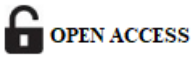
International Journal of Architecture (IJA)
Volume 7, Issue 2, July-December 2021, pp. 11-22. Article ID: IJA_07_02_002
Available online at https://iaeme.com/Home/issue/IJA?Volume=7&Issue=2
ISSN Online: 2251-5799, Journal ID: 3120-0508
Impact Factor (2021): 13.74 (Based on Google Scholar Citation)
DOI: https://doi.org/10.34218/IJA_07_02_002



DESIGNING FAULT-TOLERANT DISTRIBUTED SYSTEMS FOR HIGH-AVAILABILITY CONSUMER INTERNET PLATFORMS

Venkatramana Reddy Panyala
Production Engineer, Yahoo, USA.

ABSTRACT

One of the most difficult tasks of software engineers today is arguably the development of fault-tolerant distributed systems of internet consumer platforms. Social media and e-commerce services, video-streaming applications, and fintech applications are all types of internet consumer platforms that must be 24/7 with millions of simultaneous users living in geographically dispersed areas. Poor performance or an outage will lead to losses, damaged reputation, and churn. This paper gives a detailed discussion of architectures that have been adopted to guarantee high availability and fault tolerance of the distributed systems. It describes the theory underlying it in the context of the CAP theorem and consistency models and takes into account practical solutions such as redundancy, consensus protocols, circuit breakers, service mesh technologies, replication, chaos engineering and CI/CD release pipelines that can be used to minimize the blast radius of deployments. This paper, based on the existing state-of-art research and best practices, proposes a vivid image to professionals interested in constructing highly available internet infrastructure.

Keywords: Fault tolerance, Distributed systems, High availability, Consensus, Circuit Breaker, Chaos Engineering, Microservices

Cite this Article: Venkatramana Reddy Panyala. (2021). Designing Fault-Tolerant Distributed Systems for High-Availability Consumer Internet Platforms. *International Journal of Architecture (IJA)*, 7 (2), 11-22.
https://iaeme.com/Home/issue/IJA?Volume=7&Issue=2

1. Introduction

The increasing population of internet connected users and mobile devices in the past decade has generated high expectations towards reliability of internet services. Hundreds of millions of active users use top-tier providers such as Amazon, Google, Netflix, and Alibaba, and service level objectives (SLOs) of more than 99.99% availability, i.e. less than 52 minutes of permitted downtime per year. To achieve such an expectation, it involves a well-thought architecture methods, which consider anticipation, faults detection and recovery in an automated fashion.

Distributed computing systems tend to be failure-prone due to such factors as network partitions, hardware faults, software flaws, and unexpected bursts in traffic which may render their operation unavailable. The area of research that is concerned with the above question is fault tolerance, how the computer programs can provide correct, degraded, or partial service despite the described above problems. One of the spheres that have been especially intrigued with this issue is consumer internet platforms because of the character of the user expectation in the industry.

Contemporary consumer internet applications, including those provided by Amazon, Netflix, and Google, cater to millions to billions of users at any given time, and their services must be constantly accessible, reactive and stable. In large-scale systems, failures of systems are not unusual occurrences but are normal owing to hardware mistakes, software, network problems, and unpredictable workloads. Subsequently, the creation of fault-tolerant distributed systems has become a requirement in order to provide uninterrupted service provision and user confidence. High availability is no longer a competitive advantage, but it is a minimum requirement.

Fault tolerance in distributed systems is the capacity of a system to operate properly even upon the failure of a few of its elements. This can be accomplished through architectural techniques like redundancy, replication, load balancing and automated failure recovery. Such methods as data replication in more than one node, application of distributed consensus algorithms and application of failover mechanisms are used to make sure that there is no single point of failure that can affect the whole system. Also, contemporary solutions are based on cloud-native, microservice-based architectures, and container orchestrators to enhance resiliency and scalability.

