LAUTECH Journal of Engineering and Technology 7(1) 2011: 52-56

TRIPLE MODULAR REDUNDANCY APPROACH FOR INTERNET CONNECTIVITY

[1]O.O. Adeosun, [2]E.R. Adagunodo and [2]E.A. Olajubu

*[1]Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, Ogbomoso, Nigeria*

*[2]Department of Computer Science and Engineering,
Obafemi Awolowo University, Ile-Ife, Nigeria*

ABSTRACT
This paper discusses the issue of providing tolerance to hardware and software faults in Internet system through triplicate application servers. A replication scheme (TMR) is presented, and a detailed dependability analysis of this scheme is performed. The proposed model was designed mainly for fault-tolerant Internet connectivity system where faults will not impair the continuous services rendered by the Internet system, thereby exhibiting highly varying and dynamic system characteristics. A major feature of the model under consideration is to attempt the adaptive connections of the existing Triple Modular Redundancy (TMR) scheme for the execution of redundant modules for a required level of fault tolerance.

Keywords: Modularization, application servers, replication scheme, dependability analysis, reliability, parallel processing, real-time processing, stochastic modeling,

1. Introduction

Internet system is developed to satisfy a set of requirements that meet a need. It should be able to deploy and coordinate network resources in order to plan, operate, administer, analyze, evaluate, design, and expand communication networks to meet demand at all times, and at a reasonable cost and optimum capacity. Better control assures a high level of quality of service, which corresponds to high productivity that is a function of investment turn-around. A requirement that is important in Internet system is that it should be highly dependable. Fault tolerance is a means of achieving that dependability. *Fault-tolerant computing* is the art and science of building computing systems that continue to operate favourably to satisfy users even in the presence of faults. Fault-tolerance is achieved by applying a set of analysis and design techniques to create systems with dramatically improved availability leading to very high *dependability*. Fault tolerance systems research covers a wide spectrum of applications ranging across embedded *real-time* systems, commercial transaction systems, transportation systems, military/space systems, health management systems, communications systems and so on. The supporting research includes system architecture, design techniques, coding theory, testing, validation, proof of correctness, modeling, software reliability, operating systems, parallel processing, and real-time processing. These areas involve diverse expertise knowledge ranging from formal logic, *stochastic* modeling mathematics, graph theory, hardware design and software engineering.

Replication is one of the oldest and most important in distributed systems. Whether one replicates data, computation or component, the objective is to have some group of processes/components that handle incoming events.

Triple Modular Redundancy (TMR) is generally used to increase the reliability of real time systems where three similar modules are used in parallel and the final output is arrived at using voting methods. The adoption of *TMR for Internet connectivity* usually requires the combined utilization of a wide range of techniques, including fault tolerance techniques intended to cope with the effects of faults and avert the occurrence of failures or at least to warn a user that errors have been introduced into the state of the system. To implement failover, it requires replicating service on TMR, storing distributed checkpoint and synchronizing replicas.

2. Review of Related works

Some of existing software fault tolerance approaches was extended to the treatment of both hardware and software faults (hybrid faults). Two typical schemes are taken into account – recovery blocks (Randell, 1975) and Self-Configuring Optimistic Programming (SCOP), an *adaptive* scheme (Bondavalli et al., 1993). *N*-version programming (Avizienis and Chen, 1977) is a representative of *non-adaptive* schemes for the sake of comparison. These architectural solutions specially directed to Internet system was analyzed with respect to dependability, availability, accessibility and restartability.

Laprie et al (1987) presented a set of *hybrid-fault-tolerant* architectures and analyzed and evaluated three of them. Their architectures are based on a fixed set of hardware components and not related to the dynamicity of hardware resources available as well as the efficiency issues. Such architectural solutions cannot well match the characteristics of dependable Internet system craved for in this research in which the resources must be competed by many unrelated but concurrent service requests on the Internet. In such varying environments e.g., Internet the architectures with the fixed requirement to hardware components are either inefficient or infeasible.

