SE480 Week 2 – Availability, Safety

Steven Engelhardt

DePaul University

Autumn 2020

Table of Contents

1. Last Week
2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations
3. Safety
   - Case Studies
   - Architecting for Safety
4. Wrap-Up

Summary of Last Week

- Architecture is the bridge between business goals and concrete systems
- Architecture inhibits or enables a system's driving quality attributes
- Architects focus on entire (sub)systems rather than individual functions or components
- Architects design structures and then create views to represent the whole system from the perspective of a set of concerns

Table of Contents

1. Last Week

2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations

3. Safety
   - Case Studies
   - Architecting for Safety

4. Wrap-Up

Table of Contents

1. Last Week

2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations

3. Safety
   - Case Studies
   - Architecting for Safety

4. Wrap-Up

Failures, Faults, and Errors

- A *failure* is a deviation of a system from its specification where the deviation is externally visible.
- A failure's cause is called a *fault*, which may be internal or external
- Intermediate states between the occurrence of a fault and the occurrence of a failure are called *errors*
- A fault can occur without a failure as long as the system is able to recover and there is no observable change in behavior
  - This happens all the time in the real world!

Figure: Microsoft's $500 million data center in Northlake, IL circa 2009

What is Availability?

- *Availability* is a property of software that it is there and ready to carry out its task when the user needs it to be
- "Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval." [BCK12]

- *Failures* reduce availability; *faults* do not
- Availability measurements should include a notion of *timeliness*
  - A system that does not respond within a reasonable timeframe is not meaningfully different to a user than one that fails to respond

Calculating Availability

- For hardware, expected availability is typically calculated by the equation

$$\frac{MTBF}{MTBF + MTTR}$$

- For services, availability is typically calculated by the equation

$$\text{Availability \%} = \frac{\text{Agreed Service Time} - \text{Downtime}}{\text{Agreed Service Time}} \times 100\%$$

- *Scheduled downtime* is typically excluded from *Agreed Service Time*
    - In our 24x7, global world, the ability to schedule downtime is becoming rarer

Measuring Availability

- Defining *downtime* is surprisingly difficult. If the ‘send email’ feature of an application is broken, is it ‘down’?
- Availability can be measured either *actively* or *passively*
- A common technique used to measure availability is using *synthetic transactions* which exercise the application’s *happy path* at a fixed interval
- When designing availability tests, consider whether you’re trying to detect failures or faults
    - What if you have a caching layer in front of your database and your database fails?

Measuring Availability Example

A service provider measures availability over a month by dividing it into 5-minute intervals and performing a *synthetic transaction test* every interval. If the synthetic transaction test fails, the system is considered ‘down’ for the entire 5 minute interval. If this test fails 10 times in the entire month of January, what is the availability of the system for that month?

Measuring Availability Example

A service provider measures availability over a month by dividing it into 5-minute intervals and performing a *synthetic transaction test* every interval. If the synthetic transaction test fails, the system is considered ‘down’ for the entire 5 minute interval. If this test fails 10 times in the entire month of January, what is the availability of the system for that month?

$$\text{Num intervals} = 31 \text{ days} \times \frac{24 \text{ hours}}{\text{day}} \times \frac{60 \text{ minutes}}{\text{hour}} \times \frac{1 \text{ interval}}{5 \text{ minutes}} = 8928$$

$$\text{Availability} = \frac{8928 - 10}{8928} \times 100 = 99.89\%$$

Downtime Implications of Availability Targets

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999% (*a.k.a. 5 9's, high availability*)	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

$$\text{Allowed Annual Downtime} = (1 - \text{Availability } \%) \times \frac{365.24 \text{ days}}{\text{year}} \times \frac{24 \text{ hours}}{\text{day}} \times \frac{60 \text{ minutes}}{\text{hour}} \times \frac{60 \text{ seconds}}{\text{minute}}$$

Closely Related Metrics

Other metrics that closely relate to availability include:

- *Recovery time objective (RTO)* – how long does a service provider have to restore service after a disaster
- *Recovery point objective (RPO)* – what is the maximum amount of data that might be lost due to a major incident

These are important numbers to gather early, as they influence the system's architecture significantly!

Service-Level Agreements

- A *service-level agreement (SLA)* specifies the availability level that is guaranteed and, usually, the penalties that the service provider will suffer if the SLA is violated
  - Because this is a contractual obligation, the service provider will frequently try to minimize it
- Service providers often maintain internal *service-level objectives (SLOs)* which are their internal targets for availability
  - More stringent than SLAs
- An *operational-level agreement (OLA)* describes the responsibilities of each internal support group toward other support groups, including the process and timeframe for delivery of their services.

