

Article

Technical Support System for High Concurrent Power Trading Platforms Based on Microservice Load Balancing

Ping Shao ¹, Longda Huang ¹, Liguu Weng ^{2,*} and Ziheng Liu ^{2,3}

¹ China Electric Power Research Institute Co., Ltd., Nanjing 210000, China; huanglongda@epri.sgcc.com.cn (L.H.)

² Jiangsu Key Laboratory of Big Data Analysis Technology, Nanjing University of Information Science and Technology, Nanjing 210044, China

³ Department of Computer Science, University of Reading, Whiteknights, Reading RG6 6DH, UK

* Correspondence: 002311@nuist.edu.cn

Abstract: With the booming development of the electricity market, market factors such as electricity trading varieties are growing rapidly. The frequency of transactions has become increasingly real-time, and transaction clearing and settlement tasks have become more complex. The increasing demands for concurrent access and carrying capacity in trading systems have made it increasingly difficult for existing systems to support business. This article proposes a transaction support system for large-scale electricity trading market entities, which solves the problems of high concurrency access and massive access data calculation while ensuring system security through business isolation measures. The system uses microservices to treat various functional modules as independent service modules, thus making service segmentation and composition more flexible. By using read–write separation, caching mechanisms, and several data reliability assurance measures, data can be stored and accessed quickly and securely. The use of a three-layer load balancing module consisting of an OpenResty access entry layer, a gateway routing gateway layer, and a WebClient service inter-resource invocation layer can effectively improve the system’s ability to handle concurrent access.

Keywords: power trading; high concurrency; microservices; load balancing



Citation: Shao, P.; Huang, L.; Weng, L.; Liu, Z. Technical Support System for High Concurrent Power Trading Platforms Based on Microservice Load Balancing. *Processes* **2024**, *12*, 1270. <https://doi.org/10.3390/pr12061270>

Academic Editors: Hsin-Jang Shieh and Masoud Soroush

Received: 18 April 2024

Revised: 16 June 2024

Accepted: 17 June 2024

Published: 20 June 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Generally speaking, the electricity market technology support system (also known as the electricity market operation system) is an organic combination of computers, data networks and communication equipment, various technical standards, and application software that supports the operation of the electricity market. It is strictly based on the operation rules of the electricity market and efficiently realizes market registration, market forecasting, electricity trading, economic dispatch, security analysis, measurement and collection, and cost settlement through computer programs and network technology. A comprehensive information processing platform, which has full business functions such as information dissemination and market evaluation, can ensure the open, transparent, safe, and stable operation of the electricity market.

As the carrier of the power business operation, the technical support system of the power trading platform has experienced continuous development and challenges with the development of the power market [1]. In recent years, the technical support system of the electric power trading platform has faced increasing pressure due to the centralized declaration of market members’ transactions. With the gradual liberalization of low-voltage side users in the future market, low-voltage level users will directly participate in market transactions, and the future system will need to accommodate a large number of market members to concurrently access the carrying capacity. The existing platforms mainly support medium-term and long-term trading business. The continuous segmentation of the

trading business has put forward higher requirements for the platform to support multicyle and multivariety market operations; with the continuous opening up of electricity sales market and low-voltage electricity users, the number of accessing market members and data storage will grow exponentially. The platform needs to withstand large-scale complex business calculations and concurrent online access from tens of thousands of users, which requires it to support high concurrency access and have the ability to process massive amounts of information. In this context, the traditional power trading platform technology support system gradually shows performance bottlenecks.

The technical support system of the traditional power trading platform adopts J2EE [2], .NET, and other technical routes to develop and deploy the single architecture. The core of its design is that the business client is only responsible for the UI interface of the input and output, the middle application layer is responsible for the business logic operation, and the backend data layer is mainly responsible for data point storage and management. For some large transactional systems, the application layer mostly adopts distributed service technology [3], such as EJB, COM, and CORBA, among which EJB is the most successful.

However, the above traditional monomer constructions are only suitable for the technical support systems of small-sized and medium-sized power trading platforms, which make them not suitable for the current demands. At present, there are the following problems:

(1) The speed of business generation selection is slow. Due to the system architecture not fully decoupling the business, modifications to a single functional module will affect the functional application of the entire system, thus resulting in a partial upgrade and then affecting the passive upgrade of the entire system. This will seriously affect the rapidly changing business characteristics of the power trading market.

(2) The slow response speed of the access and transaction data declaration for high concurrency users [4] is mainly due to the fact that most systems do not adopt internet access architecture based on microapplication architecture. Additionally, heavy reliance on the computing power of the database results in bottlenecks in data access and writing.

(3) Under the traditional architecture, deploying multiple web container instances on web servers leads to complex resource management, thus resulting in the ineffective utilization of server resources and poor scalability after the single instance deployment of server web containers. The low performance of market trading and transaction settlement computing cannot meet the business needs of daily clearing and monthly settlement. This is mainly due to the lack of elastic expansion computing capabilities of cloud computing container technology (Docker container) in terms of architecture, which cannot provide precise resource allocation without decoupling of business [5].

