Quorum-Based Perfect Failure Detection Service

Wei Chen* Xuezheng Liu† Yunni Xia‡ Lidong Zhou§

Microsoft Research Technical Report

MSR-TR-2009-62

May 2009

Abstract

A failure detection service is perfect *if it eventually detects all failures and every detection correctly identifies a failure that has already occurred. Such a perfect failure detection service serves as a basic building block for many reliable distributed systems, for example in primary/backup replication protocols and distributed lock services. In this paper, we present a comprehensive study on applying quorum systems to the perfect failure detection service in order to enhance the fault tolerance of the service. We provide the precise system model and specification for a quorum-based failure detection service. We prove that stable storage is necessary if the server processes may crash and recover in the middle of the service. We present two novel algorithms that implement the failure detection service and have complementary characteristics. We further develop a set of quality-of-service (QoS) metrics for quorum-based perfect failure detection services, and apply probabilistic analysis to quantify the QoS metrics of the two algorithms.*

keywords: *perfect failure detection service, quorum system, quality of service*

*Microsoft Research Asia, email:weic@microsoft.com
†Microsoft Research Asia, email:xueliu@microsoft.com
‡Chongqing University
§Microsoft Research Asia, email:lidongz@microsoft.com

1. Introduction

Failure detectors are a fundamental abstraction in building fault-tolerant distributed systems. They offer simple semantics that encapsulate timing assumptions. Solutions to fundamental problems such as consensus and group membership hinge on the properties of failure detectors used, as demonstrated in [4].

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.[1] 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 employ a set of processes. We call such a service *quorum-based* because the service remains available as long as a quorum of processes in the service are. 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 quorum-based 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 *observers*, monitor a process q and service queries from client q. The observers may crash and recover, which match the practical situation for a long-lived service. We then present a specification of a quorum-based perfect failure detection service, which includes important properties that eliminate trivial and useless implementations. We prove that to implement such a service, observers must write its state infinitely often to some stable storage that survives observer crashes and recoveries. 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 covers 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. As an example, we study the effect of quorum size, and find that, unlike the suggestions in [12, 6] to use majority quorums, an asymmetric quorum setting such that p's survival depends on a smaller quorum while q's query depends on a larger quorum may provide much better overall QoS in a typical system environment.

To summarize, our paper contributes to the study of the perfect failure detection service through its precise model and specification, its proof of the necessity of stable storage, its two algorithms, its proposal of QoS metrics, and its probabilistic analysis of the QoS metrics. We believe that our work will both enrich the theory of failure detectors and enhance the practical significance of perfect failure detection services.

2. 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 *observers*, and use set $X = \{x_1, x_2, \ldots, x_n\}$ to denote the set of n observers. The third type is client processes that query

[1] A similar condition on leaders is not required in Paxos though.

observers 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 observers 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. Observers 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,[2] 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 observer only takes reactive steps. That is, an observer x only sends out a message to process y (y being p or q) in x's step that processes a message received from x. Thus, we call a message sent by an observer a *reponse* to the message it receives. An observer 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 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 suicide() interface may have several ways to be implemented in systems, such as using a hardware watchdog to guarantee immediate termination [6]. We assume that p does not recover, or equivalently recovers with a different process identifier.

Each Observer may crash at any time and later recover. An observer 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 observer loses all other state values after a crash and recovery. In particular, if an observer 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 quorum systems for the observers. In our setting, a *quorum system* $\mathcal{Q}$ consists of two non-empty sets $\mathcal{Q}_p$ and $\mathcal{Q}_q$ of subsets of observers X such that for all $Q_1 \in \mathcal{Q}_p$ and all $Q_2 \in \mathcal{Q}_q$, Q_1 and Q_2 intersect. Each $Q \in \mathcal{Q}_p$ is called a *survival quorum* and each $Q \in \mathcal{Q}_q$ is called a *query quorum*. Informally, p's survival depends on p's having timely communications with at least one survival quorum, while q's success in querying p's status depends on q's having reliable communication with at least one query quorum.

