ISSN 2047-3338

Analysis of Crash Recovery Failure Detection with Quality of Services

B. Sushma[1], B. V. Rama Krishna[2] and Muni Sekhar Velpruru[3]
[1,3]Department of Information Technology, Vardhaman College of Engineering, Hyderabad, India
[2]St. Mary's College of Engineering & Technology, India

***Abstract*– We develop a probabilistic model of the behavior of a crash-recovery target, i.e. one which has the ability to recover from the crash state. We show that the fail-free and the crash-stop are special cases of the crash-recovery run with mean time to failure (MTTF) approaching to infinity and mean time to recovery (MTTR) approaching to infinity, respectively. We compare the previous work QoS metrics to allow the measurement of the recovery speed, and the definition of the completeness property of a failure detector. Then, the impact of the dependability of the crash-recovery target on the QoS bounds for such a crash-recovery failure detector is analyzed using general dependability metrics, such as MTTF and MTTR, based on an approximate probabilistic model of the two-process failure detection system. Then according to our approximate model, we show how to estimate the failure detector's parameters to achieve a required QoS, based NFD-S algorithm analytically, and how to execute the configuration procedure of this crash-recovery failure detector. Our analysis indicates that variation in the MTTF and MTTR of the monitored process can have a significant impact on the QoS of our failure detector.**

***Index Terms*– Crash recovery, failures, Metrics and QoS**

I. INTRODUCTION

THE purpose of failure detection is to discover abnormal software behaviors. Recognizing the occurrence of failures is one of the most important steps towards achieving fault-tolerance and dependability, challenging problem in this research is to tolerate the Byzantine failure also called as arbitrary failure means process may behave in an arbitrary manner and produces responses at an arbitrary time [1]. It is the most difficult failure to detect adopting consensus algorithms. To achieve K fault tolerance, 3K+1 service replications are needed [2]. In the worst case, the K faulty services may send incorrect values, or incorrectly represent the values of others, but the remaining 2K+1 services can still return the same. Crash failure detection is the building blocks to achieve a successful consensus. However, detecting crash failures is a difficult problem. In [3], Fischer et al. show the impossibility of separating a crashed process and a very slow one, in a pure asynchronous system, known as the Fischer-Lynch-Paterson's impossibility result. Subsequently, failure detector oracles, which give possibly erroneous information about the state of the monitored target, have been proposed.

Another approach to consider the crash-recovery is proposed by Guerraoui and Rodrigues [4]. A process can keep crashing and recovering infinitely often and it is eventually always up and running. In theory, a process recovery can be achieved by adopting stable storage and the state information of the process can be stored and retrieved from the storage. After a crash is detected, the recovery procedure can be used to retrieve the latest stored process information. In practice, in order to provide high availability, self-repairing and self-healing mechanisms are widely adopted in fault-tolerant systems to achieve automatic recovery after a crash occurs. Particularly, in middleware systems, many techniques and algorithms have been introduced to achieve the self-repairing or self-healing goal, e.g., [5], [6].

In such systems, it is assumed that the system undergoes periodic crashes. During a crash period, the system is unable to service any requests or send any messages, externally behaving as if the system is unreachable. The end of the crash period is marked by a recovery, after which the system returns to normal service and its internal state is restored to the state before the crash failure occurred.

Crash-recovery failure needs to be considered as a frequently occurring failure type to be detected. However, the crash-recovery case has studied due to the fact that there are more possible discrepancies between the failure detector and the monitored target, increasing the size of the state space of the monitoring process, making the quality of service analysis for such a paradigm more complicated. In this we analyzed the QoS of a crash-recovery failure detector based on a simple time-out algorithm. A crash-recovery target was modeled as an alternating renewal process that the crash-recovery behavior of the monitored target will impact the QoS of such a failure detector, which implied that the crash-recovery paradigm merited further studied. We outlined how to model the failure detection pair in a crash-recovery run and how to configure the failure detector to satisfy a given QoS requirement.

This paper represents a substantial expansion of support the results with analytical results, derived directly from the equations in this paper, are also plotted and compared and able to present a detailed analysis for each of the QoS metrics, which shows the validity of our model.

II. TYPES OF CRASH-RECOVERY FAILURES

A failure occurs when an actual running system deviates from its specified behavior.

