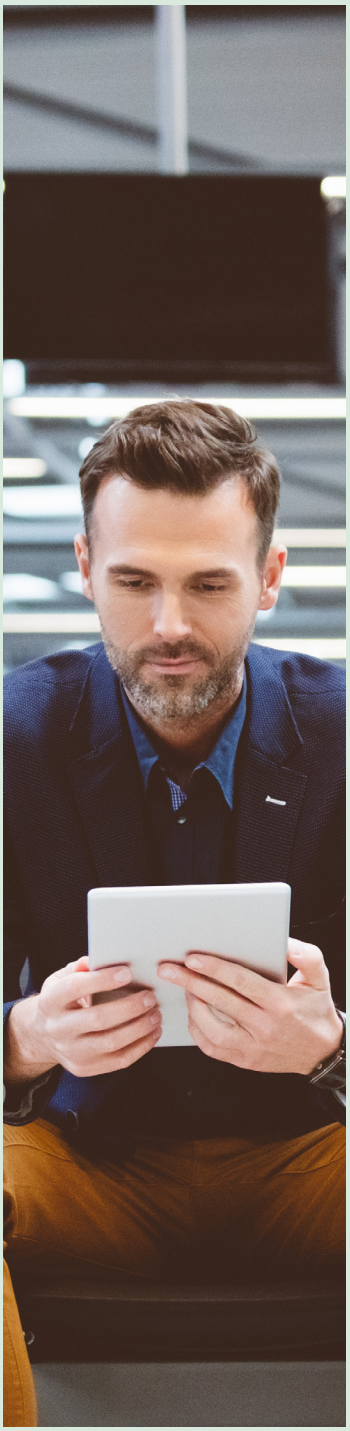
WHITE PAPER



RESILIENCE IN DISTRIBUTED SYSTEMS

Abstract

With the rapid increase in the number of internet users and frequent changes in consumer trends, traditional systems have no choice but to scale out, distribute, and decentralize. To give you an idea of the extend of scaling involved, Facebook and YouTube each would have had 40,000 to 45,000 hits from desktop users alone in the 5 seconds it took you to read this paragraph. [5].

With the peta and exa bytes of data being generated every day and the growing adoption of IoT, this scaling is only going to become more exponential and systems will need to scale out and depend on each other more than ever.

Distributed systems are driven by various Architecturally Significant Requirements (ASRs) [24] and one such ASR is resilience.

This paper is about the SPEAR (Software Performance Engineering and ARchitecture services) team's perspective on Application Resilience in distributed systems – what it means in simple terms, how to study it, the factors affecting it, and what patterns/best practices can help us in improving the same.

Infosys®
Navigate your next

Who is this document for?

This is presented from the perspective of a lead application designer or an architect, but there are some theories and methods for IT managers, developers, architects, tech leads, and program managers who are looking to understand and improve resilience in a distributed system.

Some basic definitions first.

What is a distributed system?

A distributed system is one where multiple components of a system are physically or logically separated and governed by common and component-specific requirements.

We say common because if you are designing a system to be 99.99% available you cannot have a crucial cache component in the system which is available only 97%. If that is the case, you need to have a trade-off in place to make sure the service availability is met in spite of only 97% availability of the cache component.

We say component specific because the design of the cache component is driven by speed and will be leveraging much of the primary storage disk, whereas the design of a persistent database will be to effectively manage secondary disk.

What is resilience?

Resilience of an application, in simple language, is the capability of the application to spring back to an acceptable operational condition after it faces an event affecting its operating conditions.

['capability' - what you have inside your systems to put them back in acceptable operating condition),

'event' – failure of responsibility of some module within the application or a failure of responsibility of some dependent system or a force majeure situation]

A 'failure of responsibility' or simply a failure could be a breach of SLA or some sort of agreement regarding an application. It could be big, like a failure of Amazon Route 53 services or a bug in the implementation of the Set interface which behaves unexpectedly under normal operating conditions.

The flipside of resilience is all about understanding and preparing for failures. So resilience can also be defined as the capability of the system to understand and manage failures occurring in the system effectively, with minimal disruption to business operations.

Why is this important? – Because it affects business, trust and lives.

A downtime of 1 minute at Amazon can result in a potential loss of USD 234,000 through their online channels alone. [6][7] [8] A technical glitch caused an outage on the New York Stock Exchange, leading to a drop in share prices and indexes.

Can you imagine an outage in a critical hospital system, air traffic control system, core trading platform or on a police or emergency contact system?

Some metrics like RPO (Recovery Point Objective), RTO (Recovery Time Objective), MTTR (Mean Time to Recovery), Number of failures/bugs detected in the system, SLA variance, etc., are some ways to measure resilience of a system. We will not be going into detail in this paper about the measurement and tools used but take a look at the various failures and patterns which can be used to improve the MTTR or RPO or RTO of a system.

