УДК 33 **DOI: 10.34670/AR.2023.81.38.036**

Методы создания отказоустойчивых георасределенных систем

Костиков Юрий Александрович

Кандидат физико-математических наук,
заведующий кафедрой 812,
Московский авиационный институт
(национальный исследовательский университет),
125993, Российская Федерация, Москва, Волоколамское шоссе, 4;
e-mail: jkostikov@mail.ru

Романенков Александр Михайлович

Кандидат технических наук,
доцент кафедры 812,
Московский авиационный институт
(национальный исследовательский университет),
125993, Российская Федерация, Москва, Волоколамское шоссе, 4;
e-mail: romanaleks@gmail.com

Аннотация

Отказоустойчивость информационных систем имеет большое значение для достижения требуемого уровня надежности. Необходимость обеспечить высокий уровень стабильности системы особенно возрастает (с увеличением количества клиентов). На сегодняшний день создано множество инструментов и выработаны подходы, позволяющие обеспечить высокий уровень отказоустойчивости. В данной статье рассмотрены некоторые из перспективных подходов, используя которые была разработана и запущена георасределенная система, предоставляющая унифицированный доступ к API торговых площадок. Акцент делается не на подробную инструкцию создания системы, а на ключевые моменты и инструменты.

Для цитирования в научных исследованиях

Костиков Ю.А., Романенков А.М. Методы создания отказоустойчивых георасределенных систем // Экономика: вчера, сегодня, завтра. 2023. Том 13. № 2А. С. 388-397. DOI: 10.34670/AR.2023.81.38.036

Ключевые слова

Балансировка нагрузки, сервисы, GCP, AWS, контейнеры.

Введение

Основными атрибуты, которые присущи почти всем информационным системам, являются интерфейсы доступа к системе, бизнес-логика и хранимые данные. Каждый из этих атрибутов может являться узким местом и в течение своей работы выйти из строя и привести к краху всей систему, особенно этот факт касается распределенных систем [Бёрнс, 2019]. В связи с этим следует предпринять ряд мер, направленных на повышение стабильности каждой из отдельно взятых частей системы.

К таким мерам можно отнести построение системы с учетом возлагаемых на нет задач: определение необходимых компьютерных ресурсов для непрерывной работы, тестирование компонентов и системы в целом (модульное, интеграционное, нагрузочное, приёмочное, ручное тестирование) [Тестирование программного обеспечения. Базовый курс. EPAM Systems, 2015–2020], балансировка нагрузки и трафика для оптимального уровня загрузки сервисов, репликация баз данных (БД), детальный мониторинг системы и другие более специализированные меры.

В работе рассмотрены вышеперечисленные аспекты и их применение касательно автоматически масштабируемых георасределенных информационных систем [Атчинсон, 2018; Dhall, 2018]. В качестве модельной тестовой информационной системы, демонстрирующей подходы разработки отказоустойчивых и стабильных систем, была разработана специальная георасределенная система, которая предоставляет унифицированный доступ к API различных веб-ресурсов (торговая аналитика и операции биржевой торговли).

Непрерывная интеграция

Прежде чем приступать к разработке системы и реализации бизнес-логики, необходимо решить вопрос с доставкой новых версий продукта на продуктовую, тестовую и другие среды. Данный вопрос желательно решать на ранних этапах, чтобы доставка не производилась вручную. Настройка процессов непрерывной интеграции в начале разработки займет некоторое время, зато в дальнейшем сэкономит много человеко-часов.

Код проекта размещен на хостинге репозиториев GitLab, который включает в себя множество интегрированных инструментов для реализации DevOps методологии. Поэтому непрерывная интеграция реализована с использованием .gitlab-ci.yml файлов, в которых прописаны инструкции для компиляции проекта и его развертывания. Каждый коммит будет запускать pipeline.

Программные инструменты и технологии

Определимся с технологией для написания бизнес-логики системы. Так как задача состоит в том, чтобы сделать унифицированные API без какой-либо серьезной вычислительной нагрузки, можно выбрать актуальные технологии с большим количеством библиотек и модулей, возможностью создавать web-приложения, наличием активного сообщества, а также обеспечивающие высокую скоростью разработки конечных решений. Наиболее подходящие под такие требования технологии – это Python и Node.js. Однако у Node.js есть преимущество в том, что сервисы для данной платформы можно разрабатывать на языке программирования TypeScript.

