CIS 505: Software Systems
Lecture Note on Consensus

Insup Lee
Department of Computer and Information Science
University of Pennsylvania

CIS 505, Spring 2007

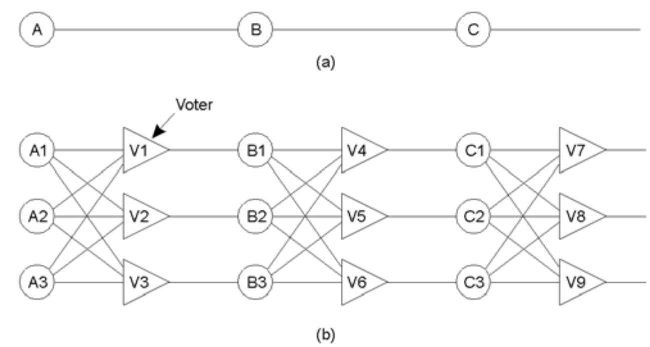
Concepts

- Dependability
  - o Availability –ready to be used; e.g.,
    - one millisecond every hour
    - 99.9999% available
  - o Reliability – run continuously without failure; e.g., 24 hrs of meantime to failure
  - o Safety – operate correctly
  - o Maintainability – how easy to repair a failed system
- Fault-tolerance
  - o Failure – a system cannot perform correctly
  - o Error – a part of a system's state that may lead to failure
  - o Fault – the cause of an error
    - Transient fault, intermittent fault, permanent fault
  - o Fault-tolerant – a system can provide its services even in the presence of faults

Techniques for masking faults

- Information redundancy
  - o E.g., send extra bits
- Time redundancy
  - o E.g., repeat if needed
- Physical redundancy
  - o TMR (Triple Modular Redundancy)

Failure Masking by Redundancy



Why is reaching agreement important?

- If system has *shared* state, and each node has a local view of state, must agree (roughly) on what shared state is.
- If system is *cooperating* must agree on a plan of action.

Must bootstrap this process by agreeing (in advance, and/or off-line) on how to reach agreement
Must agree on *agreement protocol(s)*

Why is reaching agreement hard?

- Agents die
- Agents lie
- Agents sleep (and wake up)
- Agents don't hear all messages
- Agents hear messages incorrectly
- Groups of agents split into cliques (partition)

More formally, these are known as
Failure Modes

Failure Models

- Different types of failures.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Failure modes: Processors

- Fail-stop: dies, stays dead, you know it's dead.
- Crash: dies, stays dead, maybe you don't know
- Receive omission: either dies or only gets *some* of the msgs sent to it.
- Send omission: either dies or only sends *some* of the msgs it tries to send.
- General omission: either send or receive omission, or both.
- Byzantine failure: can do *anything* - violate any protocol, lie, random behavior, evil/malicious behavior

Failure modes: Links

- Fail-stop: stops xmiting or recving, stays broken, detected (rare model)
- Crash: stops xmiting or recving, stays broken, maybe undetected
- Byzantine failure: can do *anything* - duplicate packets, fabricate packets, duplicate after arbitrarily long delay… (e.g., babbling idiots)

- Note: additional failure modes arise when an assumed property (ordering, reliability, low error rate) disappears

Types of Systems

- Synchronous
  - o Relative processors speeds are bounded
  - o Communication delays are bounded
- Asynchronous
  - o Can make no assumptions
- Intuitively: In Synchronous systems we can assume things happen in "rounds", (nobody is too slow) but this also means that you have to wait for a round before you can progress (nobody can be too fast)

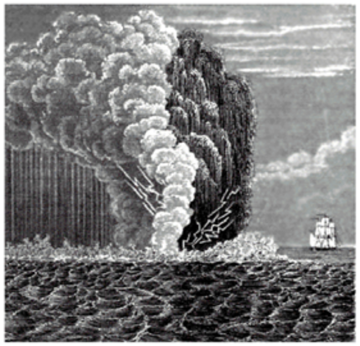
Classical types of agreement problems

- Synchronization (or Mutual Exclusion)
- Leader Election
- Coordination (two General's problem)
- Consensus
- Atomic Commitment

Partial Recap on Synchronization

- also known as Mutual Exclusion problem
- P processes, only 1 may proceed
- Token
- Voting (> 1/2)
- Timestamp or causal order or order