A. Muteness Failure

Muteness failures are malicious failures in which a process stops sending messages but might continue to send other messages. When muteness failure occurs the service will stop executing its designed features but might still be able to generate liveness messages such failures cannot be detected by crash failure detectors. Muteness failure is a particular case of omission failure which fails to send [7] only some message but no all detecting process could be an application-specific.

Adopting the muteness failure detecting algorithm in which proposes a protocol that forces the monitored service to send "Iam-not-mute" message to the muteness failure detector periodically when service is not mute but stop sending such messages when a muteness failure occurs.

B. Timing Failure

Timing failure occurs when a service response lies outside the specified time interval. Example if the service-hosting machine or network is overloaded or some other resources on which the service depends are overloaded then the service response might be delayed and a timing failure [7] might occur. Detection of timing failure should be based on the specified deadline or time constraints. In order to detect a timing failure recording the time when the conversation between a service pair starts can be adopted. If the service instance cannot return the answer before the specified deadline is regarded as a timing failure. Moreover there are more sophisticated timing failure detectors such as the one reported in which uses group communication to detect timing failure in a quasi-synchronous system or the timely computing base model can [9] deal with timeliness requirements without synchronized clocks.

C. Omission Failure

When a service fails to send a response or receive a message an omission failure occurs behaves as a communication failure will cause message transmission fail.

The simplest way to detect omission failures is to enable the service to provide failure information. If the service can throw a fail to send or fail to receive message exception or send this information to the failure detector then the failure is regarded as an omission failure.

D. QoS Failure

A service even if it provides a correct result might still fail to meet the consumers desired level of service fails to satisfy a specified property by the service consumer by a certain level of QoS constraints. Example 95% [8] confidence that the mean time to get results is smaller than 10 seconds assuming that initially 99% confidence of this property. QoS failure can be tracked by matching the given QoS specification with the QoS delivered by the service.

E. Response Failure

Response failure occurs when a service response is incorrect. In general, response failures can be separated into two types. The first type is value failure: the response value is wrong; the second type is state transition failure: the service deviates from the [8] correct flow of control [9]. To detect value failure, voting algorithms can be adopted if multiple service replications are deployed. To detect state transition failure, the service design specification should be available to check whether a service has deviated from its expected state or not.

F. Partial Failure

For a composed application, a component failure may result in a partial failure of the composed service. Identifying such a partial service failure still remains challenging. Here we regard a component of a service as atomic and consider dependencies among these components. Failure of a component might potentially cause other failures of the composed service or the failure of the composing procedure. For a composed service, due to service internal fault-tolerance policies, partial failure might not be visible externally by a failure detector, which only observes the composed service. In order to discover such partial failures, sensors must be implemented at the atomic component level to track the status information of each atomic component of a composed service. The implementation of the sensor for a component should be based on the failure mode that the sensor is concerned with.

G. Composition Failure

Service composition is an important characteristic of web services. Any service partial failure or unmatched composition requirements would result in a service composition failure. To detect such failures in a composing service, each composition step should be checked and tested. If the current composition procedure is verified without any mistake having occurred and the composition conditions are satisfied, then proceed to the next step, otherwise a composition failure might have occurred.

H. Byzantine Failure

The Byzantine failure is also sometimes called the arbitrary failure. It means a service may behave in an arbitrary manner, produce arbitrary responses at arbitrary time [9]. It is the most complicated failure to detect. According to the detection, Byzantine failures can be separated into undetectable and detectable failures [10]. Undetectable Byzantine failures refer to failures that are unobservable by other processes based on the messages they receive or failures that are undiagnosable. Detectable Byzantine failures have two categories:

Commission (Response) failures: the service does not behave correctly according to its semantics.

Omission failure: the service behaves correctly but fails to send or receive messages.

III. FAILURES, FAULT TOLERANCE AND DEPENDABILITY

The software and hardware may contain internal or external bugs, errors that can make the run-time services unstable. Computer system shows that bugs are one of the important reasons for system crashes; faults are accepted as inevitable and may lead to a system failure. To improve the critical systems survivability when failures occur used the fault-tolerance mechanisms. Fault tolerance is the ability or the property to enable a system to continuously operate correctly when some abnormal internal or external events occur.