Let's begin with a couple of modeling techniques needed for studying resilience – Flow and failure modeling.

Flow modeling

Traditionally we study the various flows in the system via use case analysis, control/ data flows, sequence diagrams etc., but will this linear and branching flow study really help?

Let's consider the control flow of this example: an online food ordering website lets the user select the food (A), check out the same (B), check coupons (C) via an external service, recalculate the checkout amount (B) if there are any discounts, select an address (D), external service for making payment (E1) which in turn automatically places an order (E2) via the restaurant API.

While this is more of a happy flow or ideal flow of a business operation, for resilience study we need to look at the alternate flows. Alternate flows are the control and data flows which are taken by the application if an unexpected behavior occurs. So to understand and design alternate flows we need to include the list of implicit services and dependencies at each step.

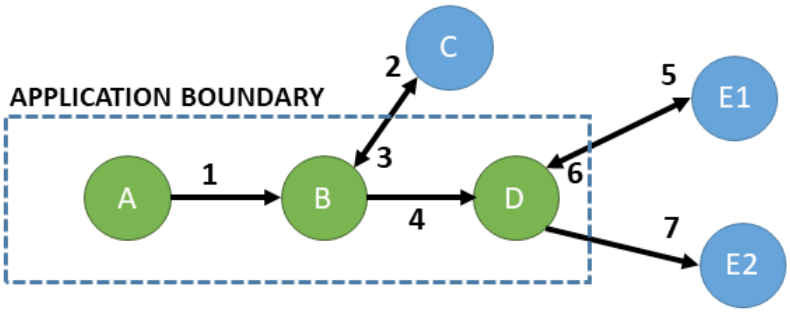


Figure 1 Control Flow of Order Booking

For example, let's look at the context of Step B – the check-out service. If we examine carefully, the check-out service, apart from coupon and address service, inherently banks on:

1. The DNS services.
2. The infrastructure services like
   a. Operating System
   b. CPU
   c. RAM
   d. Storage – Disk, File
   e. Network
   f. Any other infra components like routers, switches, firewalls, etc.,
3. The trust domain established by the security context (authentication, authorization, tokens etc.,)
4. Persistent Data Storage Services.
5. The state of the service (is it configured correctly and running correctly).
6. Technology of the service (Language, frameworks, dependent libraries, etc.)
7. Etc .

If we observe all the services some are shared between multiple components and some are not. Some are external and some are internal. Some are in the same layer as the application and some are not. Studying these dependencies and relations quickly takes the form of a reticulated mesh, which requires us to study the various events which can occur in each of these services, and then design and architect the application to handle the same.

With the current linear and branching model this is difficult and we should start using a multi-dimensional graphical model when trying to understand the control and data flows. This perspective is very important to design resilient systems.

Where to start – Cyclic and acyclic graphs, predictive models like PGM (Probabilistic Graph Modeling) etc., are some of the places to start. [21]

Failure modeling

Once we have a fair understanding of the multi-dimensional flow of control and data in the application, we need to perform a failure modeling exercise – where we list down the kinds of known failures and perform a what-if analysis and incorporate failure handling mechanisms in the system.

['what-if' – a what-if analysis is essentially an exercise to simulate a failure from the known list and check if there are mechanisms present in the system to handle it.

For example, what if there is a network failure? Do we have an alternate network or retry of services at regular intervals implemented?]

Figure (2) can help in understanding the various stages of failure handling in a system.

F_P – Fault Prevention - Capabilities present in the system to handle known failures that are expected to occur in the system.

F_D – Fault Detection – Capabilities present in the system to detect a fault. t_{fd} - time gap between the actual fault occurrence and the detection of the same.

F_I – Fault Isolation – Capability of the system to isolate the fault and treat it separately so that the normal functioning of the system is not affected. If there is no fault isolation mechanism, then the time taken to treat the fault would affect the normal processing of the system. t_{fi} – It is the time taken by the system to isolate the fault after it has been detected.

F_T – Fault Treatment – Capability present in the system to treat a fault once it has been detected and/or isolated. t_{ft} – It is the time taken to categorize the type of fault and treat it accordingly. For example, faults like a dependent system not reachable via network can be retried, but a fault like 'order message cannot be parsed' is something which will fail no matter how many times it is retried. Such faults cannot be treated and will be logged and failed gracefully.

