1

Introduction

Distributed systems bring many benefits to us, for example, we can share resources such as data storage and processing cycles much more easily; we can collaborative on projects efficiently even if the team members span across the planet; we can solve challenging problems by utilizing the vast aggregated computing power of large scale distributed systems. However, if not designed properly, distributed systems may appear to be less dependable than standalone systems. As Leslie Lamport pointed out: "You know you have one (a distributed system) when the crash of a computer you've never heard of stops you from getting any work done" [10]. In this book, we introduce various dependability techniques that can be used to address the issue brought up by Lamport. In fact, with sufficient redundancy in the system, a distributed system can be made significantly more dependable than a standalone system because such a distributed system can continue providing services to its users even when a subset of its nodes have failed.

In this chapter, we introduce the basic concepts and terminologies of dependable distributed computing and system security, and outline the primary approaches to achieving dependability.

1.1 Basic Concepts and Terminologies for Dependable Computing

The term "dependable systems" has been used widely in many different contexts and often means different things. In the context of distributed computing, **dependability** refers to the ability of a distributed system to provide correct services to its users despite various threats to the system such as undetected software defects, hardware failures, and malicious attacks.

To reason about the dependability of a distributed system, we need to model the system itself as well as the threats to the system clearly [2]. We also define common attributes of dependable distributed systems and metrics on evaluating the dependability of a distributed system.

1.1.1 System Models

A system is designed to provide a set of **services** to its users (often referred to as clients). Each service has an **interface** that a client could use to request the service. What the system should do for each service is defined as a set of **functions** according to a *functional specification* for the system. The status of a system is determined by its **state**. The state of a practical system is usually very complicated. A system may consist of one or more processes spanning over one or more nodes, and each process might consist of one or more threads. The state of the system is determined collectively by the state of the processes and threads in the system. The state of a process typically consists of the values of its registers, stack, heap, file descriptors, and the kernel state. Part of the state might become visible to the users of the system via information contained in the responses to the users' requests. Such state is referred to as **external state** and is normally an abstract state defined in the functional specification of the system. The remaining part of the state that is not visible to users is referred to as **internal state**. A system can be recovered to where it was before a failure if its state was captured and not lost due to the failure (for example, if the state is serialized and written to stable storage).

From the structure perspective, a system consists of a one or more **components** (such as nodes or processes), and a system always has a **boundary** that separates the system from its **environment**. Here environment refers to all other systems that the current

system interact with. Note that what we refer to as a system is always relative with respect to the current context. A component in a (larger) system by itself is a system when we want to study its behavior and it may in turn have its own internal structures.

1.1.2 Threat Models

Whether or not a system is providing correct services is judged by whether or not the system is performing the functions defined in the functional specification for the system. When a system is not functioning according to its functional specification, we say a service failure (or simply failure) has occurred. The failure of a system is caused by part of its state in wrong values, *i.e.,* **errors** in its state. We hypothesize that the errors are caused by some **faults** [6]. Therefore, the threats to the dependability of a system are modeled as various faults.

A fault might not always exhibit itself and cause error. In particular, a software defect (often referred to as software bug) might not be revealed until the code that contains the defect is exercised when certain condition is met. For example, if a shared variable is not protected by a lock in a multithreaded application, the fault (often referred to as race condition) does not exhibit itself unless there are two or more threads trying to update the shared variable concurrently. As another example, if there is no boundary check on accessing to an array, the fault does not show up until a process tries to access the array with an out-of-bound index. When a fault does not exhibit itself, we say the fault is **dormant**. When certain condition is met, the fault will be **activated**.

When a fault is activated, initially the fault would cause an error in the component that encompasses the defected area (in programming code). When the component interacts with other components of the system, the error would propagates to other components. When the errors propagate to the interface of the system and render the service provided to a client deviate from the specification, a service failure would occur. Due to the recursive nature of common system composition, the failure of one system may cause a fault in a larger system when the former constitutes a component of the latter, as shown in Figure 1.1. Such relationship between fault, error, and failure is referred to as ”chain of threats” in [2]. Hence, in literature the terms ”faults” and ”failures” are often used interchangeably.

