6.852: Distributed Algorithms
Fall, 2015

Lecture 9

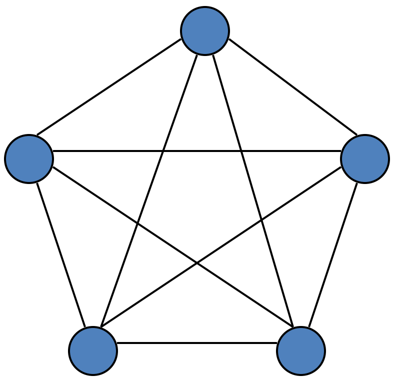
Today's plan

- Distributed commit
- Formal modeling of asynchronous systems:
  - I/O automata
  - Executions and traces
  - Operations: composition, hiding
  - Properties and proof methods:
    - Invariants
    - Simulation relations
- Reading: Section 7.3, Chapter 8
- Next:
  - Asynchronous network algorithms: Leader election, breadth-first search, shortest paths, spanning trees.
  - Reading: Chapters 14 and 15

Distributed Commit

Distributed Commit

- Motivation: Distributed database transaction processing
  - A database transaction performs work at several distributed sites.
  - Transaction manager (TM) at each site decides whether it would like to "commit" or "abort" the transaction.
    - Based on whether the transaction's work has been successfully completed at that site, and results made stable.
  - All TMs must agree on whether to commit or abort.
- Assume:
  - Process stopping failures only.
  - n-node, complete, undirected graph.



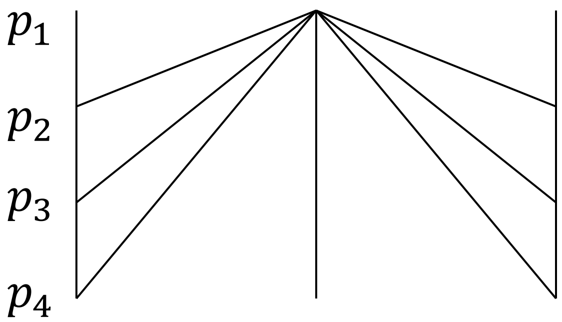
- Require:
  - Agreement: No two processes decide differently (faulty or not, uniformity)
  - Validity:
    - If any process starts with 0 (abort) then 0 is the only allowed decision.
    - If all start with 1 (commit) and there are no faulty processes then 1 is the only allowed decision.

Correctness Conditions for Commit

- Agreement: No two processes decide differently.
- Validity:
  - If any process starts with 0 then 0 is the only allowed decision.
  - If all start with 1 and there are no faulty processes then 1 is the only allowed decision.
- Note asymmetry: Guarantee abort (0) if anyone wants to abort; guarantee commit (1) if everyone wants to commit and no one fails (best case).
- Termination:
  - Weak termination: If there are no failures then all processes eventually decide.
  - Strong termination (non-blocking condition): (Even if there are failures), all nonfaulty processes eventually decide.

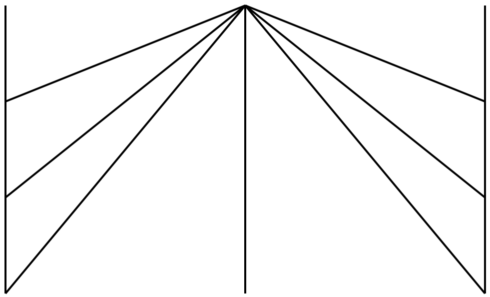
2-Phase Commit

- Traditional, blocking algorithm (guarantees weak termination only).
- Assumes distinguished process 1, acts as "coordinator" (leader).
- Round 1: All send initial values to process 1, who decides.
  - If it sees 0, or doesn't hear from someone, it decides 0; otherwise it decides 1.
- Round 2: Process 1 sends the decision to everyone else.

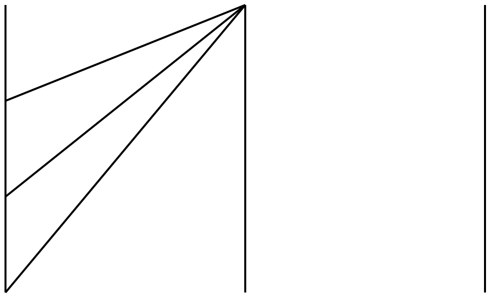


- Q: When can the processes decide?
- Anyone with initial value 0 can decide at the beginning.
- Process 1 decides after receiving round 1 messages.
- Everyone else decides after round 2 (if there are no failures).

Correctness of 2-Phase Commit

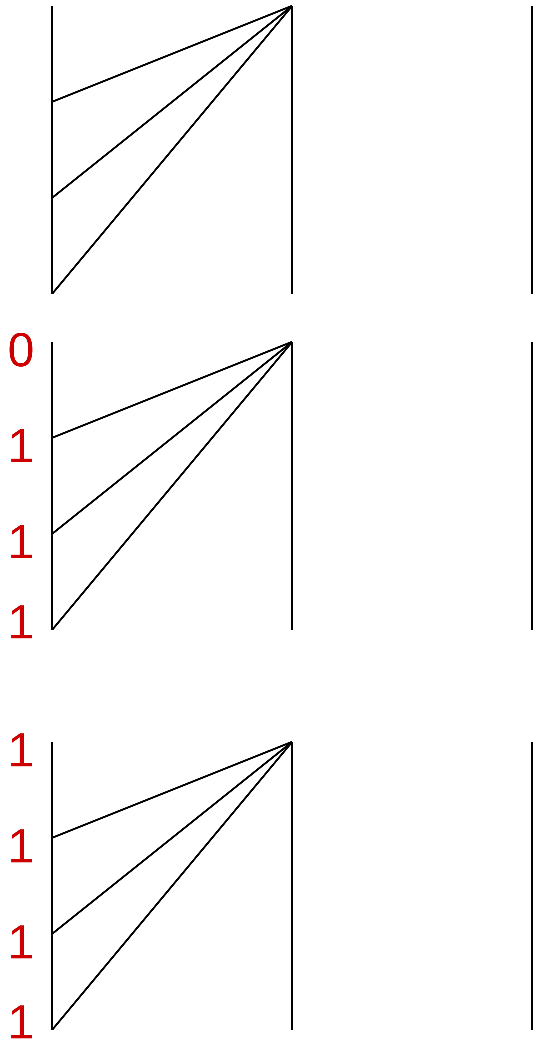


- Agreement:
  - Because decision is centralized (and consistent with any individual initial decisions).
- Validity:
  - Because of how the coordinator decides.
- Weak termination:
  - If no one fails, then everyone terminates by the end of round 2.
- Strong termination?
  - No: If the coordinator fails before sending its round 2 messages, then others with initial value 1 will never terminate.



Add a termination protocol?

- We might try to add a termination protocol: other processes try to detect failure of coordinator and finish agreeing on their own.
- But this can't always work:
  - If initial values are 0,1,1,1, then by validity, everyone is required to decide 0.
  - If initial values are 1,1,1,1 and process 1 fails just after deciding, and before sending out its round 2 messages, then:
    - Process 1 decides 1.
    - By agreement, others must decide 1.
  - But the other processes can't distinguish these two situations.

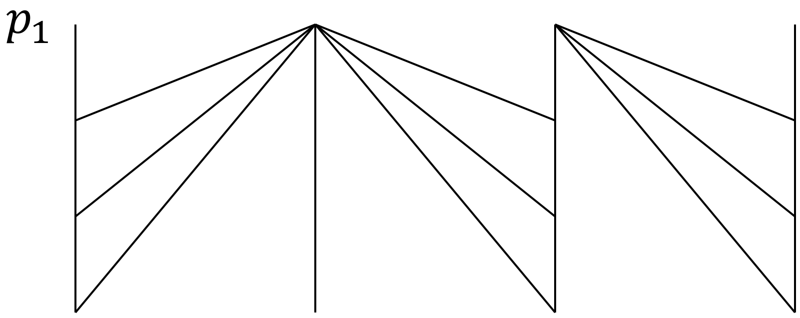


Complexity of 2-phase commit

- Time:
  - 2 rounds
- Communication:
  - At most $2n$ messages

3-Phase Commit [Skeen]

- Yields strong termination.
- Trick: Introduce intermediate stage, before actually deciding.
- Process states are now classified into four categories:
  - $dec0$: Already decided 0.
  - $dec1$: Already decided 1.
  - $ready$: Ready to decide 1 but hasn't yet.
  - $uncertain$: Otherwise.
- Again, process 1 acts as "coordinator".
- Communication pattern:

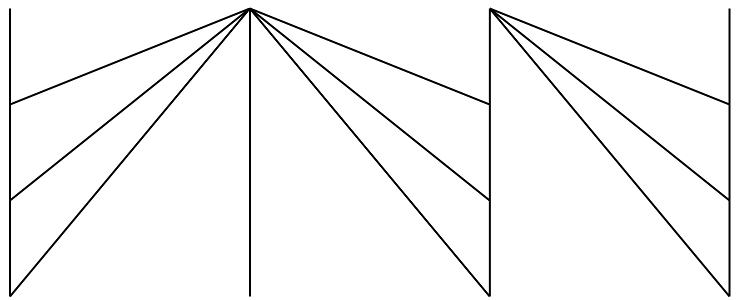


3-Phase Commit

- All processes are initially *$uncertain$.*
- Round 1:
  - All other processes send their initial values to p_1.
  - All with initial value 0 decide 0 (and enter $dec0$ state)
  - If p_1 receives 1s from everyone and its own initial value is 1, p_1 becomes $ready$, but doesn't yet decide.
  - If p_1 sees 0 or doesn't hear from someone, p_1 decides 0.

