6.852: Distributed Algorithms
Fall, 2015

Lecture 10

Today's plan

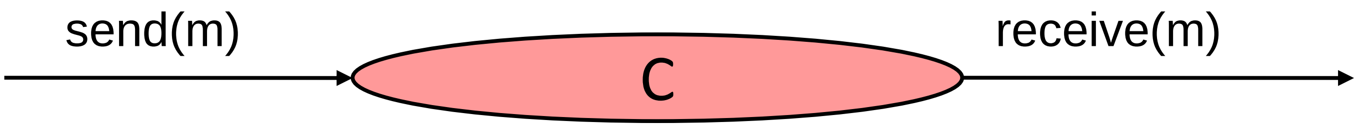
- I/O Automata, cont'd
- Asynchronous network model
- Asynchronous network algorithms:
  - Leader election
  - Constructing a spanning tree
- Readings:
  - Chapter 8
  - Chapter 14
  - Section 15.1-15.3
- Next:
  - Breadth-first search
  - Shortest paths
  - Minimum spanning trees
  - Readings:
    - Section 15.3-15.5
    - [Gallager, Humblet, Spira]

Input/Output automaton

- $sig = (in, out, int)$
  - $act = in \cup out \cup int$
  - $ext = in \cup out$
  - $local = out \cup int$
- $states$
- $start \subseteq states$
- $trans \subseteq states \times acts \times states$
- $tasks$, partition of locally controlled actions

- Action π is enabled in a state s if $trans$ contains a step (s, π, s') for some s'.
- I/O automata are input-enabled.

Channel automaton



- signature
  - input actions: $send(m), m \in M$
  - output actions: $receive(m), m \in M$
- states
  - $queue$: FIFO queue of M, initially empty
- 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 execution of an I/O automaton is a finite or infinite sequence:
  - $s_0\ \pi_1\ s_1\ \pi_2\ s_2\ \pi_3\ s_3\ \pi_4\ s_4\ \pi_5\ s_5$ ... (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)
- Execution fragment: Same, but might not begin in a start state.
- The trace of an execution is the subsequence of external actions in the execution.
- A trace of an I/O automaton is the trace of any execution of the automaton.

Properties and Proof Methods

- Compositional reasoning
- Invariants
- Trace properties
- Simulation relations

Compositional reasoning

- Use Theorems 1-6 (projection, pasting, substitutivity) to infer properties of a system from properties of its components.
- And vice versa.

Invariants

- A state is reachable if it appears in some execution (or, at the end of some finite execution).
- An invariant is a predicate that is true for every reachable state.
- Most important tool for proving properties of concurrent and distributed algorithms.
- Proving invariants:
  - Typically, by induction on length of execution.
  - Often prove batches of interdependent invariants together.
  - Step granularity is finer than round granularity, so proofs are more complicated and detailed than those for synchronous algorithms.

Example: Incrementing

- Two processes, P_1 and P_2, communicating via channels C_{12} and C_{21}: $send(v)_{12}, receive(v)_{12}, send(v)_{21}, receive(v)_{21}$.
- Each process has a local variable val.
- Initially $P_1.val = 1$, $P_2.val = 2$.
- Transitions:
  - $send(v)$, where $v = val$, at any time.
  - When $receive(v)$: $val := v + 1$.
- Invariant 1: $P_1.val$ is odd and $P_2.val$ is even
- Proof: By induction.
  - Base: Yes
  - Inductive step:
    - Cases based on various kinds of send/receive actions.
    - Strengthen invariant?
    - Add that any value in C_{12} is odd, and any value in C_{21} is even.

Example: Incrementing

- Initially $P_1.val = 1, P_2.val = 2$.
- Transitions:
  - $send(v)$, where $v = val$, at any time.
  - When $receive(v)$: $val := v + 1$.
- Invariant 1: $P_1.val$ is odd and $P_2.val$ is even
- Invariant 2: $|P_2.val - P_1.val| \leq 1$
- Proof: By induction.
  - Base: Yes
  - Inductive step:
    - Cases based on various send/receive actions.
    - Strengthen invariant?
    - LTTR.

Trace properties

- A trace property is essentially a set of allowable external behavior sequences.
- Formally, a trace property P is a pair consisting of:
  - $sig(P)$: External signature (no internal actions).
  - $traces(P)$: Set of sequences of actions in $sig(P)$.
- Automaton A satisfies trace property P if $extsig(A) \ = sig(P)$ and (two notions, depending on whether we consider fairness):
  - $traces(A) \subseteq \ traces(P)$
  - $fairtraces(A) \subseteq traces(P)$
- All problems we consider for asynchronous systems can be formulated as trace properties.
- When we care about liveness, we use the second def.

Safety and liveness properties

- Safety property: "Bad" thing doesn't happen:
  - Nonempty (null trace is always safe).
  - Prefix-closed: Every prefix of a safe trace is safe.
  - Limit-closed: Limit of sequence of safe traces is safe.
- Liveness property: "Good" thing happens eventually:
  - Every finite sequence over $acts(P)$ can be extended to a sequence in $traces(P)$.
  - "It's never too late."
- Define safety/liveness for executions similarly.

Safety property S

- traces(S) are nonempty, prefix-closed, limit-closed.
- Examples:
  - 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 MSTs,…
    - Outputs do not yield any incorrect answers.
  - Mutual exclusion: No two grants without intervening return.

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.
- Use invariants:
  - Find an invariant corresponding to the trace safety property.
  - Example: Consensus
    - Record decisions in the state.
    - Express agreement and validity in terms of recorded decisions.
  - Then prove the invariant by induction.

Liveness property L

- Every finite sequence over sig(L) has an extension in 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 logical reasoning using temporal logic.
  - Methods are less agreed-upon than those for safety properties.

Safety and liveness

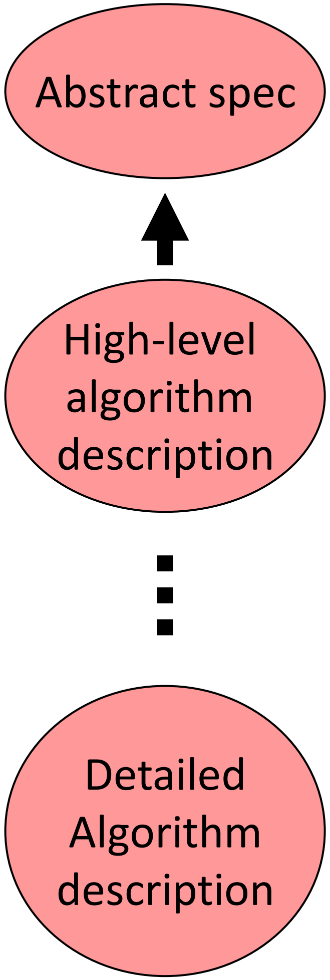
- Theorem: Every trace property can be expressed as the intersection of a safety property and a liveness property.
- So, to define a problem to be solved by an asynchronous system, it's enough to specify safety requirements and liveness requirements separately.
- This explains why typical specifications of problems for asynchronous systems consist of:
  - A list of safety properties.
  - A list of liveness properties.
  - And nothing else.

Automata as specifications

- Every I/O automaton specifies a trace property $(extsig(A), traces(A))$.
- So we can use an automaton as a problem specification.
- Automaton A "implements" automaton B if
  - $extsig(A) = extsig(B)$
  - $traces(A) \subseteq traces(B)$

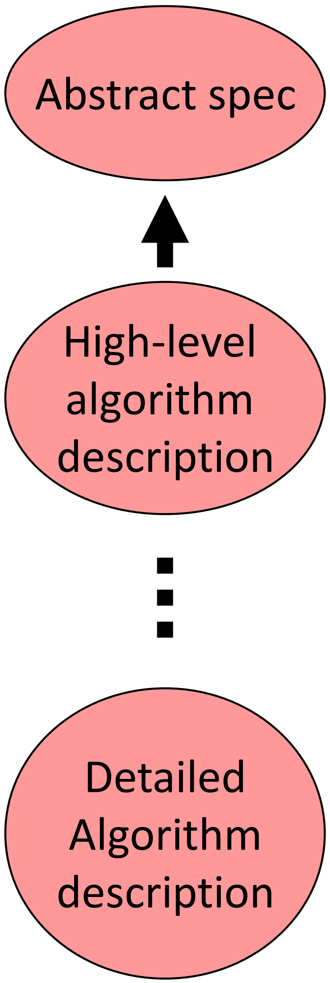
Hierarchical proofs

- Important strategy for proving correctness of complex asynchronous distributed algorithms.
- Define a series of automata, each implementing the previous one (“successive refinement”).
- Highest-level automaton model captures the “real” problem specification.
- Next level is a high-level algorithm description.
- Successive levels represent more and more detailed versions of the algorithm.
- Lowest level is the full algorithm description.



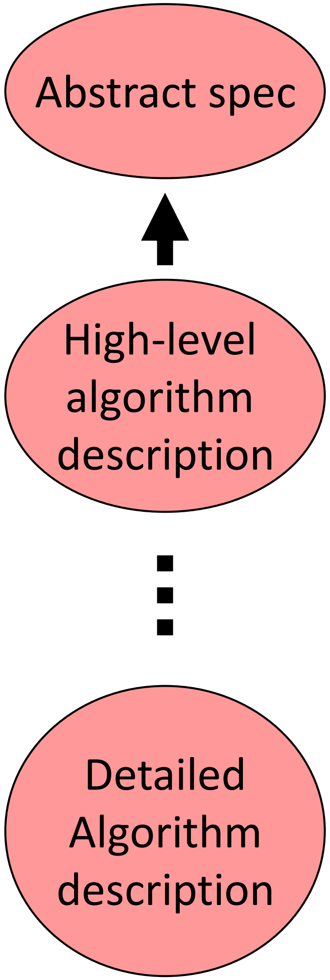
Hierarchical proofs

- For example:
  - 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.
- In all these cases, lower levels are harder to understand and reason about.
- So instead of reasoning about them directly, relate them to higher-level descriptions.
- Method similar to what we saw for synchronous algorithms.



