International Journal of Latest Engineering and Management Research (IJLEMR)
ISSN: 2455-4847
www.ijlemr.com || Volume 09 – Issue 02 || February 2024 || PP. 43-50

Modern Approaches to Scaling High-Load Backend Systems

Mykyta Machekhin
Senior Software Engineer, Curbside Technologies Inc.
US, New York City

Abstract: In the era of digitalization and exponential data growth, the efficiency and scalability of backend systems are becoming critically important. With the growing volume of user data, transactions and interactions, traditional approaches to the architecture and infrastructure of the backend are often unable to cope with the load, which leads to a decrease in performance and availability of services.

This article is devoted to an overview of modern methods and strategies for scaling high-load backend systems. The main focus is on horizontal and vertical scaling, the use of microservice architecture, containerization and orchestration using tools like Docker and Kubernetes. The issues of database optimization, autoscaling and monitoring, as well as the role of continuous integration and deployment (CI/CD) in maintaining the efficiency and stability of highly loaded systems are also discussed.

In the context of scaling highly loaded systems, it is important not only to apply new technologies, but also to rethink approaches to data development and management. The key aspect is the transition from monolithic architecture to microservices, which allows for more flexible resource allocation and simplifies the scaling process. Containerization and container orchestration greatly simplify application deployment and management, making it easy to scale systems to meet changing needs. Automation of CI/CD and monitoring processes helps to maintain continuous performance and optimize system resources.

The purpose of the work is to consider the possibilities of modern approaches to scaling high-load backend systems. The methodological basis was the scientific works of domestic and foreign authors.

Keywords: backend systems, scaling of high-load backend systems, modern technologies, digitalization, IT.

Introduction

The amount of data processed by web resources is constantly expanding, especially in the field of high-load projects. This necessitates the use of innovative approaches to optimizing and scaling databases in order to ensure stable functionality and prompt access to information.

Database scaling is the process of increasing their capacity and efficiency in order to meet the requirements of high-load web projects. Various approaches are used for this, including vertical and horizontal scaling. Vertical scaling involves increasing server performance, installing more powerful processors, or increasing the amount of RAM. In turn, horizontal scaling provides for the addition of new servers to evenly distribute the load.

There are a number of advanced technologies and platforms that facilitate efficient database scaling. One of these platforms is Apache Cassandra, a distributed data management system capable of efficiently processing vast amounts of information and ensuring reliability in operation. Another popular database scaling platform is MongoDB, which is a NoSQL solution that guarantees horizontal scaling and storage of data in the format of JSON-like documents.

Database optimization also plays a key role in the context of scaling high-load web projects. A variety of tools and techniques are used to achieve this, including indexing, data fragmentation, and caching systems. These approaches help to speed up data access and reduce the load on servers.

Scaling the backend is a key aspect that controls everything that happens on the backend of the application. This component is divided into several subsystems, each of which is responsible for performing certain tasks.

Horizontal scaling, in turn, involves adding multiple servers to an existing one. However, this is not always justified. Thus, it is important to determine when it is really necessary to expand the server space and divide the backend components into them.

If the data actions are applicable to the entire backend, then the separation becomes redundant. If these actions are diverse in nature, you can consider allocating independent components to separate servers based on the increasing load on these actions. However, each approach has its pros and cons, and the choice between them depends on the specific needs of the project.

Splitting the backend into multiple servers can be an expensive undertaking, raising questions about the interaction between the separated components. In the case of using a single server, another task arises — efficient data processing, determining what to cache, what to store and extract from memory, and what to leave unchanged [1].

1. Client-server architecture

The classic client-server model is the simplest example of a distributed system. The server is a synchronization point, it allows multiple clients to do something together in a coordinated manner.