TypeScript – это компилируемый JavaScript язык программирования, который в 2012 году

представила компания Microsoft. Ключевое преимущество данного языка программирования заключается в том, что он является статически типизированным языком. В таком случае тип переменной или функции определяется на этапе объявления. Данная характеристика чрезвычайно важна при разработке отказоустойчивых систем, так как значительную часть ошибок можно предотвратить благодаря строгой системе типов.

При создании программного кода необходимо учитывать следующие фундаментальные аспекты:

1) Код должен быть понятен другим разработчикам, которые будут его поддерживать.
2) Добавление тестов не должно изменять существующий код.
3) Добавление нового функционала не должно усложнять структуру проекта и увеличивать время для реализации последующих обновлений бизнес-логики.

Эти пункты являются следствием из общепризнанных принципов разработки, таких как SOLID, KISS, DRY и других. Доминирующие позиции для разработки web-приложений в мире Node.js занимают следующие библиотеки: express, sails.js, loopback.js, koa.js, fastify, polka, restify и другие, а также основанные на их базе фреймворки. Изучив документацию каждой из библиотек, можно заметить, что построены они все на схожих принципах, но их производительность различается.

Эмпирически установлено, что наиболее удобным для разработки оказался фреймворк express из-за большого сообщества разработчиков, множества инструментов, расширяющих функционал express и богатую документацию. Однако, несмотря на все достоинства express, он не является самым производительным среди своих аналогов на платформе Node.js. Но время, затрачиваемое на работу самого фреймворка, незначительно по сравнению со временем выполнения бизнес-логики, запросов к данным и обращению к API внешних ресурсов. Предпочтение было отдано фреймворку Nest JS [NestJS. Documentation, www], потому что этот фреймворк позволяет определить минимальный каркас приложения, включающий модульную структуру, и глобальные перехватчики исключений (что важно при разработке отказоустойчивого приложения). Данный фреймворк также позволяет декларативно описывать правила для автоматического преобразования и валидации входящих DTO и разделять бизнес логику, уровень доступа к данным, web уровень.

Минимальное приложение на Node.js приведено в листинге 1:

```
import { NestFactory, Module } from '@nestjs/core';
import { Injectable, Controller, Get } from '@nestjs/core';

@Injectable()
export class ApplicationService {
  getVersion(): number {
    return 1;
  }
}
@Controller()
export class ApplicationController {
  constructor(
    private readonly applicationService: ApplicationService,
  ) {}
@Get('version')
```

```
  getVersin() {
    const version = this.applicationService.getVersion();
    return { version };
  }
}
@Module({
  imports: [],
  controllers: [ApplicationController],
  providers: [ApplicationService],
})
export class ApplicationModule {}

async function bootstrap() {
  const application = await NestFactory.create(ApplicationModule);
  await application.listen(3000);
}
bootstrap();
```

Листинг 1

В листинге 1 запускается web-сервер на порту 3000, который при выполнении HTTP GET запроса по пути /version вернет следующий JSON объект:

```
{
  "version": 1
}
```

Листинг 2

В данном примере бизнес-логика отделена от web-контроллера, что позволяет поддерживать чистую архитектуру, которая в дальнейшем обеспечит возможность гибкой быстрой модернизации программного продукта.

Для запуска разработанной системы необходимы сервера, которые можно арендовать у облачных провайдеров (Infrastructure-as-a-Service, IaaS) либо использовать свои собственные вычислительные мощности. Однако разворачивать собственные сервера очень дорого, особенно если необходимо располагать их в различных частях света. Таким образом, для запуска системы было отдано предпочтение облачным провайдерам, среди которых было выделено три провайдера: Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, инфраструктура которых развернута по всему миру.

После анализа достоинств и недостатков каждого из провайдеров для разрабатываемой системы было принято решение использовать GCP по следующим причинам:

1) AWS и Azure больше ориентированы на корпоративных клиентов.
2) Документация у GCP лучше, чем у AWS и Azure.
3) Самая большая в мире сеть, глобальный load balancing.

Система управления базой данных (СУБД)

Для надежного хранения данных следует выбирать такие инструменты, которые были разработаны именно для этих целей, так будет меньше непредвиденных проблем. Одним из таких инструментов является СУБД CockroachDB. Ключевые преимущества данной СУБД

следующие:

1) Поддержка SQL API как у PostgreSQL (с некоторыми ограничениями).

2) Встроенная поддержка масштабирования.

