Homework 3

ECE 590: Towards More Reliable Software

Jeff Fan || zf70@duke.edu

February 12, 2024

Question 1: Review the slides of the lecture on software architecture by Dr. Ivan Mura

a. List and explain ideas discussed that help fault avoidance.

Fault Prevention/Avoidance: This involves proactive measures to eliminate potential faults from the beginning of the software development process. The goal is to reduce the need for later fixes and patches.

Learning from Past Mistakes: This emphasizes the importance of understanding previous faults, analyzing their root causes, and applying this knowledge to prevent similar issues in future projects. By doing so, developers can identify patterns or recurring issues and take proactive steps to prevent similar faults in future projects. This approach turns past failures into valuable learning experiences, directly contributing to the development of more robust and reliable software.

Managing Complexity: Here, the focus is on handling the complexity inherent in software development. This includes ensuring clear communication, focused attention, and maintaining a clear understanding of objectives. By effectively managing complexity through clear objectives, focused attention, and structured approaches, developers can maintain a better overview of the system. This oversight helps in identifying potential fault points early and simplifies the development process, thereby reducing the likelihood of faults.

Methods for Fault Avoidance: This encompasses various techniques like disciplined requirements management, structured design, and automated code generation. It also involves using formal methods like model checking and proof of correctness, as well as practices like software reuse, design patterns, and pair programming.

Architectural Design Considerations: Identifying and steering clear of typical architectural mistakes, known as anti-patterns, like spaghetti code, which leads to tangled, unmanageable code structures. Emphasizing the need for well-thought-out architectural decisions that support maintainability, scalability, and

robustness. By avoiding common pitfalls and following best practices in software architecture, developers can build systems that are easier to test, maintain, and debug. This reduces the chance of faults occurring and makes any faults that do arise easier to address.

Design Principles: It advocates for adhering to principles like: **Don't Repeat Yourself (DRY)**: Avoiding redundancy in code to make maintenance easier and reduce the likelihood of inconsistencies. **Principle of Least Surprise**: Ensuring that components behave in expected ways to avoid unintended errors. **Single Responsibility Principle**: Each module or class should have one, and only one, reason to change, simplifying debugging and enhancement. **Open/Closed Principle**: Designing modules to be open for extension but closed for modification, thus promoting stability. **Low Coupling and High Cohesion**: Encouraging designs where modules have minimal dependencies on each other (low coupling) and where the elements within a module are strongly related (high cohesion).

[1] The slides of the lecture on software architecture by Dr. Ivan Mura

Question 2: Review the slides of the lecture by Dr. Veena Mendiratta

a. List and explain ideas discussed that help fault avoidance.

Microservices Architecture: It aids in fault avoidance primarily due to its structure of small, independent services. Each microservice is designed to perform a specific function. This modularity means that if one service encounters an issue, it does not necessarily impact the others. In a monolithic architecture, a single fault could cause widespread system failure. However, in a microservices setup, the isolation of services ensures that faults are contained, reducing the risk of a system-wide collapse. Additionally, the simplicity of individual microservices makes them easier to manage and debug, which also contributes to fault avoidance.

Service Mesh and Sidecars: These provide resilience by managing communication among microservices, ensuring reliable network functions. They act as intermediaries, handling network communication, service discovery, load balancing, and security protocols. This setup allows individual services to focus on their core functionality without being burdened by these complex interactions. By centralizing communication logic, Service Mesh and Sidecars reduce the likelihood

of faults caused by misconfigured or overwhelmed network communications between services. This results in a more robust and reliable system where service-to-service communication is less prone to failure.

Domain-Oriented Architecture: This approach emphasizes separation of concerns and clear domain boundaries, reducing the risk of widespread failures due to tightly coupled components. This separation ensures that issues within one domain do not easily propagate to others. It promotes a clearer understanding and management of each domain, reducing the chances of errors during development and maintenance. By isolating the domains, any faults that occur are contained within a specific area, preventing them from affecting the entire system. This approach leads to a more stable and reliable system overall.

b. List and explain ideas discussed that help fault tolerance.

Circuit Breakers and Timeouts: These mechanisms detect unresponsive services and prevent cascading failures by rerouting traffic or triggering retries.

Circuit Breakers monitor the success or failure of operations. If failures reach a certain threshold, the circuit breaker trips, and the system temporarily halts operations to that particular service. This prevents the system from being overwhelmed by continuous requests to a failing service, allowing it to maintain functionality in other areas.

Timeouts define how long a service will wait for a response from another service. If the timeout is reached without a response, the operation is aborted. This prevents the system from waiting indefinitely for a response, which could tie up resources and degrade overall system performance.

Resilience with Service Meshes and Sidecars: Service meshes offer a controlled environment to handle communication failures and service unavailability. Resilience in microservices architecture, enhanced by Service Meshes and Sidecars, contributes to fault tolerance by efficiently managing network communication and service interaction complexities. Service Meshes handle tasks like load balancing, service discovery, and failure recovery, allowing individual services to recover from failures more quickly. Sidecars, deployed alongside each microservice, abstract and manage network communication, making it easier to implement resilient communication patterns. This setup enables the system to maintain operations even when individual services or components fail, thus enhancing overall fault tolerance.

Model-Based Analysis: It helps in fault tolerance by providing a structured approach to predict and analyze system behavior under various failure scenarios. It involves creating models, like Continuous Time Markov Chains (CTMC),

which simulate different states of a system and transitions between these states, including failures and recoveries. By analyzing these models, engineers can identify potential weaknesses in the system and implement strategies to mitigate them. This proactive approach to understanding and preparing for failures ensures that the system can remain operational and recover quickly from disruptions, thereby increasing its overall fault tolerance.

[2] The slides of the lecture on software architecture by Dr. Veena Mendiratta