European Journal of AI, Computing & Informatics

Vol. 1 No.1 2025

Article **Open Access**

High Availability Architecture Design and Optimization Practice of Cloud Computing Platform

Yifan Yang [1,*]

[1] Viterbi school of Engineering, University of Southern California, Los Angeles, CA, 90089, USA
* Correspondence: Yifan Yang, Viterbi school of Engineering, University of Southern California, Los Angeles, CA, 90089, USA

Received: 10 April 2025
Revised: 15 April 2025
Accepted: 27 April 2025
Published: 29 April 2025

Abstract: In order to ensure service availability and continuous operation of cloud computing platform services, it is necessary to study and optimize the high availability architecture of cloud computing platform. Based on the development history of the high availability architecture of cloud computing platforms, this paper analyzes and studies the basic principles of high availability design. It first discusses how to apply various redundancy strategies to solve hardware redundancy problems, then focuses on addressing network redundancy issues, and finally provides corresponding solutions. It also provides how to improve the troubleshooting ability and rapid recovery ability of the system by adopting containerized architecture and big data monitoring.

Keywords: cloud computing platform; high availability architecture; fault tolerance mechanism; fault recovery; big data monitoring

1. Introduction

With the rapid development of cloud computing technology, enterprises have increasing requirements for its availability and scalability, aiming not only for the system's quick recovery from failures but also for the continuous and stable operation of business services. This paper describes the foundation and characteristics of HADR in cloud computing environment and it outlines the high availability structure and highlights the key significance and challenges of HADR design. After that, the key elements and optimization methods of building HADR with excellent performance and high reliability are elaborated with a practical case study, which provides feasible theoretical basis and reference method for the construction of cloud computing platform HADR.

2. Theoretical Basis

2.1. Definition and Development of Cloud Computing Platform

Cloud computing platform is a network-based computing platform that uses computing, storage and network systems and platform services on the Internet to enable customers to use resources according to needs and charge according to actual usage, greatly reducing IT costs. Service types can be classified into IaaS, PaaS, and SaaS. IaaS is the basic computing resource, PaaS is used to build and publish applications, and SaaS delivers applications directly to users to run. The development process of the cloud platform has evolved from simple virtualization technology to complex distributed computing systems and now has a wide range of applications in big data, AI, and other fields [1].

2.2. Basic Concepts of High Availability Architecture

HA architecture refers to the use of specific technologies and methods to ensure that the system can continue to provide services when problems occur and minimize service interruption time. Its purpose is to avoid a single point of failure (SPOF), improve system reliability and fault tolerance, and ensure business continuity.

In general, high availability network architecture is mainly realized through the concept of redundancy design, backup mechanisms, and self-healing capabilities. Redundancy prevents a single point of failure by backing up hardware, data, or services. For example, a group of servers are distributed in different areas, so that when one server or data center goes down, its requests will be replaced by other servers or data centers. In the backup mechanism, the system actively detects faults and switches to the fault-free node or standby device to maintain service continuity. An automatic recovery policy can take this approach and restore service availability by repairing or restarting the faulty component in a timely manner with its own recovery function after an error occurs [2].

3. High Availability Architecture Design

3.1. Overall Framework for High Availability Architecture Design of Cloud Computing Platform

The high availability architecture of the cloud platform is designed to ensure that the platform can continue providing normal external services even when abnormal conditions such as hardware failures, software errors, or network disruptions occur (Figure 1 [3].

