Electronics and Information Technologies, 2025, 31, 45–52
http://publications.lnu.edu.ua/collections/index.php/electronics/index

UDC 004.3, 004.4, 004.9

FEATURES OF DESIGNING SOFTWARE DISTRIBUTED SYSTEMS ARCHITECTURE

Ivan Rovetskii
Lviv State University of Life Safety,
35 Kleparivska St. UA-79007, Lviv, Ukraine

Rovetskii, I. (2025). Features of Designing Software Distributed Systems Architecture. *Electronics and Information Technologies*, 31, 45–52. https://doi.org/10.30970/eli.31.4

ABSTRACT

Background. A pressing problem in designing distributed software systems is that they must operate stably under high load conditions when thousands of users want to receive certain resources provided by system services. To ensure high availability and stability of highly loaded software services, they are deployed in managed multiprocessor distributed systems (clusters). These kinds of resources have a high value, so in practice, various cloud platforms (Google Cloud, Amazon, Azure, etc.) are most often used, which provide these resources, charging only for the time of direct use of them. Services must be able to fully utilize the provided resources during data processing, so they must be designed using special architectural solutions. Therefore, the purpose of this research is to theoretically investigate the architectural solution features when designing software services of distributed systems.

Materials and Methods. The paper presents a theoretical research design of features of distributed software systems architecture which is based on the analysis and comparison of facts obtained from scientific sources and the author's practical experience as well.

Results and Discussion. The article shows that Kubernetes is one of the main software solutions designed for deploying software applications in parallel distributed systems. It was established that microservice architecture is the optimal architectural solution for designing software services of distributed systems, given the specifics of deploying software systems under Kubernetes management.

Conclusion. A multithreaded design must be used for effective scaling distributed software system under high load. However, performance improvement occurs only if the parallel algorithm uses parallel rather than concurrent multithreading mode. The concurrent mode of thread operation is advisable to use only in the case of blocking operations when some threads are in a waiting mode. Synchronous architecture has a good performance, but there are limitations associated with blocking threads until the results of client requests are received. The asynchronous model allows for a larger number of requests than the synchronous one but requires a fully asynchronous API when working with external services, and is also more difficult to debug the service and fix errors.

Keywords: distributed systems, clusters, cloud platforms, microservices, microservice architecture, multithreading.

INTRODUCTION

The current problem is to ensure high availability and stability of distributed software systems under high load conditions [1-5]. This problem must be solved by certain architectural solutions already at the stage of designing the system architecture. A necessary condition for the stable operation of a software system under high load conditions is its deployment in some software-controlled parallel distributed system - a cluster [1-5]. Such systems consist of separate, independent of each other, microprocessor nodes [6] with their own RAM and operating system, connected by specialized high-speed

communication channels, which work as a single system. In this case, the microprocessors of each cluster node are, in most cases, multi-core Chip Multiprocessor (CMP). Although in the general case, the RAM in the cluster is distributed, the memory is common to all processors of one processor node of the cluster, which significantly improves the efficiency of data exchange between processes during parallel calculations within one node. It is clear that in order to ensure the coordinated operation of the entire cluster, special software tools are required that are installed on the cluster nodes and provide performance monitoring, problem diagnosis, and automation of resource management on all available cluster nodes. Cluster deployment and scaling are currently very often performed remotely with the help of cloud platform services [7-8]. (Amazon, Google, Azure)

Special software tools are required to ensure the coordinated operation of the entire cluster. These tools are installed on the cluster's main node and provide performance monitoring, problem diagnostics, and automated resource management for all accessible nodes in the cluster. Cluster creation and management are currently mostly conducted remotely using virtual machines on cloud platforms [7-8] (Amazon, Google, Azure).

However, deploying a software service in a cluster is not a sufficient condition to ensure its efficiency [1-5]. To fully utilize the resources of a parallel distributed system, it needs to be designed using multithreading principles [23-25]. The use of threads allows tasks to be executed concurrently. However, in practice, there are often cases of service performance degradation in multithreaded environments. This is primarily due to peculiarities in working with threads, including their creation, maintenance, and management. Therefore, this paper also examines and analyzes the key aspects of designing software service architectures that efficiently use threads for parallel computations, aiming to improve service performance and maximize resource utilization within the cluster

MATERIALS AND METHODS

The paper involves theoretical research on designing features of distributed software systems architecture, based on the analysis and comparison of facts obtained from scientific sources and the practical experience acquired by the author as well.

