What is Global Snapshot?

- Distributed Snapshot
  - What does a global snapshot even mean?
- In The Cloud
  - In a cloud: each application or service is running on multiple servers
  - Servers handling concurrent events and interacting with each other
  - The ability to obtain a global photograph of the system is important
  - Some uses of having a global picture of the system
    - Checkpointing: can restart distributed application on failure
    - Garbage collection of objects: objects at servers that don't have any other objects (at any servers) with pointers to them
    - Deadlock detection: Useful in database transaction systems
    - Termination of computation: Useful in batch computing systems like Folding@Home, SETI@Home
- What is a Global Snapshot
  - Global Snapshot = Global State =
    - individual state of each process in the distributed system + Individual state of each communication channel in the distributed system
  - Capture the instantaneous state of each process
  - And the instantaneous state of each communication channel, i.e., message in transit on the channels
- Obvious First Solution
  - Synchronize clocks of all processes
  - Ask all processes to record their states at known time t
  - Problems?
    - Time synchronization always has error
    - Also, does not record the state of messages in the channels
  - Again: synchronization not required - causality is enough!
- Moving from state to state
  - Whenever an event happens anywhere in the system, the global state changes
    - process receives message
    - process sends message
    - process takes a step
  - State to state movement obeys causality
    - Next: Causal algorithm for Global Snapshot calculation

Global Snapshot Algorithm

- System Model
  - Problem: Record a global snapshot
  - System Model:
    - N processes in the system
    - There are two uni-directional communication channels between each ordered process pair
    - communication channels are FIFO-ordered
    - No failure
    - All messages arrive intact, and are not duplicated
      - other papers later relaxed some of these assumptions
- Requirements
  - Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages

- Each process is able to record its own state
  - Process state: Application-defined state or, in the worst case:
  - its heap, registers, program counter, code, etc. (essentially the coredump)
- Global state is collected in a distributed manner
- Any process may initiate the snapshot
  - We'll assume just one snapshot run for now
- Chandy-lamport global snapshot algorithm
  - First, Initiator P_i records its own state
  - Initiator process creates special messages called "Marker" messages
    - Not an application message, doe not interfere with application messages
  - for j = 1 to N except i
    - P_i sends out a Marker message on outgoing channel C_ij
  - Starts recording the incoming messages on each of the incoming channels at P_i:C_ij (for j = 1 to N except i)
  - Whenever a process P-i receives a Marker message on an incoming channel C_ki
    - if (this is the first Marker P_i is seeing)
      - P_i records its own state first
      - Marks the state of channel C_ki as "empty"
      - For j = 1 to N except i
        - P_i sends out a Marker message on outgoing channel C_ij
      - Starts recording the incoming messages on each of the incoming channels at P_i: C_ji (for j = 1 to N except i and k)
    - else // already seen a Marker message
      - mark the state of channel C_ki as all the messages that have arrived on it since recording was turned on for C_ki
  - The algorithm terminates when
    - all processes have received a Marker
      - To record their own state
    - all processes have received a Marker on all the (n-1) incoming channels at each
      - to record the state of all channels
  - Then, (if needed), a central server collects all these partial state pieces to obtain the full global snapshot
- Next
  - Global snapshot is causally correct

Consistent Cuts

- Cuts
  - cut = time frontier at each process and at each channel
  - events at the process/channel that happen before the cut are "in the cut"
    - and happening after the cut are "out of the cut"
- Consistent Cuts
  - a cut that obeys causality
  - a cut C is a consistent cut if and only if
    - for (each pair of events e, f in the system)
      - such that event e is in the cut C, and if f->e (f happens-before e)
        - Then: Event f is also in the cut C
- Any run of Chandy-Lamport Global Snapshot algorithm creates a consistent cut

- Chandy-lamport global snapshot algorithm creates a consistent cut
  - Let e_i and e_j be events occurring at P_i and P_j, respectively such that
    - e_i -> e_j (e_i happens before e_j)
  - The snapshot algorithm ensures that
    - if e_j is in the cut then e_i is also in the cut
  - That is: if e_j -> <P_j records its state>, then
    - if must be true that e_i -> <P_i records its state>
  - if e_j -> <P_j records its state>, then it must be true that e_i -> <P_i records its state>
    - By contradiction, suppose e_j -> <P_j records its state> and <P_i records its state> -> e_i
    - Consider the path of app messages (through other processes) that go from e_i -> e_j
    - Due to FIFO ordering, markers on each link in above path will precede regular app messages
    - Thus, since <P_i records its state> -> e_i, it must be true that P_j received a marker before e_j
    - Thus e_j is not in the cur -> contradiction

