Today’s plan

• Partially synchronous (timed) distributed systems
• Modeling timed systems
• Proof methods
• Mutual exclusion in timed systems
• Consensus in timed systems
• Clock synchronization
• Reading:
  – Chapters 23, 24, 25
  – [Attiya, Welch], Section 6.3, Chapter 13
We’ve studied distributed algorithms in synchronous and asynchronous distributed models.

Now, intermediate, partially synchronous models.

- Involve some knowledge of time, but not synchronized rounds:
  - Bounds on relative speed of processes,
  - Upper and lower bounds for message delivery,
  - Local clocks, proceeding at approximately-predictable rates.

Useful for studying:
- Distributed algorithms whose behavior depends on time.
- Practical communication protocols.
- (Newer) Mobile networks, embedded systems, robot control,…

Needs new models, new proof methods.

Leads to new distributed algorithms, impossibility results.
Modeling Timed Systems
Modeling timed systems

MMT automata [Merritt, Modugno, Tuttle]
- Simple, special-cased timed model
- Immediate extension of I/O automata

GTA, more general timed automata

Timed I/O Automata
- Still more general
- [Kaynar, Lynch, Segala, Vaandrager] monograph
- Mathematical foundation for Tempo.

Textbook cover image removed due to copyright restrictions.

MMT Automata

• **Definition:** An MMT automaton is an I/O automaton with finitely many tasks, plus a boundmap (lower, upper), where:
  - lower maps each task T to a lower bound lower(T), \(0 \leq \text{lower}(T) < \infty\) (can be 0, cannot be infinite),
  - upper maps each task T to an upper bound upper(T), \(0 < \text{upper}(T) \leq \infty\) (cannot be 0, can be infinite),
  - For every T, \(\text{lower}(T) \leq \text{upper}(T)\).

• **Timed executions:**
  - Like ordinary executions, but with times attached to events.
  - \(\alpha = s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), s_2, \ldots\)
  - Subject to the upper and lower bounds.
    - Task T can’t be continuously enabled for more than time upper(T) without an action of T occurring.
    - If an action of T occurs, then T must have been continuously enabled for time at least lower(T).
  - Restricts the set of executions (unlike having just upper bounds):
    - No fairness anymore, just time bounds.
MMT Automata, cont’d

• **Timed traces:**
  – Suppress states and internal actions.
  – Keep info about external actions and their times of occurrence.

• **Admissible timed executions:**
  – Infinite timed executions with times approaching $\infty$, or
  – Finite timed executions such that $\text{upper}(T) = \infty$ for every task enabled in the final state.

• **Rules out:**
  – Infinitely many actions in finite time (“Zeno behavior”).
  – Stopping when some tasks still have work to do and upper bounds by which they should do it.

• **Simple model, not very general, but good enough to describe some interesting examples:**
Example: Timed FIFO channel

- Consider our usual FIFO channel automaton.
  - **State:** queue
  - **Actions:**
    - Inputs: send(m), m in M
    - Outputs: receive(m), m in M
  - **Tasks:** receive = \{ receive(m) : m in M \}

- **Boundmap:**
  - Associate lower bound 0, upper bound d, with the receive task.

- ** Guarantees delivery of oldest message in channel (head of queue), within time d.**
Composition of MMT automata

• Compose MMT automata by
  – Composing the underlying I/O automata,
  – Combining all the boundmaps.
  – Composed automaton satisfies all timing constraints, of all components.

• Satisfies pasting, projection, as before:
  – Project timed execution (or timed trace) of composition to get timed executions (timed traces) of components.
  – Paste timed executions (or timed traces) that match up at boundaries to obtained timed executions (timed traces) of the composition.

