Lecture 10: Failure Detectors

Aurojit Panda

1 Failure Detectors

1.1 Motivation/Why?

In a previous class we saw that the ability to assume partial synchrony makes it possible to solve some problems, including consensus. As a reminder, solving consensus here means that there exists a protocol that meets

- Agreement
- Validity
- Termination.

However, timing information (e.g. what is the latency bound, when is GST, etc.) changes by deployment, and in many cases reasoning about systems that rely on these assumptions is hard. Failure detectors are an alternative approach where rather than using timing information, there is an attempt to decide on exactly what additional information one needs.

1.2 Model

1.2.1 Processes

As usual we consider a set of processes Π connected by an asynchronous network.

1.2.2 Failure patterns

$F : \mathbb{N} \rightarrow 2^{\Pi}$ is a function from time to sets of processes.

For what follows we assume the fail-stop model, which in turn means that the set of failed processes can only grow, i.e., $t < t' \implies F(t) \subseteq F(t')$.

1.2.3 Failure detector (simplified)

We model failure detectors as libraries or modules that a process can query to find current failure information.

We are first going to start by talking about a simple class of failure detectors that can be queried to find a set of processes that are suspected to have **failed**.

Note, that by virtue of processes knowing the initial set of processes, this also means that a failure detector can be queried to determine the set of processes that are suspected to be functioning correctly.

1.2.4 System model/trace

We are going to continue with the model we have been using so far where each event consists of a process

- Receiving a message
- Performing some computation and updating its state
- Sending out 0 or more messages

To extend this model to make use of failure detectors we assume that a process queries its failure detector when it receives a message. So the new steps are now:

- Process receives a message
- Process queries failure detector
- Performing some computation and updating its state
- Sending out 0 or more messages

1.3 Failure detector properties

We will look at the capabilities of failure detectors in terms of their completeness and accuracy, as defined below.

1.3.1 Failure detector completeness

Completeness requirements look at the ability to detect failed nodes.

- Strong completeness: Eventually **every process** that has failed is permanently suspected by **every correct process**.

- Weak completeness: Eventually **every process** that has failed is permanently suspected by **some correct process**.

Note that the main distinction here lies in which correct processes detect a failure.

1.3.2 Failure detector accuracy

Accuracy requirements restrict when correct processes can be suspected.

- Strong accuracy: No **process is suspected before it crashes**. Note, that this means no **correct process** is suspected.
- Weak accuracy: Some **correct process** is never suspected. Note, this means that correct processes can be suspected, but there is at least **one** correct process is never suspected.

Note, that as opposed to completeness, the accuracy properties do not say "eventual". In reality we would like to allow for errors initially and things that eventually settle down:

- Eventual Strong accuracy: There is a time t after which no **correct process is suspected**.
- Eventual Weak accuracy: There is a time t after which some **correct process** is never suspected.

1.3.3 Failure Detector Classes

Given this we can get a whole bunch of failure detectors:

- Perfect: Strong completeness, strong accuracy. (P)
- Eventually perfect: Strong completeness, eventual strong accuracy. ($\diamond P$)
- Strong: Strong completeness, weak accuracy (S)
- Eventually strong: strong completeness, eventually weak accuracy. ($\diamond S$)
- Weak: Weak completeness, weak accuracy. (W)
- Eventually weak: Weak completeness, eventually weak accuracy ($\diamond W$).

Notice that the ones which are not eventual cannot

2 Reductions between failure detectors

2.1 Comparing failure detectors

It is often useful (e.g., in the paper you just read) to compare the "power" of different failure detectors. The particular definition we use in this case is whether one failure detector can be used to **implement** another.

We say a failure detector D can be reduced to D' if given D' there exists a distributed algorithm $T_{D' \to D}$ that can take the output of D' and compute D's output. We also say D' is reducible to D or $D' \succeq D$.

- We say D and D' are equivalent if D can be reduced to D', and D' can

be reduced to D.

- We say D is weaker than D' if D can be reduced to D' but not the other way, and write $D' \succ D$.

2.2 A simple comparison

2.2.1 Detectors with strong completness are reducible to detectors with weak completness

This holds from the definition of strong and weak completeness.

2.2.2 Detectors with weak completeness are reducible to detectors with string completeness

```
output_p = empty # Initial output
run in parallel(
    loop forever
        suspects = query_fd()
        broadcast({whoami(), suspects})
    loop forever
        receive do
           {q, suspects} -> output_p = union(output_p, suspects) - {q}
        end
```

- Why does this get from weak to strong completeness?

  output$_\text{p}$ is the union of suspects at all processes, which is a superset of the set of failed processes by definition

- Why does this preserve strong accuracy?

