RELIABILITY AND REPLICATION TECHNIQUES FOR IMPROVED FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

Presentation · April 2020

DOI: 10.13140/RG.2.2.10586.49607

1 author:

Orogun Adebola
Adekunle Ajasin University
13 PUBLICATIONS **22** CITATIONS

SEE PROFILE

RELIABILITY AND REPLICATION TECHNIQUES FOR IMPROVED FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

Orogun A.O

adebola.orogun@aaua.edu.ng

ABSTRACT

A lot of issues have been raised on reliability of distributed systems these days. A distributed system consists of many hardware/software components that are likely to fail or develop fault eventually. In many cases, the failure of a distributed system can result in anything from easily repairable errors to catastrophic melt downs, a reliable distributed system is designed to be as fault tolerance as possible. With the ever increasing dependency being placed on distributed systems, the numbers of users requiring fault tolerance are on the increase. The design and understanding of fault-tolerant distributed systems is a very difficult task. Measures must be put in place to deal with not only all the complex problems of distributed systems when all the components are well, but also the more complex problems when some of the components fail. This paper introduces the basic concepts about system reliability and replication techniques that relate to fault-tolerant computing.

Key words: Reliability, distributed systems, dependancy, replication, fault tolerance, markov's model

1.0 INTRODUCTION

A distributed system is a collection of independent computers that appears to its users as a single coherent system or as a single system. Also, a distributed computing system is a system composed of a large number of computers and communication links, must almost always function with some part of its broken. Over time, only the identity and number of the failed component change. Failures arise from software bugs, human operator errors, performance overload, congestion, magnetic media failure, electronic component failure, or malicious subversion. A distributed system can also be said to be a system consisting of a collection of autonomous machines connected by communication networks and equipped with software systems designed to produce an integrated and consistent computing environment. Distributed systems enable people to cooperate and coordinate their activities more effectively and efficiently.

If we are to talk about a distributed system being reliable, there are two key issues that we must not fail to discuss and these are **fault avoidance** and **fault tolerance**. The goal of fault avoidance is to reduce the likelihood of failures by using conservative design practices, high-reliability components, etc. while the goal of fault tolerance is to ensure that the system continue to function correctly even in the presence of faults.

- **Fault:** A fault is a source that has the potential of generating errors.
- **Error:** An error is the manifestation of a fault within a program, a data structure, a component, or a system
- **Failure:** A failure occurs when the delivered services deviate from the specified services.

1.1 Relationship between fault, error and failure

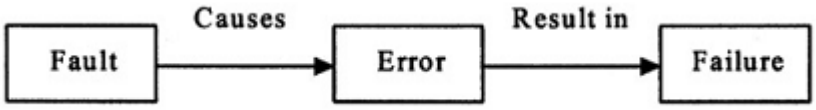

Figure 1: Fault, error and failure

Error occurs as a result of fault which will then result in failure. If faults are prevented or tolerated in a system, there will not be error and we won't have to talk about failure. There are different sources of errors in a particular system and identifying these sources will help us a great deal to prevent such errors and subsequently reducing failure in our systems.

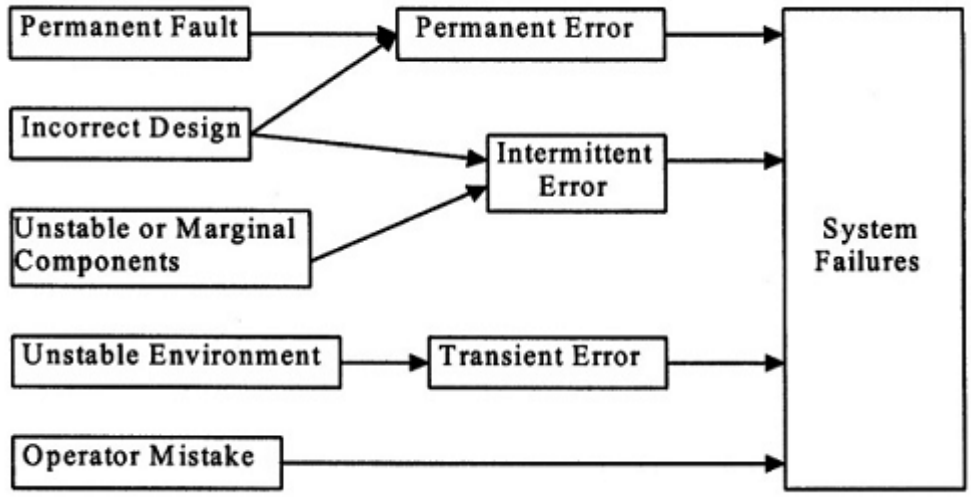

Figure 2: Sources of errors

Permanent error: this is a continuous and stable error, in hardware; permanent failure reflects an irreversible physical change.

Intermittent error: this type of error is only occasionally present due to unstable hardware or varying hardware or software state, such as a function of load or activity.

Transient error: this type of error result from temporary environmental conditions.

Failure rate: Whenever error occurs in a system which leads to failure of the system, it is expected that we look into the failure rate of the particular system. Failure Rate is defined as the expected number of failures of a type of device or system per a given time period. For example, a 32K RAM chip may have a failure rate of 7 failures per million hours.

Sometimes it is convenient to define the failure rate function as a function of time. From the failure rate function we can calculate the failure rate (failures per unit of time) in a given time. The commonly accepted relationship between the failure rate function and time for electronic components is called the bathtub curve, as shown in the figure below. Here is the failure rate and is normally expressed in units of failures per hour (or per million hours).

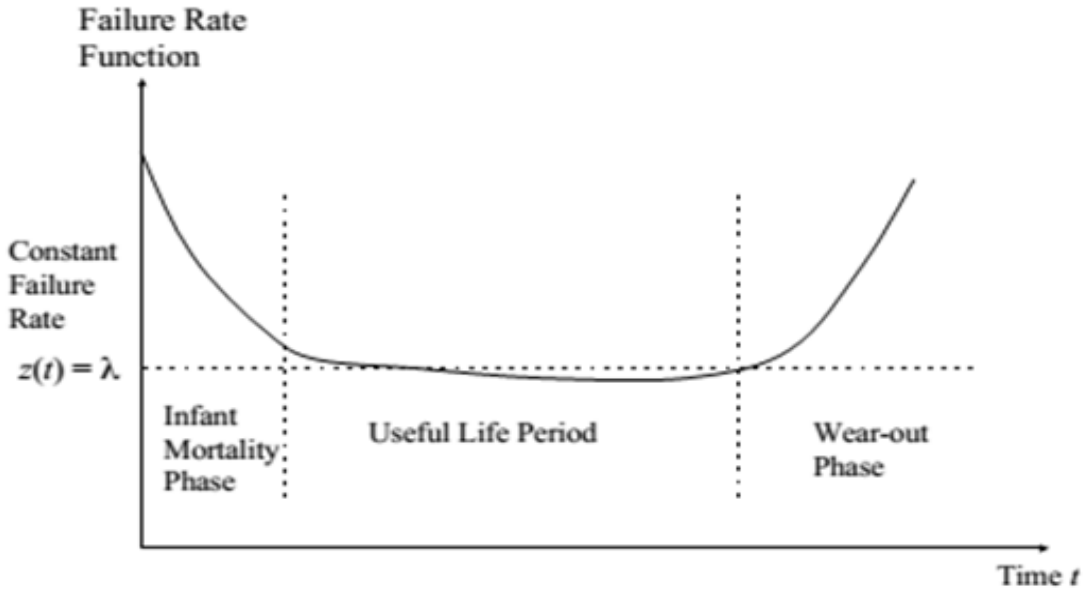

Figure 3: The bathtub curve

Fault coverage: Fault coverage is a measure of a system's ability to perform fault tolerance. There are four primary types of fault coverage: fault detection, fault location, fault containment, and fault recovery coverage, respectively. All these are measures of a system's ability to detect, locate, contain, and recover from faults. The fault recovery coverage is the most commonly considered, and the general term "fault coverage" is often used to mean fault recovery coverage. The mathematical representation for fault coverage is given below:

$$C = P(\text{fault recovery} \mid \text{fault existence}) .$$

Where: C is the coverage

P is the probability

It is explained as the conditional probability that, given the existence of a fault, the system recovers.

1.2 Reliability and Availability

Reliability denotes the ability of a distributed system to deliver its services even when one or several of its software or hardware components fail. It definitely constitutes one of the main expected advantages of a distributed solution, based on the assumption that a participating

machine affected by a failure can always be replaced by another one, and not prevent the completion of a requested task. For instance, a common requirement of large electronic Web sites is that a user transaction should never be canceled because of a failure of the particular machine that is running that transaction. An immediate and obvious consequence is that reliability relies on redundancy of both the software components and data. At the limit, should the entire data center be destroyed by an earthquake, it should be replaced by another one that has a replica of the shopping carts of the user. Clearly, this has a cost and depending on the application, one may more or less fully achieve such resilience for services, by eliminating every single point of failure.

