Potpourri: Failure Detection, Election Algorithms & Mutual Exclusion

Dr. Yingwu Zhu
Failure Detection
Real Word: Failure is the Norm!

• You’ve been put in charge of a datacenter, and your manager has told you, “Oh no! We don’t have any failures in our datacenter!”

• Do you believe him/her?

• What would be your first responsibility?
  – Build a failure detector

• What are some things that could go wrong if you didn’t do this?
To Detect Failures

• You have a few options

1. Hire 1000 people, each to monitor one machine in the datacenter and report to you when it fails.
2. Write a failure detector program (distributed) that automatically detects failures and reports to your workstation.

Which is more preferable, and why?
Two Different Distributed Systems

Whenever someone gives you a distributed computing problem, the first question you want to ask is, “What is the model under which I need to solve the problem?”

- **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)
  
  Examples: Multiprocessor systems

- **Asynchronous Distributed System**
  - No bounds on message transmission delays
  - No bounds on process execution
  - (The drift of a clock is arbitrary)
  
  Examples: Internet, wireless networks, datacenters, most real systems
Failure Models

• Process omission failure
  – 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

• Our focus on *Crash-stop* failures
  – They are easy to detect in synchronous systems
  – Not so easy in asynchronous systems
What’s a failure detector?

pi

pj
What’s a failure detector?

Crash-stop failure
(pj is a *failed* process)
What’s a failure detector?

needs to know about pj’s failure
(pi is a *non-faulty* process
or *alive* process)

Crash-stop failure
(pj is a *failed* process)

There are two main flavors of Failure Detectors...
I. Ping-Ack Protocol

needs to know about pj’s failure

- pi queries pj once every T time units
- if pj does not respond within another T time units of being sent the ping, pi detects pj as failed

If pj fails, then within T time units, pi will send it a ping message. pi will time out within another T time units.

Worst case Detection time = 2T

The waiting time ‘T’ can be parameterized.
II. Heartbeating Protocol

needs to know about pj’s failure

- pj maintains a sequence number
- pj sends pi a heartbeat with incremented seq. number after every T time units

-if pi has not received a new heartbeat for the past, say 3*T time units, since it received the last heartbeat, then pi detects pj as failed

\[ \text{If } T \gg \text{round trip time of messages, then worst case detection time } \sim 3*T \] (why?)

The ‘3’ can be changed to any positive number since it is a parameter
In a Synchronous System

- The Ping-ack and Heartbeat failure detectors are always correct
  - Ping-ack: set waiting time ‘T’ to be > round-trip time upper bound
  - Heartbeat: set waiting time ‘3*T’ to be > round-trip time upper bound

- The following property is guaranteed:
  - If a process pj fails, then pi will detect its failure as long as pi itself is alive
  - Its next ack/heartbeat will not be received (within the timeout), and thus pi will detect pj as having failed
Failure Detection Properties

• **Completeness** = every process failure is eventually detected (no misses)
• **Accuracy** = every detected failure corresponds to a crashed process (no mistakes)

- What is a protocol that is 100% complete?
- What is a protocol that is 100% accurate?

• Completeness and Accuracy
  - Can both be guaranteed 100% in a synchronous distributed system
  - Can never be guaranteed simultaneously in an asynchronous distributed system

  Why?
In Asynchronous Systems

• Completeness and Accuracy both cannot be guaranteed!
• Why?
In Asynchronous Systems

• Completeness and Accuracy both cannot be guaranteed!
• Impossible because of arbitrary message delays, message losses
  – If a heartbeat/ack is dropped (or several are dropped) from pj, then pj will be mistakenly detected as failed => inaccurate detection
  – How large would the T waiting period in ping-ack or 3*T waiting period in heartbeating, need to be to obtain 100% accuracy?
  – In asynchronous systems, delay/losses on a network link are impossible to distinguish from a faulty process
• Heartbeating – satisfies completeness but not accuracy (why?)
• Ping-Ack – satisfies completeness but not accuracy (why?)
• Most failure detector implementations are willing to tolerate some inaccuracy, but require 100% Completeness
Failure Detection in Distributed Systems

- That was for one process $p_j$ being detected and one process $p_i$ detecting failures
- Let’s extend it to an entire distributed system
- Difference from original failure detection is
  - We want failure detection of not merely one process ($p_j$), but all processes in system
Centralized Heartbeating

$pj$, Heartbeat Seq. $l++$

$pi$

Downside?
Ring Heartbeating

$pj$, Heartbeat Seq. $l++$

Downside?
All-to-All Heartbeating

pj, Heartbeat Seq. i++

Advantage: Everyone is able to keep track of everyone
Downside?
Metrics for Failure Detection