We now make it formal the meaning of a process being able to communicate with a quorum of observers. The definitions follow the similar ones in [13]. Let $r(y, x)$ denote the round-trip channel from a process y to an observer 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 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 *quorum reliable* at time t if there is a survival (resp. query) quorum Q of observers such that for all $x \in Q$ round-trip channel $r(p, x)$ (resp. $r(q, x)$) is reliable at time t. We say that process y (p or q) is *fairly quorum reliable* if given any time sequence $(t_1, t_2, \ldots)$ that tends to infinity, there exists time t_i at which y is quorum reliable. Intuitively, if y tries to communicate with the observers infinitely often, then it should be successful at least once.

The system has a known parameter Δ that is used to define timeliness. We say that $r(y, x)$ is *timely* 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 receive a response from x within Δ time from t. Process p (resp. q) is said to be *quorum timely* at time t if there is a survival (resp. query) quorum Q of observers such that for all $x \in Q$ $r(p, x)$ (resp. $r(q, x)$) is timely at time t. We say that process y (p or q) is *always quorum timely* if for all time t, y is quorum timely at time t. We say that y is *fairly quorum timely* if given any time sequence $(t_1, t_2, \ldots)$ that tends to infinity, there exists time t_i at which y is quorum timely.

3. Specification of Perfect Failure Detection Service

In our model, failure detection is provided as a service by the observers, which are constantly monitoring the status of process p. To retrieve the status of p, client q invokes a query interface check(), which communicates with the ob-

[2] Bounded clock drifts can also be easily handled.

servers and returns either $Alive$ or $Dead$ status of p to q. Client q may enter the system and invoke check() at any time and may leave the system after the 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 check() and check() returns, the return value must be $Dead$.
- *Strong Accuracy*: If client q invokes check(), which returns $Dead$ at time t to q, then p must have crashed or committed suicide before time t.

Since p may commit suicide, there could be a trivial implementation in which p always commits suicide at the beginning and all check() invocations return $Dead$. This implementation satisfies the above two properties but is useless. We provide the following Integrity property to exclude such trivial implementations. Intuitively, p cannot commit suicide if p always has timely communications with a quorum of observers, even though the quorum may change over time.

- *Integrity*: If p is always quorum timely, then p does not commit suicide.

Moreover, we also need to avoid trivial implementations in which check() is blocked forever without returning any value. The following property is the weaker version requiring that check() should terminate if q is able to achieve at least one round of timely communication with a quorum of observers as long as it tries infinitely often.

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

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

- *Strong Query Termination*: If client q is fairly quorum 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 observers 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 a survival quorum of observers, 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.

4. Necessity of stable storage

In this section, we study the necessity of using stable storage on the observers. In our model, observers may crash and recover at any time. If the observers do not have stable storage, their failures will bring them back to an old state, which may result in q obtaining wrong status about p. In fact, we show that the observers have to write to stable storage an infinite number of times in any algorithm that claims to implement the perfect failure detection service. The following theorem and the proof formalize this idea.

We say a failure detection algorithm is *finite-write* if it guarantees that in every run, there is only a finite number of writes to the stable storage on the observers. Note that if p crashes or commits suicide, it should be easy to guarantee finite-write, so the property is mainly for the case when p keeps alive.

Theorem 1 *There is no finite-write failure detection algorithm that implements a weakly perfect failure detection service. This is true even if all links are timely at all times.*

Proof. Suppose, for a contradiction, that there is a finite-write algorithm A that implements a weakly perfect failure detection service.

Let ϵ be a constant such that $\Delta > 2\epsilon$. We separate the timeline into disjoint time periods, each of which has length ϵ. We use numbers $1, 2, 3, \ldots$ to number the periods, with the first period being $[0, \epsilon)$, the second period being $[\epsilon, 2\epsilon)$, and so on.

For every message that p sends to an observer at time t, if t is in an odd-numbered period, then the message is scheduled to arrive the observer at the next even-numbered period; if t is in an even-numbered period, then the message is scheduled to arrive the observer at the same even period. When the observer receives the message, it process the message immediately, and if there is a reponse sent by the observer back to p, the response is scheduled to arrive at p in the same even-numbered period. By the above scheduling, we know that at any time t process p will receive its response from all observers within at most $2\epsilon < \Delta$ time, so p is always quorum timely. Therefore, by the Integrity property, p will not commit suicide. Moreover, observers only receive messages from p and send responses back to p in even-numbered periods.

