6.852: Distributed Algorithms
Fall, 2015

Lecture 22

Today's plan

- Asynchronous shared memory model vs. asynchronous network model
- Consensus in asynchronous networks
- The Paxos consensus protocol
- Reading:
  - Chapter 17
  - [Lamport] The Part-Time Parliament (The Paxos paper)
- Next time:
  - Failure detectors
  - Reading:
    - [Chandra, Toueg] Unreliable FDs for reliable distributed systems
    - [Cornejo, Lynch, Sastry] Asynchronous FDs
    - [Pike, Song, Sastry] FDs for Dining Philosophers
    - [Sastry, Pike, Welch] Weakest FD for Wait-Free Dining Philosophers
    - [Chandra, Hadzilacos, Toueg] Weakest FD for Consensus
    - [Lynch, Sastry] Weakest Asynchronous FD for Consensus

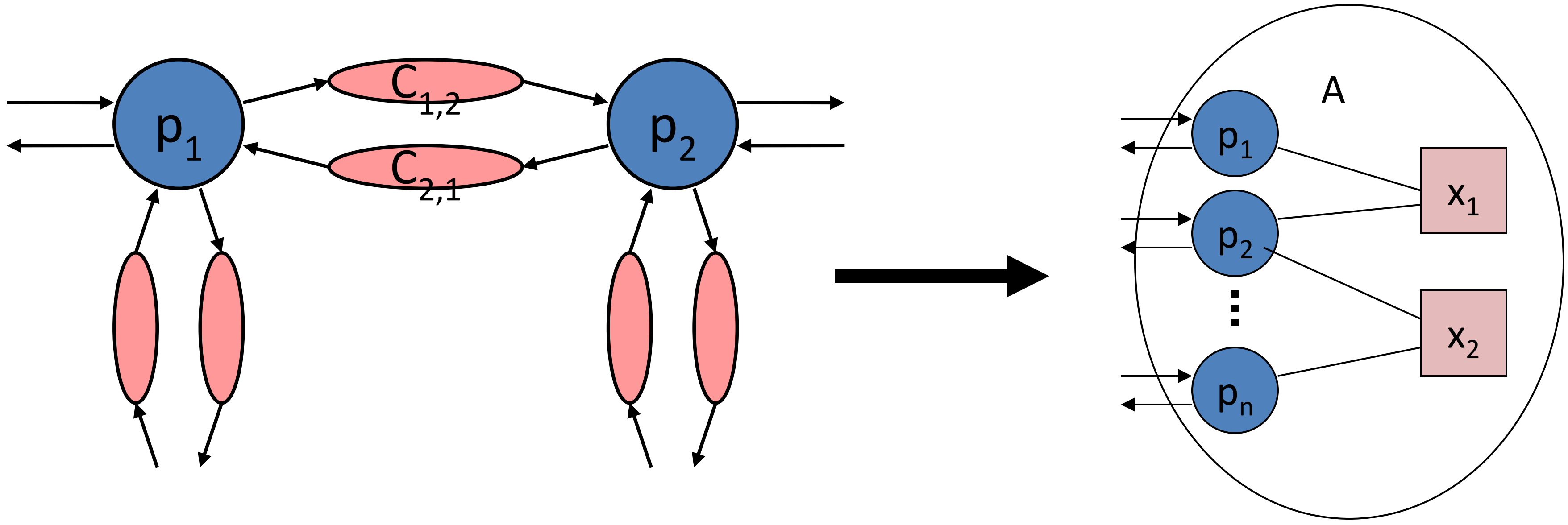
Shared memory vs. networks

- Simulating shared memory in distributed networks:
  - A popular method for simplifying distributed programming.
  - Distributed Shared Memory (DSM).
  - Easy if there are no failures.
  - Possible if $n > 2f$; impossible if $n \leq 2f$.
  - [Attiya, Bar-Noy, Dolev] fault-tolerant algorithm.
- Simulating networks using shared memory:
  - Easier, because shared memory is "more powerful".
  - Works for any number of processes and failures.
  - Useful mainly for lower bounds, impossibility results.
    - Carry over impossibility results for the shared memory model to the network model
    - E.g., for fault-tolerant consensus.

Paxos Consensus Algorithm [Lamport]

- A fault-tolerant consensus algorithm for distributed networks.
- Can use it to implement a fault-tolerant replicated state machine (RSM), to produce the appearance of centralized shared memory, for any data types, in a distributed network.
- Generalizes Lamport’s timestamp-based non-fault-tolerant RSM results.
- Consensus algorithm uses ideas from [Dwork, Lynch, Stockmeyer 88].

Simulating networks using shared-memory systems



Simulating networks using shared-memory systems

- Easy transformation, because the shared-memory model is more powerful:
  - Has reliable, instantaneously-accessible shared memory.
  - No delays as for channels.
- Transformation preserves fault-tolerance, even for $f \geq n/2$.
- Assume an asynchronous network system A for a directed graph network.
  - $stop_i$ event disables P_i and has no effect on channels.
- Design an asynchronous read/write shared-memory system B that simulates A as follows:
  - For any execution α of the shared-memory system $B \times U$, there is an execution α′ of the network system $A \times U$ such that:
    - $\alpha | U = \alpha' | U$.
    - $stop_i$ events occur for the same locations i in α and α‘ .
    - If α is fair then α’ is also fair.

An easy algorithm

- Replace channel $C_{i,j}$ with a 1-writer, 1-reader shared variable $x(i,j)$, written by i, read by j.
- $x(i,j)$ contains a queue of messages, initially empty.
- Process i adds messages, never removes any.
- Process i simulates automaton P_i, step by step.
  - To simulate $send(m)_{i,j}$, process i adds m to $x(i,j)$.
    - Does this using a $write$, by remembering what it wrote earlier.
  - Meanwhile, process i keeps checking its incoming variables $x(j,i)$, looking for new messages.
    - Does this by remembering what it read earlier.
    - When it finds a new message, process i handles it just as P_i would.

