Consensus & Agreement

Group Communication
- Unicast messages: from a single source to a single destination
- Multicast messages: from a single source to multiple destinations (designated as a group)
- Issues:
  - Fault tolerance: two kinds of faults in distributed systems
    - "Crash faults" (also known as fail-stop or benign faults): process fails and simply stops operating
    - "Byzantine faults": process fails and acts in an arbitrary manner (or malicious agent is trying to bring down the system)
  - Ordering:
    - Achieve some kind of consistency in how messages of different multicasts are delivered to the processes

Unicast messages: from a single source to a single destination
Multicast messages: from a single source to multiple destinations (designated as a group)

Issues:
- Fault tolerance: two kinds of faults in distributed systems
  - "Crash faults" (also known as fail-stop or benign faults): process fails and simply stops operating
  - "Byzantine faults": process fails and acts in an arbitrary manner (or malicious agent is trying to bring down the system)
- Ordering:
  - Achieve some kind of consistency in how messages of different multicasts are delivered to the processes

Basic Multicast
- Channels are assumed to be reliable (do not corrupt messages and deliver them exactly once)
- A straightforward way to implement B-multicast is to use a reliable one-to-one send operation:
  - B-multicast(g,m): for each process p in g, send (p,m).
  - receive(m): B-deliver(m) at p.
- A basic multicast primitive guarantees a correct process will eventually deliver the message, as long as the multicaster (sender) does not crash.

Reliable Multicast
- Desired properties:
  - Integrity: A correct (i.e., non-faulty) process p delivers a message m at most once.
  - Validity: If a correct process multicasts message m, then it will eventually deliver m. (Local liveness)
  - Agreement: If a correct process delivers message m, then all the other correct processes in group(g) will eventually deliver m.
  - Property of "all or nothing."
- Validity and agreement together ensure overall liveness
- Question: how do you build reliable multicast using basic multicast?

Reliable multicast (contd.)

On initialization:
Received = {};
For process p to B-multicast message m to group g
B-multicast(g,m); // p ∈ g is included as a destination
On B-deliver(m) at process q with g = group(m)
if (m ∈ Received)
then
   Received = Received ∪ {m};
   if (q ∈ p) then B-multicast(g,m); end if
   R-deliver m;
end if

Ordered Multicast
- Desirable ordering properties:
  - FIFO ordering: if a correct process issues multicat(g,m) and then multicat(g,m’), then every correct process that delivers m’ will deliver m before m’.
  - Causal ordering: if multicat(g,m) → multicat(g,m’) then any correct process that delivers m’ will deliver m before m’.
  - Total ordering: If a correct process delivers message m before m’, then any other correct process that delivers m’ will deliver m before m’.
- Causal ordering implies FIFO ordering
- Causal ordering does not imply total ordering
- Total ordering does not imply causal ordering
Implementing Total Ordering

1. Multicast a message, solicit sequence numbers from processes, multicast a sequence number that is computed based on solicited values.

2. Each process, $q$, keeps:
   - $A_q$: the largest agreed sequence number it has seen
   - $P_q$: its own largest proposed sequence number

3. Process $p$ multicasts $<m, i>$ to $g$, where $i$ is a unique identifier for $m$.

4. Each process $q$ replies to the sender $p$ with a proposal for the message’s agreed sequence number of $P_q := \text{Max}(A_q, P_q) + 1$.

5. Places it in its hold-back queue.

6. $p$ collects all the proposed sequence numbers and selects the largest as the next agreed sequence number, $a$.

7. It B-multicasts $<i, a>$ to $g$.

8. Recipients set $A_q := \text{Max}(A_q, a)$, attach $a$ to the message and reorder hold-back queue.

Consensus

- **Consensus**: N Processes agree on a value.
- For example, synchronized action (go / abort)
- Consensus may have to be reached in the presence of failure.
- Process failure – process crash (fail-stop failure), arbitrary failure.
- Communication failure – lost or corrupted messages.

