LOCALIZED ENTITY FAILURES

Using Fault Detection & Preexecution Failures

LOCALIZED ENTITY FAILURES

Using Fault Detection

Introduction

The availability of any **reliable fault detector** in a **asynchronous system** would let to distinguish a slow entity from a failed entity. However, the problem in asynchronous system is how to construct reliable fault detectors.

In **fully synchronous system**, if messages are never lost, synchrony ensures that the absence of an anticipated message indicates a faulty sender.

In asynchronous systems:

- Do we need a completely reliable crash detector?
- What is the weakest detector we can usefully employ?

Asynchronous system : additional assumptions (FDA)

1. Connectivity, Bidirectional Links
2. The network is a complete graph
3. Entities have unique ids
4. Entities can fail only by crashing
5. Each entity knows the ids of its neighbors

Failure Detectors and Their Properties

A distributed detector of entity failures ia a set of n failure detection modules, one per entity. Each module keeps a list of entities it suspects to be faulty.

Failure detectors are defined in terms of the property they satisfy, instead of the actual implementation. We consider two properties:

Completeness

Accuracy

Completeness & Accuracy

Are both properties necessary?

Example: the trivial failure detector "PARANOID"

Each entity permanent suspect every other entity

- Paranoid will satisfy any completeness property
- Paranoid provides no real information about the actual failures

Example: the trivial failure detector "NAIVE"

Each entity never suspect any other entity

- Naive will satisfy any accuracy property
- Naive provides no real information about the failures

Completeness & Accuracy

Are both properties necessary?

Example: the trivial failure detector "PARANOID"

Each entity permanent suspect every other entity

- Paranoid will satisfy any completeness property
- Paranoid provides no real information about the actual failures

Example: the trivial failure detector "NAIVE"

Each entity never suspect any other entity

- Naive will satisfy any accuracy property
- Naive provides no real information about the failures

Types of Completeness & Accuracy

Completeness

- Strong Completeness
- Weak Completeness

Accuracy

- Perpetual Strong Accuracy
- Perpetual Weak Accuracy
- Eventual Strong Accuracy
- Eventual Weak Accuracy

Types of Completeness & Accuracy

Completeness

- Strong Completeness
- Weak Completeness

Accuracy

- Perpetual Strong Accuracy
- Perpetual Weak Accuracy
- Eventual Strong Accuracy
- Eventual Weak Accuracy

Perpetual strong accuracy is very difficult to achieve in many systems. However, even the perpetual weak accuracy may be a strong requirement for any failure detector to achieve, for example, because of an high network traffic and a too long delaying of messages. Fortunately, there are also the eventual strong accuracy and eventual weak accuracy.

Definition: Perfect Failure Detector

A failure detector is **perfect** if it satisfies the strong completeness and the perpetual strong accuracy.

In synchronous systems there is a perfect failure detector.

Theorem

Any failure detector that satisfies only weak completeness and eventual weak accuracy is sufficient for reaching consensus if at most f < $\frac{n}{2}$ entities can crash.

We denote by Ω any failure detector that satisfies only weak completeness and eventual weak accuracy.

The Weakest Failure Detector

What is the weakest detector?

Notations

F(t) : function describing the set of entities that have crashed through time. $\forall t\ F(t) \subseteq F(t+1)$

crashed(F) = $\bigcup_t F(t)$ denote the set of entities that crash under the failure pattern F

correct(F) = ϵ - crashed(F) denotes the set of entities that not crash under F

D(F) denote the set of all failure detector histories that can occur in executions with failure pattern F and failure detector D

$\mathbf{v}^r$ denote the history of the variable v during the execution r

Reduction Algorithm

Now we want to define an algorithm T to transform a failure detector D into another failure detector D'.

In particular, the algorithm T transforms D into D' (D $\geqslant$ D') if and only if, for every execution $\in$ of T using D, the histories of the variables output(x) in execution r can actually occur in some executions with detector D' and pattern F.

output(x) is a variable contained in every entity x that, using the failure detector D, emulates the output of D' in x.

Reduction Algorithm

How we can use actually the algorithm T?