Pseudocode, for process i

- State variables
  - $pstate$, a state of P_i
  - $sent(j)$ for each out-neighbor j, a sequence of M, initially empty
  - $rcvd(j), processed(j)$ for each in-neighbor j, a sequence of M, initially empty
- Transitions
  - $send(m, j)_i$:
    - pre: $send(m)_{i,j}$ enabled in $pstate$
    - eff: append m to $sent(j)$; $x(i, j) \coloneqq sent(j)$;
      update $pstate$ as for $send(m)_{i,j}$
  - $receive(m, j)_i$
    - pre: true
    - eff: $rcvd(j) := x(j, i)$;
      update $pstate$ using messages in $rcvd(j) - processed(j)$;
      $processed(j) \coloneqq rcvd(j)$
  - All others: As for P_i, using $pstate$

Theorem and Corollary 1

- Theorem: This simulation produces an asynchronous shared-memory system B that simulates A, in the sense that, for any execution α of the shared-memory system $B \times U$, there is an execution α' of the network system $A \times U$ such that:
  - $\alpha \mid U = \alpha' \mid U$.
  - $stop_i$ events occur for the same i in α and α'.
  - If α is fair then α' is also fair.
- Corollary 1: Consensus is impossible in asynchronous networks, with 1 stopping failure [Fischer, Lynch, Paterson].
- Proof:
  - If such an algorithm existed, we could simulate it in an asynchronous read/write shared-memory system using the simulation just given.
  - This would yield a 1-fault-tolerant consensus algorithm for (1-writer 1-reader) read/write shared memory.
  - We know this is impossible [Loui, Abu-Amara].

Corollary 2

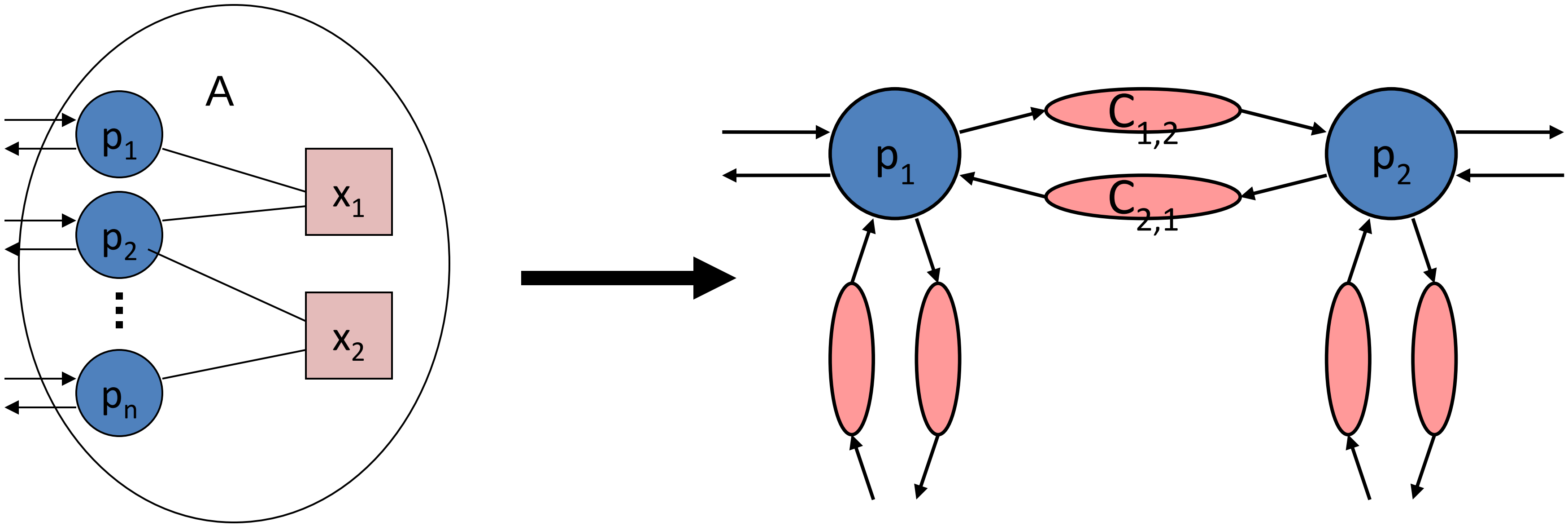
- Corollary 2: Consensus is impossible in asynchronous broadcast systems, with 1 stopping failure [Fischer, Lynch, Paterson].
- Asynchronous broadcast system: A process can put a message in all its outgoing channels in one step, and all are guaranteed to eventually be delivered.
  - That is, a process cannot fail in the middle of a broadcast.
- Proof:
  - If such an algorithm existed, we could simulate it in an asynchronous shared-memory system using a simple extension of the simulation above.
  - Extension uses 1-writer multi-reader shared variables to represent the broadcast channels.
  - This would yield a 1-fault-tolerant consensus algorithm for 1-writer multi-reader read/write shared memory.
  - We already know this is impossible [Loui, Abu-Amara].
- Q: Is this counterintuitive?

Is this counterintuitive?

- Corollary 2: Consensus is impossible in asynchronous broadcast systems, with 1 stopping failure [Fischer, Lynch, Paterson].
- Asynchronous broadcast system: Process can put a message in all its outgoing channels in one step, and all are guaranteed to eventually be delivered.

- Recall that in the synchronous model, impossibility results for consensus depended on processes failing in the middle of a broadcast.
- Now every broadcast is completed, and guaranteed to be delivered everywhere.
- But we still get impossibility, this time because of asynchrony.

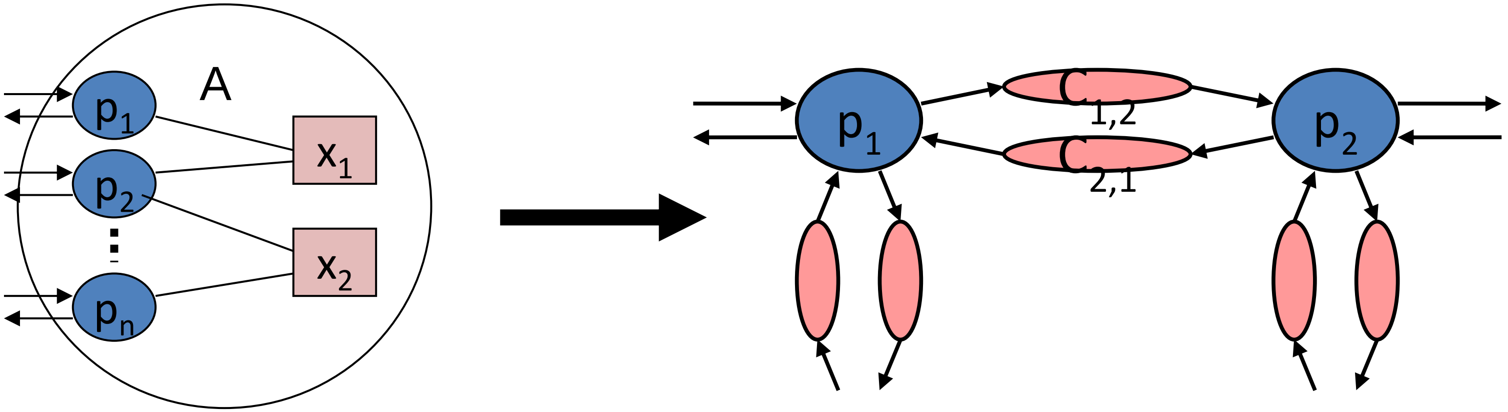
Simulating shared-memory systems using networks