The present paper is devoted to the design principles, architectural patterns, and technologies that facilitate the fault tolerance of distributed systems designed to be used in highly available consumer internet platforms. It discusses how top organizations embrace dynamic monitoring, self-healing and smart traffic routing to overcome failures. Moreover, it discusses the trade-offs of consistency, availability and partition tolerance, as defined by the CAP theorem and the impacts of such trade-offs on practical system design choices.Fault-tolerant measures that can be understood and applied help organizations to develop resilient systems that can gracefully deal with failure so that user experiences remain smooth even when circumstances are unfavorable.

2. Background and Theoretical Foundations

Brewer suggested the CAP theorem which was mathematically proved by Gilbert and Lynch in 2002. According to the theorem, a distributed data storage system cannot ensure three

significant properties simultaneously: consistency (read operations receive the latest write), availability (read operations cannot fail) and partition tolerance (the system can continue to operate even in case any messages are lost across the network) [1]. In the case of consumer internet services, which must operate in a low-quality network environment, partition tolerance cannot be compromised, and designers must take due care of consistency versus availability.

Eventual consistency is a weakening of the notion of consistency in real-world systems: eventually all nodes will be consistent and identical in their state assuming no other updates are made. Amazon DynamoDB and Apache Cassandra are two well-known examples of distributed systems that have availability and partition tolerance, with tunable consistency choices. Instead, strong consistency needs to apply certain coordination algorithms, such as Paxos or Raft, hence, it is not fully available.

The availability of systems is typically measured using a concept known as 'nines' where systems providing availability of 99.9% (3 nines) are expected to have less than 8.76 hours of downtime per year, whereas systems with 99.999% (5 nines) availability can afford no more than 5.26 minutes of downtime each year. SLAs are availability agreements with the customers, and SLOs are the internal targets that must be exceeded so as to ensure that there is a sufficient margin to meet the agreed SLAs. SLIs, in their turn, mean the metric-based assessment of the meeting of SLOs [2].

The traditional SLIs incorporate the measures like the number of requests served successfully over the number of requests that the system receives within a measured time. The modern approach towards availability considers composite SLIs that take into account the impact of latency on system availability besides correctness, whereby a response received after an unreasonable period of time will be considered a failure in meeting the SLO.

There are different categories of failures that can be categorized by the systems engineers. The former is referred to as a crash-stop failure where the part ceases to operate and becomes incapable of communication. The second is a crash-recovery failure, in which the component might fail, but then restart with potentially outdated state. The third kind is omission failure which occurs when a particular component is unable to transmit or receive particular messages. Byzantine faults are the most generic and severe type of failure, in which the component is able to do anything, even send contradicting messages to two or more components.

The paper by S. Liu (2021) is a thorough review of fault tolerance in distributed optimization and machine learning systems and is concerned with the ability of large-scale distributed environments to respond to node failures, communication delays, and data inconsistencies. The research identifies important methods of redundancy, gradient coding and strong methods of aggregation to help in reliable model training in case of failure. It further describes the trade-offs between computational efficiency and fault tolerance and the importance of scalable and resilient algorithms in current distributed learning systems.[3]

J. Hellings and M. Sadoghi (2019) explore the fault-tolerant cluster-sending problem, a problem that has an issue of ensuring reliable transmission of data in distributed clusters. Their work suggests models to guarantee consistency and availability even in case of partial system failures. The research is relevant to enhance the reliability of communication by maximizing the delivery of the message which is essential to the integrity of the system in large-scale distributed systems.[4]

E. Taheri et al. (2021) present DeFT, an algorithmic routing algorithm with no deadlock and fault tolerance, that is optimized to work on 2.5D chiplet-based networks. Their design provides greater reliability at the network layer, eliminating deadlocks, and maintaining data flow even in the presence of faults. The work is especially important in the high-performance computing systems where effective and reliable interconnect communication is critical to the stability of the entire system.[5]