  Strong accuracy means a correct process is never suspected, hence never added to ‘$\mathrm{output}_\mathrm{p}$‘

- Why does this preserve weak accuracy?

  The chosen correct process is never added to $\mathrm{output}_\mathrm{p}$ due to the same argument as we used above.

- Why does this preserve eventual weak accuracy?

  Correct processes send failure detector updates infinitely often, hence the chose correct process is removed from $\mathrm{output}_\mathrm{p}$ infinitely often. Additionally after some time t it is never added back (by definition).

2.2.3 Consequence

Weak and strong completeness are equivalent.

3 Leader election and Omega

3.1 What do we want from leader election?

Choose a single leader: a single **correct** process that all nodes agree on. But this is very similar to the description of weak accuracy above.

3.2 Leader Election

This is **stronger** than the leader election we looked at last time, and is formally equivalent to the perfect failure detector P. Meets the following requirement:

- Eventual correctness: Eventually every correct process trusts some correct process which we label the leader.
- Agreement: No two correct processes disagree on leaders.
- Local accuracy: If a process p_i becomes leader, then all previously elected leaders have failed.

Note, that local accuracy is a stronger condition than is met by either of the leader election algorithms we looked at last week. This is in part due to the impossibility of implementing P in an asynchronous system.

3.3 Eventual leader election

Referred to as Ω, this is formally equivalent to the eventually perfect ($\diamond P$) failure detector, which is formally equivalent to $\diamond S$ (the eventually strong failure detector). Result in the paper you studied shows that in reality $\diamond P$, Ω and $\diamond W$ are all equivalent.

Anyways, what Ω provides is:

- Eventual correctness: Eventually every correct process trusts some correct process which we label the leader.
- Eventual agreement: After some time t, no two correct processes disagree on leaders.

Observe that local accuracy is gone in this case.

4 Using failure detectors

So far we have looked at how failure detectors add functionality to asynchronous systems. But how does this help solve consensus. Two approaches are briefly mentioned below.

4.1 Consensus using the strong failure detector

Just as a reminder, the strong failure detector S provides strong completeness and weak accuracy. Since we know that strong and weak completeness are equivalent, this protocol also works with the weak failure detector (which provides weak completeness, and weak accuracy).

As a further reminder weak accuracy requires that there is at least one correct process that is never suspected by any correct process. The key to the protocol is to use this processes view of proposed values to decide.

Put a different way, the existence of a process that will never be suspected means that there is a 1-process quorum that is safe.

```
wait_until_receive_or_fail(process, msg_pattern)
    # This is a function that waits until either a message
    # is received from each process p or process p is suspected
    # to have failed. We use this to wait for messages from
    # all n processes.

v_p = [null, null, ..., null] # Process p's estimate of proposed values.
```

```
v_p[p] = input # p knows its own proposal
d_p = v_p

# Phase 1: Proposal phase (This actually corresponds to phase 1 and 2 of the
# Chandra and Toueg paper.)
for r = 0 to n
   broadcast({r, d_p, p}) # Send a message with round numbers
   D = [[null, null, null, ..., null], ....] # Receive buffer
   for q in processes:
     {r, d, q} = wait_until_receive_or_fail(q, {r, d, q})
     D[q] = d
   for q in processes:
      if v_p[q] = null and exists q' s.t. D[q'][q] != null
         v_p[q] = D[q'][q]
   d_p[q] = v_p[q] # Exchange v_q again.

# Phase 2 decide
decide(first non-null component of v_p)
```

See Chandra and Toueg for proof of correctness, but the interesting thing here is that up to n-1 processes can fail.

4.2 Consensus using eventual strong failure detector

Just as a reminder, the eventually strong failure detector $\diamond S$ provides strong completeness and eventually weak accuracy. Since we know that strong and weak completeness are equivalent, this protocol also works with the eventually weak failure detector (which provides weak completeness, and eventually weak accuracy).

Eventual weak accuracy means that we eventually stop suspecting a process, i.e., one can eventually choose a correct leader.

The protocol here is precisely any of the leader election based protocols we have discussed before, with the failure detector used to trigger and decide on the leader.

5 Building Failure detectors

5.1 Classic approach: Timeouts

Processes send heartbeats, a lack of heartbeats for a certain duration of time causes processes to suspect each other. This is essentially the mechanism you have (or will) implement for Raft.

5.2 Falcon: Move away from eventual accuracy

What is we wanted to implement (weak or strong) accuracy, but still work in the asynchronous environment. Well the problem is that a lack of heartbeats can be due to delays or due to failure. One approach to resolve this uncertainty is to find a way to explicitly kill any process suspected of having failed.

How does this impact correctness?

5.3 Pigeon: Provide a bit more information, let application decide