DoC 437 - 2008

Distributed Algorithms

Part 6: Failure Detectors

Consensus in asynchronous systems

an impossibility

- Fischer, Lynch, and Paterson (1985)

  impossible to solve the consensus problem for an asynchronous system in the presence of even a single process failure

  intuition: a "failed" process may just be slow, and can inject a message at exactly the wrong time, thereby disrupting the state of agreement

- The FLP impossibility result extends to

  reliable ordered multicast communication in groups

  transaction commit for coordinated atomic updates

  consistent replication

How do we get around this?

- Weaken the problem

  randomization, leading to probabilistic consensus

  or

  admit to multiple agreed values, not just one

- Strengthen the model

  synchronization

  or

  use *failure detectors*

Failure detection

practical motivation

- Most distributed execution environments provide clocks and timers in some way

  theoretical studies reveal for what tasks these primitives are necessary and to what degree they can be used, but are they absolutely required?

- These environments are also often designed to return an error message upon an attempt to communicate with a crashed process

  however, such error messages are not always reliable

  so, how reliable do they need to be in order to usefully solve real problems such as consensus?

Failure detectors

- A *failure detector* is a module that provides to each process a collection of *suspected* processes

  predicate $j \in D$ is true if j is suspected of being in a crashed state at the moment of evaluation

  (assumed failure model: non-recovery after halt)

- Detectors in different processes need not agree

  correct p may suspect r while correct q does not

  and need not be just at any time

  correct p may suspect correct r or not suspect failed r

Reasoning about failure detectors

- To be able to reason about algorithms that use failure detectors we must express the properties of the detectors, particularly the relation between the detector output and actual failures

- Actual failures are expressed in a *failure pattern*

  the pattern uses a notion of "time", but this is not an observed time, simply a means for reasoning

Reasoning about failure detectors

basic definitions

- A *failure pattern* is a function $F : T \rightarrow 2^P$, where T is a set representing all time instances and 2^P is the power set of the set of processes P

  $F(t)$ is the collection processes that have crashed at time t

  no recovery, so $t_1 \leq t_2 \Rightarrow F(t_1) \subseteq F(t_2)$

  $Crash(F) = \cup_{t \in T} F(t)$, the set of defective processes

  $Corr(F) = P \setminus Crash(F)$, the set of correct processes

Reasoning about failure detectors

basic definitions

- Suspicions may differ from time to time and from process to process, so we model *suspected processes* by a function $H : P \times T \rightarrow 2^P$

  $H(q,t)$ is the collection of processes suspected by q at time t

- To allow non-deterministic failure detectors (i.e., so a given failure pattern can lead to different responses), we model a *detector* D as a mapping from failure patterns to collections of failure detector *histories*

  $H \in D(F)$ is a failure detector history for pattern F

Reasoning about failure detectors

properties of histories

- For a failure detector to be useful, there must be a relationship between its output (a history) and its input (a pattern), modeled as properties

- *Completeness:* the detector will suspect crashed processes

    bounds the set of suspect process "from below"

- *Accuracy:* the detector will not suspect correct processes

    bounds the set of suspected process "from above"

Basic detector classes

completeness, formally

- Failure detector *D* is (strongly) *complete* if every crashed process will eventually be suspected by every correct process

$$\forall F : \forall H \in D(F) : \exists t : \forall p \in Crash(F) : \forall q \in Corr(F) : \forall t' \geq t : p \in H(q,t')$$

Basic detector classes

accuracy, formally

- Failure detector D is *strongly accurate* if no process is ever suspected if it has not crashed

$$\boxed{\forall F : \forall H \in D(F) : \forall t : \forall p,q \notin F(t) : p \notin H(q,t)}$$

- Failure detector D is *weakly accurate* if there exists a correct process that is never suspected

$$\boxed{\forall F : \forall H \in D(F) : \exists p \in Corr(F) : \forall t : \forall q \notin F(t) : p \notin H(q,t)}$$

Basic detector classes

accuracy, formally

- Failure detector *D* is *eventually strongly accurate* if there exists a time after which no correct process is suspected

$$\forall F : \forall H \in D(F) : \exists t : \forall t' \geq t : \forall p,q \in Corr(F) : p \notin H(q,t')$$

- Failure detector *D* is *eventually weakly accurate* if there exists a time and a correct process that is not suspected after that time

$$\forall F : \forall H \in D(F) : \exists t : \exists p \in Corr(F) : \forall t' \geq t : \forall q \in Corr(F) : p \notin H(q,t')$$

Basic detector classes

completeness vs. accuracy

- Easy to have a complete detector

  one that suspects every process: $H(q,t) = P$

- Easy to have an accurate detector

  one that suspects no process: $H(q,t) = \emptyset$

- Interesting and useful detectors combine (or compromise) between completeness and accuracy

Basic detector classes

- A failure detector is said to be…

  perfect if complete and strongly accurate

  strong if complete and weakly accurate

  eventually perfect if complete and eventually strongly accurate

  eventually strong if complete and eventually weakly accurate

Why do we need detectors?

