A literature review of failure detection

Within the context of solving the problem of distributed consensus

by

Michael Duy-Nam Phan-Ba

B.S., The University of Washington, 2010

AN ESSAY SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2015

Abstract

As modern data centers grow in both size and complexity, the probability that components fail becomes significant enough to affect user-facing services [3]. These failures have the apparent consequence of invoking the impossibility result for distributed consensus in the presence of even one failure [15]. One way to solve the impossibility result is to use failure detectors [8]. In this essay, we present the theoretical models that allow us to solve consensus. Then, we discuss practical refinements to the models for the purposes of implementing failure detectors in practice. Finally, we conclude by surveying common design patterns for building distributed failure detectors.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I would like to thank Ivan Beschastnikh and Norm Hutchinson for their invaluable support and feedback on this essay.

Dedication

To my parents.

Chapter 1

Introduction

As modern data centers grow in both size and complexity, the probability that components fail becomes significant enough to affect user-facing services [3]. Indeed, the presence of failures has both theoretical and practical consequences. When it comes to the problem of distributed consensus, the impossibility result states that consensus cannot be reached in the presence of even one faulty process [15]. In more practical terms, the failure of a component can cause entire systems to stop working properly. Fortunately, Chandra et al. [8] proved that if we have access to a suitable failure detector, then we can solve consensus.

However, correctly implementing failure detection in a way that is both accurate and efficient is no trivial task. For example, Facebook, in designing the Cassandra distributed database, chose to base their implementation of failure detection on the Φ accrual failure detector after determining that a gossip-based model would be too slow to detect failures [23]. As many readers may be aware, the sheer number of users on Facebook's social networking platform means that using a too-slow failure detector would cause service disruptions to a significant number of people. In another example, Google's Chubby service for distributed locks extensively used heartbeats to detect failures [7]. The sheer scale of Google's operations necessitated a hierarchical design for failure detection in order to avoid crippling the network from an overload of heartbeat messages. Indeed, there are many ways that a naïve implementation of failure detection could unintentionally disrupt or degrade a distributed application.

In this essay, we explore how failure detectors help us solve the problem of distributed consensus. We start in Chapter 2 by defining the problem of consensus in asynchronous systems, what it means to have an asynchronous model of computation, and how the concept of failure detection helps us solve the impossibility result. Next, in Chapter 3 we introduce the two basic timeout-based methods for implementing failure detectors. In Chapter 4, we adopt algorithms for estimating the optimal timeout to learn how to build failure detectors that adapt to changing network conditions and application requirements. Finally, in Chapter 5 we end with a survey of common design patterns for building distributed failure detectors.

Chapter 2

The theory behind failure detection and consensus

Failure detection was devised as a way to address the impossibility result for distributed consensus [9]. The consensus impossibility result states that the problem of distributed consensus in fully asynchronous systems cannot be solved in the presence of even one faulty process, because we cannot determine whether a process has failed or is just very slow [15]. In this section, we define the formal models of asynchronous computation in Section 2.1 and failure detection in Section 2.2, followed by a description of the weakest failure detector for solving the problem of distributed consensus in Section 2.3. We will see that the concept of failure detection shifts the dialog from the impossibility of consensus towards the design of failure detectors.

2.1 The impossibility result for consensus

In our primal model of asynchronous computation, we consider a system in which independent processes implement deterministic automata to perform serial computation on inputs received from the network. The network reliably delivers messages between processes. The asynchrony in the system implies that we make no assumptions about how much time it takes to perform computation or to send and receive messages between processes.[1]

The impossibility result for distributed consensus from Fischer et al. is fundamental a limitation in the asynchronous model of computation [15]. The formal proof of the impossibility result shows that no consensus protocol P can ensure that $N \geq 2$ processes all agree on the same value for an arbitrary number of registers x_i, where i identifies a single register, wherein processes communicate by sending messages over the network and assign a value to x_i based on the contents of those messages.

[1] Alternatively, asynchrony could also describe an extreme instance of the General Theory of Relativity [28] in which no two processes exist in physical spaces with the same relative time reference, making accurate time measurements between processes impossible.

In the generalized proof, x_i takes on a value in $\{b, 0, 1\}$, where b is the initial value for x_i that must transition to either 0 or 1. When the value for x_i is b at a process p, the process p is said to be in a bivalent state in that it may transition to either $x_i = 0$ or $x_i = 1$. When the process p assigns x_i a value in $\{0, 1\}$, it is said to be in a univalent state in that there is only one possible transition for the value of x_i to the same value.

If any process q fails to receive a decision for the value of x_i, either because it takes longer to respond to messages than other processes or through long delays in message delivery, the consensus protocol P will never terminate. As time is not measurable in the asynchronous model, P cannot determine whether the process q has failed or is just very slow to respond. No additional state exchange could guarantee the detection of the failed process q and there remains the possibility that the process q has not failed and thus remains in a bivalent state for x_i, leading to the impossibility result for consensus.

The impossibility result is only applicable in the asynchronous model of computation. While it is tempting to disregard asynchrony and consider time as an essential in a model of computation, the asynchronous model lends well to simpler, more robust and portable software implementations in many practical applications [9]. In the next section, we introduce the theoretical concept of failure detection which allows us to continue using the asynchronous model of computation to solve consensus.

2.2 A model of failure detection

Continuing our exploration of the impossibility result for consensus in the asynchronous model of computation, we now introduce the concept of failure detection. In this section, we consider failure detectors as all-knowing oracles without assuming *how* they could be implemented in practice; we defer to Chapters 3-5 for a review of concrete failure detectors and a discussion of practical design patterns. Instead, this section describes the abstract classes of failure detectors, their equivalences, and how the theoretical results allow us to solve the problem of consensus.

Let us define *failure* as the event in which a process halts without prior notice and *failure detection* to mean the event in which a failed process is marked as suspected of failure. In addition, we also make available a global, monotonically increasing virtual clock. Rather than describing physical time, the virtual clock advances only if some event occurs in the system. From the viewpoint of the virtual clock, an event occurs in the system if any process performs an arbitrary unit of computation, which may include sending and receiving messages on the network or experiencing a failure. The clock is not available to individual processes or the failure detectors and exists only to aid in the analysis. In this section, we will refer to time as defined by the virtual clock.

Processes form a failure detection group wherein each member process shares information about process failures in the group. Each process maintains an instance of the failure detector that provides, possibly incorrect, information about process failures in the group. Member processes in the monitoring group share information gathered from their local failure detectors. The global failure detector $\mathscr{D}$ describes the aggregate failure detection capabilities of the failure detection group.

Within this model, Chandra and Toueg defined two completeness properties and four accuracy properties that the failure detector $\mathscr{D}$ may satisfy. The completeness and accuracy properties are loosely related to the true failure and false positive rates, respectively. We provide the informal definitions here; curious readers are referred to [9] for the formal definitions and proof of equivalence.

Completeness The failure detector $\mathscr{D}$ is said to have *strong completeness* if every failed process is permanently suspected by *every* correct process. On the other hand, if every failed process is only permanently suspected by *some* correct process, $\mathscr{D}$ is said to have *weak completeness*.

We say that weak completeness is equivalent to strong completeness in that one can emulate the other [9]. With weak completeness, at least one correct process will suspect a failed process. That process can then share that information with the rest of the failure detection group to achieve strong completeness in aggregate. The reverse is trivially true: strong completeness trivially satisfies the weak completeness property. This equivalence allows us to focus solely on the four classes of failure detectors described by the accuracy property.

Accuracy The failure detector $\mathscr{D}$ is said to have *strong accuracy* if *no* process is suspected before it fails. Similarly, $\mathscr{D}$ is said to have *weak accuracy* if only *some* correct process is never suspected of failure. It follows that strong accuracy satisfies the weak accuracy property.

For both these properties, it may be difficult to guarantee that at least one correct process is never suspected of failure. Thus, Chandra and Toueg introduce the concept of *eventual* satisfiability for the accuracy properties.

