Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements: Indy Gupta and Nikita Borisov

Midterm 1 next week Wed-Fri

- Detailed instructions shared on CampusWire (post #98).
  - Go over them again.
  - Reserve a slot if you haven't already.
  - Submit your Letters of Accommodations to CBTF, if required.
  - Syllabus: everything covered in class upto and including Global State.
  - Closed-book exam: cannot refer to any materials.
    - We will provide a cheatsheet over PrairieLearn.
    - Shared in post #98
  - CBTF will provide calculator and scratch paper.
  - *Dummy Quiz has been released on PrairieLearn.*

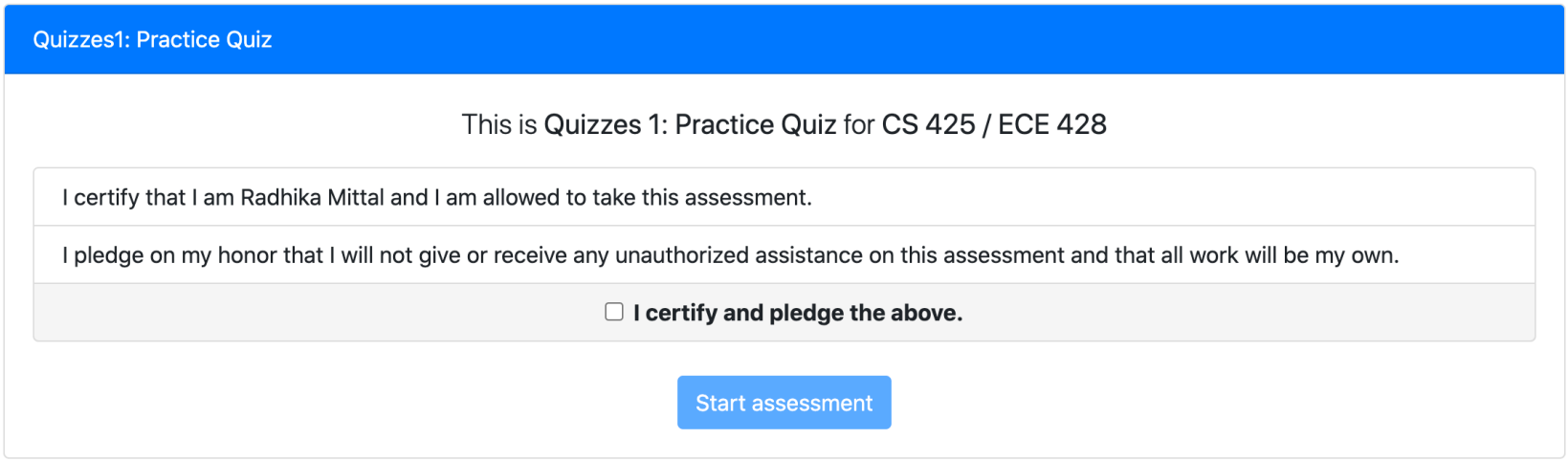
Midterm exam

- Syllabus:
  - everything up to and including Global State.
- Exam duration: 50mins
  - Extra time to check-in and settle in.

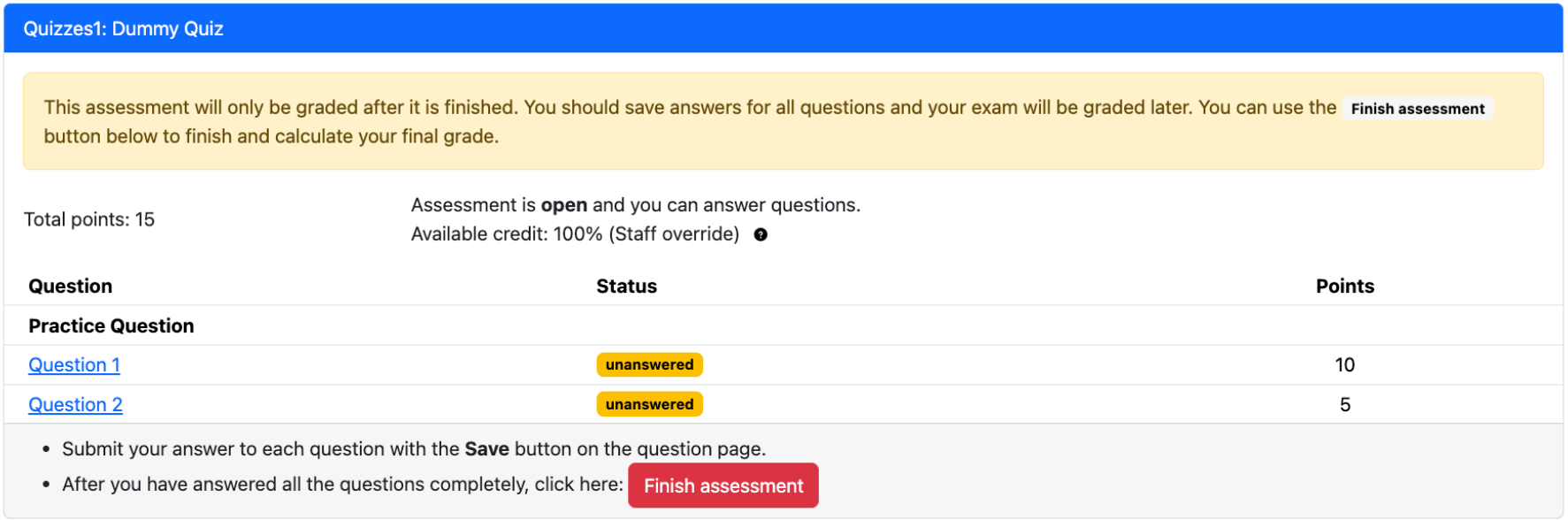
PrairieLearn

- Exam format:
  - Multiple choice questions:
    - Single answer correct; True/False
    - Multiple answers may be correct.
  - Numerical questions
    - No step marking!
  - Ensure all your responses are "saved" and none are "invalid".

PrairieLearn: interface example

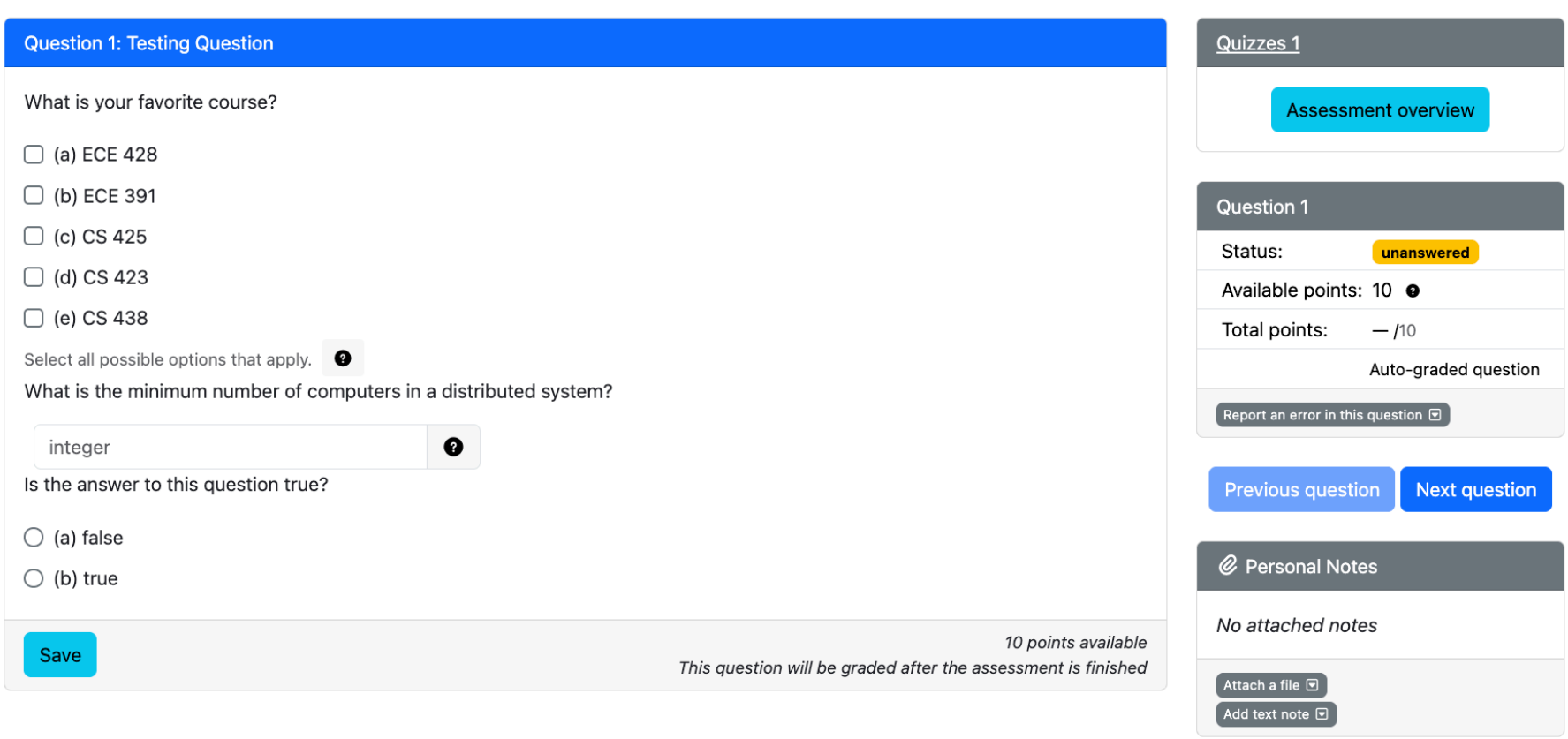


PrairieLearn: interface example

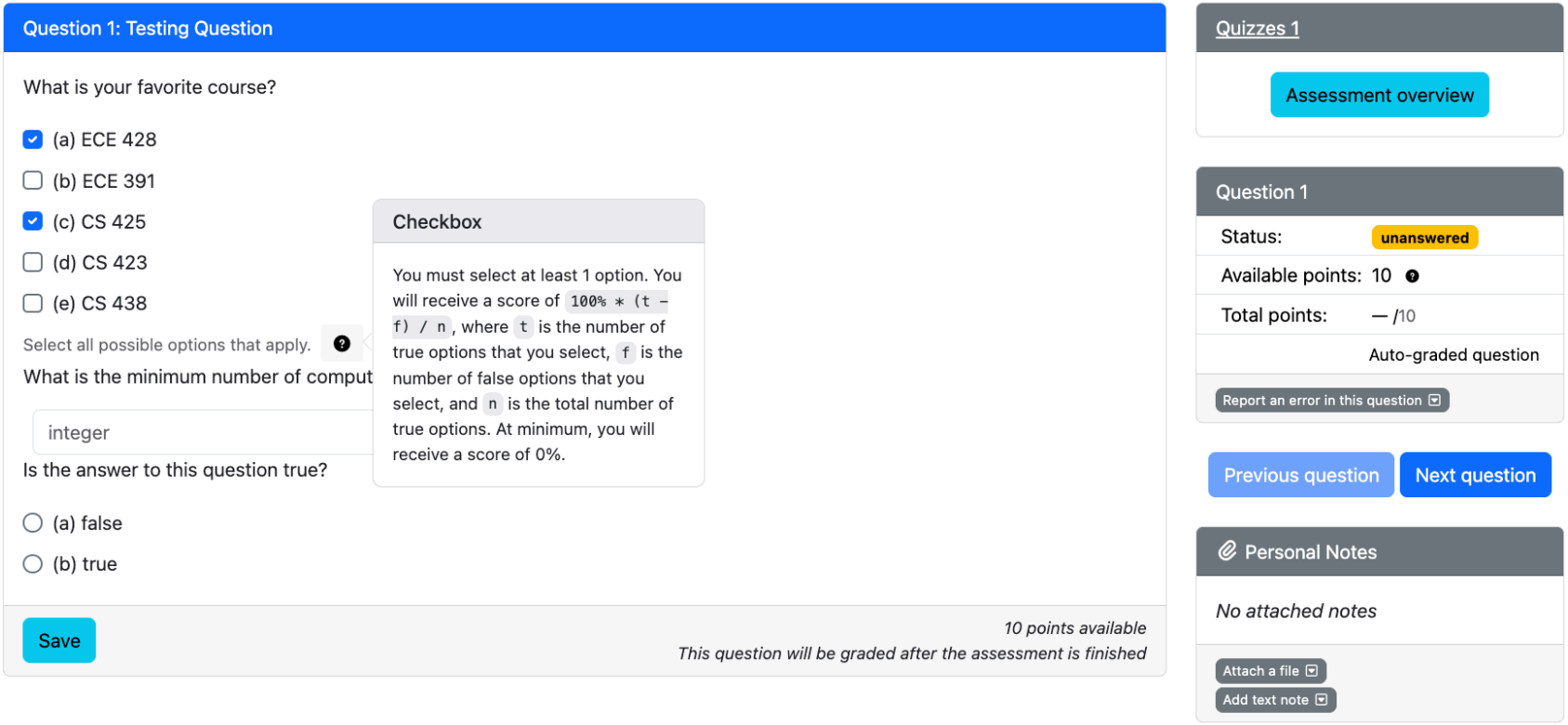


This demo has only two questions. Your midterm will display more questions.