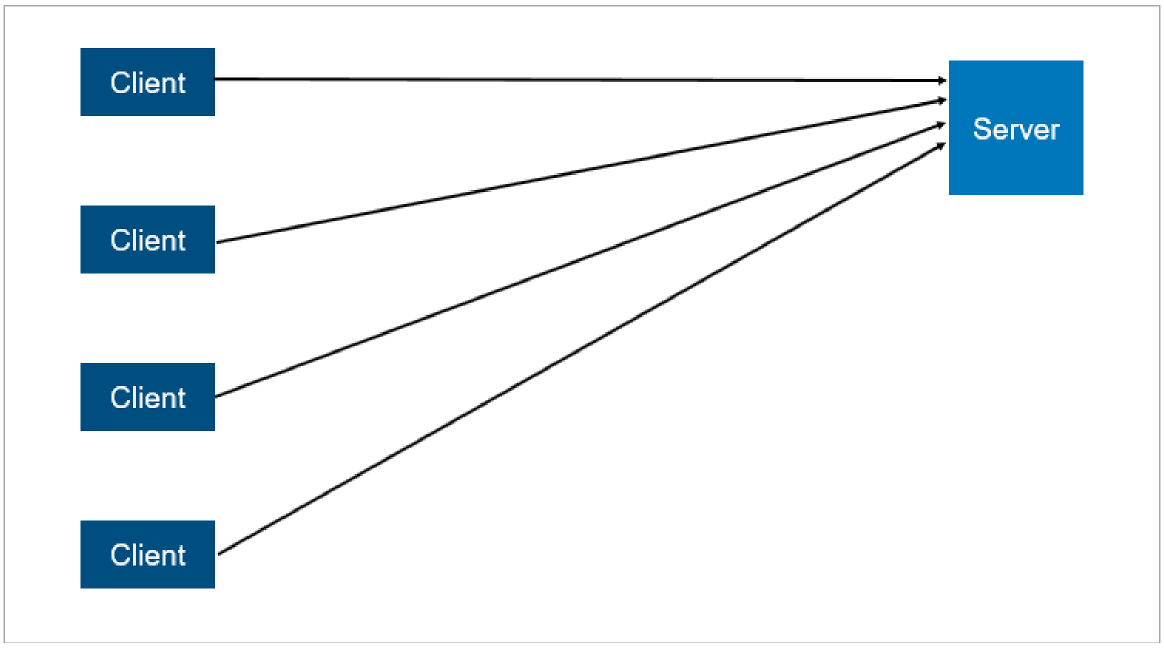


Fig. 1. Client-server interaction diagram

To avoid this, people came up with a master-slave connection (which is now politically correctly called leader-follower). The bottom line is that there are two servers, all clients communicate with the main one, and all data is simply replicated to the second one.