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect $\mathscr{P}$	Strong $\mathscr{S}$	Eventually Perfect $\diamond\mathscr{P}$	Eventually Strong $\diamond\mathscr{S}$
Weak	$\mathscr{Q}$	Weak $\mathscr{W}$	$\diamond\mathscr{Q}$	Eventually Weak $\diamond\mathscr{W}$

Table 2.1: Eight classes of failure detectors and their symbolic representations. Failure detectors on the same column are equivalent, while failure detectors with weak accuracy are weaker than ones with strong accuracy. Likewise, failure detectors with eventual accuracy are weaker than ones with perpetual accuracy. See also Figure 2.1 for an illustration of equivalences.

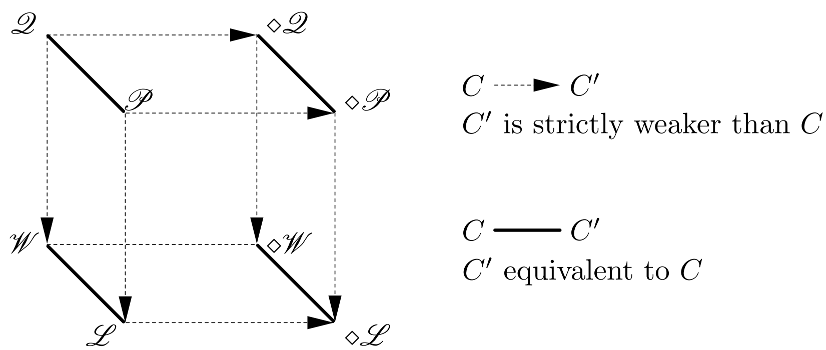


Figure 2.1: The equivalences of failure detector classes. See Table 2.1 for a mapping from the symbolic representations to the failure detection classes.

That is, the failure detector $\mathscr{D}$ satisfies *eventual strong accuracy* if there is a time after which correct processes are not suspected by any correct process. Likewise, $\mathscr{D}$ is said to have *eventual weak accuracy* if there is a time after which only *some* correct processes are not suspect by any correct process.

The eight classes of failure detectors are listed with their symbolic representations in Table 2.1 and their equivalences illustrated in Figure 2.1. We will see in the next section that we are able to solve the consensus problem using the weakest failure detector $\diamond\mathscr{W}$ that satisfies the weak completeness and eventual weak accuracy properties.

2.3 The weakest failure detector

In this section, we introduce the result from Chandra et al. [8, 9] showing that the weakest failure detector $\mathscr{D}$ for solving consensus for $n > 2f$ only needs to satisfy the weak completeness and eventual weak accuracy properties, where n is the total number of processes in the failure detection group and f is the number of failed processes in the group. We thus begin our discussion by showing that we may use a strongly complete, weakly accurate failure detector to solve the consensus problem.

Using a strongly complete, weakly accurate failure detector Let us begin by considering Algorithm 2.1 which uses a strongly consistent failure detector $\mathscr{L}$ satisfying the weak accuracy property. Recall from Section 2.1 that in our model of computation for the consensus problem, every process maintains an arbitrary number of registers x_i. Consensus is reached when all processes agree and commit to a single, globally consistent value for x_i. For our discussion, let us generalize the consensus problem to allow x_i to accept *any* value.

Algorithm 2.1 consists of three phases. In the first two phases, the algorithm collects all proposed values from correct processes. The strong consistency and weak accuracy properties guarantee that all failed processes are detected and at least one correct process is never suspected. This in turn guarantees that all processes are able to construct a consistent view V_p at the end of phase 2. In phase 3, processes deterministically decide on a value for x_i and the algorithm trivially solves the consensus problem.

Using a weakly complete, weakly accurate failure detector More impressively, we can also solve the consensus problem using a weakly consistent, weakly accurate failure detector $\mathscr{W}$. Recall that we learned in the previous section that a weakly complete failure detector can be converted to a strongly complete failure detector by allowing processes to share failure information. Thus, we can modify Algorithm 2.1 to use a weakly complete, weakly accurate failure detector by instructing processes to broadcast a message after detecting that a process has failed. By the weak completeness property, a failed process is detected by at least one process and the failure detector thus behaves as if it satisfies the strongly complete property.

Naturally, even with the strong completeness property, we could consider it difficult to design a weakly accurate failure detector in which at least

Algorithm 2.1 Solving consensus using any failure detector $\mathscr{L}$ satisfying the strong completeness and weak accuracy properties. Every processes p executes the `propose` function to reach consensus on a value for x_i.

function PROPOSE(v_p)
  Let $V_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$ be p's view of all proposed values

  Phase 1: collect all proposed values
  for all $n - 1$ other processes **do**
    Send v_p and V_p to all processes
    Wait to receive v_q and V_q from q or until $\mathscr{D}$ suspects q
    for all processes q that did not fail **do**
      ▷ query the failure detector
      for all processes k participating in consensus **do**
        Update $V_p[k] \leftarrow V_q[k]$ where $V_p[k] \neq \perp$
      end for
    end for
  end for

  Phase 2: update V_p for failed processes
  Send V_p to all processes
  for all processes q in the failure detection group **do**
    Wait to receive V_q from q or until $\mathscr{D}$ suspects q
    if q did not fail **then** ▷ query the failure detector
      for all processes k participating in consensus **do**
        Update $V_p[k] \leftarrow V_q[k]$ where $V_p[k] \neq \perp$
      end for
    end if
  end for

  Phase 3: deterministically decide on a non-$\perp$ value in V_p

end function

one correct process is never suspected. Instead, observe that the consistency problem does not require that the failure detection properties hold *forever*. Rather, a failure detector $\mathscr{D}$ only needs the properties to hold for a *sufficiently long time* for the consensus algorithm to complete. Thus, it is sufficient for $\mathscr{D}$ to eventually satisfy the weak accuracy property.

Let us now turn our attention to the second algorithm from Chandra et al., which uses the weaker eventually weakly accurate failure detector to solve the consensus problem with $n > 2f$ processes, where n is the total number of processes and f is the number of failed processes [9].

Using a strongly complete, eventually weakly accurate failure detector Algorithm 2.2, adapted from [9], solves the consensus problem using a weakly consistent, eventually weakly accurate failure detector $\diamond\mathscr{L}$.

The algorithm proceeds through three epochs. In the first epoch, more than one decision value is possible. In the second epoch, a majority of processes have accepted the coordinator's proposed value and the value is *locked*: no other decision value is possible. In the third epoch, processes decide the locked value. In all epochs and phases, the strong completeness property ensures that the algorithm makes progress. Progression through the three epochs then relies on the eventual weak accuracy property to ensure that the majority of correct processes eventually accept the locked value in phases 3 for the leader to commit the value in phase 4.

In the first epoch, either the processes are unaware of other processes' proposed values for x_i (round $r_p = 1$) or fewer than $\lceil (n+1)/2 \rceil$ have chosen the same value for x_i, leaving the possibility for more than one value for x_i to be chosen. In the second epoch, the eventual weak accuracy property ensures that at least $\lceil (n+1)/2 \rceil$ processes eventually accept the coordinator's proposed value in phase 3. In the third epoch, once a majority of processes agree on and lock the proposed value for x_i, the coordinator is guaranteed to choose the locked value in phase 2. From then on, by the eventual weak accuracy property, a majority of processes will eventually accept the locked value in phase 3 and decide on a consistent value after phase 4.

This algorithm is correct with the assumption that $n > 2f$, where n is the number of processes participating in consensus and f is the number of processes that fail. The constraint arises from this contradiction: if we instead assume $n \leq 2f$, then the algorithm could lock an inconsistent value in the third epoch. As we are using an eventually weakly accurate failure detector $\diamond\mathscr{W}$, the $2f$ suspected processes could actually be alive. The $2f$ processes could have independently locked a value inconsistent with the

$n \leq 2f$ correct processes, thereby violating consistency and resulting in the contradiction. Thus, we require $n > 2f$ to solve the problem of distributed consensus with $\diamond\mathscr{W}$.

