Failure Detectors for Distributed Systems

Anurag Aggarwal
Department of Computer Science & Engg.
Indian Institute of Technology
Kanpur - 208016
Email: anuragag@cse.iitk.ac.in

Diwaker Gupta
Department of Computer Science & Engg.
Indian Institute of Technology
Kanpur - 208016
Email: gdiwaker@cse.iitk.ac.in

Abstract— **Failure detectors are an integral part of any fault-tolerant distributed system. Most of the implementable failure detectors are based on simple timeouts. The timeout period is a critical factor for the performance of a failure detector. In this paper we briefly look at the classes of failure detectors that have been proposed in theory and some of their implementations. We then study the *Quality of service* (QoS) of failure detectors. We first look at some QoS metrics that have been proposed and how they relate to the performance of the failure detector. Then we survey recent propositions that try to find optimal timeout values which satisfy some QoS constraints**

I. Introduction

Designing fault-tolerant distributed applications has been the goal of application developers almost ever since the inception of the area of distributed systems but it has not turned out to be easy to do so. Developing a failure detector which can identify the faulty processes seems to be the first step toward building such a system. Failure detectors have also gained importance in recent times due to their application in solving problems like *Consensus* and *Atomic Broadcast*. Consensus in *asynchronous systems* where there is no bound on the message delay is an attractive model because it has simple semantics and is more general than the synchronous model. But according to the FLP result [1] it is known that Consensus can't be solved in an asynchronous system subject to even a single failure. The intuition behind this result is that it is impossible for an application to distinguish between a "slow process" and a "failed process". To circumvent this result, research has focused on solving Consensus in partially synchronous systems by relaxing some conditions of asynchrony. Most of these models use a failure detector for identifying failed processes. Thus we find that the failure detectors are central to the paradigm of Distributed Computing. In this paper we will look at the impact of timeout value on the performance of the failure detectors and how to choose a timeout value to satisfy the given QoS constraints.

This paper is organized in 7 sections. Section II discusses the concept of Failure Detectors along with the some implementation issues. In section III we take a look at the QoS for Failure Detectors and how do they affect the performance of the failure detectors. In section IV and section V we discuss two approaches for finding the optimal timeout value given some parameters. In section VI we briefly discuss some other approaches and finally section VII gives some concluding remarks and future work.

II. Failure Detectors

A. Theoretical Concepts

The *unreliable failure detectors* introduced in [2] run as a module with each process and return a list of suspected processes. Depending upon the properties the failure detectors need to satisfy, they have been classified into different classes. Out of these classes the ones that are of interest are $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$. $\diamond\mathcal{P}$ includes all the failure detectors that after some unknown but finite time,make no mistake i.e. the list of suspects

includes all the crashed processes and no correct processes. $\diamond S$ is the weakest failure detector to solve Consensus problem in asynchronous distributed systems [3]. (Failure Detectors with different aims and properties are presented in [4])

Garg and Mitchell [5] introduced a class of failure detectors which can actually be implemented in asynchronous systems. But apart from that there are not many useful failure detectors that can be implemented in asynchronous systems because they would otherwise contradict the impossibility result [1]. So additional assumptions about the underlying system are taken which can be used to implement the class of failure detectors in which we are interested.

B. Implementation Issues

The implementation of failure detectors is generally based on timeouts. There are two *flow policies* between the failure detectors and monitored components which abstract behaviors of monitoring protocols used by failure detectors to monitor system components and the way the information about component failures is propagated in the system [6]. These models are known as *push* and *pull* models.

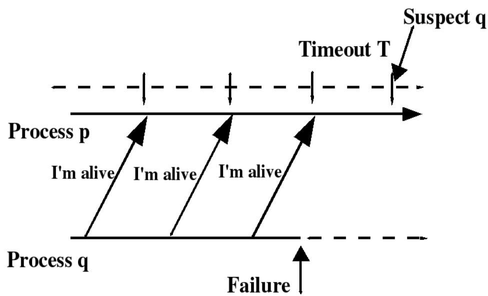

Fig. 1. Push Model

1) The Push Model: In the Push model, monitored components are active and the monitor (failure detector) is passive. Each monitored component periodically sends heartbeat messages to the failure detector which is monitoring the process. The failure detector suspects a process when it does not receive a heartbeat message from the monitored component within a certain time interval (timeout)T. If the failure detector receives a message from a suspected process then it removes it from the suspected list and starts monitoring it again(figure 1).