Participation Activity (10-15 minutes)

- Find a SLA on the Internet. Calculate the allowed downtime per the SLA, and estimate the monetary penalties the service provider would incur for violation of the SLA. Share a summary of what you've found.
- General discussion: If you were CEO of a SaaS company, would you find these SLAs adequate for your business?

Business Costs of Lack of Availability

- In July, 2018, a breakdown in an internal system called Sable caused Amazon to fail to have enough servers to handle the traffic surge on Prime Day. The loss in sales is estimated to be about $90 million. [Kim18]
- On August 26, 2017, Google published invalid BGP peer prefixes to Verizon, and corrected them 8 minutes later. About half of Japan lost or had slow Internet access for several hours. Online trading was halted, and East Japan Railway Co. riders weren't able to buy tickets or board trains. Japan's Internal Affairs & Communications ministries are investigating. [Chi17]
- On February 28, 2017, a typo by an Amazon employee caused a failure of Amazon S3 for approximately 4 hours. This cost companies in the S&P 500 approximately $150 million. [Sve17]
- In 2015, Apple's App Store & iTunes were down for about 12 hours. This cost the company approximately $25 million in lost sales. [Spa15]
- In August 2013, Google went down for 5 minutes. Internet traffic dropped 40%. [Sve13]

Table of Contents

1. Last Week
2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations
3. Safety
   - Case Studies
   - Architecting for Safety
4. Wrap-Up

Architecting for Availability

- *Plan for faults!*
- Prevent faults if you can
- Use recovery techniques to ensure faults don't become failures
  - During recovery, time is of the essence
- Use fault detection techniques to ensure faults can be repaired before they become failures

Key Questions to Ask

- What kinds of things are likely to fault?
- How is system failure detected?
- How frequently is a fault likely to occur?
- What happens when a fault occurs?
- How long is a system allowed to be out of operation?
- When can faults occur safely?
- How can faults be prevented?
- What kind of notifications are required?

Availability Tactics

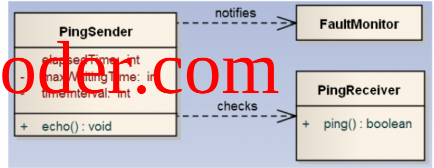
- Purpose: To keep faults from becoming failures.
- Availability tactics are categorized as:
  - Detect faults
  - Recover from faults
  - Prevent faults

Table of Contents

1. Last Week
2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations
3. Safety
   - Case Studies
   - Architecting for Safety
4. Wrap-Up

Ping/Echo

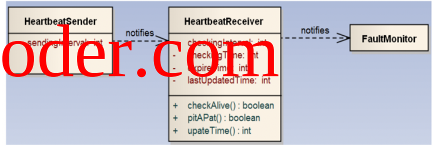
- A *monitor* is a component that it is used to monitor the state of health of various other parts of the system.
- PingSender sends a ping message at specified time intervals and waits for an echo from a ping receiver until the maximum waiting time. If an echo is not received an exception occurs, and it is detected by the fault monitor.



Where is the safety critical component?

Heartbeat

- HeartbeatSender sends a heartbeat message periodically.
- HeartbeatReceiver receives the heartbeat and updates the last received time of a heartbeat message.
- The aliveness of the heartbeat sender is checked regularly by comparing the latency time between the current time and last updated time in consideration of the max waiting time for the next heartbeat message.



Where is the safety critical component?

Quick Summary

- Ping/echo – Are you alive?
- Heartbeat – I am alive!
- Watchdog – A special case of heartbeat where the monitor has a timer that must be explicitly reset before it expires

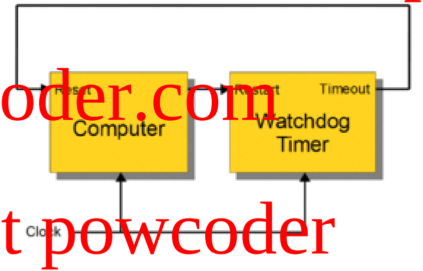


Figure: Watchdog Timer

Condition Monitoring

- *Condition monitoring* involves checking conditions in a process or device, or validating assumptions made during the design.
- A common example of this tactic is computing *checksums*
- Checksums are ubiquituous in computing. Examples include:
  - Parity bits in ECC RAM
  - 32-bit CRC in Ethernet frame
  - 16-bit checksum in TCP
  - Block-level checksums in ZFS
  - MD5 checksums to check integrity of downloaded files
- Food for thought: How much computing horsepower worldwide is wasted on calculating redundant checksums?

Checksum Example – Parity Bit

sequence of seven bits	with eighth even parity bit	with eighth odd partiy bit
0100010	01000100	01000101
1000000	10000001	10000000

Figure: Parity Bit Example

- A *parity bit*, or *check bit*, is a bit added to a string of binary code to ensure that the total number of 1-bits in the string is even or odd.
- How many bit errors can a parity bit detect?
- How many bit errors can a parity bit correct?

