейки нашлось несколько идентификаторов, значит объекты, которым данные идентификаторы принадлежат, стоит рассмотреть алгоритмом узкой фазы (рис. 3.13).

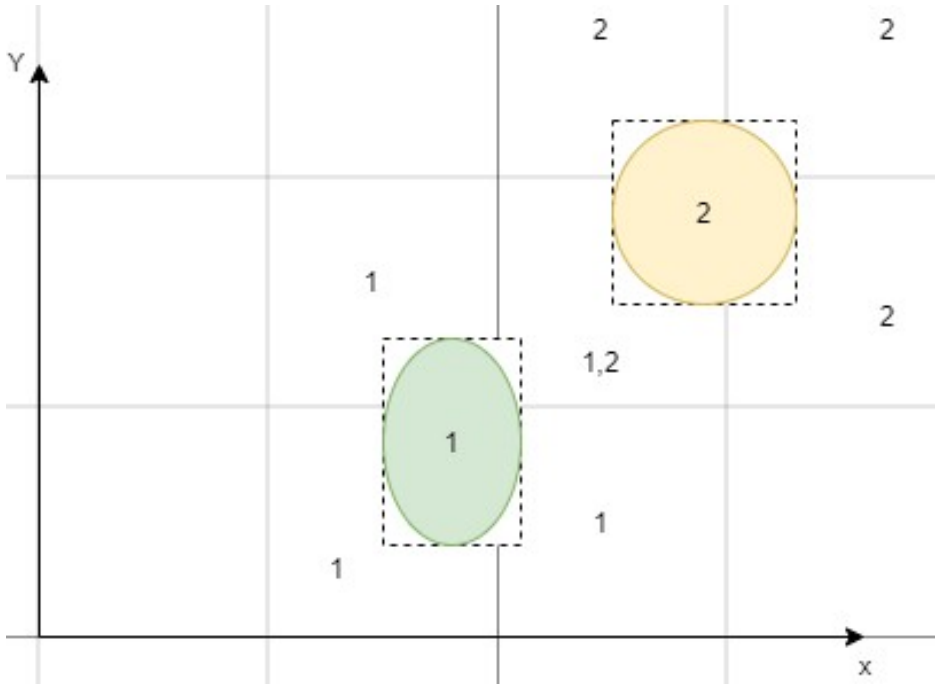


Рисунок 3.13 – Пример иерархической сетки

Преимущества иерархических сеток:

- простые в реализации;
- хорошо работает для маленьких относительно разрешения сетки объектов;

Недостатки иерархических сеток:

- сложно подобрать правильное разрешение сетки;
- большие затраты памяти при хранении больших сеток, особенно в трёхмерном случае;
- медленный пересчёт сетки для больших движущихся объектов.

На основании всех перечисленных преимуществ и недостатков, в частности скорости и средней по сложности реализации, для данной работы был выбран Sweep and Prune. Пример его реализации для движущегося аватара пользователя можно найти в Приложении И.

3.3.2. Алгоритмы узкой фазы

При переходе к узкой фазе сперва необходимо провести небольшую подготовительную работу – просмотреть многоугольники, и если какие-то из них не являются выпуклыми, то преобразовать их в выпуклые. Для этого есть 2 подхода:

- обернуть невыпуклые многоугольники в выпуклые. Для данной задачи существует алгоритм quickhull. Однако это приводит к потере данными многоугольниками некоторых качеств их геометрии, которые могут привести к коллизиям там, где их быть не должно;
- разбить невыпуклые многоугольники на выпуклые. Для данной задачи существует такой алгоритм как V-HACD.

В данной работе имеется предопределённое разбиение всех невыпуклых непроходимых препятствий на выпуклые, поэтому необходимости использовать данные алгоритмы отпадает.

Исходя из допущений, сделанных для проекта, всего есть 3 возможных ситуации, в которых необходимо рассматривать коллизии. Для каждой из них были использованы различные алгоритмы узкой фазы:

- атака игрока – сама атака представляет собой сектор круга. Так как модель коллизии для атак аватара также представляет собой круг был использован простейший алгоритм проверки наложения круга на круг с дополнительным расчётом угла сектора наложения кругов;
- при успешной атаке аватар, по которому атака попала, отбрасывается назад на небольшую относительно всех объектов дистанцию. Поэтому в конце каждого шага сервера будут рассчитываться статичные коллизии всех аватаров и непроходимых областей. Непроходимые области представляют собой набор выпуклых многоугольников, соединённых между собой, а модели аватаров – круги. Для этого был использован метод областей Воронова;
- для любых других ситуаций использовался метод, основанный на теореме о разделяющей оси.

Коллизия двух кругов. Данный случай является элементарным – необходимо проверить расстояние между центрами кругов: если оно меньше суммы радиусов, то коллизии нет, иначе коллизия есть.

Однако конкретно его применение для проверки попадания атак вносит некоторое небольшое усложнение – есть необходимость проверить попадает ли конкретно сектор атаки по кругу (рис. 3.14).

Для данной проверки вводится сектор попадания, изображённый на рисунке 3.14 синим цветом. Он образуется 2-мя треугольниками, имеющих 3

стороны: радиус первого круга, радиус второго круга и отрезок, соединяющий центры кругов. После этого достаточно найти углы каждого из конца секторов атаки и концов сектора попадания и проверить, имеют ли данные секторы общий сектор.

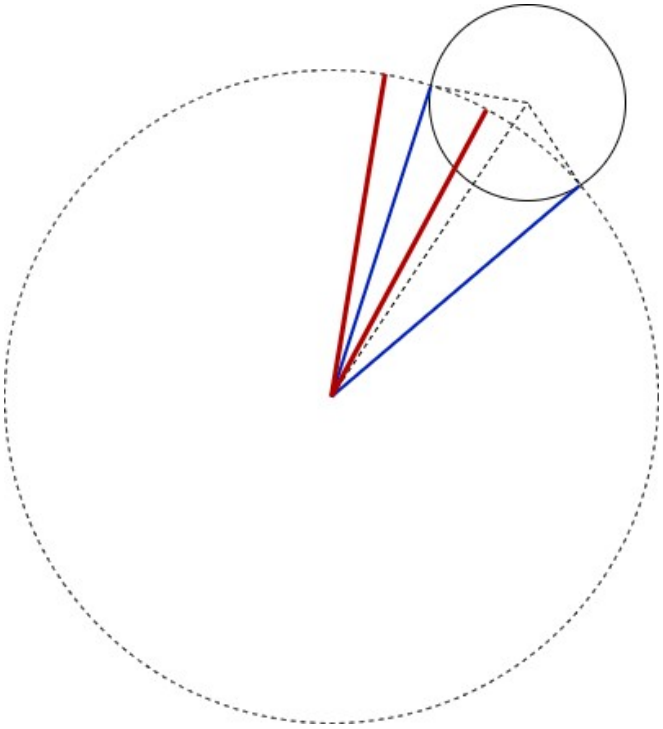


Рисунок 3.14 – Попадание красного сектора по кругу

Коллизия круга и выпуклого многоугольника (полигона). Для рассмотрения этой задачи сперва необходимо пояснить такое определение как регионы Воронова.

Начнём с диаграммы Вороного. Диаграмма Вороного для конечного множества точек *T* на плоскости представляет собой набор множеств точек, которые получаются из разбиения плоскости по критерию близости к точкам множества *T*, то есть каждое множество точек является более близким к одной из точек множества *T*, чем к каким-либо другим из множества *T*.

Регионы Вороного представляют собой перенесение данного понятия с множества точек на фигуры. Каждая фигура разбивается на множество элементов – например двумерный многоугольник можно разбить на его рёбра и вершины, а трёхмерный на вершины, рёбра и грани. Затем пространство, которому принадлежит эта фигура, разбивается на набор множеств точек, при этом каждое множество точек является более близким к одному из элементов фигуры, чем к какому-либо другому элементу фигуры (рис. 3.15).

Алгоритм проверки на коллизию при помощи регионов Вороного в двумерном варианте имеет следующий вид:

1. Выбрать одна из вершин многоугольника как начальная – A.
2. Для вершины A вычислить вектор $\acute{a}$, началом которого является

вершина A, а концом центр круга C.

3. Выбрать ребро b многоугольника, начинающееся в вершине, по часовой стрелке.
4. Проанализировать принадлежность центра круга C к региону Вороного ребра b на основании его векторного представления $\acute{b}$ по формуле (1):

$$region = \begin{cases} ¿\ scalar < 0 \\ middle, 0 \le scalar \le len2, \\ ¿, \wedge scalar \ge len2 \end{cases} \quad (1)$$
$$где\ scalar = (\acute{a}, \acute{b}), len2 = (\acute{b}, \acute{b})$$

5. В зависимости от того, какой результат дала формула произвести следующие действия:
   А. Если $region = ¿$, выбрать ребро и вершину против часовой стрелки относительно рассматриваемых и вычислить вектор a'_{prev} аналогично $\acute{a}$. К полученному вектору a'_{prev} и векторному представления ребра b'_{prev} применить формулу (1).
      а. Если $region = ¿$, значит центр круга принадлежит региону Вороного вершины A. Необходима дополнительная проверка на расстояние между вершиной A и центром круга по формуле (2). Если $collision = true$, то коллизия присутствует, если $collision = false$, то коллизия отсутствует. Итерация на этом прекращается.

$$collision = \begin{cases} true, |\acute{a}| \le r \\ false, |\acute{a}| > r \end{cases}, где\ r - радиус\ круга \quad (2)$$

      б. Другие случаи игнорируются, так как они будут рассмотрены далее по итерации.

   Б. Если $region = ¿$, то выбрать ребро и вершину A_{next} по часовой стрелке относительно рассматриваемых и вычислить вектор a'_{next} аналогично $\acute{a}$. К полученному вектору a'_{next} и векторному представления ребра b'_{next} применить формулу (1).
      а. Если $region = ¿$, значит центр круга принадлежит региону Вороного вершины A_{next}. Аналогично шагу 5.А.а проделать те же операции с заменой A на A_{next} и $|\acute{a}|$ на $|a'_{next}|$.
      б. Другие случаи игнорируются, так как они будут рассмотрены далее по итерации.

   В. Если $region = middle$, то это значит, что центр круга принадлежит региону Вороного ребра b. Для проверки на коллизию

необходимо вычислить расстояние от центра круга до ребра b по формуле (3). Если дистанция меньше, чем радиус круга, то коллизия присутствует, иначе она отсутствует. Итерация на этом заканчивается.

$$distance = \frac{\left|(\acute{a}, \acute{b})\right|}{\left|\acute{b}\right|} \tag{3}$$

6. Если итерация не была закончена, выбрать следующую вершину A по часовой стрелке и перейти к шагу 2.

Геометрическое представление алгоритма представлен на рисунке 3.15.

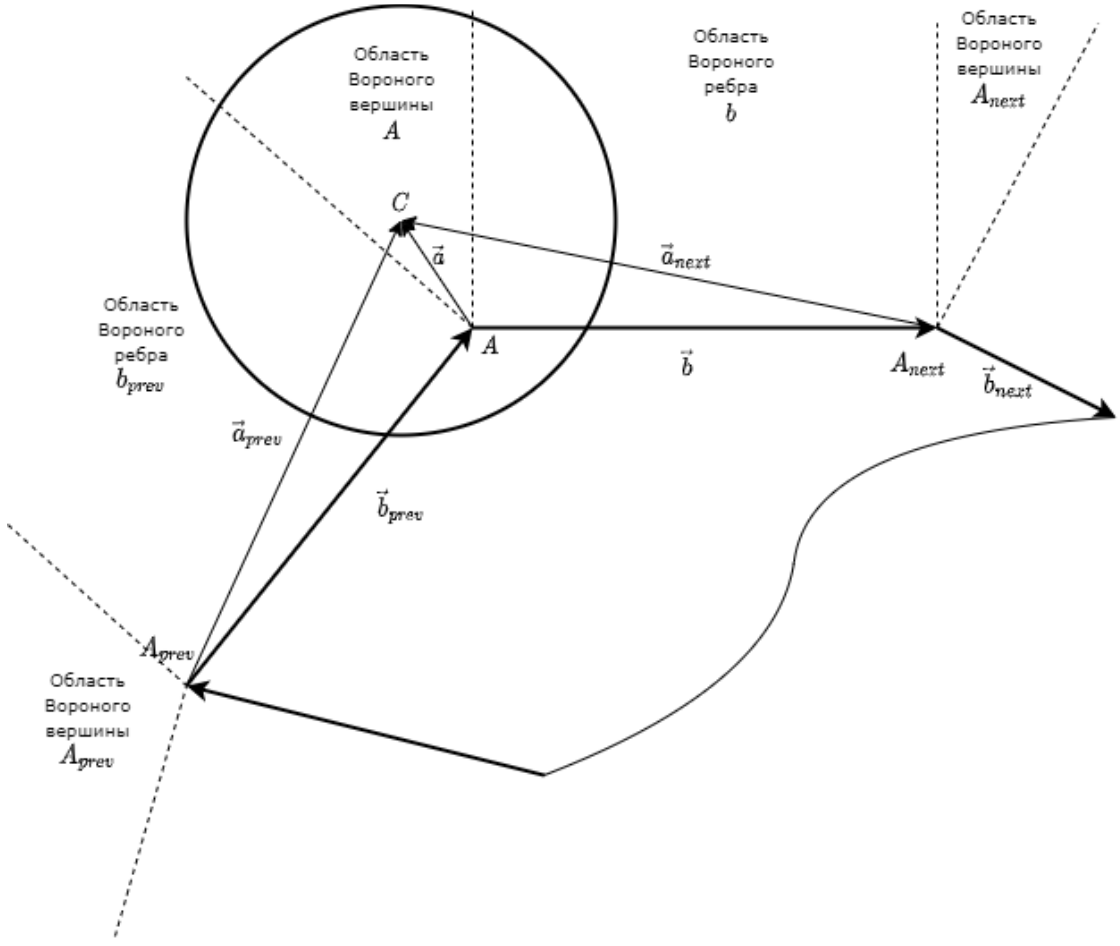


Рисунок 3.15 – Геометрическое представление алгоритма

Коллизия 2-х выпуклых многоугольников (полигонов). Данный алгоритм базируется на теореме о разделяющей гиперплоскости (для двумерной геометрии – оси) – две выпуклые геометрии не пересекаются тогда и только тогда, когда существует гиперплоскость, которая их разделяет. Ортогональная ось разделяющей гиперплоскости называется разделяющей осью, а проекции фигур на нее не пересекаются.

Соответственно, для того чтобы определить в двумерном случае

пересекаются ли 2 выпуклых многоугольника, необходимо проверить, не существует ли для них разделяющей прямой. Аналогом можно привести попытку найти стену, на которой тени многоугольников, висящие перед этой стеной, не пересекаются.

Так как количество возможных осей бесконечно для построения проекций обычно используют нормали к рёбрам многоугольников. Если проекции хотя бы на одну из нормалей не пересекаются, значит данные многоугольники также не пересекаются. Поэтому алгоритм для многоугольников A и B выглядит следующим образом:

1. Выбрать один из многоугольников.
2. Выбрать одно из его рёбер a, которое не было рассмотрено до этого.
3. Вычислить вектор нормали $\acute{n}$ к векторному представлению ребра $\acute{a}$ – если вектор $\acute{a}$ имеет координаты (x, y), то вектор нормали $\acute{n}$ будет иметь координаты $(\frac{y}{\sqrt[2]{x^2+y^2}}, \frac{-x}{\sqrt[2]{x^2+y^2}})$.
4. Вычислить минимальные и максимальные коэффициенты для проекции каждого из многоугольников на вектор нормали по формуле (4).

$$projection = ¿ \tag{3}$$

где $\acute{v} \in V$, а V – это множество радиус-векторов вершин многоугольника.

Если важна оптимизация, можно заменить шаги 3 и 4 на следующую формулу, в которой не придётся использовать квадратный корень для нахождения нормали (4):

$$projection = ¿ \tag{4}$$

где $\acute{v} \in V$, V – это множество радиус-векторов вершин многоугольника, а $\acute{n}$ – нормаль к $\acute{a}$.

5. Получив $projectionA = (minA, maxA)$ и $projectionB = (minB, maxB)$, необходимо провести сравнение (5):

$$separating = \begin{cases} true, minA > maxB \vee ¿ minB > maxA \\ false, иначе \end{cases} \tag{5}$$

Если $separating = true$, значит была найдена разделяющая ось – многоугольники не пересекаются и итерация заканчивается. Иначе требуется продолжить поиск.

6. Если остались нерассмотренные рёбра многоугольника, перейти к шагу

2. Иначе выбрать другой многоугольник и перейти к шагу 2. Если оба многоугольника были полностью рассмотрены, значит они пересекаются.

Реализация данных алгоритмов находится в репозитории [19].

3.4. Тестирование серверной части

Для тестирования функционала серверной части были использованы 2 метода: функциональное тестирование на уровне модулей (юнит-тестирование) – для общей проверки на дефекты, а также функциональное тестирование на уровне компонент – для тестирования взаимодействия между микросервисами GameService и Matchmaker. Юнит-тестирование также включено в процесс CI для каждого из репозиториев для автоматической проверки на дефекты перед сборкой.

Тестирование микросервиса UsersService представляет собой особый интерес, ведь данный микросервис в каждом методе внешних интерфейсов имеет вызовы к базе данных. Для создания имитационной базы данных был использован модуль Golang go-sqlmock. Однако данный подход не позволяет устраивать специальные сбои при взаимодействии с базой данных, что приводит к плохому покрытию некоторых ошибочных путей в коде (рис. 3.14). Кроме того, данный метод плохо совместим с ORM-библиотекой, так как для имитации ответов из базы данных необходим сам SQL-запрос, на который данный ответ должен прийти, что заставляет разработчика вручную переписывать SQL-запросы, что является весьма затратным по времени занятием.

Тестирование GameService является достаточно прямолинейным. Внутренняя логика GameSession, отвечающая за все расчёты состояния игры, отлично покрывается unit-тестами. Внешний интерфейс, предоставляемый GameManager также требует лишь создания тестового GRPC-сервера, с которым юнит-тесты будут взаимодействовать с целью нахождения дефектов взаимодействия (рис. 3.15).

