

An efficient technique for fault monitoring in Grid and Distributed Systems

Shujaat Hussain, Muhammad Abdul Qadir
Center for Distributed & Semantic Computing,
Mohammad Ali Jinnah University, Islamabad, Pakistan
{shujaat,aqadir}@jinnah.edu.pk

Fault monitoring is an important issue to be addressed for fault tolerant distributed system. With the help of an efficient fault monitoring scheme, it would be easy to determine the crash and quickly take the recovery steps. Fault monitor typically detects faults by sending and receiving messages to remote objects. One of the major responsibilities of the monitor is to adapt timeouts according to the dynamic network and system conditions, and set timeouts very close to the real delays in the system. The timeouts must not fluctuate with large amplitudes around the actual time delays. It should not adapt with sudden transients behaviors. Otherwise the number of false alarms would increase, which may trigger a heavy fault recovery mechanisms. The relationship between timeouts and monitoring intervals need to be managed intelligently. Our technique adapts the timeout on the previous history which gives us a fair idea about the work load and we use it to our advantage. When we tested the existing schemes against the three points just mentioned, to our surprise, none of the scheme complies with these points. We experimented with our technique along with some other proposed techniques, our scheme; ACID gave very good results when compared with the schemes.

Keywords: Fault Tolerance, failure detection, fault monitoring, timeout, monitoring interval

1. Introduction

As the very nature of distributed systems is, the objects are geographically spread, so monitoring of the objects aliveness (crash faults) becomes inevitable. To observe and diagnose faults, distributed entities are monitored by sending and receiving messages. Typically, a monitor entity waits for some specific time interval (timeout) for the reply from the entity being monitored and if the reply is received within the interval (termed as the occurrence of **response**), the reply is interpreted for further action. However, if the reply is not received within the timeout (termed as the occurrence of **timeout**), the remote entity could be considered as suspected.

The algorithm used to monitor remote entities is termed as Failure Detector [1, 2, 6, and 7]. There main objective is to identify and suspect the failed

processes. The failure detector has the responsibility of identifying the current status of a process. As it is almost impossible to differentiate between a crashed process and a process which is communicating slow due to some reason like network and other load conditions, so there is always a chance that a correct process can be tagged as a suspected process. In this sense the failure detector can be said an unreliable failure detector as there may always be some correct process in its list that are broadcasted as suspected process.

Failure detectors may eventually realize its mistake and change the status of a correct process based on the algorithm or system which is being implemented for all this procedure. The two properties that a failure detector should ultimately achieve are the completeness and accuracy [13]. Completeness property implies that at the end of the day all the correct processes are not suspected by the other processes and all the crashed processes have their status suspected by the other processes. The accuracy property tells us how accurate is the failure detector has identified the status of the process.

As mentioned earlier entities are distributed, there could be many reasons for the delay in receiving the reply which could be network overloads / congestion or load at the object that is being monitored. The system has to be intelligent enough to accurately fix the timeouts for each monitoring message depending upon the dynamic conditions of the networks. The load on the hosting machine (machine upon which the remote entity is hosted) may vary at different times, too. This varying load conditions make it even more difficult to fix accurate timeouts. A timeout value greater than transmission time and load conditions can delay the detection mechanism and the value smaller than these varying parameters may trigger false fault occurrences. So accurate timeouts are very difficult to determine and at the same time it is very important, too [1]. We need to have monitoring algorithms which take into account the dynamic network and host-load conditions to adjust the timeouts and monitoring intervals. In other words the algorithm has to be **adaptive**.

Another very important parameter against which the fault monitoring algorithm should be judged is convergence to very close to the actual network delays and host response times. Transient situations should not change the timeouts and monitoring intervals. There could be a sudden transient change in the network or host conditions, and adapting the parameters on these transients may lead to false alarms as these conditions do not normally last longer. Therefore, adaptation should not be frequent enough to take in these kinds of temporary conditions. If the remote entity is suspected, what should the system do to avert the suspicion if it is a fake? Also, the algorithm should not take too much time to adapt. In other words, timeout should be the sum (or very close to it) of the entity processing time for *isAlive* or *iAmAlive* methods and the network delay at that time. The adjustment of timeouts should not oscillate with large peak to peak value with reference to the actual timeout. Order of the responses with last reading larger than the previous one (or smaller than the previous one) should not influence the timeout significantly. The monitoring system that accurately adjusts timeouts in minimum possible time with bypassing transients is termed as **convergent** in our paper.