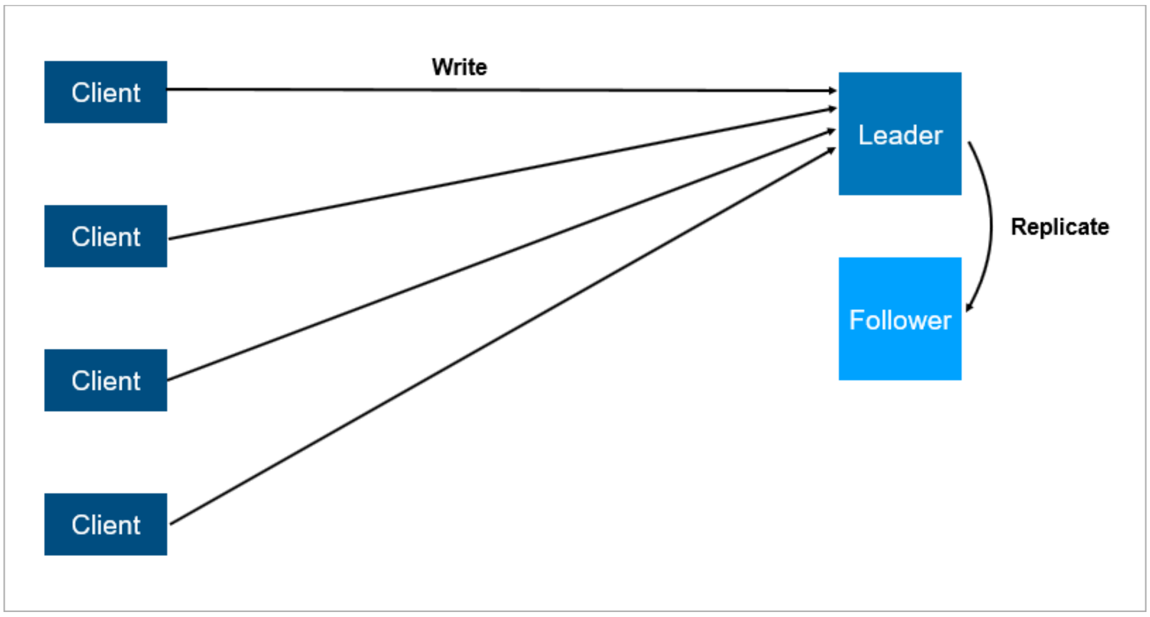


Fig. 2. Client-server architecture with data replication on follower

It is clear that such a system provides high reliability: in case of failure of the main server, a copy of all data is located on the follower, which allows you to quickly restore the system.

It is important to consider the replication mechanism. With synchronous replication, transactions are saved simultaneously on both the leader and the follower, which can slow down the process. Asynchronous replication, in turn, can lead to data loss after a failover.

When the leader fails at night, when everyone is asleep, the follower has data, but he does not know that he is now the leader, and clients do not connect to him. You can provide the follower with logic that begins to consider him the main one when the connection with the leader is lost. However, this can lead to a split brain conflict when both servers consider themselves to be in charge. Such situations are found, for example, in RabbitMQ, a popular queue technology.

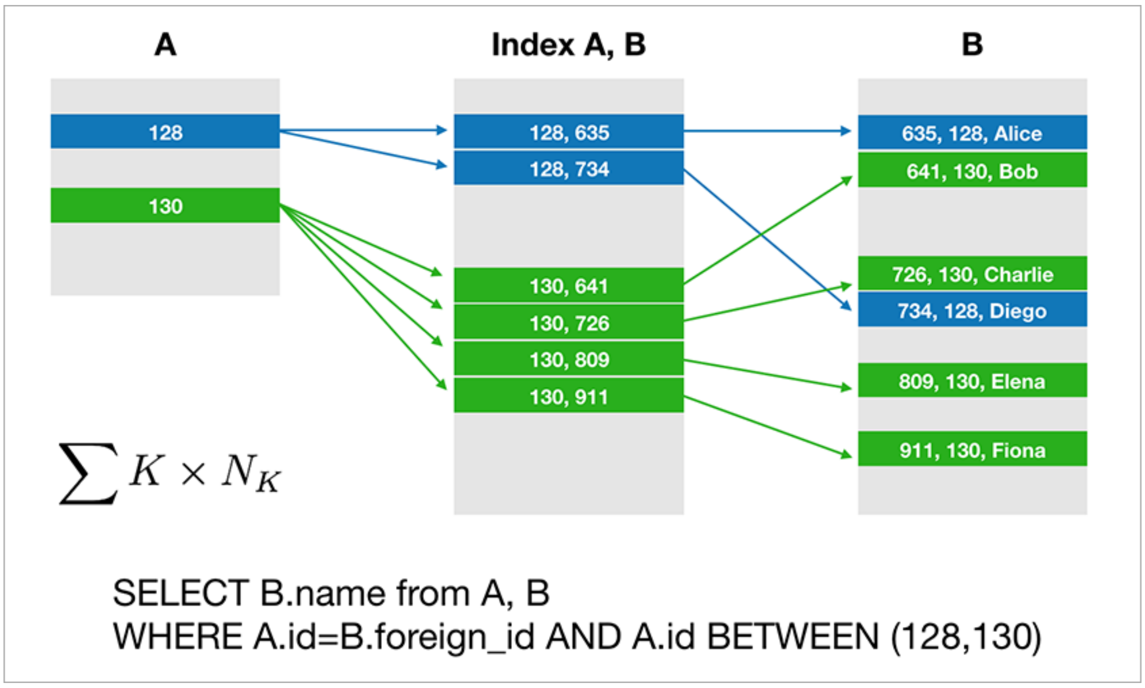
To solve these problems, auto failover is used — adding a third server (witness). It ensures that there is only one leader. If the leader refuses, the follower turns on automatically with a minimum timeout, which can be reduced to a few seconds. Clients should know the addresses of the leader and follower in advance, as well as implement the logic of automatic reconnection between them.