3) Поддержка высокой производительности при масштабировании.

4) Встроенная поддержка распределения данных между экземплярами.

5) Спроектирован с учетом запуска в Kubernetes.

Для обеспечения сохранности данных и непрерывной доступности в случае отказа или ошибки в контейнере с СУБД или недоступности центра обработки данных (ЦОД) была следующим образом настроена CockroachDB:

1) В каждом регионе в одной зоне запущено по три экземпляра CockroachDB.

2) Используется репликация между узлами внутри региона.

3) Используется репликация между узлами в разных регионах.

Для подключения к СУБД из Node.js приложения используется TypeORM [TypeORM. Documentation, www] – ORM библиотека с обширным функционалом, декларативным описанием сущностей, которые являются отображением таблиц в БД, множеством функций для работы с данными и встроенной поддержкой миграций. Миграции – инструмент для изменения схемы БД, который представляет собой набор SQL инструкций, которые выполняются при запуске приложения. Для описания таблицы с пользователями используется класс UserEntity, приведенный в листинге 3. При помощи декораторов описываются свойства таблицы и колонок.

```
import { Entity, Column, PrimaryColumn, Generated } from 'typeorm';
import { CreateDateColumn, UpdateDateColumn } from 'typeorm';
@Entity('users')
export class UserEntity {
  @PrimaryColumn({ name: 'id' })
  @Generated('uuid')
  id!: string;
  @Column({ type: 'varchar', length: 64, unique: true, name: 'id' })
  username!: string;
  @Column({ type: 'varchar', name: 'password_hash' })
  passwordHash!: string;
  @CreateDateColumn({ name: 'created_timestamp' })
  createdTimestamp!: Date;
  @UpdateDateColumn({ name: 'updated_timestamp' })
  updatedTimestamp!: Date;
 }
```

Листинг 3

Несмотря на все достоинства CockroachDB, она все же не обеспечивает такую высокую скорость доступа, как in-memory решения для хранения данных. К наиболее зарекомендовавшим решениям с возможностью хранения данных в памяти можно отнести Redis, Memcached и MongoDB. Все эти инструменты могут использоваться для наиболее быстрого доступа к данным.

Такой способ хранения данных отлично подходит для кэширования результатов из БД, хранения jwt или других часто используемых данных, которые не критично потерять в случае остановки или перезапуска хранилища. Стоит понимать, что реляционные СУБД и перечисленные in-memory решения предназначены для решения разного рода задач. Поэтому

часто на практике применяется комбинация различных хранилищ для достижения оптимального результата.

Выбирая между ПО для хранения данных в оперативной памяти, следует уделить внимание на такие вещи, как скорость доступа к данным, поддержка механизмов горизонтального масштабирования, поддерживаемые типы данных и дополнительный функционал, который может быть полезен. Исходя из перечисленных требований и характеристик выбранных инструментов, было решено использовать Redis в связи с тем, что у Redis множество встроенных типов данных, master-slave репликация и поддержка подписок и уведомлений.

В разработанной системе в CockroachDB будут храниться пользовательские данные, такие как email, login, хэш пароля, дата регистрации и другое. А в Redis будут храниться access и refresh токены, кэшироваться некоторые результаты из CockroachDB. Кроме того, благодаря поддержке подписок и уведомлений в Redis, он пригоден для общения между сервисами внутри региона. Однако для общения между регионами Redis не подходит, для этого используется Google Pub/Sub – ПО для общения между сервисами посредством подписок и публикацией сообщений.

Оркестрация контейнеров

Для того чтобы управлять запущенными сервисами, необходим инструмент, который умеет следить за состоянием сервисов (health check, readiness probe) и в случае ошибки перезапускать их. В качестве такого инструмента был выбран Kubernetes [Джиджи, 2019], потому что он способен запускать множество сервисов, упакованных в контейнеры, балансировать нагрузку между ними и работать на нескольких хостах. Также есть возможность использовать несколько кластеров, что особенно важно для георасределенной системы [Лукша, 2019].