Hierarchical proofs

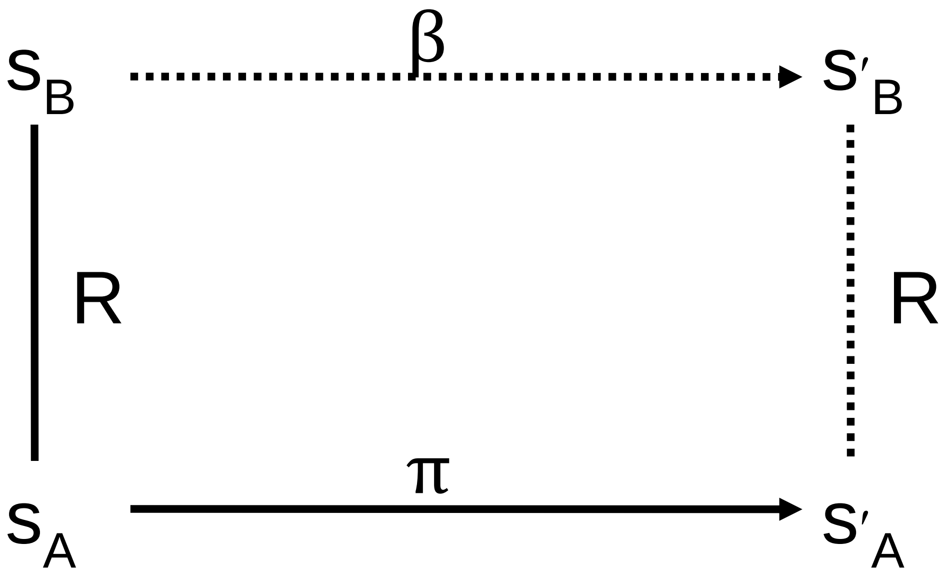
- Recall, for synchronous algorithms:
  - Optimized algorithm runs side-by-side with unoptimized version, and "invariant" proved to relate the states of the two algorithms.
  - Prove using induction.
- For asynchronous systems, it's harder:
  - Asynchronous model has more nondeterminism (in choice of new state, in order of steps).
  - So, it's harder to determine which executions to compare.
- One-way implementation relationship 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: it 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 binary relation from $states(A)$ to $states(B)$.
- Then R is a simulation relation from A to B provided:
  - $s_A \in start(A)$ implies that there exists $s_B \in start(B)$ such that $s_A\ R\ s_B$.
  - If s_A, s_B are reachable states of A and B respectively, $s_A\ R\ s_B$ and (s_A, π, s'_A) is a step of A, then there is an execution fragment β of B, starting with s_B and ending with s'_B such that $s'_A\ R\ s'_B$ and $trace(\beta) = trace(\pi)$.

Simulation relations



- R is a simulation relation from A to B provided:
  - $s_A \in start(A)$ implies that there exists $s_B \in start(B)$ 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, π, s'_A) is a step, then there is an execution fragment β starting with s_B and ending with s'_B such that $s'_A \; R \; s'_B$ and $trace(\beta) = trace(\pi)$.

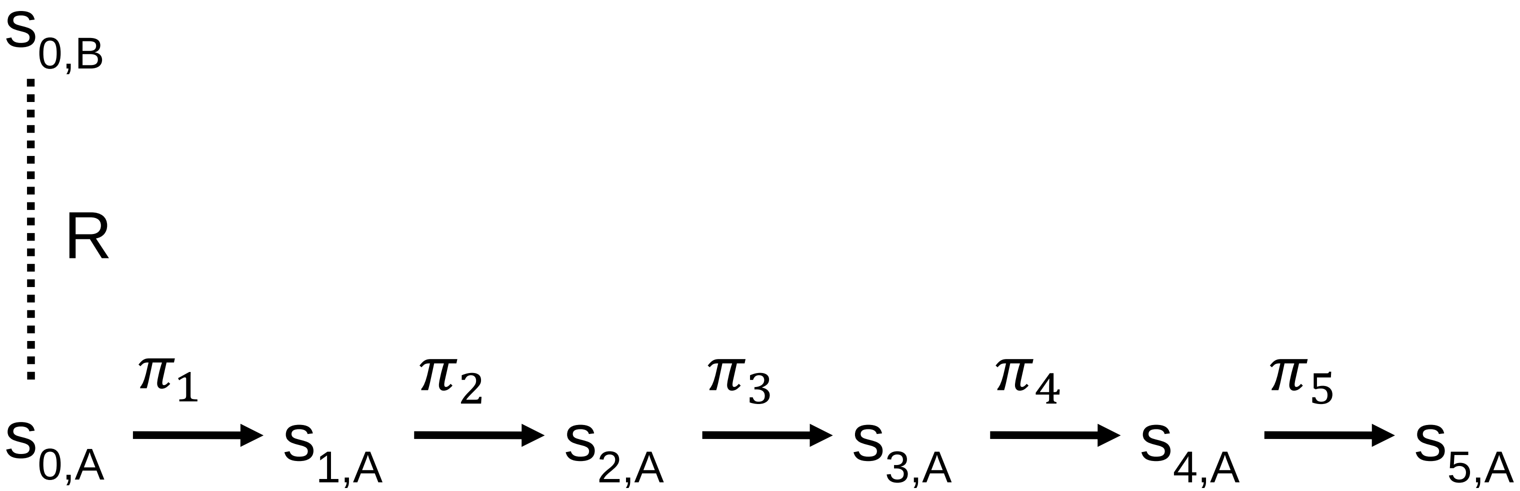
Simulation relations

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



Simulation relations

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

$$
\begin{tikzcd}
s_{0,B} \arrow[r, dotted, "\beta_1"] \arrow[d, dash, "R"] & s_{1,B} \arrow[d, dash, dotted, "R"] & & & & \\
s_{0,A} \arrow[r, "\pi_1"] & s_{1,A} \arrow[r, "\pi_2"] & s_{2,A} \arrow[r, "\pi_3"] & s_{3,A} \arrow[r, "\pi_4"] & s_{4,A} \arrow[r, "\pi_5"] & s_{5,A}
\end{tikzcd}
$$

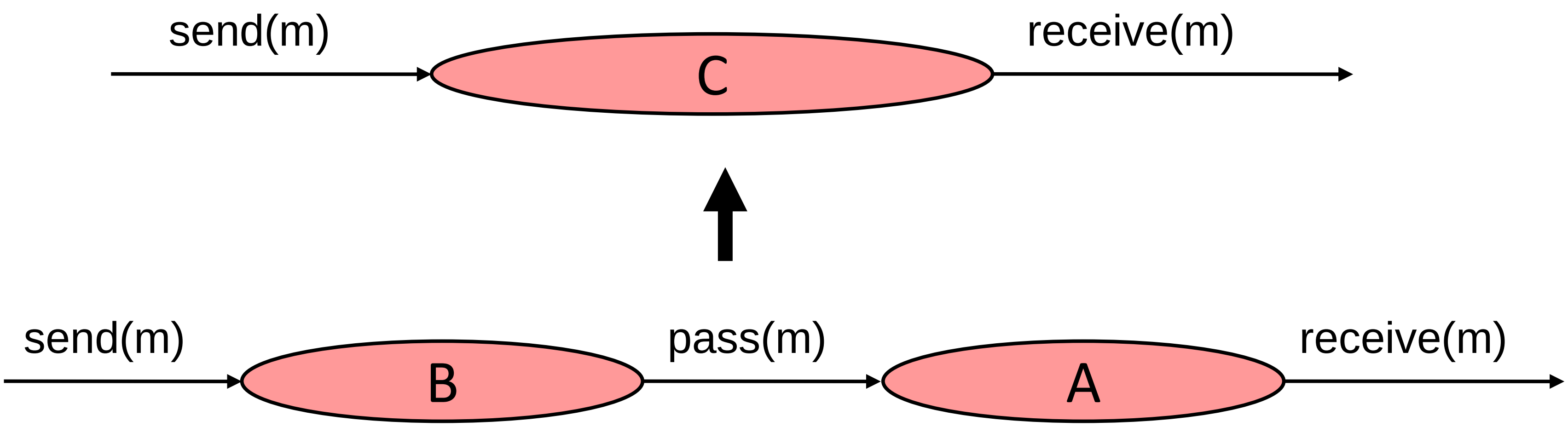
Simulation relations

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

$$\begin{tikzcd}
s_{0,B} \arrow[r, dotted, "\beta_1"] \arrow[d, no head, "R"] & s_{1,B} \arrow[r, dotted, "\beta_2"] \arrow[d, no head, "R"] & s_{2,B} \arrow[r, dotted, "\beta_3"] \arrow[d, no head, dotted, "R"] & s_{3,B} \arrow[r, dotted, "\beta_4"] \arrow[d, no head, dotted, "R"] & s_{4,B} \arrow[r, dotted, "\beta_5"] \arrow[d, no head, dotted, "R"] & s_{5,B} \arrow[d, no head, dotted, "R"] \\
s_{0,A} \arrow[r, "\pi_1"] & s_{1,A} \arrow[r, "\pi_2"] & s_{2,A} \arrow[r, "\pi_3"] & s_{3,A} \arrow[r, "\pi_4"] & s_{4,A} \arrow[r, "\pi_5"] & s_{5,A}
\end{tikzcd}$$

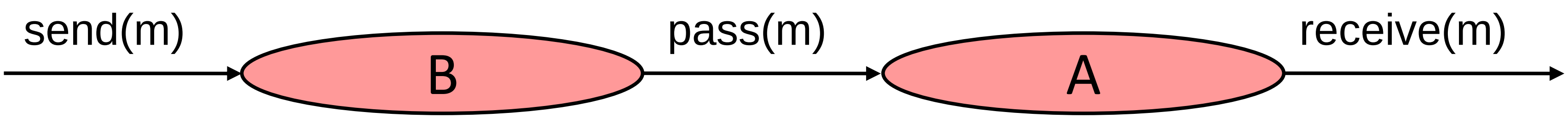
Example: Channels

- Show two channels implement one.



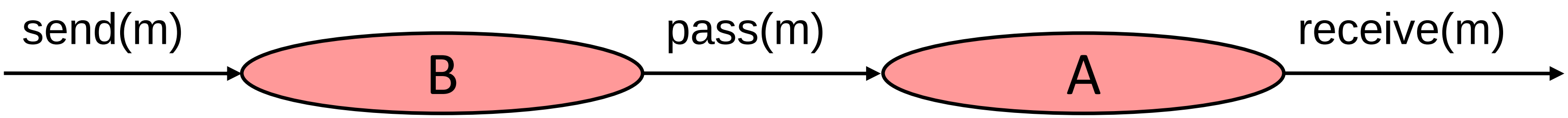
- Rename $receive(m)$ of B and $send(m)$ of A to $pass(m)$.
- Let $D = hide_{\{pass(m)\}}\,(A \times B)$.
- Show that $traces(D) \subseteq traces(C)$.

Two channels implement one



- Let $D = hide_{\{pass(m)\}}\ (A \times B)$.
- Show that $traces(D) \subseteq traces(C)$.
- Define relation R: For $s \in states(D)$ and $u \in states(C)$, define:
  - $s\ R\ u$ iff $u.queue$ is the concatenation of $s.A.queue$ and $s.B.queue$.
- Prove that R is a simulation relation:
  - Start condition: All queues are empty, so start states correspond.
  - Step condition: Define "step correspondence":

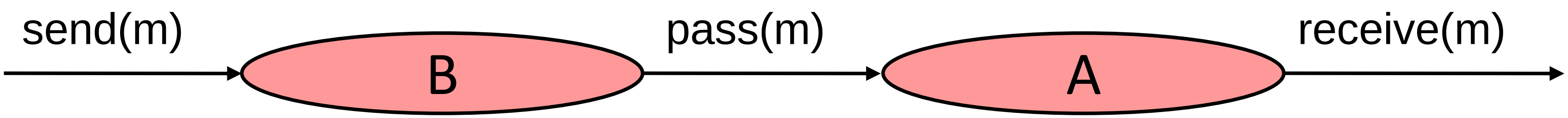
Two channels implement one



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

- Step correspondence:
  - For each step $(s, \pi, s') \in trans(D)$ and u such that $s\ R\ u$, define execution fragment β of C:
    - Starts with u, ends with u' such that $s'\ R\ u'$.
    - trace(β) = trace(π)
  - Here, actions in β depend only on π, and uniquely determine the states.
    - Same action if external, empty sequence if internal.

