# Methods of Creating Fault-Tolerant Geographically Distributed Systems

**Romanenkov Aleksandr Mikhailovich**
Federal State Budgetary Educational Institution of Higher Education "Moscow Aviation Institute (National Research University)",
Federal State Budgetary Educational Institution of Higher Education "Russian State University named after A.N. Kosygin (Technologies. Design. Art)" romanaleks@gmail.com
Orcid 0000-0002-0700-8465

**Temnov Aleksei Andreevich**
Federal State Budgetary Educational Institution of Higher Education "Moscow Aviation Institute (National Research University)" temnov.aleksei@gmail.com
Orcid 0000-0002-3685-7802

**Kostoev Adam Timurovich**
Federal State Budgetary Educational Institution of Higher Education "Russian State University named after A.N. Kosygin (Technologies. Design. Art)" a.kostoev@inbox.ru
Orcid 0000-0003-1012-7980

***Article Info***
***Page Number: 168 – 176***
***Publication Issue:***
***Vol 71 No. 2 (2022)***

***Article History***
***Article Received**: 30 December 2021*
***Revised**: 28 January 2022*
***Accepted**: 12 March 2022*
***Publication**: 31 March 2022*

***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). Today, many tools have been created and approaches have been developed to ensure a high level of fault - tolerance [9-11]. In this work, some of the promising approaches are considered, using which a geographically distributed system was developed and launched that provides unified access to the API of trading platforms. The emphasis is not on detailed instructions for creating the system, but on the key points and tools.

**Keywords:** fault - tolerance, information systems, geographically distributed systems.

## Introduction

The main characteristics in almost all information systems are system access interface, business logic and stored data. Each of these places can fail and disable the system, especially for distributed systems [8]. In this regard, a number of measures should be taken to improve the stability of each of the individual parts of the system.

Such measures include building a system taking into account the tasks assigned to it: determining the necessary computer resources for continuous operation, testing components and the system as a whole (modular, integration, load, acceptance, manual testing) [1], load and traffic balancing, for optimal level of service loading, database (db) replication, detailed system monitoring and other more specialized measures.

Further, the above aspects and their application regarding automatically scalable geographically distributed information systems are considered [3, 5].

**Problem statement**

As a task demonstrating approaches to the development of fault-tolerant and stable systems, it was decided to develop a geographically distributed system that provides unified access to the API of various web resources (trading analytics and exchange trading operations).

**Continuous integration**

Before proceeding with the development of the system and the implementation of business logic, it is necessary to solve the issue of delivering new versions of the product to the product, test, and other environments. It is advisable to solve this issue at an early stage so that delivery is not carried out manually. Setting up continuous integration processes at the beginning of development will take some time, but it will save a lot of man-hours later on.

The project code is hosted on the GitLab repository hosting, which includes many integrated tools for implementing the DevOps methodology. Therefore, continuous integration is implemented using .gitlab-ci.yml files that contain instructions for compiling the project and deploying it. Each commit will start a pipeline.

**Tools**

Let's decide on the technology for writing the business logic of the system. Since the task is to make unified APIs without any serious computational load, one can choose current technologies with a large number of libraries and modules, the ability to create web applications, the presence of an active community, as well as providing a high-speed development of final solutions. The most suitable technologies for such requirements are Python and Node.js. However, the Node.js has the advantage that services for this platform can be developed in the TypeScript programming language.

TypeScript is a compiled JavaScript programming language introduced by Microsoft in 2012. The key advantage of this programming language is that it is a statically typed language. In this case, the type of variable or function is determined at the declaration stage. This characteristic is extremely important when developing fault-tolerant systems, since a significant part of errors can be prevented thanks to a strict type system.

**Service architecture**

When writing code, the following aspects should be taken into account:

1) The code should be understandable to other developers who will support it.

2) Adding tests should not change existing code.

3) Adding new functionality should not complicate the project structure and increase the time for implementing subsequent business logic updates.

These aspects are a consequence of generally accepted development principles such as SOLID, KISS, DRY and others.

The following libraries occupy the dominant positions for developing web applications in the world of Node.js: express, sails.js, loopback.js, koa.js, fastify, polka, restify and others, as well as frameworks based on them. After studying the documentation of each of the libraries, one can see that they are all built on similar principles, but their performance differs.

The express framework turned out to be the most convenient for development due to the large developer community, many tools that extend the express functionality and rich documentation. However, despite all the advantages of express, it is not the most productive among its counterparts on the Node.js platform. But the time spent on the work of the framework itself is insignificant compared to the execution time of business logic, data requests and access to the API of external resources.