2.1 Client-Server distributed computing systems

Modern client-server distributed computing systems may be seen as implementations of N-tier architecture. In a typical four tier architecture the first tier (client tier) consists of client applications containing browsers, with the remaining three tiers deployed within an enterprise representing the server side; the second tier (Web tier) consists of a Web server that receives client requests typically via HTTP and passes on the requests to specific applications residing in the third tier (business tier) that is capable of hosting distributed applications; the fourth tier (enterprise information systems tier) contains databases and legacy applications of the enterprise. The platform providing the Web tier plus business tier is usually called an application server. Scalability can be achieved by replication of the different tiers on a cluster of machines (also called clusterization).

2.2 Redundancy system basics

The generic engineering solution to the problem of flaky components is redundancy: using multiple unreliable components in a coordinated, mutually verifying way can increase the reliability of the complete system by orders of magnitude. For example, if two identical, redundant components are each down 0.1% of the time, their failure modes are completely independent and detectable, and the rest of the system (including the arbitrator which determines which component to trust) can be approximated as 100% perfect, then the whole system should be down only 0.0001% of the time. As the example demonstrates, it is never possible to reach 100% reliability, but it is often possible to come arbitrarily close to the limit. This is the focus in this research work, to provide Internet services to clients without any interruption even at the presence of faults, thereby making the Internet system transparent and very well available to the clients.

3. Proposed Methodology for building Fault-Tolerant Internet Connectivity

Guerraoui and Schiper (1996) opine that group communication enables encapsulating a set of entities that cooperate to achieve some common service. A group has a logical address, which allows clients ignore the existence of its members. In figure 1, a set of replicated application servers composes the group. All replicas must provide access to the same methods and have to maintain the same state. To achieve this, there should be strong consistency. This will enable read-one-write-all replicas.

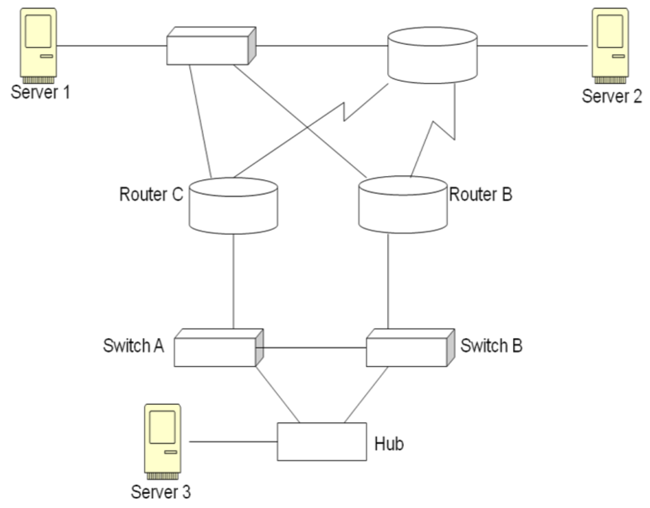

Figure 1: Proposed Replicated Servers Model

3.1 From primary to backup replication

In figure 1, one of the replicated servers (the primary) executes a transaction locally. Many classical approaches to replication are based on a *primary/backup* model where one device or process has unilateral control over one or more other processes or devices. For example, the primary might perform some computation, streaming a log of updates to a backup (standby) process, which can then take over if the primary fails, that is, it forwards updates to all other group member (backups) using the *total order multicast primitive* (TOCAST) (Guerraoui and Schiper, 1996). This primitive ensures that updates are delivered in the same order by all correct processes that work according to their specification. The termination property of the TOCAST assures the distributed system progress despite of failures, as well as its non-blocking characteristics. Typically, the primary waits for all backup answers and returns response to the client.

In order to avoid bottleneck, any replica can be enabled to play the primary role. Backup failure is transparent to the requester, but faulty primaries require achieving failover. In this study, a client detects a faulty-primary using timeout and its stub automatically re-routes a faulty request to an alternative application server.