Two channels implement one



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

- Step correspondence:
  - $\pi = send(m)$ in D corresponds to $send(m)$ in C
  - $\pi = receive(m)$ in D corresponds to $receive(m)$ in C
  - $\pi = pass(m)$ in D corresponds to λ in C
- Verify that this works:
  - Same external actions (yes).
  - Actions of C are enabled.
  - Final states related by relation R.

– Routine case analysis:

Showing R is a simulation relation

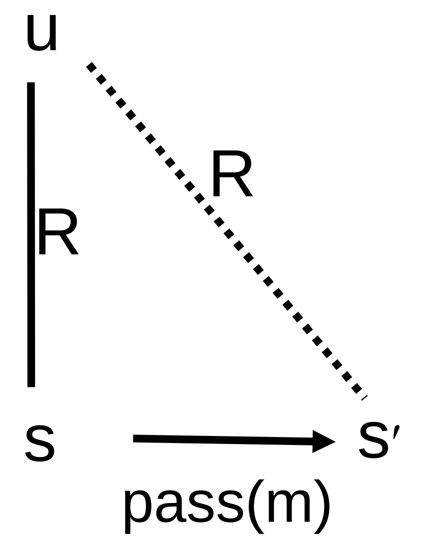
s R u iff u.queue is concatenation of s.A.queue and s.B.queue

- Case 1: $\pi = send(m)$
  - No enabling issues (input).
  - Must check that $s' \ R \ u'$.
    - Since $s \ R \ u$, $u.queue$ is the concatenation of $s.A.queue$ and $s.B.queue$.
    - Adding the same m to the end of $u.queue$ and $s.B.queue$ maintains the correspondence.
- Case 2: $\pi = receive(m)$
  - Enabling: Check that $receive(m)$, for the same m, is also enabled in u.
    - We know that m is first on $s.A.queue$.
    - Since $s \ R \ u$, m is also first on $u.queue$.
    - So $receive(m)$ is enabled in u.
  - $s' \ R \ u'$: Since m is removed from both $s.A.queue$ and $u.queue$.

Showing R is a simulation relation

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

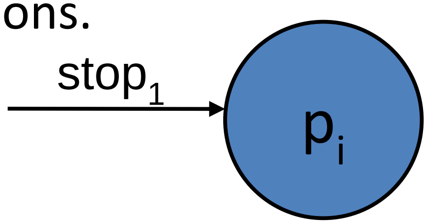
- Case 3: $\pi = \mathit{pass}(m)$
  - No enabling issues (since no high-level steps are involved).
  - Must check $s' \, R \, u$:
    - Since $s \, R \, u$, $u.\mathit{queue}$ is the concatenation of $s.A.\mathit{queue}$ and $s.B.\mathit{queue}$.
    - The concatenation of the queues is unchanged as a result of this step, so also $u.\mathit{queue}$ is the concatenation of $s'.A.\mathit{queue}$ and $s'.B.\mathit{queue}$.

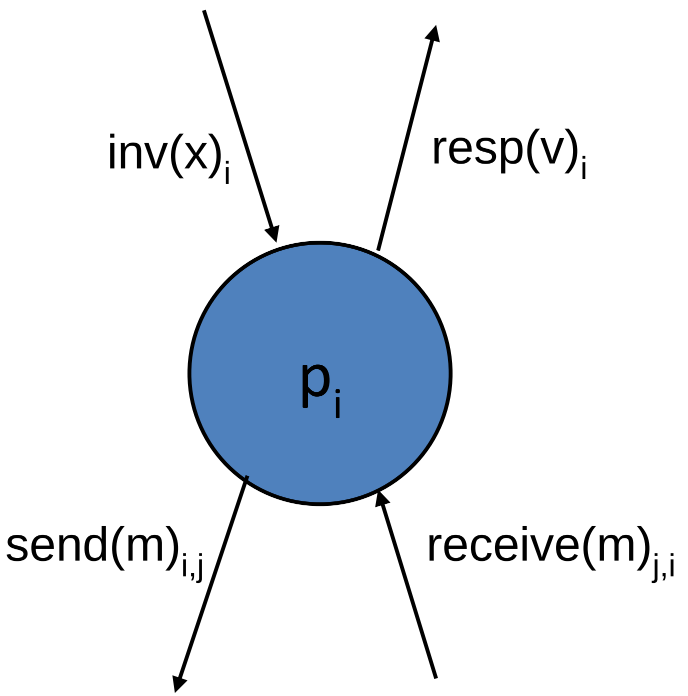


Asynchronous network model

Send/Receive System

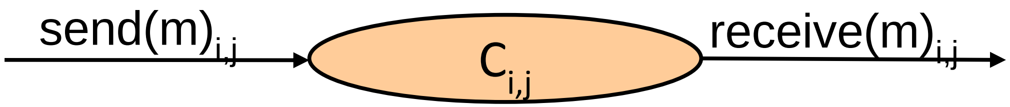
- Digraph $G = (V, E)$, with:
  - Process I/O automata associated with nodes, and
  - Channel I/O automata associated with directed edges.
- Compose all the automata to get a system automaton.
- Process:
  - User interface actions, e.g., invocations and responses
  - Send/receive actions for interaction with channels
- Problems specified in terms of allowable traces at the user interface.
  - Hide send/receive actions.
- Failure modeling:





- Having explicit *stop* actions in the external interface allows us to state requirements involving failures.

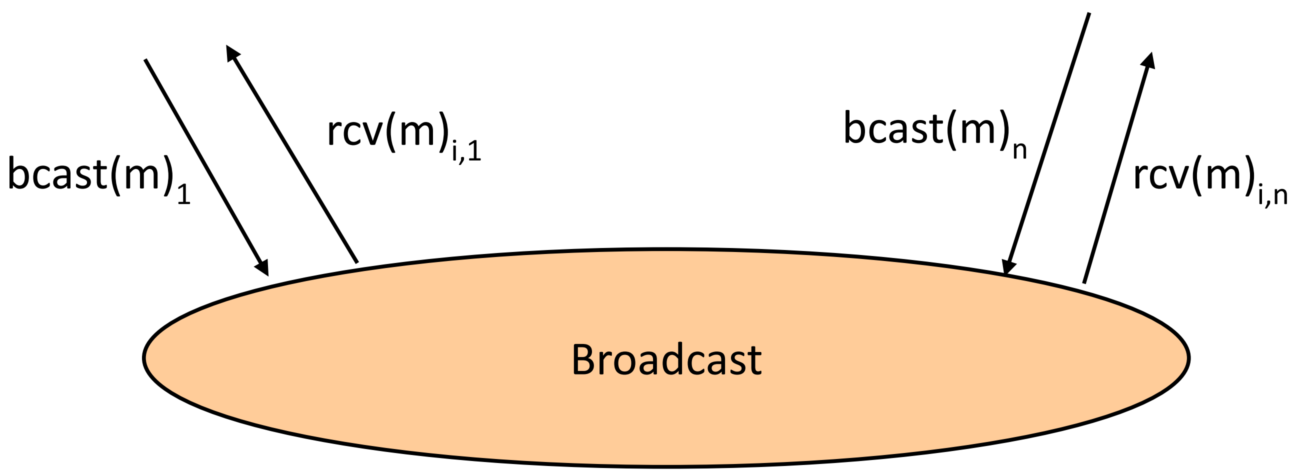
Channel automata



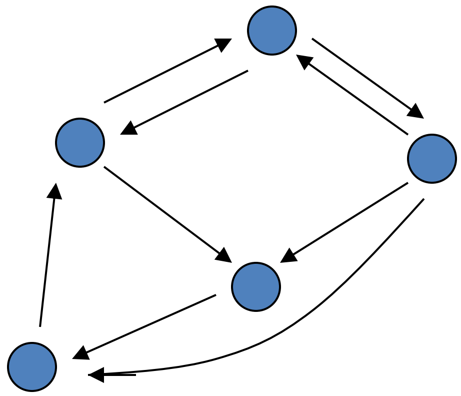
- Consider different kinds of channels with this interface:
  - Reliable FIFO, as before.
  - Weaker guarantees: Lossy, duplicating, reordering
- Can define channels by trace properties, using a *cause* function mapping receives to sends.
  - Integrity: The *cause* function preserves the message.
  - 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 conditions; weaker channels satisfy Integrity but may weaken some of the other properties.

Broadcast and multicast systems

- Broadcast
  - Reliable FIFO between each pair.
  - Different processes can receive messages from different senders in different orders.
  - Model abstractly using separate queues for each pair.
- Multicast: Processes designate recipients.



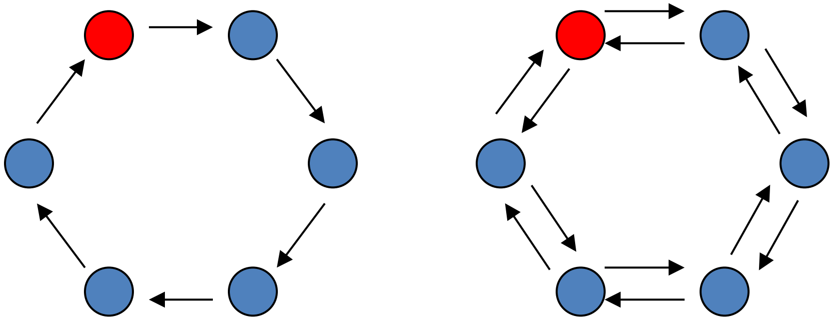
Asynchronous network algorithms



Asynchronous network algorithms

- Consider send/receive systems with reliable FIFO point-to-point channels
- Revisit problems we considered in synchronous networks:
  - Leader election, in a ring, and in general undirected networks.
  - Spanning tree
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree
- What results carry over?
- Where did we use the synchrony assumption?

Algorithms for Leader Election in a Ring



Leader election in a ring

- Assumptions:
  - G is a ring, with unidirectional or bidirectional communication
  - Local names for neighbors, UIDs
- $AsynchLCR$ [LeLann] [Chang-Roberts]
  - Send UID clockwise around ring (unidirectional).
  - Discard UIDs smaller than your own.
  - Elect yourself if your UID comes back.
- Correctness: Basically the same as for synchronous version, with a few complications:
  - Finer granularity, must consider individual steps rather than entire rounds.
  - Must consider messages in channels.