Reliability is a conditional probability that a system survives for the time interval [0, t], given that it was operational at time t=0. That is, the reliability R of a system is a function of time t:

$$R(t) = Pr\ \{0 \text{ failures in } [0, t] \mid \text{no failure at } t = 0\}.$$

Let $N_0(t)$ be the number of components that are operating correctly at time t, $N_f(t)$ be the number of components that have failed at time t, and N be the number of components that are in operation at time t:

$$R(t) = \frac{N_o(t)}{N} = \frac{N_o(t)}{N_o(t) + N_f(t)}$$

Similarly, we can define the unreliability Q as:

$$Q(t) = \frac{N_f(t)}{N} = \frac{N_f(t)}{N_o(t) + N_f(t)}$$

Of course, at any time t, R(t) = 1.0 - Q(t).

If we assume that the system is in the useful-life stage where the failure rate function has a constant value of λ, then the reliability function is well known to be an exponential function of parameter λ (exponential failure law):

$$R(t) = e^{-\lambda t}$$

Dependability: this is the quality of the delivered services. High dependability means that reliance can justifiably be placed on a system. Fault-tolerant computing is concerned with the method of achieving computer system dependability.

Availability: this is the intuitive sense of reliability. A system is reliable if it is able to perform its intended function at the moment the function is required or needed. Formally, it is the probability that the system is operational at the instance of time t. To understand the availability of a system, we have to classify the following concepts:

- **Mean time to failure (MTTF):** MTTF is the expected time that a system will operate before the first failure occurs.

$$MTTF = \frac{\sum_{i=1}^{N} t_i}{N}$$

where N is the number of identical systems measured, t_i is the time that system i operates before encountering the first failure, and the start time is t (=0). If the reliability function obeys the exponential failure law, then

$$MTTF = \int_0^\infty e^{-\lambda t} dt = \frac{1}{\lambda}$$

- **Mean time to repair (MTTR):** MTTR is the average time required to repair a system. If the i^{th} of N faults requires a time to repair, the MTTR is estimated as:

$$MTTR = \frac{\sum t_i}{N}$$

The MTTR is normally specified in terms of a repair rate which is the average number of repairs that occur per time period (hour):

$$MTTR = \frac{1}{\mu}$$

- **Mean time between failures (MTBF):** MTBF is the average time between failures of a system. For example, if there are N systems and each of them is operated for some

time T and the number of failures encountered by the i^{th} system is recorded as n_i. The average number of failures is computed as:

$$n_{avg} = \sum_{i-1}^{N} \frac{n_i}{N}$$

Finally, the MTBF is

$$MTBF = \frac{T}{n_{avg}}$$

If we assume that all repairs to a system make the system perfect once again just as it was when it was new, the relationship between the MTTF and MTBF is: MTBF = MTTF + MTTR, as illustrated in Figure 4 below.

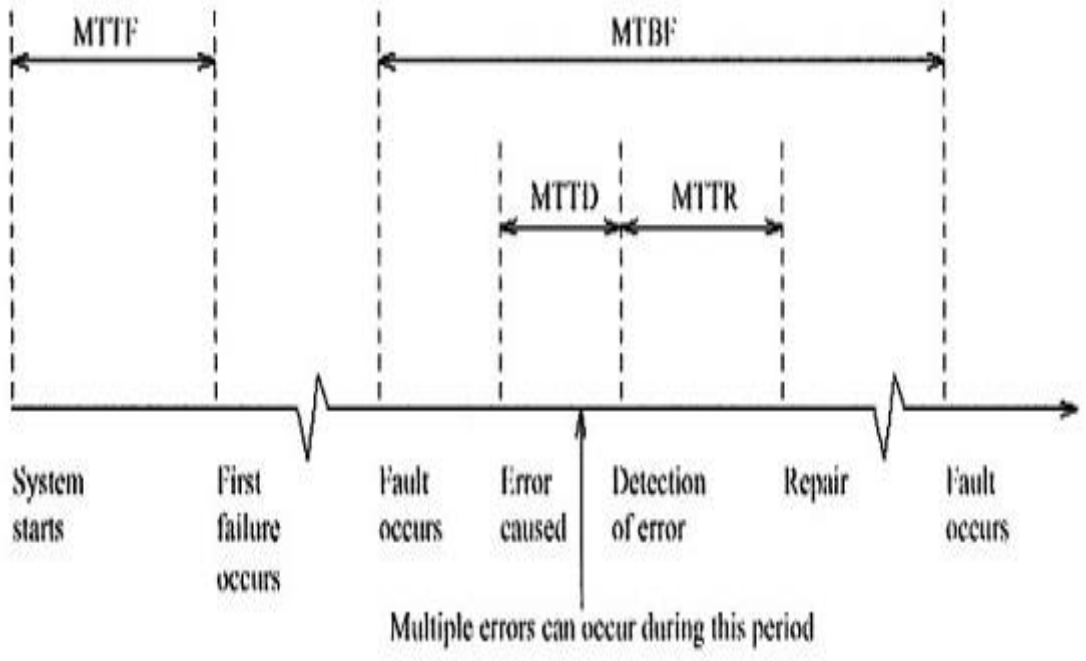

Figure 4: Relationships between MTBF, MTTF, and MTTR.

1.3 Classification of Failures

According to Cristian 1991, different classes of failure can exist in a distributed system with respect to services provided by servers. The various classes of failure are as listed below:

i. **Omission failure:** this type of failure occurs when a server omits to respond to an input. It could be due to lost requests from clients.

ii. **Timing failure:** a failure occurs when a server's response is functionally correct but untimely. Timing failures can be either early or late. The cause of this kind of failure might be that the server is busy with a high priority job, the heavy traffic of network, queuing problem, and so on.

iii. **Response failure:** a failure occurs when a server responds incorrectly. This failure could be caused by design errors incurred in implementing the server, or the protocol used to communicate between clients and the server.

iv. **Crash failure:** a crash failure occurs when a server stops running. This failure could be a hardware failure, the machine being switched off deliberately, or a detected error, possibly non-recoverable, that brings the system to a fail-stop.

2.0 TECHNIQUES TO ACHIEVE RELIABILITY

There are a number of techniques used to build reliable distributed network systems some of which are as listed here: redundancy, fault avoidance techniques, fault detection techniques, and fault tolerance techniques.

2.1 Redundancy

Redundancy is the basic requirement in any technique to achieve reliability. Redundant elements are those system components that can be removed without affecting the performance of a system (assuming that the remaining system is fault free). Redundancy provides information about the state of the system and the components that are needed for recovery from failures. In terms of using resources, there are space and time redundancies respectively:

Space redundancy:

Hardware redundancy: uses extra gates, memory cells, bus lines, functional modules, and other hardware to supply recovery information.

- Software redundancy: uses alternate or independent versions of algorithms to provide recovery information.

Time Redundancy: uses extra computing or execution by the same or different methods to retry the failed operation and provide a basis for subsequent action.

Classification of Redundancies

i. Static Redundancy: this employs redundant components to mask the effects of hardware or software failures within a given module. The output of the module remains unaffected, i.e., error free, as long as the protection due to redundancy is effective. This module assumes that failures of redundant elements are independent, not related.

ii. Dynamic Redundancy: this allows errors to appear at the output of modules. When a fault is detected, a recovery action eliminates or corrects the resulting error. Dynamic redundancy implies a requirement for fault detection, fault location and recovery processes.

2.2 Fault Avoidance Techniques

These are the techniques used to reduce the probability of failure and are intended to decrease the possibility of transient faults by manipulating factors that affect the failure rates. Environmental changes, quality changes, and complexity change are three Fault Avoidance Techniques to obtain a lower system failure rate.

2.3 Fault Detection Techniques

Fault detection techniques recognize the inevitability of eventual failure, no matter how well the system is designed. The key to fault detection is redundancy. Fault Detection Techniques include:

i. Duplication: Duplication can detect all non-overlapping, single faults except faults in the comparison unit. When a failure occurs in one copy, the two copies are no longer identical and a simple comparison detects the fault. Identical faults from identical modules are not detectable because both copies are in agreement.

ii. Error-Detection Code: e.g. Parity, Checksums or Charity Codes, they are the systematic application of redundancy to information.

iii. Self-Checking and Fail-Safe: a self-checking system checks its own behaviour against some pre-specified specifications to ensure that it is doing the right thing.

iv. Watchdog Timers and Timeouts: watchdog timers provide a simple and inexpensive way to keep track of proper process functioning.

v. Consistency and Capability Checking: consistency checking is performed by verifying that intermediate or final results are within a reasonable range either on an absolute basis or as a function of the inputs. Capability checking is usually part of the operating system, but it can also be implemented in hardware.