The weakness of primary/backup schemes is that in settings where all modules could have been active, only one is actually performing operations. It is true we are gaining fault-tolerance but spending thrice money as much to get this property. An outgrowth of this work was the emergence of schemes in which a group of replicated modular components could cooperate, with each component backup the others, and each having the same status with the others.

3.2 Communication model

We use an *asynchronous* communication in our model; even overload application server can be assumed as fault-suspected because there is no way to distinguish between overload and faulty application servers. Also, we use an underneath *group-communication layer* to provide the needed multicast primitive and also application server service. The application server service manages the replicated application servers in figure 1 and detects fault-suspected application servers removing them from the group. The group communication layer operates in the presence of message omission faults, processor crashes and recoveries as well network partitions and merges.

4. Design Approach

As the number of nodes in a distributed computation increases, so does the probability for failure. A system is a collection of functionalities that must perform specific tasks; then the design of a survivable system can be thought of as a *multistage* process. It should be noted that, in a malicious environment, each stage has its limitations.

In traditional fault-tolerance, tolerating faults is typically achieved utilizing the principle of redundancy.

(i) Information Redundancy – usually considers the inclusion of additional information as the basis for fault recovery. A typical example is an error correction code.

(ii) Time redundancy – relies on multiple executions skewed in time on the same node and is often used to mask omissions.

(iii) Spatial redundancy – uses multiple components, each computing a value, and the final value is derived from a convergence function (e.g., majority voting). The resulting *N*-modular redundant (NMR) system implements a k-of-*N* system, which implies that the system functions as long as k or more components are fault free. A typical configuration is a triple-redundant redundancy (TMR), which is a 2-of-3 system.

5. Enabling Recovery Failures and Providing Failover Service to Users

Achieving the proposed Internet fault-tolerant service using modular redundancy requires treating client-primary as well primary-backups interaction. The model handles client-primary interaction switching of the client requests to alternative application server, when the current service is interrupted. The work also handles primary-backup interaction implementing distributed checkpoints. Recover from a failed application server is easier. It just requires re-routing clients' requests.

5.1 Distributed checkpoint implementation

A *distributed checkpoint* contains all local *snapshots* placed in all the replicated application servers. Each snapshot holds information about the last executed method, the client who requested this method and the application server who executed this method. This follows a distributed checkpoint approach, which multicasts a snapshot from a primary to all other application servers. Whenever the primary receives a transactional request (using point-to-point communication) from a client, it updates its own state and multicasts synchronization messages to the backups using the TOCAST primitive. The primary verifies if the distributed checkpoint was successfully established (waiting for all backup confirmation messages) and answers the client.

Backups process the synchronization messages and automatically store updates in their own states to establish the distributed checkpoint and to reflect a single distributed global state. If an application server fails, clients are guaranteed access to the same data through the backups. When an application server connection is closed, all application servers remove the information about the distributed checkpoint for that client. Storing this information will enable automatic failover during a transaction execution. The non-finished methods will be executed in another application server used to replace the failed application server.

5.2 Propagating updates to backups

There are two possible strategies to propagate updates: deferred update and immediate update (Wiesmann et al., 2000). In deferred update, transactions are processed locally at one application server and are forward to the backups at the commit time while the immediate update synchronizes every transaction across all application servers.

6. Implementation Issues

Two *OpenSource* projects were identified: *Java-Groups* (Ban, 1999) and *JOnAS* (*Java Open Application Server*) (Danes et al., 2000). Our replicated server is been developed to match the two OpenSource. In our model (figure 1), we changed some classes of the JOnAS to include the TOCAST primitive in the application server-side. Replicated application servers join the group and use this primitive to setting the distributed checkpoint. We implement the distributed checkpoint selecting, at compiling time, updates to be forwarded during the service execution. An update is assumed to be a method without result (it returns a null value). In the client-side, we modify the client's stub to automatically re-route faulty requests.

7. Result and Discussion

According to McCarthy (2003), a TMR architecture will have its reliability to be:

$$R_{System} = [R^3 + 3R^2(1-R)]R_v \quad \text{...............} (1)$$

where

R is the reliability of individual application server working correctly

(1-*R*) is the reliability that an application server is not working

R_v is the reliability of the coordinating voting device

Since our model (figure 1) follows suit, it means our reliability model is (1).

