A Novel Failure Detection Algorithm for Reliable Distributed Systems

Yunni Xia, Chuanjian Jiang, Tianhao Sun, Ruilong Yang
School of computer Chongqing University Chongqing, China, 40030
Email: xiayunni@yahoo.com.cn

Abstract—A failure detection service is perfect if it eventually detects all failures and every detection correctly identifies a failure that has occurred. Such a perfect failure detection service serves as a basic building block for many reliable distributed systems, for example in distributed lock services. In this paper, we introduce a perfect failure detection scheme in order to improve the fault tolerance of the service. We provide the precise system model and specification for a failure detection service. We present two novel algorithms that implement the failure detection service. We further develop a set of quality-of-service (QoS) metrics for perfect failure detection services, and apply probabilistic analysis to quantify the QoS metrics of the two algorithms.

Index Terms—failure detection; distributed system; quality of service

I. INTRODUCTION

Failure detectors have already been shown to be theoretically important ([4]), and they can be found in many reliable distributed systems. Leader election in Paxos [10] implicitly relies on failure detection for replacing failed leaders. In any primary/backup replication protocol (e.g., the ones used in Harp [11], Chain Replication [14], and Boxwood [12]), failure detectors are used to trigger fail-over of the primary. In distributed lock services (e.g., the ones in Boxwood [12] and Chubby [3]), failure detectors that are based on leases [7] are used to detect the failures of the processes that are holding locks so that the locks can be reclaimed by other processes waiting for the locks.

For both primary/backup replication and distributed lock service, mutual exclusion properties are required: in the former, there cannot be two primaries active at the same time. In the latter, there cannot be two processes holding the same lock at the same time. Such mutual exclusion properties can be achieved through the use of a perfect failure detector [4, 6], the key properties of which are (a) all failures are eventually detected, and (b) if the failure detector considers a process p faulty, then p must have failed.

Implementing a perfect failure detector in an asynchronous system is impossible. However, with the assumption of bounded clock drift, an often reasonable assumption in practice, and a suicide mechanism, where a process times out and transitions into a faulty state voluntarily, perfect failure detector becomes possible. Such a suicide mechanism was used in ISIS [2] and advocated by Fetzer [6].

In this paper, we advocate the use of a perfect failure detection service. With such a service, no ad-hoc monitoring or failure detection needs to be implemented (or duplicated) any more. A system-wide monitoring service with strong properties is instead used.

A service in a fault-tolerant distributed system must itself be fault-tolerant. This is especially important for a perfect failure detection service: due to the use of the suicide mechanism, all processes in the system would commit suicide if the service were to fail. We therefore study a perfect failure detection service that employs a set of processes. Although Boxwood uses such a service, no satisfactory details are given on this particular component. The case is not covered well by Fetzer [6] either. Chubby [3] uses a replicated master group for failure detection, which means it needs a separate and more complicated replication protocol and its failure detection guarantees depend on the timing assumptions on the synchronization and fail-overs within the master group.

In this paper, we provide a rigorous study on several important aspects concerning a perfect failure detection service. We first provide a precise system model in asynchronous message-passing systems in which a number of dedicated server processes, called detectors, monitor a process q and service queries from client q. The detectors may crash and recover, which match the practical situation for a long-lived service. We then present a specification of the perfect failure detection service, which includes important properties that eliminate trivial and useless implementations. We next provide two new algorithms that have complementary characteristics. One algorithm is based on leases while the other is based on shared register abstraction.

Furthermore, we study the quality of service (QoS) of perfect failure detection services. The QoS of failure detectors was originally proposed and studied for heartbeat style failure detectors in [5]. In this paper, we propose a set of QoS metrics that cover important aspects of a perfect failure detection service, such as how well the service is in avoiding suicide, how fast it is in detecting failures, and how fast it is in responding to client queries. We then use probabilistic method to analyze the QoS metrics of the two algorithms.
II. SYSTEM MODEL

We consider message-passing systems composed of three type of processes. The first type is processes being monitored. In this paper, we only need one representative process \( p \) of this type. The second type is server processes that monitor the status of \( p \). We call these processes detectors, and use set \( X = \{ x_1, x_2, \ldots, x_d \} \) to denote the set of \( n \) detectors. The third type is client processes that query detectors to check the status of \( p \). We only need to consider one client process \( q \) in our study. Our results can be easily generalized to the case of multiple clients and multiple processes being monitored.