$AsynchLCR$, process i

- Signature
  - ***in*** $rcv(v)_{i-1,i}$, v a UID
  - ***out*** $send(v)_{i,i+1}$, v a UID
  - ***out*** $leader_i$
- State variables
  - u, a UID, initially i's UID
  - send, a FIFO queue of UIDs, initially containing i's UID
  - status, unknown, chosen, or reported, initially unknown
- Tasks
  - { $send(v)_{i,i+1}$ | v is a UID }
  - { $leader_i$ }

Transitions

- $send(v)_{i,i+1}$
  pre: v = head(send)
  eff: remove head of send
- $receive(v)_{i-1,i}$
  eff:
  if v = u then status := chosen
  if v > u then add v to send
- $leader_i$
  pre: status = chosen
  eff: status := reported

$AsynchLCR$ properties

- Safety: No process $i \neq i_{max}$ ever performs $leader_i$.
- Liveness: i_{max} eventually performs $leader_{imax}$.

Safety proof

- Safety: No process $i \neq i_{max}$ ever performs $leader_i$.
- Recall the synchronous proof, based on showing an invariant of global states: After any number of rounds:
  - If $i \neq i_{max}$ and $j \in [i_{max}, i)$ then u_i not in $send_j$.
- We can use a similar invariant for the asynchronous algorithm:
  - If $i \neq i_{max}$ and $j \in [i_{max}, i)$ then u_i not in $send_j$ or in $queue_{j,j+1}$.
- The main difference is that now the invariant must hold after any number of steps.
- Prove this by induction on number of steps.
  - Use cases based on the type of action.
  - Key case: $receive(v)_{imax-1,\ imax}$
    - Argue that if $v \neq u_{max}$ then v gets discarded.

Liveness proof

- Liveness: i_{max} eventually performs $leader_{imax}$.
- Synchronous proof used an invariant saying exactly where the max is after r rounds.
- Now we don't have rounds, so we need a different proof.
- Can establish intermediate milestones, e.g.:
  - For $k \in [0, n-1]$, u_{max} is eventually in $send_{\text{imax+k}}$
  - Prove by induction on k; use fairness for a process and a channel to prove the inductive step.

Complexity

- Messages: $O(n^2)$, 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 the first message in each channel (that is, for each channel task).
  - Measuring real time here (not counting rounds).
  - Only upper bounds, so this 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:
  - Uses bidirectional communication.
  - Send in both directions, to successively doubled distances.
  - Extends immediately to the asynchronous model.
  - $O(n \log n)$ messages.
- Peterson:
  - Unidirectional communication
  - $O(n \log\ n)$ messages
  - 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 there are 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 larger than the other two.
  - In this case, adopt the middle UID.
  - 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 before the phase remain active.
- Termination:
  - If a process sees that its immediate predecessor's UID is the same as its own, it elects itself the leader (knows it's the only active process left).

PetersonLeader

- Signature
  - ***in*** $receive(v)_{i-1,i}$, v a UID
  - ***out*** $send(v)_{i,i+1}$, v a UID
  - ***out*** $leader_i$
  - ***int*** $get\text{-}second\text{-}uid_i$
  - ***int*** $get\text{-}third\text{-}uid_i$
  - ***int*** $advance\text{-}phase_i$
  - ***int*** $become\text{-}relay_i$
  - ***int*** $relay_i$
- State variables
  - mode: active or relay, initially active
  - status: unknown, chosen, or reported, initially unknown
  - uid1, initially i's UID
  - uid2, initially null
  - uid3, initially null
  - send, FIFO queue of UIDs; initially contains just i's UID
  - receive: FIFO queue of UIDs, initially empty

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

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

- advance-phase$_i$
  pre: mode = active
  uid3 ≠ null
  uid2 > max(uid1, uid3)
  eff: uid1 := uid2
  uid2, uid3 := null
  add uid1 to send

- become-relay$_i$
  pre: mode = active
  uid3 ≠ null
  uid2 ≤ max(uid1, uid3)
  eff: mode := relay

- relay$_i$
  pre: mode = relay
  receive is nonempty
  eff: move head(receive) to send

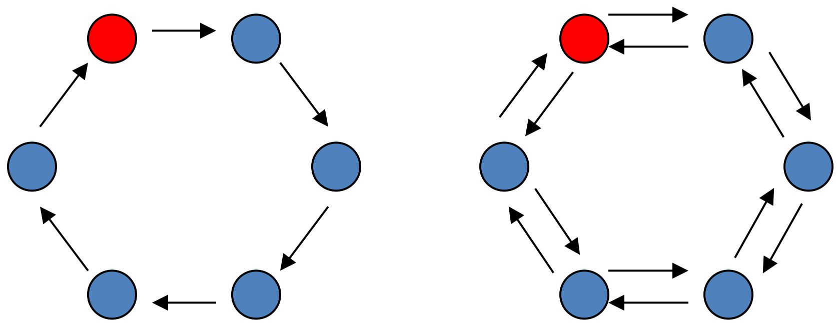
PetersonLeader

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

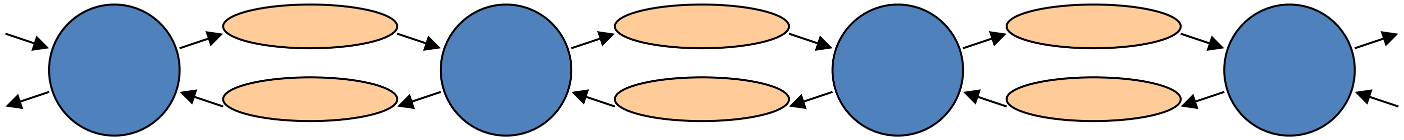
- Q: Can we do better than $O(n \log n)$ message complexity?
  - Not with comparison-based algorithms. (Why?)
  - Not at all:
    - Can prove another lower bound.
    - This one depends on asynchrony.

Lower Bound for Leader Election in a Ring



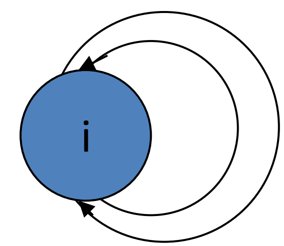
$\Omega(n \log n)$ lower bound

- Lower bound for leader election in asynchronous ring 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 are 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 at the ends.
  - Ring looks like a line if communication is delayed across a boundary.

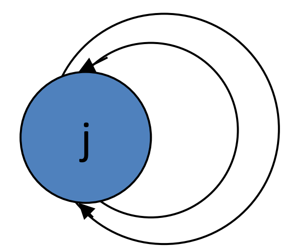


$\Omega(n \log n)$ lower bound

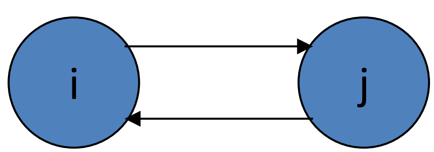
- Lemma 1: There are infinitely many process automata, each of which can send at least one message without first receiving one (in some execution).
- Proof:
  - If not, then there are two processes i, j, neither of which sends a message without first receiving one.
  - Consider 1-node ring:
    - i must elect itself, with no messages sent or received.



  - Consider another 1-node ring:
    - j must elect itself, with no messages sent or received.

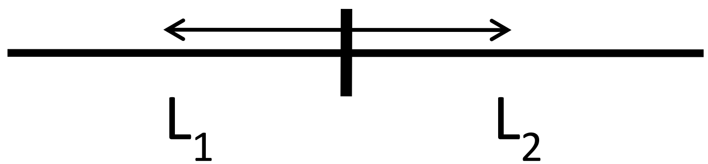


  - Now consider:
    - Both i and j elect themselves, contradiction.



$\Omega(n \log n)$ lower bound

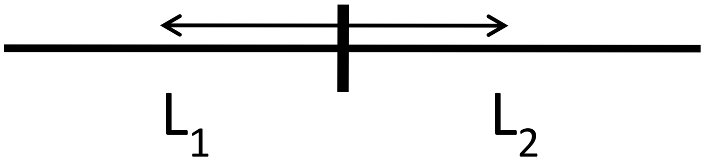
- $C(L)$ = maximum (actually, supremum) of the number of messages that are sent in a single input-free execution of line L.
- Lemma 2: If L_1, L_2, L_3 are three line graphs of even length l such that each $C(L_i) \geq k$, then $C(L_i\ join\ L_j) \geq 2k + l/2$ for some $i, j, i \neq j$.
- Proof:



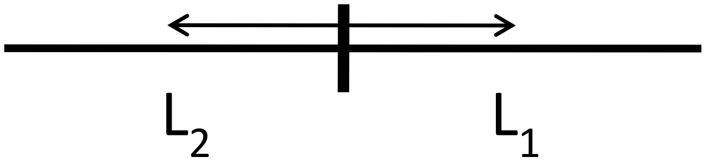
  - Suppose not.
  - Construct an execution $\alpha_{1,2}$ of $L_1\ join\ L_2$:
  - Let α_i be a finite execution of L_i with $\geq k$ messages, $i = 1, 2$.
  - Run α_1 then α_2 then continue to a quiescent state (no more messages),
  - May involve delivering delayed messages across the join boundary.
  - By assumption, $< l/2$ additional messages are sent in $\alpha_{1,2}$.
  - So, execution $\alpha_{1,2}$ quiesces before the effects of the new inputs can cross the middle edge of L_1 or the middle edge of L_2.

$\Omega(n \log n)$ 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 L_1, L_2, L_3 are three line graphs of even length l such that each $C(L_i) \geq k$, then $C(L_i\ join\ L_j) \geq 2k + l/2$ for some i, j.
- Proof, cont'd:
  - Execution $\alpha_{1,2}$ quiesces before the effects of the new inputs can cross the middle edge of L_1 or L_2.



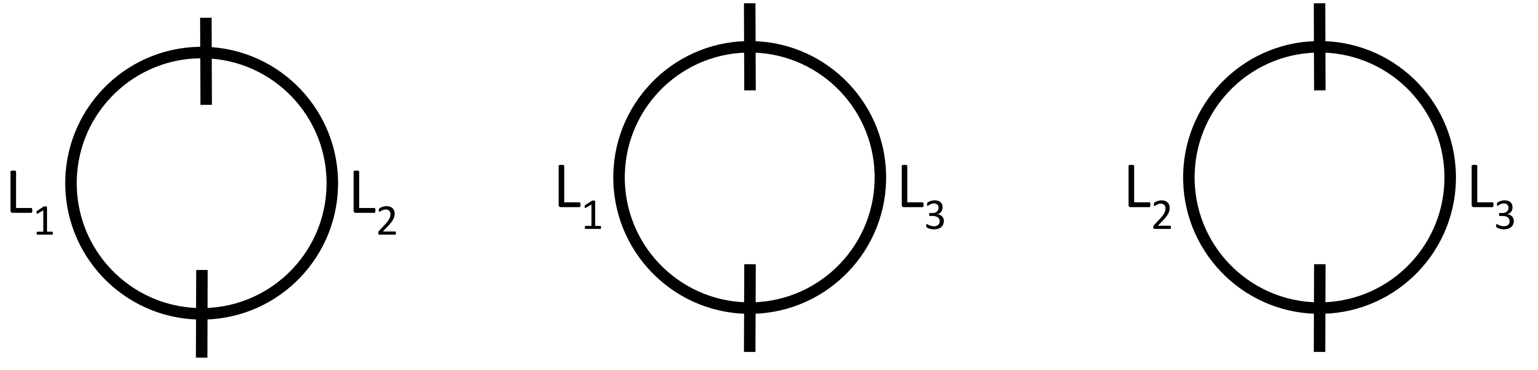
  - Similarly, construct $\alpha_{2,1}$, an execution of $L_2\ join\ L_1$.



