CS 3551: Advanced Topics in Distributed Information Systems - Building Dependable Infrastructure

Day 1: Course Introduction

Dr. Amy Babay, Fall 2024

University of Pittsburgh

Department of Computer Science

School of Computing and Information

Today's Objectives

- Understand course **logistics and expectations**
- Build **context** for the rest of the course
  - What do we mean by *dependable infrastructure*?
  - What are some of the challenges in building dependable critical infrastructure and other distributed systems?
  - What are some high-level approaches to building dependable distributed systems?

Introductions

- Instructor: Amy Babay
  - PhD from Johns Hopkins University in 2018
    - “Timely, Reliable, and Cost-Effective Internet Transport using Structured Overlay Networks”: how can we provide the **network performance** needed for highly interactive applications?
    - Intrusion-tolerant SCADA for the power grid: how can we build computer systems that continue to work correctly despite compromises and **network attacks**?
  - Brief time in industry
    - Exploring commercial applications of PhD work
    - Infrastructure to simplify management of global overlay networks
  - Joined Pitt SCI in August 2019
    - Research focuses on building dependable critical infrastructure systems, supporting demanding new Internet services, and community-based environmental monitoring

Introductions

- Instructor: Amy Babay
  - Contact: Teams (best for quick questions) or email babay@pitt.edu
  - Office hours: by appointment

Course Logistics

- **Course meetings**
  - **11:00am – 12:15pm Tue/Thu**
  - **Sennott Square, Room 6516**
- **Course website**: course info, reading schedule
  - https://sites.pitt.edu/~babay/courses/cs3551/
  - (Link is posted in canvas)
- **Canvas**: assignment submission, announcements
- **Teams**: questions, discussion, project coordination

Workload and Grading

- ~16 paper reviews (20%)
  - Normally 1 review per class
- ~5 lab days (10%)
  - In-class, hands-on work with tools for specifying, implementing, and testing distributed systems
- Discussion participation (10%)

- 1 semester-long course project (50%)
  - May be done alone or in teams of any size. Project scope must match team size.
  - 10/3: Project proposals
  - 10/29 – 10/31: Project checkpoint presentations
  - 12/10 – 12/12: Final project presentations + Final project report, webpage, and artifacts delivery
- ~2 paper presentations (10%)
  - 1 in 1st half of course (sign up for pre-selected paper)
  - 1 in 2nd half of course (need to find and propose a paper related to your project topic)

Policies

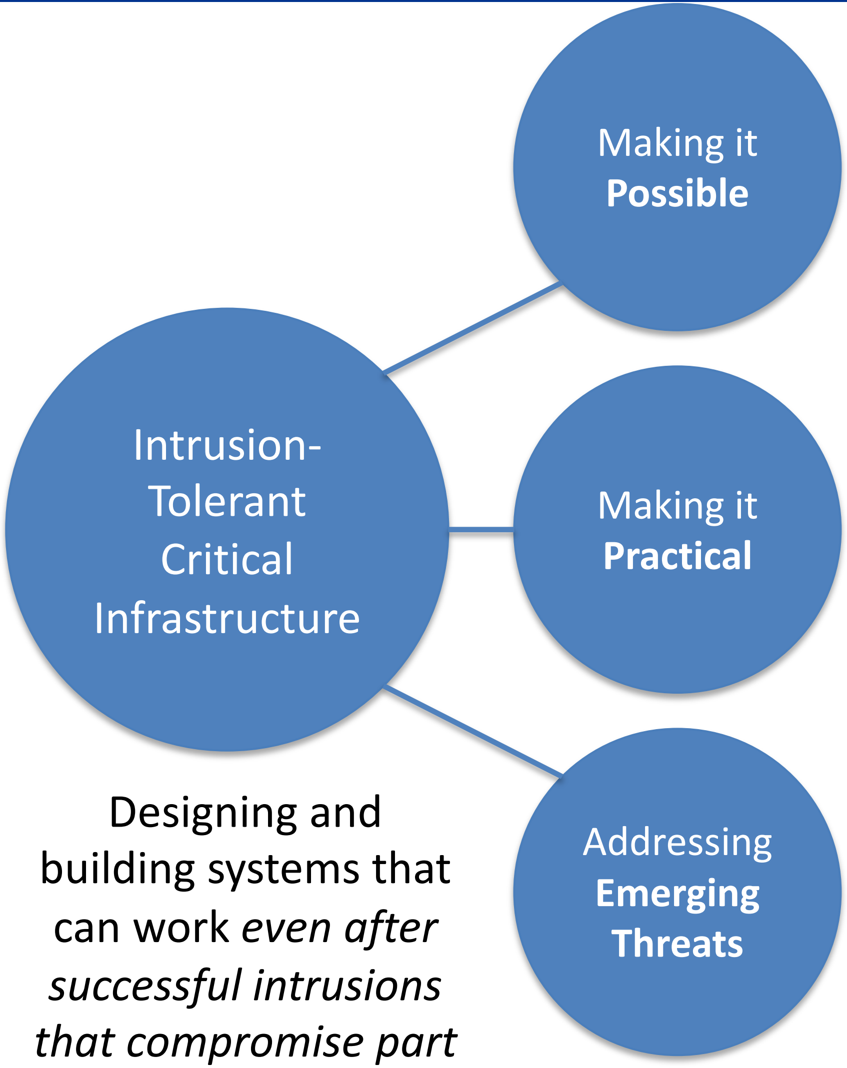
- See course website: https://sites.pitt.edu/~babay/courses/cs3551/policies.html

