6.852: Distributed Algorithms
Fall, 2009

Class 8
Today’s plan

- Basic asynchronous system model, continued
  - Hierarchical proofs
  - Safety and liveness properties
- Asynchronous networks
- Asynchronous network algorithms:
  - Leader election in a ring
  - Leader election in a general network
- Reading: Sections 8.5.3 and 8.5.5, Chapter 14, Sections 15.1-15.2.
- Next:
  - Constructing a spanning tree
  - Breadth-first search
  - Shortest paths
  - Minimum spanning trees
  - Reading: Section 15.3-15.5, [Gallager, Humblet, Spira]
Last time

• Defined basic math framework for modeling asynchronous systems.
• I/O automata
• Executions, traces
• Operations: Composition, hiding
• Proof methods and concepts
  – Compositional methods
  – Invariants
  – Trace properties, including safety and liveness properties.
  – Hierarchical proofs
Input/output automaton

- **sig** = (in, out, int)
  - input, output, internal actions (disjoint)
  - acts = in ∪ out ∪ int
  - ext = in ∪ out
  - local = out ∪ int

- **states**: Not necessarily finite
- **start** ⊆ states
- **trans** ⊆ states × acts × states
  - Input-enabled: Any input “enabled” in any state.
- **tasks**, partition of locally controlled actions
  - Used for liveness.
Channel automaton

- Reliable unidirectional FIFO channel between two processes.
  - Fix message alphabet $M$.

- signature
  - input actions: $\text{send}(m)$, $m \in M$
  - output actions: $\text{receive}(m)$, $m \in M$
  - no internal actions

- states
  - queue: FIFO queue of $M$, initially empty
Channel automaton

- **trans**
  - send(m)
    - effect: add m to (end of) queue
  - receive(m)
    - precondition: m is at head of queue
    - effect: remove head of queue

- **tasks**
  - All receive actions in one task.
Executions

- An I/O automaton executes as follows:
  - Start at some start state.
  - Repeatedly take step from current state to new state.
- Formally, an execution is a finite or infinite sequence:
  - $s_0 \pi_1 s_1 \pi_2 s_2 \pi_3 s_3 \pi_4 s_4 \pi_5 s_5 \ldots$ (if finite, ends in state)
  - $s_0$ is a start state
  - $(s_i, \pi_{i+1}, s_{i+1})$ is a step (i.e., in trans)

$\lambda, \text{send}(a), a, \text{send}(b), ab, \text{receive}(a), b, \text{receive}(b), \lambda$
Execution fragments

- An I/O automaton executes as follows:
  - Start at some start state.
  - Repeatedly take step from current state to new state.
- Formally, an execution is a sequence:
  - \( s_0, \pi_1, s_1, \pi_2, s_2, \pi_3, s_3, \pi_4, s_4, \pi_5, s_5, \ldots \)
  - \( s_0 \) is a start state
  - \( (s_i, \pi_{i+1}, s_{i+1}) \) is a step.
Traces

- Models external behavior, useful for defining correctness.
- A trace of an execution is the subsequence of external actions in the execution.
  - Denoted $\text{trace}(\alpha)$, where $\alpha$ is an execution.
  - No states, no internal actions.

\[
\lambda, \text{send}(a), a, \text{send}(b), ab, \text{receive}(a), b, \text{receive}(b), \lambda
\]

\[
\text{send}(a), \text{send}(b), \text{receive}(a), \text{receive}(b)
\]
Composition of compatible automata

- Compose two automata A and B (see book for general case).
- \( \text{out}(A \times B) = \text{out}(A) \cup \text{out}(B) \)
- \( \text{int}(A \times B) = \text{int}(A) \cup \text{int}(B) \)
- \( \text{in}(A \times B) = \text{in}(A) \cup \text{in}(B) - (\text{out}(A) \cup \text{out}(B)) \)
- \( \text{states}(A \times B) = \text{states}(A) \times \text{states}(B) \)
- \( \text{start}(A \times B) = \text{start}(A) \times \text{start}(B) \)
- \( \text{trans}(A \times B): \) includes \( (s, \pi, s') \) iff
  - \( (s_A, \pi, s'_A) \in \text{trans}(A) \) if \( \pi \in \text{acts}(A) \); \( s_A = s'_A \) otherwise.
  - \( (s_B, \pi, s'_B) \in \text{trans}(B) \) if \( \pi \in \text{acts}(B) \); \( s_B = s'_B \) otherwise.
- \( \text{tasks}(A \times B) = \text{tasks}(A) \cup \text{tasks}(B) \)