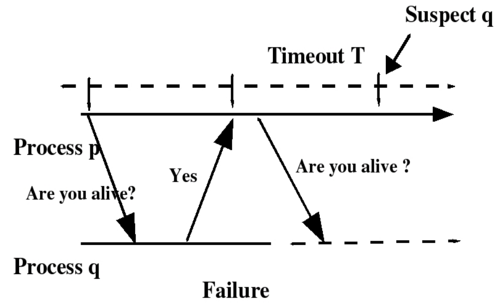

Fig. 2. Pull Model

2) The Pull Model: In the Pull model, monitored components are passive while the monitor or failure detector is active. The monitor sends liveness requests(*"Are you alive?"* messages) periodically to monitored components. If a monitored component replies, it means that it is alive. When the monitor does not receive a reply from the monitored component within a certain time interval(timeout) it starts suspecting the monitored component(figure 2).

3) Comparison of the two approaches: The Push model requires that only the monitored component send messages to the monitor whereas in Pull model both the monitor and monitored component send messages. So it appears that Push model is more efficient as it involves only *one-way* messages whereas Pull model requires *two-way* messages. But the argument in favor of Pull model is that the monitor need not send the liveness request regularly. Instead it can choose to do it when it really needs to know whether the process is alive. Also there are variations of Pull model that try to make it even more efficient. Felber et al. [6] discuss a *lazy* failure detection protocol in which processes monitor each other by using application messages whenever possible to get information on process failures. This protocol requires that each message be acknowledged. In the absence of application messages two processes control messages are used instead. Thus this protocol tries to reduce the number of messages exchanged but its performance depends largely on the nature

of the application. There have also been attempts to use a combination of the two models [6] to have the good features of both the models.

III. QoS FOR FAILURE DETECTORS

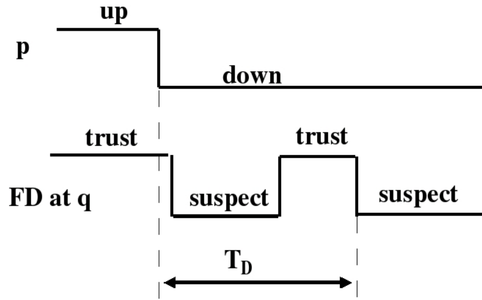

Fig. 3. Detection time T_D

Until very recently research on failure detectors was mainly concerned about their *eventual* behavior(e.g. a failure detector will eventually detect a failed process). These notions assisted in gaining a clear understanding of the implementable failure detectors in asynchronous systems which have no timing assumptions. However many applications have some timing constraints and for these applications determining the *eventual* behavior is not good enough. For example, if an application needs to solve consensus within a minute then a failure detector, which gives a guarantee of detecting a crash *eventually*, is useless for the application. Due to these application requirements failure detectors must satisfy some Quality of Service(QoS) constraints.

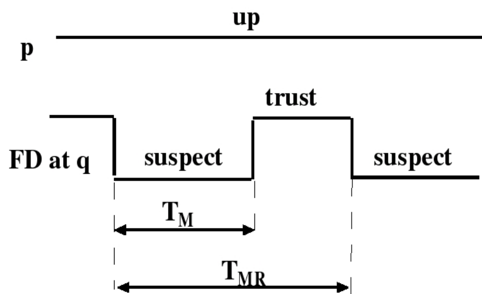

Fig. 4. Mistake duration T_M and mistake recurrence time T_{MR}

The QoS metrics should,in general, specify (a)how *fast* it detects a failure and (b)how *good* it is at avoiding the false suspicions. In other words we need to quantify the properties of *speed* and *accuracy* of failure detectors. Note that the speed of a failure detector is with respect to crashed or failed processes whereas the accuracy is with respect to correct processes. A failure detector's speed is easy to measure. It is the time elapsed from the instant when a process p crashes to the time when the failure detector starts suspecting p permanently. This QoS metric is called the *detection time* (figure 3). Quantifying the *accuracy* of failure detectors is more difficult. Consider the scenario in which a failure detector at process q checks if the process p is dead or not. Supposing that p is alive, a natural way of quantifying the accuracy of failure detector at q is the probability that it correctly reports that the process p is alive when queried randomly. This QoS metric is called the *query accuracy probability*. But consider the two failure detectors shown in the figure 5. Both of them have a query accuracy probability of 0.75 but FD_1 makes mistakes *less frequently* than FD_2. For some applications the rate of making mistakes might be more useful. So the metric needed to quantify this rate of making mistakes is termed as *mistake rate*(figure 6). But again note that this metric alone is not sufficient to characterize accuracy of failure detector. In figure 6 the two failure detectors have the same mistake rate but FD_2 has a more desirable property of short *mistake duration* i.e. FD_2 corrects its mistake earlier than FD_1. It is clear from the above examples that the metrics used to quantify failure detectors is not an easy task and depends largely on the requirements of the application. In the following section we give some more QoS metrics that are of interest to the application developers in general.