Simulating shared memory in distributed networks

- Can be used to simplify distributed programming.
- Non-fault-tolerant algorithms:
  - Single-copy
  - Multi-copy
  - Majority voting
- Fault-tolerant algorithms:
  - [Attiya, Bar-Noy, Dolev] algorithm for $n > 2f$.
  - Impossibility result for $n \leq 2f$.

Non-fault-tolerant simulation of shared memory in distributed networks



Simulating shared memory in networks

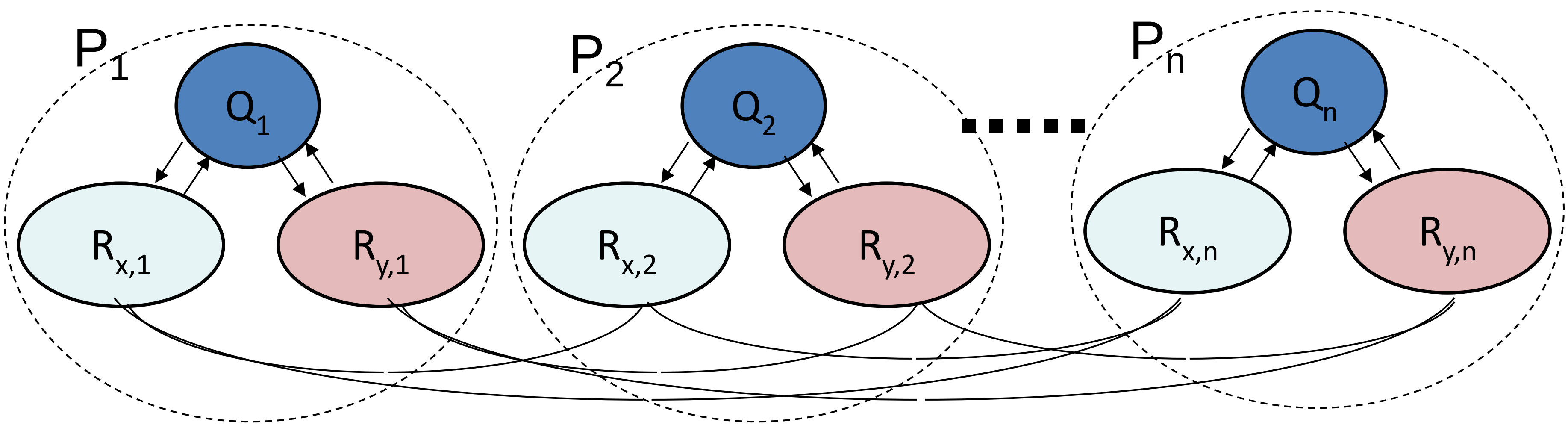
- Assume shared memory system A:
  - Ports $1, \dots, n$
  - User U_i interacts with process i on port i
  - Technical restriction: For each i, it's always either the user's turn, or process's turn to take steps (not both).
    - So we could replace shared variables with atomic object implementations without introducing new behavior.
- Design asynchronous network system B:
  - Same ports/user interface.
  - Processes and FIFO reliable channels.
  - For any execution α of the network system $B \times U$, there is an execution α' of the shared memory system $A \times U$ such that:
    - $\alpha \mid U = \alpha' \mid U$.
    - $stop_i$ events occur for the same i in α and α'.
    - If α is fair then α' is also fair (this condition will change for the FT case).

Single-copy simulation

- Non-fault-tolerant.
- Works for any object types.
- Handle each shared variable independently.
- Locate each shared variable x at some known process, $owner(x)$.
- Automaton P_i simulates process i of A, step by step.
  - All actions other than shared-memory accesses are as before.
  - To access variable x, P_i sends a message to $owner(x)$ and waits for a response; when response arrives, uses it and resumes the simulation.
  - P_i also handles requests to perform accesses to all variables x for which $i = owner(x)$.
    - Performs on local copy, in one indivisible step.
    - Sends response.

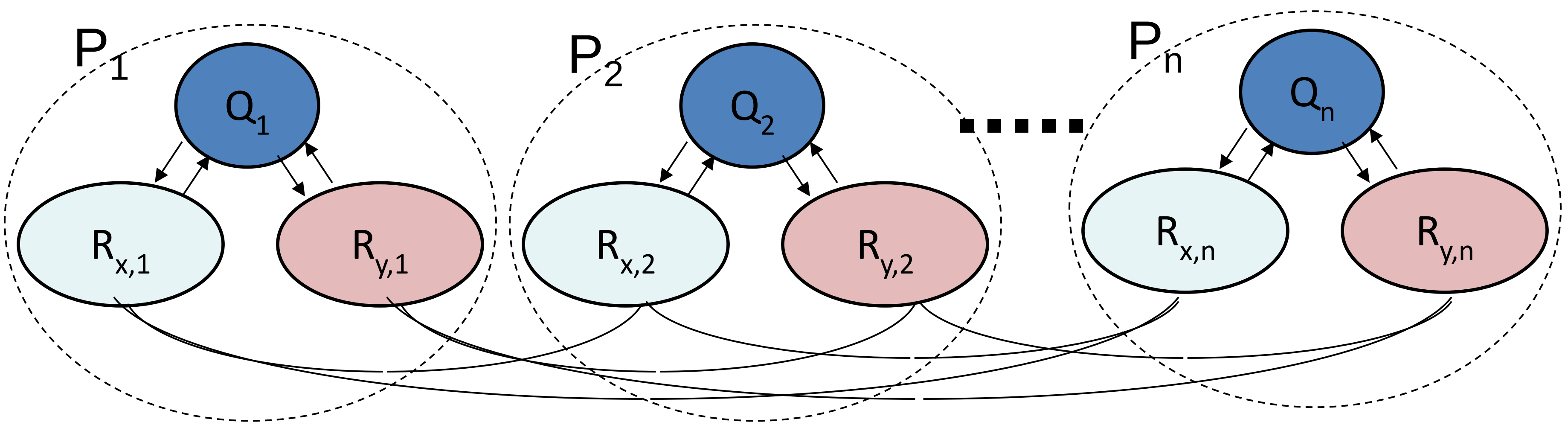
Formally…

- Each automaton P_i is the composition of:
  - Q_i, an automaton that simulates process i of the shared-memory system A,
    - Use same automata as when replacing shared variables by atomic objects.
  - $R_{x\ i}$, for every shared variable x, an automaton that manages variable x and its requests.



