International Journal on Science and Technology (IJSAT)

E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Unified System Design: A Comprehensive Study on Scalability, Access Control, and Communication Protocols

Akash Rakesh Sinha

Software Engineer 3, Walmart Inc.

Abstract

As organizations worldwide is increasingly dependent on technology for driving innovation and operational efficiency, effective systems design has become a key pillar on which long-term successful organizations are built. This paper discusses the system design basics, major architectural patterns and their use cases for performance tuning. It also explores the evolution of communication protocols, most notably HTTP/1.1, HTTP/2, and HTTP/3, and the effects they've had on API design in distributed contexts. At the heart of the conversation lies authorization frameworks like Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC) and Policy-Based Access Control (PBAC) which are pivotal to protecting security, privacy and ensuring compliance with regulations.

This paper expounds upon these pitfalls while bringing to light real constructs that challenge developers, specifically in microservices-based ecosystems, where distributed architectures complicate facets such as data consistency, caching policies, and robust error handling. Specific best practices and emerging trends shed light on how theoretical principles is transmuted into practical implementation. Focusing on this disparity, this paper attempts to empower system architect, software engineer and decision-maker to evolve applications towards scalable, secure and uninterrupted future by compiling findings between industry experiences and academic literature. It highlights the importance of aligning architectural paradigms, efficient data transfer protocols, and sophisticated access control frameworks to build robust and scalable systems that can respond to changing business and technology trends.

Keywords: System Design, Distributed Systems, Microservices, Scalability, Performance Optimization, Load Balancing, HTTP/2, HTTP/3, REST, GraphQL, Policy-Based Access Control (PBAC), Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC), OAuth, Security, Compliance

1. Introduction

With the relentless wave of digital transformation impacting just about every industry, the demand for building systems that are scalable, secure and agile are more important than ever. Modern architectures should provide high availability and effective performance under fluctuations in workload, leading many software teams to adopt distributed and microservices-based solutions. Seen in this light, although these approaches are valuable ways to enhance agility and scalability, they also present potential risks associated with data integrity, security breaches, and regulatory compliance.

Hence access control steps into the limelight with organizations moving towards structured frameworks like RBAC, ABAC and PBAC to cater to change user context and changing resource needs. At the same time, the development of communication protocols as seen in the move from HTTP/1.1 to HTTP/2 and HTTP/3 is still improving throughput and reducing latency while granting developers the ability to build flexible APIs that can scale with the future.

This paper focuses on the convergence of three crucial elements: scalability, security, and modern protocol design. This book covers system design patterns, caching patterns, higher-level authorization patterns, and the artfulness of automating distributed- systems performance. In this regard, the paper aims to emphasize how architectural considerations during the design process, the adoption of protocol standards, and carefully optimized access reviews, can allow organizations to design reliable systems, which not only exceed demanding user expectations but also can be aligned with ever increasing compliance requirements.

2. System Design Fundamentals and Architectural Patterns

At the heart of sustainable, high-performing software is system design experience. Good designs dictate the performance, stability, and portability of applications throughout their lifetimes. These principles further segment complex infrastructures into smaller, manageable components & encapsulates internal state.

Modular Design and Layered Architecture

In general, layered architectures are structured such that concerns are separated into distinct layers, including presentation, application (or domain) and data management (which can include data warehouse concerns). It abstracts out specific functional responsibility to each layer. The presentation layer deals with user interface and client-side logic, application or domain layer focuses on the application logic, and the data layer is responsible for persistence and retrieval operations, for example.

These clear boundaries make testing and maintenance easier, and encourage parallel development, since separate teams can focus on different layers without the opportunity for the massive impeding of one layer upon another.

Monolithic vs Microservices Architectures Many organizations start off with a monolithic architecture, as it is easier to manage for smaller applications and also easier to deploy. But as more systems are added, monoliths often become complex to maintain, scale, and upgrade. Microservices architectures address these challenges by breaking down applications into multiple self-contained services, organized around a specific business function. This allows teams to develop, test, and deploy these services in parallel, enabling a rapid iteration process. Microservices may bring agility and service-level scalability, but they also introduce complexities from inter-service communication, data synchronization, and operational overhead.