J. Xu et al. (2021) pay attention to the complexity of concurrency management of distributed systems with fault tolerance. The article discusses how to regulate the interactions between the multi-processes running simultaneously with ensuring consistency of the system in case of failures. It emphasizes the significance of the coordination models, the synchronisation methods, and the recovery methods in order to make sure that distributed operations are reliable and predictable in dynamic environments.[6]

K. K. Pattan (2021) introduces the notion of an AI-enhanced fault tolerance approach that applies to distributed systems by optimizing the replica management. The paper takes advantage of machine learning algorithms to dynamically control replicas to enhance the availability and performance of the system. The work illustrates the application of AI in improving resilience and adaptability in contemporary distributed architectures by integrating smart decision-making within the conventional fault-tolerance approaches.[7]

3. System Architecture for High Availability

3.1 Overview

An architecture that is fault tolerant should incorporate redundancy at all levels, namely compute, network, storage, and software. The multi-layer architecture depicted in Figure 1 below can serve as a good example. The requests will be forced to go through a load balancer spread out to various locations and using DNS technology, through the API gateways, through stateless microservices and message buses and then to a redundant database.

3.2 Traffic and load balancing.

The major mitigation of the threat of local outages is load balancing. Contemporary systems have a number of layers of load balancing: global load balancing through DNS balances traffic between geographic locations, hardware or software load balancers balance traffic among availability zones, and service meshes balance traffic among individual service instances. DNS-based failover is based on DNS records which offer failover with low TTLs where requests are redirected to new geographical locations within tens of seconds or minutes of a failure event taking place [8].

Consistent hashing is popularly used for distributed caching systems and sharded databases to reduce the remapping of keys if there are changes to the underlying infrastructure due to growth or a node outage. Virtual nodes (vnodes) are more load balanced and fault tolerant in that each real node has multiple non-adjacent positions on the hash ring.