Processes \( p \) and \( q \) communicate with the detectors by sending and receiving messages over asynchronous bidirectional links. The links cannot create or duplicate messages, but they may delay or drop messages. Message delay is always greater than zero and may be arbitrarily large. Detectors do not communicate among themselves, nor do \( p \) and \( q \) communicate with each other.

Each process is equipped with a local clock, which can be used to read clock values and set timers. Local clocks are drift-free, but they are not necessarily synchronized. After a timer is set with a certain time interval \( T \), it will expire exactly \( T \) time units after setting the timer, unless the timer is cancelled or reset to a different interval. In reasoning about process behaviors, we often refer to the global time, which is continuous and cannot be accessed by processes. Each process executes a sequence of steps, which are triggered by message receptions, timer expirations, or interface function invocations. In each step, a process may change its local state, send messages and operate on timers. For simplicity, we assume that the time to complete one step is zero. An detector only takes reactive steps. That is, an detector \( x \) only sends out a message to process \( y \) (being \( p \) or \( q \) ) in \( x \)'s step that processes a message received from \( x \). Thus, we call a message sent by an detector a response to the message it receives. An detector will not send a message in a step triggered by a timer expiration event.

Process \( p \) may crash involuntarily at any time or it may commit suicide voluntarily in one of its steps by invoking a special \textit{suicide()} interface function. We say that \( p \) fails if it crashes or commits suicide. After \( p \) fails, it does not take any more steps. The \textit{suicide()} interface may have several ways to be implemented in systems, such as using a hardware watchdog to guarantee immediate termination [6].

Each Detector may crash at any time and later recover. An detector may store (part of) its state into stable storage. The value stored in the stable storage will not be lost after a crash and recovery. The local clock value is also not affected by crash and recovery. Except for the local clock value and the value stored in the stable storage, an detector loses all other state values after a crash and recovery. In particular, if an detector starts a timer and then crashes, it loses the state about this timer. We assume that client \( q \) does not crash, since if it crashes in the middle of a query, the query automatically fails and we do not enforce any requirement on \( q \)'s query if \( q \) fails.

Our specification and algorithms are based on the status detection systems. In our setting, a detection system consists of two non-empty sets \( D_p \) and \( D_q \) of subsets of detectors \( X \) such that for all \( d_1 \in D_p \) and all \( d_2 \in D_q \), \( d_1 \) and \( d_2 \) intersect. \( S_A \) is a survival amount and \( QA \) is a query amount. Informally, \( p \)'s survival depends on \( p \)'s having at least \( S_A \) timely communications, while \( q \)'s success in querying \( p \)'s status depends on \( q \)'s having at least \( QA \) reliable communications.

We now make it formal the meaning of a process being able to communicate with \( S_A \) detectors. The definitions follow the similar ones in [13]. Let \( r(y,x) \) denote the round-trip channel from a process \( y \) to an detector \( x \). We say that \( r(y,x) \) is reliable at time \( t \) if for any message \( m \) that \( y \) would send to \( x \) at time \( t \), if \( x \) would send a response to \( y \) for \( m \), then \( y \) would eventually receive the response from \( x \). Note that it is a property of the system requiring all the following conditions to hold: (a) the communication link from \( y \) to \( x \) would not drop the message sent by \( y \) to \( x \) at time \( t \), (b) \( x \) would process the message and would not crash during the processing, and (c) \( x \)'s response (if any) would not be dropped by the link from \( x \) to \( y \). It does not depend on whether or not \( x \) and \( y \) actually send messages.

Process \( p \) (resp. \( q \) ) is said to be reliable at time \( t \) if there is at least \( S_A \) detectors such that for every included detector \( x \) the round-trip channel \( r(p,x) \) (resp. \( r(q,x) \) ) is reliable at time \( t \). We say that process \( y \) (being \( p \) or \( q \) ) is fairly reliable if given any time sequence \( (t_1, t_2, \ldots) \) that tends to infinity, there exists time \( t_x \) at which \( y \) is reliable. Intuitively, if \( y \) tries to communicate with the detectors infinitely often, then it should be successful at least once.

