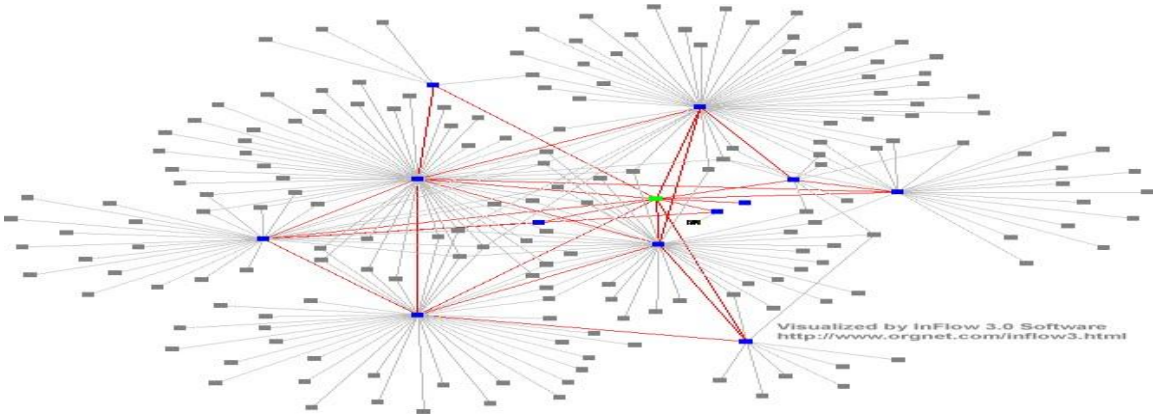
DCS Fault Tolerance Issues

Part-2



Dr. Sunny Jeong. spjeong@uic.edu.cn

With Thanks to Prof. G. Coulouris, Prof. A.S. Tanenbaum and Prof. S.C Joo

More on Software Reliability Models

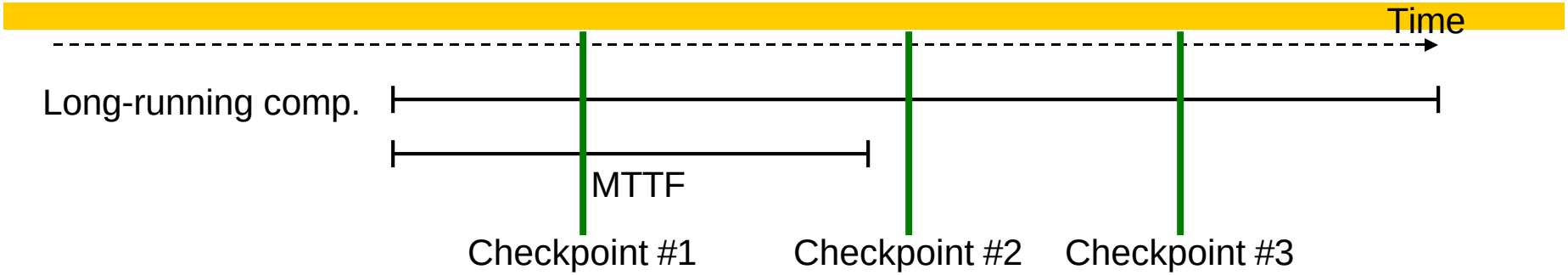
Exponentially decreasing flaw removal rate is more realistic than linearly decreasing, since flaw removal rate never really becomes 0

Estimating the model constants

- Use handbook: public ones, or compiled from in-house data
- Match moments (mean, 2nd moment, . . .) to flaw removal data
- Least-squares estimation, particularly with multiple data sets
- Maximum-likelihood estimation (a statistical method)

-> Fault Prediction, Monitoring Systems Operation, Risk Management…

Checkpointing and Rollback



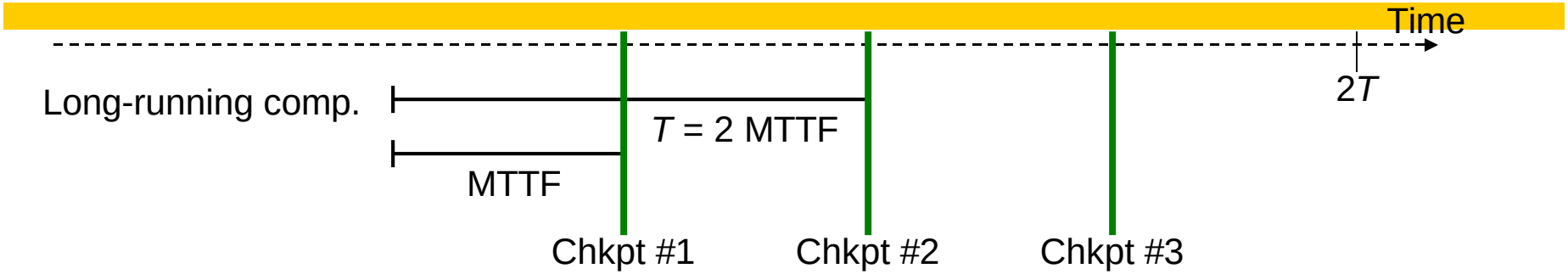
If MTTF is shorter than the running time, many restarts may be needed