- Round 2:
  - If p_1 has decided 0, it broadcasts "decide 0", else it broadcasts "ready".
  - Anyone else who receives "decide 0" decides 0.
  - Anyone else who receives "ready" becomes $ready$.
  - Now p_1 decides 1 if it hasn't already decided.

- Round 3:
  - If p_1 has decided 1, it bcasts "decide 1".
  - Anyone else who receives "decide 1" decides 1.



3-Phase Commit

- Key invariants (after 0, 1, 2, or 3 rounds):
  - If any process is in $ready$ or $dec1$, then all processes have initial value 1.
  - If any process is in $dec0$ then:
    - No process is in $dec1$, and no non-failed process is $ready$.
  - If any process is in $dec1$ then:
    - No process is in $dec0$, and no non-failed process is $uncertain$.
- Proof: LTTR.
  - Key step: Third condition is preserved when p_1 decides 1 after round 2.
  - In this case, p_1 knows that:
    - Everyone's input is 1.
    - No one decided 0 at the end of round 1.
    - Every other process has either become $ready$ or has failed (without deciding).
  - Implies the third condition.
- Note critical use of synchrony here:
  - p_1 infers that non-failed processes are $ready$ just because round 2 is completed.
  - Without synchrony, this would require explicit acknowledgments.

Correctness conditions (so far)

- Agreement and validity follow, for these three rounds.
- Weak termination holds
- Strong termination:
  - Doesn’t hold yet---must add a termination protocol.
  - Allow process 2 to act as coordinator, then 3,…
  - “Rotating coordinator” strategy

3-Phase Commit

- Round 4:
  - All processes send current status ($dec0$, $uncertain$, $ready$, $dec1$) to p_2.
  - If p_2 receives any $dec0$'s and hasn't already decided, then p_2 decides 0.
  - If p_2 receives any $dec1$'s and hasn't already decided, then p_2 decides 1.
  - If all received values, and its own value, are $uncertain$, then p_2 decides 0.
  - Otherwise (all values are $uncertain$ or $ready$ and at least one is $ready$), p_2 becomes $ready$, but doesn't decide yet.

- Round 5 (analogous to round 2):
  - If p_2 has (ever) decided 0, broadcasts "decide 0", and similarly for 1.
  - Else broadcasts "ready".
  - Any undecided process who receives "decide()" decides accordingly.
  - Any process who receives "ready" becomes $ready$.
  - Now p_2 decides 1 if it hasn't already decided.

- Round 6 (analogous to round 3):
  - If p_2 has decided 1, broadcasts "decide 1".
  - Anyone else who receives "decide 1" decides 1.

- Continue with subsequent rounds for $p_3, p_4, \ldots$

Correctness

- Key invariants still hold:
  - If any process is in $ready$ or $dec1$, then all processes have initial value 1.
  - If any process is in $dec0$ then:
    - No process is in $dec1$, and no non-failed process is $ready$.
  - If any process is in $dec1$ then:
    - No process is in $dec0$, and no non-failed process is $uncertain$.
- Imply agreement, validity
- Strong termination:
  - Because eventually some coordinator will finish the job (unless everyone fails).

Complexity

- Time until everyone decides:
  – Normal case 3
  – Worst case $3n$
- Messages until everyone decides:
  – Normal case $O(n)$
    - Technicality: When can processes stop sending messages?
  – Worst case $O(n^2)$

Practical issues for 3-phase commit

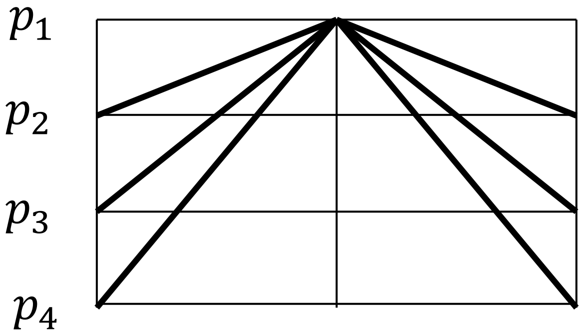
- Depends on strong assumptions, which may be hard to guarantee in practice:
  - Synchronous model:
    - Could emulate with approximately-synchronized clocks, timeouts.
  - Reliable message delivery:
    - Could emulate with acks and retransmissions.
    - But if retransmissions add too much delay, then we can't emulate the synchronous model accurately.
    - Leads to unbounded delays, asynchronous model.
  - Accurate diagnosis of process failures:
    - Get this "for free" in the synchronous model.
    - E.g., 3-phase commit algorithm lets process that doesn't hear from another process i at a round conclude that i must have failed.
    - Very hard to guarantee in practice: In Internet, or even a LAN, how to reliably distinguish failure of a process from lost communication?
- Other consensus algorithms can be used for commit, including some that don't depend on such strong timing and reliability assumptions.

Paxos consensus algorithm [Lamport]

- A more robust consensus algorithm, can be used for commit.
- Tolerates process stopping and recovery, message losses and delays,…
- Runs in partially synchronous model.
- Similar to algorithm by [Dwork, Lynch, Stockmeyer].
- Algorithm idea:
  - Processes use an unreliable leader election subalgorithm to choose a coordinator, who tries to achieve consensus.
  - Coordinator decides based on active support from a majority of the processes.
  - Does not assume anything based on not receiving a message.
  - Subtleties arise when multiple coordinators are active---must ensure consistency.
- Practical difficulties with fault-tolerance in the synchronous model motivate studying the asynchronous model (later today).

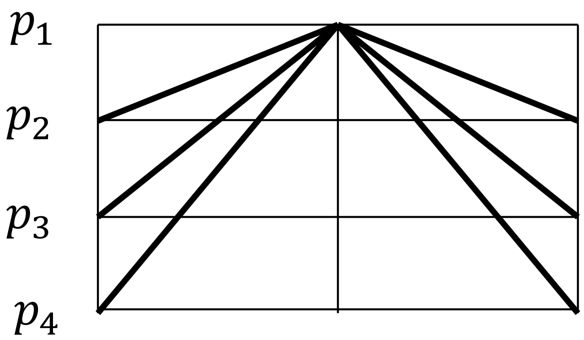
A Lower Bound for Commit

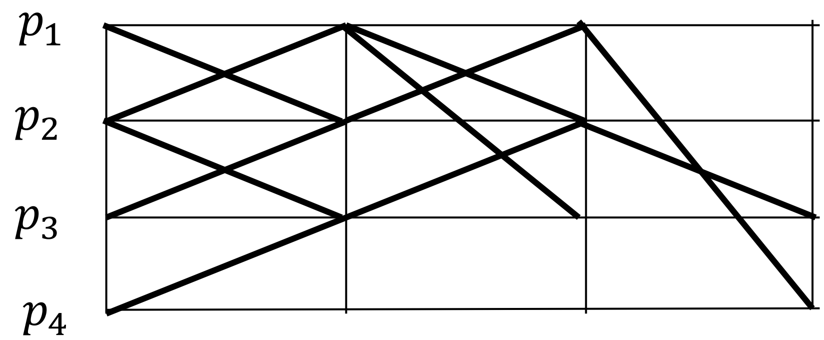
- How many messages are needed to solve the commit problem?
- Theorem [Dwork, Skeen]: Any algorithm that solves the commit problem, even with weak termination, uses at least $2n - 2$ messages in the failure-free execution α in which all inputs are 1.
- Note: That's what 2-phase commit uses, so 2-phase commit is "optimal":



