Dependable and Distributed Systems

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 3 - Software Fault Tolerance

Software Fault Tolerance

Given a specification $SPEC$, and a program P that satisfies $SPEC$, and a fault model F

Consider the fault model F to be a function that transforms P into a program $P' = F(P)$

P' is said to be F-tolerant if P' satisfies $SPEC$

Now, we know from our formal methods that a system specification consists of two parts:

Safety

Liveness

Software Fault Tolerance

What if P' satisfies only safety or liveness but not both?

If P' satisfies *safety only* in presence of F, then P' is said to be **fail-safe F-tolerant**.

If P' satisfies *liveness only* in presence of F, then P' is said to be **non-masking F-tolerant**.

If P' does *not satisfy either safety or liveness* in presence of F, it is called **F-intolerant**.

F-tolerant is also known as **masking F-tolerant**

Designing For Fault Tolerance Classes

$$\begin{tikzcd}
 & \text{Masking fault tolerance} \arrow[dl] \arrow[dr] & \\
\text{Non-masking fault tolerance} \arrow[dr] & & \text{Fail-safe fault tolerance} \arrow[dl] \\
 & \text{Fault intolerance} &
\end{tikzcd}$$

Stronger than

$\longrightarrow$

Notes On Fault Tolerance Classes

What we have talked about up until now can be applied to hardware too

How can we enforce fail-safety or non-masking fault tolerance?

Is it possible to design program such that these properties are enforced?

If yes, how? What do we need?

Program Components

To satisfy safety and liveness, we need only two program components

Detectors

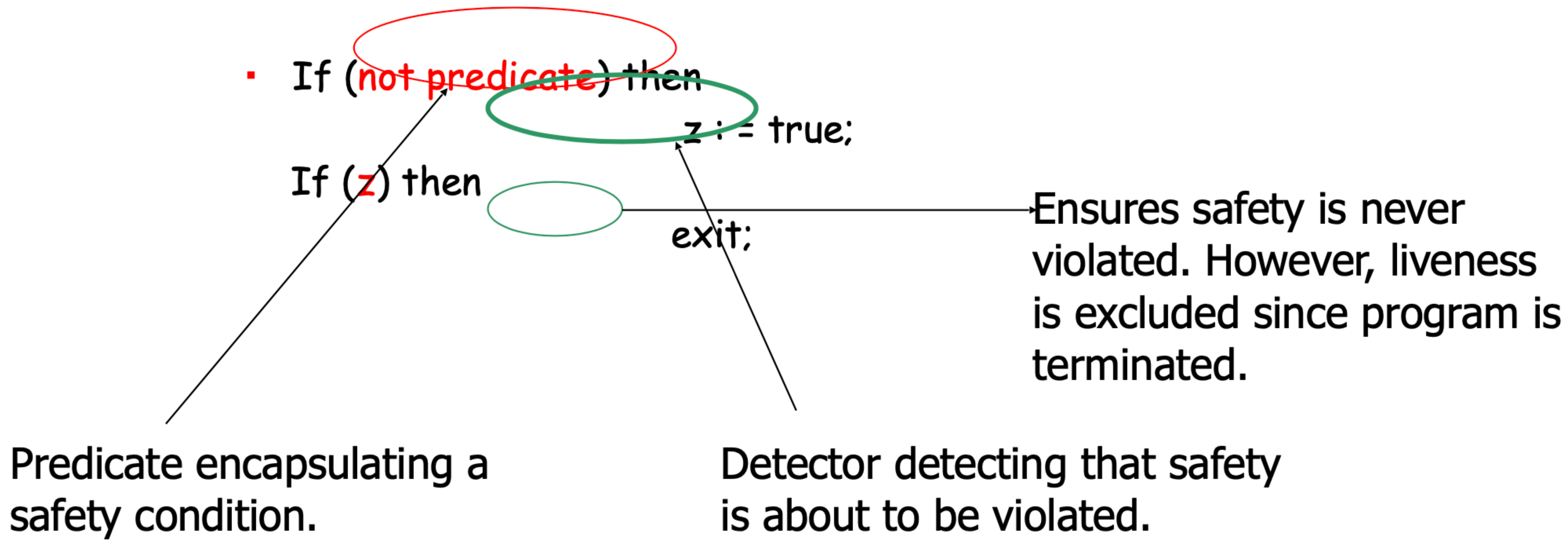
Correctors

A **detector** is a class of program components that asserts the validity of a predicate in a running program.

A **corrector** is a class of program components that imposes a given predicate on a running program

Detectors

In its simplest form:



```
If (not predicate) then
                  z := true;
If (z) then
                  exit;
```

Ensures safety is never violated. However, liveness is excluded since program is terminated.

Predicate encapsulating a safety condition.

Detector detecting that safety is about to be violated.

Correctors

In its simplest form:

```
▪ If (not safety) then
                      enforce predicate;
```

Predicate enforcement techniques ensure that invariant is satisfied again, after it has been violated. Remember invariant captures the safety notion for the program.

