6.852 Lecture 8

- Finish up formal model
  - fairness
  - composition
- Basic asynchronous network algorithms
- Reading: Chapters 8 (continued), 14, 15
- Next lecture: Finish Chapter 15.

Last lecture

- Defined I/O automaton model
  - sig(A): input, output, internal actions
  - states(A) (typically defined by state variables)
  - start(A) $\subseteq$ states(A)
  - trans(A) $\subseteq$ states(A) $\times$ acts(A) $\times$ states(A) ("steps")
    - typically defined using precondition-effect form
  - tasks(A): fairness partition (must be countable)
  - defined executions, traces
- Hierarchical proofs and simulation relations
  - automata as specs: prove one automaton *implements* another
- Safety and liveness properties

Fairness

- Task (set of actions) corresponds to "thread of control"
  - used to define "fair" executions
    - a "thread" that is continuously enabled gets to take a step
  - needed to prove liveness
- Formally, an execution α is **fair** to $C \in$ tasks (A) if:
  - α is finite and C is not enabled in final state
  - α is infinite and either
    - infinitely many events in C occur in α; or
    - C is not enabled in infinitely many states in α

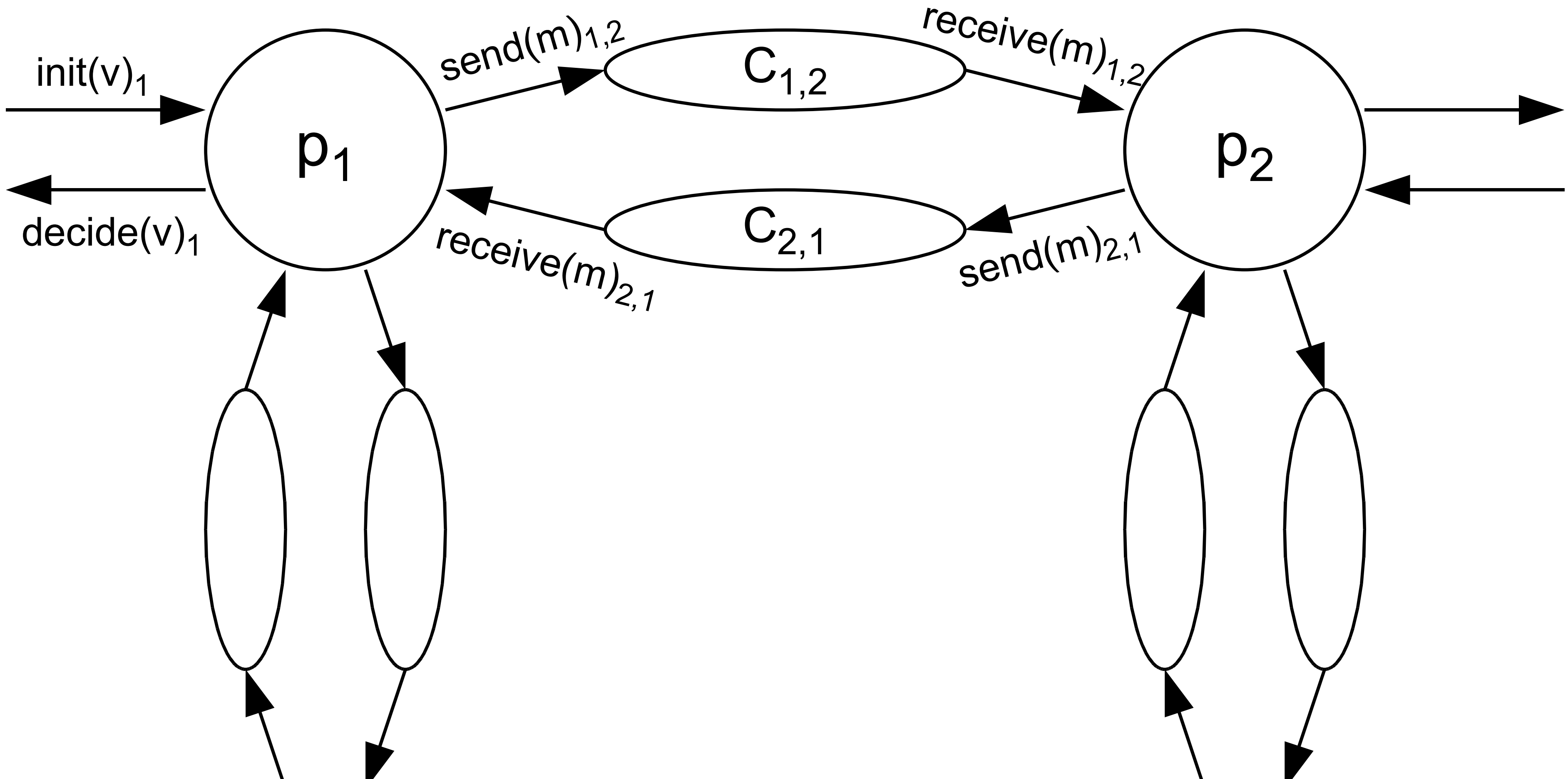
Specifications

- Trace property: Problem specification
  - (sig(P), traces(P))
- Automaton A satisfies trace property P if
  - extsig(A) = sig(P) and traces(A) $\subseteq$ traces(P)
  - extsig(A) = sig(P) and fairtraces(A) $\subseteq$ traces(P)
- Automata as specifications
  - (extsig(A), traces(A))
    - use simulation relations to prove
  - (extsig(A), fairtraces(A))

Safety and liveness

- **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) has an extension (is a prefix) of some sequence in traces(P)
    - “it's never too late”
- Every trace property is intersection of a safety and a liveness property.
- Every (closed) safety property can be specified as automaton.

Composition: Asynchronous network