• Also, a hiding operation, which makes some output actions internal.
Example: Timeout system

- **P₁**: Sender process
  - Sends “alive” messages at least every time \( l \), unless it has failed.
  - Express using one send task, bounds \([0,l]\).
- **P₂**: Timeout process
  - Decrements a count from \( k \); if reaches 0 without a message arriving, output timeout.
  - Express with 2 tasks, decrement with bounds \([l₁, l₂]\), and timeout with bounds \([0,l]\).
  - Need non-zero lower bound for decrement, so that steps can be used to measure elapsed time.
- Compose \( P₁, P₂, \) and timed channel with bound \( d \).
- Guarantees (assuming that \( k l₁ > l + d \)):
  - If \( P₂ \) times out \( P₁ \) then \( P₁ \) has in fact failed.
    - Even if \( P₂ \) takes steps as fast as possible, enough time has passed when it does a timeout.
  - If \( P₁ \) fails then \( P₂ \) times out \( P₁ \), and does so by time \( k l₂ + l \).
    - \( P₂ \) could actually take steps slowly.
Example: Two-task race

• One automaton, two tasks:
  – **Main** = \{ increment, decrement, report \}
    • Bounds \([ l_1, l_2 ]\).
  – **Interrupt** = \{ set \}
    • Bounds \([ 0, l ]\).
• Increment **count** as long as flag = false, then **decrement**.
• When **count** returns to 0, output **report**.
• **Set** action sets flag true.
• **Q:** What is a good upper bound on the latest time at which a report may occur?
• \( l + l_2 + ( l_2 / l_1 ) \ l \)
• Obtained by incrementing as fast as possible, then decrementing as slowly as possible.
General Timed Automata

• MMT is simple, but can’t express everything we might want:
  – Example: Perform actions “one”, then “two”, in order, so that “one” occurs at an arbitrary time in \([0,1]\) and “two” occurs at time exactly 1.

• GTAs:
  – More general, expressive.
  – No tasks and bounds.
  – Instead, explicit time-passage actions \(\nu(t)\), in addition to inputs, outputs, internal actions.
  – Time-passage steps \((s, \nu(t), s')\), between ordinary discrete steps.
Example: Timed FIFO Channel

• Delivers oldest message within time d

• States:
  
  queue
  now, a real, initially 0
  last, a real or $\infty$, initially $\infty$

• Transitions:

  send(m)
  Effect:
  add m to queue
  if |queue| = 1 then last := now + d

  receive(m)
  Precondition:
  m = head(queue)
  Effect:
  remove head of queue
  if queue is nonempty then last := now + d else last := $\infty$

  $\nu(t)$
  Precondition:
  now + t $\leq$ last
  Effect:
  now := now + t
Another Timed FIFO Channel

• Delivers every message within time d

• States:
  queue, FIFO queue of (message, real) pairs
  now, a real, initially 0

• Transitions:
  send(m)
    Effect:
      add (m, now + d) to queue
  receive(m)
    Precondition:
      (m,t) = head(queue), for some t
    Effect:
      remove head of queue
  υ(t)
    Precondition:
      now + t ≤ t', for every (m, t') in queue
    Effect:
      now := now + t
Transforming MMTAs to GTAs

• Program the timing constraints explicitly.
• Add state components:
  – *now*, initially 0
  – For each task $T$, add *time-valued variables*:
    • $\text{first}(T)$, initially $\text{lower}(T)$ if $T$ is enabled in initial state, else 0.
    • $\text{last}(T)$, initially $\text{upper}(T)$ if $T$ is enabled in initial state, else $\infty$.
• Manipulate the *first* and *last* values to express the MMT upper and lower bound requirements, e.g.:
  – Don’t perform any task $T$ if $\text{now} < \text{first}(T)$.
  – Don’t let time pass beyond any $\text{last}(\cdot)$ value.
  – When task $T$ becomes enabled, set $\text{first}(T)$ to $\text{lower}(T)$ and $\text{last}(T)$ to $\text{upper}(T)$.
  – When task $T$ performs a step and is again enabled, set $\text{first}(T)$ to $\text{lower}(T)$ and $\text{last}(T)$ to $\text{upper}(T)$.
  – When task $T$ becomes disabled, set $\text{first}(T)$ to 0 and $\text{last}(T)$ to $\infty$. 
Two-task race

• New state components:
  now, initially 0
  first(Main), initially \( l_1 \)
  last(Main), initially \( l_2 \)
  last(Interrupt), initially 1

