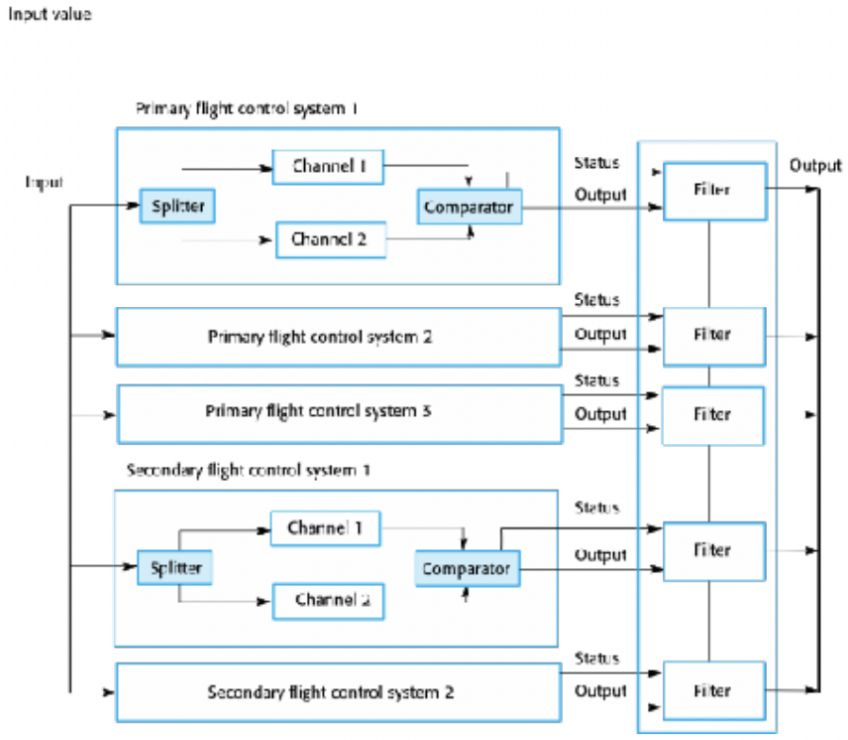
CAS CS 411
Lec 16

Reliability and Availability

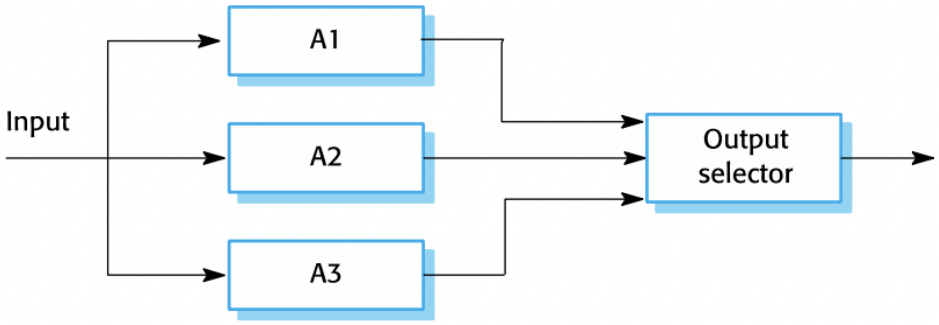
1. Self-Monitoring Systems
    a. Hardware in each channel has to be diverse so that common mode hardware failure will not lead to each channel producing the same results.
    b. Software in each channel must also be diverse, otherwise the same software error would affect each channel.
    c. If high-availability is required, you may use several self-checking systems in parallel.
        i. This is the approach used in the Airbus family of aircraft for their flight control systems.
2. Airbus Flight Control System Architecture



    a.
3. Airbus Architecture Discussion
    a. The Airbus FCS has 5 separate computers, any one of which can run the control software.
    b. Extensive use has been made of diversity
        i. Primary systems use a different processor from the secondary systems.
        ii. Primary and secondary systems use chipsets from different manufacturers.