There is a main database, a backup database, there is a witness and yes — when sometimes we come in the morning and see that there was a switch at night. But this scheme also has drawbacks. Imagine that the user installs the service packages or updates the OS on the leader server.

To summarize, the redundancy should be equal to two. A redundancy equal to one is not enough. For this reason, in disk arrays, people began to use the RAID6 scheme instead of RAID5, experiencing the fall of two disks at once.

Scaling of relational databases

Most of the databases that developers are used to working with support relational algebra. Data is stored in tables and sometimes you need to connect data from different tables by using the JOIN operation. Let's look at an example of a database and a simple query to it.



```
SELECT B.name from A, B
WHERE A.id=B.foreign_id AND A.id BETWEEN (128,130)
```

Fig. 3. Scaling of relational databases

We assume that A.id — this is a primary key with a clustered index. Then the optimizer will build a plan that will most likely first select the necessary records from table A and then take from the appropriate index (A,B) the corresponding links to the records in table B. The execution time of this query grows logarithmically from the number of records in the tables.

Now imagine that the data is distributed across four cluster servers and you need to run the same query:

If the DBMS does not want to analyze all records in the cluster, the program may try to find records with A.id equal to 128, 129 or 130, and then find the corresponding entries from Table B. However, if A.id it is not a sharding key, the DBMS will not be able to determine in advance which server the data in table A. This forces you to access all servers to check for records A.id . Then each server can perform a JOIN, but this is not enough. For example, if an entry on node 2 is needed in the selection, but there is no entry with A.id=128, nodes 1 and 2 will perform JOIN independently, which will lead to an incomplete query result.

Thus, in order to fulfill this request, each server must contact each other. The execution time increases quadratically depending on the number of servers. (Good luck if you manage to shard all tables with one key, and then traversing all servers is not required. However, in practice this is unrealistic — there will always be queries where sampling is not required by the sharding key.) [2].

2. Replication Data

Replication Data replication is a key database scaling method for high-load websites. It is possible by creating multiple copies of the database, which are easily hosted on different servers or platforms.

There are several innovative solutions and platforms for data replication. Some of them provide optimization options and flexible replication settings, which ensures that the optimal strategy is selected in accordance with the unique requirements of each project.

Data replication provides an opportunity to improve performance by efficiently distributing the load across multiple servers. It also guarantees high data availability: in case of a failure on one server, other copies of the database can be used promptly.

When implementing data replication, it is extremely important to take into account the specifics of each database and choose a replication strategy that meets specific needs. For example, asynchronous replication allows you to write data locally on a server and then asynchronously transfer it to other servers. Synchronous replication, in turn, involves instant data recording to all servers, ensuring maximum reliability, but at the same time it can affect performance.

To effectively manage the data replication process, it is recommended to use a variety of mechanisms and tools, including automatic fault detection and recovery, as well as mechanisms for automatic data synchronization between servers.

Data replication is an important and advanced technology in the field of database scaling for high-load websites. This technology allows you to ensure high data availability and improve system performance.

3. Database sharding

Database sharding is an innovative solution for optimizing and scaling databases, ensuring efficient management and high performance of highly loaded websites. The basic concept of sharding is to divide data into small fragments, or shards, and distribute them across different servers or database clusters.

Advanced sharding strategies allow you to evenly distribute the load and ensure reliable data storage, while ensuring resistance to possible failures. This technique ensures efficient scaling of databases to handle high volume of requests, ensuring stable operation of the system.

Using sharding, each shard contains only a part of the data, which allows you to evenly distribute the load and ensure prompt access to data. The choice of criteria for dividing data into shards may depend on various factors, such as the geographical location of users or the type of data.