Checksum Example – Parity Bit

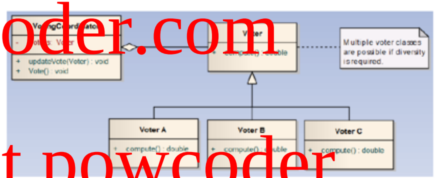
sequence of seven bits	with eighth even parity bit	with eighth odd partiy bit
0100010	01000100	01000101
1000000	10000001	10000000

Figure: Parity Bit Example

- A *parity bit*, or *check bit*, is a bit added to a string of binary code to ensure that the total number of 1-bits in the string is even or odd.
- How many bit errors can a parity bit detect? *1*
- How many bit errors can a parity bit correct? *0*
- *For error correction, other techniques (e.g. CRCs, Hamming codes) are used.*

Voting

- Purpose: Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter.
- Coordination mechanisms:
  - Majority rules (w/ quorum)
  - Preferred component
- Voter diversity:
  - Same algorithm → processor failures only
  - Diverse algorithm → processor & algorithmic failures



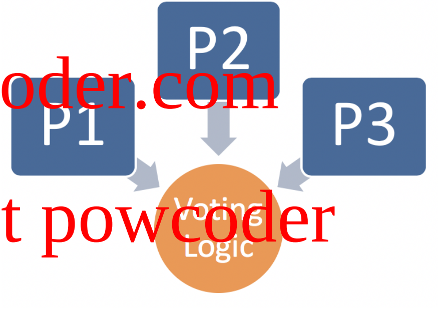
Voting (Continued)

- Forms of *voting* include:
  - *Replication* is the simplest form of voting where components are exact clones of each other
  - *Functional redundancy* is a form of voting where every component must always give the same output given the same input, but they are diversely designed and diversely implemented
  - *Analytic redundancy* permits diversity not only in design and implementation, but also in inputs and outputs. It is intended to tolerate specification errors by using separate requirement specifications.

Triple Modular Redundancy (TMR)

- A form of *voting* with three identical processing units
- Any inconsistency is treated as a system fault

- Relatively uncommon in undistributed, non-safety-critical systems, but extremely common in distributed data storage systems!



Calculating Impact of TMR

A system is built using TMR where each individual component is 99.9% available, and at least two components need to fail in order for the entire system to fail. The components fail independently. What is the reliability of the entire system?

Calculating Impact of TMR

A system is built using TMR where each individual component is 99.9% available, and at least two components need to fail in order for the entire system to fail. The components fail independently. What is the reliability of the entire system?

$$
\begin{aligned}
P_{2\ \text{failures}} &= 3 \times 0.001^2 \times (1 - 0.001) = 0.000002997 \\
P_{3\ \text{failures}} &= 0.001^3 = 0.000000001 \\
P_{\text{failure}} &= P_{2\ \text{failures}} + P_{3\ \text{failures}} \\
&= 0.000002998 = 0.0003\% \\
\text{Availability} &= 1 - P_{\text{failure}} = \boxed{99.9997\%}
\end{aligned}
$$

Over 300 times more reliable!

Message Ordering

- *Time stamps* are used to detect incorrect sequences of events primarily in distributed message-passing systems
- Many systems, like TCP, use an incrementing *sequence number* rather than a clock time
- More sophisticated distributed systems use *vector clocks*, *Lamport timestamps*, or other mechanisms for event ordering and conflict resolution
- Distributed system synchronization overhead can be costly, but it can be largely eliminated by keeping a globally-synchronized clock. This is exceptionally hard, but not impossible – Google did it with Spanner. [BBB+17]

Other Fault Detection Tactics (Continued)

- *Sanity checking* checks the validity or reasonableness of specific operations or outputs of a component (e.g. asserts)

- *Exception detection* refers to the detection of a system condition that alters the normal flow of execution. Examples include:
  - *System exceptions* such as divide by zero
  - *Parameter fences* are sentinel data patterns placed immediately before or after any variable-length parameters of an object to allow runtime detection of memory buffer overflows or underflows
  - *Parameter typing* is a pattern for implementing message parsers to reduce bugs
  - *Timeout* is a tactic that raises an exception when a component detect that it or another component has failed to meet its timing constraints

- *Self-tests* are where components can run procedures to test themselves for correct operation

Table of Contents

1. Last Week
2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations
3. Safety
   - Case Studies
   - Architecting for Safety
4. Wrap-Up

Active Redundancy (Hot Spare)