Hence, correctors try to reinstate the invariant again once it has been violated.

Because of this, the eventual program satisfies liveness, but not safety (safety is also eventually satisfied)

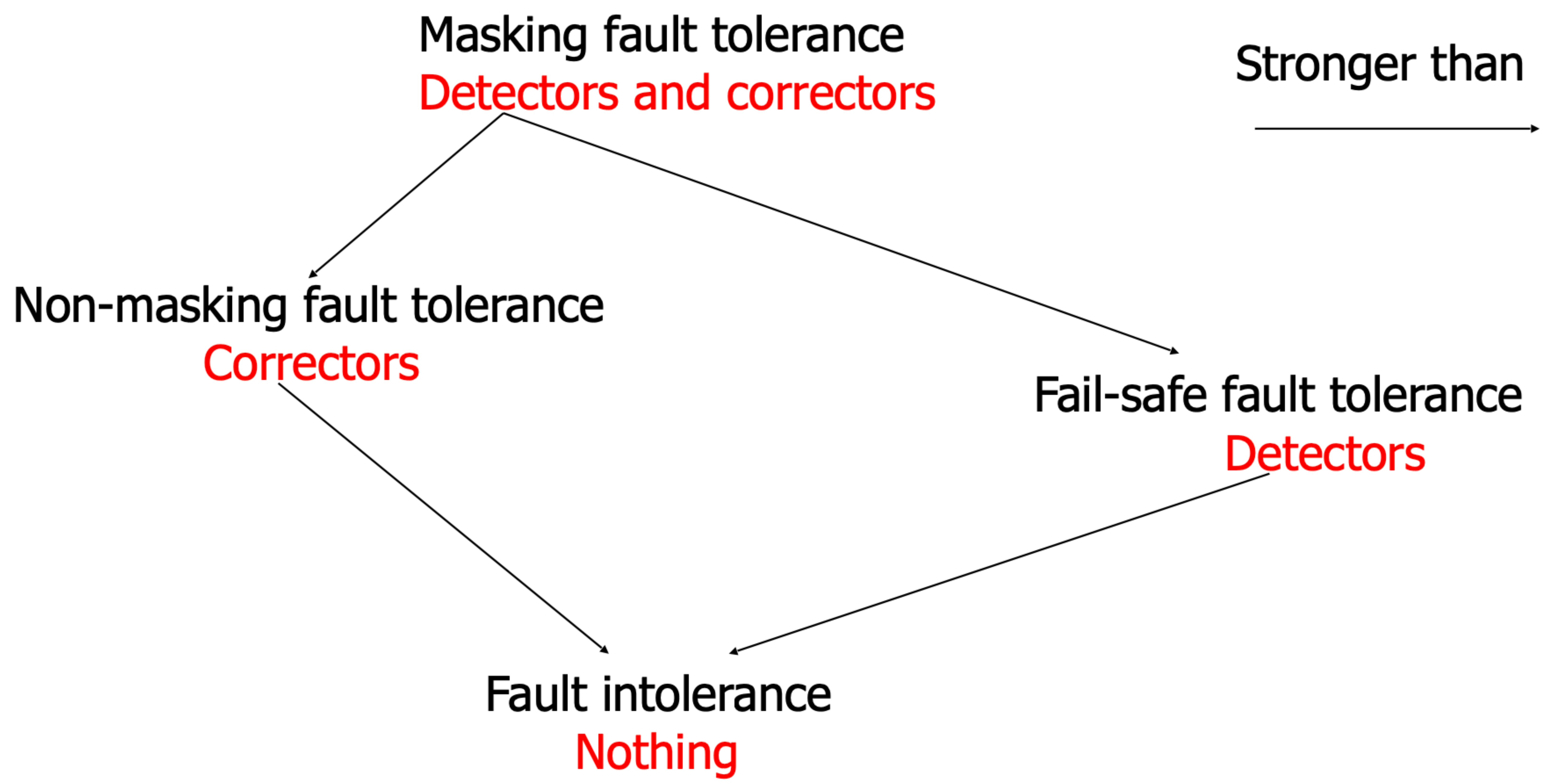
Important Results In Software Dependability

Detectors are both necessary and sufficient to design fail-safe fault tolerance

Correctors are both necessary and sufficient to design non-masking fault tolerance

Detectors and **correctors** are both necessary and sufficient to design masking fault tolerance

