# Lecture 4

## Fault-Tolerant Design

ECE 422: Reliable and Secure Systems Design


Instructor: An Ran Chen
Term: 2024 Winter

# Schedule for today

- Key concepts from last class
- Fault tolerance
- Fault recovery techniques
  - Backward error recovery
  - Forward error recovery
- Multiple versions techniques
  - Recovery blocks
  - N-version programming
  - N self-checking programming

# SRE (Site Reliability Engineering)

SRE is a practice of automating IT operations tasks.

- Focuses on response time and software reliability
- Operations-oriented automation

Examples of IT operations tasks that are automated:

- Incident response
- Production system management
- Performance monitoring
- Emergency response

# SRE (Site Reliability Engineering)

SRE has three key components:

- Service Level Agreement (SLA)
  - Agreement that the business team make with the customer
- Service Level Objective (SLO)
  - Objectives that the SRE team agrees to meet
- Service Level Indicator (SLI)
  - Real metrics on the system performance

Site reliability engineers build Service Level Indicators to drive Service Level Objectives which are then used to negotiate the Service Level Agreements with the customer.

# SRE (Site Reliability Engineering)

There are two important -ability in SRE:

- Reliability is the likelihood that a system will perform its function without failure.
  - Calculated by MTBF (Mean Time Between Failure)
  - The higher MTBF, the more reliable and available the system becomes
- Availability

# Availability

Availability is the percentage of time that a system is operational.

- Expressed in terms of the number of “nines” in the availability percentage

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF}+\text{MTTR}}$$

A SLA is a contract that outlines the expectations and responsibilities of both parties, and includes performance metrics for services the business offers.

Examples of SLA:

- Availability

# How to improve availability?

To increase availability, there are two solutions:

- Reduce the frequency of faults
  - By upgrading the hardware, expensive
  - By continuously improving the software, slow and expensive
  - Challenge: we can never reduce the probability of faults to zero
- Design systems that continue to work despite some faulty components; this solution is called Fault Tolerance

# Software fault tolerance

Fault tolerance is the ability for systems to continue functioning as a whole despite the faults

- Example of fault tolerance: while a PDF reader may crash when editing a corrupted PDF file, the PDF reader will restart itself after saving the information
- Why? Building safety-critical systems where there is more requirements on building safe and reliable softwares
  - E.g., aircraft, electronic banking, automobiles
  - Failures in these systems can cause catastrophic consequences
- Fault tolerance is implemented and prioritized in these systems

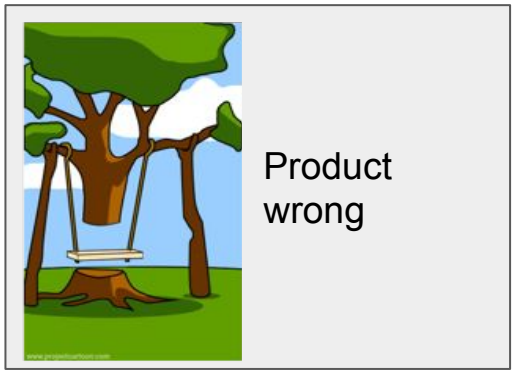
# State-of-the-practice on software faults

With the current state-of-the-practice, such as DevOps and SRE (People + Practices + Tools together):

- Fewer faults introduced, but the software is never assumed fault-free
- Some faults can be allowed to remain in the system

Unique challenges to software engineering, faults can happen due to design and specifications:

- Building the product wrong
- Building the wrong product





Images from www.projectcartoon.com

# Software fault vs hardware faults

- Physical vs logical faults
  - Software faults are logical faults written in programming language, which are difficult to visualize and detect.
  - Hardware faults are generally physical faults, which are limited by the number of physical components.
- Cause of faults
  - Software components fail due to bugs, hard to predict and monitor.
  - Hardware components generally fail due to wear and tear, which can be characterized and predicted over time.
- Resolution
  - Software faults are harder to fix, possibility of regression problems
  - Hardwares can simply be replaced when they break down

# Fault tolerance techniques

Fault tolerance techniques provide mechanisms to the software system to prevent failures in the present of a fault.