• **Bandwidth**: the number of messages sent in the system during steady state (no failures)
  – Small is good

• **Detection Time**
  – Time between a process crash and its detection
  – Small is good

• **Scalability**: Given the bandwidth and the detection properties, can you scale to a 1000 or million nodes?
  – Large is good

• **Accuracy**
  – High is good (lower inaccuracy is good)
Metrics for Accuracy

• **False Detection Rate**: Average number of failures detected per second, when there are in fact no failures

• Fraction of failure detections that are false

• Tradeoffs: If you increase the T waiting period in ping-ack or 3*T waiting period in heartbeating what happens to:
  – Detection Time?
  – False positive rate?
  – Where would you set these waiting periods?
Summary

• Failure detectors are required in distributed systems to keep system running in spite of process crashes
• Properties – completeness & accuracy, together unachievable in asynchronous systems but achievable in synchronous systems
  – Most apps require 100% completeness, but can tolerate inaccuracy
• 2 failure detector algorithms - Heartbeating and Ping
• Distributed FD through heartbeating: Centralized, Ring, All-to-all
• Metrics: Bandwidth, Detection Time, Scale, Accuracy
Election Algorithms
Why Election?

• Example 1: Your Bank maintains multiple servers in their cloud, but for each customer, one of the servers is responsible, i.e., the leader.
  – What if there are two leaders per customer?
  – What if servers disagree about who the leader is?
  – What if the leader crashes?
Why Election?

• Example 2: Group of cloud servers replicating a file need to elect one among them as the primary replica that will communicate with the client machines

• Example 3: Group of NTP servers: who is the root server?
What is Election?

• In a group of processes, elect a *Leader* to undertake special tasks.

• What happens when a leader fails (crashes)
  – Some process detect this
  – Then what?

• Our focus: *Election Algorithm*
  – Elect one leader only among the non-faulty processes
  – All non-faulty processes agree on who is the leader
Election Algorithms

• Many distributed systems require one process to act as coordinator/initiator, or perform some special role
• Elect one to fit into that role
• In general, election algorithms attempt to locate the process with the highest process number as the coordinator
Assumptions

• Any process can call for an election.
• A process can call for at most one election at a time.
• Multiple processes can call an election simultaneously.
  – All of them together must yield a single leader only
• The result of an election should not depend on which process calls for it.
• Messages are eventually delivered.
Problem Specification

• At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
  – Attribute examples: leader has highest id or address, or fastest cpu, or most disk space, or most number of files, etc.

• A run (execution) of the election algorithm must always guarantee at the end:
Algorithm 1: Ring Election (1)

- N Processes are organized in a logical ring
  - \( P_i \) has a communication channel to \( p_{(i+1) \mod N} \)
  - All messages are sent clockwise around the ring

- Any process \( p \) that discovers the old coordinator has failed initiates an “election” message that contains \( p \)’s own id:attr.
Algorithm 1: Ring Election (2)

• When a process receives an election message, it compares the attr in the message with its own attr.
  – If the arrived attr is greater, the receiver forwards the message.
  – If the arrived attr is smaller and the receiver has not forwarded an election message earlier, it overwrites the message with its own id:attr, and forwards it.
  – If the arrived id:attr matches that of the receiver, then this process’s attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an “elected” message to its neighbor with its id, announcing the election result.

• When a process $p_i$ receives an elected message, it
  – Sets its variable $elected_i \leftarrow$ id of the message.
  – Forwards the message if it is not the new coordinator.
The worst-case scenario occurs when the counter-clockwise neighbor (@ the initiator) has the highest attr.

In the example: The election was started by process 17. The highest process identifier encountered so far is 24.

(final leader will be 33)
Analysis

- The worst-case scenario occurs when the counter-clockwise neighbor has the highest attr.

In a ring of N processes, in the worst case:
- A total of N-1 messages are required to reach the new coordinator-to-be (election messages).
- Another N messages are required until the new coordinator-to-be ensures it is the new coordinator (election messages – no changes).
- Another N messages are required to circulate the elected messages.
- Total Message Complexity = 3N-1
- Turnaround time = 3N-1
Problem

• What happens if there are failures during the election run?
Example: Ring Election

1. P2 initiates election after old leader P5 failed
2. P2 receives “election”, P4 dies
3. Election: 4 is forwarded for ever?