Event-Driven Systems

Event-driven designs emphasize asynchronous interactions, where services broadcast and react to events rather than perpetually waiting for direct API calls. This decoupling increases overall responsiveness of the system and improves fault tolerance. Approaches for processing event flows include the use of message brokers or streaming platforms (e.g. RabbitMQ, Apache Kafka), which guarantee that each service handles the events independently. However, teams utilizing event-driven approaches need to carefully manage event choreography and the danger of inconsistent data states.

Balancing Trade-Offs

To choose a suitable architectural paradigm, you need to balance cost, time to market, operational complexity, and performance needs. Monolithic systems can speed up early development, but they face scaling issues as the application scales.

Microservices facilitate rapid evolution and robustness but also demand advanced orchestration and communication controls. The event-driven framework promotes flexibility and fault tolerance but requires careful consideration of event ordering and data consistency. Ultimately, architecture must be aligned with organizational goals, available technical resources, and expected patterns of growth.

3. Scalability and Performance Optimization Strategies

The systems that provide responsive user experiences under increasing or variable load are essential in a competitive landscape. Scalability and performance optimization strategies focus on ensuring that applications maintain robust throughput, minimal latency, and consistent reliability when faced with changing user demands.

Load Balancing Approaches

Load balancing lay the foundation for horizontal scaling by distributing incoming requests evenly between multiple servers or containers. Hardware- based load balancers like F5 which may be the best in dedicated performance but also not always easy on the budget. Software-based solutions such as HAProxy and Nginx are usually more versatile and less expensive to run; however, they can require extra configuration work. Careful placement of load- balancers helps distribute server load for responsive service times.

Horizontal vs Vertical Scaling

Vertical scaling which is adding more CPU, memory, or disk to a server can be agile in the presence of moderate workloads. However, it has fundamental hardware constraints and can quickly become economically expensive. Horizontal scaling, on the other hand, simply adds additional servers or containers behind a load balancer, sharing traffic loads. While such an approach usually enhances resilience and performance, it creates additional challenges around preserving the same session state and maintaining data consistency across nodes.

Performance Tuning and Caching

Caching enhances performance by returning the information directly from memory instead of reading from a slow disk or remote database. In-memory caching solutions such as Redis or Memcached can hold key-value pairs, and distributed caching is necessary for keeping data consistent among microservices-based architectures. They also can help reduce latency caused by geographic distance by caching static resources at edge nodes close to end- users (Content Delivery Networks/CDNs). To avoid serving stale data, each caching layer must involve well-defined invalidation policies, which means a careful plan on when to refresh the cache and a constant background refresh to keep it up to date.

Database Performance Profiling and Query Optimization

Well, databases are often the bottleneck in high- traffic systems. Profiling on a regular basis can help find slow queries and bad indexing strategy, leading to optimization options like denormalization, sharding, or specialized indexing. For a sizable amount of data, reading replicas can reduce backup queries and decreasing the burden on primary nodes, applying asynchronous replication can be used to avoid performance drops. When choosing between relational SQL databases (PostgreSQL, MySQL) and NoSQL options (Cassandra, MongoDB), it all boils down to how your data is structured, your consistency requirements, and your growth patterns.

Balancing Cost and Benefits

The horizontal scale with distributed caching is more flexible, but taxes administration for synchronization and possible data duplication. On the other hand, advanced solutions like container orchestration platforms (Kubernetes, Docker Swarm) offer simplified scaling but introduce operational complexity. Balancing cost vs. user experience is an ongoing process of capacity planning, load testing, and iterative tuning.

4. Communication Protocols and API Design

Robust communication protocols and well-designed APIs are the need of the hour for modern distributed systems. Technological progress in HTTP coupled by innovations such as gRPC and WebSockets has expanded the range of available solutions for real- time data exchange, resource management, and service orchestration.