It consists of:

- Error detection: identification of an error state
- Error diagnosis: assessment of the damage caused by the error
- Error confinement: prevention of further damages
- Error recovery: replacing the error state with an error-free state

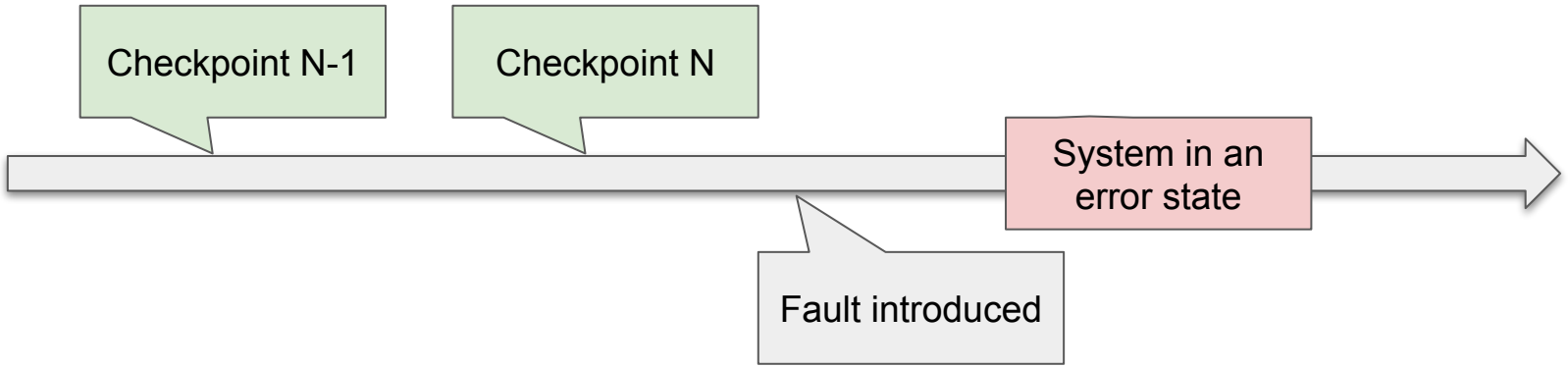
# Schedule for today

- Key concepts from last class
- Fault tolerance
- Fault recovery techniques
  - Backward error recovery
  - Forward error recovery
- Multiple versions techniques
  - Recovery blocks
  - N-version programming
  - N self-checking programming

# Backward error recovery

The system state is saved at predetermined recovery points (also known as checkpoints)

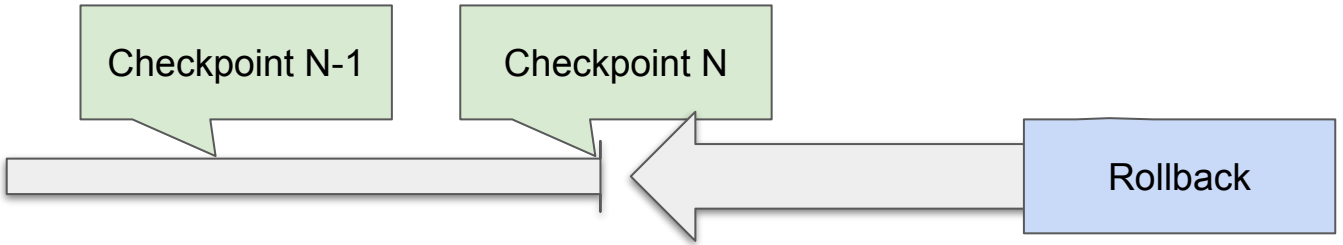
- Incremental checkpointing on error-free state
- When the system is in an error state, recover the system by rolling back to a previously saved state



# Backward error recovery

Example of implementations

- Roll back the previous checkpoint
- Wait and test the checkpoint for the next request
- Repeat until the system is in an error-free state



# Backward error recovery

Advantages

- No knowledge on the errors
- Genelizable solution
- Perfect for faults that are arbitrary or unpredictable

Disadvantages

- Requires significant resources
- Requires identification of consistent states (hard to define)
- No guarantee that the error will not persist when recovering (e.g., design errors)

# Forward error recovery

The system builds a new error-free state from which it can continue to operate with error compensation for the corrupted and missed data.