RESULTS AND DISCUSSION

Today, a primary tool for deployment, scaling, and management of software applications is the open-source software platform Kubernetes [9, 10] developed by Google. Kubernetes cluster deployments are mostly performed remotely on virtual clusters hosted on cloud platforms within their data centers (data centers). Since Kubernetes has become a standard for managing software applications in a cluster, let's look at the key architectural features of software services design related to using Kubernetes.

The most important feature of Kubernetes is that it can only manage containerized software applications where the application and everything needed for its operation (code, libraries, configuration files, and even the operating system) are packaged into a container [9, 10]. Currently, Docker-based containers are the most widely used platform for containerization [11, 12]. Docker containers utilize kernel-level virtualization, functioning as independent processes with their own address space. Unlike virtual machines, containers share a single operating system and its kernel while loading only the binaries and libraries required to run the application. This eliminates the need to install separate operating systems—multiple containers can operate within the same system. This approach saves overhead costs associated with virtual hardware emulation and launching full-fledged operating system instances, making containers lightweight, fast, and cost-effective. Everything inside the container is isolated from the outside world, enabling the simultaneous operation of multiple containers on a single system without concern that they could influence each other [11, 12].

Thus, to effectively scale a software application within a Kubernetes-managed cluster [9, 10], it must be designed as separate, loosely coupled software components that can be easily deployed in containers. This type of architecture is called microservices architecture [13-15], and its components are referred to as microservices. Splitting an application into individual microservices is typically based on specific aspects (models) of the domain. For instance, in the context of the financial domain, one microservice might handle user authentication and authorization, another might process payments, a third might manage documents, and so on. This approach is known as Domain-Driven Design (DDD) [13-15].

Each microservice should control access only to the data it is responsible for. No other microservices should have direct access to databases managed by other microservices such data should only be accessible through the API of the respective service. This ensures that each microservice controls the consistency and structure of the database. It manages and allows the use of the most appropriate database management system for its specific needs without being constrained by overall system requirements. For example, one microservice might use a normalized SQL database while another could benefit from a NoSQL database [13-15].

When implementing microservices architecture, interaction mechanisms between microservices must be carefully designed. The simplest and most general form of microservice communication is message-driven communication, where microservices exchange messages according to a specific protocol. This technique is called Message-Driven Architecture (MDA). In this case, one microservice sends a message (request), other processes it, and then sends a response message back. Here, the requesting service is referred to as the client and the responding one as the server. In high-load systems, communication may be event-driven instead, relying on distributed message queues. This communication approach is called Event-Driven Architecture (EDA) [19].

The rules governing the exchange of messages in the client-server model are encapsulated in the Representational State Transfer architectural style (REST API) [16-18]. The essence of the REST API architectural style is that the microservice that processes messages should provide a unified interface for identifying resources, should not cache data from previous requests, and each request should contain all the necessary information for its processing. Client-server interfaces (protocols) facilitate message transmission, with the most common protocols being HTTP(S), GRPC(S), and WebSockets. Resource data is exchanged in specified formats determined by the protocol. Popular formats include JSON, XML, HTML, YAML, PlainText, and FormData. Each request processing microservice follows a multi-layer architecture, where upper-layer components interact only with directly lower-layer components. A prominent example of this architecture is the Model-View-Controller (MVC) pattern [23]. The model defines the business logic corresponding to the domain-specific model. The view represents the layer that formats data for the user. The controller manages system interaction and modifies the model in response to incoming requests. In practice, microservices also have a persistence layer to ensure data consistency within the database and provide access to it.