HTTP Evolution (HTTP/1.1, HTTP/2, HTTP/3)

At first, HTTP/1.1 brought persistent connections and chunked data transfers to general use. But it still struggled with the head of line blocking problem, which limited requests concurrency. HTTP/2 [HTTP2] addressed these limitations with multiplexing, header compression, and server push, allowing it to interleave multiple concurrent request- response calls. HTTP/3, which runs on the QUIC protocol, goes a step further in reducing latency and increasing reliability by utilizing a UDP-based approach rather than TCP, eliminating head-of-line blocking at the transport layer and speeding up secure connection setup.

REST vs. GraphQL

REST defines resources and how they are manipulated through stateless HTTP operations. It has evolved into a de facto standard for microservices, thanks to its simplicity and clarity. Some clients experience over-fetching and under- fetching problems if they require only a few fields of a resource. GraphQL takes that to a level more flexible and less wasteful: clients specify exactly which bits of the data they want, eliminating unnecessary payloads. That said, while this can dramatically improve client performance, it also centralizes data logic, so careful schema design and query optimization are of utmost importance.

gRPC and WebSockets

gRPC is based on HTTP/2 and allows the creation of unary and streaming Remote Procedure Calls (RPCs) commonly required for high-performance and low- latency microservices. This design pattern works well for internal service-to-service communication, particularly if the system components need to exchange data back and forth or send streams of information back and forth continuously.

WebSockets also allows full-duplex communication. While useful for chat services, collaborative editing or real-time gaming, they add extra complexity over simpler request-response style paradigms (that could be per-message, per-transaction, etc.)

API Design Best Practices

But no matter what protocol is selected, APIs always benefit from similar considerations for design (consistency, versioning, automated documentation, security depth). Keeping consistent naming and structural conventions helps in avoiding confusion between developers, while versioning prevents disruption during feature updates. Documentation usually written using OpenAPI or similar specs need to remain up to date as endpoints change. Security is of utmost importance at all layers, from authentication and authorization to encryption of transport.

Protocol Selection Considerations

The choice of an underlying protocol is based on domain-specific considerations like data volume, real-time latency, concurrency, and developer familiarity. While a traditional CRUD microservice may only need RESTful HTTP/2 endpoints, something like a streaming or chat application may take advantage of WebSockets or gRPC. Reviewing latency limits, shape of data, and possibility of extension is important to make sure that the developers have a protocol that best fits to their immediate requirement and future expansions as well.

5. Access Control and Authorization Models

Managing which resources users and services can access and which they can't is important for both organizational security and user privacy. As systems become more distributed, and data becomes more sensitive, strong authorization becomes part of the design.

Role Based Access Control (RBAC)

RBAC takes a simpler approach to permissions management by organizing sets of privileges into discrete roles, like "admin," "editor," or "viewer." In a large organization with well-defined operational structures, users or user groups "inherit" the access control permissions defined for the roles assigned to them, minimizing complexity. Yet, in dynamic environments with many exceptions or ever-changing responsibilities, RBAC can become cumbersome, as roles or role-permission mappings pile up.

Attribute Based Access Control (ABAC)

Not only does ABAC allow more granularity to the definition of access but it also extends the scope of authorization. For example, from user role or departmental affiliation to contextual and/or environmental factors (time, location, state of the device, etc). This lets for fine-grain policies, e.g., "Allow resource read if user. department matches resource. department and current_time is in
working_hours." ABAC, while providing more granular control, can have complex policy definitions requiring a more sophisticated policy engine and more rigorous auditing.

Policy Based Access Control (PBAC)

PBAC provides a cohesive framework for specifying, assessing, and maintaining rules that can include roles, attributes, and contextual conditions. These policies can be systematically expressed using policy specification languages (e.g., XACML) or custom rules. PBAC engines (often consisting of a Policy Decision Point (PDP) and Policy Enforcement Point (PEP)) grant access decisions in real time, allowing enterprises to quickly adapt new business rules. This layered architecture facilitates audits and consistency, notably within microservices, where individual services can delegate authorization decisions to a centralized PDP.