Suppose that D $\geqslant$ D' (D is reducible to D') and supppose an algorithm A using failure detector D' to solve the problems but only D is available.

Then we can use the algorithm T in this way:

- Concurrently with A, entities run T to transform D into D'
- Whenever A requires that x query its failure detector module, x reads the current value of output(x) instead.

Example of Reduction Algorithm: REDUCE algorithm

The REDUCE algorithm transforms any given failure detector D that satisfies weak completeness into a failure detector D' that satisfies strong completeness.
Every entity x in REDUCE executes the following:

- **Initially**:
  output(x) $\leftarrow \oslash$
- **Repeatedly the entity x queries its local failure detector module** D_x :
  suspect(x) $\leftarrow D_x$
  send <x, suspects(x)> to N(x)
- **When receiving** <y, suspect(y)> **from** y:
  output(x) := output(x) $\bigcup$ suspect(y) - y

Properties satisfied by the REDUCE algorithm

Suppose H $\in$ D(F) denote an history of failure detector D for pattern F

Lemma : Transforming weak completeness into strong completeness

Let z be any entity that crashes; if eventually some correct entity permanently suspects y in H, then eventually all correct entities permanently suspect y in output$^{\in}$

Proof

Let y be any entity that crashes. Suppose that there is a time t after which some correct entity x permanent suspects y in H.
We must show that there is a time after which every correct entity suspects y in output$^{\in}$.

As y crashes, there is a time t' after which no entity receives a message from y.
Consider the execution by entity x after time t" = max(t,t').
Entity x sends a message of the type <x,suspects(x)> with y $\in$ suspects(x) to all entities. Eventually, every correct entity receives <x,suspects(x)> and adds y to its output.
As no correct entity receive any message from y after time t' and t" $\geqslant$ t', no correct entity removes y from its output after time t".
Thus, there is a time after which every correct entity permanently suspects y in output$^{\in}$.

■

Lemma: Preserving perpetual accuracy

Let y be any entity; if no entity suspects y in H before time t, then no entity suspects y in output$^{\in}$ before time t.

Lemma: Preserving eventual accuracy

Let y be any correct entity; if there is a time after which no correct entity suspects y in H, then there is a time after which no correct entity suspects y in output$^{\in}$.

So we can conclude that REDUCE transforms any failure detector D that satisfies weak completeness into a failure detector D' that satisfies strong completeness.

Theorem

REDUCE strengthens completeness while preserving accuracy.

Theorem

Ω is the weakest failure detector that can possibly be used to achieve consensus in presence of crash failures.

LOCALIZED ENTITY FAILURES

Preexecution Failures

Partial Reliability

Even in a complete graph if all the failure occur before the execution of the algorithm, the proof of the Single Failure Disaster does not hold because it relies heavily on the fact that the "adversary" of each entity can choose which entity fails as well as the moment when the failure occurs.

Instead of Total Reliability restriction ("No faults have occurred nor will occur"), we can suppose the **Partial Reliability** restriction: "No faults will occur during the computation tasks"

Example: Election in Complete Network

Consider the Election problem in a complete graph where some entities might have crash. Under Partial Reliability, it is possible to perform the election without synchrony, randomization and without fault detection even if f $\leq \lceil \frac{n}{2} \rceil$ - 1 have crashed.
We construct an algorithm based on the algorithm CompleteElect that was realized under Total Reliability.
Here is the algorithm **CompleteElect**:

1) A candidate entity x sends a Capture message to a neighbor y.

2) If y is candidate, the outcome of the attack depends on the stage and the id of the two entities:

(a) If **stage(x) > stage(y)**, the attack is successful.
(b) If **stage(x) = stage(y)**, the attack is successful if id(x) < id(y); otherwise x becomes passive.
(c) If **stage(x) < stage(y)**, x becomes passive.

3) If y is passive, the attack is successful.

4) If y is already captured, then x has to defeat y's owner z before capturing y. Specifically, a Warning message with x's id and stage is send by y to its owner z.

(a) **If z is a candidate in a higher stage**, or in the same stage but with a smaller id than x, then the attack to y is not successful: z will notify y that, in turn, will notify x.