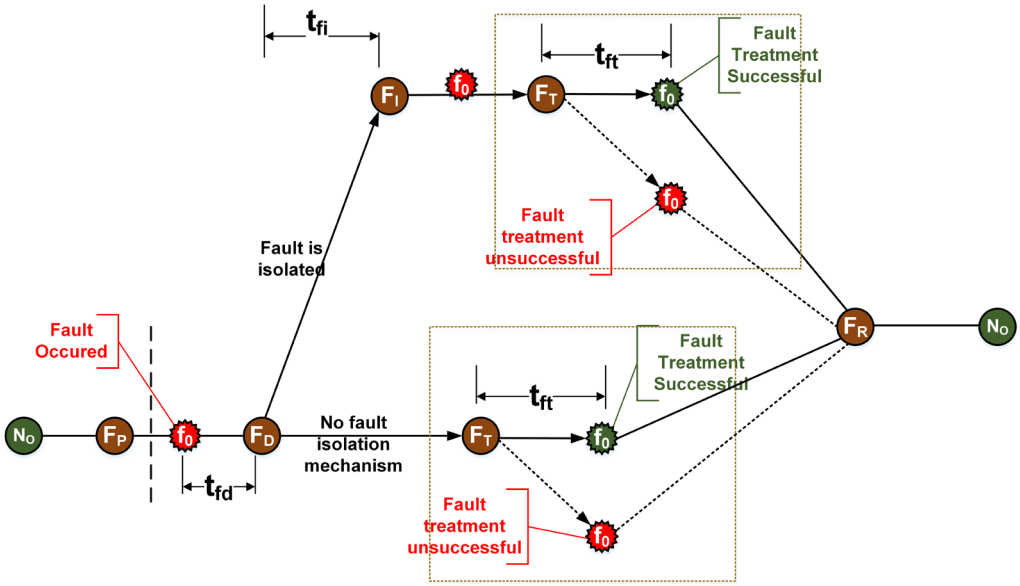


Figure 2 Failure Handling in a software system

F_R – Fault Recovery – Capability present in the system to recover from the fault after it has occurred and/or has been treated. If the fault cannot be treated or if the treatment of the fault affects the overall system state, then recovery mechanisms are needed to set the system back to operational state. The only goal of the recovery mechanism(s) is to get the system back up and running in normal operating mode.

In one of the architecture assessments we found that the key orchestrating system was running on a fault-tolerant box, but when we did a what-if analysis on the physical failure of the system, we found that the time taken to achieve the RTO and RPO was getting affected. As a result, we recommended adding an additional machine to handle box failure.

Where to start - FMEA (Failure Mode Effects Analysis), Event and Fault tree analysis, etc., are some places to start.

A highly resilient system should scientifically study each fault, categorize them based on severity and risk, have proper mechanisms in place to handle the system in case of a failure, and decrease the time taken to detect or treat a fault.

Key failure categories

Let us see some failure categories and check what mechanisms can be put in place to handle them efficiently.

I. Architecture and design Issues:

A poorly designed or architected distributed software can lead to various issues in the system today, tomorrow or any time till the end of life of the software. Remember, "Prevention is always better than cure".

Couple of examples are mentioned below:

1. Intelligent retries – "Intelligent Retries" decide the strategy to retry a failed operation. For example, retry with a timeout, logarithmic retry, arithmetic retry, geometric retries, priority retries, failover retries, etc.

Retries can make an application fault-tolerant, that is, if a module fails to connect to a service, it silently fails-over to another equivalent service which can fulfil the same request. This is called "idempotent failover" [4] and is employed in many stateless services.

2. Actor model [14] – Actor model is an alternate, highly concurrent and resilient model where actors never lock on a single resource like shared memory or shared object, instead talking to each other via messages. Akka is one such open source library which is built based on Actor model.

Architecture and design is a vast topic and to retain brevity, we recommend the below material for further reading on this.

Where to start? - Martin Fowler, Chaos engineering groups, EIP patterns by Gregor Hohpe and Bobby Woolf, highscalability.com, cloud resiliency patterns, Erlang's 'Let it Fail' [17], digital twins, bulkhead, 8 fallacies of distributed computing, stand-in processing, claim-check, throttling, sidecar, circuit breaker, fencing, redundancy, auto scaling, stateless architectures, Infrastructure as Code, reconciliation strategies, Policy centralization, Immutable infrastructure, event/service meshes and mesh based architectures (like MASA), traffic mirroring, Unbreakable pipelines, streaming patterns, Byzantine fault, consensus algorithms like raft or Paxos, CEP (Complex Event Processing), real-time and near real-time replication strategies, EDA (Event Driven Architecture), choreography patterns, distributed transaction handling patterns (like SAGA), data bus concepts from LinkedIn or MySQL or MongoDB, etc .

Daunted by this list? Don't be, and you are not alone. Understanding the requirements of the distributed software and making intelligent choice of algorithms and patterns will solve many forthcoming issues.

II. Failure risk analysis:

To build more confidence in the system it is important that a failure categorization is done accordingly and risk is assessed. The following quadrant-based segregation helps in understanding a distributed system.