To address the aforementioned issues, many researchers have proposed different solutions. Zimmermann [6] proposed a general principle, architecture design, and corresponding case explanation for system construction through cloud computing architecture and microservices. Jacopo et al. [7] analyzed the differences between microservices architecture and distributed architecture, thus detailing the advantages of using microservices architecture to develop systems compared to distributed architecture and also highlighting the challenges that microservices architecture faces in service management and load balancing. Jitendra et al. [8] proposed a dynamic weight load balancing strategy. This strategy dynamically changes the weight value of each server node by continuously monitoring its load information, and it allocates request tasks based on the size of the weight value, thus achieving the goal of load balancing for each service node. However, in cases of high concurrency, this strategy cannot adjust the weights of each server in a timely manner and cannot achieve load balancing well. Kunihiro et al. [9] proposed a scalable load balancing strategy. The central idea behind this strategy is that each node within the cluster reassigns its load by stealing IP addresses from other nodes. When the cluster load is unevenly distributed, lightweight servers will steal IP addresses from heavyweight servers to provide services instead of the latter, thus ensuring load balancing among all nodes in the cluster. However, this strategy does not offer an effective evaluation of the severity of the server load and fails to achieve true load balancing. In summary, the current integration of

microservices and load balancing algorithms still has some performance issues. On servers that traditionally employ microservice load balancing methods, there will be a multitude of monitoring programs within the microservices cluster. When concurrency is high, it will consume a significant amount of server resources, thereby failing to achieve the desired load balancing effect. However, the multilayer load balancing strategy of microservices is still most commonly utilized in traditional polling strategies, which have not fully considered or ignored the performance issues of backend servers.

Regarding the current issues, this project optimizes and expands the microservices architecture and multilayer load balancing strategy, and we design a storage scheme, which considers the speed and security of data storage. This article proposes an overall solution for the high concurrency access of the power trading platform technology support system based on the microservices framework. By studying key technologies such as microservices technology, load balancing, data storage access optimization, and containers, a power trading platform technology support system that can meet the processing needs of high concurrency, high frequency, and massive data is designed and proposed.

2. Overall Architecture

Compared to traditional system design, microservices architecture has advantages such as good scalability, low coupling, high fault tolerance, and convenient development [10]. It has great advantages for achieving timely support platforms for power transactions. Based on the microservices architecture, we mainly consider the following performance requirements for high concurrency and high frequency:

(1) Good openness: The system has a wide range of external interfaces, which can access external databases, power grid dispatch and control systems, medium- and long-term trading systems, and small-scale power grid trading databases, among others. Good openness can greatly expand the applicability of the system.

(2) Scalability: The system adopts a modular design based on microservices, which can develop new functions according to the new requirements of new businesses without leaving any impression on the original architecture and functions of the system. At the same time, the system has comprehensive module management and a call library, thereby making function expansion more convenient.

(3) Security: The system has comprehensive information security protection measures, including permission management, resource management, account management, operation management, etc., which can ensure the privacy of transaction users and the security of transaction data. At the same time, the system has a data encryption protection function with digital signatures at its core. Electronic seals, timestamps, and other methods are utilized to safeguard the storage and transmission of transaction data.

(4) High availability: Different business models adopt parameterized designs, which can be flexibly configured according to business requirements. The system adopts technologies such as clustering, a circuit breaker, and backup technology to enhance its availability. When the system is abnormal, it has the ability to self-recover and prevent downtime. The high concurrency access scenario consumes a lot of resources, and each transaction in the trading system involves the economic rights and interests of the market members. The system must avoid any single point of failure and ensure the high reliability of the entire process.

System Structure

As shown in Figure 1, the overall architecture of the power trading platform technical support system proposed in this article includes four parts:

(1) Access layer: Power trading customers can log in, register, query, and publish transaction declarations on the power trading platform through the client. During the peak hours of power trading, the high concurrency of user access and declaration release requires a high load bearing capacity of the client, so multilayer load balancing technology is applied to enhance the load bearing capacity of the client. In practical engineering,

in addition to being suitable for hardware-based F5, NGINX and LVS have also been added as software implementations for load balancing.

(2) Application layer: The application layer provides clear and direct frontend pages for login, registration, transaction declaration, and other functions of the access layer, thus achieving data communication and decoupling between frontend pages and backend microservices.

(3) Business layer: The business layer utilizes microservices to implement relevant business functions, complete transaction operations, generate business records and data, and transmit them to the data layer.

(4) Data layer: The data layer stores the data of the power trading platform, introduces database table partitioning and read–write separation technology, realizes the deployment of relational databases from centralized to distributed, and improves the read–write performance of relational databases. Distributed cache libraries improve I/O access efficiency while breaking through single-machine memory capacity bottlenecks. Distributed message queues provide serialization functionality for massive concurrent requests and buffering for asynchronous inbound operations.