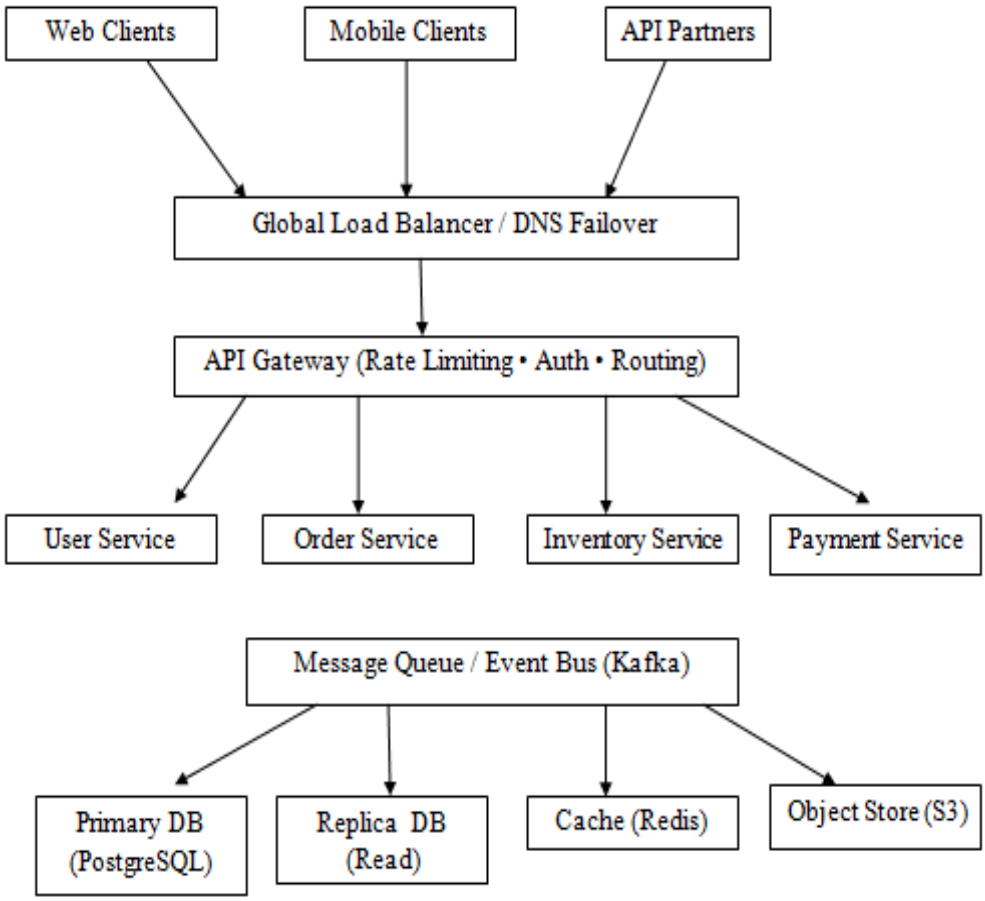


Figure 1: Fault-Tolerant Distributed System Architecture for Consumer Internet Platforms

3.3 Stateless Service Design

Services that are stateless, i.e. do not store any information on a session state between calls, are far easier to scale and recover failures. Any type of request can be served by every stateless service, which can easily failover with the load balancer. The session state information is moved outside to distributed caches like Redis/Memcached or tokenizing methods like JWT tokens.

The design practice of the twelve-factor app, which is a popular way of developing cloud-native, involves a stateless service as well as other characteristics like configuration externalization, disposability, and horizontal scaling [9]. Containerization systems like Docker and orchestration platforms like Kubernetes allow executing this design philosophy.

3.4 Service Mesh

A service mesh implements fault tolerance mechanisms like retries, circuit breakers, mutual TLS, load balancing, and observability at an infrastructure level that do not need any changes to the applications' source code. The service mesh can be described as a proxy sidecar (primarily Envoy), which is deployed with each service instance that oversees all the communication between services.Service meshes take care of implementing retry rules, timeout periods, and circuit breaker values. Otherwise, in case of one of the services failing, there might be a scenario of another service calling it and collapsing in a domino effect. Service meshes offer very granular observability capabilities via distributed tracing.

4. Fault detection and recovery mechanisms.

4.1 Health Checks and Heartbeats

The foundation of a fault detection solution of distributed computing systems is based on automated health checks. Service endpoints, such as an HTTP request on /health or /readiness, are used to make health checks available. Kubernetes uses liveness probes to decide when a container should be restarted, and readiness probes to decide when a container should accept traffic. When the service is live, but not ready (i.e. it is warming up dependencies or some other reason), it will not be added to the load balancer, but will not be killed.

Gossip-based failure detection schemes such as Cassandra and Serf spread the load of monitoring out across all nodes in the cluster. Nodes sample other nodes at random, and exchange state information, so that failure detection and propagation can be done without a centralized monitoring tool. SWIM (Scalable Weakly-consistent Infection-style Membership) ensures detection time and false positives limitations [10].

4.2 Circuit Breaker Pattern

Circuit Breaker pattern, presented by Michael Nygard and later officially introduced in the Hystrix framework of Netflix, helps avert cascading failures by observing the calls to the downstream services and halting any communication with a faulty one when detected [11]. The circuit breaker goes through three different states, Closed, where the service operates normally and allows all requests to go through, Open, where the circuit breaker fails after a fault threshold is reached, and Half-Open, where a probe request can be sent out to check for recovery.

Circuit breakers should be set accordingly to ensure that settings are not too high or too low to the extent that they will interfere with customer experience. Here, an overly large or overly small fault threshold will cause the breaker to either disrupt normal operations or fail to aid in preventing cascading failures. Sliding window algorithms are used to cope with the variability in traffic.