Questions?

Seminar theme: Building Dependable Infrastructure

Resilient Systems and Societies Lab

www.rsslab.io



Designing and building systems that can work *even after successful intrusions that compromise part of the system*

- **Spire** intrusion-tolerant SCADA for the power grid
- **Spines** intrusion-tolerant network

www.spire-sys.org
www.spines-org.github.io

- **Cloud-based** intrusion-tolerant SCADA systems reduce the cost of resilience, without exposing sensitive data to cloud providers — DSN 2021 (Best paper runner-up), SRDS 2023, Maher Khan PhD thesis (2024)
- **Digital twins** can provide a transition path to intrusion-tolerant architectures — DOE-funded Cyber Energy Center project
- **Seamless intrusion-tolerant networks** support legacy applications without changes — Defense Logistics Agency funded project

- **Compound threats** involve natural hazards combined with cyberattacks — DoD/DOE funded SERDP project, SRDS 2024
- **Integrated architecture** for power grid systems, provides intrusion tolerance from substation to control center

Motivation – Personal Experience

- Resilience to failures and attacks is crucial
- But:
  - Resilient system designs become complex
  - Complexity introduces more opportunities for errors
  - Proving correctness of protocols is challenging
    - New protocols are often introduced without rigorous proofs
  - There is often a big gap between the abstract specification of a protocol and its implementation in a real system

Motivation – White House Report

- Back to the Building Blocks: A Path Toward Secure and Measurable Software
- “A proactive approach that focuses on eliminating entire classes of vulnerabilities reduces the potential attack surface and results in more reliable code, less downtime, and more predictable systems.”

