Fast Failure Detection in a Process Group

Xinjie Li and Monica Brockmeyer*
Department of Computer Science
Wayne State University, Detroit, MI 48202
Fax: +1-313-577-6868
{xinjieli, mbrockmeyer}@wayne.edu

Abstract

Failure detectors represent a very important building block in distributed applications. The speed and the accuracy of the failure detectors is critical to the performance of the applications built on them. In a common implementation of failure detector based on heartbeats, there is a tradeoff between speed and accuracy so it is difficult to be both fast and accurate. Based on the observation that in many distributed applications, one process takes a special role as the leader, we propose a Fast Failure Detection (FFD) algorithm that detects the failure of the leader both fast and accurately. Taking advantage of spatial multiple timeouts, FFD detects the failure of the leader within a time period of just a little more than one heartbeat interval, making it almost the fastest detection algorithm possible based on heartbeat messages. FFD could be used stand alone in a static configuration where the leader process is fixed at one site. In a dynamic setting, where the role of leader has to be assumed by another site if the current leader fails, FFD could be used in collaboration with a leader election algorithm to speed up the process of electing a new leader.

1 Introduction

The notion of failure detectors has played an very important role in the theoretic work of distributed systems involving a group of processes. The study of failure detectors started as a response to the famous impossibility result of consensus in a asynchronous distributed system with unreliable networks [8]. It is well known that in a asynchronous distributed system with unreliable networks, it is impossible to distinguish a slow process from a crashed process so it is impossible to reach consensus across a process group [8]. To work around the impossibility result, one thought is to augment the asynchrony model with a failure detector satisfying certain properties such that consensus could be reached. A failure detector is a distributed *oracle* outputting the trusted or suspected remote peers at local processes. Chandra and Toueg [5] first classified failure detectors by two properties, *completeness* and *accuracy*. Completeness characterizes the ability on detecting failures. Accuracy characterizes the ability of not mistakenly suspecting correct processes. Different *classes* of failure detectors are defined according to the level of completeness and accuracy they provide. They also proved that the weakest failure detector to be able to solve consensus problem is the class of *eventually weak* detectors, denoted $\diamond\mathscr{W}$. The $\diamond\mathscr{W}$ satisfies two conditions, *weak completeness*, which means eventually every crashed process is suspected by some process; and eventual *weak accuracy*, which means eventually some correct process is never suspected by any process.

Failure detectors are both a powerful abstraction to augment the asynchronous model and also a useful building block in solving real world problems. However, completeness and accuracy can not both be deterministically satisfied in an asynchronous system [5]. Notice that completeness on its own is not very useful, as the detector can stratify completeness simply by suspecting every process. So it is essential that both properties be satisfied. Chandra and Toeug [5] propose a partial synchrony model under which $\diamond\mathscr{W}$ is implementable. In this partial synchrony model, the timing attributes (maximum message delay) are bounded, but the bounds are unknown and hold only after an unknown stabilization interval. While failure detectors are implementable in partial synchronous model, we notice that the accuracy property only holds *eventually*, as the stabilization time is unknown and the bounds on the message delay is also unknown. Chandra and Toueg also propose another failure detector, the Ω failure detector in [4]. The Ω failure detector has the property that eventually all correct processes trust the same correct process. It is also shown in [4] that Ω and $\diamond\mathscr{W}$ are reducible to each other.

The Ω is of special interest to us in this paper because

*This material is based upon work supported by the National Science Foundation under grant CAREER-0347222.

it is also seen as a leader election algorithm, as eventually it outputs the same trusted leader at every process. We will use Ω and term *leader election algorithm* interchangeably.

Though there is no restriction on how to implement failure detectors, almost all implementations use timeouts. By letting the monitored process sending out heartbeat messages, other processes could detect the failure from the missing of timeouts. In partially synchronous systems, the timeout usually starts out with a small value and adaptively increases on receiving a late arriving heartbeat messages from a suspected process.