Designing For Fault Tolerance Classes



Phases of Fault Tolerance

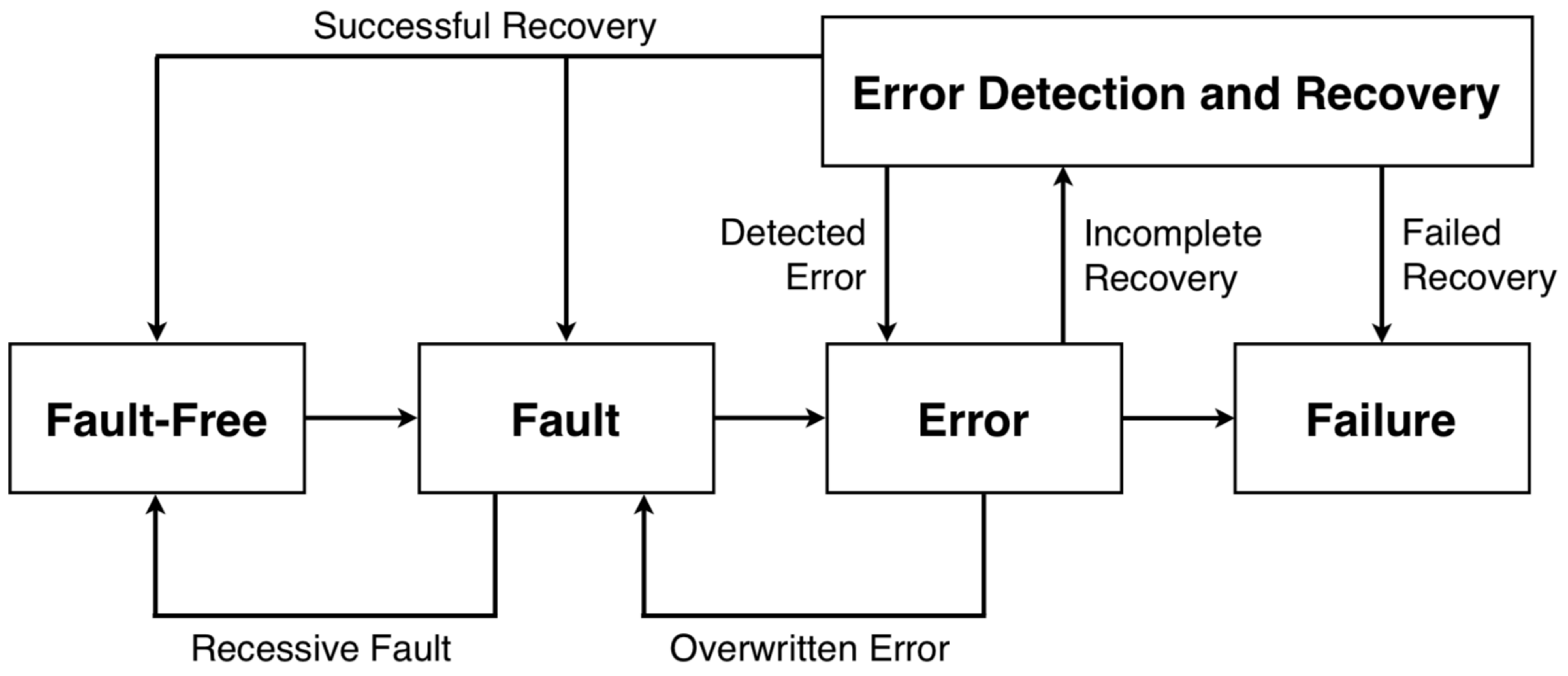
Error detection – achieved by detector components

Error recovery – achieved by corrector components

Damage assessment – investigate how far error has propagated, e.g., how many modules affected

More detectors, less damage!

Fault treatment – usually done offline to rectify problem



Fault Tolerance Techniques

There are several techniques for providing fault tolerance (fail-safe, non-masking or masking)

There are generally classified according to single module or multiple modules

Some examples of these where is it is easy to see the detection and correction elements include run-time checks and exception handlers

Detector Example - Run-time Checks

Used for detecting errors, hence used in fail-safe fault tolerance design

Many flavours:

Replication checks – compare outputs of matching modules
Timing checks – use of timers to check timing constraints
Reversal checks – reverse output and check against inputs
Coding checks – parity, Hamming codes, etc.
Reasonable checks – use semantic properties of data
Structural checks – redundancy in data structures
Validity checks – divide by 0, check array bounds, overflow

Corrector Example - Exception Handlers

Detectors raise exception (interrupt signal)

Interruption of normal operation to handle exceptional events.

Three main classes:

Interface exception – Invalid service request detected by interface detectors and corrected by service requestor

Local / internal exception – Problems with own internal operations detected by local detectors and corrected by local correctors

Failure exception – When internal errors propagate to interface, and detected by interface detectors. Global correction may be needed, e.g., look for another server

Recovering From Errors

Two main classes: **forward error recovery** and **backward error recovery**

Forward error recovery

Upon detection of error, program attempts to get into a state which is no longer erroneous (we will see how recovery blocks enable this)

Backward error recovery

Upon detection of error, program rolls back to a previously "recorded" good point (we will see how checkpoints enable this) from which it can restart executing

