5. Failure Detectors and Consensus

Local Failure Detection

- i.e., detection of failed processes from inside a host
- Detection can be considered reliable
  - Why?
- Various mechanisms:
  - Self-checking routines (e.g. parity checks)
  - Guardian components (e.g. memory access checkers)
  - Watch-dogs (hardware-based, software-based)
- Temporal problems may still affect accuracy
  - Why?

- *(from now on: distributed failure detection)*
- *(in this lecture: failure = crash)*

Basic Failure Detectors: Ping

```
Alyssons-MBP:~ anbessani$ ping www.google.com
PING www.google.com (172.217.16.228): 56 data bytes
64 bytes from 172.217.16.228: icmp_seq=0 ttl=56 time=12.662 ms
64 bytes from 172.217.16.228: icmp_seq=1 ttl=56 time=13.100 ms
64 bytes from 172.217.16.228: icmp_seq=2 ttl=56 time=13.238 ms
64 bytes from 172.217.16.228: icmp_seq=3 ttl=56 time=13.011 ms
64 bytes from 172.217.16.228: icmp_seq=4 ttl=56 time=12.356 ms
64 bytes from 172.217.16.228: icmp_seq=5 ttl=56 time=13.119 ms
64 bytes from 172.217.16.228: icmp_seq=6 ttl=56 time=14.157 ms
64 bytes from 172.217.16.228: icmp_seq=7 ttl=56 time=13.245 ms
64 bytes from 172.217.16.228: icmp_seq=8 ttl=56 time=72.393 ms
^C
--- www.google.com ping statistics ---
9 packets transmitted, 9 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 12.356/19.698/72.393/18.636 ms
Alyssons-MBP:~ anbessani$
```

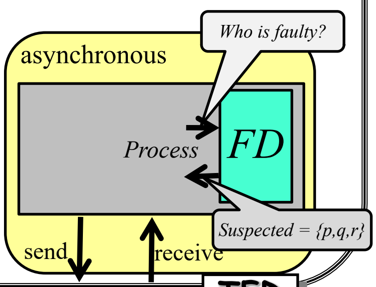
- Typically ICMP: echo-request, echo-reply

Engineering Failure Detection

- Heartbeat mechanism:
  - Observed component periodically sends messages
  - Usually called “I'm alive” messages or “heartbeats”
  - When these messages are not received, the component is considered faulty
- Probe mechanism (similar to a ping):
  - Observed component waits for a probe message and replies
  - Probe comes from the failure detector
  - Component replies with “I'm alive" message
- Which one is better?

Failure Detectors (FDs)

- Ping provides suspicions (**hints**) of failures (crashes)
- Conceptually, FDs are **distributed oracles** that provide hints about process failures
  - local modules attached to processes
  - modules may communicate using the network
- Using FDs allow the specification of algorithms that does not express timing assumptions
  - System can be asynchronous
    - (no time assumptions)
  - But failures can be detected
    - (through FD module)
  - Failure detection is **modularized**
  - Separation of concerns!



Recalling Safety and Liveness

- Safety
  - The measure in which a service/program does not do bad things
  - Safety properties specify that wrong events never take place
  - A safety property specifies that a predicate *P* is always true
- Liveness
  - The measure in which a service/program does good things
  - Liveness properties specify that good events eventually take place
  - A liveness property specifies that predicate *P* will eventually be true

(Recall the properties we used to define reliable broadcast, etc.)

Failure detection
properties and problems

- Consistency of distributed failure detection:
  - A fundamental problem in asynchronous distributed systems **is how to differentiate a slow process from a crash**
  - Ideally, when a process goes down all the other processes must know about it to coordinate their actions to take corrective actions
  - Such FD consistency can be required for **safety** or **liveness**
    - What's the difference?

- One of the biggest issues when designing a practical distributed system is if we can reliably detect process failures
  - And, if not, how we deal with misdetections?

Failure detection
properties and problems

Properties of Consistent Failure Detection

- Strong Accuracy
  - A **safety** requirement, specifying that no correct process is ever considered faulty
- Strong Completeness
  - A **liveness** requirement, specifying that a failure must be eventually detected by every correct process

Is it possible to satisfy these properties on the internet?