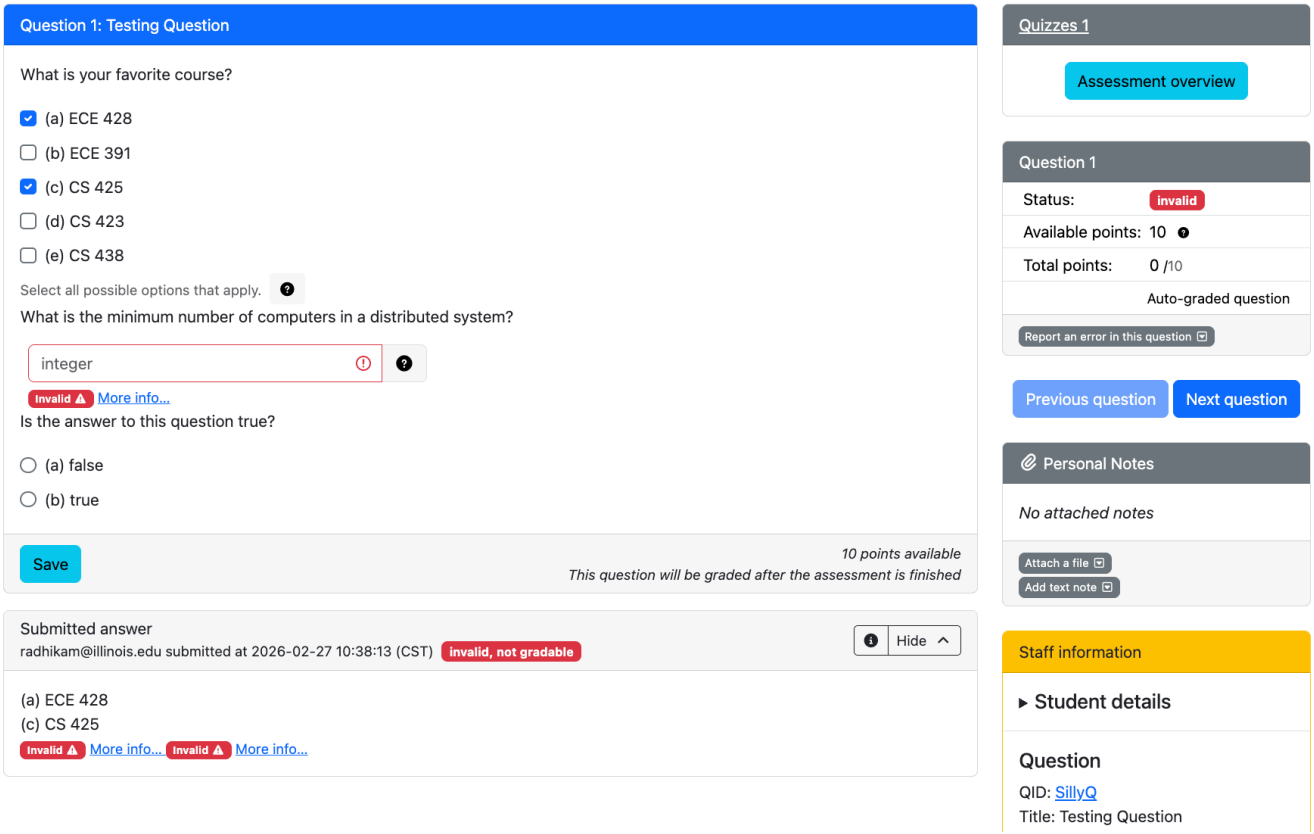
PrairieLearn: interface example



PrairieLearn: interface example

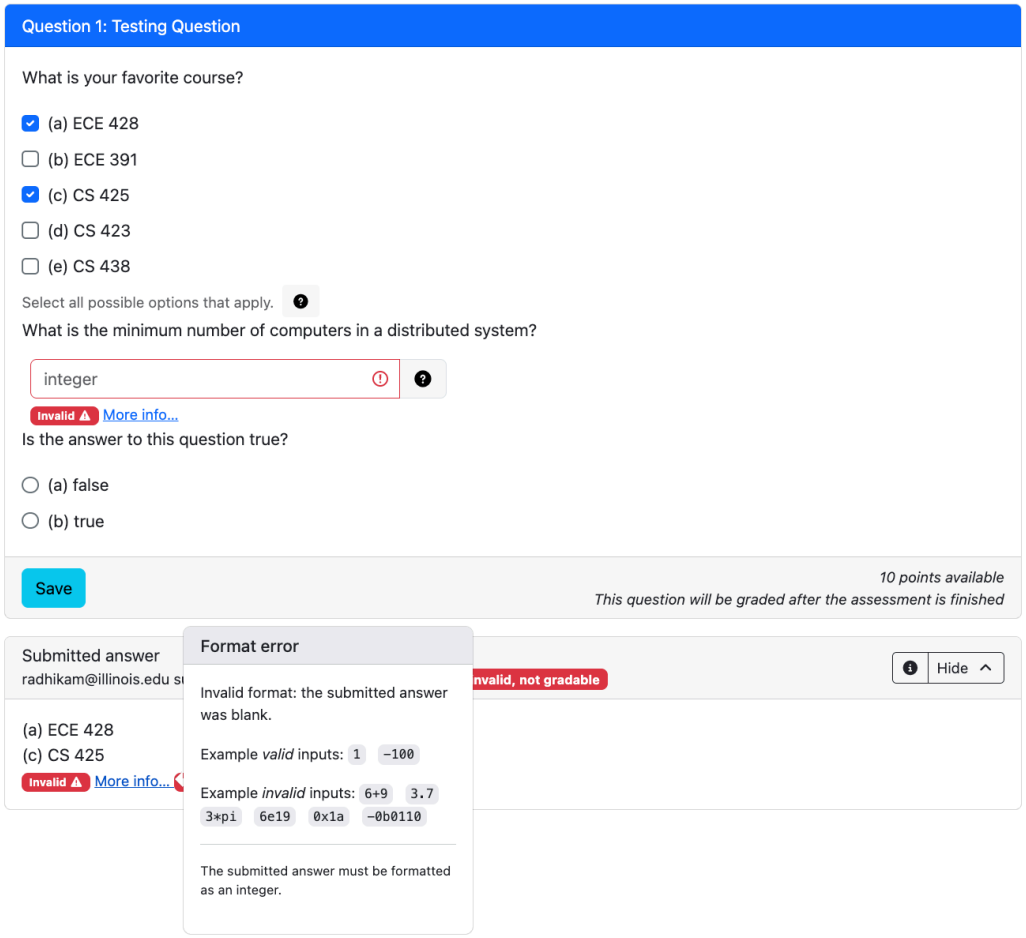


PrairieLearn: interface example



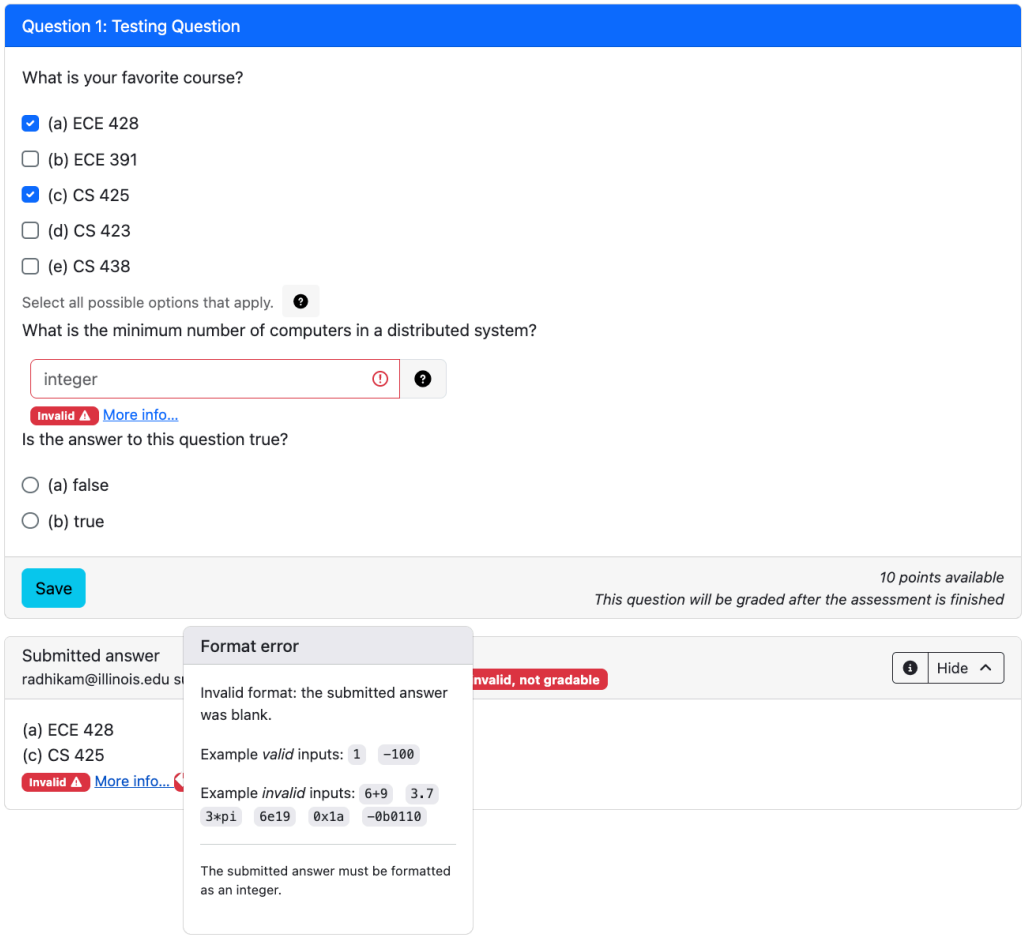
You must attempt all subparts of a given question for the question to be gradable.

PrairieLearn: interface example



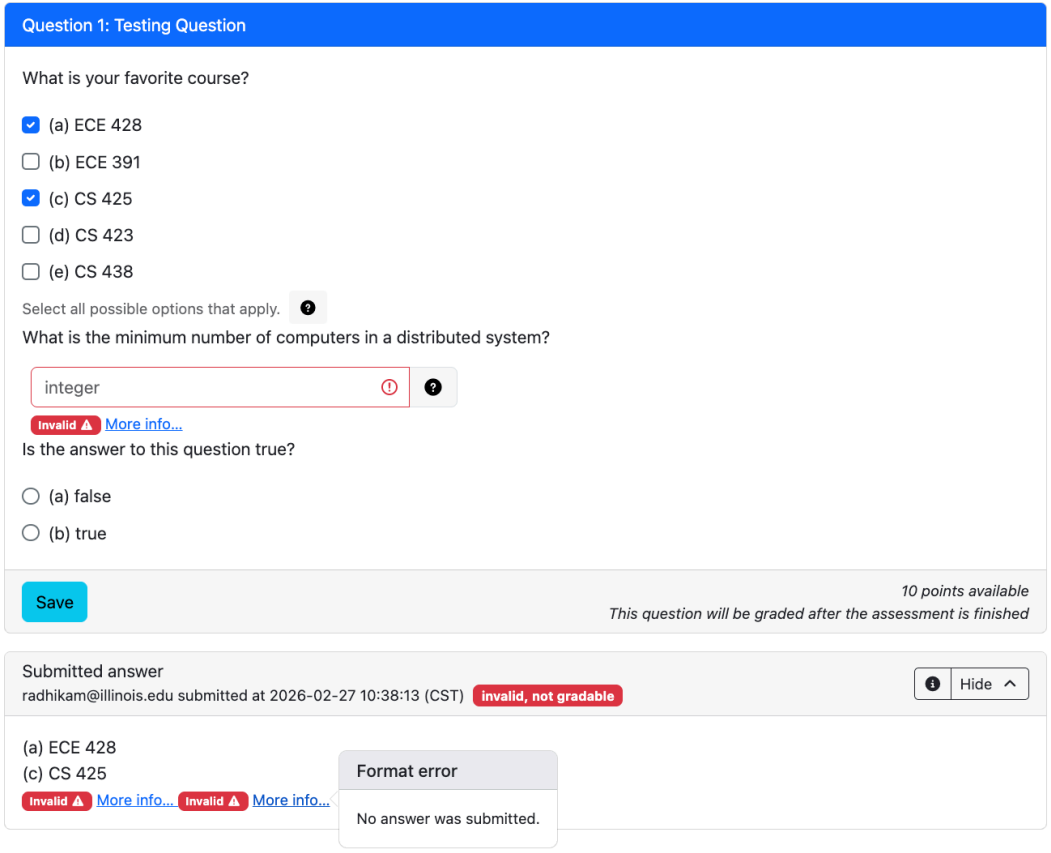
You must attempt all subparts of a given question for the question to be gradable.