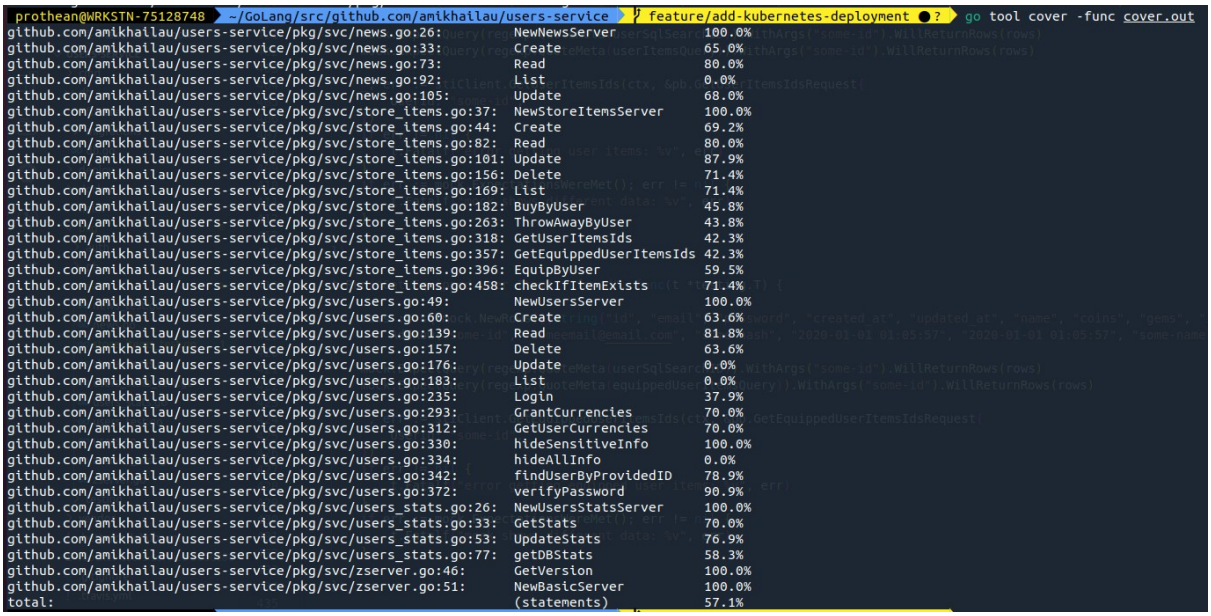


Рисунок 3.16 – Покрытие юнит-тестами методов внешних интерфейсов микросервиса UsersService

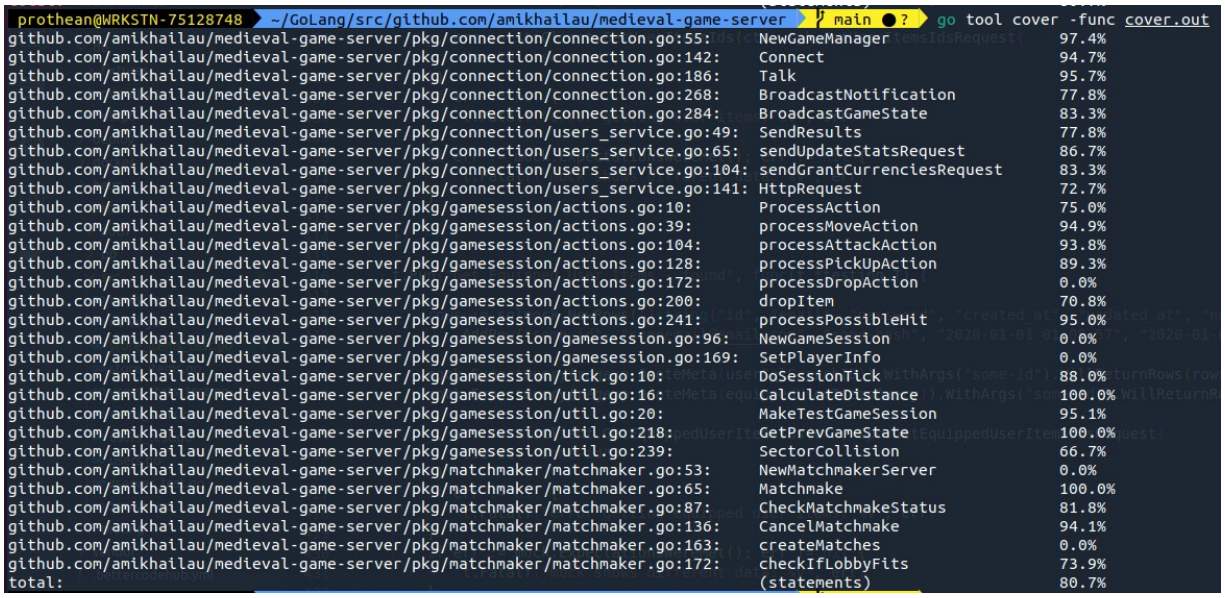


Рисунок 3.17 – Покрытие юнит-тестами функционала микросервисов GameService и Matchmaker

Однако сложности возникают при попытке тестирования синхронизации с целью имитировать ситуацию, когда несколько игроков взаимодействуют с сервером. Для решения этой проблемы было создано описание небольшой тестовой карты, на которую могут быть помещены 2 и более фиктивных игрока, которыми будут управлять юнит-тесты, описывающие игровой сценарий, с целью выявления проблем синхронизации, такие как тупики или состояния гонки. Таким же образом тестируется и интеграция компонент GameSession и GameManager.

Для тестирования интеграции Matchmaker и GameService без взаимодействия с OpenMatch был написан отдельный тестовый клиент, который создаёт несколько фиктивных JWT для имитации нескольких клиентов, посылает со всех имитированных клиентов запросы на участие в игре в Matchmaker и после этого на основе полученных данных подключает все имитационные клиенты, для которых была создана сессия игры, к серверу (рис. 3.16).

Как можно видеть, при первом запросе о готовности сессий игр сразу же после отправки запроса на участие в игре в ответ приходит информация о том, что игра ещё не готова, даже несмотря на небольшую задержку перед отправкой запроса. Это обусловлено частотой стандартного алгоритма организации игр. При локальном развёртывании он установлен в 2 секунды. Однако после второго запроса, который произошёл через 5 секунд после первого, 4 клиента получили в ответ информацию об адресе сервера, на который следует подключиться для участия в игре, в то время как 1 из них не получил ничего. Стоит заметить, что клиенты получили одинаковую информацию по парам, так как размер сессии игры при локальном развёртывании установлен в 2 игрока.

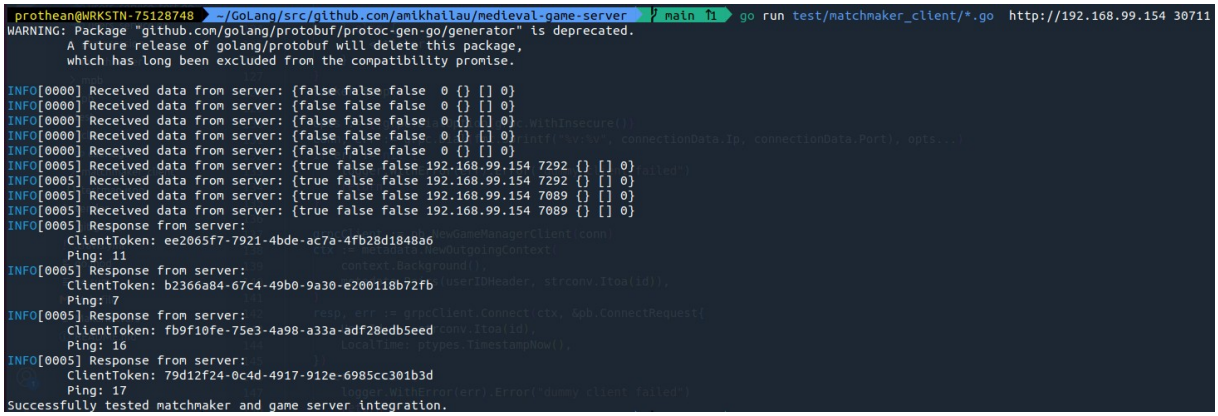


Рисунок 3.18 – Интеграционное тестирование взаимодействия Matchmaker и GameService

Данное тестирование требует локального кластера Kubernetes с развёрнутыми в нём микросервисами Matchmaker и GameService, а также сервисами платформы Agones. Для этого используется minikube - инструмент, который применяют для проведения локальных экспериментов с Kubernetes, представляющий собой несколько облегчённую версию самого Kubernetes (рис. 3.17).

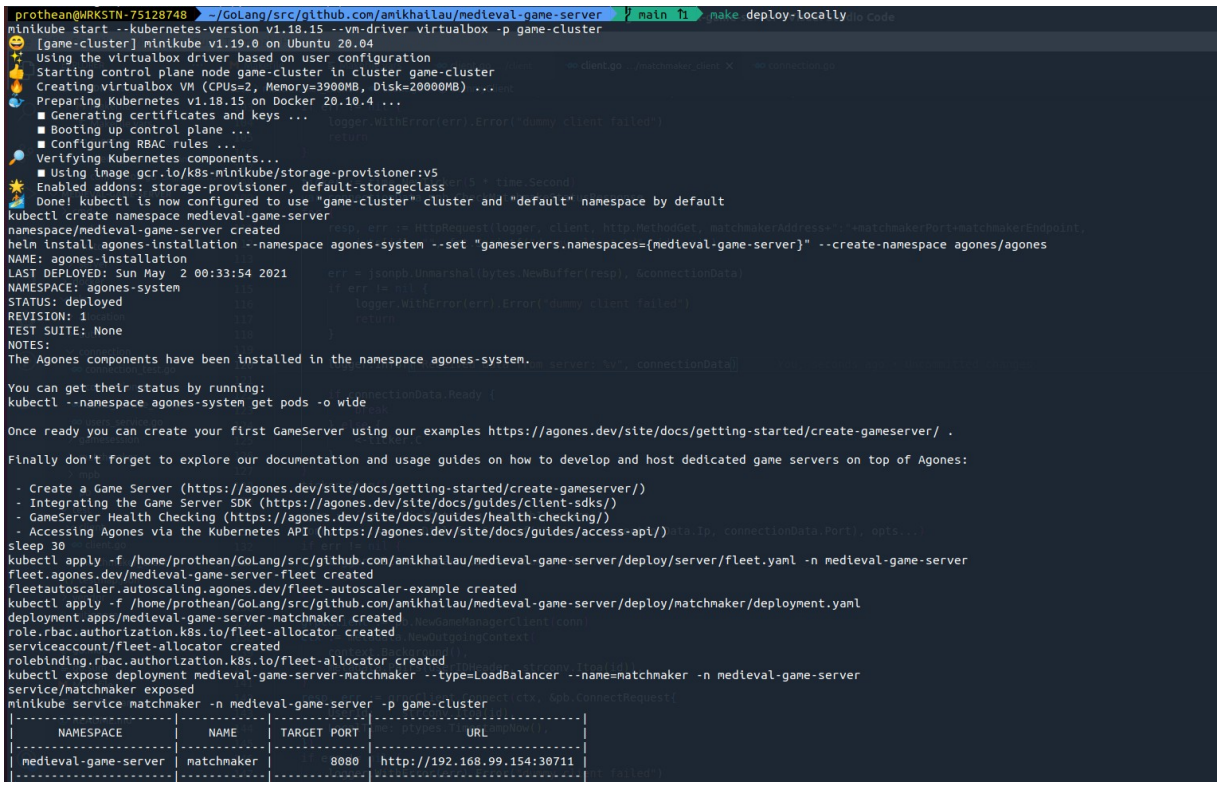


Рисунок 3.19 – Локальное развёртывание Matchmaker и GameService

ВЫВОДЫ

1. Проведён анализ требований к серверной части проекта Medieval.io.
2. Создан макет архитектуры серверной части.
3. Выделены контексты для компонент микросервиса UsersService. Создана схема базы данных. Спроектированы внешние интерфейсы, предоставляемые UsersService. Создано описание структур данных и интерфейсов, используемых в UsersService, на языке protobuf.
4. Проведён анализ проблем масштабирования серверной части игры. Предложено возможное решение данных проблем. Проведён поиск подходящих технологий с функционалом, похожим на функционал возможного решения.
5. Проведён анализ проблем организации сессий игр. Предложено возможное решение для организатора игр. Проведён поиск

подходящих технологий с функционалом, похожим на функционал возможного решения.

6. Спроектированы внешние интерфейсы, предоставляемые микросервисами GameService и Matchmaker. Создано описание структур данных и интерфейсов, используемых в GameService и Matchmaker, на языке protobuf.
7. Разработан микросервис UsersService. Рассмотрен шаблон «интерцептор» для авторизации запросов.
8. Разработан микросервис Matchmaker.
9. Разработан микросервис GameService.
10. Рассмотрены алгоритмы широкой фазы обнаружения коллизий и реализован алгоритм Sweep and Prune для GameService.
11. Реализованы алгоритмы узкой фазы обнаружения коллизий для GameService.
12. Проведено покрытие юнит-тестами функционала всех микросервисов. Проведено интеграционное тестирование взаимодействия Matchmaker и GameService.

ГЛАВА 4. РАЗВЁРТЫВАНИЕ СЕРВЕРНОЙ ЧАСТИ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ НА МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ

1
2
3
4

4.1. Контейнеризация. Docker

Как уже упоминалось в Главе 1, микросервисы должны развёртываться независимо друг от друга, быть изолированными. Кроме того, довольно важным свойством микросервисов является их гибкость – в одном проекте может быть множество микросервисов, написанных на абсолютно разных языках программирования, и их развёртывание может проводиться в различных средах окружения.

Для реализации данного свойства чаще всего используется или виртуализация, или контейнеризация. В данной работе была использована контейнеризация при помощи Docker.

4.1.1. Контейнеризация

Контейнеризация – это легковесная виртуализация на уровне операционной системы, которая позволяет запускать приложения и его зависимости как изолированный процесс. Все необходимые компоненты для запуска приложения собраны в одном месте – образе контейнера – и могут быть повторно использованы. Образ приложения запускается в изолированной среде окружения и не использует память или процессор хостовой операционной системы.

Чем же контейнеризация отличается от виртуализации? В виртуализации обычно используются полноценные операционные системы (гостевая ОС), на которых запускаются приложения, из-за этого размеры виртуальных машин легко достигают нескольких гигабайт. Контейнеры же в свою очередь являются легковесными и содержат в себе только то, что необходимо для запуска приложения, что позволяет им иметь размер в несколько сотен мегабайт. Для того чтобы запустить виртуальную машину происходит эмулирование аппаратного окружения, в то время как контейнер может быть запущен только с таким же ядром, что и у хостовой операционной системы. Из-за необходимости запускать полноценную операционную систему виртуальные машины очень

медленно запускаются, в то время как контейнеры способны запускаться и инициализировать приложение почти моментально.

Преимущества контейнеризации:

- быстрое развёртывание;
- высокая производительность;
- возможность контроля версий образов;
- портативность – один и тот же образ может запускаться как на Windows, так и на почти всех дистрибутивах Linux;
- безопасность – из-за изолирования всех процессов внутри контейнера изменения в одном контейнере не повлияют на другой контейнер.

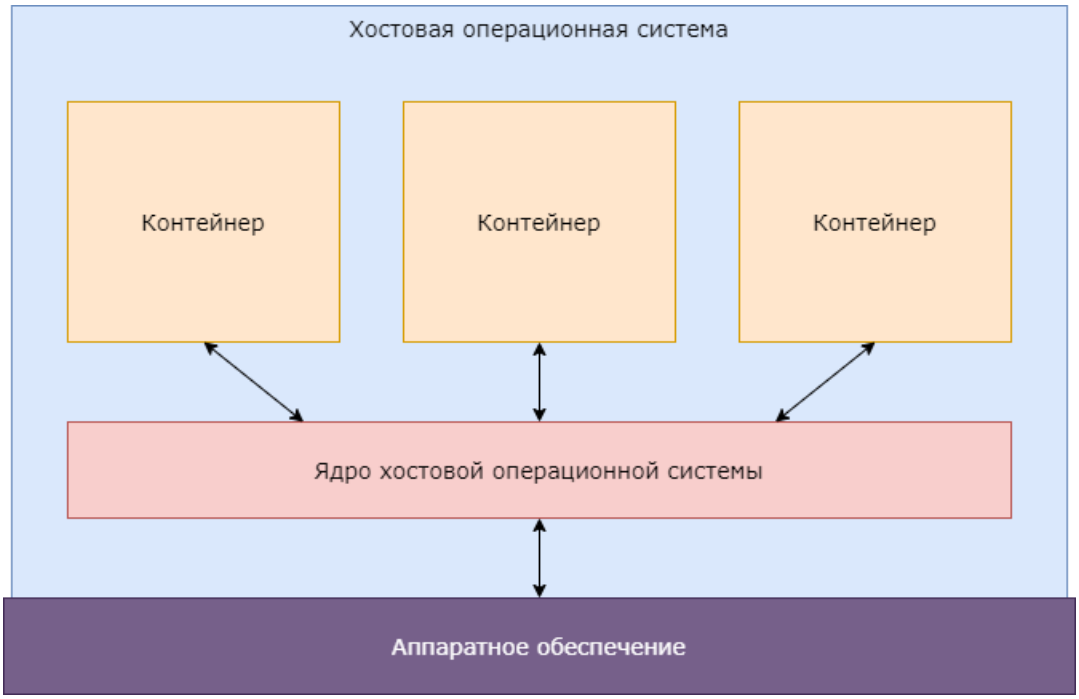


Рисунок 4.1 – Контейнеризация

Недостатки контейнеризации:

- повышение сложности развёртывания;
- хоть Windows и поддерживает контейнеризацию, всё-таки сама технология (и тот же Docker) базируются на Linux контейнерах, что приводит к некоторым проблемам при развёртывании контейнеров на Windows.

Повышение сложности развёртывания сильно отталкивала разработчиков первое время, ведь приходилось работать напрямую со средствами операционной системы Linux – cgroups и namespace, которые отвечают за управление и изолирование ресурсов процессов. Однако со временем появились технологии, совершающие те же самые действия за них при помощи единого командного интерфейса. Одной из таких технологий является Docker.

4.1.2. Контейнеризация при помощи Docker

Docker – это программное обеспечение, написанное на Golang, позволяющее создавать контейнеры – упакованные приложения со всеми необходимыми настройками окружения и зависимостями – и предоставляющее сервисы для управления и развёртывания данных контейнеров.

Для каждого контейнера существует соответствующий образ. Образы являются своеобразными шаблонами, содержащими окружение программы и саму программу. Образы состоят из слоёв, что позволяет при изменении программы не пересоздавать образы полностью, а лишь изменить тот слой, в котором содержалась часть, подвёргнутая изменению.

Для хранения данных образов Docker предоставляет DockerHub – реестр образов, отдалённо напоминающий систему контроля версий. Для каждой программы создаётся отдельный репозиторий, в котором хранятся различные версии образа. Если docker не найдёт необходимый образ для создания контейнера в локальном реестре, он обратится к DockerHub для его скачивания.

При работе с Docker пользователь не использует средства операционной системы напрямую, а вместо этого взаимодействует с единым командным интерфейсом. Данный интерфейс позволяет обращаться к docker-клиенту, который получает данные команды и контактирует с docker-демоном, которым запускает новые контейнеры и управляет существующими.