Dependability is one of the most important issues for computer systems which is a complex attribute, the concept of dependability as the property of a computer system such that reliance can justifiably be placed on the service it delivers. In addition by recording the lifetime information of a system, the systems dependability can be described quantitatively. Dependability of a system can be measured according to the reliability, availability, consistency, usability and security. In order to simplify the measurements which are related to failure detection.

Reliability can be defined as the probability that the system will operate correctly in a specified operating environment up until time t (t>0).

Availability can be defined as the probability that the system will be operational at time t.

Consistency can be defined as the probability that the system will return to normal operation correctly after a failure has occurred within a specified operating environment within time t.

Generally reliability presents how long a system will operate correctly and can be captured by mean time to failure which records the probability of a service to persist without a failure. Availability presents the probability that a system is accessible or reachable with correct operation at any time and can be captured by mean time to failure divided by mean time between failures. Consistency presents the ability of a system to recover from a failure state to the correct operation state and can be captured by meantime to recovery (MTTR) which records how fast a system recovers. Furthermore from the system design perspective different systems might desire different aspects of dependability features such as the highly available system which requires the system to be accessible with correct operation most of the time or the highly consistent system which requires fast recovery of the system after failures occur.

IV. PROBLEM DEFINITION

Global distributed systems, various types of failure may occur during the execution. In our work we address crash failure detection. Furthermore, many researchers have drawn their attentions on the QoS of crash failure detectors' implementations and failure detection algorithms. However, most previous work on the QoS of crash failure detection is based on the crash-stop or the fail-free assumption and relies on predicting the liveness message transmission behavior and estimating a suitable timeout threshold to achieve a better QoS of crash failure detection. In contrast, we regard a crashed process or service as recoverable, since many fault-tolerance techniques can be adopted to achieve such recovery. For high-level applications, a more realistic crash failure model would be crash-recovery.

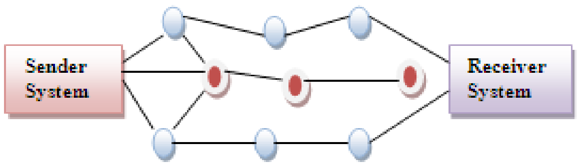


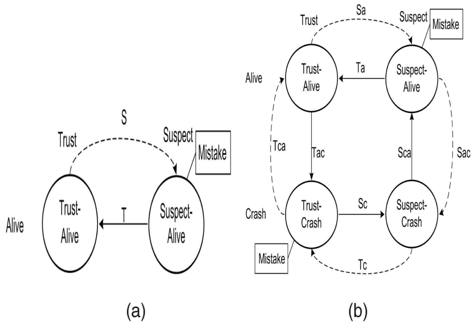
Fig. 1: Network topology with node failures

Fig.1 shows the problem definition where Red Color Node represents the network failure.

Our solution implements the Sender sends some data to receiver where some nodes exist between networks of sender to receiver, where different port numbers assigned to all these intermediate nodes. When program begins a file accepted by sender to transmit to receiver NFDS (Network Failure Detection Algorithm) monitors the transmission for failures. If any failure detected NFDS reports the specific node where data lost and recovers the data from previous node. Retransmission from recovered node done by NFDS this process continues until all data packets safely reaches the destination. For successful delivery to destination all the intermediate nodes maintain a copy of data to ensure data transmission reliability over network

A. System Model

We consider a distributed system model with two services: one FDS and CR-TS distributed over a wide-area network. The FDS and the CR-TS are connected by an unreliable communication channel. Liveness (heartbeat) messages are transmitted through the channel. The communication channel does not create or duplicate liveness messages, but the messages might be lost or delayed indefinitely during transmission.1 The CR-TS can fail by crashing but can be repaired and restart to run again after some repair time, i.e., it behaves as a crash-recovery model. The drift of the local clocks of the FDS and the CR-TS is small enough to be ignored and their local clocks are sufficiently [11] synchronized (this can be guaranteed by some time synchronization service such as the Network Time Protocol used to be regarded as a clock synchronized system. The failure detection algorithm we adopt is the NFD-S algorithm.