In practice, however, the message delay is hard to bound. Measurement of round trip delays over [14] the Internet has demonstrated that there is a large temporal variation in round-trip time and the distribution has a long tail on the right hand side. This means even a very large timeout value might not be large enough. On the other hand, if the performance requirements of an application puts restrictions on the timeout value, the accuracy guarantee of failure detection becomes probabilistic.

If a failure detector could make mistakes at any time, the application would want to know and have requirements on the speed and accuracy of the failure detector. Intuitively there is a tension between speed and accuracy in timeout based failure detectors, suspecting upon multiple or longer timeout will increase the accuracy at the cost of slower detection. Chen *et al.* [6] conducted the first quantitative study on the Quality of Service (QoS) of pair-wise failure detectors and proposed a set of QoS metrics.

In many distributed applications, different processes take vastly differently roles. Quite often, one process is more important than the rest and takes the *leader* role. We say that such applications are *asymmetric*. This paper proposes a new algorithm, Fast Failure Detection (FFD) to improve the QoS of distributed failure detection of the leader in asymmetric applications. The basic idea is that if multiple monitor all timed out on the leader, then the accuracy in suspecting the leader is much higher. We call this scheme *spatial multiple timeouts*. In the traditional point-to-point failure detectors, since one timeout is usually not accurate enough, the monitor process usually suspects the monitored process after multiple consecutive timeouts. We refer to this scheme as *temporal multiple timeouts*. Since the metrics proposed in the point-to-point settings [6] do not fit perfectly in the group settings, we will simply define accuracy and speed as the metrics on the detection of the leader process. FFD is fast, as it detects the failure of the leader within a time period of just a little more than one heartbeat interval (Proposition 4.2). It is also highly accurate due to multiple independent timeouts. FFD is also communication efficient, requiring that only $n-1$ links are permanently active. Note that FFD is not a failure detector because it is not a distributed oracle that outputs at every local process. However we will see that it naturally complements a failure detector algorithms and increases the performance of the latter.

Now look at some example applications of FFD. As the first example, consider a distributed service replicated using primary-backup scheme. Here the leader role is manually assigned to a process. FFD could be used on its own to detect the failure of the primary server much faster than traditional detection algorithms.

As the second example, consider an distributed, replicated data storage service running over Internet. The replica processes could crash and the Internet could lose messages or delay messages for an extended period of time. The data storage system needs to maintain data consistency and tolerate any number of process crash failures. For example, Lamport's Paxos [11, 12] algorithm addresses this problem by having one leader propose and enforce the agreed on order of command on all replicas. Paxos could be built using a Ω [3]. In dynamic environments, FFD has to work in collaboration with a leader election algorithm. A leader election algorithm uses its own timeouts to monitor the current leader. However, FFD usually detects the failure much faster and prompts the leader election algorithm to select a new leader. On the other hand, FFD depends on the leader election algorithm to tell it what the current trusted leader is.

The rest of the paper is organized as follows. Section 2 briefly summarizes related work in this area. Section 3 describes the system model. Section 4 introduces our fast failure detection algorithm and gives proofs on its performance. Section 5 shows the application of FFD to leader election algorithms. Section 6 discusses the issues on the assumptions we make. Finally, Section 7 concludes the paper.

2 Related Work

Chandra and Toueg [5] first formally classified failure detectors by completeness and accuracy. The classification was mainly to capture the synchrony requirements on the system in order to solve the consensus problem. They also give a reference implementation of failure detection based on heartbeat messages [5]. In heartbeat method, process p sends a heartbeat message to q at regular time interval; q trusts p when q receives the message and starts a timeout timer; q suspects p if the timer expires before a new heartbeat from p is received. The heartbeats method is simple and it guarantee s completeness. The fully distributed heartbeats method is not scalable as it results in $O(n^2)$ periodic heartbeat messages.