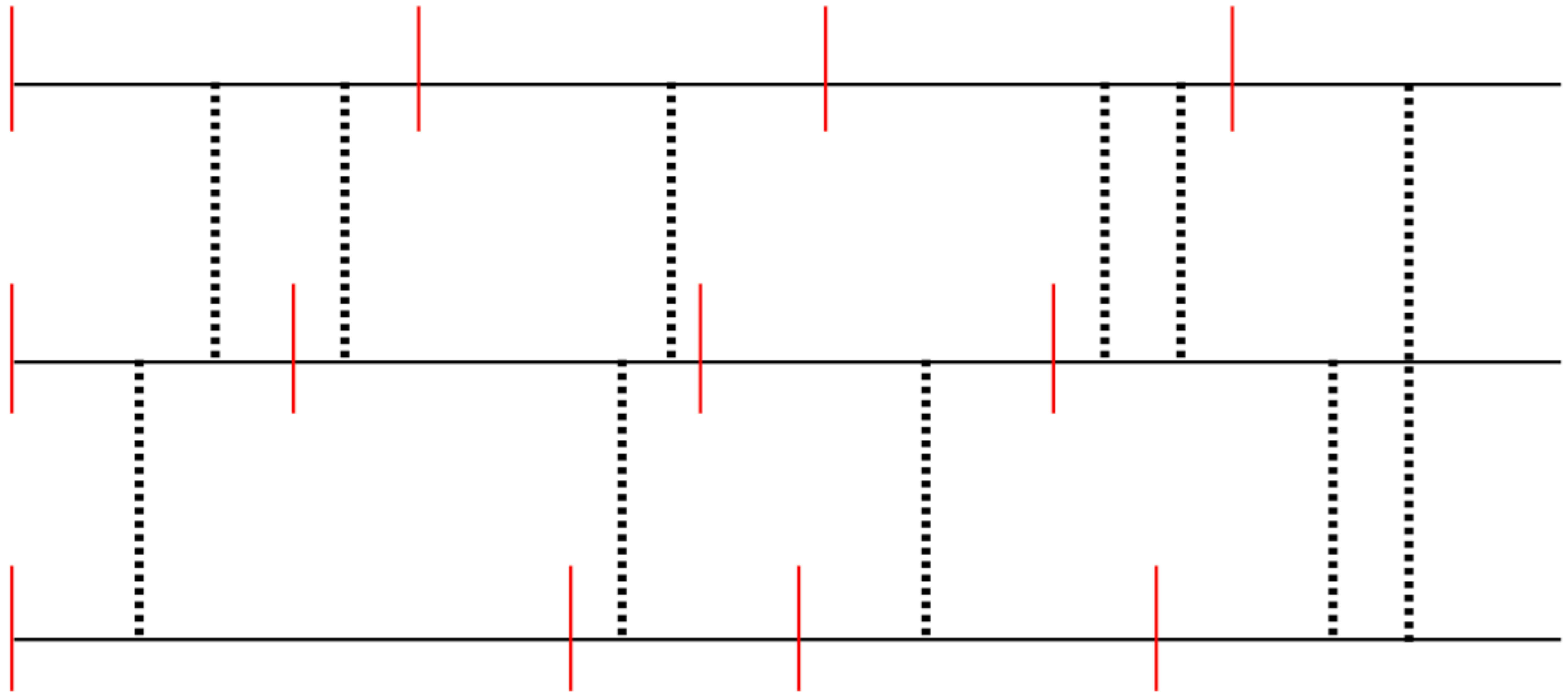
Checkpointing

How Do We Take Checkpoints?

How can we solve problems relating to domino effects?