2.4 Fault Tolerance Techniques

Fault tolerance techniques can be categorized into Static Redundancy and Dynamic Redundancy.

- Static redundancy uses redundancy to provide fault tolerance by either isolating or correcting faults before the faulty results reach module outputs.
- Dynamic redundancy techniques involve the reconfiguration of system components in response to failures. The reconfiguration prevents failures from contributing their effects to the system operation.

3.0 SOFTWARE FAULT TOLERANCE

Why do we require fault tolerance in software? Software does not degrade physically as a function of time or environmental stresses. A program that has once performed a given task as specified will continue to do so provided that none of the following change: the input, the computing environment, or user requirements.

However, the above factors do change. So, past and current failure-free operation cannot be taken as a dependable indication that there will be no failures in the future. Failure experience in current software has shown that existing software products exhibit a fairly constant failure frequency.

Software failures usually differ in their impact on the operations of an organisation. Therefore, we need to classify them by severity and come up with the failure

intensity or reliability for each classification. At least three classification criteria are in common use: cost impact, human impact, and service impact.

Difficulties in software test and verification:

- Difficulties in testing: testing can be used to show the presence of bugs, but never show their absence. Complete testing of any practical programs is impossible because of the vast number of possible input combinations.
- Difficulties in verification: formal verification has been applied on an experimental basis and usually to small programs (too formal, difficult to use, expensive in handling large programs, difficult in real-time program verification, and difficult in translating a natural language specification into a formal specification).

3.1 Techniques for Software Fault-tolerance

The Design techniques for software fault tolerance include the following:

- N-version programming and
- recovery blocks.

N-version programming: The independent generation of N (>=2) functionally equivalent programs, called versions, from the same initial specification.

When N=2, N-version programming can be expected to provide good coverage for error detection but may be found wanting in assuring continued operation of the software. Upon disagreement among the versions, three alternatives are available:

- Retry or restart (in this case fault containment rather than fault tolerance is provided);
- Retransmission to a “safe state,” possibly followed by later retries;
- Reliance on one of the versions, either designed in advance as more reliable or selected by a diagnostic program.

For N >= 3, a majority voting logic can be implemented. N=3 is the most commonly used method. A 3-version programming requires:

- Three independent programs, each furnishing identical output formats.
- An acceptance program that evaluates the output of requirement 1.
- A driver (or executive segment) that invokes requirements 1 and 2 and furnishes the results to other programs.

Recovery block: A recovery block is a block in the normal programming language sense, except that, at the entrance to the block, it is an automatic recovery point and at the exit it is an acceptance test. The acceptance test is used to test if the system is in an acceptable state after the first execution of the block, or primary module as it is often called. The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and the second alternative module being executed. If the second module also fails the acceptance test, then the program is restored to the recovery point and yet another alternative module is executed. If all the alternatives are exhausted, the system fails. Therefore, the recovery must take place at a higher level. The recovery block is used for safely accessing critical data. It also detects errors through the acceptance test after running each alternative. It could be designed to detect further errors by thoughtfully designing alternatives.

A recovery block consists of three software elements which are as stated below:

- A primary routine, which executes critical software functions;
- An acceptance test, which tests the output of the primary routine after each execution;
- An alternate routine, which performs the same function as the primary routine (but may be less capable or slower), and is invoked by the acceptance test upon detection of a failure.

The structure of a recovery block can be described as follows:

```
Structure:
Ensure T
    By P
    Else by Q
Else Error
```

where T is the acceptance test condition that is expected to be met by successful execution of either the primary routine P or alternate routine Q. The structure is easily expanded to accommodate several alternates Q_1, Q_2, …, Q_n

Error Recover: Error recovery is a mechanism which brings a system to an errorfree state. There are two approaches to recovery: backward recovery and forward recovery. The backward recovery restores the system to a prior state, which is referred to as a recovery point. The forward recovery attempts to continue from an erroneous state by making selective corrections to the system state.

The act of establishing the recovery point is the checkpointing which is a process of checking and saving the current system state for recovery. Rollback is one of the backward recovery mechanisms. Rollback is often employed by database systems to return the system state from the unfinished transaction to the previous consistent state by undoing all the steps that have been done since the transaction was started.

Domino effect: If two concurrent processes are interacting with each other, one process that rolls back to its recovery point might cause the other one to roll back to its recovery point as well. This procedure could go on until the whole system returns to its original state. This phenomenon is called the domino effect. The cause of this phenomenon is that each process is designed to have its own recovery point which does not consider others' recovery points.

Logging: Logging (or audit trail) is a way used to keep track of modifications or operations since last checkpointing, so the system can be able to roll back to the latest consistent state. There are various logging mechanisms designed to suit a variety of applications. Logs are saved to permanent storage (disk files) to survive any crashes.

4.0 RELIABILITY MODELLING

This session introduces how to build a model to properly describe system reliability.

4.1 Combinatorial Models

This discussed under two categories

a. Series System

A series system is one in which each element is required to operate correctly for the system to operate correctly (no redundancy).

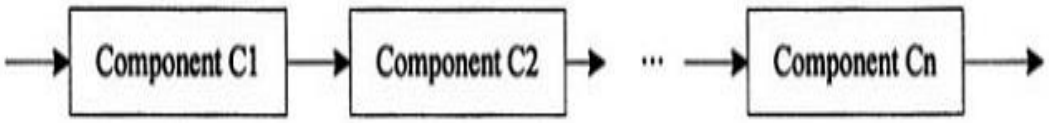

Figure 5: Reliability block diagram of a series system

Let $C_{iw}(t)$ represent the event that component C_t is working properly at time t, $R_i(t)$ is the reliability of component at time t, and $R_{series}(t)$ is the reliability of the series system. The reliability at any time t is the probability that all N components are working properly:

$$R_{series}(t) = P(C_{1w}(t) \cap C_{2w}(t) \cap ... \cap C_{Nw}(t))$$

Assume that the events are independent, then

$$R_{series}(t) = R_1(t)R_2(t)...R_N(t) = \prod_{i=1}^{N} R_i(t)$$

Suppose each component satisfies the exponential failure law such that the reliability of each component is. Then

$$R_{series}(t) = e^{-\lambda_1 t}...e^{-\lambda_N t} = e^{-\lambda_{system} t}$$

Where $\lambda_{system} = \sum_{i=1}^{N} \lambda_i$ and corresponds to the failure rate of the system**.**

Parallel Systems

A parallel system is one in which only one of several elements must be operational for the system to perform its functions correctly.

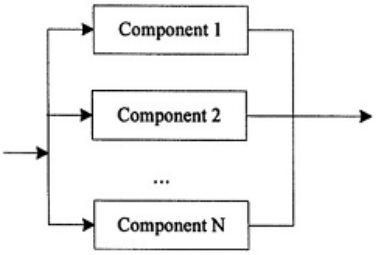

Figure 6: Reliability block diagram of the parallel system

The unreliability of a parallel system can be computed as the probability that all the N elements fail. Let $C_{if}(t)$ represent the event that element i has failed at t, $Q_{parallel}(t)$ be the unreliability of the parallel system, and $Q_i(t)$ be the unreliability of the i^{th} element. Then

$$Q_{parallel}(t) = P(C_{1f}(t) \cap C_{2f}(t) \cap \ldots \cap C_{Nf}(t))$$
$$= Q_1(t)Q_2(t)\ldots Q_N(t) = \prod_{i=1\ldots N} Q_i(t)$$

So the reliability of the parallel system is:

$$R_{parallel}(t) = 1.0 - Q_{parallel}(t) = 1.0 - \prod_{i=1\ldots N} Q_i(t) = 1.0 - \prod_{i=1\ldots N} (1.0 - R_i(t)).$$

4.2 Markov Models

In probability theory, a Markov model is a stochastic model used to model randomly changing systems where it is assumed that future states depend only on the present state and not on the sequence of the events that preceed it (that is, it assumes the markov property). Generally, this assumption enables reasoning and computation with the model that would otherwise be intractable.

This diagram below represents the Markov model in which each state transition (the change of state that occurs within a system; as time passes, the system goes from one state to another) is associated with a transition probability that describes the probability of that state transition occurring within a specified period of time.

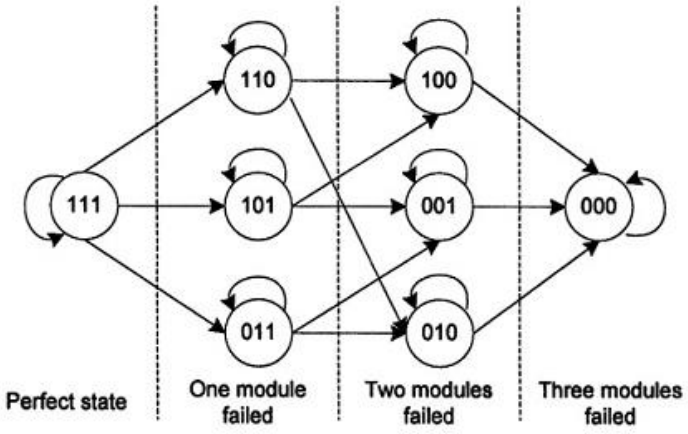

Figure 7: State diagram of a TRM system

The states 000, 001, 010, 100 in Figure 9.11 represent states in which the system has ceased to function correctly. Assume that the system does not contain repair and that only one failure will occur at a time. The above states can be partitioned into three categories: the perfect state (111); the one-failed states (110), (101), and (011), the system failed states (100), (001), (101), and (000). They can be used to reduce the Markov model.