Synchronous and asynchronous architecture of distributed software systems. Nowadays, REST API are commonly implemented in synchronous communication models where a client sends a request and receives a response within the same connection. In synchronous architectures, each client request is handled by allocating a dedicated thread, as is common with traditional web servers. This architecture works well for scaling services under varying loads by dynamically increasing or decreasing the number of threads in the service. However, thread creation [25] involves not only generating an object within the user space of a process but also invoking low-level operating system APIs and kernel-level instructions for allocating and initializing hardware resources, such as stack memory and task scheduling. Consequently, this process is not instantaneous, and its latency becomes more noticeable under high-load conditions. To mitigate the negative impact of dynamic thread creation during service operation, it is advisable to pre-create an optimal number of

threads upon program startup and reuse them during execution. Threads are added to a thread pool for efficient reuse, and the optimal number of threads can only be determined experimentally through performance testing.

In cases where computational tasks are resource-intensive, parallel algorithms can improve execution. Tasks are divided into independent subtasks processed concurrently, and their results are merged afterward. However, this approach works well only for additive tasks; tasks dependent on execution order may suffer degraded performance due to synchronization and blockage of threads.

Despite its benefits, synchronous REST API architecture still employs blocking sockets, limiting the maximum number of simultaneous requests to the number of available threads. For higher scalability asynchronous architecture is preferred. Here, a single thread, which is known as the EVENT LOOP, handles all client requests using non-blocking sockets (NIO). Request processing occurs asynchronously within a thread pool, where callback functions or pipelines execute upon receiving results from databases or external services. Using asynchronous APIs, clients are notified of response completion through additional channels like email or SMS.

Another microservices interaction pattern is the Publisher-Subscriber architecture [23], which operates using distributed data queues [20-22] (e.g., Kafka, RabbitMQ, JMS, Google Pub/Sub). Changes in the state of one microservice generate an event that is published to the queue in a predefined topic. Other subscribed microservices consume and process the events, enabling a fully asynchronous Event-Driven Architecture (EDA) [19]. Such systems can be used for ensuring data consistency across distributed databases (e.g., using the SAGA pattern) [24].

Deploying distributed software systems. While microservices architecture offers considerable advantages, challenges must also be addressed. Deployment and scaling processes must be carefully described for each microservice to enable load balancing within a Kubernetes cluster [10]. Continuous Integration/Continuous Delivery/Continuous Deployment (CI/CD pipelines) are defined based on organizational requirements and technical specifications. Popular tools include Jenkins pipelines [26] or event-based pipelines (e.g., GitHub and GitLab CI/CD) [27]. Complexity management is another concern, particularly with large numbers of microservices. Furthermore, security and data integrity require additional attention.

Monitoring and logging in distributed environments present unique challenges, necessitating centralized analysis and logging tools. Practical implementations often involve the ELK stack [28]: ElasticSearch, Logstash, and Kibana. Logstash collects and transforms data, ElasticSearch indexes and analyzes it while providing search capabilities, and Kibana visualizes the results.

Technologies for designing distributed software systems. Currently, either the JAX-RS specification, which is part of the Jakarta Enterprise Edition (Java EE) standard, or the Spring framework is used to create enterprise-level RESTful microservices.

JAX-RS (Java API for RESTful Web Services) is a Java specification that provides a standardized way to create RESTful APIs, which uses annotations to simplify the process. The @Path annotation is used to define REST endpoints. The @GET, @POST, and @PUT annotations are used to define the methods that will be used to process the request - reading a resource, adding a new resource, and updating a resource, respectively. The @Produces, @Consumes annotations are used to define the data format. However, JAX-RS is only a specification, so its use requires an implementation. The main implementations of JAX-RS that work with data via the HTTP protocol and are used in practice are the Jersey, RESTEasy, and Apache CXF frameworks.

The Spring Web framework, which is part of the Spring platform, should be highlighted separately. Spring Web provides its own set of annotations for creating RESTful APIs, for example, @RestController, @RequestMapping, while immediately providing mechanisms for their processing. Therefore, in this case, we get a full-fledged framework for creating

RESTful APIs, which also works based on the HTTP protocol but does not implement the JAX-RS standard.