- Proof considers the communication pattern for α:

Information flow in a communication pattern

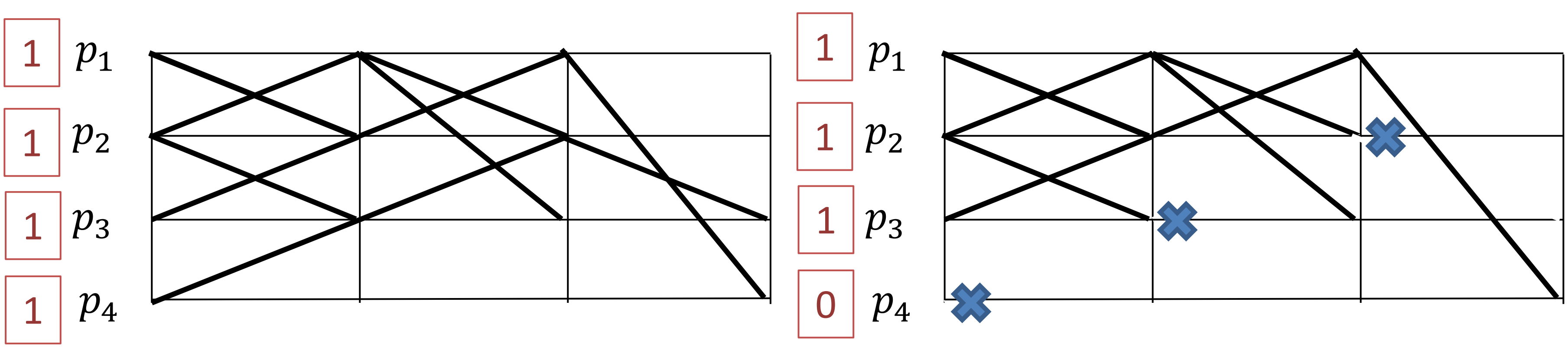




- i affects j in a pattern if there is a path in the pattern from i at time 0 to j at some later time.
- In Pattern 1, all processes affect all processes.
- In Pattern 2, 4 does not affect 1.
- Lemma: In the failure-free, all-1-input run α, every i affects every j in the communication pattern of α.
- Corollary: The communication pattern of α has at least $2n - 2$ edges.

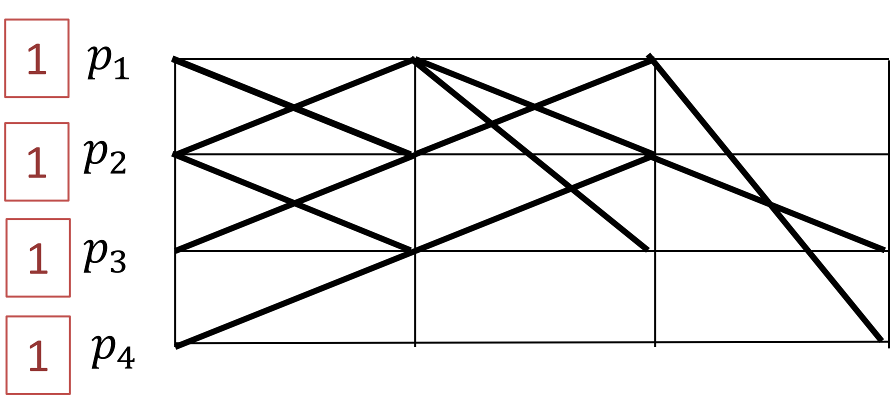
Proof of the Lemma

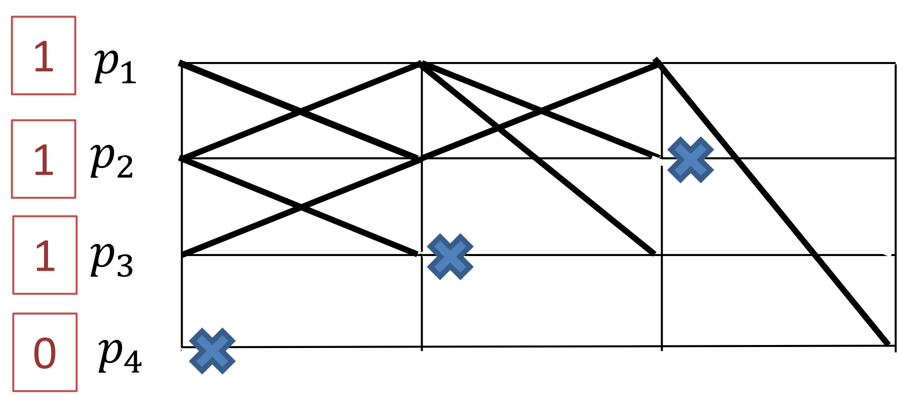
- Lemma: In the failure-free, all-1-input run α, every i affects every j in the communication pattern of α.
- Proof:
  - By contradiction. Suppose i does not affect j (for some particular i, j).
  - Then $i \neq j$.
  - Construct execution α', which is the same as α except that:
    - i's input is 0, and
    - Every process that is affected by process i in α fails just after it first gets affected by process i in α.
- Example: Process 4 does not affect process 1.



Proof of the Lemma

- Lemma: In the failure-free, all-1-input run α, every i affects every j in the communication pattern of α.
- Proof, cont'd:
  - Construct execution α':
    - i's input is 0, and
    - Every process that is affected by process i in α fails just after it first gets affected by process i in α.
  - In α, all processes eventually decide 1.
  - α' is indistinguishable from α to process j.
  - So process j decides 1 in α' , which contradicts the requirements.