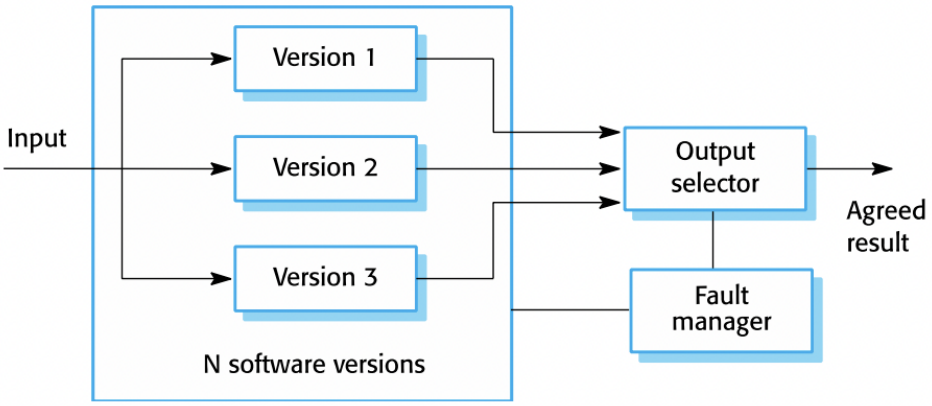
iii. Software in secondary systems is less complex than in primary system – provides only critical functionality.
iv. Software in each channel is developed in different programming languages by different teams.
v. Different programming languages used in primary and secondary systems.

4. N-Version Programming
    a. Multiple versions of a software system carry out computations at the same time. There should be an odd number of computers involved, typically 3.
    b. The results are compared using a voting system and the majority result is taken to be the correct result.
    c. Approach derived from the notion of triple-modular redundancy, as used in hardware systems.
5. Hardware Fault Tolerance
    a. Depends on triple-modular redundancy (TMR).
    b. There are three replicated identical components that receive the same input and whose outputs are compared.
    c. If one output is different, it is ignored and component failure is assumed.
    d. Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure.
6. Triple Modular Redundancy



a.

7. N-Version Programming



a.

    b. The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.
    c. There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.
8. Software Diversity
    a. Approaches to software fault tolerance depend on software diversity where it is assumed that different implementations of the same software specification will fail in different ways.
    b. It is assumed that implementations are (a) independent and (b) do not include common errors.
    c. Strategies to achieve diversity
        i. Different programming languages
        ii. Different design methods and tools
        iii. Explicit specification of different algorithms
9. Problems with Design Diversity
    a. Teams are not culturally diverse so they tend to tackle problems in the same way.
    b. Characteristic errors
        i. Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place;
        ii. Specification errors;
        iii. If there is an error in the specification then this is reflected in all implementations;
        iv. This can be addressed to some extent by using multiple specification representations.
10. Specification Dependency
    a. Both approaches to software redundancy are susceptible to specification errors. If the specification is incorrect, the system could fail
    b. This is also a problem with hardware but software specifications are usually more complex than hardware specifications and harder to validate.
    c. This has been addressed in some cases by developing separate software specifications from the same user specification.
11. Improvements in Practice
    a. In principle, if diversity and independence can be achieved, multi-version programming leads to very significant improvements in reliability and availability.
    b. In practice, observed improvements are much less significant but the approach seems leads to reliability improvements of between 5 and 9 times.

c. The key question is whether or not such improvements are worth the considerable extra development costs for multi-version programming.

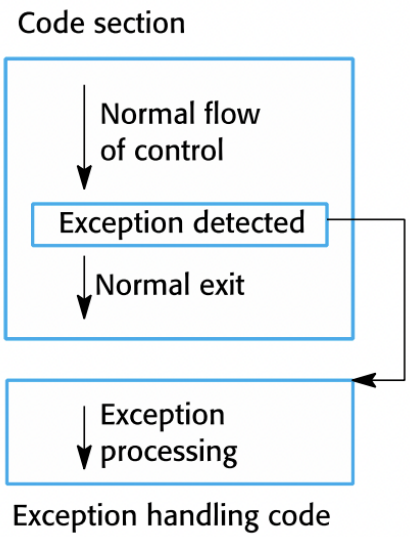
12. Dependable Programming
    a. Good programming practices can be adopted that help reduce the incidence of program faults.
    b. These programming practices support
        i. Fault avoidance
        ii. Fault detection
        iii. Fault tolerance
13. Good Practice Guidelines for Dependable Programming
    a.