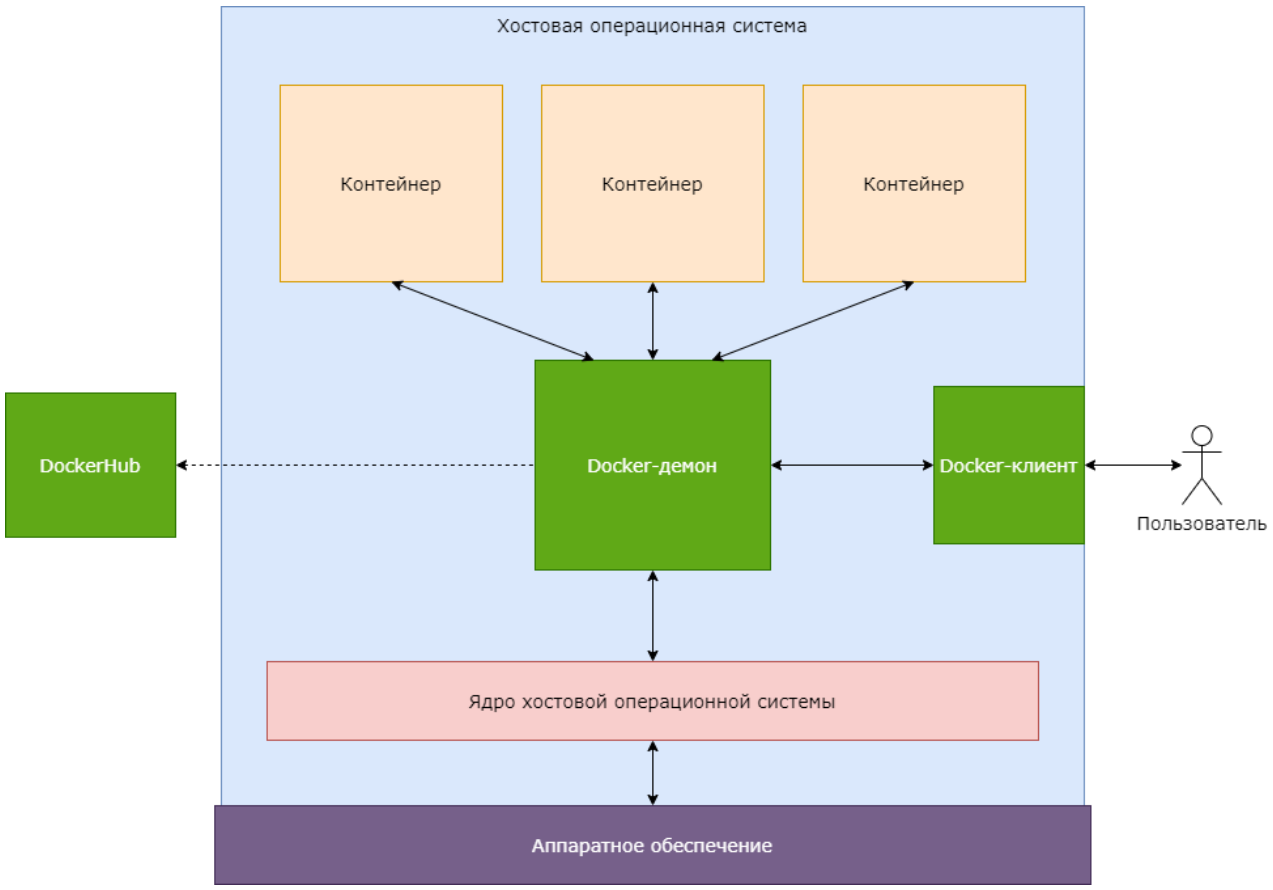


Рисунок 4.2 – Docker

Для создания нового образа необходимо создать файл Dockerfile, в котором будет прописан базовый образ, на основе которого будет создаваться новый, а также прописаны все инструкции, которые необходимо совершить docker для создания контейнера и запуска самой программы внутри контейнера. Dockerfile, написанный для контейнеризации сервера игры – микросервиса GameService – можно найти в Приложении З.

Данный Dockerfile разделён на 2 части. В первой используется базовый образ golang для сборки бинарного файла сервера игры в промежуточном контейнере, что можно видеть из инструкций FROM (объявление базового образа), COPY (копирование файлов из памяти хоста в контейнер) и RUN (выполнение команды через командную строку). После сборки бинарного файла он копируется в другой контейнер, базовым образом для которого выступает alpine – более легковесный чем golang образ, в котором и запускается полученный бинарный файл. За запуск отвечает инструкция ENTRYPOINT.

4.2. Оркестратор контейнеров Kubernetes

Docker позволяет изолировать микросервисы друг от друга и создать среду окружения для их запуска. Но как быть если микросервисов много или если какие-то из них нужно масштабировать? Как распределять нагрузку между несколькими контейнерами одного и того же микросервиса? Здесь на помощь приходят оркестраторы контейнеров такие как Kubernetes.

4.2.1. Общий обзор Kubernetes

Kubernetes (от греч. κυβερνήτης – «кормчий», «рулевой») - это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными приложениями, являющаяся на текущий момент индустриальным стандартом [20]. Написана она, как и Docker, на Golang.

Kubernetes предлагает следующий функционал для оркестрации контейнеризированных приложений:

- мониторинг сервисов и распределение нагрузки – Kubernetes обнаруживает развёрнутые контейнеры и назначает им IP-адрес. Если нагрузка на определённый контейнер оказывается слишком высокой, Kubernetes старается распределить его между всеми такими же контейнерами для стабильной работы сервиса;
- оркестрация хранилищ – при необходимости внешнего хранилища для контейнеризированного приложения, Kubernetes предлагает

возможность монтирования как локального хранилища, например etcd или redis, а также использования облачных хранилищ, таких как AWS или GCP;

- автоматическое развёртывание и откаты – при помощи ресурсов Kubernetes таких как Deployment есть возможность описывать желаемое состояние контейнеров, а также изменять состояние уже развёрнутых контейнеров, что позволяет автоматизировать процесс развёртывания приложения;
- самоконтроль – в случае если какой-либо из контейнеров отказывает, в зависимости от политики перезапуска Kubernetes может самостоятельно удалить такой контейнер и создать вместо него новый. Кроме того, Kubernetes имеет возможность добавить проверку работоспособности контейнера, и в случае невыполнения данной проверки контейнер не будет использован для распределения нагрузки;
- автоматическое распределение нагрузки – при создании кластера Kubernetes требуется выделить несколько вычислительных узлов для него. Благодаря указанию количества ресурсов таких как память и количество ядер процессора, необходимых для контейнера, Kubernetes может максимально эффективно распределить нагрузку между вычислительными узлами;
- управление конфиденциальной информацией и конфигурацией – за счёт таких ресурсов как Secret и ConfigMap Kubernetes способен хранить конфиденциальную информацию, такую как JWT-токены, пароли от баз данных и ключи SSH, и конфигурации контейнеризованных приложений прямо на кластере, что позволяет не раскрывать данную информацию в репозиториях контроля версий или сервисах непрерывной интеграции и развёртывания.

Архитектура Kubernetes выглядит достаточно интересно. С одной стороны есть мастер-планировщик, к которому можно обращаться через командную строку при помощи команды kubectl, а с другой стороны узлы, на которых развёртываются все необходимые ресурсы для приложения, управляемые через kubelet-агент мастером-планировщиком. На каждом узле настраивается kube-proxy, который занимается перенаправлением запросов на поды, в которых развёрнуты контейнеры микросервисов. Если посмотреть на рисунок 4.3, можно увидеть, что Kubernetes сам по себе имеет микросервисную архитектуру.

Определение слова «*под*» (Pod) не просто подразумевает под собой совокупность развёрнутых контейнеров, под – это представление запроса на запуск одного или более контейнеров. То есть просто так контейнер сам по себе не может существовать в архитектуре Kubernetes, он всегда принадлежит

какому-то поду. Чаще всего один под содержит в себе контейнеры одного микросервиса с контейнерами вспомогательных сервисов, необходимых данному контейнеру, например контроллеры, кэши (redis) или базы данных.

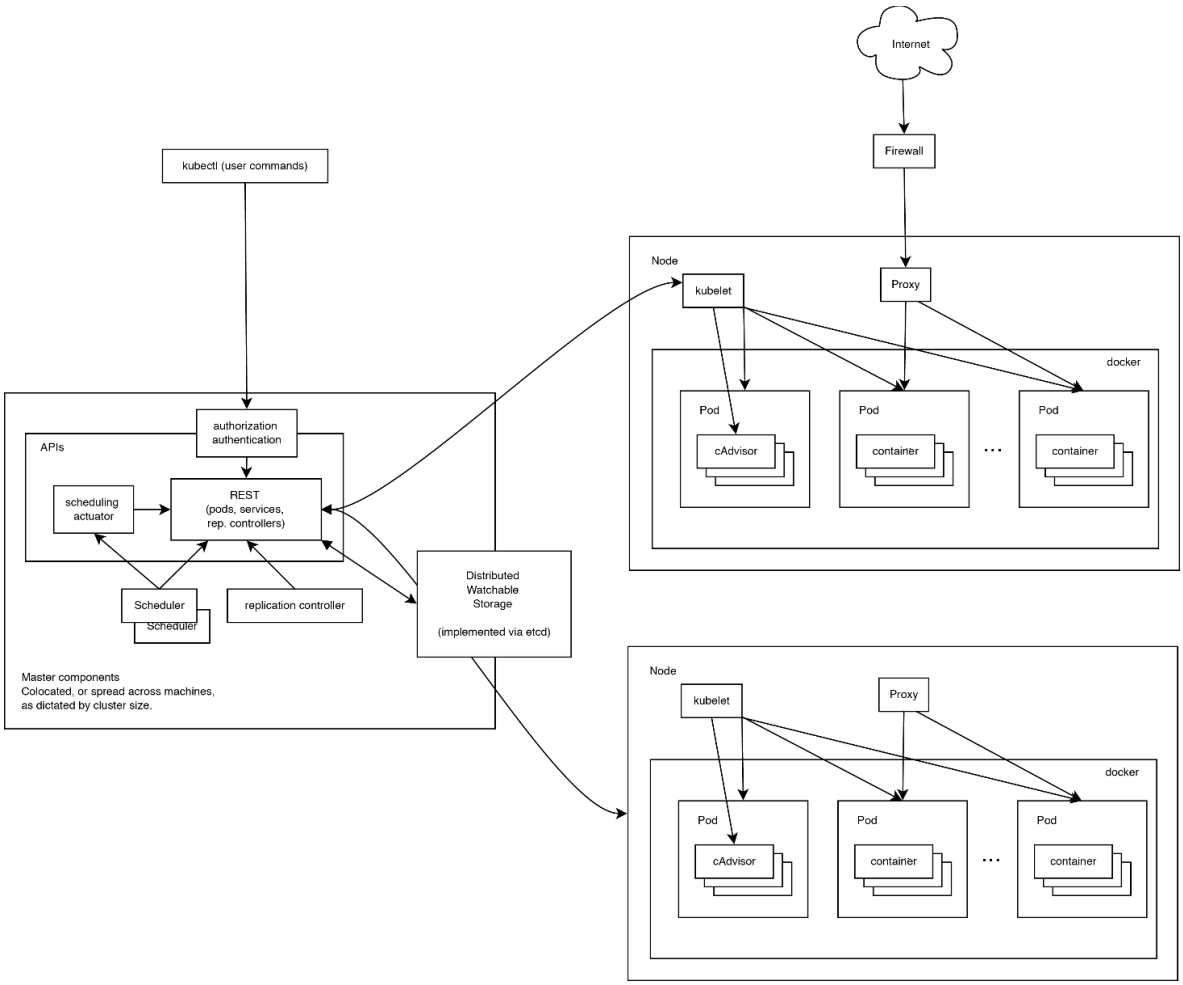


Рисунок 4.3 – Архитектура Kubernetes [24]

4.2.2. Ресурс Deployment

Весь описанный функционал будет в том или ином виде задействован в данной работе при развёртывании как сервера игры, так и вспомогательных микросервисов таких как Matchmaker. Пример ресурса Deployment в формате YAML для микросервиса UsersService можно найти в Приложении И.

Стоит отметить, хоть Deployment и не предоставляет динамического масштабирования в зависимости от нагрузки, её предлагает другой ресурс – Horizontal Pod Autoscaler. Так как нет разницы, конкретно какой экземпляр Matchmaker или UsersService обработает запрос, нет необходимости использования Agenos для развёртывания данных микросервисов. Horizontal Pod Autoscaler изменяет количество экземпляров сервера в зависимости от нагрузки на процессор или от другой метрики, обозначенной пользователем.

Данный ресурс имеет следующие поля:

- metadata описывает имя Deployment ресурса, пространство имён, в которое он будет помещён, а также пометки для данного ресурса;
- spec.selector отвечает за указание пометок *под*, которыми данный ресурс будет управлять;
- spec.replicas указывает на то, сколько *под* должно быть развёрнуто;
- spec.template используется для описания самих *под*, которые будут развёрнуты. spec.template.labels используется для пометок, которые были упомянуты чуть выше – за счёт них Kubernetes понимает, за какие *поды* отвечает данный ресурс;
- spec.template.spec описывает конфигурацию *под*. spec.template.spec.restartPolicy определяет политику перезапуска, о которой речь шла немного раньше;
- spec.template.spec.containers указывает какие контейнеры должны будут развёрнуты в *подах*. Для каждого контейнера указывается тег образа, который нужен для создания контейнера, порты, которые использует контейнер, параметры конфигурации для контейнера, а также ресурсы, которые нужно выделить данному контейнеру, и лимит выделяемых ресурсов.

4.3. Платформа для развёртывания серверов игр Agones

4.3.1. Общий обзор Agones

В ходе поиска подходящих решений для масштабирования и оркестрирования контейнеров сервера игры была рассмотрена такая технология как Agones, которая удовлетворяла всем критериям поиска.

Agones (от греч. agōn – «соревнование», «состязание») – это платформа с открытым исходным кодом для развёртывания, хостинга, масштабирования и оркестрации игровых серверов для крупномасштабных многопользовательских игр, построенная на современном стандарте облачной индустрии – платформы Kubernetes. Данная платформа не является инструментарием для разработки самой логики игры, её фокус находится в создании инфраструктуры для развёртывания и поддержки серверов игры, что позволяет разработчикам игр сконцентрироваться на реализации логики.

Заявленные способности и преимущества Agones:

- снижение затрат при разработке и эксплуатации хостинга, масштабирование и оркестрация многопользовательских игровых серверов;
- любой сервер игры, который можно запустить на Linux, можно развернуть при помощи Agones;

- так как Agones по сути является надстройкой над Kubernetes, его можно размещать везде, где можно разместить кластер Kubernetes – например в облаке или на персональном компьютере. Кроме того, все базовые функции и возможности Kubernetes также будут доступны для использования;
- серверы игр и любые другие вспомогательные сервисы могут быть размещены, используя единую платформу, что позволяет упростить архитектуру серверной части;
- Agones является бесплатной платформой с открытым исходным кодом;
- SDK Agones предоставляет возможность управления жизненного цикла сервера, в том числе проверку состояния, управление состоянием, точную настройку, метрики активности и другое;
- за счёт использования Docker контейнеров, достигается изоляция экземпляров сервера игры друг от друга.

Agones состоит из следующих трёх компонент:

- GameServer контроллер отвечает за выделение IP-адресов и портов экземплярам сервера, создание таких ресурсов Kubernetes как Pod, которые в свою очередь ответственны за развёртывание каждого из экземпляров, и управление жизненным циклом данных ресурсов;
- Allocation контроллер отвечает за выделение экземпляров сервера для обработки игровых сессий;
- SDK контроллер, который отвечает за изменение состояния экземпляров при получении сообщений от самих экземпляров, проверку их состояния и удаление ненужных или отработавших экземпляров. В самой последней версии Agones экземпляр данного контроллера создаётся вместе с экземпляром сервера в одной *поде*.

Для интеграции с Agones необходимо использовать специально предназначенное для этого SDK. Пример его использования для микросервиса GameService находится в Приложении К.

Описание функций SDK в примере:

- Ready() – при вызове этой функции экземпляр сервера сообщает о том, что он готов принимать соединения клиентов. Как только данная функция вызывается, GameServer контроллер добавляет данные о IP-адресе и порте, выделенного данному экземпляру, в его описание и изменяет его состояние на Ready. Также данная функция позволяет перевести экземпляр из состояния Allocated в Ready;
- Health() – при вызове этой функции экземпляр сервера посылает сигнал SDK контроллеру чтобы показать, что данный экземпляр находится в рабочем состоянии. Если данные сигналы перестают

поступать, через некоторое время, определённое в конфигурации сервера игры, экземпляр сервера будет помечен как нерабочий;
- Shutdown() – экземпляр, вызвавший данную функцию, оповещает Agones, что он завершил работу и его необходимо удалить, после чего удаляется соответствующий ресурс Kubernetes Pod.

4.3.2. Развёртывание сервера игры при помощи Agones

Для развёртывания сервера необходимо объявить соответствующие ресурсы, описывающие настройки развёртывания и самого сервера игры.

Таким ресурсом является Fleet, пример которого можно найти в Приложении Л. Описание параметров ресурса:

- replicas указывает на количество экземпляров сервера, которые должны быть развёрнуты;
- scheduling отвечает за то, как будут развёрнуты экземпляры, в случае, если кластер Kubernetes имеет несколько узлов – «Packed» означает, что как можно больше экземпляров будут развёрнуты на одних и тех же узлах, в то время как «Distributed» будет стараться расположить экземпляры на как можно большем количестве узлов;
- в strategy описывается стратегия обновления сервера. Всего существует 2 стратегии: RollingUpdate и Recreate. Recreate удаляет все не выделенные экземпляры и создаёт новые экземпляры, работающие на обновленной версии сервера, в то время как RollingUpdate будет обновлять экземпляры по очереди в количестве, зависящего от остальных параметров – maxSurge и maxUnavailable;
- template является шаблоном для создания GameServer ресурса, который содержит информацию о конфигурации самого сервера игры. В данном параметре указываются порты, которые использует сам контейнер сервера игры, и стратегия выделения внешних портов (ports), настройки оповещения о рабочем состоянии экземпляра (health), порты для Agones SDK (sdkServer) и описание Kubernetes Pod, (template), который необходимо развернуть.

Для развёртывания сервера игры данный ресурс необходимо разместить в кластере Kubernetes. Как только хотя бы один экземпляр сервера перейдёт в состояние Ready, сервер готов к обработке запросов клиентов.