May not terminate when process failure occurs during the election! Consider above example where attr==highest id
Algorithm 2: Modified Ring Election

- N Processes are organized in a logical ring
- Any process $p$ that discovers the old coordinator has failed initiates an "election" message. This is the *initiator* of the election.
- The message is circulated around the ring, bypassing failed nodes.
- Each node adds its id:attr to the message as it passes it to the next node (without overwriting what is already in the message)
- Once the message gets back to the initiator, it elects the node with the best election attribute value.
- It then sends a "coordinator" message with the id of the newly-elected coordinator. Again, each node adds (appends) its id to the end of the message.
- Once "coordinator" message gets back to initiator,
  - Election is over if "coordinator" is in id-list.
  - Otherwise the algorithm is repeated (handles election failure).
Example: Ring Election

1. P2 initiates election

2. P2 receives “election”, P4 dies

3. P2 selects 4 and announces the result

4. P2 receives “Coord”, but P4 is not included

5. P2 re-initiates election

6. P3 is finally elected
Modified Ring Election

• How many messages?
• What is the turnaround time?
• How would you redesign the algorithm to be fault-tolerant to an initiator’s failure?
  – One idea: Have the initiator’s successor wait a while, then re-initiate a new election. Do the same for this successor’s successor, and so on...

• Reconfiguration of ring upon failures
  – Can be done if all processes “know” about all other processes in the system
• Supports concurrent elections – an initiator with a lower id blocks other initiators’ election messages
What about that Impossibility?

• The Election problem is a special form of the *consensus problem*

• In an asynchronous system with no bounds on message delays and arbitrarily slow processes, can we have a generic leader election protocol?
  – No! [Marzullo and Sabel] + Impossibility of Consensus

• Where does the modified Ring election start to give problems with the above asynchronous system assumptions?
  – Pi may just be very slow, but not faulty (yet it is not elected as leader!)
  – Also slow initiator, ring reorganization
Algorithm 3: Bully Algorithm

• Assumptions
  – Synchronous system
    • All messages arrive within $T_{\text{trans}}$ units of time.
    • A reply is dispatched within $T_{\text{process}}$ units of time after the receipt of a message.
    • If no response is received in $2T_{\text{trans}} + T_{\text{process}}$, the node is assumed to be faulty (crashed).
  – Attr = id
  – Each process knows all the other processes in the system (and thus their id’s)
Algorithm 3: Bully Algorithm

- \( P \) sends an \textit{ELECTION} message to all processes with higher numbers.
  - If no answer within timeout, announce itself to lower nodes as coordinator.
  - If any answer received, then there is some non-faulty higher node \( \rightarrow \) so, wait for coordinator message. If none received after a timeout, start a new election.
- A node that receives an “election” message replies with answer, & starts its own election protocol (unless it has already done so).
- When a process finds the coordinator has failed, if it knows its id is the highest, it elects itself as coordinator, then sends a \textit{coordinator} message to all processes with lower identifiers.
(a) Process 4 holds an election.
(b) Processes 5 and 6 respond, telling 4 to stop.
(c) Now 5 and 6 each hold an election.
Example

(d) Process 6 tells 5 to stop.
(e) Process 6 wins and tells everyone.
Analysis of Bully Algorithm

• Best case scenario: The process with the second highest id notices the failure of the coordinator and elects itself.
  – $N-2$ coordinator messages are sent.
  – Turnaround time is one message transmission time.
Analysis of Bully Algorithm

- Worst case scenario: When the process with the lowest id in the system detects the failure.
  - N-1 processes altogether begin elections, each sending messages to processes with higher ids.
  - The message overhead is $O(N^2)$.
  - Turnaround time is approximately 5 message transmission times if there are no failures during the run:
    1. Election from lowest id process
    2. Answer to lowest id process from 2$^{nd}$ highest id process
    3. Election from 2nd highest id process
    4. Timeout for answers @ 2nd highest id process
    5. Coordinator from 2$^{nd}$ highest id process
Summary of Election Algorithms

• Coordination in distributed systems requires a leader process.
• Leader process might fail.
• Need to (re-) elect leader process.
• Three Algorithms:
  – Ring algorithm
  – Modified Ring algorithm
  – Bully Algorithm
Mutual Exclusion
Why Mutual Exclusion?

• **Bank’s Servers in the Cloud:** Think of two simultaneous deposits of $10,000 into your bank account, each from one ATM.
  – Both ATMs read initial amount of $1000 concurrently from the bank’s cloud server
  – Both ATMs add $10,000 to this amount (locally at the ATM)
  – Both write the final amount to the server
  – What’s wrong?

• The ATMs need *mutually exclusive* access to your account entry at the server (or, to executing the code that modifies the account entry)
Mutual Exclusion

- **Critical section** problem: Piece of code (at all clients) for which we need to ensure there is at most one client executing it at any point of time.