Dependable programming guidelines
1. Limit the visibility of information in a program **2. Check all inputs for validity** **3. Provide a handler for all exceptions** **4. Minimize the use of error-prone constructs** **5. Provide restart capabilities** **6. Check array bounds** **7. Include timeouts when calling external components** **8. Name all constants that represent real-world values**

14. (1) Limit the Visibility of Information in a Program
    a. Program components should only be allowed access to data that they need for their implementation.
    b. This means that accidental corruption of parts of the program state by these components is impossible.
    c. You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().
15. (2) Check all Inputs for Validity
    a. All program take inputs from their environment and make assumptions about these inputs.
    b. However, program specifications rarely define what to do if an input is not consistent with these assumptions.
    c. Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.
    d. Consequently, you should always check inputs before processing against the assumptions made about these inputs.
16. Validity Checks
    a. Range checks
        i. Check that the input falls within a known range.

b. Size checks
    i. Check that the input does not exceed some maximum size e.g. 40 characters for a name.
c. Representation checks
    i. Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.
d. Reasonableness checks
    i. Use information about the input to check if it is reasonable rather than an extreme value.

17. (3) Provide a Handler for All Exceptions
    a. A program exception is an error or some unexpected event such as a power failure.
    b. Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
    c. Using normal control constructs to detect exceptions needs many additional statements to be added to the program. This adds a significant overhead and is potentially error-prone.
18. Exception Handling



    a.
    b. Three possible exception handling strategies
        i. Signal to a calling component that an exception has occurred and provide information about the type of exception.
        ii. Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.
        iii. Pass control to a run-time support system to handle the exception.
    c. Exception handling is a mechanism to provide some fault tolerance

19. (4) Minimize the Use of Error-Prone Constructs
    a. Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system
    b. This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don't check for mistakes when they could do so.
    c. Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.
20. Error-Prone Constructs
    a. Unconditional branch (goto) statements
    b. Floating-point numbers
        i. Inherently imprecise. The imprecision may lead to invalid comparisons.
    c. Pointers
        i. Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
    d. Dynamic memory allocation
        i. Run-time allocation can cause memory overflow.
    e. Parallelism
        i. Can result in subtle timing errors because of unforeseen interaction between parallel processes.
    f. Recursion
        i. Errors in recursion can cause memory overflow as the program stack fills up.
    g. Interrupts
        i. Interrupts can cause a critical operation to be terminated and make a program difficult to understand.
    h. Inheritance
        i. Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.
    i. Aliasing
        i. Using more than 1 name to refer to the same state variable.
    j. Unbounded arrays
        i. Buffer overflow failures can occur if no bound checking on arrays.
    k. Default input processing
        i. An input action that occurs irrespective of the input.
        ii. This can cause problems if the default action is to transfer control elsewhere in the program. In incorrect or deliberately malicious input can then trigger a program failure.

21. (5) Provide Restart Capabilities
    a. For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
    b. Restart depends on the type of system
        i. Keep copies of forms so that users don't have to fill them in again if there is a problem
        ii. Save state periodically and restart from the saved state
22. (6) Check Array Bounds
    a. In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
    b. This leads to the well-known 'bounded buffer' vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
    c. If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.
23. (7) Include Timeouts When Calling External Components
    a. In a distributed system, failure of a remote computer can be 'silent' so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
    b. To avoid this, you should always include timeouts on all calls to external components.
    c. After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.
24. (8) Name all Constants that Represent Real-World Values
    a. Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
    b. You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
    c. This means that when these 'constants' change (for sure, they are not really constant), then you only have to make the change in one place in your program.
25. Bottom Line
    a. Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment and by including fault tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
    b. Reliability requirements can be defined quantitatively in the system requirements specification.
    c. Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF) and availability (AVAIL).

d. Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its nonfunctional reliability requirements.
e. Dependable system architectures are system architectures that are designed for fault tolerance.
f. There are a number of architectural styles that support fault tolerance including protection systems, self-monitoring architectures and N-version programming.
g. Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
h. Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.