FDAE: A failure detector for asynchronous events

A. Farruggia, M. Ortolani, G. Lo Re

In Proceedings of the Sixth International Conference on Networked Computing and Advanced Information Management (NCM), 2010, pp. 197-202

Publisher: IEEE

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5572274

FDAE: A Failure Detector for Asynchronous Events

Alfonso Farruggia, Marco Ortolani, and Giuseppe Lo Re
Department of Computer Engineering
University of Palermo
Viale delle Scienze, ed.6 - Palermo
Email: {farruggia,ortolani,lore}@dinfo.unipa.it

***Abstract*—Detecting element failures is a relevant issue in distributed systems. A fault tolerant system needs to detect a failure and recover from it promptly. In fact, traditional approaches to fault tolerance are usually not completely free from errors during the failure detection phase; a good failure detector is thus a very important component of them to minimize these errors. In this paper we present a failure detector able to monitor both asynchronous and synchronous elements of a distributed system by exchanging messages with the monitored elements. In order to assess the health status of monitored elements our failure detector relies on a simple query/ACK mechanism, which however requires a reliable timeout estimate in order to properly set the monitoring interval. To this purpose our failure detector uses the history of past estimates to compute new values for both quantities. The model proposed here introduces a new label to tag monitored elements, besides those used in traditional failures detectors. To evaluate this work, we compared it with two other algorithms by computing performance metrics, such as specificity and sensitivity, and by considering the number of required control packets. We also compared the performance of the failure detectors by computing their detection time.**

I. Introduction

Network interfaces are nowadays commonly available in virtually all computing devices, ranging from complex industrial equipments to simple appliances for home automation; besides mere data exchange, they may be fruitfully exploited for remote administration tasks, such as for instance system-wide monitoring of remote elements. The failure detector is an essential part of the monitoring subsystem and typical requirements include prompt failure detection and timely error reporting to the system administrator.

In a distributed system, the QUERY/ACK method is a basic mechanism to monitor an element, and it is characterized by two parameters: the *monitoring interval* and the *timeout*.The *monitoring interval* is the time elapsed between two checks, whereas the *timeout* is the time spent by the failure detector waiting for a response message from a monitored element. Computing accurate values for the two previous parameters may be very challenging, because of variations in both the network state and in the workload of each monitored element.

Chandra and Toueg [1] defined the basic concepts for unreliable failure detectors; those may incur into errors about the health status of a monitored element especially due to variations in the system workload. The same authors define possible tags for an element as: `alive` or `suspected`. An element will be tagged as `alive` if the failure detector receives a response to a control message within the timeout value. Otherwise, it is `suspected`, implying that the failure detector believes that the monitored element has crashed.

Furthermore, we can distinguish between two types of elements with respect to their activity. *Periodic* elements are characterized by a cyclic job; for instance, they may gather environmental data at fixed time intervals. On the other hand, the activity of *non periodic* elements is modeled via asynchronous events: for instance they may be only triggered on demand. One of the tasks of a failure detector is to estimate the timeout and the monitoring interval, so as adapt to the workload of any element; however this may be very challenging in the asynchronous scenario. This work is aimed to address generic scenarios, possibly including both kinds of elements. For instance, in an asynchronous scenario where an element has not been used in a long time, the failure detector would adapt to a low workload; however, the element could later be checked while its workload is high, which would surely affect the response time of the element and would have the failure detector erroneously generate premature timeout events, resulting in incorrect labeling of the element as `suspected`.

The approach proposed here aims to avoid this kind of premature deductions. If the failure detector checks an element during a high workload, our approach forces the element to signal its status to the detector, which would label it by using a novel tag (`busy`), specifically introduced here to this purpose; the timeout and the monitoring interval are computed by using a statistical approach that takes into account the history of past estimates.

This paper is organized as follows. The next Section presents the related works; in Section III we present the details of our work, namely the model used for the detector and the approach to estimating the timeout and monitoring time. In Section IV, the algorithm proposed is compared with the works presented in [2] e [3]. The last Section presents our conclusion.