Описание конфигурации вычислительных ресурсов, настроек запущенных сервисов, переменных окружения и других параметров происходит при помощи манифестов – специальных файлов с расширением yaml. Таким образом реализуется подход Infrastructure-as-Code (IaC), который автоматизируя процесс конфигурирования и настройки среды ускоряет запуск продукта, снижает цену и уменьшает риски, связанные с человеческим фактором. Для данной работы использовался следующий манифест для развертывания сервиса:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-service
  template:
    metadata:
      labels:
        app: api-service
    spec:
      containers:
```

```
- name: api-service
  image: <IMAGE>
  ports:
  - containerPort: 5000
  livenessProbe:
    httpGet:
      path: /health
      port: 5000
    initialDelaySeconds: 15
  env:
  - name: NODE_ENV
    value: 'production'
```

Листинг 4

Манифест, приведенный в листинге 4, описывает, как будет происходить развертывание сервиса, задает переменные окружения для контейнера и указывает endpoint для проверки работоспособности контейнера. В случае если в контейнере случится ошибка, не позволяющая продолжать дальнейшую работу, то Kubernetes перезапустит контейнер. Для применения этого файла конфигурации используется команда, приведенная в листинге 5:

```
kubectl apply –f <file_name>
```

Листинг 5

Наиболее распространенным контейнером, который поддерживает Kubernetes, является Docker. Данное ПО позволяет создавать контейнеры с приложениями и их зависимостями и запускать их на любой машине, поддерживающей Docker. Основные операционные системы (Mac OS, Linux-системы, Windows) на которых ведется разработка и эксплуатация ПО, имеют поддержку Docker. Для построения контейнера используется Dockerfile из листинга 6. Данный файл определяет рабочую директорию для контейнера, скачивает необходимые зависимости и компилирует TypeScript код в JavaScript код. В данном Dockerfile для уменьшения размера конечного контейнера используется node:14-alpine в качестве базового image.