III. SPECIFICATION OF PERFECT FAILURE DETECTION

In our model, failure detection is provided as a service by the detectors, which are constantly monitoring the status of process \( p \). To retrieve the status of \( p \), client \( q \) invokes a query interface \textit{check()}, which communicates with the detectors and returns either \textit{Alive} or \textit{Dead} status of \( p \) to \( q \). Client \( q \) may enter the system and invoke \textit{check()} at any time and may leave the system after the \textit{check()} returns. This flexible query model and the failure-detection-as-a-service architecture differ from the original failure detector abstraction [4], where processes are assumed to always be part of the system monitoring each other. As a result, the specification of our failure detection service covers aspects not in [4]. The specification includes a number of properties. The first two properties correspond to the two properties of perfect failure detector defined in [4]:

**Strong Completeness:** If process \( p \) crashes or commits suicide, then there is a time after which if client \( q \) invokes \textit{check()} and \textit{check()} returns, the return value must be \textit{Dead}.

**Strong Accuracy:** If client \( q \) invokes \textit{check()}, which returns \textit{Dead} at time \( t \) to \( q \), then \( p \) must have crashed or committed suicide before time \( t \).

The following property is the weaker version requiring that \textit{check()} should terminate if \( q \) is able to achieve at
least one round of timely communication with at least $SA$ detectors as long as it tries infinitely often.

**Weak Query Termination:** If client $q$ is fairly timely, then every $check()$ invoked by $q$ eventually returns.

We also give the stronger version in which communication between $q$ and the detectors are not required to be timely:

**Strong Query Termination:** If client $q$ is fairly reliable, then every $check()$ invoked by $q$ eventually returns.

We say that a failure detector service is weakly (resp. strongly) perfect if it satisfies Strong Completeness, Strong Accuracy, Integrity, and Weak (resp. Strong) Query Termination properties. Finally, we address the bootstrap issue of how detectors and clients learn about $p$ so that they can monitor and query the status of $p$. We assume that when $p$ starts running, it must first complete one round of communication with at least $SA$ detectors, and then it can register its existence (to some directory service). Only after its registration, other clients may start query its status by invoking $check()$. This bootstrap requirement matches practical situations.

**IV. ALGORITHMS OF PERFECT FAILURE DETECTION**

In this section, we consider two algorithms, the first of which implements a weakly perfect failure detection service while the second of which implements a strongly perfect failure detection service. We then compare the two algorithms and show that they have complementary features.

In our pseudocode, we use several primitives to represent operations related to clocks and timers. In particular, $getClockTime()$ primitive is for a process to get its current clock value; $setTimer(t, \delta)$ is for a process to set or reset its timer $t$ to be expired at $\delta$ time units later from the current time; $clearTimer(t)$ is to clear timer $t$; and $expireTimer(t)$ is triggered when timer $t$ expires.

We use the term stable variable to represent the use of storage on the detectors. For a stable variable $v$, every write to $v$ completes only after the value is written to both the memory and the stable storage. When a detector recovers from a crash failure, it loads the value of $v$ in the stable storage back to memory, and every read of $v$ afterwards are from memory directly.

The correctness proofs of the two algorithms are not very difficult and are omitted due to space constraint.

**A. Algorithm I: Lease-based**

The first algorithm is based on the lease mechanism [7]. In the basic lease-based failure detector with two processes $p$ and $q$ (as illustrated in Figure 1(a)), $p$ periodically requests a lease from $q$. The lease period on process $p$ starts when $p$ sends a lease request to $q$, while the lease period on process $q$ starts when $q$ receives the request from $p$. The lease periods on $p$ and $q$ have the same length, and thus $q$’s lease period always ends later than the corresponding lease period on $p$. Before one lease period expires on $p$, $p$ has to receive a response from $q$ that grants $p$ a new lease period. If $p$ does not receive this response in time, its lease expires and for the purpose of perfect failure detection, it needs to commit suicide. If $q$ does not receive a new request before its lease period ends, it starts to declare $p$ as dead. Since $q$’s lease period ends later, $q$’s declaration of $p$ being dead always comes after $p$ crashes or commits suicide. The basic failure detector is easily extend to a perfect failure detection service with a single detector $x$. We use $x$ to replace $q$ so that detector $x$ correctly detects the status of $p$, and client $q$ only communicates with $x$ to query the status of $p$.