II. Related works

The authors of [1] classify failure detectors according to two features: *completeness* and *accuracy*. *Completeness* characterizes the failure detector capability of suspecting every faulty process. *Accuracy* characterizes the failure detector capability of avoiding to label correct processes as supected.

In [4] two strategies for implementing a failure detector are presented, namely the *hear-beat* (or *push*) model, and the *interrogation* (or *pull*) model. In the former model, the monitored element periodically sends a heartbeat message to

the failure detector. If the failure detector receives the message within a specific time, the status of the monitored element is `alive`, otherwise it is `suspected`. In the latter model, it is the failure detector that periodically sends a message to the monitored elements. If the element replies to the failure detector then its status is `alive`, otherwise it is `suspected`.

Another model has been presented in [5], called *dual* model. This model combines the advantages of both the previous models; as in the pull model, the failure detector sends a message to an element, which however replies as described in the push model. The failure detector tags an element as `suspected` if a reply is missing twice consecutively.

The authors of [1] also define a taxonomy for classifying fault detectors with respect to completeness and accuracy; in particular, they define:

- *strong* completeness: eventually every process that crashes is permanently suspected by every correct process;
- *eventual weak* accuracy: there is a time after which some correct process is never suspected by any correct process.

In this taxonomy, our approach may be classified as *eventually strong* ($\diamond S$).

The following works show different failure detectors based on Chandra e Toueg's considerations. In [2] a failure detector, named ADAPTATION, that implements the *dual* model is presented. It uses two algorithms to adapt the timeout and the monitoring time to the actual workload of the system using the history of past values. The estimate is based on an empirical approach.

In [6] a failure detector based on the heart-beat model is presented, which however does not rely on a centralized failure manager. Each element has an instance of the failure detector. The novelty of this approach is that the failure detector is used only if the failure actually occurs. Two types of messages are used: the *application message* and the *control message*. The first one is used for data exchange, whereas the second one is used to discover the status of other elements. If the elements are communicating through an *application message*, *control messages* are not used. Only in case of communication problems, the failure detector uses a *control message*.

The authors of [3] presents DPCP (Discard Past Consider Present), which is applied both to the push and pull model. The algorithm calculates the timeout and the monitoring time, and those values are controlled by the frequency of fault monitoring messages. It is worth noting that increasing the monitoring frequency, the algorithm adapts the timeout to the system workload; moreover applying the algorithm to the push model fewer timeout false events are produced. DPCP does not use a particular technique for estimating the timeout; the results of the experiments are proven by running the algorithm with different values of monitoring interval and the history of previously samples is not considered.

In [7] the authors present a failure detector that implements the heart-beat strategy. This work introduces an estimator for the arrival time of messages, and an adaptor of QoS with respect to the needs of the application. This algorithm is based on *all-to-all* communication, where an element periodically broadcasts a heart-beat message via IP-Multicast. The algorithm is formed by two layers. In the first the arrival time of heart-beat message is calculated, and the algorithm finds a tradeoff between the number of false timeout events and the precision in the detection of a failure. In the second layer the *timeout moderator* is introduced. This value is incremented when a timeout event occurs.

In [8] a failure detector based on the previous algorithm is described, and similarly to the previous approach, the arrival time of *heartbeat* messages is estimated. The difference between the two works lies in the estimator of that parameter.

All the mentioned works implement similar approaches to failure detection in that elements are allowed only two statuses; we intend to propose here a variant of the traditional approach, by introducing a new label to refine the diagnosis of the status of an element, as will be explained in the following.

III. The Proposed Failure Detector: FDAE

Our work aims to detect potential failures in a distributed system, and considers its composing elements, or processes, as communicating through a QUERY/ACK mechanism. Following the traditional approach, we assume a reliable and secure communication channel for message exchange among elements, so that the only possible reason why a message does not reach its intended destination is that the receiving element has crashed.