Similarly to Algorithm 2.1, we can modify Algorithm 2.2 to use a weakly consistent, eventually weakly accurate failure detector $\diamond\mathscr{W}$ by sharing failure information between processes. The result in [8] shows that $\diamond\mathscr{W}$ is indeed the weakest possible failure detector capable of solving the consensus problem with $n > 2f$ processes.[2]

Relation to 2PC and 3PC Alert readers may notice the similarity of Algorithms 2.1 and 2.2 to two-phase commit (2PC) and three-phase commit (3PC), respectively. In 2PC's prepare and commit phases, a coordinator initiates a transaction in the prepare phase by asking processes if they can commit a value. The coordinator notifies the participants in the commit phase to either commit the value (if all processes replied yes) or to abort the transaction (if one or more processes replied no) [4].

Whereas 2PC may block processes indefinitely if the coordinator fails before sending the commit or abort message, 3PC adds the pre-prepare phase to prevent blocking. At the beginning of a transaction, the coordinator asks processes to pre-prepare and, if all processes reply yes, the coordinator proceeds to the prepare and commit phases of 2PC, otherwise the transaction is aborted. The pre-prepare phase avoids blocking processes indefinitely by allowing processes to timeout on the coordinator during the prepare phase.

Indeed, the failure detector-based algorithms we have described could be derived from 2PC and 3PC by replacing the explicit timeout handling with a default response based on the failure information of a process from an abstract failure detector. In both cases, the use of a sufficiently strong failure detector allows us to simplify the termination protocol when failed processes are detected [17]. For example, in Algorithm 2.1, the use of a weakly accurate failure detector prevents the algorithm from blocking because the failure detector will yield a "default" answer to abort the procedure. Likewise, the eventually weakly accurate failure detector in Algorithm 2.2 prevents the algorithm from blocking because failed processes are eventually detected.

In this section, we defined the formal models of computation leading to an understanding of how failure detectors allow us to solve the impossibility result for distributed consensus. We also showed that the weakly consistent, eventually weakly accurate failure detector $\diamond\mathscr{W}$ is capable of solving the

[2] We refer the reader to [8] for the full proof of this result.

Algorithm 2.2 Solving consensus using any failure detector $\diamond\mathscr{L}$ satisfying the strong completeness and eventual strong accuracy properties. Every processes p executes the `propose` function to reach consensus on a value.

function PROPOSE(v_p)
  Let $r_p \leftarrow 0$ be the current round number
  Let $ts_p \leftarrow 0$ be latest round number in which p updated v_p
  Let $state_p \leftarrow undecided$
  while $state_p = undecided$ **do**
    Advance $r_c \leftarrow r_c + 1$ and elect a coordinator $c_p \leftarrow (r_p + 1) \mod n$
    Phase 1: all processes send their proposed values to c_p
    Send (p, v_p, r_p) to c_p
    Phase 2: c_p gathers $\lceil (n+1)/2 \rceil$ proposals and sends a new value
    if $p = c_p$ **then**
      Wait to receive $\lceil (n+1)/2 \rceil$ proposals (q, v_q, r_q)
      Update v_p to the proposal v_q with the highest r_q if any
      Send (p, r_p, v_p) to all
    end if
    Phase 3: all processes wait for the new proposal from c_p
    Wait to receive (c_p, r_p, v_c) from c_p or until $\mathscr{D}$ suspects c_p
    if c_p failed **then** $\triangleright$ query the failure detector
      Send $(p, r_p, nack)$ to c_p
    else
      Update $v_p \leftarrow v_c$ and $ts_p \leftarrow r_p$
      Send (p, r_p, ack) to c_p
    end if
    Phase 4: c_p waits for $\lceil (n+1)/2 \rceil$ acks
    if $p = c_p$ **then**
      Wait to receive responses from $\lceil (n+1)/2 \rceil$ processes
      if p received $\lceil (n+1)/2 \rceil$ acks **then**
        Send $(r_p, v_p, commit)$ to all
      end if
    end if
    if p receives a $(r_p, v_c, commit)$ message at between phases **then**
      Update $v_p \leftarrow v_c$, $ts_p \leftarrow r_p$, and $state_p \leftarrow decided$
    end if
  end while
end function

consensus problem with $n > 2f$ processes, where n is the total number of processes and f is the number of failed processes. Chandra et al. [8] further proved that $\diamond\mathscr{W}$ is the weakest class of failure detectors for solving consensus with $n > 2f$ processes.

While there exist weaker failure detectors, they provide even less information than the model of failure detection we have presented here and solve different classes of computational problems. For example, the Υ failure detector solves the *wait-free set agreement* problem by informing that *some* set of processes are cannot be the set of correct processes [18]. Of course, the wait-free set agreement is not consensus. In the remainder of this essay, we introduce the concrete failure detection algorithms that satisfy these properties.

Chapter 3

Binary failure detection with partial synchrony

In the asynchronous model of failure detection introduced in Chapter 2, we assumed that there are no bounds on the message delay and that physical time either could not be reliably measured or there is no way of deterministically predicting execution time. In addition, we also assumed that messages are eventually reliably delivered. This last assumption is easy to ensure in practice if all processes make infinitely many attempts to send and infinitely many attempts to receive messages [15].[3]

Thus, if we know ahead of time that f processes will fail out of n total processes, then solving the problem of consensus is trivial: we simply wait to receive $n - f$ messages before terminating the algorithm [29]. If we only know that *up to* f processes *may* fail, then we have the unreliable failure model described in Section 2.3 and only need to ensure that no more than $\lfloor n/2 \rfloor$ processes fail [8]. Nevertheless, this solution is predicated on having access to a failure detection oracle.

In practice, we rarely know f a priori, messages *can* be delayed indefinitely, and we do not have access to a failure detection oracle. As a result, to implement a suitable eventual weak failure detector $\diamond\mathscr{W}$ to solve the problem of consensus, we need to enrich our model of computation with additional assumptions. In particular, we relax the asynchrony assumption *for the model of failure detection* to allow processes to reach an approximate common notion of time to enable the use of timeouts. As we cannot achieve full synchrony in the sense that we cannot guarantee that computation, including the delivery of network messages, will complete within an explicit time bound, the use of timeouts gives us what is called a partially synchronous system. In the partially synchronous model, an upper bound on computation and network delays exists (or is enforced), but is not known in advance [13].

[3] We will revisit this inefficiency in Chapter 5.

This subtle distinction on the application of partial synchrony allows us to restrict the concept of physical time to the failure detection model, while maintaining the simpler asynchronous model of computation for solving consensus. In essense, we are allowing algorithms for solving consensus to disregard the concept of time and continue to assume that any failed processes will "notify" the algorithm of its failure.

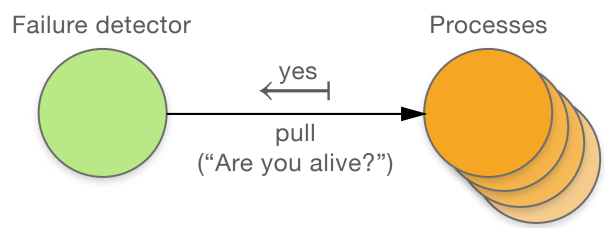
With partial synchrony, the unidirectional *pull* and *push* models form the basis of all currently known failure detector algorithms [14, 20]. In this section, we define and compare the *pull* and *push* models and conclude with a description of the more flexible *dual* that combines features from both *push-pull* models.

3.1 Pull failure detection

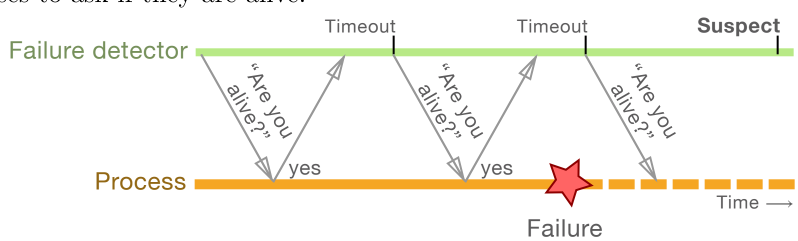
A failure detector implementing the *pull* model of interaction periodically sends *liveness requests* to processes [14, 20]. If a process responds to the request before a timeout, the failure detector considers it alive. Otherwise, the process is marked as suspected of failure. The pull control flow is illustrated in Figure 3.1a while the flow of the monitoring messages are illustrated in Figure 3.1b.