However, extending the single detector case to the multiple detectors case is not so straightforward. A naive extension is that each detector $x_i$ simply behaves as the single detector, and $p$ needs to receive responses from at least $SA$ detectors in order to extend its lease, while $q$ needs to collect $p$’s status from at least $SA$ detectors to derive the status of $p$. If all detectors in the $SA$ detectors believe that $p$ is dead, then $q$ declares $p$ dead; otherwise as long as one detector in the $SA$ detectors believes that $p$ is alive, $q$ declares $p$ alive. Figure 1(b) illustrates a problematic scenario of this naive extension, where we have three detectors and any two detectors forma quorum. In this scenario, detector $x_2$ misses the second message from $p$ and thus after its first lease period ends it starts to declare $p$ as dead, but $p$ receives responses from $x_1$ and $x_3$ so $p$ successfully renewed its lease. Later when $x_2$ receives the third message from $p$, it starts a new lease period and declares $p$ alive again, but $x_3$ misses the third message and thus declare $p$ dead after the second lease period ends. If $q$ obtains $p$’s status from $x_2$ at point $A$ and from $x_3$ at point $B$, both status will say $p$ dead and thus $q$ will declare $p$ dead. But $p$ is still alive since it always receives two responses in time. Therefore, Strong Accuracy property is violated.
On process $p$ (to be monitored):

1. **Variable:**
   - `counter`: initially 0
2. repeat every $\eta$ time units:
   - `counter = counter + 1`
   - send (LEASE-REQUEST, `counter`) to all detectors
   - `setTimer(time[`counter`], $\delta_p + \eta \cdot \delta_Q`)
3. Upon expiry of timer(`i`):
   - if not received (LEASE-GRANT, `j`) with $j \geq i + 1$ from at least $5\delta$ detectors then suicide()

**On detector $x$:**

1. **Variable:**
   - `latest`: stable variable, the latest lease request number received from $p$, initially 0
   - `deadline`: stable variable, the ending time of $x$'s current lease, initially 0
2. On receipt of (LEASE-REQUEST, `entry`) from $p$:
   - if `entry` = `latest` then
     - `latest = entry`
     - `deadline = getClockTime() + \delta_o / (\delta_o + \delta_p + A^\eta)`
     - send (LEASE-GRANT, `entry`) to $p$
   - On receipt of (CHECK, `ts`) from client $q$:
     - if `getClockTime()` = `deadline` then
       - send (CHECK-ACK, `ts`, `latest`, Alive) to $q$
     - else send (CHECK-ACK, `ts`, `latest`, Dead) to $q$
3. **On client $q$:**
   - `check()`:
     - repeat
     - `ts = getClockTime()`
     - send (CHECK, `ts`) to all detectors
     - `wait until` one of the following conditions holds:
       - (a) received (CHECK-ACK, `ts`, `entry`, $\sigma$) from at least $5\delta$ `QA` detectors;
       - (b) $\delta_o - \delta_q$ time elapsed;
     - **until** (a) is true
     - `cnt1 =` the largest `cnt` received in (CHECK-ACK, `ts`, `entry`, Dead) or 0
     - `cnt2 =` the largest `cnt` received in (CHECK-ACK, `ts`, `entry`, Alive) or 0
     - if `cnt1 < cnt2` return Dead else return Alive

Figure 2. Failure Detection Algorithm I: lease-based

If we use the same length for lease period on $p$ and on detectors, the two points $A$ and $B$ could be made arbitrarily close in time, so we have no chance for $q$ to avoid this scenario. To fix this problem, we let detectors use a longer lease period. Let $\delta_q$ be the length of the lease period on $p$ and $\delta_p$ be the length of the lease period on $q$. We require that $\delta_q \geq \delta_p + \Delta$, where $\Delta$ is the timeliness parameter defined in Section 2. With this setting, we guarantee that points $A$ and $B$ be at least $\delta_q - \delta_p$ time units apart. Then we require that $q$ has to collect at least $QA$ responses within $\delta_q - \delta_p$ time units. If it fails to do so, it has to resend a new set of messages to detectors to collect responses again.