(b) **In all other cases** (z is already passive or captured, z is a candidate in a smaller stage, or in the same stage but with a larger id than x), the attack to y is successful: z notifies x via y, and if candidate it becomes passive.

5) If the attack is successful, y is captured by x, x increments stage(x) and proceeds with its conquest.

The most important changes are essentially two:

1. **In the original protocol CompleteElect**, a candidate entity x starts by sending a "Capture" message to a single neighbor and waits for its reply.

   In this algorithm, because at most f entities have crashed, at the beginning x will send the "Capture" message to f + 1 entities. When x receives an "Accept" message, it enters the next stage and sends it message to another entity;

2. **In the original protocol Complete Elect**, a candidate entity x has only one pending "Capture" message and waits for its reply; if the reply is "Reject" x become passive.

   In this algorithm, instead, x has f + 1 pending "Capture".

Alert!!!

It may happen that while x waiting for the reply from y, x receives some "Accept" messages whose effect is to increase the stage number. So x must reject the "Reject" command.

Solution: x will send a new "Capture" message to y with its new stage number and close all its other ports waiting for the reply from y. When the reply arrives, x will increase the stage (if it's "Accept") or become passive (if it's "Reject") before reopening all the ports.

Correctness

For proving the correctness of the algorithm, we must consider the effects of the changes we have made on the original protocol.

In particular, in the second change, several entities may close all their ports but one and waiting for a reply from that one. To ensure correctness, we must prove that settlements must no create deadlocks.

Every request ("Capture" or "Warning") sent to a nonfaulty entity receives a reply. After a settlement (between x and y) at most one of them is still candidate and if so it has increased its stage.
If an entity x has an owner, then both x and its owner are nonfaulty. A nonfaulty entity always replies to a "Warning" message.

Lemma

Every entity eventually reaches stage greater than $\frac{n}{2}$, or it ceases to be a candidate.

Lemma

Assume an entity x ceases to be candidate as a result of a message originated by candidate y. Then, at any time after the time this message is processed by x, either the stage of y is greater than the stage of x or x and y are in the same stage, but $\mathrm{id}(x) < \mathrm{id}(y)$.

Lemma

At least one entity always remains a candidate.

Lemma

Let x be a candidate and s be its final size. The total number of times a capture message was sent by x is at most 2s + f.

Proof

When x initiates the algorithm, it sends f + 1 "Capture" messages. Every other "Capture" message it sends follows the reception of either an "Accept" or a "Reject" message.
The number of "Accept" messages it receives as a candidate is s - 1.
The number of "Reject" messages it receives as a candidate is at most s.

■

Costs

Lemma

For every l $\geqslant$ 2, if there are l - 1 candidates whose final size is not smaller that of a candidate x, then the stage of x is at most ln.

Let k denote the number of spontaneous initiators.
#messages for the leader announcement: n-1
A "Capture" message from x to y, in addition to the reply from y, will cause at most two additional messages: a "Warning" from y to its owner z and the reply.

$$\#\textbf{Total messages} \leq \textbf{n - 1 +} \ 4\#\textbf{Capture messages}$$

Let x be a candidate and s be its final size. Thanks to lemma 7.6.4 we know that the number of times this candidate ha sent a "Capture" message does not exceed 2s + f. Entities that didn't wake up spontaneously never become candidates.

$$\#\textbf{Total messages} \leq \textbf{n - 1 + 4} \sum_{1 \leq j \leq k} \textbf{(2} \frac{\textbf{n}}{\textbf{j}} \textbf{ + f)}$$

$$\textbf{M[FT - CompleteElect] = O(n log k + k f)}$$

Is this cost optimal?

The bound of FT-CompleteElect algorithm made up by two components: O(n log k) and O(k f).

Ω(n log k) messages are needed for election in complete networks even in the absence of failures.

Each initiator will send at least f + 1 messages so also Ω(k f) are needed. (k = #initiators)

Finally, we can see that the complete lowerbound Ω(n log k + k f) matches the upperbound O(n log k + k f).