Safety and Liveness

- Correctness in distributed systems
  - liveness and safety
- Liveness
  - guarantee that something good will happen, eventually
  - eventually == does not imply a time bound, but if you let the system run long enough, then ….
  - Examples in Real World
    - Guarantee that “at least one of the athletes in the 100m final will win gold” is liveness
    - A criminal will eventually be jailed
  - Examples in Distributed System
    - Distributed computation: Guarantee that it will terminate
    - “Completeness” in failure detectors: every failures is eventually detected by some non-faulty process
    - In Consensus: All processes eventually decide on a value
- Safety
  - Safety = guarantee that something bad will never happen
  - Examples in Real World
    - A peace treaty between two nations provides safety
      - War will never happen
    - An innocent person will never be jailed
  - Examples in Distributed Systems
    - There is no deadlock in distributed transaction system
    - No object is orphaned in a distributed object system
    - “Accuracy” in failure detectors
    - In Consensus: No two processes decide on different values
- Can’t we guarantee both?
  - Can be difficult to satisfy both liveness and safety in an asynchronous distributed system
      - Failure Detector:
        - Completeness (Liveness) and Accuracy (Safety) cannot both be guaranteed by a failure detector in an asynchronous distributed system
      - Consensus:

        - Decisions (Liveness) and correct decisions (Safety) cannot both be guaranteed by any consensus protocol in an asynchronous distributed system
      - very difficult for legal systems (anywhere in the world) to guaranteed that all criminals are jailed (liveness) and no innocents are jailed (Safety)
- In the language of global states
  - recall that a distribute system moves from one global state to another global state via causal steps
  - Liveness w.r.t. a property Pr in a given state S means
    - S satisfies Pr, or there is some causal path of global states from S to S' where S' satisfies Pr
  - Safety w.r.t. a property Pr in a given state S means
    - S satisfies Pr, and all global states S' reachable from S also satisfy Pr
- Using global snapshot algorithm
  - Chandy-lamport algorithm can be used to detect global properties that are stable
    - stable = once true, stays true forever afterwards
  - Stable Liveness examples
    - Computation has terminated
  - Stable Non-Safety examples
    - There is a deadlock
    - An object is orphaned (no pointer point to it)
  - All stable global properties can be detected using the algorithm
    - Due it its causal correctness
- Summary
  - don't want to interrupt running distributed application
  - Output of chandy-lamport algorithm calculates global snapshot
  - can be used to detect stable global properties
  - Safety vs. Liveness

Multicast Ordering
- Multicast problem
  - a message that needs to be sent out in a group of processes
- Other communication forms
  - Multicast -> message sent to group of processes
  - Broadcast -> message sent to all processes (anywhere)
  - Unicast -> message sent from one sender process to one receiver process
- Who uses multicast?
  - A widely-used abstraction by almost all cloud systems
  - Storage systems like Cassandra or a database
    - replica servers for a key: Writes/reads to the key are multicast within the replica group
    - All servers: membership information is multicast across all servers in cluster
  - Online scoreboards
    - multicast to group of clients interest in the scores
  - Stock exchanges
    - group is the se of broker computers
    - groups of computers for High frequency trading
  - Air traffic control system
    - All controllers need to receive the same updates in the same order
- FIFO Ordering
  - Multicasts from each sender are received in the order they are sent, at all receivers
  - Don't worry about multicast from different senders