Кроме ресурса Fleet в приложении также указан ресурс FleetAutoscaler. Данный ресурс позволяет автоматизировать масштабирование сервера игры в зависимости от нагрузки на сервер и какое количество существующих экземпляров сервера игры уже выделено для обработки сессий игр. Для FleetAutoscaler типа Buffer должны быть указаны следующие параметры:

- bufferSize – количество невыделенных экземпляров сервера игры, которое необходимо поддерживать (когда это возможно);
- minReplicas – минимальное количество экземпляров сервера игры, которое должно быть развёрнуто в любой момент времени;
- maxReplicas – максимальное количество экземпляров сервера игры, которое должно быть развёрнуто в любой момент времени.

Пример жизненного цикла экземпляра сервера можно найти на рисунке 4.4. В нём упоминается такой ресурс как GameServerAllocation, пример которого можно найти в Приложении М. Данный ресурс может содержать информацию для выбора определённого экземпляра сервера игры, а также метаданные, которые будут добавлены к метаданным выделенного экземпляра.

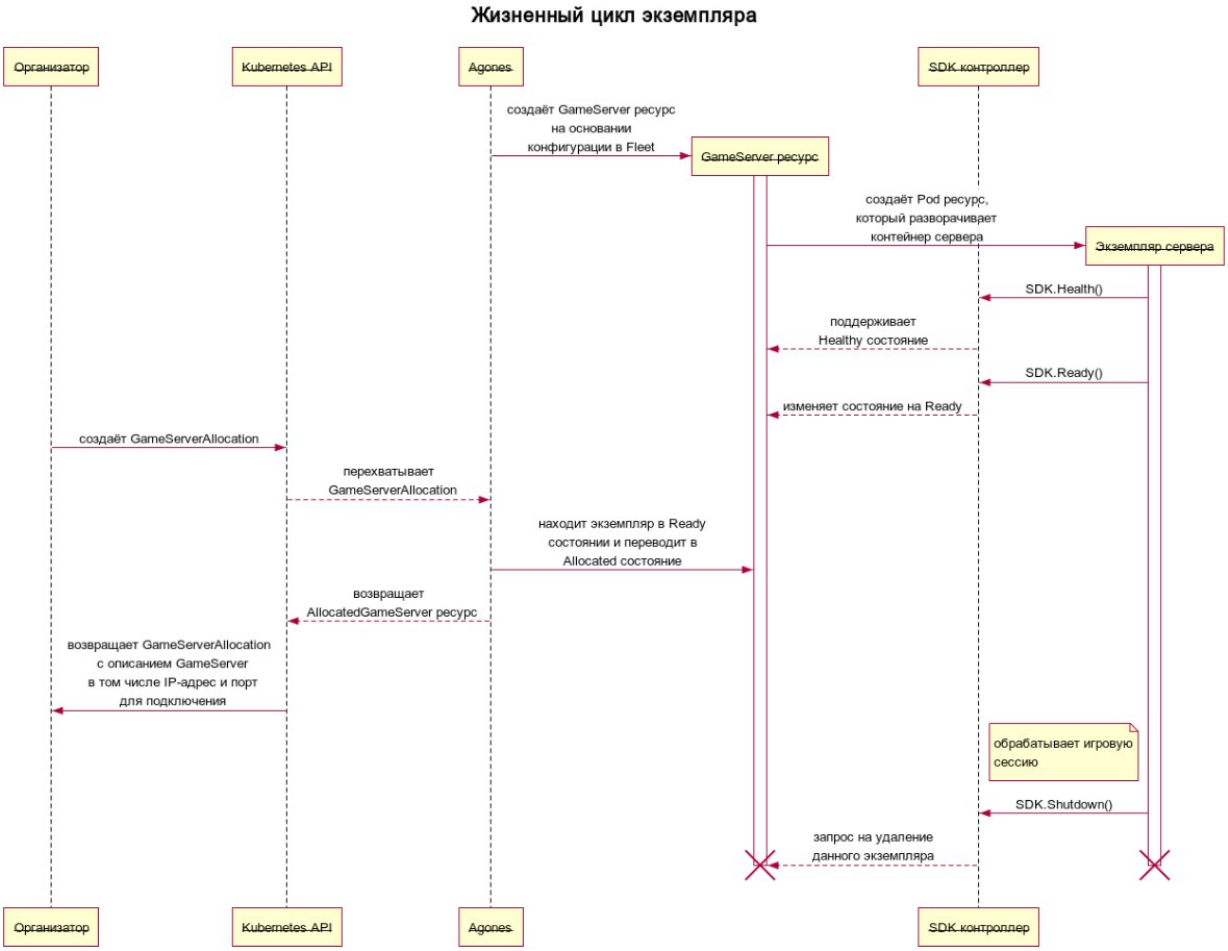


Рисунок 4.4 – Диаграмма последовательности для жизненного цикла экземпляра сервера

Данный пример подойдёт далеко не всем жанра многопользовательских игр, в его основе лежит идея создания сервера для игры жанра «королевская битва», где один экземпляр сервера обрабатывает одну игровую сессию и завершает свою работу, после чего для новых сессий будут создаваться новые экземпляры. В случае MMORPG, где необходимо сохранять состояние сервера в независимости от того, покинули ли сервер все игроки, жизненный цикл

будет выглядеть по-другому.

4.4. Организатор игр на платформе OpenMatch

4.4.1. Общий обзор OpenMatch

В ходе описания жизненного цикла экземпляра сервера в предыдущем пункте был затронут такой компонент, как организатор игр, необходимый для создания сессии игры на основе списка игроков, которые желают принять участие в игре. В процессе исследования существующих технологий была обнаружена платформа OpenMatch, которая удовлетворяла всем критериям поиска.

OpenMatch – это платформа с открытым исходным кодом, предназначенная для упрощения создания масштабируемого организатора игр. Она предоставляет инфраструктуру, которая позволяет решить такие проблемы как обработка большого потока игроков и эффективный и быстрый поиск игроков в очереди на основе поставленных критериев. Данная платформа, как и Agones, использует Kubernetes как платформу для развёртывания и поддержки инфраструктуры организатора игр.

OpenMatch состоит из набора сервисов, развёрнутых в Kubernetes кластере:

- FrontendService отвечает за создание, просмотр и удаление билетов – объекта, который представляет собой игрока или группу игроков, запрашивающих принять участие в игре;
- BackendService отвечает за создание матчей – списка игроков, которые будут участвовать в сессии игры, – а также получении от Director и передаче FrontendService назначений – информации о сервере, который будет обрабатывать сессию игры определённого списка игроков;
- QueryService занимается обработкой запросов на получение билетов, которые удовлетворяют определённым критериям;
- MatchFunction и Evaluator предоставляются разработчиками игры. Они занимаются созданием матчей по определённым критериям, для чего извлекается список игроков из QueryService и передаётся BackendService для его дальнейшего обработки. Для упрощения разработки OpenMatch предоставляет стандартный Evaluator, который подойдёт под большинство возможных ситуаций;
- GameFrontend подразумевает под собой компоненту игры, которая осуществляет аутентификацию игроков, отправку запросов на участие в игре и обработку полученной впоследствии информации об игровом сервере, который будет заниматься обработкой сессии

игры клиента, сделавшего запрос;

- Director занимается получением матчей от BackendService, отправкой запроса на выделение сервера для обработки новой сессии игры и впоследствии передачей информации о выделенном сервере BackendService. Данный сервис также предоставляется разработчиками игры.

Как упоминалось ранее, при развёртывании сервера игры с OpenMatch, микросервис Matchmaker будет выступать в роли Director и GameFrontend. Диаграмму компонент OpenMatch можно увидеть на рисунке 4.5.

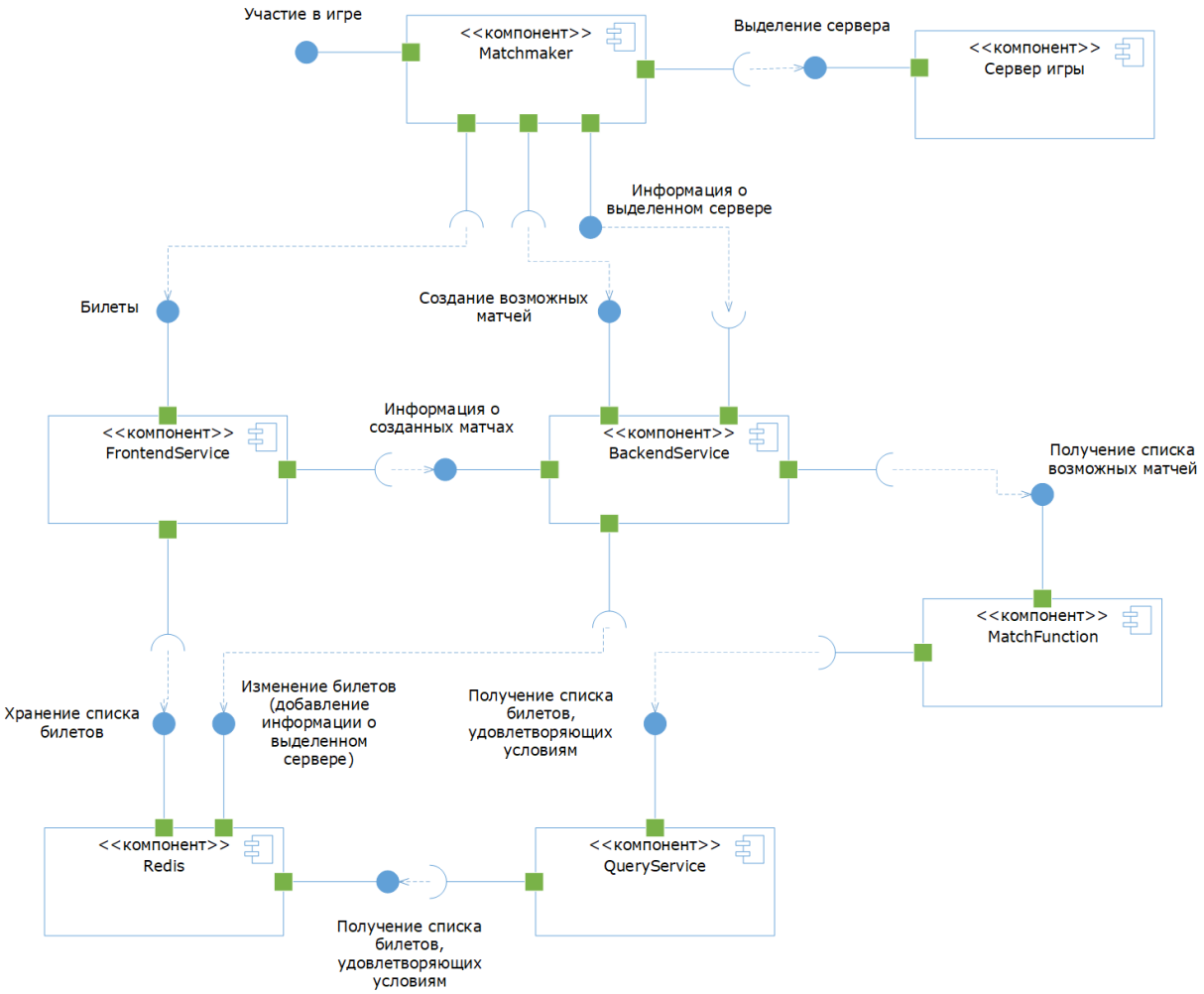


Рисунок 4.5 – Диаграмма компонент OpenMatch

Процесс организации игр с помощью OpenMatch в данной работе можно разбить на несколько шагов:

1. Matchmaker аутентифицирует пользователя и создаёт билет в OpenMatch.
2. Периодически Matchmaker посылает запросы на создание матчей на BackendService. BackendService при помощи MatchFunction и QueryService создаёт список возможных матчей и посылает их

Matchmaker.

3. Matchmaker запрашивает выделение сервера и передаёт информацию о выделенном сервере обратно в OpenMatch. После этого через FrontendService Matchmaker получает информацию для всех игроков, для которых была создана сессия игры.

Диаграмму последовательности с подробностями данного процесса можно увидеть на рисунке 4.6.

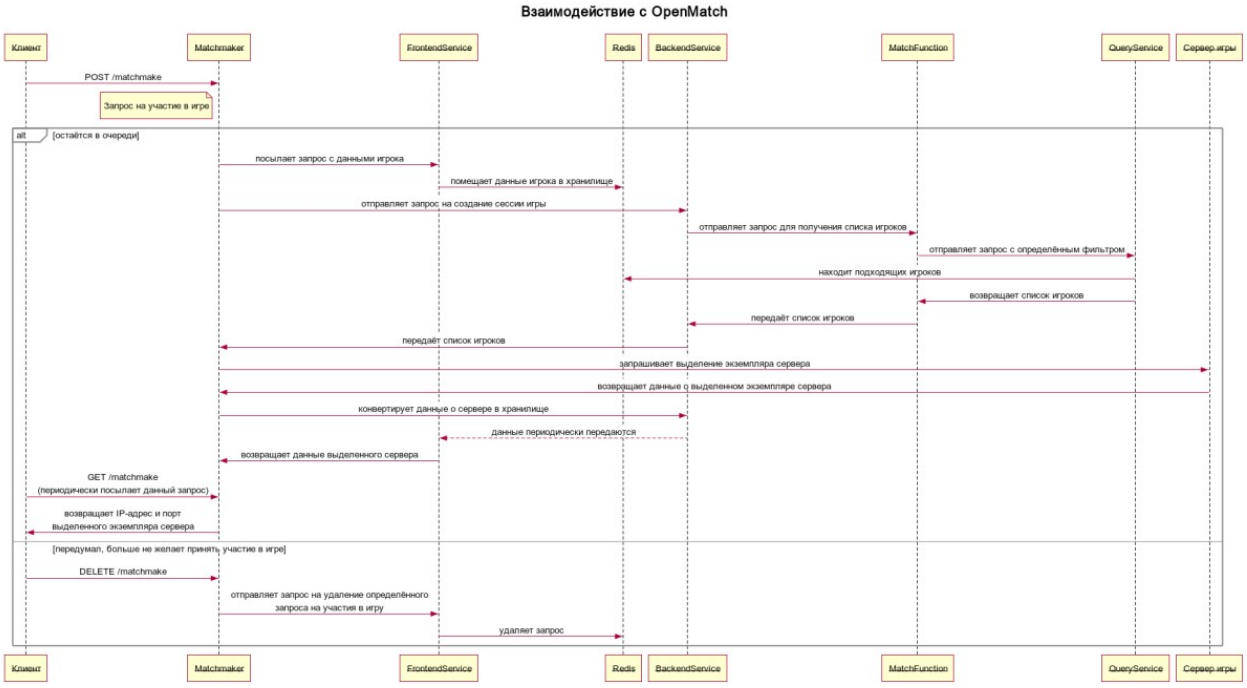


Рисунок 4.6 – Диаграмма последовательности организации игр при помощи OpenMatch

4.5. Развёртывание сервера игры на GKE

4.5.1. Услуга GKE облачной платформы GCP

Google Cloud Platform – это облачная платформа, предоставляющая множество различных услуг таких как облачные вычисления, хранение и анализ данных, машинное обучение и др. В частности, она предлагает такие услуги как «платформа как услуга» и «инфраструктура как услуга», которые и будут использованы в данной работе.

Для лучшего пояснения таких концептов как IaaS и PaaS приведен рисунок 4.7.

IaaS является самым низким уровнем обслуживания, предоставляемый облачными провайдерами. В данной модели обслуживания пользователю предоставляется базовая инфраструктура, которая обычно включает в себя такое аппаратное оборудование, как процессоры, графические процессоры,

сетевые кабели, оперативную память, хранилища данных (по сути, вычислительные узлы с доступом в сеть), а также базовые образы требуемых операционных систем. Данная модель обслуживания обычно необходим для высококвалифицированных разработчиков, которым необходима индивидуальная настройка большинства компонент для проведения определённых исследований или построения специфического сервера для производственных нужд.

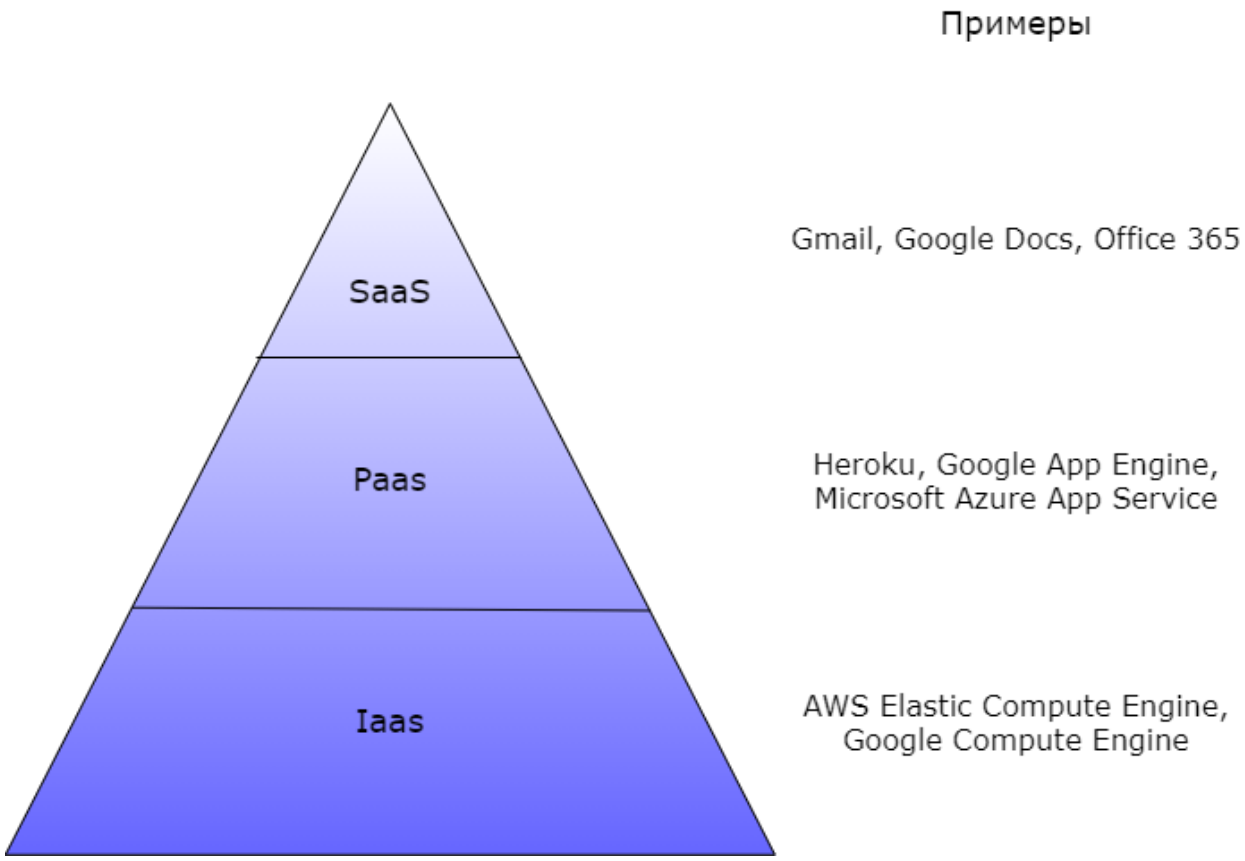


Рисунок 4.7 – Уровни обслуживания, предоставляемые облачными провайдерами

PaaS модель обслуживания предполагает под собой уже наличие определённой платформы, на основе которой можно собрать и развернуть необходимый проект. PaaS является своеобразной абстракцией над IaaS, где нет необходимости самостоятельно заниматься администрацией ресурсов вычислительных узлов, таких как время процессора, оперативная память, сеть и т.д. С одной стороны данная модель обслуживания предоставляет намного меньшую свободу над предоставленными вычислительными узлами, с другой стороны она намного проще и чаще всего прямой контроль ресурсов не нужен.