- Assume that the location and cause of errors can be accurately assessed
- Build a new error-free state from it

The system continues to run in a “degraded” mode

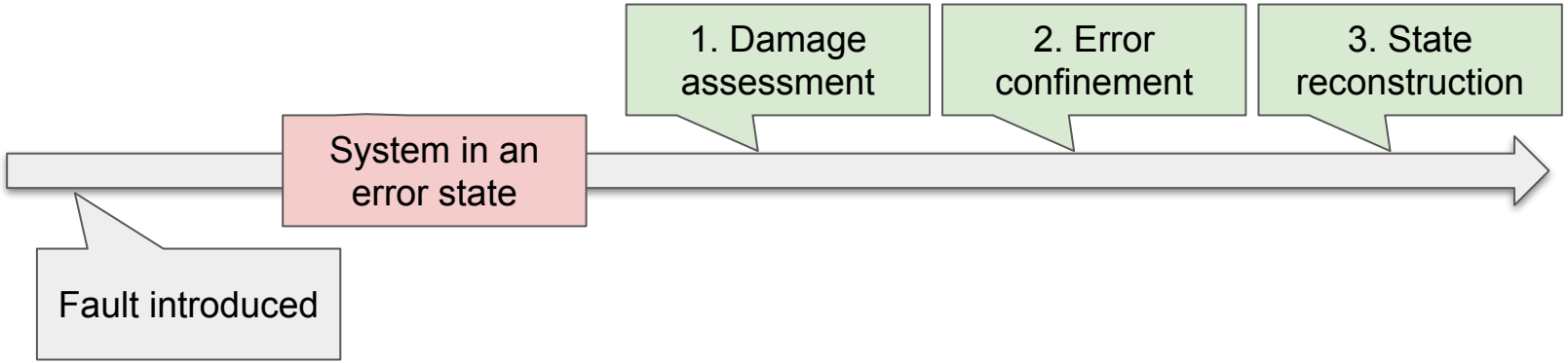
- Example: A printer operating in forward error recovery
  - Printing jobs can enter their critical sections
  - But not ordered by timestamp

# Error compensation

Error compensation is based on the algorithm that uses redundancy to select the correct/acceptable version.

- For example, in recovery blocks, the state transformation can be induced by switching from a failed state to a non-failed one executing the same task
- Error compensation may be applied all the time, whether or not an error occurred
    - Fault masking (e.g., N-version programming)

# Forward error recovery




Example of implementation

1. Determine to what extent the system has been corrupted
2. Structure the system to minimize the damage caused by the error
3. Make selective corrections to the system state

# Forward error recovery

Advantages

- More efficient use of time/memory

Disadvantages

- Design specifically for a particular system
- Depends on the accuracy of damage assessment and prediction
- Anticipate the potential errors in advance
- Not applicable for unpredicted errors

# Schedule for today

- Key concepts from last class
- Fault tolerance
- Fault recovery techniques
  - Backward error recovery
  - Forward error recovery
- Multiple versions techniques
  - Recovery blocks
  - N-version programming
  - N self-checking programming

# Classification of fault tolerance techniques

- Single version techniques: use the redundancy applied to a single version of a software module to detect and recover from faults.
  - Exception handling
- Multiple version techniques
  - Recovery blocks
  - N-version programming
  - N self-checking programming
- Multiple data representation techniques (not covered in class)
  - Retry blocks
  - N-copy programming

# Exception handling

Exception handling is the interruption of normal operation to handle abnormal responses.