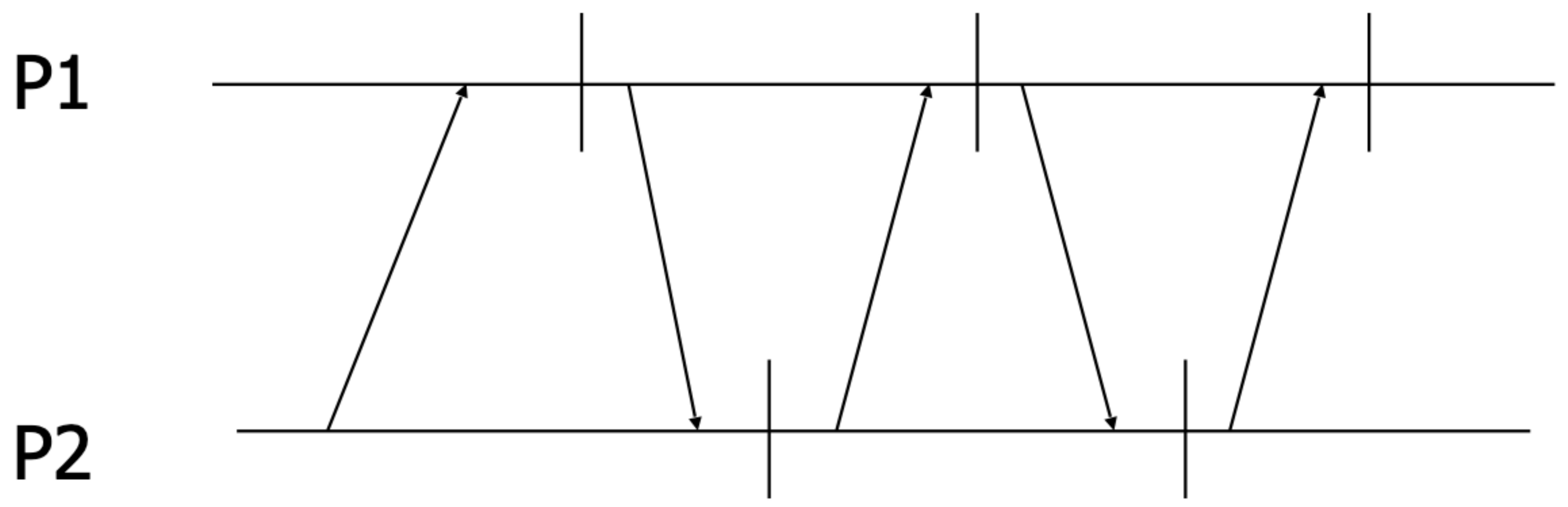
Checkpoints must be taken in a con

We need a checkpointing scheme t



Recovery Line

A recovery line is a set of checkpoints across all processes to which the programs can be rolled back, in the event of failure, to ensure consistent error-free state of the system



Recovery Line - What Can We Observe?

OK to checkpoint after a message send

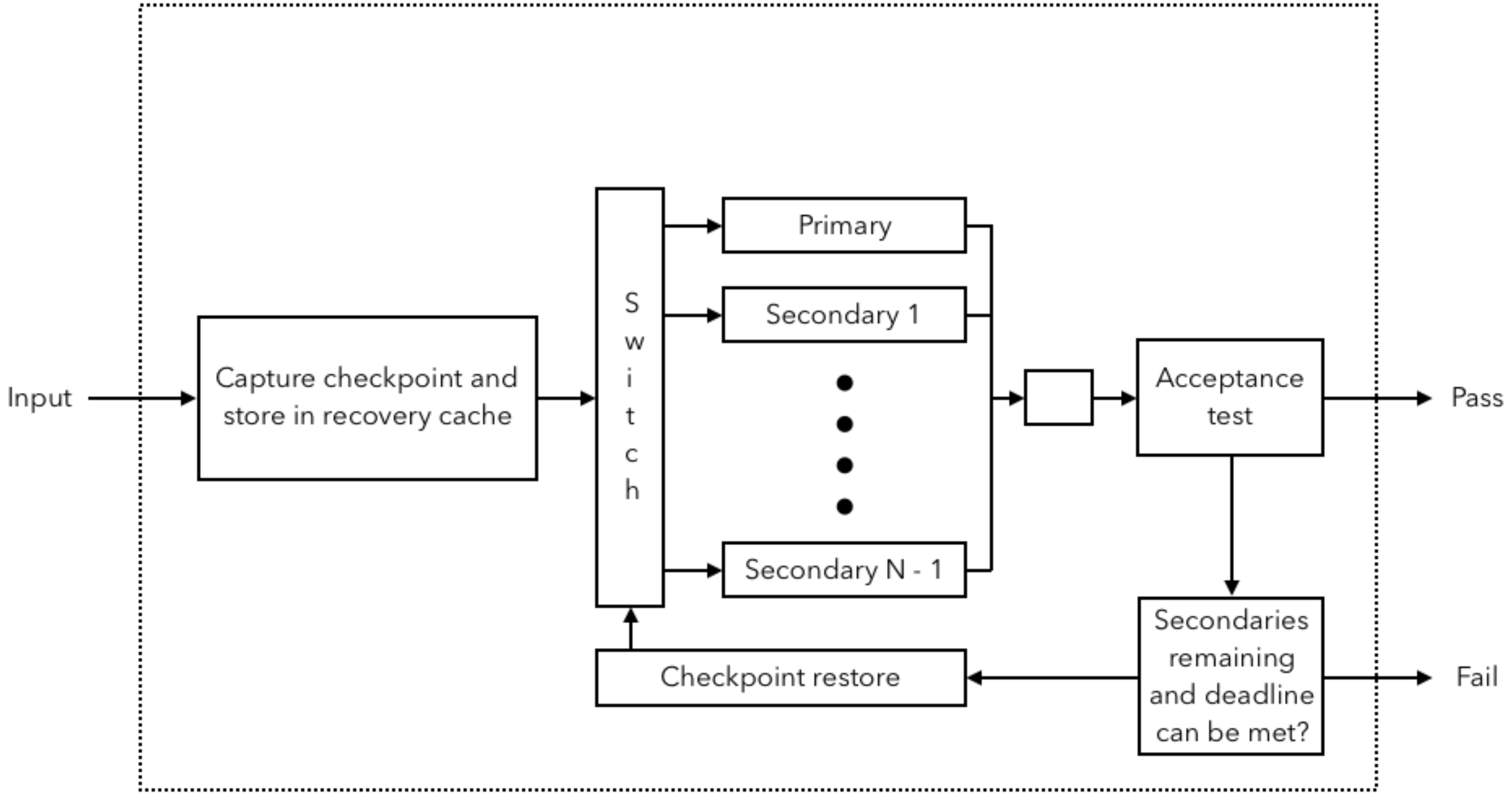
Not ok to checkpoint prior to a message receive

A recovery line is said to be consistent if there are no messages that originate (temporally speaking) after the line and terminate before it

In other words, along the recovery line, there can not be a receive without a corresponding send - the reverse is allowed, because temporally speaking, receive will occur after send

Recovery Blocks

How Can We Design Recovery Blocks?