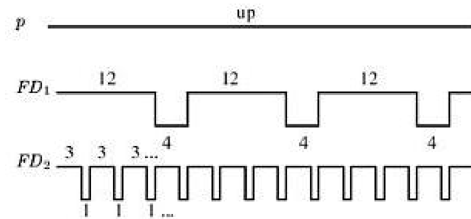

Fig. 5. FD_1 and FD_2 have same query accuracy probability of 0.75, but the mistake rate of FD_2 is four times that of FD_1

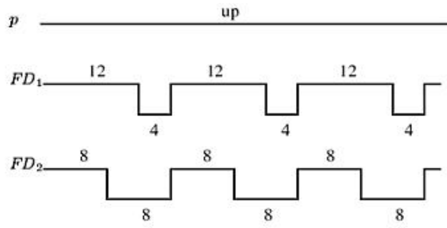

Fig. 6. FD_1 and FD_2 have same mistake rate of 1/16, but the query accuracy probability of FD_1 and FD_2 are .75 and .50 respectively

The QoS metrics are conditional upon the requirements of the application and the scenario in which we are interested in. We will take a look at two different settings and the metrics relevant in those settings.

A. QoS Metrics considering a pair of processes

Chen et al.[7] have given an extensive list of metrics for QoS of complete and efficient failure detectors. Although they have a considered a model consisting of two processes p and q interacting with one another, the metrics are relevant in most of the scenarios. We will first take a look at the failure detector model and the metrics proposed by them.

1) The failure detector model: Consider a system of two processes p and q with a failure detector at process q monitoring p. Also assume that during this period the process q does not crash.

At any instant of time the failure detector can either suspect (S) or trust(T) the process p. A transition occurs when the output of the failure detector changes. A S*-transition* occurs when the output of failure detector changes from T to S. Similarly, a T*-transition* occurs when the output of failure detector changes from S to T. It is also assumed that there are finite number of transitions in a given time interval. Only those failure detectors are considered which eventually reach *steady* state. By this we mean that after running for some time the behavior of the failure detector is not dependent on the initial conditions anymore. In other words the probability law governing the behavior of the failure detectors does not change over time once it reaches the steady state. The metrics given below refer to the state of the failure detector when it is in steady state.

2) Some metrics for QoS:

- **Detection time**(T_D):As described earlier in this section this is a measure of the *speed* of a failure detector. Mathematically, T_D is a random variable representing the time that elapses from the moment p crashes to the instant when the final S-transition takes place and there are no transitions afterwords(figure 3). If there is no such final S-transition then T_D=∞;If the transition takes place before p crashes then T_D=0.
- **Mistake recurrence time**(T_{MR}):This metric is a measure of the accuracy of the failure detector. Mathematically, T_{MR} is a random variable representing the time that elapses from an S-transition to the next one(figure 4).
- **Mistake duration**(T_M):This is also an accuracy metric for failure detectors. Mathematically, T_M is a random variable representing the time it takes from an S-transition to the next T-transition (figure 4).
- **Average mistake rate**(λ_M):This measures the rate at which a failure detectors makes mistakes,i.e. it is the average number of S-transitions per time unit. It is important for the applications where each failure detector mistake(each S-transition) is costly.
- **Query accuracy probability**(P_A):This is the probability that the failure detector's output is correct at a random time. It is important for applications which query the failure detectors at random times
- **Good period duration**(T_G):This measures the length of a *good period* – period in which the failure detector makes no mistakes. Mathematically, T_G is a random variable representing the time that elapses from a T-transition to the next S-transition.
- **Forward good period duration**(T_{FG}):For some short lived applications it might be useful if it can complete its work in the good period starting from a random point in the good period. Mathematically, T_{FG} is defined as a random variable representing the time that elapses from

a random time at which q trusts p, to the time of the next S-transition.

B. QoS Metrics for distributed failure detectors

Gupta et al.[8] have proposed a model consisting of distributed failure detectors running on a *group* of uniquely identifiable processes, which are subject to failures and recoveries and communicate over an unreliable network. We will first look at this model in detail and then go over some relevant metrics in this case.

1) Failure detector model: The model here consists of a large *group* of n members. Each member has a unique identity and is known to all other members. Members may suffer crash failures and recover subsequently. The probability of failure of a random group member at a random time is given by p_f. Also the probability of a message getting lost due to some network problem is given by p_{ml}.Also q_f and q_{ml} refer to (1-p_f) and (1-p_{ml}) respectively.

