Unreliable Failure Detectors for Reliable Distributed Systems

Mikel Larrea

Departamento de Arquitectura y Tecnología de Computadores

UPV / EHU

Contents

References

(1) *Unreliable Failure Detectors for Asynchronous Distributed Systems*

Tushar Deepak Chandra

PhD Thesis, Cornell University, May 1993. TR93-1377, Cornell University

(2) *Unreliable Failure Detectors for Reliable Distributed Systems*

Tushar Deepak Chandra and Sam Toueg

Journal of the ACM, 43(2): 225-267, March 1996

Introduction

Consensus is a fundamental problem of fault tolerant distributed computing (common denominator between many agreement type problems: atomic broadcast, group membership, atomic commitment, leader election, etc.)

Informally, Consensus allows processes to reach a common decision, which depends on their initial inputs, despite failures

We focus on solutions to Consensus in the *asynchronous* model of distributed computing: no timing assumptions

FLP Impossibility result (Fischer, Lynch, and Paterson, 1985): Consensus cannot be solved deterministically in an asynchronous system that is subject to even a single crash failure. Essentially, the impossibility stems from the inherent difficulty of determining whether a process has actually crashed or is only ‘very slow’

Introduction

To circumvent the FLP impossibility result, Chandra and Toueg propose to augment the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes (unreliable failure detector)

Consensus can be solved using a ‘perfect’ failure detector (one that does not make mistakes). But is perfect failure detection necessary to solve Consensus?

Possibility result (Chandra and Toueg, 1991): Consensus can be solved in asynchronous systems with unreliable failure detectors, even if they make an infinite number of mistakes

Certain failure detectors can be used to solve Consensus despite any number of crashes, while others require a majority of correct processes

Introduction

How much information about failures is necessary and sufficient to solve Consensus?

The *Eventually Weak Failure Detector* (◊*W*), a failure detector that provides surprisingly little information about which processes have crashed, is sufficient to solve Consensus in asynchronous systems with a majority of correct processes

Moreover, to solve Consensus, any failure detector has to provide at least as much information about failures as ◊*W*. Thus, ◊*W* is indeed the *weakest* failure detector for solving Consensus in asynchronous systems with a majority of correct processes

Reference: *The Weakest Failure Detector for Solving Consensus.* T.D. Chandra, V. Hadzilacos, and S. Toueg. Journal of the ACM, 43(4): 685-722, July 1996

System Model

Asynchronous distributed system: there is no bound on message delay, clock drift, or the time necessary to execute a step

The system consists of a finite set of processes:

$$\Pi = \{p_1, p_2, \ldots, p_n\}$$

Message passing model. Every pair of processes is connected by a *reliable* communication channel

Processes can fail by *crashing*. Once a process crashes, it does not recover

An algorithm *A* is a collection of *n* deterministic automata, one for each process in the system. Computation proceeds in *steps* of *A*. In each step, a process $p_i \in \Pi$ may (1) send a message to a single process, (2) receive a message that was sent to it, (3) perform some local computation (e.g., query its failure detector module), or (4) fail

System Model

A run is an infinite execution of the system. Given any run σ, *crashed(t, σ)* is the set of processes that have crashed by time *t* in σ, and *correct(t, σ)* = Π – *crashed(t, σ)*

$$\mathit{crashed}(\sigma) = \cup_t\, \mathit{crashed}(t, \sigma)$$
$$\mathit{correct}(\sigma) = \Pi - \mathit{crashed}(\sigma)$$

If $p \in$ *correct(σ)* then *p* is *correct* in σ. Otherwise, we say that *p* is *faulty* in σ, and $p \in$ *crashed(σ)*. We consider only runs with at least one correct process, i.e., *correct(σ)* $\neq \emptyset$

Failure Detectors

A failure detector is a distributed oracle that provides hints about the operational status of other processes

Each process $p \in \Pi$ has access to a local failure detector module D_p. Each local failure detector module monitors a subset of the processes in the system, and maintains a list of those that it currently suspects to have crashed

Each failure detector module can make mistakes by erroneously adding processes to its list of suspects. If it later believes that suspecting a given process was a mistake, it can remove this process from its list. At any given time, the modules at two different processes may have different lists of suspects

The mistakes made by an unreliable failure detector should not prevent any correct process from behaving according its specification, even if that process is (erroneously) suspected to have crashed by all the other processes

Properties of Failure Detectors

Failure detectors are abstractly characterised in terms of two properties: *completeness* and *accuracy*

Completeness characterises the degree to which crashed processes are permanently suspected by correct processes