PrairieLearn: interface example



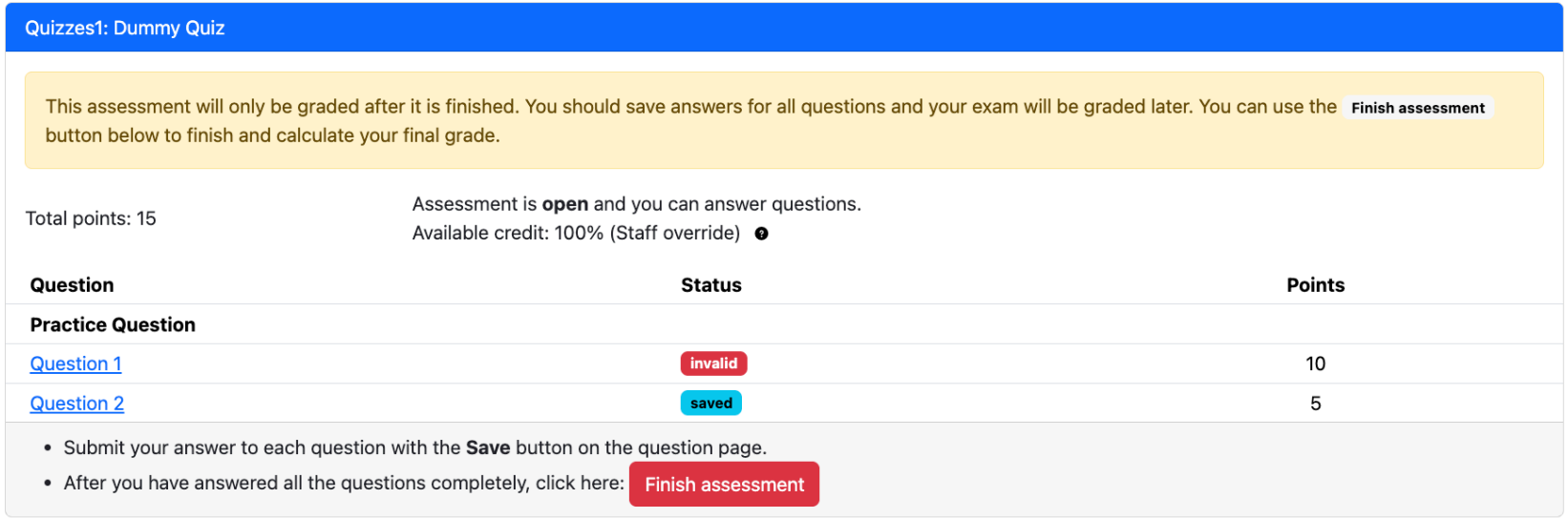
You must attempt all subparts of a given question for the question to be gradable.

PrairieLearn: interface example



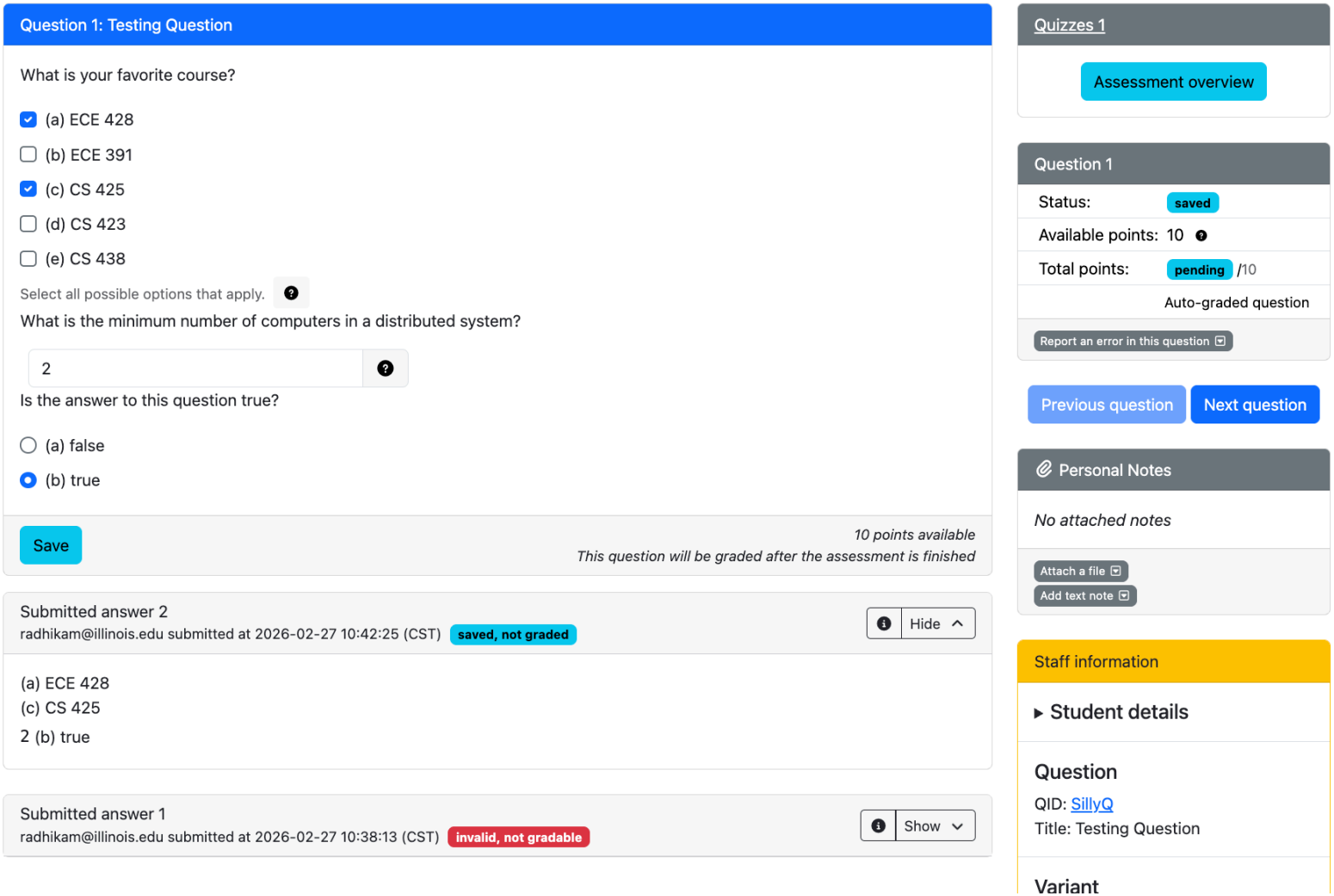
You must attempt all subparts of a given question for the question to be gradable.

PrairieLearn: interface example

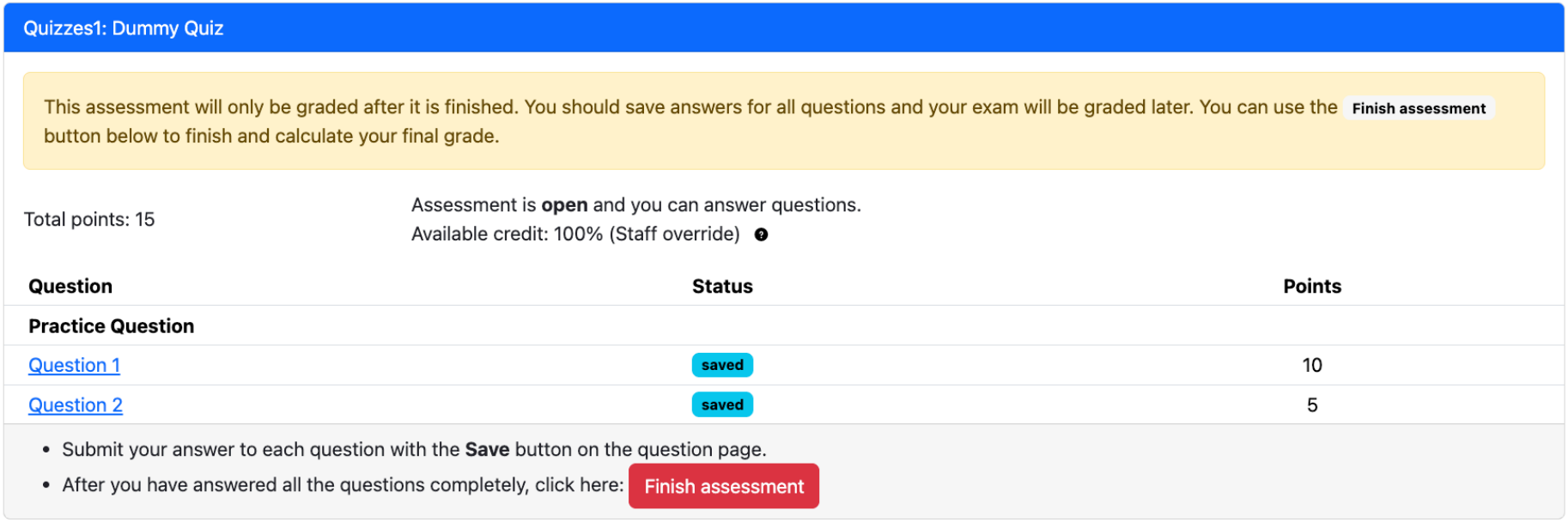


Valid responses saved for other questions will not be affected.

PrairieLearn: interface example



PrairieLearn: interface example



Today's agenda

- **Leader Election (contd)**
  - Chapter 15.3
- **Goal:**
  - What is leader election in distributed systems?
  - How do we elect a leader?
  - To what extent can we handle failures when electing a leader?
- **Exam Review**

Calling for an Election

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes are allowed to call an election simultaneously.
  - All of them together must yield only a single leader
- The result of an election should not depend on which process calls for it.

Election Problem, Formally

- A run of the election algorithm must always guarantee:
  - **Safety**: For all non-faulty processes *p*:
    - *p* has elected:
      - (q: a particular non-faulty process with the *best attribute value*)
      - or Null
  - **Liveness**: For all election runs:
    - election run terminates
    - & for all non-faulty processes *p*: *p*'s elected is not Null
- At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
  - Common attribute : leader has highest id
  - Other attribute examples: leader has fastest cpu, or most disk space, or most number of files, etc.

System Model

- *N* processes.
- Messages are eventually delivered.
- Failures may occur during the election protocol.
- **Each process has a unique id.**
  - Each process has a unique attribute (based on which Leader is elected).
  - If two processes have the same attribute, combine the attribute with the process id to break ties.

Classical Election Algorithms

- Ring election algorithm
- Bully algorithm

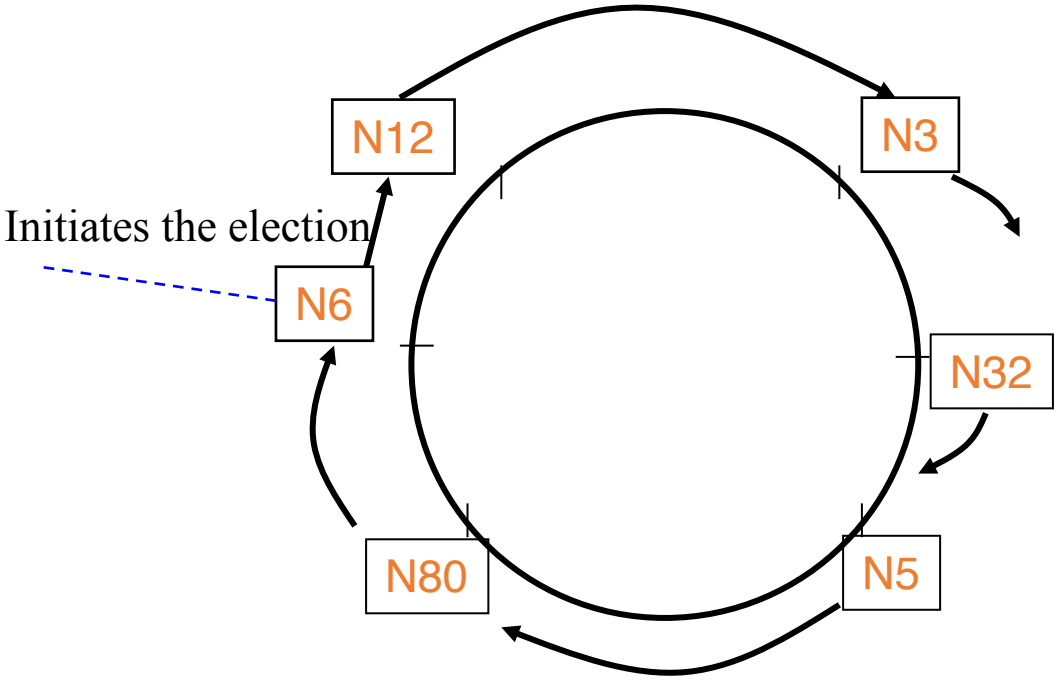
Ring Election Protocol [Chang & Roberts'79]

- When Pi start election
  - send election message with Pi's <$attr_i$, i> to ring successor.
  - set state to participating
- When Pj receives message (election, <$attr_x$, x>) from predecessor
  - If ($attr_x$, x) > ($attr_j$, j):
    - forward message (election, <$attr_x$, x>) to successor
    - set state to participating
  - If ($attr_x$, x) < ($attr_j$, j)
    - If (not participating):
      - send (election, <$attr_j$, j>) to successor
      - set state to participating
  - If ($attr_x$, x) = ($attr_j$, j) : Pj is the elected leader (why?)
    - send elected message containing Pj's id.
- elected message forwarded along the ring until it reaches the leader.
  - Set state to not participating when an elected message is received.