Early computers had a short MTTF
It was impossible to complete any computation that ran for several hours

Checkpoints are placed at convenient points along the computation path (not necessarily at equal intervals)

Checkpointing entails some overhead
Too few checkpoints would lead to a lot of wasted work
Too many checkpoints would lead to a lot of overhead

Why Checkpointing Helps

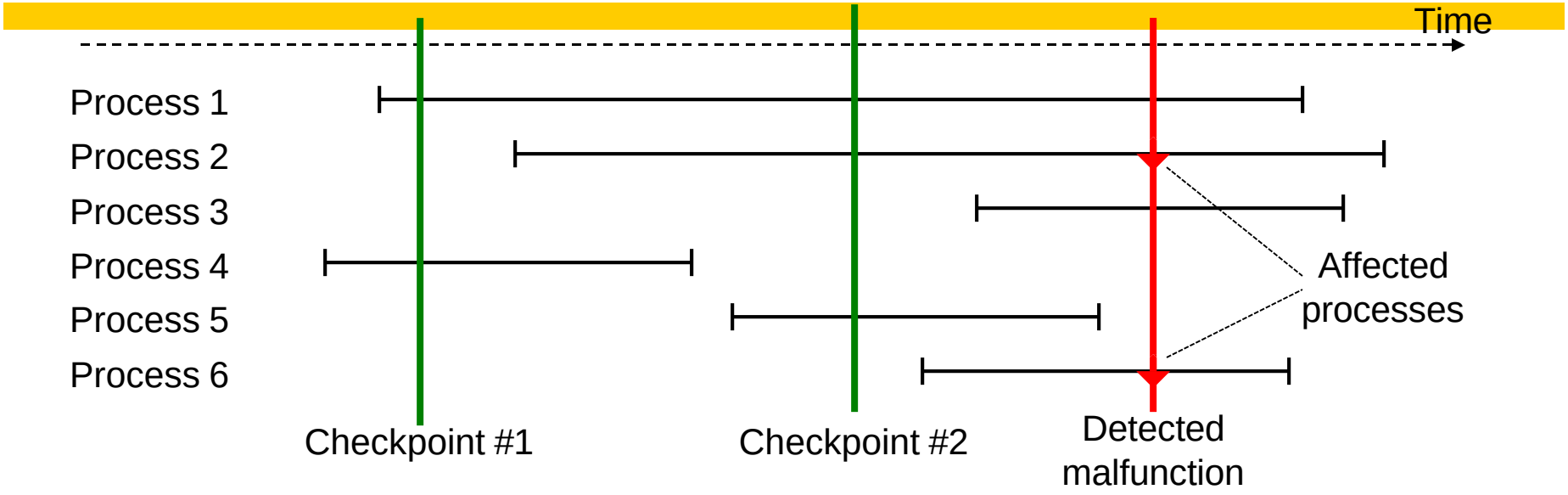


A computation's running time is T = 2 MTTF = 2/λ. (fault rate $\lambda = 1/$ MTTF)

What is the probability that we can finish the computation in time 2T:
a. Assuming no checkpointing
b. Assuming checkpointing at regular intervals of T/2
c. Ignore all overheads.

Cf. Reliability Function
R(T) = e^(-λT)

Recovery via Rollback



Roll back process 2 to the last checkpoint (#2)
Restart process 6

Rollback or restart creates no problem for tasks that do I/O at the end
Interactive processes must be handled with more care
e.g., bank ATM transaction to withdraw money or transfer funds
(check balance, reduce balance, dispense cash or increase balance)

Three Common Causes of Failure

- Errors in the specification or design
  - Specification must be unambiguous, complete, and correct (very difficult to ensure)
  - The design that is developed from a formal specification can, theoretically, be checked formally
- Defects in the components
  - May arise from manufacturing defects, or form defects caused by the wear and tear of use
- Environmental effects