Input
Capture checkpoint and store in recovery cache
S w i t c h
Primary
Secondary 1
Secondary N - 1
Acceptance test
Pass
Secondaries remaining and deadline can be met?
Fail
Checkpoint restore

Recovery Blocks

One hardware channel and N software components

Secondary components perform similar function but in a different way

Acceptance testing decides whether results are acceptable

If primary component fail acceptance test:, we reload a checkpoint, and execute a secondary from a "clean" state

Checkpoint is a snapshot of the system state at the point of primary execution

Recovery Blocks - When Can Acceptance Testing Reject?

- Error in module operation
- Time out expired, i.e., not terminated
- Raised exception due to, say, division-by-zero
- If not accepted, recovery procedure does:
    - Restore system state (example of backwards recovery)
    - Execute first available secondary module
    - When complete failure, exception / error raised to the environment

A Possible Implementation

```
ENSURE good_enough_result(possible_result)
BY
    possible_result := prim_module();
ELSE_BY
    possible_result := fst_sec_module();
ELSE_ERROR
    possible_result := fail();
END
```

Recovery Blocks

Redundancy in space (multiple versions) and time (executing the various versions)

Tolerate design flaws / bugs and hardware faults

Difficult to design good acceptance checks

- Can have problems of false positives / negatives
- Analysis has shown that this is the most crucial aspect of the recovery block scheme
- Imperfect acceptance test can even decrease reliability of the entire system compared to one with a highly reliable primary only
- Acceptance needs to accept results from last module, which can be possibly least reliable - what can we do?

Nested Recovery Blocks

```
ENSURE True
      BY
            ENSURE primary-acceptance-test()
            BY
                  primary-module();
            END
      ELSEBY
            ENSURE fst-secondary-acceptance-test()
            BY
                  fst-secondary-module()
            ELSE_ERROR
                  fail();
      ...
      ELSE_ERROR
            fail();
END
```

N-Version Programming

N-Version Programming

N-Version programming (NVP) is defined as the independent generation and utilisation of N functionally equivalent programs, where $N \geq 2$

Typically implemented with majority voting but this isn't strictly required

Characterised as N hardware channels and N software channels

N-Version Programming Without Majority Voting

When $N = 2$ majority voting is not possible

- Logging and error detection are preferable in some applications

Lack of agreement indicates that some problem occurred

- Error detection for situations where it is enough to raise a flag and trigger the desired correction mechanism

N-Version Programming Without Majority Voting

Generally impossible to precisely identify the erroneous version but we have still detected an error

Restart or retry

Starting state is know to be a safe state

Specification can be re-enforced once execution resumes from that safe state, hence the restart provides liveness

We can provide masking fault tolerance (program remained safe and liveness is eventually satisfied) using N-version programming, even without majority voting

N-Version Programming Without Majority Voting

Transition to a predefined safe state followed by later retries

Similar line of reasoning to restart, only a predefined safe state for a whole system (when you don't know what state you're coming from) can be difficult to identify

Can be used where infrequent "halt" is acceptable (fail-safe fault tolerance)

However, not recommended where continuous provision of service, *i.e.*, *liveness*, is critical, e.g., avionics or traffic lights

N-Version Programming Without Majority Voting

Reliance on a designating a version as being more reliable

- Could be a form non-masking fault tolerance, especially if output violates specification
- Liveness is preserved
- Routinely used in domains where liveness is more important than safety

Diagnostic selection

- Selection of “best” based on the analysis of the output
- Typically done by a diagnostic program - what does this means for safety and liveness?

N-Version Programming With Majority Voting

$N \geq 3$ but often $N = 3$

Then a majority voting logic is applied

- Three independent programs each providing identical output formats
- Acceptance program that evaluates the first output requirement and selects the result for the N-version output
- A driver invokes the above points and provides the N-version output to other programs or physical system

NVP Notes

Simple in its nature provides broad applicability

All versions need to be synchronised

- It is required to compare results from the “rounds”, otherwise, comparisons do not make sense
- May require long wait states for synchronisation the results

The N hardware channels may have very different properties

NVP Notes

Accuracy between versions may differ

- The N versions are often implemented using different programming languages or numerical methods, particularly in scientific and financial computing
- Each may have different floating point accuracy or mechanisms for the provision of its output
- It is necessary to find a meaningful notion of equality among version
- Most frequent errors are not in the N versions themselves

NVP Notes

If comparisons are rare, long wait could be acceptable

- Otherwise, there might be a performance penalty when providing fault tolerance
- Versions can rarely be fully independent
- General aim to minimise comparisons, however it should be possible to make a comparison without violating tolerances on accuracy or incurring excessive waiting times

Fault Tolerant Operations In NVP

Results are being provided to each computer and compared locally

Communicating the comparisons to the other computers

Analysing the comparisons and if they are in agreement, provide majority as output for the program

In the case that these do no agree, all the above steps should be repeated or exception handling will have to be invoked.