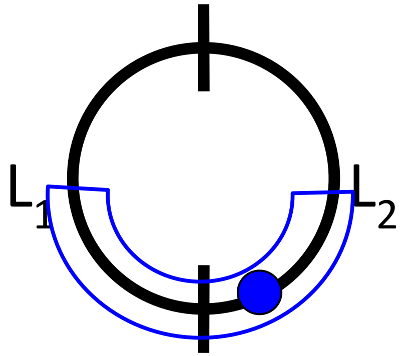
  - Execution $\alpha_{2,1}$ quiesces before the effects of the new inputs can cross the middle edge of L_1 or L_2.

Proof of Lemma 2

- Now consider three rings:

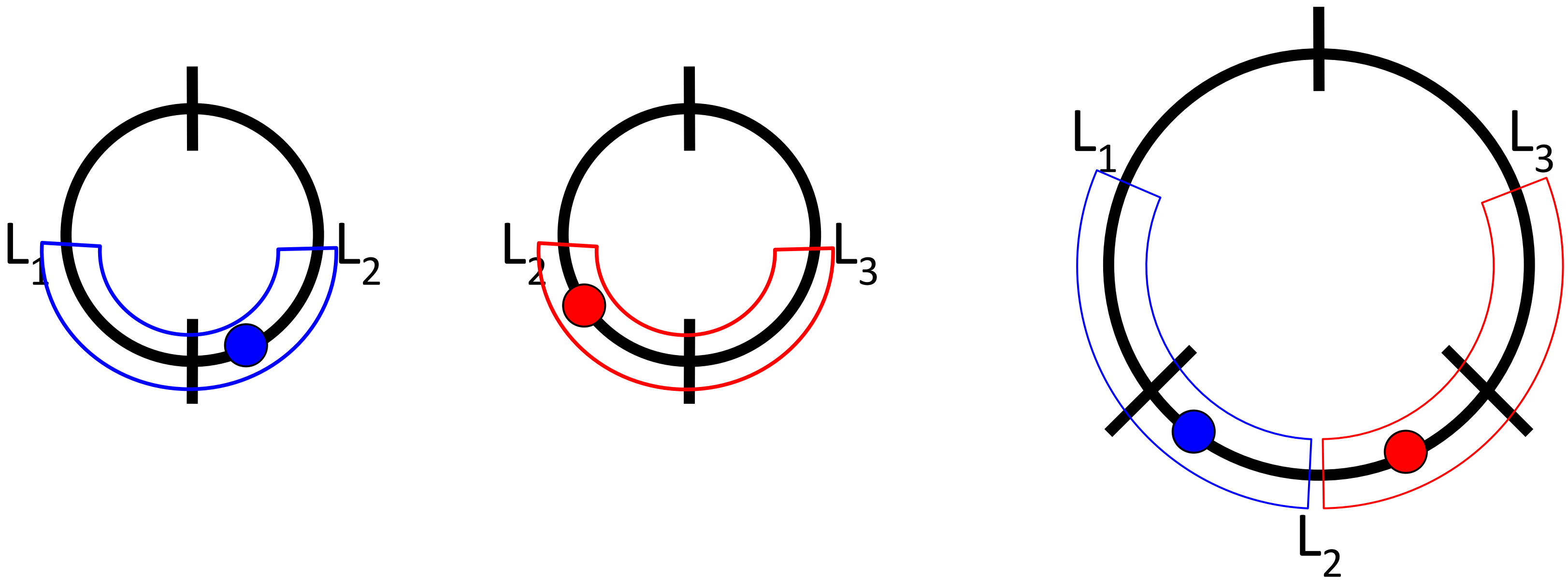


Proof of Lemma 2

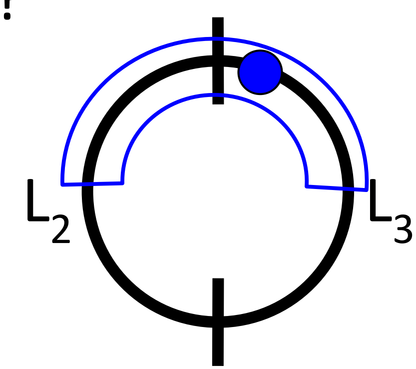


- Connect both ends of L_1 and L_2.
  - Right neighbor in line is clockwise around ring.
- Run α_1, then α_2, then finish $\alpha_{1,2}$, then finish $\alpha_{2,1}$.
  - No interference between the last parts of $\alpha_{1,2}$ and $\alpha_{2,1}$.
  - Quiesces: Eventually no more messages are sent.
  - Must eventually elect a leader.
- Assume WLOG that elected leader is in the "bottom half".

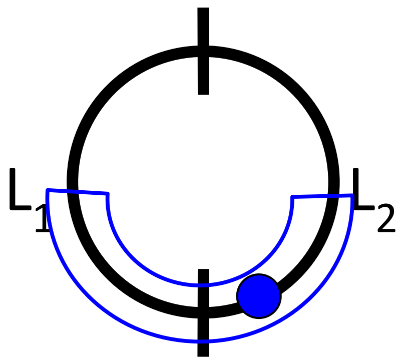
Proof of Lemma 2

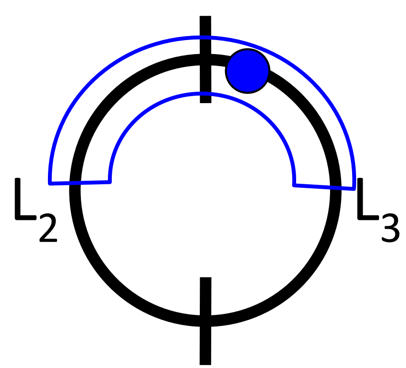


- Same argument for ring constructed from L_2 and L_3.
- Can the leader be in the bottom half?
- No!
- So it must be in top half.

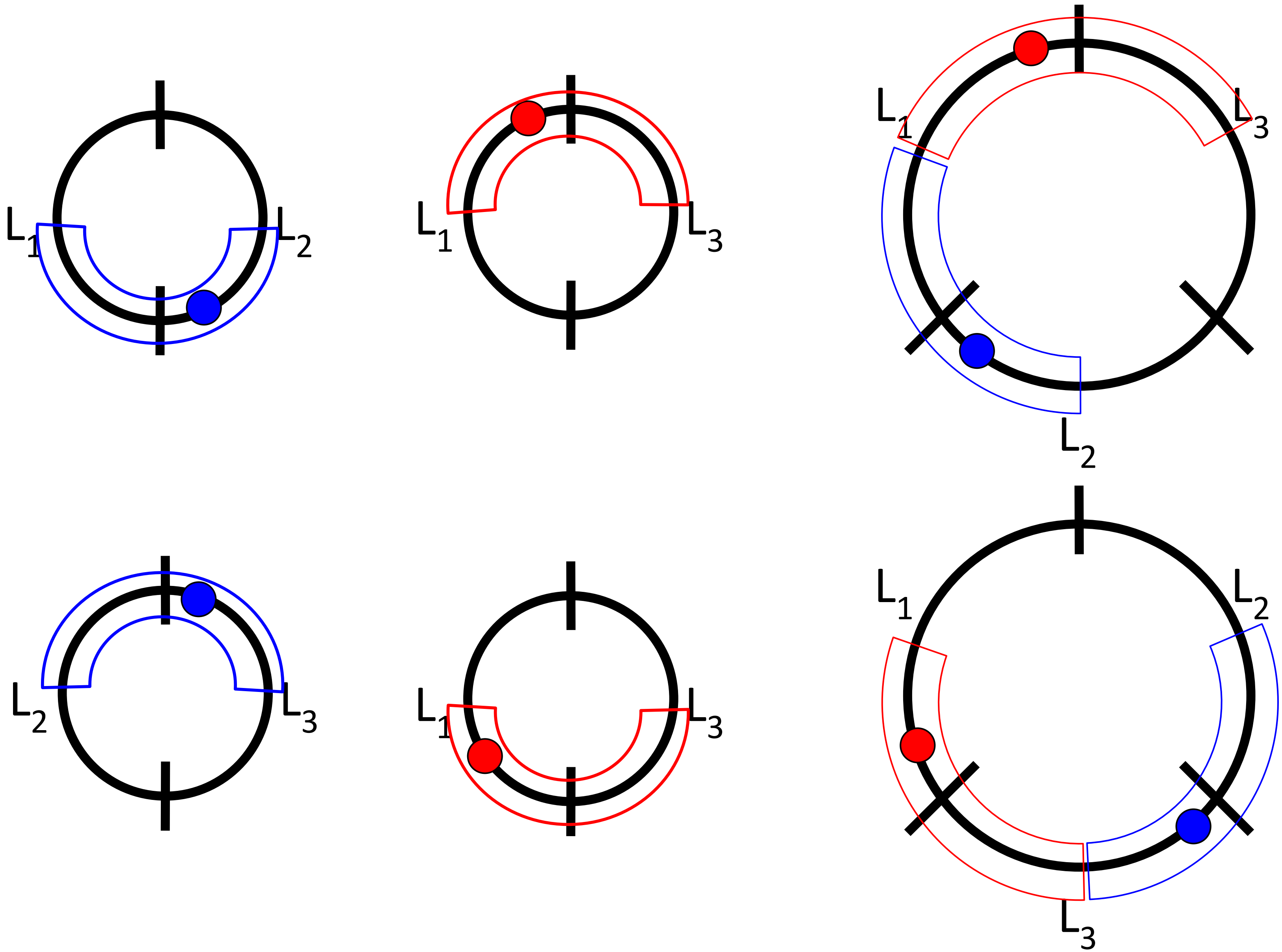


Proof of Lemma 2





Proof of Lemma 2



Lower bound, cont'd

- Summarizing:
- 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 each $C(L_i) \geq k$, then $C(L_i\ join\ L_j) \geq 2k + l/2$ for some $i \neq j$.
- Combining, we get:
- Lemma 3: For any $r \geq 0$, there are infinitely many disjoint line graphs L of length 2^r such that $C(L) \geq r\ 2^{r-2}$.
- Proof: Induction on r.
  - 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 $C(L_i) \geq (r-1)\ 2^{r-3}$.
    - By Lemma 2, for some i, j, $C(L_i\ join\ L_j) \geq 2(r-1)2^{r-3} + 2^{r-1}/2 = r\ 2^{r-2}$.