Quadrant 1 - All failures, once they occur or are known failure scenarios, are studied and solutions are provided to handle them through various architectural choices and design mechanisms. The same needs to be built into various testing strategies to ensure that any changes made in the system are tested against these known failures and are handled accordingly. For example, a network failure is anticipated and recovery mechanisms are in place to handle the same.

Quadrant 2 – It is possible that some of the failures/bugs lie unearthed in the system and if found earlier in the system could have been isolated and handled better by adding the required design and processes. An unknown failure once unearthed becomes a known failure and after studying it, is moved into Quadrant 1.

For example, an unearthed design bug of infinite retry occurs only when the network goes down. Such scenarios could be missed in the usual testing methods. Using failure injection techniques, it is possible to study the failure and develop a solution for the same.

Quadrant 3 – In this case it is not only difficult to find the unknown failures in the system, as setting up such testing context can be difficult, it is possible that such failures have no known solutions that can be applied.

Quadrant 4 - This case is when we know some failures may occur but truly there is no solution which can be applied. Force majeure situations like acts of god, or any man-made disaster fall into this category and such risks are to be covered in contracts

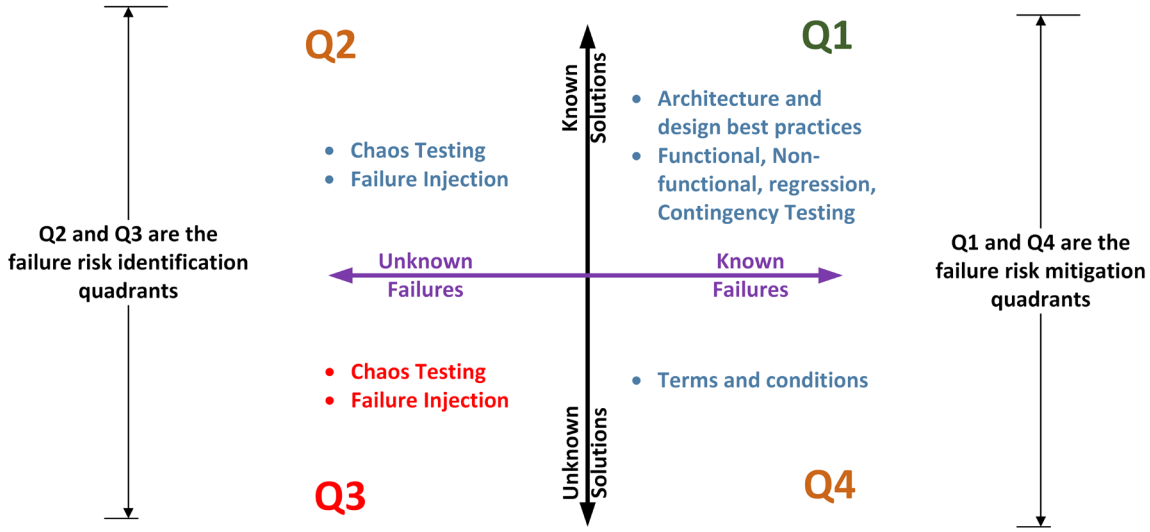


Figure 3 Known and Unknown Failure domains

III. Testing methods:

The application landscape has changed and the world is getting rapidly rewritten in code for tomorrow, even as we speak.

To keep up with this change, it is imperative that software testing methods have to be made more robust. One such method is Chaos testing [15] which is the process of testing failures in a distributed system by injecting known failures in the systems and observing the behavior.

For example, inject a JVM memory exception in a remote JVM instance and observing the response time of the system.

Some tools which can be used for chaos testing are – Chaos Monkey, Simian Army, gremlin etc.

Another way is to employ shift left testing strategy where the testing methods can start engaging very early in the cycle and act as a gateway to promote the code. Tool chains in the DevOps, CI/CD pipeline should integrate testing tools and practically all kinds of testing, from unit, performance, security, integration etc., can be executed and studied in a controlled environment and it can be decided if the software or the patch can go live. Tools like kubernetes, JMeter, Jenkins, Docker, Dynatrace etc., enable us to model newer testing approaches. Robust monitoring systems and tools today also allow us to detect and rollback the deployments made in case of an error found in production.

The testing tools and methods today are sophisticated and continually evolving. Embracing these new testing methods, tools, and processes is imperative for building a robust system.

IV. Deployment issues:

An intelligent deployment strategy can prevent issues from being caused due to software problems. Let's look at a couple of examples below:

1. Blue/Green deployment – The idea is to maintain two production environments

- blue and green - which are identical, with one of them being active (say blue). A new feature is deployed into the inactive environment (green) and tested and once it is 'Okay', then the traffic is switched from blue to green. The next feature is deployed into blue and traffic is routed through blue again. This ensures that at any point of time, the system is operable and no new features rupture the business operations after go live.