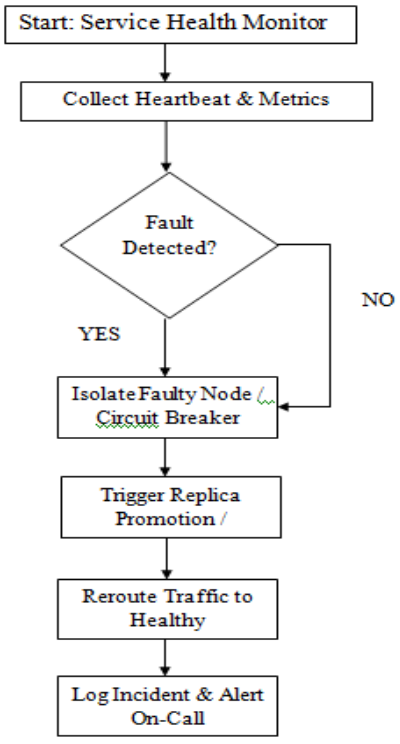


Figure 2: Fault Detection and Automated Recovery Flowchart

4.3 Retry Strategies and Timeout Budgets.

A retry is a straightforward yet effective approach to recover from temporary failures. Nonetheless, inadequate retry strategies might lead to additional load on the failed downstream system, causing a failure and hence, a retry storm. A retry is recommended with an exponential backoff algorithm as per best practice that introduces random jitter to the retry time to prevent desynchronization [12].

The timeouts are supposed to restrict the number of seconds that are spent on waiting to have all the downstream callswithin the capacity of a given request satisfied. The caller can set no timeouts and an upstream service running slowly can consume the thread or connection pools of its upstream caller, eventually causing its failure. As an example, gRPC has a deadline propagation feature that enables deadlines to be propagated to all call endpoints to abort incomplete requests when needed.

4.4 Bulkhead Pattern

Bulkhead Pattern: This pattern is used to segregate the resources like thread pool, connection pool, semaphore based on different types of downstream dependencies. The resources allocated to any downstream dependency are limited; hence, in case one of the downstream dependencies malfunction or work at a slow pace, it cannot possibly exhaust its resources and the system cannot take the requests by other healthy downstream dependencies.

5. Data Consistency and Data Replication.

5.1 Replication Strategies

The solution to creating fault-tolerant storage is data duplication. The fundamental decision is between synchronous data replication which ensures that no data is lost in case the primary fails but increases the write latency and asynchronous data replication which reduces the write latency at the expense of losing data that may have been committed before the replication of data is finished. Several commercial database systems offer configurable durability parameters to assist operators to make tradeoffs.

The most popular form of data replication that is employed by relational database systems is Leader/Follower (Primary/Replica) Data Replication. The leader is the recipient of write requests and copies it to the followers who do the data reading. The failover is automatic in that in an event of failure of the leader, a new leader is automatically promoted among the followers.

5.2 Consensus Algorithms

A distributed system’s leader election and coordination are based on consensus, which means that all non-failure nodes should reach agreement on a certain value. The family of distributed consensus protocols that were invented by Leslie Lamport in 1989 is called Paxos [13]. However, as Paxos is highly difficult to get right, Raft has been created as the consensus algorithm oriented towards simplicity to understand and implement [14].Raft consensus has three loosely coupled tasks: leadership election, replication of log entries as well as consistency guarantees. The leader of the Raft is chosen between the nodes and accepts the requests of clients to the leader, puts them into the log, and sends them to other nodes. Entry can be considered committed when acknowledged by a majority of nodes. As a result, when a small group of nodes fails, this does not affect durability.

5.3 Multi-Region Replication

Internet services that are consumer facing tend to be available in most geographic locations to minimize latency to their geographically distributed user base, and tend to be resistant to regional problems like natural disasters, power outages, and regional failures in cloud provider infrastructure. Multi region replication is difficult due to large latencies which exist between various regions (between 50 ms and 150 ms) and a synchronous replication is not practical.