- In a consensus algorithm:
  - All $P_i$ start in an “undecided” state.
  - Each $P_i$ proposes a value $v_i$ from a set $D$ and communicates it to some or all other processes.
  - A consensus is reached if all non-failed processes agree on the same value, $d$.
  - Each non-failed $P_i$ sets its decision variable to $d$ and changes its state to “decided.”

- **Consensus Requirements**
  - **Termination**: Eventually each correct process sets its decision value.
  - **Agreement**: The decision value is the same for all correct processes, i.e., if $p_i$ and $p_j$ are correct and have entered the decided state, then $d_i = d_j$.
  - **Integrity**: If all correct processes $P_i$ propose the same value, $d$, then any correct process in the decided state has decision value = $d$.

- **Rich problem space**:
  - Synchronous vs. asynchronous systems
  - Fail-stop vs. byzantine failures
  - Process vs. message failures

Interactive Consistency Problem

- Interactive consistency is a special case of consensus where processes agree on a vector of values, one value for each process.

Byzantine Generals Problem

- 3 or more generals need to agree to attack or to retreat.
- **Problem**
  - The commander issues the order.
  - One or more of the generals (including the commander) could be a traitor who will give wrong information.
  - Each general sends his/her information to all others (assuming reliable communication).
  - Once each general has collected all values, it determines the right value (attack or retreat).

- The requirements are termination, agreement, and integrity.
Problem Equivalence

- Interactive consistency (IC) can be solved if there is a solution for Byzantine Generals (BG) problem:
  - Just run BG “n” times
- Consensus (C) can be solved if there is a solution for IC:
  - Run IC to produce a vector of values at each process
  - Then apply the majority function on the vector
  - Resulting value is the consensus value
- If no majority, choose a “bottom” value
- BG is solvable if there is a solution to C:
  - Commander sends its proposed value to itself and each of the other generals
  - All processes run C with the values received
  - Resulting consensus value is the value required by BG

Consensus in a synchronous system

- For a system with at most f processes crashing, the algorithm proceeds in f+1 rounds, using basic multicast.
- Values_{i}^{r}: the set of proposed values known to P_{i} at the beginning of round r.
- Initially Values_{i}^{0} = {} ; Values_{i}^{1} = \{v_{i}\}
  
  for round = 1 to f+1 do
    B-multicast (Values_{i}^{r} – Values_{i}^{r-1})
    Values_{i}^{r+1} = Values_{i}^{r+1} \cup V_{j}
  end
  d_{i} = \text{minimum}(Values_{i}^{f+1})

Proof of correctness

- Proof by contradiction.
- Assume that two processes differ in their final set of values.
- Assume that p_{i} possesses a value v that p_{j} does not possess.
  - A third process (p_{k}) sent v to p_{i} and crashed before sending v to p_{j}.
  - Any process sending v in the previous round must have crashed; otherwise, both p_{k} and p_{i} should have received v.
  - Proceeding in this way, we infer at least one crash in each of the preceding rounds.
- But we have assumed at most f crashes can occur and there are f+1 rounds → contradiction.

Byzantine Generals in a synchronous system

- A faulty process may send any message with any value at any time; or it may omit to send any message.
- In the case of arbitrary failure, no solution exists if N<=3f.

Solution

- To solve the Byzantine generals problem in a synchronous system, we require N>=3f+1
- Consider N=4, f=1
  - In the first round, the commander sends a value to each of the other generals
  - In the second round, each of the other generals sends the value it received to its peers.
  - The correct generals need only apply a simple majority function on the set of values received.
Consensus Algorithms for Byzantine Failures

- Minimum number of rounds is $f + 1$
- Exponential tree algorithm:
  - Each processor maintains a tree data structure in its local state
  - Each node of the tree is labeled with a sequence of processor indices with no repeats
    - Root's label is an empty sequence
    - Root has $n$ children labeled 0 through $n-1$
    - Child node labeled "i" has $n-1$ children labeled 0 through $i-1$ and $i+1$ through $n-1$
    - In general, node at level $d$ with label $v$ has $n - d$ children skipping any index already present in $v$
    - Nodes at level $f+1$ are the leaves