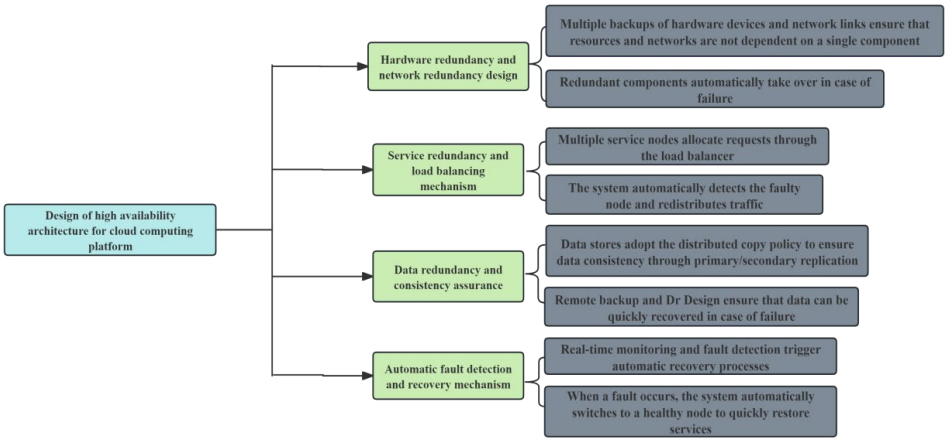

Figure 1. Flowchart of the Overall Framework for High Availability Architecture Design.

Design hardware redundancy and network redundancy: Hardware redundancy and network link redundancy are designed to avoid the single point of failure in the system. In case of device or link failure, services can be processed continuously.

Load balancing and service redundancy: At the service layer, all requests are dynamically scheduled through multiple service instances by the load balancer. The load balancer is responsible for health monitoring, and if an unhealthy instance occurs, it dynamically directs traffic to the healthy instance to ensure service availability.

Data redundancy and data consistency maintenance: Distributed storage and master/slave replication are used to implement data redundancy, ensuring that when a storage node fails, other nodes can still access the stored data.

Automatic fault detection and recovery mechanism: The automatic fault detection and recovery module of the system dynamically monitors the node status. When the system detects a fault, it automatically recovers the fault [4].

3.2. Implementation of Multi-Layer Redundancy and Fault Tolerance Mechanism

In a high availability system to ensure the normal operation of services, its redundancy and fault-tolerant design are important guarantee mechanisms. This redundancy and fault tolerance can prevent single points of failure (SPOF) in the system. In case of an interruption, the system can seamlessly switch to a redundant component, ensuring uninterrupted service delivery. The following describes the multilevel redundancy and fault-tolerant system architecture in detail.

Redundant design:

1) Hardware redundancy: Load balancing can be implemented, that is, load balancing can be performed among multiple servers to prevent a server failure from causing a system crash. The formula is as follows:

$$P_{fail} = (1 - P_{Success})^n \quad (1)$$

Where P_{fail} indicates the probability that at least one device fails, $P_{Success}$ indicates the probability that a single device does not fail, and n indicates the number of devices. With the increase of the number of redundant devices, the probability of failure decreases significantly.

2) Network redundancy: Since we connect different nodes with multiple network paths, when there is a problem in some network paths, we can also use other network paths to transmit data.
3) Data redundancy: Technical measures such as data backup number and RAID (redundant array of independent disks) can ensure that data can still be used normally even if a storage device fails. For example, the active/standby replication function in a distributed database can quickly transfer the failure of the active node to the standby node, as shown in formula (2).

$$R = \frac{\sum_{i=1}^{n}(T_{replicai})}{n} \quad (2)$$

Here, R represents the data recovery time and $T_{replicai}$ is the data recovery time of the i-th replica.

3.3. Load Balancing and Fault Recovery Design

HA is a load balancing and fault recovery technology that distributes traffic across multiple servers to avoid overloading a single server. With fault recovery, you can ensure that problems can be repaired in the first time. The main load balancing and fault recovery designs are described below.

Load balancing:

1) Application layer load balancing: In practical applications, we hand the requested load to the load balancer. The load balancer will implement intelligent scheduling based on the actual load and response time of the server. For example, we can use a weighted polling algorithm for load balancing and allocate requests based on the capabilities of individual servers.

$$L = \frac{1}{N}\sum_{i=1}^{N} L_{server\ i} \quad (3)$$

Where $L_{\text{server i}}$ indicates the load of server i and N is the number of servers.

2) Global load balancing: Applications are deployed across different data centers. Global load balancing is an intelligent routing policy based on factors such as data center location and network delay.