Failure detection
properties and problems

- **On the possibility of implementing strong accuracy and completeness…**

- Assuming that message processing requires negligible time, processes have clocks with bounded drift, and "**perfect channels**" are available, heartbeat/probe exchanges meet strong accuracy and completeness
  - Perfect channel = synchronous + reliable
- Such a detector is called perfect failure detector:
  - If a process crashes, all correct processes will note the absence of the heartbeat and detect the failure
  - No correct process is ever suspected

Failure detection
properties and problems

- What happens when the channels are not perfect?
  - Either imperfection can be fixed by some simple protocol
  - Or imperfection is impossible to overcome

- Communication channel is not perfect but has some mild imperfection:
  - For instance, it makes at most *k* consecutive omissions
  - Solution: transform the imperfect channel into a perfect channel
  - E.g., each heartbeat can be retransmitted *k + 1* times, effectively ensuring that it is observed by all correct processes.

Failure detection
properties and problems

- Channel imperfection impossible to overcome:
  - Lack of bounds on the number and type of faults that may happen (e.g., number of channel omission faults not bounded)
- Consequence:
  - Now, if a process does not receive any heartbeat message from the other process this may have two causes:
    - Because the other process is failed or
    - Because the channel has dropped all heartbeats sent so far!!!

Failure detection
properties and problems

- Channel imperfection impossible to overcome:
  - Lack of bounds for the timely behavior of system components (processes or links) – *asynchrony*
- Consequence:
  - No way to distinguish missing from "extremely slow" heartbeats
  - Happens if a link can delay a message arbitrarily, or if a process can take an arbitrary amount of time to make a processing step
  - Perfect failure detection cannot be implemented in asynchronous systems!!!

Hierarchy of FDs

- Chandra and Toueg (1992 and 1996) have defined the notion of **unreliable FD** and a hierarchy of such FDs

- Perfect failure detector $\mathcal{P}$
  - Strong Accuracy. No process is suspected before it crashes.
  - Strong Completeness. Eventually every process that crashes is permanently suspected by every correct process.

Failure detection
properties and problems

These properties can be relaxed:

- Weak Accuracy
  - At least one correct process is never suspected by all correct processes
- Weak Completeness
  - Eventually every process that crashes is permanently suspected by at least one correct process

Even Weak Accuracy is impossible in non-synchronous systems!
Therefore:

- Eventual Strong Accuracy
  - There is a time after which all correct processes are never considered failed by any correct processes
- Eventual Weak Accuracy
  - There is a time after which some correct process is never considered failed by any correct processes

Eventually weak FD: $\Diamond\mathcal{W}$

- The weakest FD of the Chandra-Toueg hierarchy is define by the following properties:
- Eventual Weak Accuracy
  - There is a time after which some correct process is never considered failed by any correct processes
- Weak Completeness
  - Eventually every process that crashes is permanently suspected by at least one correct process
- Consensus is solvable in an asynchronous system with an eventually weak FD
  - Notice that this system model is stronger than the asynchronous system model; the eventually weak FD is not implementable in the latter!
  - **It is impossible to solve consensus with a weaker detector**, as proved by Chandra, Hadzilacos and Toueg (1996)

Hierarchy of FDs

- Note:
  - the diamond stands for "eventually"
  - *S* is an important FD for "practical" synchronous systems

Completeness	**Accuracy**			
	Strong	Weak	Eventually strong	Eventually weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually weak* $\diamond\mathcal{W}$

Weakest FD revisited

Completeness	Accuracy: Strong	Weak	Eventually strong	Eventually weak
Strong	Perfect $\mathcal{P}$	Strong $\mathcal{S}$	Eventually perfect $\diamond\mathcal{P}$	Eventually Strong $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	Weak $\mathcal{W}$	$\diamond\mathcal{Q}$	Eventually weak $\diamond\mathcal{W}$

w/ crash faults it is trivial to obtain **Strong Completeness** from **Weak Completeness** (so using $\diamond\mathcal{S}$ is more or less the same as $\diamond\mathcal{W}$):

Here is a transformation

```
output_p ← ∅

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

|| Task 2: when receive (q, suspects_q) for some q
        output_p ← (output_p ∪ suspects_q) − {q}          {output_p emulates D'_p}
coend
```

 TFD

Summary: FD classes and properties

Completeness	Accuracy: Strong	Weak	Eventually strong	Eventually weak
Strong	Perfect $\mathcal{P}$	Strong $\mathcal{S}$	Eventually perfect $\diamond\mathcal{P}$	Eventually Strong $\diamond\mathcal{S}$
~~Weak~~	~~$\mathcal{Q}$~~	~~Weak $\mathcal{W}$~~	~~$\diamond\mathcal{Q}$~~	~~Eventually weak $\diamond\mathcal{W}$~~