Possible events triggering the exceptions:

- Interface exceptions
  - signaled by a module when it detects an invalid service request
- Local exceptions
  - signaled by a module when its fault detection mechanism detects a fault
- Failure exceptions
  - signaled by a module when it has detected that its fault recovery mechanism is enable to recover successfully

# Classification of fault tolerance techniques

- Single version techniques
  - Exception handling
- Multiple version techniques: use two or more versions of the same software module to achieve fault tolerance through software redundancy.
  - Recovery blocks
  - N-version programming
  - N self-checking programming
- Multiple data representation techniques (not covered in class)
  - Retry blocks
  - N-copy programming

# Multiple version techniques

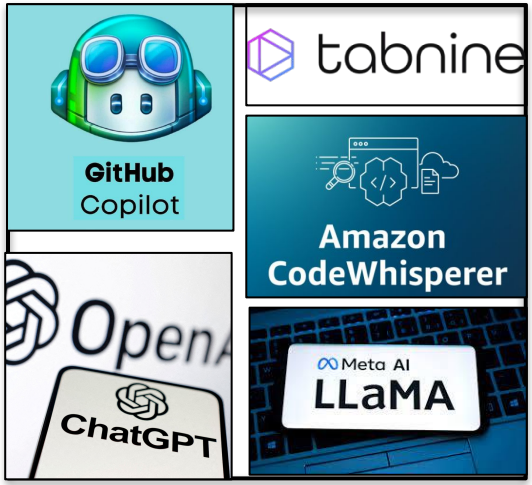
Recovery blocks (by Randell, 1975, 2714 citations)


N-version programming (by Chen and Avizie, 1978, 1001 citations)

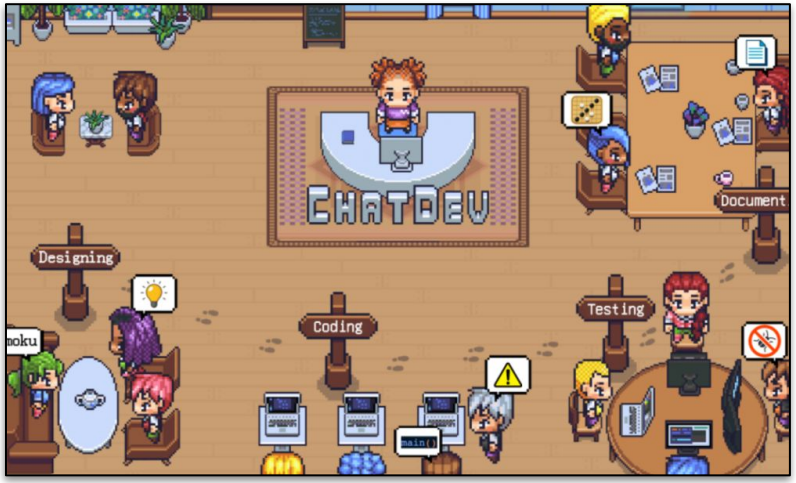

N self-checking programming (by Laprie, 1987, 117 citations and 1990, 549 citations)


# Multiple version techniques

Can we use fault tolerant techniques to assist in building better AI technologies?








https://github.com/OpenBMB/ChatDev

# Classification of fault tolerance techniques

- Single version techniques
  - Exception handling
- Multiple version techniques: use two or more versions of the same software module to achieve fault tolerance through software redundancy.
  - Recovery blocks
  - N-version programming
  - N self-checking programming
- Multiple data representation techniques (not covered in class)
  - Retry blocks
  - N-copy programming