Performance and Scalability in Authorization While fine-grained methods offer the best security, they at the same time add to the computational burden. Frequent policies evaluation can also lead to performance penalties, especially for high-volume microservices that need to validate access for each command. Overhead, however, can be mitigated by techniques such as local caching, short-lived tokens (e.g., JWT), or distributed policy caches. In addition, using asynchronous or batched authorization checks in the right scenarios can reduce the pressure on real- time policy engines as well.

Emerging Directions in Access Control Advanced frameworks incorporate contextual awareness assessing real-time risk based on factors
such as geolocation, device health or user behavioral patterns. In the long run, machine learning could potentially be used to detect unauthorized attempts or update rules in policy automatically based on

anomalous user activity. This constant evolution points to a critical need for open, future-proof authorization solutions.

6. Security, Privacy, and Compliance

Security is a foundational pillar to system design, but it can be intertwined with requirements for privacy and compliance, which are more prominent in domains like finance, healthcare, and government services. With the unwieldy adoption of distributed and cloud-native solutions, the issues get compounded, and defense-in-depth techniques are required.

HTTP Security Considerations

TLS/SSL encryption on HTTP traffic has become a minimum requirement for protecting data in transit. The use of HTTP Strict Transport Security (HSTS) ensures that browsers only connect over HTTPS and protects against downgrade attacks. Certificate management needs to be reliable and automated, so it doesn't disrupt things / services. Beyond encryption, secure headers (Content-Security-Policy, X-Frame-Options, etc.) protect you from common web exploits.

OAuth and JWT

For delegated authorization, OAuth is a proven standard protocol that grants secure third-party access without sharing logins. This paradigm is extended for JSON Web Tokens (JWTs) where the user claims are embedded in a stateless, cryptographically signed token. These tokens simplify microservices interactions by removing redundant server-side lookups. However, the key to preventing malicious use is effective token management strategies, including token revocation and token expiration.

Threat Modeling

Frameworks that are systematic in nature for threat modeling, like STRIDE, promote developers to discover, classify, assess, and mitigate potential threats at architectural designs. Helping teams catching and mitigating vulnerabilities early by malicious simulation of activities such as spoofing, tampering, repudiation, information disclosure, denial of service and privilege escalation. These efforts may also steer test automation and help communicate security trade-offs with stakeholders.

Regulatory Compliance

There are strict regulations like GDPR, or General Data Protection Regulation, HIPAA and PCI-DSS (Payment Card Industry Data Security Standard) that provide specific guidelines around the handling, retention and reporting of data. Most secure and compliant systems require encryption at rest, access logs, strong audit capabilities, and agreed-upon incident response plans. Although compliance may create extra friction, it also builds trust by demonstrating organizational due diligence and responsibility in data stewardship.

Balancing Security and User Experience Excessively strict security processes may discourage users or impede effective workflows. A risk based approach helps calibrate controls to context, imposing stronger safeguards, when handling sensitive data, in dubious situations or circumstances, where breaches may happen, and reducing burdens at routine usage. Part of thorough planning is ensuring that systems are User friendly while meeting relevant security expectation.

7. Implementation Challenges in Distributed Systems and Microservices

Microservices and distributed architectures enable teams to deploy features quickly, scale them granularly, and improve resilience. They do however exacerbate issues around authorization, data consistency, error handling, and observability that makes them anything but trivial to implement or manage.

Authorization Overheads in High-Traffic Environments

In microservices ecosystems, where each service may call multiple peer services, repeating authorization checks is required. If these checks involve a remote policy server lookup or extensive rule evaluation, the checks can quickly become a bottleneck. Using carefully cached tokens or colocating policy engines with microservices can reduce the latency. Likewise, asynchronous or event-driven approaches for authorization assist in distributing computational workloads.

Caching Strategies and Data Invalidation Although caching also increases performance by minimizing direct calls to the database, stale data is an ever-present threat. Cache invalidation is one of the great challenges of distributed systems, and without a coordinated approach to this task, services can end up in inconsistent states. This can be mitigated with event-driven invalidation, time-based expiration, or potentially a library used more
transparently. It's important to robustly test that data updates and cache sync will not get out of sync in certain scenarios for stable operations, or that if they do, that is okay.