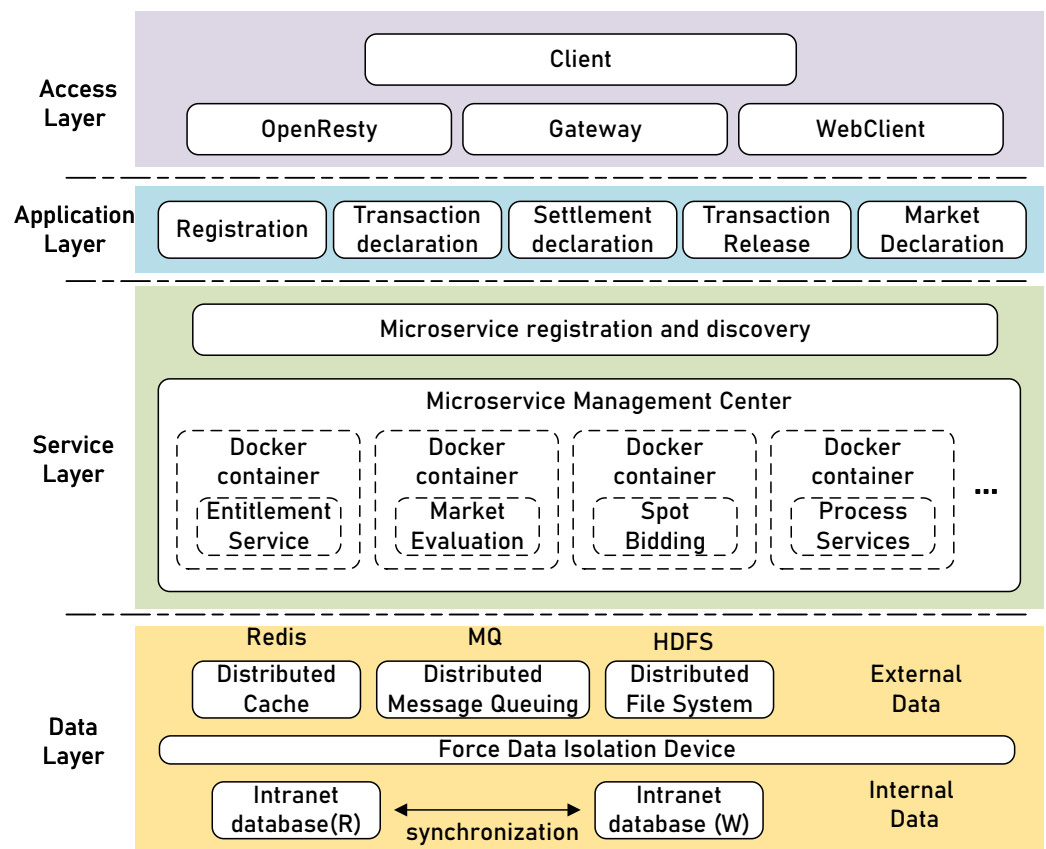


Figure 1. High concurrent access architecture of electricity market technical support system.

The technical optimization design of the four levels of the platform architecture provides the underlying technical support for the trading system to cope with more than ten thousand levels of concurrency.

3. Key Technology

Based on the overall scheme design, high concurrency access to transactions mainly includes technologies such as microservices, data storage access optimization, load balancing, containers, and public clouds. The main technical points will be explained separately below.

3.1. Microservices Architecture

Microservices embody the design concept of business logic as the main focus, thus treating each specific business function as an independent service and providing services externally through API. Essentially, microservices are a development of the SOA [11] design concept, thus inheriting the idea of componentization of functional modules while downplaying the concept of ESBs and treating each functional module as an independent service. This decentralized design concept makes the segmentation and composition of services more flexible and easier to respond to business changes and developments.

The microservices architecture abstracts independent logical units into a service, thus reducing the degree of coupling between the logical units and decoupling the system [12]. Furthermore, each project can be independently released and deployed, thus significantly enhancing the iteration speed of the project. In the microservices architecture, each independent service is perceived as a standalone system and maintained by a dedicated team so that changes to one service do not necessitate synchronous updates from other services. Microservices can be independently scaled. If a service encounters high load and performance bottlenecks, it only requires the allocation of additional resources for that service. By leveraging service registration and discovery technologies, smooth horizontal scaling can be achieved without the need to restart the service. Within the microservices architecture, each service is independently deployed, and services remain independent of one another. If a service encounters an exception, it will not cause the entire system to crash [13]. The structure is shown in Figure 2.

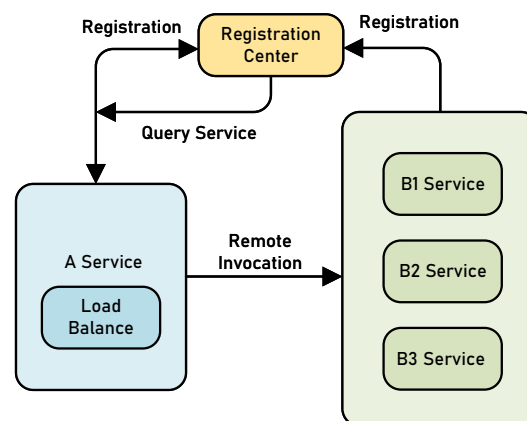


Figure 2. Microservices Architecture.

This article adopts (1) Eureka [14] to implement service registration discovery. Eureka has advantages such as client caching and fast host switching, thus resulting in higher availability and superior performance when registering and discovering services. It also adopts (2) Sleuth and Zipkin to implement distributed link tracking. Any high latency or error in any dependent service in each link of a microservice architecture system will cause the final failure of the request [15]. Sleuth provides link tracking for calls between services, including time analysis, error collection, and link optimization functions. Zipkin supports service data collection, storage, search, and visual display to address latency issues in microservice architectures. Sleuth and Zipkin are combined to achieve service request tracking and time delay statistics for each processing unit. (3) Hystrix achieves fault blocking. The methods for handling fault blocking include the following: The first is Degradation. They perform service degradation processing when request timeout, resource shortage, and other situations occur. The second is Isolation. Isolate requests using thread pools and semaphores ensure the integrity of the service invocation chain [16]. The third is Blown. When the proportion of abnormal requests reaches the threshold within a certain period of time, the fuse is activated. Whether downgraded or blown, once started, it will

stop calling the real service logic and quickly return the failure, which can ensure the integrity of the service chain.

3.2. Multilayer Load Balancing

In the microservices architecture, in order to prevent individual servers from crashing due to overload, it is necessary to make traffic distribution more uniform. This technology is called load balancing [17]. There are many techniques for achieving load balancing, including software implementation and hardware implementation. Depending on the application scenario, different load balancing techniques can be selected [18].

Figure 3 shows the overall architecture of optimized multilayer load balancing, which is mainly divided into three layers of load balancing modules: an OpenResty access entry layer, a Gateway routing gateway layer, and a WebClient service inter-resource invocation layer. All requests to access the architecture first go through OpenResty load balancing and then arrive at the Gateway routing gateway, where secondary load balancing is performed. In microservices, there are mutual calls between some services. Here, the asynchronous nonblocking component WebClient is used to perform three load balancing calls between services.