Composition

- "Put multiple automata together"
  - output actions of one may be input actions of others
- Look first at composing two automata
  - generalize to composing infinitely many automata (in book)
- Recall:
  - sig(A) = (in(A), out(A), int(A))
  - local(A) = out(A) $\cup$ int(A)
- Two automata A and B are **compatible** if
  - local(A) and local(B) are disjoint
  - int(A) and acts(B) are disjoint
  - int(B) and acts(A) are disjoint

Composition

- $A \times B$, composition of A and B
  - $\mathrm{int}(A \times B) = \mathrm{int}(A) \cup \mathrm{int}(B)$
  - $\mathrm{out}(A \times B) = \mathrm{out}(A) \cup \mathrm{out}(B)$
  - $\mathrm{in}(A \times B) = \mathrm{in}(A) \cup \mathrm{in}(B) - (\mathrm{out}(A) \cup \mathrm{out}(B))$
  - $\mathrm{states}(A \times B) = \mathrm{states}(A) \times \mathrm{states}(B)$
  - $\mathrm{start}(A \times B) = \mathrm{start}(A) \times \mathrm{start}(B)$
  - $\mathrm{trans}(A \times B)$: includes (s, π, s') iff
    - $(s_A, \pi, s'_A) \in \mathrm{trans}(A)$ if $\pi \in \mathrm{acts}(A)$; $s_A = s'_A$ otherwise
    - $(s_B, \pi, s'_B) \in \mathrm{trans}(B)$ if $\pi \in \mathrm{acts}(B)$; $s_B = s'_B$ otherwise
  - $\mathrm{tasks}(A \times B) = \mathrm{tasks}(A) \cup \mathrm{tasks}(B)$
- $\prod_{i \in I} A_i$, composition of $\{ A_i : i \in I \}$ (I countable)

Composition: Basic results

- Projection
  - execution of composition “looks good” to each component
- Pasting
  - if execution “looks good” to each component, it is good.
- Substitutability
  - can replace a component with one that implements it

Composition: Basic results

- Projection
  - If $\alpha \in execs(\prod A_i)$ then $\alpha|A_i \in execs(A_i)$ for all i.
  - If $\beta \in traces(\prod A_i)$ then $\beta|A_i \in traces(A_i)$ for all i.
  - If $\alpha \in fairexecs(\prod A_i)$ then $\alpha|A_i \in fairexecs(A_i)$ for all i.
  - If $\beta \in fairtraces(\prod A_i)$ then $\beta|A_i \in fairtraces(A_i)$ for all i.