- Notation: \( \prod_{i \in I} A_i \), for composition of \( A_i : i \in I \) (I countable)
Hierarchical proofs
Hierarchical proofs

- Important strategy for proving correctness of complex asynchronous distributed algorithms.
- Define a series of automata, each implementing the previous one ("successive refinement").
- Highest-level = Problem specification.
- Then a high-level algorithm description.
- Then more and more detailed versions, e.g.:
  - High levels centralized, lower levels distributed.
  - High levels inefficient but simple, lower levels optimized and more complex.
  - High levels with large granularity steps, lower levels with finer granularity steps.
- Reason about lower levels by relating them to higher levels.
- Similar to what we did for synchronous algorithms.
Hierarchical proofs

• For synchronous algorithms (recall):
  – Optimized algorithm runs side-by-side with unoptimized version, and “invariant” proved to relate the states of the two algorithms.
  – Prove using induction.

• For asynchronous algorithms, it’s harder:
  – Asynchronous model has more nondeterminism (in choice of new state, in order of steps).
  – So, harder to determine which execs to compare.

• One-way implementation is enough:
  – For each execution of the lower-level algorithm, there is a corresponding execution of the higher-level algorithm.
  – “Everything the algorithm does is allowed by the spec.”
  – Don’t need the other direction: doesn’t matter if the algorithm does everything that is allowed.
Simulation relations

- Most common method of proving that one automaton implements another.
- Assume A and B have the same extsig, and R is a relation from states(A) to states(B).
- Then R is a simulation relation from A to B provided:
  - $s_A \in \text{start}(A)$ implies there exists $s_B \in \text{start}(B)$ such that $s_A R s_B$.
  - If $s_A, s_B$ are reachable states of A and B, $s_A R s_B$ and $(s_A, \pi, s'_A)$ is a step, then there is an execution fragment $\beta$ starting with $s_B$ and ending with $s'_B$ such that $s'_A R s'_B$ and $\text{trace}(\beta) = \text{trace}(\pi)$. 
Simulation relations

- \( R \) is a simulation relation from A to B provided:
  - \( s_A \in \text{start}(A) \) implies \( \exists s_B \in \text{start}(B) \) such that \( s_A R s_B \).
  - If \( s_A, s_B \) are reachable states of A and B, \( s_A R s_B \) and \( (s_A, \pi, s'_A) \) is a step, then \( \exists \beta \) starting with \( s_B \) and ending with \( s'_B \) such that \( s'_A R s'_B \) and \( \text{trace}(\beta) = \text{trace}(\pi) \).
Simulation relations

- **Theorem:** If there is a simulation relation from A to B then traces(A) ⊆ traces(B).
- This means all traces of A, not just finite traces.
- **Proof:** Fix a trace of A, arising from a (possibly infinite) execution of A.
- Create a corresponding execution of B, using an iterative construction.

\[ S_0,A \xrightarrow{\Pi_1} S_1,A \xrightarrow{\Pi_2} S_2,A \xrightarrow{\Pi_3} S_3,A \xrightarrow{\Pi_4} S_4,A \xrightarrow{\Pi_5} S_5,A \]
Simulation relations

- **Theorem:** If there is a simulation relation from A to B then \( \text{traces}(A) \subseteq \text{traces}(B) \).
Simulation relations

- Theorem: If there is a simulation relation from $A$ to $B$ then $\text{traces}(A) \subseteq \text{traces}(B)$. 

\[
\begin{align*}
\pi_1 : S_{0,A} &\rightarrow S_{1,A} \\
\pi_2 : S_{1,A} &\rightarrow S_{2,A} \\
\pi_3 : S_{2,A} &\rightarrow S_{3,A} \\
\pi_4 : S_{3,A} &\rightarrow S_{4,A} \\
\pi_5 : S_{4,A} &\rightarrow S_{5,A}
\end{align*}
\]
Simulation relations

- Theorem: If there is a simulation relation from \( A \) to \( B \) then \( \text{traces}(A) \subseteq \text{traces}(B) \).
Example: Channels

- Show two channels implement one.