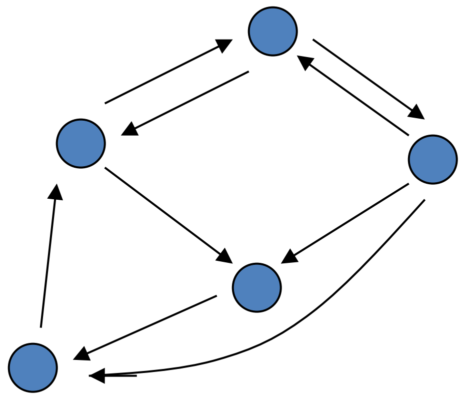


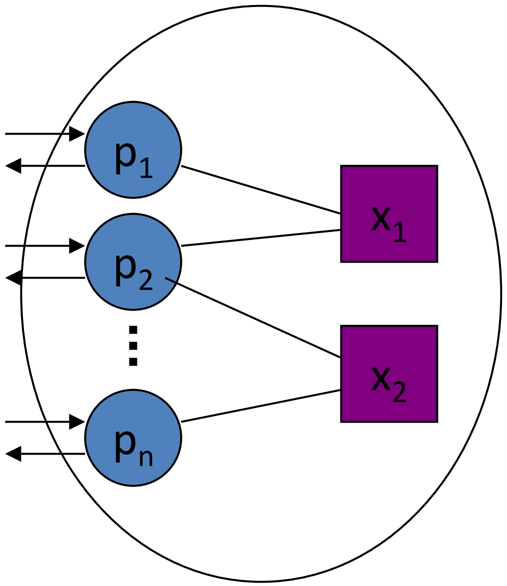


Asynchronous Systems

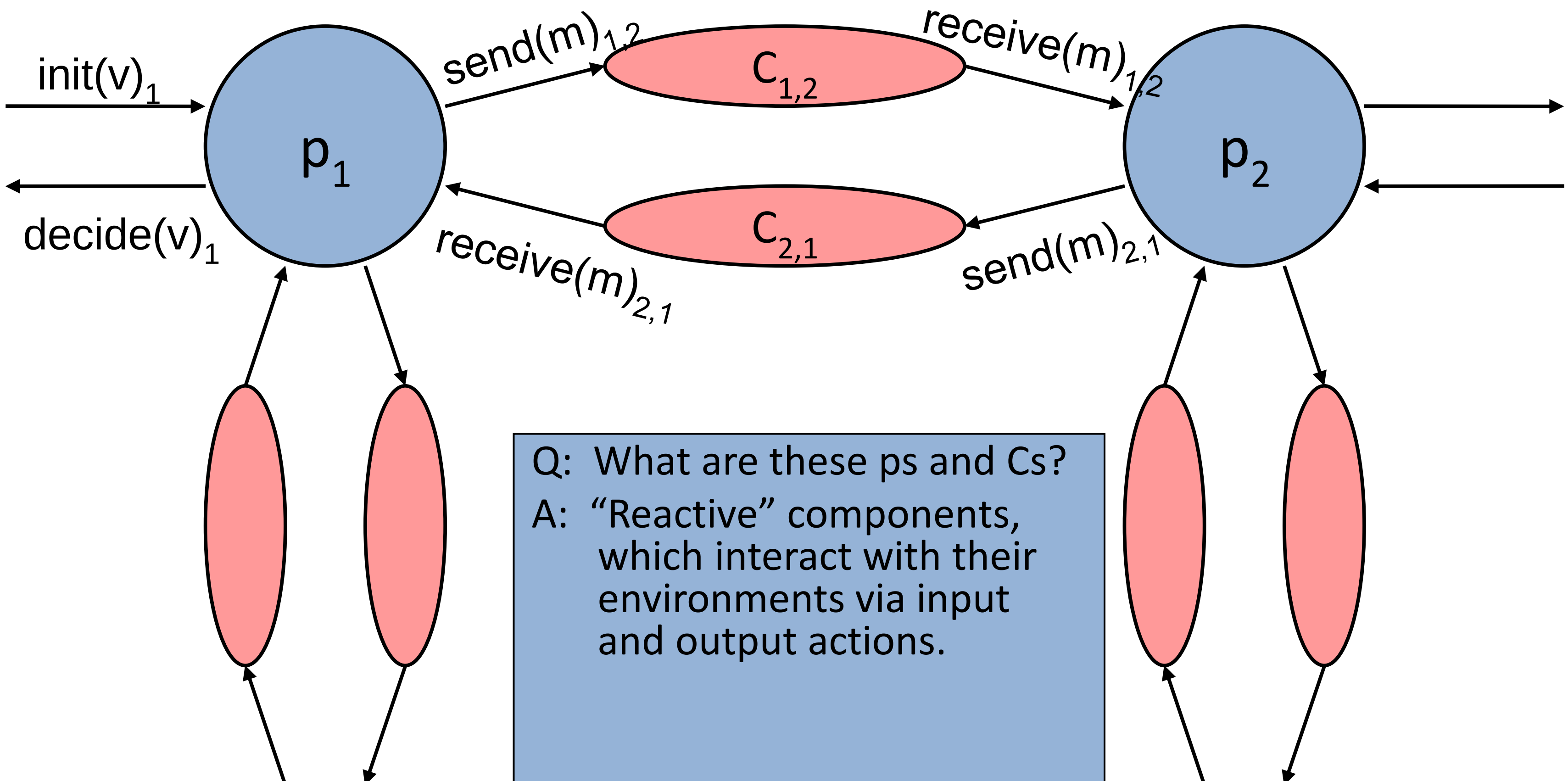
Asynchronous systems

- No timing assumptions
  - No rounds
- Two kinds of asynchronous models:
  - Asynchronous networks
    - Processes communicating via channels
  - Asynchronous shared-memory systems
    - Processes communicating via shared objects





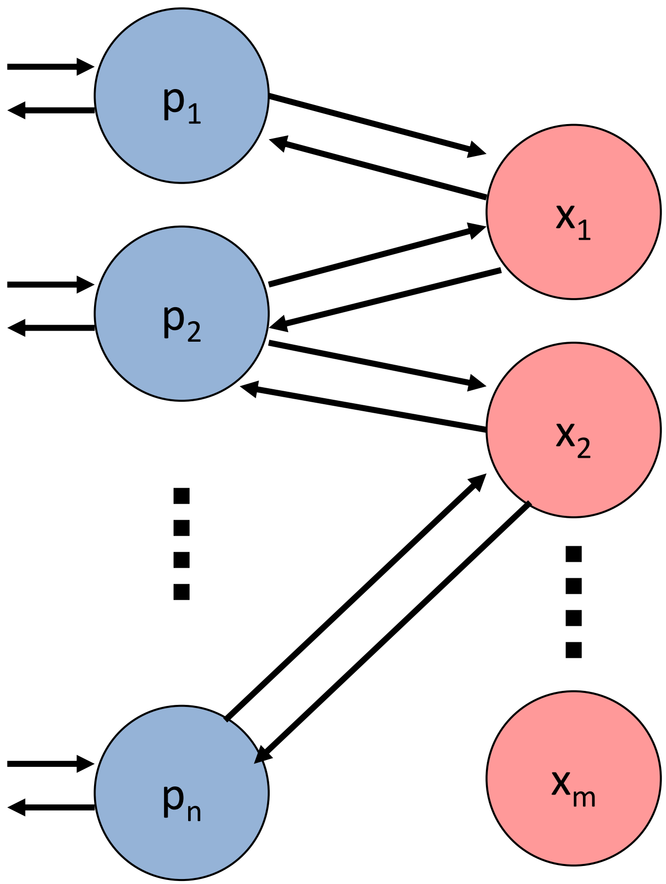
Asynchronous network: Processes and channels



Q: What are these ps and Cs?

A: “Reactive” components, which interact with their environments via input and output actions.

Asynchronous shared-memory system: Processes and objects



These processes and objects are also "reactive" components.

In both cases, we have reactive components.

We need a general model for reactive components.

Specifying problems and systems

- Processes, channels, and objects are automata
  - Perform actions while changing state.
  - Reactive
    - Interact with environment via input and output actions.
    - Not just functions from input values to output values; they may have more kinds of interactions.
- Execution:
  - Sequence of states and actions
  - Interleaving semantics
- External behavior (trace):
  - We observe external actions.
  - States and internal actions are hidden.
  - Problems are defined in terms of allowable traces.

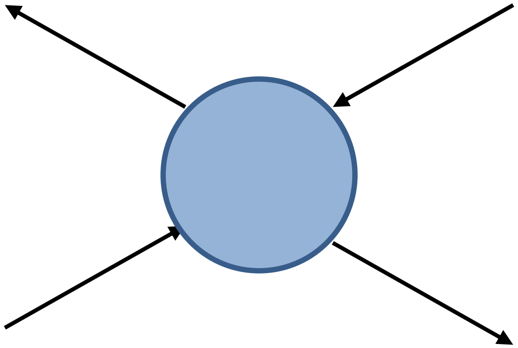
Input/Output Automata

Input/Output Automata

- General mathematical modeling framework for reactive system components.
  - Little structure---must add special structure to specialize it for networks, shared-memory systems,…
- Designed for describing systems in a modular way:
  - Supports description of individual system components, and how they compose to yield a larger system.
  - Supports description of systems at different levels of abstraction, e.g.:
    - Detailed implementation vs. higher-level algorithm description.
    - Optimized algorithm vs. simpler, un-optimized version.
- Supports several standard proof techniques:
  - Invariants
  - Simulation relations (like running 2 algorithms side-by-side and relating their behavior step-by-step).
  - Compositional reasoning (prove properties of individual components; use compositional reasoning to infer properties for the overall system).

Input/Output Automaton

- State transition system
  - Transitions labeled by actions
- Actions classified as input, output, internal
  - Input, output are external.
  - Output, internal are locally controlled.



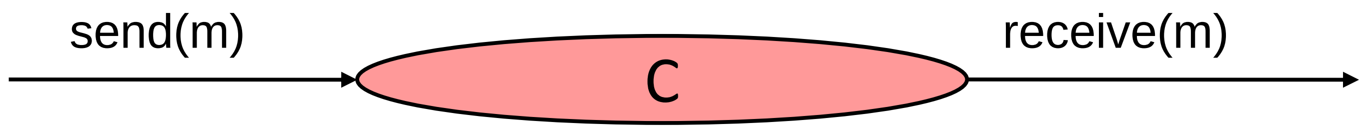
Input/Output Automaton, formally

- $sig = (in, out, int)$
  - input, output, internal actions (disjoint)
  - $in \cup out \cup int$
  - $ext = in \cup out$
  - $local = out \cup int$
- $states$: Not necessarily finite
- $start \subseteq states$
- $trans \subseteq states \times acts \times states$
  - Input-enabled: Any input “enabled” in any state.
- $tasks$, partition of locally controlled actions
  - Used to specify liveness.