NVP Assumptions

The NVP axiom: NVP is most efficient if version failures occur independently

NVP will not work if all versions fail on same inputs

NVP will not work if fail in similar ways

A Case Study - Testing The NVP Axiom

A large-scale experiment in the effectiveness of NVP, operated by the Computer Science Department of University of Virginia and University of California at Irvine

One requirements specification for an anti-missile detection and response system

All programs written had to produce same output for the same inputs

Each program was subjected to 1,000,000 randomly selected tests

A Case Study - The Application

Anti-missile system for an aerospace company

Program required to read some a data set representing radar reflection

Using a collection of decision it had to decide whether the reflections originated from an object that represents a threat

If the object appeared as a threat, a signal was sent to launch the interceptor

Conditions were heavily parameterised (named Launched Interceptor Conditions)

A Case Study - Computing Resources

Programs written in Pascal

UVa: University Hull V-mode Pascal compiler for Prime computer using PRIMOS

UCI: Berkeley PC for VAX 11/750 using UNIX

Programming had the specification explained and were provided with access to the same resources

Any potential flaw in the specification was broadcasted to all programmers

A Case Study - Development

Each student was supplied with 15 input data sets with expected outputs

Acceptance test used to determine when a program is complete

Each program had its own acceptance test

To prevent problems with floating point operations, all programmers were supplied with a function to perform comparisons of real quantities with limited precisions, which needed to be used

A Case Study - Golden Version

Golden Run

- A non-faulty version which can be treat as being an omniscient version
- Output of all versions were compared against the output of gold version
- Thoroughly tested, extensive walk-through to ensure bugs present
- 1,000,000 validation tests run on 27 versions and gold version

A Case Study - The Programmers

14 BSc degrees

8 MSc degrees

4 PhDs

1 unknown background

Some programmers had little programming experience while others extensive

Deliberate range distribution in knowledge and programming background

A Case Study - Experimental Results

Failure occurs for a version on a particular test case when there are discrepancies between the results produced by that version and those from the golden version

Conclusions

Each program was extremely reliable when combined by NVP

The number of tests in which more than one program failed was substantially higher than expected

Individual versions must be carefully (read: probably not) used

Reliability analysis must also include the effects of dependent errors

Fault Injection Analysis

Fault Injection Analysis

Fault injection analysis involves subjecting a system to abnormal conditions, i.e., introducing faults / errors, so that behaviour can assessed

Usually tests the effectiveness of fault tolerance mechanisms

Can be performed during:

Design - Computer aided design (CAD) environments are used to evaluate design via simulation, including simulation-based fault injection

Testing - Early feedback provided based on the analysis of units, subsystems or operational prototypes

Fault Injection Analysis

Fault injection can be performed on a prototype system

- System runs under controlled workload conditions
- Controlled fault injection used, including some assessment of coverage
- Fault injection on a real system provides information about failure process, error latency, error detection, error propagation, error recovery, etc., which is lost when using a prototype
- Can not extrapolate in relation to attributes or analytic measures

Fault Injection Analysis

Fault injection can be performed on an operational / in-production system

- Operating under under a representative workload
- Provides information about naturally occurring errors / failures
- Limited to detecting errors (unless you have a very forgiving set of users)

Coverage

Dependable programs have dependability mechanisms in-built

The **coverage** of these mechanisms is the ratio of number of test cases successfully passed to the number of test cases subjected to

Coverage 1 means system can handle all test cases

Coverage 0 means system cannot handle any test case

Parameter Estimation

Statistical Approaches To Fault Tolerance

Data from fault injection and / or operational systems can be used to identify potential faults or undetected error in software systems (including those are are being tolerated already!)