2. Canary releases – This strategy is similar to blue/green deployment but a release is only made to a subset of the infrastructure or a scalable but downsized identical deployment. Initially some % of the users are routed to the new deployment. Once the new feature is stable, all users will be routed to the new version.

For example, we all do see pop-ups in or mobile apps and web apps asking us if we want to try the new feature or the new beta version of a software. When we opt 'Yes', we will be routed to the new feature while users who opted 'No' will be routed to the old version. Practices like this ensures that feature roll outs don't introduce failures and even if they do, it doesn't stop the business from servicing the clients.

V. Alignment of builders:

It is important that builders of software are provided a formal initial alignment or induction process, which educates them on the nature of distributed systems, the likely problems they will face, and the common set of patterns which can be used to avoid those. It is natural that in a distributed system with many moving parts and many teams, people tend to fall into what I call as the 'farther from repercussion' problem. Since everyone is farther from the larger picture in an enterprise and there are always deployments and developments from multiple teams happening at the same time, it is possible that the team 'feels' that it is 'someone else's problem' and that 'I have no control on the final picture' or 'how do I know this will happen.' This needs to be addressed and avoided. It is important that programmers are evolved to developers. A developer is the one who sees beyond programming a specific task and puts together the multiple facets of what he writes.

To signify the importance of this, similar to the Y2K bug, a software bug which couldn't detect the year 2010 rendered up to 30 million debit cards in Germany unusable. [1][2]

A well aligned developer will be able to write software which will not have such issues.

VI. Architecturally Significant Requirements study

It is important that the architecturally significant requirements (ASR) derived during the initial requirement or 'solutioning' phase are studied carefully and related to logic and math. If there are specific SLAs and metrics which the software needs to meet, capturing them beforehand and asking the software builders to 'mind them' while developing will help.

For example, instead of just stating the system should be up 99.99%, it should be written in a more clearly articulated statement, like 'the users should be able to submit loans 99.99% of the time in the system and should be able to view the loan status response within 3 seconds of submitting the loan in the system.'

Also, by studying the various ASRs and performing a trade-off or a risk analysis matters, because the more we try to design a perfect failure-free system or resilient system, the more the cost goes up. Sometimes it is okay to let it fail and retry later. Such decisions can be taken only if there is a good study and clear articulation of the ASRs in the system.

VII. Adoption of DevOps:

A deployment should take care of ease of testing and provide the team a sense of ownership and the confidence that whatever they have developed will work well in production. DevOps processes ensure that the ride from development to deployment phase is not jittery. A lot of resources are available about the benefits of DevOps and usage of DevOps tools. Incorporate it as part of the software development-to-deployment cycle.

VIII. Security issues:

Security is a significant area where failures occur in the system. Cyber security has always been a battle of today with the vulnerabilities of yesterday.

Unfortunately, hackers today are more targeted and motivated than ever before.

Example: In January 2018, three major banks in Netherlands were victims of a DDoS attack which resulted in slow response times and service outages. [9]

Following are some of the preventive mechanisms that can be incorporated from an application and process perspective to avoid such a scenario–

a. In case of a security threat, isolation of application or services from the network should be possible.

b. Security checks, OWASP (Open Web Application Security Project) compliance checks, app scans, and customized security testing should be included in your DevOps pipeline.

c. SecOps should be inculcated as practice in the organization. As developers and leads deal with DevOps, Sysadmins and Security Architects should come together to define and practice SecOps in their organization.

For example, it could be decided that all OS and other software used should always be (n-1) version in production.

d. Adoption of enterprise-ready, proven, and a good community-backed Open Source Software (OSS) is one area where security has done well in terms of publishing quick patches for vulnerabilities as opposed to proprietary software which are in general influenced by product roadmaps and differences in support levels.

e. Block chain, AI, and ML are into the cybersecurity space and newer models

in fields like cognitive and semantic security help to identify new patterns which are difficult to manually detect.

For example, AI/ML algorithms detect intrusions and anomalies which are normally left undetected by rule configurations done by humans.

f. Distribution of the security context between multiple services and solutions is a proven strategy to avoid attacks on a singular context.

g. Last but not the least, the weakest part of any cyber security system are humans. Design systems which prevent this vulnerability. Example of a poor design is having a software storing a password in a format readable by humans or having a single-factor authentication for key use cases.

h. Multi-factor authentication, maker-checker process for critical business services, encryption, OTP mechanisms etc., are some of the methods which can avoid human vulnerabilities.

A Robust system should also handle existing and newer vulnerabilities (sites like https://oval.cisecurity.org/, https://cve.mitre.org/cve/ and https://nvd.nist.gov/) are some of the open public sites which publish vulnerabilities.

