International Journal of Research
Available at https://edupediapublications.org/journals

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 03
March 2017

A Study on Important Considerations for Designing Fault Tolerant Systems

Gouthami Shiramshetty[1] *Jose Mary Golamari*[2] *Srinivas Rao Pulluri*[3]

gouthami.shiramshetty@gmail.com, golamarijosemary@gmail.com srithanrao@gmail.com

Department of Computer Science and Engineering

Jayamukhi Institute of Technological Sciences, Narsampet, Warangal (TS)

***Abstract*— All the embedded and software systems should be reliable and should be operational even when the system is performing any tasks even in extreme conditions. So, in order to make a system continues operational, we intend the systems to be Fault Tolerant.Fault Tolerance is the ability to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification. In order to adequately understand the Fault tolerance it is important to understand the nature of the problem that software is supposed to solve. Software faults are all design faults. The source of the problem being solely design faults is very different than almost any other system in which fault tolerance is a desired property. The software faults are the result of human error. In this, paper will discuss about the Architectural design of Fault Tolerant system, Redundancy, Application requirements which mainly intends to Software design, Synchronization Interface, Fault Detection logic and the modes of operation.Current software fault tolerance methods are based on traditional hardware fault tolerance. Firstly, we shall discuss the basic terminology which clearly explains about the different terms used in Fault Tolerance. And go on to discuss various fault tolerance design considerations**

I. INTRODUCTION

Distributed Real-time Embedded (DRE) systems are a evolving group of systems that combine the strict real-time characteristics of embedded platforms with the dynamic, unpredictable characteristics of distributed platforms. As these DRE systems increasingly become part of critical domains, such as defense, aerospace, telecommunications, and healthcare, fault tolerance (FT) becomes a critical requirement that must coexist with their real-time performance requirements. DRE systems have several characteristics affecting their fault tolerance: DRE systems typically consist of many independently developed elements, with different fault tolerance requirements. This means that any fault tolerance approach must support mixed-mode fault tolerance (i.e., the coexistence of different strategies) and the coexistence of fault tolerance infrastructure (e.g., group communication) and non-fault tolerance infrastructure (e.g., TCP/IP). DRE systems' stringent real-time requirements mean that any fault tolerance strategy must meet real-time requirements with respect to recovery and availability of elements and the overhead imposed by any specific fault tolerance strategy on real-time elements must be weighed as part of the selection of a fault tolerance strategy for those elements. DRE applications are increasingly component-oriented, so that fault tolerance solutions must support component infrastructure and their patterns of interaction. DRE applications are frequently long-lived and deployed in highly dynamic environments. Fault tolerance solutions should be evolvable at runtime to handle new elements. This paper makes two major contributions. First, it describes the particular characteristics and challenges of component-oriented DRE systems and describes three advances we have made in the state of the art in fault tolerance for DRE systems: 1) A new approach to communicating with replicas that supports the coexistence of non-replicated and replicated elements for DRE systems with varying FT requirements, with no extra elements and no extra overhead on nonreplicated elements that only communicate with other nonreplicated elements. 2) An approach to self-configuration of replica communication, which enables replicas, non-replicas, and groups to discover one another automatically as the number of, and fault tolerance requirements of, elements change dynamically. 3) An approach to duplicate management that supports replicated clients and replicated servers, necessary to support the complicated calling patterns of DRE applications. A second contribution of this paper is that we demonstrate these advances in the context of an integrated fault tolerance capability for a real-world DRE system with strict real-time and fault tolerance requirements, a multi-layered resource manager (MLRM) used in shipboard computing systems. The fault tolerance we developed for this context utilizes off-the-shelf fault tolerance and component middleware with the above enhancements; and supports a mixture of fault tolerance strategies and large numbers of inter-operating elements, with varying degrees of fault tolerance. We then evaluate the performance of the replicated MLRM to meet its real-time and fault tolerance requirements and present analysis of the performance overhead of our fault tolerance approach.
All the embedded and software systems should be reliable and should be operational even when the system is performing any tasks even in extreme conditions. So, in order to make a system continues operational, we intend the systems to be Fault Tolerant.

Fault Tolerance is the ability to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification.

In order to adequately understand the Fault tolerance it is important to understand the nature of the problem that software is supposed to solve. Software faults are all design faults. The source of the problem being solely design faults is very different than almost any other system in which fault tolerance is a desired property. The software faults are the result of human error. In this, paper will discuss about the Architectural design of Fault Tolerant system, Redundancy, Application requirements which mainly intends to Software design, Synchronization Interface, Fault Detection logic and the modes of operation
Current software fault tolerance methods are based on traditional hardware fault tolerance. Firstly, we shall discuss the basic terminology which clearly explains about the different terms used in Fault Tolerance. And go on to discuss various fault tolerance design considerations.