Similarly, we can schedule all messages from q to arrive at observers in the odd-numbered periods (either the same period as when q sends the message, or the next period), and thus q is always quorum timely. By the Weak Query Termination property, all q's queries eventually return.

We now construct a run R_1 as follows. In this run, process p does not crash. Process q periodically issues check() (the exact timing of the queries is not important). All the messages are scheduled as described above. At the end of each period, all observers crash and immediately recover. Since these crashes and recoveries occur at the end of each period, they do not interfere with observers' communication with p and q.

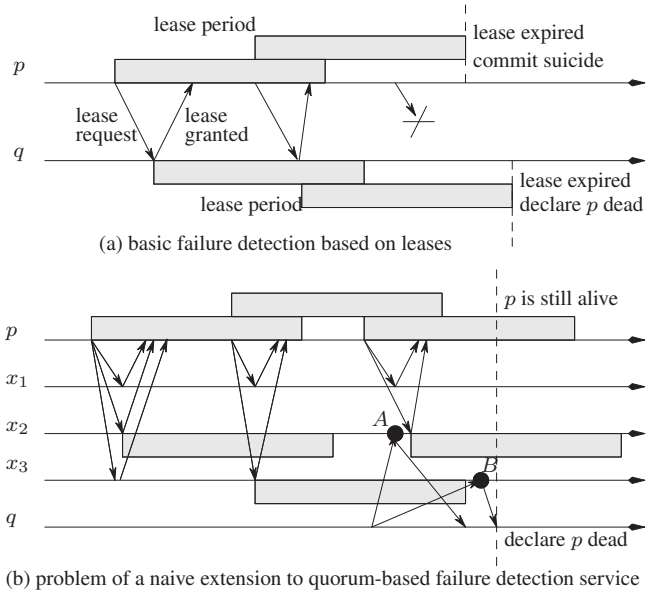
In run R_1, since p does not crash and cannot commit suicide, it keeps alive, and by the Strong Accuracy property, all the queries of q return $Alive$ eventually. Since the algorithm is finite-write, there is a time t_0 after which no observers write to stable storage any more after t_0. Therefore, at the end of each period after time t_0, after all observers recover, they always start with the same set of states restored from their stable storage (of course except for their local clock values which keep increasing).

We now construct another run R_2 based on R_1. Let $[t_1, t_1 + \epsilon)$ be the first odd-numbered period after time t_0 (excluding the period containing t_0). R_2 behaves exactly the same as R_1 before time t_1. At time t_1, process p crashes. After time t_1, all observers are kept in the crashed state in all even-numbered periods, that is, all observers crash at the beginning of every even-numbered period after t_1 and recover at the end of the same even-numbered period. All communications between q and the observers at all times are the same as in R_1. This is possible because observers only receive and process messages from q in the odd-numbered periods. More importantly, observers in the odd-numbered periods cannot distinguish between runs R_1 and R_2, because they recover in the same states, process the same set of messages with the same clock values in the odd-numbered periods, and then crash through the entire even-numbered periods.

Therefore, from the point of view of q, the two runs behave exactly the same since q only communicate with the observers in odd-numbered periods. This means that in R_2 q's queries will always return $Alive$ as in run R_1. This is a contradiction, however, since by the Strong Completeness property, q's query should eventually return $Dead$ because p crashes at time t_1 in R_2. □

5. Algorithms for Perfect Failure Detection Service

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.



(a) basic failure detection based on leases

(b) problem of a naive extension to quorum-based failure detection service

Figure 1. Illustration of lease-based algorithm

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(τ, δ) is for a process to set or reset its timer τ to be expired at δ time units later from the current time; clearTimer(τ) is to clear timer τ; and expireTimer(τ) is triggered when timer τ expires.

We use the term *stable variable* to represent the use of stable storage on the observers. 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 an observer recovers from a crash failure, it loads the value of v in the stable storage back to memory, and every reads 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.