- Rename some actions.
- Claim that $D = \text{hide}_{\{\text{pass}(m)\}} A \times B$ implements $C$, in the sense that $\text{traces}(D) \subseteq \text{traces}(C)$. 
Recall: Channel automaton

- Reliable unidirectional FIFO channel.
- signature
  - Input actions: send(m), m ∈ M
  - output actions: receive(m), m ∈ M
  - no internal actions
- states
  - queue: FIFO queue of M, initially empty
Channel automaton

- **trans**
  - send(m)
    - effect: add m to queue
  - receive(m)
    - precondition: m = head(queue)
    - effect: remove head of queue

- **tasks**
  - All receive actions in one task
Composing two channel automata

- Output of B is input of A
  - Rename receive(m) of B and send(m) of A to pass(m).
- \( D = \text{hide} \{ \text{pass}(m) \mid m \in M \} \) \( A \times B \) implements C
- Define simulation relation \( R \):
  - For \( s \in \text{states}(D) \) and \( u \in \text{states}(C) \), \( s R u \) iff \( u.\text{queue} \) is the concatenation of \( s.A.\text{queue} \) and \( s.B.\text{queue} \)
- Proof that this is a simulation relation:
  - Start condition: All queues are empty, so start states correspond.
  - Step condition: Define “step correspondence”:
Composing two channel automata

\[
\text{s } R \text{ u iff u.queue is concatenation of s.A.queue and s.B.queue}
\]

- **Step correspondence:**
  - For each step \((s, \pi, s') \in \text{trans}(D)\) and u such that \(s R u\), define execution fragment \(\beta\) of C:
    - Starts with \(u\), ends with \(u'\) such that \(s' R u'\).
    - \(\text{trace}(\beta) = \text{trace}(\pi)\)
  - Here, actions in \(\beta\) happen to depend only on \(\pi\), and uniquely determine post-state.
    - Same action if external, empty sequence if internal.
Composing two channel automata

\[ s \; R \; u \; \text{iff} \; u.\text{queue} \; \text{is concatenation of} \; s.A.\text{queue} \; \text{and} \; s.B.\text{queue} \]

- **Step correspondence:**
  - \( \pi = \text{send}(m) \) in D corresponds to \( \text{send}(m) \) in C
  - \( \pi = \text{receive}(m) \) in D corresponds to \( \text{receive}(m) \) in C
  - \( \pi = \text{pass}(m) \) in D corresponds to \( \lambda \) in C

- **Verify that this works:**
  - Actions of C are enabled.
  - Final states related by relation R.

- **Routine case analysis:**
Showing R is a simulation relation

\[ s \ R \ u \ \text{iff} \ u.\text{queue} \text{ is concatenation of } s.A.\text{queue} \text{ and } s.B.\text{queue} \]

- **Case:** \( \pi = \text{send}(m) \)
  - No enabling issues (input).
  - Must check \( s' \ R \ u' \).
    - Since \( s \ R \ u \), \( u.\text{queue} \) is the concatenation of \( s.A.\text{queue} \) and \( s.B.\text{queue} \).
    - Adding the same \( m \) to the end of \( u.\text{queue} \) and \( s.B.\text{queue} \) maintains the correspondence.

- **Case:** \( \pi = \text{receive}(m) \)
  - Enabling: Check that \( \text{receive}(m) \), for the same \( m \), is also enabled in \( u \).
    - We know that \( m \) is first on \( s.A.\text{queue} \).
    - Since \( s \ R \ u \), \( m \) is first on \( u.\text{queue} \).
    - So enabled in \( u \).
  - \( s' \ R \ u' \): Since \( m \) removed from both \( s.A.\text{queue} \) and \( u.\text{queue} \).
Showing $R$ is a simulation relation

$s \ R \ u$ iff $u$.queue is concatenation of $s$.A.queue and $s$.B.queue

- **Case**: $\pi = \text{pass}(m)$
  - No enabling issues (since no high-level steps are involved).
  - Must check $s' \ R \ u$:
    - Since $s \ R \ u$, $u$.queue is the concatenation of $s$.A.queue and $s$.B.queue.
    - Concatenation is unchanged as a result of this step, so also $u$.queue is the concatenation of $s'$.A.queue and $s'$.B.queue.
Safety and liveness properties
Specifications

• **Trace property:**
  – Problem specification in terms of external behavior.
  – \(( \text{sig}(P), \text{traces}(P) )\)

• Automaton \(A\) satisfies trace property \(P\) if \(\text{extsig}(A) = \text{sig}(P)\) and (two different notions, depending on whether we’re interested in liveness or not):
  – \(\text{traces}(A) \subseteq \text{traces}(P)\), or
  – \(\text{fairtraces}(A) \subseteq \text{traces}(P)\).