When used as part of a distributed failure detector, it is also easy to see that we can achieve the results from Section 2.3 without requiring all processes to monitor all other processes. Instead, we achieve strong completeness by allowing individual failure detectors to share failure information. While individual failure detectors may make mistakes, given a long enough timeout, we can ensure that the distributed failure detector is eventually weakly accurate. Thus, the pull model for failure detection satisfies the properties of the weakest failure detector capable of solving the problem of consensus.

A benefit of using the pull model is that it is simple to implement because monitored processes are passive participants. That is, processes only need to react to external liveness requests. They do not need to maintain a local clock to send regular messages and may operate within the asynchronous model of computation. In the next section, we describe the *push* model of interaction, which does necessitates that processes and failure detectors both have access to a clock, for a more efficient way to implement failure detection.



(a) The *pull* models of failure detection in which the failure detector regularly polls processes to ask if they are alive.



(b) Monitoring messages in the *pull* model of failure detection.

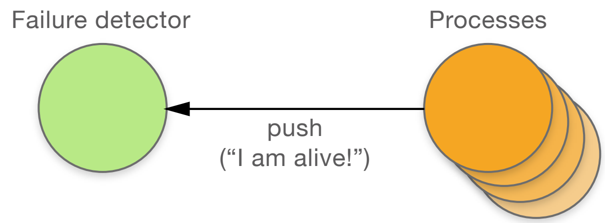
Figure 3.1: The *pull* model of failure detection illustrated.

3.2 Push failure detection

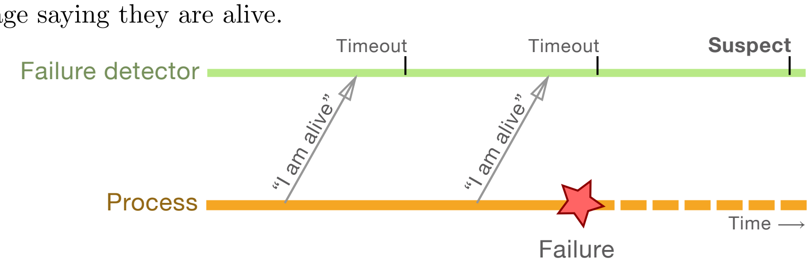
In the push model of failure detection, monitored processes become active participants. They periodically send *heartbeat* messages to the failure detector [14, 20]. Processes are suspected of failure when they stop sending heartbeat messages and become *quiescent*. The push control flow is illustrated in Figure 3.2a while the flow of the monitoring messages are illustrated in Figure 3.2b.

When used as part of a distributed failure detector, the push model does not require that all processes monitor all other processes. Rather, we may organize processes into a sufficiently structured monitoring topology to ensure that all processes are associated with at least one individual failure detector. Individual failure detectors then share failure information to achieve the strong completeness and eventual weakly accurate properties [1].

The result from [1] shows that the push model does not require the explicit use of timeouts and instead counts the total of heartbeat messages received from each process, marking a as suspected of failure when its heartbeat counter stops increasing. However, the heartbeat messages are still required to have some degree of periodicity for this method to work. This means that processes must adopt a more complex partially synchronous model of computation. In exchange, the push model halves the number of



(a) The *push* model of failure detection in which processes regularly broadcast a message saying they are alive.



(b) Monitoring messages in the *push* model of failure detection.

Figure 3.2: The *push* model of failure detection illustrated.

network messages needed to implement a failure detector suitable for solving the consensus problem [1, 24].

3.3 Push-pull failure detection

In a heterogeneous environment, it may be desirable (or necessary) to deploy both *push*- and *pull*-based failure detectors. For such scenarios, Felber et al [14] proposed the *dual* model of failure detection that combines the push and pull models. A failure detector implementing the dual strategy accommodates both models by accepting heartbeat messages when available and sending liveness requests otherwise.

The dual model could be particularly useful, for example, to allow two data centers separated by an intercontinental network connection to efficiently monitor processes for failure across the both systems. Within each data center, we can efficiently synchronize process clocks and reliably support the push model of failure detection. Between the data centers, network delays make it more difficult for heartbeat messages to be reliably and periodically delivered. Instead, we send liveness requests over the intercontinental link for the reason that the pull model does not require accurate timekeeping.

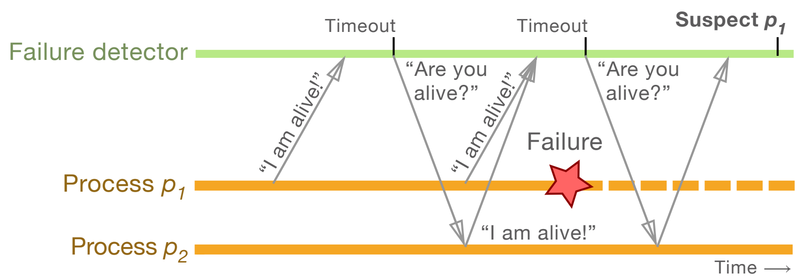


Figure 3.3: The *push-pull* model of failure detection illustrated. Both the push and pull models of interaction are represented.

Figure 3.3 illustrates an example of the dual model in which process p_1 is push-aware and process p_2 is pull-aware. When the failure detector is started, it accepts heartbeat messages from p_1. After a timeout, it detects that p_2 is not sending heartbeat messages and starts periodically sending liveness requests to p_2. Failures are then detected using the appropriate model of failure detection [14].

Thus far, we have assumed that we know the optimal timeout delays a priori as global, unchanging values. This assumption presupposes that the network delays are predictable and that we have access to clocks with negligible drift to time those delays [10, 21]. In practice, both these assumptions are impractical. In Chapter 5, we will explore the topic of how to implement practical failure detection in more detail. Leading up to that discussion, we will explore in Chapter 4 the more sophisticated class of graded failure detectors that output a numeric estimate of a process's failure, rather than a simple binary answer. We will see shortly that these graded failure detectors give us much flexibility in implementing failure detection as a shared service.

Chapter 4

Accrual failure estimation for adaptive failure detection

In Chapter 3, we relaxed our asynchronous model of computation to make it possible to implement concrete failure detectors. The partial synchrony in the revised model manifested as the timeouts used in the push and pull models of failure detection. However, we assumed that the timeout durations were known a priori, possibly by measuring the expected message delay in the network. The use of a single, unchanging timeout also presupposes that we have access to clocks with negligible drift for reliable failure detection [10, 21]. In this section, we instead relax these assumptions and explore algorithms for adaptively estimating the optimal timeout for the purposes of failure detection.

The goal of estimating the optimal timeout is simple. The longer we wait to timeout, the longer it takes to detect a failure. On the other hand, if we timeout too early, we make more mistakes when reporting suspected processes. Thus, we begin the chapter by describing three algorithms for estimating network delays: an algorithm estimating the round-trip time for the pull model of failure detection in Section 4.1 and two algorithms for estimating heartbeat arrival times in the push model in Section 4.2. We then conclude the chapter by presenting the accrual class of failure detectors that reimagines the use of timeout estimation to return a probabilistic estimate of a process's failure status, rather than a simple binary answer.

4.1 Estimating round-trip time

In the pull model of failure detection, Jacobson's algorithm, used in the Transmission Control Protocol (TCP) for estimating the round-trip time (RTT), is likely the most widely used [22]. As we'll see in Section 4.2, Jacobson's algorithm is of particular interest to us because it is used in Bertier et al.'s algorithm for estimating the next arrival time for heartbeat messages [6].