Fault tolerance is an important design consideration for distributed real-time and embedded systems, which combine the real-time characteristics of embedded platforms with the dynamic characteristics. Traditional Fault tolerance methods do not address features that are common in distributed real-tme and embedded systems. Most of the existing research in Fault tolerance aimed at client-server object systems, whereas distributed real-time and embedded systems are increasingly based on component-based architectures, which support peer-to-peer interactions. This paper describes various design considerations to develop Fault tolerance technology for distributed real-time and embedded systems.

II. MAJOR COMPONENTS

Fault Definition

It is essential to define what/which system faults are severe enough for the redundant/backup system to take over. We can handle some kinds of application faults by just restarting the application/task on the same system in case of non-critical system which doesn't need to investigate the failure. Like a video streaming application failure causing it to restart the process.

Fault Detection

The Fault Detection logic is the main heart for the fault tolerance to succeed. It could be hardware fault, data fault, logic fault or storage error, etc. The system should have sound logic to detect which errors are severe and which are minor to detect the faults. Heart beat miss is the most common of the fault detection.

Recovery Logic

The recovery mechanism would/may need extra hardware support based on the configuration of the system. IO recovery may need extra connections be made or remade. On fault detection to recover the task or sub-task is assigned to some other process / component or hardware to complete the operation.

III. FAULTS

It can be termed as "defect" at the lowest level of abstraction. It can lead to erroneous system state. Faults may be classified as transient, intermittent or permanent. They are of following types:

A. Processor Faults (Node Faults):

Processor faults occur when the processor behaves in an unexpected manner. It may be of classified into three kinds:

1> Fail- Stop:
Here a processor can both be active and participate in distribute protocols or is totally failed and will never respond. In this case the neighboring processors can detect the failed processor.

2> Slowdown:
Here a processor might run in degraded fashion or might totally fail.

3> Byzantine:
Here a processor can fail, run in degraded fashion for some time or executed at normal speed but tries to fail the computation.

B. Network Faults (Link Faults):

Network Faults occur when (live and working) processors are prevented from communicating with each other. Link faults causes the following type of problems:

1> One way Links:
Here one processor can send messages to other is not able to receive messages. This kind of problem is similar to that faced due to processor slowdown.

2> Network Partition:
Here a portion of network is completely isolated with the other.

a. Failure:
Faults due to unintentional intrusions and hardware faults (RAM bit flip, etc).

b. Error:
Undesirable system state or data state that may lead to failure of the system or inconsistent results.

c. Recovery:
Recovery is a passive approach in which the state of the system is maintained and is used to roll back the execution to a predefined checkpoint.

d. Fault Tolerance:
Ability of system to behave in a well-defined manner upon occurrence of faults.

e. Redundancy
With respect to fault tolerance it is replication of hardware, software components or computation.

f. Security:
Robustness of the system characterized by secrecy, integrity, availability, reliability and safety during its operation.

IV. Design considerations

- How Real Time is your application?

A system is said to be real-time if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed.[4] Real-time systems, as well as their deadlines, are classified by the consequence of missing a deadline:
Hard – missing a deadline is a total system failure.
Firm – infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline.
Soft – the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service.
Thus, the goal of a hard real-time system is to ensure that all deadlines are met, but for soft real-time systems the goal becomes meeting a certain subset of deadlines in order to optimize some application-specific criteria. The particular criteria optimized depend on the application, but some typical examples include maximizing the number of deadlines met, minimizing the lateness of tasks and maximizing the number of high priority tasks meeting their deadlines.
Also non-real time application can also be running on the system.

- How distributed is your application?

Distributed applications (distributed apps) are applications or software that runs on multiple computers within a network at the same time and can be stored on servers or with cloud computing.
A system which does a single algorithm computation or IO operation like fetch and send would not be much benefitted if the application is sub-divided into smaller tasks because it doesn't optimize anything if we further divide the application task(s). Now fault tolerance for these kind of systems versus a system which caters to an application which is very distributed in nature would vary. For example simple distributed storage would need fault tolerant storage. If a storage operation fails we retry on another.

Let's take the application is very distributed in nature. It has to do N sub-tasks to complete a request/task. So now the sub-tasks could be running in a distributed computing environment and distributed data. Here if a sub-task fail the system has to detect the failure and schedule somebody else to do the same operation.

- How much data your application is processing and its read/write latency?