Fault recovery design:

Automatic fault detection and failover: The monitor can detect each service node and switch to another service node when the service node is not working, thus ensuring continuous service provision. The fault recovery formula is as follows:

$$R_{time} = T_{detect} + T_{failover} \quad (4)$$

Where R_{time} represents the total process from the generation of the problem to the end of recovery, T_{detect} represents the time required for the problem discovery process, and $T_{failover}$ represents the time required for the problem transfer.

3.4. Data Backup and Disaster Recovery Design

When a disaster occurs, you can use data backup and disaster recovery policies to quickly recover data and ensure continuous service running.

Data backup strategy:

1) Incremental backup and full back up: A full backup copies all data at once, while an incremental backup only saves data that has changed since the last backup.

Combining the two methods can reduce the backup duration and capacity without compromising data security.

2) Remote backup: Remote data backup is performed to ensure data recovery when disasters occur in the future.

The formula is as follows:

$$D_{restore} = \frac{T_{local}+T_{remote}}{2} \quad (5)$$

$D_{restore}$ is the data recovery time, T_{local} is the local recovery time, and T_{remote} is the remote recovery time.

Disaster recovery design:

1) Multi-regional disaster recovery architecture: Cloud platforms often adopt the multi-regional disaster recovery mode, that is, they operate through data centers distributed in different geographic areas. For example, if a service in one region fails, the service in another region is switched to the disaster recovery regional service.
2) Automatic recovery process: When a disaster recovery event occurs, the system automatically detects the fault and performs actions, such as data recovery, application reinstallation, and traffic switchover.

$$T_{recovery} = T_{detect} + T_{restore} + T_{switch} \quad (6)$$

$T_{recovery}$ is the time from fault occurrence to service recovery, T_{detect} is the time for fault detection, $T_{restore}$ is the time for data recovery, and T_{switch} is the time for service switchover.

4. Key Difficulties in the High Availability Architecture

4.1. Difficulties in Implementing Hardware Redundancy and Fault Tolerance

Although implementing hardware redundancy and fault-tolerant hardware infrastructure is one way, there are many problems in practice. The use of excess hardware means an increase in the total cost, especially when a large number of uses, additional hardware purchase and maintenance require a larger amount of capital expenditure. In addition, hardware redundancy relies on the cooperation between different components and the consistency of data. If one component fails, it may cause the entire system to fail [5].

4.2. Multi-Layer Network Redundancy and Failover Problems

Although it is necessary to consider the problems of multilevel network reuse and failover to build a highly reliable system, there are still many difficulties in the process of realizing this strategy. Effective redundant network design scheme is difficult to obtain, especially in the large-scale distributed system, how to reasonably design the lines and equipment use of redundant network, to prevent the problem of line overload or idle redundant equipment.

Network link faults of different nature, such as delay and packet loss, may degrade the quality of service, or routing protocols in distributed networks may be delayed. When a network fault occurs, the unavailable time of service quality will be increased.

4.3. Conflict between Distributed Data Consistency and High Availability

In the distributed architecture, the conflict over the consistency and reliability of information is contradictory. In the CAP theory, if the network is divided, the integrity and

availability of information cannot be provided, thus forcing a trade-off between consistency and availability during system design. For example, in the design of the master-slave structure, when the master machine fails, the backup machine starts to provide services, which may lead to data loss or delay synchronization, and the data stored on each device is not in the same state. This happens frequently in distributed databases. In order to avoid consistency problems, many systems adopt the ultimate consistency strategy, that is, allow a brief inconsistency, and then make all the data consistent. However, this method can lead to serious conflicts and increase the difficulty of handling.

4.4. Technical Challenges of Automated Fault Detection and Recovery

Although the automatic fault diagnosis and repair strategy is based on the highly reliable basic structure system, it has many technical challenges. Due to the nature of large-scale distributed systems, fault phenomena in large-scale distributed systems are often reflected through subtle indicators such as minor performance degradation or transient error logs, which traditional monitoring tools may not promptly detect, resulting in delayed fault identification. In addition, the states of components in the system change quickly and complex, how to build an effective, timely and efficient fault detection method to locate the fault location is still a challenging technical problem.

During data recovery, maintaining data synchronization across multiple service nodes presents a significant challenge, particularly in avoiding data loss and ensuring a smooth recovery process. When data is synchronized between multiple service nodes, how to avoid data loss and make the recovery process as smooth as possible is also a problem to be solved during the recovery process.