Motivation – White House Report

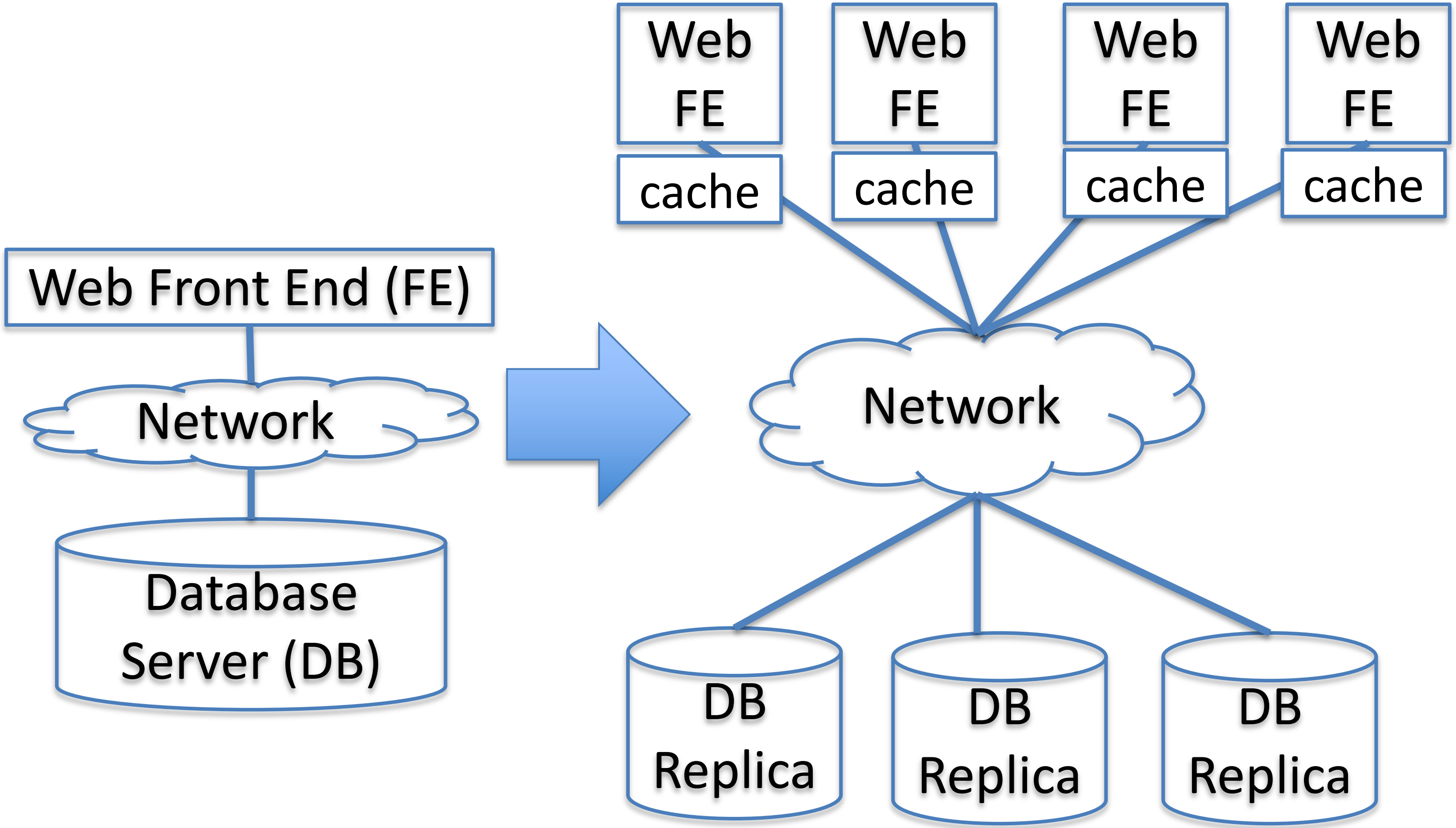
- Recommendations:
  - **Memory safe programming languages**
    - Avoid classic memory safety bugs (null pointer, buffer overflow, use after free) possible (and common) in C/C++
    - Challenge for systems where predictable low-latency performance is critical: garbage collection
    - DARPA TRACTOR program – “Translating All C to Rust”
  - Memory safe hardware
    - For embedded systems where moving to memory safe programming languages may not (yet) be feasible
  - **Formal methods**
    - **Prove** that software is correct (meets specific requirements)
    - Static analysis, model checking, assertion-based testing

Motivation – White House Report

- Recommendations:
  - Measuring the cybersecurity quality of software
    - How can developers identify/choose secure open source libraries to build on?
    - How can customers select secure products?
  - Policy: make *software manufacturers* responsible for vulnerabilities, not only software users
    - Echoed by software "users" in the power industry – rely on vendors for compliance, but limited incentives for vendors to fully meet requirements

