JES
Journal of Engineering Sciences

Vol 10, Issue 6, JUNE /2019
ISSN NO:0377-9254

DESIGN FOR FAULT TOLERANCE: AN APPROACH TO OBTAIN HIGH DEPENDABILITY

Dr Y.Rajasree Rao, Vimmigari Karthik,S.Mallikarjuna

LORDS Institute of Engineering and Technology

Email : rajasreerao@lords.ac.in , karthik@lords.ac.in , settimallimalli@gmail.com

ABSTRACT

Fault tolerance is the ability of a system to continue performing its intended function in spite of faults. In abroad sense, fault tolerance is associated with reliability, with successful operation, and with the absence of breakdowns. A fault-tolerant system should be able to handle faults in individual hardware or software components, power failures or other kinds of unexpected disasters and still meet its specification. In the secure-critical fields like avionics, nuclear engineering, aircraft and transportation, it highly requires reliability to reduce the occurrence of failures. Within the area of system reliability, the fault tolerance is an effective and dependable method to enable the system (mostly embedded system) to still work in the correct way.

Keywords:security,failures,redundancy,reliability

1.INTRODUCTION

There are countless ways in which a system can fail. To make it a fault tolerant, we need to identify potential failures, which a system might encounter, and design counteractions. Each failure's frequency and impact on the system need to be estimated to decide which one a system should tolerate. Here are just a few examples of potential issues to think of:

- program experiences an unrecoverable error and crash (unhandled exceptions, expired certificates, memory leaks)
- component becomes unavailable (power outage, loss of connectivity)
- data corruption or loss (hardware failure, malicious attack)
- security (a component is compromised)
- performance (an increased latency, traffic, demand)

1. FAILURES CATEGORIES

Most of the typical failures can be divided into two categories:

2.1. Fail-stop behaviors

Fail-stop failures are relatively easy to deal with. Here are a couple of basic strategies:

1. checkpoint strategy
    A. Conditions:
        i. hardware is still working
        ii. a consistent copy of a system's state is available (backup)

Vol 10, Issue 6, JUNE /2019
ISSN NO:0377-9254

B. Action: restart a server and load the last good version of the system
C. Downsides:
    i. a component is unavailable for a time of reboot
    ii. there is a chance that reboot won't help
    iii. a component may end up in an endless crash loop

2. Replicate state and failover

A. Condition: there are valid state's backups and redundant hardware available
B. Action: start a second instance of the system and switch the traffic to it
C. Downside: requires a spare hardware, which would possibly linger away for most of the time

2.2 Byzantine failures

Byzantine failures are situations, where a component starts to work in an incorrect, but the seemingly valid way (e.g. data gets corrupted) – possibly due to faulted hardware (a flipped bit) or malicious attack. Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. Here are a couple of basic solutions:

1. turn byzantine into fail-stop behaviour
    - Action: use checksums, assertions or timeouts. If verification fails, the system should automatically stop and recover – hopefully in a better state
    - Downsides: same as in the Fail-stop strategies
    - Example: internet routers drop corrupted packets
2. Fix corrupted data in runtime
    - Action: use error detection and correction algorithms
    - Downside: performance impact, because the system must use its computing power to verify data at every processing step
    - Example: CRC, ARQ, ECC

Each solution to byzantine failures has its disadvantages, but they seem to outweigh the alternative, which is having corrupted data in a system. Unfortunately, there may be no solution to byzantine failure where all data is stored and processed by a single process.

3.DISTRIBUTED SYSTEMS

Another way to handle failures is to design a distributed system, but with it, things get more complicated. A distributed system is the one where a state and processing are shared by multiple computers – unlike a centralized system, where everything is stored in a single piece of hardware – that appears to a user as a single coherent system. Distributed systems can be found everywhere. Here are just a few examples:

- Domain Name System (DNS)
- Content Delivery Network (CDN)
- Phone networks (landline and cellular)
- Traffic control networks (lights, train, airplanes)
- Crypto currencies (Bit coin, Ethereum)

Usually, distributed systems are designed to achieve some non-functional requirements like:

- reliability – elimination of a single point of failure by using redundant nodes which take over workload in the case a node presents a fail-stop behavior
- performance – latency reduction by placing nodes closer to clients
- scaling – ability to tune a system's computing capacity according to the current demand by juggling with the amount of the available hardware in a system

While distributed systems may help to tolerate some of the typical failures of centralized systems, they increase complexity of a solution and come with their own set of problems such as network partition state consistency

4.FAULT TOLERANCE APPROACHES

Fault tolerance approaches can be classified into two types: Proactive and Reactive. Proactive approaches predict errors, faults and failures and replace the suspected components where as reactive approaches reduce the effect of faults by taking necessary actions. Some fault treatment policies can also be used to prevent faults from being reactivated.