- Multiple systems maintained in identical state so that any one can be used
- All redundant components respond to events in parallel and are therefore maintained in the same state
- Only one response is used (usually first responder), others are discarded
- When faults occurs, downtime is usually in milliseconds
- Redundancy can be either at the processor level or the communication path (i.e. single path failure will not make all components inaccessible)

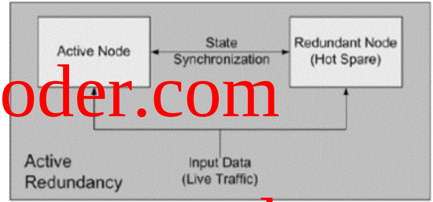


Figure: Active Redundancy (Hot Spare)

Passive Redundancy (Warm Spare)

- One component responds to events and informs the standby components of state updates they must make
- When a fault occurs, the system must check that backup state is sufficiently fresh
- Considerations:
  - Which component decides when the secondary component should take over from the primary one?
  - How to prevent *flapping*, which is repeatedly moving back-and-forth between the active node and warm spare?
  - How tightly coupled are primary and secondary components?

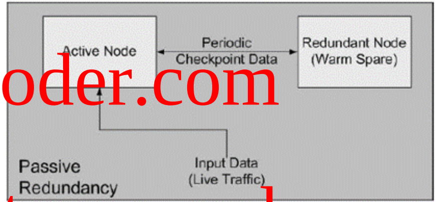


Figure: Passive Redundancy (Warm Spare)

Sparing (Cold Spare)

- A standby spare computing platform is configured to replace failed components.
- Must be rebooted when a failure occurs.
- State must be refreshed before spare goes online.
- Downtime usually measured in minutes.
- More suited for systems with ONLY high reliability (MTBF) as opposed to high availability requirements

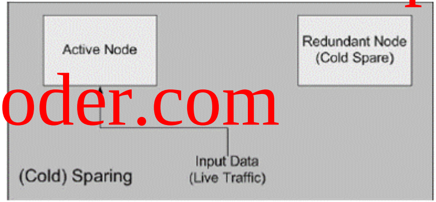


Figure: Sparing (Cold Spare)

Exception Handling

- *Exception handling* refers to the system's policy for what to do when it encounters an exception
- The book claims that the simplest and worst way to respond to exceptions is to crash. But is it so terrible?
  - Can your system maintain all invariants in the face of all exceptions?
  - Can you test all of these scenarios?

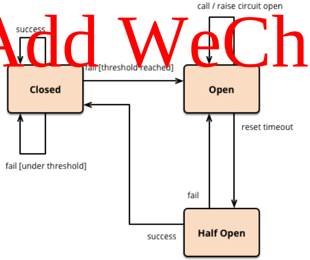
- Some architects believe that we can build more reliable systems by building crash-only software [CF03]

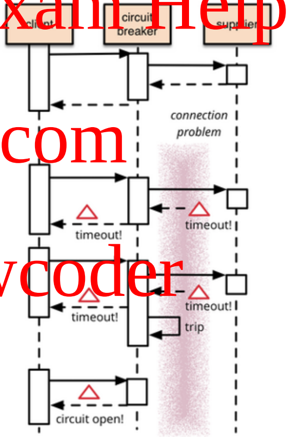
Retry

- Many developers, when dealing with network services that aren't 100% reliable, introduce a "retry loop" in their code
- Common problems with retry loops:
  - Lack of delay between retries
  - Lack of limits to number of retries
  - Lack of backoff between retries
  - Retries not randomly distributed across systems
  - What if A calls-with-retry service B, which calls-with-retry service C, which calls-with-retry service D, and service D fails?

Degradation

- The *degradation* tactic is about continuing to provide degraded service in the face of component failures
- Degradation is commonly implemented with *circuit breakers*
- Once failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.





Load Shedding

- *Load shedding* is about flatly ignoring (or immediately returning failure codes to) *some* requests rather than overwhelming a system and making it fail to serve *any request*

- In some circumstances, a request that has exceeded its response time SLA mineaswell never return

Rollback

- *Rollback* refers to allowing the system to revert to a previous known good state upon the detection of a failure
- This requires the system to know how to mark a state as good using *checkpoints*

- SQL databases support rollback exceptionally well
- Software upgrades (e.g. Windows updates) frequently use checkpoints & rollback techniques as well

Software Upgrade

- *Software upgrade* refers to techniques to perform code updates without affecting services
- Many of you will work on systems that requires zero-downtime upgrades
- For stateless systems this is easy. For stateful systems, it can get very tricky, but it is possible.
- Don't forget that there may be both correctness as well as performance requirements for successful zero-downtime upgrades

Blue-Green Deployments

- *Blue-green deployment* is a technique that reduces downtime and risk by running two identical production environments called Blue and Green.
- At any time, only one of the environments is live, with the live environment serving all production traffic.
- The process of switching from blue to green is intended to be quick, reliable, and easy to reverse, allowing for rollback