• All the problems we’ll consider for asynchronous systems can be formulated as trace properties.

• And we’ll usually be concerned about liveness, so we will use the second notion.
Safety property S

- traces(S) are nonempty, prefix-closed, and limit-closed.
- “Something bad” never happens.
- Violations occur at some finite point in the sequence.

- **Examples** (we’ll see all these later):
  - Consensus: Agreement, validity
    - Describe as set of sequences of init and decide actions in which we never disagree, or never violate validity.
  - Graph algorithms: Correct shortest paths, correct minimum spanning trees,…
    - Outputs do not yield any incorrect answers.
  - Mutual exclusion: No two grants without intervening returns.
Proving a safety property

• That is, prove that all traces of A satisfy S.
• By limit-closure, it’s enough to prove that all finite traces satisfy S.
• Can do this by induction on length of trace.
• Using invariants:
  – For most trace safety properties, can find a corresponding invariant.
  – Example: Consensus
    • Record decisions in the state.
    • Express agreement and validity in terms of recorded decisions.
  – Then prove the invariant as usual, by induction.
Liveness property L

- Every finite sequence over \( \text{sig}(L) \) has some extension in \( \text{traces}(L) \).

- Examples:
  - Temination: No matter where we are, we could still terminate in the future.
  - Some event happens infinitely often.

- Proving liveness properties:
  - Measure progress toward goals, using progress functions.
  - Intermediate milestones.
  - Formal reasoning using temporal logic.
  - Methods less well-established than those for safety properties.
Safety and liveness

• **Theorem:** Every trace property can be expressed as the intersection of a safety and a liveness property.

• So, to specify a property, it’s enough to specify safety requirements and liveness requirements separately.

• Typical specifications of problems for asynchronous systems consist of:
  – A list of safety properties.
  – A list of liveness properties.
  – Nothing else.
Asynchronous network model
Send/receive systems

- Digraph $G = (V,E)$, with:
  - Process automata associated with nodes, and
  - Channel automata associated with directed edges.
- Model processes and channels as automata, compose.
- Processes

  - User interface: $\text{inv}$, $\text{resp}$.
  - Problems specified in terms of allowable traces at user interface
    - Hide send/receive actions
  - Failure modeling, e.g.:
  - Having explicit stop actions in external interface allows problems to be stated in terms of occurrence of failures.
Different kinds of channel with this interface:
- Reliable FIFO, as before.
- Weaker guarantees: Lossy, duplicating, reordering

Can define channels by trace properties, using a “cause” function mapping receives to sends.
- Integrity: Cause function preserves message.
- No loss: Function is onto (surjective).
- No duplicates: Function is 1-1 (injective).
- No reordering: Function is order-preserving.

Reliable channel satisfies all of these; weaker channels satisfy Integrity but weaken some of the other properties.
Broadcast and multicast

- **Broadcast**
  - Reliable FIFO between each pair.
  - Different processes can receive msgs from different senders in different orders.
  - Model using separate queues for each pair.

- **Multicast:** Processes designate recipients.