What to Check and Common Terms

- **Hardware fault**: physical defect that can cause a component to malfunction
  Ex: a broken wire
- **Software fault**: a “bug” that can cause the program to fail for a given set of inputs
- **Fault/Error latency**: the duration between the onset of a fault and Its manifestation as an error
- **Error recovery**: the process by which the system attempts to recover from the effects of an error
  - **Forward error recovery**: the error is masked without any computations having to be redone
  - **Backward error recovery**: the system is rolled back to a moment in time before the error is believed to have occurred and the computation is carried out again

Examples : How to Check

- Parity Coding
  - Adding an extra bit to each word
    - Odd: the number of 1's in it is always odd
      - 0001101 → 00011010
    - Even: the number of 1's in it is always even
      - 0001101 → 00011011
  - It is possible to *detect* all one-bit errors
  - It is not always possible to detect errors if two or more bits are wrong
- Checksum -> adding 1's complement
  - Divide the original data into the 'm' number of blocks with 'n' data bits in each block.
  - Adding all the 'k' data blocks.
  - The addition result is complemented using 1's complement.
  - The obtained data is known as the Checksum.

Fault Classification

- Fault is **said** to be:
  - ***active*** when it is physically capable of generating errors
  - ***benign*** when it is not

- Faults are **classified** according to:
  - ***temporal*** *behavior*
  - ***output*** *behavior*

Temporal Behavior Classification

- Permanent
  - fault does not die away with time, remains until it is repaired or replaced
- Intermittent
  - fault cycles between the fault-active and fault-benign state
- Transient
  - fault dies away after some time

Output Behavior Classification

- Non-malicious
  - Erroneous output from a unit will be interpreted in a consistent way by all of the receivers
- Malicious
  - Consistency breaks down; receivers see non-identical values / erroneous output may be interpreted inconsistently by the receivers
  - Harder to handle → *Byzantine failures*, *Arbitrary failures*

Output Behavior : Fail-stop/Fail-safe Unit

- Fail-stop unit responds to up to a certain maximum number of failures by simply stopping, rather than putting out incorrect output
- Fail-stop units typically consist of multiple processors running the same tasks and comparing results.
  - If the outputs are different, the whole unit turns itself off

- A system is said to be fail-safe if its failure mode is biased so that the application process does not suffer catastrophe upon failure
- Classic example: traffic-light controller
  - When it fails:
    - If it sets all the lights to green, catastrophe ensues
    - If it sets all the lights to red or to flashing yellow, it is fail-safe

How to Prevent/Responses to failure

- Short-term responses
  - Quickly correcting for a failure to allow immediate deadline to be met
- Long-term responses
  1. Locating the failure
  2. Determining the best response to it
  3. Initiating a recovery & reconfiguration procedure

Fault & Error Containment

- Faults and errors must be prevented from spreading through the system → *containment plan*
  - Classic example: ships have long been divided into separate watertight compartments, so that damage to a small numbers of compartments does not sink the ship
- The function on error containment zone(ECZ) is to prevent errors from propagating across zone boundaries
- Typically achieved by voting redundant outputs

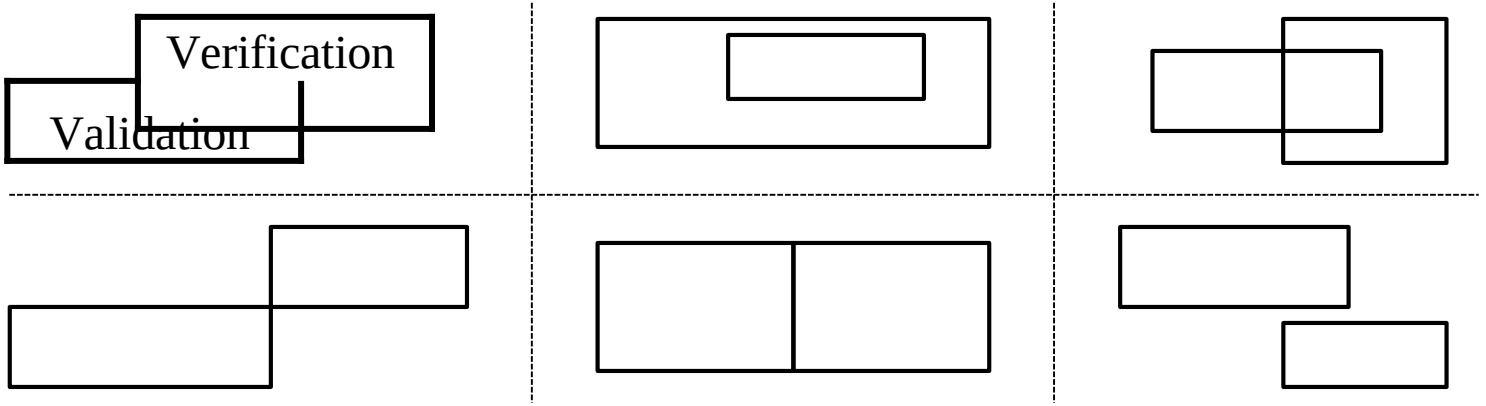
Software Verification and Validation

Verification: “Are we building the system right?” (meets specifications)