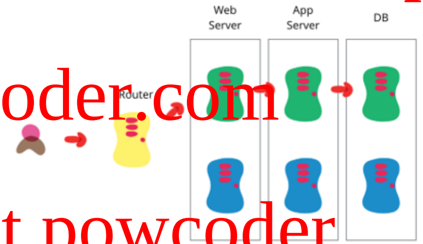


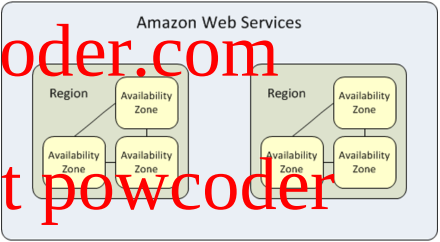
Figure: Green is currently live and Blue is idle

Availability Zones

- *Availability zones* are a way to partition a region (or data center) into pieces which are explicitly designed to fail independently

- Software architects can then ensure their system is designed & deployed in such a way to handle the failure of an availability zone

- Availability zones are frequently used as both *fault domains* and *patch domains*



Shadow

- *Shadow* refers to operating a previously failed or in-service upgraded component in a "shadow mode" for a predefined duration of time prior to reverting the component back to an active role.

- Useful to "warm up" a component before having it serve traffic again as well as to observe its behavior for correctness before activating it
- Particularly useful for caching systems, e.g. a Varnish HTTP Cache

Non-Stop Forwarding

- *Non-stop forwarding* originated in router design and splits functionality into two parts: supervisory, or control plane (which manages connectivity and routing information) and data plane (which does the actual work of routing packets from sender to receiver).
- Done correctly, this means that much of the router functionality continues to work even if the control plane crashes.

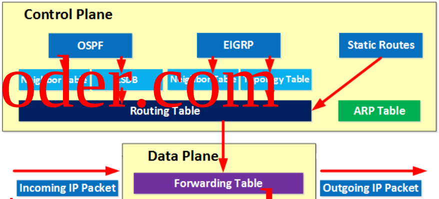


Figure: Architecture of Non-Stop Forwarding Router

Other Fault Recovery Tactics

- The *ignore faulty behavior* tactic calls for ignoring messages sent from a particular source when we determine that those messages are spurious.

- *Reconfiguration* attempts to recover from component failures by reassigning responsibilities to the (potentially restricted) resources left functioning, while maintaing as much functionality as possible.

- *Escalating restart* is a reintroduction tactic that allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.

Table of Contents

1 Last Week

2 Availability

Intro to Availability

Architecting for Availability

Fault Detection Tactics

Fault Recovery Tactics

Fault Prevention Tactics

Additional Techniques and Considerations

3 Safety

Case Studies

Architecting for Safety

4 Wrap-Up

Removal from Service

- The *removal from service* tactic refers to temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures.

- A common example of this tactic in practice is periodic server reboots to scrub latent faults
- This tactic also relates closely to other tactics like zero-downtime software upgrades

Transactions

- Architects can embrace *transactional semantics* to improve availability by, for example, ensuring asynchronous messages are *atomic*, *consistent*, *isolated*, and *durable*
- *Two-phase commit* (2PC) is a common way to implement the transactions tactic in distributed systems
- The *Unit of Work* design pattern is one implementation of this tactic
- At the programming language level, *software transactional memory* is an implementation of this tactic which continues to be a focus of research efforts
- However, transactions can impede *concurrency* and thus limit scalability

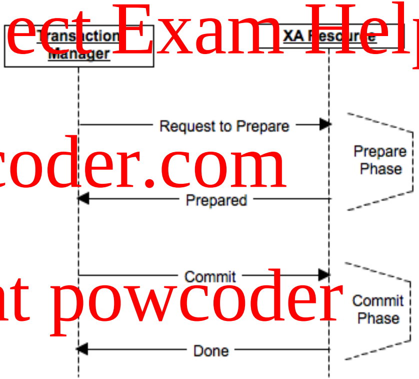


Figure: Two-Phase Commit

Predictive Model

- A *predictive model*, when combined with a monitor, is used to take corrective action when conditions are detected that are predictive of likely future faults.
- Examples include:
  - Automatically restarting a process when resource utilization exceeds a maximum threshold because of known resource leaks
  - *Auto scaling* in cloud services

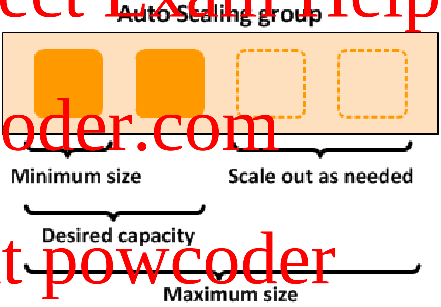


Figure: Auto Scaling

Other Fault Prevention Tactics

- *Exception prevention* refers to programming techniques to reduce the number of exceptions or automatically handle them when they occur.
- *Increase competence set* means designing a system to handle more cases – faults – as part of its normal operation.

