American Research Journal of Computer Science and Information Technology

Volume 7, Issue 1, 6-10 Pages
Research Article | Open Access
ISSN (Online)- 2572-2921
DOI : 10.21694/2572-2921.24002

AMERICAN RESEARCH JOURNALS
An Academic Publishing House

Designing System Architecture with High Availability and Scalability

Nikhil Badwaik

Software Engineer at NIKE INC, Portland OR, USA.

ABSTRACT

The article discusses modern approaches and methods of designing systems that can adapt to changing operating conditions and market requirements. The emphasis is on the need to integrate scalability and reliability into architectural solutions that ensure smooth operation under various loads. This paper elaborates on the principles of designing scalable systems, encompassing the utilization of templates, antipatterns, subsystem decomposition, and asynchronous data processing. Strategies for high availability and fault tolerance, including data replication and the use of distributed databases, are also highlighted. The importance of collaboration of various components of the system with minimizing dependencies is emphasized to achieve optimal performance and save resources.

KEYWORDS: *scalability, architecture development, programming, software, modern technologies.*

INTRODUCTION

In modern online application development, it is important to build platforms that not only function efficiently under various workloads but can also scale in response to changing market demands. Designing a system architecture with high availability and scalability is a critical task in today's world where businesses and services depend on the smooth operation of IT infrastructure. As the number of users and the amount of data to be processed increases, traditional monolithic systems face scalability and reliability challenges. High system availability ensures that services are available to users at all times, minimizing downtime and financial losses. Scalability allows the system to efficiently handle increasing workloads, enabling growth without significant changes in the architecture. The relevance of this topic is due to the rapid development of digital technologies, increased requirements for quality of service, and the need to adapt to rapidly changing market conditions. The main challenge is to find a balance between availability, scalability, performance, and cost-effectiveness, which requires a deep understanding of different approaches and technologies in the field of system architecture.

GENERAL CHARACTERIZATION

When designing database architectures for cloud applications, it is critical to ensure high availability, performance, scalability, fault tolerance, and disaster recovery. Traditional on-premises web application development environments often have infrastructure constraints, limited access to developer resources, operating group support, the need to scale resources to cope with peak loads, and ongoing infrastructure maintenance requirements. The development process is dependent on the availability of new technologies and infrastructure, which can delay the realization of business functions.

Alternatively, databases can be categorized as relational database management systems (RDBMS) and non-relational NoSQL databases. The choice between these technologies depends on specific requirements and usage scenarios. NoSQL databases, such as document-oriented or key-value stores, support flexible schemas and are often chosen for their ability to scale horizontally, which is ideal for high-performance scenarios.

In localized environments, scaling is often a challenge. Estimating expected load and tuning hardware resources requires going through several levels of approval and justification. In contrast, cloud environments offer the flexibility to dynamically scale resources in response to performance changes, which also helps to optimize resource utilization and manage the cost of cloud services [1].

Scaling is commonly categorized into horizontal and vertical scaling (Fig.1.). Vertical scaling increases the allocated resources (CPU, memory, storage) for the database, which provides immediate performance improvement and the ability to process more transactions. In a cloud environment,

the process of vertical scaling is simplified by the ability to resize database instances on the fly, without the need to physically deploy new hardware. This allows quick adaptation to changing performance requirements, reducing unnecessary costs.

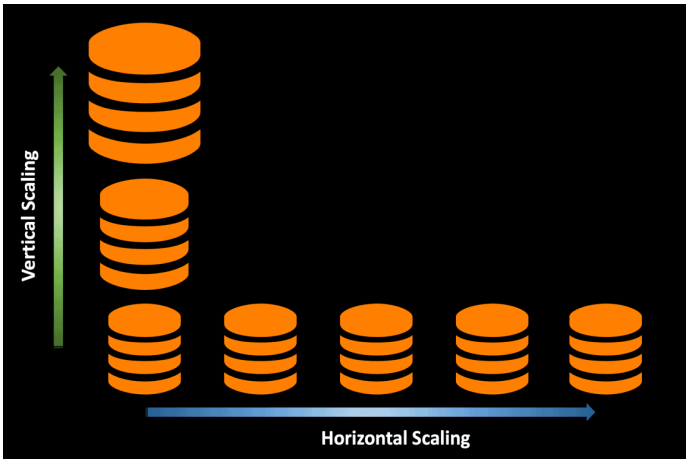

Fig 1. Horizontal and vertical scaling [2].