SaaS напрямую подразумевает предоставление необходимых сервисов конечному пользователю. Доступ к ним обычно происходит через веб-сайты – прямым примером являются Gmail и Google Docs.

Google Kubernetes Engine – это одна из услуг, предоставляемых GCP. Она

представляет собой абстракцию над Google Compute Engine, которая относится к IaaS, но между тем отнести её конкретно к IaaS или PaaS достаточно сложно. Иногда для данного типа услуг выносят отдельный уровень между IaaS и PaaS: KaaS или CaaS – «Kubernetes как услуга» (Kubernetes as a Service) или «Контейнер как услуга» (Container as a Service).

GKE предлагает разработчикам удобное и быстрое развёртывание в полностью контролируемой среде Kubernetes. Хоть Kubernetes и будет выполнять координацию и управление контейнеров, развёрнутых в нём, создание кластера Kubernetes, настройка вычислительных узлов для Kubernetes, создание брандмауэров и др. лежит на плечах разработчика, желающего развернуть свой проект.

Несмотря на все преимущества GKE, Google признал, что Kubernetes чаще всего сложная для освоения технология и поэтому добавил Autopilot режим для GKE, который сам занимается управлением вычислительными узлами, динамически выделяет ресурсы для каждого из развёрнутых контейнеров и имеет множество предустановленных настроек безопасности и сети. В случае данной работы Autopilot использоваться не будет как раз из-за предустановленных настроек, которые не позволяют использовать такие платформы как Agones и OpenMatch. Кроме того, Autopilot поддерживает только Containerd-образы контейнеров, однако Docker-образы можно легко мигрировать на Containerd-образы.

4.5.2. Развёртывание сервера игры

Для развёртывания сервера игры сперва необходимо создать кластер Kubernetes. Google Cloud Platform предоставляет специальный командный интерфейс gcloud для выполнения подобных операций через командную строку на локальном компьютере.

При создании кластера были созданы 3 набора вычислительных узлов: набор для OpenMatch, набор для Agones и набор для самого сервера игры (рис. 4.8). Данный подход рекомендуется для стабильной работы всех компонент.

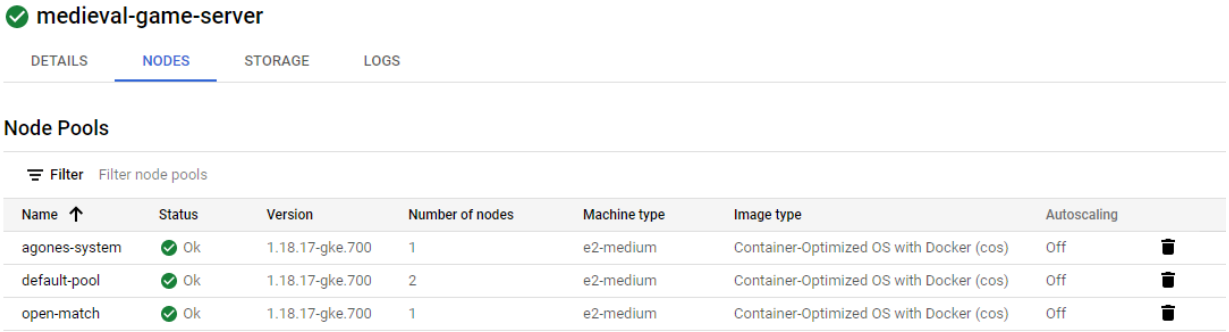
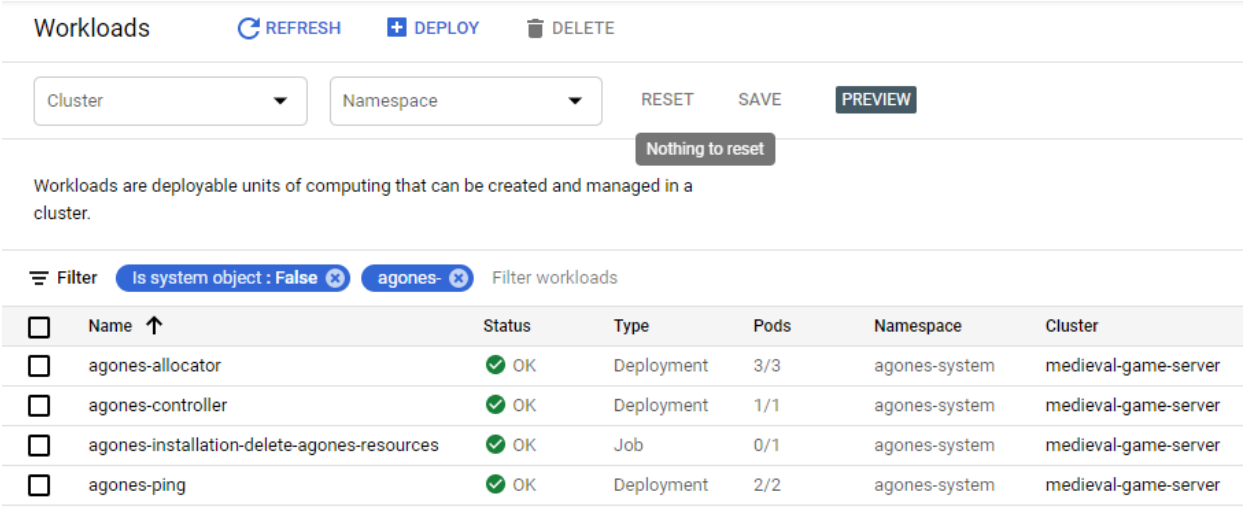


Рисунок 4.8 – Наборы вычислительных узлов для кластера

Следующим шагом является развёртывание Agones и OpenMatch. Обе платформы предлагают инструкции для развёртывания как при помощи интерфейса командной строки kubectl – данный интерфейс будет взаимодействовать с кластером Kubernetes напрямую. Однако для упрощения процесса развёртывания компонент, которые внешне выглядят единым целым и их можно представить в виде пакета, используется пакетный менеджер для Kubernetes – Helm. Используя его, Agones и OpenMatch были развёрнуты всего



за 2 команды каждая: добавление репозитория платформы в известные пакетному менеджеру и непосредственная команда установки со всеми необходимыми параметрами (рис. 4.9 и 4.10).

Рисунок 4.9 – Развёрнутые компоненты Agones

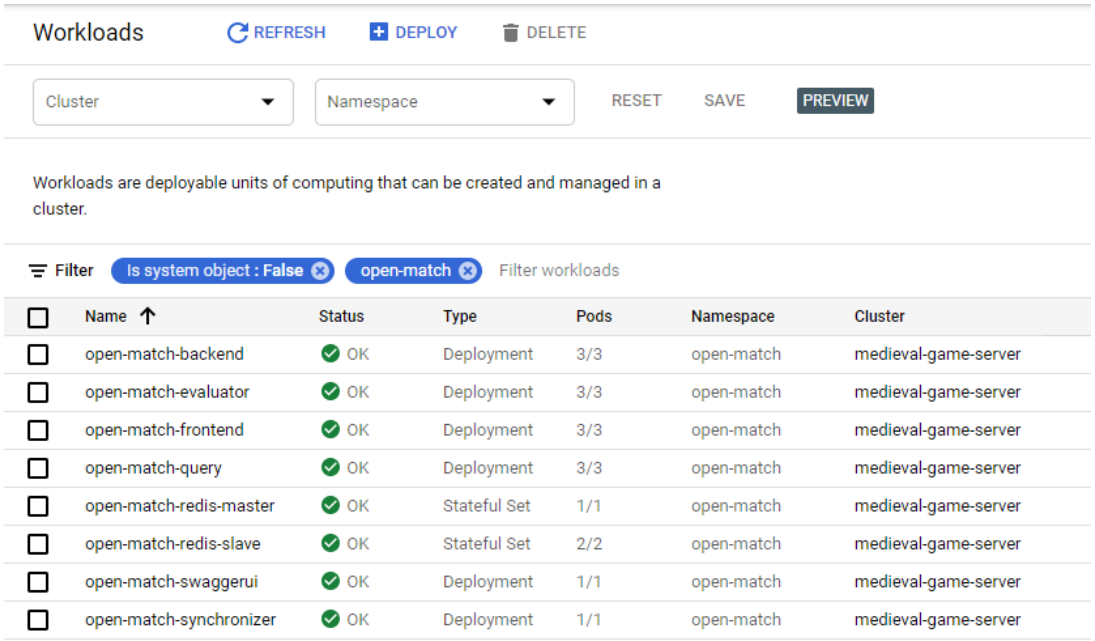


Рисунок 4.10 – Развёрнутые компоненты OpenMatch

После успешного развёртывания Agones и OpenMatch можно приступить к

развёртыванию самого сервера игры. Для это понадобятся файлы, уже описанные в приложениях И и Л, а также понадобится отдельный файл с описанием Deployment для микросервиса Matchmaker. Для их применения был использован интерфейс командной строки kubectl.

Так как данное развёртывание происходит на облачном сервисе, необходимо поместить образы данных микросервисов в DockerHub. Для автоматизации данного процесса, а также для введения CI и CD для ускорения процесса сборки всех микросервисов и их развёртывания был использован сервис Travis CI в сочетании с отдельно написанным Makefile. Пример файла для инструкций Travis CI по сборке микросервиса GameService и целей Makefile, которые используются в CI, находится в Приложении Н и О.

Результат развёртывания можно увидеть на рисунке 4.11. Стоит отметить, что, как и было указано в Приложении Л, было развёрнуто количество экземпляров сервера игры равное размеру буфера.

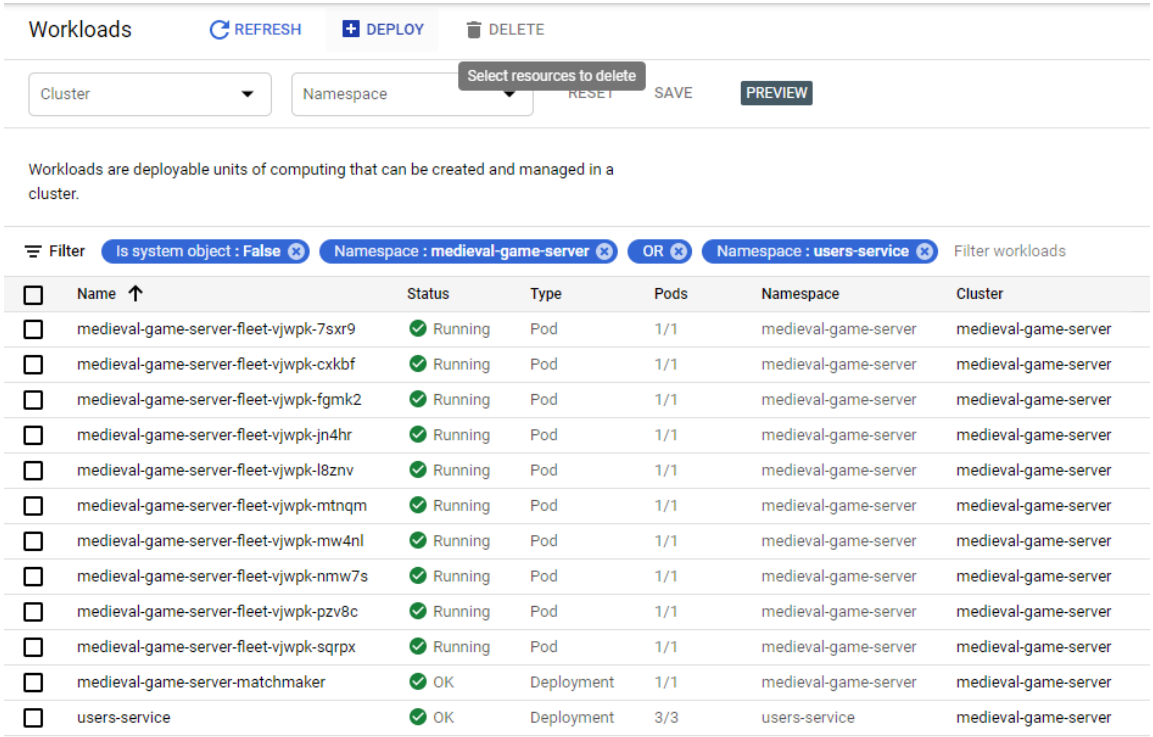


Рисунок 4.11 – Развёрнутые компоненты сервера игры

Для завершения развёртывания необходимо обеспечить доступ извне кластера как к HTTP-методам микросервисов Matchmaker и UsersService, так и к GRPC-методам микросервиса GameService.

Для обеспечения доступа извне кластера, а также балансировки нагрузки для микросервисов Matchmaker и UsersService были созданы LoadBalancer и Service ресурсы. На рисунке 4.12 можно увидеть полученные результаты – в том числе и адреса, которые были выделены данным микросервисам. При отправке запросов в UsersService или при желании принять участие в игре

клиент будет отправлять запросы именно по этим адреса.

Для обеспечения доступа к GRPC-методам GameService необходимо настроить со стороны самого GKE специальный брандмауэр для кластера, на котором размещён сервер игры.

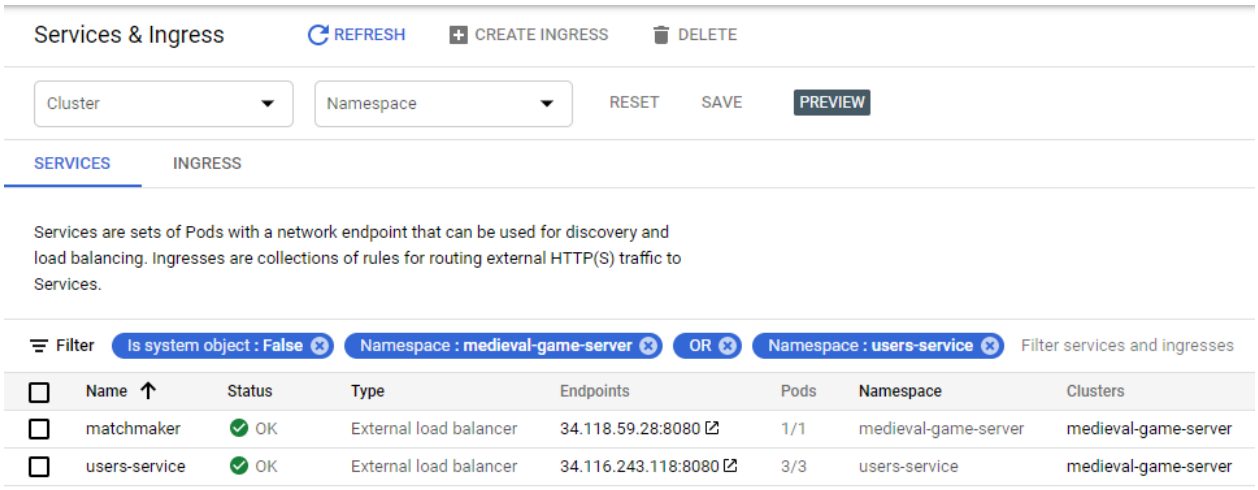


Рисунок 4.12 – Адреса микросервисов в глобальной сети

На рисунке 4.13 можно увидеть созданный брандмауэр. Порты 7000-8000 выделены специально, ведь именно в этом диапазоне Agones выдаёт порты созданным экземплярам сервера игры.

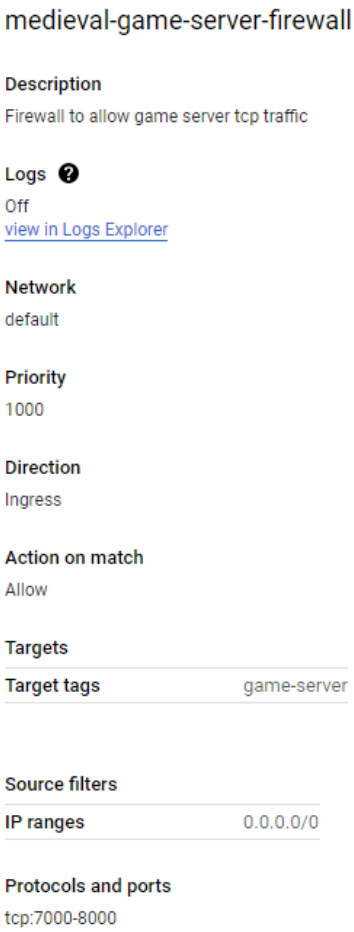


Рисунок 4.13 – Брандмауэр для сервера игры

4.5.3. Тестирование развёртывания сервера игры

Для тестирования развёртывания UsersService была использована программа Postman для отправки запросов по адресу, указанному на рисунке 4.12 (рис. 4.14).