Composition: Basic results

- Pasting
  - Suppose β is a sequence of external actions of $\prod A_i$.
  - If $\alpha_i \in$ execs(A_i) and $\beta|A_i$ = trace(α_i) for all i
    then there is an execution α of $\prod A_i$
    such that β = trace(α) and $\alpha_i = \alpha|A_i$ for all i.
  - If $\alpha_i \in$ fairexecs(A_i) and $\beta|A_i$ = trace(α_i) for all i
    then there is a fair execution α of $\prod A_i$
    such that β = trace(α) and $\alpha_i = \alpha|A_i$ for all i.
  - If $\beta|A_i \in$ traces(A_i) for all i then $\beta \in$ traces($\prod A_i$).
  - If $\beta|A_i \in$ fairtraces(A_i) for all i then $\beta \in$ fairtraces($\prod A_i$).

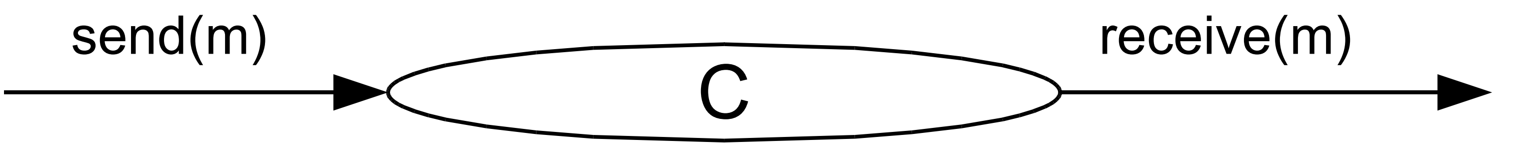
Composition: Basic results

- Substitutability
  - If A_i implements A'_i for all i then $\prod A_i$ implements $\prod A'_i$ (assuming $\prod A_i$ and $\prod A'_i$ are defined).
    - follows from trace projection and pasting
  - Analogous result for “fair implementation”.

Other operations on I/O automata

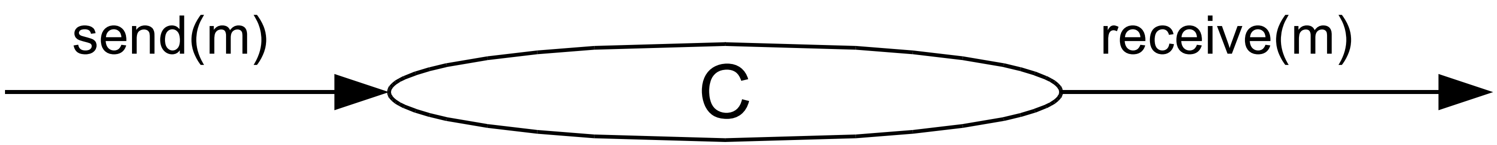
- Hiding
  - make some output actions internal
  - hides internal communication among components of system
- Renaming
  - change names of some actions (changes sig, trans, tasks)
  - important because communication between automata is through shared actions
  - typically just make names right in first place

Channel automaton



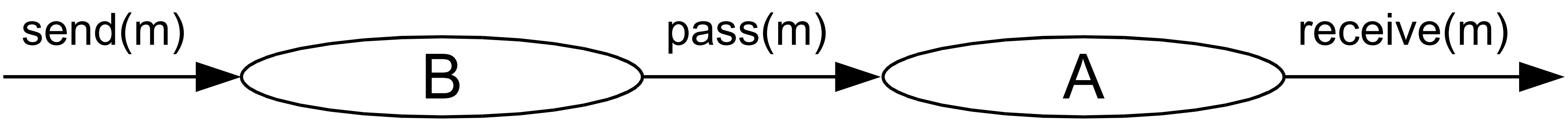
- Reliable unidirectional FIFO channel for 2 processes
  - fix message “alphabet” M
- signature
  - input actions: send(m) for m $\in$ M
  - output actions: receive(m) for m $\in$ M
  - no internal actions
- states
  - **queue**: FIFO queue of M, initially empty