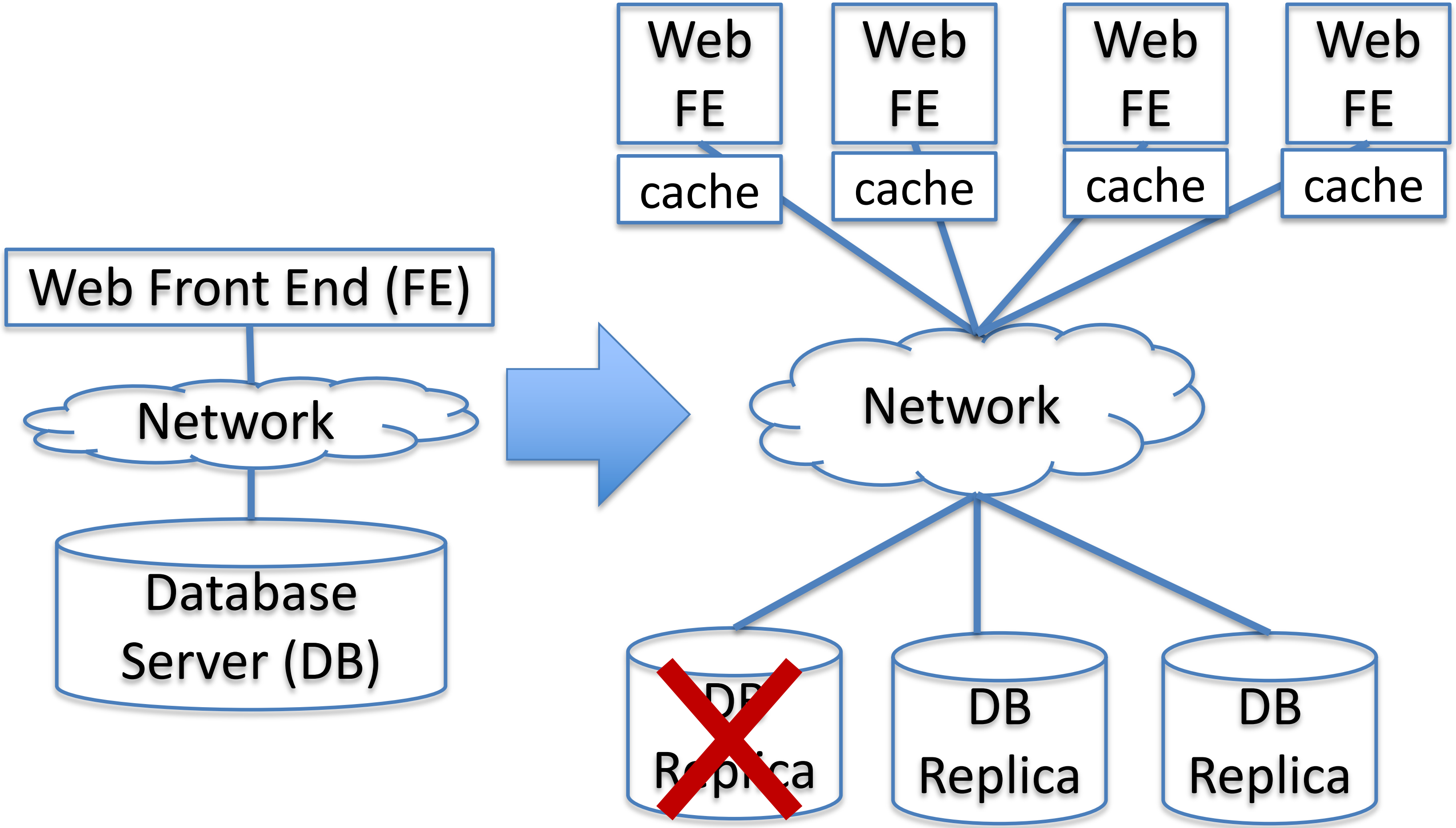
Overview: Fault Tolerance

Fault tolerance requires distribution



- To withstand failures, typically need to **replicate** services over multiple machines
  - Failure of a single replica does not impact availability of the service
  - Achieving this in practice can be challenging…need to synchronize replicas

Fault tolerance requires distribution



- Example: One DB replica fails
- No problem…another can take over
- But, can we guarantee that it has the same state?

Overview: Testing and Verifying Distributed Systems

Some slides adapted from Roxana Geambasu's Distributed Systems course @ Columbia:
https://systems.cs.columbia.edu/ds1-class/lectures/13-testing-model-checking.pdf

Properties of Distributed Systems – More on this next class!