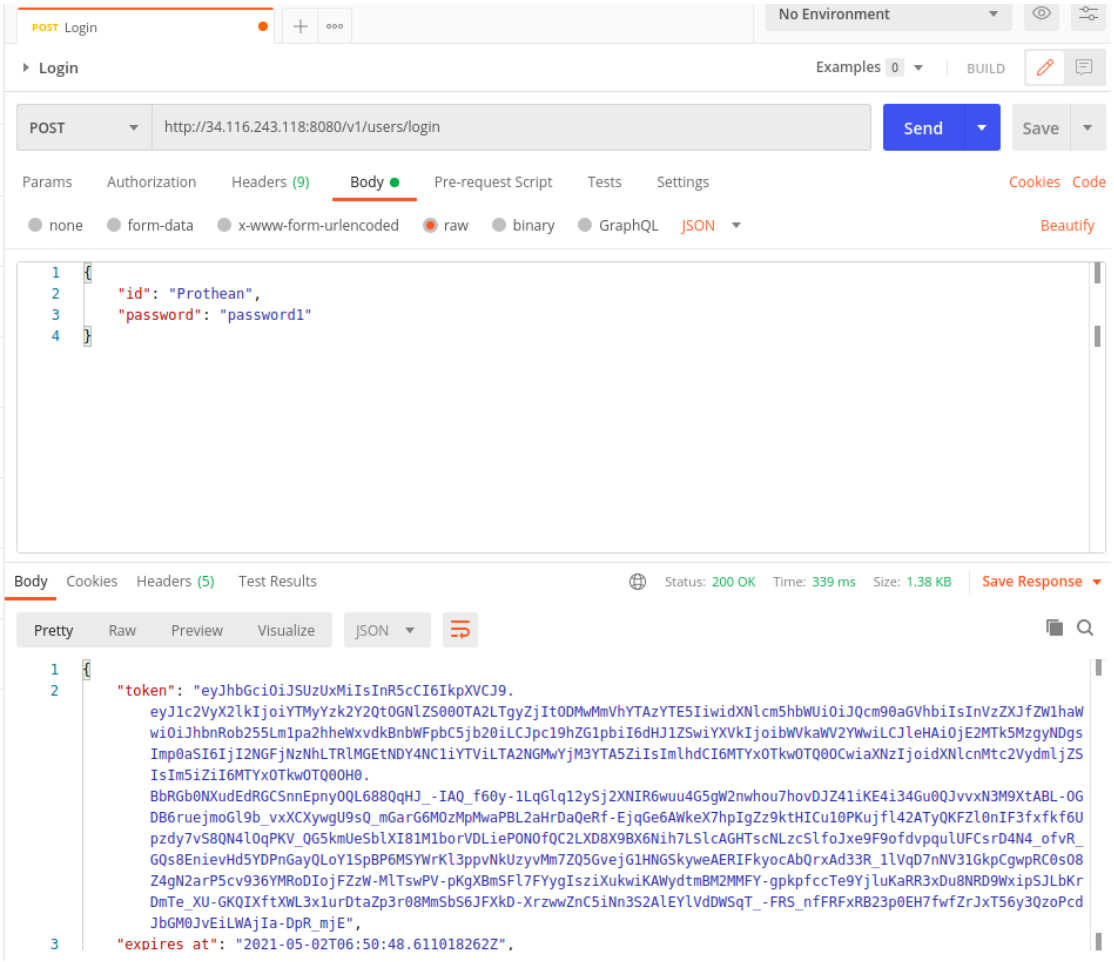


Рисунок 4.14 – Отправка запроса для аутентификации

Как можно видеть на рисунке, запрос был успешно обработан и в ответ был получен JWT-токен для совершения остальных запросов к серверу игры.

Для полного тестирования API UsersService в Postman была создана отдельная коллекция запросов, представленная на рис. 4.15.

Для тестирования развёртывания Matchmaker и GameService был использован клиент, описанный в пункте 3.4, который использовался для локального интеграционного тестирования. В рамках данного тестирования размер сессии был изменён до 20 игроков при помощи изменения ресурса Fleet, развёрнутом на GKE, и общее количество имитированных клиентов, который

будет использовать клиент для интеграционного тестирования, повышено с 5 до 500. Для обработки такого количества клиентов понадобится создать минимум 25 экземпляров микросервиса GameService – это означает, что реальная нагрузка будет превосходить ожидаемую в тот момент в 2.5 раза (рис. 4.16).

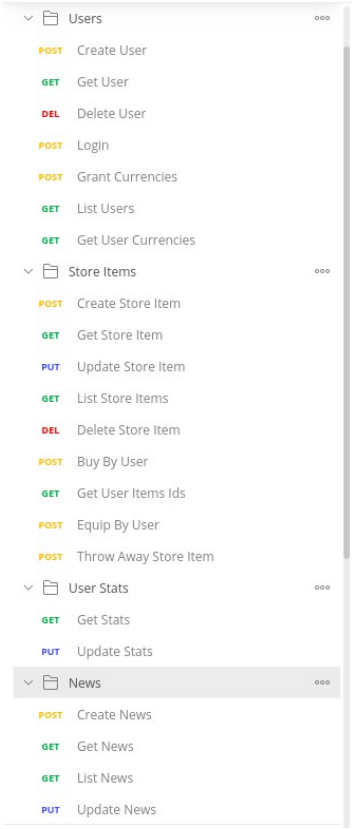


Рисунок 4.15 – Коллекция запросов для тестирования API сервера игры

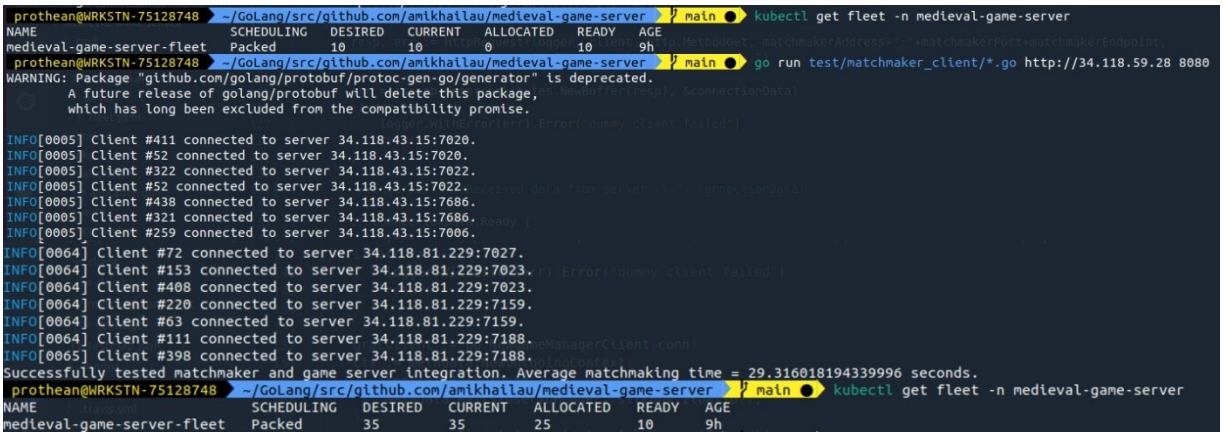
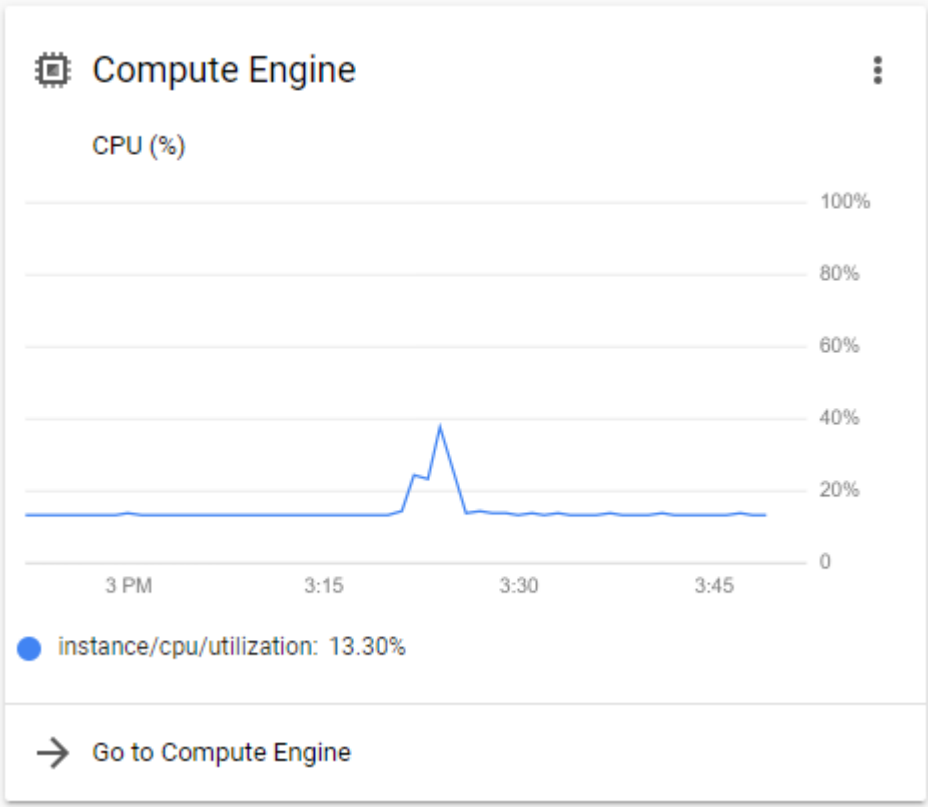


Рисунок 4.16 – Динамическое масштабирование сервера игры для обработки большего количества игровых сессий

Как можно увидеть на рисунке 4.16 всего было развёрнуто 35 экземпляров сервера игры, что легко объяснить 25 выделенными экземплярами и 10 невыделенными экземплярами, которые являются буфером. Кроме того, на

организацию сессий игр и выделение экземпляров сервера под обработку этих сессий понадобилось всего 1 минута. Среднее время ожидания в очереди – 29.3 секунды. Учитывая, что для выделения необходимого количества экземпляров



серверу пришлось пройти через 2 этапа масштабирования, данные показатели является весьма впечатляющими, так как данный эксперимент был проведён на 3-х вычислительных узлах типа e2-medium, имеющих по 1 физичекому процессору – один из самых дешёвых типов вычислительных узлов, предоставляемых GKE, при этом максимальное использование процессора не превысило 40% (рис. 4.17).

Рисунок 4.17 – Использование процессора вычислительных узлов во время эксперимента

ВЫВОДЫ

1. Проведён общий обзор контейнеризации.
2. Рассмотрена технология Docker.
3. Проведена контейнеризация всех микросервисов серверной части.
4. Рассмотрена платформа Kubernetes.
5. Написаны ресурсы Deployment для развёртывания UsersService и Matchmaker на Kubernetes.
6. Проведён общий обзор платформы Agones. Рассмотрена интеграция

микросервиса GameService с Agones SDK.

7. Проанализирован процесс развёртывания сервера игры при помощи Agones и жизненный цикл экземпляра сервера. Созданы ресурсы Fleet и FleetAutoscaler для развёртывания микросервиса GameService.
8. Проведён общий обзор платформы OpenMatch.
9. Рассмотрены структура OpenMatch и процесс организации игр при помощи OpenMatch.
10. Проведён общий обзор облачной платформы Google Cloud Platform и, в частности, услуги Google Kubernetes Engine. Рассмотрены различные виды услуг, предоставляемых облачными платформами.
11. Проведено развёртывание сервера игры на GKE. Репозитории микросервисов добавлены в Travis CI для CI/CD.
12. Проведено тестирование развёртывания и динамического масштабирования сервера игры.

ЗАКЛЮЧЕНИЕ

Микросервисные технологии активно развиваются и захватывают рынок программного обеспечения. По итогам опроса, проведённого между IT-сотрудниками из разных частей света на разных позициях, 49,3% думают, что микросервисная архитектура станет стандартом для больших и сложных проектов, а 36,2% считают, что данная архитектура станет промышленным стандартом для разработки бэкенда [30].

Микросервисная архитектура ни в коем случае не является «серебряной пулей», которая мгновенно решит все проблемы с разработкой, развёртыванием и поддержкой больших и сложных приложений, но в то же время возможности гибкого масштабирования логики сервера позволяют оперативно реагировать на появление дополнительной нагрузки, как например бывает при наплыве большого количества новых игроков в многопользовательской игре, и также избавиться от проблем с полной недоступностью функционала при проведении технических работ в одной из частей игры. Облегчение логгирования и выявления проблемной логики, приводящей к недоступности функционала, также позволяет более оперативно реагировать на внезапные сбои и облегчает поддержку серверов. Поэтому применение микросервисной архитектуры именно в данной сфере является достаточно выгодной идеей для более глубокого исследования.

Применение фреймворка GRPC в микросервисной архитектуре не является инновацией, сам фреймворк входит в инициативу Cloud Nature Computing Foundation, которая является «домом» для множества проектов с открытым кодом, направленных на проектирование и разработку программного обеспечения для облачных приложений. Однако идея его применения в сфере многопользовательских игр относительна нова, одна из самых известных платформ разработки игр Unity имеет лишь экспериментальную поддержку данного фреймворка.

При помощи использования новых технологий Agones и OpenMatch появляется возможность разработать решение, которое сводит проблемы динамического масштабирования и обработки резких ростов нагрузки к минимуму. Кроме предоставления необходимого функционала для динамического масштабирования, обновления сервера и организатора игр, данные технологии облегчают процесс разработки за счёт предоставления практически полной инфраструктуры для развёртывания сервера игры, что позволяет разработчикам сконцентрироваться на решении более важных задач, таких как дизайн игры, разработка логики синхронизации клиента и сервера и т.п. Кроме того, данные технологии имеют открытый исходный код и могут применяться как в личных проектах, так и в производственных масштабах.

Использование Agones в своей основе является достаточно простым, однако требует некоторых знаний о таких понятиях как контейнеризация и распределённые вычислительные системы и о таких технологиях как Docker и Kubernetes.

В данной работе были совершены следующие действия:

1. Проведён общий обзор микросервисной архитектуры. Поянено понятие «микросервис».
2. Проведено сравнение микросервисной архитектуры с монолитной архитектурой и SOA.
3. Описаны используемые в проекте шаблоны проектирования приложений с микросервисной архитектурой.
4. Проведён общий обзор фреймворка gRPC.
5. Рассмотрена проблема жульничества пользователя.
6. Рассмотрены подходы и приёмы, применяемые для ликвидации проблем синхронизации состояний клиента и сервера.
7. Рассмотрены подходы и приёмы, применяемые для синхронизации состояний сессии игры между несколькими клиентами.
8. Проведён анализ требований к серверной части проекта Medieval.io
9. Создан макет архитектуры серверной части.
10. Выделены контексты для компонент микросервиса UsersService. Создана схема базы данных. Спроектированы внешние интерфейсы, предоставляемые UsersService. Создано описание структур данных и интерфейсов, используемых в UsersService, на языке protobuf.
11. Проведён анализ проблем масштабирования серверной части игры. Предложено возможное решение данных проблем. Проведён поиск подходящих технологий с функционалом, похожим на функционал возможного решения.
12. Проведён анализ проблем организации сессий игр. Предложено возможное решение для организатора игр. Проведён поиск подходящих технологий с функционалом, похожим на функционал возможного решения.
13. Спроектированы внешние интерфейсы, предоставляемые микросервисами GameService и Matchmaker. Создано описание структур данных и интерфейсов, используемых в GameService и Matchmaker, на языке protobuf.
14. Разработан микросервис UsersService. Рассмотрен шаблон «интерцептор» для авторизации запросов.
15. Разработан микросервис Matchmaker.
16. Разработан микросервис GameService.

17. Рассмотрены алгоритмы широкой фазы обнаружения коллизий и реализован алгоритм Sweep and Prune для GameService.
18. Реализованы алгоритмы узкой фазы обнаружения коллизий для GameService.
19. Проведено покрытие юнит-тестами функционала всех микросервисов. Проведено интеграционное тестирование взаимодействия Matchmaker и GameService.
20. Проведён общий обзор контейнеризации.
21. Рассмотрена технология Docker.
22. Проведена контейнеризация всех микросервисов серверной части.
23. Рассмотрена платформа Kubernetes.
24. Написаны ресурсы Deployment для развёртывания UsersService и Matchmaker на Kubernetes.
25. Проведён общий обзор платформы Agones. Рассмотрена интеграция микросервиса GameService с Agones SDK.
26. Проанализирован процесс развёртывания сервера игры при помощи Agones и жизненный цикл экземпляра сервера. Созданы ресурсы Fleet и FleetAutoscaler для развёртывания микросервиса GameService.
27. Проведён общий обзор платформы OpenMatch.
28. Рассмотрены структура OpenMatch и процесс организации игр при помощи OpenMatch.
29. Проведён общий обзор облачной платформы Google Cloud Platform и, в частности, услуги Google Kubernetes Engine. Рассмотрены различные виды услуг, предоставляемых облачными платформами.
30. Проведено развёртывание сервера игры на GKE. Репозитории микросервисов добавлены в Travis CI для CI/CD.
31. Проведено тестирование развёртывания и динамического масштабирования сервера игры.

Для проектирования серверной части использовалась среда Microsoft Visio и облачный сервис app.diagrams.net.

Для реализации серверной части использовалась среда Visual Studio Code.

