## Types of Redundancy for Software Fault Tolerance

A key supporting concept for fault tolerance is redundancy, that is, additional resources that would not be required if fault tolerance were not being implemented. **Redundancy can take several forms**: hardware, software, information, and time. Redundancy provides the additional capabilities and resources needed to detect and tolerate faults.

- **Hardware redundancy:** includes replicated and supplementary hardware added to the system to support fault tolerance. It is perhaps the most common use of redundancy. Redundant or diverse software can reside on the redundant hardware to tolerate both hardware and software faults.

- **Software redundancy:** includes the additional programs, modules, or objects used in the system to support fault tolerance.
- **Information or data redundancy**: includes the use of additional information with data (typically used for hardware fault tolerance) and additional data used to assist in software fault tolerance. Sometimes grouped with software redundancy, but will be treated separately here.

- **Temporal redundancy**: involves the use of additional time to perform the tasks required to effect fault tolerance.

  Several aspects of redundancy (e.g., hardware, software/information, and time) can be used in varying ranges (e.g., none, single, dual, and so on) in a single software fault tolerance technique. For example, the NVP technique may use three hardware units and three software variants, but no time or data redundancy. The scope of redundancy for fault tolerance is the use of any or all of the dimensions of redundancy and diversity to support software fault tolerance by a particular technique.

## 1- Software Redundancy

The concept of software redundancy was borrowed from hardware fault tolerance approaches. Hardware faults are typically random, due to component aging and environmental effects. While the use of redundant hardware elements can effectively tolerate hardware faults, tolerance of software faults must employ other approaches. Software faults overwhelmingly arise from specification and design errors or implementation (coding) mistakes. To tolerate software faults, failures arising from these design and implementation problems must be detected

Software design and implementation errors cannot be detected by simple replication of identical software units, since the same mistake will exist in each copy of the software. If the same software is copied and a failure occurs in one of the software replicas, that failure will also occur in the other replicas and there will be no way to detect the problem. (This assumes the same inputs are provided to each copy.) A solution to the problem of replicating design and implementation faults is to introduce diversity into the software replicas. When diversity is used, the redundant software components are termed variants, versions, or alternates. Diversity can be introduced using many differing aspects of the software development process.

The basic approach to adding diversity is to start with the same specification and have different programming teams develop the variants independently. This results in functionally equivalent software components. The goals of increasing diversity in software components are to decrease the probability of similar, common-cause, coincident, or correlated failures and to increase the probability that the components (when they fail) fail on disjoint subsets

of the input space. **The achievement of these goals** increases the systems reliability and increases the ability to detect failures when they occur. The use of diverse software modules requires some **means** to adjudicate, arbitrate, or otherwise decide on the acceptability of the results obtained by the variants. The component that performs this task is called the adjudicator (or decision mechanism). There are many available algorithms with which to perform adjudication.

**Since this adjudication module is not replicated and typically does not have an alternate, it is very important that it be free from errors itself.**

Software redundancy can be structured in many forms. As illustrated in Figure 1.5, the structure can vary depending on the underlying hardware. That is, the system can have

(a) All replicas on a single hardware component,

(b) Replicas on multiple hardware components,

(c) The adjudicator on a separate hardware component.

(d) The software that is replicated can range from an entire program to a few lines of code (program segment).

The choices to be made in structuring software redundancy are based on **available resources** and the **specific application.**

## 2- Information or Data Redundancy

Information (or data) redundancy is sometimes grouped with software redundancy. However, it will be treated separately to introduce the concept of data diversity. Information or data redundancy includes the use of information with data and the use of additional forms of data to assist in fault tolerance. The addition of data to information is typically used for

hardware fault tolerance. Examples of this type of information redundancy include error-detecting and error-correcting codes. Diverse data (not simple redundant copies) can be used for tolerating software faults. A data re-expression algorithm (DRA) produces different representations of a modules input data. This transformed data is input to copies of the module in data diverse software fault tolerance techniques.