- We want to ensure that systems we build maintain certain properties:
  - Safety (correctness at every step – nothing bad happens)
  - Liveness (eventually, something good will happen)
  - Performance (something will happen within a certain time or with a certain amount of resources)

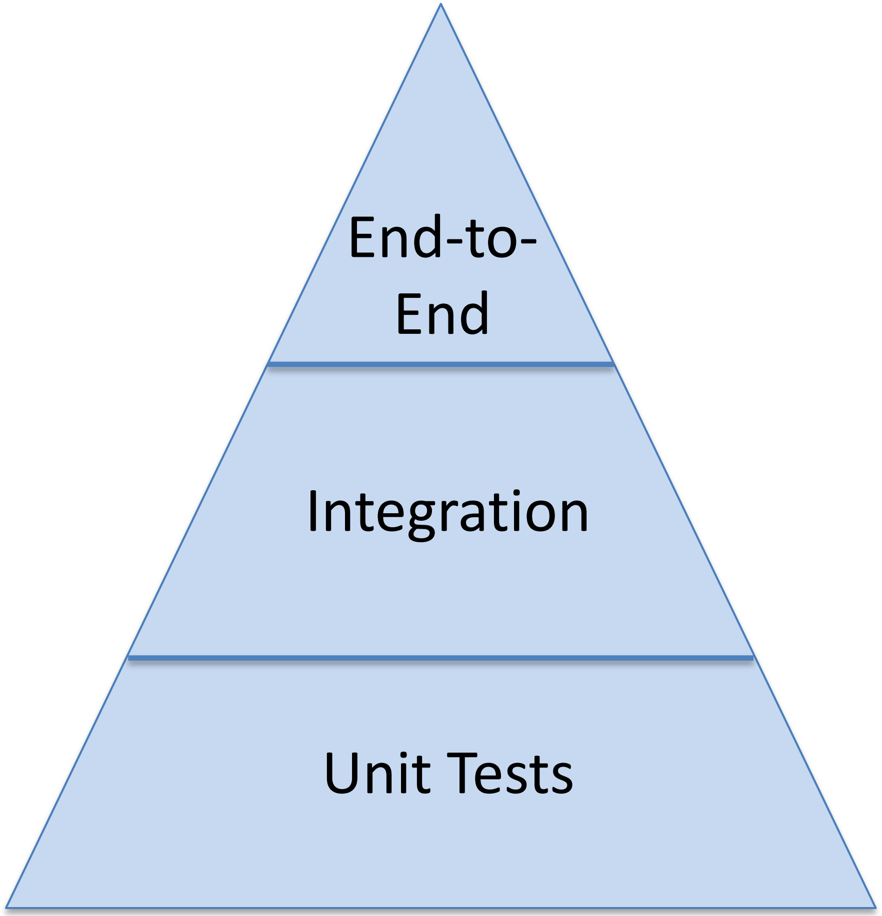
- How do we ensure a distributed system meets these properties in practice?