The possibilities of implementing sharding include horizontal and vertical approaches. Horizontal sharding involves splitting data into rows, where each shard contains the same number of records. In turn, vertical sharding divides data into columns, where each shard contains specific columns.

Special load balancing and data synchronization algorithms are used to ensure data integrity and control the uniform distribution of requests between shards. In addition, data backup and recovery strategies are being developed to ensure the reliability and security of information.

As a result, database sharding is an effective tool for optimizing and scaling highly loaded websites. Its use ensures efficient data management, guarantees fast access and high availability. With proper implementation and configuration, database sharding can significantly improve the performance and efficiency of web applications.

4. Advanced approaches to database optimization:

- Vertical and horizontal scaling
- Using indexes
- Caching techniques
- Transaction Management
- Selecting the appropriate data model
- Monitoring and performance analysis

In general, database optimization is an integral part of scaling and ensuring high performance of online platforms. The choice of suitable solutions and advanced approaches allows you to optimize the operation of databases and increase the efficiency of the entire platform [3].

5. Containerization of the Application

Containerization, also known as container virtualization, is an innovative approach to virtualization in which a single "user space" in the operating system kernel is divided into several independent logical partitions, known as containers or zones. A separate application can be run in each container, completely isolated from the rest of the system.

The container acts as a standalone executable software, where the binary code of the application is adjacent to all the necessary components for its operation, including configuration files and the runtime environment. Each application in the container has its own private network and virtual file system, completely isolated from other containers and the host system.

Although application containers resemble virtual machines (VMs) in functionality and purpose, which operate on the basis of hardware virtualization, it is important to understand that "containerization" and "hardware virtualization" are different virtualization technologies that solve similar tasks using different methods.

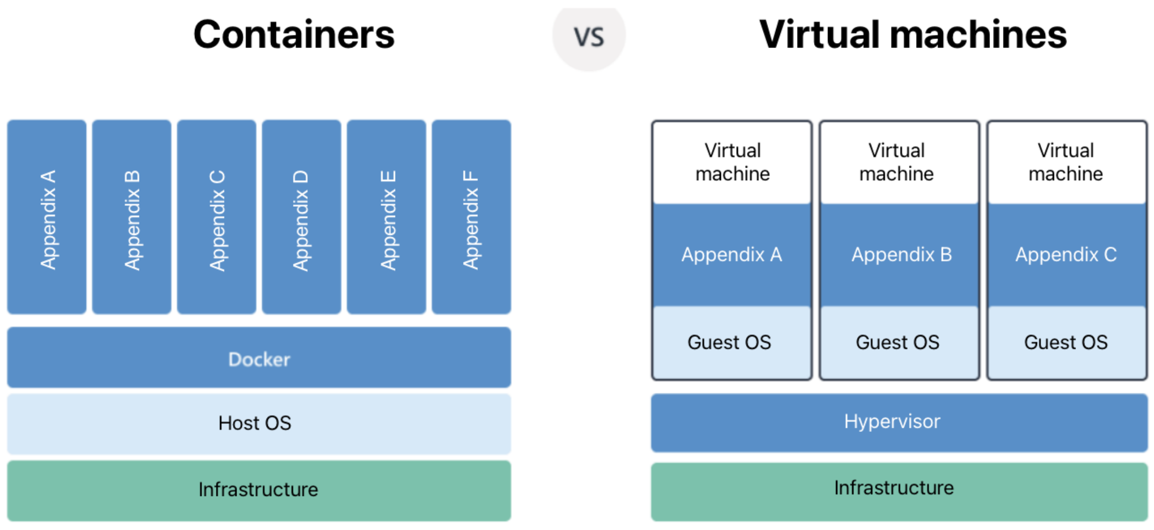


Fig. 4. Comparison of containers with virtual machines

A virtual machine is a full-fledged operating system embedded inside another OS, with its own core and isolated resources. On the other hand, the container is not an autonomous "computer", but only a dedicated mechanism for running a separate application.