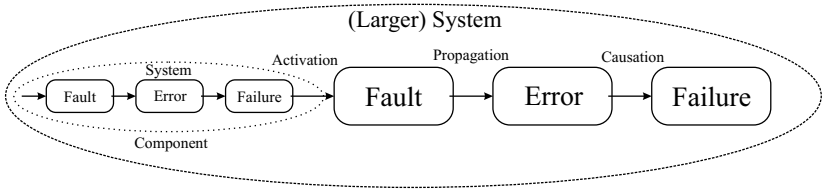

Figure 1.1 An example of a chain of threats with two levels of recursion.

Of course, not all failures can be analyzed with the above chain of threats. For example, a power outage of the entire system would immediately cause the failure of the system.

Faults can be classified based on different criteria, the most common classifications include:

- Based on the source of the faults, faults can be classified as:
  - Hardware faults, if the faults are caused by the failure of hardware components such as power outages, hard drive failures, bad memory chips, etc.
  - Software faults, if the faults are caused by software bugs such as race conditions and no-boundary-checks for arrays.
  - Operator faults, if the faults are caused by the operator of the system, for example, misconfiguration, wrong upgrade procedures, etc.
- Based on the intent of the faults, faults can be classified as:
  - Non-malicious faults, if the faults are not caused by a person with malicious intent. For example, the naturally occurred hardware faults and some remnant software bugs such as race conditions are non-malicious faults.
  - Malicious faults, if the faults are caused by a person with intent to harm the system, for example, to deny services to legitimate clients or to compromise the integrity of the service. Malicious faults are often referred to as commission faults, or Byzantine faults [5].
- Based on the duration of the faults, faults can be classified as:
  - Transient faults, if such a fault is activated momentarily and becomes dormant again. For example, the

race condition might often show up as transient fault because if the threads stop accessing the shared variable concurrently, the fault appears to have disappeared.

  - Permanent faults, if once a fault is activated, the fault stays activated unless the faulty component is repaired or the source of the fault is addressed. For example, a power outage is considered a permanent fault because unless the power is restored, a computer system will remain powered off. A specific permanent fault is the (process) crash fault. A segmentation fault could result in the crash of a process.

- Based on how a fault in a component reveals to other components in the system, faults can be classified as:
  - Content faults, if the values passed on to other components are wrong due to the faults. A faulty component may always pass on the same wrong values to other components, or it may return different values to different components that it interacts with. The latter is specifically modeled as Byzantine faults [5].
  - Timing faults, if the faulty component either returns a reply too early, or too late alter receiving a request from another component. An extreme case is when the faulty component stops responding at all (*i.e.,* it takes infinite amount of time to return a reply), *e.g.,* when the component crashes, or hangs due to an infinite loop or a deadlock.
- Based on whether or not a fault is reproducible or deterministic, faults (primarily software faults) can be classified as:
  - Reproducible/deterministic faults. The fault happens deterministically and can be easily reproduced. Accessing a null pointer is an example of deterministic fault, which often would lead to the crash of the system. This type of faults can be easily identified and repaired.
  - Nondeterministic faults. The fault appears to happen nondeterministically and hard to reproduce. For example, if a fault is caused by a specific interleaving of several threads when they access some shared variable,

it is going to be hard to reproduce such a fault. This type of software faults is also referred to as Heisenbugs to highlight their uncertainty.

- Given a number of faults within a system, we can classify them based on their relationship:
  - Independent faults, if there is no causal relationship between the faults, *e.g.*, given fault A and fault B, B is not caused by A, and A is not caused by B.
  - Correlated faults, if the faults are causally related, *e.g.*, given fault A and fault B, either B is caused by A, or A is caused by B. If multiple components fail due to a common reason, the failures are referred to as common mode failures.

When the system fails, it is desirable to avoid catastrophic consequences, such as the loss of life. The consequence of the failure of a system can be alleviated by incorporating dependability mechanisms into the system such that when it fails, it stops responding to requests (such systems are referred to as **fail-stop** systems), if this is impossible, it returns consistent wrong values instead of inconsistent values to all components that it may interact with. If the failure of a system does not cause great harm either to human life or to the environment, we call such as system a **fail-safe** system. Usually, a fail-safe system defines a set of safe states. When a fail-safe system can no longer operate according to its specification due to faults, it can transit to one of the predefined safe states when it fails. For example, the computer system that is used to control a nuclear power plant must be a fail-safe system.