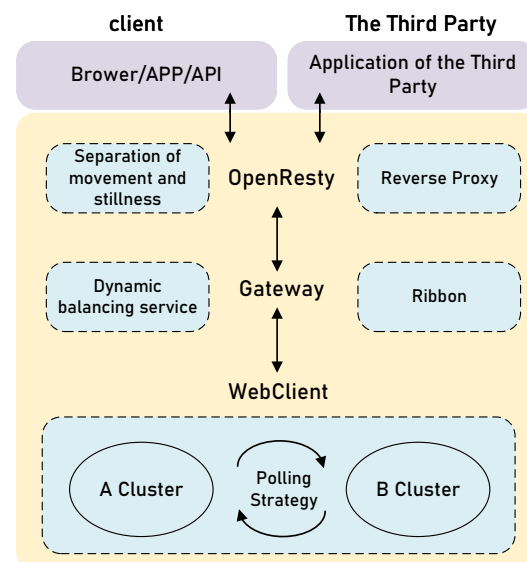


Figure 3. High concurrent access architecture of electricity market technical support system.

3.2.1. OpenResty

OpenResty is a high-performance web server that integrates Nginx [19] and LuaJIT. We can use Lua scripts to call various Nginx C modules and Lua libraries integrated internally to customize load balancing algorithms instead of the most commonly used polling load balancing algorithm by Nginx, thus increasing the load balancing ability of the access layer. The load balancing process of the access gateway layer is shown in the figure. The design concept steps are as follows:

- (1) Analyze various factors that affect the server's load capacity, and approximate the weight coefficients of each factor to obtain the weight value $K_i (i \in (1, 2, \dots, n))$. Calculate the load capacity value $X_i (i \in (1, 2, \dots, n))$. Calculate the load capacity value of each server proportionally to obtain the load balance weight value of each server.
- (2) Divide the intervals based on the weight values of each server obtained from step (1).
- (3) Randomly generate a random number that is not greater than the sum of the weight values of each server in the cluster.
- (4) Determine which service interval the random number is in step (2), and then select the server to provide services.

In the multilayer load balancing system architecture of microservices, the most commonly used strategy is the traditional polling load balancing strategy. This strategy simply assigns request tasks to each service node in turn, thus ensuring absolute task balance. However, this strategy cannot guarantee the rationality of task allocation. When assigning tasks to each service node, it does not reasonably consider the load capacity of each server. In actual production, there are many factors that affect the server's load capacity. In addition to being affected by factors such as the server's CPU performance, IO speed, memory size, etc., other objective factors such as network broadband and server connection numbers should also be considered. Due to the varying degrees of impact of different factors on the server load capacity, in order to reasonably quantify the impact of various factors on the server load capacity, it is necessary to construct an evaluation function for the server load capacity. This paper adopts a linear weighted evaluation model to evaluate the server load capacity [20].

In contrast to traditional polling strategies, this approach weighs various factors that influence a server's load balancing capabilities and determines the probability of randomly assigning tasks to a server based on the magnitude of the weight value, shown as in Figure 4. As OpenResty functions as the access entry layer of the architecture, it aggregates a multitude of requests. As the volume of access increases, the impact of this strategy will tend toward that of a weighted polling strategy, which facilitates task allocation based on weight values. This policy is stateless, and thus, it does not require maintaining the last selection state of the polling policy. It outperforms the polling policy in terms of performance.

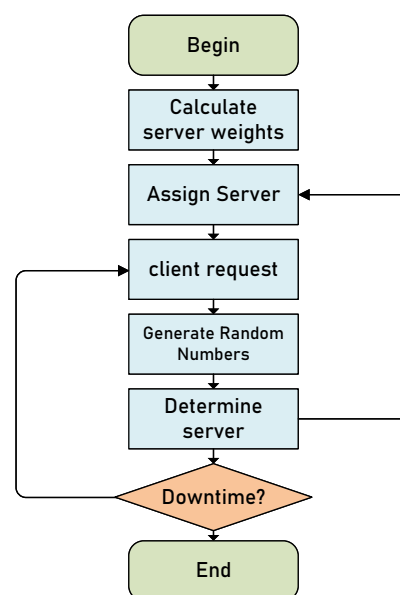


Figure 4. Flow chart of load balancing algorithm for access entry layer.

3.2.2. Gateway

When building a microservices system, each service is deployed in the form of a cluster. If you directly access the backend cluster system, there will be a large number of illegal requests flooding into the service cluster, thus posing significant security risks to the system. At the same time, the massive number of requests will also cause uneven load distribution in the microservices cluster [21]. Therefore, a service gateway is added between the client and the cluster, and through the routing, filtering, and load balancing of the service gateway, request tasks are evenly and securely allocated to the microservices cluster, thus enabling it to execute tasks reasonably and improving system availability.

At present, there are many popular microservice gateways on the market. The Gateway component is a new gateway based on the responsive WebFlux framework developed by Spring Cloud to replace Zuul gateways [22]. Compared to traditional nonresponsive

frameworks, the responsive framework WebFlux can utilize limited computer resources to improve system throughput. Therefore, in terms of performance, the Gateway gateway components are superior to other gateway components. Not only that, the Gateway component also provides rich routing and filtering functions, which can forward specific requests to the corresponding server cluster; To ensure the reliability of forwarding, the Gateway gateway component integrates Hystrix components to increase the system's fault tolerance. The load balancing function of the Gateway component is achieved by integrating the load balancing module Ribbon. Therefore, in this layer, the load balancing function is achieved by dynamically sending requests to the backend server cluster using the polling strategy in Ribbon, as shown in Figure 5.

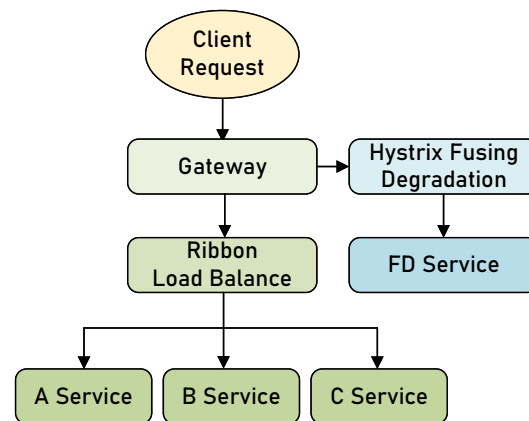


Figure 5. Microservice gateway load balancing.