Performance Analysis

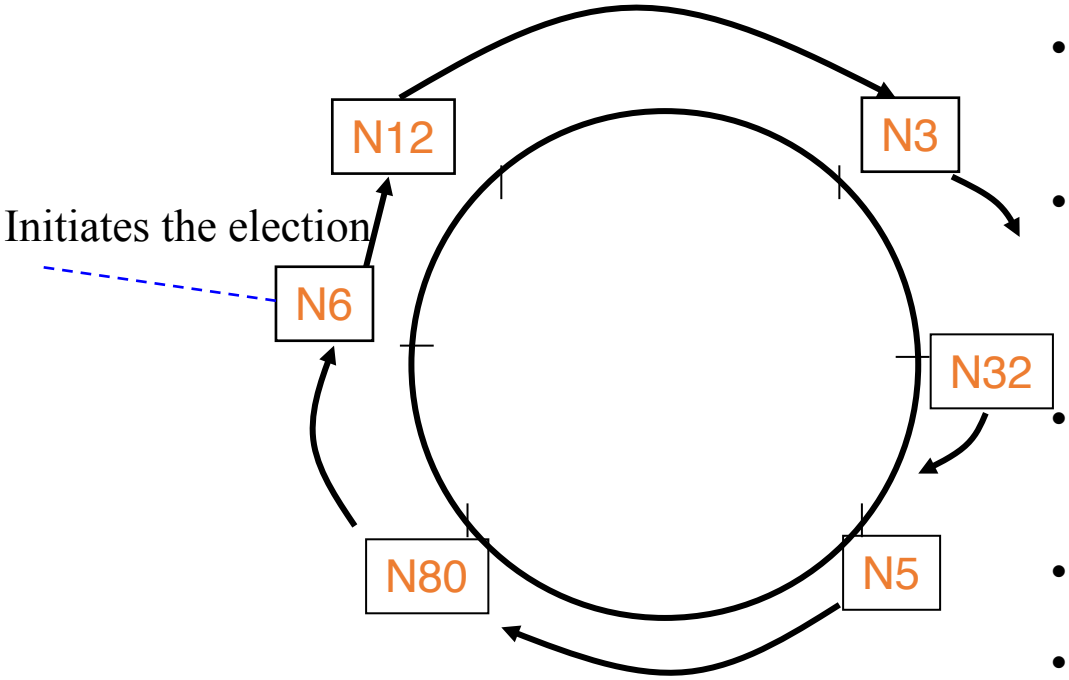
- Let's assume no failures occur during the election protocol itself, and there are *N* processes.
- Let's also assume that only one process initiates the algorithm
- **Bandwidth usage:** Total number of messages sent.
- **Turnaround time:** The number of serialized message transmission times between the initiation and termination of a single run of the algorithm.

Worst-case



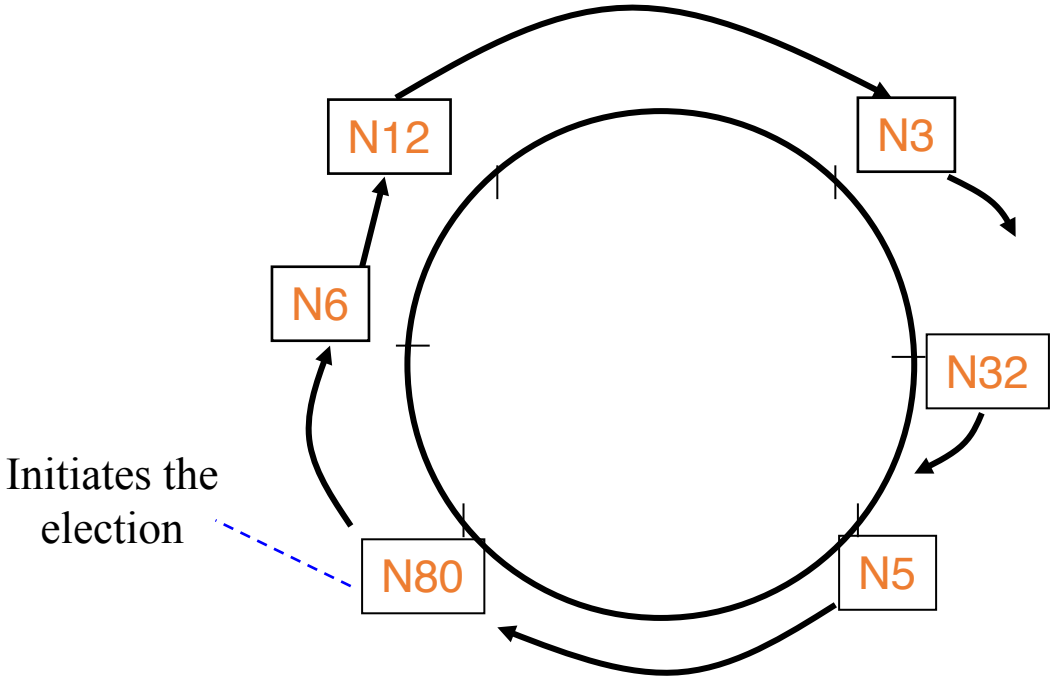
When the initiator is the ring successor of the would-be leader.

Worst-case



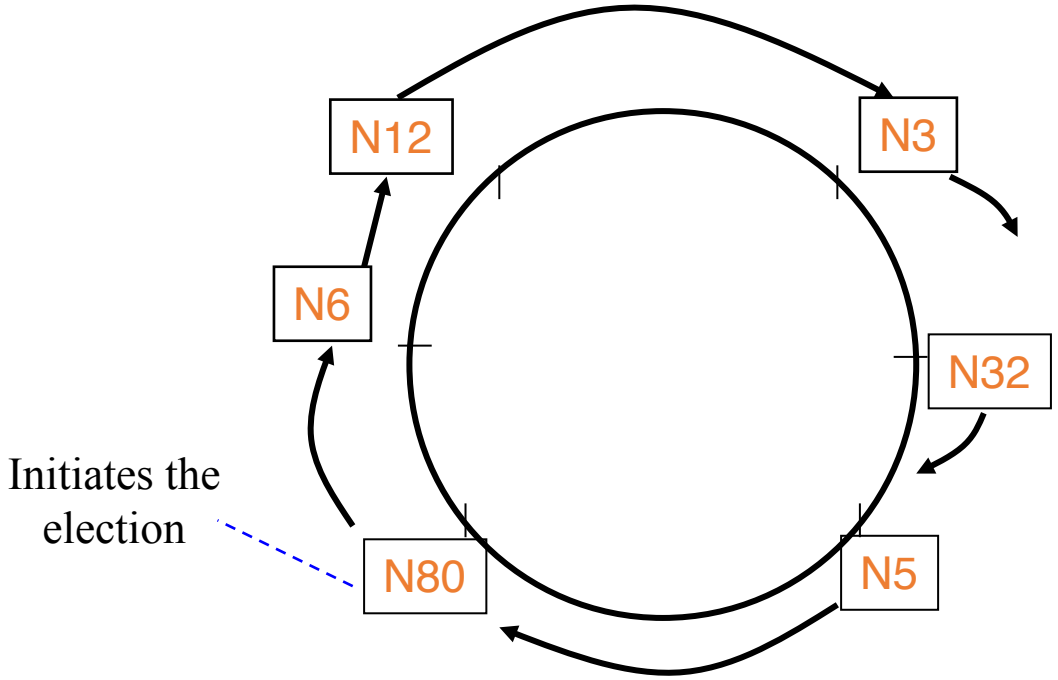
- (N-1) messages for Election message to get from N6 to N80.
- N messages for Election message to circulate around ring without message being changed.
- N messages for Elected message to circulate around the ring
- **No. of messages: (3N-1)**
- **Turnaround time: (3N-1) message transmission times**

Best-case



When the initiator is the would-be leader.

Best-case



When the initiator is the would-be leader.

No. of messages: 2N

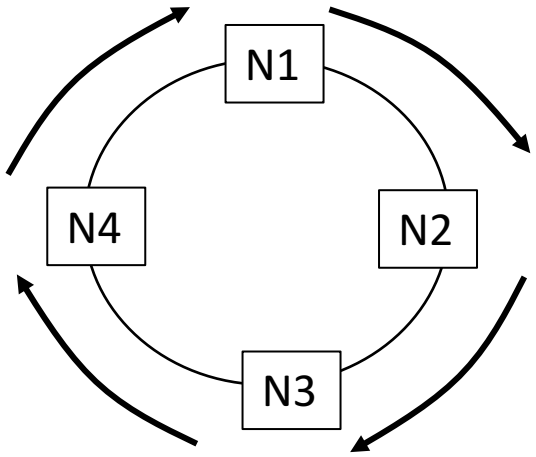
Turnaround time:
2N message transmission times

Performance Analysis

- Let's assume no failures occur during the election protocol itself, and there are *N* processes.
- Let's also assume that only one process initiates the algorithm
- Bandwidth usage (total number of messages)
  - O(N): Worst case = 3N - 1; Best case = 2N.
- O(N) turnaround time.

Performance Analysis

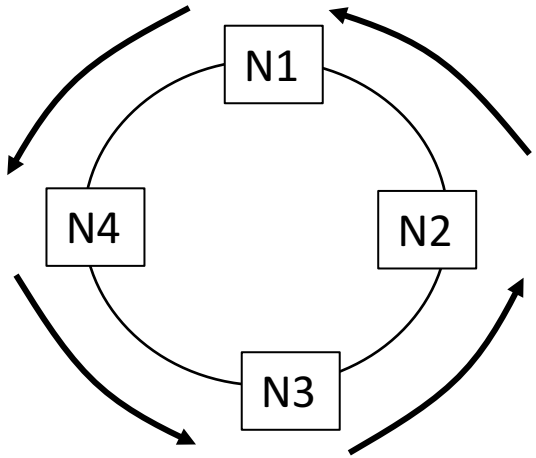
- Let's assume no failures occur during the election protocol itself, and there are *N* processes.
- When each process initiates the algorithm?
  - O(N) messages in best-case.



- N election messages generated at the start of algorithm.
- Only one survives, and completes a full round.
  - N-1 more messages.
- One round for the elected message
  - N messages.
- Total: 3N-1 messages

Performance Analysis

- Let's assume no failures occur during the election protocol itself, and there are *N* processes.
- When each process initiates the algorithm?
  - O(N) messages in best-case.
  - O(N^2) in worst-case.



- N election messages generates at the starts of algorithm.
- N - 1 survive the next time step.
- N – 2 survive the next time step.
- ….

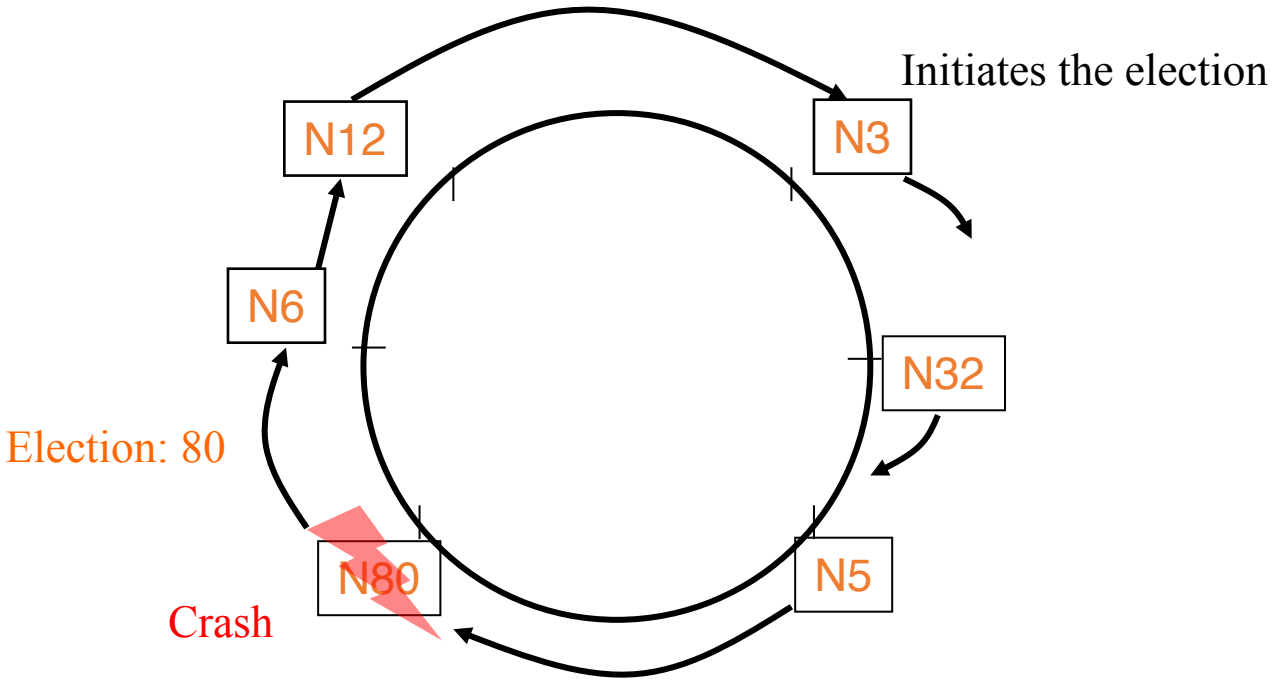
Performance Analysis

- Let's assume no failures occur during the election protocol itself, and there are *N* processes.
- When each process initiates the algorithm?
  - O(N) messages in best-case.
  - $O(N^2)$ messages in worst-case.
  - O(N) turnaround time.

Correctness

- Assuming no process fails.
- Safety:
  - Process with highest attribute elected by all nodes.
- Liveness:
  - Election completes within 3N – 1 message transmission times.