Exponential Tree Algorithm

- Each processor fills in the tree nodes with values as the rounds go by
- Initially, store your input in the root (level 0)
- Round 1: send level 0 of your tree (the root); store value received from $p_j$ in node $j$ (level 1)
- Round 2: send level 1 of your tree; store value received from $p_j$ for node $k$ in node "k:j" (level 2)
  - This is the “value that $p_j$ told me that $p_k$ told $p_j$”
  - Continue for $f + 1$ rounds

Computing Decision Value

- In the last round, each processor uses the values in its tree to compute its decision
  - Decision is $\text{resolve}(\lambda)$
  - Where $\text{resolve}(\pi)$ equals:
    - Value in tree node labeled "\pi" if it is a leaf
    - $\text{majority}(\text{resolve}(\pi')) : \pi' \text{ is a child of } \pi$

Building Tree: top-down phase

- Assume that nodes 0, 1, and 2 are legitimate; they contribute value 5
- Assume that node 3 is byzantine

Resolving nodes

- Resolve a leaf node: return the value of the node
- Resolve an internal node: return the majority value of children
- Decision by processor: resolve the root
Proof of algorithm

- **Resolve Lemma:** Non-faulty processor \( p_i \)'s resolved value for node \( \pi = \pi' j \) equals what \( p_j \) has stored for \( \pi' \).
  - **Proof:** By induction on the height of \( \pi \).
    - **Basis:** \( \pi \) is a leaf.
      1) Then \( p_i \) stores in node \( \pi \) what it received from \( p_j \) in the last round.
      2) For leaves, the resolved value is the tree value.
  - **Proof (contd.)**
    - **Induction Step:** \( \pi \) is not a leaf.
      Suppose in contradiction \( \pi \) is not common.
      Then every \( \pi' \)-to-leaf path has the property that every \( \pi' \)-to-leaf path has a common node.
      Since the height of \( \pi' \) is smaller than the height of \( \pi \), the inductive hypothesis implies that \( \pi' \) is common.
      Therefore, all non-faulty processors compute the same resolved value for \( \pi \), and thus \( \pi \) is common.
Prove that root has the property
- Show that every root-to-leaf path has a common node:
  - There are f+2 nodes on a root-to-leaf path
  - The label of each non-root node on a root-to-leaf path ends in a distinct processor index (the processor from which the value is to be received)
  - At least one of these indices is that of a non-faulty processor
  - "Resolve Lemma" implies that the node whose label ends with a non-faulty processor is a common node

Polynomial Algorithm for Byzantine Agreement
- Can reduce the message size with a simple algorithm that increases the number of processors to n > 4f and number of rounds to 2(f + 1)
- Phase King Algorithm: Uses f + 1 phases, each taking two rounds
  Code for p
  pref = my input
  First round of phase k:
  send pref to all
  receive prefs of others
  let "maj" be the value that occurs > n/2 times among all prefs (0 if none)
  let "mult" be the number of times "maj" occurs

Algorithm (contd.)
Second round of phase k:
if my Proc == k then send "maj" // I am the phase king
receive tie-breaker from p_k
if mult > n/2 + f
  then pref = maj
else pref = tie-breaker
if k == f+1 then decide pref

Proof of Phase King Algorithm
- Lemma: If all non-faulty processors prefer v at start of phase k, then all do at end of phase k.
  Proof:
  - Each non-faulty processor receives at least n - f preferences (including its own) for v in the first round of phase k
  - Since n > 4f:
    - n/2 > 2f
    - (n - n/2) > f + f
    - n - f > n/2 + f.
  - Thus the processors still prefer v.
- Validity: follows from above lemma
  - All non-faulty processors start with the same value