Horizontal scaling is a strategy to increase query processing and manage the growing workload that exceeds the capacity of a single database instance. This technique allows adding additional database instances, thus realizing scaling through expansion [2].

There are several approaches to horizontal database scaling:

- Adding read replicas to reduce the load on the main database by processing read operations on separate servers.
- Caching data before querying the main database to reduce the frequency of database accesses.
- Sharding or splitting the database into multiple parts that are distributed across different servers to improve both read and write data performance.

In a cloud environment, horizontal scaling is made easier by the ability to dynamically manage database instances according to the current workload. This eliminates the need to maintain large instances during periods of low activity, thereby enabling cost optimization.

There are costs associated with using automatic database scaling in a cloud environment, but these costs are directly correlated to application traffic and workload. Conducting load tests on different database instance sizes can help determine optimal read and write performance metrics that meet business requirements and service level agreements (SLAs).

Next, let's look at the basic principles of scalability. The principles of scalability form a framework for understanding and designing systems that can adapt to changes in workload. These fundamentals include the behavioral and structural aspects on which all scalable design patterns, rules, and anti-patterns are based. Mastering these principles enables the design of scalable systems without the need for in-depth study of every detail of such systems. Architectural and design choices should be guided by these fundamental principles:

- Simplicity: Simplicity of design facilitates not only scalability but also system development, deployment, maintenance, and support.
- Partitioning: Effectively dividing the system into smaller, manageable subsystems allows each subsystem to perform autonomous functions. These subsystems can function independently in separate processes or threads, scaling through a variety of load balancing and customization techniques.
- Asynchronous: Asynchronous processing allows the system to perform tasks without blocking resources, although it requires complex design and testing approaches due to its peculiarities.
- Loose coupling: Reducing coupling and increasing coupling is key to increasing application scalability. Low coupling provides flexibility in choosing optimization strategies for different subsystems, whereas high coupling can complicate the interaction between components by requiring a combination of local and remote calls to perform operations.
- Parallelism and Parallelization: Parallelism allows multiple tasks to execute simultaneously using shared resources, while parallelization means dividing a single task into many independent tasks capable of simultaneous execution.
- Economical use of system: Economical use of system resources is important to the architect and developer. Efficient use of resources such as CPU, disk, memory, network, and databases not only increases efficiency but also minimizes the consumption of scarce resources.
- Decentralization/Distribution: Distributed systems consist of subsystems that run on independent servers and are perceived by users as a single system. Such systems provide high scalability and availability by allowing additional servers to be added [3].

EXISTING STRATEGIES

Database caching. In the development of distributed systems that require high performance to meet SLA (Service Level Agreement) standards, database optimization is key. A sound caching strategy can significantly improve application performance and reliability, reducing database load and lowering operational costs. Implementing a cache allows frequently requested data to be stored directly in memory, minimizing the need for repeated costly database queries. Queries that require accessing multiple tables or executing complex stored procedures become much more efficient if values are retrieved from the cache in a fraction of a millisecond instead of each database access (Fig.2.).

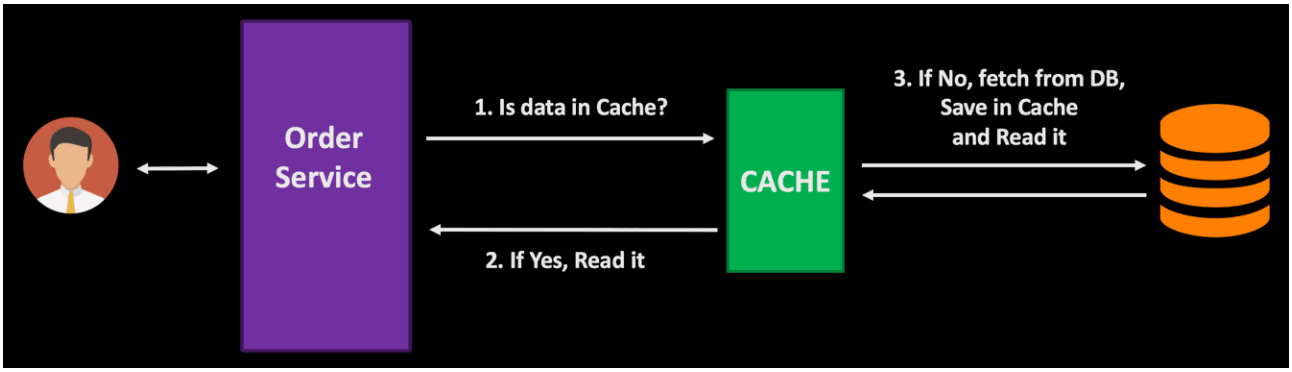

Fig 2. Example of caching [2].