Validation: “Are we building the right system?” (meets requirements)

Both verification and validation use testing as well as formal methods

Example: overlap of rectangles for verification and validation



Formal Proofs for Software Verification

Program to find the greatest common divisor of integers $m > 0$ and $n > 0$

```
input m and n
x := m
y := n
while x ≠ y
   if x < y
   then y := y – x
   else x := x – y
   endif
endwhile
output x
```

m and *n* are positive integers

x and *y* are positive integers, $x = m$, $y = n$

Loop invariant: $x > 0$, $y > 0$, $\gcd(x, y) = \gcd(m, n)$

$x = \gcd(m , n)$

Steps 1-3: “partial correctness”
Step 4: ensures “total correctness”

The four steps of a correctness proof relating to a program loop:

1. Loop invariant implied by the assertion before the loop (precondition)
2. If satisfied before an iteration begins, then also satisfied at the end
3. Loop invariant and exit condition imply the assertion after the loop
4. Loop executes a finite number of times (termination condition)

Formal Proofs for Software Validation

- **Validating user interface (UI) elements:**
  checking if buttons function as expected,
- **Verifying data integrity:**
  is data is stored and retrieved accurately?
- **Confirming system performance:**
  validating response times meet user expectations
- **Ensuring compliance with regulations:**
  adheres to industry standards or laws
- **Testing interoperability with other systems:**
- **Validating system security measures:**

Software Flaw Tolerance Techniques

Flaw avoidance strategies include (structured) design methodologies, software reuse, and formal methods

Given that a complex piece of software will contain bugs, can we use redundancy to reduce the probability of software-induced failures?

The ideas of masking redundancy, standby redundancy, and self-checking design have been shown to be applicable to software, leading to various types of fault-tolerant software

"Flaw tolerance" = "Fault tolerance"

Masking redundancy: N-version programming

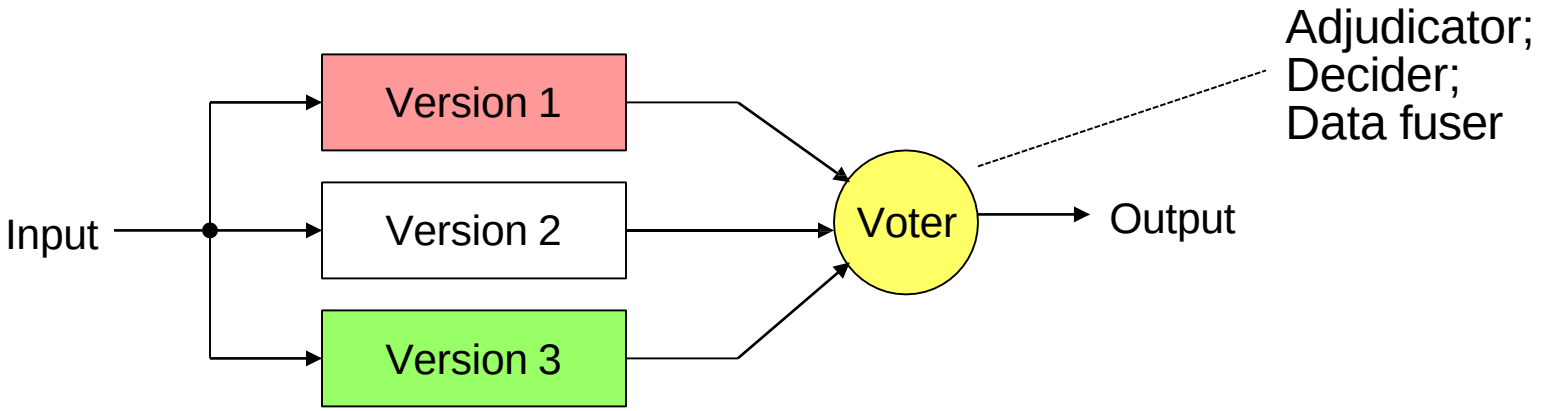
Standby redundancy: the recovery-block scheme

Self-checking design: N-self-checking programming

Sources: *Software Fault Tolerance*, ed. by Michael R. Lyu,
(on-line book at http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html)