Proof (contd.)
- Lemma: If the king of phase k is non-faulty, then all non-faulty processors have the same preference at the end of phase k.
  Proof:
  - Consider two non-faulty processors p_i and p_j
  - Case 1: p_i and p_j both use p_k's tie-breaker. Since p_k is non-faulty, they agree
  - Case 2: p_i uses its majority value and p_j uses the king's tie-breaker
    - p_i's majority value is v
    - p_i receives more than n/2 + f preferences for v
    - p_i receives more than n/2 preferences for v
    - p_i's tie-breaker is v
  - Since there are f + 1 phases, at least one has a non-faulty king
  - At the end of that phase, all non-faulty processors have the same preference
  - From that phase onward, the non-faulty preferences stay the same
  - Thus the decisions are the same.
Fischer-Lynch-Patterson (1985)

- No completely asynchronous consensus protocol can tolerate even a single unannounced process death

Assumptions

- Fail-stop failure:
  - Impossibility result holds for byzantine failure
- Reliable message system:
  - Messages are delivered correctly and exactly once
- Asynchronous:
  - No assumptions regarding the relative speeds of processes or the delay time in delivering a message
  - No synchronized clock
    - Algorithms based on time-out cannot be used
  - No ability to detect the death of a process

The weak consensus problem

- Initial state: 0 or 1 (input register)
- Decision state:
  - Non-faulty process decides on a value in \{0, 1\}
  - Stores the value in a write-once output register
- Requirement:
  - All non-faulty processes that make a decision must choose the same value.
  - For proof: assume that some processes eventually make a decision (weaker requirement)
- Trivial solution is ruled out
  - Cannot choose 0 arbitrarily
  - Processes modeled by deterministic state machines

Notation

- A configuration consists of
  - All internal state of each process, the contents of message buffer
- Message system (think of the undelivered messages stored in a bag)
  - send(p, m)
  - receive(p) \text{ returns some message to be received by } p \text{ or an empty message}
- A step is a transition of one configuration \( C \) to another \( e(C) \), including 2 phases:
  - First, receive(p) to get a message \( m \)
  - Based on p's internal state and \( m \), p enters a new internal state and sends finite messages to other
- \( e = (p, m) \) is called an event and said \( e \) can be applied to \( C \)

Schedule, run, reachable and accessible

- A schedule from \( C \)
  - a finite or infinite sequence of events that can be applied, in turn, starting from \( C \)
  - The associated sequence of steps is called a run
  - \( (C) \) denotes the resulting configuration and is said to be reachable from \( C \)
- An accessible configuration \( C \)
  - If \( C \) is reachable from some initial configuration

Lemma 1

- Suppose that from some configuration \( C \), the schedules \( \gamma_1 \) and \( \gamma_2 \) lead to configuration \( C_1 \) and \( C_2 \) respectively.
  - If the sets of processes taking steps in \( \gamma_1 \) and \( \gamma_2 \), respectively, are disjoint:
    - Then \( \gamma_1 \) can be applied to \( C_2 \) and \( \gamma_2 \) can be applied to \( C_1 \), and both lead to the same configuration.
Definitions
- A process is non-faulty
  - If it takes infinitely many steps
- A configuration $C$ has decision value $v$ if some process $p$ is in a decision state with output register containing $v$.
- Deciding run
  - Some process reaches a decision state
- Admissible run
  - At most one process is faulty and all messages sent to non-faulty processes are eventually received

Bivalent, 0-valent/1-valent
- Let $C$ be a configuration, $V$ the set of decision values of configurations reachable from $C$
  - $C$ is bivalent if $|V| = 2$.
  - $C$ is univalent if $|V| = 1$.
- 0-valent or 1-valent according to the corresponding decision value.

