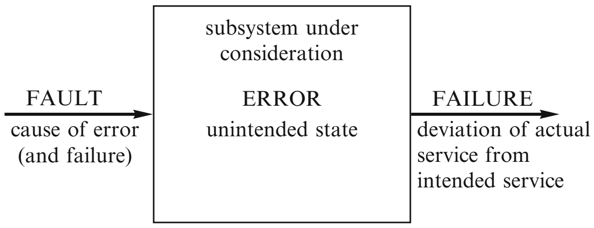
Chapter 6
Dependability

Overview It is said that Nobel Laureate *Hannes Alfven* once remarked that *in Technology Paradise no acts of God can be permitted and everything happens according to the blueprints*. The real world is no technology paradise – *components* can fail and *blueprints* (software) can contain design errors. This is the subject of this chapter. The chapter introduces the notions of *fault*, *error*, and *failure* and discusses the important concept of a *fault-containment uni*t. It then proceeds to investigate the topic of *security* and argues that a security breach can compromise the safety of a safety-critical embedded system. The direct connection of many embedded systems to the Internet – the Internet of Things (IoT) – makes it possible for a distant attacker to search for *vulnerabilities*, and, if the intrusion is successful, to exercise remote control over the physical environment. Security is thus becoming a prime concern in the design of embedded systems that are connected to the Internet. The following section deals with the topic of anomaly detection. An anomaly is an out-of-norm behavior that indicates that some exceptional scenario is evolving. Anomaly detection can help to detect the early effects of a random failure or the activities of an intruder that tries to exploit system vulnerabilities. Whereas an anomaly lies in the grey zone between correct behavior and failure, an error is an incorrect state that requires immediate action to mitigate the consequences of the error. Error detection is based on knowledge about the intended state or behavior of a system. This knowledge can stem either from a priori established regularity constraints and known properties of the correct behavior of a computation, or from the comparison of the results that have been computed by two redundant channels. Different techniques for the detection of temporal failures and value errors are discussed. The following two sections deal with the design of fault-tolerant systems that are capable of masking faults that are contained in the given fault hypothesis. The most important fault-tolerance strategy is *triple modular redundancy* (TMR), which requires a deterministic behavior of replicated components and a deterministic communication infrastructure. Robustness, which is discussed next, is a system property that tries to provide an acceptable level of service despite unforeseen *perturbations*.

Fig. 6.1 Faults, errors, and failures


subsystem under consideration
FAULT
cause of error (and failure)
ERROR
unintended state
FAILURE
deviation of actual service from intended service

6.1 Basic Concepts

The seminal paper by Avizienis et al. [Avi04] establishes the fundamental concepts in the field of dependable computing. The core concepts of this paper are: *fault*, *error,* and *failure* (Fig. 6.1).

Computer systems are provided to offer a dependable service to system users. A user can be a human user or another computer system. Whenever the *behavior* of a system (see Sect. 4.1.1), as seen by the user of the system, deviates from the *intended service*, the system is said to have *failed*. A failure can be *pinned down* to an *unintended state* within the system, which is called an *error*. An error is *caused* by an adverse phenomenon, which is called a *fault*.

We use the term *intended* to state the *correct* state or behavior of a system. Ideally, this correct state or behavior is documented in a precise and complete specification. However, sometimes the specification itself is wrong or incomplete. In order to include specification errors in our model, we introduce the word *intended* to establish an abstract reference for correctness.

If we relate the terms *fault*, *error*, and *failure* to the levels of the four universe model (Sect. 2.3.1), then the term *fault* refers to an adverse phenomenon at any level of the model, while the terms *error* and *failure* are reserved for adverse phenomena at the digital logic level, the informational level, or the external level. If we assume that a sparse global time base is available, then any adverse phenomenon at the digital logic level and above can be identified by a specific bit pattern in the value domain and by an instant of occurrence on the sparse global time base. This cannot be done for phenomena occurring at the physical level.

6.1.1 *Faults*

We assume that a system is built out of components. A component is a *fault-containment unit* (*FCU*), if the direct effect of a single fault influences only the operation of a single component. Multiple FCUs should fail independently. Figure 6.2 depicts a classification of faults.

Fault-Space. It is important to distinguish faults that are related to a deficiency *internal to the FCU* or to some adverse phenomena occurring *external to the FCU*.

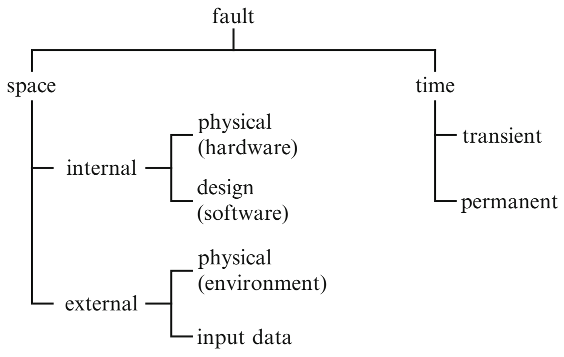


Fig. 6.2 Classification of faults

An *internal fault* of a component, i.e., a fault within the FCU can be a *physical fault*, such as the random break of a wire, or a *design fault* either in the software (a *program error*) or in the hardware (an *erratum*). An *external fault* can be a physical disturbance, e.g., a lightning stroke causing spikes in the power supply or the impact of a cosmic particle. The provision of incorrect input data is another class of an external fault. *Fault containment* refers to design and engineering efforts that ensure that the immediate consequences of a fault are limited to a single FCU. Many reliability models make the tacit assumption that FCUs fail independently, i.e., there is no single fault that can affect more than one FCU. This *FCU independence assumption* must be justified by the design of the system.

Example: The physical separation of the FCUs of a fault-tolerant system reduces the probability for *spatial proximity faults*, such that fault at a single location (e.g., impact in case of an accident) cannot destroy more than a single FCU.

Fault Time. In the temporal domain a fault can be *transient* or *permanent*. Whereas physical faults can be *transient* or *permanent*, design faults (e.g., software errors) are always *permanent*.

A *transient fault* appears for a short interval at the end of which it disappears without requiring any explicit repair action. A *transient* fault can lead to an *error*, i.e., the corruption of the *state* of an FCU, but leaves the physical hardware undamaged (*by definition*). We call a transient external physical fault a *transitory fault*. An example for a transitory fault is the impact of a cosmic particle that corrupts the state of an FCU. We call a transient internal physical fault an *intermittent fault*. Examples for intermittent faults are oxide defects, corrosion or other fault mechanisms that have not yet developed to a stage where the hardware fails permanently (refer to Table 8.1). According to Constantinescu [Con02], a substantial number of the transient faults observed in the field are intermittent faults. Whereas the failure rate of transitory faults is constant, the failure rate for intermittent faults increases as a function of time. An increasing intermittent failure rate of an electronic hardware component is an indication for the *wear-out* of the component. It suggests that preventive maintenance – the replacement of the faulty component – should be performed in order to avoid a permanent fault of the component.

A *permanent fault* is a fault that remains in the system until an explicit repair action has taken place that removes the fault. An example for a *permanent external fault* is a lasting breakdown of the power supply. A *permanent internal fault* can be in the physical embodiment of the hardware (e.g., a break of an internal wire) or in the design of the software or hardware. The mean time it takes to repair a system after the occurrence of a permanent fault is called *MTTR* (*mean time to repair*).

6.1.2 Errors

The immediate consequence of a fault is an *incorrect state* in a component. We call such an incorrect state, i.e., a wrong data element in the memory, a register, or in a flip-flop circuit of a CPU, an *error*. As time progresses, an error is *activated* by a computation, *detected* by some error detection mechanism, or *wiped out*.

An error is *activated* if a computation accesses the *error*. From this instant onwards, the computation itself becomes incorrect. If a fault impacts the contents of a memory cell or a register, the consequent error will be activated when this memory cell is accessed by a computation. There can be a long time-interval between error occurrence and error activation (the *dormancy* of an error) if a memory cell is involved. If a fault impacts the circuitry of the CPU, an immediate activation of the fault may occur and the current computation will be corrupted. As soon as an incorrect computation writes data into the memory, this part of memory becomes *erroneous* as well.

We distinguish between two types of software errors, called *Bohrbugs* and *Heisenbugs* [Gra85]. A *Bohrbug* is a software error that can be reproduced L-deterministically in the data domain by providing a specific input pattern to the routine that contains the Bohrbug, i.e., a specific pattern of input data that always leads to the activation of the underlying Bohrbug. A *Heisenbug* is a software error that can only be observed if the input data and the exact timing of the input data – in relation to the timing of all other activities in the computer – are reproduced precisely. Since the reproduction of a *Heisenbug* is difficult, many software errors that pass the development and testing phase and show up in operational systems are *Heisenbugs*. Since the temporal control structure in event-triggered systems is dynamic, *Heisenbugs* are more probable in event-triggered systems than in time-triggered systems, which have a data-independent static control structure.

Example: A typical example for a *Heisenbug* is an error in the synchronization of the data accesses in a concurrent system. Such an error can only be observed, if the temporal relationships between the tasks that access the mutually exclusive data are reproduced precisely.