```
FROM node:14-alpine
USER node
WORKDIR /home/node
COPY . /home/node
RUN npm ci
RUN npm run build
CMD ["node", "dist/main.js"]
```

Листинг 6

Однако данный Dockerfile возможно улучшить, применив технологию многоэтапного построения (multi-stage builds) – отдельно устанавливая все зависимости, в том числе и dev-зависимости, для компиляции проекта, после чего оставлять только продуктовые зависимости.

Балансирование и мониторинг нагрузки

Для балансирования нагрузки между серверами используется связка из Google HTTP Load Balancer, глобального Multi Cluster Ingres и локального Ingress контроллера с Kubernetes Ingress внутри. Такая комбинация балансировщика нагрузки и веб-серверов для маршрутизации

трафика позволяет перенаправлять трафик сначала на ближайший Nginx, который перенаправляет запрос на конечный сервер.

Помимо этого, за счет использования Ingress появляется возможность отказаться от жесткой привязки маршрута к конкретному узлу, чего нельзя было бы добиться, используя, например, только Nginx.

Вышеописанного хватает для запуска системы, устойчивой к отказам компонентов, однако при возникновении различного рода ошибок они останутся незамеченными до тех пор, пока о них не сообщит пользователь или администратор не увидит их в логах системы. К тому же с увеличением числа компонентов в системе и расширением функционала потенциальных точек отказа будет становиться все больше и больше. Это все приводит к необходимости настроить системы мониторинга и уведомления в случае чрезвычайных ситуаций.

GCP позволяет настроить мониторинг на основе множества встроенных метрик, но, кроме этого, можно создавать собственные метрики на основе логов системы. Поэтому было принято решение реализовать мониторинг, который в случае обнаружения проблем в логах будет создавать публикацию в Pub/Sub, которая, в свою очередь, будет запускать cloud function, внутри которой будет посылаться сообщение с ошибкой в Telegram канал. Для реализации этого необходимо:

1) Записывать логи в соответствии с требованиями GCP, чтобы они учитывались.
2) Создать метрику для анализа логов.
3) Создать тему в Pub/Sub.
4) Создать Cloud Function для отправки сообщения в Telegram, запускаемую при появлении сообщения с Pub/Sub.
5) Создать канал для мониторинга, выполняющий публикацию в Pub/Sub.
6) Создать политику уведомлений, связывающую канал уведомлений и метрики, анализирующие логи.

Такая система позволит оперативно узнавать об исключительных ситуациях, возникающих во время работы системы.

Заключение

Разработанная система успешно выполняет поставленную перед собой задачу, и в случае отказа одного из компонентов (падение СУБД, обработчика запросов или временная недоступность региона) переключается на другой работоспособный компонент.

В то же время существуют возможности улучшения данной системы. Они заключаются в использовании более продвинутой системы комплексного мониторинга серверов и сервисов, в задачи которых входит мониторинг таких вещей, как размер свободного места на диске, среднее время ответа, health check's и многое другое. Также если появится задача производить какие-либо сложные вычисления, требующие значительные ресурсы, то имеет смысл эти места переписать с использованием более производительных технологий.

Библиография

1. Атчинсон Л. Масштабирование приложений. Выращивание сложных систем. СПб.: Питер, 2018. 256 с.
2. Бёрнс Б. Распределенные системы. Паттерны проектирования. СПб.: Питер, 2019. 224 с.
3. Джиджи С. Осваиваем Kubernetes. Оркестрация контейнерных архитектур. СПб.: Питер, 2019. 400 с.
4. Лукша М. Kubernetes в действии. М.: ДМК Пресс, 2019. 672 с.

5. Тестирование программного обеспечения. Базовый курс. EPAM Systems, 2015–2020.
6. Dhall C. Scalability Patterns: Best Practices for Designing High Volume Websites. 2018.
7. NestJS. Documentation. URL: https://docs.nestjs.com.
8. TypeORM. Documentation. URL: https://typeorm.io.
9. Qureshi A. Power-demand routing in massive geo-distributed systems : дис. – Massachusetts Institute of Technology, 2010.
10. Abu-Lebdeh M. et al. NFV orchestrator placement for geo-distributed systems //2017 IEEE 16th International Symposium on Network Computing and Applications (NCA). – IEEE, 2017. – С. 1-5.

Methods for creating fault-tolerant geo-distributed systems

Yurii A. Kostikov

PhD in Physical and Mathematical Sciences,
Head of the Department 812,
Moscow Aviation Institute (National Research University),
125993, 4 Volokolamskoe highway, Moscow, Russian Federation;
e-mail: jkostikov@mail.ru

Aleksandr M. Romanenkov

PhD in Technical Sciences, Associate Professor,
Department 812,
Moscow Aviation Institute (National Research University),
125993, 4 Volokolamskoe highway, Moscow, Russian Federation;
e-mail: romanaleks@gmail.com

Abstract

The fault tolerance of information systems is of great importance for achieving the required level of reliability. The need to ensure a high level of system stability especially increases (with an increase in the number of clients). To date, many tools have been created and approaches have been developed to ensure a high level of fault tolerance. This article discusses some of the promising approaches used to develop and launch a geo-distributed system that provides unified access to the API of trading platforms. The emphasis is not on detailed instructions for creating a system, but on key points and tools.

For citation

Kostikov Yu.A., Romanenkov A.M. (2023) Metody sozdaniya otkazoustoichivykh georaspredelennykh sistem [Methods for creating fault-tolerant geo-distributed systems]. *Ekonomika: vchera, segodnya, zavtra* [Economics: Yesterday, Today and Tomorrow], 13 (2A), pp. 388-397. DOI: 10.34670/AR.2023.81.38.036

Keywords

Load balancing, services, GCP, AWS, containers.

References

1. Atchinson L. (2018) *Masshtabirovanie prilozhenii. Vyrashchivanie slozhnykh system* [Application scaling. Growing complex systems]. Saint Petersburg: Piter Publ.
2. Berns B. (2019) *Raspredelennye sistemy. Patterny proektirovaniya* [Distributed systems. Design patterns]. Saint Petersburg: Piter Publ.
3. Dhall C. (2018) *Scalability Patterns: Best Practices for Designing High Volume Websites*.
4. Dzhidzhi S. (2019) *Osvaivaem Kubernetes. Orkestratsiya konteinernykh arkhitektur* [Orchestration of container architectures]. Saint Petersburg Piter Publ.
5. Luksha M. (2019) *Kubernetes v deistvii* [Kubernetes in action]. Moscow: DMK Press.
6. *NestJS. Documentation*. Available at: https://docs.nestjs.com [Accessed 12/01/2023].
7. *Testirovanie programmnogo obespecheniya. Bazovyi kurs. EPAM Systems* [Software testing. Basic course. EPAM Systems] (2015–2020).
8. *TypeORM. Documentation*. Available at: https://typeorm.io [Accessed 19/01/2023].
9. Qureshi, A. (2010). Power-demand routing in massive geo-distributed systems (Doctoral dissertation, Massachusetts Institute of Technology).
10. Abu-Lebdeh, M., Naboulsi, D., Glitho, R., & Tchouati, C. W. (2017, October). NFV orchestrator placement for geo-distributed systems. In 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA) (pp. 1-5). IEEE.