Reduced Markov Model: Let state 3 correspond to the state in which all three modules are functioning correctly; state 2 is the state in which two modules are working correctly; state 1 is the failed state in which two or more modules have failed.

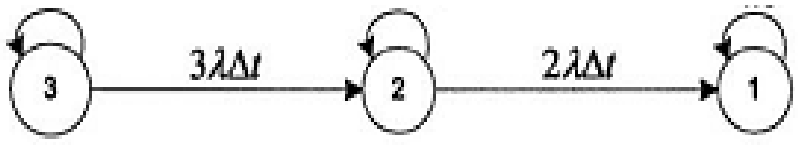

Figure 8: Reduced state diagram of a TRM system

4.3 Fault Coverage and Its Impact on Reliability

Consider a simple parallel system consisting of two identical modules shown in Figure 9.13.

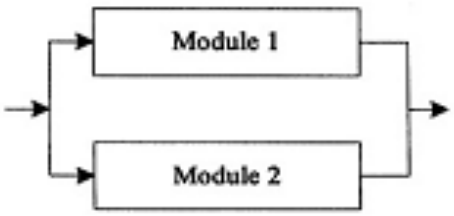

Figure 9: Reliability block diagram of a simple parallel system

Assume that module 1 is the primary and module 2 is switched in if module 1 fails. But this depends on the ability to detect and handle the faults. So,

$$R_{system}(t) = R_1(t) + (1 - R_1(t))C_1R_2(t)$$

where is the fault coverage of module 1, is the reliability of module i. If the reliabilities and fault coverage factors of the two modules are the same, then

$$R_{system}(t) = R(t) + (1 - R(t))CR(t).$$

This is a linear function between the system reliability R and the fault coverage C, as depicted in Figure 9.14. It is observed that with the increase of C, the system is more and more reliable. From Figure 9.14, we can see: If C = 1.0 (perfect parallel system), then

$$R_{system}(t) = 2R(t) - R(t)^2 = 1 - (1 - R(t))^2.$$

If C = 0.0, the reliability expression reduces to the reliability of one module.

4.4 M-of-N Systems

M-of-N systems is the generalisation of the ideal parallel system, where M of the total N identical modules are required to function for the system to function. TMR (triple module redundancy) is a 2-of-3 system. Suppose that we have a TMR system. If we ignore the reliability of the voter, then

$$\begin{aligned} R_{TMR} &= R_1(t)R_2(t)R_3(t) + R_1(t)R_2(t)(1 - R_3(t)) \\ &\quad + R_1(t)(1 - R_2(t))R_3(t) + (1 - R_1(t))R_2(t)R_3(t) \end{aligned}$$

where is the reliability of the module. If then

$$R_{TMR} = R^3(t) + 3R^2(t)(1 - R(t)) = 3R^2(t) - 2R^3(t).$$

Comparison of the reliabilities of TMR and a single module is shown in Figure 9.15. It is easy to find the crossover point: which implies the quadratic equation

$$R^2 - \frac{3}{2}R + 0.5 = 0.$$

The two solutions of this function are 0.5 and 1.0. This example also shows that a system can be tolerant of faults and still have a low reliability. We can also calculate the reliability of a general M-of-N system:

$$R_{M-of-N} = \sum_{i=0}^{N-M} \binom{N}{i} R^{N-i}(t)(1-R(t))^i$$

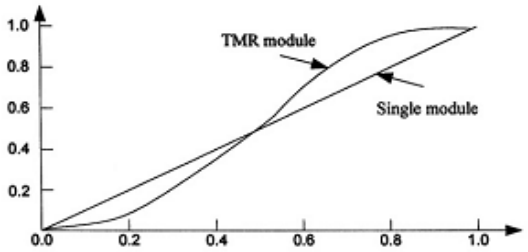

Figure 10: Reliability comparison of TMR and a single module

where

$$\binom{N}{i} = \frac{N!}{(N-i)!i!}.$$

For example, the TMR system reliability can be derived from the above formula:

$$R_{TMN} = \sum_{i=0}^{1} \binom{3}{i} R^{3-i}(t)(1-R(t))^i = 3R^2(t) - 2R^3(t).$$

5.0 FAULT TOLERANT DISTRIBUTED ALGORITHMS

The various fault tolerant distributed algorithm for distributed systems are discussed in this section.

5.1 Distributed Mutual Exclusion

The problem of mutual exclusion frequently arises in distributed systems whenever concurrent access to shared resources by several sites/processes is involved. For correctness, it is necessary to ensure that the shared resource be accessed by a single site/process at a time. This requires that concurrent access to a shared resource by several uncoordinated user-requests be serialised to secure the integrity of the shared resource. It requires that the action performed by a user on a shared resource must be atomic. That is, if several users concurrently access a shared resource, then the actions performed by a user, as far as the other users are concerned, must be instantaneous and indivisible. Hence the net effect on the shared resource is the same as if the actions were executed serially, as opposed to an interleaved manner.

Mutual exclusion is a fundamental issue in the design of distributed systems and provides an efficient and robust technique for their viable design.

Mutual exclusion algorithms

At any instant, a site/process may have several requests for the critical section (CS). The site/process queues up these requests and serves them one at a time. A site/process can be in one of the following three states: requesting CS, executing CS, or idle (neither requesting nor executing CS). In the requesting CS state, the site/process is blocked and cannot make further requests for CS. In the idle state, the site/process is executing outside its CS. In the token based algorithms, a site/process can also be in the idle token state. That is, the site/process holds the token and is executing outside the CS.

The primary objective of a mutual exclusion algorithm is to guarantee that only one request accesses the CS at a time. In addition, the following characteristics are considered important in a mutual exclusion algorithm:

- Freedom from deadlocks. Two or more sites/processes should not endlessly wait for messages that will never arrive.
- Freedom from starvation. A site/process should not be forced to wait indefinitely to execute CS while other sites/processes are repeatedly executing CS. That is, in a finite time, all requesting sites/processes should have an opportunity to execute the CS.
- Fairness. Fairness implies freedom from starvation (but not vice-versa). It requires that requests are executed in a certain order (e.g., the order in which they arrive at the CS, or in which they were issued).
- Fault tolerance. A mutual exclusion algorithm is fault-tolerant if in the wake of a failure, it can reorganise itself so that it continues to function without any (prolonged) disruptions.

In a simple solution to distributed mutual exclusion, a site/process, called the control site/process, is assigned the task of granting permission for the CS execution. To request the CS, a site/process sends a REQUEST message to the control site/process. The control site/process queues up the requests for the CS and grants them permission one by one.

Token-based and non-token-based algorithms

During the 1980s and 1990s, the problem of mutual exclusion received considerable attention and several algorithms to achieve mutual exclusion in distributed systems were proposed. They tend to differ in their network topology (e.g., bus, tree, ring, etc.) and in the amount of information maintained by each site/process about other sites/processes. These algorithms can be classified into the following two groups:

- Token-based: a unique token (also known as the PRIVILEGE message) is shared among the sites/processes. A site/process is allowed to enter CS if it possesses the token and it continues to hold the token until the execution of the CS is over. These algorithms essentially differ in the way a site/process carries out the search for the token.
- Non-token-based (time ordering): they require two or more rounds of message exchanges among the sites/processes. These algorithms are assertion based because a site/process can enter its CS when an assertion defined on its local variable becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site/process at any given time.

5.2 Election Algorithms

An election algorithm carries out a procedure to choose a site/process from a group, for example, to take over the role of a failed site/process. The main requirement is that a unique site/process is elected, even if several sites/processes call elections concurrently.

Basically, there are two types of election algorithms: one is the ring-based election algorithms and the other is the broadcast election algorithms. The ring-based election algorithms assume that an order (a physical or logical ring) exists among all the sites/processes and the election messages flow along the ring. A site/process (say, i) wanting to be elected sends a message of (REQ, i) along the ring. Another site/process (say, j) forwards the request to its successor if i > j (or equivalently, i < j). Otherwise, the message (REQ, j) is sent instead. At the end of the message circulation, the site/process with the highest (lowest) number will be elected and the message of (ELECTED, k) (k is the highest (lowest) number among all sites/processes) is sent to all sites/processes.