An error is *detected*, when a computation accesses the *error* and finds out that the results of the computation deviates from the *expectations* or the *intentions of the user*, either in the domain of value or the domain of time. For example, a simple parity check detects an error if it can be assumed that the fault has corrupted only a single bit of a data word. The time interval between the instant of *error* (*fault*) *occurrence* and the instant of *error detection* is called the *error-detection latency*. The probability that an error is detected is called *error-detection coverage*. Testing is a technique to detect *design faults* (software errors and hardware errata) in a system.

An error is *wiped-out*, if a computation overwrites the error with a new value before the error has been activated or detected. An error that has neither been activated, detected, or wiped-out is called a *latent error*. A latent error in the *state* of a component results in a *silent data corruption* (SDC), which can lead to serious consequences.

Example: Let us assume that a *bitflip* occurs in a memory cell that is not protected by a parity bit and that this memory cell contains sensory input data about the intended acceleration of an automotive engine. The consequent *silent data corruption* can result in an unintended acceleration of the car.

6.1.3 Failures

A *failure* is an event that denotes a *deviation* between the actual behavior and the intended behavior (the *service*) of a component, occurring at a particular instant. Since, in our model, the behavior of a component denotes the sequence of messages produced by the component, a failure manifests itself by the production of an *unintended* (or *no intended*) message. Figure 6.3 classifies the *failure* of a component:

Domain. A failure can occur in the *value domain* or in the *temporal domain*. A value failure means that an *incorrect value* is presented at the component-user interface. (Remember, the user can be another system). A *temporal failure* means that a value is presented outside the *intended interval of real-time*. Temporal failures only exist if the system specification contains information about the intended temporal behavior of the system. Temporal failures can be subdivided into *early temporal failures* and *late temporal failures*. A component that contains internal error detection mechanisms in order to detect any *error* and suppresses a result that contains a *value error* or an *early temporal failure* will only exhibit a *late temporal failure,* i.e., an *omission*, at the interface to its users. We call such a failure an *omission failure*. A component that only makes omission failures is called a *fail-silent component*. If a component stops working after the first

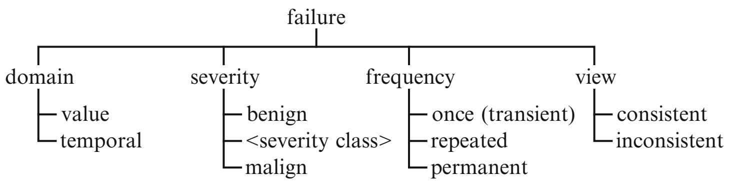


Fig. 6.3 Classification of failures

omission failure, it is called a *fail-stop component*. The corresponding failure is sometimes called a *clean failure* or a *crash failure*.

Example: A self-checking component is a component that contains internal failure detection mechanisms such that it will only exhibit *omission failures* (or *clean failures*) at the *component-user interface*. A self-checking component can be built out of two deterministic FCUs that produce two results at about the same time and where the two results are checked by a *self-checking* checker.

Example: Fault-injection experiments of the MARS architecture have shown that between 1.9% and 11.6% of the observed failures were temporal failures, meaning that a message was produced at an unintended instant. An independent time-slice controller, a guardian, has detected all of these temporal failures [Kar95, p. 326].

Severity. Depending on the effect a failure has on its environment, we distinguish between two extreme cases, *benign* and *malign* failures (see also Sect. 1.5). The cost of a benign failure is of the same order of magnitude as the loss of the normal utility of the system, whereas a malign failure can result in failure costs that are orders of magnitude higher than the normal utility of a system, e.g., a malign failure can cause a catastrophe such as an accident. We call applications where malign failures can occur, *safety-critical* applications. The characteristics of the application determine whether a failure is benign or malign. In between these two extreme cases of benign and malign, we can assign a *severity class* to a failure, e.g., based on the *monetary impact* of a failure or the impact of the failures on the *user experience*.

Example: In a multimedia system, e.g., a digital television set, the failure of a single pixel that is overwritten in the next cycle is masked by the human perception system. Such a failure is thus of negligible severity.

Frequency. Within a given time interval, a failure can occur only *once* or *repeatedly*. If it occurs only once, it is called a *single* failure. If a system continues to operate after the failure, we call the failure a *transient failure*. A frequently occurring transient failure is called a *repeated failure*. A special case of a single failure is a *permanent* one, i.e., a failure after which the system ceases to provide a service until an explicit repair action eliminates the cause of the failure.

View. If more than one user looks at a failing component, two cases can be distinguished: all users see the *same failing behavior* – we call this a *consistent failure* – or different users see *different behaviors* – we call this an *inconsistent failure*. In the literature, different names are used for an *inconsistent failure*: *two-faced failure*, *Byzantine failure*, or *malicious failure*. Inconsistent failures are most difficult to handle, since they have the potential to *confuse* the correct components (see Sect. 3.4.1). In high-integrity systems, the occurrence of Byzantine failures must be considered [Dri03].

Example: Let us assume that a system contains three components. If one of them fails in an *inconsistent failure mode*, the other two correct components will have different views of the behavior of the failing component. In an extreme case, one correct component classifies the failing component as correct, while the other correct component classifies the failing component as erroneous, leading to an inconsistent view of the failed component among the correct components.

Example: A *slightly-out-of-specification* (*SOS*) *failure* is a special case of a Byzantine. SOS failures can occur at the interface between the analog level and the logical level of the four-universe model (see Sect. 2.3.1). If, in a bus system, the voltage of the high-level output of a sender is slightly below the level specified for the high-level state, then some receivers might still accept the signal, assuming the value of the signal is *high*, while others might not accept the signal, assuming the value is *not high*. SOS failures are of serious concern if signals are marginal w.r.t. voltage or timing.

Propagation. If an error inside a component is activated and propagates outside the confines of the component that has been affected by the fault then we speak of *error propagation*. Let us make the simplifying assumption that a component communicates with its environment solely by the exchange of messages and there is no other means of interaction of components (such as a common memory). In such a system, an error can propagate outside the affected component solely by the transmission of an *incorrect message*.

In order to avoid that a propagated error infects other – up to that time healthy – components and thus invalidates the *component independence assumption, error propagation boundaries* must be erected around each component. A message can be incorrect either in the *value domain* (the data field of the message contains a corrupted value) or in the *time domain*, i.e., the message is sent at an unintended instant or not at all (omission failure). Temporal message failures can be detected by the communication system, provided the communication system has a priori knowledge about the correct temporal behavior of a component. Since a communication system is *agnostic* about the contents of the value field of a message (see Sect. 4.6.2), it is the responsibility of the receiver of the message to detect corrupted values, i.e., *errors*, in the data field of a message.

In a cyclic system, the corruption of the *g-state* (see Sect. 4.2.3) is of particular concern, since the g-state contains the information of the current cycle that influences the behavior of the next cycle. Since a *latent error* in the g-state can become an incorrect input to a computation in the next cycle, a gradual increase in the number of errors in the g-state can occur (called *state erosion*). If the g-state is empty, then there is no possibility of error propagation of an error from the current cycle to the next cycle. In order to avoid error propagation from one cycle to the next cycle, the integrity of the g-state should be monitored by a special error detection task of an independent diagnostic component.

6.2 Information Security

Information security deals with the *authenticity*, *integrity*, *confidentiality*, *privacy*, and *availability* of information and services that are provided by a computer system. In the following section, we always mean *information security* when we use the term *security*. We call a *deficiency* in the design or operation of a computer system that can lead to a security incident a *vulnerability* and the successful exploitation of a vulnerability an *intrusion*. The following reasons make clear why information security has become *a prime concern* in the design and operation of embedded systems [Car08]:

1. *Controllers are computers*. Over the past few years, hard-wired electronic controllers have been replaced by programmable computers with non-perfect operating systems, making it possible for an outsider to exploit the vulnerabilities of the software system.
2. *Embedded systems are distributed*. Most embedded systems are distributed, with wire-bound or wire-less channels connecting the nodes. An outside intruder can use these communication channels to gain access to the system.
3. *Embedded systems are connected to the Internet*. The connection of an embedded system to the Internet makes it possible for an intruder anywhere in the world to attack a remote system and to systematically exploit any detected vulnerability.

As of today, there is normally a human mediator between the *cyberspace* (*e.g., the Internet*) and *actions in the physical world*. Humans are supposed to have *common sense* and *responsibility*. They are able to recognize an evidently wrong computer output and will not set any actions in the physical world based on such a wrong output. The situation is different in embedded systems connected directly to the Internet – the *Internet-of-Things* (*IoT*), where the *smart object* at the edge of the Internet (e.g., a robot) can immediately interact with the physical world. An adversary can compromise the integrity of the embedded system by breaching the security walls, thus becoming a safety hazard. Alternatively, an adversary can carry out a *denial-of-service* attack and thus bring down the availability of an important service. *Security and safety are thus interrelated and of utmost concern in embedded systems that are connected to the Internet*.

Example: Let us assume that an owner of a vacation home can set the temperature of the thermostat of his electric furnace in the vacation home remotely via the Internet. If an adversary gets control of the thermostat he can elevate the temperature to a high level and increase the energy consumption significantly. If the adversary executes this attack on all vacation homes in a neighborhood, then the total power consumption in the neighborhood might increase beyond the critical level that leads to a blackout (example taken from Koopman [Koo04]).