5.1. Algorithm I: Lease-based

The first algorithm is based on the lease mechanism [7] and is extended to quorum systems. 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 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 observer x: We use x to replace q so that observer x correctly detects the status of p, and client q only communicates with x to query the status of p.

However, extending the single observer case to multiple observers with a quorum system is not so straightforward. A naive extension is that each observer x_i simply behave as the single observer, and p needs to receive responses from a quorum of observers in order to extend its lease, while q needs to collect p's status from a quorum of observers to derive the status of p. If all observers in the quorum believe that p is dead, then q declares p dead; otherwise as long as one observer in the quorum believes that p is alive, q declares p alive. Figure 1(b) illustrates a problematic scenario of this naive extension, where we have three observers and any two observers form a quorum. In this scenario, observer 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 declare 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.

If we use the same length for lease period on p and on observers, 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 observers use a longer lease period. Let δ_p be the length of a lease period on p and δ_o be the length of a lease period on the observers. We require that $\delta_o \geq \delta_p + \Delta$, where Δ is the timeliness parameter defined in Section 2. With this setting, we guarantee that points A and B be at least $\delta_o - \delta_p$ time units apart. Then we require that q has to collect a quorum of responses within $\delta_o - \delta_p$ time units. If it fails to do so, it has to resend a new set of messages to observers 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 quorum timely, then eventually q will have a communication round that completes within $\delta_o - \delta_p \geq \Delta$.

On process p (to be monitored):

1 Variable:
2 $counter$: initially 0

3 repeat every η time units:
4 $counter \leftarrow counter + 1$
5 send (LEASE-REQUEST, $counter$) to all observers
6 setTimer($timer[counter], \delta_p$) /* $\delta_p \geq \eta + \Delta$ */

7 Upon expireTimer($timer[i]$)
8 **if** not received (LEASE-GRANT, j) with $j \geq i + 1$ from a survival quorum of observers **then** suicide()

On observer x:

9 Variable:
10 $latest$: stable variable, the latest lease request number received from p, initially 0
11 $deadline$: stable variable, the ending time of x's current lease, initially 0

12 On receipt of (LEASE-REQUEST, cnt) from p:
13 **if** $cnt > latest$ **then**
14 $latest \leftarrow cnt$
15 $deadline \leftarrow$ getClockTime() $+ \delta_o$ /* $\delta_o \geq \delta_p + \Delta$ */
16 send (LEASE-GRANT, cnt) to p

17 On receipt of (CHECK, ts) from client q:
18 **if** getClockTime() $< deadline$ **then**
19 send (CHECK-ACK, ts, $latest$, $Alive$) to q
20 **else** send (CHECK-ACK, ts, $latest$, $Dead$) to q

On client q:

21 check() :
22 **repeat**
23 $ts \leftarrow$ getClockTime()
24 send (CHECK, ts) to all observers
25 **wait until** one of the following conditions holds:
(a) received (CHECK-ACK, ts, cnt, $*$) from a query quorum of observers;
(b) $\delta_o - \delta_p$ time elapsed;
26 **until** (a) is true
27 $cnt1 \leftarrow$ the largest *cnt* received in (CHECK-ACK, ts, cnt, $Dead$) or 0
28 $cnt2 \leftarrow$ the largest *cnt* received in (CHECK-ACK, ts, cnt, $Alive$) or 0
29 **if** $cnt1 \geq cnt2$ **return** $Dead$ **else return** $Alive$

Figure 2. Failure Detection Algorithm I: lease-based

Figure 2 provides the complete pseudocode for the lease-based algorithm. Process p periodically sends LEASE-REQUEST messages to all observers (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 a survival quorum of observers (line 8).

Each observer 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. Observer 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 observers until it receives responses from a query quorum of observers within $\delta_o - \delta_p$ time units (lines 22–26). The computation of the final return value of check() is a little more sophisticated then described above (lines 27–29). Instead of returning $Alive$ as long as one observer returns $Alive$, the algorithm returns $Alive$ if and only if all responses with the highest lease number indicates that p is alive. This allows q to detect the failure of p earlier.