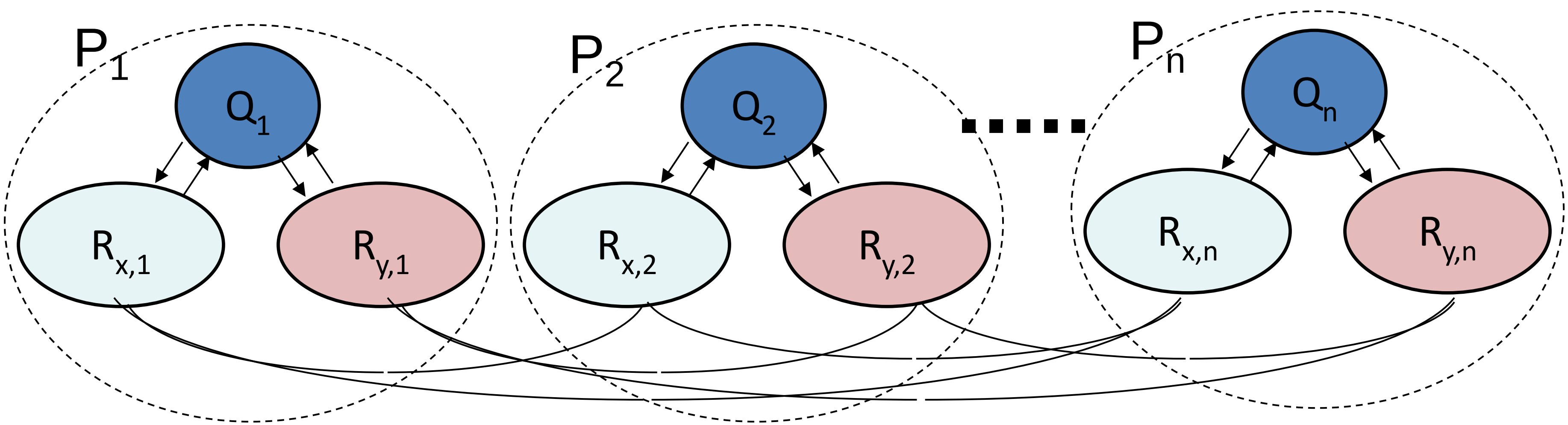
Formally…

- Q_i and $R_{x,i}$ interact using invocations, responses on object x.
- For each x, the $R_{x,i}$ automata communicate over FIFO send/receive channels, and cooperate to implement an atomic object for x.
- $owner(x)$: Collects requests via local invocations and messages from others, processes on them on its local copy.
- Non-owners: Send invocation to $owner(x)$, await response.



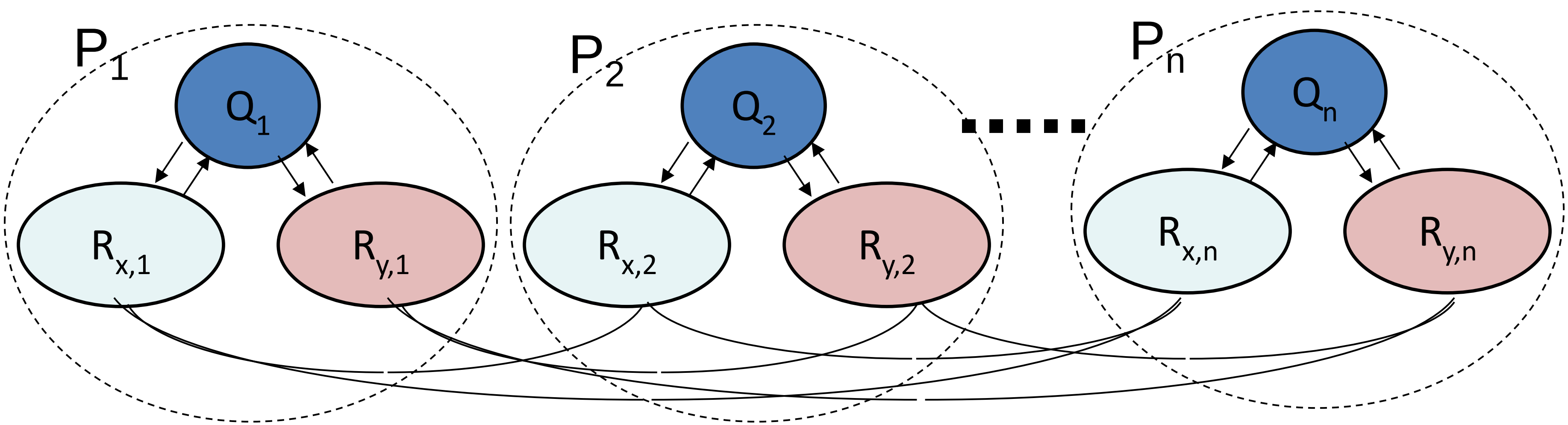
Formally…

- Correctness: Obvious, since the $R_{x,i}$ automata and the channels between them implement an atomic object for x.
  - Serialization point for an operation: When the owner performs the operation on the local copy.
- Fault-tolerance: None. Any process failure kills its owned variables, which can block everyone.



Remarks

- Optimization: Avoid busy-waiting on a remote shared variable: Send one request, let owner notify sender when the value of the variable changes.
- Q: Where are the best locations for the copies?