Unlike hardware virtualization, containerization provides separation of resources at the operating system level, making containers more lightweight, less demanding and completely dependent on the "parent" OS.

Containers play an important role in ensuring code portability by eliminating possible inconsistencies between the local development environment and the production environment of the application. Packaging the code in a container allows you to isolate it from the underlying infrastructure, ensuring stable operation in various environments.

The benefits of containerization include a key contribution to the continuous deployment and reduction of the product innovation lifecycle. By breaking monolithic architectures into flexible container microservices, companies reduce the time to bring a product to market to days, despite earlier monthly deadlines.

Container technology simplifies the configuration of applications, making them versatile and ready to work in various environments. It is also effective in improving development productivity by enabling local development and deployment of applications, followed by instant deployment in test and production environments [4].

6. General characteristics of Docker and Kubernetes

It is an extensive set of tools for developers that provides the ability to create, publish, run and effectively manage container applications.

Docker Build creates a basic container image that contains all the necessary components for the successful launch of the application. This image includes code, binaries, scripts, dependencies, configurations, environment variables, and other key elements. Docker Compose is often used to manage multi-container applications, integrated with code repositories (for example, GitHub) and continuous integration (CI) and continuous deployment (CD) tools such as Jenkins.

Docker Hub, which is a Docker registry, is used to search and access container images, similar to the functionality of GitHub. Users can use Docker Hub for limited (developer-only) or shared access to containers.

Docker Engine, a container runtime environment, operates efficiently on various platforms, including Mac and Windows computers, Linux and Windows servers, cloud solutions, and edge devices. The Docker Engine, which is a wrapper for the containerd engine (an open source DNCF project), provides container execution.

Docker Swarm, an embedded tool, manages a cluster of Docker modules or a "swarm". Usually this cluster is created on various nodes and intersects with Kubernetes functionality.

In turn, Kubernetes (K8s) is an open container orchestration platform that automates the deployment and scaling of applications on different hosts. K8s allows you to manage entire container clusters, ensuring the scheduled launch of containers (Docker, containerd and CRI-O) in accordance with the computing resources and needs of each container.

K8s is chosen by many companies (88%) using containerization in production. Developed by Google, Kubernetes is supported by many cloud service providers (AWS, Azure, GCP), and is also included in popular distributions such as Red Hat OpenShift, VMware Tanzu and others. This broad support avoids dependence on specific vendors, providing more freedom for DevOps engineers to focus on developing their own products.

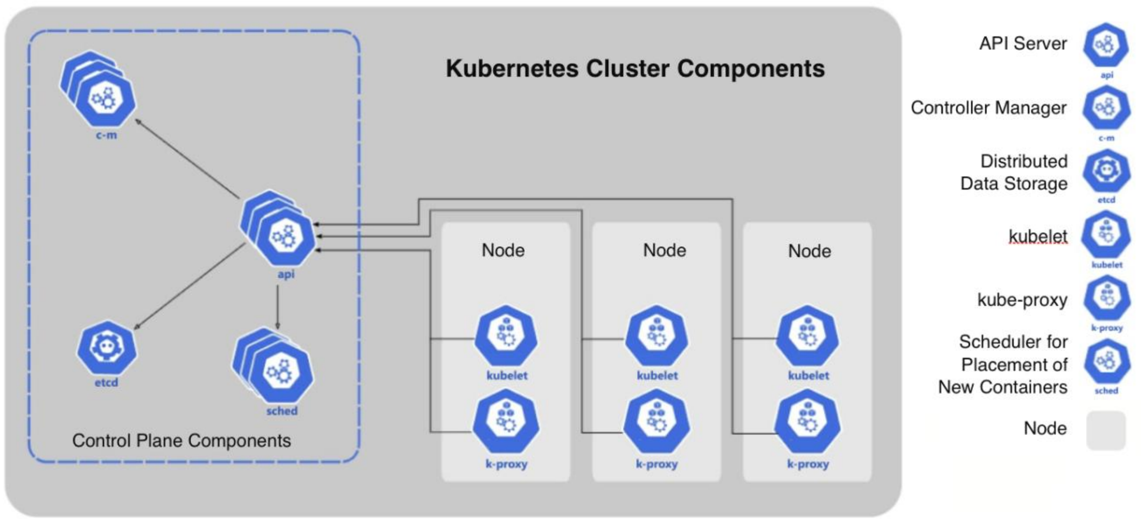


Fig. 5. Kubernetes operating principle

Containers within Kubernetes are combined into abstract entities called pods. Pods are groups of one or more base units ready to be deployed on nodes (physical or virtual machines).