Ceteris paribus, a less secure system is more likely to fail. We've only scratched the surface and this is a topic and study on its own. We recommend cyber-security practices that address both human and non-human vulnerabilities and how all systems, processes, and humans can be patched at regular intervals.

IX. Currency issues:

As a general practice, all software and products used in production should be at the most be (n-2) versions or within two years of a major release. This should be a part of product upgrade strategy to avoid foreseen and unforeseen issues. All patches should be taken into a security and operations group -- here is where the role of the SecOps team is vital -- and should be patched accordingly.

Why this is important - After the discovery of critical Spectre-NG vulnerability, the following statement was issued by Intel – "Protecting our customers' data and ensuring the security of our products are critical priorities for us. We routinely work closely with customers, partners, other chipmakers and researchers to understand and mitigate any issues that are identified, and part of this process involves reserving blocks of CVE numbers. We believe strongly in the value of coordinated disclosure and will share additional details on any potential issues as we finalize mitigations. As a best practice, we continue to encourage everyone to keep their systems up-to-date." [11]

It is also to be mandated that all maintenance scripts and contingency scripts written for the software are kept updated and any release made to production is inclusive of updates made to the contingency scripts and SOP documents

X. Transition state issues:

Many of the failures occur during an inconsistent or transient state of systems. A failure occurring during an environment startup and shutdown, can cause more damage to the data and the state of the application than a failure occurring during normal business operating conditions. Extra care should be taken to add testing methods and detailed designs to handle such scenarios.

Transition scenarios are special cases and the variables governing the system during transition states are sometimes different than normal operational conditions.

Some examples of transition states are – During DR processes, replication backups, reverse DR times, while writing point-in-time disk snapshots to storage, a false positive event in the system triggers a contingency process and needs to be reversed, etc.

XI. Dependency SLA issues:

When the application we build has dependencies, it is imperative that we read the fine print of the documents governing their availability or durability or backup mechanisms.

For example, a customer was using AWS EBS for storing core application data without storing snapshots of the same in a more durable and available S3 storage. If the fine print of AWS can be read (as of March 2019) it reads, "Amazon EBS volumes are designed for an annual failure rate (AFR) of between 0.1% - 0.2%, where failure refers to a complete or partial loss of the volume, depending on the size and performance of the volume. This makes EBS volumes 20 times more reliable than typical commodity disk drives, which fail with an AFR of around 4%. For example, if you have 1,000 EBS volumes running for 1 year, you should expect 1 to 2 will have a failure. EBS also supports a snapshot feature, which is a good way to take point-in-time backups of your data" [10]

A dependency system having 97% availability should not be used for use cases requiring 99.99% availability. If used, then care has to be taken to incorporate additional replicas and other mechanisms to achieve the intended availability.

XII. Lack of robust monitoring:

One of the key areas of failure handling is the ability of the system to monitor itself and detect a failure or a pattern which can cause potential disruption of services.

A good monitoring system today should incorporate event correlation, detect and provide graphical information of the data flows, trace processes across distributed systems, predict event streams for failure, alert users and systems, trigger contingency scripts for automatic recovery, collect logs, search through them, detect anomalies in traffic and network, have configurable rule addition capabilities to identify newer patterns in the logs, trace flows (this is very important as the tracing gets very graphical with multiple dimensions of dependencies), and to top it all, have AI and ML capabilities to sort, cluster, learn, and identify patterns from the vast amount of monitoring data collected. To have a single monitoring tool with all these features is very tough and we may need to have a mix of tools.

Example: A rare component failure in a backup switch at a VISA data center caused an outage in June 2018, resulting in the failure of 5.2 million card payments. Visa admitted that the software to automatically detect failure was not in place and the same was corrected. [13]

The key to resilience is in case of a failure event, the system should quickly detect and deploy counter measures aided by a good monitoring system.

XIII. Mechanical sympathy:

Mechanical sympathy [22] can be understood as the state of mind a developer is in, when he tries to understand the environment where his code or tool runs and tries to design and optimize the code or tool based on that information. This can be static or dynamic. For example, if a developer writes code to observe the RAM usage dynamically and performs garbage collection, then he is said to have designed his code with 'mechanical sympathy'.

On one side, it tends to couple with the dependencies and introduce failures when the operating environment changes (say a version of the Linux kernel or package of Linux). On the other end, asking questions like, "Is my program going to run in a memory-constrained environment like PoS terminal?" brings out good design and algorithms which are frugal and efficient.

In short, Mechanical sympathy is to be exercised wisely.

XIV. DR switchover processes:

Software should participate in the DR and reverse DR switch over processes as failure recovery is never an Ops team-only issue. Recovering from failures involves understanding of the data storage, data replication strategy, what is needed to recover the state of the system when it went down, having enough logs to investigate the issues, and software plays a major role in all of this.