The preference was given to the Nest JS framework [2] because it allows to define an application framework that includes a modular structure and global exception interceptors (which is important when developing fault-tolerant application). This framework also allows to declaratively describe rules for automatic conversion and validation of incoming DTOs and separate business logic, data access layer, web level.

Minimal Node.js application is shown in Listing 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();
```

Listing 1

Listing 1 starts a web server on port 3000, which, when executing an HTTP GET request along the route/version, will return the following JSON object:

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

Listing 2

In this example, business logic is separated from the web controller, which allows to maintain a clean architecture.

**Infrastructure**

To run the developed system, one need servers that can be rented from cloud providers (Infrastructure-as-a-Service, IaaS) or use own computing power. However, it is very expensive to deploy own servers, especially if one need to locate them in different parts of the world. Thus, to launch the system, preference was given to cloud providers, among which three providers were singled out: Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, whose infrastructure is deployed worldwide.

After studying the advantages and disadvantages of each of the providers, it was decided to use GCP for the following reasons:

1) AWS and Azure are more focused on corporate clients.
2) GCP has better documentation than AWS and Azure.
3) The world's largest network, global load balancing.

**Fault-tolerant database management system (DBMS)**

For reliable data storage, one should choose tools that have been developed specifically for this purpose, so there will be fewer unforeseen problems. One of these tools is CockroachDB DBMS. The key advantages of this DBMS are:

1. SQL API support like PostgreSQL (with some limitations).
2. Built-in zoom support.
3. Support high performance when scaling.
4. Built-in support for data distribution between instances.
5. Designed to run on Kubernetes.

To ensure data integrity and continuous availability in the event of failure or error in the DBMS container or the unavailability of the data processing center (DPC), CockroachDB was configured as follows:

1. Each region has 3 instances of CockroachDB running in one zone.
2. Replication is used between nodes within the region.
3. Replication is used between nodes in different regions.

To connect to the DBMS from the Node.js application, TypeORM [4] is used - an ORM library with extensive functionality, a declarative description of entities that represent tables in the database, many functions for working with data and built-in support for migrations. Migrations is a tool for changing the database schema, which is a set of SQL statements that are executed when the application is launched. The UserEntity class, shown in Listing 3, is used to describe a table with users. Decorators are used to describe table and column properties.

```
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;
}
```

**Listing 3**

**In-memory DBMS**

Despite all the advantages of CockroachDB, it still does not provide such high access speed as in-memory data storage solutions. The most established in-memory storage solutions include Redis, Memcached, and MongoDB. All of these tools can be used for the fastest access to data.

This way of storing data is great for caching results from the database, storing jwt or other frequently used data that is not critical to lose if the storage is stopped or restarted. It should be understood that relational DBMS and the listed in-memory solutions are designed to solve various kinds of problems. Therefore, in practice, a combination of different storage facilities is often used to achieve optimal results.

When choosing between software for storing data in RAM, one should pay attention to such things as: data access speed, support for horizontal scaling mechanisms, supported data types and additional functionality that may be useful. Based on the listed requirements and the characteristics of the selected tools, it was decided to use Redis due to the fact that Redis has many built-in data types, master-slave replication, and support for subscriptions and notifications.

In the developed system, CockroachDB will store user data, such as email, login, password hash, registration date, and more. And Redis will store access and refresh tokens, cache some results from CockroachDB. In addition, thanks to Redis' support for subscriptions and notifications, it is suitable for communication between services within region. However, Redis is not suitable for communication between regions; Google Pub / Sub is used for this i.e., software for communication between services through subscriptions and posting messages.

**Container orchestration**

In order to manage running services, one need a tool that can monitor the status of services (health check, readiness probe) and restart them in case of an error. Kubernetes [6] was chosen as such a tool because it is able to run many services packaged in containers, load balance between them, and run on multiple hosts. It is also possible to use multiple clusters, which is especially important for geographically distributed system [7].

The description of the configuration of computing resources, settings of running services, environment variables and other parameters occurs using manifests - special files with yaml extension. Thus, the Infrastructure-as-Code (IaC) approach is implemented by automating the process of configuring and setting up the environment accelerates the launch of the product, reduces the price, and reduces the risks associated with the human factor. For this work, the following manifest was used to deploy the service:

```
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'
```

**Listing 4**

The manifest given in Listing 4 describes how the service will be deployed, sets environment variables for the container, and specifies the endpoint for checking the container's health. If an error occurs in the container that does not allow further work to continue, Kubernetes restarts the container. To apply this configuration file, the command given in listing 5 is used:

```
kubectl apply –f <file_name>
```

**Listing 5**

The most common container that supports Kuberenetes is Docker. This software allows to create containers with applications and their dependencies and run them on any machine that supports Docker. The main operating systems (Mac OS, Linux systems, Windows) on which the software is developed and operated have Docker support. The Dockerfile from Listing 6 is used to build the container. This file defines the working directory for the container, downloads the required dependencies, and compiles the TypeScript code into JavaScript code. This Dockerfile uses node:14-alpine as the base image to reduce the size of the final container.