Let's take the case of an old conventional application which operates on a large data. So in case of a significant fault we may have to do a lot of data copy to the redundant/back up system which would cause the failover time to be high because of the data copy involved. It would always help if the application logic or database is designed such that whenever the data is operated upon the synchronization happens across. Also the distributed database latency would also play a major factor especially when the application/service has to process the data in real time and respond with results. In these cases it calls for use of faster data synchronization mechanisms across peers.
Read/write database latencies can be improved by using in-memory databases and faster memory access technologies.

- Is the application Connection Oriented?

In connection oriented application(s)/services(s) the failure of connected node/station can cause the service to pause/not available for quite some time. The detection of network failure or node failure in case of connection oriented network application needs to be sound. Mechanism to notify the task scheduler in-case of network prolonged/multiple connection disruption(s) would need to be defined properly. Also the heart beat mechanism between the task scheduler and task listener needs to be optimized for the faster fault detection and recovery.

The time taken for the other application to reconnect and provide the same connection oriented service should be as minimal as possible. Even in case of connection-less communication we would have to take care of the QoS and service not available scenario(s) detection.

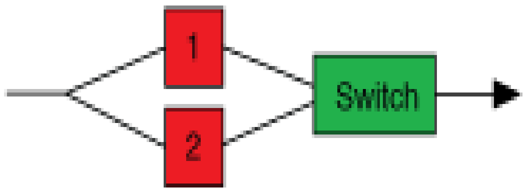
- What type of redundancy is desired?

Standby redundancy is also known as Backup Redundancy i.e., when you have an identical secondary unit (Shadow) to back up the primary unit. The secondary unit typically does not process/monitor the system, but it's just there as a spare. Standby unit is not usually kept in sync with the primary unit, so it must reconcile its input and output signals.

We need a third party called watchdog, which monitors the system to decide when a switchover condition is met and command the systemto switch control to the standby unit. The system cost increase for this type of redundancy is usually about 2X or less depending on your software development costs. In Standby redundancy there are two basic types, they are as follows: Cold Standby and Hot Standby.

- Cold Standby:

In cold standby, the secondary unit is powered off, thus preserving the reliability of the unit. The drawback of this design is that the downtime is greater than in hot standby, because you have to power up the standby unit and bring it online into a known state. This makes it more challenging to reconcile synchronization issues, but due to the length of the time it takes to bring the standby unit on line, you will usually suffer a big bump on switchover.



V. RUNNING MODE

A. Only 1 Active (single Module Up):
Here the Control Processor running as standalone one where it will be monitoring the complete processing i.e.., only one CP is active.

B. Simultaneous Active Modules (Both Modules Up)
Here the control Processor runs with one more extra module. Where the first module and the second Module performs the same operations. Here the both modules are said to be in simultaneously active state where both will be running the same operation at a time.
The advantage is the concept of Fault Tolerance (i.e. Redundant Modules) if suddenly the first active goes wrong, the second module now will behave as First Active (Main Module) which is helpful in running any critical operations without any interrupt.

VI. SYNCHRONIZATION INTERFACE

The following are some of the synchronization interfaces possible for the fault tolerant system to operate:

A. Over the Network
Ethernet using fibre or tradional RJ45 connection over internet or private TCP/IP network.

B. Serial Interface (HDLC)
Serial cable running HDLC protocol in many to many operations.

C. USB interlink
We can run customized USB based connection protocols.

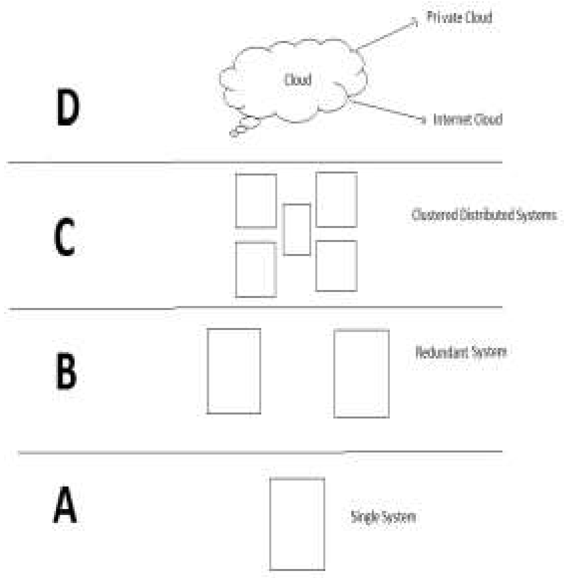
D. Hardware protocols
We can use many customized fault detection and link up detection and communication protocols. The automotive industry has quite a few hardware interlink / synchronization interface.