Table of Contents

1. Last Week

2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations

3. Safety
   - Case Studies
   - Architecting for Safety

4. Wrap-Up

Statelessness

- A *stateless* service is a service that doesn't store data on the host.
- Stateless services are:
  - More reliable
  - Easy to manage failures
  - Easy to deploy
  - Easy to upgrade
  - Easy to scale

Proxying Load Balancers

Proxying load balancers, like F5's BIG-IP Local Traffic Manager (LTM), include the ability to perform many availability-related functions such as:

- Monitoring health checks
- Automatically marking services in- and out-of-service based on health checks
- Traffic shaping
- Instance failover

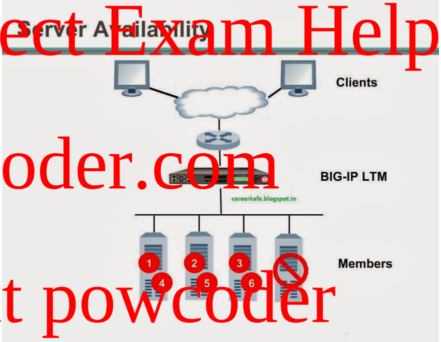


Figure: Proxying Load Balancer

Netflix's Simian Army

- The problem with faults is they are *low-probability, high-impact scenarios* (black swans). Humans are generally terrible at predicting and dealing with these scenarios.
- Furthermore, as production systems become complex, it becomes nearly impossible to reason about them holistically.
- Netflix's solution: *Make faults common*, and require components to be able to handle them.
- This philosophy has been given a name: *chaos engineering*

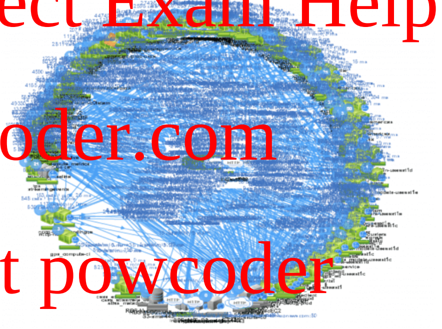


Figure: Netflix's Architecture (2013)

Chaos Engineering

- "Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production." [pri]
- Principles:
  - Build a Hypothesis around Steady State Behavior
  - Vary Real-world Events
  - Run Experiments in Production
  - Automate Experiments to Run Continuously
  - Minimize Blast Radius

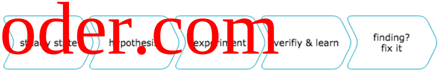


Figure: Phases of Chaos Engineering

When to Use Chaos Engineering

- Talk to your colleagues about the planned chaos experiments in advance!
- If you know your chaos experiment will fail, don't do it!
- Chaos shouldn't come as a surprise, your aim is to prove the hypothesis.
- Chaos Engineering helps you understand your distributed systems better.
- Limit the blast radius of your chaos experiments.
- Always be in control of the situation during the chaos experiment!

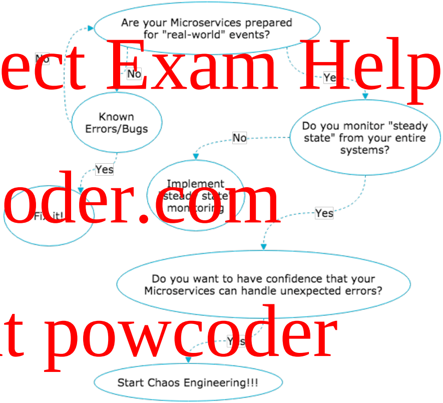

Are your Microservices prepared for "real-world" events?
No
Yes
Known Errors/Bugs
Do you monitor "steady state" from your entire systems?
No
Yes
Fix it!
Implement steady state monitoring
Yes
No
Do you want to have confidence that your Microservices can handle unexpected errors?
Yes
Start Chaos Engineering!!!

Figure: Chaos Engineering Decision Helper [Wil18]

The Human Element

- What if you implement an availability tactic which requires human judgment, decision making, or action?
- What if the fault happens at 3AM on a Sunday?
- How will this affect your availability figures?

Other Thoughts

- Distribution without redundancy can hurt availability significantly!
- You cannot divorce availability from capacity. Does your system have the capacity to handle the fault of one or more elements?
- How reliable is a data center, really?
- What are the lessons we can learn on the balance between economies of scale and availability? Consider the electrical grid.