2) Some QoS metrics: The requirements of efficiency of the failure detector need to specified by the application in terms of the following parameters:

- **Speed**:Every member failure is detected by *some* non-faulty group member within $\mathcal{T}$ time units after its occurrence($\mathcal{T} \gg$ worst-case message round trip time)
- **Accuracy**:At any time instant, for every non-faulty member M_i not yet detected as failed, the probability that no other non-faulty group member will (mistakenly) detect M_i as faulty within next $\mathcal{T}$ time units is at least (1-$\mathcal{P}M(\mathcal{T})$).
- **Network load**:The *worst-case network load L* of a failure detector protocol is the maximum number of messages transmitted by any run of the protocol within any time interval of length $\mathcal{T}$, divided by $\mathcal{T}$. This load L should be minimum or optimal.

C. Choice of Primary Metrics

The metrics that are of importance are not always independent. We need to determine the relationship between metrics so that we can identify the sets of independent metrics and just monitor those. The relationships are also dependent on the network model considered. Chen et. al [7] have shown that average mistake rate, query accuracy probability, good period duration and forward good period duration can be expressed in terms of detection time, mistake recurrence time and mistake duration. So the three metrics detection time, mistake recurrence time and mistake duration are chosen as the *primary metrics*. Another reason for choosing them as the primary metrics is that if one failure detector FD_1 is better that another failure detector FD_2 in terms of these three metrics then it would also be better in terms of the other metrics. So the comparison of two failure detectors becomes easier if we choose the above stated primary metrics.

IV. QoS FOR FAILURE DETECTORS FOR A PAIR OF PROCESSES

The model proposed by Chen et al.[7] was discussed in the section III. We now take a look at the failure detector protocol satisfying the QoS constraints provided by them.

A. Problem with the common Push protocol

The Push protocol discussed in section II has two problems, one regarding its accuracy and another related to its detection time. Consider the i-th heartbeat message m_i sent from process p to process q. The timer for m_i is started as soon as soon as m_{i-1} is received by the process q. This would mean that the timeout for m_i is dependent not only on the delay of m_i itself but also on the delay of message m_{i-1} which is clearly not desirable. The other problem is in the case when a process sends a heartbeat just before crashing. If the delay of this message is d and the timeout is TO then the worst case detection time for the algorithm is d+TO. Dependency of detection time on message delay is not desirable as this can be arbitrarily large. The proposed algorithm does away with this dependencies.

B. The Probabilistic Network Model

We assume that the link connecting processes p and q does not duplicate messages but can delay or drop some messages. The message loss and message delay are characterized by (1) message loss probability, p_L and (2) a random variable, D, denoting the expected delay of a message. Although the locals clocks need not be synchronized but there should not be any clock drift.

C. The Algorithm

The monitored process p periodically sends heartbeat messages m_1, m_2, m_3, ... to q every η time units. Let σ_i denote the send times of the messages. The monitoring process maintains the sequence τ_i=σ_i+δ where δ is a parameter of the algorithm. Consider a time period [τ_i,τ_{i+1}). At time τ_i, q checks whether it has received a message m_j with $j \geq i$. If it has then q trusts p throughout the interval [τ_i,τ_{i+1}). If it does not receive any such message then q starts suspecting p. If a message m_j is received before $\tau i + 1$ then q starts trusting p till the time τ_{i+1} otherwise p is suspected for the whole time interval. The intuition behind the algorithm is that we want to consider only those messages which are still *fresh*. The detailed algorithm with parameters η and δ is called NFD-S and is given in figure 7

Process p:
(1) for all $i \geq 1$, at time $\sigma_i = i.\eta$, send heartbeat m_i to q;
Process q:
(2) Initialization: *output* = S;
(3) for all $i \geq 1$, at time $\tau_i = \sigma_i + \delta$:
(4) **if** did not receive m_j with $j \geq i$ **then** $output \leftarrow S$; $\{$suspect p if no fresh message is received $\}$
(5) upon receive message m_j at time $t \in [\tau_i, \tau_{i+1})$:
(6) **if** $j \geq i$ **then** $output \leftarrow T$;

Fig. 7. Failure Detector algorithm NFD-S with parameters η and δ

D. The QoS Analysis of the Algorithm

Definitions

1) For any $i \geq 1$,let k be the smallest integer such that for all $j \geq i + k$, m_j is sent at or after time τ_i
2) For any $i \geq 1$,let $p_j(x)$ be the probability that q does not receive message m_{i+j} by time $\tau_i + x$, for every $j \geq 0$ and every $x \geq 0$; let p_0 = $p_0(0)$.
3) For any $i \geq 2$,let q_0 be the probability that q receives message m_{i-1} before time τ_i
4) For any $i \geq 1$,let $u(x)$ be the probability that q suspects p at time $\tau_i + x$, for every $x \in [0, \eta)$.
5) For any $i \geq 2$,let p_s be the probability that an S-transition occurs at time τ_i.