With the above restriction, we eliminate the scenario depicted in Figure 1(b). The downside is that $q$ may need multiple rounds of communication to complete its query. If $q$ is fairly timely, then eventually $q$ will have a communication round that completes within $\delta_q - \delta_p \geq \Delta$.

Figure 2 provides the complete pseudocode for the lease-based algorithm. Process $p$ periodically sends LEASE-REQUEST messages to all detectors (lines 3–6), and when the current lease expires, it can continue running only if it receives LEASE-GRANT messages for higher-numbered lease periods from at least $SA$ detectors (line 8).

Each detector $x$ maintains two stable variables `latest` and `deadline`: Variable `latest` keeps the latest lease number and `deadline` keeps the ending time of the current lease. Detector $x$ simply responds $p$ with a LEASE-GRANT message, and responds $q$ with its latest lease number and $p$'s status based on whether the latest lease has expired or not (lines 12–20).

Client $q$ continues sending CHECK messages to the detectors until it receives responses from at least $QA$ detectors within $\delta_q - \delta_p$ time units (lines 22–26). The computation of the final return value of `check()` is a little more sophisticated than described above (lines 27–29). Instead of returning Alive as long as one detector returns Alive, the algorithm returns Alive if and only if all responses with the highest lease number indicate that $p$ is alive. This allows $q$ to detect the failure of $p$ earlier.

As indicated already, this algorithm works if $q$ is fairly timely. Therefore, it implements a weakly perfect failure detection service. The algorithm, however, is not strongly perfect, because if $q$ cannot obtain at least $QA$ timely responses, `check()` will not terminate. Our second algorithm fixes this problem.

**B. Algorithm II: Register-based**

The second algorithm (Figure 3) takes a different approach to implement a perfect failure detection service. The basic idea is for $p$ to write increasing counter values into the detectors to indicate that it is alive, and $q$ reads out these values. If $q$ cannot read higher values after a significant amount of time, then $q$ can declare $p$ dead.

Since it resembles the use of a shared read-write register [8, 9], we call this algorithm register-based.

More specifically, $p$ increments its counter variable every $\eta$ time units and writes the value of counter into the detectors to indicate that it is alive, and $q$ reads out these values. If $q$ cannot read higher values after a significant amount of time, then $q$ can declare $p$ dead.

With this read() interface, when $q$ invokes `check()`, it issues a sequence of `read()` calls, each of which is
separated $\delta_p$ time units after the previous one returns. This separation period ensures that if $p$ is still alive, then each $\text{read}(i)$ must return a value higher than the previously read value. Therefore, if $q$ sees that any $\text{read}(i)$ does not return a higher value, it returns Dead. Otherwise, it continues until it completes a total number of $[\delta_p/q]+2$ $\text{read}(i)$’s, then it returns Alive. We need multiple $\text{read}(i)$’s because if $p$ is dead, its last few counter values may not reach $S_A$, and thus it takes several reads to exhaust all these possible values and discover a non-increasing value. The algorithm is proven to implement a strongly perfect failure detection service.

C. Comparison between two algorithms

The above two algorithms have some complementary features that worth a comparison here. The lease-based algorithm has the drawback that the $\text{check}(i)$ may not terminate if $q$ cannot have timely communication with the detectors (i.e., $q$ is not fairly timely). This may not be an issue if the system is symmetric, that is, $q$ is also being monitored by other processes. In this case, if $q$ has no timely communication with the detectors, $q$ will commit suicide itself, so the termination of $\text{check}(i)$ may not be an issue any more. But in general, comparing with the register-based algorithm, the lease-based algorithm does have stronger requirements for query termination.