High-level View

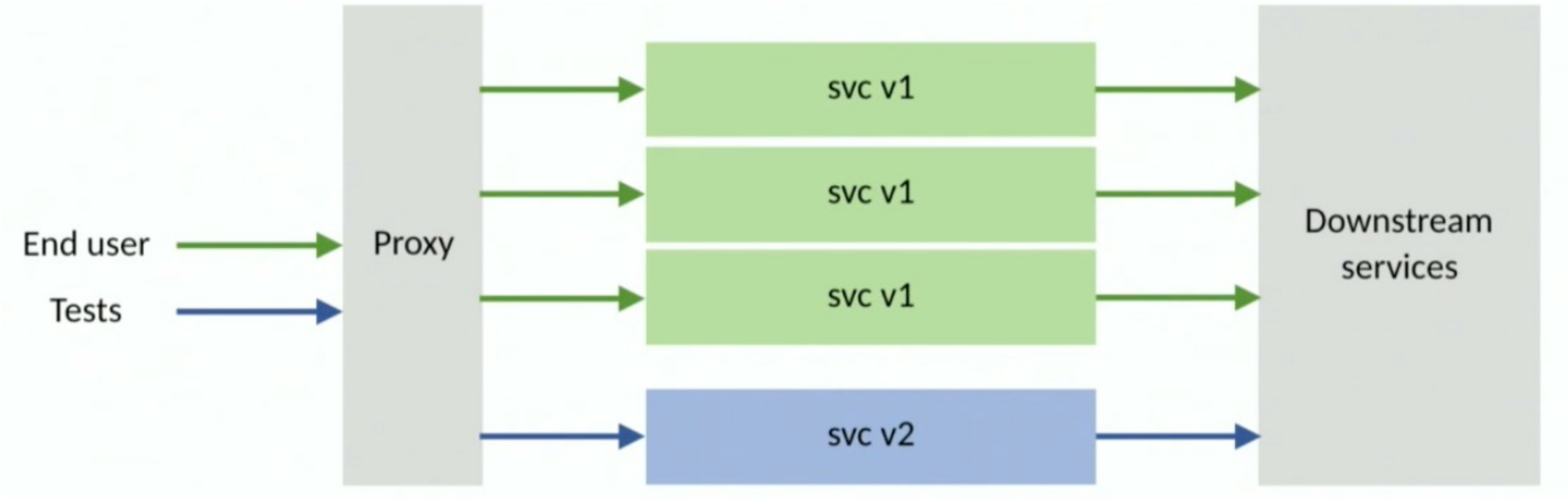
- **Testing**
  - Directly test the system implementation to evaluate whether it meets safety/liveness/performance properties
    - In the normal case and under failures
  - Best effort – typically impossible to test *all* possible cases
- **Formal Verification**
  - Comprehensive checking of safety and liveness properties
  - **Proves** that the system will behave correctly under stated assumptions
  - Usually applied on design, not on the implementation (often requires design to be written in formal specification language)
- See also: "The Verification of a Distributed System"

Types of Testing

- **Unit tests:**
  - Basis to catch most bugs pre-production
  - Test every function, module, microservice separately
  - Stub all other components (mocks, contract tests)
  - Aim for >95% line coverage
- **Integration:**
  - Test multiple integrated components, still with some stubbing for external dependencies
  - Often rely on growing list of scenarios
- **End-to-end:**
  - Run on deployment in production(-like) environment, often with mirrored traffic