The strategy adopted in this work extends the *dual-model* presented in [5], and also adopted in [2]. In our model after two missing replies to check messages, the suspected element is excluded from the list of elements to be checked later; however a suspected element that later becomes `alive` again will be recognized by the failure detector during a *discovery* phase that would restore the normal control loop.

In an actual application scenario, elements would act as service providers for the users, so that it is crucial that they are always alive, and we can distinguish the two mentioned kinds of elements: those which periodically execute a task, and those which are only occasionally triggered; we thus need a monitoring system to promptly detect potential element failures. We describe the system model in the following subsection using the same mathematical notation as in [7] and [6].

System Model

The system is composed of a finite set Π of elements Π=$\{p_1, p_2, ..., p_n\}$ monitored by a failure detector q. Each monitored element has a unique id and it may rely on other elements to complete its primary activity. The failure detector keeps a model q_{p_i} for each monitored element. We introduced in the model a variable $\triangle_w$, it keeps the time spent by each element for its activity.

We want to specifically address the case when an element is checked while it is carrying on a time-consuming task, as this might lead the failure detector to generate false timeout events following one of the traditional approaches, and consequently

Algorithm 1 Discovery

```
(1)  begin thread_Sender
(2)    sendDiscoveryMsg();
(3)    setNextDiscoveryTime(discoveryTime);
(4)    sleep(discoveryTime);
(5)  end
(7)  begin thread_Receiver
(8)    element p = ReadElement();
(9)    p.status = alive;
(10)   thread_Check_Element(p);
(11) end
```

label the element as `suspected`. In the considered scenario, this might happen because the reply of such element would likely be slower than the one of a low-workload element. In oder to avoid premature timeouts, and to minimize the number of false failures detections, the tag `busy` has been introduced, and its use is as follows.

The Failure Detection Algorithm

The failure detection algorithm proposed here is composed of two phases:

- *discovery*: where the failure detector looks for the element to monitor;
- *check*: where the failure detector assesses the status of a monitored element.

The *discovery* phase, presented in Algorithm 1, is performed during the initialization of the system, when the failure detector q sends a multicast discovery message via the *sendDiscoveryMsg()* function in order to identify all the elements installed in the distributed system. A generic element p_i receives the discovery message and replies to the failure detector q by sending its id. The id stores the element network address, and will be used by q during the *check* phase to access p_i. Upon reception of p_i's id, q will create the corresponding model q_{p_i}. This is a periodic phase, this needs to update the list of elements attached to the distributed system, and discover new elements. This phase implements a plug and play protocol. The discovery phase is repeated with a period equal to *discoveryTime* (it is important to note that the discovery time is different from the monitoring interval), and each element found in this phase is tagged as `alive`. The errors of evaluation made by the failure detector are repaired thanks to the periodicity of this phase. In fact, if q tags p_i as `suspected` in two subsequent check phases, the element is no longer checked. If p_i is not really `suspected`, it could be attached to q in the next *discovery* phase, and its label will be again `alive`. In this case, where the failure detector makes mistake, the p_i element will remain `suspected` for a max of $2 * discoveryTime$.

Algorithm 2 illustrates the *check* phase. Here the main job of the failure detector q is to check the status of an element p_i. Through the function *controlMsgTo(element)*, presented in Algorithm 3, q starts a QUERY/ACK session to perform the check, and depending an the return value it tags q as `alive`, `busy`, or `suspected`; afterwards, q goes into a sleep state.

Algorithm 2 Check

```
(1)  begin thread_Check_Element(element p)
(3)    test = controlMsgTo(p)
(5)    if (test.event == alive)
(6)      p.timeout = SetNextTimeout(p.rtt);
(7)      p.status = test.event;
(8)      p.testDelay = SetNextTestDelay();
(10) else if (test.event == busy)
(11)       p.timeout = SetNextTimeout();
(12)       p.status = test.event;
(13)       p.testDelay =
(14)         SetNextTestDelay(p.workingDelay);
(16)   else if (test.event == suspected)
(17)         p.timeout = SetNextTimeout();
(18)         p.status = test.event;
(19)         p.testDelay = SetNextTestDelay();
(20)       fi
(21)     end
(23)     sleep(p.testDelay);
(24)   end
```