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

5.2. 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 observers 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 η time units and writes the value of $counter$ by sending a $(\text{WRITE}, counter)$ message to all observers (lines 3–6). In the mean time, p starts a timer $timer[counter]$ that expires δ_p time units later. To match this algorithm to the first algorithm, we set the condition under which p commits suicide (line 8) to be exactly the same as the one in the first algorithm. The algorithm on p can be viewed as p invoking several concurrent write operations to write increasing counter values.

The observers' job is very simple. It stores the highest value received in the WRITE messages from p, responds to p with a WRITE-ACK message, and whenever receiving a READ message from q, responds with a READ-ACK message with the highest value it stores.

On client q, it implements a simple read() interface (lines 23–29), in which q periodically sends READ messages to the observers until it receives responses from a query quorum of observers for a particular round of READ message. The return value of read() is the highest value received in the responses. As long as q is fairly quorum reliable, q's read() eventually returns, and it is guaranteed to read at least some value that has reached a survival quorum of observers.

With this read() interface, when q invokes check(), it issues a sequence of read() calls, each of which is separated δ_p time units after the previous one returns. This separation period ensures that if p is still alive, then each read() must return a value higher than the previously read value. Therefore, if q sees that any read() does not return a higher value, it returns $Dead$. Otherwise, it continues until it completes a total number of $\lfloor \delta_p/\eta \rfloor + 2$ read()'s, then it returns $Alive$. We need multiple read()'s because if p is dead, its last few counter values may not reach a survival quorum of observers, 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.

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

5.3. 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 check() may not terminate if q cannot have timely communication with the observers (i.e., q is not fairly quorum 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 observers, q will commit suicide itself, so the termination of check() 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 LEASE-REQUEST message that p sends out before p crashes has a long delay, we may have a situation in which the first check() already returns $Dead$, but the second check() still returns $Alive$, and the time after which check() always return $Dead$ depends on the delay of p's messages. If q issues check() periodically and remembers the query results, this

On process p (to be monitored):

```
1  Variable:
2    counter: initially 0
3  repeat every η time units:
4    counter ← counter + 1
5    send (WRITE, counter) to all observers
6    setTimer(timer[counter], δ_p)          /* δ_p ≥ η + Δ */
7  Upon expireTimer(timer[i])
8    if not received (WRITE-ACK, j) with j ≥ i + 1 from
         a survival quorum of observers then suicide()
```

On observer x:

```
9  Variable:
10   latest: stable variable, the latest counter value
         received from p, initially 0
11 Upon receipt of (WRITE, v) from p
12   if v > latest then latest ← v
13   send WRITE-ACK to p
14 Upon receipt of (READ, ts) from q
15   send (READ-ACK, ts, latest) to q
```

On client q:

```
16 check() :
17   v1 = read()
18   repeat ⌊δ_p/η⌋ + 1 times
19     wait for δ_p time units
20     v2 = read()
21     if v2 ≤ v1 then return Dead else v1 ← v2
22   return Alive
23 Implementation of read():
24   ts ← getClockTime()
25   repeat periodically
26     send (READ, ts) to all observers
27   until [received (READ-ACK, ts, v_i) from
         a query quorum of observers]
28   v ← the largest v_i received in
         (READ-ACK, ts, v_i)
29   return v
```

Figure 3. Failure Detection Algorithm II: register-based

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 check() 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 observers.

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

Overall, we can say that the lease-based algorithm is good at query response time but rely on timely communication between q and the observers, while the register-based algorithm guarantees detection without timeliness assumption. Depending on the actual usage, an application may choose one algorithm or some combination of the two algorithms.

6. Quality of Service of the Perfect Failure Detection Service

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 quorum-based 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.

6.1. Definitions of QoS metrics

For the study of QoS, we assume that the system behavior is probabilistic, and it is caused by probabilistic events in the system such as message delays, the details of which is provided in Section 6.2. We define a set of metrics that captures the important QoS aspects of a perfect failure detection service. These QoS metrics are random variables based on the probabilistic behavior of the system.