Database performance improvement can be achieved through the use of advanced query optimization techniques. However, for data that is frequently accessed, it is possible to significantly reduce the load on the database and speed up the response time by implementing in-memory caching of this data. Based on the analysis of performance requirements and data access patterns, develop a personalized caching strategy by determining which data should be cached and setting cache retention times [4].

Database sharding. With increasing traffic and application scale, it may be necessary to consider techniques to optimize database performance, among which sharding occupies an important place. Sharding allows data to be distributed across different servers, which satisfies the scalability needs of modern distributed systems and allows large amounts of data to be managed more efficiently. This technique improves the performance of both read and write operations as each database serves a smaller amount of data. Sharding can be thought of as the process of creating separate databases, each functioning as an independent segment.

When implementing sharding, data is distributed across different nodes based on the so-called shard key. Each shard contains only part of the data, which facilitates faster and more efficient data management in each shard. Operations are performed in parallel across all segments, which can be compared to horizontal partitioning, in which data is divided among multiple stores to achieve horizontal scalability (Figure 3).

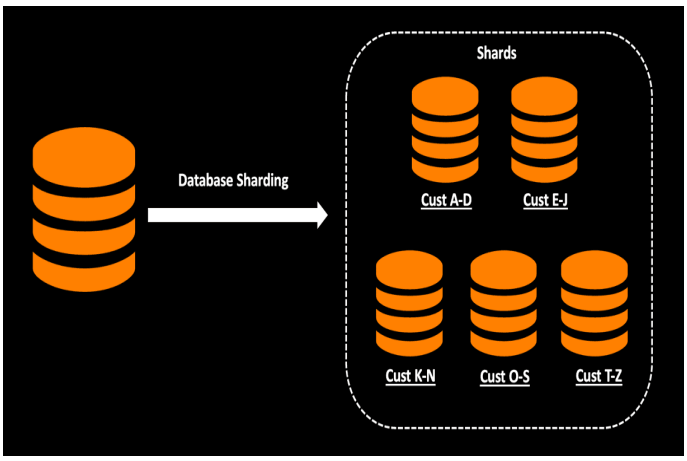

Fig 3. An example of database sharding [2].

Database design in microservice architecture. Handling database changes in a microservices architecture is challenging. When developing cloud services, each microservice must have its own separate database. This will allow deployment and scaling of microservices independently. In the diagram below, all four services will have different loads, so it makes sense to have separate data stores. This type of design can be called decentralized data management architecture and is a very common pattern when designing highly scalable distributed systems.

When these services have a single monolithic shared database, it is very difficult to scale the database based on traffic spikes. This design pattern is typical for traditional applications in which data is often distributed among different components. However, tight coupling between services will prevent service changes from being deployed independently. The only option is to scale the entire monolithic database - scaling a single component is not possible [5].

LOAD BALANCING METHODS

In the realm of scalable system design, load balancing emerges as a pivotal element that ensures optimal performance and reliability. By evenly distributing incoming requests across multiple servers, load balancing prevents any single server from becoming a bottleneck, thereby enhancing both availability and efficiency.

Round Robin load balancing is a fundamental technique that distributes incoming requests sequentially across a set of servers. This method is predicated on the assumption that all servers have equivalent capacity and performance. The essence of Round Robin lies in its simplicity: each server receives a request in turn, forming a continuous loop.

In practice, this method operates by cycling through the list of available servers. For instance, if there are three servers, the first request goes to Server 1, the second to Server 2, the third to Server 3, and the fourth request returns to Server 1. This cyclical allocation ensures that no single server is disproportionately loaded, provided that the servers are homogenous in capability [6].