Unreliable communication



Two generals' problem

- Two generals on opposite sides of a valley have to agree on whether to attack or not (at a pre-agreed time)
- Goal: Each must be sure that the other one has made the same decision
- Communicate by sending messenger who may get captured
- Can never be sure whether the last messenger reached the other side (every message needs an ack), so no perfect solution
- Impossibility of consensus is as fundamental as undecidability of the halting problem !
- In practice: probability of losing a repeatedly sent message decreases (so agreement with high probability possible)

Impossibility Proof

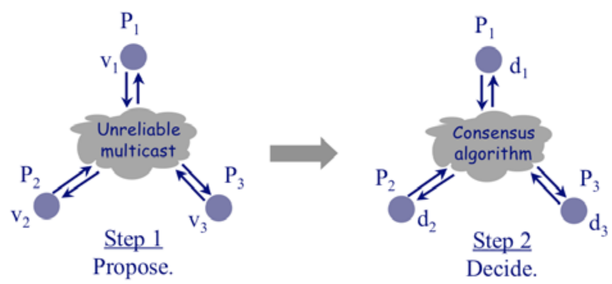
Theorem. If any message can be lost, it is not possible for two processes to agree on non-trivial outcome using only messages for communication.

Proof. Suppose it is possible. Let m[1],…,m[k] be a finite sequence of messages that allowed them to decide. Furthermore, let's assume that it is a minimal sequence, that is, it has the least number of messages among all such sequences. However, since any message can be lost, the last message m[k] could have been lost. So, the sender of m[k] must be able to decide without having to send it (since the sender knows that it may not be delivered) and the receiver of m[k] must be able to decide without receiving it. That is, m[k] is not necessary for reaching agreement. That is, m[1],…,m[k-1] should have been enough for the agreement. This is a contradiction to that the sequence m[1],…,m[k] was minimum.

Four Dimensions of Failure Models

- Reliable vs. unreliable network
  - *Reliable*: all messages are eventually delivered exactly once.
- Synchronous vs. asynchronous communication
  - *Synchronous*: message delays (and process delays) are bounded, enabling communication in *synchronous rounds*.
- Byzantine vs. fail-stop
  - *Fail-stop*: faulty nodes stop and do not send.
  - *Byzantine*: faulty nodes may send arbitrary messages.
- Authenticated vs. unauthenticated
  - *Authenticated*: the source and content of every message can be verified, even if a Byzantine failure occurs.

Consensus



Generalizes to N nodes/processes.

[Chase]

Assumptions

- For now we assume:
  - Nodes/processes communicate only by messages.
  - The network may be synchronous or asynchronous.
  - The network channels are *reliable*.
    - Is this realistic?

Properties for Correct Consensus

There are at least two possible values, 0 and 1.

- *Termination*: All correct processes *eventually* decide.
- *Agreement*: All correct processes select the same d_i.
  - Or…(stronger) all processes that do decide select the same d_i, even if they later fail.
- *Integrity*: All deciding processes select the "right" value.
  - As specified for the variants of the consensus problem.

[Chase]

Consensus: synchronous with no failures

- The solution is trivial in one round of proposal messages.
- Intuition: all processes receive the same values, the values sent by the other processes.
- Step 1. Propose.
- Step 2. At end of round, each P_i decides from received values.
  - *Consensus*: apply any deterministic function to $\{v_0,\ldots, v_{N-1}\}$.
  - *Command consensus*: if v_{leader} was received, select it, else apply any deterministic function to $\{v_0,\ldots, v_{N-1}\}$.
  - *Interactive consistency*: construct a vector from all received values.

[Chase]

Problem Definition

- Generals = Computer Components
- The abstract problem…
  - Each division of Byzantine army is directed by its own general.
  - There are n Generals, some of which are traitors.
  - All armies are camped outside enemy castle, observing enemy.
  - Communicate with each other (private) by messengers.
  - Requirements:
    - G1: All loyal generals decide upon the same plan of action
    - G2: A small number of traitors cannot cause the loyal generals to adopt a bad plan
  - Note: We **do not** have to identify the traitors.

Naïve solution

- i^{th} general sends v(i) to all other generals
- To deal with two requirements:
  - All generals combine their information v(1), v(2), .., v(n) in the same way
  - Majority (v(1), v(2), …, v(n)), ignore minority traitors
- Naïve solution does not work:
  - Traitors may send different values to different generals.
  - Loyal generals might get conflicting values from traitors
- **Requirement:** Any two loyal generals must use the same value of v(i) to decide on same plan of action.