• Transitions:
  increment:
  Precondition:
    flag = false
    now \geq first(Main)
  Effect:
    count := count + 1
    first(Main) := now + l_1
    last(Main) := now + l_2

decrement:
  Precondition:
    flag = true
    count > 0
    now \geq first(Main)
  Effect:
    count := count - 1
    first(Main) := now + l_1
    last(Main) := now + l_2

report:
  • Precondition:
    flag = true
    count = 0
    reported = false
    now \geq first(Main)
  • Effect:
    reported := true
    first(Main) := 0
    last(Main) := \infty
Two-task race

set:

Precondition:
flag = false

Effect:
flag := true
last(Interrupt) := ∞

υ(t):

Precondition:
now + t ≤ last(Main)
now + t ≤ last(Interrupt)

Effect:
now := now + t
More on GTAs

• Composition operation
  – Identify external actions, as usual.
  – Synchronize time-passage steps globally.
  – Pasting and projection theorems.

• Hiding operation

• Levels of abstraction, simulation relations
Timed I/O Automata (TIOAs)

- Extension of GTAs in which time-passage steps are replaced by trajectories, which describe state evolution over time intervals.
  - Formally, mappings from time intervals to states.
  - Allows description of interesting state evolution, such as:
    - Clocks that evolve at approximately-known rates.
    - Motion of vehicles, aircraft, robots, in controlled systems.
- Composition, hiding, abstraction.
Proof methods for GTAs and TIOAs.

- Like those for untimed automata.
- Compositional methods.
- Invariants, simulation relations.
  - They work for timed systems too.
  - Now they generally involve time-valued state components as well as "ordinary" state components.
  - Still provable using induction, on number of discrete steps + trajectories.
Example: Two-task race

- **Invariant 1:** \( \text{count} \leq \left\lfloor \frac{\text{now}}{l_1} \right\rfloor \).
  - \( \text{count} \) can’t increase too much in limited time.
  - Largest \( \text{count} \) results if each \( \text{increment} \) takes smallest time, \( l_1 \).

- Prove by induction on number of discrete + time-passage steps? Not quite:
  - Property is not preserved by \( \text{increment} \) steps, which increase \( \text{count} \) but leave \( \text{now} \) unchanged.

- So we need another (stronger) invariant.

- **Q:** What else changes in an \( \text{increment} \) step?
  - Before the step, \( \text{first(Main)} \leq \text{now} \); afterwards, \( \text{first(Main)} = \text{now} + l_1 \).
  - So \( \text{first(Main)} \) should appear in the stronger invariant.

- **Invariant 2:** If not reported then \( \text{count} \leq \left\lfloor \frac{\text{first(Main)}}{l_1 - 1} \right\rfloor \).
- Use Invariant 2 to prove Invariant 1.
Two-task race

• **Invariant 2:** If not reported then 
  \[ \text{count} \leq \left\lfloor \frac{\text{first(Main)}}{l_1} - 1 \right\rfloor \]

• **Proof:**
  – By induction.
  – **Base:** Initially, \( \text{LHS} = \text{RHS} = 0 \).
  – **Inductive step:** Dangerous steps either increase LHS (increment) or decrease RHS (report).
    • **Time-passage steps:** Don’t change anything.
    • **report:** Can’t cause a problem because then \( \text{reported} = \text{true} \).
    • **increment:**
      – \( \text{count} \) increases by 1
      – \( \text{first(Main)} \) increases by at least \( l_1 \): Before the step, \( \text{first(Main)} \leq \text{now} \), and after the step, \( \text{first(Main)} = \text{now} + l_1 \).
      – So the inequality is preserved.
Modeling timed systems (summary)

- **MMT automata** [Merritt, Modugno, Tuttle]
  - Simple, special-cased timed model
  - Immediate extension of I/O automata
  - Add upper and lower bounds for tasks.