Previous research addressed the efficiency in implementing failure detectors. [1, 2, 13] give implementations of Ω that require only $n-1$ channels that permanently carry periodic messages.

The aforementioned work were all based on the partial synchrony models. In the asynchronous network model, no failure detector could be accurate [5]. As a result, the application can only demand a probabilistic accuracy. Usually, an application also has requirements on the speed of failure detectors. It is easily seen that there is a tradeoff between speed and accuracy. In order to get a higher accuracy, usually a process is suspected only after several consecutive timeouts instead of one.

In the traditional timeout based failure detectors, a timer is restarted upon receiving a heartbeat. The deficiency in this scheme is that the probability of timing out on a message depends not only on the delay of that message, but also on the delay of the previous message. Chen *et al.* [6] propose a modified heartbeat method, called the "freshness points" scheme. In freshness points scheme, the monitored process p sends heartbeat message m_i at time $\sigma_i's$, separated by fixed interval η; and the detecting process q will, at fixed freshness points $\tau_i = \sigma_i + \delta$, check if it has received m_j with $j \geq i$. If it has not, it suspects the monitored process. We will borrow the "freshness points" scheme in our paper. This scheme requires synchronized clocks, which is not unreasonable. For example, GPS could provide highly precise clocks [15]. When synchronized clock is not assumed, Chen *et al.* [6] show how past message arrival times could be used to estimate the freshness points. Chen *et al.* also conduct the first quantitative study on QoS of failure detectors in the same paper [6]. Their work is restricted to pair-wise failure detectors.

The first work on *distributed* failure detectors is due to Gupta, *et al.* [9]. They propose a randomized distributed failure detector with low total message load on the system. They also take advantage of spatial multiple timeouts as a tool to increase the accuracy of the failure detector. This paper differs from [9] in the following aspects: this research develops a low cost protocol especially tailored for asymmetric configurations while [9] targets a symmetric configuration; our algorithm is deterministic and guarantee s a strict upper bound on detection time of failure while the algorithm in [9] is randomized and guarantee s a upper bound only on the expected detection time.

The term *fast failure detector* (FastFD) was introduced in [10], referring to failure detectors build upon expedited messages. Our algorithm does not need special expedited messaging support.

3 System Model

We consider a group of processes, Π, of size n, collaborating over a wide area network. Each member in the group is uniquely identified by its *id*. The knowledge of all member ids are known to each node *a priori*. This assumption is convenient for analysis but does not limit the scope of the application of our algorithm only to static configurations. In practice, dynamic membership could be supported by invoking a distributed consensus on the new member composition, as done in Paxos [11].

In our network model, a message is either lost with probability p_L or delivered within t_{lat}. We do not model the message latency distribution. We will show in Section 6.1 that our simple model suffices.

Since we borrow the "freshness point" method in heartbeating from [6], we make the same assumption that the clocks are synchronized. Though not an impractical assumption by itself, we could always use the same methodology as in [6] when this assumption is not met.

The system imposes a constraint on the smallest possible interval between two consecutive heartbeat messages, t_{itv}, usually on the order of seconds or tens of seconds. Setting t_{itv} too low is impractical because first it will take up too much processing power and network bandwidth; and second, temporally close messages tend to be highly correlated, providing little new information. Heartbeat messages separated temporally more than t_{itv} are assumed to be independent.

The typical message latency t_{lat}, which is usually on the order of tens or hundreds of milliseconds, is much smaller than t_{itv} ($t_{lat} \ll t_{itv}$). If a message is delayed for too long, it will be effectively considered lost by the freshness point scheme at the receiving process. The message round trip time, t_{rtt}, equals $2 * t_{lat}$.

