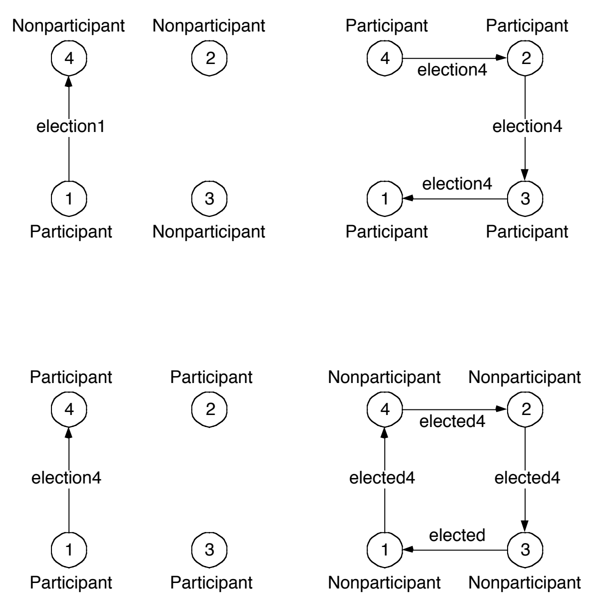
Chang and Roberts' Ring Algorithm (1979)

- Threads are arranged in a logical ring
  - Every thread is initially a *non-participant*
- The election:
  - A thread begins an election by
    - Marking itself as a *participant*
    - Sending an *election* message (containing its identifier) to its neighbor
  - When a thread receives an *election* message, it compares the identifier that arrived in the message to its own:
    - If the arrived identifier is greater, then it:
      - If it is not a *participant*, it:
        - » Marks itself as a *participant*
      - Forwards the message to its neighbor
    - If the arrived identifier is smaller:
      - If it is not a *participant*, it:
        - » Marks itself as a *participant*
        - » Substitutes its own identifier in the *election* message and sends it on
      - If it is already a *participant*, it does nothing

Chang and Roberts' Ring Algorithm (cont.)

- The election:
  - When a thread receives an *election* message, it compares…:
    - If the arrived identifier is that of the receiving thread, then its identifier is the largest, so it becomes the coordinator
      - It marks itself as a *non-participant* again,
      - It sends an *elected* message to its neighbor, announcing the results of the election and its identity
  - When a thread receives an *elected* message, it
    - Marks itself as a *non-participant*, and
    - Forwards the message to its neighbor
- Evaluation:
  - 3N–1 messages in worst case
    - N–1 *election* messages to reach immediate neighbor in wrong direction, N *election* messages to elect it, then N *elected* messages to announce result

Chang and Roberts' Ring Algorithm (cont.)



Agreement

- In a distributed system, it is often necessary for a set of processors to reach *mutual agreement (consensus)*
  - Mutual exclusion — agree who has the right to enter the critical section
  - Maintain replicated data, monitor a distributed computation, detect failed processors, etc.
  - This is one of the most fundamental problems in distributed system design
- In normal situations, this isn't a problem
  - Exchange values, take average, etc.
  - However, this is difficult if the system contains *failures* (also called *faults*)
    - Faulty processors can send erroneous values to other processors
    - Faulty network links can prevent values from reaching other processors

Adversaries

- One way to think about agreement is to imagine an all-powerful ***adversary***
  - Adversary is a demon with complete control over the system who will try to make your algorithm fail
  - Adversary knows global system state (but you can not!) and can arbitrarily interleave process execution, event execution, message delivery, etc.
  - Adversary can make processors and links fail at arbitrary times, even intermittently
- You must design an agreement algorithm that always works
  - Can't say "but that's highly unlikely!", because that's what the adversary will do

System Model

- There are N processors in the system trying to reach agreement
  - A subset F of those N processors are *faulty*, and others are non-faulty
  - Each processor P_i has a value V_i
- To reach agreement, each processor calculates an agreement value A_i
  - Every N–F non-faulty processor computes the same agreement value A_i
    - This A_i does not depend on the value V_i of any of the faulty processors
  - We don't care what agreement value A_i the F faulty processors compute
- Any processor can communicate directly with any other processor, and the communication mechanism is reliable (no messages are lost or corrupted)

Processor Failure

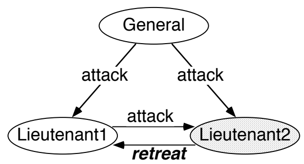
- Types of failures (Christian, 1991):
  - Omission failure — server doesn't respond to a request
  - Response failure — server responds incorrectly to a request
    - Returns wrong value, has wrong effect on resources (e.g., sets wrong values)
  - Timing failure — server responds too late (e.g., it's overloaded) or too early
  - Crash failure — repeated omission failure; server repeatedly fails to respond to requests until it is restarted
    - Amnesia crash — restarts in initial state
    - Pause crash — … in state before crash
    - Halting crash — never restarts
- A failure that exhibits many of these is called Byzantine failure (Lamport, 1982)
  - Goal: system should function correctly!

Byzantine Generals Problem

- There is one general, and N–1 lieutenants
  - The general gives an order "attack" or "retreat" to the lieutenants
  - The general and the lieutenants are either "loyal" or "traitors"
    - A traitor may act maliciously to prevent agreement (think of the adversary)
- Goal: to reach agreement:
  - All loyal lieutenants should agree on the order to perform
  - If the general is loyal, then the order the loyal lieutenants agree on should be the order he sent
  - Even if the general is a traitor, the loyal lieutenants should agree with each other
  - It is irrelevant what order the traitorous officers want to perform

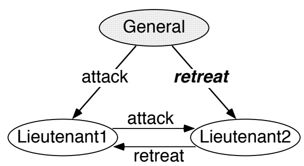
1 General, 2 lieutenants (1 Traitor, 2 Loyal)

- What if a lieutenant is a traitor?