Remarks

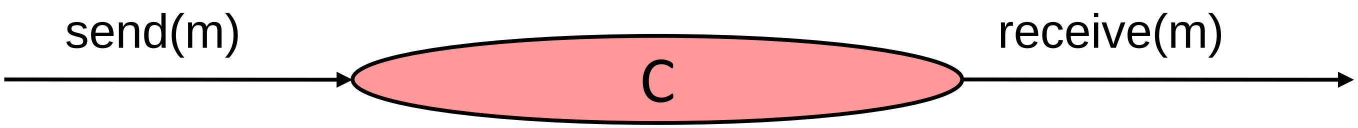
- A step of an automaton is an element of $trans$.
- Action π is enabled in a state s if $trans$ contains a step (s, π, s') for some s'.
- I/O automata must be input-enabled.
  - Every input action is enabled in every state.
  - Captures the idea that an automaton cannot control its inputs.
    - If we want restrictions, model the environment as another automaton and express restrictions in terms of the environment.
    - Could allow a component to detect bad inputs and halt, or exhibit unconstrained behavior for bad inputs.
- Tasks correspond to “threads of control”.
  - Used to define fairness (give turns to all tasks).
  - Needed to guarantee liveness properties (e.g., the system keeps making progress, or eventually terminates).

Example: Channel automaton



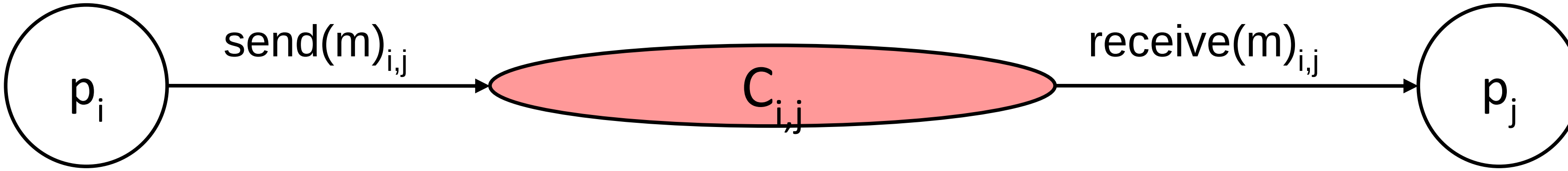
- Reliable unidirectional FIFO channel between two processes.
  - Fix message alphabet M.
- signature
  - input actions: $send(m), m \in M$
  - output actions: $receive(m), m \in M$
  - No internal actions
- states
  - $queue$: FIFO queue of M, initially empty

Channel automaton



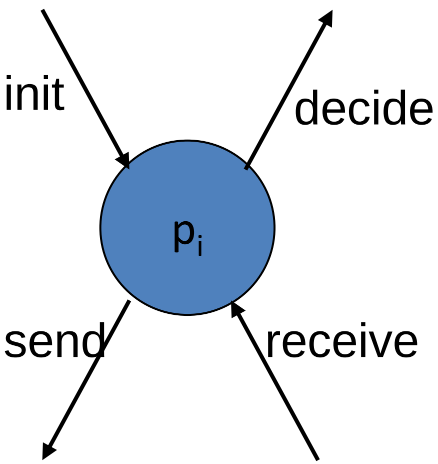
- trans
  - $send(m)$
    - effect: add m to (end of) $queue$
  - $receive(m)$
    - precondition: m is at head of $queue$
    - effect: remove head of $queue$
- tasks
  - All $receive$ actions in one task.

Channel automaton



- trans
  - $send(m)_{i,j}$
    - effect: add m to (end of) $queue$
  - $receive(m)_{i,j}$
    - precondition: m is at head of $queue$
    - effect: remove head of $queue$
- tasks
  - All $receive$ actions in one task.

A process



- E.g., in a consensus protocol.
- See book, p. 205, for code details.

- Inputs arrive from the outside.
- Process sends/receives values, collects vector of values, one for each process.
- When vector is filled, outputs a decision obtained as a function of the vector.
- Can get new inputs, change values, send and output repeatedly.

- Tasks for:
  - Sending to each individual neighbor.
  - Outputting decisions.

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:
  - $s_0\ \pi_1\ s_1\ \pi_2\ s_2\ \pi_3\ s_3\ \pi_4\ s_4\ \pi_5\ s_5$ ... (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)

$$\lambda, send(a), a, send(b), ab, receive(a), b, receive(b), \lambda$$

Execution fragments

- An I/O automaton executes as follows:
  - Start at some start state.
  - Repeatedly take step from current state to new state.

execution fragment

- 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.

Invariants and reachable states

- A state is reachable if it appears in some execution.
  - Equivalently, 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/distributed algorithms.
  - Typically proved by induction on length of execution.

Traces

- Traces allow us to focus on components' external behavior.
- Useful for defining correctness.
- A trace of an execution is the subsequence of external actions in the execution.
  - No states, no internal actions.
  - Denoted $trace(\alpha)$, where α is an execution.
  - Models "observable behavior".

$$\lambda, send(a), a, send(b), ab, receive(a), b, receive(b), \lambda$$

$$send(a), send(b), receive(a), receive(b)$$

Operations on I/O Automata

Operations on I/O automata

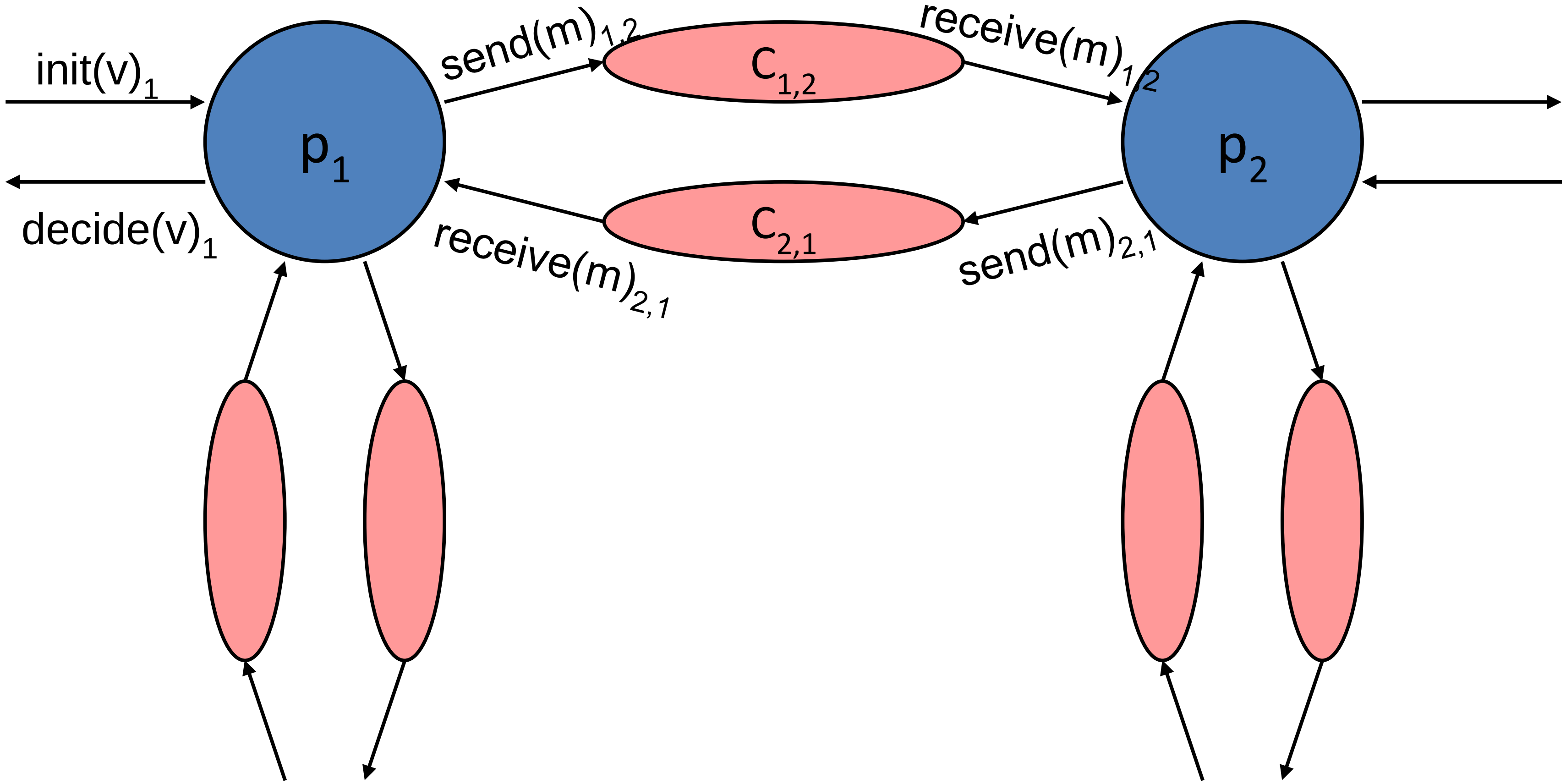
- To describe how systems are built out of components, the model has operations for composition, hiding, renaming.
- Composition:
  - "Put multiple automata together."
  - Output actions of one may be input actions of others.
  - All components having an action perform steps involving that action together ("synchronize on actions").
- Composing finitely many (or countably many) automata $A_i, i \in I$:
- Need compatibility conditions:
  - Internal actions aren't shared:
    - $int(A_i) \cap acts(A_j) = \emptyset$
  - Only one automaton controls each output:
    - $out(A_i) \cap out(A_j) = \emptyset$
  - But output of one automaton can be an input of one or more others.
  - No action is shared by infinitely many A_is.