# Recovery blocks

The recovery blocks technique

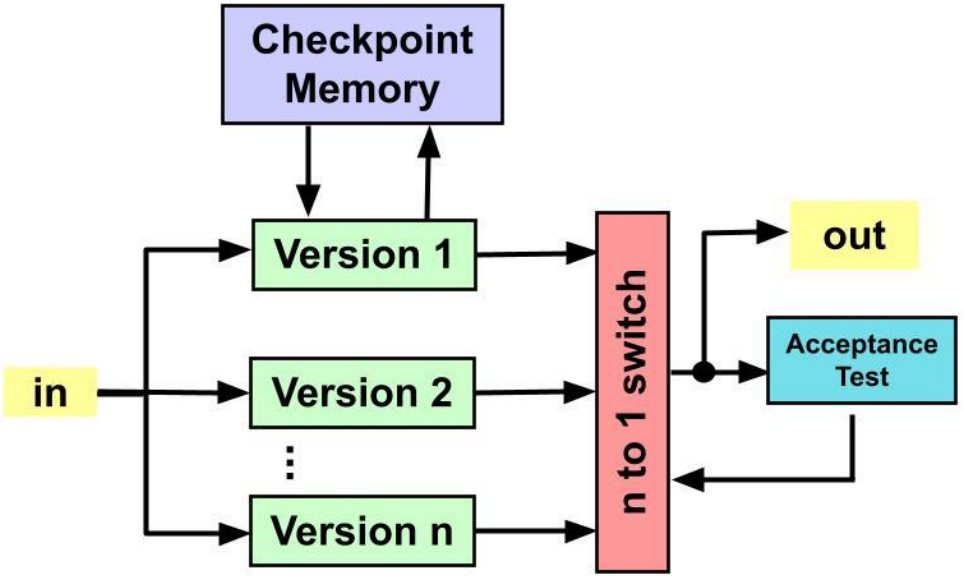
- Views the entire system as fault recoverable blocks
- Operates with an acceptance test, which confirms the results of different versions of the same function

There are three main actors in a recovery blocks technique:

- Primary version
  - The main implementation of a function
- Acceptance tests
  - Condition that the system is expected to meet
- Alternative versions
  - Each version is a different implementation of the same function

# Recovery blocks

Example




1. Run the primary version for the acceptance test
2. If the primary version fails, roll back the state of the system before the execution
3. Run the next version for the same acceptance test until there is an acceptable output
4. If none produce acceptable outputs, the system fails

# Recovery blocks

Cons:

- Reliability of the approach depends on the coverage and quality of the acceptance test (the decision mechanism for versions selection)
- State saving mechanisms can cause overheads
- Acceptance test needs to be small and fast to execute
  - Faster than the function itself
- Acceptance test used in an ad-hoc basis
  - If such tests exist, the faults would have already been fixed

# N-version programming

The N-version programming technique

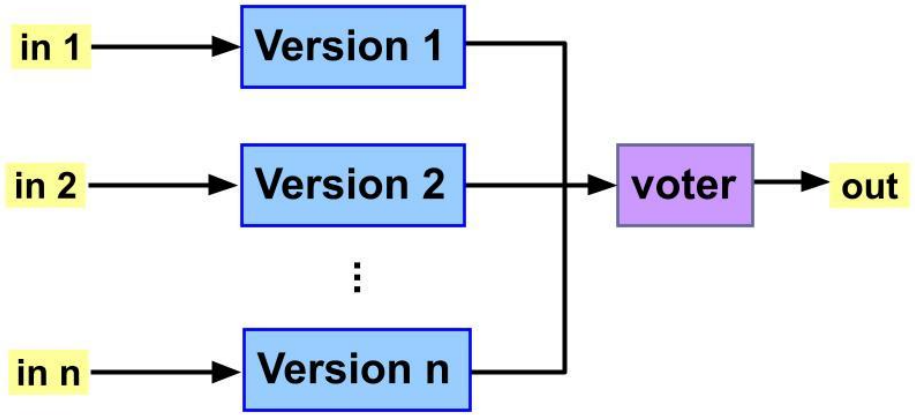
- Executes N versions of the function in parallel on identical input
- Returns the results based on a voting of the individual outputs

There are two main components:

- Versions
  - Different implementations of the same function
- Voter
  - A selection algorithm based on the individual outputs, or the consequences of the error
  - E.g., majority win, generalized median

# N-version programming

Example




1. Run all versions concurrently
2. Run the decision mechanism based on the voter (e.g., majority win)
3. Return the most common output from the individual versions

# N-version programming

Intuition behind N-version programming

- Have N teams of developers managing their own implementation of the software.
- If one of the versions fail, we can use the others as backups.
- The probability of two or more versions of a software system containing the same fault is very low.