3.2.3. WebClient

The load balancing component of the interservice resource call layer, WebClient, often uses traditional polling load balancing strategies [23], which allocate user requests in turn to internal server clusters for processing. This algorithm benefits from its simplicity in implementation and showcases the advantage of absolute load balancing. However, given the variances in the ability of each server to process requests, this strategy does not guarantee the rationality of task allocation. As such, a polling load balancing algorithm based on the server weight has been introduced.

The polling load balancing algorithm based on the server weight is based on traditional polling load balancing strategies. By first analyzing the load capacity of each service node in the cluster and then assigning corresponding load balancing weight values to each node, the number of times a node is scheduled is subsequently determined according to its weight value. If there are three service nodes a, b, c in the backend cluster, and their corresponding load balancing weight values are 3, 2, 1, then every six scheduling cycles is a scheduling cycle, and the scheduling order of each scheduling cycle is a, a, a, b, b, c. This scheduling order is not very friendly for the server cluster, because it can cause servers with larger weight values to experience instantaneous high-load situations. Therefore, it is necessary to introduce a new load balancing algorithm to address this issue in the WebClient layer when calling resources between services. This algorithm is optimized and improved on the basis of a weight-based polling load balancing algorithm. It still assigns tasks to each service node according to its load balancing weight value in each scheduling cycle. However, to ensure smooth task allocation as much as possible, it no longer continuously assigns tasks to service nodes according to the weight value; this ensures that the load will not be concentrated on the same server with higher load balancing weight values. The steps for the load balancing strategy for resource invocation between services are as follows: (1) Calculate the fixed weight value F_i of i th server node in the cluster. (2) Calculate the total fixed weight values for each server S . (3) Calculate the weight value W_i for service node i as $W_i = F_i + L_i$, where L_i is the temporary weight, which is initialized to 0. (4) Select

the service instance j with the highest weight value to provide services. (5) Modify the weight value of the service instance k to $W_k = W_k - S$. (6) Repeat steps 5 and 6.

3.3. Data Storage Access Optimization

Optimizing data storage access is a key step in improving the backend data access capability of trading systems [24]. The smooth and fast declaration of transaction data is achieved through caching and asynchronous writing mechanisms. The transaction information release query is mainly based on the internal database read–write separation strategy of the relational databases and the external network distributed caching strategy. The components are as follows:

(1) Read–Write Separation: Add a slave database on top of the master database, with data writing oriented toward the master database and reading oriented toward the slave database. The data are synchronized in real time from the master database to the slave database. One is the read/write access routing, which dynamically captures the type of data read/write request based on the reflection mechanism when configuring the master/slave data source data access, and it determines the connection to different read/write data sources. The second is data synchronization. On the premise of master–slave database network interoperability, the relational database products all provide supporting real-time data synchronization tools. The transaction system databases are deployed on the information intranet, and real-time data synchronization from the main database to the slave database of the transaction system can be achieved through data synchronization tool configuration.

(2) Cache: This article includes three caching modes. Firstly, preload caching is conducted. For a large amount of data that needs to be frequently accessed, it is necessary to preload it from the relational database to the distributed cache, such as the login permission information of the market members. Secondly, read misses are loaded into the cache. When unable to determine if the data are denied access, the system uses this mode for processing. If the data read from the cache misses, it will be read from the relational library and loaded into the cache so that it can be directly read from the cache in the future for viewing historical declaration data during the declaration period. Thirdly, the cache real-time reads and writes, and it asynchronously writes to the database through message queues. The data are cached for real-time read and write, and they are asynchronously written to the internal network relationship database through message queues and message logs. This pattern is applied to transaction declaration scenarios, thus deploying the message queue to an external network and providing an expandable buffer pool for short-term submitted declaration data requests by subscribing to topic messages on the message queue. There is a request serialization to quickly write data to the distributed cache while asynchronously and smoothly writing declared data to the internal network database. The serialization of massive concurrent requests based on message queues reduces the concurrency pressure on isolated devices and internal network databases.

(3) Data reliability assurance: Firstly, strong isolation devices, distributed caching, and message servers are deployed in a cluster architecture to ensure the security of internal and external network data transmission, data access, and message redundancy. Secondly, to avoid extreme caching and message anomalies, local small file logs are used to backup the declaration data and declaration time information, thus serving as reliable source data for recovering and storing declaration data anomalies. the system deletes local logs only when the declared data are stored normally. Thirdly, during the declaration process, real-time monitoring of the status of key resources, such as web servers, isolation devices, databases, and the writing of the declaration data, as well as alerting and the timely handling of abnormal situations, is required. Among them, isolation device status monitoring mainly monitors whether the isolation device status is normal by detecting the isolation device port through heartbeat and simulating sending simple SQL query requests. Fourthly, there is deploying a resident program on each web server to supplement and store data from log files that exceed a certain length of time. The resident program parses the declaration

time information of the log file to ensure that the data entry timing is correct, that is, the final declaration data entry takes effect. If the postdeclaration records are stored first, then when scanning the postdeclaration log file, there will be no storage overwriting, because the declaration time is earlier than the database data. In addition, different web servers can solve the problem of timing mismatch caused by inconsistent server time by configuring system clock synchronization. Through these four points, it can be ensured that the transaction declaration data can be properly processed as soon as possible in abnormal situations, ensuring that the data are not lost.

(4) Database table partitioning design: To cope with the massive data storage of a single table in a relational database, a distributed relational database provides technical support for subtable storage operations. When the amount of data in a single table increases to a certain scale, the time required for complex table queries or modification operations will sharply increase. This article uses mature tools such as distributed relational database middleware to achieve unified data storage and access for transaction system data after table partitioning.

