ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 23 Chaos Engineering

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 5, 2025

Outline

Reading Assignment

- This lecture: 9, Chao Engineering
  - What is chaos engineering?
    https://www.ibm.com/think/topics/chaos-engineering
- Next Lecture: 10

Outline

Health Check

Chaos Enigeering

Service Redundancy

- Duplicate critical components or functions to improve reliability.
  - Deploy component to multiple server instances.
  - Ideally across multiple zones or even across multiple regions.
- Autoscaling helps to maintain certain level of redundancy as demand fluctuates. However, it takes time to start an instance so there should be room for redundancy without scaling.
- Fault masking: a system fault is invisibly compensated for without being explicitly detected.
  - Without careful planning, redundancy will lead to fault masking that conceals progressive faults.
  - E.g. loss of nodes for a service are not observed until all nodes are lost, causing a sudden and catastrophic outcome.

Health Check: Pull Model

- An API endpoint for clients to decide if a service instance is alive and healthy.
  - For clients that are aware of the redundancy, e.g. Cassandra and Kafka clients, as well as load balancers, monitoring services, service registries, etc.
- Usually implemented as an HTTP endpoint for simplicity.
  - E.g. available from /health that returns 200 OK for a health service or 503 Service Unavailable otherwise.
- Trade-offs between latency and scalability.
  - Frequent health checks may lead to inefficiency, in particular when there are a lot of services and a lot of clients.
  - For longer intervals between health checks, clients may miss critical information like when a service actually dies.

Health Check: Push Model

- Let services send health information to clients.
  - Periodically, e.g. heartbeats.
  - Proactively when health status changes.
- A more complex system.
  - Where are the clients?
  - What if there are more information than what a single client can handle?
  - Use message queues to decouple services from clients and to handle scalability better.
- What does it mean for an instance to be "healthy"?
  - Is a response of 200 OK from /health sufficient for both the clients and the instance?

“Healthy” Instances

- Simple definition: “healthy” means “available”
  - But availability of instances may be impacted by availability of services these instances depending on.
  - Restarting/replacing these instances won’t help at all.
- Need to make choices depending on services.
  - Liveness checks: a simple response to indicate the service instance is reachable and responding, confirming correctness of network, security, and service configuration.
  - Shallow health checks: ensure local resources (memory, CPU, disk etc.) and dependencies (monotoring etc.) are available so the service instance is likely to be able to funciton.
  - Deep health checks: inspect the ability to interact with other subsystems, identifying potential issues like networking – however, it is costly and it is possible to have all instances reporting unhealthy.

Outline

Health Check

Chaos Enigeering

Chaos Engineering

- How do we prove mechanisms designed and implemented for resilience actually works?
  - Waiting for failure to happen and then discovering the implementaion has a bug would lead to disasters.
- Intentional and controlled causing of failures.
  - To understand their impacts in complex distributed systems.
  - To have a better plan for when failures actually happen – time is precious when the system is actually down.

Production vs. Pre-Production Environments

- Production environment provides the most accurate environment for understanding how an incident impacts the customer experience.
- For pre-production environments like development,
  - Some issues could only be triggered when a level of live traffic is presented.
  - Security configurations could be very different.
- Still, it is reasonable to start practices of chaos engineering in a development environment to understand the process before applying them to a production environment that will affect actual customers.

A Netflix Story

- An outage in 2008 led to a three-day interruption.
  - When Netflix was transiting to online streaming.
- Chaos Monkey: an open source tool to create random incidents in services and infrastructure
  - Implemented when Netflix moved from private data center to AWS which is less reliable.
  - Identify weaknesses that can be fixed or addressed through automatic recovery procedures
  - Minimize the damage if and when an unavoidable failure occurs

Experiments

- Latency injection
  - Create scenarios that emulate a slow or failing network connection
  - Understand how system performs with unexpected network delays or slower response times.
- Fault injection
  - Introduce errors into the system, e.g. disk failures, processes termination, host shutdown.
  - Determine how faults propagate to other dependent systems and whether they interrupt services.

Experiments (Cont.)

- Load generation
  - Stress the system by sending significant traffic levels well beyond normal operations.
  - Understand where the bottlenecks are in the system, which in turn allows to build more scalable systems.
- Canary testing
  - Release a new product or feature to a small group of users.
  - Allow to introduce product or feature to production environment while limiting the impacts of glitches or bugs to only a few of clients.

Scope of Experiments

- Component or pod
  - E.g. failure of a single pod or throttle CPU/memory for a single instance.
- Node and cluster
  - E.g. failure of the whole instance or a set of instances in the same data center.
- Service and cross-service
  - E.g. make a database unavailable that will impact other services.
- Business or application logic
  - E.g. a bug in a new product or feature.

Best Practices

- For chaos engineering to be effective, several principles need to be addressed.
- Understand the system: how subsystems interact?
- Enbrace failure: failures will always happen, it's better to plan ahead than trying to solving issues right after they appear.
- Establish steady-state behavior: how the system behave when running correctly?
- Identify real-world incidents: explore incidents that are likely to happen, e.g. network failures and bad configurations.
- Create a game day: schedule a day for multiple experiments to maximize findings.
- Use automation: make the process reproducible and less labor intensive.

Best Practices (Cont.)

- Keep in mind the experiments could be running in production environment with actual customers.
  - Minimize the blast radius so that the actual harm to customers is as minor as possible.
- Target a subset of services
- Run the experiment for a finite time
- Run the experiment away from peak traffic
- Run the experiment in the development environment (but understand this is different than the production environment).
- Have multiple runs of experiments to cover every component when system is continually changing.

Challenges and Limitations

- Human factors
  - Unintended customer impact
  - Organizational inertia and cultural challenge.
  - Cost and resource overhead need to be justified.
- Technology factors
  - False negatives where system appears resilient but not.
  - Limitations on scope and coverage of experiments as some faults are hard to inject.
  - Gap from observation to action.

Summary

- Use health check to monitor system status
- Apply chaos enigneering to understand system behavior when faults are presented and plan ahead.