The broadcast election algorithms assume that a site/ process knows the identifiers and addresses of all other sites/processes. A site/process (say A) begins an election by sending an

election message to those sites/processes that have a higher identifier. If none of the sites/processes return an answer message within a certain time, A considers itself as elected. A then sends a coordinator message to all sites/processes with a lower identifier and these sites/processes will treat A as elected. If a site/process (say, B) receives an election message from A (that means A has a lower identifier than B), B sends back an answer message to A and then B starts another round of election.

5.3 Deadlock Detection and Prevention

In a distributed system, a process can request and release local or remote resources in any order and a process can request some resources while holding others. If the sequence of the allocation of resources to processes is not controlled in such environments, deadlock can occur.

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any events that could wake up any of the other members of the set, and all the processes continue to wait forever.

A resource allocation graph (a directed graph) can be used to model deadlocks. In such a graph:

- Circles represent processes.
- Squares represent resources.
- An arc from a resource node to a process node means that the resource previously has been requested by, granted to, and is currently held by that process.
- An arc from a resource node to a process node means that the resource previously has been requested by, granted to, and is currently held by that process.
- An arc from a process node to a resource node means that the process is currently blocked or waiting for that resource.

Figure 11, shows a deadlock using a resource allocation graph, where process A holds resource R1 and is requesting resource R2 At the same time, process B holds resource R2 and is requesting resource R1

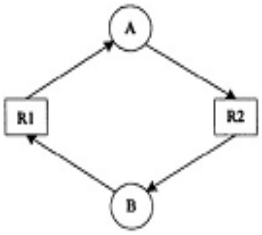

Figure 11: Deadlock

Distributed Deadlock Detection

Centralised algorithm. A central coordinator maintains the resource graph for the whole system (the union of all the individual graphs on each site). Whenever there is a change in a local graph, the coordinator must know this by some message-passing facility. When the coordinator detects a cycle, it kills off one process to break the deadlock.

Because of the message delays, false deadlock may occur. Figure 12 shows a false deadlock example. At a particular time, the resource allocation graphs on Sites 1, 2, and 3 are shown in Figure 9.17 (a), (b) and (c), respectively. Now assume that two messages are sent by Site 1 and Site 2 to the coordinator:

- **Msg$_0$**: Site 1 sends a message to the coordinator announcing the release of by B;
- **Msg$_1$:** Site 2 sends a message to the coordinator announcing that B is waiting for resource R_3.

A false deadlock occurs if arrives first. Figure 12 (d) shows the false deadlock.

Distributed algorithm. Suppose that processes are allowed to request multiple resources simultaneously. The Chandy-Misra-Haas algorithm works as follows:

1. When a process has to wait for some resources held by other process(es), a probe message is generated and sent to the process(es) holding the needed resources. The message consists of three numbers: the process that is just blocked, the process sending the message, and the process to whom it is being sent.

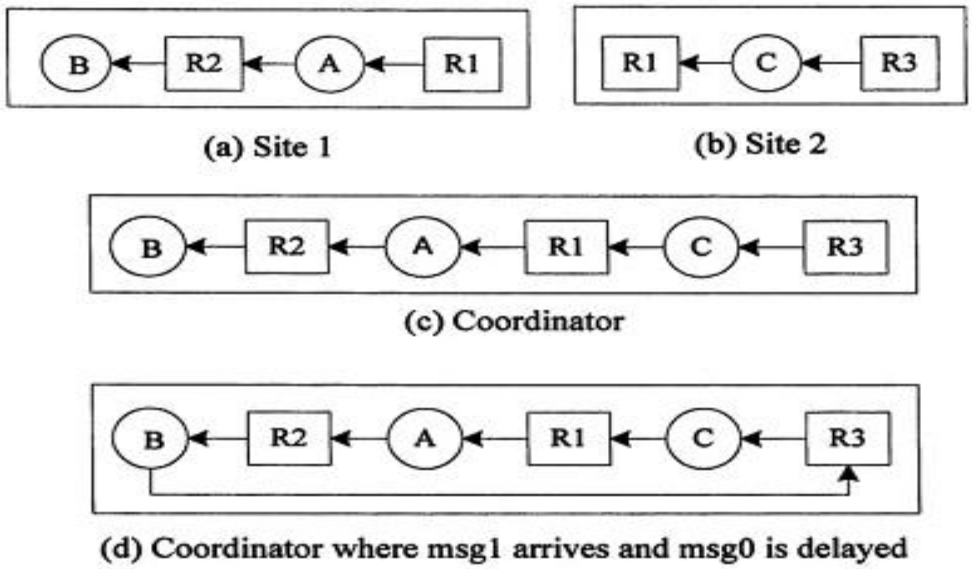

Figure 12: False deadlock

2. When the message arrives, the recipient checks to see if it itself is waiting for any processes. If so, the message is updated, keeping the first field but replacing the second field by its own process number and the third one by the number of the process it is waiting for. The message is sent to the process on which it is blocked. If it is blocked on multiple processes, all of them are sent messages.
3. If a message goes all the way around and comes back to the original sender, that is, the process listed in the first field, a cycle exists and the system is deadlocked.

Distributed Deadlock Prevention

To prevent the deadlock, a system is carefully designed so that deadlocks are structurally impossible. Some possible techniques are: A process can only hold one resource at a time. All processes must request all their resources initially. Processes must release all resources when asking for a new one. All resources are ordered and processes acquire them in strictly increasing order. So a process can never hold a high resource and ask for a low one, thus making cycles impossible.

When one process is about to block waiting for a resource that another process is using, a check is made to see which one has a larger timestamp (younger). We can then allow the wait only if the waiting process has a lower timestamp (older) than the process waited for. So, following any chain of waiting processes, the timestamps only increase, so cycles are impossible. Alternatively, we can use higher timestamps (older).

6.0 REPLICATION AND RELIABILITY

High-speed networks make it possible to run distributed software on multiple machines efficiently. With the ever-growing dependency being placed on distributed computing systems, the requirement for their reliability has increased enormously. A number of techniques have been proposed for the construction of reliable and fault-tolerant systems. One of these techniques is to replicate critical system service on multiple machines connected by networks so that if one copy (replica) fails, other replicas can still provide the continuing service.

When failures occur in hardware/software, the system may generate incorrect results or may simply stop before finishing the intended computation. Therefore, failures in distributed systems can have different semantics, and in turn, they require individual treatments [Cristian 1991]. Distributed systems are typically subject to two kinds of failures: site failure and communication link failure, which can result in the following failure semantics:

- Fail-stop failure [Schlichting and Schneider 1983]. Fail-stop failure is used to describe a process/processor which either works correctly, or simply stops working without taking any incorrect action. The fail-stop process/processor has the property of informing others by a notification service upon the failure or remaining in a state that the failure is detectable to others. There is also another term Fail-silent failure [Power 1994]. Fail-silent failure exhibits the same halt-on-failure semantics as the fail-stop failure. But the failed process/processor may not have the capability of notifying others, nor is able to be detectable to others.
- Network link failure [Tanenbaum 1996]. This refers to the breakdown of a communication link between sites. The link failure makes it impossible to send or receive messages over the failed links. Also messages in transmission can be lost.
- Network partition failure [Birman 1996]. Network link failures can lead to partition failure, where a group of sites involved in a distributed system is partitioned into a set of subgroups, of which members of the same subgroup can communicate but not with members of different subgroups.
- Timing failure [Johnson 1989]. This refers to a violation of assumed temporal property of the system, such as clock drift bound between machines, or a message transmission delay between sites linked by networks.

- Byzantine failure [Lamport et al 1982]. This refers to any violation of the system behavior. In particular, it is used to refer to corrupted messages, such as malicious messages, that give wrong instructions, and as a result, may bring down the system.

To be able to detect the failure of a process/processor, failure-detecting techniques are needed. Traditionally, a technique often used is that the detecting process sends a message asking "Are you alive?" to the remote process/processor to check whether the remote process/processor is operating or not. If the remote process/processor responds within a predetermined time period, it indicates the aliveness of the remote process/processor; otherwise, the detecting process will time out and assume that the remote process/processor is dead. This method is based on a pre-assumption that the system is a synchronous distributed system. A synchronous distributed system has an upper-bound for message transmission delay [Mullender 1993]. Therefore, the time-out can be set statically to a value larger than the upper-bound.

On the contrary, an asynchronous distributed system does not have such an upper-bound. Thus, the time-out can not be used as a criterion for detecting the failure of a process/processor. The asynchronous distributed environment is close to reality where a message transmission can be delayed indefinitely. However, such a system is hard to implement because of the uncertainty of message transmission delay. An implementation that works for an asynchronous distributed system should work for a synchronous system [Cristian 1991, 1996].