The first metric captures how well the service does in preventing p from committing suicide. We denote it as *TTS*, which stands for *time to suicide*, and define it to be the time elapsed from the time p starts sending its first message to the observers 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 check() to return $Dead$. The delay in detection can be separated into two periods. The first period is for check() to stabilize its return value to $Dead$ while the second period is to complete check() operation. More precisely, let t_1 be the earliest time $t \geq t_0$ such that if q invokes check() at time t and if the check() returns, it is guaranteed that the check() returns $Dead$. Let $t_2 > t_1$ be the time at which the 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 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 check() to return a value after it stablizes. 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 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 check() to time t_2 when check() returns in runs in which p does not fail. We call this metric *time to normal response* and denote it as *TTNR*.

6.2. 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 $F(\Delta/2) > 0$. 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 observer failures. Message losses, and to some extent observer 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 fairly quorum timeliness, since if process p (or q) sends an infinite number of messages to all observers, because of the fact that $F(\Delta/2) > 0$, with probability one at least one message will generate timely responses from a quorum of observers.

We study the following type of general quorum systems. A survival quorum is any subset of observers with t observers, and a query quorum is any subset of observers with $n - t + 1$ observers, 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 quorum related parameters n (the number of observers) and t (the size of survival quorums).

In the analysis, we frequently use order statistics [1], which is defined as follows. Given a set of m independent and identically distributed (i.i.d) 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, $X_2, \ldots$, and X_m, where i could be 1, 2, ..., or m. If X_j 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.

Evaluation of *TTS*

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

Let random variable RTD_j denote the round-trip delay from p sending a message to an observer x_j to p receiving the response to this message from x_j. Let D_1 and D_2 be two i.i.d. random variables with distribution function F. Then $RTD_j = D_1 + D_2$.[3]

When the timer $timer[i]$ expires on p, p commits suicide if and only if p has not received responses to its $(i+1)$-th message from a survival quorum of observers. The duration from p sending its $(i+1)$-th message to $timer[i]$ expires is $\delta_p - \eta$. Therefore, the probability of p committing suicide when $timer[i]$ expires can be given as $p_s = Pr(RTD^{(t)} > \delta_p - \eta)$, where $RTD^{(t)}$ is the t-th order statistic of $\{RTD_1, \ldots, RTD_n\}$.

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_s be a random variable representing the number i such that p commits suicide when $timer[i]$ expires. Thus N_s follows a geometric distribution with probability p_s. Considering that a timer expires on p every η time units with the first expiration time δ_p, we have

$$TTS = \eta \cdot (N_s - 1) + \delta_p. \tag{6.1}$$

[3] The distribution of RTD_j can be calculated by a convolution integral. In general, we omit the details of such formulas in the paper.

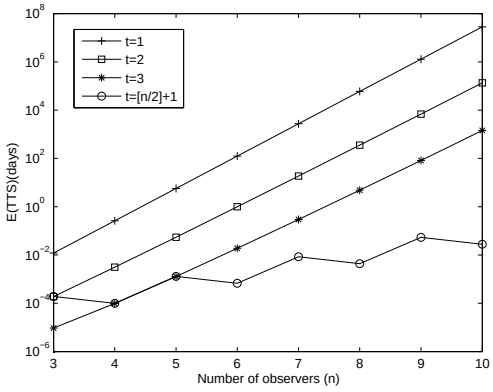


Figure 4. $E(TTS)$ **as the function of the number of observers.**

Then the expected value and the variance of *TTS* are given as: $E(\textit{TTS}) = \eta(1/p_s - 1) + \delta_p$, $Var(\textit{TTS}) = (1-p_s)/p_s^2$.

For the computation of $E(\textit{TTS})$ 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: $\eta = 0.1$s, $\delta_p = 0.15$s, and $\delta_o = 0.2$s.

Figure 4 shows the result of $E(\textit{TTS})$ when the number n of observers varies from 3 to 10, with the size t of survival quorums being 1, 2, 3 or $(\lfloor n/2 \rfloor + 1)$ (the majority quorum) respectively. The results show that when we use constant survival quorum size, $E(\textit{TTS})$ increases exponentially as n increases or t decreases, because p has more options or relies on less number of observers to survive. As an example, when $n = 5$ and $t = 2$, the mean time to suicide is about 1.4 hours, while if $t = 1$, it is increased to 138 hours. Moreover, the majority quorum system (with $t = \lfloor n/2 \rfloor + 1$) does not behave as well and it has the zig-zag behavior, which is inevitable when t increases with n.