In addition to timeouts, another important parameter need to be adjusted by the fault-monitoring algorithms is the monitoring interval. Monitoring interval is the time delay between two consecutive monitor messages for the situations, timeout or response. Relating the timeouts with monitoring interval (if time out increases, the monitoring interval will increase, and vice versa) could not be defended, too. If an entity is not suspected (response occurs), why should we reduce the monitoring interval along with the timeout. We can compare it with the analogy that if a person is healthy then why she should go to the doctor very often (the monitoring time interval could be increased). Similarly in case of response, should we increase the monitoring interval or decrease it. What should be the ratio of the increase or decrease? All these questions must be answered sensibly and intelligently by the monitoring algorithms. The system that fixes monitoring intervals the way as just described is termed as **intelligent** in our paper.

To summarize the three points bench-mark (we believe that this bench mark is also our contribution) which is the base point for the analysis of existing algorithm to set timeouts and monitoring intervals for fault monitoring in distributed systems, ofcourse the system has to be adaptive if we want it to converge,

the timeouts and monitoring intervals, to very close to the network and host dynamic conditions. However, every adaptive system may not converge in reasonable amount of time with bypassing transients. Similarly, the adjustment of monitoring intervals has to be intelligent enough that one can defend it with argument.

There is basically two well known fault monitoring strategies; the push model [1,2,10] and the pull model [1,2,10]. In case of the pull model, a fault monitor (objects that detect the fault occurrences) pings periodically by calling an *isAlive* method of the monitorable object (object to be monitored). If the monitor does not receive reply within the time interval (timeout) then the process is considered suspected and faulty. In case of the push model, the monitorable object calls periodically an *iAmAlive* method of itself informing to the monitoring objects that it is still alive. If the monitor object does not receive the message within some predefined time interval then monitorable object is suspected.

Both the models have their own pros and cons [1,2]; as the push (heart beat) strategy has only one way traffic for the messages and consumes the network resources less. The pull model gives us the freedom of monitoring the process on our own will.

We have considered the criteria described in the previous paragraphs (adaptive, convergent, and intelligent) to examine the existing algorithms as well as our proposed algorithm (termed as failure detectors) to setup timeouts and monitoring intervals to monitor aliveness of distributed objects. First let us consider the related work and some terminology used in defining these algorithms.

2. Related Work

One of the first was Chandra and Teoug [9] who proposed to increase the timeout after the wrong suspicion of the process. Chandra and Teoug classified the failure detectors in classes to differentiate their power according to accuracy and completeness. A failure detector has also been proposed in [10] to tolerate mobility and topology changes. Failure detectors have been considered as first class objects in [11] i.e. all of the responsibilities of the failure detector are hidden and the failure detection works on high abstraction. In [19] the failure detectors are said to be considered as the operating system service. In [12], a two way adaptive

failure detection is used, as there is a normalized value attached to every process which tells us about the status of a particular process. There have been several instances of adapting the failure detectors to network load [1, 2, 13, 14, 15].

In [14] a quality of service (QoS) for failure detectors has been proposed for the message delays and losses. A probability is assigned to the message delays and losses. In [15] it has also been emphasized on how hard it is to identify between a crashed process and a slow process. It has classified messages as application messages and control messages. A finite average response model has been proposed in [16] which is based on an assumption that there are stubborn channels with finite response time. Moreover some more assumptions are taken like the average response time will be finite and the channel will have a mechanism for some basic flow. Most importantly there is no upper or lower bound on the processes to respond.

The technique in [1] has claimed optimal adaptation in setting the timeout value. The claim is quite justified when it comes to consistent and smooth overload. Whenever the response comes very quickly or very slowly, then the timeout adapted severely fluctuates, thus triggering a false alarm. One of the problems with the technique is its dependency on the last response time, which is the most critical in this scheme, whereas it almost ignores the previous set of responses when adapting the timeout. Moreover there is a very close relationship between monitoring interval and the timeout. This phenomenon creates an unnecessary burden on the object that is to be monitored when the regular responses are coming from the object.

3. System Model

The distributed system consists of a collection of processes who can communicate by sending and receiving messages. The channels used for communication guarantees no message corruption and no malicious messages [18]. Moreover it guarantees message will eventually reach the desired destination. The only reason the process cannot receive or send the messages when the process is crashed. The messages are never altered or lost, which implies that the message will eventually arrive at the process it is meant for unless the process is crashed.