Multi-copy simulation

- Not fault-tolerant.
- Just read/write objects.
- Handle each shared variable independently.
- Locate each shared variable x at some known collection of processes, $owners(x)$.
- How P_i accesses variable x:
  - READ: Read any copy.
  - WRITE: Write all copies, asynchronously, in any order.
  - "Read-one, write-all."
- Can be faster than single-copy, most of the time, if reading is much more common than writing.
  - E.g., in peer-to-peer systems, sharing files.
- But, without some constraints, we get consistency issues…

Bad examples

- Multi-writer, inconsistent order of WRITEs
  - P_1 and P_2 WRITE the same shared variable x.
  - $owners(x) = \{P_3, P_4\}$.
  - P_1 and P_2 send write request messages to P_3 and P_4.
  - P_3 and P_4 receive the write requests in different orders, so end up with different final values.
  - Later READs may get either value, inconsistent.
- Single-writer, inconsistent READs
  - $owners(x) = \{P_2, P_3\}$.
  - Writer P_1 sends write request messages to P_2 and P_3.
  - Message arrives at P_2, which writes its local copy.
  - Then a READ gets processed at P_2, getting the new value.
  - Later, a READ happens at P_3, getting the old value.
  - Then P_1's write message arrives at P_3, which writes its local copy.
  - The READs are sequential, but are concurrent with the WRITE.
  - Out-of order READ behavior is not allowed by atomic R/W objects.

Multi-copy simulation

- So we need some more clever protocols…
- Idea: Use atomic transactions:
- E.g., to do a WRITE, perform all the writes to all copies as a single atomic transaction, so that they appear to occur instantaneously, as far as READ operations can tell.
- Can implement such a transaction using 2-phase locking:
  - Phase 1: Lock all copies and write them.
  - Phase 2: Release all the locks.
- Must solve deadlock problems for lock acquisition.
- Works because serialization point for WRITE can be placed at a “lock point”, when all the locks are held.

Majority-voting algorithms

- Not fault-tolerant.
- Just read/write objects.
- Handle each shared variable independently.
- Locate each shared variable x at some known collection of processes, $owners(x)$.
- How P_i accesses variable x:
  - READ: Read from a majority of copies.
  - WRITE: Write to a majority of copies.
- Run each READ or WRITE operation as an atomic transaction, using an underlying concurrency-control strategy like 2-phase locking.
- More precisely:...

Majority-voting algorithms

- Each copy of x includes an integer tag, initially 0, as well as a $value$ for x.
- How P_i accesses variable x:
  - Performs an atomic transaction, implemented by 2-phase locking.
  - READ:
    - Read from a majority of copies.
    - Return the value associated with the largest tag.
  - WRITE(v):
    - First do an embedded-read of a majority of copies.
    - Determine the largest tag t.
    - Write $(v, t + 1)$ to a majority of copies.
  - Each READ or WRITE appears to be instantaneous, because the operations are implemented as transactions.

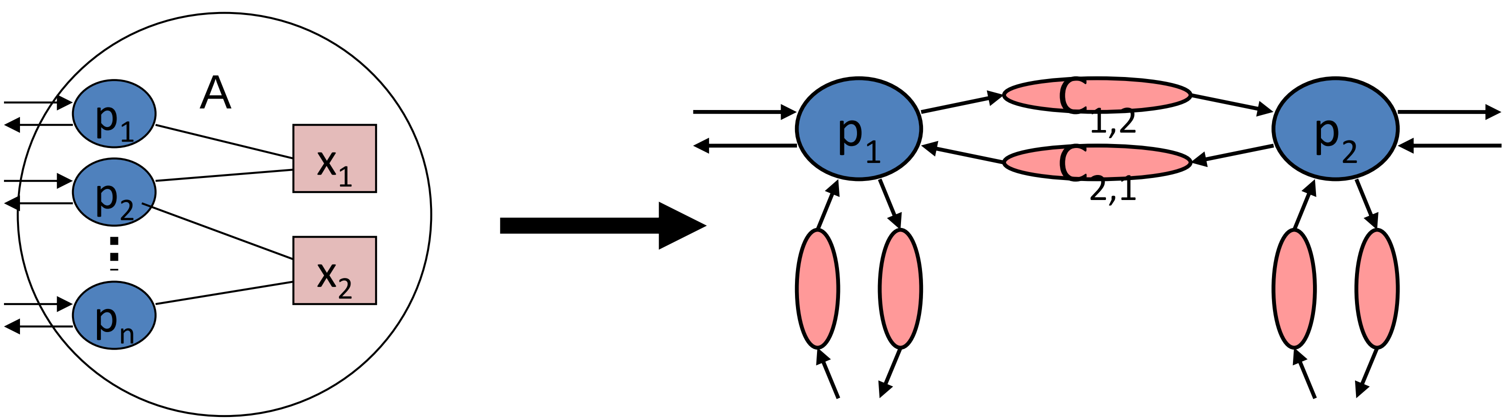
Why does this work?

- To see that this implements an atomic Read/Write object:
- Choose serialization points for the READ and WRITE operations to be the serialization points for their transactions.
  - These are guaranteed by the transaction implementation, e.g., lock points for 2-phase locking.
- Show that the operations behave as if they occurred at their transactions' serialization points:
  - WRITE operations are assigned $tags$ 1,2, … in order of their transactions' serialization points.
  - READ or embedded-read obtains the largest tag that was written by a WRITE operation serialized before it (0 if there are none), together with the associated value for the object.
  - These statements both depend on the key fact that each READ or embedded-read reads a majority of the copies, the largest tag gets written to a majority of the copies, and all majorities intersect.

Remarks

- This is still not fault-tolerant:
  - Because standard transaction implementations like 2-phase locking aren't fault-tolerant.
  - A process that fails while holding locks "kills" the locked objects.
- Can generalize majorities to quorum configurations.
- Quorum configuration:
  - A set of read-quorums, finite subsets of process indices,
  - A set of write-quorums, finite subsets of process indices, such that
  - $R \cap W \neq \emptyset$ for every read-quorum R and write-quorum W.
- READ operation accesses any read-quorum.
- WRITE operation accesses both a read-quorum and a write-quorum (in its two phases).
- Allows tuning for smaller read-quorums, which can speed up READs.
  - E.g., read-one, write-all.

Fault-tolerant simulation of shared memory in distributed networks



Fault-tolerant simulation of shared memory in distributed networks

- [Attiya, Bar-Noy, Dolev], 2011 Dijkstra Prize
- Tolerates f stopping failures, works provided $n > 2f$.
- Assume reliable channels.
- Just for read/write objects, in fact, 1-writer multi-reader objects (not hard to extend to MWMR, see HW).
- Modeling failures:
  - Use a $stop_i$ input at each external port (of the shared-memory system A, or of the network system B).
  - $stop_i$ disables all locally-controlled actions of process i, in either system.
  - Does not affect messages in transit (in system B).