- **Solutions:**
  - Semaphores, mutexes, etc. in single-node operating systems
  - Message-passing-based protocols in distributed systems
    - `enter()` the critical section
    - `AccessResource()` in the critical section
    - `exit()` the critical section
  - Distributed mutual exclusion requirements:
    - **Safety** – At most one process may execute in CS at any time
    - **Liveness** – Every request for a CS is eventually granted
    - **Ordering** (desirable) – Requests are granted in the order they were made
How are semaphores used?

One Use: Mutual Exclusion – Bank ATM example

semaphore S=1;

ATM1:
    wait(S); // enter
    // critical section
    obtain bank amount;
    add in deposit;
    update bank amount;
    signal(S); // exit

extern semaphore S;

ATM2
    wait(S); // enter
    // critical section
    obtain bank amount;
    add in deposit;
    update bank amount;
    signal(S); // exit
Assumptions/System Model

- Each pair of processes is connected by reliable channels (such as TCP).
- Messages are eventually delivered to recipients’ input buffer in FIFO order.
- Processes do not fail.
A Centralized Algorithm

(a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted (a token is given).
(b) Process 2 then asks permission to access the same resource. The coordinator does not reply and queue the request.
(c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.
Mutual Exclusion
A Centralized Algorithm

• Simple: 3 messages: request, grant, release

• Downsides
  • Simple point of failures
  • Performance bottleneck
A Distributed Algorithm (Ricart & Agrawala)

• The requestor broadcasts a message containing the requested resource, process id, and logical time

• Three different cases:
  1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
  2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
  3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. (Lamport’s clock vector to implement tm)
A Distributed Algorithm (Ricart & Agrawala)

- Processes requiring entry to critical section multicast a request, and can enter it only when all other processes have replied positively.
- Messages requesting entry are of the form \(<T, p_i>\), where \(T\) is the sender’s timestamp (from a Lamport clock) and \(p_i\) the sender’s identity (used to break ties in \(T\)).

To enter the CS
- set state to wanted
- multicast “request” to all processes (including timestamp)
- wait until all processes send back “reply”
- change state to held and enter the CS

On receipt of a request \(<T_v, p_i>\) at \(p_j\):
- if (state = held) or (state = wanted & \((T_v, p_j)<(T_v, p_i))\), // lexicographic ordering
  enqueue request
- else “reply” to \(p_i\)

On exiting the CS
- change state to release and “reply” to all queued requests.
Ricart & Agrawala’s Algorithm

On initialization
   \textit{state} := \textsc{released};

To enter the section
   \textit{state} := \textsc{wanted};
   Multicast request to all processes;
   \(T := \text{request’s timestamp};\)
   \textit{Wait until} (number of replies received = \((N - 1)\));
   \textit{state} := \textsc{held};

On receipt of a request \(<T_i, p_i>\) at \(p_j\) \((i \neq j)\)
   \textbf{if} \ ((\textit{state} = \textsc{held} \text{ or } (\textit{state} = \textsc{wanted} \text{ and } (T, p_j) < (T_i, p_i))) \textbf{then}
      \quad \text{queue request from } p_i \text{ without replying;}
   \textbf{else}
      \quad \text{reply immediately to } p_i;
   \textbf{end if}

To exit the critical section
   \textit{state} := \textsc{released};
   reply to any queued requests;
A Distributed Algorithm (Ricart & Agrawala)

(a) Two processes want to access a shared resource at the same moment.
(b) Process 0 has the lowest timestamp, so it wins.
(c) When process 0 is done, it sends an OK also, so 2 can now go ahead.
A Distributed Algorithm (Ricart & Agrawala)

- Message complexity: $2(n-1)$ per entry
  - $(n-1)$ for multicast and $(n-1)$ for replies
- Magnify the single point of failure problem in centralized algorithms ($n$ points)
- Group membership is known
- Bottleneck: each machine handles same load, but machines may be heterogeneous
Token Ring Approach

Processes are organized in a logical ring: $p_i$ has a communication channel to $p_{(i+1) \mod (n)}$.

Operations:
- Only the process holding the token can enter the CS.
- To enter the critical section, wait passively for the token. When in CS, hold on to the token.
- To exit the CS, the process sends the token onto its neighbor.
- If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.

Features:
- Safety & liveness are guaranteed, but ordering is not.
- Bandwidth: 1 message per exit
- Client delay: 0 to N message transmissions.
- Synchronization delay between one process’s exit from the CS and the next process’s entry is between 1 and N-1 message transmissions.
Token Ring Algorithm

• Problems
  – Lost tokens, how to detect them?
  – Process failures, how to detect them?