Lecture Notes for CSCI 351: Distributed Computing
Set 8-Failure Detectors[2]

Professor Talmage

May 8, 2023

---

1 Definition

One way to solve problems in failure-prone distributed systems is to detect failures and proceed with only the correct processes. This allows us to avoid scenarios when we must wait for crashed processes to ensure that they do not wake up and perform an action contradictory to one we have already taken, such as deciding a different value in a consensus algorithm.

Of course, in a truly asynchronous system, detecting failures is generally impossible, since the same indistinguishability of a crashed versus slow process holds. What failure detectors do, instead, is allow us to characterize the minimum amount of synchrony that is required for solving consensus. If we can determine the *weakest failure detector* to solve a particular problem, then we can ask how much synchrony is required to implement that failure detector, and that minimum synchrony is also the minimum amount required to solve the original problem. We typically focus more on comparing weakest failure detectors, though, than actually trying to quantify the synchrony required to implement one, as defining/quantifying synchrony is difficult.

We will work in asynchronous, crash-prone systems, similar to that from the FLP result. We may work in either message-passing or shared memory. A *failure detector* is an algorithm which stores a variable $suspect_i$ at each process p_i, and which acts independently of any other algorithm running (other algorithms may not update $suspect_i$). $suspect_i$ is a set of processes which p_i currently suspects of having crashed. $suspect_i$ can grow and shrink, adding and removing processes multiple times. If a process c is in $suspect_i$ from some point in time and never leaves it, we say that p_i *permanently suspects* c from that point in time on.

To specify the behavior of a failure detector, we define conditions on the relation of processes' $suspect_i$ variables and the set of processes which have actually crashed. Recall that a *correct* process is one which does not crash, a *faulty* process is one that crashes at some point in an execution, then within an execution we talk about *crashed* and *alive* processes at a particular point in time as those which have or have not crashed before that point in time.

The first property we define, *Completeness*, discusses how effectively the failure detector reports crashed processes. Eventually, every correct process should suspect every crashed process. That is, for any correct p_i, $suspect_i$ should eventually contain every process that crashes.

Exercise: What is the simplest complete failure detector?

To avoid the trivial solution of always suspecting all processes, we add conditions demanding that processes not over-suspect. These are known as *Accuracy* conditions, and there are several different conditions a failure detector may provide:

- *Strong Accuracy*: No process suspects any live process. That is, no process appears in any $suspect_i$ before it crashes.

- *Weak Accuracy*: Some correct process is never suspected.
- *Eventual Strong Accuracy*: Eventually, no correct process is in any $suspect_i$.
- *Eventual Weak Accuracy*: Eventually, some correct process is not in any $suspect_i$.

This gives a set of four possible failure detector classes:

Completeness:	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong Completeness	Perfect: P	Strong: S	Eventually Perfect: $\Diamond P$	Eventually Strong: $\Diamond S$

2 Reductions

We want to understand how these four different failure detectors relate. Specifically, if one is stronger than another, and we can solve a problem with the weaker one, then we can solve that problem with the stronger one, as well. We can then ask "What is the weakest failure detector required to solve problem X?".

To reduce failure detector A to failure detector B, we give an algorithm which relies on B and provides a set $output_i$ at each process. If $output_i$ provides the guarantees of A, then we have reduced A to B by showing that it is sufficient for the system to provide B for us to have the knowledge guaranteed by A.

For a first result, we will show that our definition of completeness is stronger than we need. Instead of eventually having *every* correct process suspect every crashed process, we can just require that eventually, for every crashed process c, there is *some* correct process p that permanently suspects c. This is known as *Weak Completeness*.

Completeness:	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong Completeness	Perfect: P	Strong: S	Eventually Perfect: $\Diamond P$	Eventually Strong: $\Diamond S$
Weak Completeness	Q	Weak: W	$\Diamond Q$	Eventually Weak: $\Diamond W$

Exercise: How do you think we can get from a each failure being detected at one process to being detected at all processes?

The following code provides strong completeness from any failure detector that provides weak completeness.

Algorithm 1 Code for each p_i to implement a strongly complete failure detector, assuming D is a weakly-complete failure detector.

Initially:
1: $output_i = \emptyset$
Repeat Forever:
2: $send(i, D.suspects_i)$ to all
3: **Upon** RECEIVE$(j, suspects_j)$
4: $output_i = (output_i \cup suspects_j) \setminus \{j\}$

Exercise: Choose an accuracy condition and argue that if D satisfies that condition, then the above algorithm will, as well.

What this means is that if we can solve a problem with a failure detector in the top row, then we can also solve it with the failure detector in the same column of the second row. Thus, we can write algorithms using stronger tools, and automatically convert them to algorithms which work in less-friendly systems.

3 Solving Consensus

Recall that if we can implement consensus, then we can implement any other ADT we like. Thus, we want to know whether we can use a particular failure detector to solve consensus. It is important to realize that we are modifying our system model, since we know that consensus is impossible in an asynchronous, crash-tolerant model. Adding a failure detector adds some level of synchrony, though how much is sometimes unclear.

We will outline a consensus algorithm which relies on $\Diamond S$, which means that it is actually possible to solve consensus in a model which only provides $\Diamond W$. Note that this algorithm is in a shared memory model, for simplicity.

Exercise: Restate $\Diamond S$'s and $\Diamond W$'s guarantees in English.