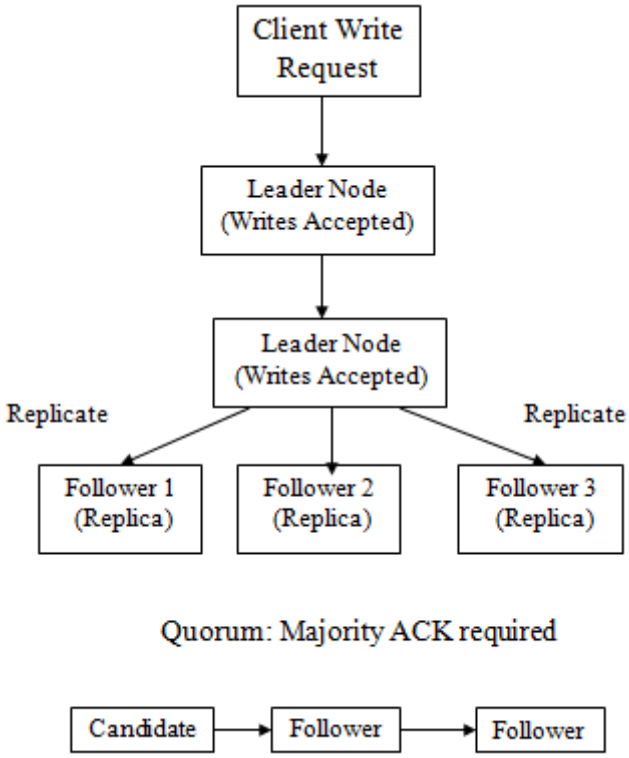


Figure 3: Leader-Follower Replication Topology and Raft Consensus State Machine

Active-active architecture gives the user the ability to update data on any part, but methods to recognize and duplicate any conflicts in the event of multiple writes have to be put in place. Methods are vector clock, Conflict-free Replicated Data Type (CRDT) algorithms, and last writer wins logic. Google Spanner applies its TrueTime API, which is based on atomic clocks and GPS devices, to ensure external consistency among replicas geographically distributed around the world [15].

5.4 Caching Strategies

Redis and Memcached distributed caches reduce the load on persistent storage, and offer lower response times, but introduce issues of cache invalidation as far as fault tolerance is concerned. Wrong or old cache information can lead to inconsistency to the users when there is a change in the main storage. Potential ways of revoking the cache include TTL cache invalidation, write-through cache and cache-aside.

6. Automation of deployments and chaos engineering.

6.1 Fault-Tolerant Deployments CI/CD.

The CI/CD pipeline is an automated process of creating, testing, and releasing software by automation to reduce the likelihoods of introducing bugs into the system due to errors by the

developers when manually implementing code. An example of a CI/CD pipeline is presented in Figure 4; When deploying high availability services, the following strategies can be used: Rolling Update -a portion of the traffic between service instances is replaced with another part; Blue-green Deployment -there are two identical environments, and traffic is swapped between them immediately; Canary Release -a portion of traffic is redirected to the new instance and tested as shown in Figure 3. [16].

6.2 Chaos Engineering

Chaos engineering is the field where deliberate failures are introduced into a production or production-like environment to ensure that the resilience measures of the system perform as expected. The concept was initially brought forth by Netflix through creation of the Chaos Monkey software in 2011. Chaos engineering has now developed and has literature and tools of its own [17]. The Chaos Monkey kills off randomly chosen instances in a production environment, which makes Netflix engineers develop their services in such a way that unexpected instance deaths would not have a significant impact as shown in Figure 4.

Modern chaos engineering frameworks like Chaos Kong (by Netflix) to introduce failure on a region-wide basis, Gremlin, and Litmus offer more fault injection features such as adding network latency, CPU or memory starvation, disabled disk I/O performance, and dependency blackholing. The experiments of chaos should always include a clear hypothesis regarding the steady states, small blast radius and quick rollbacks. Experiments performed using chaos engineering provide insights on how to improve the resilience measures of the system and its runbooks.