Standard implementations of JAX-RS and Spring Web frameworks are built on classic web servers (Tomcat), which use a synchronous (blocking) architecture for processing requests. The synchronous architecture has its limitations associated with blocking threads during the processing of client requests, and therefore, all these limitations are inherited by the frameworks. Such architectures are very common in the financial or health insurance sectors, where data processing includes a lot of data validation, document signing, and report generation. All these data processing stages are usually performed sequentially and technically involve working with different databases and integrating with external services. Typical technical tasks that are often encountered in practice in this area:

a) The task involves executing a database query that requires reading or updating a large data set;
b) The task involves executing several database queries within a single transaction;
c) Data for calculations must be obtained from several different sources using external service calls, which are often also synchronous;

This type of task is well divided into several parts that can be executed in parallel in separate threads. After the data is received, it can be processed asynchronously using callback code. In practice, in the Java world, CompletableFuture is used for this purpose.

If data processing is long-term, even using parallel threads, then we can provide an API to check the execution status. In this case, the software client periodically polls the server about the operation execution status.

If the task is additive and all the necessary initial data is present, but we need to speed up their processing, then in this case we can also parallelize the task into independent parts, execute them in a parallel thread pool, and then combine the results. ForkJoin thread pool is effectively used for this purpose in Java. However, it should be emphasized once again that performance will improve only if the result does not depend on the order of execution of its parts. In this case, the main thread is blocked until we get the intermediate results necessary for the combination.

If, according to technical requirements, it is necessary to accept as many client requests as possible or to carry out continuous data transfer in real time, then in this case, in practice, an asynchronous (non-blocking) Restful API is used. Asynchronous approaches are very often used in online games, the Internet of Things (continuous data collection from various sensors), and the creation of video conferencing platforms. An asynchronous API can be built using the Spring Web Flux framework or using distributed data queues.

If we talk about the Spring Web Flux framework, then its work is built only on the basis of the non-blocking Netty server. A feature of the asynchronous architecture, which is built on a non-blocking server, is the use completely non-blocking API in all parts of the software microservice to avoid blocking the main thread. Otherwise, the work of such a service under some conditions may simply stop.

In the case of distributed data queues, microservices can be built on both blocking and non-blocking servers. In practice, ready-made solutions provided by leading cloud platform vendors, such as Google Pub/Sub, are usually used.

CONCLUSION

As a result of the study, it was found that a feature designing of distributed software systems is designing microservice architecture. It was analyzed that microservice architecture can be built based on REST architecture or event-driven architecture.

Synchronous (blocking) REST architecture shows good performance but has limitations associated with blocking threads until the results of client requests are received. Therefore, it is suitable for areas where there is no high load on services, in particular for the financial and health insurance sectors.

Asynchronous (non-blocking) REST architecture and event-driven architecture (EDA) are used in areas where continuous data transfer in real time is required (online games, Internet of Things, streaming video conferences). Event-driven architecture is also used in systems with microservice architecture in which the state of one microservice depends on the state of another. The asynchronous model allows for a larger number of requests than the synchronous model but requires a fully asynchronous API when working with external services and is also more complex when debugging the service and fixing errors.

The study also found that architecture of distributed parallel systems should be built using parallel threads. The use of threads makes it possible to increase the performance of software systems but only if a parallel not a concurrent multithreading mode is used by parallel algorithm. The concurrent mode threads operation is advisable to use only in case of blocking operations when some threads are in standby mode. In general, the number of active threads in parallel mode should coincide with the number of available processors (processor cores).

The results obtained can be used in the practical design of microservice architectures and microservice integration. Also, the results of the study can be used in further scientific research related to the use of software system protection mechanisms discussed in this article.

COMPLIANCE WITH ETHICAL STANDARDS

The author declare that he have no competing interests.

AUTHOR CONTRIBUTIONS

Conceptualization, [I.R]; investigation, [I.R.]; writing – original draft preparation, [I.R.]; writing – review and editing, [I.R.];

The author has read and agreed with the published version of the article.