(a) (b)

Fig. 2: (a) Fail-free transition, (b) Crash-recovery transition

B. Quality of Service Metrics for Crash Recovery

In order to measure the speed with which a FDS can discover a recovery of the CR-TS, Recovery detection time (TDR): represents the time that elapses from CR-TS's recovery time to the time when the FDS discovers the recovery. If the recovery is not detected, then TDR = +1. Since in a crash-recovery run there is no eventual behavior of a CR-TS, a fast recovery could make a failure undetectable by a FDS. Under such circumstances, the completeness property of a failure detector proposed in [11] cannot be always satisfied. In order to reflect this situation, we refine the definition of completeness as follows:

- Strong completeness: every crash failure of a recoverable process will be detected.
- Weak completeness: a proportion of crash failures of recoverable process will be detected, satisfying a specified requirement.

To measure the completeness of a crash recovery failure detector, another new QoS metric:

Detected failure proportion (RDF): the ratio of the detected failures over the occurred failures (0 _ RDF _ 1). When no crash failure is detected, RDF = 0. When all crash failures occurrences are detected, RDF = 1. The strong completeness property of a FDS's requires that E(RDF) = 1. The weak completeness property requires E(RDF) _ RL DF, where RL DF is the required lower bound on the detected failure proportion and 0 < RL DF < 1.

V. ESTIMATION OF QUALITY OF SERVICE & NFD-S ALGORITHM IN A CRASH-RECOVERY RUN

In a crash-recovery run, the state of CR-TS can switch between Alive and Crash. There is a sequence of regeneration points for the CRTS, each of which is the recovery time of the CR-TS. In the following these are also regeneration points of the system consisting of the failure detection pair. In order to study the steady state behavior of CR-TS throughout its lifetime, only need to observe the time period between two consecutive recovery times of the CR-TS. Fig shows the relationship between a FDS [10] and CR-TS on the interval t_2 $[t_0, t_3)$, where both t_0 and t_3 are regeneration points. Obviously, the mean time between t0 and t_3 is the MTBF. We split $[t_0, t_3)$ into $[t_0; t_1)$, $[t_1;, t_2)$, $[t_2, t_3)$,

- t_1 is the time when the FDS detects the recovery of the CR-TS from the Crash state to the Alive state,
- t_2 is the time when the service crashes,
- Ss is the first liveness message sending time after a recovery,
- Sf is the sending time of the last liveness message before a crash,
- Si is the sending time of a liveness message between ss and sf,
- h is the liveness sending interval; ts is the first decision time after recovery5,
- t_b is the last decision time before crash,
- t_f is the freshness point according to sf,
- TDR is the time to detect a recovery.

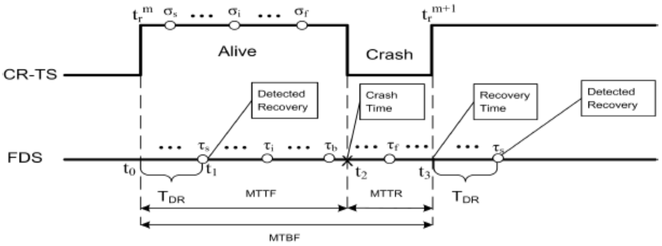


Fig. 3: Analysis of the Crash-Recovery NFD-S Algorithm

Let tr be a recovery time of the current MTBF period. The following definitions are based on the NFD-S algorithm.

For the fail-free duration $[t_1\ t_2)$ within each MTBF period 1. K : for any $i \geq 1$, let k be the smallest integer such that, for all $j \geq i+k$, m_i j is sent at or after time t_i.

For any $i \geq 1$, let p_i j(x) be the probability that the FDS does not receive just the (i+ j)th message m_{i+j} by time t_{i+x}, for every $j \geq 0$ and every $x \geq 0$; let pi $_0$ = pi $_{0(0).}$

For any $i \geq 2$, let q_0^i be the probability that the FDS receives message mi□1 before time $t_{i.}$

A. Comparative Study

Previous work focused on the QoS of crash failure detectors is based on the crash stop at that time or fail-free assumption. The fail-free assumption assumes that failures do not occur. The crash-stop assumption assumes that there is only one failure and the monitoring procedure terminates once that crash failure is detected. The algorithms based on these assumptions focus on how to estimate the probabilistic message arrival time and a suitable time-out period for a failure detector to ensure a required QoS.

We have drawbacks with previous work. In such systems, it is assumed that the system undergoes periodic crashes. During a crash period, the system is unable to service any requests or send any messages, externally and behaving as if the system is unreachable.