Three key tools are used to effectively configure and manage the hearths: kubectl, kubelet and kubeadm. In addition to the basic kubelet management agent, each node is also provided with a kube-proxy network proxy, which forms the necessary network rules.

Hearth lifecycle management provides flexibility in scaling and maintaining the required state. This provides a high level of control over the performance and reliability of deployed applications [5].

Kubernetes Application:

- Managing containers on multiple hosts at the same time.
- Efficient use of equipment resources.
- Automatic deployment and updating of applications.
- Adding and connecting storage for applications with tracking their status.
- Scaling container applications and their resources dynamically.
- Declarative service management for full control over application deployment.
- Automated health monitoring and recovery of applications using autorun, autocorrect, auto-replication and autoscaling.

Interaction between Kubernetes and Docker: Despite the fact that Kubernetes used Docker as the default container runtime environment for a long time, Docker was not designed specifically to work inside Kubernetes. To provide more flexibility and reduce dependence on Docker, a proprietary API called Container Runtime

Interface (CRI) has been implemented. This interface allows you to choose the container runtime environment, making Kubernetes more flexible. Despite the lack of Docker support for CRI, an adapter called Dockerhim was introduced in Kubernetes, which translates CRI commands into the Docker language.

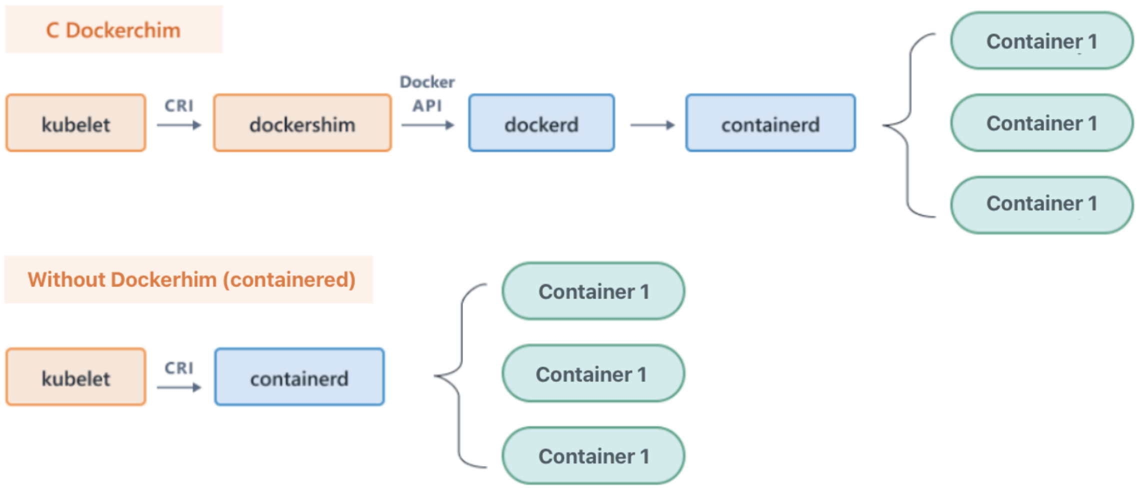


Fig. 6. Interaction of the kubectl control agent and containers

The main point of contact between Kubernetes and Docker is the orchestration of container clusters. Although Docker Swarm is also an orchestration tool, Kubernetes has become the de facto standard due to its flexibility and scalability.

Container orchestration, whether Docker Swarm or Kubernetes, faces common problems. A modern application can consist of many container microservices hosted on multiple host machines called nodes. These nodes are combined into a cluster to ensure smooth operation.