Channel automaton



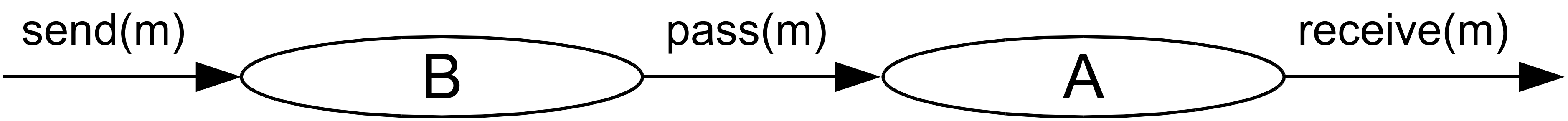
- trans
  - send(m)
    - effect: add m to (end of) **queue**
  - receive(m)
    - precondition: m is at head of **queue**
    - effect: remove head of **queue**
- tasks
  - all receive actions in one task

Composing two channel automata



- Output of B is input of A
  - rename receive(m) of B and send (m) of A to pass(m)
- hide$_{\{\ \mathrm{pass}(m)\ |\ m \in M\ \}}$ $A \times B$ implements C
  - define simulation relation R:
    - for $s \in$ states($A \times B$) and $u \in$ states(C),
      s R u iff u.queue is concatenation of s.A.queue and s.B.queue
  - start: all queues empty, so start states correspond
  - step: define "step correspondence"

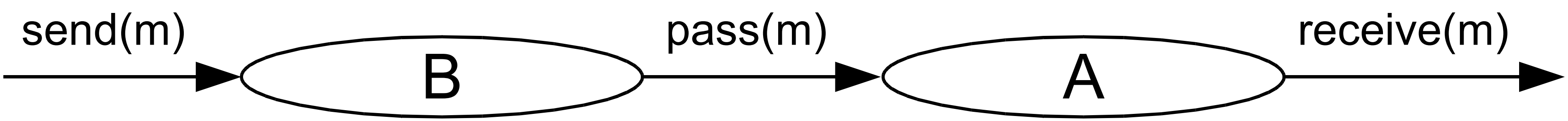
Composing two channel automata



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

- step correspondence:
  - for each step (s, π,s') $\in$ trans(A $\times$ B) and u such that s R u, define execution fragment α of C
    - starts with u ends with u' such that s' R u'
    - trace(α) = trace(π)
  - actions in α depends only on π, uniquely determine post-state
    - same action if external, λ otherwise

Composing two channel automata



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

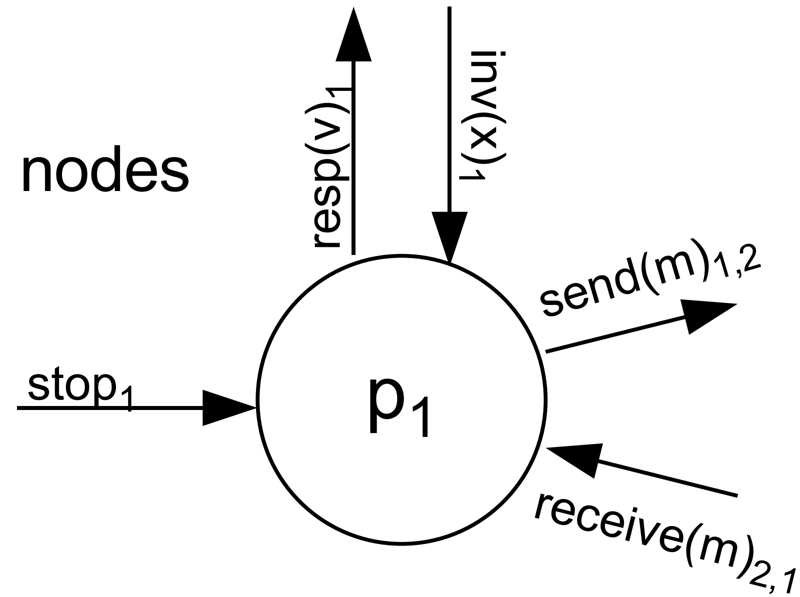
- step correspondence:
  - π = send(m) corresponds to send(m) in C
  - π = receive(m) corresponds to receive(m) in C
  - π = pass(m) corresponds to λ in C
- verify actions are enabled, preserve simulation relation
  - boring case analysis