REFERENCES

[1] Burns, B. (2025). Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Systems Using Kubernetes: O'Reilly. https://www.oreilly.com/library/view/designing-distributed-systems/9781098156343/

[2] Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Oprea, A., Stubblefield, A. (2020). Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems: O'reilly. https://www.oreilly.com/library/view/building-secure-and/9781492083115/

[3] Gorton, I. (2022). Foundations of Scalable Systems: Designing Distributed Architectures 1st Edition: O'Reilly. https://www.oreilly.com/library/view/foundations-of-scalable/9781098106058/

[4] Tanenbaum, A. S., Steen, M. (2006). Distributed Systems: Principles and Paradigms: Pearson. https://pdf-up.com/download/distributed-systems-principles-and-paradigms-andrew-s-tanenbaum-4951969

[5] Vitillo, R. (2021). Understanding Distributed Systems: What every developer should know about large distributed applications. https://cdn.bookey.app/files/pdf/book/en/understanding-distributed-systems.pdf

[6] Herlihy, M., Shavit, N. (2012). The Art of Programming Multiprocessor: Elsevier. https://www.educate.elsevier.com/book/details/9780123973375

[7] Sosinsky, B. (2011). Cloud Computing Bible: Wiley Publishing. DOI: 10.1002/9781118255674

[8] Buyya, R., Broberg, J., Goscinski, A.M. (2011). Cloud Computing: Principles and Paradigms: Wiley Publishing. DOI:10.1002/9780470940105

[9] Luksa, M. (2023). Kubernetes in Action: Manning. https://www.scribd.com/document/659803971/Kubernetes-in-Action-Second-Edition-MEAP-V15

[10] Boorshtein, M., Surovich, S. (2024). Kubernetes – An Enterprise Guide: Master containerized application deployments, integrate enterprise systems, and achieve scalability: Packt Publishing. https://www.packtpub.com/en-SK/product/kubernetes-an-enterprise-guide-9781835081754?srsltid=AfmBOor_Cra9iEVmhmtw_DirzU_mzs3KGN4Cfqff4tglkvTX6Zf9dHBk
[11] Nickoloff, J, Kuenzli, S. (2019). Docker in Action: Manning. https://www.manning.com/books/docker-in-action-second-edition
[12] Poulton, N. (2023). Docker Deep Dive: Packt Publishing. https://cdn.bookey.app/files/pdf/book/en/docker-deep-dive.pdf
[13] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems: O'Reilly. https://www.oreilly.com/library/view/building-microservices/9781491950340/
[14] Bruce, M., Pereira, P.A. (2019). Microservices in Action: Manning. https://www.oreilly.com/library/view/microservices-in-action/9781617294457/
[15] Fowler, S. J. (2016). Production-Ready Microservices: Building Standardized Systems Across Engineering an Organization: O'Reilly. https://www.oreilly.com/library/view/production-ready-microservices/9781491965962/
[16] Biehl, M. (2016). RESTful API Design: Createspace Independent Publishing Platform. https://books.google.com.ua/books?id=DYC3DwAAQBAJ&printsec=frontcover&hl=uk&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
[17] Varanasi, B., Bartkov, M. (2022). Spring REST. Building Java Microservices and Cloud Applications: Apress. https://www.oreilly.com/library/view/spring-rest-building/9781484274774/
[18] Amundsen, M. (2022). RESTful Web API Patterns and Practices Cookbook: Connecting and Orchestrating Microservices and Distributed Data: O'Reilly. https://www.oreilly.com/library/view/restful-web-api/9781098106737/
[19] Bellemare, A. (2023). Building an Event-Driven Data Mesh: Patterns for Designing & Building Event-Driven Architecture s: O'Reilly. https://www.oreilly.com/library/view/building-an-event-driven/9781098127596/
[20] Narkhede, N., Shapira, G., Palino, T. (2017). Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale: O'Reilly. https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/
[21] Videla, A., Williams, J.J.W. (2012). RabbitMQ in Action Distributed Messaging for Everyone: Manning. https://classpages.cselabs.umn.edu/Spring-2018/csci8980/Papers/PublishSubscribe/RabbitMQinAction.pdf
[22] Richards, M., Monson-Haefel, R., (2009). Chappell D. A. Java Message Service: O'Reilly. https://www.oreilly.com/library/view/java-message-service/9780596802264/
[23] Fowler, M. (2003). Patterns of Enterprise Application Architecture: Longman (Pearson Education). https://ptgmedia.pearsoncmg.com/images/9780321127426/samplepages/0321127420.pdf
[24] Joshi, U. (2023). Patterns of Distributed Systems: Addison-Wesley Professional. https://www.oreilly.com/library/view/patterns-of-distributed/9780138222246/
[25] Goetz, B. Bloch, J., Peierls, T., Bowbeer, J., Holmes, D., Lea, D. (2006). Java Concurrency in Practice: Longman (Pearson Education). https://ptgmedia.pearsoncmg.com/images/9780321349606/samplepages/9780321349606.pdf
[26] Leszko, R. (2017). Continuous Delivery with Docker and Jenkins: Packt Publishing. https://www.packtpub.com/en-us/product/continuous-delivery-with-docker-and-jenkins-3rd-edition-9781803237480