- Also consider bcast, mcast with failures, and/or with additional consistency conditions.
Asynchronous network algorithms
Asynchronous network algorithms

- Assume reliable FIFO point-to-point channels
- Revisit problems we considered in synchronous networks:
  - Leader election:
    - In a ring.
    - In general undirected networks.
  - Spanning tree
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree
- How much carries over?
  - Where did we use synchrony assumption?
Leader election in a ring

• Assumptions:
  – G is a ring, unidirectional or bidirectional communication
  – Local names for neighbors, UIDs

• LeLann-Chang-Roberts (AsynchLCR)
  – Send UID clockwise around ring (unidirectional).
  – Discard UIDs smaller than your own.
  – Elect self if your UID comes back.
  – Correctness: Basically the same as for synchronous version, with a few complications:
    • Finer granularity, consider individual steps rather than entire rounds.
    • Must consider messages in channels.
AsynchLDR, process i

- **Signature**
  - in \( \text{rcv}(v)_{i-1,i} \), \( v \) is a UID
  - out \( \text{send}(v)_{i,i+1} \), \( v \) is a UID
  - out \( \text{leader}_i \)

- **State variables**
  - \( u \): UID, initially i’s UID
  - send: FIFO queue of UIDs, initially containing i’s UID
  - status: unknown, chosen, or reported, initially unknown

- **Tasks**
  - \{ \( \text{send}(v)_{i,i+1} \) | \( v \) is a UID \} and \{ \( \text{leader}_i \) \}

- **Transitions**
  - \( \text{send}(v)_{i,i+1} \)
    - pre: \( v = \text{head}(\text{send}) \)
    - eff: remove head of send
  - \( \text{receive}(v)_{i-1,i} \)
    - eff:
      - if \( v = u \) then \( \text{status} := \text{chosen} \)
      - if \( v > u \) then add \( v \) to send
  - \( \text{leader}_i \)
    - pre: \( \text{status} = \text{chosen} \)
    - eff: \( \text{status} := \text{reported} \)
AsynchLCR properties

- **Safety**: No process other than $i_{\text{max}}$ ever performs leader$_i$.
- **Liveness**: $i_{\text{max}}$ eventually performs leader$_i$. 
Safety proof

- **Safety**: No process other than $i_{\text{max}}$ ever performs leader$_i$.

- Recall synchronous proof, based on showing invariant of global states, after any number of **rounds**:
  - If $i \neq i_{\text{max}}$ and $j \in [i_{\text{max}}, i)$ then $u_i$ not in send$_j$.

- Can use a similar invariant for the asynchronous version.

- But now the invariant must hold after any number of **steps**:
  - If $i \neq i_{\text{max}}$ and $j \in [i_{\text{max}}, i)$ then $u_i$ not in send$_j$ or in queue$_{j,j+1}$.

- Prove by induction on number of steps.
  - Use cases based on type of action.
  - Key case: receive($v$)$_{i_{\text{max}}-1, i_{\text{max}}}$
    - Argue that if $v \neq u_{\text{max}}$ then $v$ gets discarded.
Liveness proof

- **Liveness**: \( i_{\text{max}} \) eventually performs leader \( i \).

- Synchronous proof used an invariant saying exactly where the max is after \( r \) rounds.
- Now no rounds, need a different proof.
- Can establish intermediate milestones:
  - For \( k \in [0,n-1] \), \( u_{\text{max}} \) eventually in send \( i_{\text{max}+k} \)
  - Prove by induction on \( k \); use fairness for process and channel to prove inductive step.
Complexity

- **Msgs:** $O(n^2)$, as before.

- **Time:** $O(n(l+d))$
  - $l$ is an upper bound on local step time for each process (that is, for each process task).
  - $d$ is an upper bound on time to deliver first message in each channel (that is, for each channel task).
  - Measuring real time here (not counting rounds).
  - Only upper bounds, so does not restrict executions.
  - Bound still holds in spite of the possibility of “pileups” of messages in channels and send buffers.
    - Pileups can be interpreted as meaning that some tokens have sped up.
    - See analysis in book.
Reducing the message complexity

- **Hirschberg-Sinclair:**
  - Sending in both directions, to successively doubled distances.
  - Extends immediately to asynchronous model.
  - $O(n \log n)$ messages.
  - Use bidirectional communication.