- **GTA**, more general timed automata
  - Explicit time-passage steps

- **Timed I/O Automata**
  - Still more general
  - Instead of time-passage steps, use trajectories, which describe evolution of state over time.
  - [Kaynar, Lynch, Segala, Vaandrager] monograph
  - Tempo support
Simulation relations

• These work for GTAs/TIOAs too.
• Imply inclusion of sets of timed traces of admissible executions.
• Simulation relation definition (from A to B):
  – Every start state of A has a related start state of B. (As before.)
  – If s is a reachable state of A, u a related reachable state of B, and (s, π, s’) is a discrete step of A, then there is a timed execution fragment α of B starting with u, ending with some u’ of B that is related to s’, having the same timed trace as the given step, and containing no time-passage steps.
  – If s is a reachable state of A, u a related reachable state of B, and (s, υ(t), s’) is a time-passage step of A, then there is a timed execution fragment of B starting with u, ending with some u’ of B that is related to s’, having the same timed trace as the given step, and whose total time-passage is t.
Example: Two-task race

- Prove upper bound of $l + l_2 + (l_2 / l_1) l$ on time until report.
- Intuition:
  - Within time $l$, set flag true.
  - During time $l$, can increment count to at most approximately $l / l_1$.
  - Then it takes time at most $(l / l_1) l_2$ to decrement count to 0.
  - And at most another $l_2$ to report.
- Could prove a simulation relation, to a trivial GTA that just outputs report, at any time $\leq l + l_2 + (l_2 / l_1) l$.
- Express this using time variables:
  - now
  - last(report), initially $l + l_2 + (l_2 / l_1) l$.
- The simulation relation has an interesting form: inequalities involving the time variables:
Simulation relation

- s = state of race automaton, u = state of time bound spec automaton
- u.now = s.now, u.reported = s.reported
- u.last(report) ≥
  s.last(Int) + (s.count + 2) l_2 + (l_2 / l_1) (s.last(Int) – s.first(Main)),
  if s.flag = false and s.first(Main) ≤ s.last(Int),
  s.last(Main) + (s.count) l_2, otherwise.

- **Explanation:**
  - If flag = true, then time until report is the time until the next decrement, plus
    the time for the remaining decrements and the report.
  - Same if flag = false but must become true before another increment.
  - Otherwise, at least one more increment can occur before flag is set.
  - After set, it might take time (s.count + 1) l_2 to count down and report.
  - But current count could be increased some more:
    - At most 1 + (last(Int) – first(Main)) / l_1 times.
    - Multiply by l_2 to get extra time to decrement the additional count.
Timed Mutual Exclusion Algorithms
Timed mutual exclusion

- Model as before, but now the Us and the algorithm are MMT automata.
- Assume one task per process, with bounds $[l_1, l_2]$, $0 < l_1 \leq l_2 < \infty$.
- Users: Arbitrary tasks, boundmaps.

- **Mutual exclusion problem**: guarantee well-formedness, mutual exclusion, and progress, in all admissible timed executions.
- No high-level fairness guarantees, for now.
- Now, algorithm’s correctness is allowed to depend on timing assumptions.
Fischer mutual exclusion algorithm

- Famous, “published” only in email from Fischer to Lamport.
- A toy algorithm, widely used as a benchmark for modeling and verification methods for timing-based systems.
- Uses a single read/write register, `turn`.
- Compare: In asynchronous model, need n variables.

- Incorrect, asynchronous version (process i):
  - Trying protocol:
    - wait for `turn = 0`
    - `turn := i`
    - if `turn = i`, go critical; else go back to beginning
  - Exit protocol:
    - `turn := 0`
Incorrect execution

- To avoid this problem, add a timing constraint:
  - Process $i$ waits long enough between $\text{set}_i$ and $\text{check}_i$ so that no other process $j$ that sees $\text{turn} = 0$ before $\text{set}_i$ can set $\text{turn} := j$ after $\text{check}_i$.
  - That is, interval from $\text{set}_i$ to $\text{check}_i$ is strictly longer than interval from $\text{test}_j$ to $\text{set}_j$.