Most techniques for achieving fault-tolerance rely on introducing extra redundant components to the system in order to detect and recover from component failures. Employing redundant components is a common concept in real life. An example of such application is aircraft, it has four engines so that if any of the engines shuts down, the remaining engines can still keep the craft in the air and land safely. Computer hardware systems often employ duplicated parts (i.e., dual processors) to survive partial failures as well.

The client/server distributed computing model cannot be regarded as a reliable model, as the server is the single processing point. In the face of a site failure or a communication link failure, the server becomes inaccessible. Distributed replication is then a technique to solve this problem. With a large number of powerful machines available on the network, it becomes possible to duplicate a critical server on multiple machines so that if one server fails,

the failure can be masked by its replicas. In turn, the service is continued. Distributed replication systems make three major contributions:

- increasing the availability,
- achieving fault-tolerance, and
- improving the performance.

Increasing the availability. Availability refers to the accessibility of a system service. For a non-replicated server system, if the server breaks down, the service becomes unavailable. With a replicated server system, high availability can be achieved by software and/or data being available on multiple sites. Thus, if one server is down, the remaining replicas can still provide the service.

Fault tolerance. The ability to recover from component failures to a system consistent state without performing incorrect actions is said to be fault-tolerance. In a distributed replication system, the failure of one replica due to a process/processor crashing can be tolerated (masked) by its replicated counterparts. However, some synchronisation among the remaining replicas has to be performed to reach a consistent system state in the event of a crash. Then an illusion of continuous service can be presented as if nothing has happened. Availability and fault tolerance are very closely related issues.

NOTE: Availability requires service being available, whereas fault tolerance imposes failures being tolerable.

Enhanced performance. Replication is also a key to providing better performance. As now the service is running at multiple sites, clients do not have to line up at one site, instead, they can line up at different sites. If the replicated service is shared by a large community of clients, the response time can be dramatically improved.

7.0 REPLICATION SCHEMES

There are two major styles of replication schemes presented in the literature, namely the **primary-backup replication scheme** [Budhiraja et al 1993], and the **active replication scheme** [Guerraoui and Schiper 1997] [Schneider 1990]. The two schemes can also be combined in an integrated replication scheme that can accommodate the active replication scheme, as well as the primary-backup scheme in a unified form. The integrated scheme is based on the active replication, but is configurable to the primary-backup scheme. Some

terminology used hereafter needs to be clarified so that ambiguity can be avoided. A list of terms we like to differentiate in the distributed system context are: service/application, server, client, replica/member and replica/server group.

- Service/application. A service in a distributed system provides a set of well-defined operations exported to clients. The set of operations is often defined by an abstract service interface.
- Server. A server is a software entity running on an autonomous machine. The server implements the set of operations exported to clients. The implementation details of the service can be hidden from clients, and clients only see the abstract service interface that defines the set of operations.
- Client. A client is the user of a service. It typically invokes the operations provided by the server. Often, the client and the server are running on different machines, in turn their communications have to go through the underlying networks.
- Stateful versus stateless server [Birman 1996] [Zhou and Goscinski 1997]. If a server maintains some form of data on behalf of clients, we refer to it as a stateful server. A notorious example of this stateful server is the database server. On the contrary, if no client requests have any effect on the state of a server, whether the server does or does not maintain any data, this kind of server is referred to as a stateless server. A good example of the stateless server is the WWW (World Wide Web) server, where the server manages WWW pages, but WWW clients (browsers) cannot change the content of WWW pages (browsers can only retrieve the pages).
- Replica/member and replica/server group. A replica is a software entity representing the replicated server. The functionality of a replica is two-fold: providing the service and implementing the underlying replication control protocol. A collection of replicas forms a replica group, and replicas are assumed to be identical copies. We also call a replica in a replica group a member, and call a replica group a server group as well.
- Query versus update operation. Operations exported by a server can be categorised as either query or update operations. A query operation does not change the state of the data maintained by the server, but an update operation does. An operation invoked by a request contains the name of the operation and a list of actual parameters.

Now let us take a look into several replication schemes mentioned earlier in this section:

Case Study 1: The Primary-Backup Scheme

The primary-backup scheme has been researched extensively by many researchers [Borg et al 1989] [Budhiraja et al 1992, 1993] [Powell 1994] [Jalote 1994] [Mehra et al 1997]. The Figure below depicts the general architecture of this scheme. In essence, it is a simple scheme especially used in tolerating a process/processor failure by crashing. However, the primary backup scheme can become complicated when employing multiple backups. The complexity derives from keeping the consistency among backups are as stated:

(1) When the primary propagates its state to backups, the atomicity property, i.e. either all of backups receive a propagation or none of them receives it, should be guaranteed.

(2) When the primary crashes, the backups have to elect a new primary. The election algorithm has to run a consensus protocol to guarantee that only one candidate satisfies as the primary.

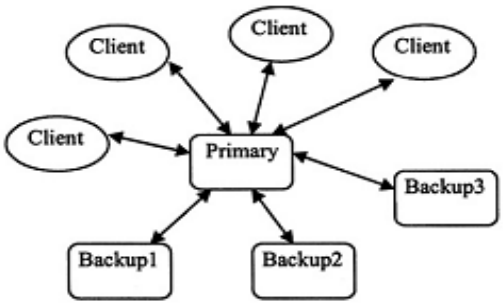

Figure 13: The Primary-Backup Replication Scheme

Replicas in a primary-backup scheme are distinguished as either the primary or a backup. At any time, there is only one replica acting as the primary. In other words, there is not a time when two replicas are the primary. Clients send requests to the primary only, backups do not receive any request directly from clients. The data consistency between the primary and backups is preserved by the primary propagating state changes (or updates directly) to backups. The scheme exhibits a fail-over time that is a time period between the primary crashing and the new primary taking over the process. During the fail-over time period, no replica is operating.

According to how often state changes are propagated to backups, the primary backup scheme is further divided into the hot (every state change is sent to the backup/backups right away), the warm (a collection of state changes is sent out at a time interval), and the cold (no state change is sent to backups while the primary is operating) strategies [Wellings 1996].

Obviously, different strategies affect the failover time. In general, the more frequently the propagation is performed, the shorter the fail-over time incurs.

Also the choice of propagation strategies can be affected by the environment within which a primary-backup system is running. For example, in an environment where all replicas can access the same file system, the primary can save the update requests to a log file to allow the backups access instead of sending requests directly to backups, thus the cold strategy is enough. In fact, the cold strategy is most suitable to a server providing only read-like service, in other words, no state changes. Cold strategy is considered to be the simplest propagation strategy of all.

The hot strategy is most applicable to a system that requires a real-time fail-over period so that when the primary crashes, the switch-over time (the fail-over time) is minimum. But the hot strategy can affect the response time depending on how the hot propagation is implemented. Figure 14 describes three different implementations. In the implementation (a), the response time for an update request is the worst compared with implementation (b) and (c). The implementation (b) gives the best response time as it executes the request and sends a reply to the client before the propagation. The performance of implementation (c) is in the middle.

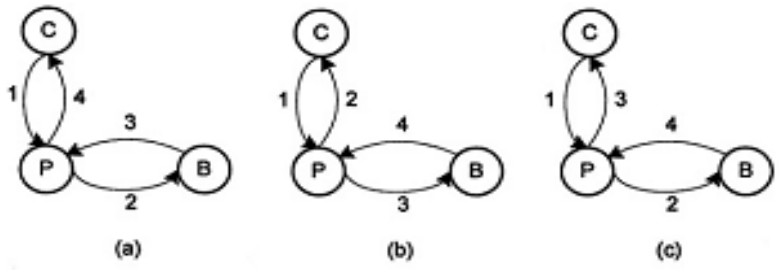

Figure 14: Hot replication implementations -- P represents the primary; B represents the backup; and C represents the client. In (a), 1, 2, 3 and 4 denote requests, propagations, acknowledgements and replies respectively; In (b), 1, 2, 3 and 4 denote requests, replies, propagations and acknowledgements respectively;In (c), 1, 2, 3 and 4 denote requests, propagations, replies and acknowledgements respectively.

The warm strategy follows the implementation of Figure 9.20 (b), but the propagation is not sent out per update. This strategy is often applicable to business-oriented servers, such as

database servers where fast responses have a higher priority. The propagation frequency is determined by the semantics of the business application domain.

In summary, choosing a right propagation strategy (hot/warm/cold) depends on the requirement of the fail-over time and the environment in which a replication system is running. The primary-backup scheme has been described as the passive replication in the literature [Budhiraja et al 1992, 1993] for the reason that backups sit back and passively react to state changes without being involved in any interaction with clients. The scheme is also labelled as easily implementable, less redundant processing, thus, less costly and more prevalent in practice.

However, the major drawback of the scheme is that the primary becomes the communication bottleneck due to the fact that all requests are sent to the primary. Other downsides of this scheme are: (1) In the event of the primary's failure, there exists a fail-over time during which there is no server available. (2) There can be request losses when the primary fails, e.g., the requests received by the primary but not yet being propagated are lost. The request loss problem is solvable, however, it needs to introduce some extra handling at the client side (see next chapter about the discussion of how a client switches to a new replica).