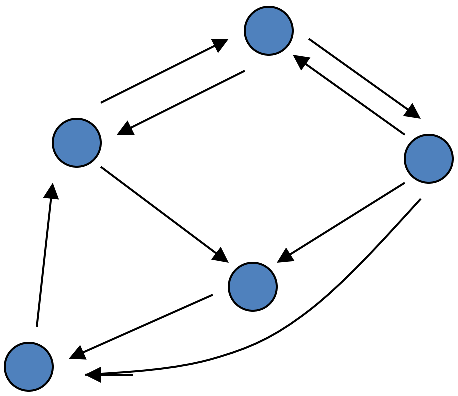
Lower bound, cont'd

- 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.

- Theorem can be extended to non-powers of 2. LTTR.

Leader Election in General Networks



Leader election in general networks

- Consider undirected graphs.
- We can get an asynchronous version of the synchronous $FloodMax$ algorithm:
  - Simulate rounds with local counters.
  - Need to know the diameter for termination.
- We'll see several 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 algorithms
  - Consistent global snapshots to detect termination

Spanning Trees and Searching

Spanning trees and searching

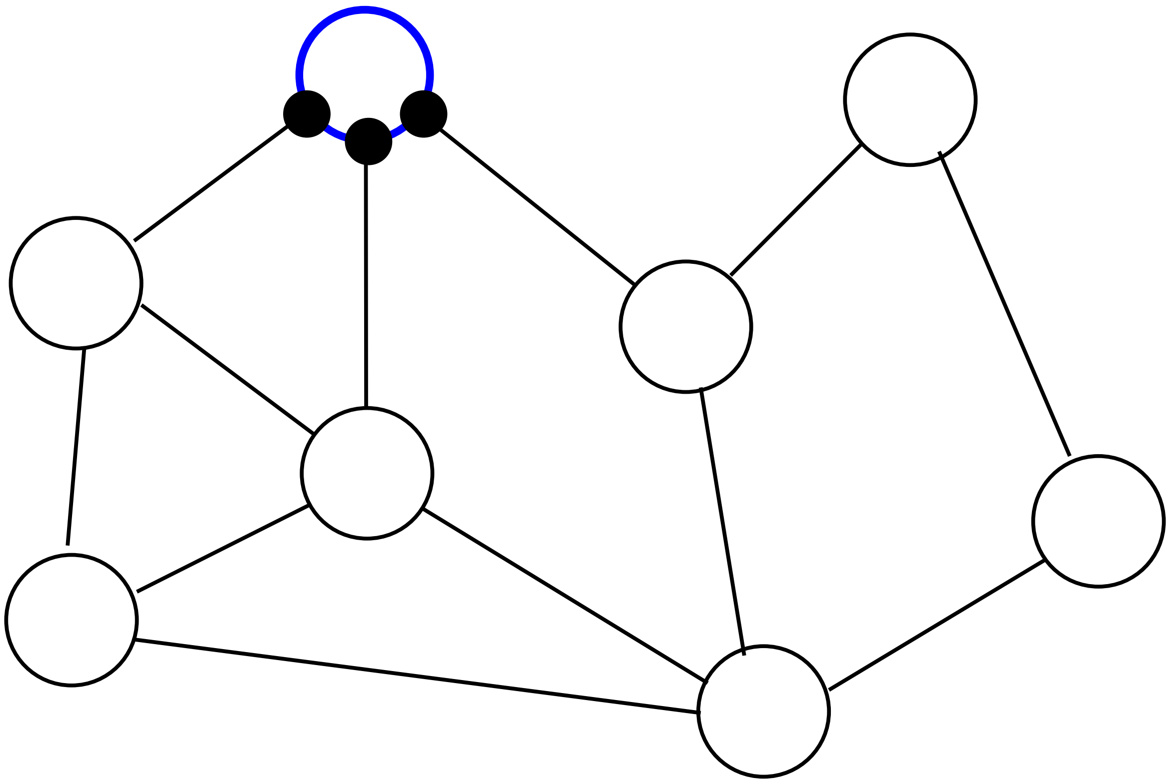
- Spanning trees are used for communication, e.g., bcast/ccast
- Start with the simple task of setting up some (arbitrary) spanning tree with a (given) root i_0.
- Assume:
  - Undirected, connected graph (i.e., bidirectional communication).
  - Root i_0
  - Size and diameter unknown.
  - UIDs, with comparisons for equality.
  - Can identify in- and out-edges to same neighbor.
- Require: Each process should output its parent in tree, with a $parent$ output action.
- Starting point: $SynchBFS$ algorithm:
  - i_0 floods a $search$ message; parent of a node is the first node from which it receives a $search$ message.
  - Q: What if we try to run the same algorithm in an asynchronous network?
  - Still yields a spanning tree, but not necessarily a breadth-first tree.

$AsynchSpanningTree$, Process i

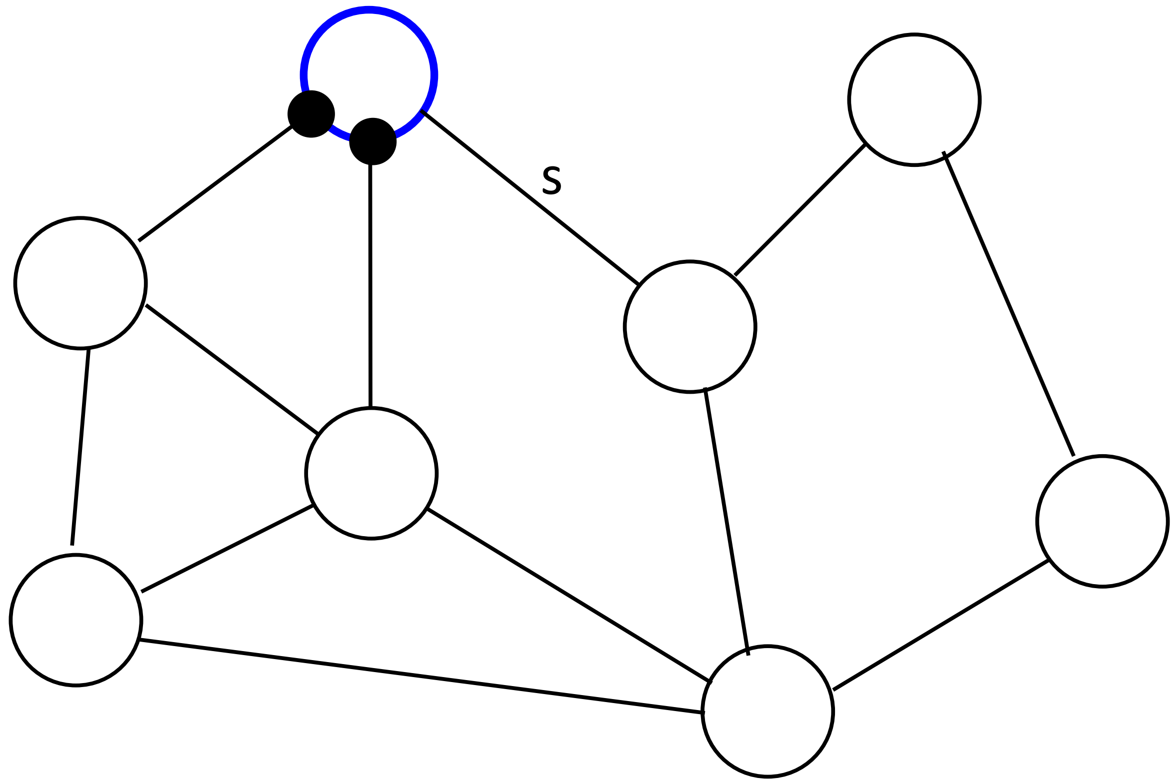
- Signature
  - ***in*** receive(search)$_{j,i}$, j ∈ nbrs
  - ***out*** send(search)$_{i,j}$, j ∈ nbrs
  - ***out*** parent(j)$_i$, j ∈ nbrs
- State
  - parent: nbrs U { ⊥ }, init ⊥
  - reported: Boolean, init false
  - for each j ∈ nbrs:
    - send(j) ∈ {search, ⊥},
      init search if i = i_0, else ⊥

- send(search)$_{i,j}$
  pre: send(j) = search
  eff: send(j) := ⊥
- receive(search)$_{j,i}$
  eff: if i ≠ i_0 and parent = ⊥ then
  parent := j
  for k ∈ nbrs - { j } do
  send(k) := search
- parent(j)$_i$
  pre: parent = j
  reported = false
  eff: reported := true