Proof:

If r (survival probability) is the reliability of an individual replicated application server then, the reliability of the *k*-out-of-*N* structure (figure 1) under the assumption that failures are independent events is given by the expression:

$$R(k-out-of-N)=\sum_{i=k}^{N}\binom{N}{i}r^i(1-r)^{N-i} \qquad \text{..........} \ (2)$$

where:

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

k is the number of application servers in use.
N is the total number of application servers available for use

This reliability expression is simply the summation of all the successful events; i.e. the system (2) survives provided *k*, *k*+1, *k*+2, …, *N*-1, or *N* application servers survive. The probability of exactly *i* application servers (modules) surviving is r^i. The probability of exactly $N-i$ application servers having failed is $(1-r)^{N-i}$, and the number of ways in which this event can occur is *N*-combinatorial-*i*. The summation of all these events from *i* = *k* to *N* yields the general expression (2). This general expression (2) has a number of special cases, which represent many of the commonly used protectively redundant structures.

In this case, where *3* application servers are used, the system (figure 1) can tolerate the failure of up to $\left\lceil \frac{N}{2} \right\rceil$ application servers. Therefore, the fault-tolerance of the proposed system is equal to $\left\lceil \frac{3}{2} \right\rceil$ thereby leading to high availability of Internet system, which is improving availability of Internet services to users.

8. Conclusion

Transactional systems could benefit from high availability Internet system to achieve fault tolerance and high dependability. The Internet system is more available for service delivery and provides good performance cum high Internet stability.

Also, we expect that server modularization improves the application servers' response time, when compared with non-replicated application servers, by allowing requests to be handled by several modules rather than one besides eliminating a single point-of-failure. In addition, deployment and redeployment of new and recovered application servers are necessary to maintain the Internet availability and dependability.

References

Avizienis, A. and Chen, L. (1977): "On the Implementation of N-Version Programming for Software Fault Tolerance During program execution," in COMPSAC 77, pp.149-155.

Agarwal, T.**;** Pathak, A. and Mohan, A. (2011): "A Novel Hybrid Voter Using Genetic Algorithm and Performance History," International Journal of Artificial Intelligence And Expert Systems (IJAE), Volume (2), Issue (3), pp117-121.

Ban, B. (1999): JavaGroups user's Guide, Department of Computer Science, Cornell University, 73p. http://JavaGroups sourceforge.net/

Bondavalli, A.; Di Giandomenico, F. and Xu, J. (1993): 'A Cost-Effective and Flexible Scheme for Software Fault Tolerance,' Journal of Computer Systems Science and Engineering, CRL Publishing Ltd., Vol. 8, No. 4, pp.234-244.

Danes, A.; Dechamboux, P.; Riveill, M. and Vandome, G. (2000): "Technologie a base de composants EJB experience et perspectives avec JOnAS. OCM 2000, Nantes, Mai 2000, pp 11-13. http://www.objectweb.org/jonas/

Guerraoui, R. And Schiper, A. (1996): "Fault_Tolerance by Replication in Distributed Systems." Department d'Informatique Ecole Polytechnique Federale de Lausanne, 1996.

Laprie, J.C.; Arlat, J.; Beounes, C.; Kanoun, K. and Hourtolle, C. (1987): "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," IEEE Computer, Vol. 23, No. 7, pp.39-51.

McCarthy, M. (2003): "Fault-Tolerant", Tech Target, Volume 3, Number 1, pp13-21.

Randell, B. (1975): "System Structure for Software Fault Tolerance," IEEE TSE, Vol. SE-1, No. 2, pp.220-232.

Wiesmann, M.; Pedone, F.; Schiper, A.; Kemme, B. and Alonso, G. (2000): Understanding replication in databases and distributed systems. Proceedings of ICDCS 2000, pp.264-274, Taipei, Taiwan, R.O.C., April 2000.