- More formally
    - If a correct process issues (sends) multicast(g,m) to group g and then multicast(g,m'), then every correct process that delivers m' would already have delivered m.
  - order of different senders does not matter
- Causal Ordering
  - Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
  - Formally
    - If multicast(g,m) -> multicast(g,m') then any correct process that delivers m' would already have delivered m.
    - (-> is Lamport's happens-before)
- Causal Vs. FIFO
  - Causal Ordering -> FIFO Ordering
  - Why?
    - If two multicasts M and M' are sent by the same process P, and M was sent before M', then M -> M'
    - Then a multicast protocol that implements causal ordering will obey FIFO ordering since M -> M'
  - Reverse is not true! FIFO ordering does not imply causal ordering
- Why Causal At All?
  - Group = set of your friends on a social network
  - A friend sees your message m, and she post a response (comment) m' to it
    - If friends receive m' before m, it wouldn't make sense
    - But if two friends post messages m'' and n'' concurrently, then they can be seen in any order at receivers
  - A variety of systems implement causal ordering: Social networks, bulletin boards, comments on websites, etc.
- Total Ordering
  - Also known as "Atomic Broadcast"'
  - Unlike FIFO and causal, this does not pay attention to order of multicast sending
  - Ensures all receivers receive all multicasts in the same order
  - Formally
    - If a correct process P delivers message m before m' (independent of the senders), then any other correct process P that delivers m; would already have delivered m.
  - May need to delay delivery of some messages at sender
- Hybrid Variants
  - Since FIFO/Causal are orthogonal to Total, can have hybrid ordering protocols too
    - FIFO-total hybrid protocol satisfies both FIFO and total orders
    - Causal-total hybrid protocol satisfies both Causal and total orders
- Implementation?

Implementing Multicast Ordering 1 FIFO

- Multicast Ordering
  - FIFO ordering
  - Causal ordering
  - Total ordering
- FIFO Multicast: Data Structures
  - Each receiver maintains a per-sender sequence number (integers)
    - processes P1 through PN
    - Pi maintains a vector of sequence numbers Pi[1…N] (initially all zeros)

    - Pi[j] is the latest sequence number Pi has received from Pj
- FIFO Multicast: Updating Rules
  - Send multicast at process Pj:
    - Set Pj[j] = Pj[j] + 1
    - Include new Pj[j] in multicast message as its sequence number
  - Receive multicast: If Pi receives a multicast from Pj with sequence number S in message
    - if(S == Pi[j] + 1) then
      - deliver message to application
      - Set Pi[j] = Pi[j] + 1
    - else buffer this multicast until above condition is true
- Total Ordering
  - Ensures all receivers receive all multicasts in the same order
  - Formally
    - If a correct process P delivers message m before m' (independent of the senders), then any other correct process P' that delivers m' would already have delivered m.
- Sequencer-Based Approach
  - Special process elected as leader or sequencer
  - Send multicast at process P_i:
    - send multicast message M to group and sequencer
  - Sequencer:
    - Maintains a global sequence number S (initially 0)
    - When it receives a multicast message M, it sets S = S + 1, and multicast <M,S>
  - Receive multicast at process P_i:
    - Pi maintains a local received global sequence number S_i (initially 0)
    - If Pi receives a multicast M from P_j, it buffers it until it both
      - P_i receives <M,S(M)> from sequencer, and
      - Si + 1 = S(M)
      - Then deliver it message to application and set Si = Si + 1

Implementing Multicast Ordering 2 Causal Ordering

- Causal Ordering
  - Multicast whose send events are causally related, must be received in the same causality-obeying order at all receivers
  - Formally
    - If multicast(g,m) -> multicast(g,m') then any correct process that delivers m' would already have delivered m.
    - (-> is Lamport's happens-before)
- Causal Multicast: Data structures
  - Each receiver maintains a vector of per-sender sequence numbers (integers)
    - Similar to FIFO multicast, but updating rules are different
    - Process P1 through PN
    - Pi maintains a vector Pi[1…N] (initially all zeros)
    - Pi[j] is the latest sequence number Pi has received from Pj
- Causal Multicast: updating rules
  - Send multicast at process Pj:
    - Set Pj[i] = Pj[i] + 1
    - include new entire vector Pj[1…N] in multicast message as its sequence number
  - Receive multicast: If Pi receives a multicast from Pj with vector M[1…N] (=Pj[1…N]) in message, buffer it until both
    - This message is the next one Pi is expecting from P, i.e.,