Standard security techniques are based on a sound security architecture that controls the information flow among subsystems of different *criticality* and *confidentiality*. The architectural decisions are implemented by the deployment of cryptographic methods, such as *encryption*, *random number generation*, and *hashing*. The execution of cryptographic methods requires extra energy and silicon real estate, which are not always available in a small (portable) embedded systems.

6.2.1 Secure Information Flow

The main security concerns in embedded systems are the *authenticity and integrity of the real-time data* and of the *system configuration*, and, to a lesser extent, the control of access to data. The security policy must specify which processes are authorized to modify data (*data integrity*) and which processes are allowed to see the data (*confidentiality of data*). A security policy for *data integrity* can be established on the basis of the Biba model, while a security policy for the confidentiality of data can be derived from the Bell-LaPaluda model [Lan81]. Both models classify the *processes* and the *data files* according to an ordered sequence of levels, from highest to lowest. A process may read and modify data that is at the same level as the process. The respective security models govern the access and modification of data at a level that is different from the level of the reading or writing process.

The concern of the Biba model is the *integrity of the data*, a concern that is highly relevant in *multi-criticality embedded systems*. The classification of the data files and the processes is determined by the *criticality* from the point of view of the safety analysis (see Sect. 11.4.2). In order to ensure the integrity of a (high-critical) process, the (high-critical) process must not read data that is classified at a lower level than the classification of the (high-critical) process. In order to ensure that a (low-criticality) process will not corrupt data of a higher criticality level, the Biba model states that no (low-criticality) process may modify data that is at a higher criticality level than that of the (low-criticality) process.

The concern of the *Bell-LaPaluda model* is the *confidentiality of the data*. The classification of the data files and the processes is determined by the *confidentiality* of the data from *top secret* to *unclassified*. In order to ensure the confidentiality of top-secret data, it must be made certain that no (unclassified) process may read data that is classified at a higher level than the classification of the (unclassified) process. In order to ensure that a (top secret) process will not publish confidential data to a (unclassified) lower level, the *Bell-LaPaluda* states no (top secret) process may write data to a data file that is at a lower confidentiality level than that of the (top secret) process.

The classification of processes and data from the point of view of integrity will normally be different from classification according to the point of view of confidentiality. These differences can lead to a conflict of interest. In case of such a conflict, the integrity concern is the more important concern in embedded systems.

The selected security policy must be enforced by *mechanisms* that establish the authenticity of processes and the integrity of the data that is exchanged. These mechanisms make wide use of the well-understood cryptographic methods discussed in Sect. 6.2.3.

6.2.2 Security Threats

A systematic security analysis starts with the specification of an *attack model*. The attack model puts forward an *attack hypothesis*, i.e., it lists the *threats* and makes assumptions about the *attack strategy* of an adversary. It then outlines the conjectured steps taken by an adversary to break into a system. In the next phase a *defense strategy* is developed in order to counter the attack. There is always the possibility that the attack hypothesis is incomplete and a clever adversary finds a way to attack the system by a method that is not covered by the attack hypothesis.

The typical attacker proceeds according to the following three phases: *access to the selected subsystem*, *search for and discovery of a vulnerability*, and finally, *intrusion and control of the selected subsystem*. The control can be *passive* or *active*. In *passive* control, the attacker observes the system and collects confidential information. In *active* control, the attacker modifies the behavior of the system such that the system will contribute to the attacker's mean purpose. A security architecture must contain observation mechanisms, i.e., *intrusion detection* mechanisms, to detect malicious activities that relate to any of these three phases of an attack. It also must provide firewalls and procedures that mitigate the consequences of an attack such that the system can survive.

Access to the system must be prevented by requiring strict adherence to a mandatory access control procedure, where every person or process must authenticate itself and this authentication is verified by *callback* procedures. *Security firewalls* play an important role to limit the access to sensitive subsystems to authorized users.

The attacker's *search for vulnerabilities* can be detected by *intrusion detection mechanisms,* which can be part of an anomaly detection subsystem (see Sect. 6.3.1). Anomaly detection is needed in order to detect the consequences of random physical faults as well as the activities of a malicious intruder.

The *capture of control* over a subsystem can be prevented by a structured security architecture, where different criticality levels are assigned to different processes and a formal security policy, based on a formal model, controls the interactions among these criticality levels.

The attainment of topmost security is not only a technical challenge. It requires high-level management commitment in order to ensure that the users strictly follow the organizational rules of the given security policy. Many security violations are not caused by technical weaknesses in a computer system, but by a failure of the users to comply with the existing security policies of the organization.

Example: Beautement et al. [Bea09] state: *It is widely acknowledged in security research and practice that many security incidents are caused by human, rather than technical failures. Researchers approaching the issue from a Human–Computer Interaction* (*HCI*) *perspective demonstrated that many human failures are caused by security mechanisms that are too difficult for a non-expert to use.*

The following list of security attacks is only an indication of what has been observed. It is by no means complete:

Malicious Code Attack. A malicious code attack is an attack where an adversary inserts *malicious code*, e.g., a *virus*, a *worm*, or a *Trojan horse*, into the software in order that the attacker gets partial or full control over the system. This malicious code can be inserted statically, e.g., by a malicious maintenance action (*insider attack*), by the process of downloading a new software version, or dynamically during the operation of a system by accessing an infected Internet site or opening an infected data structure.

Spoofing Attack. In a spoofing attack an adversary masquerades as a legitimate user in order to gain unauthorized access to a system. There are many versions of spoofing attacks in the Internet: replacement of a legitimate web-page (e.g., of a bank) by a seemingly identical copy that is under the control of the adversary (also called *phishing*), the replacement of the correct address in an email by a fake address, a *man-in-the middle attack* where an intruder intercepts a session between two communicating partners and gains access to all exchanged messages.

Password Attack. In a password attack, an intruder tries to guess the password that protects the access to a system. There are two versions of password attacks, *dictionary attacks* and *brute force attacks*. In a dictionary attack, the intruder guesses commonly used password strings. In a brute force attack, the intruder searches systematically through the full code space of the password until he is successful.

Cipher-Text Attack. In this attack model the attacker assumes to have access to the cipher text and tries to deduce the plaintext and possibly the encryption key from the cipher text. Modern standardized encryption technologies, such as the AES (Advanced Encryption Standard), have been designed to make the success of cipher-text attacks highly improbable.

Denial of Service Attack. A *denial of service attack* tries to make a computer system *unavailable* to its users. In any wireless communication scenario, such as a sensor network, an adversary can jam the ether with high-power signals of the appropriate frequency in order to interfere with the communication of the targeted devices. In the Internet, an adversary can send a coordinated burst of service requests to a site to overload the site such that legitimate service requests cannot be handled any more.

Botnet Attack. A *botnet* (the word *bot* is an abbreviation of *robot*) is a set of infected networked nodes (e.g., thousands of PC or set top boxes) that are under the control of an attacker and cooperate (unknowingly to the owner of the node) to achieve a malicious mission. In a first phase an attacker gets control over the botnet nodes and infects them with malicious code. In the second phase he launches a *distributed denial-of-service attack* to a chosen target website to make the target website *unavailable* to legitimate users. Botnet attacks are among the most serious attack modes in the Internet.

Example: A study in Japan [Tel09, p. 213] showed that *it takes about four minutes, on average, for an unprotected PC to be infected when connected to the Internet* and that an estimated 500,000 PCs are infected. A total of around 10 Gbps of traffic from Japanese IP addresses are wasted by botnets (SPAM mail traffic via botnets is not included).

6.2.3 Cryptographic Methods

The provision of an adequate level of integrity and confidentiality in embedded systems that are connected to the Internet, the IoT, can only be achieved by the judicious application of cryptographic methods. Compared to general computing systems, the security architecture of embedded systems must cope with the following two additional constraints:

- *Timing constraints*. The encryption and decryption of data must not extend the response time of time critical tasks; otherwise the encryption will have a negative impact on the quality of control.
- *Resource constraints*. Many embedded systems are resource constrained, concerning memory, computational power, and energy.

Basic cryptographic concepts. The basic cryptographic primitives that must be supported in any security architecture are *symmetric key encryption*, *public key encryption*, *hash functions,* and *random number generation*. The proper application of these primitives, supported by a *secure key management system,* can ensure the authenticity, integrity, and confidentiality of data.

In the following paragraphs we use the term *hard* to mean: it is beyond the capabilities of the envisioned adversary to break the system within the time period during which the security must be provided. The term *strong cryptography* is used if the system design and the cryptographic algorithm and key selection justify the assumption that a successful attack by an adversary is highly improbable.

In cryptography, an algorithm for *encryption* or *decryption* is called a *cipher*. During encryption, a *cipher* transforms a *plaintext* into a *ciphertext*. The ciphertext holds all the information of the plaintext but cannot be understood without knowledge of the algorithm and the keys to decrypt it.

A *symmetric key encryption* algorithm encrypts and decrypts a *plaintext* with the same (or trivially related) keys. Therefore both the encryption and decryption key must be kept secret. In contrast, an *asymmetric key algorithm* uses different keys, a *public key* and a *private key*, for encryption and decryption. Although the two keys are mathematically related, it is *hard* to derive the private key from the knowledge of the public key. Asymmetric key algorithms form the basis for the widely used *public key encryption technology* [Riv78].