4. Working of Proposed Algorithm

The Flag value shows how many consecutive timeouts have occurred. It is set to zero initially. Current_elapsed_time stores the latest response time and Counter is set to 3 which shows after how many consecutive responses the timeout value should be changed. Current_ratio and Reverse_ratio are the ratios of the responses that arrive within the timeout. Last_elapsed_time is the previous response that came in time and was previously the Current_elapsed_time. Total_Ratio is the sum of reverse ratio Reverse_ratio and current_ratio.

An enhancement has been proposed in the technique in [1] which improves the timeout considerably. The averaging scheme is the heart of the algorithm for finding the timeout. The figure 2 algorithm executes when the response arrives. The status of the process is set to active as it has a given a response. The status is then broadcasted. The timeout_flag is set to zero as a response has occurred. The first time the algorithm executes the current_ratio is set to 1. The current ratio is calculated by the ratio of current elapsed time to last elapsed time. The reverse ratio is calculated by the ratio of last elapsed time to current elapsed time. The limit to which the timeout is adapting is three. When the counter reaches 3 the timeout variable is going to have a new value. The Ratio is calculated by the sum of current ratio and reverse ratio divided by 5 as the current ratio and reverse ratio are calculated twice and one for the first time. The average elapsed time is the average of the three consecutive responses. The new timeout is calculated by the product of average ratio and average elapsed time. The monitoring interval is increased by the mfactor which can be increased from 1.1 to 2. Increasing the factor more than this will compromise the intimacy of the monitorable object. When the count is equal to 2 than we can assume the network load is moderate due to two consecutive responses. Now we should not increase the timeout value aggressively if the timeout occurs, so we decrease the timeout flag

```

ACID_RESPONSE ()
{
  Status ← active;
  Current_elapsed_time ← timervalue
  Flag++;
  If( last_elapsed_time = 1)
    Current_ratio ← 1
  Else
    Current_ratio ← Current_elapsed_time /
      last_elapsed_time;

  If(counter > 0)

    Reverse_ratio ← last_elapsed_time /
      current_elapsed_time;

    Total_ratio ← Reverse_ratio + Total_ratio +
      Current_ratio;

    Total_elapsed_time ← Total_elapsed_time +
      Current_elapsed_time;

    Last_elapsed_time ← Current_elapsed_time;

    Counter ← Counter+1;

  If( count==2)
    If(flag>0)
      Flag ← flag-1;

  If(Counter==3)
    Ratio ← Total_ratio / 5;
    Average_elapsed_time ← Total_elapsed_time / counter;
    Timeout ← Average_elapsed_time x Ratio;
    Monitoring_interval ← Monitoring_interval x mfactor;
    Flag ← 0;
    Ratio ← 0
    Counter ← 0;

  Current_elapsed_time ← 0;
}

```

Figure 1: Response Algorithm

When the timeout event occurs, the algorithm in figure 3 executes. The status is set to suspected and broadcasted. The timeout flag is incremented. If the timeout for the first time after a string of responses, it is increased very softly by a factor of 1.4.

```

ACID_TIMEOUT ()
{
  Status ← suspected;
  Flag++;
  If(flag = 1)
    Timeout ← timeout x 1.4;
    Monitoring_interval ← Monitoring_interval / 1.4;

  If(flag = 2)
    Timeout ← timeout x 1.6;
    Monitoring_interval ← Monitoring_interval / 1.6;

  If(flag > 2)
    Timeout ← timeout x 1.8;
    Monitoring_interval ← Monitoring_interval / 1.8;

  If(flag > threshold)
    Monitoring_interval ← 0;
    Timeout ← 0

  Counter ← 0
}

```

Figure 2: Timeout Algorithm

If it occurs for the second consecutive time the timeout is increased by a factor of 1.6. If the timeout occurs consecutively more than twice, it is increased by a factor of 1.8. The monitoring interval is decreased by the same factor by which the timeout has decreased. Moreover when the timeout flag is incremented to a certain threshold which is used to indicate there are very strong chances that the object has crashed, then we reset the timeout and monitoring intervals as decreasing the monitoring interval will increase the burden on the monitor object. This phenomenon will only waste the resources and pollute the network unnecessarily. We have experimented with many values that we have used as factors for increasing factor. They were ranged from 1.05 to 2, but the factors that are mentioned in the timeout algorithm produced the optimum results.