```
FROM node:14-alpine

USER node
WORKDIR /home/node

COPY . /home/node

RUN npm ci
RUN npm run build

CMD ["node", "dist/main.js"]
```

**Listing 6**

However, this Dockerfile can be improved by using the multi-stage builds technology - separately installing all dependencies, including dev dependencies, to compile the project, after which only product dependencies are left.

**Load balancing**

To balance the load between servers, a bundle of Google HTTP Load Balancer, global Multi Cluster Ingres, and local Ingress controller with Kubernetes Ingress inside are used. This combination

of load balancer and web servers to route traffic allows to redirect traffic first to the nearest Nginx, which redirects the request to the destination server.
In addition, due to the use of Ingress, it becomes possible to abandon the hard binding of the route to the specific node, which could not be achieved using, for example, only Nginx.

**Monitoring**

The above is enough to start a system that is resistant to component failures, however, if various kinds of errors occur, they will go unnoticed until they are reported by the user, or the administrator sees them in the system logs. In addition, with the increase in the number of components in the system and the expansion of the functionality, the potential points of failure will become more and more. This all leads to the need to set up monitoring and notification systems in case of emergencies.

GCP allows to configure monitoring based on many built-in metrics, but one can also create own metrics based on system logs. Therefore, it was decided to implement monitoring, which, if problems are detected in the logs, will create a publication in Pub / Sub, which in turn will launch a cloud function, inside which an error message will be sent to the Telegram channel. To implement this, one need:

1. Record logs in accordance with GCP requirements so that they are considered.
2. Create a metric for analyzing logs.
3. Create a topic in Pub/Sub.
4. Create a Cloud Function to send a message to Telegram, triggered when a message appears from Pub/Sub.
5. Create a monitoring channel that publishes to Pub/Sub.
6. Create a notification policy linking the notification channel and metrics that analyze logs.

Such system will allow to quickly learn about exceptional situations that arise during the operation of the system.

## Conclusions

The developed system successfully fulfills the task set for itself, and in the event of failure of one of the components (DBMS crash, query processor or temporary unavailability of the region), it switches to another operable component.

At the same time, there are opportunities to improve this system. They consist in using a more advanced system of comprehensive monitoring of servers and services, whose tasks include monitoring such things as the size of free disk space, average response time, health check, and much more. Also, if there is a task to perform any complex calculations that require significant resources, then it makes sense to rewrite these places using more productive technologies.

## References

[1]. Software testing. Basic course. EPAM Systems, 2015–2020.
[2]. NestJS. Documentation. [Electronic resource]. URL: https://docs.nestjs.com/
[3]. Atchinson L. Application scaling. Growing of complex systems. Saint Petersburg: Saint Petersburg, 2018. 256 p.
[4]. TypeORM. Documentation. [Electronic resource]. URL: https://typeform.io/
[5]. Dhall C. Scalability Patterns: Best Practices for Designing High Volume Websites. 2018.

[6]. Gigi S. Mastering Kubernetes. Orchestration of container architectures. Saint Petersburg: Saint Petersburg, 2019. 400 p.

[7]. Luksha M. Kubernetes in action. M.: DMK Press, 2019. 672 p.

[8]. Burns B. Distributed systems. Design patterns. Saint Petersburg: Saint Petersburg, 2019. 224 p.

[9]. Dimitrova, Rayna and Bernd Finkbeiner. "Synthesis of Fault-Tolerant Distributed Systems." ATVA (2009).

[10]. Sari, Arif & Akkaya, Murat. (2015). Fault Tolerance Mechanisms in Distributed Systems. International Journal of Communications, Network and System Sciences. 8. 471-482. 10.4236/ijcns.2015.812042.

[11]. Zawirski, Marek & Bieniusa, Annette & Balegas, Valter & Duarte, Sérgio & Baquero, Carlos & Shapiro, Marc & Preguiça, Nuno. (2013). SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Proceedings of the IEEE Symposium on Reliable Distributed Systems. 10.1109/SRDSW.2014.33.