N-Version Programming Model

Independently develop *N* different programs (known as “versions”) from the same initial specification



The greater the diversity in the *N* versions, the less likely that they will have flaws that produce correlated errors

Diversity in:

- Programming teams (personnel and structure)
- Software architecture
- Algorithms used
- Programming languages
- Verification tools and methods
- Data (input re-expression and output adjustment)

Some Objections to N-Version Programming

1) **Developing programs is already a very expensive and slow process**;

Cannot produce flawless software, regardless of cost

2) Diversity does not ensure independent flaws (It has been amply documented that multiple programming teams tend to overlook the same details and to fall into identical traps, thereby committing very similar errors)

This is a criticism of reliability modeling with independence assumption, not of the method itself

3) Imperfect specification can be the source of common flaws

Multiple diverse specifications?

4) With **truly diverse implementations, the output selection mechanism is complicated** and may contain its own flaws

Increasing Entropy

Redundancy (spare capacity)

I. Hardware redundancy

- The system is provided with far more hardware (typically two or three times) that it would need if all the components were perfectly reliable

II. Software redundancy

- The system is provided with different software versions of tasks, preferably written independently by different teams
- When one version of a task fails under certain inputs, another version can be used

Redundancy

III. Time redundancy

- The task schedule has some slack in it, so that some task can be rerun if nece ssary and still meet critical deadlines

IV. Information redundancy

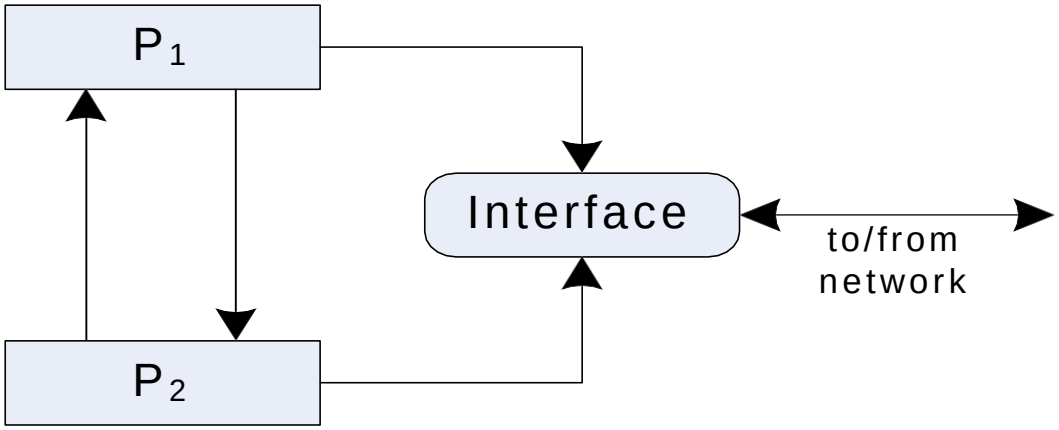
- The data are coded in such a way that a certain number of bit errors can be d etected and/or corrected

I. Hardware redundancy

- Two ways:
  - Fault detection, correction, and masking
    - Multiple hardware units may be assigned to do the same tas k in parallel and their results compared
    - Short-term: neutralize the effects of the observed failure
  - Replace the malfunctioning unit
    - It is possible for system to be designed so that spares can be switched in to replace any faulty units
    - Long-term

Static pairing

- The simplest schemes of all
- Processors in pairs and discard the entire pair when one of the processors fails

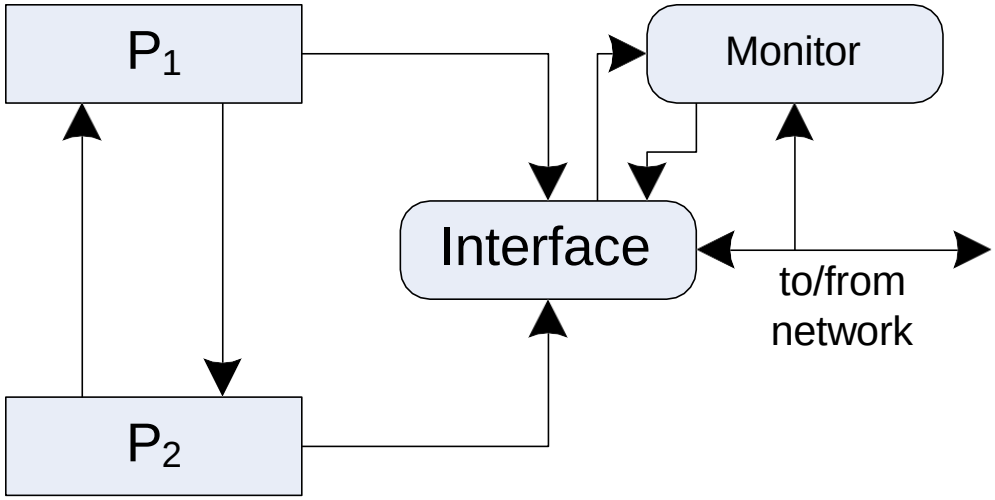


If either processor detects non identical outputs
→ at least one processor is faulty
The processor that detect this discrepancy, switches off the interface to the rest of the system
→ isolating this pair