Proposition

1) $k = \lceil \delta/\eta \rceil$.
2) For all $j \geq 0$ and for all $x \geq 0$, $p_j(x) = p_L + (1 - p_L)Pr(D > \delta + x - j\eta)$.
3) $q_0 = (1 - p_L)Pr(D < \delta + \eta)$.
4) For all $x \in [0, \eta), u(x) = \prod_{j=0}^{k} p_j(x)$.
5) $p_s = q_0.u(0)$

Properties of NFD-S

1) The detection time is bounded by $T_D \leq \delta + \eta$
2) The average mistake recurrence time is

$$E(T_{MR}) = \frac{\eta}{p_s}$$

3) The average mistake duration is

$$E(T_M) = \frac{\int_0^\eta u(x)dx}{p_S}$$

E. Configuring the failure detector to satisfy QoS requirements

We are given the QoS requirements of the failure detector and we need to compute the parameters η and δ of the NFD-S algorithm so that the QoS are satisfied. We assume that (a) The local clocks of the processes are synchronized and (b) The message loss probability p_L and the distribution of message delays $Pr(D \leq x)$ are given. The QoS requirements that are specified are

1) T_D^U, the upper bound on detection time
2) T_{MR}^L, the lower bound on mistake rate
3) T_M^U, the upper bound on the average mistake duration

Note that we would like to maximize the value of η satisfying the above requirements so that the number of messages

are minimized. To compute such a solution the following procedure is followed:

Step 1 : Compute $q_0' = (1 - p_L)Pr(D < T_D^U)$, and let $\eta_{max} = q_0' T_M^U$. If $\eta_{max} = 0$ then output "QoS cannot be achieved" and stop; else continue.

Step 2 : Let

$$f(\eta) = \frac{\eta}{q_0' \prod_{j=1}^{\lceil T_D^U/\eta \rceil - 1} [p_L + (1 - p_L)Pr(D > T_D^U - j\eta)]}$$

Find the largest $\eta \leq \eta_{max}$ such that $f(\eta) \geq T_{MR}^L$. Such an η always exists. To find this we can use some simple numerical method, such as binary search.

Step 3 : Set $\delta = T_D^U - \eta$, and output η and δ .

F. Dealing with more generic situation

In the previous section for finding the parameters η and δ we needed to know p_L, probability of message loss and $Pr(D \leq x)$,the probability distribution of the message delays. We can estimate the $Pr(D \leq x)$ in terms of p_L,$V(D)$ (Variance of message delays) and $E(D)$ (Expected value of message delays). We can then go on to estimate p_L, $E(D)$ and $V(D)$ by looking at the history of the algorithm. This way we don't need the prior information about the message behavior. The other assumption about the local clocks being synchronous still remains there. We needed this so as to set the receiving times τ_i by shifting the sending times of heartbeat.To do away with this assumption we can use the *expected arrival times* of the heartbeats instead of the actual time. Here the assumption is that there is no clock drift which is a reasonable assumption to make. The new algorithm is called NFD-U. If we don't know the *expected arrival times* of the messages then we can estimate them as well using the history of message arrival times. The algorithm using the *estimates* of expected arrival times is referred to as NFD-E.

G. Results

Both the new failure detector algorithms proposed (NFD-S and NFD-E) and the simple algorithm commonly used were simulated. The values of the various parameters used for setting up the simulation environment and reasons for choosing them are as follows :

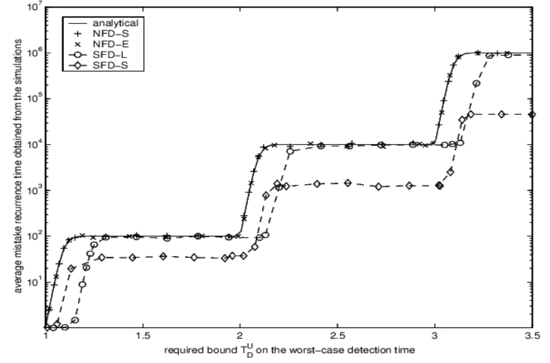

Fig. 8. The average mistake recurrence times obtained by : (a) simulating the new algorithms NFD-S and NFD-E (shown by + and ×), (b) simulating the simple algorithm (shown by -○- and -◇-), and (c) plotting the analytical formula for $E(T_{MR})$ of the new algorithm NFD-S (shown by –).