Correctness
- A consensus protocol $P$ is totally correct in spite of one fault:
  - No trivial solutions (there are some configurations that lead to result 0 and some that lead to result 1)
  - No accessible configuration has more than one decision value
  - Every admissible run is a deciding run

Theorem 1
- No consensus protocol is totally correct in spite of one fault.
  - Proof strategy:
    - There must be some initial configuration that is bivalent
    - Consider some event $e = (p, m)$ that is applicable to a bivalent configuration, $C$
      - Consider the set of configurations reachable from $C$ w/o applying $e$ (let this set be $\Sigma$)
      - Apply $e$ to each one of these configurations to get the set $D$
      - Show that $D$ contains a bivalent configuration
    - Construct an infinite sequence of stages where each stage starts with a bivalent configuration and ends with a bivalent configuration

Lemma 2
- $P$ has a bivalent initial configuration (Proof by contradiction)
  - Consider configuration $C_1 = \{ 0, 0, 0, \ldots, 0 \}$
    - Every processor starts with input value 0
    - $C_1$ is 0-valent
  - Consider configuration $C_2 = \{ 1, 1, 1, \ldots, 1 \}$
    - $C_2$ is 1-valent
  - Transform $C_1$ to $C_2$ with at most one processor changing its input value
    - There must be two configurations $C_3$ and $C_4$
      - $C_3$ is 0-valent, $C_4$ is 1-valent
        - Some processor $p$ changed its value from 0 to 1
      - Consider some admissible deciding run from $C_3$ involving no $p$-events.
        - Let $e$ be associated schedule.
        - Let $e$ be associated schedule.
        - Apply $e$ to $C_4$. Clearly, resulting state should be 0.
        - Implies contradiction.

Lemma 3
- Let $C$ be a bivalent configuration of $P$.
  - Let $e = (p, m)$ be an event that is applicable to $C$.
    - Let $\Sigma$ be the set of configurations reachable from $C$ without applying $e$, and let $D = e(\Sigma) = \{ e(E) | E \in \Sigma \}$.
    - Then, $D$ contains a bivalent configuration.
Proof

There must be two states such that:
\[ C \sim E_0 \text{ and } C \sim E_1 \]
where \( E_0 \) is 0-valent and \( E_1 \) is 1-valent.

Consider \( E_0 \):
- If \( E_0 \) belongs to \( \Sigma \), then \( e(E_0) = F_0 \) belongs to \( D \).
- If \( E_0 \) does not belong to \( \Sigma \), then there is a \( F_0 \):
  - Such that \( F_0 \) belongs to \( D \).
  - \( F_0 \sim E_0 \).
- In either case, there is a \( F_0 \in D \) and \( F_0 \) is 0-valent.

Similarly there exists a \( F_1 \) which is 1-valent and \( F_1 \in D \).

\( D \) contains 0-valent and 1-valent configurations.

Proof (contd.)

Assume that the event that transforms \( G_0 \) to \( G_1 \) is \( e' = (p', m') \) and let \( p' \neq p \).
- Recall that \( p \) is the processor with the delayed message (and the delayed event \( e \)).
- \( e' \) is applicable to \( D_0 \) and transforms \( D_0 \) to \( D_1 \) (commutativity lemma).
- What does this imply?

Proof (contd.)

If \( p' \) is same as \( p \), consider some configuration \( A \) that is reachable from \( G_0 \) that involves no events to \( p \), and is deciding. Let \( \omega \) be the schedule.
Proof (contd.)

Construction of infinite run

Proof Wrapup

Goal is to construct an infinite sequence of events:
- No processor fails
- Each processor executes an infinite steps
- All messages sent to a processor is delivered in finite time
- Every configuration in the sequence is bivalent

Previous theorem states that:
- Start with a bivalent configuration
- Delay some message
- Can always find some other bivalent configuration that is reached by delivering the message

Construction of infinite run (contd.)

Block a message for the next processor, construct another possible bivalent configuration

Construction can go on for ever:
- No faults (infinite steps for each processor, messages delivered in finite time)
- Always goes from one bivalent configuration to another bivalent configuration

Paxos Consensus

Assume that a collection of processes that “can” propose values, choose a value
- Only a value that has been proposed may be chosen
- Only a single value is chosen

Three classes of agents: proposers, acceptors, and learners
- A single process may act as more than one agent

Model:
- Asynchronous messages
- Agents operate at arbitrary speed, may fail by starting, and may restart. (If agents fail and restart, assume that there is non-volatile storage.)
- Guarantee safety and not liveness