Algorithm 3 Send Control Message

```
(1)  begin controlMsgTo(element p)
(2)    test_e = null;
(4)    try{
(5)      rttSession = timestamp(now);
(6)      connectTo(p);
(7)      event = waitingEvent(p.rtt);
(8)     if (event == alive)
(9)        rttSession = timestamp(now) − rttSession;
(10)       test_e.rtt = rttSession;
(11)       test_e.status = event.status;
(12)       test_e.workingDelay = event.workingDelay;
(13) else if (event == suspected)
(14)         test_e.status = event.status;
(15)      fi
(17)      }catch(busy_exception)
(18)      {
(19)       test_e.status = busy;
(20)      }
(21)      return test;
(22)   end
```

The traditional approach is shown in Figure 1. The failure detector q tags an element p_i as `alive` after receiving a reply to check message within the timeout value. It computes the round trip time (rtt), and estimates the values for the *timeout* and *interval time* for next check session. In our solution, the element includes an average value for its $\triangle_w$ into the reply message.

Fig. 1. Messages sequence: `alive`

Fig. 2. Messages sequence: `busy`

Figure 2 presents the case where the new tag `busy` is used. When q wants to check p_i but this is carrying on some time consuming task, it breaks the checking phase generating a *busy exception*; q catches it and suspends the check tagging p_i as `busy`. The failure detector in this case sets the next *monitoring interval* based on $\triangle_w$, because it expects to receive the reply to check message before that time, in this case q tags p_i again as `alive`. If q does not receive an answer from p_i, it starts the second check; if during it the status is again `busy` then q tags p_i as `suspected`, and it is excluded from the next check phase. In the scenario where the result of the first check is `suspected`, the failure detector does not set the *monitoring interval* based on $\triangle_w$, as presented in Figure 3. The excluded elements can be re-checked from the failure detector thanks to discovery phase, already discussed.

In this work we introduce an estimator for the *timeout* and the *monitoring interval*. The estimator is implemented for elements with unsynchronized clocks. We estimate the arrival time and the timeout using a statistical approach, and we compute the safety margin dynamically. The failure detector is modeled to be used a over long period of time, so that it needs to adapt at run-time to variations of workload of monitored elements and in network bandwidth. The rtt is measured only when the status of element is `alive`. If the element is `busy` or `suspected` the reply to check message is missing, so the rtt value cannot be properly measured. In the present work the rtt is used to compute the *timeout* and the *monitoring interval*. The first one depends directly on state of the element, while the second is computed from first, considering the $\triangle_w$. Our estimation follows the same

Fig. 3. Messages sequence: `suspected`

principles followed for the assessment of the *timeout* in the TCP protocol, presented in [9]. Before estimating the *timeout* and the *monitoring interval*, the estimated round trip time ($ertt$) is computed using the Exponential Mobile Average (EMA) considering a window w of samples; Simple Mobile Average (SMA)is used to compute the first w values of $ertt$:

$$ertt_i(ertt, w) = SMA_i(rtt, w) = \frac{\sum_{j=1}^{w-1} rtt_{i-j}}{w}$$

for the following values the formula below is used:

$$ertt_i(rtt_i, w) = ertt_{i-1} + k * (rtt_i - ertt_{i-1})$$

the estimate may be tuned by acting on the k parameter, which allows recent samples to weigh more than past ones. The parameters is set as follows:

$$k = \frac{2}{w+1}$$

Finally, the value of timeout is computed by increasing to $ertt$ estimate with safety margin $\triangle_\sigma$, which varies based on the status of an element, as specified below;

$$timeout = ertt_i + \triangle_\sigma * \begin{cases} C_a & if \quad status = \texttt{alive} \\ C_b & if \quad status = \texttt{busy} \\ C_s & if \quad status = \texttt{suspected} \end{cases}$$

where $\triangle_{\sigma(i)} = \beta * \triangle_{\sigma(i-1)} + (1-\beta) * \|ertt - rtt^*\|$
and the coefficients of C_a, C_b, and C_s are respectively set to 4.2, 4.0, and 3.8; they are empirically chosen.