Evaluation of *TTNR*

The metric *TTNR*, time to normal response, has different results for the two algorithms. Henceforth, to distinguish random variables for the two algorithms when necessary, we add a super script L to a random variable for the lease-based algorithm, and add a super script R for the register-based algorithm.

For the lease-based algorithm, its response time is determined by the number of rounds that q needs to communicate with the observers. Each round last for at most $\delta_o - \delta_p$ time units. The check() query terminates in a round if and only if it receives responses from a query quorum of observers.

Since the message delays have the same distribution, we also use RTD_j to denote the round trip delay between q and observer x_j. The probability of check() terminates in a certain round is given by $p_r = Pr(RTD^{(n-t+1)} \leq \delta_o - \delta_p)$, where $RTD^{(n-t+1)}$ is the $(n-t+1)$-th order statistic of $\{RTD_1, \ldots, RTD_n\}$.

Let N_r be a random variable representing the number of rounds needed to terminate a check() query. Since rounds are independent of one another, N_r has a geometric distribution with probability p_r. Let R_r be the random variable representing the duration of the last round in which the check() terminates. Variable R_r is independent of N_r, and its distribution is given below as a conditional probability: $Pr(R_r \leq t) = Pr(RTD^{(n-t+1)} \leq t \mid RTD^{(n-t+1)} \leq \delta_o - \delta_p)$. With N_r and R_r, we know that the $\textit{TTNR}^L$ for the lease-based algorithm is given as

$$\textit{TTNR}^L = (N_r - 1)(\delta_o - \delta_p) + R_r. \qquad (6.2)$$

For the register-based algorithm, under the condition that p is alive, a check() query must invoke a total of $\lfloor \delta_p/\eta \rfloor + 2$ read()'s. Each read() is completed when q receives responses from a query quorum of observers, and between two consecutive read()'s there are a constant period of length δ_p. Let RD_j be a random variable representing the duration of the j-th read operation, with $j = 1, 2, \ldots, \lfloor \delta_p/\eta \rfloor + 2$. All RD_j's are i.i.d, with distribution the same as $RTD^{(n-t+1)}$. Therefore, the $\textit{TTNR}^R$ for the register-based algorithm can be easily derived as follows:

$$\textit{TTNR}^R = \delta_p \cdot (\lfloor \delta_p/\eta \rfloor + 1) + \sum_{j=1}^{\lfloor \delta_p/\eta \rfloor + 2} RD_j. \qquad (6.3)$$

Figure 5 shows the result of $E(\textit{TTNR})$ for the two algorithms with 7 observers and survival quorum 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 survival quorums, because it makes the query quorum smaller and thus it is faster for q to complete the communication with a query quorum. However, the change in response time is not very significant with different survival quorum sizes.

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 invocations 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 $\textit{TTGD} = T_g + T_d$. 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.

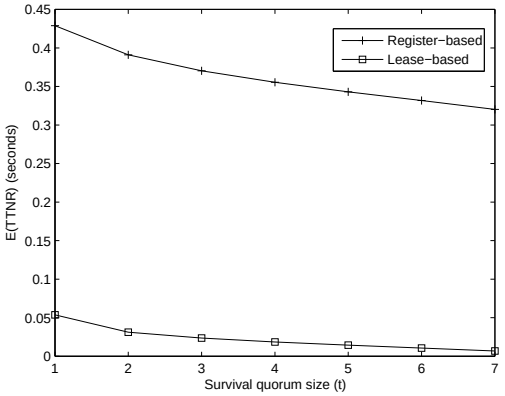


Figure 5. $E(TTNR)$ **as the function of survival quorum size, with** $n = 7$**.**