Lamport's 1982 Result, Generalized by Pease

- The Lamport/Pease result shows that consensus is impossible:
  - with byzantine failures,
  - if one-third or more processes fail (N ≤ 3m),
    - Lamport shows it for 3 processes, but Pease generalizes to N.
  - even with synchronous communication.
- Intuition: a node presented with inconsistent information cannot determine which process is faulty.
- The good news: consensus can be reached if N > 3m, no matter what kinds of node failures occur.

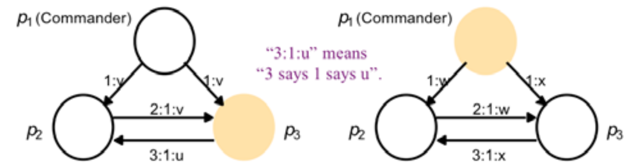
Assumptions

- System Model:
  - n processors, at most m are faulty
  - fully connected, message passing
  - receiver always knows the identity of the sender
  - reliable communication, only processors fail
  - the value communicated is 0 or 1 (or v or w)
- Synchronous computation: processes run in a lock step manner.
  - In each step a process receives one or more messages, performs a computation, and sends one or more messages to other processes
  - A process knows all messages it expects to receive in a round.
- Byzantine failure: process can behave randomly.
- Oral messages:
  - process can change the contents of a message before it relays the message to other processes, i.e. it can lie about what it received from another process.
- Performance Aspects: number of rounds (time) and number of messages

An Impossibility Result

- The Byzantine Agreement Problem (restated):
  - o Agreement- All nonfaulty processes agree on the same value.
  - o Validity- If the source process is nonfaulty, then the common agreed upon value by all nonfaulty processors should be the initial value of the source.
- An Impossibility Result:
  - o Byzantine Agreement cannot be reached among three processors where one processor is faulty.
- A Stronger Result:
  - o No solution with fewer than 3m+1 processes can tolerate m faulty processes.

Impossibility with three byzantine generals



Faulty processes are shown shaded

[Lamport82]

Intuition: subordinates cannot distinguish these cases.

Each must select the commander's value in the first case, but this means they cannot agree in the second case.

Lamport-Shostak-Pease Algorithm

- Algorithm OM(0).
  1. The source process sends its value to every other process.
  2. Each process uses the value it received from the source. (If it receives no value, then it uses a default value of 0.)
- Algorithm OM(m), m>0.

  1. The source process sends its value to every other process.

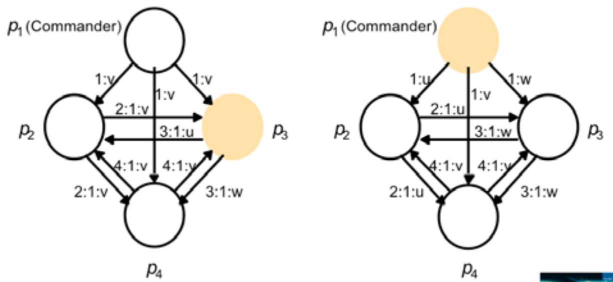
  2. For each i, let vi be the value process i receives from the source. (If it receives no value, then it uses a default value of 0.) Processor i acts as the new source and initiates OM(m-1) by sending vi to each of the (n-2) other processes.

  3. For each i and each j (<>i), let vj be the value process i received from process j in Step 2, using OM(m-1). (If it receives no value, then it uses a default value of 0.) Process i uses the value majority (v1,v2,...,vn-1).

Ex. (m=1, n=3m+1=4), assume p3 is faulty

- OM(1): p1 sends v to p2, p3, p4. (n-1 messages)
- OM(0): Each pi acts as a source process, sends its value to each otherpj, j<>i.
  - o So p2 sends v to p3, p4; p4 sends v to p2, p3.
  - o Since p3 is faulty it sends v to p3, w to p4.
  - o ((n-1)(n-2) messages)
- Step 3 of OM(1):
  - o p2 receives (v,v,v) and decides v;
  - o p4 receives (v,w,v) and decides v;
  - o p3 can decide anything.
- Number of rounds: 2
- Number of messages: (n-1) + (n-1)(n-2)

Solution with four byzantine generals



Faulty processes are shown shaded

Intuition: vote.