Figure: 2003 Northeast Electrical Blackout

Optional Reading

- Throughput autoscaling: Dynamic sizing for Facebook.com
- Why Serverless Apps Fail and How to Design Resilient Architectures
- Zoom Video Communications, Inc. Global Infrastructure and Security Guide
- Deploys at Slack
- Understanding, detecting, and localizing partial failures in large system software
- Simple Systems Have Less Downtime
- Meaningful Availability
- Failure Modes and Building Resilient Systems: Adrian Cockcroft at QCon SF
- Managing reliability with SLOs and Error Budgets
- Faliure Modes and Continuous Resilience
- Hot Patching SQL Server Engine in Azure SQL Database

Table of Contents

1. Last Week
2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations
3. Safety
   - Case Studies
   - Architecting for Safety
4. Wrap-Up

What is Safety?

- Safety is about the software's ability to avoid entering states that cause or lead to damage, injury, or loss of life to actors in the software's environment, and to recover and limit the damage when it does enter into bad states

Table of Contents

1. Last Week

2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations

3. Safety
   - Case Studies
   - Architecting for Safety

4. Wrap-Up

737 Max

- October 29, 2018: Lion Air Flight 610, a 737 MAX 8, registration PK-LQP, on a flight from Jakarta, Indonesia to Pangkal Pinang, Indonesia, crashed into the sea 13 minutes after takeoff, killing all 189 people aboard.
- March 10, 2019: Ethiopian Airlines Flight 302, a 737 MAX 8, registration ET-AVJ, on a flight from Addis Ababa, Ethiopia to Nairobi, Kenya, crashed 6 minutes after takeoff, killing all 157 people aboard.
- The 737 Max has much larger engines than previous 737s. These engines were extended up and well in front of the wing. However, doing so also meant that the centerline of the engine's thrust changed. Now, when the pilots applied power to the engine, the aircraft would have a significant propensity to "pitch up," or raise its nose. [Tra19]

737 Max (Continued)

- This propensity to pitch up with power application thereby increased the risk that the airplane could stall when the pilots “punched it”. It’s particularly likely to happen if the airplane is flying slowly.
- Apparently the 737 Max pitched up a bit too much for comfort on power application as well as at already-high angles of attack.
- Instead of going back to the drawing board and getting the airframe hardware right, Boeing relied on something called the “Maneuvering Characteristics Augmentation System,” or MCAS.
- The MCAS pushes the nose of the plane down when the system thinks the plane might exceed its angle-of-attack limits; it does so to avoid an aerodynamic stall. It also does something else: Indirectly, via something Boeing calls the “Elevator Feel Computer,” it pushes the pilot’s control columns (the things the pilots pull or push on to raise or lower the aircraft’s nose) downward.

737 Max (Continued)

- The Elevator Feel Computer can put a lot of force into that column – so much force that a human pilot can quickly become exhausted trying to pull the column back, trying to tell the computer that this really, really should not be happening.
- In the 737 Max, only one of the flight management computers is active at a time – either the pilot's computer or the copilot's computer. And the active computer takes inputs only from the sensors on its own side of the aircraft.
- This means that if a particular angle-of-attack sensor goes haywire – which happens all the time in a machine that alternates from one extreme environment to another, vibrating and shaking all the way – the flight management computer just believes it.
- In the MCAS system, the flight management computer is blind to any other evidence that it is wrong, including what the pilot sees with his own eyes and what he does when he desperately tries to pull back on the control column.

Therac-25

- In the mid-80s, a Canadian-built radiation-treatment device called the Therac-25 began dealing fatal doses of radiation to patients. [Ros94]
- In June, 1985, Katie Yarborough was supposed to receive a dose of about 200 rads as part of her cancer treatment. Instead, the device delivered an estimated 15,000 − 20,000 rads on a dime-sized space. 1,000 rads can be fatal if it is spread over the entire body. She died 5 years later.
- It took four deaths over roughly a year before anyone thought to blame the software behing the Therac-25 and another death before the errors were resolved.

Therac-25 (Continued)

- On March 21, 1986, a technician was operating the Therac-25. At the computer console she typed in the prescription data for an electron beam of 180 rads, then noticed she'd made an error by typing in command x (for x-ray treatments) instead of e (for electron). She ran the cursor up the screen to change the command x to e, as the patient's prescription required. She verified everything else and turned on the beam. The machine stopped and the computer screen flashed "Malfunction 54," a mysterious message not even mentioned in the Therac-25 manual.
- The technicians who operated the Therac-25 were used to computer glitches. The Therac-25 did so by displaying "Malfunction" plus a number, from 1 through 64. No explanation was offered by the computer nor was there any reference to the malfunction codes in the operator's manual. Technicians could, in most cases, bypass the irritating malfunctions simply by pressing the "p" key, for "proceed." Doing so became a matter of habit.