        - M[j] = Pi[j] + 1
    - All multicasts, anywhere in the group, which happened-before M have been received at Pi, e.e.,
        - For all k != j M[k] <= Pi[k]
        - i.e., Receiver satisfies causality
    - When above two conditions satisfied, deliver M to applications and set Pi[j] - M[j]
- Summary: Multicast ordering
    - Ordering of multicasts affect correctness of distributed systems using multicasts
    - Three popular ways of implementing order
        - FIFO, Causal, Total
    - And their implementations
    - What about reliability of multicasts?
    - What about failures

Reliable Multicast

- Reliable multicast
    - multicast loosely says that every process in the group receives all multicasts
        - Reliability is orthogonal to ordering
        - Can implement Reliable-FIFO, or Reliable-Causal, or Reliable-Total, or Reliable-Hybrid protocols
    - What about process failures?
    - Definition becomes vague
- Reliable Multicast (under failures)
    - Need all correct (i.e., non-faulty) processes to receive the same set of multicasts as all other correct processes
        - Faulty processes are unpredictable, so we won't worry about them
- Implementing Reliable Multicast
    - Let's assume we have reliable unicast (TCP) available to us
    - First-cut: Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients.
    - First-cut protocol does not satisfy reliability
        - If sender fails, some correct processes might receive multicast M, while other correct processes might not receive M
- Really Implementing Reliable Multicast
    - Trick: have receivers help the sender
    - Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients
    - When a receiver receives multicast M, it also sequentially sends M to all the group's processes
- Analysis
    - Not the most efficient multicast protocol, but reliable
    - Proof is by contradiction
    - Assumption two correct processes Pi and Pj are so that Pi received a multicast M and Pj did not receive that multicast M
        - Then Pi would have sequentially sent the multicast M to all group members, including Pj, and Pj would have received M
        - A contradiction
        - Hence our initial assumption must be false
        - Hence protocol preserves reliability
-

Virtual Synchrony

- Virtual Synchrony or View Synchrony
  - Attempts to preserve multicast ordering and reliability in spite of failures
  - Combines a membership protocol with a multicast protocol
  - Systems that implemented it (like Isis) have been used in NYSE, French Air, traffic control system, swiss stock exchange
- Views
  - Each process maintains a membership list
  - The membership list is called a View
  - An update to the membership list is called a View Change
    - Process join, leave, or failure
  - Virtual synchrony guarantees that all view changes are delivered in the same order at all correct processes
  - Views may be delivered at different physical times at processes, but they are delivered in the same order
- VSync Multicasts
  - A multicast M is said to be “delivered in a view V at process Pi” if
    - Pi receives view V, and then sometime before Pi receives the next view it delivers multicast M
  - Virtual synchrony ensures that
    - The set of multicasts delivered in a given view is the same set at all correct processes that were in that view
      - What happens in a View, stays in that View
    - The sender of the multicast message also belongs to that view
    - If a process Pi does not deliver a multicast M in view V while other processes in the view V delivered M in V, then Pi will be forcibly removed from the next view
- What about multicast ordering?
  - Again, orthogonal to virtual synchrony
  - The set of multicasts delivered in a view can be ordered either
    - FIFO
    - Causally
    - Totally
    - Hybrid scheme
- About that name
  - called “virtual synchrony” since in spite of running on an asynchronous network, it gives the appearance of a synchronous network underneath that obeys the same ordering at all processes
  - So can this virtually synchronous system be used to implement consensus?
  - No! VSync groups susceptible to partitioning
    - E.g., due to inaccurate failure detections
- Summary
  - Multicast an important building block for cloud computing systems
  - Depending on application need, can implement
    - Ordering
    - Reliability
    - Virtual synchrony
-
-
-