For example, writing a software which doesn't produce enough information to debug issues and understand the failures is clearly a design flaw which has not factored a DR switchover process and RPO.

XV. Automation, AI & ML:

The ability of the system to recover from failures sometimes involves executing manual procedures and hence is prone to error, time consuming, and introduces key-man dependency.

For example, in March 2016 a major outage occurred at Telstra, associated to a human-made error. IT operations consultant Sam Newman of Thought Works responded to it, "It's about the system you create, it's not about individuals." and continued "… Looking for a single cause of failure is like looking for a single cause of success,". [12]

Automation scripts and robotic processes are some of the mechanisms which can be employed in production to avoid delays in running recovery mechanism in production. Self-healing, auto-healing, and self-stabilization mechanisms should be adopted as much as possible.

Adoption of AI and ML is important as with Giga/Tera bytes of logs and events generated every day, it is humanly impossible to peruse and correlate issues. AI and ML pattern recognition algorithms can identify, predict, and report failures and anomalies from the piles of monitoring data.

For example, ML algorithms based on 'survival analysis' [23] models can be used to predict failures.

XVI. Lack of training:

It is important that Ops Team and Application Team should train together with mock DR drills.

By engaging in such activities, developers realize that the software which they have built has failed to cope up with the resilience SLA. For example, the lead could realize that the algorithm used to defer database writes affects RTO. Also most of the application maintenance and contingency scripts are written by the application team and any scope to reduce manual work and automate the same can be done.

Training also detects problems like the SOP document not being fool-proof and while executing steps, new scenarios or failures could be encountered.

XVII. Infrastructure Failures:

Software depends on infrastructure services like network, storage, OS, etc. While choice of architectural design, replication, and deployment strategies takes care of the software side of issues, the infrastructure side of things needs to be handled separately and if the infrastructure is down, then everything built on top of it will go down. Redundancy, clustering, replication, a good monitoring and alerting system, selection of reliable infrastructure etc., are some of the many processes and tools which are adopted today to handle infrastructure failures.

XVIII. Processes – a closer look:

In an enterprise, processes are the key delivery vehicles and it is important that we take a closer look at the various processes governing software.

In one of our assessments we found that a software was tested and pushed into production and the same was also pushed to the DR environment without testing, as it was identical. This gap in testing and change rollout processes introduces an additional risk which 'could've been mitigated' if we had done a basic sanity testing in DR environment. We identified this and the risk was mitigated.

In March 2019, operation ShadowHammer attacked thousands of users by infecting the live update server in ASUS with malware. According to the verge, the origin of this issue was likely a 'supply chain' attack where malicious software or components are installed on systems before or while building the systems. [16]

The key lesson here is to pay attention to all processes in the enterprise which directly or indirectly affect the software.

XIX. Primacy of architecture principles:

Like they decide on the various good patterns and best practices, it is also equally important that developers and architects always have an internal compass pointing to the common principles of architecture to guide them at every point of time.

It is common that developers and architects are getting trained in many distributed technologies today and with the technology marketing heave, it is common that developers are lost and make choices in technology and design which may look like it is paying off now or give a 'feeling' that it will solve their issues today, but may eventually cause other issues. Though architecture checks and balances can be setup to review and approve new designs, sometimes things pass through many such measures due to other forces like time constraints, delivery constraints, cost cutting, and lack of expertise.

Some principles to look at - YAGNI – You Ain't Gonna Need It, KISS – Keep it Simple and Short, Data is shared, Compliance with Law, Always think ahead of the quick fix, Don't use a cannon to kill a mosquito, Open-Closed principle, Illusion of control, Premature optimization is evil, Never trust anything which is coming into your module or system, etc.

One example is on the principle of 'always document your design' - All design and documents at a high level should be captured in some sort of document so that it is reviewed. Agile prefers working software over comprehensive documentation, but never mentions 'no documentation' [18]. Not knowing how a complex distributed cache works in your enterprise will cause an issue, if not today, then sometime tomorrow.

Why this is important – Let's take the general case of attaching a node to a load balancer when one of the servers goes down. It is a pretty common use case, but when we simply follow this pattern without understanding the after effects of it, this will lead to what is called as a 'black hole effect' [19]. A black hole effect happens when the load balancer basically finds out that the new server added actually has no sessions affined to it and starts routing all requests to the new server. This new server now 'sucks' all requests and hence is over flooded with requests causing sudden imbalance and slowness.

This anti-pattern of just attaching a node to a load-balancer was later discovered and was fixed by adding various weightage, throttling, and limits to the load balancer. What principles could have prevented this during design? How about, always think ahead of the quick fix?