- **Peterson's algorithm:**
  - $O( n \log n)$ messages
  - Unidirectional communication
  - Unknown ring size
  - Comparison-based
Peterson’s algorithm

- Proceed in asynchronous “phases” (may execute concurrently).
- In each phase, each process is active or passive.
  - Passive processes just pass messages along.
- In each phase, at least half of the active processes become passive; so at most \( \log n \) phases until election.
- **Phase 1:**
  - Send UID two processes clockwise; collect two UIDs from predecessors.
  - Remain active iff the middle UID is max.
  - In this case, adopt middle UID (the max one).
  - Some process remains active (assuming \( n \geq 2 \), but no more than half.
- **Later phases:**
  - Same, except that the passive processes just pass messages on.
  - No more than half of those active at the beginning of the phase remain active.
- **Termination:**
  - If a process sees that its immediate predecessor’s UID is the same as its own, elects itself the leader (knows it’s the only active process left).
PetersonLeader

- **Signature**
  - \textit{in} receive(v)_{i-1,i}, v is a UID
  - \textit{out} send(v)_{i,i+1}, v is a UID
  - \textit{out} leader_i

- \textit{int} get-second-uid_i
- \textit{int} get-third-uid_i
- \textit{int} advance-phase_i
- \textit{int} become-relay_i
- \textit{int} relay_i

- **State variables**
  - \textit{mode}: active or relay, initially active
  - \textit{status}: unknown, chosen, or reported, initially unknown
  - \textit{uid1}; initially i's UID
  - \textit{uid2}; initially null
  - \textit{uid3}; initially null
  - \textit{send}: FIFO queue of UIDs; initially contains i's UID
  - \textit{receive}: FIFO queue of UIDs
PetersonLeader

- **get-second-uid_i**
  pre: **mode** = active  
  **receive** is nonempty  
  **uid2** = null  
  eff: **uid2** := head(**receive**)  
    remove head of **receive**  
    add **uid2** to **send**  
    if **uid2** = **uid1** then  
      **status** := chosen

- **advance-phase_i**
  pre: **mode** = active  
  **uid3** ≠ null  
  **uid2** > max(\(\text{uid1}, \text{uid3}\))  
  eff: **uid1** := **uid2**  
    **uid2** := null  
    **uid3** := null  
    add **uid1** to **send**

- **get-third-uid_i**
  pre: **mode** = active  
  **receive** is nonempty  
  **uid2** ≠ null  
  **uid3** = null  
  eff: **uid3** := head(**receive**)  
    remove head of **receive**

- **become-relay_i**
  pre: **mode** = active  
  **uid3** ≠ null  
  **uid2** ≤ max(\(\text{uid1}, \text{uid3}\))  
  eff: **mode** := relay

- **relay_i**
  pre: **mode** = relay  
  **receive** is nonempty  
  eff: move head(\(\text{receive}\)) to **send**
PetersonLeader

- **Tasks:**
  - \{ send(v)_{i,i+1} \mid v \text{ is a UID} \}
  - \{ \text{get-second-uid}_i, \text{get-third-uid}_i, \text{advance-phase}_i, \text{become-relay}_i, \text{relay}_i \}
  - \{ \text{leader}_i \}

- **Number of phases is** \(O(\log n)\)

- **Complexity**
  - Messages: \(O(n \log n)\)
  - Time: \(O(n(l+d))\)
Leader election in a ring

- Can we do better than $O(n \log n)$ message complexity?
  - Not with comparison-based algorithms. (Why?)
  - Not at all: Can prove a lower bound.
Lower bound for leader election in asynchronous network.

Assume:
- Ring size $n$ is unknown (algorithm must work in arbitrary size rings).
- UIDS:
  - Chosen from some infinite set.
  - No restriction on allowable operations.
  - All processes identical except for UIDs.
- Bidirectional communication allowed.

Consider combinations of processes to form:
- Rings, as usual.
- Lines, where nothing is connected to the ends and no input arrives there.
- Ring looks like line if communication delayed across ends.
\( \Omega(n \log n) \) lower bound

- **Lemma 1:** There are infinitely many process automata, each of which can send at least one message without first receiving one (in some execution).
- **Proof:**
  - If not, there are two processes \( i, j \), neither of which ever sends a message without first receiving one.
  - Consider 1-node ring:
    - \( i \) must elect itself, with no messages sent or received.
  - Consider:
    - \( j \) must elect itself, with no messages sent or received.
  - Now consider:
    - Both \( i \) and \( j \) elect themselves, contradiction.
**Ω(n log n) lower bound**

- **C(L)** = maximum (actually, supremum) of the number of messages that are sent in a single input-free execution of line L.