Ex. (m=2, n=3m+1=7), assume p2 and p3 are faulty

- OM(2): p0 sends 1 to p1,...,p6. (n-1 messages)
- OM(1): Each pi acts as a source process to communicate what it received from the previous source; other processes need to agree on what it received at the end of this round.
- p1: sends 1 to p2,...,p6
- OM(0),p1: p2,...,p6 send the values they received to each other.
- p2 and p3 can lie, but p4,p5,p6 must communicate 1.
- p2, p3 can decide anything, say they decide 0
- p4,p5,p6 receive (?,?,1,1,1) ==> 1 (i.e. "p1 received 1 from p0")
- p2: sends 1 to p1, p3 and 0 to p4,p5,p6
- OM(0),p2: p1,p3,p4,p5,p6 send the values they received to each other.
- p3 can lie, but p1 must send 1, p4,p5,p6 must send 0.
- p3 can decide anything, say it decides 1
- p1 receives (1,0,0,0,0) ==> 0 (i.e. "p2 received 0 from p0")
- p4,p5,p6 receive (1,1,0,0,0) ==> 0 and so on (p3 can do anything, say it is like p2; p4,p5,p6 must behave like p1)
- Step 3 of OM(1):
- p1,p4,p5,p6 see (1,0,0,1,1,1) ==> 1
- p2, p3 see whatever

Note:

- Each of the (n-1) occurrences of OM(1) execute in parallel (i.e. during the same round). So each pi, acting as the new source, sends (n-2) messages at the beginning of round 2.
- Each of the (n-1)(n-2) occurrences of OM(0) also execute in parallel in round 3, so at the beginning of round 3, each pi sends (n-2) messages to each other pj.

- Number of rounds: 3
- Number of messages:
  Round 1: (n-1)
  Round 2: (n-1)(n-2)
  Round 3: (n-1)(n-2)(n-3)

  (n-1)+(n-1)(n-2)+(n-1)(n-2)(n-3)

Complexity

- In general:
  - Number of rounds: m+1
  - Number of messages: (n-1)+(n-1)(n-2)+...+(n-1)(n-2)...(n-m)
  - O(n**m)

Correctness

- Lemma. For any m and k, Algorithm OM(m) satisfies the validity condition if there are more than 2k+m processes and at most k faulty processes.
- Proof by induction:
- Base case - OM(0) holds since the source is non-faulty by hypothesis and all messages sent are delivered.
- Inductive step - We now assume it is true for m-1, m>0 and prove it for m.
  - Step 1: The non-faulty source process sends v to the other (n-1) processes.
  - Step 2: Each non-faulty processes applies OM(m-1) with (n-1) processes.
  - By hypothesis, n>2k+m so n-1>2k+m-1 and we can use the inductive hypothesis to conclude that every non-faulty process gets vi=v for each non-faulty process pj. Since there are at most k faulty processes, and n-1>2k+(m-1)>=2k a majority of the n-1 processes are non-faulty.
  - Hence, each non-faulty process has vi=v for a majority of the n-1 values i, so it obtains v is step 3, proving the validity condition.
- QED

Correctness

- Theorem: For any m, Algorithm OM(m) satifies the correctness and validity conditions if there are more than 3m processes and at most m faulty processes.
- Proof by induction on m:
- Base case - OM(0) trivially holds (no traitors)
- Inductive step - Assume that the theorem is true for OM(m-1) and show it for OM(m), m>0.
  - Assume that p0 is non-faulty. By taking k=m in Lemma 1, we see that OM(m) satisfies validity. Correctness follows from validity if the source is non-faulty.
  - Assume that p0 is faulty. Then at most (m-1) of the remaining processes can be faulty. Since there are more than 3m processes, there are more than 3m-1 processes other than p0 and 3m-1>3(m-1). We may therefore apply the induction hypothesis to conclude that OM(m-1) satisfies correctness and validity. Hence, for each j, any two non-faulty processes get the same value for vj in step 3. Any two non-faulty processes will therefore obtain the same vector of values and therefore obtain the same value using majority, proving correctness.
- QED