Error Handling, Resiliency and Observability Partial failures are common in distributed systems. Circuit breakers, retry patterns, and bulkhead patterns protect against cascading failures. Full observability needs logs, metrics, and distributed traces that give developers insight into a transaction that leaps across the services. Prometheus (metrics) & Jaeger (tracing) these tools provide almost real-time insight into the health of the system, allowing us to detect anomalies and performance regression quickly.

Data Consistency and The CAP Theorem According to the CAP theorem (Consistency, Availability, Partition Tolerance), distributed systems must make trade-offs in the presence of network partitions. In fact, many architectures use eventual consistency or sagas (orchestrated distributed transactions) for business operations that span across multiple services. Though they address data clashes, these solutions also increase design complexity with the need to tightly knit business logic and service boundaries.

Organizational and Cultural Factors

Moving away from monoliths to distributed systems requires reimagining development processes, communication protocols, and operational mindsets. In fact, some of the codas which uphold the watermark of DevOps principles such as blurring the lines between development and operations teams are often the very fiber for seamless continuous integration, high throughput testing, and successful deployments. If no organizational culture permitting for iterative improvement is present at your organization microservices transformations will stall or underdeliver.

8. Real-World Use Cases, Best Practices, and Emerging Trends

Use Cases

Transitioning an E-Commerce Application to Microservices

An online retail website that moved from monolithic architecture to microservices by exposing HTTP/2-based REST endpoints had a significant reduction in response latency. They also used a combination of role-based and policy-based controls to manage granular merchant permissions. And while distributed caching soaked up the heavy traffic spikes around the holidays, strategic instrumentation uncovered an impressive 30% average response time improvement. Developer autonomy was enhanced as well, which reduced release cycles from weeks to days.

For Regulated Data: A Healthcare Application A healthcare solution handling patient records faced stringent data privacy and access control demands under HIPAA. The engineering team introduced PBAC for dynamic user attributes and integrated OAuth-based workflows to ensure fine-grained consent and delegation. They maintained rigorous compliance standards while providing real-time updates to patients by implementing strong SSL
settings, ongoing threat modeling, and a global event- driven notification pipeline.

Best Practices

Organizations find incremental strategies to be more beneficial than sweeping changes all at once. Phased rollouts of new functionalities (whether they are new systems, advanced protocols or access controls), and breaking existing systems into modularized applications, provide teams an opportunity to validate assumptions and address bottlenecks early on. To prevent adding faulty features deployed in production systems I also introduce automation from CI/CD pipelines, so every production feature must be robust against performance regressions and security issues. Centralized policy management can apply consistently across microservices to prevent permission inconsistency, and a comprehensive observability stack logs, metrics, and distributed traces allow for proactive debugging.

Emerging Trends

- **Zero-Trust Architectures:** Zero-trust frameworks operate under the assumption that no trust exists by default in network boundaries and strict identity verification must be enforced at each and every step. This method relies a lot on micro-segmentation and continuous authentication and is more adapted to the modern distributed ecosystem.
- **Edge Computing:** The offload of computation and data storage, closer to the appropriate edge nodes, enables reduction of latency in IoT and high-volume analytics scenarios, and also offloads the core data center. However, it brings up complex caching, synchronization and security models for geographically distributed nodes.
- **Serverless Computing:** Function-as-a- Service (FaaS) paradigms allow developers to minimize operational overhead by letting cloud providers dynamically manage infrastructure resources. Nevertheless, serverless methods require considerate design for caching, ephemeral storage, and cold-start latencies.
- **AI-Enhanced Security and Policy:** Machine learning techniques are used more regularly to identify abnormal traffic or malicious action. Smart policy engines might adapt to changing conditions across the organization, adjusting rules about who (or who may not) to access resources or even shaping microservices routing on the fly.

Combining these strategies, organizations can move closer to building resilient infrastructures that can effectively incorporate evolving communication pipelines and authorization frameworks while remaining robust and agile within dynamic markets.