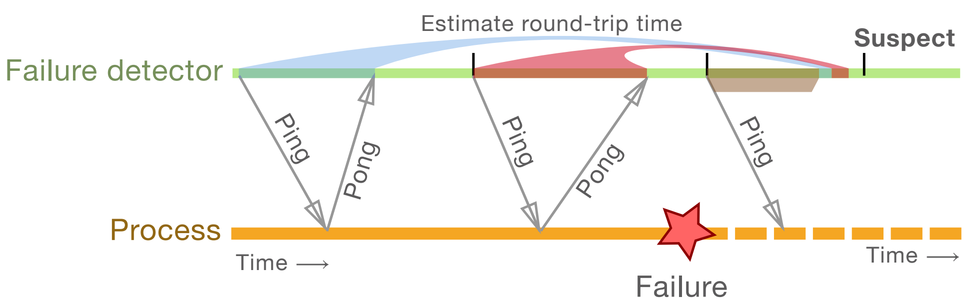


Figure 4.1: Round-trip time estimation based on previous round-trip delays. The shaded area to the right of the rightmost ping message sent from the failure detector represents the weighted historical RTT estimate. The next two shaded areas represent the weighted RTT contributions from the two most recent RTT delays.

Jacobson's algorithm Illustrated in Figure 4.1, Jacobson's algorithm calculates a running estimate of the RTT, giving more weight to more recently observed RTT delays. The RTT estimation algorithm is formally described in Equations 4.1-4.3.

$$A = \gamma A + (1 - \gamma)M \tag{4.1}$$
$$D = \beta D + \beta(|M - R| - D) \tag{4.2}$$
$$R = A + \phi D \tag{4.3}$$

Here, A is an estimate of the mean RTT, M is the most recent RTT sample, and D is an estimate of the mean deviation in the RTT. R is the estimated RTT that incorporates both the observed average and deviation in the RTT. The parameters γ and β determine how much weight to give past RTT samples and have suggested values of 0.9 and 0.125, respectively [22]. The parameter ϕ determines how much deviation from the mean RTT to tolerate and has a suggested value of 2. The algorithm makes relatively few assumptions about the network and adapts to changing conditions as rapidly as the values of γ, β, and ϕ allow.

As round-trip time estimation algorithms are widely discussed elsewhere in the literature, we limit our discussion to Jacobson's algorithm here. In the next section, we discuss Chen's algorithm for estimating the next heartbeat arrival times for use with the push model of interaction, followed by Bertier's algorithm, which combines Chen's algorithm with Jacobson's algorithm to adapt to changing network conditions.

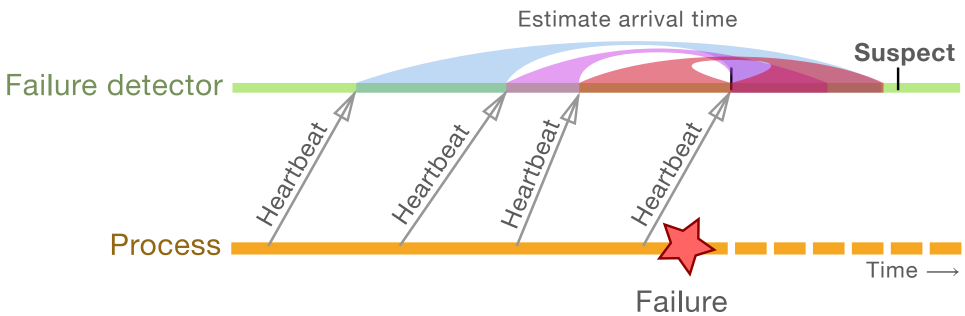


Figure 4.2: Heartbeat estimation based on previous arrival times. The shaded area to the right of the rightmost heartbeat represents the next estimated heartbeat delay based on the weighted mean of the three most recently measured heartbeat intervals.

4.2 Estimating heartbeat arrival times

For the push model of failure detection, Chen et al. [10] first proposed an adaptive algorithm based on probabilistic analysis of network traffic to estimate the arrival time of the next heartbeat in the push model of failure detection.[4] The basic idea behind heartbeat estimation is illustrated in Figure 4.2. Due to network fluctuations, we can expect the time between heartbeats to vary over time. The timeout t_{timeout} at the failure detector is set based on an estimation of the mean and variance in the observed delays between heartbeats, with the addition of a constant safety margin α. As a follow-up, Bertier et al. [6] then proposed to combine Chen's estimation algorithm with Jacobson's estimation of round-trip time. We describe both algorithms in this section.

Chen's algorithm Algorithm 4.1 describes Chen's algorithm. The core functionality of the algorithm depends on the accuracy of $EA_{\ell+1}$, the estimated arrival time for the next heartbeat, where ℓ is the sequence number of a heartbeat. The failure monitor p estimates $EA_{\ell+1}$ by

$$EA_{\ell+1} \approx \frac{1}{n}\left(\sum_{i=1}^{n} A_i - \eta s_i\right) + (\ell+1)\eta \tag{4.4}$$

[4]Chen et al. [10] actually described two estimation algorithms: one that depends on highly accurate synchronized GPS and Cesium clocks and one that does not make this assumption. As we are interested in methods for relaxing the requirement for synchronized clocks, we describe only the latter in this review.

where $s_1, \ldots, s_n$ are the sequence numbers of heartbeats received from p and $A_i, \ldots, A'_n$ the receipt times of those messages. In the summation component, the estimation function takes the average of the difference between the expected arrival time and the actual arrival time $(A_i - \eta s_i)$. This average essentially describes the drift in q's local clock relative to p's local clock. The estimated drift is then added to the next expected heartbeat arrival time $((\ell + 1)\eta)$. Based on their algorithmic analysis and simulation results, Chen's algorithm provides good estimates of the arrival time for the next heartbeat.

In both the main algorithm and the estimation function for $EA_{\ell+1}$, η is the configurable heartbeat interval and α is a constant safety margin. Chen et al. additionally provide methods for calculating the parameters α and η. We refer curious readers to [10] for more information.

Bertier's algorithm Algorithm 4.2 describes Bertier's algorithm. Whereas Chen's algorithm assumes a constant, probabilistic value for the error margin α, Bertier's algorithm uses Jacobson's algorithm to estimate α.

In addition, Bertier's algorithm specially handles the initialization of the failure detector, in which there are fewer than n previous heartbeat arrival times with which to calculate EA'_i. The initial estimates for EA'_i use the algorithm described in Equations 4.5 and 4.6.

$$U_{i+1} = \frac{t}{i+1} \cdot \frac{i}{i+1} \cdot U_i \tag{4.5}$$

$$EA'_{i+1} = U_i + \frac{i+1}{2} \cdot \eta \tag{4.6}$$

The values U_0 and EA'_0 are initially set to 0 and both quantities are calculated at the same time. When $i > n$, EA_{i+1} is calculated using Equation 4.4. Based on their network measurements, Bertier's algorithm is competitive with Chen's algorithm, trading shorter detection times (by adjusting the timeout lower as network conditions allow) for an increase in the number of false failure detected (because the estimated timeout will not immediately respond to increases in the network delay).

Chen's and Bertier's algorithms for estimating the optimal timeout have real practical implications: having a good estimate of the optimal timeout allows us to use a wider variety of clocks with non-insignificant drift and possibly lower cost. In the next section, we introduce the accrual failure detectors that further refine the idea of timeout estimation to decouple the interpretation of failure data from the failure monitoring mechanism [21].