- Strong Completeness. Eventually every process that crashes is permanently suspected by every correct process.
- Strong Accuracy. No process is suspected before it crashes.
- Weak Accuracy. At least one correct process is never suspected by all correct processes
- Eventual Strong Accuracy. There is a time after which all correct processes are never considered failed by any correct processes
- Eventual Weak Accuracy. There is a time after which some correct process is never considered failed by any correct processes

 TFD

Implementing a FD of class $\Diamond\mathcal{P}$

(that also implements weaker FDs: $\Diamond S$ and $\Diamond\mathcal{W}$)

- Consider the following system model:
  - Partial synchrony
    - For every execution there is a Global Stabilization Time (GST) after which there are bounds on the relative process speeds and communication delays (both GST and the bounds are unknown)
  - Reliable channels
  - Processes can fail by crashing

Implementing a FD of class $\Diamond\mathcal{P}$

```
Every process p executes the following:

output_p ← ∅
for all q ∈ Π                          {Δ_p(q) denotes the duration of p's time-out interval for q}
    Δ_p(q) ← default time-out interval

cobegin
|| Task 1: repeat periodically
    send "p-is-alive" to all

|| Task 2: repeat periodically
    for all q ∈ Π
        if q ∉ output_p and
            p did not receive "q-is-alive" during the last Δ_p(q) ticks of p's clock
            output_p ← output_p ∪ {q}          {p times-out on q: it now suspects q has crashed}

|| Task 3: when receive "q-is-alive" for some q
    if q ∈ output_p                       {p knows that it prematurely timed-out on q}
        output_p ← output_p − {q}              {1. p repents on q, and}
        Δ_p(q) ← Δ_p(q) + 1                    {2. p increases its time-out period for q}
coend
```

Solving consensus with S and $\Diamond S$

- Consensus is impossible in asynchronous systems, but solvable with failure detectors
- Consensus:

– Termination: Every correct process eventually decides on some value.
– Validity: If a process decides v, then v was proposed by some process.
– Agreement: No two correct processes decide differently.

- Let's study the protocol by Mostefaoui and Raynal (1999)

Solving consensus with S and $\Diamond S$

Step by Step

- Step 0
  - Initially, all processes P_i have their own $est_i = v_i$ of decision
  - Some process must send a proposal
  - Let's use the rotating coordinator approach
    - All processes know n and the coordinator c for each round r ($c = r \bmod n$)
  - Coordinator reliably sends its *est* to all processes
  - All processes wait for receiving *est* from coordinator
  - When they receive *est*, they decide *est*

- Doesn't work. Why?
  - If the coordinator fails (by crashing), then termination is violated!!
  - Failure of the coordinator must be detected (and recovered)

Solving consensus with $\mathcal{S}$ and $\Diamond\mathcal{S}$

Step by Step

- Step 1: Proposal
  - Let's use a Failure Detector to allow progress when *c* crashes
  - Consider a ***Strong FD*** or an ***Eventually Strong FD***
    - If c crashes, all processes will eventually detect the failure (SC)
    - But it is possible that some process will not wait enough time
  - All processes record the *est_from_c*, received from *c*
    - At some point (the end of initial phase) *est_from_c* can be equal to v (*c*'s proposal), or can be empty (⊥)

(3) $c \leftarrow (r_i \bmod n) + 1$; $est\_from\_c_i \leftarrow \bot$; $r_i \leftarrow r_i + 1$; % round $r = r_i$ %
(4) **case** $(i = c)$ **then** $est\_from\_c_i \leftarrow est_i$ *and send it to all*
FD
(5) $(i \neq c)$ **then wait** ((EST(r_i, v) is received from p_c)$\vee$($c \in suspected_i$));
(6) **if** (EST(r_i, v) has been received) **then** $est\_from\_c_i \leftarrow v$
(7) **endcase**; % $est\_from\_c_i = est_c$ or $\bot$ %

- It's not yet possible to decide! Why?
  - How to ensure that all others will decide the same value?