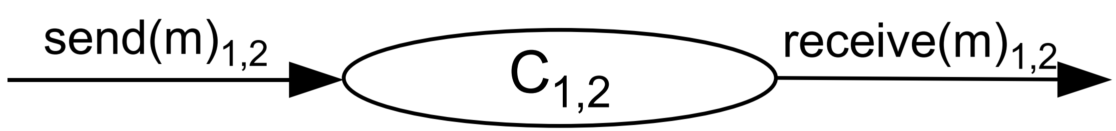
Asynchronous networks

- Processes communicate via channels
  - point-to-point
    - digraph G = (V, E); like synchronous networks, but no rounds
  - broadcast, multicast
- Model processes and channels as I/O automata
  - communicate via send, receive actions
- Basic algorithms on asynchronous networks
  - leader election, set up spanning tree, breadth-first search, shortest paths, minimum spanning tree
  - compare with synchronous algorithms

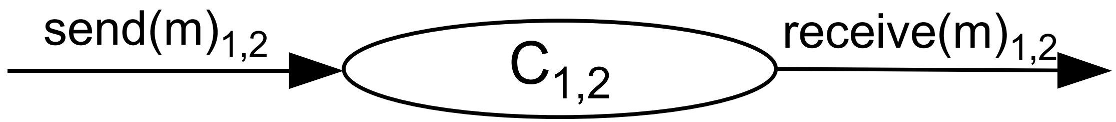
Send/receive systems

- Point-to-point networks
  - process automata associated with nodes
    - problems specify inv, resp and allowable traces
      - hide send/receive actions
    - failures
      - stopping
      - Byzantine
  - channel automata associated with (directed) edges





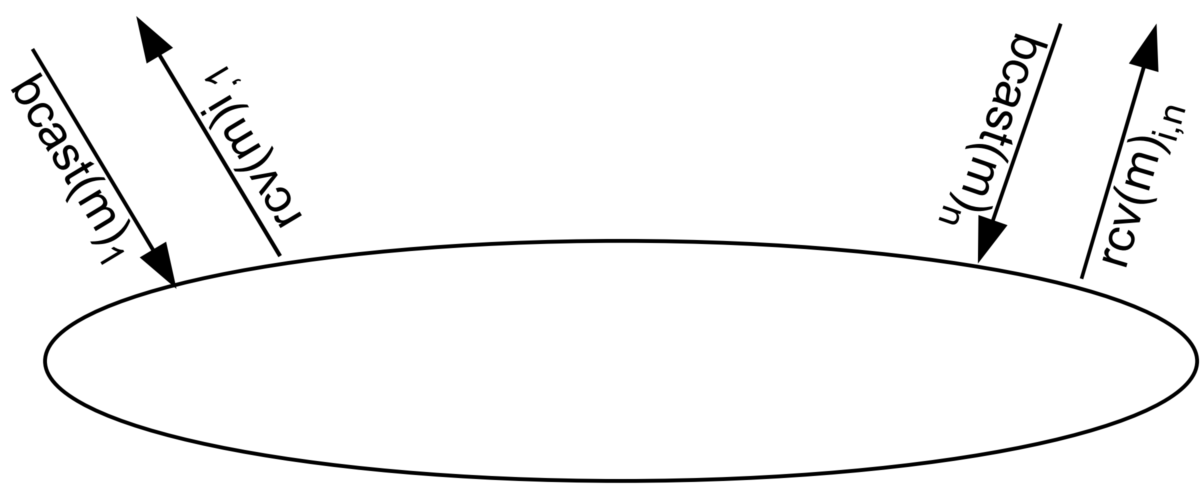
Channel automata



- Different kinds of channel with this interface
  - reliable FIFO
  - weaker guarantees: lossy, duplicating, reordering
- Can also define trace properties (use “cause” fn)
  - integrity: map preserves message
  - no loss: map is onto
  - no duplicates: map is 1-1
  - no reordering: map is order-preserving (monotone)

Broadcast and multicast

- Broadcast
  - reliable FIFO between each pair, but different processes can receive msgs from different senders in different orders
  - model: separate queues for each pair
  - failures, consistency conditions (e.g., atomic bcast)
- Multicast: processes designate recipients



Asynchronous network algorithms

- Assume reliable FIFO point-to-point channels
- Look at problems solved for synchronous network
  - Leader election in ring
  - Leader election in general networks
  - Spanning tree construction
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree
- How much holds over?
  - where did we use synchronous assumption?