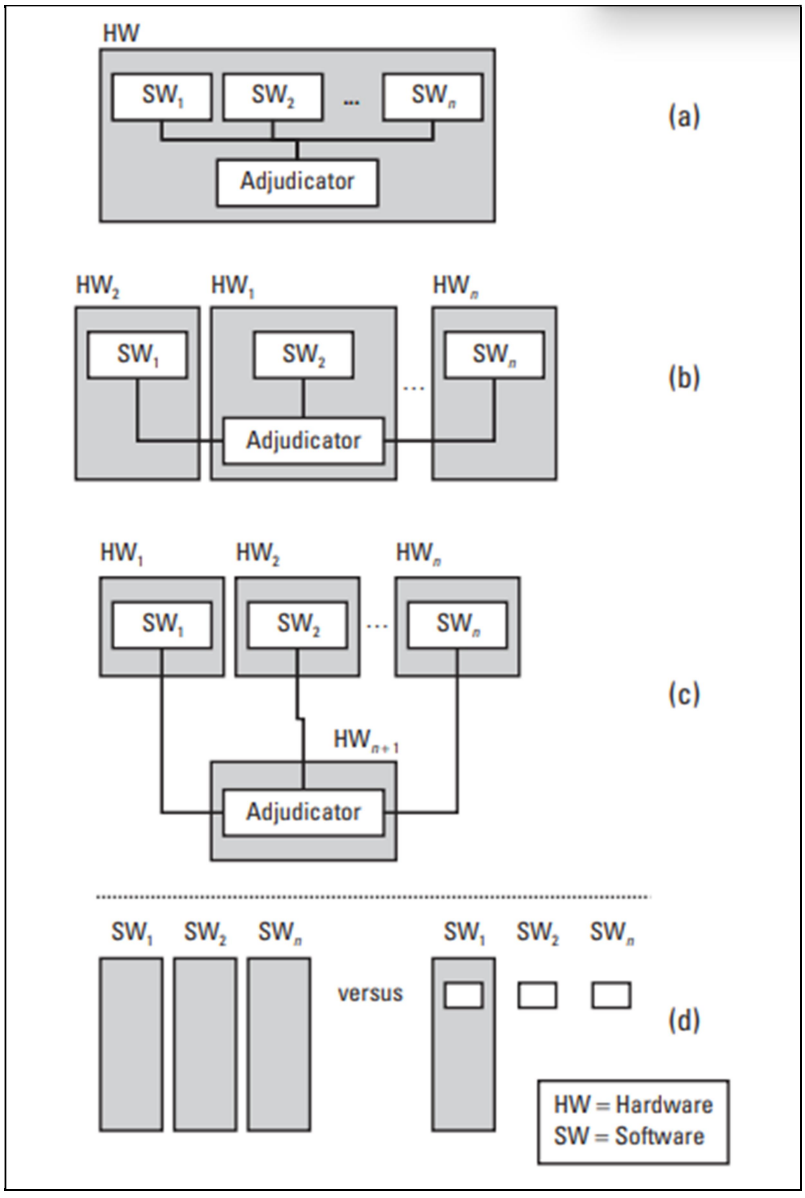


**Figure 1.5** Various views of redundant software: (a) all replicas on a single hardware componenet, (b) replicas on multiple hardware components, (c) adjudicator on separate hardward component, and (d) complete program replicas versus program segment replicas.

## 3- Temporal Redundancy

It involves the use of additional time to perform tasks related to fault tolerance. It is used for both hardware and software fault tolerance. Temporal redundancy commonly comprises repeating an execution using the same software and hardware resources involved in the initial, failed execution. This is typical of hardware backward recovery (roll-back) schemes.

Backward recovery schemes used to recover from software faults typically use a combination of temporal and software redundancy. Timing or transient faults arise from the often complex interaction of hardware, software, and the operating system. These failures, which are difficult to duplicate and diagnose, are called **Heisenbugs**. Simple replication of redundant software or of the same software can overcome transient faults because prior to the reexecution time, the temporary circumstances causing the fault are then usually absent. If the conditions causing the fault persist at the time of reexecution, the reexecution will again result in failure. Temporal redundancy has a great advantage for some applications it does not require redundant hardware or software**. It simply requires the availability of additional time to reexecute the failed process.** Temporal redundancy can then be used in applications in which time is readily available, such as many human-interactive programs.

Applications with hard real-time constraints, however, are **not** likely candidates for using temporal redundancy. The additional time used for reexecution may cause **missed deadlines**. Forward recovery techniques using software redundancy are more appropriate for these applications