The Consensus Problem

- Give it a thought
  - have you ever wondered why distributed server vendors always only offer solutions that promise five-9's reliability, seven-9's reliability, but never 100% reliable?
  - The fault does not lie with the companies themselves, or the worthlessness of humanity
  - The fault lies in the impossibility of consensus
- What is common to all of these
  - A group of servers attempting:
    - make sure that all of them receive the same updates in the same order as each other
      - **Reliable Multicast**
    - To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously
      - **Membership/Failure Detection**
    - Elect a leader among them, and let everyone in the group know about it
      - **Leader Election**
    - To ensure mutually exclusive (one process at a time only) access to a critical resource like a file
      - **Mutual Exclusion**
- So what is common?
  - Let's call each server a "process" (think of the daemon at each server)
  - All of these were groups of processes attempting to coordinate with each other and reach agreement on the value of something
    - The ordering of messages
    - The up/down status of a suspected failed process
    - Who the leader is
    - Who has access to the critical resource
  - All of these are related to the Consensus problem
- What is Consensus
  - Formal problem statement
    - N processes
    - Each process p has
      - input variable xp: initially either 0 or 1
      - output variable yp: initially b (can be changed only once)
    - Consensus problem: design a protocol so that at the end, either:
      - All processes set their output variables to 0 (all-0's)
      - Or all processes set their output variables to 1 (all-1's)
  - Every process contributes a value
  - Goal is to have all processes decide the same (some) value
    - Decision once made can't be changed
  - There might be other constraints
    - Validity = if everyone proposes same value, then that's what's decided
    - Integrity = decided value must have been proposed by some process
    - Non-triviality = there is at least one initial system state that leads to each of the all-0's or all-1's outcomes
- Why is it important
  - Many problems in distributed systems are equivalent to (or harder than) consensus
    - Perfect Failure detection
    - Leader election (select exactly one leader, and every alive process knows about it)
    - Agreement (harder than consensus)

- So consensus is very important problem, and solving it would be really useful
- So, is there a solution to Consensus?
- Two Different Models of Distributed Systems
  - Synchronous System Model and Asynchronous System Model
  - Synchronous Distributed System
    - Each message is received within bounded time
    - Drift of each process' local clock has a known bound
    - Each step in a process takes lb < time < ub
      - e.g. A collection of processors connected by a communication bus, e.g., a Cray supercomputer or a multicore machine
- Asynchronous System Model
  - Asynchronous Distributed System
    - No bounds on process execution
    - The drift rate of a clock is arbitrary
    - No bounds on message transmission delays
    - E.g. The internet is an asynchronous distributed system, so are ad-hoc and sensor networks
  - This a more general (and this challenging) model than the synchronous system model. A protocol for an asynchronous system will also work for a synchronous system (but not vice, versa)
- Possible or Not
  - In the synchronous system model
    - Consensus is solvable
  - In the asynchronous system model
    - Consensus is impossible to solve
    - Whatever protocol/algorithm you suggest, there is always a worst-case possible execution (with failures and message delays) that prevents the system from reaching consensus
    - powerful result (see the FLP proof in the Optional lecture of this series)
    - Subsequently, safe or probabilistic solutions have become quite popular to consensus or related problems

Consensus In Synchronous Systems
- Let's try to solve consensus
  - Uh, what's the system model?
  - Synchronous system: bounds on
    - message delays
    - upper bound on clock drift rates
    - max time for each process step
      - e.g., multiprocessor (common clock across processors)
  - Processes can fail by stopping (crash-stop or crash failures)
- Consensus in Synchronous Systems
  - For a system with at most f processes crashing
    - All processes are synchronized and operate in "rounds" of time
    - the algorithm proceeds in f + 1 rounds (with timeout), using reliable communication to all members
    - Values^r_i: the set of proposed values known to p_i at the beginning of round r
- Consensus in Synchronous System
  - For a system with at most f processes crashing
    - All processes are synchronized and operate in "rounds" of time

    - the algorithm proceeds in f+1 rounds (with timeout), using reliable communication to all members
    - values^r_i: the set of proposed values known to p_i at the beginning of round r
  - Initially Values^n_i = {}; Values^1_i = {v_i}
    - for round = 1 to f+1 do
      - multicast(values^r_i - values^r-1_i) //iterate through processes, send each a message
      - Values^r+1 <- values^r_i
      - for each V_j received
        - values^r+1_i = values^r+1_i union V_j
      - end
    - end
    - d_i = minimum(values^f+1_i)
- Why does the algorithm work?
  - After f+1 rounds, all non-faulty processes would have received the same set of Values. Proof by contradiction
  - Assume that two non-faulty processes, say p_i and p_j, differ in their final set of values(i.e. after f+1 rounds)
  - Assume that p_i possesses a value v that p_j does not possess
    - p_i have received v in the very last round
      - else, p_i would have sent v to p_j in that last round
    - so, in the last round: a third process p_k, must have sent v to p_i, but then crashed before sending v to p_j
    - similarly, a fourth process sending b in the last-but-one -round must have crashed; otherwise, both p_k and p_j should have received v.
    - proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds
    - This means a total of f+1 crashes, while we have assumed at most f crashes can occur => contradiction