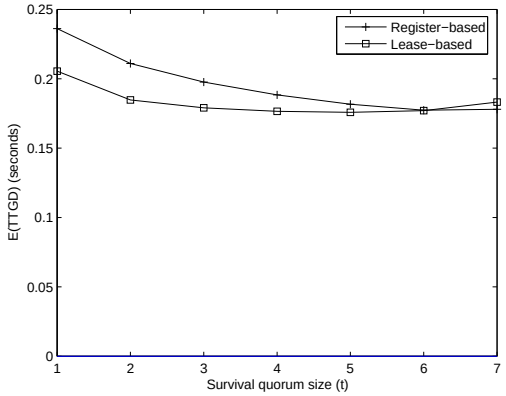


Figure 6. $E(TTGD)$ **as the function of survival quorum size, with** $n = 7$**.**

Suppose that the last message p sends to the observers is its i-th message. We use a random variable FL to represent the duration from p sending its last message to the observers 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 indepdent of message delays.

For the lease-based algorithm, the first period T_g^L ends at time t_1 when a survival quorum of observers have received the i-th message from p and their last lease periods for this message ends. This is because after time t_1, whenever q sends CHECK messages to the observers and receives responses from a query quorum of observers, at least one of the response will be $Dead$ with the largest counter value, so check() can only return $Dead$, but before t_1 q may still receive reponses from a query quorum of observers that all indicate p being alive.

Let LD_j be the delay of the last message p sends to observer x_j. So LD_j's are i.i.d with distribution function F. The t-th order statistic of $\{LD_1, \ldots, LD_n\}$, $LD^{(t)}$, represents the duration from p sending its last message to the time when a survival quorum of observers have received p's last message. Therefore, we have $T_g^L = LD^{(t)} + \delta_o - FL$.[4]

For the lease-based algorithm, its second period T_d^L is the duration of the check() invoked at time t_1. Since the termination of check() is independent of whether check() returns $Dead$ or $Alive$, T_d^L is simply $TTNR^L$. Since T_g^L only depends on the messages p sent while T_d^L only depends on the messages between q and the observers, T_g^L and T_d^L are independent of each other. Therefore, for the $TTGD^L$ metric, we have

$$TTGD^L = LD^{(t)} + \delta_o - FL + TTNR^L. \qquad (6.4)$$

For the register-based algorithm, it guarantees that any check() invoked after p fails could only return $Dead$. Therefore, its first period T_g^R is simply 0. The computation of the second period T_d^R is tedious and is omitted due to space constraint. The basic idea is to compute the probability that the check() invoked at the time when p fails requires three read()'s and use this probability to compute the probability distribution and the mean of the duration of the check().

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

When choosing a quorum size, we see that a small survival quorum size dramatically improves time to suicide while not significantly degrading response time and detection time, therefore one should prefer using small survival quorums, as long as leaving enough room for query quorums to tolerate observer 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 majority quorums in failure detection.

7. Conclusion

In this paper we have argued for the investigation of quorum-based perfect failure detection service. We make

[4] This is an approximation, because it omits the case when p crashes after $timer[i-1]$ expires, in which case $LD^{(t)}$ must be less than $\delta_p - \eta$.

significant contributions both in theory and in practice. From a theoretical point of view, we have provided a specification of the perfect failure detection service as well as an accurate description of the system model. We further presented two novel protocols that implement such a perfect failure detection service and proved their correctness. From a practical point of view, the specification and the system model are both inspired by real distributed systems. The protocols are simple to implement in practice and have indeed been used in some of the systems we build. Finally, QoS metrics we define have direct practical applicability and allow practitioners to evaluate different protocols easily. The future work includes studying alternative models that do not require stable storage, and studying different application usage of the failure detection service (such as periodical monitoring rather than random query).

References

[1] N. Balakrishnan and A. C. Cohen. *Order Statistics and Inference*. Academic Press, 1991.

[2] K. P. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[3] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, Nov. 2006.

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

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

[6] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb. 2003.

[7] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Dec. 1989.

[8] L. Lamport. On interprocess communication; part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[9] L. Lamport. On interprocess communication; part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

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

[11] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 226–238, Oct. 1991.

[12] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec. 2004.

[13] D. Malkhi, F. Oprea, and L. Zhou. Omega meets Paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Symposium on Distributed Computing*, Sept. 2005.

[14] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec. 2004.