  - Solution: assume the general is loyal

- But — what if the general is the traitor?



  - If each lieutenant assumes the general is loyal, they can't reach agreement

- 3 processors can **not** reach agreement in the presence of a single faulty processor

Lamport, Shostak, and Pease's Oral Message Algorithm (1982)

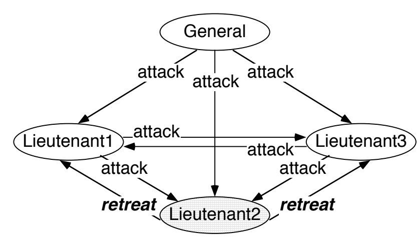
- Solves the Byzantine Generals problem for 3M+1 officers, with at most M traitors
- Officers can send "oral" (non-authenticated) messages:
  - Every officer can send a message to every other officer
    - But the officer may modify a received message before sending it on, or may forge a message from another officer
  - Every message that it sent is delivered correctly (i.e., no messengers captured)
    - The receiver of a message knows who sent it, and the absence of a message can be detected (communicate in "rounds")
- Other assumptions:
  - A traitorous general may or may not send a message
  - A lieutenant's default order is "retreat"

Lamport, Shostak, and Pease's Oral Message Algorithm (cont.)

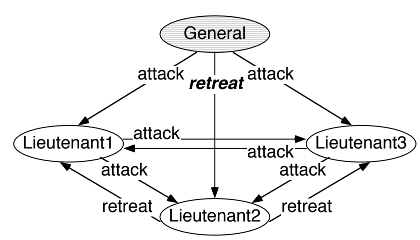
- Solves the Byzantine Generals problem for 3M+1 officers, with at most M traitors
- Algorithm for 4 officers, at most 1 traitor:
  - General sends order to each lieutenant
  - A lieutenant's initial order is the value received from the general, or "retreat" if no order was received
  - Each lieutenant sends his initial order to all the other lieutenants
  - Each lieutenant's final order is the majority of 3 orders it received (1 from the general, 1 from each of the 2 lieutenants)

1 General, 3 lieutenants (1 Traitor, 3 Loyal)

- What if a lieutenant is a traitor?



- What if the general is the traitor?



- 4 processors **can** reach agreement in the presence of a single faulty processor

Agreement Problems

- Byzantine agreement
  - Source processor broadcasts its initial value to all other processors
  - All non-faulty processors must agree on the same value
  - If the source processor is non-faulty, then the commonly-agreed-upon value of all the non-faulty processors must be the initial value of the source
- Consensus
  - Every processor broadcasts its initial value to all other processors
  - All non-faulty processors must agree on the same single value
  - If the initial value of every non-faulty processor is V, then the commonly-agreed-upon value of all the non-faulty processors must be V

Agreement Problems (cont.)

- Interactive Consistency
  - Every processor broadcasts its initial value to all other processors
  - All non-faulty processors must agree on the same vector V = (v1, v2, …, vn)
  - If the *i*-th processor is non-faulty and its initial value is v*i*, then the commonly-agreed-upon value of all the non-faulty processors for the *i*-th value must be v*i*

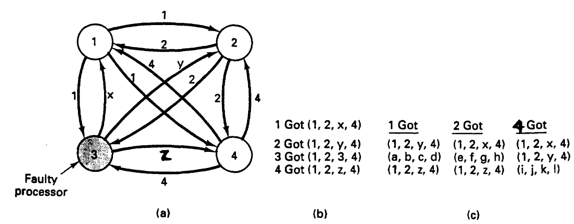


Fig. 4-23. The Byzantine generals problem for 3 loyal generals and 1 traitor. (a) The generals announce their troop strengths (in units of 1K). (b) The vectors that each general assembles based on (a). (c) The vectors that each general receives in step 2.

Distributed Operating Systems, Tanenbaum, Prentice Hall, 1995

Fault-Tolerant Physical Clock Synchronization

- 3 basic assumptions:
  - All clocks are initially synchronized to approximately the same value
  - A non-faulty process's clock runs at approximately the correct rate
  - A non-faulty process can read the clock value of another non-faulty clock with at most a small error
- Interactive Convergence Algorithm:
  - Each process reads the value of all other processes' clocks, and sets its clock value to the average of these values
    - If a clock value differs from its own clock by more than δ, it replaces that value by its own clock value in taking the average
  - If the clocks are synchronized often enough, they will converge to within a desired degree

Fault-Tolerant Physical Clock Synchronization (cont.)

- Interactive Consistency Algorithm:
  - Improvements
    - Take median of clock values (instead of mean)
      - Provides a better estimate, since number of faulty clocks should be low
    - Overcomes problem of two-faced clocks
  - Two processes compute approximately the same median if:
    - Any two processes obtain approximately the same value for a process P's clock (even if process P is faulty)
    - If Q is a non-faulty process, then every non-faulty process obtains approximately the correct value for process Q's clock
  - Algorithm for clock synchronization:
    - Use solution to Interactive Consistency problem (e.g., Oral Message Algorithm) to collect clock values for all clocks
    - Set local clock to be median of the collected clock values