Результаты исследования представлены на 78-ой Научной конференции студентов и аспирантов БГУ.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Chris Richardson, Microservice Patterns with examples in Java. / Richardson, Chris. — M.: Manning, 2018. — 520 с. — ISBN 9781617294549.
2. Бессерверные вычисления [Электронный ресурс] – Режим доступа: https://aws.amazon.com/ru/serverless/ – Дата доступа: 01.05.2021
3. Microservices vs Monolith: which architecture is the best choice for your business? [Electronic resource] – Mode of access: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ – Date of access: 01.05.2021
4. SOA vs. microservices: What's the difference [Electronic resource] – Mode of access: https://www.ibm.com/blogs/cloud-computing/2018/09/06/soa-versus-microservices/#:~:text=The%20main%20difference%20between%20SOA,architecture%20has%20an%20application%20scope. – Date of access: 01.05.2021
5. Переход от монолита к микросервисам [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/305826/ – Дата доступа: 01.05.2020
6. What are microservices? [Electronic resource] – Mode of access: https://microservices.io/ – Date of access: 01.05.2021
7. gRPC – A high-performance, open source universal RPC framework [Electronic resource] – Mode of access: https://grpc.io/ – Date of access: 01.05.2021
8. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization [Electronic resource] – Mode of access: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization – Date of access: 01.05.2021
9. Fast-Paced Multiplayer (Part I): Client-Server Game Architecture [Electronic resource] – Mode of access: https://www.gabrielgambetta.com/client-server-game-architecture.html – Date of access: 01.05.2021
10. Advanced Load Balancing and Sticky Sessions with Ambassador, Envoy and Kubernetes [Electronic resource] – Mode of access: https://danielbryantuk.medium.com/advanced-load-balancing-and-

sticky-sessions-with-ambassador-envoy-and-kubernetes-6f924b9d4b7e – Date of access: 01.05.2021

11. Репозиторий users-service [Электронный ресурс] – Режим доступа: https://github.com/amikhailau/users-service – Дата доступа: 01.05.2021
12. go-cache Golang module [Electronic resource] – Mode of access: https://pkg.go.dev/github.com/patrickmn/go-cache – Date of access: 01.05.2021
13. Репозиторий medieval-game-server [Электронный ресурс] – Режим доступа: https://github.com/amikhailau/medieval-game-server – Дата доступа: 01.05.2021
14. Введение в дискретно-ориентированные многогранники для задачи определения столкновений [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/257339/ – Дата доступа: 01.05.2021
15. Christer Ericson, Real-Time Collision Detection. / Ericson, Christer. — Morgan Kaufmann Publishers, 2005. — 592 с. — ISBN 1-55860-732-3.
16. Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal [Electronic resource] – Mode of access: https://www.math.ucsd.edu/~sbuss/ResearchWeb/EnhancedSweepPrune/SAP_paper_online.pdf – Date of access: 01.05.2021
17. Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects [Electronic resource] – Mode of access: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects – Date of access: 01.05.2021
18. Lesson 04. Broad Phase Collision Detection [Electronic resource] – Mode of access: https://dai.fmph.uniba.sk/upload/8/87/Ca15_lesson05.pdf – Date of access: 01.05.2021
19. Репозиторий Collision2D [Электронный ресурс] – Режим доступа: https://github.com/Tarliton/collision2d – Дата доступа: 01.05.2021
20. Контейнеризация понятным языком: от самых азов до тонкостей работы с Kubernetes [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/southbridge/blog/530226/ – Дата доступа: 01.05.2021
21. Docker [Electronic resource] – Mode of access:

https://www.docker.com/ – Date of access: 01.05.2021

22. Container Technologies Overview [Electronic resource] – Mode of access: https://dzone.com/articles/container-technologies-overview?roistat_visit=899735 – Date of access: 01.05.2021
23. Kubernetes [Electronic resource] – Mode of access: https://kubernetes.io/ru/ – Date of access: 01.05.2021
24. Основы Kubernetes [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/258443/ – Дата доступа: 01.05.2021
25. Agones [Electronic resource] – Mode of access: https://agones.dev/site/ – Date of access: 01.05.2021
26. Introducing Agones: Open-source, multiplayer, dedicated game-server hosting built on Kubernetes [Electronic resource] – Mode of access: https://cloud.google.com/blog/products/containers-kubernetes/introducing-agones-open-source-multiplayer-dedicated-game-server-hosting-built-on-kubernetes – Date of access: 01.05.2021
27. OpenMatch [Electronic resource] – Mode of access: https://open-match.dev/site/ – Date of access: 01.05.2021
28. Open Match simplified matchmaking for developers is now 1.0 [Electronic resource] – Mode of access: https://cloud.google.com/blog/products/gaming/open-match-1-0-ready-for-deployment-in-production – Date of access: 01.05.2021
29. GCP: Разбор вычислительного стека Google Cloud Platform [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/otus/blog/467749/ – Дата доступа: 01.05.2021
30. State of Microservices 2020 [Electronic resource] – Mode of access: https://tsh.io/state-of-microservices/#future – Date of access: 01.05.2021