- Can ensure by counting steps:
  - Before checking, process $i$ waits $k$ steps, where $k > \frac{l_2}{l_1}$.
  - Shortest time from $\text{set}_i$ to $\text{check}_i$ is $k \cdot l_1$, which is greater than the longest time $l_2$ from $\text{test}_j$ to $\text{set}_j$.
Fischer mutex algorithm

• Pre/effect code, p. 777.
• Not quite in the assumed model:
  – That has just one task/process, with bounds \([l_1, l_2]\).
  – Here we use another task for the check, with bounds \([a_1, a_2]\), where \(a_1 = k l_1, a_2 = k l_2\),
  – This version is more like the ones used in most verification work.
• Proof?
  – Easy to see the algorithm avoids the bad example, but how do we know it’s always correct?
Proof of mutex property

• Use invariants.
• One of the earliest examples of an assertional proof for timed models.
• Key intermediate assertion:
  – If $pc_i = \text{check}$, $\text{turn} = i$, and $pc_j = \text{set}$, then $\text{first}(\text{check}_i) > \text{last}(\text{main}_j)$.
  – That is, if $i$ is about to check $\text{turn}$ and get a positive answer, and $j$ is about to set $\text{turn}$, then the earliest time when $i$ might check it is strictly after the latest time when $j$ might set it.
  – Rules out the bad interleaving.
• Can prove this by an easy induction.
• Use it to prove main assertion:
  – If $pc_i \in \{ \text{leave-try}, \text{crit}, \text{reset} \}$, then $\text{turn} = i$, and for every $j$, $pc_j \neq \text{set}$.
• Which immediately implies mutual exclusion.
Proof of progress

• Easy event-based argument:
  – By contradiction: Assume someone is in T, and no one is thereafter ever in C.
  – Then eventually region changes stop, everyone is in either T or R, at least one process is in T.
  – Eventually turn acquires a contender’s index, then stabilizes to some contender’s index, say i.
  – Then i proceeds to C.

• Refine this argument to a time bound, for the time from when someone is in T until someone is in C:
  – $2a_2 + 5l_2 = 2k l_2 + 5l_2$
  – Since k is approximately $L = l_2 / l_1$ (timing uncertainty ratio), this is $2L l_2 + O(l_2)$
  – Thus, timing uncertainty stretches the time complexity.
• **Q:** Why is the time complexity “stretched” by the timing uncertainty $L = (l_2/l_1)$, yielding an $L l_2$ term?

• Process $i$ must ensure that time $t = l_2$ has elapsed, to know that another process has had enough time to perform a step.

• Process $i$ determines this by counting its own steps.

• Must count at least $t/l_1$ steps to be sure that time $t$ has elapsed, even if $i$’s steps are fast ($l_1$).

• But the steps could be slow ($l_2$), so the total time could be as big as $(t/l_1) l_2 = (l_2/l_1) t = L t$.

• Requires real time $Lt$ for process in a system with timing uncertainty $L$ to be sure that time $t$ has elapsed.

• Similar stretching phenomenon arose in timeout example.
Lower bound on time

**Theorem:** There is no timed mutex algorithm for 2 processes with 1 shared variable, having an upper bound of $L l_2$ on the time for someone to reach $C$.

**Proof:**
- Like the proof that 1 register is insufficient for 2-process asynchronous mutual exclusion.
- By contradiction; suppose such an algorithm exists.
- Consider admissible execution $\alpha$ in which process 1 runs alone, slowly (all steps take $l_2$).
- By assumption, process 1 must enter $C$ within time $L l_2$.
- Must write to the register $x$ before $\rightarrow C$.
- Pause process 1 just before writing $x$ for the first time.
Lower bound on time

- Proof, cont’d:
  - Now run process 2, from where process 1 covers x.
  - p2 sees initial state, so eventually →C.
  - If p2 takes steps as slowly as possible ($l_2$), must →C within time $L \cdot l_2$.
  - If we speed p2 up (shrink), p2 →C within time $L \cdot l_2 (l_1 / l_2) = L \cdot l_1$.
  - So we can run process 2 all the way to C during the time p1 is paused, since $l_2 = L \cdot l_1$.
  - Then as in asynchronous case, can resume p1, overwrites x, enters C, contradiction.
The Fischer algorithm is fragile

- Depends on timing assumptions, even for the main safety property, mutual exclusion.
- It would be nice if safety were independent of timing (e.g., like Paxos).
- Can modify Fischer so mutual exclusion holds in all asynchronous runs, for n processes, using 3 registers [Section 24.3].
- But this fails to guarantee progress, even assuming timing eventually stabilizes (like Paxos).
- In fact, progress depends crucially on timing:
  - If time bounds are violated, then algorithm can deadlock, making future progress impossible.
- In fact, we have:
Another impossibility result!