Following the freshness point scheme, the monitored process sends one heartbeat every t_{itv} ($\sigma_i = i * t_{itv}$). On the monitoring process, it defers the freshness point by a detection delay δ, which is to say $\tau_i = \sigma_i + \delta$. Under the point-to-point setting that one single process is monitoring another process, the length of δ determines how accurate the detection is. Roughly, if δ is as long as $m * t_{int} + t_{lat}$, it will accommodate m heartbeat delay or loss, and the probability that all of them lost is p_L^m. The smallest value δ can assume is t_{lat}, corresponding to $m = 0$. To have high accuracy in the point-to-point setting, m has to be fairly large. For example, if $p_L = 0.1$ and we want the detection accuracy be 0.99999, m has to be at least 5. When the value of δ is large, we will use another symbol Δ to denote it.

4 The Fast Failure Detection Algorithm

In distributed applications, quite often the protocol is based on a special leader process and a number of participants. The Paxos [11] algorithm described in Section 1 is one example of many such applications. In such applications, the current leader should be monitored so if fails, a new leader could be elected. If the detection is slow, then when the current leader fails, the system becomes either unavailable or appears slow to the clients; if the detection

keeps making mistakes, the system falls into an unstable state as it will keep electing new leaders.

In the algorithm, we call the process that is monitored the *watched process*, or simply the *watched*, and monitoring process *monitors*. In the static settings, the watched will always be the leader process. In the dynamic settings, the watched is set to be currently believed leader due to a leader election algorithm. Since the leader elections could make mistakes, the watched will be the same as the real leader only when the leader election algorithm stops making mistakes and stabilizes on one leader.

Inspired by the application requirement notion in [9], we define the following QoS metrics on FFD:

FAILURE DETECTION ACCURACY The probability of mistakenly detecting a non-faulty watched process as faulty is at most P_m.

FAILURE DETECTION SPEED The failure of the watched is detected, subjected to the accuracy requirement P_m, within $T_{detect}(P_m)$ seconds.

The algorithm is shown in two parts. Figure 1 lists function definitions, and the task `watched`. Figure 2 shows the main monitor task `monitor` and another transient task `collectVotes`.

Among the monitors, the one with smallest id is said to be the *primary monitor*. The watched sends heartbeats (`WATCHED-OK` messages) to all other processes (Line 26 of Figure 1). The heartbeat messages contain a sequence number and the *id*s of all processes it believes to be its current monitors. Upon receiving a heartbeat, a monitor will record the information of the current peer monitors (Line 15 of Figure 2). From this information, the primary monitor could be self determined when necessary (Line 6 of Figure 2).

Upon *the first timeout*, the primary monitor asks the rest of the monitors if they also timed out on the same heartbeat (Line 8 of Figure 2). If all responses are positive, it will suspect the current watched process and report to the upper layer about this suspicion and the number of timeouts it has collected (The `collectVotes` task starting at Line 29 of Figure 2).

The upper layer has the ultimate decision on whether to suspect the current leader and ask the leader election algorithm to elect a new leader. The number of timeouts (k) provides the information for the upper layer to calculate the (in)accuracy probability of the detection, for example, by using the equation $P_m = p_L^k$ since we assume independence of message loss. Notice the report of a leader failure is a suggestion, due to the probabilistic nature of the detection.

Any other deterministic function suffices.

If message loss is not independent, then a different equation which makes use of the correlation information between message loss could be employed if the correlation is known.

Some details are omitted in the algorithm for clarity of presentation. For example, when the primary monitor is collecting vote, a delayed vote for a previous ballot should not be confused as a new vote for this ballot. This could be easily accomplished by adding a nonce or timestamp to the messages.

We prove two propositions showing that to meet the same accuracy requirement, FFD is much faster in detecting leader failures, by comparing its lower bound on detection time with that of the traditional algorithm based on temporal multiple timeouts.