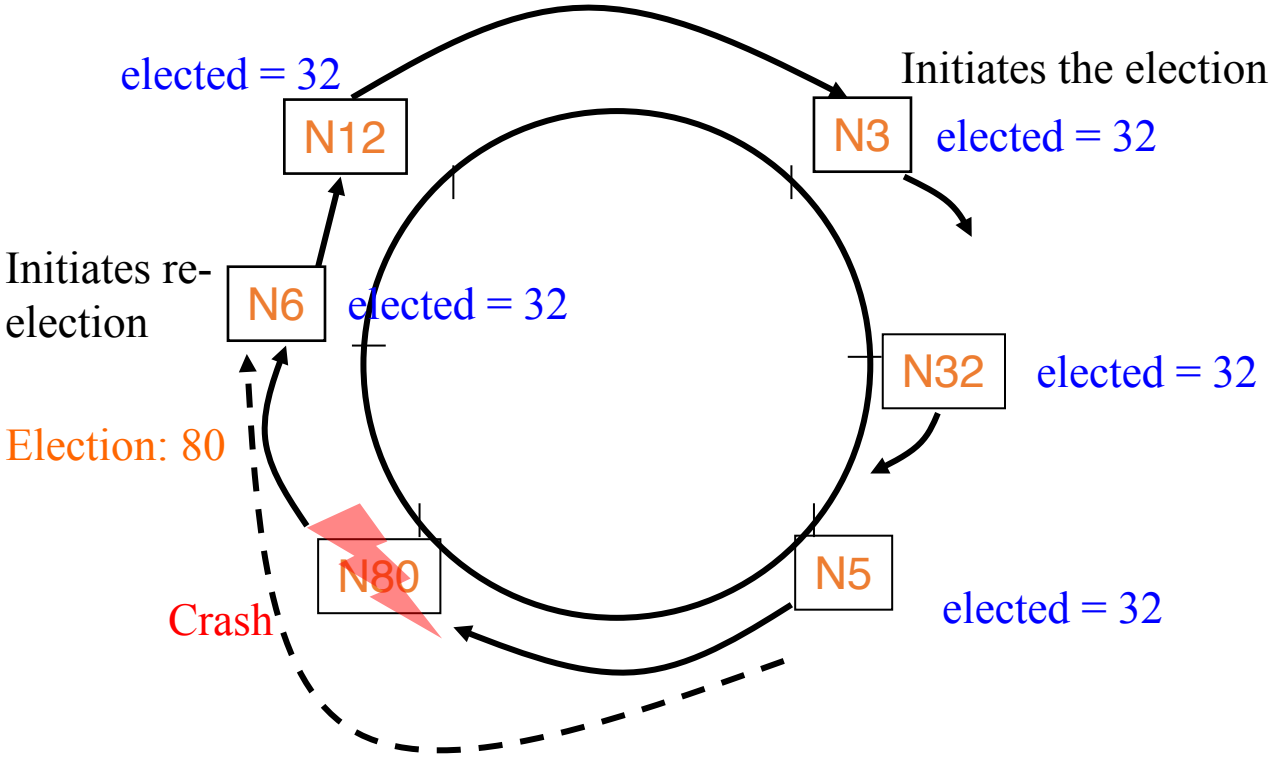
Handling Failures



Handling failures

- Use the failure detector.
- A process can detect failure of N80 via its own local failure detector:
  - Repair the ring.
  - Stop forwarding Election:80 message.
  - Start a new run of leader election.

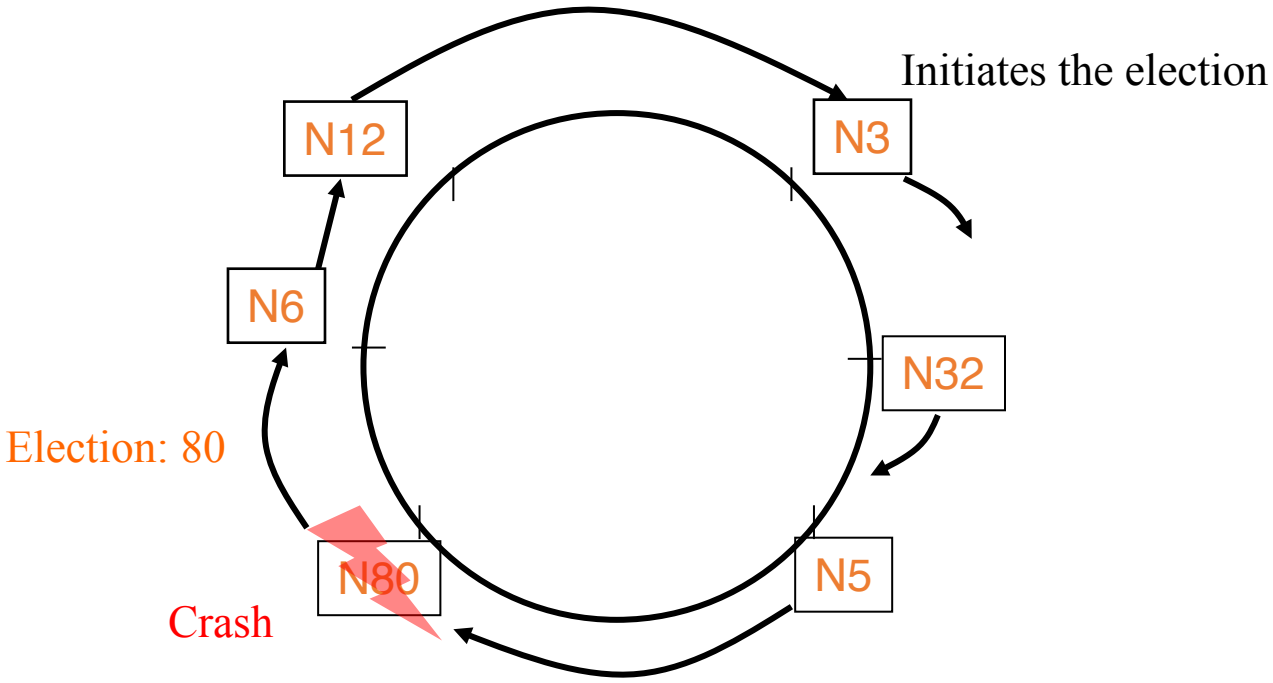
Handling Failures



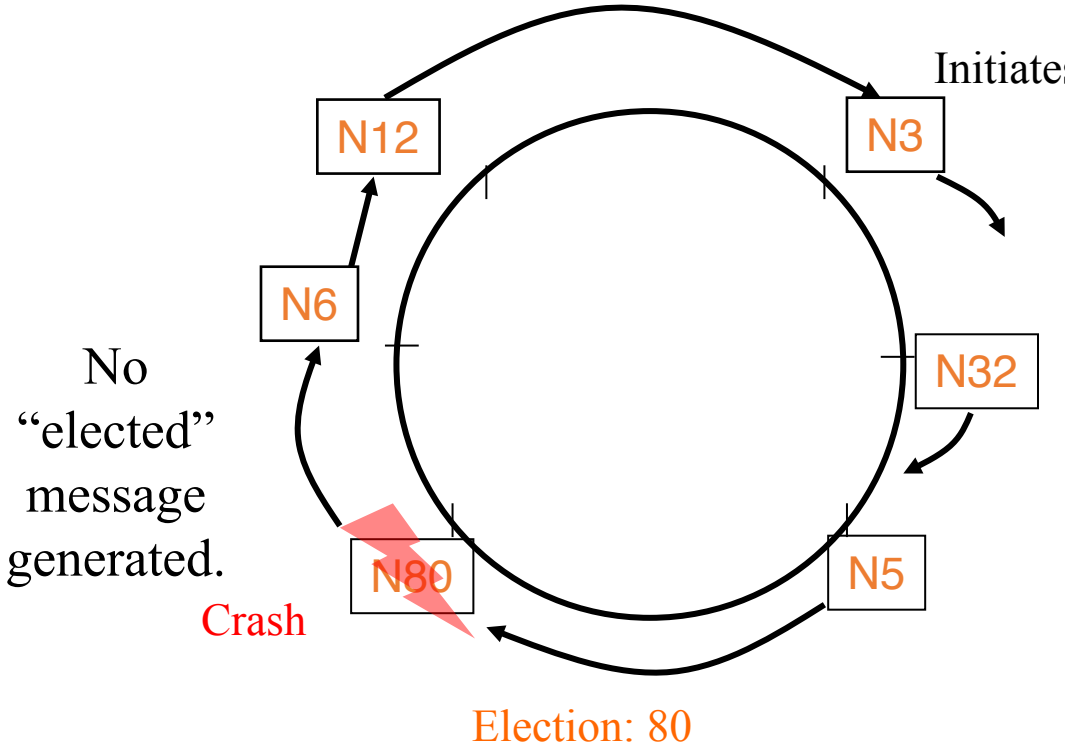
Handling failures

- Use the failure detector.
- A process can detect failure of N80 via its own local failure detector:
  - Repair the ring.
  - Stop forwarding Election:80 message.
  - Start a new run of leader election.
- But failure detectors cannot be both complete and accurate.
  - Incomplete FD => N80's failure might be missed .

What happens if a process failure is undetected?



What happens if a process failure is undetected?



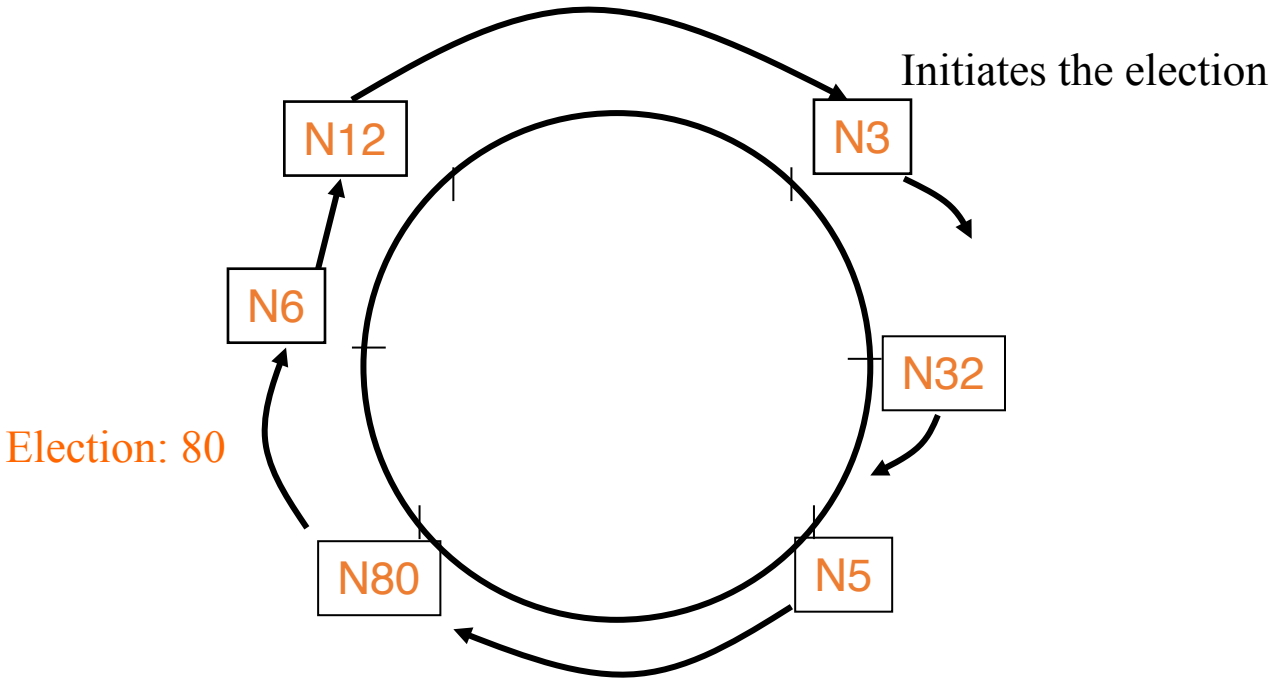
Algorithm does not terminate.

Liveness violated.

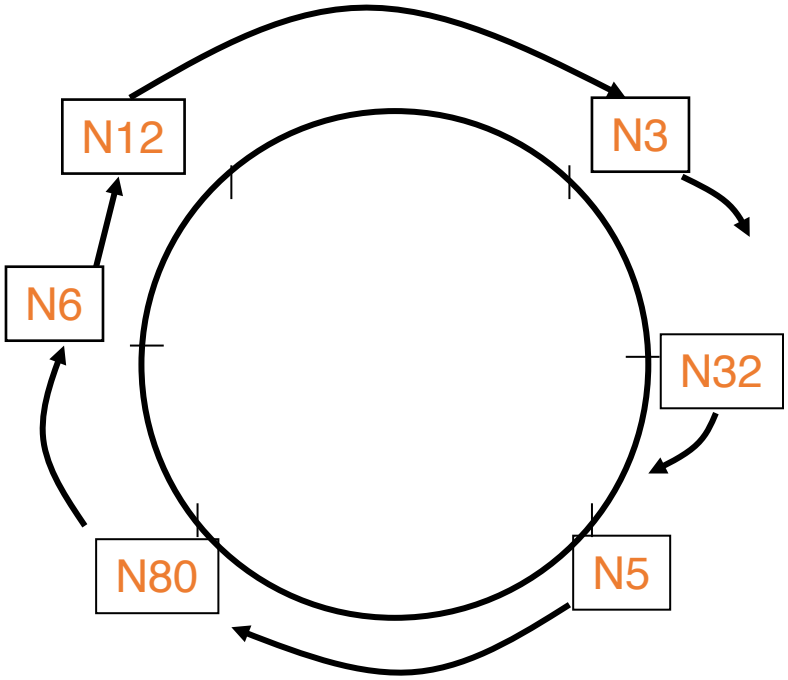
Handling failures

- Use the failure detector.
- A process can detect failure of N80 via its own local failure detector:
  - Repair the ring.
  - Stop forwarding Election:80 message.
  - Start a new run of leader election.
- But failure detectors cannot be both complete and accurate.
  - Incomplete FD => N80's failure might be missed
    - violation of liveness.
  - Inaccurate FD => N80 mistakenly detected as failed

What can happen if an alive process is detected as failed?