The procedure for key distribution is called *key management*. In *public key encryption systems*, the security of the system depends on the *secrecy of the private keys* and the establishment of a *trusted relationship* between the *public key* and the *identity of the owner of the respective private key*. Such a trusted relationship can be established by executing a secure network authentication protocol to an *a priori known security server*. An example of such a network authentication protocol is the KERBEROS protocol that provides mutual authentication [Neu94] and establishes a secure channel between two nodes in an open (insecure) network by using a *trusted security server*.

Random numbers are required in both symmetric and asymmetric cryptography for key generation and for the generation of *unpredictable* numbers that are used only once (called a *nonce*) in order to ensure the uniqueness of a key of a session. In *public key encryption,* the node that needs a *private key* must generate the asymmetric pair of keys out of a *nonce*. The private key is kept secret to the node, while the public key is disseminated over open channels to the public. A signed copy of the public key must be sent to a *security server* in order that other nodes can check the trusted relationship between the public key and the identity of the node that generated the public key.

In order to ensure the secrecy, a private key should not be stored in plain text but must be sealed in a *cryptographic envelope*. To operate on such an envelope a non-encrypted key is required, which is usually called the *root key*. The root key serves as the starting point for the *chain of trust*.

The computational effort required to support *public key encryption* is substantially higher than the computation effort needed for *symmetric key encryption*. For this reason, public key encryption is sometimes only used for the secure distribution of keys, while the encryption of the data is done with symmetric keys.

A *cryptographic hash function* is an *L-deterministic* (see Sect. 5.6) mathematical function that converts a large variable-sized *input string* into a fixed size bit-string, called the *cryptographic hash value* (or for short a *hash*) under the following constraints:

- An accidental or intentional change of data in the input string will change the hash value.
- It should be *hard* to find an input string that has a given hash value.
- It should be *hard* to find two different input strings with the same hash value.

Cryptographic hash functions are required to establish the authenticity and integrity of a plain text message by an electronic signature.

Authentication. Anyone who knows the sender's public key can decrypt a message that is encrypted with the sender's private key. If a *trusted relationship* between the sender's public key and the identity of the sender has been ascertained, then the receiver knows that the identified sender has produced the message.

Digital signature. If both, the authenticity and integrity of a plain-text message must be established, the plain text is taken as the input to a cryptographic hash function. The hash value is then encrypted with the author's private key to generate the *digital signature* that is added to the plain text. A receiver who is in the possession of the author's public key must check whether the decrypted signature is the same bit string as the recalculated hash value of the received text.

Privacy. Anyone who uses a receiver's public key for the encryption of a message can be sure that only the receiver, whose public key has been used, can decipher the message.

Resource Requirements. The computational effort for cryptographic operations measured in terms of *required energy, time* and *gate count of an implementation* depends on the selected algorithm and its implementation. In 2001 the US National Institute of Standards has selected the AES (Advanced Encryption Standard) Algorithm as the Federal Information Processing Standard for symmetric encryption. AES supports key sizes of 128, 192 and 256 bits. Table 6.1 gives an estimate of the resource requirements of different hardware implementations for the AES. From this table it is evident that there is an important design trade-off between required time and required silicon area.

Table 6.1 Comparison of requirements of different hardware AES implementations (Adapted from Feldhofer et al. [Fel04a])

AES 128 encryption	Gate equivalent	Clock cycles
Feldhofer	3,628	992
Mangard	10,799	64
Verbauwhede	173,000	10

The different implementations of the AES algorithm depicted in Table 6.1 show the tradeoff between *silicon area* (gate count) and *speed* (clock cycles) that applies to many algorithms. The resource requirements for public key encryption are higher than listed in Table 6.1. However, there is a concerted research effort ongoing to find resource-aware implementations for public-key cryptography that use elliptic curve cryptography [Rom07] in order to deploy public key cryptography in small embedded systems at a reasonable cost. The results of this research on one side and the progress of the semiconductor industry on the other side will provide the technical and economic basis for the pervasive use of cryptography in all but very small embedded systems in the near future.

6.2.4 Network Authentication

In the following section, we outline a sample of a network authentication protocol that uses public key cryptography to establish the *trusted relationship* between a *new node* and its *public key*. For this purpose we need the *trusted security server*. Let us assume all nodes know the public cryptographic key of the security server and the security server knows the public cryptographic keys of all nodes a priori.

If a node, say node A, wants to send a encrypted message to a yet unknown node, say node B, then node A takes the following steps:

1. Node A forms a signed message with the following content: *current time, node A wants to know what is the public key of node B?*, *signature of node A*. It then encrypts this message with the public key of the security server, and sends the ciphertext message to the security server over an open channel.
2. The security server decrypts the message with its private key and checks whether the message has been sent recently. It then examines the signature of the message with the a priori known public key of the signature of node A to find out whether the contents of the message from node A are authentic.
3. The security server forms a response message with the contents: *current time, address of node B, public key of node B*, *signature*, encrypts this message with the public key of node A, and sends this ciphertext message to node A over an open channel.
4. Node A decrypts the message with its private key, checks whether the message has been sent recently and whether the signature of the security server is authentic. Since node A trusts the information of the security server, it now

knows that the public key of node B is authentic. It uses this key to encrypt the messages it sends to B.

A network authentication protocol that establishes a secure channel between two nodes by using symmetric cryptography is the aforementioned KERBEROS protocol [Neu94].

6.2.5 Protection of Real-Time Control Data

Let us assume the following *attack model* for a real-time process control system in an industrial plant. A number of sensor nodes distributed throughout the plant periodically send real-time *sensor values* by open wireless channels to a controller node which calculates the set points for the control valves. An *adversary* wants to sabotage the operation of the plant by sending counterfeit *sensor values* to the controller node.

In order to establish the authenticity and integrity of a sensor value, a standard security solution would be to append an electronic signature to the sensor value by the *genuine* sensor node and to check this signature by the controller node that receives the message. However, this approach would extend the duration of the control loop by the time it takes for generating and checking the electronic signature. Such an extension of the length of the control loop period has a negative effect on the quality of control and must be avoided.

In a real-time control system, the design challenge is to find a solution that detects an *adversary* without any extension of the duration of the control-loop period. The above example shows that the two requirements, *real-time performance* and *security* cannot be dealt with separately in a real-time control system.

There are characteristics of real-time control systems that must be considered when designing a security protocol:

- In many control systems, a single corrupted setpoint value is not of serious concern. Only a sequence of corrupted values must be avoided.
- Sensor values have a short *temporal accuracy* (see Sect. 5.4.2) – often in the range of a few milliseconds.
- The resources of many mobile embedded system, both computational and energy, are constrained.

Some of these characteristics are helpful; others make it more difficult to find a solution.

Example: It is possible to take the signature generation and the signature check of real-time data out of the control loop and perform it in parallel. As a consequence, the detection of an *intrusion* will be delayed by one or more control cycles (which is acceptable considering the characteristics of control system).

Further research is needed to find effective protection techniques for real-time data under the listed constraints.

6.3 Anomaly Detection

6.3.1 *What Is an Anomaly?*

If we look at the state space of *real-life embedded systems*, we find many examples that show a *grey zone* between the *intended* (*correct*) *state* and an *error*. We call states in this intermediate grey zone between intended and erroneous states or behavioral patterns an *anomaly* or an *out-of-norm state* (see Fig. 6.4). Anomaly detection is concerned with the detection of states or behavioral patterns outside the expected, i.e., the normal behavioral patterns, but do not fall into the category of errors or failures. There are many reasons for the occurrence of *anomalies*: activities by an *intruder* to find a vulnerability, exceptional circumstances in the environment, user mistakes, degradation of sensors resulting in imprecise sensor readings, external perturbations, specification changes, or imminent failures caused by an error in the design or the hardware. The detection of anomalies is important, since the occurrence of an anomaly is an indication that some atypical scenario that may require immediate corrective action (e.g., the imminent *intrusion* by an adversary) is developing.

Application-specific a priori knowledge about the restricted ranges and the known interrelationships of the values of RT entities can be used to detect anomalies that are undetectable by syntactic methods. Sometimes these application-specific mechanisms are called *plausibility checks.*

Example: The constraints imposed on the speed of change of the RT entities by the inertia of a technical process (e.g., change of temperature) form a basis for very effective plausibility checks.

Plausibility checks can be expressed in the form of *assertions* to check for the plausibility of an intermediate result or at the end of a program by applying an *acceptance test* [Ran75]. Acceptance tests are effective to detect anomalies that occur in the value domain.

Advanced dynamic anomaly detection techniques keep track of the *operational context* of a system and autonomously *learn* about the normal behavior in

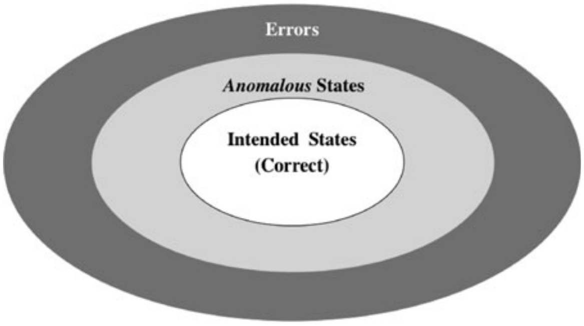


Fig. 6.4 Grey zone between intended and erroneous states

specific contexts in order to be able to detect anomalies more effectively. In a real-time control system that exhibits periodic behavior, the analysis of the time series of real-time data is a very effective technique for anomaly detection. An excellent survey of anomaly detection techniques is contained in Chandola et al. [Cha09].