Composition of compatible automata

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

- Notation: $\Pi_{i \in I} A_i$, for composition of A_i, $i \in I$.

Composition of channels and consensus processes



Composition: Basic results

- Projection
  - Execution of composition "looks good" to each component.
- Pasting
  - If execution "looks good" to each component, it is good overall.
- Substitutivity
  - Can replace a component with one that implements it.

Composition: Basic results

Theorem 1: Projection

- If $\alpha \in execs(\Pi A_i)$ then $\alpha | A_i \in execs(A_i)$ for every i.
- If $\beta \in traces(\Pi A_i)$ then $\beta | A_i \in traces(A_i)$ for every i.

Composition: Basic results

Theorem 2: Pasting

Suppose β is a sequence of external actions of ΠA_i.

- If $\alpha_i \in execs(A_i)$ and $\beta|A_i = trace(\alpha_i)$ for every i, then there is an execution α of ΠA_i such that $\beta = trace(\alpha)$ and $\alpha_i = \alpha|A_i$ for every i.
- If $\beta|A_i \in traces(A_i)$ for every i then $\beta \in traces(\Pi A_i)$.

Composition: Basic results

Theorem 3: Substitutivity

- Suppose A_i and A'_i have the same external signature, and $traces(A_i) \subseteq traces(A_i')$ for every i.
  - A kind of "implementation" relationship.
- Then $traces(\Pi A_i) \subseteq traces(\Pi A_i')$ (assuming compatibility).

Proof:

- Follows from trace pasting and projection, Theorems 1 and 2.

Other operations on I/O automata

- Hiding
  - Reclassify some output actions as internal.
  - Hides internal communication among components of a system.
- Renaming
  - Change names of some actions.
  - Action names are important for specifying component interactions.
  - E.g., define a "generic" automaton, then rename actions to define many instances to use in a system.
    - As we did with channel automata.

Fairness

Fairness

- Task T (a set of actions) corresponds to a "thread of control".
- Used to define "fair" executions: a task that is continuously enabled eventually takes a step.
- Tasks are used to state and prove liveness properties, e.g., that something eventually happens, like an algorithm terminating.

- Formally, an execution (or fragment) α of A is fair to task T if one of the following holds:
  - α is finite and T is not enabled in the final state of α.
  - α is infinite and contains infinitely many events in T.
  - α is infinite and contains infinitely many states in which T is not enabled.
- Execution of A is fair if it is fair to all tasks of A.
- Trace of A is fair if it is the trace of a fair execution of A.

Example

- Channel
  - Only one task (all receive actions).
  - A finite execution of Channel is fair iff $queue$ is empty at the end.
  - Q: Is every infinite execution of Channel fair?
- Consensus process
  - Separate tasks for sending to each other process, and for output.
  - Means it "keeps trying" to do these forever.

Fairness and composition

- Fairness "behaves nicely" with respect to composition---results analogous to non-fair results:

- Theorem 4: Projection
  - If $\alpha \in fairexecs(\Pi A_i)$ then $\alpha|A_i \in fairexecs(A_i)$ for every i.
  - If $\beta \in fairtraces(\Pi A_i)$ then $\beta|A_i \in fairtraces(A_i)$ for every i.

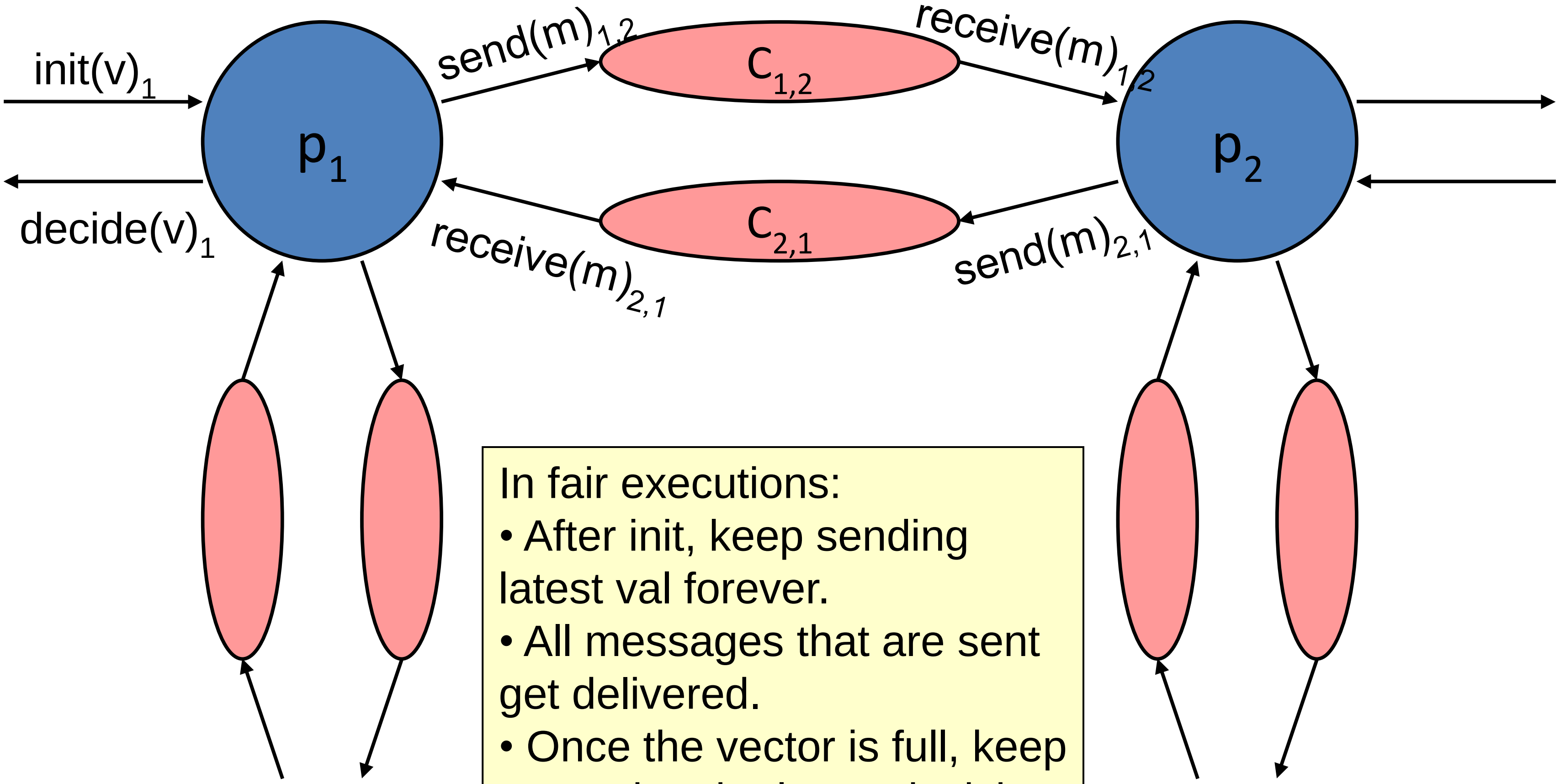
- Theorem 5: Pasting

  Suppose β is a sequence of external actions of ΠA_i.
  - If $\alpha_i \in fairexecs(A_i)$ and $\beta|A_i = trace(\alpha_i)$ for every i, then there is a fair execution α of ΠA_i such that β = trace(α) and $\alpha_i = \alpha|A_i$ for every i.
  - If $\beta\ |A_i \in fairtraces(A_i)$ for every i then $\beta \in fairtraces(\Pi A_i)$.