Simple solutions

Have a single acceptor agent
- Proposers send a proposal to the acceptor:
  - Acceptor chooses the first proposed value
  - Rejects all subsequent values
  - Failure of acceptor means no further progress

Let’s use multiple acceptor agents
- Proposer sends a value to a large enough set of acceptors
- What is large enough?
  - Some majority of acceptors, which implies that only one value will be chosen
  - Because any two majorities will have at least one common acceptor
Some Other Ground Rules

- There might be just one proposer
  - Number of proposers is unknown
- No liveness requirements:
  - If a proposal does not succeed, you can always restart a new proposal
- The three important actions in the system are:
  - Proposing a value
  - Accepting a value
  - Choosing a value (if a majority of acceptors accept a value)

Solutions that don’t work

- There could be just one proposed value
  - An acceptor should accept the first value

Refinements

- Allow an acceptor to accept multiple proposals
  - Which implies that multiple proposals could be chosen
  - Trivially satisfies the condition that only a single value is chosen
  - Requires coordination between proposers and acceptors
- Let proposals be ordered
  - One possibility: each proposal is a 2-tuple [proposal-number, processor-number]
- Ensure the following property:
  - P2: If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has the value v

More refinements

- Consider the following property:
  - P3: If a proposal with value v is chosen, then every higher-numbered proposal that is accepted has the value v
  - P3 ==> P2 ==> P1
- Consider an even stronger property:
  - P4: If a proposal with value v is chosen, then every higher-numbered proposal that is proposed by any processor has value v
  - P4 ==> P3 ==> P2 ==> P1

One more refinement

P5: For a proposal numbered n with value v:
- It is issued only if there is a set S consisting of a majority of acceptors such that either:
  - No acceptor in S has accepted any proposal numbered less than n, or
  - v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S

One can satisfy P4 by maintaining the invariant P5

How does one enforce P5?

Phase 1: prepare request

1. A proposer chooses a new proposal version number n, and sends a prepare request ("prepare", n) to a majority of acceptors:
   
   (a) Can I make a proposal with number n?
   
   (b) If yes, do you suggest some value for my proposal?
Phase 1 (receive prepare request)

(2) If an acceptor receives a prepare request
("prepare", n) with n greater than that of any
prepare request it has already responded, sends
out ("ack", n, n', v') or ("ack", n, ⊥, ⊥)

(a) responds to the request with a promise not to accept
any more proposals numbered less than n.

(b) suggest the value v of the highest-number proposal
that it has accepted if any, otherwise ⊥.

In Well-Behaved Runs

Phase 2: accept request

(3) If the proposer receives the requested responses from a
majority of the acceptors, then it can issue a propose request
("propose", n, v) with number n and value v.

(a) n is the number that appears in the prepare request.

(b) v is the value of the highest-numbered proposal among the
responses.

(4) If the acceptor receives a request ("propose", n, v), it accepts
the proposal unless it has already responded to a prepare
request having a number greater than n.

Example

In Well-Behaved Runs

Example

Example
Paxos: other issues

- A proposer can make multiple proposals
  - It can abandon a proposal in the middle of the protocol at any time
  - Probably a good idea to abandon a proposal if some processor has begun trying to issue a higher-numbered one

- If an acceptor ignores a prepare or accept request because it has already received a prepare request with a higher number:
  - It should probably inform the proposer who should then abandon its proposal

- Persistent storage:
  - Each acceptor needs to remember the highest numbered proposal it has accepted and the highest numbered prepare request that it has asked.

Progress

- Easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers
  - P completes phase 1 for a proposal numbered n1
  - Q completes phase 1 for a proposal numbered n2 > n1
  - P's accept requests in phase 2 are ignored by some of the processors
  - P begins a new proposal with a proposal number n3 > n2
  - And so on...

Announcements

- Class lecture notes updated
- Upcoming topics:
  - Secure routing (avoiding denial of service attacks)
  - Overlay/sensor networks
- Project checkpoint due