6.3 Observability

It demands sufficient observability - the ability to determine the internal condition of the system given its output - to effectively detect and manage faults in the system. The key elements of observability, metrics, logs, and traces, offer different views on the operation of the system. The metrics can be used to identify an anomaly in the system in a short time because they quantify the parameters, such as requests per second, error rate, and so on. Distributed traces enable tracking the sequence of events concerning particular requests that are handled by multiple microservices. When investigating incidents, logs can provide information that can be essential.[18]

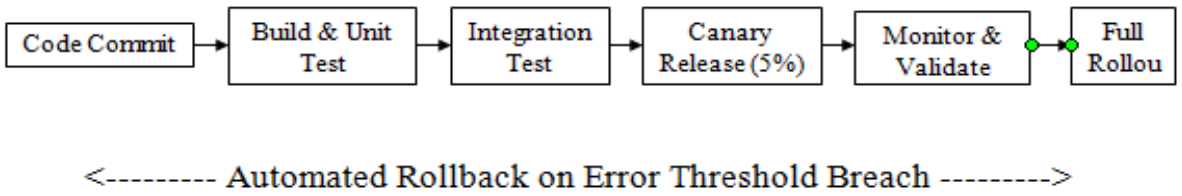


Figure 4: CI/CD Pipeline with Canary Deployment and Automated Rollback

USER and RED techniques (Utilization, Saturation, Errors and Rate, Errors, Duration) enable the identification of bottlenecks in the underlying infrastructure resources, and in the applications/services themselves. Creating the systems alerts, it is necessary to pay attention to the SLO burn rate because the former can give a precise method of predicting the upcoming SLO violations.

6.4 Runbooks and Incident Management.

Technical strategies are not enough to attain operational resilience because they need to have mechanisms to detect and scale up problems as well as deal with them. Critical elements are on-call shifts, runbooks that properly address the known cases of failure, and the post-mortem analysis that does not place anyone in the hot seat. Google has a methodology titled Site Reliability Engineering (SRE), incorporating the notion of an error budget, which enables a person to measure acceptable amounts of system unavailability.

7. Discussion and Tradeoffs

7.1 Complexity vs. Resilience

The introduction of every one of the resilient entities into the distributed system introduces complexity and new areas of failure. Failure of circuit breaker elements may cause undue system shutdowns. Thundering herds can occur due to incoherent retry systems. Latency is present in consensus algorithms. To decide whether a resilient element is worth the extra complexity, designers ought to calculate the cost-benefit ratio of each resilient element, particularly when the failure associated with it is uncommon or not disastrous.[19]

7.2 Consistency vs. Availability

CAP theory states that there is a conflict between consistency and availability in the case of partitions within the network. Trade-off between the two properties is appropriate depending on the nature of workload in the case of consumer Internet applications. In the case of financial transactions, strong consistency is required to prevent such problems as double spending and incorrect accounting. In the case of social media feeds, eventual consistency would not be an issue since the end user is not much concerned with the latency in delivering the content to them.[20]

7.3 Cost Considerations

Redundant architecture by definition means that there is redundancy in components which do not serve any traffic during normal operations. The expense of maintaining additional replica servers, moving data between regions and implementing services across regions can be very high. The business ought to weigh the cost of having such redundancy and the cost of going down considering not just the direct financial effect but also the brand damage. Spot and preemptive cases can be employed to cut down on the computing expenses.

7.4 Limitations and Open Challenges.

Although much advancement has been made in fault-tolerant distributed computing, there are many outstanding problems as well. The diagnosis and correction of heisenbugs in a distributed computing environment pose an incredibly challenging task. Modeling languages such as TLA+ and Alloy can be used to model the behaviour of a system and identify any flaws in the design of a system before it is implemented, but industry has not yet adopted them due to the cognitive cost of learning them. The noted machine learning methods in detecting anomalies are still young.[16][17]

8. Conclusion

Consumer internet platform distributed systems offering fault tolerance are some of the most challenging engineering challenges that we have ever faced. Their theoretical background has been discussed in the CAP theorem, consistency models and fault taxonomy and practical

solutions such as redundancy, load balancing, circuit breaker, consensus-based replication, chaos engineering, and CI/CD automation. With the introduction of microservices architecture, container orchestration, service mesh, and cloud-native infrastructure, it has become easier to implement such patterns, and thus, availability became accessible to more people.