Algorithm 2

```
1:  function DECIDE(x)
2:      r = 0
3:      while true do
4:          c = r mod n
5:          if i == c then
6:              ans = safe − phase(r, x)
7:              if ans ≠ ⊥ then decisionVal = ans
8:              end if
9:          else
10:             wait until c ∈ suspect_i or R_c.phase ≥ r
11:         end if
12:         r+ = 1
13:     end while
14: function SAFE-PHASE(r, x)
15:     write r into own register R_i
16:     read all other processes' registers, abort if any has larger phase number
17:     choose value from other process' register with largest phase number as preference, write it or x if
        none found
18:     read all other processes' registers, abort if any has larger phase number
19:     decide preference
```

4 Implementing Failure Detectors

First, consider a high-level idea: each process will periodically announce that it is still alive. This is known as a *heartbeat* algorithm. Processes can use timeouts to note that they have not heard from a particular other process for a while and suspect that they have crashed.

Exercise: What completeness and accuracy guarantees will this provide? On what system assumptions does it depend?

As long as processes have reasonably accurate local clocks, this will give strong completeness, as eventually every live process will suspect every crashed process. There are no true accuracy guarantees, however, as each correct process' heartbeats can be repeatedly delayed past whatever timeout other processes use, so that they repeatedly suspect it. In a real system, however, we will probably increase the timeout every time we find that we erroneously suspected a process, and eventually we are highly unlikely to incorrectly suspect a process.

4.1 Ω

Another way to think of failure detectors is to think of them as reporting which processes are still live. The Ω failure detector is one such. Instead of a $suspect_i$ array at each process, it maintains a $trust_i$ variable at each process, which stores one process it thinks is currently live. Ω guarantees that eventually every process' $trust_i$ will hold the ID of the same correct process.

Exercise: Given an Ω failure detector, construct a $\Diamond S$ failure detector. Do this with no additional communication.

- For completeness, we can suspect every process that Ω does not trust: $suspect_i = P \setminus \{trust_i\}$.
- For accuracy, the correct process which is eventually in all $trust_i$'s is no longer in any $suspect_i$, and thus satisfies the requirements of eventual weak accuracy.

Thus, to implement $\Diamond S$, it is sufficient to implement Ω. We consider two kinds of failures: processes may crash or be arbitrarily slow, and communication links may drop messages or take arbitrarily long to deliver them. We will restrict ourselves to considering *fair* communication links: if a message is sent infinitely many times, it will eventually arrive. We also assume that there is some process which is *eventually timely*– it must have a minimum execution speed and there is an upper bound d on the delay of messages from that process, though d is only required to hold eventually.

These constraints are a type of synchrony assumption, that we are describing more explicitly than by assuming a failure detector exists. As we saw, these failure detectors cannot exist in a truly asynchronous, fault-prone system, since if they exist we can solve consensus and FLP precludes solving consensus in an asynchronous, fault-prone system. Thus, to implement a failure detector, we must assume a certain amount of synchrony.

See pseudocode on next page. This algorithm still uses heartbeats and timeouts, with the following optimizations:

- Adaptive timeouts: When we hear from a process, we increase the length of its timeout.
- Message efficiency: Only the current candidate leader sends heartbeats. This means that we only need $n-1$ links to stay active forever, once we stabilize on a single trusted process. However, it also means that we may timeout on and suspect a process who has already ceded the *leader* (trusted) role.
- Accusations: When we timeout on a process, we send an *accuse* message to it. Note that we only need to accuse the process itself, not publicize our accusation. If that process is crashed, then we already suspect it, and everyone else will, as well, when they timeout on it. If it has not crashed, it will update its *counter*, unless it realizes that the accusation is out of date because it has already ceded the *leader* role and moved to a new *phase*. A higher *counter* means that the process will lose *leader* competitions.

This algorithm can be modified to work in a system where only one node has all fair links. The two challenges are (1) that *accuse* messages can be lost, so we must broadcast and forward them and (2) that two processes may think that they are leader. We could forward *alive* messages, but this would mean all links must continue sending messages forever. Instead, a process that hears from two different leaders will tell them about each other and let them duel by accusing each other.

Algorithm 3 Implementation of Ω in system with fair links[1]

Initialization:

$\forall 0 \leq j < n, counter[j] = 0, phase[j] = 0$ ▷ Local view of all processes
$\forall 0 \leq j < n, i \neq j, timeout[j] = e + 1, timer[j] = -1$ ▷ Initialize timers
$active = \{i\}, leader = \perp$
while true **do**
    updateLeader()
    if $sendAliveTimer = 0$ **then**
        send $(alive, counter[i], phase[i])$ to all others
        $sendAliveTimer = e$
    end if
    for $0 \leq j < n, i \neq j$ **do**
        if receive $(alive, jcounter, jphase)$ from p_j **then**
            add j to $active$
            $counter[j] = \max\{counter[j], jcounter\}$ ▷ Do similar for $phase[j]$
            reset $timer[j]$ to $timeout[j]$
        end if
        if $timer[j] == 0$ **then**
            send $(accuse, phase[j])$ to p_j
            remove j from $active$
            increment $timeout[j]$
            turn off $timer[j]$ by setting to -1
        end if
        if receive $(accuse, phase)$ from p_j **then**
            if $phase == phase[j]$ **then**
                increment $counter[i]$
            end if
        end if
    end for
    if $sendAliveTimer > 0$ **then** decrement $sendAliveTimer$
    for $0 \leq j < n, i \neq j$ **do**
        if $timer[j] > 0$ **then** decrement $timer[j]$
    end for
end while
function UPDATELEADER
    $newLeader = argmin_\ell\{(counter[\ell], \ell) \mid \ell \in active\}$
    if $newLeader \neq leader$ **then**
        If the new leader is me, $sendAliveTimer = 0$ ▷ Start sending heartbeats
        If the old leader was me, increment $phase[i]$ and set $sendAliveTimer = -1$
            ▷ move to new phase to ignore old accusations, stop sending heartbeats
        $leader = newLeader$
end function

5 References

References

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Comput.*, 21(4):285–314, 2008.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 325–340. ACM, 1991.