Moreover, for the lease-based algorithm if the last $\text{LEASE-REQUEST}$ message that $p$ sends out before $p$ crashes has a long delay, we may have a situation in which the first $\text{check}(i)$ already returns Dead, but the second $\text{check}(i)$ still returns Alive, and the time after which $\text{check}(i)$ always return Dead depends on the delay of $p$’s messages. If $q$ issues $\text{check}(i)$ periodically and remembers the query results, this is not an issue, but if $q$ only issues ad-hoc queries and do not remember query results (or query results are lost due to failures) then it may take time for $q$ to detect $p$’s failure. The register-based algorithm is better in this regard, because it guarantees that $\text{check}(i)$ always return Dead as long as it is issued after $p$ crashes. So the detection latency is not affected by the delay of $p$’s messages. Of course, it still depends on the delay of messages between $q$ and the detectors.

The register-based algorithm also has its drawback. When $p$ is alive, each $\text{check}(i)$ requires a number of $\text{read}(i)$ calls and each of them has to be separated by $\delta_p$ time units, making the response time of $\text{check}(i)$ quite long. The lease-based algorithm, on the other hand, can complete $\text{check}(i)$ in one round of communication if it is timely. If $q$ issues $\text{check}(i)$ periodically and can keep its local state, then each $\text{check}(i)$ only needs to call $\text{read}(i)$ once in the register-based algorithm and the read cost is amortized among multiple $\text{check}(i)$’s. However, in general the register-based algorithm depends on multiple rounds of communication with the detectors to determine $p$’s status while in the lease-based algorithm each round of communication is independent.

V. QUALITY OF SERVICE ANALYSIS

In the previous section, we only provide qualitative assessment to the different features of the algorithms. Applications may want to quantify these features and know how to tune system parameters to balance different aspects of a failure detection service. In this section, we address how to quantify the quality of a failure detection service using probabilistic analysis. Quality of service (QoS) of heartbeat-style failure detectors has been studied in [5]. In this section, we provide a matching study on the QoS of our perfect failure detection service. In particular, we first define a set of QoS metrics for such a service, and then provide a probabilistic analysis to the two algorithms described in the previous section.

We may optimize the algorithm such that when $q$ sends the $\text{READ}$ message, it can piggyback the value $v$ it previously read into the $\text{READ}$ message, which essentially means that it writes $v$ back to the detectors. Furthermore, $q$ can also indicate that $v$ is outdated when it writes $v$, because after $\delta_p$ time units, there must be a value higher than $v$ already stored on the detectors if $p$ is still alive. Then when $q$ later reads a value $v'$, $v'$ must be the highest value among those that has not been marked as outdated. This optimization helps subsequent $\text{check}(i)$ to reduce the number of $\text{read}(i)$’s when $p$ is dead, but for the first $\text{check}(i)$ it may still need to go through all $[\delta_p/q]+2$ number of $\text{read}(i)$’s, so we do not include this optimization directly in the pseudocode.
A. Definitions of QoS metrics

The first metric captures how well the service does in preventing $p$ from committing suicide. We denote it as $TTS$, which stands for the time elapsed from the time $p$ starts sending its first message to the detectors to the time $p$ commits suicide, in runs in which $p$ does not crash. The second metric captures how fast the service detects failures. After $p$ fails at a time $t_0$, it may take a while for $q$’s $\text{check}()$ to return Dead. The delay in detection can be separated into two periods. The first period is for $\text{check}()$ to stabilize its return value to Dead while the second period is to complete $\text{check}()$ operation. More precisely, let $t_1$ be the earliest time $t \geq t_0$ such that if $q$ invokes $\text{check}()$ at time $t$ and if the $\text{check}()$ returns, it is guaranteed that the $\text{check}()$ returns Dead. Let $t_2 > t_1$ be the time at which the $\text{check}()$ invoked at time $t_1$ returns. Thus the first period from $t_0$ to $t_1$ represents the time it takes for the service to stabilize $\text{check}()$ and guarantee the detection of $p$’s failure, while the second period from $t_1$ to $t_2$ represents the time it takes for $\text{check}()$ to return a value after it stabilizes. The two periods together indicate how fast the service detects $p$’s failures. Therefore, we define our second metric to be the duration from time $t_0$ to time $t_2$, which we call time to guaranteed detection and denote as $TTGD$. The third and the last metric captures the response time of $\text{check}()$ during the normal operation when $p$ is alive. More precisely, we define the metric to be the duration from a random time $t_1$ when $q$ invokes $\text{check}()$ to time $t_2$ when $\text{check}()$ returns in runs in which $p$ does not fail. We call this metric time to normal response and denote it as $TTNR$.