Cons:

- The technique depends on design diversity, which may lead to other problems (e.g., specification/design errors)

# Recovery Blocks vs N-version programming

N-version programming has the versions executed concurrently, while the recovery blocks execute the versions in sequence.

- In real-time systems, cost in time is always prioritized.
- N-version programming is generally more applicable

Different decision mechanisms on version selection

- N-version programming run a single decision mechanism (the voter)
- Recovery blocks require each version to run the acceptance test

Recovery blocks assume that a single acceptance test is enough to detect the fault

- Not always the case in real-world applications

# N self-checking programming

The N self-checking programming technique combines the recovery blocks and N-version programming together.

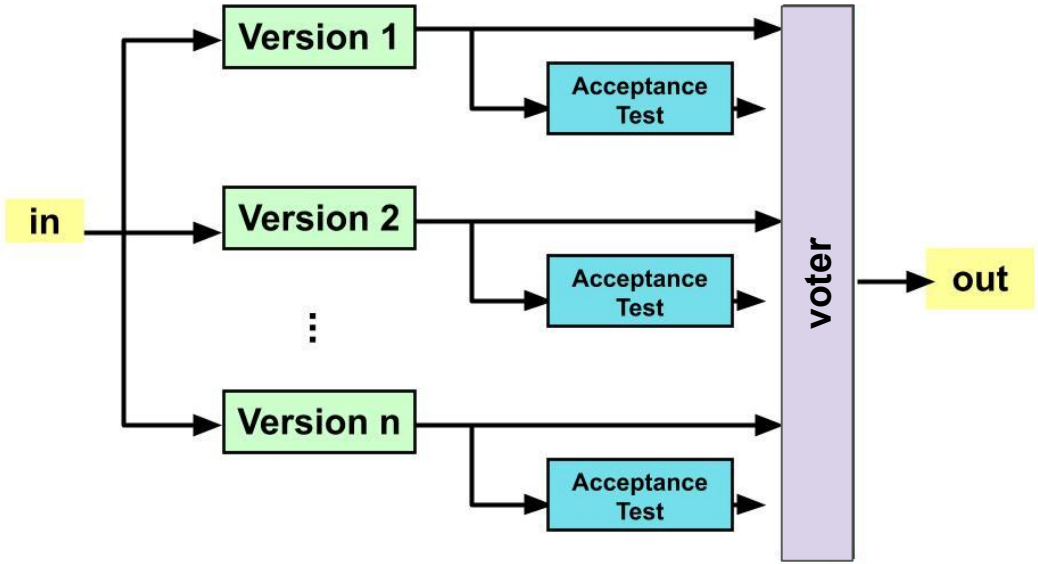
- Executes N versions of the function concurrently, or in sequence on identical input
- Return the results based on the acceptance tests

There are two main components:

- Versions
- Acceptance tests
    - Separate tests for each version

# N self-checking programming

Example: N self-checking by acceptance tests




1. Run all versions concurrently or sequentially
2. Run the corresponding acceptance test for each version
3. Run the voter
4. Return the majority agreement as output (only applied to versions that have passed their acceptance test)

# N self-checking programming

Cons:

- Include more overheads than N-version programming and “N” times the overhead of recovery blocks
- Vulnerable against design and specification errors
- Extra efforts in deriving individual acceptance tests

Example of applications:

- Airbus A-340 airplane (put in service 30 years ago)

# Other challenges on multiple version techniques

- Decision mechanisms themselves must be free from faults
  - do not involve an alternative mechanism
- Difficult to maintain and document
  - “Versions” can range from a few lines of code to an entire program
- Difficult to generalize
  - Design choices are made based on the available resources and on the specific applications

# Questions

- What is the goal of software fault tolerance?
- Name the two error recovery strategies, and briefly explain how they work.
- Name the three multiple version techniques, and briefly explain how they work.