To compute the *monitoring interval* we used the value of the *timeout*, correcting by adding the previous $ertt$ to give it a safety margin.

$$\begin{array}{l} monitoring\ interval = timeout + ertt + \\ \begin{cases} \triangle_w & if \quad status = \texttt{busy} \\ 0 & if \quad status = \texttt{suspected} \vee \texttt{alive} \end{cases} \end{array}$$

IV. Evaluation of failure detection

The present Section deals with the evaluation of our previously described approach to failure detection in terms of assessing the classification of the status of each monitored element. On one hand, we would like to minimize the number

Fig. 4. T_D: Detection Time

of false negative detections, so that most failures are correctly discovered; on the other hand, the number of false positive detections must also be kept low since a false alarm on a healthy element would trigger an expensive and unnecessary mechanism of failure repair. Any element in our experimental scenario is characterized by an unpredictable workload, which makes the estimation of appropriate values for the timeout and monitoring interval difficult; overestimating such quantities would result in delaying failure detection, whereas underestimating them may possibly lead to false detections. In [10], the authors present the metric to evaluate the speed of failure detectors, which is the time elapsed from the moment when an element p_i crashes to the time when the failure detector starts suspecting it permanently, as shown Figure 4; this time is called *detection time* (T_D).

In order to assess the behavior of our failure detector, and to compare it with other approaches proposed in literature, we consider two parameters, namely *specificity*, and *sensitivity*; moreover, in order to get some insight on the performance of FDAE, we consider its impact on the network load, by measuring the number of required control packets. The two former indices are statistical measures of the reliability of a classifier; in particular, sensitivity measures the ratio between the number of elements correctly labeled as healthy and the total number of elements labeled healthy; specificity analogously measures the proportion of elements correctly labeled as faulty.

In our experimental setting, the failure detector q and all monitored elements p_i reside on different computers connected via a LAN; we systematically generate failures on the p_i's and log the behavior of q. We consider two representative scenarios: in the former one, an element p_i carries on a periodic task, resembling the typical process for acquiring environmental data, whereas in the latter we consider a non periodic task, varying both in duration and in the time of occurrence, similar to the one of a process controlling some mechanical device (e.g. an RFId device for access control). We artificially add short random failures in both cases.

For the periodic element, we set an `alive` period of 120 s, followed by a `busy` phase of 2 s; while assessing algorithms that do not consider the `busy` tag, we assume a period of 122 s for a comprehensive `alive` phase. As regards the non periodic element the duration of the `alive` phase is in the range $1 - 120$ s, while the `busy` one lasts between 5 s and 10 s. In both cases, failures occur randomly according to a uniform distribution with a probability of 0.2, and their duration varies in the range $10 - 15$ s. Each experiment has been repeated 10 times and its duration was 20 m; during each run, FDAE was executed together with instances of the DCPC and ADAPTATION algorithms, for comparison.

TABLE I
PERIODIC SCENARIO

	DPCP	ADAPTATION	FDAE
sensitivity	0.96	0.91	0.96
specificity	0.97	0.49	0.95
# ctrl_pkts	109,558	7,020	4,430

TABLE II
NON PERIODIC SCENARIO

	DPCP	ADAPTATION	FDAE
sensitivity	0.95	0.90	0.96
specificity	0.96	0.36	0.93
# ctrl_pkts	113,081	8,552	4,089

In our context, *specificity* measures the percentage of correctly detected failures, and is defined as:

$$specificity = \frac{\#true\ failures}{\#true\ failures + \#erroneous\ failures}$$

Sensitivity indicates the percentage of correctly detected healthy elements, and is defined as:

$$sensitivity = \frac{\#true\ healthy}{\#true\ healthy + \#erroneous\ healthy}$$

Table I and Table II show the average values measured during the experiments for all three algorithms.