Paxos, Simply

- Consensus problem
  - impossible to solve in asynchronous systems
    - Key to the proof: it is impossible to distinguish a failed process from one that is just very very (very) slow. Hence the rest of the alive processes may stay ambivalent (forever) when it comes to deciding
  - But consensus important since it maps to many distributed computing problems
  - Um, can't we just solve consensus?
- Yes we can
  - Paxos Algorithm
    - Most popular “consensus-solving” algorithm
    - Does not solve consensus problem (which would be impossible, because we already proved that)
    - But provides safety and eventual liveness
    - A lot of system use it
  - Paxos invented by?
    - invented by leslie lamport
- Yes we can
  - Paxos provides safety and eventual liveness
    - Safety: Consensus is not violated

        - Eventual Liveness: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.
    - FLP result still applies: Paxos is not guaranteed to reach Consensus (ever, or within any bounded time)
- Political science 101
    - Paxos has rounds; each round has a unique ballot id
    - Rounds are asynchronous
        - Time synchronization not required
        - If you're in round j and hear a message from round j+1, abort everything and move over to round J+1
        - use timeouts; may be pessimistic
    - Each round itself broken into phases (which are also asynchronous)
        - Phase 1: A leader is elected (election)
        - Phase 2: Leader proposes a value, processes ack (Bill)
        - Phase 3: Leader multicasts final value (law)
- Phase 1 - Election
    - Potential leader chooses a unique ballot id, higher than seen anything so far
    - Sends to all processes
    - Processes wait, respond once to highest ballot id
        - If potential leader sees a higher ballot id, it can't be a leader
        - Paxos tolerant to multiple leaders, but we'll only discuss 1 leader case
        - Processes also log received ballot ID on disk
    - If a process has in a previous round decided on a value v', it includes value v' in its response
    - If majority (i.e., quorum) respond OK then you are the leader
        - If no one has majority, start new round
    - (If things go right) A round cannot have two leaders (why?)
- Phase 2 - Proposal (Bill)
    - Leader sends proposed value v to all
        - use v=v' if some process already depicted decided in a previous round and sent you its decided value v'
    - recipient logs on disk; responds OK
- Phase 3 - Decision (Law)
    - If leader hears a majority of OKs, it lets everyone know of the decision
    - Recipients receive decision, log it on disk
- Which is the point of no-return?
    - That is, when consensus reached in the system
    - If/when a majority of processes hear proposed value and accept it(i.e., are about to/have respond(ed) with an OK!)
    - processes may not know it yet, but a decision has been made for the group
        - even leader does not know it yet
    - What if leader fails after that?
        - keep having rounds until some round completes
- Safety
    - If some rounds has majority (i.e. quorum) hearing proposed value v' and accepting it (middle of Phase 2), then subsequently at each round either: 1) the round chooses v' as decision or 2) the round rails
    - Proof:
        - Potential leader waits for majority of OKs in Phase 1

        - At least one will contain v’ (because two majorities or quorums always intersect)
        - It will choose to send out v’ in Phase 2
    - Success requires a majority, and any two majority sets intersect
- What could go wrong
    - Process fails
        - majority does not include it
        - when process restarts, it uses log to retrieve a past decision (if any) and past-seen ballot ids. Tries to know of past decisions
    - Leader fails
        - Start another round
    - Messages dropped
        - If too flaky, just start another round
    - Note that anyone can start a round any time
    - Protocol may never end - tough luck, buddy!
        - Impossibility result not violated
        - If things go well sometime in the future, consensus reached
- What could go wrong?
    - see original paper
- Summary
    - Consensus is a very important problem
        - equivalent to many important distributed computing problems that have to do with reliability
    - Consensus is possible to solve in a synchronous system where message delays and processing delays are bounded
    - Consensus is impossible to solve in an asynchronous system where these delays are unbounded
    - Paxos protocol: widely used implementation of a safe eventually-live consensus protocol for asynchronous systems
        - Paxos (or variants) used in and many other cloud computing systems
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
- adsfdsf
-