Highload Backend Assignment 3 Report

Prepared by:

Bogdatov Chingis

ID: 21B030792

Prepared for:

Highload Backend class

Date:

October 31, 2024

Table of Contents

Introduction

In modern software engineering there is a tendency towards creating applications that are able to handle a huge number of requests per second. Why is it important? Because modern business has to implement information systems in order to gain a competitive advantage. Many services are now available online, making it easier and faster to use. Millions of people can use the same application simultaneously. So, the modern software engineers are implementing this kind of high-load resistance. However, there are a lot of things to consider, such as data consistency, scaling, monitoring and how to properly implement these points. This paper is devoted to explaining different technologies that are used to provide users with a better experience.

Distributed Systems and Data consistency

Overview

Distributed information systems are networks of multiple, dispersed by either geographical or logical level computers or nodes that collaborate to process, store, and retrieve information in a coordinated manner. These systems play a fundamental role in modern computing, enabling organizations to manage large volumes of data, support real-time applications, and handle distributed workloads across various locations.

The positive outcome is that such systems are more likely to safely handle a huge number of requests by different users, therefore making companyies' services more available and providing operational excellence. However, it leads to one question. The question is "How do we make sure that all of the user data is

consistent amongst all of the nodes?”. Making data consistent is a pretty hard challenge to accomplish.

Exercises

Exercise 1 was about implementing a simple application with key value storage. The challenge was to set multiple nodes of application and make sure that data stays consistent. How do we properly check that? We can use a technique called quorum reads and quorum writes. As the name says, the system is considered to work correctly when a certain number of nodes are responding similarly to each other.

So, apart from setting app the django application, I also wrote a custom python script that checks whether all nodes respond in similar way.

```
import requests

class QuorumTester:
    def perform_test(self, hosts: list[str]) -> bool:
        pass
class ReadQuorumTester(QuorumTester):
    R = 2
    def perform_test(self, hosts: list[str]) -> bool:
        responses = []
        for host in hosts:
            try:
                response = requests.get(url=f"{host}/api/name/")
                if response.status_code == 200:
                    responses.append(response.json()["value"])
            except:
                continue
        return max(responses, key=responses.count) >= self.R
class WriteQuorumTester(QuorumTester):
    W = 2
    def perform_test(self, hosts: list[str]) -> bool:
        responses_count = 0
        for host in hosts:
            try:
                response = requests.put(url=f"{host}/api/name",
data={"value": "new_value"})
                if response.status_code == 200:
                    responses_count += 1
```

The approach is a pretty straightforward method, allowing to customize your quorum testability and enabling auto tests as well.

Exercise 3

Load balancing is a technique aimed at implementing distributed systems. The point is that a user connects to a load balancing system that decides which of the application nodes should respond to a call. I chose to use Nginx as a load balancing system, since the configuration is pretty easy and familiar.

```
events {}
http{
  upstream django_servers{
    server django_app_1:8000;
    server django_app_2:8001;
    server django_app_3:8002;
  }
  server{
    listen 80;
    location / {
      proxy_pass http://django_servers;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $host;
```

That is how my configuration looks like. I basically tell all of the applications to a config file and redirect them appropriately. So, when a user connects to 80th port of the host and then gets redirected to an application. It allows you to handle more connections simultaneously.

```
cingisbogdatov@MBP-Cingis-2 ~ % wrk -t12 -c400 -d30s http://localhost:8001/api/name
Running 30s test @ http://localhost:8001/api/name
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   450.29ms   46.83ms 487.55ms   96.34%
    Req/Sec    72.75     35.79   230.00     68.78%
  16519 requests in 30.02s, 4.84MB read
  Socket errors: connect 0, read 337, write 0, timeout 0
Requests/sec:    550.22
Transfer/sec:    164.96KB
```

Without load balancer – directly to django instance

```
cingisbogdatov@MBP-Cingis-2 ~ % wrk -t12 -c400 -d30s http://localhost:80/api/name
Running 30s test @ http://localhost:80/api/name
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   270.26ms  412.82ms   1.91s    83.86%
    Req/Sec     1.66k     1.64k    6.72k    69.43%
  172619 requests in 30.03s, 51.69MB read
  Socket errors: connect 0, read 390, write 0, timeout 1030
  Non-2xx or 3xx responses: 172619
Requests/sec:   5748.80
Transfer/sec:      1.72MB
```

With load balancer

Here are the test results with use of the wrk tool that allows to generate artificial traffic for test purposes. As it's seen from the results, application with implementation of the load balancing handles roughly more than 10 times number of the requests.

Exercise 4

Exercise 4 was aimed at implementing database replication techniques. It allows a developer to distribute the database load appropriately. It is obvious that most of the systems are designed for retrieving information rather than changing it. So, replication is a technique that creates copies of main database and handles reads from replicas rather than the main database instance. This makes readings from database faster but as a drawback it takes longer to implement and maintain the whole process.

Findings

After implementing all of the exercises connected to distributing an information system, I found out that the before mentioned techniques

get the job done, dramatically changing the computational capabilities of a system. There are lot of challenges during implementation of distributed systems, but the result is more than satisfactory.