We can see that both FDAE and DPCP outperform ADAPTATION in the periodic scenario, thanks to their capability to adapt to the periodicity of the element workload; both those algorithms present few detection errors, however in this scenario our approach uses fewer control packets as compared to the DPCP algorithm.

On the other hand, in the non periodic scenario the three algorithms present similar values of *sensitivity*; however considering the *specificity*, our approach present a better tradeoff between the number of packets used to monitor an element and the performance metric.

The number of packets is used to highlight the overhead that each protocol imposes on the use of the bandwidth; FDAE outperforms the other algorithms in this respect. More specifically, our experiments allowed us to notice that, while the number of packets sent when an element is `alive` is comparable for all algorithms, FDAE heavily reduces the number of packet sent during the `busy` period. Our failure detector does not repeatedly check the element, but it just simply waits for a potential reply (see Figure 3); finally, our adopted policy allows the element to signal when it busy period ends so that in the `suspected` case it sends a maximum of two packets. Considering the same scenarios, we measured T_D for each of the implemented algorithms; the results are shown in Table III. These values emphasize the

TABLE III
SPEED OF FAILURE DETECTORS

	DPCP	ADAPTATION	FDAE
detection time (ms)	7.10	49.20	4.76

speed of our algorithm as compared to the two other reference algorithms. It is worth noting that FDAE outperforms both DCPC and ADAPTATION, and only DCPC gets closer to our algorithm, although it requires a much larger number of control packets, as discussed before.

V. CONCLUSION

Monitoring systems rely mainly on the reactivity of the failure detector. Our work shows a failure detector whose difference with the traditional approaches consists in precise identification of an uncertain status of a monitored element, due to its workload, via the introduction of a new tag.

Additionally, our implementation allows easy installation of additional elements in the system, thanks to its *discovery* phase that allows for their recognition at run-time. Our experiments proved that our approach has significative performance with respect to other state-of-the-art algorithms, as shown by the analysis of proper performance metrics.

We intend to pursue further research on this topic, possibly using this algorithm as the basis for a *self-healing* component of an autonomic system.

REFERENCES

[1] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, pp. 225–267, 1995.

[2] I. Sotoma and E. R. M. Madeira, "ADAPTATION - Algorithms to ADAPTive FAulT MonItOriNg and Their Implementation on CORBA," in *Proc. of the Third International Symposium on Distributed Objects and Applications, DOA'01*, 2001, pp. 219–228.

[3] ——, "DPCP (Discard Past Consider Present) – A Novel Approach to Adaptive Fault Detection in Distributed Systems," in *Proc. of The Eighth IEEE Workshop on Future Trends of Distributed Computing Systems, FTDCS 2001*, 2001, pp. 76–82.

[4] N. Sergent, X. Défago, and A. Schiper, "Failure detectors: implementation issues and impact on consensus performance," Laboratoire de Systémes d'Exploitation, École Polytechnique Fédérale de Lausanne, Switzerland, Tech. Rep., 1999.

[5] P. Felber, "The CORBA Object Group Service: a Service Approach to Object Groups in CORBA," Master's thesis, PhD thesis, École Polytechnique Fédérale de Lausanne, the Netherlands, 1998.

[6] C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol," in *In Proc. 8th IEEE Pacific Rim Symp. on Dependable Computing (PRDC 01*, 2001, pp. 146–153.

[7] M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," in *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 354–363.

[8] J. Zhou, G. Yang, L. Dong, and G. Liu, "Implementation and performance evaluation of an adaptable failure detector for distributed system," in *CIS '07: Proceedings of the 2007 International Conference on Computational Intelligence and Security*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 266–270.

[9] I. S. Institute, "RFC 793," 1981, edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html. [Online]. Available: http://rfc.sunsite.dk/rfc/rfc793.html

[10] W. Chen, S. Toueg, and M. Aguilera, "On the quality of service of failure detectors," *Computers, IEEE Transactions on*, vol. 51, no. 5, pp. 561 –580, may 2002.