In general, good architecture principles guide us to develop good resilient systems and also help weed out anti-patterns in our systems. TOGAF framework has published some of the key principles [20] and the same can be read from the reference section below.

Summary

Designing and developing a distributed software is challenging in many ways, but the benefits far outweigh the cons and rest assured, the ecosystem of distributed system is growing and continues to grow rapidly, with technology and software fast evolving to handle the failures and issues which are envisioned in distributed systems. Here are some of the golden takeaways for resilience.

1. Resilience is not related to a singular context in distributed systems and is not dependent on humans to be error-free. All systems, processes, tools, libraries, software, hardware, infrastructure, dependent services, etc., come together to achieve resilient business operations.
2. There are no failure-free systems. No system can be declared 100% resilient or failure-free.
3. Understanding and handling failures is key to resilience. All processes governing software - design, development, deployment, operations etc., can contribute to a failure.
4. Practically all software built by major technology companies have experienced failures and caused downtime in some form or another, such as at Facebook, Apple, Boeing, Amazon Google, Microsoft, Linux to name a few. So rest assured, you are not alone.
5. There is no book listing all failures in software, however there are lists of things which are known to occur.
6. In a distributed set-up, sometimes failures are not in your control and you need to carry on trusting the dependent systems. You can never be completely free of failures; you can only prevent or embrace them. Embracing them and continually building mechanisms to handle them is the only way to make your system more robust.

Resiliency is a process continuum and should keep evolving from time to time.

About SPEAR:

SPEAR stands for Software Performance Engineering and ARchitecture services and is an integral part of Infosys STAR (Solution Technology and ARchitecture) group aiding primarily the banking and financial vertical in Infosys. We are a large and niche group of Technical and Senior Technical Architects, specializing in assessing and improving performance, resilience, scalability, availability, and reliability in distributed systems through our tailored services and offerings.

For more details contact us at: FS-STAR-SPEAR@infosys.com

References

1. German bank error - http://content.time.com/time/business/article/0,8599,1952305,00.html
2. Year 2010 problem - https://www.dw.com/en/millions-of-german-bank-cards-hit-by-software-bug/a-5088075
3. Reliability - http://www.cse.msu.edu/~stire/HomePage/Papers/wadsChapter05.pdf
4. Idempotent failover - https://www.springer.com/us/book/9783540407270
5. Traffic stats - https://www.similarweb.com/website/facebook.com#overview
6. Downtime and lag time - https://medium.com/@vikigreen/impact-of-slow-page-load-time-on-website-performance-40d5c9ce568a
7. Downtime of amazon - https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#5eabc9b6495c
8. Amazon revenue by segment - online sales - https://www.statista.com/statistics/672747/amazons-consolidated-net-revenue-by-segment/
9. 3 Banks DDoS attacks - https://www.cshub.com/attacks/news/incident-of-the-week-ddos-attack-hits-3-banks
10. EBS - AWS - https://aws.amazon.com/ebs/features/#Amazon_EBS_Snapshots
11. Intel's comments - https://www.theregister.co.uk/2018/10/08/intel_security_commitment/
12. Telstra human error - https://www.news.com.au/technology/gadgets/mobile-phones/telstra-explains-network-outage-as-worker-faces-the-music/news-story/7e3f2214350094c3c2096ad14f7480ae
13. VISA outage - https://www.cbronline.com/news/visa-outage
14. Actor Model - https://doc.akka.io/docs/akka/2.5/guide/actors-intro.html
15. Chaos Engineering - https://principlesofchaos.org/
16. ASUS attack - https://techhq.com/2019/03/asus-breach-highlights-software-supply-chain-risk/
17. Let it Fail approach - http://ward.bay.wiki.org/view/let-it-fail
18. Agile Manifesto - https://agilemanifesto.org/
19. Black Hole - https://www.brianmadden.com/opinion/Dealing-with-the-Black-Hole-Effect-Throttling-Logons-to-New-Servers
20. TOGAF Architecture principles - http://pubs.opengroup.org/architecture/togaf8-doc/arch/chap29.html
21. Flow Modeling references:
    a. FMEA - https://wiki.ece.cmu.edu/ddl/index.php/FMEA
    b. PGM - http://pgm.stanford.edu/algorithms/
22. Mechanical Sympathy - https://dzone.com/articles/mechanical-sympathy
23. Survival analysis - https://en.wikipedia.org/wiki/Survival_analysis
24. Architecturally Significant Requirements:
    a. https://www.ida.liu.se/~TDDD09/openup/core.tech.common.extend_supp/guidances/concepts/arch_significant_requirements_1EE5D757.html
    b. https://www.ibm.com/developerworks/rational/library/4706.html

For more information, contact askus@infosys.com

Infosys®
Navigate your next