The interaction scheme of containers and nodes implies the presence of special tools for coordinating such a distributed system. Container orchestration systems can be compared to a conductor skillfully directing the instruments of an orchestra to perform complex symphonic works [6].

7. The difference between Kubernetes and Docker Swarm

Both Docker Swarm and Kubernetes are production—level container orchestration platforms, but each has its own strengths.

The built-in Docker Swarm tool (Docker Swarm mode) is the easiest to deploy and manage orchestrator. This is an excellent choice for companies just starting to use containers in production.

Swarm effectively covers 80% of all possible container orchestration scenarios, while its set of tools is much easier to master compared to Kubernetes, by about 5 times.

The easy integration of Docker Swarm with the rest of the Docker ecosystem tools such as Docker Compose and Docker CLI provides a familiar user interface with a smooth learning curve. In the container world, Docker Swarm is considered safer and easier to troubleshoot than Kubernetes.

Advantages of Docker Swarm:

- Simple installation using the same command-line interface as Docker provides convenient configuration.
- Full compatibility with other Docker tools, which provides a unified user interface.
- High-speed application launch in a dynamic Docker Swarm environment.
- Well-documented functionality, constantly updated to provide up-to-date information.
- Ability to control versions of docker containers for effective change management.

Disadvantages of Docker Swarm:

- Dependence on a platform supported only by Linux operating systems.
- Lack of built-in storage, which makes Docker Swarm less convenient for connecting containers to storage.
- Limited monitoring tools that do not provide extended real-time information.

The main advantages of the Kubernetes orchestration platform are almost endless scalability, flexible configuration, and an extensive technological ecosystem that includes a variety of open source frameworks for monitoring, management, and security.

Disadvantages of Kubernetes:

- **Setup:** Using different settings for each operating system can complicate the setup process.
- **Migration:** Applications that are already clustered or stateless may encounter failures in the configuration of hearths when trying to migrate to Kubernetes, requiring configuration reworking.
- **Compatibility:** Kubernetes is incompatible with the Docker CLI and Compose tools, which can create difficulties in integrating with existing environments [7].

Conclusion

Based on the above, it can be concluded that modern approaches to scaling highly loaded backend systems are a complex and multifaceted process that requires in-depth knowledge and an integrated approach. With the constant growth of data volumes and performance requirements, developers and engineers are actively using technologies such as microservice architecture, containerization, automated scaling and cloud computing.

Successful implementation of scaling requires not only the selection of appropriate technologies, but also careful design of the system taking into account its future growth. Monitoring and management of resources is also an important aspect to ensure efficient use of infrastructure and rapid identification of possible problems.

References

[1]. Scaling of the application backend . [Electronic resource] Access mode: https://unetway.com/blog/masstabirovanie-bekenda-prilozenij?ysclid=lrwfncpdi5397792116.– (accessed 01/25/2024).

[2]. Database scaling in high-load systems. [Electronic resource] Access mode: https://h.amazingsoftworks.com/en/articles/440306 /.– (accessed 25.01.2024).

[3]. Scaling databases for high-load websites. [Electronic resource] Access mode: https://ibsystems.kz/backend-development/baz-dannyh-dlya-vysokonagruzhennyh-veb-sajtov-resheniya /.– (accessed 25.01.2024).

[4]. Docker and Kubernetes — what is the difference between containerization technologies. [Electronic resource] Access mode: https://eternalhost.net/blog/razrabotka/docker-kubernetes ?ysclid=lrwfo7ffrf138901719.– (accessed 01/25/2024).

[5]. Using Docker for microservices architecture. [Electronic resource] Access mode: https://appmaster.io/ru/blog/arkhitektura-mikroservisov-docker .– (accessed 01/25/2024).

[6]. Kubernetes and other orchestrators. [Electronic resource] Access mode: https://habr.com/ru/companies/kts/articles/591355 /.– (accessed 25.01.2024).

[7]. Evstratov V. V. Container orchestration by the example of Kubernetes // Young Scientist. 2020. No. 51 (341). pp. 11-13.