- It’s **impossible** to guarantee n-process mutual exclusion in all asynchronous runs, progress if timing stabilizes, with < n registers:

- **Theorem:** There is no asynchronous read/write shared-memory algorithm for n ≥ 2 processes that:
  - Guarantees **well-formedness and mutual exclusion** when run asynchronously,
  - Guarantees **progress** when run so that each process’ step bounds eventually are in the range \([l_1, l_2]\), and
  - Uses < n shared registers.

- !!!

- **Proof:** Similar to that of impossibility of asynchronous mutex for < n registers (tricky).
Timed Consensus Algorithms
Consensus in timed systems

- **Network model:**
- **Process:**
  - MMT automaton, finitely many tasks.
  - Task bounds $[l_1, l_2]$, $0 < l_1 \leq l_2 < \infty$, $L = l_2 / l_1$
  - Stopping failures only.
- **Channels:**
  - GTA or TIOA
  - Reliable FIFO channels, upper bound of $d$ for every message.
- **Properties:**
  - Agreement,
  - Validity (weak or strong),
  - Failure-free termination
  - f-failure termination, wait-free termination

- In general, we’re allowed to rely on time bounds for both safety + liveness.
- **Q:** Can we solve fault-tolerant agreement? How many failures? How much time does it take?
Consensus in timed systems

• Assumptions:
  – $V = \{0, 1\}$,
  – Completely connected graph,
  – $l_1, l_2 << d$ (in fact, $n l_2 << d, L l_2 << d$).
  – Every task always enabled.

• Results:
  – Simple algorithm, for any number $f$ of failures, strong validity, time bound $\approx f L d$
  – Simple lower bound: $(f+1) d$.
  – More sophisticated algorithm: $\approx Ld + (2f+2) d$
  – More sophisticated lower bound: $\approx Ld + (f-1) d$

• [Attiya, Dwork, Lynch, Stockmeyer]
Simple algorithm

- **Implement a perfect failure detector**, which times out failed processes.
  - Process i sends periodic “alive” messages.
  - Process i determines process j has failed if i doesn’t receive any messages from j for a large number of i’s steps ($\approx (d + l_2) / l_1$ steps).
  - Time until detection at most $\approx L \cdot d + O(L \cdot l_2)$.
  - $Ld$ is the time needed for a timeout.

- **Use the failure detector to simulate a round-based synchronous consensus algorithm for the required $f+1$ rounds.**

- **Time for consensus at most $\approx f \cdot L \cdot d + O(f \cdot L \cdot l_2)$.**
Simple lower bound

- Upper bound (so far): $\approx f L d + O(f L l_2)$.
- Lower bound $(f+1)d$
  - Follows from $(f+1)$-round lower bound for synchronous model, via a model transformation.
- Note the role of the timing uncertainty $L$:
  - Appears in the upper bound: $f L d$, time for $f$ successive timeouts.
  - But doesn’t appear in the lower bound.
- Q: How does the real cost depend on $L$?
Better algorithm

- **Time bound:** \( Ld + (2f+2)d + O(f I_2 + L I_2) \)
  - Time for just one timeout!
  - Tricky algorithm, LTTR.
    - Uses a series of rounds, each involving an attempt to decide.
    - At even-numbered rounds, try to decide 0; at odd-numbered rounds, try to decide 1.
    - Each failure can cause an attempt to fail, move on to another round.
    - Successful round takes time at most \( \approx Ld \).
    - Unsuccessful round \( k \) takes time at most \( \approx (f_k + 1) d \), where \( f_k \) is the number of processes that fail at round \( k \).
Better lower bound

• Upper bound: \( \approx Ld + (2f+2)d \)
• Lower bound: \( Ld + (f-1) d \)

• Interesting proof---uses practically every lower bound technique we’ve seen:
  – Chain argument, as in Chapter 6.
  – Bivalence argument, as in Chapter 12.
  – Stretching and shrinking argument for timed executions, as in Chapter 24.