The anomaly detection subsystem should be separated from the subsystem that performs the operational functions for the following reasons:

- The anomaly detection subsystem should be implemented as an independent fault-containment unit, such that a failure in the anomaly detection subsystem will have no direct effect on the operational subsystem and vice versa.
- Anomaly detection is a well-defined task that must be performed independently from the operational subsystem. Two different engineering groups should work on the operational subsystem and the anomaly detection subsystem in order to avoid common mode effects.

The multi-cast message primitive, introduced in Sect. 4.1.1, provides a means to make the g-state of a component accessible to an independent anomaly-detection subsystem without inducing a probe effect. The anomaly detection subsystem classifies the observed anomalies on a severity scale and reports them either to an off-line *diagnostic system* or to an on-line *integrity monitor*. The integrity monitor can take immediate corrective action in case the observed anomaly points to a safety-relevant incident.

Example: It is an *anomaly* if a car keeps accelerating while the brake pedal is being pressed. In such a situation, an on-line integrity monitor should autonomously discontinue the acceleration.

All detected anomalies should be documented in an *anomaly database* for further on-line or off-line analysis. The depth of investigation into an anomaly depends on the severity of the anomaly – the more severe the anomaly the more information about the occurrence of the anomaly should be recorded. The off-line analysis of the anomaly database can expose valuable information about *weak spots* of a system that can be corrected in a future version.

In a safety-critical system, every single observed anomaly must be scrutinized in detail until the final cause of the anomaly has been unambiguously identified.

6.3.2 Failure Detection

A failure can only be detected if the *observed behavior* of a component can be judged in relation to the *intended behavior*. Failure detection within a system is only possible if the system contains some form of redundant information about the *intended behavior*. The coverage of the *failure detector*, i.e., the probability that a failure will be detected if it is present, will increase if the information about the intended behavior becomes more detailed. In the extreme case, where every failure in the behavior of a component must be detected, a second component that provides the basis for the comparison – a *golden reference component* – is needed, i.e., the redundancy is 100%.

Knowledge about the regularity in the activity pattern of a computation can be used to detect temporal failures. If it is a priori known that a result message must arrive every second, the *non-arrival* of such a message can be detected within 1 s. If it is known that the result message must arrive *exactly at every full* second, and a global time is available at the receiver, then the failure-detection latency is given by the precision of the clock synchronization. Systems that tolerate jitter do have a longer failure-detection latency than systems without jitter. The extra time gained from an earlier failure detection can be significant for initiating a mitigation action in a safety-critical real-time system.

In real-time systems, the *worst-case execution time* (WCET see Sect. 10.2) of all real-time tasks must be known in advance in order to find a viable schedule for the task execution. This WCET can be used by the operating system to monitor the execution time of a task. If a task has not terminated before its WCET expires, a temporal failure of the task has been detected.

6.3.3 Error Detection

As mentioned before, an *error* is an incorrect data structure, e.g., an incorrect *state* or an incorrect *program*. We can only detect an error if we have some *redundant information* about the intended properties of the data structure under investigation. This information can be part of the data structure itself, such as a CRC field, or it can come from some other source, such as *a priori knowledge* expressed in the form of *assertions* or a *golden channel* that provides a result that acts as *golden reference data structure*.

Syntactic knowledge about the code space. The code space is subdivided into two partitions, one partition encompassing syntactically correct values, with the other containing detectably erroneous code-words. This a priori knowledge about the syntactic structure of valid code words can be used for error detection. One plus the maximum number of bit errors that can be detected in a codeword is called the *Hamming distance* of the code. Examples of the use of error-detecting codes are: error-detecting codes (e.g., parity bit) in memory, CRC polynomials in data transmission, and check digits at the man-machine interface. Such codes are very effective in detecting the corruption of a value.

Example: Consider the scenario where each symbol of an alphabet of 128 symbols is encoded using a single byte. Because only seven bits ($2^7 = 128$) are needed to encode a symbol, the eighth bit can be used as a parity bit to be able to distinguish a *valid codeword* from an *invalid codeword* of the 256 code words in the code space. This code has a *Hamming distance of* two.

Duplicate channels. If two independent deterministic channels calculate two results using the same input data, we can compare the results to detect a failure but cannot decide which one of the two channels is wrong. Fault-injection experiments [Arl03] have shown that the duplicate execution of application tasks at different times is an effective technique for the detection of transient hardware faults. This technique can be applied to increase the failure-detection coverage, even if it cannot be guaranteed that *all* task instances can be completed twice in the available time interval.

There are many different possible combinations of hardware, software, and time redundancy that can be used to detect different types of failures by performing the computations twice. Of course, both computations must be *replica determinate*; otherwise, many more discrepancies are detected between the redundant channels than those that are actually caused by faults. The problems in implementing replica-determinate fault-tolerant software have already been discussed in Sect. 5.6.

Golden reference. If one of the channels acts as a golden reference that is considered correct by definition, we can determine if the result produced by the other channel is correct or faulty. Alternatively, we need three channels with majority voting to find out about the single faulty channel, under the assumption that all three channels are synchronized.

Example: David Cummings reports about his experience with error detection in the software for NASA's Mars Pathfinder spacecraft [Cum10]:

Because of Pathfinder's high reliability requirements and the probability of unpredictable hardware errors due to the increased radiation effects in space, we adopted a highly "defensive" programming style. This included performing extensive error checks in the software to detect the possible side effects of radiation-induced hardware glitches and certain software bugs. One member of our team, Steve Stolper, had a simple arithmetic computation in his software that was guaranteed to produce an even result (2, 4, 6 and so on) if the computer was working correctly. Many programmers would not bother to check the result of such a simple computation. Stolper, however, put in an explicit test to see if the result was even. We referred to this test as his "two-plus-two-equals-five check." We never expected to see it fail. Lo and behold, during software testing we saw Stolper's error message indicating the check had failed. We saw it just once. We were never able to reproduce the failure, despite repeated attempts over many thousands if not millions of iterations. We scratched our heads. How could this happen, especially in the benign environment of our software test lab, where radiation effects were virtually nonexistent? We looked carefully at Stolper's code, and it was sound.

What can we learn from this example? We should never build a safety-critical system that relies on the results of a single channel only.

6.4 Fault Tolerance

The design of any fault-tolerant system starts with the precise specification of a *fault hypothesis*. The fault hypothesis states what types of faults must be tolerated by the fault-tolerant system and divides the fault-space into two domains, the domain of *normal faults* (i.e., the faults that must be tolerated) and the domain of *rare faults*, i.e., faults that are outside the fault hypotheses and are assumed to be *rare events*. Figure 6.5 depicts the state space of a fault-tolerant system. In the center we see the correct states. A *normal failure* will bring the system into *a*

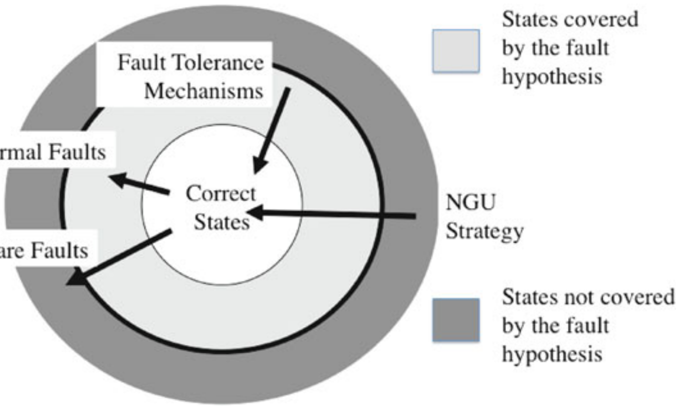

Fault Tolerance Mechanisms
Normal Faults
Correct States
Rare Faults
NGU Strategy
States covered by the fault hypothesis
States not covered by the fault hypothesis

Fig. 6.5 State space of a fault-tolerant system

normal fault-state (i.e., a state that is covered by the fault hypothesis). A normal fault will be corrected by an available fault-tolerance mechanism that brings the system back into the domain of correct states. A *rare fault* will bring the system into a state that is outside the specified fault hypothesis and therefore will not be covered by the provided fault-tolerance mechanisms. Nevertheless, instead of giving up, *a never-give-up* (*NGU*) strategy should try to bring the system back into a correct state.

Example: Let us assume that the fault hypothesis states that during a specified time interval a fault of any single component must be tolerated. The case that two components fail simultaneously is thus outside the fault hypothesis, because it is considered to be a *rare fault*. If the simultaneous failure of two components is detected, then the NGU strategy kicks in. In the NGU strategy it is assumed that the simultaneous faults are transient and a fast restart of the complete system will bring the system back into a correct state. In order to be able to promptly activate the NGU strategy we must have a detection mechanism inside the system that detects the violation of the fault hypothesis. A distributed fault-tolerant membership service, such as the membership protocol contained in the Time-Triggered Protocol (TTP) [Kop93] implements such a detection mechanisms.

6.4.1 Fault Hypotheses

Fault-Containment Unit (FCU). The fault hypothesis begins with a specification of the *units of failure*, i.e., the fault containment units (FCUs). It is up to quality engineering to ensure that FCUs fail independently. Even a small correlation of the failure rates of FCUs has a tremendous impact on the overall reliability of a system. If a fault can cause more than one FCU to fail, then the probability of such a correlated failure must be carefully analyzed and documented in the fault hypotheses.

Example: In a distributed system, a component, including hardware and software, can be considered to form an FCU. Given proper engineering precautions concerning the power supply and the electrical isolation of process signals have been made, the assumption that components of a distributed system that are physically at a distance will fail independently is realistic. On an MPSoC, an IP-core that communicates with other IP-cores solely by the exchange of messages can be considered to form an FCU. However, since the IP-cores of an MPSoC are physically close together (potential for *spatial proximity faults*), having a common power supply and a common timing source, it is not justified to assume that the failures of IP-cores are fully independent. For example, in the aerospace domain, a failure rate of 10^{-6} FIT is assumed for a total MPSoC failure, no matter what kind of MPSoC-internal fault-containment mechanisms are available.

Failure Modes and Failure Rates. In the next step, the assumed failure modes of the FCUs are described and an estimated failure rate for each failure mode is documented. These estimated failure rates serve as an input for a reliability model to find out whether the calculated reliability of a design is in agreement with the required reliability. Later, after the system has been built, these estimated failure rates are compared with the failure rates that are observed in the field. This is to check whether the fault hypothesis is reasonable and the required reliability goals can be met. The following Table 6.2 lists orders of magnitude of typical *hardware failure rates* of large VLSI chips [Pau98] that are used in the industrial and automotive domain.

In addition to the failure modes and failure rates, the fault hypothesis must contain a section that discusses the error and failure detection mechanisms that are designed to detect a failure. This topic is discussed in Sect. 6.3.

Recovery Time. The time needed to recover after a transient failure is an important input for a reliability model. In a state aware design, the recovery time depends on the duration of the ground cycle (see Sect. 6.6) and the time it takes to restart a component.

6.4.2 Fault-Tolerant Unit

In order to tolerate the failure of a fault-containment unit (FCU), FCUs are grouped into *fault-tolerant units* (*FTU*). The purpose of an FTU is to mask the failure of a single FCU inside the FTU. If an FCU implements the fail-silent abstraction, then an FTU consists of two FCUs. If no assumptions can be made about the failure behavior of an FCU, i.e., an FCU can exhibit Byzantine failures, then four FCUs linked by two independent communication channels are needed to form an FTU. If we can assume that a fault-tolerant global time is existent at all FCUs and that

Table 6.2 Order of magnitude of hardware failure rates

Failure mode	Failure rate (FIT)
Permanent hardware failures	10–100
Non-fail silent permanent hardware failures	1–10
Transient hardware failures (strong dependence on environment)	1,000–1,000,000

the communication network contains temporal failures of an FCU, then it is possible to mask the failure of a non-fail-silent FCU by triplication, called *triple modular redundancy* (*TMR*). TMR is the most important fault-masking method.

Although a failure of an FCU is masked by the fault-tolerant mechanism and is thus not visible at the user interface, a permanent failure of an FCU nevertheless reduces or eliminates any further fault-masking capability. It is therefore essential that *masked failures* are reported to a diagnostic system so that the faulty units can be repaired. Furthermore, special testing techniques must be provided to periodically check whether all FCUs and the fault-tolerance mechanisms are operational. The name *scrubbing* refers to testing techniques that are periodically applied to detect faulty units and avoid the accumulation of errors.

Example: If the data words in memory are protected by an error-correcting code, then the data words must be accessed periodically in order to correct errors and thus avoid the accumulation of errors.

Fail-Silent FCUs. A *fail-silent FCU* consists of a computational subsystem and an error detector (Fig. 6.6) or of two FCUs and a self-checking checker to compare the results. A fail-silent FCU produces either correct results (in the value and time domain) or no results at all. In a time-triggered architecture, an FTU that consists of two deterministic fail-silent FCUs produces zero, one, or two correct result messages at about the same instant. If it produces no message, it has failed. If it produces one or two messages, it is operational. The receiver must discard redundant result messages. Since the two FCUs are *deterministic*, both results, if available, are correct and it does not matter which one of the two results is taken. If the result messages are *idempotent*, two replicated messages will have the same effect as a single message.

Triple Modular Redundancy. If a fault-containment unit (FCU) can exhibit value failures at its linking interface (LIF) with a probability that cannot be tolerated in the given application domain, then these value failures can be detected and masked in a *triple modular redundant* (*TMR*) configuration. In a TMR configuration, a fault-tolerant unit (FTU) must consist of three synchronized *deterministic*

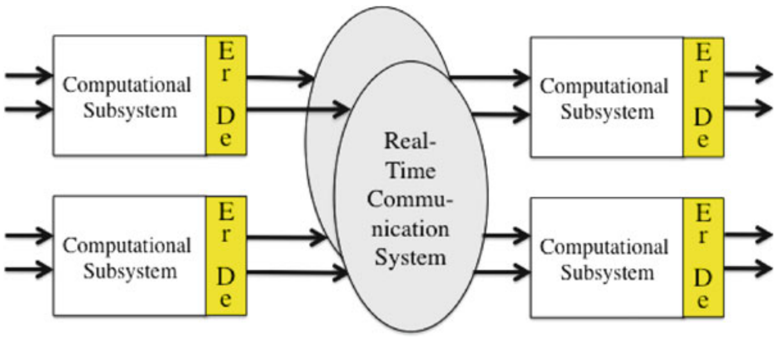


Fig. 6.6 FTU consisting of two fail-silent FCUs

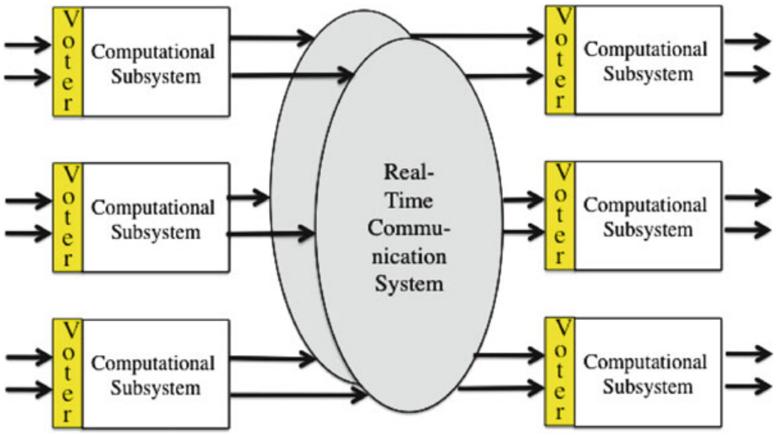


Fig. 6.7 Two FTUs, each one consisting of three FCUs with voters

FCUs, where each FCU is composed of a *voter* and the *computational subsystem*. Any two successive FTUs must be connected by two independent real-time communication systems to tolerate a failure in any of the two communication systems (Fig. 6.7). All FCUs and the communication system must have access to a fault-tolerant global time base. The communication system must perform error containment in the temporal domain, i.e., it must have knowledge about the permitted temporal behavior of an FCU. In case an FCU violates its temporal specification, the communication system will discard all messages received from this FCU in order to protect itself from an overload condition. In the fault-free case, each receiving FCU will receive six physical messages, two (via the two independent communication systems) from each sending FCU. Since all FCUs are deterministic, the *correct FCUs* will produce identical messages. The voter detects an erroneous message and masks the error in one step by comparing the three independently computed results and then selecting the result that has been computed by the majority, i.e., by two out of three FCUs.

A TMR configuration that is set up according to the above specified rules will tolerate an *arbitrary failure* of any FCU and any communication system, provided that a fault-tolerant global time base is available.

Two different kinds of voting strategies can be distinguished: *exact voting* and *inexact voting*. In *exact voting,* a bit-by-bit comparison of the data fields in the result messages of the three FCUs forming an FTU is performed. If two out of the three available messages have exactly the same bit pattern, then one of the two messages is selected as the output of the triad. The underlying assumption is that correctly operating replica-determinate components produce exactly the same results. Exact voting requires that the input messages and the g-state of the three FCUs that form an FTU are *bit-identical*. If the inputs originate from redundant sensors to the physical environment, an agreement protocol must be executed to enforce bit-identical input messages.

In *inexact voting,* two messages are assumed to contain semantically the *same* result if the results are within some *application-specific* interval. Inexact voting must be used if the replica determinism of the replicated components cannot be guaranteed. The selection of an appropriate interval for an inexact voter is a delicate task: if the interval is too large, erroneous values will be accepted as correct; if the interval is too small, correct values will be rejected as erroneous. Irrespective of the criterion defined to determine the *sameness* of two results, there seem to be difficulties.

Example: Lala [Lal94] reports about the experiences with inexact voting in the Air Force's F-16 fly-by-wire control system that uses four loosely synchronized redundant computational channels: *The consensus at the outputs of these channels caused considerable headaches during the development program in setting appropriate comparison thresholds in order to avoid nuisance false alarms and yet not miss any real faults.*

Byzantine Resilient Fault-Tolerant Unit. If no assumption about the failure mode of an FCU can be made and no fault-tolerant global time base is available, four components are needed to form a fault-tolerant unit (FTU) that can tolerate a single Byzantine (or *malicious*) fault. These four components must execute a Byzantine-resilient agreement protocol to agree on a malicious failure of a single component. Theoretical studies [Pea80] have shown that these Byzantine agreement protocols have the following requirements to tolerate the Byzantine failures of k components:

1. An FTU must consist of at least $3k + 1$ components.
2. Each component must be connected to all other components of the FTU by $k + 1$ disjoint communication paths.
3. To detect the malicious components, $k + 1$ rounds of communication must be executed among the components. A round of communication requires every component to send a message to all the other components.

An example of an architecture that tolerates Byzantine failures of the components is given in Hopkins et al. [Hop78].

6.4.3 The Membership Service

The failure of an FTU must be reported in a consistent manner to all operating FTUs with a low latency. This is the task of the *membership service*. A point in real-time when the membership of a component can be established, is called a *membership point* of the component. A small temporal delay between the membership point of a component and the instant when all other components of the ensemble are informed in a consistent manner about the current membership is critical for the correct operation of many safety-relevant applications.

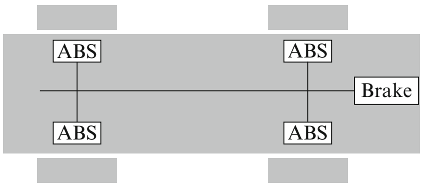


Fig. 6.8 Example of an intelligent ABS in a car

The consistent activation of a never-give-up (NGU) strategy in case the fault hypothesis is violated is another important function of the membership service.

Example: Consider an intelligent ABS (Antilock Braking System) braking system in a car with a node of a distributed computer system placed at each wheel. A distributed algorithm in each of the four nodes, one at each wheel, calculates the brake-force distribution to the wheels (Fig. 6.8), depending on the position of the brake pedal actuated by the driver. If a wheel node fails or the communication to a wheel computer is lost, the hydraulic brake-force actuator at this wheel autonomously transits to a defined state, e.g., in which the wheel is free running. If the other nodes learn about the computer failure at this wheel within a short latency, e.g., a single control loop cycle of about 2 ms, then the brake force can be redistributed to the three functioning wheels, and the car can still be controlled. If, however, the loss of a node is not recognized with such a low latency, then, the brake force distribution to the wheels, based on the assumptions that all four-wheel computers are operational, is wrong and the car will go out of control.

ET Architecture. In an ET architecture, messages are sent only when a significant event happens at a component. Silence of a component in an ET architecture means that *either* no significant event has occurred at the component, *or* a fail-silent failure has occurred (the loss of communication or the fail-silent shut-down of the component). Even if the communication system is assumed to be perfectly reliable, it is not possible to distinguish when there is *no activity at the component* from the situation when a *silent component failure* occurs in an ET architecture. An additional time-triggered service, e.g., a periodic watchdog service (see Sect. 9.7.4), must be implemented in an ET architecture to solve the membership problem.

TT Architecture. In a TT architecture, the periodic message-send times are the membership points of the sender. Let us assume that a failed component remains out-of-service for an interval with duration greater than the maximum time interval between two membership points. Every receiver knows a priori when a message of a sender is supposed to arrive and interprets the arrival of the message as a life sign at the membership point of the sender [Kop91]. It is then possible to conclude, from the arrival of the expected messages at two consecutive membership points, that the component was alive during the complete interval delimited by these two membership points (there is a tacit assumption that a transiently failed node does not recover within this interval). The membership of the FTUs in a cluster at any point in time can thus be established with a delay of one round of information exchange. Because the delay of one round of information exchange is known a priori in a TT architecture, it is possible to derive an a priori bound for the temporal accuracy of the membership service.

6.5 Robustness

6.5.1 *The Concept of Robustness*

In the domain of embedded systems, we consider a system to be robust, *if the severity of the consequences of a fault is inversely proportional to the probability of fault occurrence*, i.e., faults that are expected to occur frequently should have only a minor effect on the quality of service of the system. Irrespective of the concrete type and source of a fault, a robust embedded system will try to recover from the effects of a fault as quickly as possible in order to minimize the impact of the fault on the user. As noted above in Sect. 6.1, the immediate consequence of a fault is an error, i.e., an unintended state. If we detect and correct the error before it has had a serious effect on the quality of service, we have increased the robustness of the system. Design for robustness is not concerned with finding the detailed cause of a failure – this is the task of the *diagnostic subsystem* – but rather with the fast restoration of the normal system service after a fault has occurred.

The inherent periodicity of many real-time control systems and multimedia system helps in the design for robustness. Due to the constrained physical power of most actuators, a single incorrect output in a control cycle will – in most cases – not result in an abrupt change of a physical set point. If we can detect and correct the error within the next control cycle, the effect of the fault on the control application will be small. Similar arguments hold for multimedia system. If a single frame contains some incorrect pixels, or even if a complete frame is lost, but the next frame in sequence is correct again, then the impact of a fault on the *quality of the multimedia experience* is limited.

6.5.2 *Structure of a Robust System*

A robust system consists of at least two subsystems (Fig. 6.9) implemented as independent FCUs, one *operational component* that performs the planned operations and controls the physical environment and a second *monitoring component* that *reflects* whether the results and the g-state of the operational component are in agreement with the *intentions* of the user [Tai03].

In a periodic application such as a control application, every control cycle starts with reading the g-state and the input data, then the control algorithm is calculated,

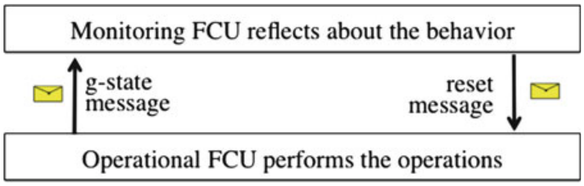

Monitoring FCU reflects about the behavior
g-state message
reset message
Operational FCU performs the operations

Fig. 6.9 Structure of a robust system

and finally the new set points and the new g-state are produced (see Fig. 3.9). A transient fault in one control cycle can only propagate to the next control cycle if the g-state has been contaminated by the fault. In a robust system, the operational component must externalize its g-state in every control cycle such that the monitoring component can check the plausibility of the g-state and perform a corrective action in case a severe anomaly has been detected in the g-state. The corrective action can consist of resetting the operational component and restarting it with a repaired g-state.

In a safety-critical application, this *two-channel approach* – one channel produces a result and the other channel, the *safety monitor,* monitors whether the result is plausible – is absolutely essential. Even if the software has been proven correct, it cannot be assumed that there will be no transient faults during the execution of the hardware. The *IEC 61508 standard* on functional safety requires such a two-channel approach, one channel for the normal function and another independent channel to ensure the *functional safety* of a control system (see also Sect. 11.4).

In a fail-safe application, the safety monitor has no other authority then to bring the application to the safe state. A *fail-silent failure* of the safety monitor will result in a loss of the safety monitoring function, while a *non-fail-silent failure* of the safety monitor will cause a reduction of the availability but will not impact the safety.

In a fail-operational application, a *non-fail silent failure* of the safety monitor has an impact on the safety of the application. Therefore the safety-monitor itself must be fault-tolerant or at least self-checking in order to eliminate non-fail-silent failures.

6.6 Component Reintegration

Most computer system faults are transient, i.e., they occur sporadically for a very short interval, corrupt the state, but do not permanently damage the hardware. If the service of the system can be reestablished quickly after a transient fault has occurred, then in most cases the user will not be seriously affected by the consequences of the fault. In many embedded applications, the fast reintegration of a failed component is thus of paramount importance and must be supported by proper architectural mechanisms.

6.6.1 Finding a Reintegration Point

While a failure can occur at an arbitrary moment outside the control of the system designer, the system designer can plan the proper point of reintegration of a repaired component. The key issue during the reintegration of a component in a real-time system is to find a future point in time when the state of the component is in synchrony with the component's environment, i.e., the other components of the cluster and the physical plant. Because real-time data are invalidated by the passage of time, rolling back to a past checkpoint can be futile: it is possible and probable that the progression of time has already invalidated the checkpoint information (see also Table 4.1).

Reintegration is simplified if the state that must be reloaded into the reintegrating component is of *small size* and fits into a single message. Since the size of the state has a relative minimum immediately after the completion of an atomic operation, this is an ideal instant for the reintegration of a component. In Sect. 4.2.3 we have introduced the notion of the *g-state* (ground state) to refer to the state at the *reintegration instant*. In cyclic systems – many embedded control and multimedia systems are cyclic – an ideal reintegration instant of a component is at the beginning of a new cycle. The temporal distance between two consecutive reintegration instants, the reintegration cycle is then identical to the duration of the control cycle. If the g-state is empty at the reintegration instant, then the reintegration of a repaired component is trivial at this moment. In many situations, however, there is no instant during the lifetime of a component when its g-state is completely empty.