3.4. Containerization Technology

Docker [25] is a mature container technology. After packaging the application as an image, it can be dynamically distributed and deployed through container orchestration tools, thus creating Docker images of different application microservices in the trading system and deploying them to a web server [26]. By starting the container, the trading application service can be started, thereby simplifying the service deployment and operation of the declaration and publishing application. Before massive market members log in to the system for centralized trading declaration, the server startup application containers can be divided on the machine cluster to enhance the performance of the declaration and publishing services. After the declaration is completed, the container can be stopped without affecting other services on the server. By utilizing container management service components, manual operation restrictions can be overcome through configuration policies, thus achieving automatic elastic scaling of container image resources.

3.5. Public Cloud

Container technology can solve the problem of dynamic expansion and the contraction of services, but server and network equipment resources still need to be managed by each power trading center. During nonpeak access periods, a large number of machine resources will be idle, thus resulting in extremely high machine and maintenance costs and low resource utilization.

If the peak concurrent scale of the trading system in the future reaches millions and tens of millions, we can deploy the system query function to the public cloud, utilize its massive machine resources, and solve the problem of dynamic hardware resources. During the peak concurrent period, we can temporarily apply for more resources on the cloud platform. These resources are released after the declaration is completed, thus enabling on-demand payment. This approach significantly enhances resource utilization and substantially reduces operation and maintenance costs.

4. Engineering Testing

On the basis of testing the performance of data access optimization, this article selects bilateral centralized power transaction declaration as the testing application scenario. Market members entering the transaction then perform multisegment power data declaration and concurrent testing on the published transaction information. The messaging middleware adopts RabbitMQ [27], the distributed cache adopts Redis [28], and the concurrent testing client uses the stress testing tool JMeter [29] for concurrent stress testing.

From Table 1, it can be seen that the read and write performance of this article generally improved by different magnitudes when compared to traditional modes. Among them, the performance advantage became more significant as the number of read performance

records decreased, and similarly, the advantage became more significant as the write performance data size increased. Currently, the spot day-ahead market unit trading requires the declaration of 48-period data. Based on the estimated number of units per market member, the read performance was improved by 6–12 times, and the write performance is improved by 5–16 times. Therefore, the data access optimization performance was significantly enhanced.

Table 1. Data access performance testing.

Data Volume	Traditional Reading/ms	Our Reading	Traditional Writing/ms	Our Writing/ms
1	103	4	19	4
10	104	5	25	4
100	121	9	64	7
1000	164	35	339	31
10,000	291	140	6051	189

4.1. Analysis of Multilayer Load Balancing Testing

This article optimizes and improves the multilayer traditional polling load balancing strategy commonly used in microservices architecture, designs an optimized multilayer load balancing architecture, and applies it to actual power trading support systems. We compare the response time and throughput of the improved multilayer load balancing strategy with the traditional multilayer load balancing polling strategy, and finally compare the experimental results of the two and analyze them.

This experiment used JMeter to simulate high-concurrency requests from users and set up a test group with the same number of concurrent requests. The throughput and average response time of the requests were used as performance testing indicators. The optimized multilayer load balancing strategy has been compared with the traditional multilayer polling load balancing strategy. The test results of the two comparison experiments are as follows.

The throughput data of the two algorithms under different levels of concurrent requests in this experiment are shown in Figure 6.

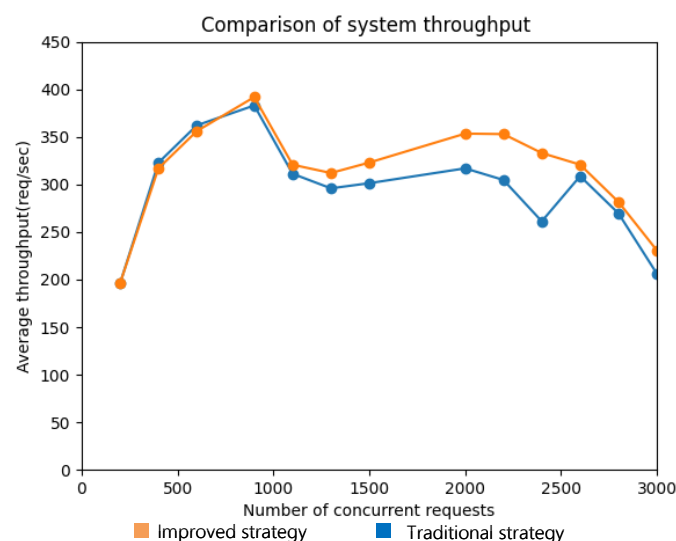


Figure 6. Comparison of system throughput.

From the above data analysis, it can be seen that when the concurrency was less than 900 req/s, the throughput of the two strategies remained basically the same. When the concurrency exceeded 900 req/s, as the number of concurrent requests gradually increased, the optimized multilayer load balancing strategy had higher throughput than

the traditional multilayer polling load balancing strategy. When the number of concurrent requests gradually increased from 800 to 1100 req/s, the throughput of both models reached its peak. Then, the throughput of both was found to gradually decrease. From the curve, it can be seen that the optimized model yielded better throughput and stability compared to traditional models.

This experiment tested the average response time of a single node and recorded the response time data for each group of concurrent requests. The experimental data for each group are shown in Figure 7.