B. Qos Analysis

For QoS analysis, we consider that the delay of a message follows a general distribution given by function $F(x)$, where $F(x)$ is a monotonic function such that $F(x)=0$ when $x \leq 0$, $F(+ \infty) = 1$, and $F(\infty) > 0$. All messages follow the same delay distribution $F$ and the delays of different messages are independent. For simplicity, we do not consider message losses or detector failures. Message losses, and to some extent detector failures, can be masked by repeatedly sending the same messages until the response is received, and thus they are transformed into message delays. This probabilistic model matches with our definition of fair timeliness, since if process $p$ (or $q$) sends an infinite number of messages to all detectors, because of the fact that $F(\infty) > 0$, with probability one at least one message will generate timely responses from a quorum of detectors.

We study the following type of general quorum systems. A survival quorum is any subset of detectors with $t$ detectors, and a query quorum is any subset of detectors with $n-t+1$ detectors, where $t$ is a parameter ranging from 1 to $n$.

With the above settings, we evaluate the QoS metrics of the two algorithms given in the previous section, and see how different parameters affect the QoS of these algorithms. As an example, we focus on the effect of related parameters $n$ (the number of detectors) and $SA$ (the size of survival amount).

In the analysis, we frequently use order statistics [1], which is defined as follows. Given a set of $m$ independent and identically distributed random variables $X_1, X_2, \ldots, X_m$, we define $X^{(i)}$ to be the $i$-th order statistic of $\{X_1, X_2, \ldots, X_m\}$, which is the random variable that represents the $i$-th smallest value among $X_1, X_2, \ldots, X_m$, where $i$ could be 1, 2, . . ., or $m$. If $X$ has distribution function $G$, then we can derive the distribution function of $X^{(i)}$ as

$$\Pr\{X^{(i)} \leq z\} = \sum_{j=i}^{m} C_m^j \times G(z)^j \times (1-G(z))^{m-j}$$

To simplify the analysis, we only consider the case when $\eta + \Delta < \delta_p < 2\eta$, in which case $p$’s suicide condition at different times are independent and only depends on one set of messages $p$ sent. For other cases when $\delta_p > 2\eta$, $p$’s suicide condition at different times are not independent, and more complicated Markov modeling may be necessary, but it does not provide additional insight comparing to the simple case. C. Evaluation of TTS

The two algorithms have the same metric $TTS$, because $TTS$ only depends on the communication pattern between $p$ and the detectors and the suicide condition, which are exactly the same for both algorithms. The analysis of $TTS$ is provided below.

Let random variable $RTD$ denote the round-trip delay from $p$ sending a message to a detector $x_j$ to receiving the response to this message from $x_j$. Let $D_1$ and $D_2$ be two random variables with distribution function $F$. Then $\text{RTD} = D_1 + D_2$.

When the timer $\text{timer}[i]$ expires on $p$, $p$ commits suicide if and only if $p$ has not received responses to its $(i+1)$th message from more than $SA$ detectors. The duration from $p$ sending its $(i+1)$th message to $\text{timer}[i]$ expires is $\delta_p - \eta$. Therefore, the probability of $p$ committing suicide when $\text{timer}[i]$ expires can be given as $\Pr(\text{RTD}[i] > \delta_p - \eta)$, where $\text{RTD}[i]$ is the $i$-th order statistic of $\{\text{RTD}[1], \ldots, \text{RTD}[n]\}$.