Therac-25 (Continued)

- The speed with which the instructions were entered made the difference. According to a computer system's analysis of FDA documents, the computer would not accept new information on a particular phase of treatment if the technician made the changes within eight seconds after reaching the end of the prescription data. That's what Malfunction 54 meant. If the changes were made so soon, all the new screen data would look correct to the technician. But inside the computer, the software would already have encoded the old information.
- That meant the beam on the Therac-25 would be set for the much stronger dose needed for an x-ray beam while the turn-table was in the electron position. The coded information within the computer apparently included no system to check that various parts of the prescription data agreed with one another.

Therac-25 (Continued)

- However, this wasn't the only accident-causing bug in the Therac-25. On the Therac-25, the part of the computer program that is often referred to as the ‘house-keeper task’ continuously checked to see whether the turntable was correctly positioned. A zero on the counter indicated to the technician that the turntable was in the correct position. Any value other than zero meant that it wasn't, and that treatment couldn't begin. The computer would then make the necessary corrections and the counter would reset itself to zero.
- But the highest value the counter could register was 255. If the program reached 256 checks, the counter automatically clicked back to zero. For that split second, the Therac-25 believed it was safe to proceed when, in fact, it wasn't. If the technician hit the “set” button to begin treatment at that precise moment, the turntable would be in the wrong position and the patient would be struck by a raw beam.

Table of Contents

1. Last Week

2. Availability
   - Intro to Availability
   - Architecting for Availability
   - Fault Detection Tactics
   - Fault Recovery Tactics
   - Fault Prevention Tactics
   - Additional Techniques and Considerations

3. Safety
   - Case Studies
   - Architecting for Safety

4. Wrap-Up

Architecting for Safety

- Formal methods exist, like IEC 61508, on how to apply, design, deploy and maintain automatic protection systems called safety-related systems.

- Basic techniques include:
  - *Simplicity* – Simple software with a small set of simple components with simple interfaces is less prone to faults.
  - *Redundancy* – See the availability section
  - *Fail Safe* – Expect the software to eventually fail, but do so in a safe way.

Table of Contents

1 Last Week

2 Availability

- Intro to Availability
- Architecting for Availability
- Fault Detection Tactics
- Fault Recovery Tactics
- Fault Prevention Tactics
- Additional Techniques and Considerations

3 Safety

- Case Studies
- Architecting for Safety

4 Wrap-Up

Wrap-Up

- Read chapters 5, 12 from SAIP
- Homework 1 is due next week: Thursday, September 24 at 5:30PM.
- Quiz 1 is available now. It is due Thursday, September 24 at 5:30PM.
- Make sure you have filled out the attendance survey at `https://d2l.depaul.edu/d2l/lms/survey/admin/modify/survey_newedit_properties.d2l?si=333024&ou=767377`

Next Week

- Topics: Security, testability
- Read chapters 9, 10 from SAIP

If Extra Time... Architectural Thinking

- Two common terms you will encounter are *high availability (HA)*, which is the system's ability to remain accessible in the event of a system component failure, and *disaster recovery (DR)*, which is the process by which a system is restored to a previous acceptable state, after a natural or man-made disaster.

- *Class discussion*: What are some of the differences between designing a system to support HA vs. DR? What are some of the key challenges?

References

[BBB+17] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford.
Spanner: Becoming a sql system.
In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, New York, NY, USA, 2017. ACM.

[BCK12] Len Bass, Paul Clements, and Rick Kazman.
Software Architecture in Practice.
Addison-Wesley Professional, 3rd edition, 2012.

[CF03] George Candea and Armando Fox.
Crash-only software.
In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[Chi17] Richard Chirgwin.
Google routing blunder sent japan's internet dark on friday.
The Register, 2017.

[Kim18] Eugene Kim.
Internal documents show how amazon scrambled to fix prime day glitches.
CNBC, 2018.

[pri] Principles of chaos engineering.

[Ros94] Barbara Wade Rose.
Fatal dose: Radiation deaths linked to aecl computer errors.
1994.

[Spa15] Matthew Sparkes.
Apple's app store and itunes go down.
The Register, 2015.

[Ste17] Laura Stevens.
Amazon finds the cause of its aws outage: A typo.
The Wall Street Journal, 2017.

[Sve13] Joe Svetlik.
Google goes down for 5 minutes, internet traffic drops 40.
CNET, 2013.

[Tra19] Gregory Travis.
How the boeing 737 max disaster looks to a software developer.
IEEE Spectrum, 2019.

[Wil18] Benjamin Wilms.
Chaos engineering – withstanding turbulent conditions in production, 2018.