Fairness and composition

- Theorem 6: Substitutivity
  - Suppose A_i and A'_i have the same external signature, and $fairtraces(A_i) \subseteq fairtraces(A_i')$ for every i.
    - Another kind of "implementation" relationship.
  - Then $fairtraces(\Pi A_i) \subseteq fairtraces(\Pi A_i')$.

Composition of channels and consensus processes



In fair executions:

- After init, keep sending latest val forever.
- All messages that are sent get delivered.
- Once the vector is full, keep outputting the latest decision.

Properties and Proof Methods

- Compositional reasoning
- Invariants
- Trace properties
- Simulation relations

Compositional reasoning

- Use Theorems 1-6 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 inter-dependent 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 (two different, alternative notions, depending on whether we want to consider fairness):
  - $extsig(A) = sig(P)$ and $traces(A) \subseteq traces(P)$
  - $extsig(A) = sig(P)$ and $fairtraces(A) \subseteq traces(P)$

Safety and liveness

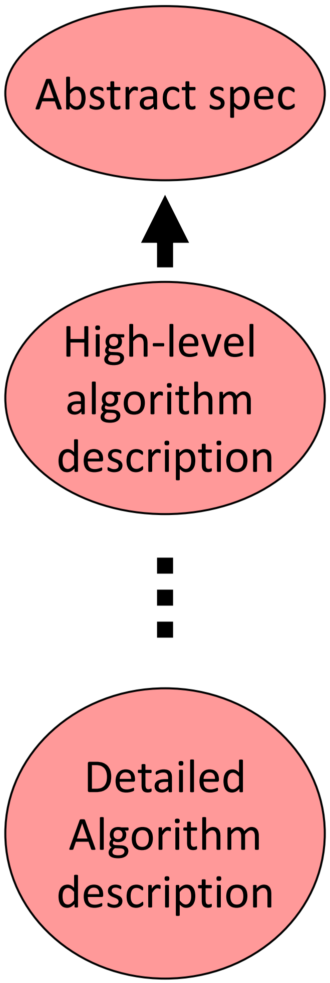
- Safety property: "Bad" thing doesn't happen:
  - Nonempty (null trace is always safe).
  - Prefix-closed: Every prefix of a safe trace is safe.
  - Limit-closed: Limit of sequence of safe traces is safe.
- Liveness property: "Good" thing happens eventually:
  - Every finite sequence over $acts(P)$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

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
  - $extsig(A) = extsig(B)$
  - $traces(A) \subseteq traces(B)$

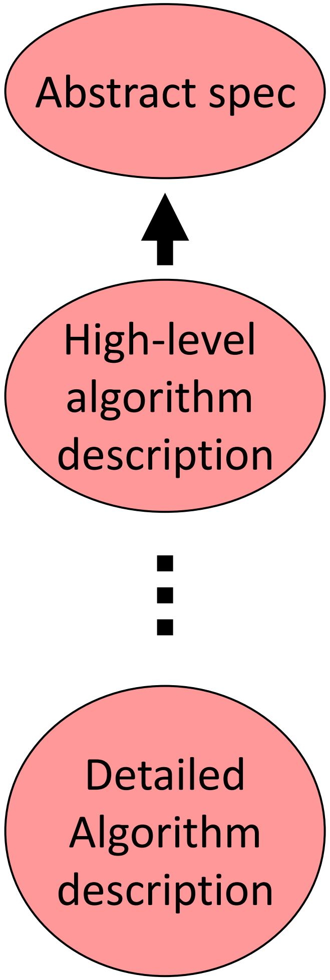
Hierarchical proofs

- 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.



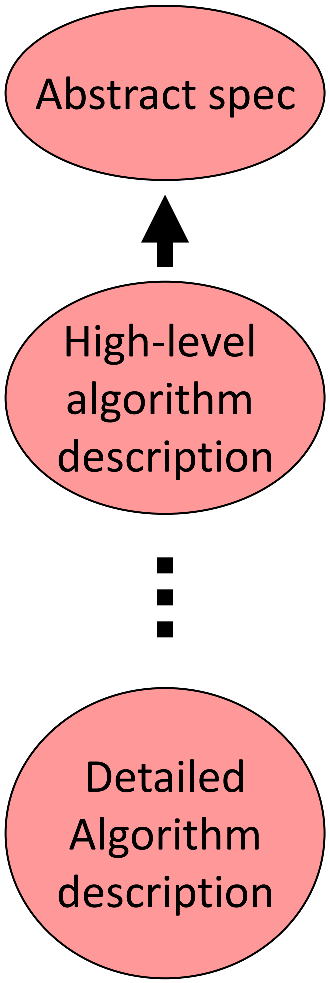
Hierarchical proofs

- 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

$$\begin{tikzcd}[column sep=huge, row sep=huge]
s_B \arrow[r, dotted, "\beta"] \arrow[d, dash, "R"] & s'_B \arrow[d, dash, dotted, "R"] \\
s_A \arrow[r, "\pi"] & s'_A
\end{tikzcd}$$

- 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.

$$s_{0,A} \xrightarrow{\pi_1} s_{1,A} \xrightarrow{\pi_2} s_{2,A} \xrightarrow{\pi_3} s_{3,A} \xrightarrow{\pi_4} s_{4,A} \xrightarrow{\pi_5} s_{5,A}$$