1) $\eta = 1$: To normalize the inter-sending time in both the algorithms
2) p_L = 0.01 : Close to the value in practical systems
3) $Pr(D \leq x) = 1 - e^{-x/E(D)}$ for all $x \geq 0$: Characteristic of message delays in many practical systems and also allows easy comparison of simulation results
4) $E(D) = 0.02$: Again chosen close to the values in practical systems (e.g. Internet)

For comparing the results of simulation, the parameters of the algorithms are chosen such that they satisfy the same bound T_D^U on the detection time. The following observations could be made from the simulation runs for values of T_D^U ranging from 1 to 3.5 :

- The accuracy of the algorithms NFD-S and NFD-E are very similar and the results of both the algorithms match the analytical formula for $E(T_{MR})$ given earlier in this section
- The simple algorithm as such does not have an upper bound on detection but by a slight modification a bound can be imposed. The idea is to have a *cutoff time* c such that any heartbeat which is delayed by more than this time is discarded. As a result the detection time T_D is bounded by $TO + c$. For the simulation two cutoff times : $c = 0.16$

and $c = 0.08$ were chosen. The algorithms corresponding to them are referred to as SFD-L and SFD-S respectively. The simulation results show that the accuracy of the new algorithms is better than both SFD-L and SFD-S.

V. Scalable and Efficient Distributed Failure Detector

Gupta et al.[8] present a failure detector which is useful in distributed systems. Large scale distributed applications need a light-weight failure detector algorithm which minimizes the *network load* in addition to being efficient. These detectors need to have good *scalability* so that they can be used even if more nodes become part of the application. A distributed algorithm has been proposed that tries to balance the load on different machines and satisfies application-defined efficiency constraints. We have already presented the model and the metrics that this model uses. Now we will discuss the algorithm and the bounds that it satisfies.

A. Quantification of Network Load

As discussed earlier the aim of this algorithm is to guarantee that the worst case network load L imposed is close to the optimal, with equal expected load per member. In this regard a theorem relating the optimal worst case network load L^* is stated below:

THEOREM : Any distributed failure detector algorithm for a group of size $n(\gg 1)$ that deterministically satisfies the *completeness*, *speed*, *accuracy* requirements above, for given values of $\mathcal{T}$ and $\mathcal{PM}(\mathcal{T})(\ll p_{ml})$, imposes a minimal worst-case network load (messages per time unit, as defined above) of:

$$L^* = n.\frac{log(\mathcal{PM}(\mathcal{T}))}{log(p_{ml}).\mathcal{T}}$$

Furthermore, there is a failure detector that achieves this minimal worst-case bound while satisfying the *completeness*, *speed*, *accuracy* requirements. (For proof of this theorem refer to [8])

L^* is thus the optimal worst-case network load required to satisfy the *completeness*, *speed*, *accuracy* requirements.

To measure the performance of a protocol in terms of network load, *sub-optimality factor* of a failure detector that imposes a worst-case network load of L is defined as $\frac{L}{L^*}$ For a simple heartbeat implementation, the sub-optimality factor varies as $\theta(n)$ for any values of p_{ml},p_f and $\mathcal{PM}(\mathcal{T})$.

B. A Randomized Distributed Failure Detection Algorithm

Integer pr; /* Local period number */

Every T' time units at M_i :

0. $pr = pr + 1$
1. Select random member M_j from view
   Send a ping(M_i, M_j, pr) message to M_j
   Wait for the worst-case message round-trip time for
   an ack(M_i, M_j, pr) message
2. If have not received an ack (M_i, M_j, pr) message yet
   Select k members randomly from view
   Send each of them a ping-req (M_i, M_j, pr) message
   Wait for an ack (M_i, M_j, pr) message
   until the end of period pr

3. If have not received an ack (M_i, M_j, pr) message yet
   Declare M_j as failed

Anytime at M_i :

4. On receipt of a ping-req (M_m, M_j, pr) ($M_j \neq M_i$)
   Send a ping(M_i, M_j, M_m, pr) message to M_j
   On receipt of an ack (M_i, M_j, M_m, pr) message
   from M_j
   Send an ack (M_m, M_j, pr) message to received to M_m

Anytime at M_i :

5. On receipt of a ping (M_m, M_i, M_l, pr) message
   from member M_m
   Reply with an ack (M_m, M_i, M_l, pr) message to M_m

Anytime at M_i :

6. On receipt of a ping (M_m, M_i, pr) message from
   member M_m
   Reply with an ack (M_m, M_i, pr) message to M_m

Fig. 9. Protocol steps at a group member M_i. Each message also contains the current incarnation number of the sender.

For this algorithm the *speed* condition is relaxed to detect

a failure within an *expected* (rather than exact) time bound of $\mathcal{T}$ time units after the failure. It satisfies the *completeness* and *accuracy* constraints and imposes an equal expected load on each group member. The worst case network load L differs from the optimal L^* by a sub-optimality factor independent of group size n. The failure detector algorithm uses two parameters: protocol period T' (in time units) and integer k, which is the size of failure detection subgroups. These parameters are known *a priori* to all group members. Here the clocks need not be synchronized but should not have any drifts.