Static pairing assumptions

- Both processors do not fail identically and around the same time
- The interface does not fail
  - Interface being a critical point of failure
  - Introduce an interface monitor

Static pairing with monitor



- Monitor ensures that the interface correctly receives/ transmits messages, and switches it off whenever faulty is detected
- Likewise, interface can check the output of the monitor, and turn itself off if it detect a fault in the monitor

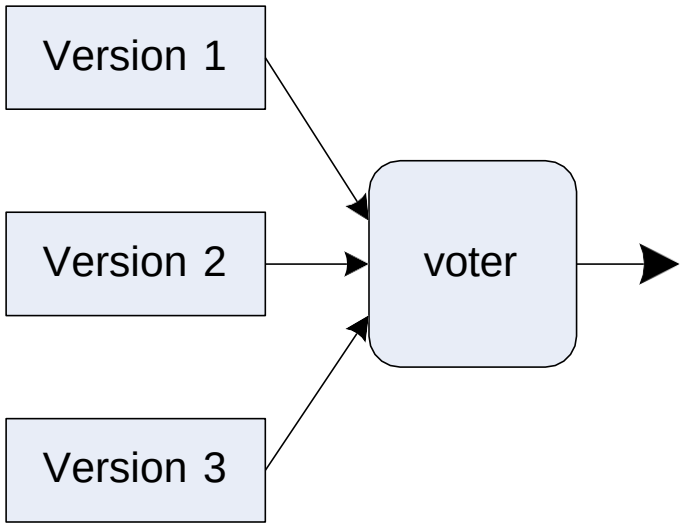
II. Software redundancy

- It is practically impossible to write any large software without faults in it
- Software never wears out
  - Faults are never generated during system operation
  - Software fault → faults in design
- To provide reliability → use redundancy
  - Simply replicating the same SW N times will not work → all N copies will fail for the same inputs
  - The N versions of the SW must be diverse

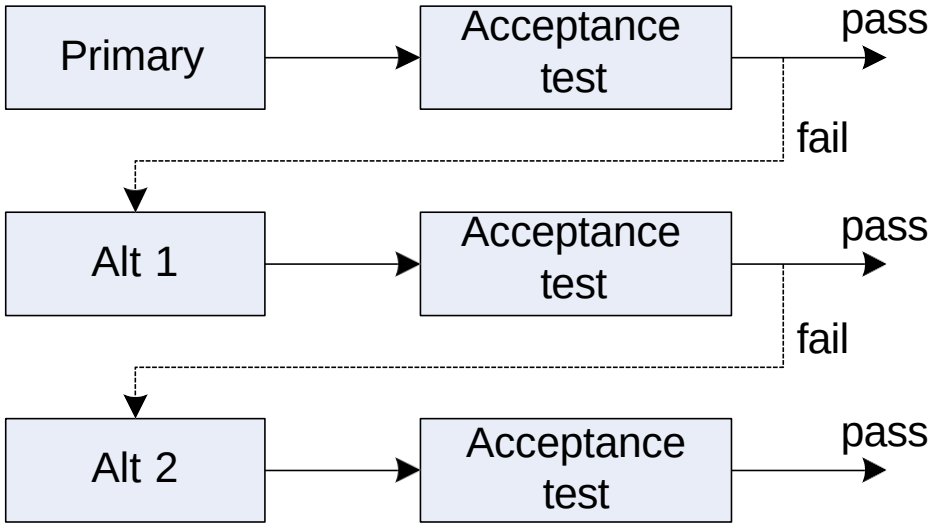
Handling multiple versions

- N-version programming
  - Running all N versions in parallel
  - Voting on the output
- Recovery-block
  - Running only one version at any one time
  - The output of this version is put through an acceptance test to see if it is in an acceptable range
    - If it is, the output is passed as correct
    - If it is not, another version is executed, and so on