Accuracy restricts the false suspicions that a failure detector can make

Strong completeness: Eventually every process that crashes is permanently suspected by *every* correct process

$$\forall \sigma, \forall p \in crashed(\sigma), \forall q \in correct(\sigma), \exists t, \forall t' \geq t: p \in D_q(t', \sigma)$$

Weak completeness: Eventually every process that crashes is permanently suspected by *some* correct process

$$\forall \sigma, \forall p \in crashed(\sigma), \exists q \in correct(\sigma), \exists t, \forall t' \geq t: p \in D_q(t', \sigma)$$

Properties of Failure Detectors

Completeness by itself is not a useful property: a failure detector may trivially satisfy this property by always suspecting *all* the processes in the system. To be useful, a failure detector must also satisfy some accuracy requirement

(Perpetual) Accuracy

Strong accuracy: No process is suspected before it crashes

$$\forall \sigma, \forall t, \forall p, q \in \Pi - crashed(t, \sigma): p \notin D_q(t, \sigma)$$

Weak accuracy: Some correct process is never suspected

$$\forall \sigma, \exists p \in correct(\sigma), \forall q \in \Pi, \forall t: p \notin D_q(t, \sigma)$$

Obviously, accuracy by itself is neither useful (e.g., “never suspect any process” trivially satisfies strong accuracy)

Properties of Failure Detectors

Eventual Accuracy

Even weak accuracy guarantees that at least one correct process is never suspected. Since this type of accuracy may be difficult to achieve, we consider failure detectors that may suspect *every* process at one time or another. Informally, we only require that strong accuracy or weak accuracy are *eventually* satisfied

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process

$$\forall \sigma, \exists t, \forall p, q \in correct(\sigma), \forall t' \geq t: p \notin D_q(t', \sigma)$$

Eventual weak accuracy: There is a time after which some correct process is never suspected by any correct process

$$\forall \sigma, \exists t, \exists p \in correct(\sigma), \forall q \in correct(\sigma), \forall t' \geq t: p \notin D_q(t', \sigma)$$

Classes of Failure Detectors

Strong completeness:	Eventually every process that crashes is permanently suspected by *every* correct process
Weak completeness:	Eventually every process that crashes is permanently suspected by *some* correct process
Strong accuracy:	No process is suspected before it crashes
Weak accuracy:	Some correct process is never suspected
Eventual strong accuracy:	There is a time after which correct processes are not suspected by any correct process
Eventual weak accuracy:	There is a time after which some correct process is never suspected by any correct process

Completeness	**Accuracy**			
	Strong	Weak	Eventual strong	Eventual weak
Strong	*Perfect* *P*	*Strong* *S*	*Eventually Perfect* ◊*P*	*Eventually Strong* ◊*S*
Weak	*Quasi-Perfect* *Q*	*Weak* *W*	*Eventually Quasi-Perfect* ◊*Q*	*Eventually Weak* ◊*W*

Implementation of Failure Detectors

Can ◊W be implemented in an asynchronous system?

Most implementations of failure detectors are based on some timeout mechanism. The definition of ◊*W* must be seen as a specification for the implementation of this mechanism: the timeout value chosen should be as small as possible (if fast reaction to process crash is required), but not too small, to guarantee the properties of ◊*W* with a probability close to 1

One possible implementation of ◊*W* could be the following:

“Every process *q* periodically sends a ‘*q* is alive’ message to all. If a process *p* times out on some process *q*, it adds *q* to its list of suspects. If *p* later receives a ‘*q* is alive’ message, *p* recognises that it made a mistake by prematurely timing out on *q*: *p* removes *q* from its list of suspects, and increases the length of its timeout period for *q* in an attempt to prevent a similar mistake in the future”

Failure Detectors: Reducibility

A failure detector *D* is said to be *stronger* than a failure detector *D'* (written $D \geq D'$) if there is a distributed algorithm $T_{D \to D'}$ that can transform *D* into *D'*. Failure detector *D'* is said to be *reducible* to *D* (*D* provides at least as much information about failures as *D'* does)