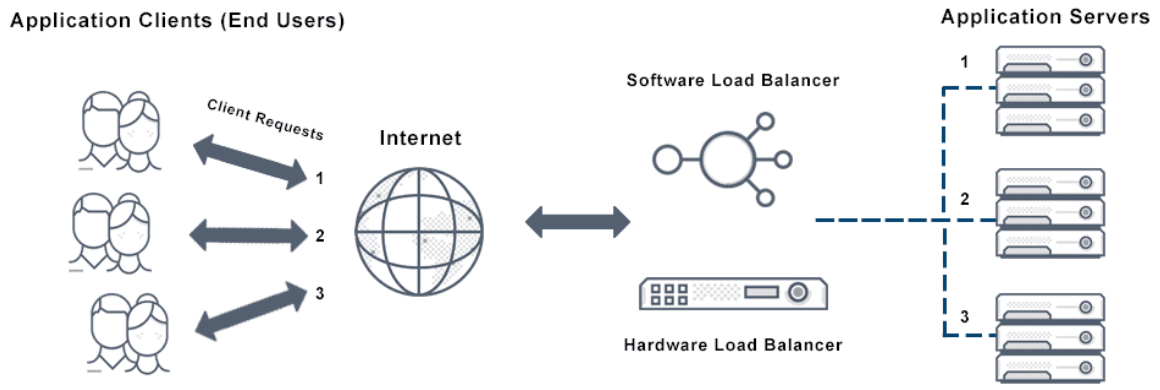

Fig 4. Round Robin load balancing

However, the simplicity of Round Robin can be a double-edged sword. In environments where server capacities vary significantly, this method can lead to inefficiencies, as it does not account for the current load or the individual server performance metrics.

The Least Connections method offers a more dynamic approach to load distribution by directing traffic to the server with the fewest active connections. This method is particularly advantageous in scenarios where connection durations are highly variable and can significantly affect server load.

Operationally, each server maintains a count of active connections. When a new request arrives, the load balancer assigns it to the server with the lowest connection count. This approach ensures a more balanced load distribution, as it dynamically adjusts to the fluctuating state of server workloads [7].

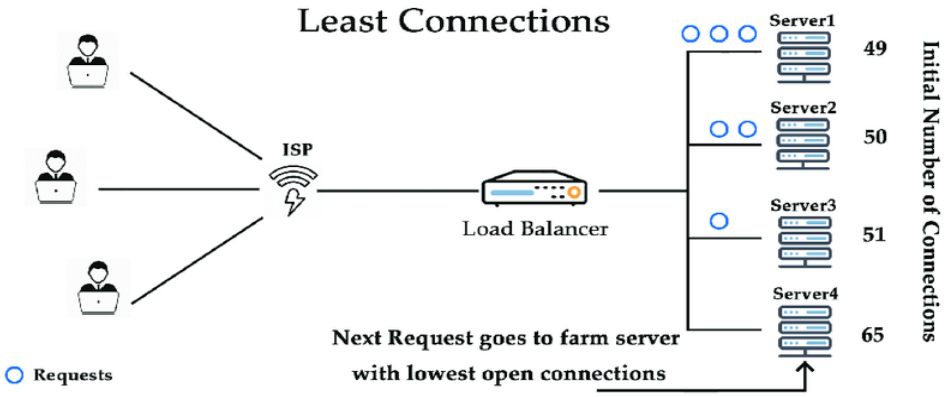

Fig 5. Least Connections Load Balancing

For example, in a web service scenario where some requests may result in long-running connections, directing new requests to the least loaded server helps maintain an even distribution of active connections, thereby optimizing response times and resource utilization.

IP Hash load balancing introduces a deterministic method of traffic distribution based on the client's IP address. By hashing the IP address and mapping the resultant hash to a specific server, this method ensures that requests from the same client are consistently directed to the same server [8].

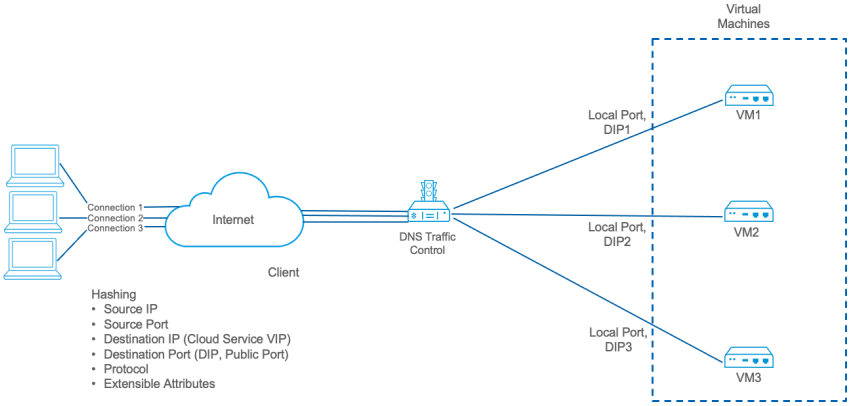

Fig 6. IP Hash Load Balancing

This method is particularly useful for maintaining session persistence, as it allows for stateful interactions where client-specific data must be stored on a particular server. The deterministic nature of IP Hash also aids in load predictability and can simplify debugging by ensuring a consistent routing of client requests.