5. Optimize the High Availability Architecture

5.1. Containerized Architecture Improves Service Fault Tolerance

Containerized architecture improves service trust and reliability through microkernel architecture. Compared to traditional virtual machine architectures, unlike traditional virtual machines, which each run a full operating system and can host multiple applications, containerized architectures share the same OS kernel, enabling multiple applications and services to run with significantly reduced overhead and improved resource efficiency. All services in this architecture are compressed into containers, making service startup, movement, and recovery faster (See Figure 2).

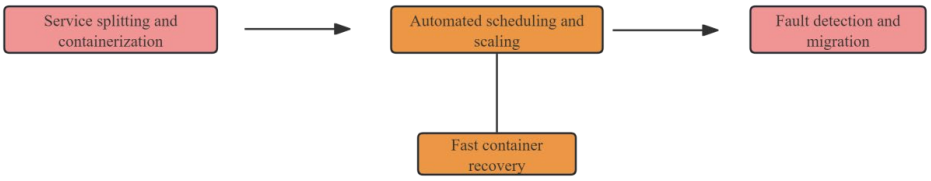

Figure 2. Architecture Process of Containerization for Enhancing Service Fault Tolerance.

The above flowchart illustrates how containerization can improve the robustness of a cloud computing service platform. Container technology subdivides services into individual, tiny containers, each of which is a packaged application or service. Therefore, even if one container fails, the other containers will still work.

First, the complex services are divided into numerous lightweight containers, each running independently to isolate faults and enhance stability, so that each service can work on its own. In this way, we can avoid the instability of a single application failure leading to an entire system deadlock. Second, with a container scheduler such as Kubernetes, the number of containers assigned to a service can be determined based on the

amount of work. As traffic increases, the number of new container instances is automatically added, and conversely, when a container has a problem, it automatically drifts or restarts. Third, the system will continuously monitor the operating status of the container and check whether it is available through health detection. If a problem occurs, traffic to that container is automatically interrupted and diverted to other normal container instances. Fourth, when a fault occurs, the device restarts quickly, quickly restores to normal through mirroring/backup, and the service interruption is shortest.

5.2. Calculating the Optimal Path Based on the Link Status

High-efficiency network architectures utilize link-state protocols such as OSPF (Open Shortest Path First) to make optimal routing decisions to make optimal path decision, but the instability of network structure will affect the efficiency and quality of network data transmission. Through the link state protocol, the status information of all nodes in the network can be obtained in real time, and the optimal path can be dynamically planned to ensure the transmission of data (Figure 3).

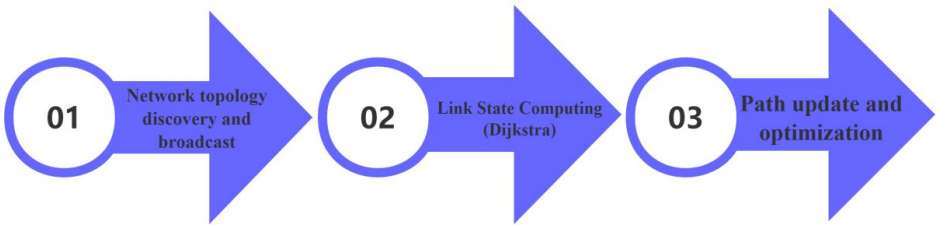

Figure 3. Flow of Optimal Path Calculation for Link State.

In the first stage, each router broadcasts its local link status information, receives topology updates from other routers, and refreshes its own topology database accordingly. After that, Dijkstra algorithm is used to optimize the path through the weight and delay of the link. Eventually, if the link breaks or the topology changes, the algorithm recalculates the shortest path to ensure flow to the healthy node.

Using link state protocol and Dijkstra algorithm, the network can automatically adjust the topology structure and provide good network connectivity.

5.3. Balancing Consistency and High Availability for Fault Recovery

In a distributed architecture, ensuring system availability and performance often requires a tradeoff between data consistency and availability, leading to the adoption of eventual consistency models for certain data. In other words, some data inconsistencies are allowed for a short period of time on some data, and are corrected to a consistent result after the data is synchronized and updated.