Algorithm 4.1 Chen's algorithm for estimating the arrival time of the next heartbeat. η and α are configuration parameters and EA_i is the estimated arrival time for the heartbeat at the ith sequence. The algorithm for estimating EA_i is provided in the text.

```
function HEARTBEAT(p)                                   ▷ using p's local clock
    for all i ≥ 1 do
        Send heartbeat m_i to q at time i · η
    end for
end function

function MONITOR(q)                                     ▷ using q's local clock
    τ_0 ← 0                     ▷ the expected arrival time of the next heartbeat
    ℓ ← −1                      ▷ the largest sequence number received from p
    loop
        if t = τ_{ℓ+1} then                             ▷ t is the current time
            Suspect p as failed                         ▷ heartbeat not received
        else if q receives a message m_j from p and j > ℓ then
            ℓ ← j                                       ▷ save new sequence number
            τ_{ℓ+1} ← EA_{ℓ+1} + α                      ▷ next estimated arrival time
                                                        ▷ (see Equation 4.4)
            if t < τ(ℓ + 1) then                        ▷ t is the current time
                Trust p as alive                        ▷ heartbeat received
            end if
        end if
    end loop
end function
```

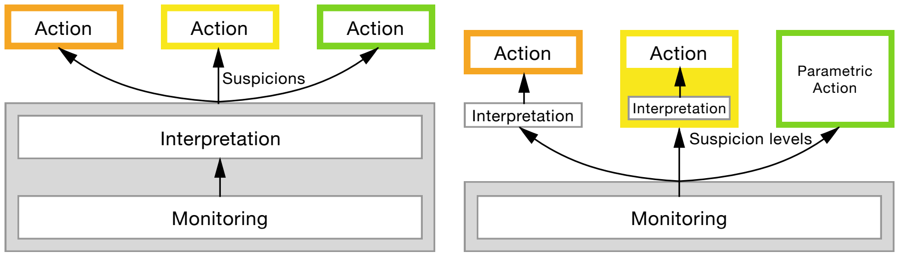
Algorithm 4.2 Bertier's algorithm for estimating the arrival time of the next heartbeat. The parameters η is the same as in Chen's algorithm and the parameters γ, β, and ϕ are described in Section 4.1. The algorithm for estimating EA'_i is provided in the text.

```
function HEARTBEAT(p)                                    ▷ using p's local clock
    for all i ≥ 1 do
        Send heartbeat m_i to q at time i · η
    end for
end function

function MONITOR(q)                                      ▷ using q's local clock
    τ_0 ← 0                   ▷ the expected arrival time of the next heartbeat
    ℓ ← −1                       ▷ the largest sequence number received from p
    loop
        if t = τ_{ℓ+1} then                             ▷ t is the current time
            Suspect p as failed                        ▷ heartbeat not received
        else if q receives a message m_j from p and j > ℓ then
            ℓ ← j                                   ▷ save new sequence number

                                       ▷ estimate α using Jacobson's algorithm
            error_j ← t − EA'_{j−1} − α_{j−1}
            delay_{j+1} ← delay_j + γ · error_j
            var_{j+1} ← var_j + γ · (|error_j| − var_j)
            α_{j+1} ← β · delay_{j+1} + φ · var_{+1}

            τ_{ℓ+1} ← EA'_{ℓ+1} + α_{j+1}              ▷ next estimated arrival time
            if t < τ(ℓ + 1) then                        ▷ t is the current time
                Trust p as alive                           ▷ heartbeat received
            end if
        end if
    end loop
end function
```



(a) Binary failure detectors (b) Accrual failure detectors

Figure 4.3: As compared to the architecture of binary failure detectors (a), accrual failure detectors (b) decouple monitoring from interpretation to allow client applications to tune the behavior of the failure detector. The parametric action modulates its behavior based on the in suspicion level, for example by sending alerts with varying levels of urgency to system administrators based on the suspicion level.

4.3 Accrual failure detection

In contrast to the binary failure detectors that we have discussed thus far, accrual failure detectors output a continuous range of values [12, 21]. While returning a binary value (*trust* or *suspect*) is more convenient to the client application in that there is no ambiguity as to the interpretation of the failure information, not all applications have the same failure tolerance and may benefit from finer interpretations of the data. Indeed, there is an *inherent* trade-off between the speed and accuracy of failure detection [12].

Part of the motivation behind the design of accrual failure detectors is to decompose failure detection into three components:

- *Monitoring* component gathers information about processes.
- *Interpretation* component decides the failure status of a process based on gathered data.
- *Actions* are executed based on the failure status of a process.

Whereas the architecture of binary failure detectors tightly couples the monitoring and interpretation components, accrual failure detectors decouple monitoring from interpretation. These differences are illustrated in Figure 4.3.

Informally, the reported values from accrual detectors represent the confidence level that a process has failed since the last time the detector received a message from the process. The suspicion output of a heartbeat-based accrual failure detector is illustrated in Figure 4.3. More precisely, an accrual failure detector outputs a suspicion level $susp\_level_p(t)$ (a floating point number) at time t for process p such that it exhibits the following properties:

1. *Asymptotic completeness* – if a process p is faulty, $susp\_level_p(t)$ increases to infinity as t increases to infinity. That is, as time passes and the faulty process stops indicating that it is alive, we can be increasingly confident that the process is truly faulty.

2. *Eventual monotonicity* – if a process p is faulty, there is a time after which $susp\_level_p(t)$ increases monotonically. This is because, as defined, the only way for the $susp\_level_p(t)$ to decrease is when it is reset to zero by property 4.

3. *Upper bound* – process p is correct if and only if there is an upper bound on $susp\_level_p(t)$ for all t. That is, the suspicion level of a correct process p never increases above a definite threshold as a consequence of (4).

4. *Reset* – if p is correct, then $susp\_level_p(t) = 0$ for some $t \geq t_0$, such as when the failure detector receives a message from p to confirm that it is alive.

In this section, we introduce two failure detectors with these properties: the Φ accrual detector [21] and Satzger's failure detector [31].

The Φ accrual failure detector The Φ accrual failure detector was the first failure detector described to satisfy these properties and accompanied the work that defined the accrual class of failure detectors [20]. The failure detector outputs a probabilistic estimate Φ that a process has failed based on the last time the detector received a heartbeat message from the process. The output value Φ is calculated using the equation

$$\Phi(t_{now}) \stackrel{\text{def}}{=} -\log_{10}\left(P_{later}\left(t_{\text{now}} - T_{\text{last}}\right)\right) \tag{4.7}$$

where t_{now} is the current time at which Φ is calculated and T_{last} is the last time the failure detector received a heartbeat message from the process in question. The value $P_{later}(t)$ is calculated using the equation

$$P_{later}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_t^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}}\, dx = 1 - F(t) \tag{4.8}$$

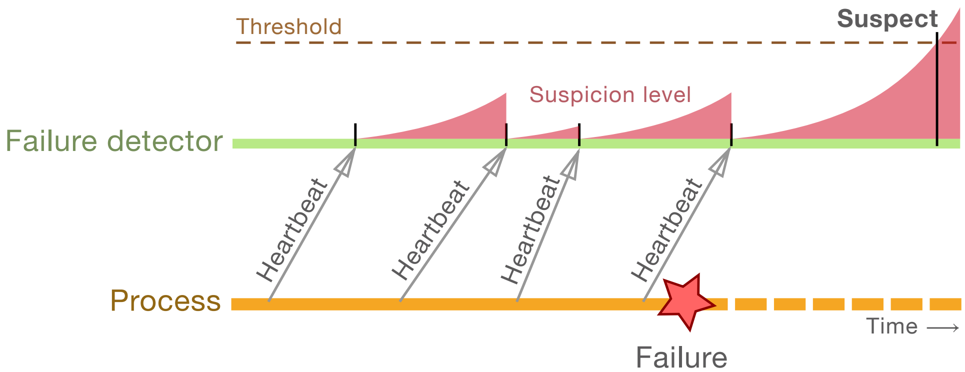


Figure 4.4: Suspicion level estimation is calculated based on the time since the last heartbeat message was received. The suspicion level of a process is illustrated as rising with the "graphs" above the failure detector line. An application sets the suspicion threshold based on their quality of service requirements. When a process's suspicion level crosses the application's threshold, the process is considered as suspected of failure.

where $F(t)$ is the cumulative distribution function of a normal distribution with mean μ and variance σ^2. The mean and variance describe the network delay and is either provided a priori or estimated using the methods described in Section 4.2. The value of $F(t)$ is usually determined using a lookup table of precalculated values.

Thus, given a value Φ for process p, we may then decide to suspect p when Φ crosses above a threshold. Hayashibara et al. estimate that for $\Phi = 1$, the probability that p has not failed is about 10%, 0.1% for $\Phi = 2$, and 0.01% for $\Phi = 3$, etc. While the Φ accrual detector's application of the normal distribution yielded an elegant and adaptive implementation of a failure detector, the method is relatively computationally intensive compared to the simpler method used by Satzger et al. [31].