- **Lemma 2**: If L₁, L₂, L₃ are three line graphs of even length l such that C(Lᵢ) ≥ k for i = 1, 2, 3, then C(Lᵢ join Lⱼ) ≥ 2k + l/2 for some i ≠ j

- **Proof**:
  - Suppose not.
  - Consider two lines, L₁ join L₂ and L₂ join L₁.
Proof of Lemma 2

- Let $\alpha_i$ be finite execution of $L_i$ with $\geq k$ messages.
- Run $\alpha_1$ then $\alpha_2$ then $\alpha_{1,2}$, an execution fragment of $L_1$ join $L_2$ beginning with messages arriving across the join boundary.
- By assumption, fewer than $l/2$ additional messages are sent in $\alpha_{1,2}$.
- So, the effects of the new inputs don’t cross the middle edges of $L_1$ and $L_2$ before the system quiesces (no more messages sent).

- Similarly for $\alpha_{2,1}$, an execution of $L_2$ join $L_1$. 

\begin{itemize}
  \item \end{itemize}
Proof of Lemma 2

• Now consider three rings:
Proof of Lemma 2

- Connect both ends of $L_1$ and $L_2$.
  - Right neighbor in line is clockwise around ring.
- Run $\alpha_1$ then $\alpha_2$ then $\alpha_{1,2}$ then $\alpha_{2,1}$.
  - No interference between $\alpha_{1,2}$ and $\alpha_{2,1}$.
  - Quiesces: Eventually no more messages are sent.
  - Must elect leader (possibly in extension, but without any more messages).
- Assume WLOG that elected leader is in “bottom half”.
Proof of Lemma 2

- Same argument for ring constructed from \( L_2 \) and \( L_3 \).
- Can leader be in bottom half?
- No!
- So must be in top half.
Proof of Lemma 2
Proof of Lemma 2
Summarizing, we have:

- **Lemma 1**: There are infinitely many process automata, each of which can send at least one message without first receiving one.

- **Lemma 2**: If $L_1, L_2, L_3$ are three line graphs of even length $l$ such that $C(L_i) \geq k$ for all $i$, then $C(L_i \text{ join } L_j) \geq 2k + l/2$ for some $i \neq j$.

Now combine:

- **Lemma 3**: For any $r \geq 0$, there are infinitely many disjoint line graphs $L$ of length $2^r$ such that $C(L) \geq r \cdot 2^{r-2}$.
  
  - **Base ($r = 0$)**: Trivial claim.
  
  - **Base ($r = 1$)**: Use Lemma 1
    
    - Just need length-2 lines sending at least one message.
  
  - **Inductive step ($r \geq 2$)**:
    
    - Choose $L_1, L_2, L_3$ of length $2^{r-1}$ with $C(L_i) \geq (r-1) \cdot 2^{r-3}$.
    
    - By Lemma 2, for some $i, j$, $C(L_i \text{ join } L_j) \geq 2(r-1)2^{r-3} + 2^{r-1}/2 = r \cdot 2^{r-2}$.
Lower bound, cont’d

- **Lemma 3:** For any \( r \geq 0 \), there are infinitely many disjoint line graphs \( L \) of length \( 2^r \) such that \( C(L) \geq r 2^{r-2} \).

- **Theorem:** For any \( r \geq 0 \), there is a ring \( R \) of size \( n = 2^r \) such that \( C(R) = \Omega(n \log n) \).
  - Choose \( L \) of length \( 2^r \) such that \( C(L) \geq r 2^{r-2} \).
  - Connect ends, but delay communication across boundary.

- **Corollary:** For any \( n \geq 0 \), there is a ring \( R \) of size \( n \) such that \( C(R) = \Omega(n \log n) \).
Leader election in general networks

- Undirected graphs.
- Can get asynchronous version of synchronous FloodMax algorithm:
  - Simulate rounds with counters.
  - Need to know diameter for termination.
- We’ll see better asynchronous algorithms later:
  - Don’t need to know diameter.
  - Lower message complexity.
- Depend on techniques such as:
  - Breadth-first search
  - Convergecast using a spanning tree
  - Synchronizers to simulate synchronous algorithm
  - Consistent global snapshots to detect termination.
Next lecture

- More asynchronous network algorithms
  - Constructing a spanning tree
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree (GHS)

- Reading: Section 15.3-15.5, [Gallager, Humblet, Spira]