The algorithm is formally described in figure 9. At the start of a protocol period of length T' a member M_i selects a random member, say M_j, and sends a ping message to it. If M_i does not receive a replying ack from M_j within some time-out (determined by the message round-trip time), which is $\ll \mathcal{T}$, it selects k members at random and sends to each a ping-req message. Each of the non-faulty members among these k which receives the ping-req message subsequently pings M_j and forwards the ack received from M_j, if any, back to M_i. In the protocol k random members are chosen to send a ping to M_j rather than sending k repeat ping messages so that if there are different message loss probabilities at different members they get evenly distributed out.

C. Analysis

1) The expected time between failure of member M_j and its detection by some non-faulty member is

$$E[\mathcal{T}] = T'.\frac{1}{1-e^{-q_f}} = T'.\frac{e^{q_f}}{e^{q_f}-1}$$

So we can get a configurable value for T' as a function of $\mathcal{T}, p_f$

2) Let $C(p_f) = \frac{e^{q_f}}{e^{q_f}-1}$. Then

$$\mathcal{P}M(\mathcal{T}) \simeq q_f.(1-q_{ml}^2).(1-q_f.q_{ml}^4)^k.C(p_f)$$

This gives us

$$k = \frac{log[\frac{\mathcal{P}M(\mathcal{T})}{q_f.(1-q_{ml}^2).\frac{e^{q_f}}{e^{q_f}-1}}]}{log(1-q_f.q_{ml}^4)}$$

3) The sub-optimality factor is given by

$$\frac{L}{L^*} = g(p_f, p_{ml}) + \frac{f(p_f, p_{ml})}{-log(\mathcal{P}M(\mathcal{T}))}$$

where $g(p_f, p_{ml})$ is :

$$[4.\frac{log(p_{ml})}{log(1-q_f.q_{ml}^4)}.\frac{e^{q_f}}{e^{q_f}-1}]$$

and $f(p_f, p_{ml})$ is :

$$[\{2-4.\frac{log(q_f.(1-q_{ml}^2).\frac{e^{q_f}}{e^{q_f}-1})}{log(1-q_f.q_{ml}^4)}\}\times(-log(p_{ml})).\frac{e^{q_f}}{e^{q_f}-1}]$$

Therefore, the sub-optimality factor is independent of the number of group size $n(\gg 1)$. More analysis shows that even the expected network load E$[L]$ can be upper-bounded from the optimal L^* by a factor that is independent of the group size n.

D. Results

The analysis of the protocol shows that it imposes a worst-case network load that differs from the optimal by a sub-optimality factor greater than 1. For very stringent accuracy requirements ($\mathcal{P}M(\mathcal{T})$ as low as e^{-30}), reasonable message loss probabilities and process failure rates in the network (upto 15% each), the sub-optimality factor is not as large as that of the traditional distributed heartbeat protocols. The test results of the implementation of the algorithm are still awaited.

E. Comparison with the previous approach

The network model and the corresponding algorithm for the model proposed by Chen et al. [7] takes into account only two processes whereas this algorithm is unique and more practical in that it works for a group of processes. However,the previous work is significant as it is the first to propose QoS metrics for failure detectors formally and considers many QoS metrics that are useful for applications. The second approach mainly focuses on the scalability metric and does not discuss in

detail the reasons for choosing the other metrics. Also the first approach is more general in the sense that it does not require any knowledge of the network behavior and can estimate the parameters like p_L. On the other hand the second approach requires the network behavior parameters for estimating the algorithm parameters. So we see that both the approaches have their pros and cons and have relevance in their respective domains.

VI. Other Approaches

A. Globus failure detection service

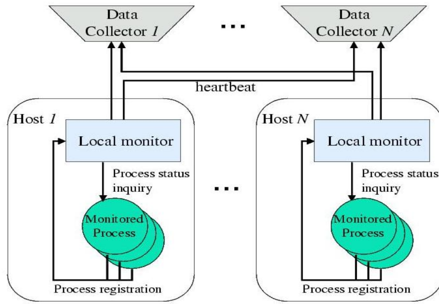

Fig. 10. Globus failure detection service

Stelling et al. [9] proposed a failure detection service for the *Globus toolkit*. Their model treats two components of the same computer in a different way than two components on two different computers. This is done for efficiency reasons and is more closer to the real life situation where more than one components on the same machine need to be monitored. The architecture of the proposed failure detector service has two layers: the lower layer includes *local monitors* and the upper layer includes *data collectors*(see figure 10). The local monitor is responsible for monitoring the host on which it runs as well as selected processes on that host. It periodically sends heartbeat messages to data collectors including information on the monitored components. The data collectors receive heartbeats from local monitors, identify failed components, and notify applications about relevant events concerning monitored components. This approach improves the failure detection time in a grid. However, its major drawback is that the architecture is static and is not adaptable to a dynamic environment in which components can join and leave at runtime.