9. Conclusion and References

Building resilient, future-ready systems requires a holistic approach covering areas like architectural patterns, scalability strategies, protocol buffing and access management. From monoliths to microservices, evolving protocols from HTTP/2 to HTTP/3, the modern system design fabric consistently adapts to the new performance and security constrains introduced by its components. Similarly, sophisticated authorization systems, including anything from RBAC to PBAC enforce detailed-grained controls that protect data integrity, user privacy, and compliance requirements.

Looking forward, increasing adoption of zero-trust approaches, edge computing paradigms and intelligent orchestration will likely reshape the structural and operational norms of systems design. Automation, real-time observability, and policy- driven governance are rapidly emerging as critical pillars. As things get more complex, it is important for software architects and developers to continue to respect basic design principles. Make sure you build on solid architectural principles and choose and hone protocols that are right for you and implement access control as carefully as possible to find the right balance between security and usability.
By maintaining this holistic discipline, bringing together architecture, performance engineering, communication protocols, and strong authorization, organizations can have a platform that consistently fit stakeholder expectations, adapts to changing workloads, and helps drive continuous compliance. Essentially, the road to construct robust, thriving systems hinges upon a blend of bedrock principles and future-ready strategies, guaranteeing flexibility and endurance in a perpetually evolving technological landscape.

References

1. Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2013). *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons.
2. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192
3. Cristea, V., Pop, F., Dobre, C., Costan, A. (2011). Distributed Architectures for Event- Based Systems. In: Helmer, S., Poulovassilis, A., Xhafa, F. (eds) Reasoning in Event-Based Distributed Systems. Studies in Computational Intelligence, vol 347. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-19724- 6_2
4. Ibrahim, Ibrahim Mahmood, Siddeeq Y. Ameen, Hajar Maseeh Yasin, Naaman Omar, Shakir Fattah Kak, Zryan Najat Rashid, Azar Abid Salih, Nareen O. M. Salim, and Dindar Mikaeel Ahmed. 2021. "Web Server Performance Improvement Using Dynamic Load Balancing Techniques: A Review". *Asian Journal of Research in Computer Science* 10 (1):47-62. https://doi.org/10.9734/ajrcos/2021/v10i130 234
5. Toby Teorey, Sam Lightstone, Tom Nadeau. 2006. "Database Modelling and Design": *An Example of Logical Database Design:* 139-145. https://doi.org/10.1016/B978-0-12-685352-0.X5000-9
6. M. Trevisan, D. Giordano, I. Drago and A. S. Khatouni, "Measuring HTTP/3: Adoption and Performance," 2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet), Ibiza, Spain, 2021, pp. 1-8, doi: 10.1109/MedComNet52149.2021.9501274.
7. Wang, V., Salim, F., Moskovits, P. (2013). The WebSocket API. In: The Definitive Guide to HTML5 WebSocket. Apress, Berkeley, CA. https://doi.org/10.1007/978- 1-4302-4741-8_2
8. Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17). Association for Computing Machinery, New York, NY, USA, Article 27, 1–36. https://doi.org/10.1145/3147704.3147734

9. Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17). Association for Computing Machinery, New York, NY, USA, Article 27, 1–36. https://doi.org/10.1145/3147704.3147734
10. Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17). Association for Computing Machinery, New York, NY, USA, Article 27, 1–36. https://doi.org/10.1145/3147704.3147734
11. Jennifer Bayuk, Data-centric security, Computer Fraud & Security, Volume 2009, Issue 3, 2009, Pages 7-11, ISSN 1361-3723, https://doi.org/10.1016/S1361- 3723(09)70032-6.
12. M. S. Khan, A. W. Khan, F. Khan, M. A. Khan and T. K. Whangbo, "Critical Challenges to Adopt DevOps Culture in Software Organizations: A Systematic Review," in IEEE Access, vol. 10, pp. 14339-14349, 2022, doi: 10.1109/ACCESS.2022.3145970.
13. Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 2 (June 2002), 51–59.https://doi.org/10.1145/564585.564601