Building fault-tolerant distributed systems encompasses proper architecture design and a culture of learning through the mistakes made in the past, developing systems to monitor the system better, and ensuring regular testing of the system through chaos engineering. Numerous new fault-tolerance solutions have been suggested in recent years, such as self-healing systems, capacity management based on AI, and formal verification. The future research and development in the area would focus on the creation of quantitative means of the resilience level of distributed systems, automation of the creation of fault-tolerant systems based on high-level system reliability requirements, and the use of formal techniques to verify distributed consensus algorithms.

References

[1] B. Burns et al."Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70–93, 2016.

[2] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA: O'Reilly Media, 2016.

[3] S. Liu, A Survey on Fault-Tolerance in Distributed Optimization and Machine Learning, arXiv preprint arXiv:2106.08545, pp. 1–20, 2021.

[4] J. Hellings and M. Sadoghi, The Fault-Tolerant Cluster-Sending Problem, arXiv preprint arXiv:1908.01455, pp. 1–18, 2019..

[5] E. Taheri, S. Pasricha, and M. Nikdast, DeFT: A Deadlock-Free and Fault-Tolerant Routing Algorithm for 2.5D Chiplet Networks, arXiv preprint arXiv:2112.09234, pp. 1–15, 2021.

[6] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, and A. F. Zorzo, Supporting and Controlling Complex Concurrency in Fault-Tolerant Distributed Systems, arXiv preprint arXiv:2111.06339, pp. 1–18, 2021.

[7] K. K. Pattan, Optimizing Fault-Tolerance in Distributed Systems with AI-Augmented Replica Management, International Journal of Intelligent Systems and Applications in Engineering, vol. 9, no. 1, pp. 139–160, 2021.

[8] A. D. Kshemkalyani and M. Singhal, Distributed Computing: Principles, Algorithms, and Systems. Cambridge, UK: Cambridge University Press, 2008.

[9] A. Karve, T. Kichkaylo, G. Pacifici, A. Spreitzer, V. Steinder, A. Tantawi, and A. Youssef, "Dynamic application placement in enterprise data centers," in Proc. IEEE Int. Conf. Autonomic Computing, 2006, pp. 163–172.

[10] A. Wiggins, "The Twelve-Factor App," Heroku, San Francisco, CA, 2012. [Online]. Available: https://12factor.net/. [Accessed: Aug. 2021].

[11] C. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70–93, Jan. 2016.

[12] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in Proc. IEEE Int. Conf. Dependable Systems and Networks, 2002, pp. 303–312.

[13] M. T. Nygard, Release It! Design and Deploy Production-Ready Software. Raleigh, NC: Pragmatic Bookshelf, 2007.

[14] M. Brooker, "Exponential backoff and jitter," AWS Architecture Blog, 2015. [Online]. Available: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/.

[15] L. Lamport, "The part-time parliament," ACM Trans. Comput. Syst., vol. 16, no. 2, pp. 133–169, May 1998.

[16] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in Proc. USENIX Annual Technical Conf. (ATC), 2014, pp. 305–319.

[17] J. C. Corbett et al., "Spanner: Google's globally distributed database," ACM Trans. Comput. Syst., vol. 31, no. 3, pp. 1–22, Aug. 2013.

[18] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley, 2010.

[19] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," IEEE Software, vol. 33, no. 3, pp. 35–41, May–Jun. 2016.

[20] H. Meng, S. Zhang, Y. Liu, R. Nie, and Y. Zhang, "Localizing failure root causes in a microservice through causality inference," in Proc. IEEE Int. Symp. Reliable Distributed Systems (SRDS), 2020, pp. 227–236.