- Basic structure of an asynchronous algorithm
  1. each process performs a "shout"

     (sends a message to every other process)
  2. each process collects $n - f$ of the shouted messages

     (f is the number of failed processes; $f \leq F$, limit on failures)
- A process should never wait for the arrival of *more than* $n - F$ messages or it risks deadlock in the case of crashes (due to "external" blocking)
- A process should never wait for a message from a *specific process* because it may become blocked if that specific sender has crashed

Why do we need detectors?

observation

- Even in the absence of failures, the basic construction allows different correct processes to collect different sets of messages

  each process is free to act once $n - F$ messages arrive

  although they may have collected the same number of messages, those messages may have been sent by different processes

How do we make use of detectors?

- Communication using failure detectors

  1. each process performs a "shout"
  2. each process waits until a message arrives from process q or q becomes suspected

- Completeness ensures freedom from external blocking

  each process that does not send a message because of having crashed will eventually be suspected

- *Pitfall:* But still vulnerable to different correct processes receiving different sets of messages

Pitfall scenarios

- Scenario 1

  process *q* crashes, but prior to its crash sends some or all of the required messages

  process p_1 receives its message before suspecting *q*, but p_2 does not receive its message and ends its receive phase without the message from *q*

- Scenario 2

  process *q* is correct and sends all its messages

  process p_1 suspects *q* and does not collect its message, while p_2 does not suspect *q* and collects its message

Accuracy helps a bit

- A set of collected messages may include a message from a crashed process, as well as not include one from a correct process
- The size of the set may differ and even fall below $n - F$ due to erroneous suspicions
- Fortunately, the accuracy property provides a bound on the suspected processes
- These considerations lead to a common idiom

  “collect <message> from q”

  waits until message received or q is suspected

Consensus, revisited

- The basic model

  n processes using an asynchronous, message-based network with perfect links, but processes may crash

- The desired outcome

  termination: every correct process decides once

  agreement: all decisions are equal

  validity: decision originates from at least one process

Recall the FLP result: impossible without detectors

Using strong detectors

rotating coordinators algorithm

```
each process i executes:

read x
for r := 1 to n do
  if (i = r) then
    send [VAL: x,r] to all processes in G
  if (collect [VAL: v,r] from G[r]) then
    x := v
decide x
```

- How does this work?

  there exists at least one process that is never suspected, but the processes do not know which

  only when the message from this process is received, do they all receive the same information

Using strong detectors

correctness proof sketch

- Terminates because no correct process blocks forever in a round

  coordinator of a round is correct or eventually suspected (due to completeness)

- Valid because a process can only keep its value or replace it by a value received from some coordinator

- Agreement is reached because if correct process is coordinator of round *j*, then all processes receive its value, and any coordinators of rounds after *j* will only send that value

Using strong detectors

observations

- Notice that there is no "resiliency" factor F mentioned in the algorithm

    correct coordinators can be suspected

    so, it does not help to know that processes are correct, just that they are unsuspected

- Bounded exactly by n

    weak accuracy implies that there is round with unsuspected process among first n rounds, but unknown which round

    can reduce the number of rounds only by strengthening the detector (e.g., eventually perfect)

Using eventually strong detectors

- Eventual weak accuracy implies that a round with an unsuspected coordinator will occur at some time, but does not provide a bound on this

- So, a solution using eventually strong detectors must also detect when agreement has been reached

- Detecting agreement requires that the algorithm provide a bound on resiliency

What is the required resiliency?

- Theorem

  there exists no consensus algorithm using eventually perfect detectors that allows $n/2$ or more failures (i.e., F must be $< n/2$)

- Informal proof (by contradiction)

  construct three disjoint subsets of the processes, two of which are of size $n - F$ and consist of correct processes, and the other of size F whose processes all fail; the correct subsets initially suspect each other

  each correct subset can receive and decide different values, violating the property of agreement

Using eventually strong detectors

rotating coordinators algorithm: $F < n/3$

```
each process i executes:

read x  r := 0  m := 0
while (true) do
  r := r + 1
  c := (r mod n) + 1
  send [VAL: x,r] to G[c]
  if (i = c) then
    upon (receipt of N-F [VAL: xj,r] messages)
    v := majority(∀j: xj)
    d := (∀j: xj = v)
    send [OUTCOME: d,v,r] to ∀j
  if (collect [OUTCOME: d,v,r] from G[c]) then
    x := v
    if (d) then
      decide x
      if (m = c) then exit
                  else if (m = 0) then m := c
```

Implementing failure detectors

- Principal advantage of failure detectors
  - must only deal with *properties* of detectors, not their implementation
- Implementation is typically a wrapping of an asynchronous interface around the careful use of timers (i.e., synchronization mechanisms)
  - "I'm alive" messages
  - adaptive estimates of "bounded" delay