Scaling Backend systems

Overview

Scaling a system is a process that increases the computational power of the system. There are 2 types of scaling: vertical and horizontal. Vertical scaling simply increases the computational power of one node, adding more CPU power, memory both RAM and persistent memory. It’s much simpler, however it’s pricier. There are certain limitations on how one is able to increase the node’s power.

The horizontal approach concentrates on distributing the load between the smaller nodes. It’s harder, but the price is much lower. So, as a scaling type I chose the horizontal approach.

Exercises

Exercise 4 and 2 are both aimed at scaling the system. Both of these tasks allow to increase the number of computational nodes.

Findings

Scaling a system is a rough challenge, mainly because of the things a developer needs to consider while creating an application. Tools are hard to learn, everything falls out of hand. However, it is more of a requirement than a wish. Things are easier since a lot of people face the same problem.

Monitoring and Observability

Overview

In backend systems, monitoring and observability are crucial for ensuring reliability, performance, and scalability. **Monitoring** is the process of collecting metrics and log data to understand system health, track performance, and detect anomalies. **Observability** goes a step further, providing insights into system behavior and enabling

root cause analysis when issues arise. Together, monitoring and observability empower development and operations teams to respond proactively to potential issues, reduce downtime, and maintain optimal performance under varying load conditions.

Modern systems benefit from real-time data visibility across all components, allowing teams to quickly identify bottlenecks, debug problems, and make data-driven decisions to improve application efficiency. This report summarizes the setup, configuration, and findings from the exercises implemented to enhance the monitoring and observability of a Django-based application.

Exercises

Exercise 2: Analyze Consistency Models

- Researched various consistency models, including strong consistency, eventual consistency, causal consistency, and read-your-writes consistency.
- Discussed the trade-offs each model presents, with strong consistency ensuring data accuracy but at a cost to availability, and eventual consistency favoring availability and scalability.
- Outlined how each model could be implemented in Django, from using ACID-compliant databases for strong consistency to implementing a distributed cache for eventual consistency, highlighting different options for supporting consistency in Django applications.

Exercise 3: Containerize and Scale Django Application with NGINX

- Built a scalable infrastructure by introducing an Nginx load balancer to manage traffic across multiple Django instances.
- Configured Docker Compose to run multiple Django containers, promoting high availability and supporting increased load.

Exercise 4: Integrate Monitoring Tools (Prometheus and Grafana)

- Integrated Prometheus and Grafana to monitor application metrics, including request rates, response times, and error rates.
- Configured Prometheus to scrape data from Django instances, and Grafana dashboards to visualize key performance indicators (KPIs) for easy tracking.
- This setup allowed for real-time insights into application performance, making it easier to identify and respond to issues as they arise.

Exercise 5: Log Analysis using ELK Stack

- Implemented Django's built-in logging framework to capture logs and integrated ELK Stack (Elasticsearch, Logstash, Kibana) to analyze log data.
- Configured Logstash to parse logs from the Django application and store them in Elasticsearch, with Kibana providing a user-friendly interface for visualization and analysis.
- Developed dashboards in Kibana to detect patterns, errors, and performance issues, allowing for detailed log analysis that could identify trends over time.

Findings

Observability enhances monitoring by providing insights into *why* an issue is happening, rather than merely identifying that it exists. Observability in backend systems involves three key pillars:

1. **Logs**: Detailed records of events that can reveal common error patterns, response latency, and user interactions.
2. **Metrics**: Quantifiable data points like CPU usage, memory consumption, and request counts, enabling trend analysis.

3. **Traces**: Contextualized information about requests as they move through the system, helping identify slow components in the request path.

In this setup, observability allowed us to dive deeper into the data and uncover patterns, such as repeated validation errors or specific endpoints causing database bottlenecks.

Through monitoring and log analysis, several patterns emerged:

1. **High Database Latency**: Logs indicated certain views or endpoints with slow response times due to inefficient database queries. This information suggests potential for optimization by improving database indexes or query design.
2. **Frequent Validation Errors**: Analysis of logs showed repeated validation errors, indicating possible user input issues. This suggests the need for enhanced validation on the frontend, reducing unnecessary backend processing.
3. **Peak Traffic Times**: By analyzing traffic patterns in Prometheus and Kibana, we observed peak usage times, allowing us to plan for scaling resources or adjusting caching to handle higher loads.
4. **Error Trends**: By aggregating error logs, we could detect common exceptions, such as KeyError or IntegrityError, pointing to areas in the code that may require additional error handling or input validation.

Conclusion

In these exercises, we explored key aspects of building, monitoring, and analyzing a scalable Django application. From understanding consistency models to implementing monitoring tools, each exercise highlighted the importance of performance, reliability, and observability in backend systems. By setting up Prometheus and

Grafana, we enabled real-time tracking of application metrics, while the ELK Stack provided powerful log analysis capabilities for identifying trends and troubleshooting issues. This approach emphasized the significance of both monitoring and observability, which together enable a deep understanding of system behavior, facilitate quick response to issues, and support informed scaling decisions. These tools and practices help ensure a robust backend that can handle user demands effectively and maintain a high level of availability.