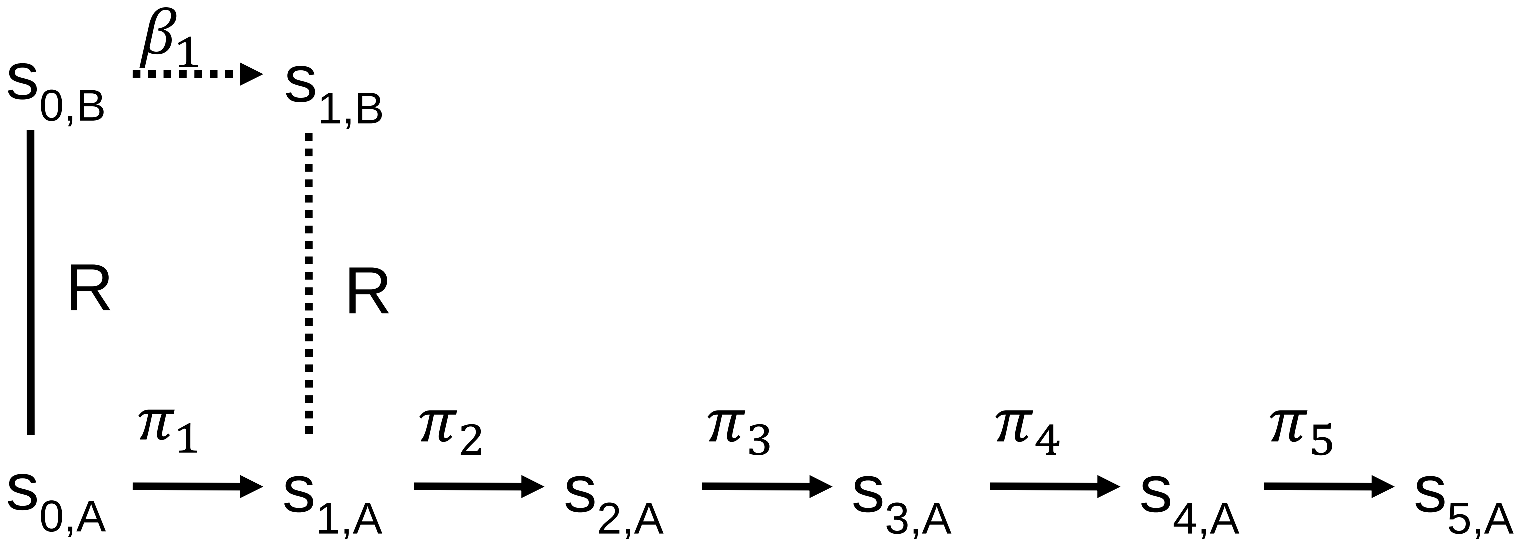
Simulation relations

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

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

Simulation relations

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



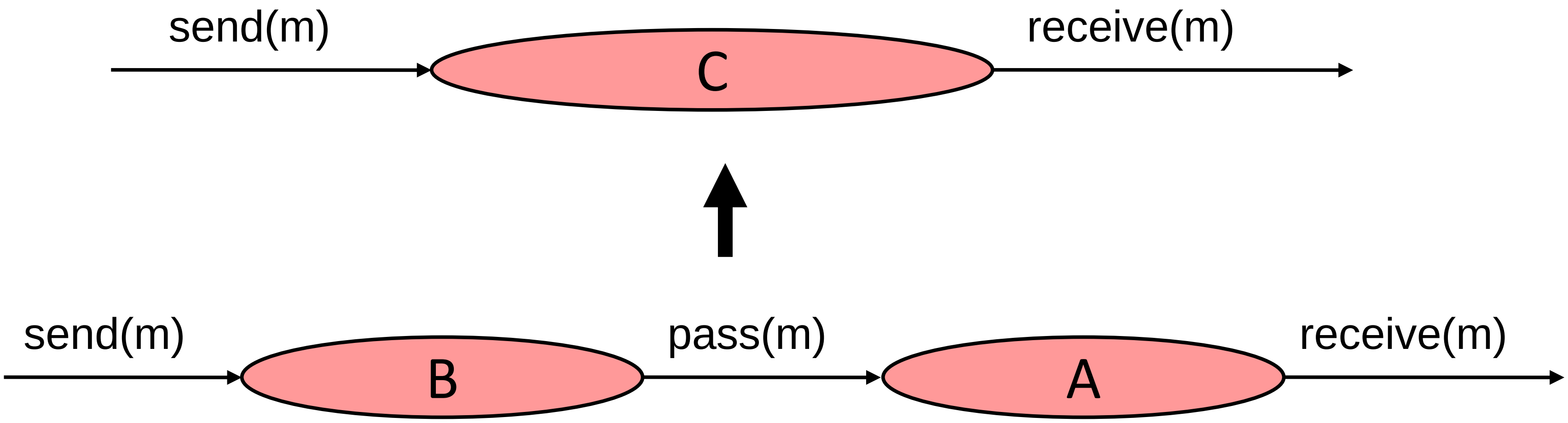
Simulation relations

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

$$
\begin{tikzcd}
s_{0,B} \arrow[r, dotted, "\beta_1"] \arrow[d, dash, "R"] & s_{1,B} \arrow[r, dotted, "\beta_2"] \arrow[d, dash, "R"] & s_{2,B} \arrow[r, dotted, "\beta_3"] \arrow[d, dash, dotted, "R"] & s_{3,B} \arrow[r, dotted, "\beta_4"] \arrow[d, dash, dotted, "R"] & s_{4,B} \arrow[r, dotted, "\beta_5"] \arrow[d, dash, dotted, "R"] & s_{5,B} \arrow[d, dash, dotted, "R"] \\
s_{0,A} \arrow[r, "\pi_1"] & s_{1,A} \arrow[r, "\pi_2"] & s_{2,A} \arrow[r, "\pi_3"] & s_{3,A} \arrow[r, "\pi_4"] & s_{4,A} \arrow[r, "\pi_5"] & s_{5,A}
\end{tikzcd}
$$

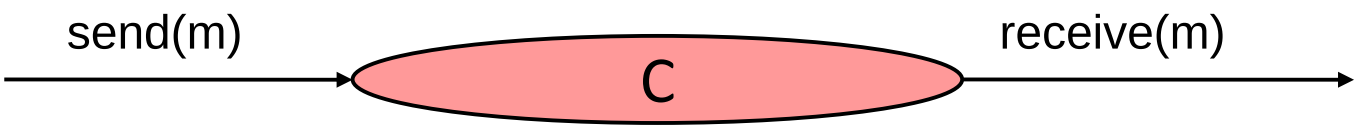
Example: Channels

- Show two channels implement one.



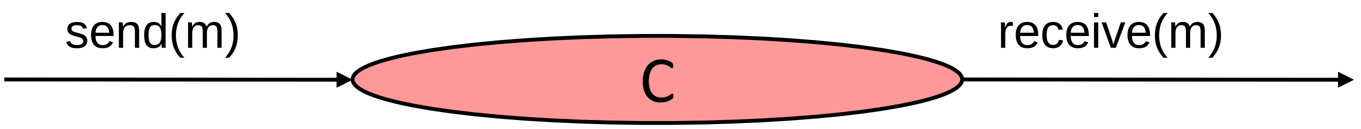
- Rename some actions.
- Let $D = hide_{\{pass(m)\}} A \times B$.
- Show that $traces(D) \subseteq traces(C)$.

Recall: Channel automaton



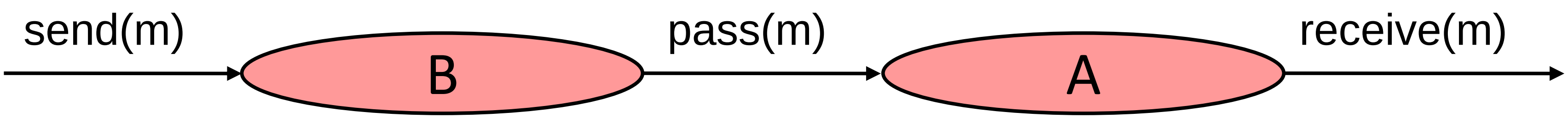
- Reliable unidirectional FIFO channel.
- sig
  - Input actions: $send(m), m \in M$
  - output actions: $receive(m), m \in M$
  - No internal actions
- $states$
  - $queue$: FIFO queue of M, initially empty

Channel automaton



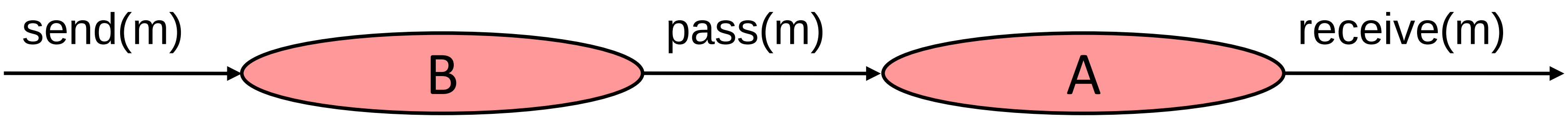
- $trans$
  - $send(m)$
    - effect: add m to $queue$
  - $receive(m)$
    - precondition: $m = head(queue)$
    - effect: remove head of $queue$
- $tasks$
  - All $receive$ actions in one task

Composing two channel automata



- Output of B is input of A
  - Rename $receive(m)$ of B and $send(m)$ of A to $pass(m)$.
- Claim $D = hide_{\{pass(m)\}}\ A \times B$ implements 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 $s.B.queue$.
- Proof that R is a simulation relation:
  - Start condition: All queues are empty, so start states correspond.
  - Step condition: Define "step correspondence":

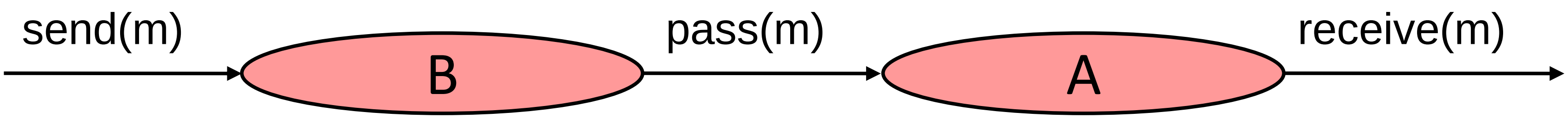
Composing two channel automata



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

- Step correspondence:
  - For each step $(s, \pi, s') \in trans(D)$ 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.

Composing two channel automata



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

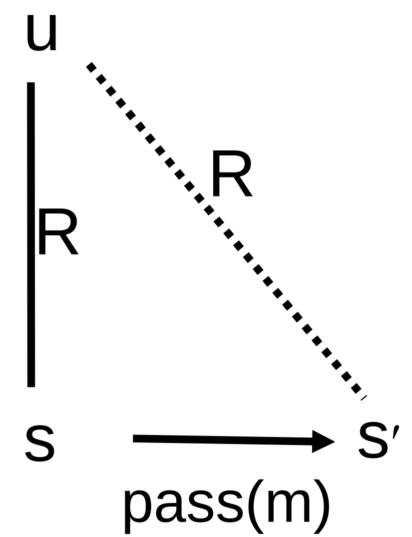
s R u iff u.queue is concatenation of s.A.queue and s.B.queue

- 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: $\pi = \ pass(m)$
  - 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 $s.B.queue.$
    - The concatenation of the queues is unchanged as a result of this step, so also $u.queue$ is the concatenation of $s'.A.queue$ and $s'.B.queue.$



Next lecture

- A bit more on safety and liveness properties.
- Then, basic asynchronous network algorithms:
  - Leader election
  - Breadth-first search
  - Shortest paths
  - Spanning trees.
- Reading:
  - Chapters 14 and 15