TFD

Solving consensus with $\mathcal{S}$ and $\Diamond\mathcal{S}$

Step by Step

- Step 2: Confirmation
  - Let's make all processes send their *est_from_c* to each other
    - Transmission is reliable, but processes may fail
    - A process can wait forever for all (n) possible *est_from_c*
    - A process finishes this phase when it has received *est_from_c* values from "enough" processes – a quorum (set Q)
    - "enough" = any two quorums must have a non-empty intersection
    - **The size of Q depends on the failure detector employed**
  - Received *est_from_c* values are stored in a *rec* (quorum) set

(8) $\forall j$ **do** *send* EST$(r_i, est_from_c_i)$ *to* p_j **enddo**;
(9) **wait until** $(\forall p_j \in Q_i$: EST(r_i, est_from_c) has been received from $p_j)$;
% Q_i has to be a *live* and *safe* quorum %
$\boxed{\begin{array}{l}\text{\% For } \mathcal{S}\text{: } Q_i \text{ is such that } Q_i \cup suspected_i = \Pi \text{ \%} \\ \text{\% For } \Diamond\mathcal{S}\text{: } Q_i \text{ is such that } |Q_i| = \lceil (n+1)/2 \rceil \text{ \%}\end{array}}$
(10) **let** $rec_i = \{est\_from\_c \mid$ EST(r_i, est_from_c) is received at line 5 or 9$\}$;
% $est\_from\_c = \bot$ or v with $v = est_c$ %
% $rec_i = \{\bot\}$ or $\{v\}$ or $\{v, \bot\}$ %

TFD

Solving consensus with $\mathcal{S}$ and $\Diamond\mathcal{S}$

Step by Step

- How to achieve overlapping quorums?
  - Let's consider an FD of class S
    - For sure, one correct process is never suspected
    - The condition to finish phase 2 is therefore waiting for messages from all correct processes
    - Even if the FD suspects all processes but one, this one will be included in all quorums
    - This means that the algorithm works for f < n
  - What if the FD of ◊S class?
    - FD can commit an arbitrary number of mistakes in a round
    - It is necessary to wait for messages of a majority of processes to ensure that any two **quorums** have at least one common process
    - The output of the FD (list of suspects) is not used in this case
    - This algorithm works because it is assumed that f < n/2

TFD

Solving consensus with $\mathcal{S}$ and $\Diamond\mathcal{S}$

Step by Step

- Step 2: Confirmation (cont.)
  - The *rec* set of any process can contain only *est_from_c* = v or ⊥
  - Given these values, is it now possible to decide?
  - Let's see the three possible outcomes for the *rec* set:

1. $rec_i = \{v\} \Rightarrow (\forall\, p_j :\ (rec_j = \{v\}) \ \vee \ (rec_j = \{v, \perp\}))$
2. $rec_i = \{\perp\} \Rightarrow (\forall\, p_j :\ (rec_j = \{\perp\}) \ \vee \ (rec_j = \{v, \perp\}))$
3. $rec_i = \{v, \perp\} \Rightarrow (\forall\, p_j :\ (rec_j = \{v\}) \ \vee \ (rec_j = \{\perp\}) \ \vee \ (rec_j = \{v, \perp\}))$

1. P_i can decide *v*, since any other process also decides *v*, or will take *v* as its own *est* for the next round
2. P_i will not decide, and no other process will decide
3. P_i will not decide, but since other processes might have decided *v*, it will update its own *est* to *v*

TFD

Solving consensus with S and $\Diamond S$

Step by Step

- Step 3: Decision
  - Just before deciding (returning *est*) the decision is sent to all processes (reliably broadcast)
    - Since the deciding process will stop executing the algorithm, this ensures that all correct processes decide, in case they have not decided in the same round
    - If the process crashes before successfully completing R-Broadcast, this may only delay decision (on same value) of correct processes

R-Broadcast

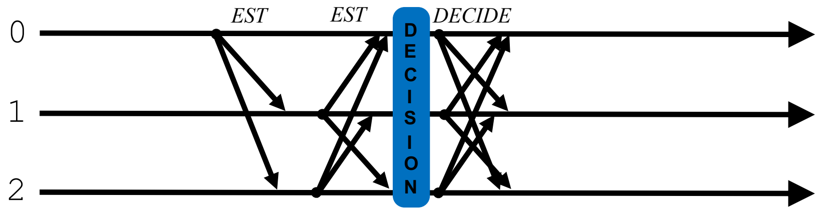
(11) **case** $(rec_i = \{\bot\})$ **then skip**
(12) $(rec_i = \{v\})$ **then** $\forall j \neq i$ **do** *send* DECIDE(v) *to* p_j **enddo**; **return**(v)
(13) $(rec_i = \{v, \bot\})$ **then** $est_i \leftarrow v$
(14) **endcase**
(15) **enddo**

(16) **task** $T2$: **upon reception of** DECIDE(v): R-Broadcast
$\forall j \neq i$ **do** *send* DECIDE(v) *to* p_j **enddo**; **return**(v)

Function Consensus(v_i)
cobegin
(1) **task** $T1$: $r_i \leftarrow 0$; $est_i \leftarrow v_i$; % $v_i \neq \bot$ %
(2) **while** *true* **do**
(3) $c \leftarrow (r_i \bmod n) + 1$; $est\_from\_c_i \leftarrow \bot$; $r_i \leftarrow r_i + 1$; % round $r = r_i$ %
(4) **case** $(i = c)$ **then** $est\_from\_c_i \leftarrow est_i$
(5) $(i \neq c)$ **then wait** ((EST(r_i, v) is received from p_c)$\vee$($c \in suspected_i$));
(6) **if** (EST(r_i, v) has been received) **then** $est\_from\_c_i \leftarrow v$
(7) **endcase**; % $est\_from\_c_i = est_c$ or $\bot$ %
(8) $\forall j$ **do** *send* EST$(r_i, est_from_c_i)$ *to* p_j **enddo**;
(9) **wait until** ($\forall p_j \in Q_i$: EST(r_i, est_from_c) has been received from p_j);
% Q_i has to be a *live* and *safe* quorum %
% For $\mathcal{S}$: Q_i is such that $Q_i \cup suspected_i = \Pi$ %
% For $\Diamond\mathcal{S}$: Q_i is such that $|Q_i| = \lceil (n+1)/2 \rceil$ %
(10) **let** $rec_i = \{est\_from\_c \mid$ EST(r_i, est_from_c) is received at line 5 or 9$\}$;
% $est\_from\_c = \bot$ or v with $v = est_c$ %
% $rec_i = \{\bot\}$ or $\{v\}$ or $\{v, \bot\}$ %
(11) **case** $(rec_i = \{\bot\})$ **then skip**
(12) $(rec_i = \{v\})$ **then** $\forall j \neq i$ **do** *send* DECIDE(v) *to* p_j **enddo**; **return**(v)
(13) $(rec_i = \{v, \bot\})$ **then** $est_i \leftarrow v$
(14) **endcase**
(15) **enddo**

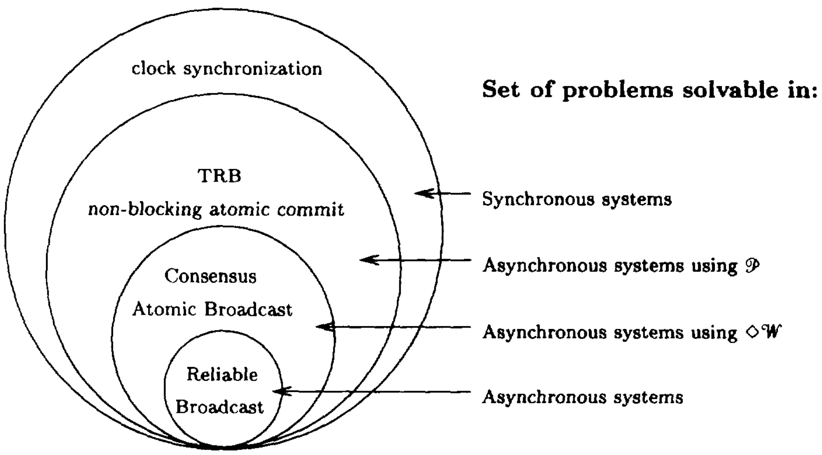
(16) **task** $T2$: **upon reception of** DECIDE(v):
$\forall j \neq i$ **do** *send* DECIDE(v) *to* p_j **enddo**; **return**(v)
coend

Solving consensus with S and $\Diamond S$



- A note about minimal number of communication steps:
  - With minimal number of processes ($n > 2f$ or $n > f$), consensus can be solved in two communication steps
  - Notice that RAFT, VSR and Paxos require three steps
  - With more processes ($n > 3f$), consensus can be solved in one communication step

What problems are solvable with (classical) failure detectors?



Bibliography

- T. Chandra and S. Toueg, **Unreliable Failure Detectors for Reliable Distributed Systems.** Journal of the ACM, 43(2):225-267, 1996.
- T. Chandra, V. Hadzilacos and S. Toueg, **The Weakest Failure Detector for Solving Consensus.** Journal of the ACM, 43(4):685–722, 1996.
- A. Mostéfaoui and M. Raynal. **Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: A General Quorum-Based Approach,** DISC, 1999.