[27] Kaufmann, M., Bos, R., Vries, M. (2024). GitHub Actions in Action Continuous integration and delivery for DevOps: Manning. https://www.oreilly.com/library/view/github-actions-in/9781633437302/
[28] Chhajed, S. (2015). Learning ELK Stack: Packt Publishing https://www.packtpub.com/en-ch/product/learning-elk-stack-9781785887154

ОСОБЛИВОСТІ ПРОЕКТУВАННЯ АРХІТЕКТУРИ ПРОГРАМНИХ РОЗПОДІЛЕНИХ СИСТЕМ

Іван Ровецький
Львівський державний університет безпеки життєдіяльності,
вул. Клепарівська 35, 79007, Львів, Україна

АНОТАЦІЯ

Вступ. На даний час актуальною проблемою проектування розподілених програмних систем є те, що вони мають стабільно працювати в умовах високого навантаження. Для того, щоб забезпечити високу доступність та стійкість високонавантажених програмних сервісів, їх розгортають у керованих мультипроцесорних розподілених системах (кластерах)**.** Такі ресурси є високо вартісними, тому на практиці, найчастіше, використовують різноманітні хмарні платформи (Google Cloud, Amazon, Azure та ін.), які надають ці ресурси віддалено. Для того, щоб використовувати розподілені апаратні ресурси в повному обсязі, програмні сервіси необхідно проектувати, використовуючи спеціальні архітектурні рішення. Тому мета даного дослідження полягає у теоретичному аналізі архітектурних рішень та особливостей їхнього використання під час проектування програмних сервісів розподілених систем.

Матеріали та методи. У роботі виконано теоретичне дослідження особливостей проектування архітектури розподілених програмних систем, яке базується на аналізі та порівнянні фактів, отриманих з наукових джерел, а також у результаті отриманого практичного досвіду автора.

Результати. У роботі показано, що основним програмним рішенням для розгортання програмних систем у паралельних розподілених системах є платформа Kubernetes. У результаті роботи встановлено, що мікросервісна архітетура є оптимальним архітектурним рішенням для проектування програмних сервісів розподілених систем, з огляду на специфіку розгортання під керуванням Kubernetes.

Висновки. Для ефективного масштабування розподіленої програмної системи необхідно використовувати багатопотокове проектування. Однак, підвищення продуктивності відбувається тільки за умови, коли у алгоритмі використовується паралельний, а не конкуретний режим багатопоточності. Конкуретний режим роботи потоків доцільно використовувати тільки у випадку блокуючих операцій. Синхронна архітектура має обмеження, пов'язані з блокуванням потоків, доки не буде отримано результати клієнтських запитів. Асинхронна модель дає можливість прийняти більшу кількість запитів, однак вимагає повністю асинхронного API під час роботи із зовнішніми сервісами, а також є складнішою під час відлагодження сервісу та виправлення помилок.

Ключові слова: розподілені системи, кластери, хмарні платформи, мікросервіси, мікросервісна архітектура, багатопоточність.

Received / Одержано	Revised / Доопрацьовано	Accepted / Прийнято	Published / Опубліковано
22 May, 2025	22 September, 2025	26 September, 2025	31 October, 2025