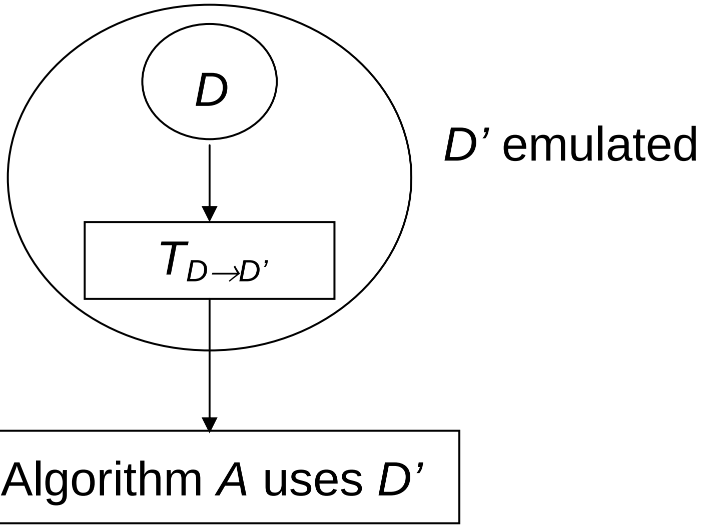
The following relations are obvious (by definition):

$$P \geq Q$$
$$S \geq W$$
$$\Diamond P \geq \Diamond Q$$
$$\Diamond S \geq \Diamond W$$

Given a reduction algorithm $T_{D \to D'}$, any problem that can be solved using failure detector *D'*, can be solved using *D* instead

Failure Detectors: Reducibility

Suppose a given algorithm *A* requires failure detector *D’*, but only *D* is available. We can still execute *A* as follows. Concurrently with *A*, processes run $T_{D \rightarrow D'}$ to transform *D* into *D’*



Transforming *D* into *D’*

Failure Detectors: Reducibility

From weak completeness to strong completeness, preserving accuracy

Every process p executes the following:

```
$output_p \leftarrow \emptyset$                                   {$output_p$ emulates $D'_p$}

cobegin
|| Task 1: repeat forever
        {$p$ queries its local failure detector module $D_p$}
        $suspects_p \leftarrow D_p$
        send ($p$, $suspects_p$) to all

|| Task 2: when receive ($q$, $suspects_q$) for some $q$
        $output_p \leftarrow (output_p \cup suspects_q) - \{q\}$
coend
```

$T_{D \rightarrow D'}$: From weak completeness to strong completeness

Failure Detectors: Reducibility

By the previous reduction algorithm, we have:

$$Q \geq P$$
$$W \geq S$$
$$\Diamond Q \geq \Diamond P$$
$$\Diamond W \geq \Diamond S$$

Two failure detectors are *equivalent* if they are reducible to each other. Thus, every failure detector with weak completeness is actually equivalent to the corresponding failure detector with strong completeness:

$$Q \cong P$$
$$W \cong S$$
$$\Diamond Q \cong \Diamond P$$
$$\Diamond W \cong \Diamond S$$

Failure Detectors: Comparison

Comparing failure detectors by reducibility

$$\begin{tikzcd}
Q \arrow[rr, dashed] \arrow[dd, dashed] \arrow[dr, equal] & & \Diamond Q \arrow[dd, dashed] \arrow[dr, equal] & \\
& P \arrow[rr, dashed] \arrow[dd, dashed] & & \Diamond P \arrow[dd, dashed] \\
W \arrow[rr, dashed] \arrow[dr, equal] & & \Diamond W \arrow[dr, equal] & \\
& S \arrow[rr, dashed] & & \Diamond S
\end{tikzcd}$$

$D \rightarrow D'$: D is strictly stronger than D'

D — D': D is equivalent to D'

Reliable Broadcast

Reliable Broadcast is a communication primitive that satisfies the following properties:

Validity: If a correct process R_broadcasts a message *m*, then it eventually R_delivers *m*

Agreement: If a correct process R_delivers a message *m*, then all correct processes eventually R_deliver *m*

Uniform Integrity: For any message *m*, every process R_delivers *m* at most once, and only if *m* was previously R_broadcast by *sender*(*m*)

Implementation of Reliable Broadcast

Reliable Broadcast is defined in terms of two primitives, *R_broadcast(m)* and *R_deliver(m)*, where *m* is the message to be broadcast

Every process p executes the following:

```
To execute R_broadcast(m):
    send m to all (including p)

R_deliver(m) occurs as follows:
    when receive m for the first time
        if sender(m) ≠ p then send m to all
        R_deliver(m)
```

Reliable Broadcast by message diffusion

The Consensus Problem

In the Consensus problem, every process proposes an input value, and correct processes (those that do not crash) must eventually decide on some common output value

We define the Consensus problem in terms of two primitives, *propose(v)* and *decide(v)*. The Consensus problem is specified as follows:

Termination: Every correct process eventually decides some value

Uniform Integrity: Every process decides at most once

Agreement: No two correct processes decide differently

Uniform Validity: If a process decides *v*, then *v* was proposed by some process

Solving Consensus using Unreliable Failure Detectors

By equivalence between failure detectors, we only need to solve Consensus using each one of the four classes of failure detectors that satisfy strong completeness, namely, *P*, *S*, ◊*P*, and ◊*S*

Two algorithms:

(1) Solving Consensus using a Strong failure detector *S*. Since by definition $P \geq S$, this algorithm also solves Consensus using a Perfect failure detector *P*

(2) Solving Consensus using an Eventually Strong failure detector ◊*S*. Since by definition $\Diamond P \geq \Diamond S$, this algorithm also solves Consensus using an Eventually Perfect failure detector ◊*P*

Solving Consensus using Unreliable Failure Detectors

Solving Consensus using a Strong Failure Detector (S)

S: strong completeness, weak accuracy. Eventually every process that crashes is permanently suspected by *every* correct process. Some correct process is never suspected

The algorithm tolerates up to n - 1 faulty processes. It runs through 3 phases: a proposition phase, an agreement phase, and a decision phase

By $W \cong S$, given any Weak Failure Detector *W*, Consensus is solvable in asynchronous systems with $f < n$ (f is the maximum number of processes that may crash)

Solving Consensus using Unreliable Failure Detectors

Solving Consensus using an Eventually Strong Failure Detector (◊S)

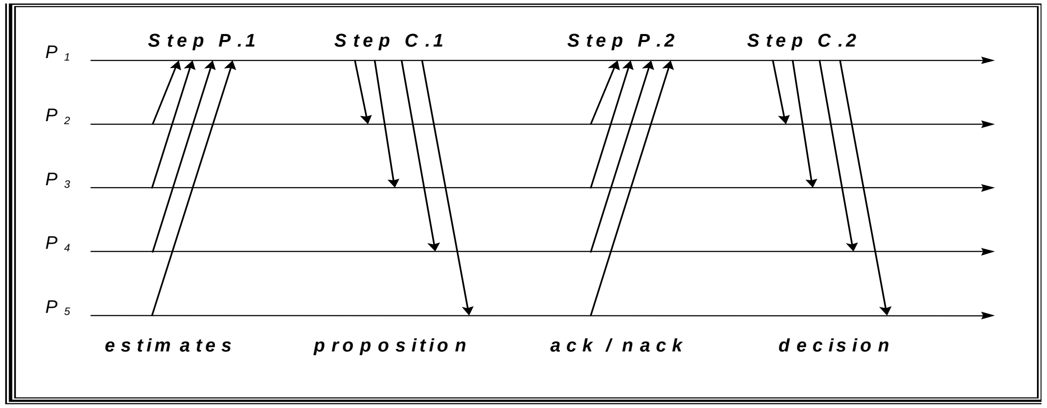
◊*S*: strong completeness, eventual weak accuracy. Eventually every process that crashes is permanently suspected by *every* correct process. There is a time after which some correct process is not suspected by any correct process

The algorithm uses the *rotating coordinator paradigm*, and it proceeds in asynchronous rounds. In each round, all messages are either to or from the 'current' coordinator. Every time a process becomes a coordinator, it tries to determine a consistent decision value. If the current coordinator is correct *and* is not suspected by any correct process, then it will succeed, and it will R_broadcast the decision value

Each round of the algorithm is divided into four asynchronous phases: a voting phase, a proposition phase, an acknowledgement phase, and a decision phase

Solving Consensus using Unreliable Failure Detectors

Solving Consensus using an Eventually Strong Failure Detector (◊S)



The algorithm goes through three asynchronous epochs, each of which may span several asynchronous rounds. In the first epoch, several decision values are possible. In the second epoch, a value gets *locked*: no other decision value is possible. In the third epoch, processes decide the locked value

By ◊*W* ≅ ◊*S*, given any Eventually Weak Failure Detector ◊*W*, Consensus is solvable in asynchronous systems with a majority of correct processes ($f < \lceil n/2 \rceil$)

Conclusions

Advantages of the failure detectors approach

It is a ‘clean’ extension of the asynchronous model

It has been used to determine the minimal information about failures necessary to solve Consensus

Lower bounds on fault tolerance: failure detectors with perpetual accuracy can be used to solve Consensus in asynchronous systems with *any* number of failures. In contrast, with failure detectors with eventual accuracy, Consensus can be solved if and only if a *majority* of the processes are correct

Algorithms based on ◊*W* (the weakest failure detector considered) always preserve *safety*: if an algorithm assumes a failure detector with the properties of ◊*W*, but the failure detector that it actually uses fails to meet these properties, the algorithm may lose its liveness properties, but its safety properties will never be violated

Conclusions

Disadvantage of the failure detector approach

Algorithms are harder to design, because they must be aware of (and deal with) the mistakes that the failure detector can make