The trigger in figure 3 starts monitoring and activates the timer to measure the response time of the object. If the response is lower than the timeout, then the response algorithm is executed else the timeout algorithm is executed. The algorithm is stopped till the value of monitoring interval. When the threshold is reached, than the monitoring is stopped as there is a strong chance that the process is crashed

```

ACID_TRIGGER()
{
    If (flag==threshold)
        stopMonitoring();
    startMonitoring();
    timer.Start();
    if (timer.Value < timeout)
        ACID_RESPONSE();
    Else
        ACID_TIMEOUT();

    Sleep(monitored_interval);
}

```

Figure 3: Trigger Algorithm

5. Implementation and Results

We implemented our algorithm and the algorithm proposed by Sotama ET all. The language was C++ and the data generated was applied to both the algorithms. The findings of this activity are discussed below.

The basic architecture is that a monitoring object inquires about the object that is being monitored. There are different techniques which are executed at the monitoring object which adjusts the timeout and monitoring interval values.

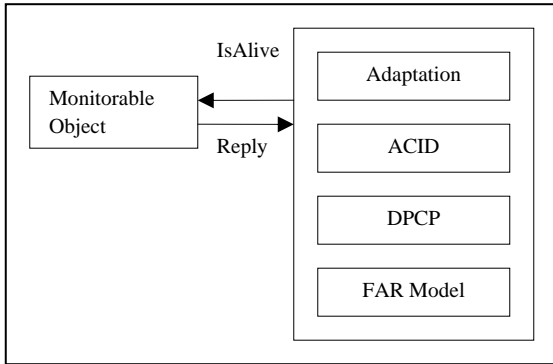


Figure 4: Basic Architecture

The different techniques [1, 2, 15] and ACID are executed and they all change the values of timeout and monitoring interval with respect to their nature.

5.1 Intelligence

As we can see that in proposed technique the monitoring interval remains stable unless a serious

load has occurred. The monitoring interval is then decreased by a factor of 1.1 so that we can monitor the object quickly and find its current status.

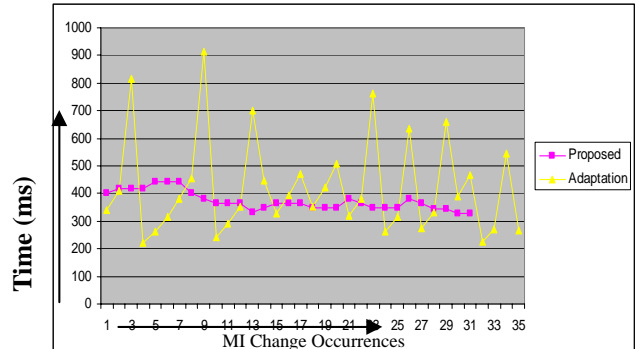


Figure 5: Intelligence in monitoring interval

It is not necessary to aggressive handle the monitoring interval when the response is not coming. We can still go with the same monitoring interval value but with a slightly increased timeout value. We increase the monitoring interval when we have three or more consecutive timeout events. This will increase the stability to the object being monitored as a little hiccup in responding should be ignored until it is persistent. The yellow strip [1] handles the monitoring interval with respect to the events that are coming i.e. RESPONSE and TIMEOUT which is not really the answer. The algorithm should be intelligent enough to increase or decrease the monitoring interval depending on the previous history. It should not briskly decrease the monitoring interval value because it will burden the object unnecessary.

5.2 Response Wastage and Overall Behavior

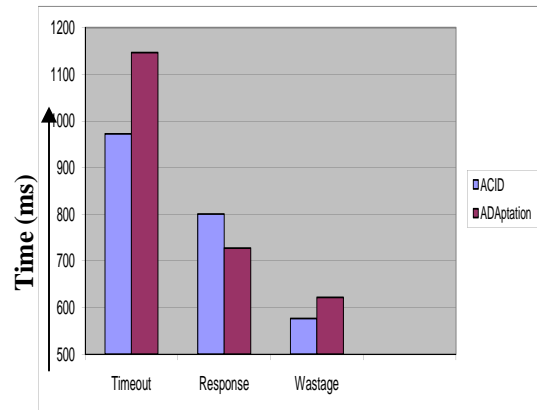


Figure 6: Response Wastage and behavior

We experimented with the responses in the range of 150 ms to 700 ms and applied the same data set to both the techniques i.e. adaptation and ACID. Now we ran five experiments with the same data range with 4000 responses each. The extra experiments were done to avoid any favorable data for either technique. The data sampled was purely random and between the ranges specified. We valued three important parameters i.e. the timeout events that have occurred during the process, the response events (three consecutive responses are considered for response event) and the responses that played no part in the adapting of timeout which we have termed as wastage. The more the timeouts occur, the weaker the technique is, mainly because it did not have the ability to cope those responses. These timeouts indicate the false timeout because the response came after these timeouts. Moving onto the second parameter i.e. the response event in which the ACID has more occurrences than the adaptation technique. The more the responses are managed or coped by the technique the better it is. The responses indicate that the timeout value is maintained optimally enough not to give false alarms.