Case Study 2: The Active Replication Scheme

The active replication scheme is proposed to give rise to system performance by letting different replicas execute client requests concurrently. The performance can be improved enormously when most requests are queries. The scheme is based on an architecture in which each replica receives and processes requests. In contrast to the primary-backup scheme, this scheme is named active for the reason that all replicas are actively involved with clients, process requests and send replies. Figure 15 depicts this architecture.

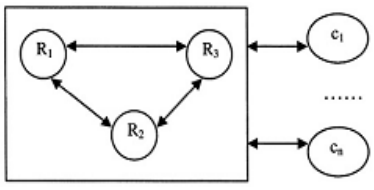

Figure 15: The active replication scheme -- represent an active replica group, whereas represent a set of clients. A client can send its requests to any replica or some replicas. Replicas that receive the request will send the reply back to the client.

The active scheme mainly has the advantage of masking server failures automatically over the primary-backup scheme. Active replication can be used to mask the server crashing automatically, provided that a request is sent to multiple replicas by a client. As long as one reply comes back, the client will not notice the failure of other replicas. The client often takes the earliest reply, and drops late ones. If an update request is not sent to full membership of a replica group, then the replicas receiving the update have to propagate the update to other members. Often this is done by competing, where multiple replicas propagate the request at the same time.

A common method proposed in [Powell 1994] to reduce unnecessary message transmissions is that, if a replica is about to send a reply to a client, it checks if there is any propagation of the request being received from other competitor replicas; if there is, it stops sending such a propagation; otherwise this replica sends the reply together with the original request to the client and all replicas, which will in turn stop their competitors sending their replies and propagations. Competing propagation and redundant replies can generate tremendous network traffic, and congest the network eventually, if the percentage of the update operations issued by clients is high. Another proposed approach is to let the client be responsible for sending an update to the full membership so that replicas are free from propagating requests. However, this can be a problem when the client fails during the procedure. The failure can leave the group in an inconsistent state whereby some replicas receive the request, some do not.

Another advantage of the active scheme over the primary-backup scheme is no failover time. As long as one replica is operating, the service remains available.

Case Study 3: Two Particular Replication Schemes

The preceding sections presented general structures for the primary-backup and the active replication schemes, their key characteristics, advantages and drawbacks. This section takes a look at two particular replication schemes, they are based on either the primary-backup or the active scheme. The coordinator-cohort scheme was developed by the ISIS project, and the leader-follower scheme was developed by the Esprit Delta-4 project [Powell 1994].

The Coordinator-Cohort Scheme

The coordinator-cohort scheme is used in ISIS as an example of testing group communication services to support building reliable distributed systems. This scheme is relatively close to the

active replication scheme. The basic idea is, for each request, a team of one coordinator and a subset of replicas (being cohorts) out of the whole replica group is formed to process the request. The coordinator is responsible for processing the request and sending the reply, whereas cohorts monitor the coordinator. One of the cohorts takes over when the coordinator fails. When sending the reply back to the client, the coordinator also sends the reply to the cohorts as well so that cohorts are informed of the completion of the request. If the cohots have not received the expected reply from the coordinator after a time-out, they conclude that the coordinator has failed and then take a corresponding action by selecting a new coordinator who takes over the process. ISIS uses its atomic multicast primitive for this purpose, and the destinations of the multicast include all cohorts and the caller.

This scheme seems to be able to perform requests concurrently between replicas, however, when a member is involved in a coordinator-cohort group, it cannot accept another request nor be in another coordinator-cohort group. The performance of the coordinator-cohort scheme is very questionable due to this reason. Also, since the coordinator-cohort group has to be formed up-front before executing each request, the response time can be considerably long.

The Leader-Follower Scheme

The leader-follower scheme developed in the project Esprit Delta-4 [Powell 1994] is rather closer to the primary-backup scheme than the active one. The general idea is that in a replica group, one replica is assigned as the leader and others as followers. All replicas receive requests and execute requests autonomously, but only the leader generates replies.

This scheme is designed with treating group non-determinism in mind. In the situation of a non-deterministic event arising, the leader makes a decision and informs followers about it so that all members can reach the same state even if they execute requests autonomously. Non deterministic factors considered are process preemption and time-related operations. The request ordering issue is included as a non-deterministic factor by the scheme.

- Request ordering. The request order is decided by the leader which sends notification messages to indicate to followers in what order requests are to be executed. This order is the same as the leader’s.
- Process preemption. Process preemption may cause inconsistency between replicas even when requests are ordered. To solve this problem, a set of preemption points has

to be defined within the server. When the leader process is preempted during an execution of a request, the leader rolls back to the last preemption point and instructs followers to roll back to the same preemption point. Solving process preemption is not an easy task. It is doubtful this method will work in general, as a preemption point may not be easily definable.

- Time-related operations. If a request involves an operation reading the local clock time in the calculation, then the leader needs to pass the reading as an extra parameter to followers so that the same request will result in the same state between the leader and the followers.

The leader-follower scheme does not solve the communication bottleneck problem, as requests are queued up at the leader for results. This scheme is basically a primary-backup scheme. The only saving is that no request propagations between the leader and followers, instead, notification messages are sent from the leader to followers. The saving is based on the assumption that notification messages are smaller than propagation messages. However, sending a request to all replicas is then shifted to the client side where the client is responsible for sending requests to the full membership of the replica group.

8.0 THE PRIMARY-PEER REPLICATION SCHEME

Although the primary-backup is relatively easy to implement, its bottleneck problem prevents it from being an advanced replication approach. The active scheme improves the performance but it brings great complexity which has to be restrained. For the two particular schemes, the coordinator-cohort has the problem of forming a group upon each request and may result in a long response time. The leader-follower scheme is very close to the primary-backup scheme that requests are lined up at the leader site, i.e., no concurrency at all. Therefore, to overcome these drawbacks, we propose a new scheme called primary-peer replication scheme (PPRS), based on the above schemes.

8.1 Description of the Scheme

A solution is to integrate the two schemes together [Wang and Zhou 1998a]. The scheme is called a primary-peer replication scheme (PPRS). It is based on the idea of an active architecture by restraining some design options in order to simplify communications among clients and replicas. Furthermore, it has the flexibility of being configurable to the primary backup scheme. This is done simply by letting all updates go to the primary of a group and

leaving other members (now backups) only receiving propagations. Figure 16 depicts the architecture of the PPRS.

Here we outline its design ideas and features:

- The PPRS allows each replica to take requests so that concurrent execution can be performed. This will give rise to a better system throughput as we have discussed.
- A client is connected with one replica at a time. The client sends all its requests to the connecting replica only. Upon the failure of the connecting replica, the client shifts to another replica. It can also switch to a different replica when the currently connecting replica becomes very slow. Clients are no longer involved in sending requests to multiple replicas. Thus, clients become lightweight software entities. Designing a lightweight client has become the trend of client/server systems nowadays [Linthicum 1997].

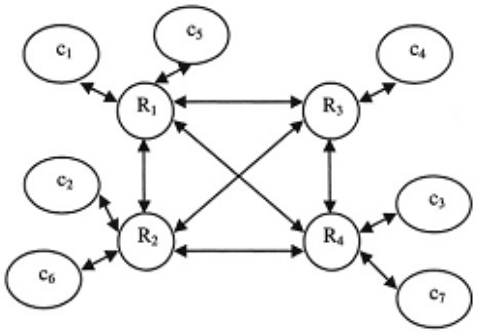

Figure 16: The Primary-Peer Replication Scheme — represent a 4- replica system, whereas represent seven clients connected to the replica system.

- The PPRS retains the idea of designating a replica as the primary. The primary is depicted in Figure 16 by the blank-filled circle. In addition to being a general member, the designated primary is responsible for making decisions on behalf of the group in certain situations, such as initiating relevant replication control protocols due to external events (group membership changes), coordinating the execution of the protocols, and informing other members about the decision. Also non-deterministic factors can be resolved if any.
- For each update request, only one replica will involve propagations. This removes the possibility of competitive propagations between multiple replicas, which happens in

the active scheme. Propagations now only happen between replicas without involving clients. This reduces the design complexity and simplifies the implementation.

However, the imperfection of the PPRS is that clients will observe the replica failure, and the fail-over time is not avoidable. The fail-over happens when a replica fails so that clients connected to the failed replica have to switch to other replicas and may need to re-send their last requests. But this fail-over time should not be very significant as requests sent to the new replica can be processed right away.

8.2 Replication Transparency

To be able to connect to a replica in the group, a client needs to know who are in the group, i.e., the references of all replicas. If replicas reside at permanent sites, clients can be hard wired to the references of the replicas. However, replicas may crash and leave voluntarily, a new replica may join the group to re-balance the load, or due to some administration reasons, a replica has to be moved to a new site, then the references that clients kept become stale.