Proposition 4.1 *To bound inaccuracy to be below P_m, the traditional temporal multiple timeouts scheme takes a time of at least*

$$T = t_{itv} * \frac{log(P_m)}{log(p_L)}.$$

Proof Within T seconds, let m be the number of heartbeat messages sent. The probability that all of them are lost is p_L^m. According to the accuracy requirement, $p_L^m \leq P_m$, so $m \geq \frac{log(P_m)}{log(p_L)}$. Given that the heartbeats must be separated by t_{itv}, the lower bound on the time of detection is $t_{itv} * \frac{log(P_m)}{log(p_L)}$.

Proposition 4.2 *If the size of the correct processes $n > m + 1$, where $m = \lceil \frac{log(P_m)}{log(p_L)} \rceil$, then the new algorithm achieves P_m within $t_{itv} + t_{lat} + t_{rtt} \approx t_{itv}$ if no messages are lost.*

Proof The worst case is that the leader fails just after sending one heartbeat. The primary monitor detects this within time length $t_{itv} + t_{lat}$ and takes another t_{rtt} to collect votes. With the assumption that messages sent to different monitors are independent, to bound the inaccuracy to be below P_m, at least $m = \frac{log(P_m)}{log(p_L)}$ number of independent timeouts are needed. Under a failure-free run, a group size of $n > m + 1$ will provide the needed number of independent timeouts. Since $t_{lat} < t_{rtt} \ll t_{itv}$, $t_{itv} + t_{lat} + t_{rtt} \approx t_{itv}$.

We use an example to illustrate the speed-up of the new algorithm. Suppose that $P_m = 0.00001$, $t_{int} = 10s$, $p_L = 0.1$, and the group size is greater than 6. From Proposition 4.1, the time required for failure detection is at least 50 seconds for a detector based on temporal multiple timeouts. From Proposition 4.2, it takes only slightly more than 10 seconds for the new algorithm while achieving the same accuracy. Also notice that the accuracy of the new algorithm gets better when the group size grows.

5 Application of FFD to the leader election algorithms

This section shows how the FFD algorithm works together with a leader election algorithm, or Ω. To be precise,

```
   /* variables used by both the watched task and the monitor task                */
1  watched = ⊥ // set externally to be the currently trusted leader
   /* variables used by the watched task                                          */
2  V = Π // V is the current view of the group at the watched
   /* variables used by the monitor task                                          */
3  missed = false // Did I missed the latest heartbeat from the watched
4  monitoring=false // set after receiving first heartbeat from the watched
5  peerMonitors = {p} // my peer monitors
   /* function definitions                                                        */
7  function OnSetNewLeader ( newLeader )// called externally
8    if watched != newLeader then
9      abort collectVote // always safe to abort a task that is not running
10     watched = newLeader
11     missed = false
12     monitoring = false // till receivs the first WATCHED-OK
13     peerMonitors = {p}
14   end
15 end function
16 for each process p: costart watched, monitor
17 task watched
19   on every time instant τ'_i = Δ + i * t_itv
     /* use a big delay Δ to accommodate several MONITOR-OK message loss or
        delays from the monitors                                                 */
20     if p == watched then
21       Q = {q | no MONITOR-OK from q for heartbeat j ≥ i is received }
22       T = Π − Q // T is the set from which fresh heartbeat is received
23       V = (V ∪ T) − Q // update the current view on live processes
24     end
26   on every time instant σ_i = i * t_int
27     send m_i =(WATCHED-OK, p, i, V ) to all processes in V
28 endtask
```