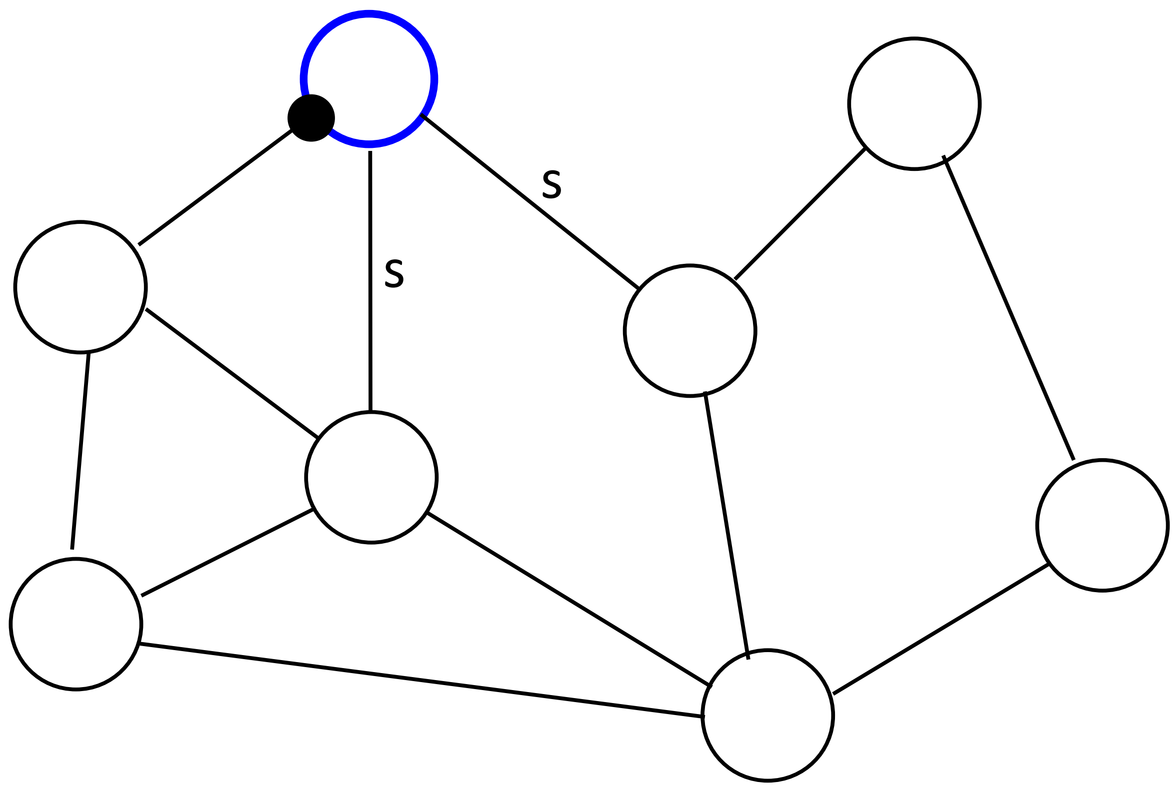
AsynchSpanningTree



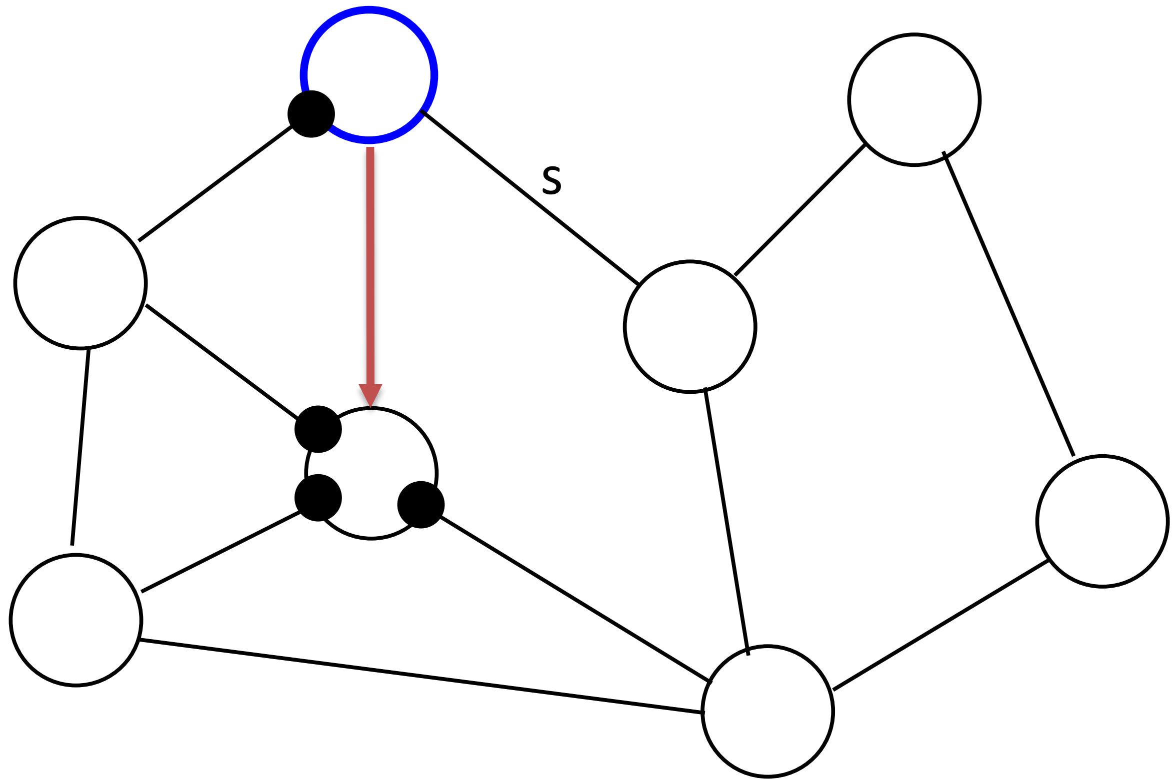
AsynchSpanningTree



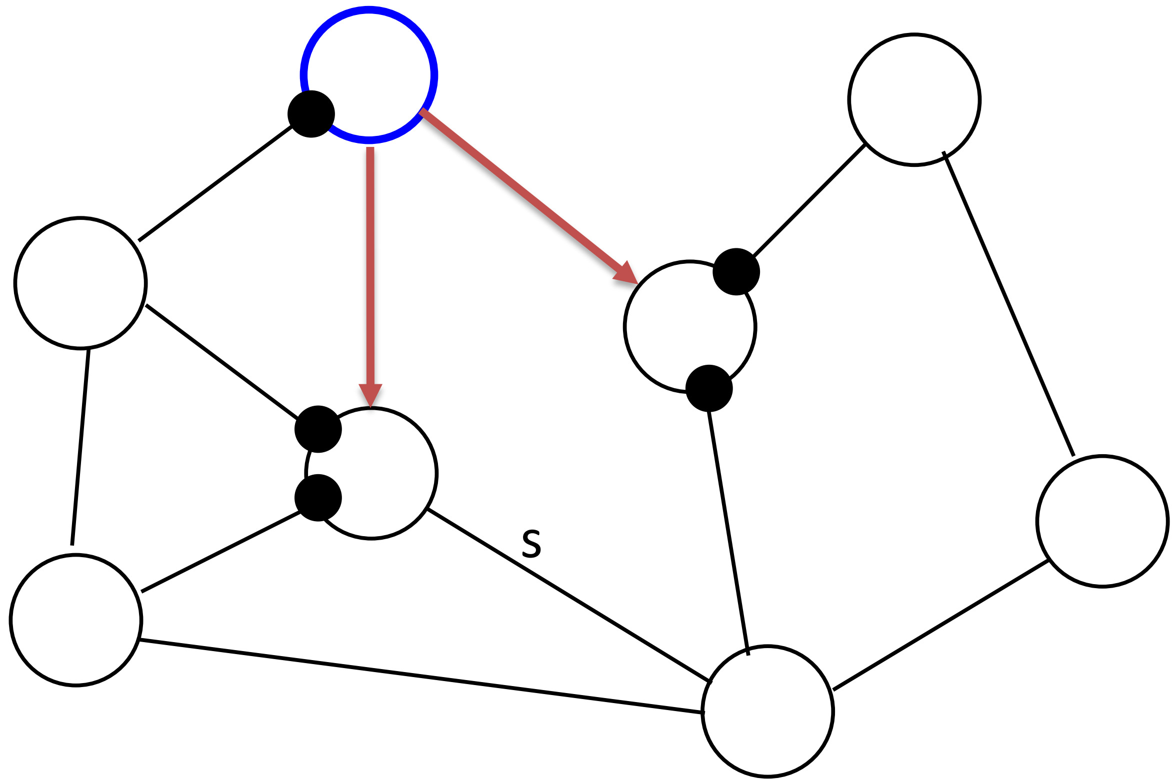
AsynchSpanningTree



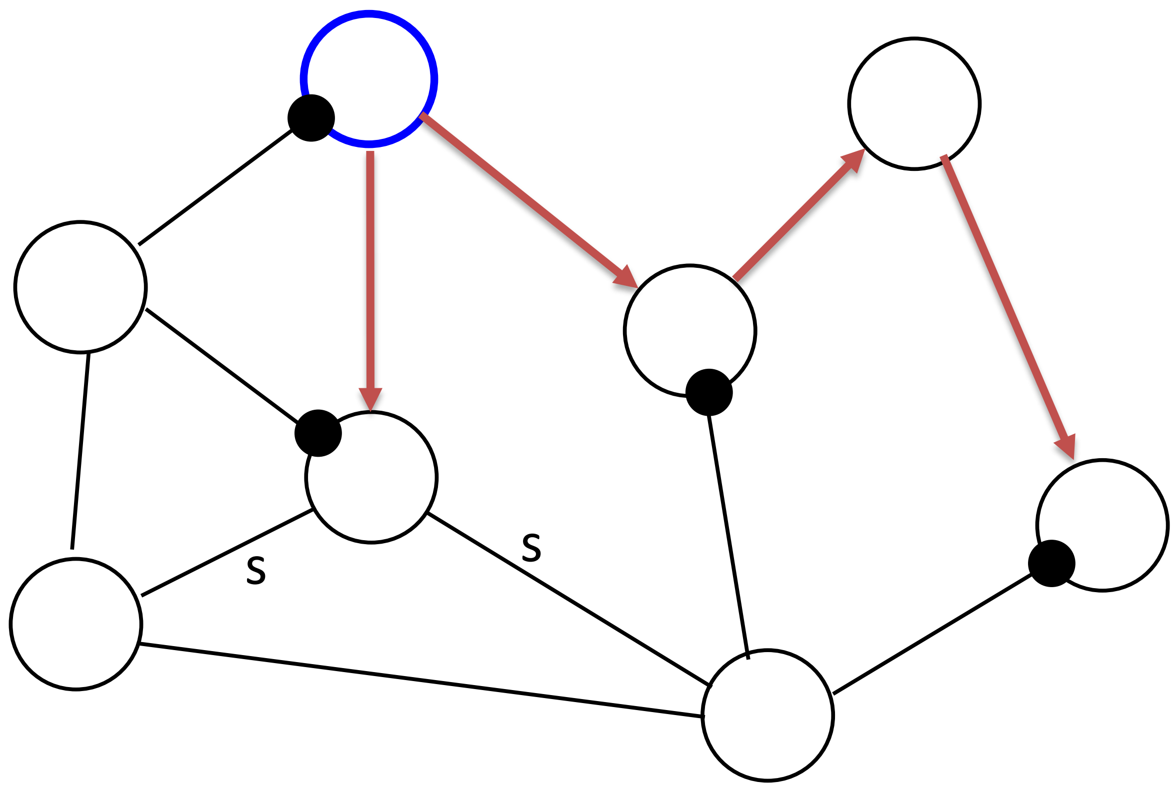
AsynchSpanningTree



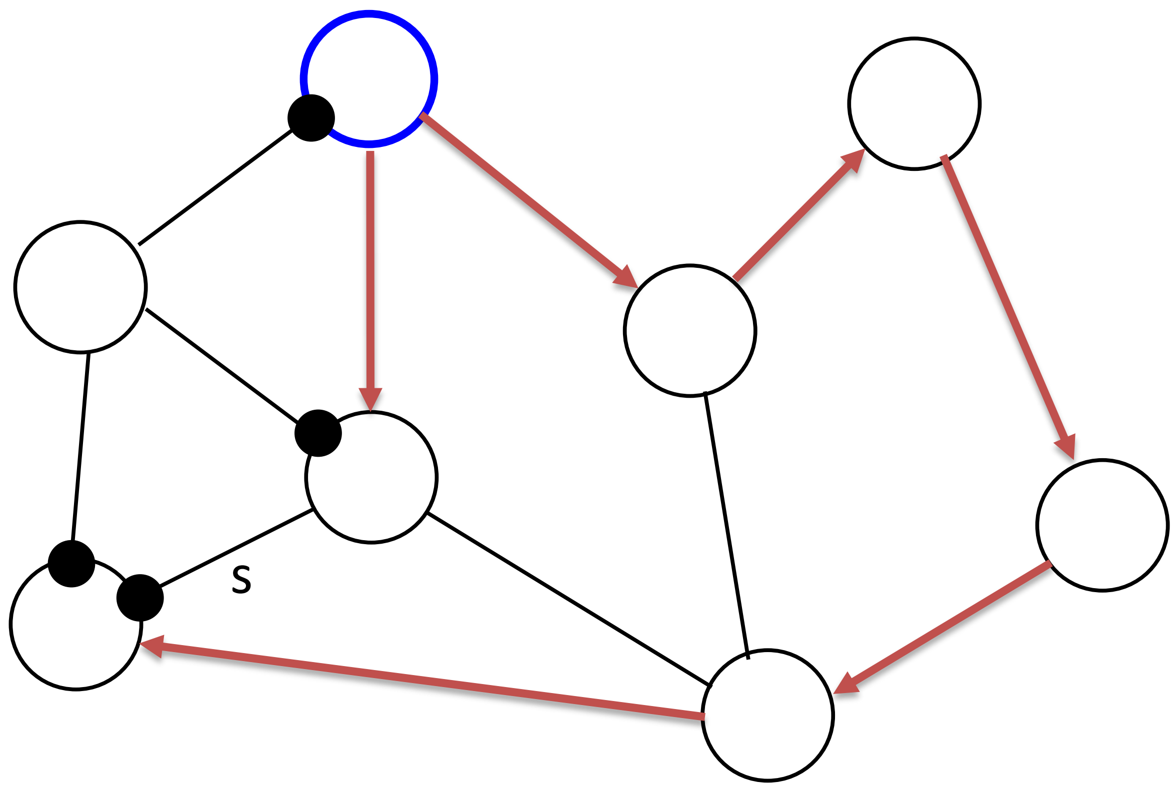
AsynchSpanningTree



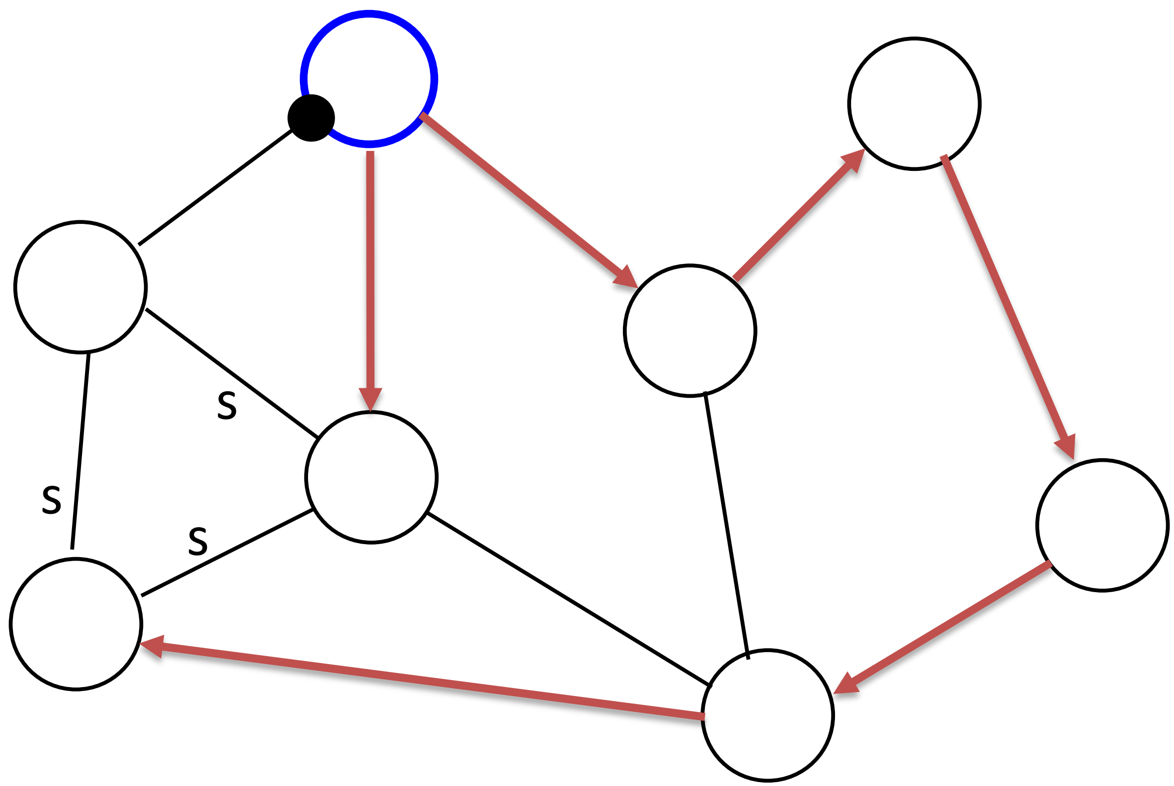
AsynchSpanningTree



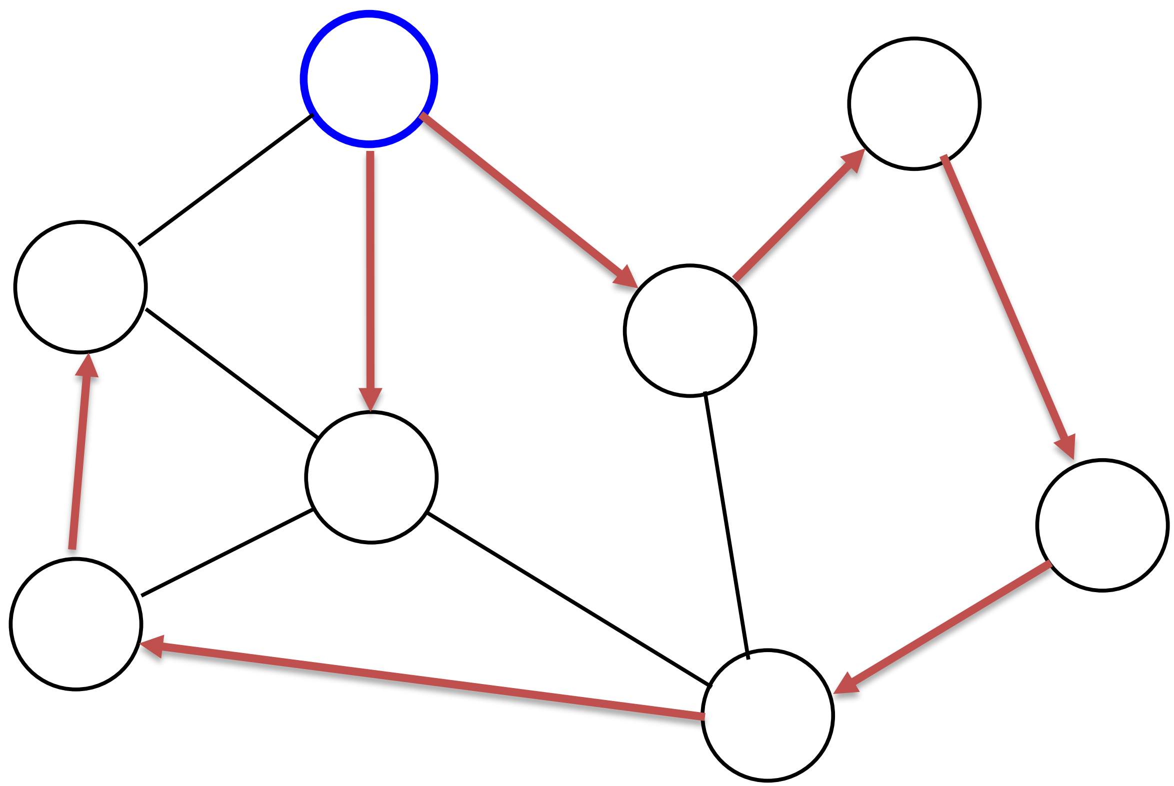
AsynchSpanningTree



AsynchSpanningTree

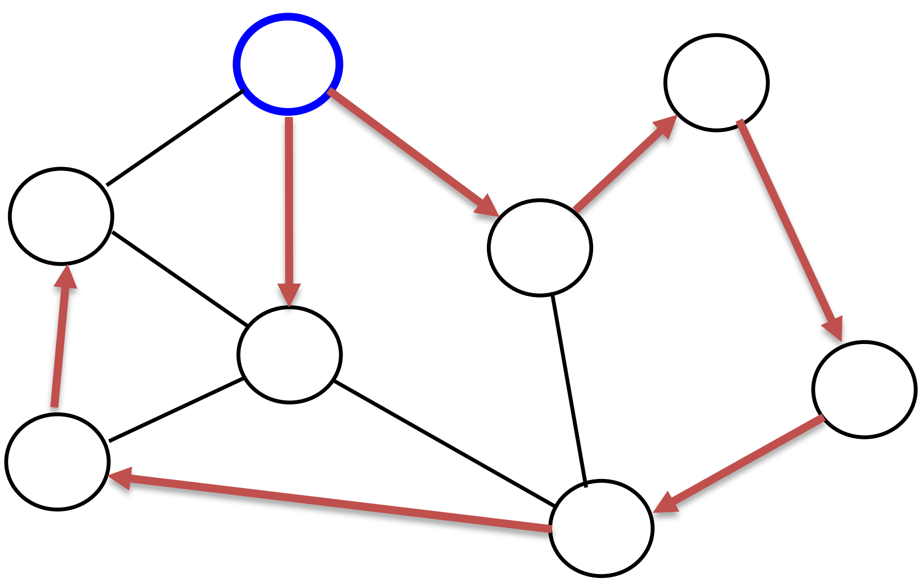


AsynchSpanningTree



AsynchSpanningTree

- Complexity
  - Messages: $O(\,|E|\,)$
  - Time: $diam\,(l + d)\; +\; l$
- Anomaly: Paths may be longer than the diameter!
  - Messages may travel faster along longer paths, in asynchronous networks.



Applications of AsynchSpanningTree

- Similar to synchronous BFS
- Message broadcast: Piggyback on $search$ message.
- Child pointers: Add responses to $search$ messages, easy because of bidirectional communication.
- Use precomputed tree for bcast/convergecast
  - Now the timing anomaly becomes significant.
  - $O(\ h(l+d)\ )$ time complexity.
  - $O(|E|)$ message complexity.
  - See book for details.

h = height of tree; may be as large as n

More applications

- Asynchronous broadcast/convergecast:
  - Can also construct spanning tree while using it to broadcast message and also to collect responses.
  - E.g., to tell the root when the bcast is done, or to collect aggregated data.
  - See book, p. 499-500, $AsynchBcastAck$.
  - Complexity:
    - $O(|E|)$ message complexity.
    - $O(\ n\ (l + d)\ )$ time complexity, timing anomaly.
    - See book for details.
- Elect leader when nodes have no info about the network (no knowledge of $n, diam$, etc.; no root, no spanning tree):
  - All independently initiate $AsynchBcastAck$, use it to determine max, max elects itself.

Next lecture

- More asynchronous network algorithms
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree (GHS)
- Readings:
  - Sections 15.3-15.5
  - [Gallager, Humblet, Spira]