Tradeoffs between consistency and high availability:

1) High Availability priority (AP model): Data inconsistency is tolerated before all replicas are synchronized, and the system ensures that data is consistent again after all replicas are synchronized.
2) Consistency priority (CP model): The system prioritizes maintaining strong data consistency even at the cost of availability. In the event of a system fault, operations are suspended until consistency across replicas can be confirmed. Such as formula (7).

$$REO = T_{detect} + T_{failover} + T_{sync} \quad (7)$$

REO is the recovery time target, T_{detect} is the fault detection time, $T_{failover}$ is the failover time, and T_{sync} is the data synchronization time. This balance ensures that the system is able to choose the right balance between consistency and availability based on business requirements.

5.4. Monitor System Indicators Using the Big Data Monitoring Platform

The construction of a high-reliability cloud computing platform is mainly reflected in the real-time monitoring of various system indicators to ensure the stable operation and uninterrupted service of the cloud computing platform. However, as cloud computing environments grow increasingly complex, traditional data collection and management methods can no longer meet the massive and dynamic information requirements. Therefore, big data monitoring tools can be used for fast and accurate collection, storage and analysis of big data, such as the big data monitoring platform of Prometheus and Grafana, enabling cloud administrators to monitor the environment in real time, promptly identify potential issues, and provide optimization recommendations. In addition to its rich and flexible dashboard visualizations, Grafana also supports multi-metric analysis, so that technicians can more intuitively understand the status of the cloud computing platform, and can set custom alarm rules to automate monitoring.

Workflow of big data monitoring platform: collect indicator information, such as CPU usage rate, memory usage rate, I/O throughput, etc., relying on services, nodes, containers and other forms; Store and analyze the collected data, store the data in the time series database, analyze and visualize the data in real time; Generate an alarm rule. Once an indicator exceeds the predefined threshold, the system automatically generates an alarm and takes the predefined measures.

6. Conclusion

This paper comprehensively analyzes the design strategies for improving the availability and reliability of cloud computing platforms. It summarizes the key technologies and challenges of high-availability architectures, proposes optimization methods such as containerization, multi-layer redundancy, and load balancing, and offers practical reference suggestions for enhancing the stability of cloud service platforms. Future work can focus on intelligent fault prediction and autonomous recovery mechanisms to further boost system resilience.

References

1. B. R. Cherukuri, "Development of design patterns with adaptive user interface for cloud native microservice architecture using deep learning with IoT," in *Proc. IEEE Int. Conf. Comput., Power Commun. Technol. (IC2PCT)*, Greater Noida, India, 2024, pp. 1866-1871, doi: 10.1109/IC2PCT60090.2024.10486720.
2. C. Cai, N. X. Zhao, R. N. Xiao, and X. Z. Wang, "Study on prosperity index system and warning monitoring of road freight transport market based on big data," *J. Highw. Transp. Res. Dev. (Engl. Ed.)*, vol. 17, no. 4, pp. 59-67, 2023, doi: 10.1061/JHTRCQ.0000882.
3. X. Xu, S. Zang, M. Bilal, X. Xu, and W. Dou, "Intelligent architecture and platforms for private edge cloud systems: A review," *Future Gener. Comput. Syst.*, 2024, doi: 10.1016/j.future.2024.06.024.
4. D. Mwale, L. Manda-Taylor, A. Likumbo, M. B. Van Hensbroek, J. Calis, W. Janssens, et al., "PP044 Topic: AS04–Emerging sciences, methodologies, big data and technology: Monitoring critically ill children in Malawi: A qualitative study," *Pediatr. Crit. Care Med.*, vol. 25, no. 11S, p. e34, 2024, doi: 10.1097/01.pcc.0001084800.19764.b6.
5. D. Xie, "Application of big data association rule algorithm in accounting network security monitoring and accounting system," *Procedia Comput. Sci.*, vol. 247, pp. 327–334, 2024, doi: 10.1016/j.procs.2024.10.038.