The end of the crash period is marked by a recovery, after which the system returns to normal service and its internal state is restored to the state before the crash failure occurred.

For such systems, crash-recovery failure needs to be considered as a frequently occurring failure type to be detected in the Proposed System by means of QOS.

Proposed system show how to remove the fail-free or crash-stop assumption and model the probabilistic behavior of a failure detector with respect to a crash-recovery target, taking into consideration general dependability metrics, such as mean time to failure (MTTF) and mean time to recovery (MTTR). We outline how the QoS of a failure detector is limited by [9] the dependability of the monitored target. Moreover, we establish that the crash-stop or fail-free models are special cases of the crash-recovery model.

In order to effectively assess the QoS of the failure detector in a crash-recovery run, we have defined new QoS metrics to measure the recovery detection speed and the proportion of the failures of the monitored target which are detected.

To make an accurate estimation of the failure detector's parameters needed to achieve a required QoS, a configuration procedure for a crash-recovery failure detector analyze how to achieve the QoS from a given set of requirements based on the NFD-S algorithm.

V. CONCLUSION

The crash-recovery target and its failure detector are analyzed as stochastic processes. We redefined previous work QoS metrics to be applicable to crash recovery failure detection and metrics to measure the recovery detection speed and the completeness property of a failure detector.

Monitored target's crash-recovery behavior on each QoS metric and showed that if a failure detector's parameters are to be accurately estimated, these dependability characteristics must be taken into account. Thus, we showed how to configure the failure detector to satisfy a given set of requirements based on the dependability characteristics in addition to the QoS of message transmission based on the NFD-S algorithm. Our analysis shows that the QoS analysis is a particular case of a crash-recovery run. Furthermore, if the recovery of the monitored target needs to be detected, future work extends with novel failure detection algorithms.

REFERENCE

[1] J. Laprie, A. Avizienis, and H. Kopetz, Dependability: Basic Concepts and Terminology. Springer-Verlag, 1992.

[2] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," ACM Trans. Programming Languages and Systems, vol. 4, no. 3, pp. 382-401, 1982.

[3] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," J. ACM, vol. 32, no. 2, pp. 374-382, Apr. 1985.

[4] R. Guerraoui and L. Rodrigues, Introduction to Reliable Distributed Programming. Springer, 2006.

[5] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, "Towards Architecture-Based Self-Healing Systems," Proc. First Workshop Self-Healing Systems (WOSS '02), pp. 21-26, 2002.

[6] M.E. Shin and D. Cooke, "Connector-Based Self-Healing Mechanism for Components of a Reliable System," Proc. 2005 Workshop Design and Evolution of Autonomic Application Software, pp. 1-7, 2005.

[7] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," IEEE Trans. Software Eng., vol. 13, no. 1, pp. 23-31, Jan. 1987.

[8] D. Manivannan and M. Singhal, "A Low-Overhead Recovery Technique Using Quasi Synchronous Check pointing," Proc. IEEE Int'l Conf. Distributed Computing Systems, pp. 100-107, 1996.

[9] J.C. Laprie, A. Avizienis, and H. Kopetz. Dependability: Basic Concepts and Terminology. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1992.

[10] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS), pages 61 – 75, 1997.

[11] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Distributed Systems. Technical Report TR93 - 1377, Department of Computer Science, Cornell University, 1993.

B. Sushma pursuing M. Tech Software Engineering at Varadhaman College of Engineering. Her areas of interest are Networking, Data Mining and Information security.

B. V. Rama Krishna Assoc. Prof. at St Mary's College of Engineering & Technology M.Tech from Punjab University currently he is pursuing Ph.D Data Mining From ANR university, Hyderabad. His areas of interest include Data Mining, Information Security.

Muni Sekhar Velpruru received the Bachelor of Technology degree with Information Technology from the Jawaharlal Nehru Technological University, Hyderabad, in 2007 and the Master of Technology degree in computer science and Engineering-Information Security from National Institute and Technology, Karnataka, Surathkal. He is currently working in Assistant Professor in the Department of Information Technology, Vardhaman College of Engineering, Hyderabad. In his two years of Research, he published near to six paper national and International Journals and Various conferences. His research interests include Network Security cryptography, Virtualization, Cloud Computing, Service Oriented Architecture and Web Designing.