Satzger's failure detector The failure detector by Satzger et al. [31] satisfies the properties of accrual failure detector, but with much lower computational costs than the Φ failure detector's use of the normal distribution. Instead, Satzger's algorithm maintains a history of the durations between past heartbeat messages to calculate an accrual failure value. Algorithm 4.3 describes the failure detector, with η being the window size for heartbeat intervals, which in turn determines the max size of the historical list of heartbeat intervals S, and α is a scaling factor. Remarkably, Satzger's failure detector matches and exceeds the performance of Φ failure detector in simulation and is competitive with Chen's failure detector from Section 4.2.

Algorithm 4.3 Satzger's algorithm for accrual failure detection. Here, η is the window size for heartbeat intervals, which in turn determines the maximum size of the historical list of heartbeat intervals S_q for all processes q, and α is a scaling factor.

```
function HEARTBEAT(p)                                   ▷ using p's local clock
    loop
        Send heartbeat message to q every Δ_i interval
    end loop
end function

S_q ← []                      ▷ list of past durations between heartbeats at q
f_q ← t_0                          ▷ receipt time of the last heartbeat at q

function MONITOR(q)                                     ▷ using q's local clock
    loop
        t_Δ ← t − f_q
        f_q ← t
        Append t_Δ to S_q
        if size of S_q > η then
            Remove the head of S_q
        end if
    end loop
end function

function PROBABILITY(q)              ▷ get failure probability of q at time t
    t_Δ ← t − f_q
    S_q^(t_Δ·α) ← subset of S_q such that each measured interval is before t_Δ·α
    return |S_q^(t_Δ·α)| ÷ |S_q|
end function
```

A major benefit of accrual failure detectors is that they allow multiple client applications to simultaneously tune the failure detection to suit their needs. In the next chapter, we'll define what this tuning means as part of the discussion on deploying failure detection as a shared service.

Chapter 5

Failure detection as a service

In Chapter 2, we learned that there exists a weakest class of failure detectors for solving the problem of consensus. In practice, the failure detection algorithms described in Chapters 3 and 4 rely on timeouts. However, the task of tuning these timeouts for optimal performance is a nontrivial task [16].[5] In this section, we revisit failure detection from a system designer's perspective and describe general strategies that have been used to implement failure detection in practice.

We begin by describing what it means for a failure detector to be "performant", we discuss the three commonly used completeness, accuracy, and timeliness metrics. Then, we discuss the practical lower bounds on the metrics. Finally, we conclude with a survey of common design patterns for implementing failure detection as a fundamental service in distributed systems [30].

5.1 Measuring quality of service

The completeness and accuracy properties we introduced in Chapter 2 make a good basis for measuring the quality of service of failure detectors. The properties roughly translate into the true positive failure detection rate and false positive (mistaken) failure detection rate and describe the fundamental trade-offs when tuning failure detectors. When failure detectors are tuned with high failure detection rates, they usually make more mistakes, and vice-versa. The failure detection algorithms we surveyed in Chapters 3-4 all used these metrics as a basis for comparing the novel designs against existing algorithms. In addition to completeness and accuracy, timeliness is also an important factor to consider when implementing failure detectors in practice [6, 19, 21, 30, 31].

Indeed, a major goal of the adaptive and accrual failure detectors from Chapter 4 is to reduce or bound the time to detect a failure, while maximiz-

[5] The majority of modern networks exhibit variable delays and make no guarantee of reliable message delivery. This makes manually tuning the timeout very difficult for human operators [16].

ing the failure detection rate and minimizing the mistake rate [10]. These failure detection algorithms likewise favor the push model of interaction because it allows for faster failure detection with fewer network messages. On the other hand, the pull model of interaction allows for on-demand failure detection and reduces the load on the network when failure detection is not regularly needed [20].

Regardless of which model of interaction we use for failure detection, the network itself bounds how completely, accurately, and quickly we are able to detect failures. Intuitively, the more messages we send, the more likely the network will fail to reliably and timely deliver those messages in practice [20]. In turn, the network unreliability affects the true failure detection rate (due to delayed messages) and causes the failure detector to return less accurate results (due to dropped messages). Likewise, the lower bound on the failure detection time is given by

$$t_{\text{detect}} \geq t_{\text{send delay}} + t_{\text{network}} + t_{\text{receive delay}} \tag{5.1}$$

where $t_{\text{send delay}}$ is the computational delay at process p in preparing and sending a network message to process q, t_{network} is the message delivery delay imposed by the network, and $t_{\text{receive delay}}$ is the computational delay at q in receiving and processing a network message from p [16]. In most physical networks, the network delay t_{network} manifests as a random variable that is difficult to predict [16].

It should be clear that the task of implementing a reliable and efficient failure detector is nontrivial. As such, it would be practically useful to decouple the implementation of failure detection from the algorithms that rely on it, such as for solving consensus [32]. In the next section, we discuss commonly used design patterns to that end.

5.2 Common design patterns

In Chapters 3-4, we progressively introduced the design dimensions of interaction (pull and push), dynamism (static and dynamic round-trip time estimation), and interpretation (binary and accrual). In this section, we expand on that list to incorporate the design dimensions described in [30]. Our adaptation of the design dimensions are listed in Table 5.1.

Interaction As described in Chapter 3, there are two basic models of interaction for failure detection: *pull* and *push*. In the pull model, the failure detector periodically sends liveness requests to processes and suspect

Parameter	Options
Interaction	Pull Push Passive
Dynamism	Static Adaptive
Interpretation	Binary Accrual

(a) Design dimensions discussed in previous chapters.

Parameter	Options
Architecture	Centralized Distributed
Isolation	Baseline Sharing
Configuration	Coarse-grained Fine-grained
Specialization	Homogeneous Heterogeneous
Monitoring	All-to-all Randomized Neighborhood
Propagation	One-to-all Structured Gossip

(b) Additional design dimensions adapted from [30].

Table 5.1: Design dimensions for failure detection.

processes of failure if they fail to respond within a timeout. In the push model, processes periodically send heartbeat messages to the failure detector and are suspected of failure when they stop sending messages after a timeout. As was shown in [14], both the pull and push models of interaction can coexist in a system.

Dynamism As we discussed in Chapter 4, we can enhance the performance of failure detectors by adapting to network conditions. In contrast to *static* failure detectors that require prior knowledge of the network delay, *adaptive* failure detectors are able to automatically adjust to transient network delays [5, 10, 22]. As networks predominantly do not guarantee the reliable and timely delivery of network messages, adaptive failure detectors are much more useful in practice [16, 33].

Interpretation In Section 4.3, we described the accrual failure detectors that decouple failure *interpretation* from failure monitoring. In contrast to *binary* failure detectors that either suspect or don't suspect a process of failure, *accrual* failure detectors return a probabilistic estimate that a process has failed. Clients are then free to set their own threshold, according to their quality of service needs, for considering whether a process has failed.

Architecture As in [30], we describe the general architecture of a failure detector as the *architecture* design dimension. In the *centralized* architecture, the failure detector is implemented as a single, monolithic component. Centralized failure detectors are easy to maintain, but represent a single point of failure. As such, modern implementations of failure detection employ the *distributed* architecture in which multiple instances of the failure detector improve the availability of the service.

Isolation With the distributed architecture, failure detectors have the choice of whether to operate in *isolation*. In the *baseline* model of isolation, the failure detector makes an independent decision about failures without consulting other instances of the failure detector. On the other hand, in the *sharing* model, instances of the failure detector share information in order to make decisions about failures [30, 34]. The main benefit of the sharing model is that neighboring processes may cooperatively monitor a third process to improve the combined quality of service for failure detection.

Configuration Complementary to the dynamism design dimension, the *configuration* design dimension applies to parameters to the failure detector that cannot be determined without operator intervention. The heartbeat interval, for example, is best set based on how quickly an application needs to detect a failure, while still balancing computational resources. This information is not readily adapted from environmental measurements.