- Q: What is guaranteed by the [ABD] simulation?

[ABD] Guarantees

- Tolerates f stopping failures, for $n > 2f$.
- For any execution α of network system $B \times U$, there is an execution α' of shared-memory system $A \times U$ such that:
  - $\alpha \mid U = \alpha' \mid U$ and
  - $stop_i$ events occur for the same i in α and α'.
- Moreover, if α is fair and contains $stop_i$ events for at most f different ports, then α' is also fair.
- This means that in the simulated shared-memory execution, all non-failed processes continue taking steps---the failed processes in the network system don't introduce any new blocking.

- Assume shared-memory system A has only 1-writer multi-reader read/write shared variables.

[ABD] algorithm

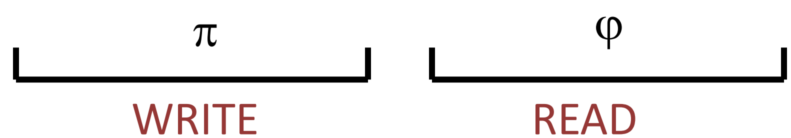
- Tolerates f stopping failures, for n > 2f.
- Implement an atomic object for each shared variable x, then combine.
- No transactions, no synchronization. Operations interleave at very fine granularity.
- Each process keeps:
  - val, a value for x, initially v_0
  - tag, initially 0
- P_1 does WRITE(v):
  - Let t be the first unused tag (P_1 knows this because it's the only writer, hence the only process generating tags).
  - Set local variables to (v, t).
  - Send message ("$write$", v, t) to all other processes.
  - When anyone receives such a message:
    - Updates local variables to (v, t) if $t > tag$.
    - In any case, sends ack to P_1.
  - When P_1 knows a majority have received its (v, t), returns ack.

[ABD] algorithm

- P_i does a READ:
  - Read own copy; send $read$ messages to all other processes.
  - When anyone receives this message, responds with its current (val, tag).
  - When P_i has heard from a majority, prepares to return v from the (v, t) pair it has seen with the largest t.
  - However, before returning v, P_i propagates this (v, t).
    - As in [Vitanyi, Awerbuch] algorithm.
    - For the same reason (prevent out-of-order reads).
  - When anyone receives this propagated (v, t):
    - Updates local variables to (v, t) if $t > tag$.
    - Sends ack to P_i.
  - When P_i knows a majority have received (v, t), returns v.

Correctness of [ABD] atomic object algorithm

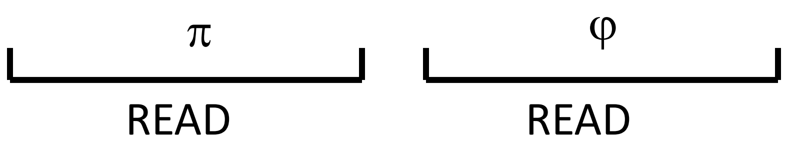
- Well-formedness (yes)
- f-failure termination, for $n > 2f$ (yes)
- Atomicity:
  - Algorithm is similar to [Vitanyi, Awerbuch], can use a similar proof, based on partial order lemma.
  - Define the partial order by:
    - Order WRITEs by tags.
    - Order READ right after WRITE whose value it gets.
  - Condition 2: If operation π finishes before operation φ starts, then φ is not ordered before π in the partial order.
  - Consider cases, based on operation types.
  - Case 1:



  - Because majorities intersect, φ gets a $tag \geq$ the tag written by π.
  - So φ is ordered after π.

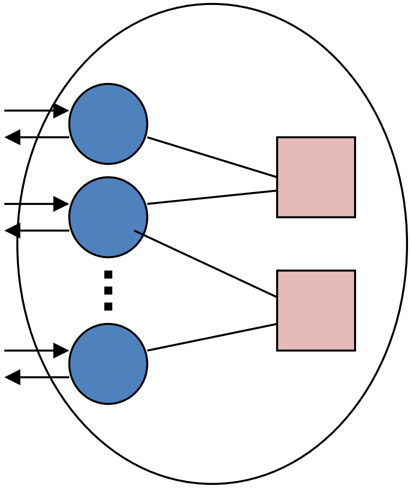
Atomicity, cont'd

- Partial order:
  - Order WRITEs by tags.
  - Order READ right after WRITE whose value it gets.
- Condition 2: If π finishes before φ starts, then φ is not ordered before π in the partial order.
- Case 2:

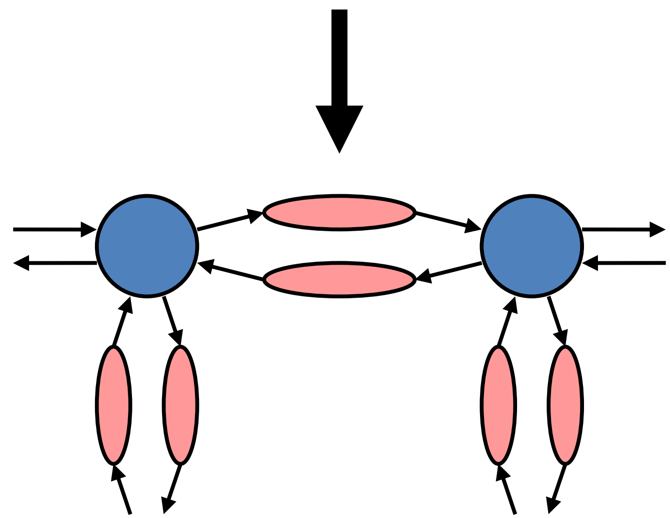


- Then φ gets a $tag \geq$ the tag obtained by π, because of propagation and majority intersection.
- So φ is not ordered before π.
- Other cases: Similar, LTTR.

[ABD] for simulating shared memory



- Use the [ABD] atomic object algorithm to construct a distributed simulation of any fault-tolerant shared-memory algorithm A that uses 1-writer multi-reader shared variables: Just replace shared vars by [ABD] atomic object implementations.
- Guarantees:
  - For any execution α of network system $B \times U$, there is an execution α' of shared-memory system $A \times U$ such that:
    - $\alpha \mid U = \alpha' \mid U$ and
    - $stop_i$ events occur for the same i in α and α'.
  - Moreover, if α is fair and contains $stop_i$ events for at most f $(< n/2)$ different ports, then α' is also fair.
- That is, we have a correct simulation, provided that there are at most f failures in the network system B.