In environments where server capacities are heterogeneous, Weighted Load Balancing becomes indispensable. This method assigns a weight to each server, reflecting its processing capacity or performance characteristics. Servers with higher weights receive a proportionally larger share of the incoming traffic [9].

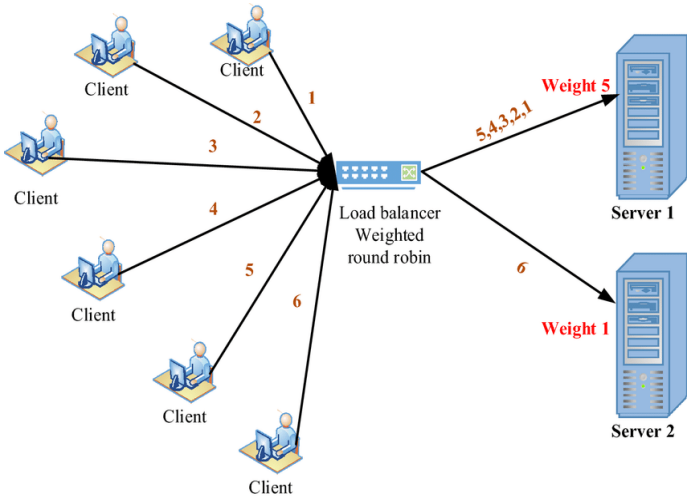

Fig 7. Weighted Load Balancing

The implementation involves calculating the proportion of requests each server should handle based on its weight. For instance, if Server A has a weight of 5 and Server B has a weight of 1, Server A will handle three times as many requests as Server B. This proportional distribution ensures that more capable servers are utilized to their full potential, thereby optimizing overall system performance.

CONCLUSION

Designing a system architecture that is highly available and scalable requires a holistic approach that considers both functional and non-functional requirements. The design of such systems involves the application of proven principles and design patterns that ensure the system is flexible, reliable, and able to adapt to changing operating conditions. Particular attention is paid to asynchrony, decomposition into subsystems, reducing the degree of coupling between components, and the use of data replication and distribution techniques to ensure fault tolerance and increase availability. Efficient use of resources and integration with modern cloud technologies allow achieving high performance at minimal cost. The approaches described in the paper show how theoretical knowledge can be successfully applied in practice to create scalable and reliable systems that can meet current and future business needs.

REFERENCES

1. Building Blocks of a Scalable Architecture. [Electronic resource] Access mode: https://dzone.com/articles/component-load-testing (accessed 8.05.2024).
2. Designing Highly Scalable Database Architecture. [Electronic resource] Access mode: https://www.red-gate.com/simple-talk/databases/sql-server/performance-sql-server/designing-highly-scalable-database-architectures / (accessed 8.05.2024).
3. Architecture design experience: high availability and scalability. [Electronic resource] Access mode: https://www.codetd.com/en/article/14194381 (accessed 8.05.2024).
4. Design for scale and high availability. [Electronic resource] Access mode: https://cloud .google.com/architecture/framework/reliability/design-scale-high-availability (accessed 8.05.2024).
5. Patterns and Anti-Patterns for Scalable and Available Cloud Architectures. [Electronic resource] Access mode: https://www.infoq.com/news/2014/04/Cloud-Architecture-Patterns / (accessed 8.05.2024).
6. Round Robin Load Balancing Definition. [Electronic resource] Access mode: https://avinetworks.com/glossary/round-robin-load-balancing/
7. Alankar B. et al. Experimental setup for investigating the efficient load balancing algorithms on virtual cloud // Sensors. – 2020. – T. 20. – №. 24. – C. 7342.
8. SOURCE IP HASH LOAD BALANCING FOR APPLICATION PERSISTENCY. [Electronic resource] Access mode: https://blogs.infoblox.com/community/source-ip-hash-load-balancing-for-application-persistency/
9. Jyoti A. et al. Cloud computing using load balancing and service broker policy for IT service: a taxonomy and survey //Journal of Ambient Intelligence and Humanized Computing. – 2020. – T. 11. – C. 4785-4814.

Citation: Nikhil Badwaik, "Designing System Architecture with High Availability and Scalability", American Research Journal of Computer Science and Information Technology, Vol 7, no. 1, 2024, pp. 6-10.