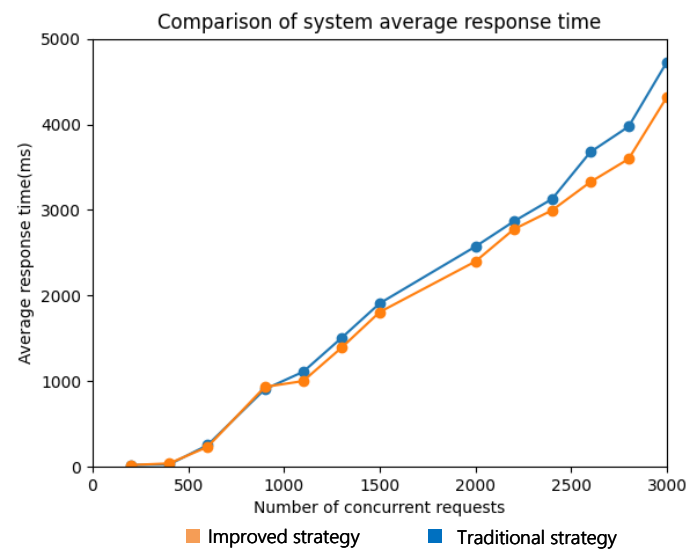


Figure 7. Comparison of system average response time.

From the above data analysis, it can be seen that when the system concurrency was less than 900 req/s, the average response time of the two multilayer load balancing strategy models remained basically the same. When the system concurrency exceeded 900 req/s, as the concurrency increased, the average response time of the two models gradually increased. However, the optimized multilayer load balancing architecture model always had a lower average response time than the traditional polling strategy's multilayer load balancing architecture model.

From the analysis of the data results from the above comparative experiments, it can be seen that when the concurrency was low, the resources of each server were relatively sufficient. The performance of the optimized multilayer load balancing architecture and the traditional multilayer polling load balancing architecture was not significantly different. Specifically, the throughput and average response time of the two were basically consistent. As the concurrency gradually increased, the throughput of both architectures generally increased to a peak and then gradually decreased, and the optimized multilayer load balancing architecture model had higher throughput than the traditional multilayer polling load balancing architecture. In terms of the average response time, the optimized multilayer load balancing architecture model was lower than the traditional multilayer polling load balancing architecture. During high concurrency, the optimized multilayer load balancing architecture exhibited a relatively smooth change in the throughput and average response time compared to the traditional multilayer polling load balancing architecture. Consequently, the optimized architecture demonstrated better and more stable performance. In summary, based on the performance of the two architectures at each stage, the optimized multilayer load balancing architecture has a better overall performance compared to the commonly used traditional multilayer polling load balancing architecture.

4.2. Concurrency Capability Testing

In order to verify the performance of the microserviceization, pressure tests were conducted on the original monomer structure scheme and the scheme after microservice transformation. The testing environment consisted of 10 servers, and in the microservice transformation plan, four servers were deployed as independent and parallel power trading module microservices. Tables 2 and 3 show the pressure testing results.

Table 2. Pressure testing to traditional single structure.

Quantity of Processes	Processing Quantity per Second	Average Response Time (/ms)	CPU Utilization
50	4913	4.24	23
80	7854	14.65	46
100	8426	41.36	59
160	8096	116.94	77

Table 3. Pressure testing to structure with microservices and load balancing.

Quantity of Processes	Processing Quantity per Second	Average Response Time (/ms)	CPU Utilization
50	4895	4.20	20
80	8246	13.54	38
100	9163	18.49	68
160	1247	34.58	74

Table 2 shows that as the number of pressure testing threads increased, the TPS (processing times per second) of the monomer structure scheme gradually reached saturation, and the CPU utilization also increased with the increase in the TPS. Table 3 shows that the implementation of microservice transformation did not significantly improve the effectiveness of adding one IO request, and even slightly decreased in the case of low concurrency. When the pressure test thread exceeded 70, the advantages of the microservices gradually emerged, with the TPS, average response time, and CPU utilization all being superior to the single solution. As the degree of concurrency increased, the advantages of the microservice solutions became more apparent. When the number of threads tested reached 150, the single architecture scheme gradually reached saturation, with a significant decrease in throughput. However, the throughput of the microservices architecture further increased, thus indicating that the microservice scheme is more suitable for high-concurrency business scenarios. From the perspective of stress testing, we can conclude that the larger the business volume, the more significant the performance of the microservices. For areas with the highest load density, microservice segmentation can optimize performance and achieve significant results. At the same time, after splitting out services, it is easier to cope with the actual needs of changing trading business and increasing trading varieties.

5. Conclusions

This paper delineated the drawbacks of the single-structured system, such as its poor scalability, challenging maintenance, and sluggish iteration speed. It elaborated on the design principles and overall technical solutions of the technical support system for the power trading platform, which was based on microservices. Additionally, it highlighted the key technologies involved in its application, including three-tier load balancing, data storage optimization, and public cloud integration. Through experiments, it was demonstrated that the implementation of microservice transformation could significantly enhance the concurrency performance of the system while also improving its robustness and the speed of version iteration.

Author Contributions: Conceptualization, L.H.; methodology, L.H. and P.S.; software, L.H.; validation, Z.L.; formal analysis, P.S.; investigation, P.S. and Z.L.; resources, L.H.; data curation, L.H.; writing—original draft preparation, L.H. and P.S.; writing—review and editing, Z.L.; visualization, P.S.; supervision, L.W.; project administration, L.W.; funding acquisition, L.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was financially supported by the Science and Technology Project of the SGCC, named the Research on Research on Key Technologies for High Concurrent Access and High Performance Processing of Massive Data in Electricity Trading Platform for Market Entities with a Scale of Ten Million (5108-202218280A-2-102-XG).

Data Availability Statement: The data and the code of this study are available from the corresponding author upon request.