5. REDUNDANCY BASED FAULT TOLERANCE

Redundancy is having more than one functionally ready components of a system other than a component that actually provides the service. At two levels we can implement the redundancy: Process level and data or object level. This approach uses replication technique to create redundancy. Replication is the process of creating and maintaining multiple copies of data objects or processes.

6. COMMUNICATION WITH GROUPS OF REPLI-CAS

Providing fault tolerance using replication requires a means to communicate with groups of replicas. A common approach is the use of a group communication system (GCS), to ensure
consistency between and among replicas. DRE systems provide several challenges for using a GCS. DRE systems can contain large numbers of elements with varying fault tolerance and real-time requirements. These requirements range from not needing FT or RT to having very strict requirements.

7. THE APPROACH

In the real embedded system, the fault happens in the component will affect other components or its parent component. Our approach can check the state of the target component with any error event within the subcomponent. Therefore, we firstly study how the component behaves with an error event inside it. In the error model, the error event triggers transition to a state. In our approach, we firstly assume a single error event with a specific type happened inside one of the subcomponents.

Component Error Behavior.get Events.

The error event is a type of error behavior event and it can generate events of different types. We enumerate every type of every error event in every turn to check the state of the control component. After choosing an error event, we iterate all the transitions whose source state is marked with the "Statekind: Working" to check whether the condition comes true with the error event.

These states which are reached with the error event will be marked with "readed" tag and then added to a stack. After the states are added to the stack, we will execute the transition whose condition is empty and source state is the state taken from the stack. Meanwhile, the state machine for the out propagations will be checked because the state trigger by the error event can trigger the out propagation to another component.
Besides, the component where the error event occurs, we take the components which receive the out propagation into consideration. As the propagation between two components are determined by the AADL architecture, we use the connection through the ports to find out the component we want to check. What we do is to iterate all the connection and then match the type of the ports and the type of propagations specified the ports. For these components which receive the out propagation, we need to execute the transitions with the incoming propagation. The process is similar with the process of the transition with the error event because the incoming propagation can be the condition in the transition too. Also some out propagations may be propagated to other components. We use the breadth-first algorithm to visit the components with the help of a stack. To avoid the endless loop, the visited incoming propagations will be marked. We do not mark the component with tag since one component can have more than one incoming propagations.

8. CONCLUSION

The fault tolerance technology is a rich, complex and growing Engineering discipline. Advances in microelectronics are removing the priced and technological barriers to design for fault tolerance and fault tolerant systems have now become a practical reality. Fault tolerance is crucial in computing systems used in defense, medical equipments and transaction processing, and is desirable in other applications. Designing a computer system to meet desired dependability specifications proves more complex. The designer faces the challenge of selecting redundancy techniques best suited to the application and integrating diverse schemes into the system. The system must, of course, also meet performance specifications. Time to mask, or detect or correct failure is an important factor in real time applications.

REFERENCES

[1] M. R. Lyu, ed., *Software Fault Tolerance* Chichester, England: John Wiley and Sons, Inc., 1995.
[2] P. Murray, R. Fleming, P. Harry, and P. Vickers, "Somersault Software Fault-Tolerance," HP Labs whitepaper, Palo Alto, California, 1998. .

[3] J. Gray and D. P. Siewiorek, "High-Availability Computer Systems," IEEE Computer, 24(9):39-48, September 1991.

[4] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1 (January 1986), pp. 96-109.

[5] A. Avizeinis, "The N-Version Approach to Fault-Tolerant Software", *IEEE Transactions of Software Engineering*, Vol. SE-11, No. 12 (December 1985), pp. 1491-1501.

[6]I. Lee and R. K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", IEEE 1993, pp. 20-29.

Vol 10, Issue 6, JUNE /2019
ISSN NO:0377-9254

[7] M. Cukier, J. Ren, C. Sabnis, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, and R.E. Schantz. AQuA: An Adaptive Architecture that provides Dependable Distributed Objects. In Proc. of the IEEE Symposium on Reliable and Distributed Systems (SRDS), pages 245–253, West Lafayette, IN, October 1998.

[8] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale : A QoS-enabled Component Deployment and Configuration Engine in Proceedings of the 3rd Working Conference on Component Deployment,Grenoble, France, November 2005.