B. Gossip-style protocols

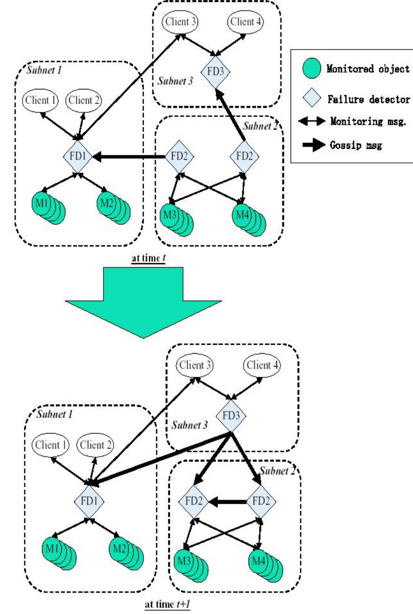

Fig. 11. Gossip-style protocols

Renesse et al. [10] distinguish two variations of gossip-style protocols: *basic gossiping* and *multi-level gossiping*(see figure 11). In the basic gossiping protocol, a failure detector module is resident at each host in the network. It maintains a list with an entry for each failure detector module known to it. This entry includes a counter called the heartbeat counter that will be used for failure detection. Every T_{gossip} seconds, each failure detector module picks another failure detector module randomly and sends it a list after incrementing its heartbeat counter. The receiving failure detector module merges its local list with the received list and adopts the maximum heartbeat counter for each member. Occasionally each member broadcasts its list to recover from eventual network partitions. Each member also maintains, for each other member in the list, the last time that its heartbeat counter was increased. If the heartbeat counter has not increased for some time interval(T_{fail}), the member is considered to have crashed.

To adapt it to large scale network, a variant of the basic gossiping protocol called multi-level gossiping protocol is proposed. The multi-level gossiping protocol uses the structure of Internet domains and subnets and their mapping into IP address to identify domains and subnet and map them into different levels. Most gossip messages are sent by the basic protocol within a subnet, and few gossip messages are sent between subnets, and fewer between domains. The values of the parameters T_{gossip} and T_{fail} are chosen so that the erroneous failure detection is less than some small threshold $P_{mistake}$. Gossip style protocols have many advantages. They are resilient against a small number of message loss and process failures. The probability that a member is falsely reported as having failed is independent of the number of processes. This algorithm also scales well in both detection time and network load.

VII. Conclusion

This paper looked into the issues concerning the implementation of failure detectors and their QoS specification. As this field is still in germinal stages, very little work has been done in it. We try to identify the important QoS metrics and discuss two failure detector protocols which try to ascertain the timeout values using the application specified QoS constraints. The effectiveness and the utility of each solution was also addressed. It was clear that the QoS metrics that need to be considered depend upon the nature and requirements of the application.

A. Future Work

As our future work we would try to extend the approach used by [7] to failure detectors based on Pull model and come up with optimal timeout values based on the application specified QoS parameters. We would also try to explore some more QoS metrics that could be of use to applications and develop protocols which can satisfy constraints on these metrics.

Acknowledgment

We would like to thank Dr. Vijay Garg for his guidance and support in this work.

References

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[2] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *Journal of the ACM (JACM)*, vol. 43, no. 4, pp. 685–722, 1996.

[4] M. Raynal, “Quiescent uniform reliable broadcast as an introduction to failure detector oracles,” *Lecture Notes in Computer Science*, vol. 2127, pp. 98–??, 2001.

[5] V. K. Garg and J. R. Mitchell, “Implementable failure detectors in asynchronous systems,” in *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. Springer-Verlag LNCS, no. 1530. Chennai, India: Springer-Verlag, 1998, pp. 158–169.

[6] P. Felber, X. Défago, R. Guerraoui, and P. Oser, “Failure detectors as first class objects,” in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA’99)*, Edinburgh, Scotland, 1999, pp. 132–141.

[7] W. Chen, S. Toueg, and M. K. Aguilera, “On the quality of service of failure detectors,” in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*. New York: IEEE Computer Society Press, 2000.

[8] I. Gupta, T. Chandra, and G. Goldszmidt, “On scalable and efficient distributed failure detectors,” in *Proceedings of 20th Annual ACM Symposium on Principles of Distributed Computing*. ACM press, 2001, pp. 170–179.

[9] P. Stelling, C. DeMatteis, I. T. Foster, C. Kesselman, C. A. Lee, and G. von Laszewski, “A fault detection service for wide area distributed computations,” *Cluster Computing*, vol. 2, no. 2, pp. 117–128, 1999.

[10] R. V. Renesse, Y. Minsky, and M. Hayden, “A gossip-style failure detection service, Tech. Rep. TR98-1687, 28, 1998.