There is altogether different scenario for the responses that are wasted. For example the timeout value is 450 ms and the next three responses are 400 ms, 320 ms and 500 ms. The first two responses will trigger the RESPONSE algorithm and wait for the third consecutive response which does not come. The third time TIMEOUT algorithm is executed and the two former responses do not play any part in adapting the timeout. These responses are termed as wastage. Now one of the main objectives of the technique should be maximizing the usage of responses and minimizing the wastage of responses. We can see in the figure that more responses are wasted in ADAPTATION technique due to its aggressive handling of the timeout i.e. a very low timeout. These three factors are the key factors for assessing the efficiency of a monitoring technique.

References

1. I.Sotama, E.R.M. Mandeira., "ADAPTATION-Algorithms to ADAPTive FAulT Monitoring and their implementation on CORBA". In: Proceedings IEEE of the 3rd International Symposium on Distributed Objects and Applications (DOA '01). Rome, Italy, Sep 2001.
2. I.Sotama, E.R.M. Mandeira, "DPCP (Discard Past Consider Present)-A Novel Approach to Adaptive Fault Detection in Distributed Systems". In: Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS' 2001). Bologna, Italy, Oct 2001.
3. Andrew S. Tanenbaum and Maarten van Steen, "Distributed Systems :Principles and Paradigms."
4. George Coulouris, Jean Dollimore, Tim Kindberg. "Distributed Systems Concepts and Design, Third Edition"
5. Object Management Group, http://www.omg.org/news/meetings/workshops/presentations/embedded-rt2002/03-1_Garon-Narasimhan_FTTutorial-OMGWkshp-2002.pdf
6. Object Management Group.: The Common Object Request Broker: Architecture and Specification, 2.6 Edition, OMG Technical Committee Document formal/02-01-02, Jan 2002
7. Object Management Group, Fault Tolerant CORBA Specification, OMG Document orbos/99-12-08 edition, December 1999.
8. T. D. Chandra , S. Toueg, "Unreliable failure detectors for reliable distributed systems. Journal of the ACM," 43(2):225–267, 1996.
9. Nigamanth Sridhar.: "Decentralized Local Failure Detection in Dynamic Distributed System"s, In: 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)
10. Pascal Felber, Xavier Defago, Rachid Guerraoui, Philipp Oser. : "Failure Detectors as First Class Objects"
11. Naohiro Hayashibara, Xavier Dfago, Takuya Katayam. , "Two-ways Adaptive Failure Detection with the Failure Detector", Workshop on Adaptive Distributed Systems (WADiS), pp.22–27, In: Sorrento, Italy, Oct. 2003.
12. M. Bertier, O. Marin, P. Sens., "Implementation and Performance evaluation of an adaptable failure detector. "In: Proc. of the 15th Int'l Conf. on Dependable Systems and Networks (DSN'02), pages 354–363, Washington, D.C., USA, Jun. 2002.
13. W. Chen, S. Toueg, and M. K. Aguilera., "On the quality of service of failure detectors." In: IEEE Transactions on Computers, 51(5):561–580, May 2002.
14. C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol". In: Proc. 8th IEEE Pacific Rim Symp. On Dependable Computing (PRDC-8), pages 146–153, Seoul, Korea, Dec. 2001.
15. Christof Fetzer Ulrich Schmid and Martin Susskraut. "On the Possibility of Consensus in Asynchronous Systems with Finite Average Response Times." In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICSCS'05).
16. Balachandran Natarajan, Aniruddha Gokhale, Shalini Yajnik, Douglas C. Schmidt. , "DOORS: Towards High-performance Fault Tolerant CORBA". In: Proceedings IEEE of the 2nd International Symposium on Distributed Objects and Applications (DOA '00). Antwerp, Belgium, Sep 2000.
17. Basu, B. Charron-Bost, S. Toueg. , "Solving problems in the presence of process crashes and lossy links. Technical Report" TR96-1609, Cornell University, USA, Sep. 1996.
18. Robert van Renesse, Yaron Minsky, Mark Hayden., "A Gossip-Style Failure Detection Service"Published by Chapman & Hall. IFIP 1996