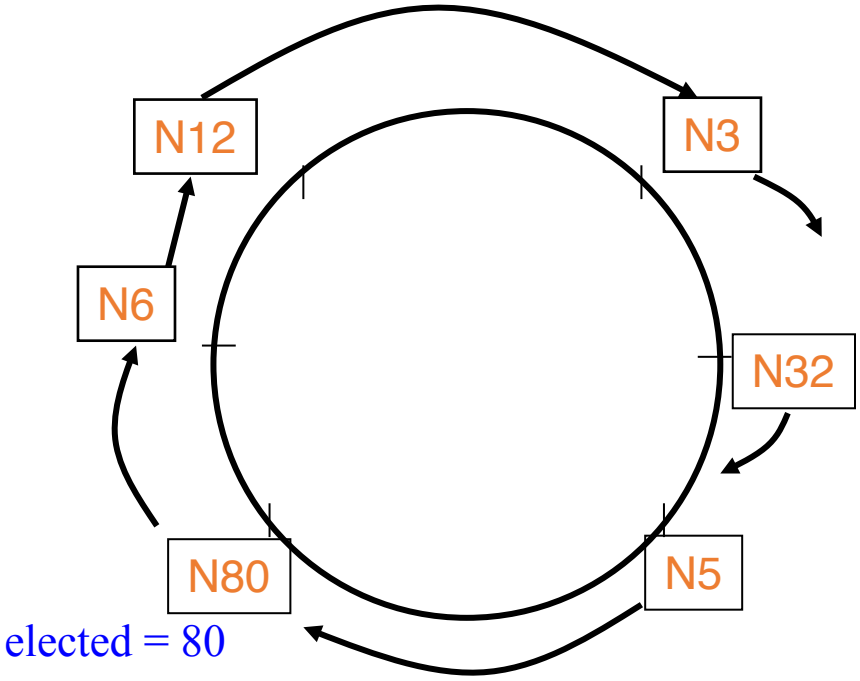


What can happen if an alive process is detected as failed?

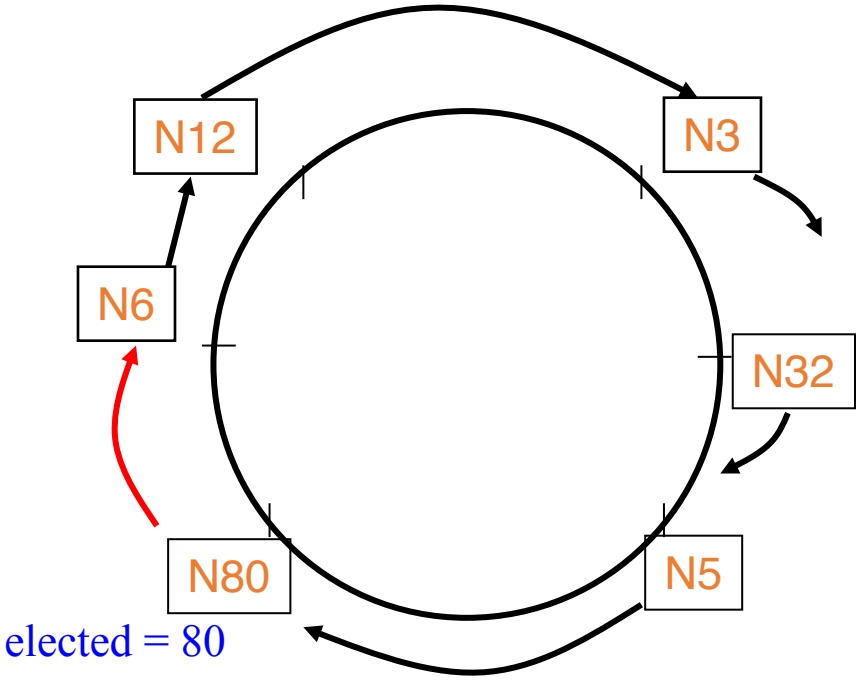


Election: 80

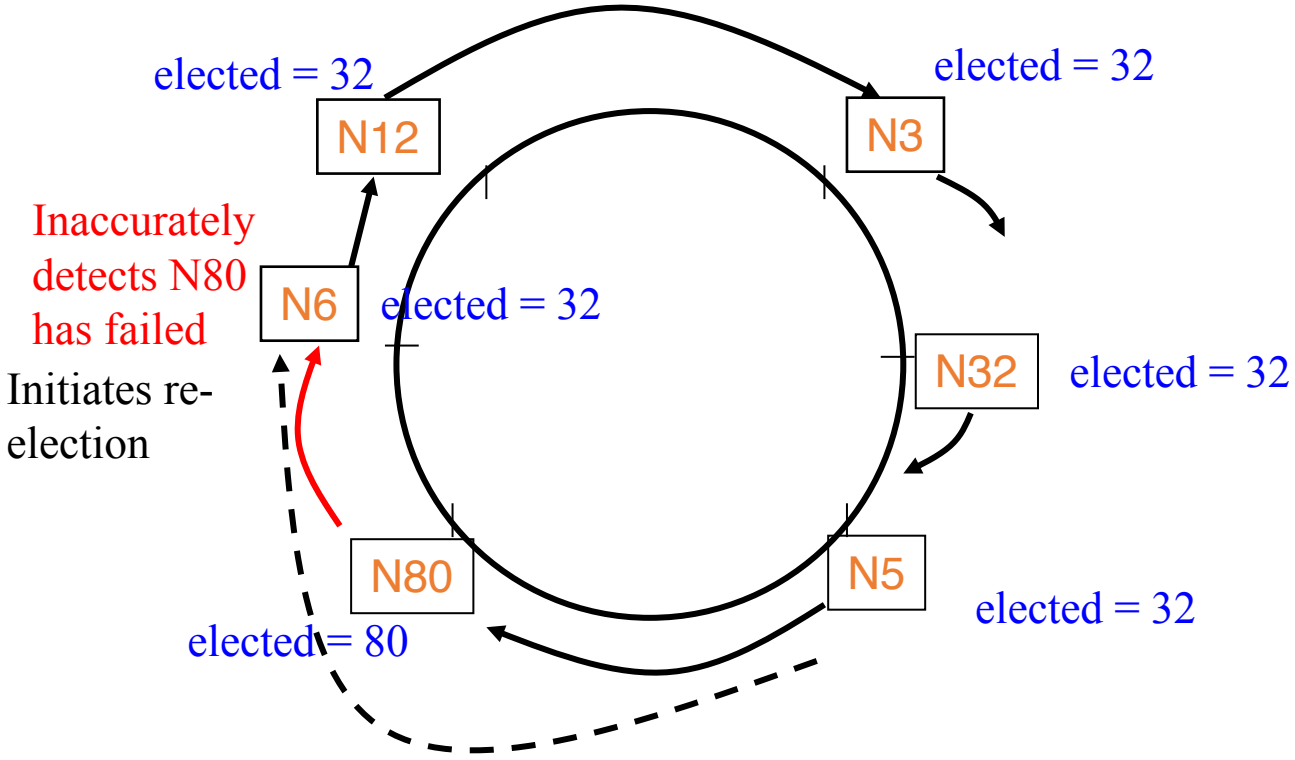
What can happen if an alive process is detected as failed?



What can happen if an alive process is detected as failed?



What can happen if an alive process is detected as failed?



Safety has been violated.

Fixing for failures

- Use the failure detector.
- A process can detect failure of N80 via its own local failure detector:
  - Repair the ring.
  - Stop forwarding Election:80 message.
  - Start a new run of leader election.
- But failure detectors cannot be both complete and accurate.
  - Incomplete FD => N80's failure might be missed
    - violation of liveness.
  - Inaccurate FD => N80 mistakenly detected as failed
    - new ring will be constructed without N80.
    - a process with lower attribute will be selected.
    - different processes end up with different elected leaders.
    - violation of safety.

Classical Election Algorithms

- Ring election algorithm
- **Bully algorithm**

Bully algorithm

- Faster turnaround time than ring election.
- Explicitly build in the notion of timeouts into the algorithm.
- Let's assume (for simplicity of exposition) that the attribute based on which leader is elected is the process id.
- Before discussing Bully algorithm, let's first discuss a simpler (related) algorithm…..

Multicast-based algorithm

- Start an election
  - Multicast <election, my ID> to all processes
  - If receive <agree> from all processes, then elected
    - Multicast <coordinator, my ID>
  - If receive <disagree> from any process
    - Give up election

- Receive <election, *ID*> from process *p*
  - If ID > my ID
    - Send <agree> to *p* (unicast)
  - If ID < my ID
    - Send <disagree> to *p*
    - Start election (if not already running)

- What about failures?

Multicast-based algorithm

- Start an election
  - Multicast <election, my ID> to all processes
  - If receive <agree> from all processes or timeout, then elected
    - Multicast <coordinator, my ID>
  - If receive <disagree> from any process
    - Give up election
- Receive <election, *ID*> from process *p*
  - If ID > my ID
    - Send <agree> to *p* (unicast)
  - If ID < my ID
    - Send <disagree> to *p*
    - Start election (if not already running)
- Can we improve on this?

Multicast-based algorithm

- Start an election
  - Multicast <election, my ID> to all processes
  - If ~~receive <agree> from all processes or~~ timeout, then elected
    - Multicast <coordinator, my ID>
  - If receive <disagree> from any process
    - Give up election
- Receive <election, *ID*> from process *p*
  - ~~If ID > my ID~~
    - ~~Send <agree> to *p* (unicast)~~
  - If ID < my ID
    - Send <disagree> to *p*
    - Start election (if not already running)
- Can we improve on this?

Bully Algorithm

- All processes know other process' ids.
- Do not need to multicast election to all processes.
- Only to processes with higher id.

Bully Algorithm

- When a process wants to initiate an election
  - **if** it knows its id is the highest
    - it elects itself as coordinator, then sends a *Coordinator* message to all processes with lower identifiers. Election is completed.
  - **else**
    - it initiates an election by sending an *Election* message
    - (contd.)

Bully Algorithm (2)

- **else** it initiates an election by sending an *Election* message
  - Sends it to only processes that have a *higher id than itself*.
  - **if** receives no answer within timeout, calls itself leader and sends *Coordinator* message to all lower id processes. Election completed.
  - **if** an answer received however, then there is some non-faulty higher process => so, wait for coordinator message. If none received after another timeout, start a new election run.
- A process that receives an *Election* message replies with *disagree* message, and starts its own leader election protocol (unless it has already done so).

Today's agenda

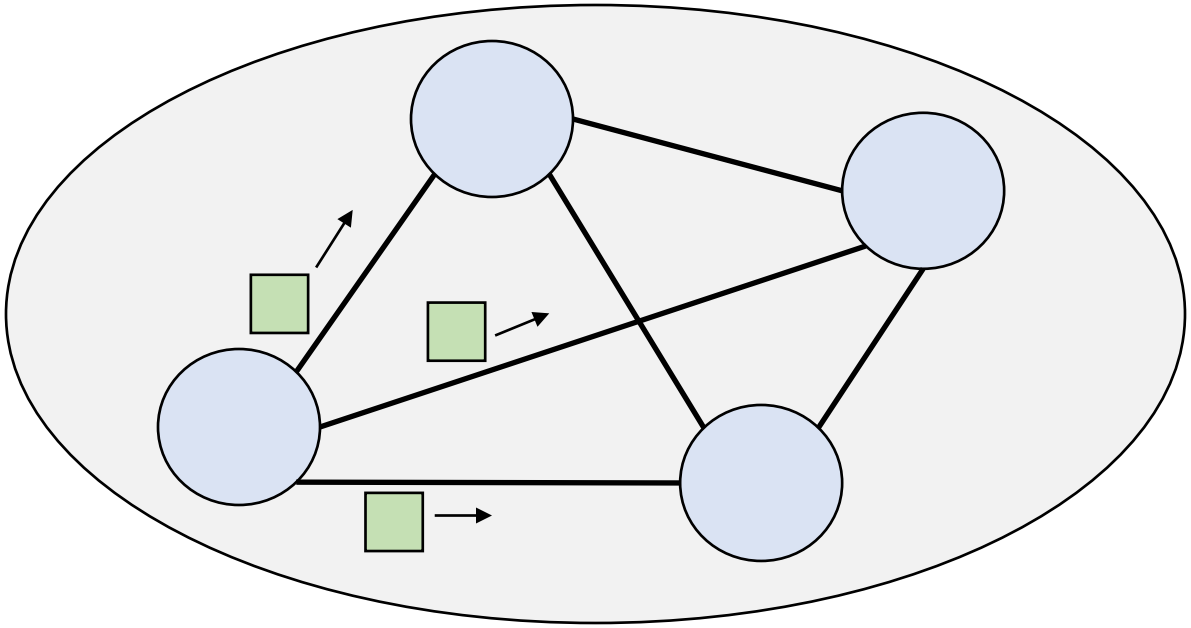
- Leader Election (contd)
  - Chapter 15.3
- Goal:
  - What is leader election in distributed systems?
  - How do we elect a leader?
  - To what extent can we handle failures when electing a leader?
- Exam Review

Disclaimer for our agenda today

- Quick reminder of the relevant topics we covered in class, that are included in your midterm.
- Not meant to be an exhaustive review!
- Go over the slides for each class.
  - Refer to lecture videos and textbook to fill in gaps in understanding.

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State

What is a distributed system?



Independent components or elements that are **connected by a network** and communicate by **passing messages** to achieve a **common goal**, appearing as a single coherent system.

Relationship between processes

- Two main categories:
  - Client-server
  - Peer-to-peer

Two ways to model

- Synchronous distributed systems:
  - Known upper and lower bounds on time taken by each step in a process.
  - Known bounds on message passing delays.
  - Known bounds on clock drift rates.