Figure 1. The FFD algorithm, part 1: the shared routine and the *watched* task.

```
1  task monitor
2      on every time instant τ_i = t_lat + i * t_itv
3          if monitoring AND did not receive (WATCHED-OK, q, j, K) from watched with j ≥ i then
4              missed = true
6              if p is the smallest among peerMonitors then
8                  send (DID-YOU-MISS-TOO, p, watched) to all in peerMonitors
9                  start task collectVotes
10             end
11         else
               // received fresh WATCHED-OK message from the watched process
12             monitoring = true
13             missed = false
15             peerMonitors = K
16         end
17     on every time instant σ_i = i * t_itv
18         send (MONITOR-OK, p, i) to watched // Ping the watched so it can update its view
19     upon receive(DID-YOU-MISS-TOO, primaryMon, suspected)
20         if monitoring AND watched == suspected then
21             if missed then
22                 send (I-MISSED-TOO, p, suspected) to primaryMon
23             else
24                 send (I-DIDNOT-MISS, p, suspected) to primaryMon
25             end
26         end
27 endtask
29 task collectVotes
30     /tcpan ephemeral task with no loops miss-count = 1
31     restart timer t_1
32     upon (t_1 > 2 * t_rtt ) // wait long enough s.t.  normal replies will arrive
34         Report (SUSPECT, p, watched, miss-count)
35     upon receive (I-MISSED-TOO, voter, suspected)
36         if suspected == watched then
37             miss-count = miss-count + 1
38         end
39     upon receive(I-DIDNOT-MISS,voter, suspected) // dismiss the suspicion
40         if suspected == watched then
41             missed = false
42             abort collectVotes
43         end
44 endtask
```

Figure 2. The FFD algorithm, part 2: the *monitor* task and the *collectVotes* task

under the asynchronous model, no Ω implementation could maintain the semantics of eventual correctness. For convenience of presentation, and with a slight abuse of terminology, we will still call such implementations Ω, knowing that it no longer has the eventual correctness semantics.

We first show an example of an existing leader election algorithm in Figure 3, then we show how to modify it to work under the asynchronous system model and how FFD works together with it.

This following leader election algorithm in Figure 3 is taken from [1] by Aguilera *et al.* It is a leader election algorithm, or in other words, an implementation of Ω. The synchrony requirement in [1] is even weaker than the partial synchrony model proposed by Chandra and Toueg. The synchrony requirements in [1] are that there is one process whose outgoing channels are eventually timely, with message delay bound α. All other channels could be lossy or untimely. As we pointed out before, the bound α, like any bounds on the message delay in a partial synchrony model, is usually big. The timeout value is 2α because the temporal gap between two consecutive heartbeats could be as long as 2α. For a formal correctness proof, please see [1].

Under the asynchronous and probabilistic network model, we have to modify the leader election algorithm. First of all, the timeout (in Line 26, Figure 3) will no longer be accurate as there will no bound on the message delay. The timeout could now be due to message loss or extended delay, both of which captured by the p_L in our model.

We modify the leader election algorithm to work together with FFD. To be consistent, we also modify it to use the freshness point heartbeating scheme. The modified algorithm is shown in Figure 4. The leader/candidate sends one heartbeat on every t_{itv}, at time points $\sigma_i = i * t_{itv}$. To increase the accuracy the detection, the detection freshness point is delayed from σ_i extendedly, i.e., $\tau_i = \sigma_i + \Delta$. As before, $\Delta = m * t_{itv} + t_{lat}$, where m is the number of heartbeat delay or loss the detector want to accommodate. Also, since the detection is distributed, any process could (mistakenly) timeout the current leader and advance to the next round, effectively demoting it. The bigger the group size is, the more likely *some* member will mistakenly timeout the current leader. So we have to increase the detection timer even more to counter this distributed "amplification" effect.

Besides adopting the freshness points detection scheme, a few more changes are needed. First, the leader should be also suspected when the FFD algorithm suspects it (Line 29 in Figure 4). Second, when a new leader is elected, the FFD should be notified about this change (Line 8 and Line 25 in Figure 4).

Under the asynchronous network model, the leader election algorithm, or Ω, could make mistakes at any time. The correctness of it could only be described in probabilistic fashion. When all the channels become very untimely such that the timeouts are often premature, there is little guarantee from the algorithm. Fortunately, many real world systems usually alternate between long "stable" periods, when the channels are timely, and short "unstable" periods, when some channels are not timely [7]. So we will consider the typical situation, that is when the system is stable and has a correct leader. Under this situation, when the leader fails, the FFD algorithm detects the failure much faster, compared to the traditional temporal multiple timeout detection. When FFD detects the failure, it prompts the leader election algorithm to elect a new leader. So overall, a new leader is elected much faster than the traditional leader election algorithm.