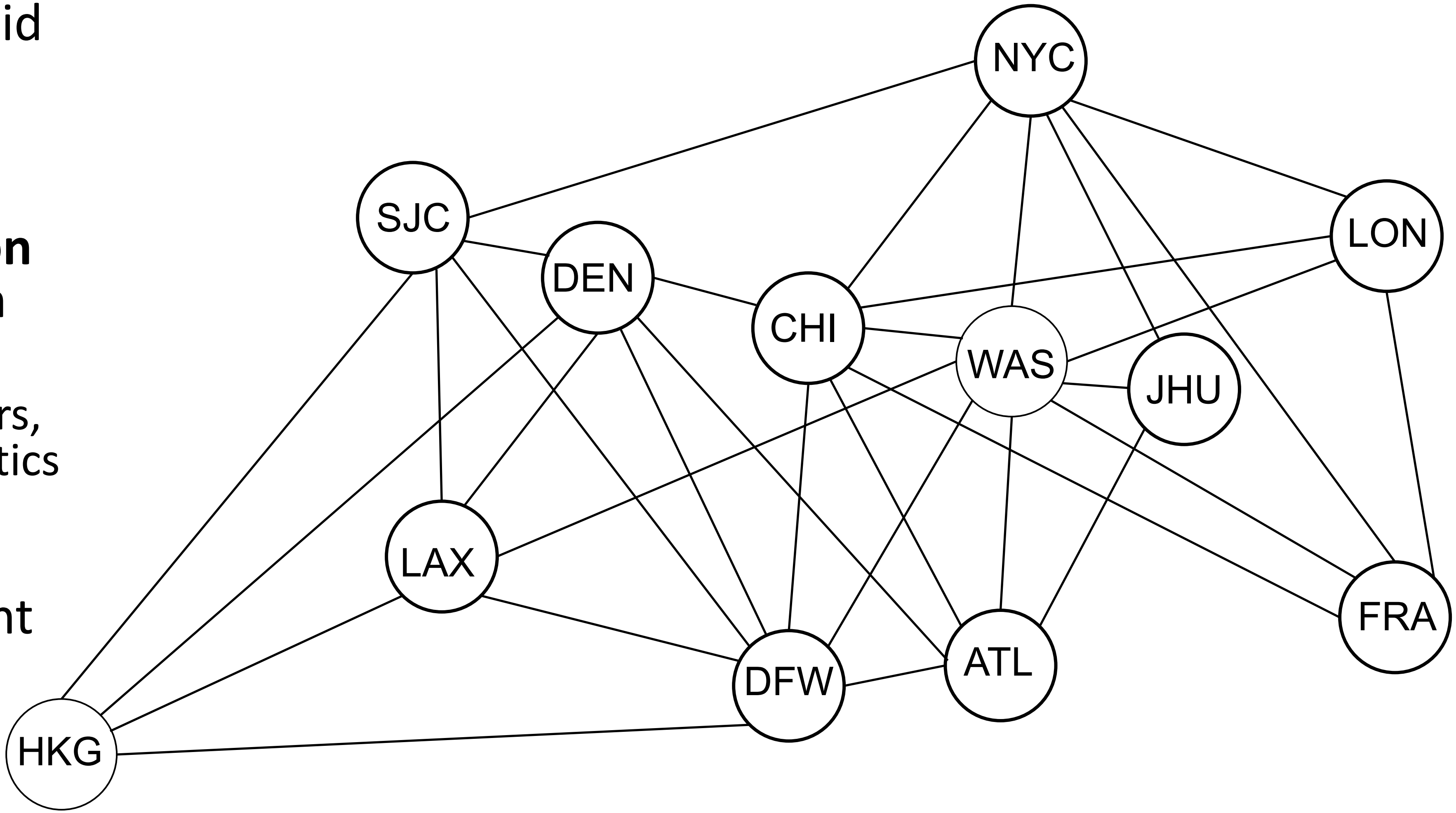
Shadowing / Traffic Mirroring



- Production traffic can be mirrored to *shadow* test service

Shadowing Example: Spines Intrusion-Tolerant Network

- Test that our group did with an **intrusion-tolerant network service**
- Shadowed **production monitoring traffic** on the LTN Global cloud
  - Status of data centers, network characteristics (latency, loss), client statuses, etc.
- 10 month deployment
- No messages lost, equally timely



Canary Deployments

- Incremental roll out of upgrades
- Start by replacing a few "canary" nodes, compare metrics/output with old version
  - Roll back if problems detected
  - Upgrade more nodes if all looks good

- Good practice, but generally tests only the ***common case***
  - Only tests under production conditions at the time the canaries are deployed; unlikely to cover failures, corner cases

Fault Injection

- Building confidence in correctness must include testing *under failure conditions*
- Perform **fault injection** for common distributed systems failure modes
  - Node failures
  - Faulty networks (latency, partitions)
  - Unsynchronized clocks
  - …

Fault Injection in Production - Chaos Engineering

- “Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production” (https://principlesofchaos.org/)
- Netflix Simian Army
  - **Chaos Monkey** (https://github.com/netflix/chaosmonkey)
    - “Randomly terminates virtual machine instances and containers that run inside of your production environment”
    - “Exposing engineers to failures more frequently incentivizes them to build resilient services”

Formal Verification

- Testing distributed systems is hard…
- Failures are often **non-deterministic** (and potentially difficult to reproduce)
  - You can run a test multiple times, but how do you know if it is **enough**?

- Recall chaos engineering goal: “to **build confidence** in the system’s capability to withstand turbulent conditions in production”
  - “build confidence” != prove correctness

Why is exhaustive testing so hard?

- Huge **state/behavior space** of possible executions
  - Concurrency
    - How many different interleavings of actions by multiple processes are possible? (a lot!)
  - Non-determinism
    - Now, each of those interleavings has multiple possible outcomes based on non-deterministic events (e.g. consider every possible place that a machine might fail)

Formal Specification and Model Checking: Process

(Ideally before implementing your system:)

1. Write a **specification** of the system in a formal specification language (think math).
2. Specify correctness properties as **invariants** on states or behaviors.
3. Use a **model checker** to exhaustively check that every state/behavior of the system, within a bounded range of configurations, satisfies your invariants.