VII. LOCATION OF BACKUP/REDUNDANT SYSTEMS

If the redundant system or backup module is also in the same chassis or machine then the data copy would be faster and save network bandwidth as well.

If it outside the chassis or switch network then the fail over would consume lot of bandwidth and could affect other communication happening on the network if we are using the Ethernet for the synchronization as well.

VIII LATENCIES



Real Time: Low Latency service
Soft Real Time: Latency service is provided in A, B and in some part of C.
Non Real Time: Web, Mobile Applications etc.., it is provided in all A, B, C, D.

A. Hardware Dependency:
We can run fault tolerance in homogenous system like in the cloud similar virtual machine would behave as backups or redundant machines.
It would be difficult to run fault tolerance on heterogeneous systems and would need to employ hardware and operating system (platform) specific backup identification and fault detection.
Let's say our application requirement is it can be deployed on any hardware cloud VM, private VM or customized/standard embedded boards available on the web (like raspberry pi).
Then we would have to find a cluster/cloud computing platform which is portable, easily deployable on any hardware or platform.

B. Application level or Cloud/Cluster level fault tolerance:
Application level fault tolerance is needed for the application to detect the application faults on its own. This would be our final detection barrier for faults. User would have to program the application recovery mechanism in case of fault.
If we employ cloud/clustering based software level fault tolerance like OpenStack, Hadoop (for big data computing),

Apache Spark and Storm the framework itself would provide the fault tolerance detection logic and recovery plan for it.

CONCLUSIONS

This paper has described advances we have made in software support for fault tolerance for DRE systems. Our approach – very successful in this project – was to utilize off-the-shelf fault tolerance software where it was applicable for our needs, customize it where necessary, and develop new reusable capabilities where none existed. The three techniques that we presented in this paper – the Replica Communicator, self-configuration for replica communication, and client- and server-side duplicate management – extend existing fault tolerance techniques to make them suitable for componentoriented DRE applications. Yet, they are complementary to, and interoperable with, other existing fault tolerance services. To illustrate this, we have instantiated them and applied them to a real-world DRE example application. Our experiments show that these solutions provide suitable real-time performance in both failure recovery and fault-free cases

References

[1] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Center for Networking and Distributed Systems, Johns Hopkins University, 1998.

[2] R. Baldoni, C. Marchetti, and A. Virgillito. Design of an Interoperable FT-CORBA Compliant Infrastructure. In Proc. of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01), 5 2001.

[3] N. Budhiraja, K. Marzullo, F. Scneider, and S. Toueg. The Primary-Backup Approach, chapter 8. ACM Press, Frontier Series. (S.J. Mullender Ed.), 1993.

[4] 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.

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

[6] P. Felber. Lightweight Fault Tolerance in CORBA. In Proc. of the International Conference on Distributed Objects and Applications, pages 239–250, Rome, Italy, September 2001.

[7] C. D. Gill, D. L. Levine, and D. C. Schmidt. Towards Real-time Adaptive QoS Management in Middleware for Embedded Computing Systems. In Proc. of the 4th Annual Workshop on High Performance Embedded Computing, Lexington, MA, September 2000. MIT Lincoln Laboratory.

[8] B. Kemme, M. Patino-Martinez, R. Jimenez-Peris, and J. Salas. Exactly-once interaction in a multi-tier architecture. In Proc. of the VLDB Workshop on Design, Implementation, and Deployment of Database Replication, August 2005.

[9] O. Marin, M. Bertier, and P. Sens. Darx - a framework for the fault-tolerant support of agent software. In Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), pages 406–417, November 2003.

[10] P. Narasimhan. Transparent Fault Tolerance for CORBA. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.

[11] P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava. MEAD: Support for Real-time Fault-Tolerant CORBA. Concurrency and Computation: Practice and Experience, 2005.

[12] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Gateways for Accessing Fault Tolerance Domains. In Middleware 2000, LNCS 1795, pages 88–103, New York, NY, April 2000.

[13] H. P. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck. Fault-Tolerant Replication Based on Fragmented Objects. In Proc. of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - DAIS 2006, pages 256–271, June 2006.

[14] P. Rubel, J. Loyall, R. Schantz, and M. Gillen. Fault Tolerance in a Multi-layered DRE System: a Case Study. Journal of Computers (JCP), 1(6):43–52, 2006.

[15] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, 22(4):299–319, 1990.

[16] A. Vaysburd and S. Yajnik. Exactly-once End-to-end Semantics in CORBA invocations across heterogeneous faulttolerant ORBs. In IEEE Symposium on Reliable Distributed Systems, pages 296–297, Lausanne, Switzerland, October 1999. 8