Leader election in a ring

- Recap assumptions
  - G is a ring, unidirectional or bidirectional communication
  - local names for neighbors, UIDs
- LeLann-Chang-Roberts (AsynchLCR)
  - send UID clockwise around ring (unidirectional)
  - throw away UIDs smaller than your own
  - elect self if your UID returns
  - correctness: basically same as for synchronous algorithm
    - but now must consider messages in channels, “pileup”
    - messages sent individually (induction on steps vs. rounds)

AsynchLCR

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

- $send(v)_{i,i+1}$

  pre: v is head of **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

- Safety: no process other than i_{max} performs $leader_i$
  - if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then u_i not in $send_j$.
- Liveness: i_{max} eventually performs $leader_i$
  - if distance from i_{max} to i is d, then u_{max} is in $send_i$ after ??

AsynchLCR

- Safety: no process other than i_{max} performs $leader_i$
  - if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then u_i not in $send_j$ or in $queue_{j,j+1}$
- Liveness: i_{max} eventually performs $leader_i$
  - for $k \in [0,n-1]$, u_{max} eventually in $send_{i_{max}+k}$ / $queue_{i_{max}+k,i_{max}+k+1}$
  - prove by induction on k; use fairness to prove inductive step
- Complexity
  - msg: $O(n^2)$, as before
  - time: ??

AsynchLCR

- Safety: no process other than i_{max} performs $leader_i$
  - if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then u_i not in $send_j$ or in $queue_{j,j+1}$
- Liveness: i_{max} eventually performs $leader_i$
  - for $k \in [0,n-1]$, u_{max} eventually in $send_{i_{max}+k}$ / $queue_{i_{max}+k,i_{max}+k+1}$
  - prove by induction on k; use fairness to prove inductive step
- Complexity
  - msg: $O(n^2)$, as before
  - time: $O(\, n(l+d) \,)$
    - l is upper bound on local step time for each process
    - d is upper bound on time to deliver first message

only upper bounds okay:
does not restrict executions

Leader election in a ring

- Reduce message complexity?
  - Hirschberg-Sinclair: O(n log n), requires bidirectional comm.
- Peterson's algorithm
  - O(n log n) messages
  - unidirectional communication
  - unknown ring size
  - comparison-based

Leader election in a ring

- Peterson's leader election algorithm
  - Proceed in phases, each process may be **active** or **passive**
    - passive process just pass messages along
  - Phase 1:
    - send UID down two processes; get two UIDs
    - remain active iff middle UID is max
      - adopt middle UID (the max one)
    - at most half processes are active “after” first phase
  - Later phases: ??
    - phases may be concurrent
  - Termination ??

PetersonLeader

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

PetersonLeader

- get-second-uid$_i$
  - pre: **mode** = active
    - **receive** is nonempty
    - **uid2** = null
  - eff: **uid2** := head of **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 of **receive**
    - remove head of **receive**
- advance-phase$_i$
  - pre: **mode** = active
    - **uid3** ≠ null
    - **uid2** > max(**uid1**, **uid3**)
  - eff: **uid1** := **uid2**
    - **uid2** := null
    - **uid2** := 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 of **receive** to **send**

PetersonLeader

- Tasks:
  - { 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
  - msg: O(n log n)
  - time: O(n(l+d))

Leader election in a ring

- Can we do better than O(n log n) msg complexity?
  - not with comparison-based algorithms (why?)
  - not at all—but we didn't cover this
- Lower bound in asynchronous network if n is unknown
  - Key: “assemble” ring from pieces which delay communication
    - **silent** state: no more messages will be sent without input
    - ring looks like “line” if communication delayed across ends
    - take 3 lines that send k messages before becoming silent
    - some pair sends 2k+l messages before becoming silent
      - l is the length of the lines
    - connect ends of line to turn into ring

Next lecture

- More asynchronous network algorithms (Chapter 15)
  - Constructing a spanning tree
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree