Distributed Algorithms

Failure detection and Consensus

Ludovic Henrio
CNRS - projet SCALE
ludovic.henrio@cnrs.fr
Acknowledgement

- The slides for this lecture are based on ideas and materials from the following sources:

  - *Introduction to Reliable Distributed Programming* by Guerraoui, Rachid, Rodrigues, Luís, 2006
  - ID2203 Distributed Systems Advanced Course by Prof. Seif Haridi from KTH (Sweden)
  - *CS5410/514: Fault-tolerant Distributed Computer Systems Course* by Prof. Ken Birman from Cornell University
  - Course from F. Bongiovanni
  - A few slides from SARDAR MUHAMMAD SULAMAN
Failure detectors
System models

- **synchronous distributed system**
  - each message is received within bounded time
  - each step in a process takes \( lb < \text{time} < ub \)
  - each local clock’s drift has a known bound

- **asynchronous distributed system**
  - no bounds on process execution
  - no bounds on message transmission delays
  - arbitrary clock drifts

*the Internet is an asynchronous distributed system*
Failure model

- First we must decide what do we mean by failure?
  - Different types of failures
    - Crash-stop (fail-stop)
      - A process halts and does not execute any further operations
    - Crash-recovery
      - A process halts, but then recovers (reboots) after a while

- Crash-stop failures can be detected in synchronous systems

Rest of this first part: detecting crash-stop failures in asynchronous systems
What's a Failure Detector?

Needs to know about $P_j$'s failure

Crash failure
1. Ping-ack protocol

- $P_i$ queries $P_j$ once every $T$ time units
- if $P_j$ does not respond within $T$ time units, $P_i$ marks $p_j$ as failed

If $p_j$ fails, within $T$ time units, $p_i$ will send it a ping message, and will time out within another $T$ time units.
Detection time = $2T$
2. Heart-beating protocol

Needs to know about $P_j$'s failure

- if $P_i$ has not received a new heartbeat for the past $T$ time units, $P_i$ declares $P_j$ as failed

- $P_j$ maintains a sequence number

- $P_j$ send $P_i$ a heartbeat with incremented seq. number after $T' (= T)$ time units
Failure Detectors

- **Basic properties**
  - **Completeness**
    - Every crashed process is suspected
  - **Accuracy**
    - No correct process is suspected

Both properties come in two flavours: Strong and Weak.

- **Strong Completeness**
  - Every crashed process is eventually suspected by *every* correct process

- **Weak Completeness**
  - Every crashed process is eventually suspected by *at least* one correct process

- **Strong Accuracy**
  - No correct process is *ever* suspected

- **Weak Accuracy**
  - There is *at least* one correct process that is *never* suspected
Perfect failure detector P

- Assume synchronous system
  - Max transmission delay between 0 and $\delta$ time units

Every $\gamma$ time units, each node:
  Sends $<$heartbeat$>$ to all nodes

Each node waits $\gamma + \delta$ time units
  If did not get $<$heartbeat$>$ from pi
  Detect $<$crash $|$ pi$>$
An algorithm for $\mathbb{P}$

Upon event (HBTimeout)
   For all $pi$ in $\mathbb{P}$
      Send HeartBeat to $pi$
      startTimer ($gamma$, HBTimeout)

Upon event Receive HeartBeat from $pj$
   alive := alive $\cup$ pj

Upon event (DetectTimeout)
   crashed := $\mathbb{P} \setminus$ alive
   for all $pi$ in crashed Trigger (crashed, pi)
   alive := $\emptyset$
   startTimer ($delta$+$gamma$, DetectTimeout)

$\mathbb{P}$: set of processes
Correctness of P

- **PFD1 (strong completeness)**
  - A crashed node doesn’t send <heartbeat>
    - Eventually every node will notice the absence of <heartbeat>

- **PFD2 (strong accuracy)**
  - Assuming local computation is negligible
  - Maximum time between 2 heartbeats
    - $\gamma + \delta$ time units
  - If alive, all nodes will recv hb in time
    - No inaccuracy
Eventually perfect failure detector $\langle\rangle P$

- For asynchronous system
  - We suppose there is an **unknown** maximal transmission delay -- **partially synchronous system**

Every $\gamma$ time units, each node:
- Sends $\langle$heartbeat$\rangle$ to all nodes

Each node waits $T$ time units
- If did not get $\langle$heartbeat$\rangle$ from $pi$
  - Indicate $\langle$suspect $| pi\rangle$ if $pi$ is not in suspected
  - Put $pi$ in suspected set
- If get $\langle$heartbeat$\rangle$ from $pi$ and $pi$ is suspected
  - Indicate $\langle$restore $| pi\rangle$
  - remove $pi$ from suspected
  - Increase timeout $T$
An algorithm for $\leftrightarrow P$

Upon event (HBTimeout)
- For all $pi$ in $P$
  - Send HeartBeat to $pi$
  - startTimer ($gamma$, HBTimeout)

Upon event Receive HeartBeat from $pj$
- $alive := alive \cup pj$

Upon event (DetectTimeout)
- for all $pi$ in $P$
  - if $pi$ not in $alive$ and $pi$ not in $suspected$
    - $suspected := suspected \cup pi$
    - Trigger ($suspected$, $pi$
  - if $pi$ in $alive$ and $pi$ in $suspected$
    - $suspected := suspected \setminus pi$
    - Trigger ($restore$, $pi$
    - $T := T + \delta$
- $alive := \emptyset$
- startTimer ($T$, DetectTimeout)
Correctness of $\langle\rangle P$

- **PFD1 (strong completeness)**
  - Idem

- **PFD2 (eventual strong accuracy)**
  - Each time $p$ is inaccurately suspected by a correct $q$
    - Timeout $T$ is increased at $q$
    - Eventually system becomes synchronous, and $T$ becomes larger than the unknown bound $\delta$ ($T > \gamma + \delta$)
  - $q$ will receive HB on time, and never suspect $p$ again

**Question:** Formalise this a bit more: why is the number of iterations finite? Prove that $p$ is never suspected again
Exercise
Eventually Perfect Failure Detector: an alternative algorithm (questions next slide)

Algorithm 2.6 Increasing Timeout

Implements:
EventuallyPerfectFailureDetector (◊P).

Uses:
PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
alive := II;
suspected := ∅;
period := TimeDelay;
startTimer (period);

upon event ⟨ Timeout ⟩ do
if (alive ∩ suspected) ≠ ∅ then
period := period + Δ;
forall pi ∈ II do
if (pi ∉ alive) ∧ (pi ∉ suspected) then
suspected := suspected ∪ {pi};
trigger ⟨ suspect | pi ⟩;
else if (pi ∈ alive) ∧ (pi ∈ suspected) then
suspected := suspected \ {pi};
trigger ⟨ restore | pi ⟩;
trigger ⟨ pp2pSend | pi, [HEARTBEAT] ⟩;
alive := ∅;
startTimer (period);

upon event ⟨ pp2pDeliver | src, [HEARTBEAT] ⟩ do
alive := alive ∪ {src};
Exercise: is this a good algorithm?

What is the delay between two heartbeats? At the beginning? At any point in time? Can you find a formula for this depending on the number of failures suspected/recovered.

Is there a maximal time before a failure is detected? (supposing there is a bound Delta on maximal communication time)
Question:
Explain why, with pessimistic message logging protocols it is possible to recover only one process at a time. Give 2 advantages and 2 drawbacks of pessimistic message logging (PML) compared to communication induced checkpointing (CIC)

Plus the exercise on the next 3 slides
exercise: an hybrid protocol

Supposing processes are split into different groups
We want to implement CIC inside a group, and pessimistic message logging between groups
Specify (i.e. write pseudo code for) an protocol that implements such an hybrid algorithm:
what happens when we send/receive a com inside a group
between groups
when is a forced checkpoint taken
Specify the recovery mechanism (NB a whole group recovers)
Test it on the examples next slides (you can do the examples first and write the pseudo code after)
execution example: 2 groups x 3 processes

place forced messages?
which messages are logged?
what happens at recovery?
Another example

Same questions

NB: the exercise is inspired by the following paper (only read it if you want to go further)
Consensus (agreement)

- In the consensus problem, the processes propose values and have to agree on one among these values.

- Solving consensus is key to solving many problems in distributed computing (e.g., total order broadcast, atomic commit, terminating reliable broadcast).
Consensus - basic properties

- **Termination**
  - Every correct node eventually decides

- **Agreement**
  - No two correct processes decide differently

- **Validity**
  - Any value decided is a value proposed

- **Integrity:**
  - A node decides at most once

- **A variant: UNIFORM CONSENSUS**
  - Uniform agreement: No two processes decide differently
Consensus

Events

• Request: <Propose, v>
• Indication: <Decide, v’>

Properties:

• C1, C2, C3, C4

algorithm I

• A P-based (fail-stop) consensus algorithm
• The processes exchange and update proposals in rounds and decide on the value of the non-suspected process with the smallest id [Gue95]

Consensus algorithm II

• A P-based (i.e., fail-stop) uniform consensus algorithm
• The processes exchange and update proposal in rounds, and after n rounds decide on the current proposal value [Lyn96]
Consensus algorithm I

• The processes go through rounds incrementally (1 to n): in each round, the process with the id corresponding to that round is the leader of the round

• The leader of a round decides its current proposal and broadcasts it to all

• A process that is not leader in a round waits (a) to deliver the proposal of the leader in that round to adopt it, or (b) to suspect the leader
Consensus algorithm I

**Implements:** Consensus (cons).

**Uses:**
- BestEffortBroadcast (beb).
- PerfectFailureDetector (P).

**upon event** `<Init>` **do**

- suspected := empty;
- round := 1; currentProposal := nil;
- broadcast := delivered[] := false;
Best effort broadcast (beb Bcast)

- **Intuition:** everything is perfect unless sender crash

- **Events:**
  - **Request:** $< \text{bebBroadcast} | m >$: Used to broadcast message $m$ to all processes.
  - **Indication:** $< \text{bebDeliver} | \text{src}, m >$: Used to deliver message $m$ broadcast by process $\text{src}$.

- **Properties:**
  - **validity:** For any two processes $p_i$ and $p_j$, if $p_i$ and $p_j$ are correct, then every message broadcast by $p_i$ is eventually delivered by $p_j$.
  - **No duplication:** No message is delivered more than once.
  - **No creation:** If a message $m$ is delivered by some process $p_j$, then $m$ was previously broadcast by some process $p_i$.

- **We will use later: Reliable broadcast**
  - **Rb:** If a message $m$ is delivered by some correct process $p_i$, then $m$ is eventually delivered by every correct process $p_j$.

For details, see Francoise’ course
upon event < crash, pi > do
  suspected := suspected U {pi};

upon event < Propose, v> do
  if currentProposal = nil then
    currentProposal := v;

upon event < bebDeliver, p\_round, value > do
  currentProposal := value;
  delivered[round] := true;

upon event delivered[round] = true or p\_round \in suspected do
  round := round + 1;

upon event p\_round = self and broadcast = false and currentProposal \neq nil do
  trigger <Decide, currentProposal>;
  trigger <bebBroadcast, currentProposal>;
  broadcast := true;
Consensus algorithm I

propose(0)  decide(0)
p1

propose(1)  decide(0)
p2

propose(0)  decide(0)
p3
Consensus algorithm I

propose(0)
propose(1)
propose(0)
decide(0)
crash
decide(0)
decide(1)
decide(1)
Consensus algorithm I late message
Correctness argument

• Let pi be the correct process with the smallest id in a run R.

• Assume pi decides v.
  • If i = n, then pn is the only correct process.
  • Otherwise, in round i, all correct processes receive v and will not decide anything different from v. They are all located after i.

Question: How do you ensure that a message does not arrive too late? (in the wrong round)
Algorithm II: Uniform consensus

- The “Hierarchical Uniform Consensus” algorithm uses a perfect failure-detector, a best-effort broadcast to disseminate the proposal, a perfect link abstraction to acknowledge the receipt of a proposal, and a reliable broadcast to disseminate the decision.

- Every process maintains a single proposal value that it broadcasts in the round corresponding to its rank. When it receives a proposal from a more importantly ranked process, it adopts the value.

- In every round of the algorithm, the process whose rank corresponds to the number of the round is the leader.
Algorithm II: Uniform consensus (2)

- A round here consists of two communication steps: within the same round, the leader broadcasts a PROPOSAL message to all processes, trying to impose its value, and then expects to obtain an acknowledgment from all correct processes.
- Processes that receive a proposal from the leader of the round adopt this proposal as their own and send an acknowledgment back to the leader of the round.
- If the leader succeeds in collecting an acknowledgment from all processes except detected as crashed, the leader can decide. It disseminates the decided value using a reliable broadcast communication abstraction.
upon event \(\langle uc, \text{Init} \rangle\) do
  \[\text{detectedranks} := \emptyset;\]
  \[\text{ackranks} := \emptyset;\]
  \[\text{round} := 1;\]
  \[\text{proposal} := \bot; \text{decision} := \bot;\]
  \[\text{proposed} := [\bot]^N;\]

upon event \(\langle \mathcal{P}, \text{Crash} \mid p \rangle\) do
  \[\text{detectedranks} := \text{detectedranks} \cup \{\text{rank}(p)\};\]

upon event \(\langle uc, \text{Propose} \mid v \rangle\) such that \(\text{proposal} = \bot\) do
  \[\text{proposal} := v;\]

upon \(\text{round} = \text{rank}(\text{self}) \land \text{proposal} \neq \bot \land \text{decision} = \bot\) do
  trigger \(\langle \text{beb}, \text{Broadcast} \mid [\text{PROPOSAL}, \text{proposal}] \rangle;\)

upon event \(\langle \text{beb}, \text{Deliver} \mid p, [\text{PROPOSAL}, v] \rangle\) do
  \[\text{proposed}[\text{rank}(p)] := v;\]
  if \(\text{rank}(p) \geq \text{round}\) then
    trigger \(\langle pl, \text{Send} \mid p, [\text{ACK}] \rangle;\)
upon \( \text{round} \in \text{detectedranks} \) do
  \begin{itemize}
  \item \text{if} \ \text{proposed}[\text{round}] \neq \bot \ \text{then}
    \item \text{proposal} := \text{proposed}[\text{round}];
    \item \text{round} := \text{round} + 1;
  \end{itemize}

upon \( \langle \text{pl}, \text{Deliver} \mid \text{q}, [\text{ACK}] \rangle \) do
  \begin{itemize}
  \item \text{ackranks} := \text{ackranks} \cup \{\text{rank}(\text{q})\};
  \end{itemize}

upon \( \text{detectedranks} \cup \text{ackranks} = \{1, \ldots, N\} \) do
  \begin{itemize}
  \item \text{trigger} \( \langle \text{rb}, \text{Broadcast} \mid [\text{DECIDED}, \text{proposal}] \rangle \);
  \end{itemize}

upon \( \langle \text{rb}, \text{Deliver} \mid \text{p}, [\text{DECIDED}, \text{v}] \rangle \) such that \text{decision} = \bot do
  \begin{itemize}
  \item \text{decision} := \text{v};
  \item \text{trigger} \( \langle \text{uc}, \text{Decide} \mid \text{decision} \rangle \);
  \end{itemize}
Example - no failure
Example - failure (1)
Example - failure (2)
Correctness

- Validity and Integrity follow from the properties of the underlying communication, and the algorithm
- Agreement

Assume two processes decide differently, this can happen if two decisions were rbBroadcast

Assume pi and pj, j > i, rbBroadcast two decisions vi and vj, because of accuracy of P, pj must have adopted the value vi
Exercise

Study the algorithm on the next slides:
1 - Show a failure free execution and 2 execution with faults
2 - is it a correct consensus? Why?
3 - is it a uniform consensus? Why?
Algorithm 5.2 Hierarchical Consensus

Implements:
  Consensus (c).

Uses:
  BestEffortBroadcast (beb);
  PerfectFailureDetector (P).

upon event \( \langle \text{Init} \rangle \) do
  detected := \emptyset; round := 1;
  proposal := \bot; proposer := 0;
  for \( i = 1 \) to \( N \) do
    delivered[\( i \)] := broadcast[\( i \)] := false;

upon event \( \langle \text{crash} \mid p_i \rangle \) do
  detected := detected \cup \{ \text{rank}(p_i) \};
Hierarchical consensus Impl. (2)

upon event \(cPropose \mid v\) \& (proposal = \(\bot\)) do
\[
\text{proposal} := v;
\]

upon (round = \(\text{rank (self)}\)) \& (proposal \neq \(\bot\)) \& (broadcast[round] = \text{false}) do
\[
\text{broadcast}[\text{round}] := \text{true};
\text{trigger} \left(c\text{Decide} \mid \text{proposal}\right),
\text{trigger} \left(beb\text{Broadcast} \mid [\text{DECIDED}, \text{round}, \text{proposal}]\right);
\]

upon (round \in \text{detected}) \lor (\text{delivered}[\text{round}] = \text{true}) do
\[
\text{round} := \text{round} + 1;
\]

upon event \(beb\text{Deliver} \mid p_i, [\text{DECIDED}, r, v]\) do
\[
\text{if} (r < \text{rank (self)}) \land (r > \text{proposer}) \text{then}
\text{proposal} := v;
\text{proposer} := r;
\text{delivered}[r] := \text{true};
\]

set node’s initial proposal, unless it has already adopted another node’s
If I am leader
Trigger once per round
Trigger if I have proposal
Permanently decide
Next round if deliver or crash
Invariant: only adopt “newer” than what you have