There are generally two methods to solve this problem. First, every replica keeps a copy of the current membership. When a replica is added to the group, each replica updates its membership to reflect the change. Then the replica piggybacks the new membership to its connecting clients. By doing so, the clients can track the membership changes. However, the drawbacks of this design are:

(i) The replica has to keep records of all connecting clients so that the membership changes can be passed to them.

(ii) The replica group is not transparent to clients, i.e., clients see the internal structure of the group.

The second method uses a group naming service (GNS) to set up separately on a stable site to manage membership changes. A replica group is registered to the GNS under a unique group name, and any membership change is sent to the GNS by the primary member of the replica group. Clients only need to contact with the GNS to get the reference to a replica.

This approach provides a set of methods to create a group, add, delete or remove replica members from a group. When a group of replicas is created, the primary of the group registers to the GNS (invoking createGroup( )). The primary is also responsible for updating the GNS with any membership changes (invoking addMember( ) or deleteMember( )). When

a replica crashes, the primary invokes deleteMember(group, backup) to remove the crashed replica. When a new replica joins the group, the primary invokes addMember(group, backup) to add the new replica. Whereas bind(group_name) is invoked by clients to get a reference to an operating replica. For an active scheme, all replicas form a ring. Upon receiving a binding request, the GNS will return the next operating replica in the ring to the client so that each replica is connected with a roughly even number of clients, thus, loads are split over replicas. For a primary-backup group, the GNS always returns the reference to the primary.

9.0 REPLICATION CONSISTENCY

Distributed replication provides high availability, fault-tolerance and enhanced performance. But these features come at a price: replication adds great complexity to the system development. Most important of all, replication jeopardises data consistency. In turn mechanisms have to be employed to enforce data consistency. Maintaining data consistency is very expensive, a common practice is then to relax the data consistency level as low as possible to give rise to better system performance.

Data replication in the transactional model has been researched extensively. Data replication mechanisms developed for the transactional model are very strict since one-copy serialisability [Attiya and Welch 1994] is often required in order to maintain the ACID (Atomicity, Consistency, Isolation and Durability) property. Basically, a write operation (i.e., an update) has to be performed on most replicas synchronously before the result is returned to the client. Therefore, a long response time may incur and a low system throughput rate is achieved.

Not all replication systems require such a strong transactional semantics [Zhou and Goscinski 1999]. Update ordering is an alternative data consistency model which has weaker semantics than that of the one-copy serialisability. The basic idea of the update-ordering model is to let replicas execute the same set of update requests in a sensible order. This order meets the requirements of both the clients and the data semantics of a replicated service application. Compared to the data replication in the transactional context, the update ordering model generally gives a better response time and a high system throughput rate because it allows updates to be executed concurrently at different replicas. Update ordering is adopted as the

general data consistency model for maintaining the data consistency in a replication system built upon the PPRS structure.

A PPRS replica group can be configured in two major styles: either a primary backup group or a primary-peer group. For a primary-backup group, ordering is not an issue of concern, as we have discussed before that requests are ordered at the primary intrinsically. Therefore, the update-ordering data consistency model applies to the primary-peer group only.

The update ordering data consistency model requires placing ordering constraints on update operations so that updates arriving at replicas are ordered. Generally, if solely from the replica group point of view, as long as updates are executed at all replicas in the same order, the data consistency is guaranteed among replicas. However, from the client point of view, it may require updates sent from the same client to be executed in the sending order at all replicas, or updates having a happened-before [Lamport 1978] relation to be executed at all replicas by keeping that happened-before relation.

Formally, update ordering is categorized in terms of FIFO, causal, total, and total+causal to reflect data consistency requirements from both clients and the replica group. Ordering constraints have different levels of strength. FIFO is the weakest one and total+casual is the strictest. The system performance, especially the system throughput, is largely affected by the strength level of the ordering constraint being placed on a replica group.

Adopting a strict data consistency model is very expensive, and it may not be needed for a service application. Depending on the data semantics of the application domain, the strength level of the data consistency model should be relaxed as much as possible to give rise to the system efficiency.

10.0 CONCLUSION

A computer system, or a distributed system consists of many hardware/software components that are likely to fail eventually. In this chapter we introduced the basic concepts and techniques that relate to fault-tolerant computing. First, we presented the concepts of the reliability of a distributed system and techniques for achieving it. We classify the basic techniques used to build reliable distributed network systems: redundancy, fault avoidance techniques, fault detection techniques, and fault tolerance techniques. Also we described

several system models to build the reliability functions properly so that we can get the reliabilities of several typical systems. Then we discussed the distributed mutual exclusion, which frequently arises in distributed systems whenever concurrent access to shared resources by several sites/processes is involved. Mutual exclusion is a fundamental issue in the design of distributed systems and an efficient and robust technique for the viable design of distributed systems.

The client/server distributed computing model cannot be regarded as a reliable model, as the server is the single processing point. In the face of a site failure or a communication link failure, the server becomes inaccessible. Distributed replication is then a technique to solve this problem. Distributed replication systems make three major contributions: increasing the availability, achieving fault-tolerance, and improving performance. There are two major styles of replication schemes presented in the literature, namely the primary-backup replication scheme, and the active replication scheme. In the chapter we addressed these two schemes as well as two other schemes: the coordinator-cohort scheme developed by the ISIS project and the leader-follower scheme developed by the Esprit Delta-4 project. Though these schemes are extremely useful, they all have drawbacks. To overcome these, we propose a new scheme called primary-peer replication scheme (PPRS), based on integration of the above schemes.

REFERENCES

[Attiya and Welch 1994] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. ACM Transaction on Computer Systems, 12(2): pp. 91--122, May 1994.

[Birman 1996] Kenneth P. Birman, Building Secure and Reliable Network Applications. Manning Publications Co., 1996.

[Budhiraja et al 1992] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Optimal primary-backup protocols. In Proc. of the Sixth International Workshop on Distributed Algorithms, pp. 362-378, Haifa, Israel, 1992.

[Budhiraja et al. 1993] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach, In Distributed Systems, Sape Mullender, editor, Addison-Wesley Publishing Company, second edition, 1993.

[Cristian 1991] Flaviu Cristian, Understanding Fault Tolerant Distributed Systems, Communications of the ACM, Vol. 34, No. 2, pp. 56-78, Feburary 1991.

[Guerraoui and Schiper 1997] Rachid Guerraoui and Andre Schiper. Software-based replication for fault tolerance. IEEE Computer, pp. 68-74, April 1997.

[Johnson 1989] B. W. Johnson, Design and Analysis of Fault-Tolerant Digital Systems, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Jussi Kangasharju 2008] Distributed Systems: What is a distributed system?

[Lamport 1978] Leslie Lamport, Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM, Vol. 21, No. 7, pp. 58-65, 1978.

[Lamport et al. 1982] Leslie Lamport, Robert Shostak, and Marshall Pease, The Byzantine generals problem, ACM Transactions on Programming Languages and Systems, 4(3): pp. 382-401, July 1982.

[Linthicum 1997] David Linthicum. David Linthicum's Guide to Client/Server and Intranet Development. John Wiley & Sons, Inc, 1997.

[Mehra et al 1997] Ashish Mehra, Jennifer Rexford and Farnam Jahanian. Design and evaluation of a window-consistent replication service. IEEE Transactions on Computers, pp. 986-996, September 1997.

[Mullender 1993] Sape Mullender. Distributed Systems. Addison-Wesley Publishing Company, second edition, 1993.

[Powell 1994] David Powell, Distributed Fault Tolerance: Lessons from Delta-4, Vol. 14, No. 1, pp. 36-47, February 1994.

[Schlichting and Schneider 1983] R. D. Schlichting and F. B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Transactions on Computer Systems, 1(3): pp. 222--38, 1983.

[Schneider 1990] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys, 22(4): pp. 299-319, December 1990.

[Tanenbaum 1996] A. S. Tanenbaum, Computer Networks, 3rd ed., Prentice-Hall, 1996.

[Wellings 1996] A. J. Wellings and A. Burns. Programming replicated systems in Ada 95. The Computer Journal, 39(5): pp. 361-373, 1996.

[Wang and Zhou 1998a] Li Wang and Wanlei Zhou. An object-oriented design pattern for distributed replication systems. In Proc. of the 10th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'98), pp. 89-94, Las Vegas, USA, October 1998.

[Zhou and Goscinski 1997] W. Zhou and A. Goscinski, Fault-Tolerant Servers for RHODOS System, The Journal of Systems and Software, Elsevier Science Publishing Co., Inc., New York, USA, 37(3), pp. 201-214, June, 1997.

Security Engineering: A Guide to Building Dependable Distributed Systems

An Introduction to Distributed Systems. http://webdam.inria.fr/Jorge/html/wdmch15.html