When under the same synchrony assumptions are made as in [1] for the algorithm in Figure 3, the correctness proof of the modified algorithm in Figure 4 is essentially the same as in [1].

6 Discussion

This section discusses the issues connected with the two assumptions we make, the adoption of a simplified network model where message latency is not modeled in detail, and the assumption on the independence of heartbeats sent to different monitors.

6.1 Justification for our simplified network model

In our network model, a message is either lost with probability p_L or delivered within t_{lat}. Alternatively, one could use a more detailed model [6] where a message is either lost with probability p_L or takes a random length of time to arrive, such that the distribution of the latency is known or at least its mean and variance are known.

Given a set of prescribed requirements on the failure detector, the knowledge on the distribution of message latency would appear to be helpful in calculating the parameters such as heartbeat interval (η) and detection delay (δ). However, it is not obvious how helpful this information would be. We use an example to demonstrate that the results calculated from the simplified model and the detailed model are essentially the same, so a detailed model is not necessary.

Let us look at one example based on one in [6]. Given the requirements on the detection speed of 30s, and the requirements on the other two QoS metrics defined in [6], and under the assumption that message loss probability $p_L = 0.01$ and that message delay is exponentially distributed with

The set of metrics defined in [6] is different from those in this paper. They are detection speed, mistake duration bound, and mistake recurrence lower bound in [6].

```
1  Code for each process p :
2  function StartRound ( s ) // executed upon start of new round
3    if  p != s mod n then
4        send (START, s) to all // bring all to new round
5        r = s // update current round
6        leader = ⊥ // demote previous leader but do not elect leader yet
7        restart timer
8    end
9  on initialization:
10 StartRound (0)
11 costart 0 and 1
12 task 0
13     loop forever /* the leader/candidate sends OK every α time                    */
14     if p == r mod n AND have not sent (OK, r) within α then
15         send (OK, r) to all
16     end
17 endtask
18 task 1
19     upon receive (OK, k) with k = r do // current leader/candidate is active
20       if leader = ⊥ AND received at least two (OK, k) messages then
22           leader = k mode n // now elect leader
23           restart timer
24       end
26       upon timer > 2α do // timeout on current leader/candidate
27         StartRound (r +1) // start next round
28       upon receive (OK,k) or (START , k) with k > r do
29         StartRound (k) // jump to round k
30       upon receive (OK,k) or (START , k) from q with k < r do
31         send(START,r) to q // update process in old round
32 endtask
```

Figure 3. The stable leader election algorithm by Aguilera *et al.* [1]. (Figure 3 in [1])

```
1  Code for each process p :
2  function StartRound ( s ) // executed upon start of new round
3    if p != s mod n then
4      send (START, s) to all // bring all to new round
5      r = s // update current round
6      leader = ⊥ // demote previous leader but do not elect leader yet
8      Invoke OnSetNewLeader( ⊥) in FFD
9    end
10 on initialization:
11 StartRound (0)
12 costart 0 and 1
13 task 0
14   on every time instant σ_i = i * t_int
15     if p == r mod n then
16       send m_i = (OK, i, r) to all
17     end
18 endtask
19 task 1
20   on every time instant τ_i = Δ + i * t_itv
     /* use a big delay Δ to accommodate several OK message loss or delays from
        the leader/candidate                                                    */
21     if has received (OK, j, k) with k = r and j >= i then
         // current leader/candidate is active
22       if leader = ⊥ then
23         leader = k mode n // now elect leader
25         Invoke OnSetNewLeader( leader ) in FFD
26       end
27     end
29   upon the call of Report(SUSPECT, p, q, miss-count) AND q = leader AND miss-count is big enough. // the
       FFD algorithm reports a suspicion
30     StartRound (r +1)
31   upon receive (OK,j, k) or (START , k) with k > r do
32     StartRound (k) // jump to round k
33   upon receive (OK,j,k) or (START , k) from q with k < r do
34     send(START,r) to q // update process in old round
35 endtask
```