The two approaches



N-version programming

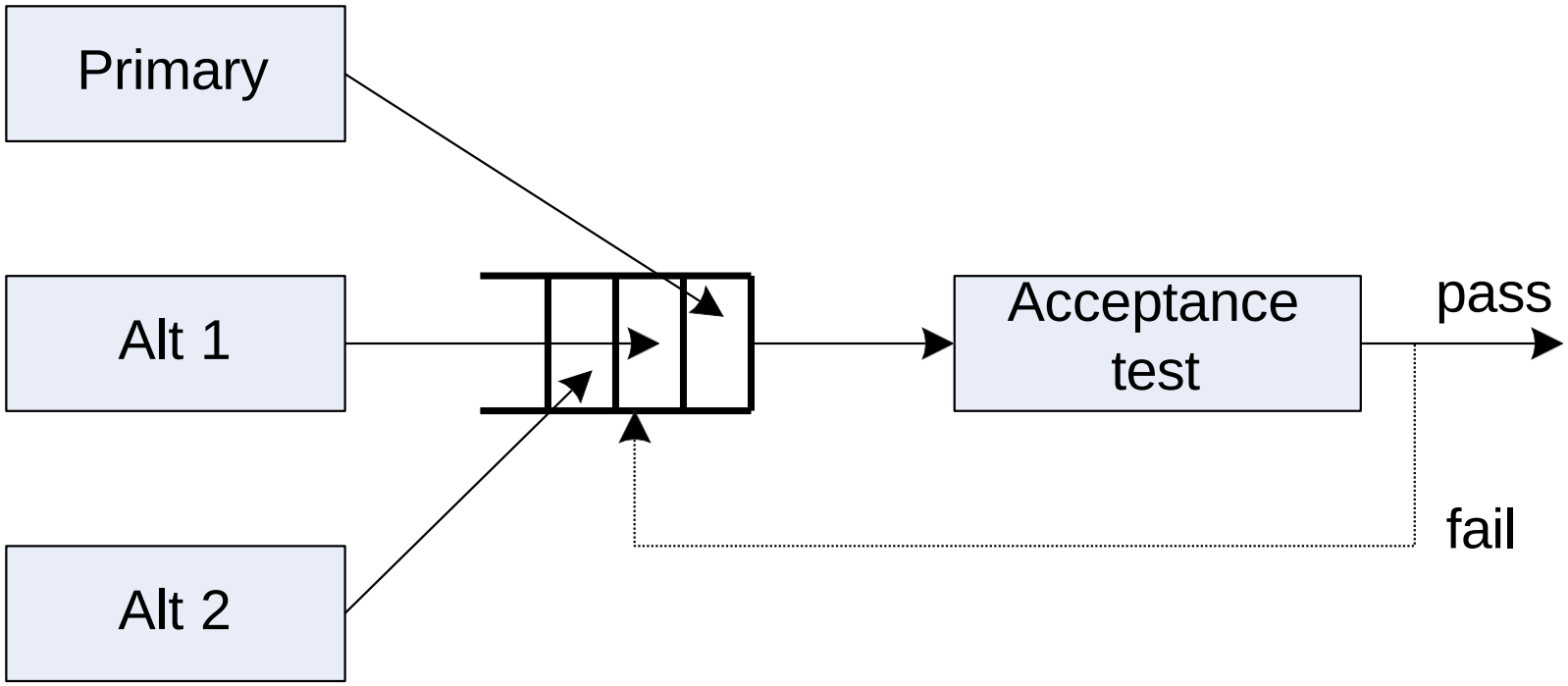


Recovery-block approach

Recovery-block approach

- The acceptance test is the weakest point
  - It has no prior way of knowing what the correct output should be
- Acceptance test ≈ ‘sanity’ check
  - Making sure that the output is within a certain acceptable *range* or that the output does not change at more than the allowed maximum *rate*
    - Ex: calculating the position of a ship, any output that claims that the ship is 10000 miles away from where it was computed to be a few milliseconds before is clearly wrong:GPS
  - These ranges and rates are functions of the application and must be specified by the designer