Perhaps counter intuitively, it is often desirable for a system to halt its operation immediately when it is in an error state or encounters an unexpected condition. The software engineering practice to ensure such a behavior is called fail fast [9]. The benefits of the fail-fast practice are that it enables early detection of software faults and the diagnosis of faults. When a fault has been propagated to many other components, it is a lot harder to pinpoint the source of the problem.

1.1.3 Dependability Attributes and Evaluation Metrics

A dependable system has a number of desirable attributes and some of the attributes can be used as evaluation metrics for

the system. We classify these attributes into two categories: (1) those that are fundamental to, and are immediate concern of, all distributed systems, including availability, reliability, and integrity; and (2) those that are secondary and may not be of immediate concern of, or be applicable to all systems, such as maintainability and safety.

The availability and reliability of a system can be used as evaluation metrics. Other attributes are normally not used as evaluation metrics because it is difficult to quantify the integrity, maintainability, and safety of a distributed system.

1.1.3.1 Availability

Availability is a measure of the readiness of a dependable system at a point in time, *i.e.,* when a client needs to use a service provided by the system, the probability that the system is there to provide the service to the client. The availability of a system is determined by two factors:

- Mean time to failure (MTTF). It characterizes how long the system can run without a failure.
- Mean time to repair (MTTR). It characterizes how long the system can be repaired and recovered to be fully functional again.

Availability is defined to be $MTTF/(MTTF+MTTR)$. Hence, the larger the MTTF, and higher the availability of a system. Similarly, the smaller the MTTR, the higher the availability of the system.

The availability of a system is typically represented in terms of how many 9s. For example, if a system is claimed to offer five 9s availability, it means that the system will be available with a probability of 99.999%, *i.e.,* the system has 10^{-5} probability to be not available when a client wants to access the service offered by the system at any time, which means that the system may have at most 5.256 minutes of down time a year.

1.1.3.2 Reliability

Reliability is a measure of the system's capability of providing correct services continuously for a period of time. It is often represented as the probability for the system to do so for a given period of time t, *i.e.,* $Reliability = R(t)$. The larger the t, the lower the reliability value. The reliability of a system is proportional to MTTF. The

relationship between reliability and availability can be represented as $Availability = \int_0^\infty R(t)$. Reliability is very different from availability. If a system fails frequently but can recover very quickly, the system may have high availability. However, such a system would have very low reliability.

1.1.3.3 Integrity

Integrity refers to the capability of a system to protect its state from being compromised under various threats. In dependable computing research, integrity is typically translated into the consistency of server replicas, if redundancy is employed. As long as the number of faulty replicas does not exceed a pre-defined threshold, the consistency of the remaining replicas would naturally imply the integrity of the system.

1.1.3.4 Maintainability

Maintainability refers to the capability of a system to evolve after it is deployed. Once a software fault is detected, it is desirable to be able to apply a patch that repairs the system without having to uninstall the existing system and then reinstall an updated system. The same patching/software update mechanism may be used to add new features or improve the performance of the existing system. Ideally, we want to be able to perform the software update without having to shutdown the running system (often referred to as live upgrade or live update), which is already a standard feature for many operating systems for patching non-kernal level components. Live upgrade has also be achieved via replication in some distributed systems [12].

1.1.3.5 Safety

Safety means that when a system fails, it does not cause catastrophic consequences, *i.e.,* the system must be fail-safe. Systems that are used to control operations that may cause catastrophic consequences, such as nuclear power plants, or endanger human lives, such as hospital operation rooms, must bear the safety attribute. The safety attribute is not important for systems that are not operating in such environments, such as for e-commerce.

1.2 Means to Achieve Dependability

There are two primary approaches to improving the dependability of distributed systems: (1) *fault avoidance*: build and use high quality software components and hardware that are less prone to failures; (2) fault detection and diagnosis: while crash faults are trivial to detect, components in a practical system might fail in various ways other than crash, and if not detected, the integrity of the system cannot be guaranteed; and (3) *fault tolerance*: a system is able to recover from various faults without service interruption if the system employs sufficient redundancy so that the system can mask the failures of a portion of its components, or with minimum service interruption if the system uses less costly dependability means such as logging and checkpointing.

1.2.1 Fault Avoidance

For software components, fault avoidance aims to ensure correct design specification and correct implementation before a distributed system is released. This objective can be achieved by employing standard software engineering practices, for example:

- More rigorous software design using techniques such as formal methods. Formal methods mandate the use of formal language to facilitate the validation of a specification.
- More rigorous software testing to identify and remove software bugs due to remnant design deficiency and introduced during implementation.
- For some applications, it may be impractical to employ formal methods, in which case, it is wise to design for testability [2], for example, by extensively use unit testing that is available in many modern programming languages such as Java and C#.

1.2.2 Fault Detection and Diagnosis

Fault detection is a crucial step in ensuring the dependability of a system. Crash faults are relatively trivial to detect, for example, we can periodically probe each component to check on its health. If no response is received after several consecutive probes,

the component may be declared as having crashed. However, components in a system might fail in various ways and they might respond promptly to each probe after they have failed. It is nontrivial to detect such faults, especially in a large distributed system. Diagnosis is required to determine that a fault indeed has occurred and to localize the source of the fault (*i.e.*, pinpoint the faulty component). To accomplish this, the distributed system is modeled, and sophisticated statistical tools are often used [3]. Some of the approaches in fault detection and diagnosis are introduced in Chapter 3.

A lot of progress has been made in modern programming language design to include some forms of software fault detection and handling, such as unexpected input or state. The most notable example is exception handling. A block of code can be enclosed with a try-catch construct. If an error condition occurs during the execution of the code, the catch block will be executed automatically. Exceptions may also be propagated upward through the calling chain. If an exception occurs and it is not handled by any developer-supplied code, the language runtime usually terminates the process.

The recovery block method, which is designed for software fault tolerance [8], may be considered as an extension of the programming language exception handling mechanism. An important step in recovery blocks is the acceptance testing, which is a form of fault detection. A developer is supposed to supply an acceptance test for each module of the system. When the acceptance test fails, a software fault is detected. Subsequently, an alternate block of code is executed, after which the acceptance test is evaluated again. Multiple alternate blocks of code may be provided to increase the robustness of the system.

1.2.3 Fault Removal

Once a fault is detected and localized, it should be isolated and removed from the system. Subsequently, the faulty component is either repaired or replaced. A repaired or replaced component can be readmitted to the system. To accommodate these steps, the system often needs to be reconfigured. In a distributed system, it is often necessary to have a notion of membership, *i.e.*, each component is aware of a list of components that are considered part of the system and their roles. When a faulty component is removed from

the system, a reconfiguration is carried out and a new membership is formed with the faulty component excluded. When the component is repaired or replaced, and readmitted to the system, it becomes part of the membership again.

A special case of fault removal is software patching and updates. Software faults and vulnerabilities may be removed via a software update when the original system is patched. Virtually all modern operating systems and software packages include the software update capability.

1.2.4 Fault Tolerance

Robust software itself is normally insufficient to delivery high dependability because of the possibility of hardware failures. Unless a distributed system is strictly stateless, simply restarting the system after a failure would not automatically restore its state to what it had before the failure. Hence, fault tolerance techniques are essential to improve the dependability of distributed systems to the next level.

There are different fault tolerance techniques that can be used to cater to different levels of dependability requirements. For applications that need high availability, but not necessarily high reliability, logging and checkpointing (which is the topic of Chapter 2), which incurs minimum runtime overhead and uses minimum extra resources, might be sufficient. More demanding applications could adopt the recovery oriented computing techniques (which is the topic of Chapter 3). Both types of fault tolerance techniques rely on *rollback recovery*. After restarting a failed system, the most recent correct state (referred to as a checkpoint) of the system is located in the log and the system is restored to this correct state.

An example scenario of rollback recovery is illustrated in Figure 1.2. When a system fails, it takes some time to detect the failure. Subsequently, the system is restarted and the most recent checkpoint in the log is used to recover the system back to that checkpoint. If there are logged requests, these requests are re-executed by the system, after which the recovery is completed. The system then resumes handling new requests.

For a distributed system that requires high reliability, *i.e.*, continuous correct services, redundant instances of the system must be used so that the system can continue operating correctly even if a portion of redundant copies (referred to as replicas) fail.

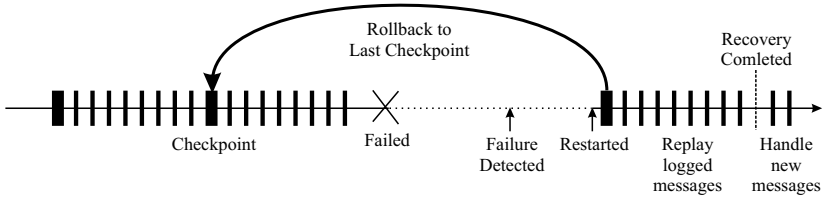

Figure 1.2 The rollback recovery is enabled by periodically taking checkpoints and usually logging of the requests received.

Using redundant instances (referred to as replicas) also makes it possible to tolerate malicious faults provided that the replicas fail independently. When the failed replica is repaired, it can be incorporated back into the system by rolling its state forward to the current state of other replicas. This recovery strategy is called *rollforward recovery*.

An example scenario of rollforward recovery is shown in Figure 1.3. When the failure of the replica is detected and the replica is restarted (possibly after being repaired). To readmit the restarted replica into the system, a nonfaulty replica takes a checkpoint of its state and transfer the checkpoint to the recovering replica. The restarted replica can rollforward its state using the received checkpoint, which represents the latest state of the system.

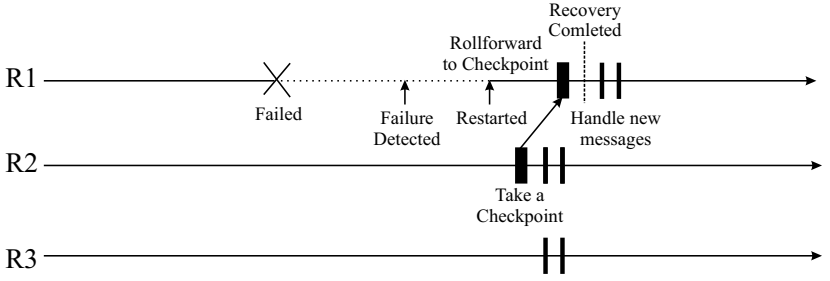

Figure 1.3 With redundant instances in the system, the failure of a replica in some cases can be masked and the system continue providing services to its clients without any disruption.

To avoid common mode failures (*i.e.*, correlated faults), it helps if each replica could execute a different version of the system code. This strategy is referred to as n-version programming [1]. Program transformation may also be used to achieve diversified replicas

with lower software development cost [4]. A special form of n-version programming appears in the recovery block method for software fault tolerance [8]. Instead of using different versions of the software in different replicas, each module of the system is equipped with a main version and one or more alternate versions. At the end of the execution of the main version, an acceptance test is evaluated. If the testing fails, the first alternate version is executed and the acceptance test is evaluated again. This goes on until all alternate versions have been exhausted, in which case, the module returns an error.

1.3 System Security

For a system to be trustworthy, it must be both dependable and secure. Traditionally, dependable computing and secure computing have been studied by two disjoint communities [2]. Only relatively recently, the two communities started to collaborate and exchange ideas, as evidenced by the creation of a new IEEE Transactions on Dependable and Secure Computing in 2004. Traditionally, security means the protection of assets [7]. When the system is the asset to be protected, it includes several major components as shown in Figure 1.4:

- *Operation*. A system is dynamic in that it is continuously processing messages and changing its state. The code as well as the execution environment must be protected from malicious attacks, such as the buffer-overflow attacks.
- *System state*. The system state refers to that in the memory, and it should not be corrupted due to failures or attacks.
- *Persistent state*. System state could be lost if the process crashes and if the process is terminated. Many applications would use files or database systems to store critical system state into stable storage.
- *Message*. In a distributed system, different processes communicate with each other via messages. During transit, especial when over the public Internet, the message might be corrupted. An adversary might also inject fake messages to the system. A corrupted message or an injected message must be rejected.

.

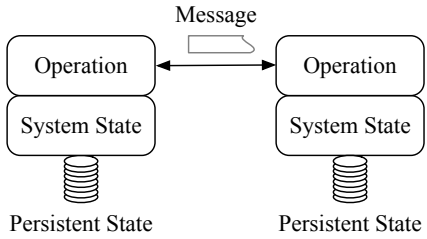

Figure 1.4 Main types of assets in a distributed system.

When we say a system is secure, we are expecting that the system exhibits three attributes regarding how its assets are protected [2]: (1) confidentiality, (2) integrity, and (3) availability. Confidentiality refers to the assurance that the system never reveals sensitive information (system state or persistent state) to unauthorized users. The integrity means that the assets are intact, and any unauthorized modification to the assets, be it the code, virtual memory, state, or message, can be detected. Furthermore, messages must be authenticated prior to being accepted, which would prevent fake messages from being injected by adversaries. The interpretation of availability in the security context is quite different from that in the dependable computing context. Availability here means that the asset is accessible to authorized users. For example, if someone encrypted some data, but lost the security key for decryption, the system is not secure because the data would no longer be available for anyone to access. When combining with dependable computing and in the system context, availability is morphing into that defined by the dependable computing community, that is, the system might be up and running, and running correctly so that an authorized user could access any asset at any time.

An important tool to implement system security is cryptography [11]. Put simply, cryptography is the art of designing ciphers, which scrambles a plaintext in such a way that its meaning is no longer obvious (*i.e.,* the encryption process) and retrieves the plaintext back when needed (*i.e.,* the decryption process). The encrypted text is often called cipher text. Encryption is the most powerful way of ensuring confidentiality and it is also the foundation for protecting the integrity of the system. There are two types of encryption algorithms, one is called symmetric encryption,

where the same security key is used for encryption and decryption (similar to household locks where the same key is used to lock and unlock), and the other one is called asymmetric encryption, where one key is used to encrypt and a different key is used to decrypt. For symmetric encryption, key distribution is a challenge in a networked system because the same key is needed to do both encryption and decryption. The asymmetric encryption offers the possibility of making the encryption key available to anyone who wishes to send an encrypted message to the owner, as long as the corresponding decryption key is properly protected. Indeed, asymmetric encryption provides the foundation for key distribution. The encryption key is also called the public key because it can be made publicly available without endangering the system security, and the decryption key is called the private key because it must remain private, *i.e.,* the loss of the private key will cripple the security of the entire system if built on top of asymmetric encryption. To further enhance the security of key distribution, a public-key infrastructure is established so that the ownership of the public key can be assured by the infrastructure.

Symmetric encryption is based on two basic operations: substitution and transposition. Substitution replaces each symbol in the plaintext by some other symbol aiming at disguising the original message, while transposition alters the positions of the symbols in the plaintext. The former still preserves the order of the symbols in the plaintext, while the latter produces a permutation of the original plaintext and hence would break any established patterns of the symbols. The two basic operations are complementary to each and would make the encryption stronger if used together. This also dictates that the symmetric encryption is going to work on a block of plaintext at a time, which is often referred to as block ciphers. When encrypting a large amount of plaintext using block ciphers, they must be divided into multiple blocks. A naive way of doing encryption would be to encrypt each block separately. Although the encryption can be done in parallel and hence can be quickly done, doing so like this would create a problem: an adversary can reorder some of the cipher texts so that the meaning is completely altered, and the receiver would have no means to detect this! To mitigate this problem, various cipher modes were introduced, such as the cipher block chaining mode and the cipher feedback mode. The essence of the cipher modes is to chain consecutive blocks together

when encrypting them. As a result, any alteration of the relative ordering of the cipher texts would break the decryption.

However, encryption alone is not sufficient to build a secure system. We still need mechanisms for authentication, authorization, and for ensuring non-repudiation, among many other requirements. Highly important cryptographic constructs include cryptographic hash functions (also referred to as one-way or secure hash functions) such as secure hash standard (SHA-family algorithms), message authentication code, and digital signatures.

A cryptographic hash function would hash any given message P and produce a fixed-length bit-string, and it must satisfy a number of requirements:

- The hash function must be efficient, that is, given a message P, the hash value of P, $Hash(P)$, must be quickly computed.
- Given $Hash(P)$, it is virtually impossible to find P. In this context, P is often referred to the preimage of the hash. In other words, this requirement says it is virtually impossible to find a preimage of a hash. It is easy to understand that if P is much longer than $Hash(P)$ in size, this requirement can be easily satisfied because information must have been lost during the hash processing. However, even if P is shorter than $Hash(P)$, the requirement must still hold.
- Given a message P, and the corresponding hash of P, $Hash(P)$, it is virtually impossible to find a different message P' that would produce exactly the same hash, that is, $Hash(P) = Hash(P')$. If the unfortunate event happens where $Hash(P) = Hash(P')$, we would say there is *collision*. This requirement states that it should be computationally prohibitive to find a collision.

The cryptographic hash function must consider every single bit in the message when producing the hash string so that even if a single bit is changed, the output would be totally different. There has been several generations of cryptographic hash functions. Currently the most common ones used are called secure hash algorithms (SHA), which are published as a federal information processing standard by the US National Institute of Standards and Technology. The SHA family of algorithms have four categories: SHA-0, SHA-1, SHA-2, and SHA-3. SHA-0 and SHA-1 both produce a 160-bit string, which

are now considered obsolete. SHA-2, which produces a 256-bit string or a 512-bit string, is used commonly nowadays.

Digital signature is another very important cryptographic construct in building secure systems. A digital signature mimics a physical signature in legal documents, and it must possess the following properties:

- The receiver of a digitally signed document can verify the signer's identity. This is to facilitate authentication of the signer. Unlike in real world, where an official could verify the signer identity by checking for government-issued identification document such as driver's license or passport, the digital signature must be designed in a way that a remote receiver of the digital signature can authenticate the signer based on the digital signature alone.
- The signer of the digital signature cannot repudiate the signed document once it has been signed.
- No one other than the original signer of the signed document could possibly have fabricated the signature.

The first property is for authenticating the signer of a signed document. The second and the third properties are essentially the same because if another person could have fabricated the digital signature, then the original signer could in fact repudiate the signed document. In other words, if the original signer cannot repudiate the signed document, then it must be true that no one else could fabricate the digital signature. Digital signatures are typically produced by using public-key cryptography on the hash of a document. This hash of a document is typically called message digest. The message digest is used because public-key cryptography must use long-keys and it is computationally very expensive compared with symmetric cryptography. In this case, the no-collision requirement for secure hash functions is essential to protect the integrity of digital signatures.

Message authentication code (MAC) is based on secure hash function and symmetric key encryption. More specifically, the sender would concatenate the message to be sent and a security key together, then hash it to produce a MAC. It is used pervasively in message exchanges to both authenticate the sender and to protect the integrity of the message. The basis for authentication is that only the sender and the receiver would know the security key used to generate the MAC. Because of the characteristic of the secure

hash function, if any bit in the message is altered during transmission, the transmitted MAC would differ from the one recomputed at the receiver. Hence, the MAC is also used as a form of checksum with much stronger protection than traditional checksum method such as CRC16.

In conventional systems, communication between a client and server is done over a session. Hence, security mechanisms were designed around this need. At the beginning of the session, the client and the server would mutually authenticate each other. Once the authentication step is done, a session key would be created and used to encrypt all messages exchanged within the session. For a prolonged session, the session key might be refreshed. For sessions conducted over the Web, the secure socket layer (SSL) (or transport layer security) protocol is typically used. The server authentication is done via a digital signature and public-key certificate protected by a public-key infrastructure. Client authentication is typically done via user-name and password. Some enterprise systems, such as directory services, adopt much more sophisticated authentication algorithms based on the challenge-response approach.

REFERENCES

1. A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the IEEE International Computer Software and Applications Conference*, pages 149–155, 1977.
2. A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
3. M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.
4. M. Franz. Understanding and countering insider threats in software development. In *Proceedings of the International MCETECH Conference on e-Technologies*, pages 81–90, January 2008.
5. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.
6. P. M. Melliar-Smith and B. Randell. Software reliability: The role of programmed exception handling. In *Proceedings of an ACM conference on*

Language design for reliable software, pages 95–100, New York, NY, USA, 1977. ACM.

7. C. P. Pfleeger, S. L. Pfleeger, and J. Margulies. *Security in Computing (5th Ed.)*. Pearson, 2015.
8. B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, pages 1–22. John Wiley & Sons Ltd, 1994.
9. J. Shore. Fail fast. *IEEE Software*, pages 21–25, September/October 2004.
10. A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, 2006.
11. A. S. Tanenbaum and D. J. Wetherall. *Computer Networks (5th Ed.)*. Pearson, 2010.
12. L. Tewksbury, L. Moser, and P. Melliar-Smith. Live upgrade techniques for corba applications. In *New Developments in Distributed Applications and Interoperable Systems*, volume 70 of *IFIP International Federation for Information Processing*, pages 257–271. Springer US, 2002.