• LTTR
[Dwork, Lynch, Stockmeyer 88]

consensus results

- 2007 Dijkstra prize
- Weaken the time bound assumptions so that they hold eventually, from some point on, not necessarily always.
- Assume $n > 2f$ (unsolvable otherwise).
- Guarantees agreement, validity, $f$-failure termination.
  - Thus, safety properties (agreement and validity) don’t depend on timing.
  - Termination does---but in a nice way: guaranteed to terminate if time bound assumptions hold from any point on.
  - Similar to problem solved by Paxos.

- Algorithm:
  - Similar to Paxos (earlier), but allows less concurrency.
[DLS] algorithm

- Rotating coordinator as in 3-phase commit, pre-allocated “stages”.
- In each stage, one pre-determined coordinator takes charge, tries to coordinate agreement using a four-round protocol:
  1. Everyone sends “acceptable” values to coordinator; if coordinator receives “enough”, it chooses one to propose.
  2. Coordinator sends proposed value to everyone; anyone who receives it “locks” the value.
  3. Everyone who received a proposal in round 2 sends an ack to the coordinator; if coordinator receives “enough” acks, decides on the proposed value.
  4. Everyone exchanges lock info.
- “Acceptable” means opposite value isn’t locked.

- Implementing synchronous rounds:
  - Use the time assumptions.
  - Emulation may be unreliable until timing stabilizes.
  - That translates into possible lost messages, in earlier rounds.
  - Algorithm can tolerate lost messages before stabilization.
Mutual exclusion vs. consensus

- Mutual exclusion with \(< n\) shared registers:
  - Asynchronous systems:
    • Impossible
  - Timed systems:
    • Solvable, time upper bound \(O(L^{1/2})\), matching lower bound.
  - Systems where timing assumptions hold from some point on:
    • Impossible to guarantee both safety (mutual exclusion) and liveness (progress).

- Consensus with \(f\) failures, \(f \geq 1\):
  - Asynchronous systems:
    • Impossible
  - Timed systems:
    • Solvable, time upper bound \(Ld + O(d)\), matching lower bound.
  - Systems where timing assumptions hold from some point on:
    • Can guarantee both safety (agreement and validity) and liveness (f-failure termination), for \(n > 2f\).
Clock Synchronization
Algorithms
Clock synchronization

- **Network model:**
- **Process:**
  - TIOA
  - Includes a physical clock component that progresses at some (possibly varying) rate in the range \([1 - \rho, 1 + \rho]\).
  - Not under the process’ control.
- **Channels:**
  - GTA or TIOA
  - Reliable FIFO channels, message delay bounds in interval \([d_1, d_2]\).
- **Properties:**
  - Each node, at each time, computes the value of a logical clock
  - Agreement: Logical clocks should become, and remain, within a small constant \(\varepsilon\) of each other.
  - Validity: Logical clock values should be approximately within the range of the physical clock values.

- **Issues:**
  - Timing uncertainty
  - Tolerating failures
  - Scalability
  - Accommodating external clock inputs
Timing uncertainty

• E.g., 2 processes:
  – Messages from $p_1$ to $p_2$ might always take the minimum time $d_1$.
  – Messages from $p_2$ to $p_1$ might always take the maximum time $d_2$.
  – Or vice versa.
  – Either way, the logical clocks are supposed to be within $\varepsilon$ of each other.
  – Implies that $\varepsilon \geq \frac{(d_2 - d_1)}{2}$

• Can achieve $\varepsilon \approx \frac{(d_2 - d_1)}{2}$, if clock drift rate is very small and there are no failures.

• For $n$ processes in fully connected graph, can achieve $\varepsilon \approx \frac{(d_2 - d_1)(1 - 1/n)}$, and that’s provably optimal.
Accommodating failures

• Several published algorithms for \( n > 3f \) processes to establish and maintain clock synchronization, in the presence of up to \( f \) Byzantine faulty processes.
  – [Lamport], [Dolev, Strong], [Lundelius, Lynch],…
  – Some algorithms perform fault-tolerant averaging.
  – Some wait until \( f+1 \) processes claim a time has been reached before jumping to that time.
  – Etc.