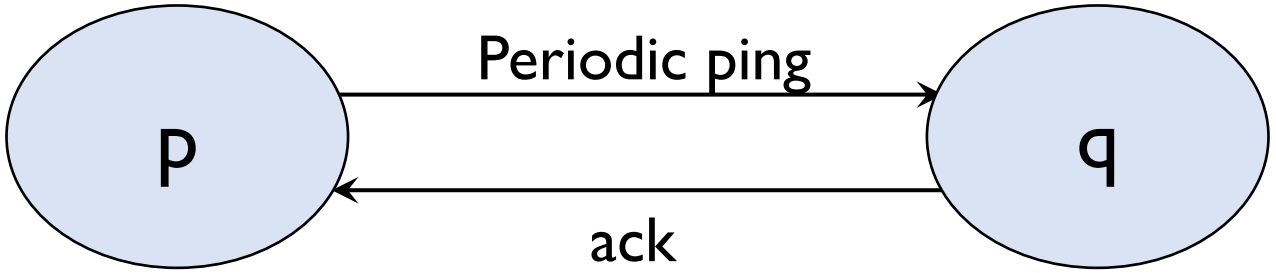
- Asynchronous distributed systems:
  - No bounds on process execution speeds.
  - No bounds on message passing delays.
  - No bounds on clock drift rates.

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State

Types of failure

- **Omission:** when a process or a channel fails to perform actions that it is supposed to do.
  - Process may **crash**.
    - Detected using ping-ack or heartbeat failure detector.
    - Completeness and accuracy in synchronous and asynchronous systems.
    - Worst case failure detection time.
  - **Communication omission**: a message sent by process was not received by another.
    - Message drops (or omissions) can be mitigated by network protocols.

How to detect a crashed process?



p sends pings to q every T seconds.

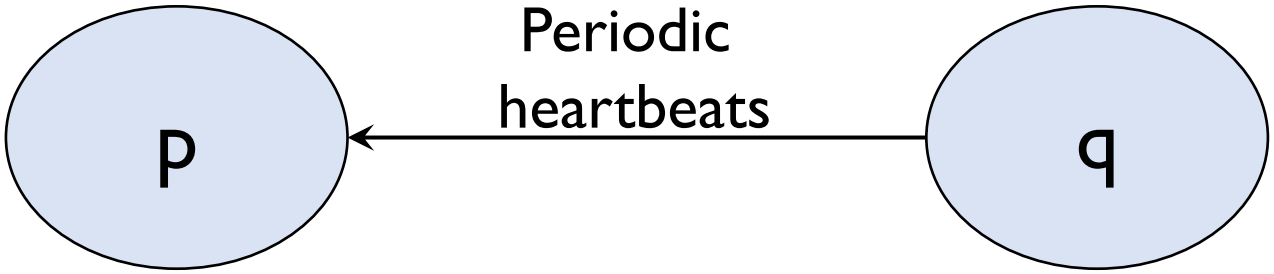
Δ_1 is the *timeout* value at p.

If Δ_1 time elapsed after sending ping, and no ack, report q crashed.

If synchronous, Δ_1 = 2(max network delay)

If asynchronous, Δ_1 = k(max observed round trip time); k >= 1

How to detect a crashed process?



q sends heartbeats to p every T seconds.

($T + \Delta_2$) is the *timeout* value at p.

If ($T + \Delta_2$) time elapsed since last heartbeat, report q crashed.

If synchronous, Δ_2 = max network delay – min network delay

If asynchronous, Δ_2 = k(observed delay); k>=1

Correctness of failure detection

- **Completeness**
  - Every failed process is *eventually* detected.
- **Accuracy**
  - Every detected failure corresponds to a crashed process (no mistakes).

Metrics for failure detection

- Worst case failure detection time
  - Ping-ack: $T + \Delta_1 - \Delta$ (where Δ is time taken for previous ping from p to reach q)
  - Heartbeat: $T + \Delta_2 + \Delta$ (where Δ is time taken for last heartbeat from q to reach p)
- Bandwidth usage:
  - Ping-ack: 2 messages every T units
  - Heartbeat: 1 message every T units.

Types of failure

- **Omission:** when a process or a channel fails to perform actions that it is supposed to do, e.g. process crash and message drops.

- **Arbitrary (Byzantine) Failures:** any type of error, e.g. a process executing incorrectly, sending a wrong message, etc.

- **Timing Failures:** Timing guarantees are not met.
  - Applicable only in synchronous systems.

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State

Clock Skew and Drift Rates

- Each process has an internal clock.
- Clocks between processes on different computers differ:
  - Clock skew: relative difference between two clock values.
  - Clock drift rate: change in skew from a perfect reference clock per unit time (measured by the reference clock).
    - Depends on change in the frequency of oscillation of a crystal in the hardware clock.
- Synchronous systems have bound on maximum drift rate.

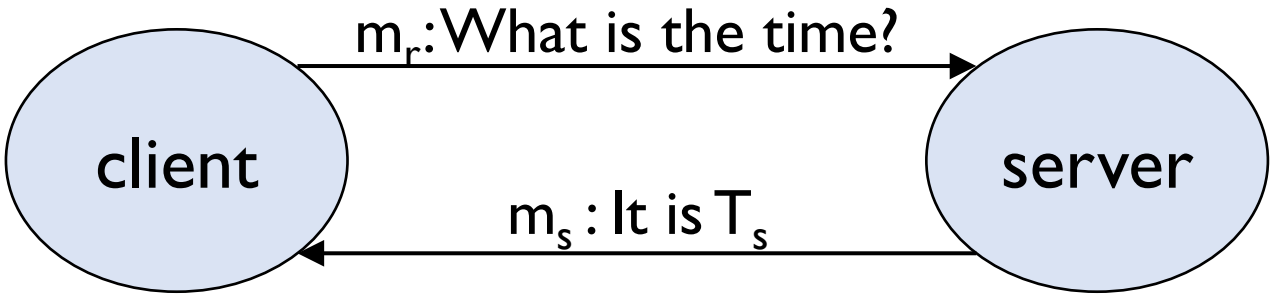
Two forms of synchronization

- External synchronization
  - Synchronize time with an authoritative clock.
  - When accurate timestamps are required.
- Internal synchronization
  - Synchronize time internally between all processes in a distributed system.
  - When internally comparable timestamps are required.
- If all clocks in a system are externally synchronized, they are also internally synchronized.

Synchronization Bound

- Synchronization bound (D) between two clocks A and B over a real time interval I.
  - $|A(t) - B(t)| < D$, for all t in the real time interval I.
    - Skew(A, B) < D during the time interval I.
    - A and B *agree* within a bound *D*.
  - If A is authoritative, D can also be called *accuracy bound*.
    - B is *accurate* within a bound of *D*.
- Synchronization/accuracy bound (D) at time 't'
  - worst-case skew between two clocks at time 't'
    - Skew(A, B) < D at time t

Synchronization in synchronous systems

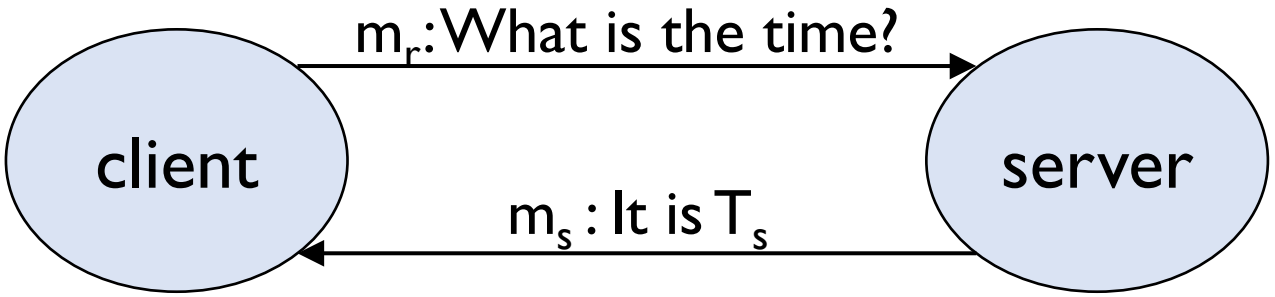


What time T_c should client adjust its local clock to after receiving m_s ?

Let *max* and *min* be maximum and minimum network delay.

If $T_c = (T_s + (min + max)/2)$, skew(client,server) $\leq$ *(max – min)/2*

Cristian Algorithm



What time T_c should client adjust its local clock to after receiving m_s ?

Client measures the round trip time (T_{round}).

$T_c = T_s + (T_{round} / 2)$

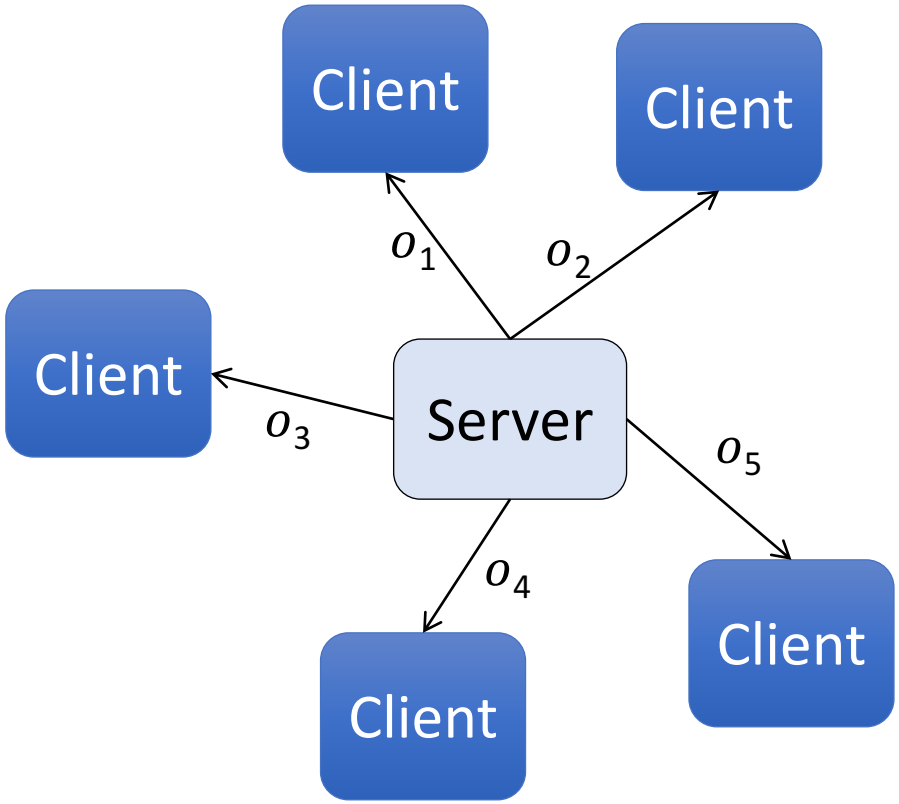
skew $\leq (T_{round} / 2) - min$

$\leq (T_{round} / 2)$

(*min* is minimum one way network delay which is atleast zero).

Berkeley Algorithm

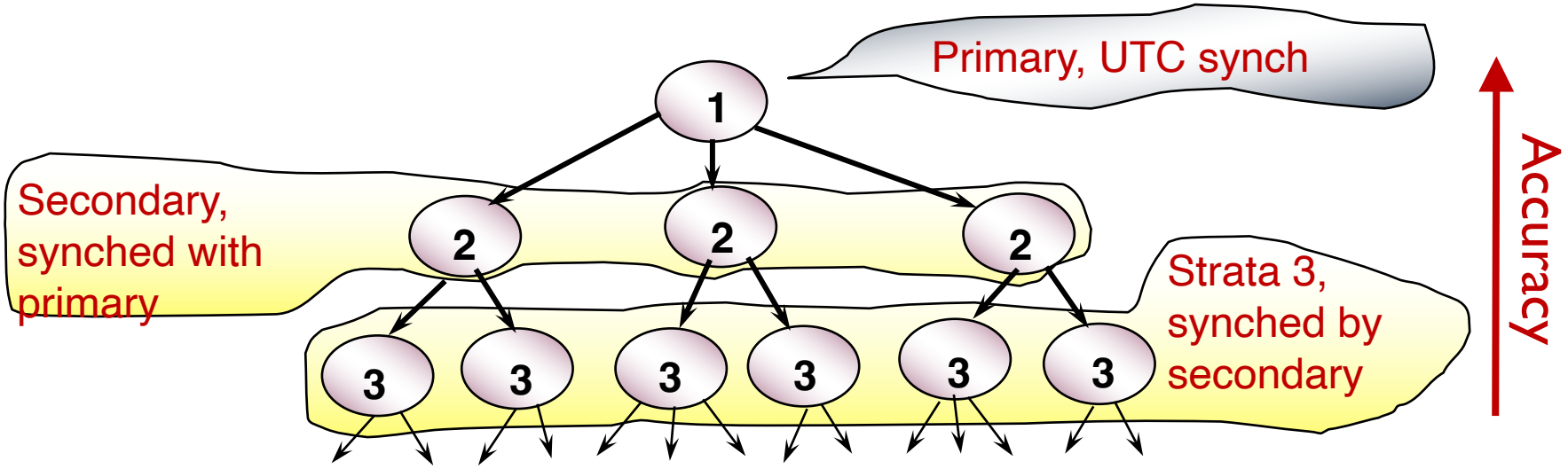
Only supports internal synchronization.