6.6.2 Minimizing the Ground-State

After a cyclic reintegration instant has been established, the g-state at this selected instant must be analyzed and minimized to simplify the reintegration procedure.

In a first phase, all system data structures within the component must be investigated to locate any hidden state. In particular, all variables that must be initialized must be identified and the state of all semaphores and operating system queues at the reintegration instant must be checked. It is good programming practice to output the g-state of a task in a special output message when a task with g-state is detected, and to re-read the g-state of the task when the task is reactivated. This identifies the g-state and makes it possible to pack all g-states of all tasks of a component into a *g-state message* particular to this component.

In a second phase, the identified g-state must be analyzed and minimized. Figure 6.10 displays a suggested division of the g-state information into three parts:

1. The first part of the g-state consists of input data that can be retrieved from the instrumentation in the environment. If the instrumentation is state-based and sends the absolute values of the RT entities (state messages) rather than their relative values (event messages), a complete scan of all the sensors in the environment can establish a set of current images in the reintegrating component and thus resynchronize the component with the external world.
2. The second part of the g-state consists of output data that are in the control of the computer and can be enforced on the environment. We call the set of the output data a *restart vector*. In a number of applications, a restart vector can be defined at development time. Whenever a component must be reintegrated,

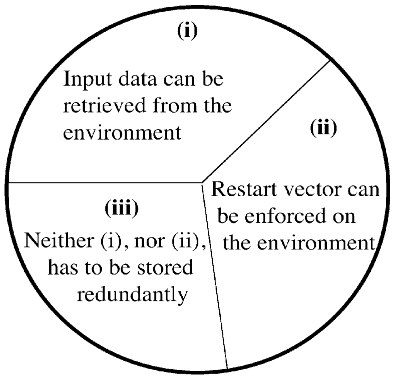

(i)
Input data can be retrieved from the environment
(ii)
Restart vector can be enforced on the environment
(iii)
Neither (i), nor (ii), has to be stored redundantly

Fig. 6.10 Partitioning of the g-State

this restart vector is enforced on the environment to achieve agreement with the outside world. If different process modes require different restart vectors, a set of restart vectors can be defined at development time, one for each mode.

3. The third part of the g-state contains g-state data that do not fall into category (1) or category (2). This part of the g-state must be recovered from some component-external source: from a replicated component of a fault-tolerant system, from the monitoring component, or from the operator. In some situations, a redesign of the process instrumentation may be considered to transform g-state of category (3) into g-state of category (1).

Example: When a traffic control system is restarted, it is possible to enforce a restart vector on the traffic lights that sets all cross-road lights first to yellow, and then to red, and finally turns the main street lights to green. This is a relatively simple way to achieve synchronization between the external world and the computer system. The alternative, which involves the reconstruction of the current state of all traffic lights from some log file that recorded the output commands up to the point of failure, would be more complicated.

In a system with replicated components in an FTU, the g-state data that cannot be retrieved directly from the environment must be communicated from one component of the FTU to the other components of the FTU by means of a g-state message. In a TT system, sending such a g-state message should be part of the standard component cycle.

6.6.3 Component Restart

The restart of a component after a failure has been detected by a monitoring component (Fig. 6.9) can proceed as follows: (1) The *monitoring component* sends a *trusted reset message* to the TII interface of the operational component to enforce a hardware reset. (2) After the reset, the operational component performs a self-test and verifies the correctness of its core image (the *job*) by checking the provided signatures in the core image data structures. If the core image is erroneous, a copy of the static core image must be reloaded from stable storage. (3) The operational component scans all sensors and waits for a cluster cycle to acquire all available current information about its environment. After an analysis of this information, the operational component decides the *mode* of the controlled object, and selects the restart vector that must be enforced on the environment. (4) Finally, after the operational component has received the g-state information that is relevant at the next reintegration instant from the monitoring component, the operational component starts its tasks in synchrony with the rest of the cluster and its physical environment. Depending on the hardware performance and the characteristics of the real-time operating system, the time interval between the arrival of the reset message and the arrival of the g-state information message can be significantly longer than the duration of a reintegration cycle. In this case, the monitoring component must perform a far-reaching state estimation to establish a relevant g-state at the proper reintegration point.

Points to Remember

- A *fault* is the adjudged cause of an error or failure.
- An *error* is that part of the state of a system that deviates from the *intended* (correct) state.
- A *failure* is an event that denotes a deviation of the actual service from the intended service, occurring at a particular point in real time.
- The failure rate for permanent failures of an industrial-quality chip is in a range between 10 and 100 FITS. The failure rate for transient failures is orders of magnitude higher.
- *Information security* deals with the *authenticity*, *integrity*, *confidentiality*, *privacy* and *availability* of information and services that are provided by computer system. The main security concerns in embedded systems are the *authenticity and integrity of data.*
- A *vulnerability* is a deficiency in the design or operation of a computer system that can lead to a security incident. We call the successful exploitation of a vulnerability an *intrusion.*
- The typical attacker proceeds according to the following three phases: *access to the selected subsystem*, *search for and discovery of a vulnerability*, and finally *intrusion and control of the selected subsystem.*
- It is widely acknowledged in security research and practice that many security incidents are caused by human rather than technical failures.
- The basic cryptographic primitives that must be supported in any security architecture are *symmetric key encryption*, *public key encryption*, *hash functions,* and *random number generation.*
- An *anomaly* is a system state that lies in the grey zone between *correct* and *erroneous*. The detection of anomalies is important, since the occurrence of an anomaly is an indication that some atypical scenario that may require immediate corrective action is developing (e.g., the intrusion by an adversary).

- In a safety-critical system, every single observed anomaly must be scrutinized in detail until the final cause of the anomaly has been unambiguously identified.
- Failure detection within a system is only possible if the system contains some form of redundant information about the *intended behavior*.
- The fault hypothesis states what types of faults must be tolerated by a fault-tolerant system and divides the fault-space into two domains, the domain of *normal faults* (i.e., the faults that must be tolerated) and the domain of *rare faults*, i.e., faults that are outside the fault hypotheses and are assumed to be *rare events*.
- A *rare fault* will bring the system into a state that is outside the specified fault hypothesis and therefore will not be covered by the provided fault-tolerance mechanisms. Nevertheless, instead of giving up, *a never-give-up* (*NGU*) strategy should be employed to try to bring the system back to a correct state.
- It is up to quality engineering to ensure that FCUs fail independently. Even a small correlation of the failure rates of FCUs has a tremendous impact on the overall reliability of a system.
- The purpose of a fault-tolerant unit (FTU) is to mask the failure of a single FCU inside the FTU. Although a failure of an FCU is masked by the fault-tolerant mechanism and is thus not visible at the user interface, a permanent failure of an FCU nevertheless reduces or eliminates any further fault-masking capability.
- In a triple-modular-redundant (TMR) configuration a fault-tolerant unit (FTU) consists of three synchronized *deterministic* FCUs, where each FCU is composed of a *voter* and the *computational subsystem*.
- A *membership service* consistently reports *the operational state* of every FTU to all operating FTUs.
- In many embedded applications, the fast reintegration of a failed component is of paramount importance and must be supported by proper architectural mechanisms.
- Design for robustness is not concerned with finding the detailed cause of a failure – this is the task of the *diagnostic subsystem* – but rather with the fast restoration of the normal system service.
- In a safety-critical application a *two-channel approach,* in which one channel produces a result and the other channel monitors whether the result is plausible is absolutely essential.

Bibliographic Notes

The seminal paper by Avizienis et al. [Avi04] introduces the fundamental concepts in the field of dependability and security. Anomaly detection is covered in the comprehensive survey by Chandola [Cha09] and on line failure prediction is the topic of [Sal10]. The yearly DSN conference (organized by the IEEE and the IFIP WG 10.4) is the most important forum for presenting research papers in the field of dependable computing.

Review Questions and Problems

6.1 Give the precise meaning of the terms failure, error, and fault. What are the characteristics of an FCU?
6.2 What are typical permanent and transient failure rates of VLSI chips?
6.3 What is an anomaly? Why is anomaly detection important?
6.4 Why is a short recovery time from transient faults important?
6.5 What are the basic techniques for error detection? Compare ET systems and TT systems from the point of view of error detection.
6.6 What is the difference between *robustness* and *fault tolerance*? Describe the structure of a robust system!
6.7 What is the difference between a *Heisenbug* and a *Bohrbug*?
6.8 Describe the characteristics of a Byzantine failure! What is an SOS failure?
6.9 Give some examples of security threats! What is a *botnet*?
6.10 What is the difference between the *Bipa model* and the *Bell-LaPaluda model* for secure systems?
6.11 What steps must be taken in a systematic *security analysis*? What is a *vulnerability*? What is an *intrusion*?
6.12 What is a membership service? Give a practical example for the need of a membership service. What is the quality parameter of the membership service? How can you implement a membership service in an ET architecture?
6.13 Describe the contents of the fault hypothesis document? What is an NGU strategy?
6.14 Discuss the different types of faults that can be masked by the replication of components. Which faults cannot be masked by the replication of components?
6.15 What is required for the implementation of fault-tolerance by TMR?
6.16 What is a restart vector? Give an example.