Since at different timer expiration points, $p$ depends on different messages to decide if it commits suicide, the conditions that $p$ commits suicide at different times are independent of each other. Let $N_i$ be a random variable representing the number $i$ such that $p$ commits suicide when $\text{timer}[i]$ expires. Thus $N_i$ follows a geometric distribution with probability $ps$. Considering that a timer expires on $p$ every $\eta$ time units with the first expiration time $\delta_p$, we have

$$\text{TTS} = \eta \cdot (N - 1) + \delta_p$$

Then the expected value and the variance of $\text{TTS}$ are given as

$$E(\text{TTS}) = \eta(1/p_s - 1) + \delta_p$$

And

$$\text{Var}(\text{TTS}) = (1 - p_s)/p_s^2$$
For the computation of $E(TTS)$ and later metrics in the figures to be shown, we use an exponential distribution for message delay distribution $F$ with mean time delay to be 0.01 second. The values of other parameters are: $\eta=0.1s$, $\delta_p=0.15s$, and $\delta_o=0.2s$.

$$Pr[R \leq t] = Pr[RTD^{(p+1)} \leq t | RTD^{(p+1)} \leq \delta_o - \delta_p]$$

With $N_r$ and $R_r$, we know that the $TTNR^L$ for the lease-based algorithm is given as

$$TTNR^L = (N_r - 1)(\delta_o - \delta_p) + R_r$$

For the register-based algorithm, under the condition that $p$ is alive, a check() query must invoke a total of $\lceil \delta_p/\eta \rceil + 2$ read()'s. Each read() is completed when $q$ receives responses from at least $QA$ detectors, and between two consecutive read()'s there is a constant period of length $\delta_p$. Let $RD_j$ be a random variable representing the duration of the $j$-th read operation, with $j=1, 2, ..., \lceil \delta_p/\eta \rceil + 2$. All $RD_j$'s are with distribution the same as $RTD^{(p+1)}$. Therefore, the $TTNR^R$ for the register-based algorithm can be easily derived as follows:

$$TTNR^R = \delta_p \cdot \left( \left\lfloor \frac{\delta_p}{\eta} \right\rfloor + 1 \right) + \sum_{j=1}^{\left\lfloor \delta_p/\eta \right\rfloor} RD_j$$

Figure 5 shows the result of $E(TTNR)$ for the two algorithms with 7 detectors and $SA$ size varies from 1 to 7. As expected, the lease-based algorithm has better response time than the register-based algorithm, because the latter requires three reads with significant time apart to complete a check(). For both algorithms, the response time is better with larger $SA$ value, because it makes the $QA$ value smaller and thus it is faster for $q$ to complete the communication. However, the change in response time is not very significant with different $SA$ sizes.

**E. Evaluation of TTGD**

Metric $TTGD$, time to guaranteed detection, consists of two periods as explained in its definition. The first period is from time $t_0$ when $p$ fails to time $t_1$ after which all activations of check() is guaranteed to return Dead. We use random variable $T_g$ to represent this period. The second period is from time $t_1$ when a check() is invoked to time $t_2$ when the check() returns. We use random variable $T_d$ to represent this period. Then $TTGD=T_g+T_d$. We now calculate $TTGD$ for each of the two algorithms. For simplicity, we choose to use approximations in some steps of the analysis to replace accurate but complicated computation.

Suppose that the last message $p$ sends to the detectors is its $i$-th message. We use a random variable $FL$ to represent the duration from $p$ sending its last message to the detectors to the time $p$ fails (crashes or commits suicide). The value range of $FL$ is $[0, \eta]$. When the
probability that $p$ crashes during any particular sending interval is very small, and the probability of $p$ committing suicide is even much smaller (for which a reasonable implementation should achieve), the distribution of $FL$ can be closely approximated by a uniform distribution, and can be viewed as independent of message delays.

Figure 6 shows the result of $E(\text{TTGD})$ for the two algorithms. The detection times of the two algorithms are at the same level, and in general decreases when $SA$ size increases, because the $QA$ size is smaller. However, the changes in $E(\text{TTGD})$ is small when $SA$ size varies.

When choosing $SA$ size, we see that a small $SA$ size dramatically improves time to suicide while not significantly degrading response time and detection time, therefore one should prefer using small $SA$ size, as long as leaving enough room for $QA$ value to tolerate detector failures. This result provides new insight different from the work in [12, 6], the two closest studies on perfect failure detection that both suggest using high $SA$ value in failure detection.

REFERENCES