Common statistics techniques applied in identifying and imparting software fault tolerance

- Parameter estimation
- Maximum likelihood
- Interval estimation - confidence intervals for means/variances
- Cluster analysis
- Correlation analysis

Parameter Estimation

Random variable

- Distribution, mean and variance
- Point estimation usually used in experimental analysis
- Estimate of detection coverage from fault injection and MTBF from operational data
- Each fault injection and failure occurrence can be treated an experiment
- Each experiment assumed to be independent

Point Estimation

Given a selection of N experimental outcomes

$x_1, x_2, \ldots, x_N$ of random variable X,

X_i is considered a value of X, where X_i is independent but has a similar distribution to X

$\{X_1, X_2, \ldots, X_N\}$ is a random sample of X

Point Estimation

Goal: Estimate the value of some parameter p, i.e., mean or variance of X, using $\{X_1, X_2, \ldots, X_N\}$

$pe(X_1, X_2, \ldots, X_N)$ estimates p (hence, it's known as an estimator of p)

$pe(x_1, x_2, \ldots, x_N)$ is a point estimate of p

Point estimate is an unbiased estimator of p if $E[pe] = p$

Point Estimation - Notes On Unbiased Estimation

It can be shown that the sample mean is the unbiased minimum variance linear estimator of the population mean

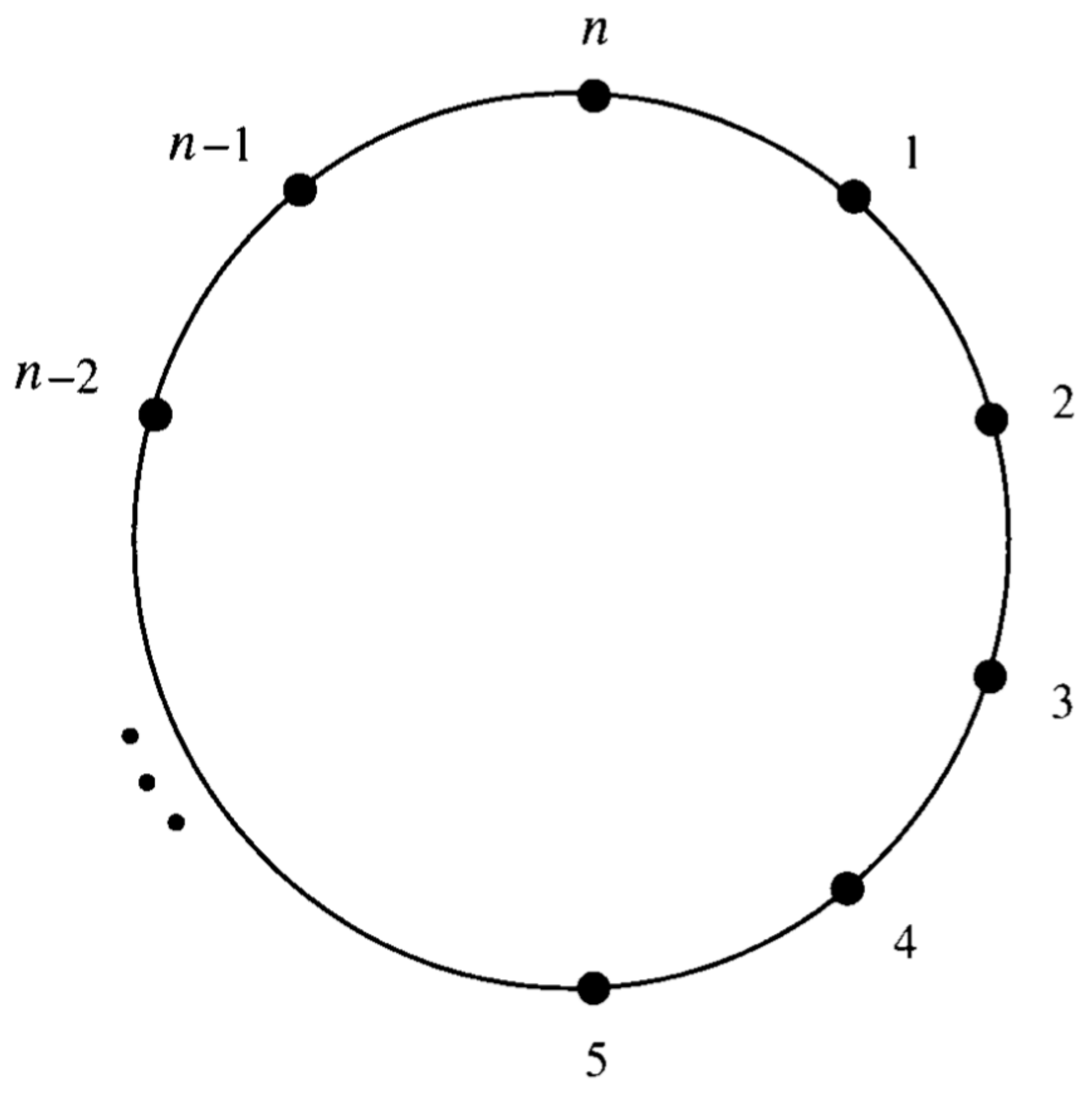
Hence, it is required to determine the values for x_i and use the equation for sample mean to estimate population mean

Thus, coverage is never the real value, but only an estimation of it

A Challenge For You - Leader Election

Can you design a simple algorithm to elect a leader in a synchronous token ring?

- Unidirectional ring of unknown size
- Operates in rounds (a node can receive and transmit in each round, if desired)
- Nodes have an identifier
- Algorithm must be run on every node (regardless of its position)



n
1
2
3
4
5
n–2
n–1