Conflicts of Interest: Authors Longda Huang and Ping Shao were employed by the company China Electric Power Research Institute Co., Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

- Stafford, G.W. Electric Wholesale Power Sales at Market-Based Rates. *Energy LJ* **1991**, *12*, 291.
- Liu, F.; Shu, X.; Luo, Y. Design of Power Marketing System Based on JavaEE. In Proceedings of the 2017 7th International Conference on Education, Management, Computer and Society (EMCS 2017), Shenyang, China, 17–19 March 2017; Atlantis Press: Amsterdam, The Netherlands, 2017; pp. 678–682.
- Dai, Y.S.; Xie, M.; Poh, K.L.; Liu, G. A study of service reliability and availability for distributed systems. *Reliab. Eng. Syst. Saf.* **2003**, *79*, 103–112. [[CrossRef](#)]
- Huang, L.; Zhuang, W.; Zhang, H.; Pan, J.; Xu, P. Research on key technologies of electricity trading platform of next generation. In Proceedings of the 2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC), Chongqing, China, 24–26 May 2019; pp. 1797–1802.
- Yuan, J.H.; Yu, S.K.; Hu, Z.G. A multi-agent trading platform for electricity contract market. In Proceedings of the 2005 International Power Engineering Conference, Singapore, 29 November–2 December 2005; pp. 1024–1029.
- Zimmermann, O. Microservices tenets: Agile approach to service development and deployment. *Comput. Sci.-Res. Dev.* **2017**, *32*, 301–310. [[CrossRef](#)]
- Soldani, J.; Tamburri, D.A.; Van Den Heuvel, W.J. The pains and gains of microservices: A systematic grey literature review. *J. Syst. Softw.* **2018**, *146*, 215–232. [[CrossRef](#)]
- Bhatia, J.; Patel, T.; Trivedi, H.; Majmudar, V. Htv dynamic load balancing algorithm for virtual machine instances in cloud. In Proceedings of the 2012 International Symposium on Cloud and Services Computing, Mangalore, India, 17–18 December 2012; pp. 15–20.
- Toumura, K.; Nemoto, N. A Scalable Server Load Balancing Method Using IP Address Stealing. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops, Kyoto, Japan, 22–26 July 2013; pp. 599–603.
- Cheng, L.; Wang, H.; Gao, C. Research on application of micro service in power transaction system. *Power Syst. Technol.* **2018**, *42*, 441–446.
- Bean, J. *SOA and Web Services Interface Design: Principles, Techniques, and Standards*; Morgan Kaufmann: Burlington, MA, USA, 2009.
- Hammad, H.; Sahmoud, T.; Ghazala, A.A.R.A. Convert Monolithic Application to Microservice Application. *arXiv* **2023**, arXiv:2306.08851.
- Fritzsche, J.; Bogner, J.; Zimmermann, A.; Wagner, S. From monolith to microservices: A classification of refactoring approaches. In Proceedings of the Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, 5–6 March 2018; Revised Selected Papers 1; Springer: Cham, Switzerland, 2019; pp. 128–141.
- Carr, K.; Kendal, R.L.; Flynn, E.G. Eureka!: What is innovation, how does it develop, and who does it? *Child Dev.* **2016**, *87*, 1505–1519. [[CrossRef](#)] [[PubMed](#)]
- Rahman, R.; Alanyali, M.; Saligrama, V. Distributed tracking in multihop sensor networks with communication delays. *IEEE Trans. Signal Process.* **2007**, *55*, 4656–4668. [[CrossRef](#)]
- Li, R.; Li, Y.; Li, W. An Integrated Load-balancing Scheduling Algorithm for Nginx-Based Web Application Clusters. *J. Phys. Conf. Ser.* **2018**, *1060*, 012078. [[CrossRef](#)]
- Roshdy, A.; Gaber, A.; Hantera, F.; ElSebai, M. Mobility load balancing using machine learning with case study in live network. In Proceedings of the 2018 International Conference on Innovative Trends in Computer Engineering (ITCE), Aswan, Egypt, 19–21 February 2018; pp. 145–150.
- Tall, A.; Altman, Z.; Altman, E. Self-optimizing load balancing with backhaul-constrained radio access networks. *IEEE Wirel. Commun. Lett.* **2015**, *4*, 645–648. [[CrossRef](#)]

19. Reese, W. Nginx: The high-performance web server and reverse proxy. *Linux J.* **2008**, *2008*, 2.
20. Ghomi, E.J.; Rahmani, A.M.; Qader, N.N. Load-balancing algorithms in cloud computing: A survey. *J. Netw. Comput. Appl.* **2017**, *88*, 50–71. [[CrossRef](#)]
21. Karimi, H.; Jadid, S. Multi-layer energy management of smart integrated-energy microgrid systems considering generation and demand-side flexibility. *Appl. Energy* **2023**, *339*, 120984. [[CrossRef](#)]
22. Ao, B. Integrated service system for multi-power business scenarios based on micro-service architecture. *Rev. Adhes. Adhes.* **2023**, *11*. [[CrossRef](#)]
23. Sanders, P. A detailed analysis of random polling dynamic load balancing. In Proceedings of the Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN), Kanazawa, Japan, 14–16 December 1994; pp. 382–389.
24. Usman, M.; Ferlin, S.; Brunstrom, A.; Taheri, J. A survey on observability of distributed edge & container-based microservices. *IEEE Access* **2022**, *10*, 86904–86919.
25. Docker Inc. Docker. linea]. [Junio de 2017]. 2020. Available online: <https://www.docker.com/what-docker> (accessed on 1 January 2013).
26. Gkamas, T.; Karaiskos, V.; Kontogiannis, S. Performance evaluation of distributed database strategies using docker as a service for industrial iot data: Application to industry 4.0. *Information* **2022**, *13*, 190. [[CrossRef](#)]
27. Johansson, L.; Dossot, D. *RabbitMQ Essentials: Build Distributed and Scalable Applications with Message Queuing Using RabbitMQ*; Packt Publishing Ltd.: Birmingham, UK, 2020.
28. Carlson, J. *Redis in Action*; Simon and Schuster: New York, NY, USA, 2013.
29. Nevedrov, D. Using jmeter to performance test web services. *Publ. dev2dev* **2006**, 1–11.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.