ПРИЛОЖЕНИЯ

Приложение А

```
message User {
  option (gorm.opts) = {
      ormable: true,
      multi_account: false
  };

  string id = 1 [(gorm.field).tag = {type: "UUID"  primary_key:
true}];
  string name = 2;
  string email = 3;
  string password = 4;
  int32 coins = 5;
  int32 gems = 6;
  repeated StoreItem items = 7 [(gorm.field).many_to_many =
{jointable: "users_store_items" preload: false}];
  UserStats stats = 8;
}

message CreateUserRequest {
  string name = 1;
  string email = 2;
  string password = 3;
}

message CreateUserResponse {
  User result = 1;
}

message ReadUserRequest {
  string id = 1;
}

message ReadUserResponse {
  User result = 1;
}

message UpdateUserRequest {
  string id = 1;
  string name = 2;
  string password = 3;
```

```
}

message UpdateUserResponse {}

message DeleteUserRequest {
  string id = 1;
}

message DeleteUserResponse {}

message ListUsersRequest {
  infoblox.api.Filtering filter = 1;
  infoblox.api.Sorting order_by = 2;
  infoblox.api.FieldSelection fields = 3;
  infoblox.api.Pagination paging = 4;
}

message ListUsersResponse {
  repeated User results = 1;
  infoblox.api.PageInfo page = 2;
}

message LoginRequest {
  string id = 1;
  string password = 2;
}

message LoginResponse {
  string token = 1;
  google.protobuf.Timestamp expires_at = 2;
  bool isAdmin = 3;
  string user_id = 4;
}

message GrantCurrenciesRequest {
  string id = 1;
  int32 add_coins = 2;
  int32 add_gems = 3;
}

message GrantCurrenciesResponse {}

message GetUserCurrenciesRequest {
```

```
  string id = 1;
}

message GetUserCurrenciesResponse {
  int32 coins = 1;
  int32 gems = 2;
}

service Users {
  option (gorm.server) = {
      autogen: true,
      txn_middleware: false,
    };

  rpc Create (CreateUserRequest) returns (CreateUserResponse) {
    option (google.api.http) = {
            post: "/users"
            body: "*"
        };
  }

  rpc Read (ReadUserRequest) returns (ReadUserResponse) {
    option (google.api.http) = {
            get: "/users/{id}"
        };
  }

  rpc Update (UpdateUserRequest) returns (UpdateUserResponse) {
    option (atlas_validate.method).allow_unknown_fields = false;
    option (google.api.http) = {
            put: "/users/{id}"
            body: "*"
            additional_bindings: {
                patch: "/users/{id}"
                body: "*"
            }
    };
  }

  rpc Delete (DeleteUserRequest) returns (DeleteUserResponse) {
    option (google.api.http) = {
            delete: "/users/{id}"
        };
```

```
    option (gorm.method).object_type = "User";
  }

  rpc List (ListUsersRequest) returns (ListUsersResponse) {
    option (google.api.http) = {
            get: "/users"
        };
  }

  rpc Login (LoginRequest) returns (LoginResponse) {
    option (google.api.http) = {
        post: "/users/login"
        body: "*"
    };
  }

  rpc GrantCurrencies (GrantCurrenciesRequest) returns
(GrantCurrenciesResponse) {
    option (google.api.http) = {
      post: "/users/{id}/currencies"
      body: "*"
    };
  }

  rpc GetUserCurrencies (GetUserCurrenciesRequest) returns
(GetUserCurrenciesResponse) {
    option (google.api.http) = {
      get: "/users/{id}/currencies"
    };
  }
}


message StoreItem {
  option (gorm.opts) = {
      ormable: true,
      multi_account: false
  };

  string id = 1 [(gorm.field).tag = {type: "UUID"  primary_key:
true}];
  string name = 2;
  string description = 3;
```

```
  int32 type = 4;
  int32 coins_price = 5;
  int32 gems_price = 6;
  string image_id = 7;
  bool on_sale = 8;
  int32 sale_coins_price = 9;
  int32 sale_gems_price = 10;
}

message CreateStoreItemRequest {
  string name = 1;
  string description = 2;
  int32 type = 3;
  int32 coins_price = 4;
  int32 gems_price = 5;
  string image_id = 6;
  bool on_sale = 7;
  int32 sale_coins_price = 8;
  int32 sale_gems_price = 9;
}

message CreateStoreItemResponse {
  StoreItem result = 1;
}

message ReadStoreItemRequest {
  string id = 1;
}

message ReadStoreItemResponse {
  StoreItem result = 1;
}

message UpdateStoreItemRequest {
  StoreItem payload = 1;
  google.protobuf.FieldMask fields = 2;
}

message UpdateStoreItemResponse {
  StoreItem result = 1;
}

message DeleteStoreItemRequest {
```

```
  string id = 1;
}

message DeleteStoreItemResponse {}

message ListStoreItemsRequest {
  infoblox.api.Filtering filter = 1;
  infoblox.api.Sorting order_by = 2;
  infoblox.api.FieldSelection fields = 3;
  infoblox.api.Pagination paging = 4;
}

message ListStoreItemsResponse {
  repeated StoreItem results = 1;
  infoblox.api.PageInfo page = 2;
}

message BuyByUserRequest {
  string user_id = 1;
  string item_id = 2;
}

message BuyByUserResponse {}

message ThrowAwayByUserRequest {
  string user_id = 1;
  string item_id = 2;
}

message ThrowAwayByUserResponse {}

message EquipByUserRequest {
  string user_id = 1;
  string item_id = 2;
}

message EquipByUserResponse {}

message GetUserItemsIdsRequest {
  string user_id = 1;
}

message UserItemInfo {
```

```
  string item_id = 1;
  bool equipped = 2;
}

message GetUserItemsIdsResponse {
  repeated UserItemInfo items = 1;
}

message GetEquippedUserItemsIdsRequest {
  string user_id = 1;
}

message GetEquippedUserItemsIdsResponse {
  repeated UserItemInfo items = 1;
}

service StoreItems {
  option (gorm.server) = {
      autogen: true,
      txn_middleware: false,
    };

  rpc Create (CreateStoreItemRequest) returns
(CreateStoreItemResponse) {
    option (google.api.http) = {
            post: "/store_items"
            body: "*"
        };
  }

  rpc Read (ReadStoreItemRequest) returns (ReadStoreItemResponse)
{
    option (google.api.http) = {
            get: "/store_items/{id}"
        };
  }

  rpc Update (UpdateStoreItemRequest) returns
(UpdateStoreItemResponse) {
    option (atlas_validate.method).allow_unknown_fields = false;
    option (google.api.http) = {
            put: "/store_items/{payload.id}"
            body: "payload"
```

```
            additional_bindings: {
                patch: "/store_items/{payload.id}"
                body: "payload"
            }
    };
  }

  rpc Delete (DeleteStoreItemRequest) returns
(DeleteStoreItemResponse) {
    option (google.api.http) = {
            delete: "/store_items/{id}"
        };
    option (gorm.method).object_type = "StoreItem";
  }

  rpc List (ListStoreItemsRequest) returns
(ListStoreItemsResponse) {
    option (google.api.http) = {
            get: "/store_items"
        };
  }

  rpc BuyByUser (BuyByUserRequest) returns (BuyByUserResponse) {
    option (google.api.http) = {
            post: "/store_items/buy"
            body: "*"
        };
  }

  rpc GetUserItemsIds (GetUserItemsIdsRequest) returns
(GetUserItemsIdsResponse) {
    option (google.api.http) = {
            get: "/store_items/user/{user_id}"
        };
  }

  rpc GetEquippedUserItemsIds (GetEquippedUserItemsIdsRequest)
returns (GetEquippedUserItemsIdsResponse) {
    option (google.api.http) = {
            get: "/store_items/user/{user_id}/equipped"
        };
  }

  rpc EquipByUser (EquipByUserRequest) returns
```

```
(EquipByUserResponse) {
    option (google.api.http) = {
            post: "/store_items/equip"
            body: "*"
        };
  }

  rpc ThrowAwayByUser (ThrowAwayByUserRequest) returns
(ThrowAwayByUserResponse) {
    option (google.api.http) = {
            post: "/store_items/throwaway"
            body: "*"
        };
  }
}

message UserStats {
  option (gorm.opts) = {
      ormable: true,
      multi_account: false
  };

  int32 id = 1 [(gorm.field).tag = {type: "serial"  primary_key:
true}];
  int32 games = 2;
  int32 wins = 3;
  int32 top5 = 4;
  int32 kills = 5;
}

message ReadUserStatsRequest {
  string username = 1;
}

message ReadUserStatsResponse {
  UserStats result = 1;
}

message UpdateUserStatsRequest {
  string username = 1;
  int32 add_games = 2;
  int32 add_wins = 3;
  int32 add_top5 = 4;
```

```
  int32 add_kills = 5;
}

message UpdateUserStatsResponse {}

service UsersStats {
  option (gorm.server) = {
      autogen: true,
      txn_middleware: false,
    };

  rpc GetStats (ReadUserStatsRequest) returns
(ReadUserStatsResponse) {
    option (google.api.http) = {
            get: "/stats/{username}"
        };
  }

  rpc UpdateStats (UpdateUserStatsRequest) returns
(UpdateUserStatsResponse) {
    option (atlas_validate.method).allow_unknown_fields = false;
    option (google.api.http) = {
            put: "/stats/{username}"
            body: "*"
            additional_bindings: {
                patch: "/stats/{username}"
                body: "*"
            }
    };
  }
}

message News {
  option (gorm.opts) = {
      ormable: true,
      multi_account: false
  };

  string id = 1 [(gorm.field).tag = {type: "UUID"  primary_key:
true}];
  google.protobuf.Timestamp created_at = 2;
  string title = 3;
  string description = 4;
```

```
  string image_link = 5;
}

message CreateNewsRequest {
  string title = 1;
  string description = 2;
  string image_link = 3;
}

message CreateNewsResponse {
  News result = 1;
}

message ReadNewsRequest {
  string id = 1;
}

message ReadNewsResponse {
  News result = 1;
}

message UpdateNewsRequest {
  string id = 1;
  string title = 2;
  string description = 3;
  string image_link = 4;
}

message UpdateNewsResponse {}

message ListNewsRequest {
  infoblox.api.Filtering filter = 1;
  infoblox.api.Sorting order_by = 2;
  infoblox.api.FieldSelection fields = 3;
  infoblox.api.Pagination paging = 4;
}

message ListNewsResponse {
  repeated News results = 1;
  infoblox.api.PageInfo page = 2;
}

service NewsService {
```

```
  option (gorm.server) = {
      autogen: true,
      txn_middleware: false,
    };

  rpc Create (CreateNewsRequest) returns (CreateNewsResponse) {
    option (google.api.http) = {
            post: "/news"
            body: "*"
        };
  }

  rpc Read (ReadNewsRequest) returns (ReadNewsResponse) {
    option (google.api.http) = {
            get: "/news/{id}"
        };
  }

  rpc Update (UpdateNewsRequest) returns (UpdateNewsResponse) {
    option (atlas_validate.method).allow_unknown_fields = false;
    option (google.api.http) = {
            put: "/news/{id}"
            body: "*"
            additional_bindings: {
                patch: "/news/{id}"
                body: "*"
            }
    };
  }

  rpc List (ListNewsRequest) returns (ListNewsResponse) {
    option (google.api.http) = {
            get: "/news"
        };
  }
}
```

```
enum EquipmentItemType {
    HELMET = 0;
    ARMOR = 1;
    WEAPON = 2;
}

enum EquipmentItemRarity {
    DEFAULT = 0;
    COMMON = 1;
    UNCOMMON = 2;
    RARE = 3;
    EPIC = 4;
    LEGENDARY = 5;
}

enum NotificationType {
    CONNECT = 0;
    DISCONNECT = 1;
}

enum ServerNotificationType {
    PLAYER_CONNECTED = 0;
    PLAYER_DISCONNECTED = 1;
    PLAYER_KILLED = 2;
    PLAYER_ATTACKED = 3;
    GAME_STARTED = 4;
    GAME_FINISHED = 5;
}

message WeaponCharacteristics {
    int32 attack_power = 1;
    float range = 2;
    float attack_cone = 3;
    float knockback_power = 4;
}

message EquipmentItem {
    EquipmentItemType type = 1;
    EquipmentItemRarity rarity = 2;
    oneof characteristics {
        WeaponCharacteristics weapon_chars = 3;
        int32 hp_buff = 4;
        int32 damage_reduction = 5;
    }
    int32 item_id = 6;
}

message DroppedEquipmentItem {
    Vector position = 1;
    EquipmentItem item = 2;
```

```
}

message PlayerEquipment {
    EquipmentItem helmet = 1;
    EquipmentItem armor = 2;
    EquipmentItem weapon = 3;
}

message Vector {
    float x = 1;
    float y = 2;
}

message PlayerStats {
    int32 damage = 1;
    int32 kills = 2;
}

message Player {
    string nickname = 1;
    int32 hp = 2;
    PlayerEquipment equipment = 3;
    string user_id = 4;
    Vector position = 5;
    float angle = 6;
    int32 player_id = 7;
    PlayerStats stats = 8;
}

message GameState {
    repeated Player players = 1;
    repeated DroppedEquipmentItem dropped_items = 2;
    int32 players_left = 3;
}

message Action {
    oneof action {
        MovementAction move = 1;
        AttackAction attack = 2;
        PickUpAction pick_up = 3;
        DropAction drop = 4;
    }
}

message MovementAction {
    Vector shift = 1;
    float angle = 2;
}

message PickUpAction {
    int32 item_id = 1;
}
```

```
message DropAction {
    EquipmentItemType slot = 1;
}

message AttackAction {

}

message ConnectRequest {
    string user_id = 1;
    google.protobuf.Timestamp local_time = 2;
    string nickname = 3;
}

message ConnectResponse {
    int32 ping = 1;
    string token = 2;
    google.protobuf.Timestamp server_time = 3;
}

message Notification {
    NotificationType type = 1;
}

message ClientMessage {
    oneof message {
        Action action = 1;
        Notification notification = 2;
    }
}

message ServerNotification {
    ServerNotificationType type = 1;
    string actor = 2;
    string receiver = 3;
}

message ServerResponse {
    oneof info {
        ServerNotification notification = 1;
        GameState game_state = 2;
    }
    google.protobuf.Timestamp server_time = 3;
}

service GameManager {
    rpc Connect (ConnectRequest) returns (ConnectResponse) {}
    rpc Talk(stream ClientMessage) returns (stream ServerResponse)
{}
}
```

```
message MatchmakeRequest {

}

message MatchmakeResponse {
    string id = 1;
}

message CheckMatchmakeStatusRequest {

}

message CheckMatchmakeStatusResponse {
    bool ready = 1;
    bool failed = 2;
    bool not_matchmaked = 3;
    string ip = 4;
    int32 port = 5;
}

message CancelMatchmakeRequest {}

message CancelMatchmakeResponse {

}

service Matchmaker {
    rpc Matchmake (MatchmakeRequest) returns (MatchmakeResponse) {
        option (google.api.http) = {
            post: "/matchmake"
            body: "*"
        };
    }
    rpc CheckMatchmakeStatus(CheckMatchmakeStatusRequest) returns
(CheckMatchmakeStatusResponse) {
        option (google.api.http) = {
            get: "/matchmake"
        };
    }
    rpc CancelMatchmake (CancelMatchmakeRequest) returns
(CancelMatchmakeResponse) {
        option (google.api.http) = {
            delete: "/matchmake"
        };
    }
```

```
type StoreItemsServerConfig struct {
     Database *gorm.DB
}

type StoreItemsServer struct {
     pb.StoreItemsServer
     cfg *StoreItemsServerConfig
}

var _ pb.StoreItemsServer = &StoreItemsServer{}

const (
     deequipQuery  = "UPDATE users_store_items SET equipped = 'f'
WHERE user_id = $1 AND store_item_id = $2"
     equipQuery  = "UPDATE users_store_items SET equipped = 't'
WHERE user_id = $1 AND store_item_id = $2"
     userItemsQuery  = "SELECT store_item_id, equipped FROM
users_store_items WHERE user_id = $1"
     equippedUserItemsQuery  = "SELECT store_item_id, equipped FROM
users_store_items WHERE user_id = $1 AND equipped = 't'"
     findEquippedQuery  = "SELECT si.id FROM store_items si JOIN
users_store_items usi ON usi.store_item_id = si.id WHERE si.type =
$1 AND usi.user_id = $2 AND usi.equipped = $3"
)

func NewStoreItemsServer(cfg *StoreItemsServerConfig)
(*StoreItemsServer, error) {
     return &StoreItemsServer{
          StoreItemsServer: &pb.StoreItemsDefaultServer{},
          cfg:              cfg,
     }, nil
}

func (s *StoreItemsServer) GetEquippedUserItemsIds(ctx
context.Context, req *pb.GetEquippedUserItemsIdsRequest)
(*pb.GetEquippedUserItemsIdsResponse, error) {
     logger := ctxlogrus.Extract(ctx).WithField("user_id",
req.GetUserId())
     logger.Debug("GetEquippedUserItemsIds")

     // Верификация запроса на основании JWT, переданного в
заголовке
     claims, _ := auth.GetAuthorizationData(ctx)
     if !claims.IsAdmin && claims.StandardClaims.Audience != "svc"
&& claims.UserId != req.GetUserId() {
          logger.Error("User can only use this endpoint for
themselves")
          return nil, status.Error(codes.Unauthenticated, "Not
authorized for another user")
```

```
     }

     // Использование GORM для получения пользователя по его
идентификатору
     var usr pb.UserORM
     if err := s.cfg.Database.Where("id = ?",
req.GetUserId()).First(&usr)  .Error; err != nil {
          if err == gorm.ErrRecordNotFound {
               logger.Error("User not found")
               return nil, status.Error(codes.NotFound, "User not
found")
          }
          logger.WithError(err).Error("Could not find user")
          return nil, status.Error(codes.Internal, "Could not find
user")
     }

     // Использование подготовленного выражения SQL
     // для получения предметов внутриигрового магазина,
     // которые данный игрок использует
     items := []*pb.UserItemInfo{}
     rows, err :=
s.cfg.Database.DB().Query(equippedUserItemsQuery, req.GetUserId())
     if err != nil {
          logger.WithError(err).Error("Could not fetch user
items")
          return nil, status.Error(codes.Internal, "Could not
fetch user items")
     }
     for rows.Next() {
          item := pb.UserItemInfo{}
          err := rows.Scan(&item.ItemId, &item.Equipped)
          if err != nil {
               logger.WithError(err).Error("Could not fetch user
items")
               return nil, status.Error(codes.Internal, "Could not
fetch user items")
          }
          items = append(items, &item)
     }

     return &pb.GetEquippedUserItemsIdsResponse{Items: items}, nil
}
```

```
// Список методов, которые доступны только администраторам и
сервисам
var (
     svcEndpoints = []string{"Users/GrantCurrencies",
"UsersService/GetVersion", "StoreItems/Create",
"StoreItems/Update",
          "StoreItems/ThrowAwayByUser", "StoreItems/Delete",
"UsersStats/UpdateStats", "NewsService/Create",
"NewsService/Update"}
)

// Интерцептор
func UnaryServerInterceptor() grpc.UnaryServerInterceptor {
     return func(ctx context.Context, req interface{}, info
*grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{},
error) {

          logger := ctxlogrus.Extract(ctx)
          logger.Debug("Authorization interceptor")

          // Получение имени вызванного метода
          str := strings.Split(info.FullMethod, ".")
          method := str[len(str)-1]

          // 2 метода должны быть вызываемыми без авторизации –
          // регистрация и логин
          if method != "Users/Create" && method != "Users/Login" {
               logger.Debug("Checking claims")
               claims, err := GetAuthorizationData(ctx)
               if err != nil {
                    return nil,
status.Error(codes.Unauthenticated, "Authorization failed -
invalid header/token")
               }
               // Проверка на то, что сессия истекла
               if claims.ExpiresAt < time.Now().Unix() {
                    return nil,
status.Error(codes.Unauthenticated, "Authorization failed - token
expired")
               }
               logger.WithField("claims", claims).Debug("Incoming
claims")
               requiresHighLevelAccess := false
               for _, svcEndpoint := range svcEndpoints {
                    if svcEndpoint == method {
                         requiresHighLevelAccess = true
                         break
                    }
               }
```

```
                // Проверка на то, что если вызванный метод требует
                // прав администратора или спецсервиса, то JWT в
заголовке
                // действительно предоставляет эти права
                if requiresHighLevelAccess && (!claims.IsAdmin &&
claims.StandardClaims.Audience != "svc") {
                     return nil,
status.Error(codes.Unauthenticated, "Authorization failed - high
level access required")
                }
          }

          return handler(ctx, req)
     }
}

// Функция для извлечения токена из заголовка запроса
func GetAuthorizationData(ctx context.Context) (*GameClaims,
error) {
     logger := ctxlogrus.Extract(ctx)
     token, err := grpc_auth.AuthFromMD(ctx, "bearer")
     if err != nil {
          logger.WithError(err).Error("Token not found")
          return nil, err
     }
     claims := &GameClaims{}
     parser := &jwt.Parser{}
     _, _, err = parser.ParseWithClaims(token, claims, func(token
*jwt.Token) (interface{}, error) {
          return publicRSAKey, nil
     })
     if err != nil {
          logger.WithError(err).Error("Not able to parse token")
          return nil, err
     }
     return claims, nil
}
```

```
// Данная функция получает конфигурацию кластера Kubernetes, в
котором
// развёртнут сервер игры/организатор, и создаёт клиент Agones
func ConnectToAgonesInCluster() (*versioned.Clientset, error) {
    config, err := rest.InClusterConfig()
    if err != nil {
         return nil, err
    }
    agonesClient, err := versioned.NewForConfig(config)
    if err != nil {
         return nil, err
    }
    return agonesClient, nil
}

// Данная функция создаёт ресурс GameServerAllocation с именем
ресурса
// Fleet, который отвечает за развёртывание сервера игры
func createAgonesGameServerAllocation()
*allocationv1.GameServerAllocation {
    return &allocationv1.GameServerAllocation{
         Spec: allocationv1.GameServerAllocationSpec{
              Required: metav1.LabelSelector{
                   MatchLabels:
map[string]string{agonesv1.FleetNameLabel: "medieval-game-server-
fleet"},
              },
         },
    }
}

func AllocateGameServer(agonesClient *versioned.Clientset)
(*allocationv1.GameServerAllocation, error) {

    // При помощи Agones клиента создаётся ресурс
GameServerAllocation
    gsa, err :=
agonesClient.AllocationV1() .GameServerAllocations("medieval-
game-server") .Create(context.Background(),
createAgonesGameServerAllocation(), metav1.CreateOptions{})
    if err != nil {
         log.Printf("error requesting allocation: %v\n", err)
         return nil, err
    }

    // Если не удалось выделить экземпляр сервиса, то требуется
    // обозначить это как ошибку
    if gsa.Status.State !=
allocationv1.GameServerAllocationAllocated {
```

```
            log.Printf("failed to allocate game server\n")
            return nil, fmt.Errorf("failed to allocate game server")
        }

        return gsa, nil
    }
```

```
// pkg/gamesession/actions.go
func (g *GameSession) processPossibleHit(attPlayerId, defPlayerId
int32) {

    <...вычисление результатов попадания по другому игроку...>

    if hpLeft <= 0 {
         //Новое сообщение посылается в канал KillNotifications
         g.KillNotifications <- KillInfo{
              Actor:    player.Nickname,
              Receiver: pPlayer.Nickname,
         }
         g.deadPlayers <- pPlayer.PlayerId
         playerCurr.PlayerInfo.Stats.Kills += 1
    }
}

// pkg/connection/connection.go
func NewGameManager(cfg *GameManagerConfig) (*GameManager, error)
{
    <...код инициализации GameSession...>
    //Запуск новой анонимной горутины
    go func() {
         <...>
         moreMessages = true
         //Чтение в цикле всех сообщений из канала
KillNotifications
         for moreMessages {
              //select - оператор, похожий по синтаксису на
switch
              //он проходит по каждому case и
              //в случае обнаружения сообщения в канале
              //выполняет код под соответствующим case
              select {
              case killInfo := <-gs.KillNotifications:
                   //Запуск новой горутины для оповещения всех
клиентов
                   //о совершённом убийстве
                   go
gm.BroadcastNotification(&pb.ServerNotification{
                        Type:
pb.ServerNotificationType_PLAYER_KILLED,
                        Actor:    killInfo.Actor,
                        Receiver: killInfo.Receiver,
                   })
              //если ни один из case не имел канала с сообщением
              //select выполняет код под default
              default:
                   moreMessages = false
```

```
            }
        }
        <...>
}}
```

```
 // Для движущегося объекта – pkg/gamesession/actions.go
func (g *GameSession) processMoveAction(moveAction
*pb.MovementAction, playerId int32) {
     player := g.GameState.Players[int(playerId)]
     minGotYou := player.PlayerInfo.Position.X -
g.cfg.PlayerRadius
     maxGotYou := player.PlayerInfo.Position.X +
g.cfg.PlayerRadius

     <...применение перемещения аватара самого игрока...>

     if moveAction.Shift != nil {
          //Построение площади, покрытой аватаром при перемещении
          playerBody :=
collision2d.NewPolygon(collision2d.NewVector(0, 0),
collision2d.NewVector(0, 0), 0, []float64{
          float64(maxGotYou),
          float64(player.PlayerInfo.Position.Y -
moveAction.Shift.Y),
          float64(minGotYou),
          float64(player.PlayerInfo.Position.Y -
moveAction.Shift.Y),
          float64(minGotYou + moveAction.Shift.X),
          float64(player.PlayerInfo.Position.Y),
          float64(maxGotYou + moveAction.Shift.X),
          float64(player.PlayerInfo.Position.Y),
          })

          //Прохождение по массиву непроходимых областей
          intervals := make(map[int]bool)
          for _, sEntity := range g.sortedEntities {
               if sEntity.value > maxGotYou+moveAction.Shift.X {
                    break
               }
               if sEntity.value < minGotYou {
                    if sEntity.start {
                         intervals[sEntity.entityId] = true
                    } else {
                         intervals[sEntity.entityId] = false
                    }
                    continue
               }
               intervals[sEntity.entityId] = true
          }

          //Выявление активных интервалов и проверка коллизий
через
          //алгоритм узкой фазы
          for entityId, in := range intervals {
```

```
                if !in {
                     continue
                }
                areColliding, _ :=
collision2d.TestPolygonPolygon(playerBody,
g.unmovableEntities[entityId])
                if areColliding < 0 {
                     continue
                }
                //Если коллизия присутствует, откат позиции игрока
                player.Lock()
                player.PlayerInfo.Position.X =
g.PrevGameStates[g.cfg.GameStatesSaved-
1] .Players[int(player.PlayerInfo.PlayerId)].Position.X
                player.PlayerInfo.Position.Y =
g.PrevGameStates[g.cfg.GameStatesSaved-
1] .Players[int(player.PlayerInfo.PlayerId)].Position.Y
                player.Unlock()
                break
           }
     }
     return
}
```

```
ARG GOPATH=/go
ARG PKG=github.com/amikhailau/medieval-game-server
ARG GODIR=$GOPATH/src/$PKG


FROM golang:1.15.0 AS builder
ARG PKG
ARG GODIR
LABEL stage=server-intermediate
WORKDIR $GODIR
COPY . $GODIR
RUN go build -v -o /gobin/server ${PKG}/cmd/server
RUN ls /gobin


FROM alpine:3.11 AS runner
ARG GODIR

RUN mkdir -p /lib64 && \
    ln -s /lib/libc.musl-x86_64.so.1 /lib64/ld-linux-x86-64.so.2

COPY --from=builder /gobin/server /gobin/server
COPY --from=builder ${GODIR}/test/* /test/

EXPOSE 9979
ENTRYPOINT ["/gobin/server"]
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: users-service
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-service
  namespace: users-service
  labels:
    app.kubernetes.io/name: "users-service"
    app.kubernetes.io/component: "main"
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: "users-service"
      app.kubernetes.io/component: "main"
  replicas: 3
  template:
    metadata:
      labels:
        app.kubernetes.io/name: "users-service"
        app.kubernetes.io/component: "main"
    spec:
      restartPolicy: Always
      containers:
      - name: users-service
        image: amikhailau/users-service:latest
        imagePullPolicy: Always
       args:
       -  --gateway.port=8080
       -  --gateway.enable=true
       -  --app.id=users-service
       -  --logging.level=info
        ports:
          - containerPort: 8080
        resources:
          limits:
            memory: 500Mi
            cpu: 500m
          requests:
            memory: 10Mi
            cpu: 1m
```

```
func main() {
    doneC := make(chan error)
    absPath, _ := filepath.Abs("")
    mapPath := filepath.Join(absPath,
viper.GetString("gamemanager.map.file"))

    // Создание клиента Agones SDK
    agones, err := sdk.NewSDK()
    if err != nil {
         log.Fatalf("Could not connect to sdk: v",err)
    }

    <...создание GameManager и запуск обслуживания внешних
запросов...>

    stop := make(chan bool)
    // Запуск отдельной горутины для отправки извещений о том,
    // что данный экземпляр сервера функционирует нормально
    go doHealth(agones, stop)

    // Отправка извещения SDK контроллеру о том, что данный
    // экземпляр сервера готов к работе
    err = agones.Ready()
    if err != nil {
         log.Fatalf("unable to send ready status: %v\n", err)
    }

    fmt.Printf("Server Initialized! Serving on %v port\n", port)

    select {
    case err = <-doneC:
         log.Fatalf("failed to serve: %v\n", err)
    case <-gm.FinishChan:
    }

    stop <- true
    // Извещение SDK контроллера о том, что данный экземпляр
сервера
    // успешно завершил работу и может быть удалён
    err = agones.Shutdown()
    if err != nil {
         log.Fatalf("unable to send shutdown signal: %v\n", err)
    }
}
```

```
func doHealth(sdk *sdk.SDK, stop <-chan bool) {
    tick := time.Tick(3 * time.Second)
    for {
         // Отправка извещения SDK контроллеру о том, что данный
         // экземпляр сервера функционирует нормально
         err := sdk.Health()
         if err != nil {
              log.Fatalf("Could not send health ping, %v", err)
         }
         select {
         case <-stop:
              log.Print("Stopped health pings")
              return
         case <-tick:
         }
    }
}
```

```
apiVersion: "agones.dev/v1"
kind: Fleet
metadata:
  name: medieval-game-server-fleet
spec:
  replicas: 5
  scheduling: Packed
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
  template:
    spec:
      ports:
      - name: default
        portPolicy: Dynamic
        containerPort: 9979
        protocol: TCP
      health:
        initialDelaySeconds: 120
        periodSeconds: 10
        failureThreshold: 3
      # Parameters for game server sidecar
      sdkServer:
        logLevel: Info
        grpcPort: 9357
        httpPort: 9358
      # The GameServer's Pod template
      template:
        spec:
          containers:
          - name: medieval-game-server
            image: amikhailau/medieval-game-server:latest
---
apiVersion: "autoscaling.agones.dev/v1"
kind: FleetAutoscaler
metadata:
  name: medieval-game-server-fleet-autoscaler
spec:
  fleetName: medieval-game-server-fleet
  policy:
    type: Buffer
    buffer:
      bufferSize: 10
      minReplicas: 10
      maxReplicas: 100
```

```
apiVersion: "allocation.agones.dev/v1"
kind: GameServerAllocation
spec:
  required:
    matchLabels:
      agones.dev/fleet: medieval-game-server-fleet
  scheduling: Packed
  metadata:
    userIDs:
      - 8b19b90c-1e36-406c-b8e5-9b79efd4bab9
      - 6116d1e7-156c-418e-9043-940a6cf76fb6
      - 05b264c7-2020-4c37-adc5-ed1b056ead5c
      - f56f4451-d8d4-46d2-863c-04dd2dfffe69
      - 994bda28-f569-4950-a1f2-dbb203a4ff30
      - 5b1c2ad6-8c0f-4120-8403-dd0cc2947f81
      - 535418e4-1f07-40f5-9f61-012f29b541cc
      - d5f3139e-8b00-4004-929b-5cf722cfe384
```

```
os: linux
arch: arm64-graviton2
dist: focal

sudo: required
language: go
go:
    - 1.15.10
services:
    - docker

branches:
    only:
        - main
        - develop

before_script:
    - echo "$DOCKER_PASSWORD" | docker login -u
"$DOCKER_USERNAME" --password-stdin
    - make fmt-local && git diff --exit-code
    - make test-local

script:
    - 'if [[ "$TRAVIS_BRANCH" == "main" ]] &&
[[ "$TRAVIS_PULL_REQUEST" == "false" ]]; then make push
IMAGE_VERSION=latest; fi'
    - 'if [[ "$TRAVIS_BRANCH" == "develop" ]] &&
[[ "$TRAVIS_PULL_REQUEST" == "false" ]]; then make push
IMAGE_VERSION=develop; fi'
    - 'if [[ "$TRAVIS_BRANCH" != "master" ]] &&
[[ "$TRAVIS_PULL_REQUEST" == "false" ]]; then make push; fi'
```

```
PROJECT_ROOT         ?= $(PWD)
PROJECT_GOLANG_PATH  := github.com/amikhailau/medieval-game-
server
DOCKERFILE_PATH      ?= $(CURDIR)/docker/Dockerfile
MM_DOCKERFILE_PATH   ?= $(CURDIR)/docker/Dockerfile.matchmaker
IMAGE_VERSION        ?= $(shell git describe --dirty=-unsupported
--always --tags || echo pre-commit)

IMAGE_REGISTRY ?= amikhailau
IMAGE_NAME     ?= medieval-game-server
SERVER_IMAGE   := $(IMAGE_REGISTRY)/$(IMAGE_NAME)

GO_TEST_FLAGS  ?= -v -cover
GO_PACKAGES    := $(shell go list ./... | grep -v vendor | grep -
v test | grep -v .*pb)

.PHONY fmt: fmt-local
fmt-local:
     @go fmt $(GO_PACKAGES)

.PHONY test: test-local
test-local: fmt-local
     @go test $(GO_TEST_FLAGS) $(GO_PACKAGES)

docker-build:
     @docker build -f $(DOCKERFILE_PATH) -t $(SERVER_IMAGE):$
(IMAGE_VERSION) .
     @docker build -f $(MM_DOCKERFILE_PATH) -t $
(SERVER_IMAGE):matchmaker-$(IMAGE_VERSION) .
     @docker image prune -f --filter label=stage=server-
intermediate

.docker-$(IMAGE_NAME)-$(IMAGE_VERSION):
     $(MAKE) docker-build
     touch $@

.PHONY: docker
docker: .docker-$(IMAGE_NAME)-$(IMAGE_VERSION)

docker-push: docker
     @docker push $(SERVER_IMAGE):$(IMAGE_VERSION)
     @docker push $(SERVER_IMAGE):matchmaker-$(IMAGE_VERSION)

.push-$(IMAGE_NAME)-$(IMAGE_VERSION):
     $(MAKE) docker-push
```

```
	touch $@

.PHONY: push
push: .push-$(IMAGE_NAME)-$(IMAGE_VERSION)
```