Corollaries

- Guarantees:
  - For any execution α of network system $B \times U$, there is an execution α' of shared-memory system $A \times U$ such that:
    - $\alpha \mid U = \alpha' \mid U$ and
    - $stop_i$ events occur for the same i in α and α'.
  - Moreover, if α is fair and contains $stop_i$ events for at most f different ports ($f < n/2$), then α' is also fair.
- Corollary: A wait-free shared-memory atomic snapshot algorithm using 1WmR registers (Chapter 13) can be transformed, using [ABD], to a distributed snapshot algorithm.
- Corollary: The [Vitanyi, Awerbuch] wait-free mWmR register implementation using 1W1R registers can be transformed, using [ABD], to a distributed register implementation.
- Note: The transformed versions are not wait-free, but guarantee only f-failure termination, where $n > 2f$.
  - Since the [ABD] implementation of atomic 1WmR registers tolerates only $f < n/2$ failures, so do the algorithms that use it.

Remarks

- Can generalize majorities to a quorum configuration:
  - Set of read-quorums, set of write-quorums.
  - $R \cap W \neq \varnothing$ for every read-quorum R, write-quorum W.
- Then:
  - A READ operation accesses both a read-quorum and a write-quorum.
  - A WRITE operation accesses just a write-quorum.
- So, we don't improve READ performance by using smaller read-quorums!
- Q: So how can we get faster READ performance?
- A: Optimize to eliminate "most" propagation phases.
  - After a WRITE with tag t completes, or a READ finishes propagating tag t, it is not necessary to propagate tag t anymore.
  - So, an operation that completes t can send messages to everyone saying that t is complete; everyone who receives such a message can mark t as complete.
  - A READ that gets tag t and sees it marked (anywhere) as complete doesn't need to propagate t.

Impossibility of n/2-fault-tolerance

- General "fact" about the distributed network model: nothing interesting can be computed with $\geq n/2$ failures.
- In contrast: There are many interesting wait-free shared-memory algorithms.
- Theorem: In the asynchronous network model with $n = m + p$ processes, no implementation of m-writer p-reader atomic registers guarantees f-failure termination for $f \geq n/2$.
- Proof:
  - By contradiction. Suppose $f \geq n/2$ and we have an algorithm…
- Assume WLOG that:
  - Initial value of implemented register = 0.
  - P_1 is a writer and P_n is a reader.
  - Partition the n processes into two subsets, each with size $\leq f$:
    - $G_1 = \{1, \dots, f\}$
    - $G_2 = \{f + 1, \dots, n\}$.
  - By f-fault-tolerance, even if one entire group fails, the other group must still give correct atomic register responses.

Proof, cont'd

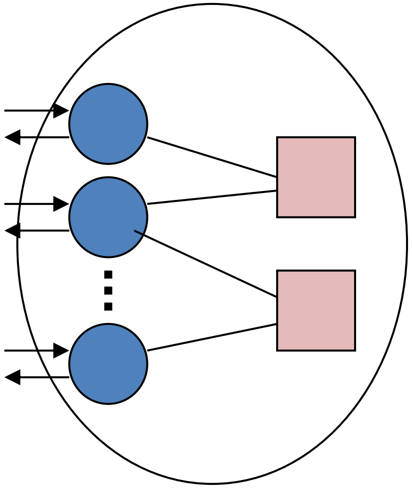
- Theorem: In the asynchronous network model with $n = m + p$ processes, no implementation of m-writer p-reader atomic registers guarantees f-failure termination for $f \geq n/2$.
- Proof, cont'd:
  - Partition the processes into $G_1 = \{1, \dots, f\}, G_2 = \{f + 1, \dots, n\}$.
  - If one group fails, the other must still give atomic register responses.
  - Execution α_1:
    - G_2 processes fail initially.
    - P_1 invokes WRITE(1).
    - WRITE must eventually terminate with response ack.
    - Let α_1' be the portion of α_1 up to the response.
  - Execution α_2:
    - G_1 processes fail initially.
    - P_n invokes READ.
    - READ must eventually terminate with response 0.
    - Let α_2' be the portion of α_2 up to the response.

Proof, cont'd

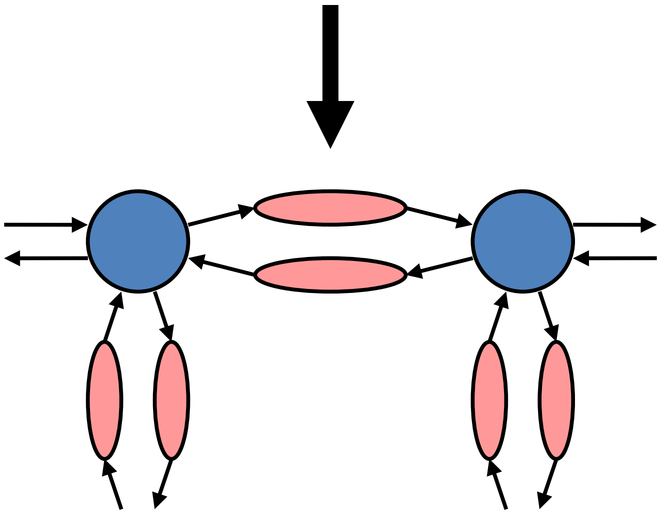
- Execution α_1:
  - G_2 processes fail initially.
  - P_1 invokes WRITE(1).
  - WRITE terminates with ack.
  - Let α_1' be the portion of α_1 up to the ack.
- Execution α_2:
  - G_1 processes fail initially.
  - P_n invokes READ.
  - READ terminates with 0.
  - Let α_2' be the portion of α_2 up to the 0.
- Execution α_3: Paste…
  - No one fails.
  - All the steps of α_1' occur first, including the ack.
  - Then all the steps of α_2' occur, including the response of 0.
  - Meanwhile, all messages between G_1 and G_2 are delayed.
- Activity in α_1' and α_2' is independent, so α_3 is an execution.
- But it is not correct for an atomic register, since the WRITE(1) completes before the start of the READ that returns 0.
- Contradiction.

Simulating shared memory in networks

- This impossibility theorem implies that there is no general simulation of shared-memory systems by networks, preserving f-fault-tolerance, for $f \geq n/2$.
  - Book, p. 567, defines f-simulation, which formalizes “preserving f-fault-tolerance”.



- Proof:
  - If there were, then we could use it to convert a (trivial) wait-free shared-memory implementation of a multi-writer, multi-reader atomic register into an f-fault-tolerant distributed network implementation, $f \geq n/2$.
  - Since the example shows that no such distributed network algorithm exists, neither does such a general simulation.