Figure 4. The modified leader election algorithm.

mean 0.02s, the parameters calculated in [6] are: heartbeats sent at 9.97s interval (η) and the detection delay (δ) is 20.03s.

To evaluate the consequence of our simplified model, we extend the analysis of [6] to show that similar results are achieved under our simplified model. Under our simplified model, we take the same message loss probability $p_L = 0.01$, to satisfy the same set of requirements, we calculate the heartbeats interval as 9.9985s and detection delay as 20.0015s, which are for all practical purposes the same as those calculated from the more complex model.

Based on this typical example, we believe that our simplified model is adequate. In general, the relationship between the more complex model and our simple model is as following. The knowledge of the distribution in the complex model could be used to calculate the probability that a message arrive later than T_{lat}. All messages arrive later than T_{lat} is considered lost and captured by the parameter p_L.

6.2 Discussion on the independence of heartbeats to different monitors

We made a key assumption that if two heartbeat messages sent from the same server to two different monitors are lost, then two things could have happened. The first possibility is that a failure occurs at the server (server crash) or at the local network where the server resides (network cable unplugged or local router down). In this case, no participant will receive the heartbeat. According our algorithm, the server will be detected as failed and that is exactly what we want. The second possibility is that the failure is not local to the server. In this case, we assumed that the loss of heartbeats is independent. This assumption is reasonable if two messages take different network paths. We believe that if the two monitors are widely dispersed, the two messages will take two different paths. Large DHT networks built over the Internet provides an ideal platform our protocol is suited for because in DHT systems, two randomly selected DHT ids have the property of being widely separated.

When the message paths are not entirely independent, we could extend our model to include the correlations between them. The correlation coefficients should be rather stable if the paths taken are stable. Thus, these coefficients are likely to be measurable.

7 Conclusion

This paper proposes a Fast Failure Detection (FFD) algorithm using spatial multiple timeouts in a process group. FFD detects the failure of the leader with high accuracy upon missing the first heartbeat, the best we can do with failure detection based on heartbeats. FFD could be used alone in a static setting where the leader process is fixed or manually reconfigured. FFD could also be used to improve the performance of leader election algorithms in a dynamic setting where different processes could assume the role of the leader. All distributed applications based on a single leader, such as Paxos, could benefit from the FFD to improve their performance.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *DISC*, pages 108–122, 2001.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC*, pages 306–314, 2003.

[3] R. Boichat, P. Dutta, S. Frlund, and R. Guerraoui. Deconstructing paxos. *SIGACT News*, (1):47–67, 2003.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *PODC '92: Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pages 147–158, New York, NY, USA, 1992. ACM Press.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, 2002.

[7] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[8] M. Fischer, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM(JACM)*, pages 374–382, 1985.

[9] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, page 170.

[10] J.-F. Hermant and G. L. Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Trans. Comput.*, 51(8):931–944, 2002.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[12] L. Lamport. Paxos made simple. *SIGACT News*, (4):18–25, 2001.

[13] M. Larrea, A. Fernandez, and S. Arevalo. Optimal implementation of the weakest failure detector for solving consensus. *srds*, 00:52, 2000.

[14] V. Paxson, A. Adams, and M. Mathis. Experiences with nimi. In *First Passive and Active Measurement Workshop*, 2000.

[15] P. Verssimo and M. Raynal. Time, clocks and temporal order. *Recent Advances in Distributed Systems, Chapter 1*, LNCS 1752, Springer-Verlag, 2000.