• Lower bound: \( n > 3f \) is necessary.
  – Original proof: [Dolev, Strong]
  – Cuter proof: [Fischer, Lynch, Merritt]
    • By contradiction: Assume (e.g.) a 3-process clock synch algorithm that tolerates 1 Byzantine faulty process.
    • Form a large ring, from many copies of the algorithm:
Accommodating failures

• Lower bound proof: \( n > 3f \) necessary
  – By contradiction: Assume a 3-process clock synch algorithm that tolerates 1 Byzantine faulty process.
  – Form a large ring, from many copies of the algorithm:

  ![Diagram](image)

  – Let the physical clocks drift progressively, as we move around the ring, fastest and slowest at opposite sides of the ring.
  – Any consecutive pair’s logical clocks must remain within \( \epsilon \) of each other, by agreement, and must remain approximately within the range of their physical clocks, by validity.
  – Can’t satisfy this everywhere in the ring.
Scalability

- Large, not-fully-connected network.
- E.g., a line:

  ![Diagram of a line network with asynchronous communication](attachment:image)

- Can’t hope to synchronize distant nodes too closely.
- Instead, try to achieve a gradient property, saying that neighbors’ clocks are always closely synchronized.
- Impossibility result for gradient clock synch [Fan 04]: Any clock synch algorithm in a line of length $D$ has some reachable state in which the logical clocks of two neighbors are $\Omega(\log D / \log \log D)$ apart.
- Algorithms exist that achieve a constant gradient “most of the time”.
- And newer algorithms that achieve $O(\log D)$ all of the time.
External clock inputs

- Practical clock synch algorithms use reliable external clock sources:
  - NTP time service in Internet
  - GPS in mobile networks
- Nodes with reliable time info send it to other nodes.
- Recipients may correct for communication delays
- Typically ignore failures.
Mobile Wireless Network Algorithms
Mobile networks

- Nodes moving in physical space, communicating using local broadcast.
- Mobile phones, hand-held computers; robots, vehicles, airplanes
- Physical space:
  - Generally 2-dimensional, sometimes 3
- Nodes:
  - Have uids.
  - May know the approximate real time, and their own approximate locations.
  - May fail or be turned off, may restart.
  - Don’t know a priori who else is participating, or who is nearby.
- Communication:
  - Broadcast, received by nearby listening nodes.
  - May be unreliable, subject to collisions/losses, or
  - May be assumed reliable (relying on backoff mechanisms to mask losses).
- Motion:
  - Usually unpredictable, subject to physical limitations, e.g. velocity bounds.
  - May be controllable (robots).
- Q: What problems can/cannot be solved in such networks?
Some preliminary results

• Dynamic graph model
  – Welch, Walter, Vaidya,…
  – Algorithms for mutual exclusion, k-exclusion, message routing,…

• Wireless networks with collisions
  – Algorithms / lower bounds for broadcast in the presence of collisions [Bar-Yehuda, Goldreich, Itai], [Kowalski, Pelc],…
  – Algorithms / lower bounds for consensus [Newport, Gilbert, et al.]

• Rambo atomic memory algorithm
  – [Gilbert, Lynch, Shvartsman]
  – Reconfigurable Atomic Memory for Basic (read/write) Objects
  – Implemented using a changing quorum system configuration.
  – Paxos consensus used to change the configuration, runs in the background without interfering with ongoing reads/writes.

• Virtual Node abstraction layers for mobile networks
  – Gilbert, Nolte, Brown, Newport,…
Some preliminary results

• Neighbor discovery, counting number of nodes, maintaining network structures,…

• Leave all this for another course.
VN Layers for mobile networks

- Add Virtual Nodes: Simple state machines (TIOAs) located at fixed, known geographical locations (e.g., grid points).
- Mobile nodes in the vicinity emulate the VSNs, using a Replicated State Machine approach, with an elected leader managing communication.
- Virtual Nodes may fail, later recover in initial state.
- Program applications over the VSN layer.
  - Geocast, location services, point-to-point communication, bcast.
  - Data collection and dissemination.
  - Motion coordination (robots, virtual traffic lights, virtual air-traffic controllers).

- Other work: Neighbor discovery, counting number of nodes, maintaining network structures,…
- Leave all this for another course.
Next time…

• There is no next time!
• Have a very nice break!