1. Server periodically polls clients: *"what time do you think it is?"*
2. Each client responds with its local time.
3. Server uses Cristian algorithm to estimate local time at each client.
4. Average all local times (including its own) – use as updated time.
5. Send the offset (amount by which each clock needs adjustment).

Network Time Protocol

Time service over the Internet for synchronizing to UTC.



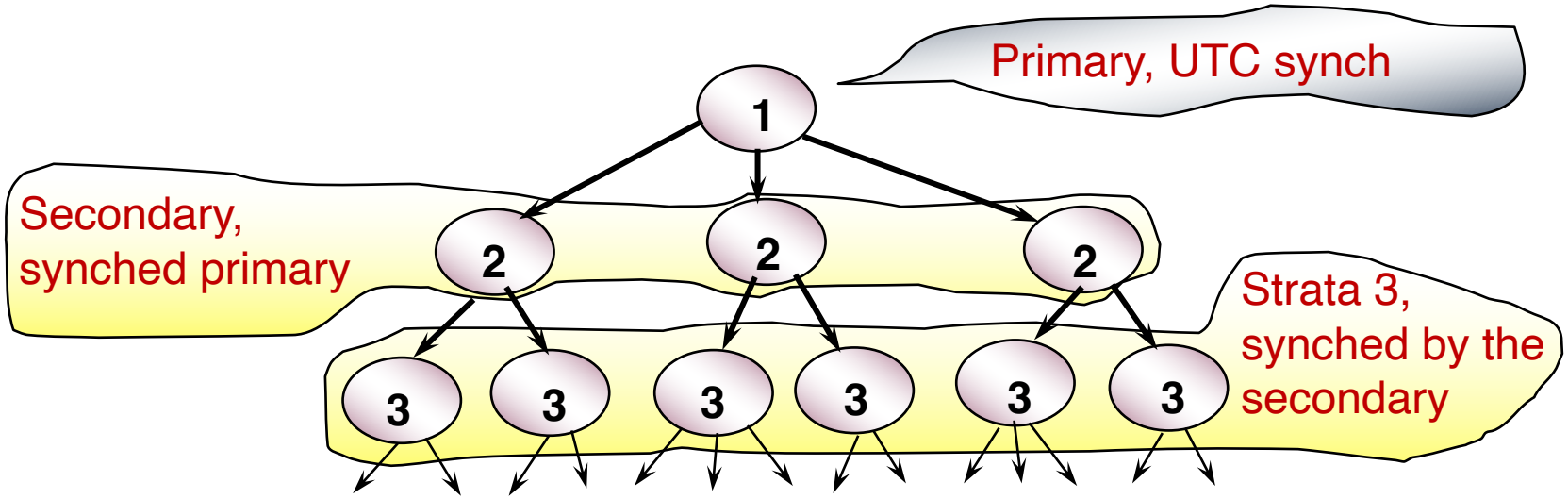
Hierarchical structure for *scalability*.

Multiple lower strata servers for *robustness*.

Authentication mechanisms for *security*.

Statistical techniques for better *accuracy*.

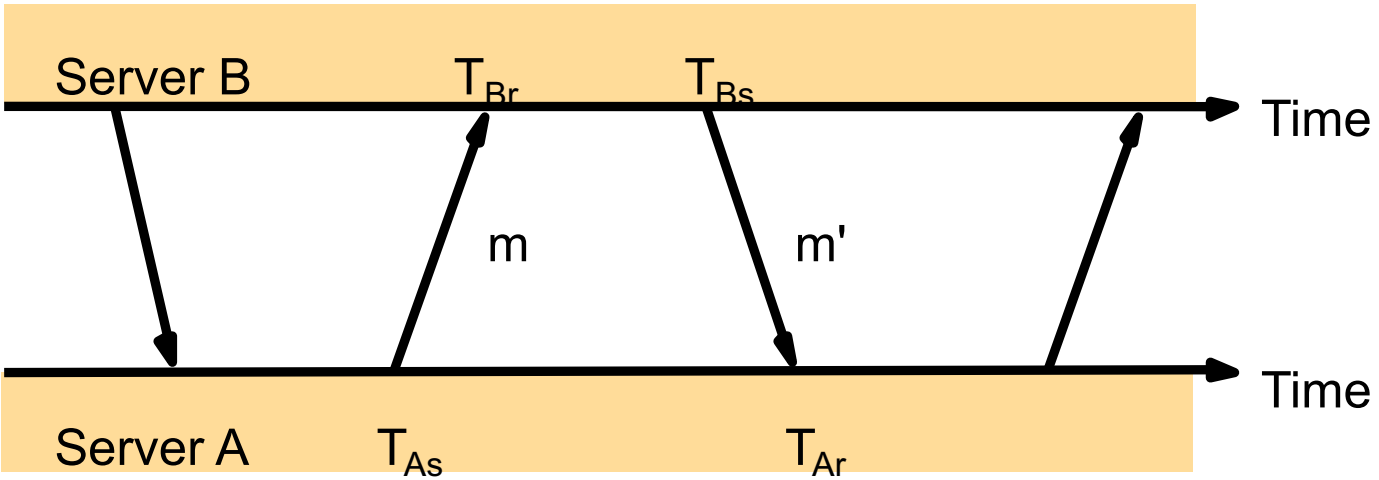
Network Time Protocol



How clocks get synchronized:

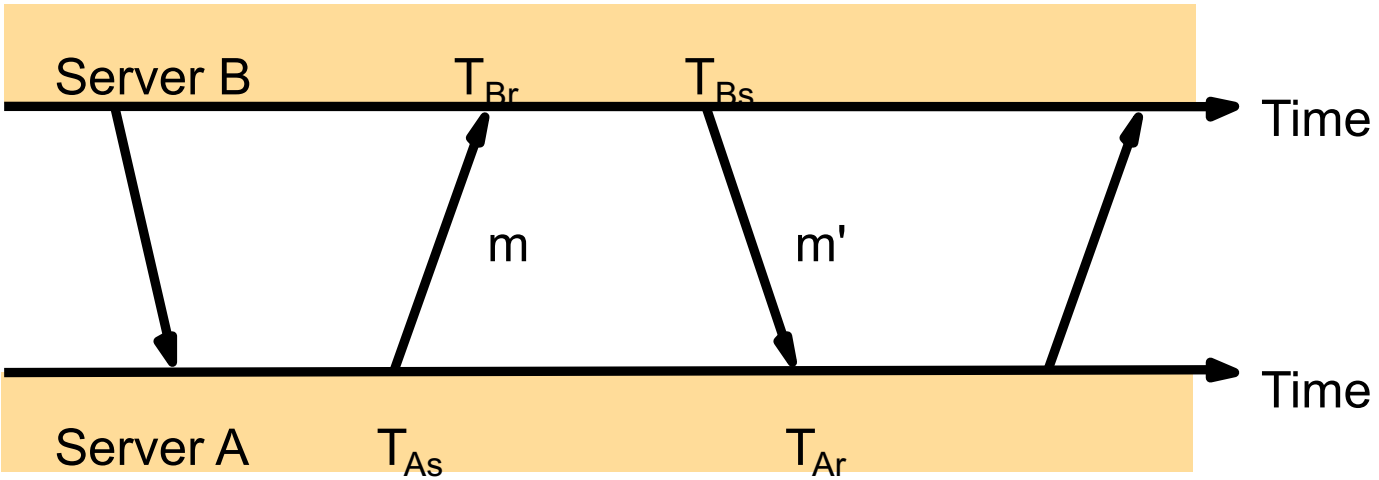
- Servers may *multicast* timestamps within a LAN. Clients adjust time assuming a small delay. *Low accuracy*.
- *Procedure-call* (Cristian algorithm). *Higher accuracy.*
- *Symmetric mode* used to synchronize lower strata servers. *Highest accuracy*.

NTP Symmetric Mode



- A and B exchange messages and record the send and receive timestamps.
  - T_{Br} and T_{Bs} are local timestamps at B.
  - T_{Ar} and T_{As} are local timestamps at A.
  - A and B exchange their local timestamp with eachother.
- Use these timestamps to compute offset with respect to one another.

NTP Symmetric Mode



- t and t': actual transmission times for m and m' (unknown)
- o: true offset of clock at B relative to clock at A (unknown)
- o_i: estimate of actual offset between the two clocks
- d_i: estimate of accuracy of o_i ; $d_i = t + t'$
- $d_i/2$: synchronization bound

$$T_{Br} = T_{As} + t + o$$
$$T_{Ar} = T_{Bs} + t' - o$$
$$o = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}) + (t' - t))/2$$
$$\mathbf{o_i = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}))/2}$$
$$o = o_i + (t' - t)/2$$
$$\mathbf{d_i = t + t' = (T_{Br} - T_{As}) + (T_{Ar} - T_{Bs})}$$

$(o_i - d_i / 2) \leq o \leq (o_i + d_i / 2)$ given $t, t' \geq 0$

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State

Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by →.
  - **e** → **e'** means **e** *happened before* **e'**.
  - **e** $\rightarrow_i$ **e'** means **e** *happened before* **e'**, as observed by $\mathbf{p_i}$.

- HB rules:
  - If ∃ $\mathbf{p_i}$, **e** $\rightarrow_i$ **e'** then **e** → **e'**.
  - For any message m, **send(m)** → **receive(m)**
  - If **e** → **e'** and **e'** → **e"** then **e** → **e"**

- Also called *"causal" or "potentially causal"* ordering.

Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- *Algorithm:* Each process $\mathbf{p_i}$
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ before timestamping the receive event (as per step 2).

Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks V_i when sending a message.
  4. upon receiving a message with vector clock value v
     - sets $V_i[j] = \max(V_i[j], v[j])$ for all $j = 1 \ldots n$.
     - increments $V_i[i]$ before timestamping receive event (as per step 2).

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State

Some more notations and definitions

- For a process p_i , where events $e_i^1, \ldots$ occur:

  history(p_i) = $h_i = \langle e_i^1, \ldots \rangle$

  prefix history(p_i^k) = $h_i^k = \langle e_i^1, \ldots, e_i^k \rangle$

  s_i^k : p_i's state immediately after k^{th} event.

- For a set of processes $\langle p_1, p_2, p_3, \ldots, p_n \rangle$:

  global history: $H = \cup_i (h_i)$

  global state: $S = \cup_i (s_i^{c_i})$

  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$

  the frontier of $C = \{e_i^{c_i}, i = 1,2, \ldots n\}$

  global state S that corresponds to cut $C = \cup_i (s_i^{c_i})$

Consistent cuts and snapshots

- A cut **C** is **consistent** if and only if

$$\forall \mathbf{e} \in \mathbf{C} \text{ (if } \mathbf{f} \rightarrow \mathbf{e} \text{ then } \mathbf{f} \in \mathbf{C})$$

- A global state **S** is consistent if and only if it corresponds to a consistent cut.

Chandy-Lamport Algorithm

- Goal:
  - Record a global snapshot
    - Process state (and channel state) for a set of processes.
  - The recorded global state is consistent.
- Identifies a consistent cut.
- Records corresponding state locally at each process.

Chandy-Lamport Algorithm

- *System model and assumptions:*
  - System of **n** processes: $\langle p_1, p_2, p_3, \ldots, p_n \rangle$.
  - There are two uni-directional communication channels between each ordered process pair : p_j to p_i and p_i to p_j.
  - Communication channels are FIFO-ordered (first in first out).
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.

- *Requirements:*
  - Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages.
  - Any process may initiate algorithm.

Chandy-Lamport Algorithm

- First, initiator $\mathbf{p_i}$:
  - records its own state.
  - creates a special **marker** message.
  - for *j= 1 to n* except *i*
    - $\mathbf{p_i}$ sends a **marker** message on outgoing channel $\mathbf{c_{ij}}$
    - starts recording the incoming messages on each of the incoming channels at $\mathbf{p_i}$: $\mathbf{c_{ji}}$ (for *j= 1 to n* except *i*).

Chandy-Lamport Algorithm

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}

Chandy-Lamport Algorithm

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
  - To record the state of all channels

More notations and definitions

- A **run** is a total ordering of events in H that is consistent with each $\mathbf{h}_i$'s ordering.
- A **linearization** is a run consistent with happens-before ($\rightarrow$) relation in H.

Global State Predicates

- A global-state-predicate is a property that is *true* or *false* for a global state.
  - Is there a deadlock?
  - Has the distributed algorithm terminated?

- Two ways of reasoning about predicates (or system properties) as global state gets transformed by events.
  - Liveness
  - Safety

Liveness

- Liveness = guarantee that something good will happen, eventually

- **Examples:**
  - Guarantee that a distributed computation will terminate.
  - "Completeness" in failure detectors.
  - All processes eventually decide on a value.

- A global state S_0 satisfies a **liveness** property P iff:
  - liveness($P(S_0)$) ≡ $\forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true
  - For any linearization starting from S_0, P is true for some state S_L reachable from S_0.

Safety

- Safety = guarantee that something bad will never happen.

- Examples:
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors.
  - No two processes decide on different values.

- A global state S_0 satisfies a **safety** property P iff:
  - safety($P(S_0)$) ≡ $\forall S$ reachable from S_0, $P(S)$ = true.
  - For all states S reachable from S_0, P(S) is true.

Stable Global Predicates

- once true for a state S, stays true for all states reachable from S (for stable liveness)
- once false for a state S, stays false for all states reachable from S (for stable non-safety)
- Stable liveness examples (once true, always true)
  - Computation has terminated.
- Stable non-safety examples (once false, always false)
  - There is no deadlock.
  - An object is not orphaned.
- *All stable global properties can be detected using the Chandy-Lamport algorithm.*

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State

Good luck!