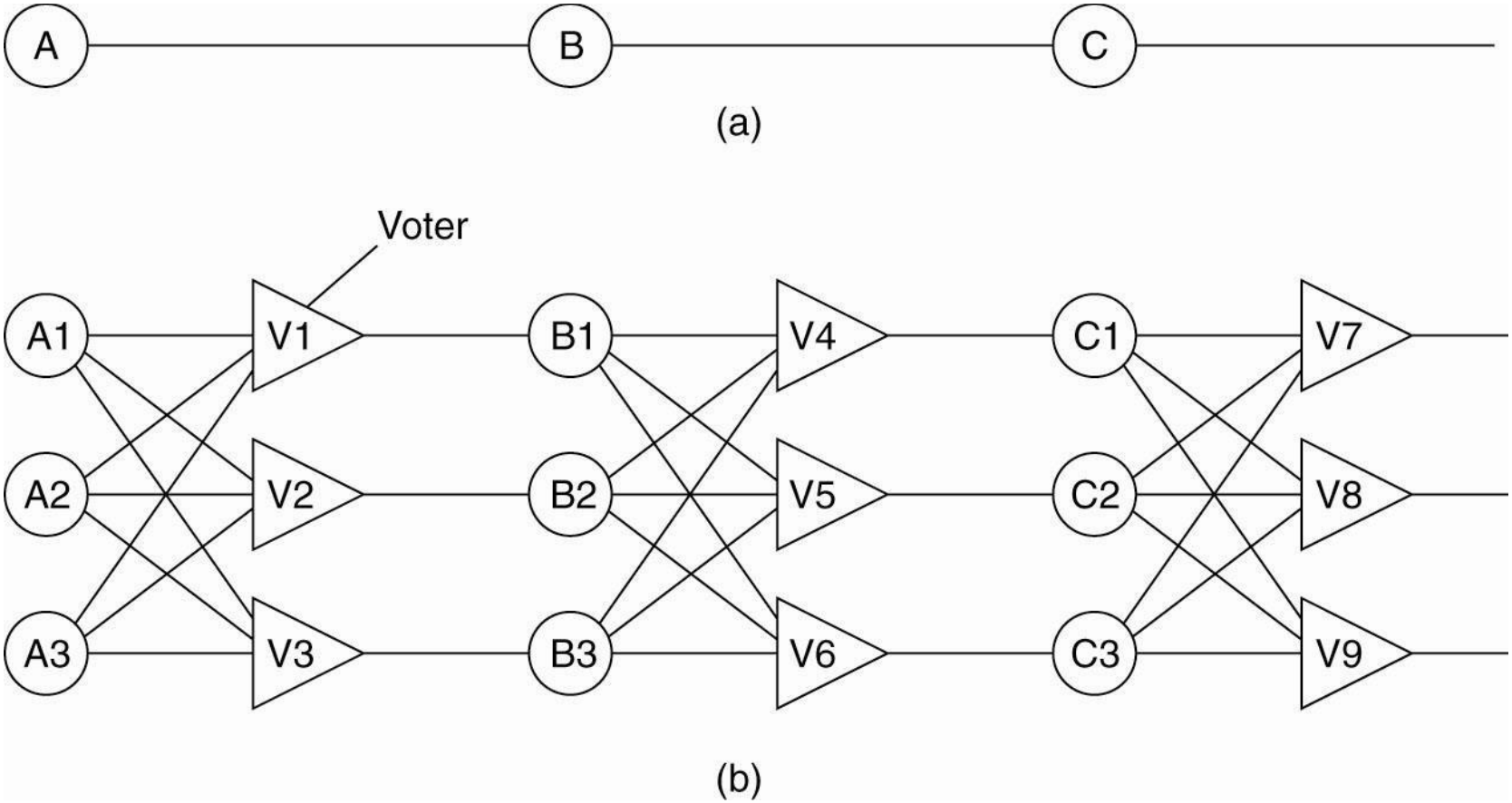
Parallel recovery-block



Parallel recovery-block approach

Failure Masking by Redundancy

- Triple modular redundancy.



III. Time redundancy

- *Backward error recovery* can take multiple forms:
  - *Retry*: the failed instruction is repeated
  - *Rolling* the affected computation back to a *previous checkpoint* and continuing from there
  - *Restarting* the computation all the way from its beginning

- Critical to a successful implementation of backward error recovery is the restoration of the state of the affected processor or system to what it was before the error occurred

IV. Information redundancy

- Adding extra information to data for error protection.
- **Purpose:** Safeguard against errors during data transmission or storage.
- **Benefit:** Enables systems to verify and fix errors.
- **Outcome:** Ensures data integrity and transmission reliability.

The use of coding to detect and/or correct errors

A code may be *separable* or *nonseparable*

Appendix : Data diversity

- Can be used in association with any of the redundancy techniques

- The idea:
  - HW or SW may fail for certain inputs, but not for other inputs that are very close to them.
  - So, instead of applying the same input data to the redundant processors, we apply slightly different input data to them

This will only work if the output's sensitivity to small changes in the input is very small or if disturbing the out put can be corrected analytically.

Summary

Part-1

- SW Dependability
- HW Dependability

Part-2

- Software Reliability Models
- Fault Classification
- **Software Flaw Tolerance Techniques**
  - **N-version Programming**
  - Redundancy