We say that a failure detector supports *coarse-grained* configuration when it only allows for a single, global configuration value. Conversely, we say that it supports *fine-grained* configuration if it supports multiple configuration values. The dichotomy between binary and accrual failure detectors illustrate the coarse-grained and fine-grained approaches, respectively.

Specialization When all processes run an instance of the failure detector, we say the design is *homogeneous*. On the other hand, in the *heterogeneous* model, failure detectors are independent agents that monitor processes of interest, which may include themselves [25–27]. In the context of providing failure detection as a shared service, heterogeneity may manifest as a way to prevent application failures from affecting the failure detection service or to aggregate failure detection requests to reduce computational costs.

Monitoring Within the architectural design dimension, we can further categorize distributed failure detectors based on their monitoring patterns: all-to-all, randomized, and neighborhood-based.

In the *all-to-all* approach, all failure detectors monitor all other processes [30]. With a small number of processes, this method is sufficiently efficient. However, with increasing numbers of processes, the number of unicast messages sent over the network increases exponentially. While hardware multicast could be used to efficiently implement all-to-all, the feature is not always available in practice [11].

The *randomized* monitoring pattern is related to the epidemic literature in that failure detectors randomly select processes to monitor, yielding an increasingly smaller probability of *not* being monitored at any given time as the number of failure detector instances increases [11].

In contrast to randomized monitoring, *neighborhood-based* monitoring patterns deterministically organize failure detectors and the monitored processes into localized groups to take advantage of the locality between processes [5, 30]. This approach is especially applicable when processes reside in physically separated networks with slow interlinks [20].

Propagation Finally, *propagation* is the last design dimension on our tour of failure detectors. Related to the monitoring design dimension, when a failure detector has news of a failure (or lack thereof), it needs to share that information with the interested parties. Here, we describe three common propagation patterns: one-to-all, gossip, and structured.

As with all-to-all monitoring, the *one-to-all* propagation method is limited to small groups of processes or requires the availability of hardware multicast to be efficiently implemented, neither of which is always practical.

Gossip-based propagation is based on the study of epidemics and, as with the randomized monitoring pattern, a process (running an instance of the failure detector) randomly selects another processes with which to share failure updates. The probability that a process does not receive an update decreases exponentially as the number of processes in the system increases [11, 25].

The *structured* propagation pattern, like the neighborhood-based monitoring pattern, organizes processes with a sufficiently structured network overlay to reduce the number of messages needed for any one process to send to reach all other processes. For example, hierarchical failure detectors implement the structured pattern by organizing processes into a hierarchy [5, 20, 30].

5.3 Practical considerations

Armed with an understanding of that failure detectors can help us solve a number of distributed problems, such as consensus, we must not forget that failure detectors also have real limitations. For example, we have only considered failures in the *crash* model. That is, we expect processes to fail by permanently halting computation, without necessarily giving prior notice. Our model of failure detection may not always provide sufficient information to solve consensus in these other failure models [16]. In fact, Aguilera et al. [2] provided an algorithm for solving consensus in the *crash-recovery* model and solutions for consensus in other failure models exist in the literature. Freiling et al. [16] also bring our attention to the fact that there were alternatives to the Chandra-Toueg model of failure detection we introduced in Chapter 2. While we do not explore these alternative models or solutions, we would like to leave the reader with the knowledge that the literature on failure detection is far richer than what is contained in this essay.

Chapter 6

Summary

In Chapter 2, we presented the seminal work by Chandra et al. [9] describing the theory behind failure detection and its utility in solving the problem of distributed consensus [8, 15]. We then presented in Chapter 3 the basic pull and push interaction patterns used in implementing real failure detectors. In Chapter 4, we described the increasingly sophisticated algorithms used to make failure detectors work in practice, leading to the elegant accrual class of failure detectors. Finally, we surveyed the common design patterns used in implementing failure detection as a shared service. We hope that readers of this essay have gained a better understanding of the failure detection abstraction and its utility in solving distributed consensus.

Bibliography

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 126–140. Springer-Verlag, September 1997.

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[3] Luiz André Barroso and U Holzle. The Datacenter as a Computer. *Morgan & Claypool Publishers (May 2009)*, 2009.

[4] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Reading, Mass. : Addison-Wesley Pub. Co, 1987.

[5] M Bertier, O Marin, and P Sens. Performance analysis of a hierarchical failure detector. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 635–644, 2003.

[6] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. IEEE Computer Society, June 2002.

[7] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 24–24. USENIX Association, November 2006.

[8] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM*, 43(4):685–722, July 1996.

[9] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

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

[11] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303–312. IEEE Computer Society, June 2002.

[12] X Defago, P Urban, N Hayashibara, and T Katayama. Definition and Specification of Accrual Failure Detectors. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05*, pages 206–215. IEEE Computer Society, June 2005.

[13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[14] P Felber, X Defago, R Guerraoui, and P Oser. Failure detectors as first class objects. In *Distributed Objects and Applications, 1999. Proceedings of the International Symposium on*, pages 132–141. IEEE, 1999.

[15] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM*, 32(2):374–382, April 1985.

[16] Felix C Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Computing Surveys (CSUR)*, 43(2):9–40, January 2011.

[17] R Guerraoui, M Larrea, and A Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Reliable Distributed Systems, 1995. Proceedings., 14th Symposium on*, pages 41–50. University of Bologna, 1995.

[18] Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy Lynch, and Calvin Newport. On the weakest failure detector ever. In *PODC*

'07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, page 235, New York, New York, USA, August 2007. ACM Request Permissions.

[19] Indranil Gupta, Tushar D Chandra, and Germán S Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179, New York, New York, USA, August 2001. ACM Request Permissions.

[20] N Hayashibara, A Cherif, and T Katayama. Failure detectors for large-scale distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 404–409. IEEE Comput. Soc, 2002.

[21] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The Φ Accrual Failure Detector. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04*, pages 66–78. IEEE Computer Society, October 2004.

[22] V Jacobson and V Jacobson. *Congestion avoidance and control*, volume 18. ACM, August 1988.

[23] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[24] M Larrea, A Fernandez, and S Arevalo. Optimal implementation of the weakest failure detector for solving consensus. In *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, pages 52–59. IEEE Comput. Soc, 2000.

[25] A Lavinia, C Dobre, F Pop, and V Cristea. A Failure Detection System for Large Scale Distributed Systems. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 482–489. IEEE, 2010.

[26] Joshua B Leners, Trinabh Gupta, Marcos K Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *nsdi'13: Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation.* USENIX Association, April 2013.

[27] Joshua B Leners, Hao Wu, Wei-Lun Hung, Marcos K Aguilera, and Michael Walfish. Detecting failures in distributed systems with the Falcon spy network. In *SOSP '11: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, page 279, New York, New York, USA, October 2011. ACM Request Permissions.

[28] David Mayer-Foulkes. Time is Being – General Relativity as a Theory of Time. *SSRN Electronic Journal*, 2010.

[29] A Mostefaoui, E Mourgaya, and M Raynal. Asynchronous implementation of failure detectors. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 351–360. IEEE, 2003.

[30] M Pasin, S Fontaine, and S Bouchenak. Failure Detection in Large Scale Systems: a Survey. *Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE*, pages 165–168, 2008.

[31] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. *A new adaptive accrual failure detector for dependable distributed systems.* ACM, New York, New York, USA, March 2007.

[32] N Sergent, X Defago, and A Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, pages 137–145. IEEE Comput. Soc, 2001.

[33] Naixue Xiong, Athanasios V Vasilakos, Jie Wu, Y Richard Yang, Andy Rindos, Yuezhi Zhou, Wen-Zhan Song, and Yi Pan. A Self-tuning Failure Detection Scheme for Cloud Computing Service. In *2012 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 668–679. IEEE, 2012.

[34] S Q Zhuang, D Geels, I Stoica, and R H Katz. On failure detection algorithms in overlay networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, pages 2112–2123. IEEE, 2005.