Summary: Byzantine Failures

- A solution exists if less than one-third are faulty (N > 3m).
- It works only if communication is synchronous.
- Like fail-stop consensus, the algorithm requires m+1 rounds.
- The algorithm is very expensive and therefore impractical.
  - Number of messages is exponential in the number of rounds.
- Signed messages make the problem easier (*authenticated byzantine*).
  - In general case, the failure bounds (N > 3m) are not affected.
  - Practical algorithms exist for N > 3m. [Castro&Liskov]

Byzantine Agreement

- Must find a way of identifying faulty nodes and corrupted/forged messages
- General strategy is to have everyone communicate not only their own info, but everything they've heard from everyone else.
  - This allows catching lying, cheating nodes
  - If there is a majority that behaves consistently with each other, can reach consensus.

Fischer-Lynch-Patterson (1985)

- No consensus can be guaranteed in an asynchronous communication system in the presence of any failures.
- Intuition: a “failed” process may just be slow, and can rise from the dead at exactly the wrong time.
- Consensus may occur recognizably on occasion, or often.
  - e.g., if no inconveniently delayed messages
- FLP implies that no agreement can be guaranteed in an asynchronous system with byzantine failures either.

Consensus in Practice I

- What do these results mean in an asynchronous world?
  - Unfortunately, the Internet is asynchronous, even if we believe that all faults are eventually repaired.
  - Synchronized clocks and predictable execution times don’t change this essential fact.
- Even a single faulty process can prevent consensus.
- The FLP impossibility result extends to:
  - Reliable ordered multicast communication in groups
  - Transaction commit for coordinated atomic updates
  - Consistent replication
- These are practical necessities, so what are we to do?

Consensus in Practice II

- We can use some tricks to apply synchronous algorithms:
  - *Fault masking*: assume that failed processes always recover, and define a way to reintegrate them into the group.
    - If you haven’t heard from a process, just keep waiting…
    - A round terminates when every expected message is received.
  - *Failure detectors*: construct a failure detector that can determine if a process has failed.
    - A round terminates when every expected message is received, or the failure detector reports that its sender has failed.
- But: protocols may block in pathological scenarios, and they may misbehave if a failure detector is wrong.

Recovery for Fault Masking

- In a distributed system, a recovered node’s state must also be consistent with the states of other nodes.
  - E.g., what if a recovered node has forgotten an important event that others have remembered?
- A functioning node may need to respond to a peer’s recovery.
  - rebuild the state of the recovering node, and/or
  - discard local state, and/or
  - abort/restart operations/interactions in progress
    - e.g., *two-phase commit* protocol
- *How to know if a peer has failed and recovered?*

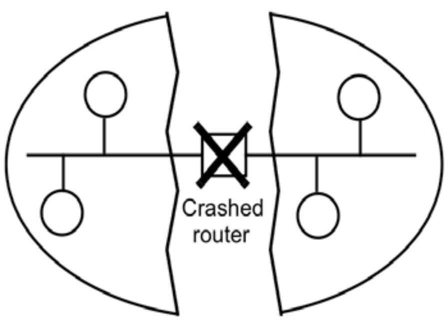
Failure Detectors

- First problem: how to detect that a member has failed?
  - o pings, timeouts, beacons, heartbeats
  - o recovery notifications
    - "I was gone for awhile, but now I'm back."
- Is the failure detector *accurate*?
- Is the failure detector *live (complete)*?
- *In an asynchronous system, it is possible for a failure detector to be accurate or live, but not both.*
  - o FLP tells us that it is impossible for an asynchronous system to agree on *anything* with accuracy and liveness!

Failure Detectors in Real Systems

- Use a failure detector that is live but not accurate.
  - Assume bounded processing delays and delivery times.
  - Timeout with multiple retries detects failure accurately with high probability. Tune it to observed latencies.
  - If a "failed" site turns out to be alive, then restore it or kill it (*fencing, fail-silent*).
- Use a recovery detector that is accurate but not live.
  - "I'm back....hey, did anyone hear me?"
- What do we assume about communication failures?
  - How much pinging is enough?
  - 1-to-N, N-to-N, ring?
- What about network partitions?

A network partition



Acknowledgments

- This lecture note is based on
  - o CIS 505 lecture notes by Honghui Lu
  - o CPS 212 lecture notes by Jeff Chase
  - o CSE 480 lecture notes by Michael Greenwald
  - o CSE 480 lecture notes by Insup Lee