Remarks

- [ABD] can be extended to dynamically-changing networks:
  - RAMBO (Reconfigurable Atomic Memory for Basic Objects) algorithm [Gilbert, Lynch, Shvartsman] works in dynamic networks.
  - Supports reconfiguration, in addition to reads and writes.
- Q: All the algorithms we have considered emulate shared read/write registers only. What about other data types?
- The situation is very different, because some objects are much more powerful than registers, e.g., CAS objects have the "power of consensus".
- ABD doesn't work.
- Now consider emulating more powerful data objects.
- Start with simpler problem: Consensus in fault-prone networks.
  - We have inherent limitations [FLP], so we must weaken requirements.
  - [Dwork, Lynch, Stockmeyer], failure-detector approaches, Paxos

Fault-Tolerant Agreement in Asynchronous Networks: The Paxos Algorithm

Fault-tolerant agreement in asynchronous networks

- It's impossible to reach agreement in asynchronous networks, even if we assume that at most one failure will occur.
- What if we really need to?
  - For transaction commit.
  - For agreeing on the order in which to perform operations on an emulated shared data object.
  - …
- Some possible approaches:
  - Randomized algorithm [Ben-Or], terminates with high probability.
  - Approximate agreement.
  - Use a failure detector service, implemented by timeouts (next time).

A nice approach: Eventual stability

- Guarantee agreement, validity in all cases.
- Guarantee termination if the system eventually "stabilizes":
  - No more failures, recoveries, message losses.
  - Timing of messages, process steps within "normal" bounds.
- Termination should be fast after system stabilizes.
- Actually, stable behavior need not continue forever, just long enough for computation to terminate.
- This general approach (safety is absolute, liveness depends on stabilization of the behavior of the underlying system) is regarded as the best approach to building practical fault-tolerant distributed data-management systems [Microsoft] [Google] [HP]…

Eventual stability: Some history

- [Dwork, Lynch, Stockmeyer] first presented a consensus algorithm with these properties.
- [Cristian] used a similar approach for group membership algorithms.
- [Lamport, Part-Time Parliament]
  - Introduced the Paxos algorithm.
  - Relationship with [DLS]:
    - Achieves similar guarantees.
    - Paxos allows more concurrency, tolerates some more kinds of failures.
    - Basic strategy for assuring safety similar to [DLS].
    - Paxos has been used as a subroutine in an algorithm to emulate powerful shared memory, which has been engineered for practical use.
  - Background:
    - Paper unpublished for 10 years because of nonstandard style. (Read it!)
    - Eventually published "as is", because others were recognizing its importance and building on its ideas.

Paxos consensus protocol

- Called Single-Decree Synod protocol.
- Assumptions:
  - Asynchronous processes, stopping failures, also recovery.
  - Messages may be lost.
- Lamport’s paper also describes how to cope with disk crashes, where volatile memory is lost (we’ll skip this).
- We’ll present the algorithm in two stages:
  - Describe a very nondeterministic algorithm that guarantees the safety properties (agreement, validity).
  - Constrain it to get termination soon after stabilization.

The "safe" algorithm: Ballots

- Uses ballots, each of which represents an attempt to reach consensus.
- Ballot = (identifier, value) pair.
  - Identifier is an element of Bid, some totally-ordered set of ballot identifiers.
  - Value in $V \cup \{ \perp \}$, where V is the consensus domain.
- Somehow, ballots get started, and get values assigned.
- Processes can vote for, or abstain from, particular ballots.
- Abstention from a ballot is a promise never to vote for it.

Quorums

- The fate of a ballot depends on the actions of quorums of processes on that ballot.
- Quorum configuration:
  - A set of read-quorums, finite subsets of process index set I, and
  - A set of write-quorums, finite subsets of I, such that
  - $R \cap W \neq \emptyset$ for every read-quorum R and write-quorum W.
- Ballot becomes dead if every node in some read-quorum abstains from it.
- A ballot can succeed only if every node in some write-quorum votes for it.

Safe algorithm, centralized version

- Someone can create a new ballot with $Bid\ b$:
  - $makeBallot(b)$
  - Provided no ballot with $Bid\ b$ has yet been created.
  - $val(b)$ is initially undefined ($\perp$).
- A process i can abstain from a set of ballots:
  - $abstain(B, i), B \subseteq Bid$
  - Provided i has not previously voted for any ballot in B.
  - B can be any set of $Bids$, which may or may not be associated with already-created ballots, e.g., all $Bids$ in some range $[b_{min}, b_{max}]$.

Safe algorithm, centralized version

- Assign a value v to a ballot id b, $assign(b, v)$, provided:
  - A ballot with $id = b$ has already been created.
  - $val(b)$ is undefined.
  - v is someone's consensus input.
  - (1) For every $b' \in Bid, b' < b$, either $val(b') = v$ or b' is dead.
- Notes on (1):
  - Recall: b' dead means some read-quorum has abstained from b'.
  - Refers to every $b' \in Bid$, not just created ones.
    - Relies on "set abstentions" above.
- Thus, we can assign a value to a ballot b only if we know it won't ever make b conflict with lower-numbered ballots b'.
- Motivation:
  - Several ballots can be created, can collect votes.
  - More than one might succeed in collecting a write-quorum of votes.
  - That's OK, if they don't have different values.

Safe algorithm, centralized version

- A process i can vote for a ballot b, $vote(b, i)$, if b is a created ballot from which i hasn't abstained.
- A ballot may succeed, $succeed(b)$, if some write-quorum W has voted for it.
- A process can decide on the value that is associated with any successful ballot, $decide(v)$.

Safety properties

- Validity:
  - Immediate. Only initial values ever get assigned to ballots.
- Agreement:
  - Follows from the careful way we avoid assigning different values to ballots that might succeed.
  - Key Invariant: If $val(b) \neq \perp$, $b' \in Bid$, and $b' < b$, then either $val(b') = val(b)$ or b' is dead.
  - Implies that all successful ballots must have the same value.

- Now let's "distribute" this centralized algorithm…

Modifying Condition (1) for assigning ballot values

- Instead of:

  (1) For every $b' \in Bid, b' < b$, either $val(b') = v$ or b' is dead.

  consider:

  (2) Either every $b' \in Bid, b' < b$, is dead, or there exists $b' < b$ with $val(b') = v$, and such that every b'' with $b' < b'' < b$ is dead.

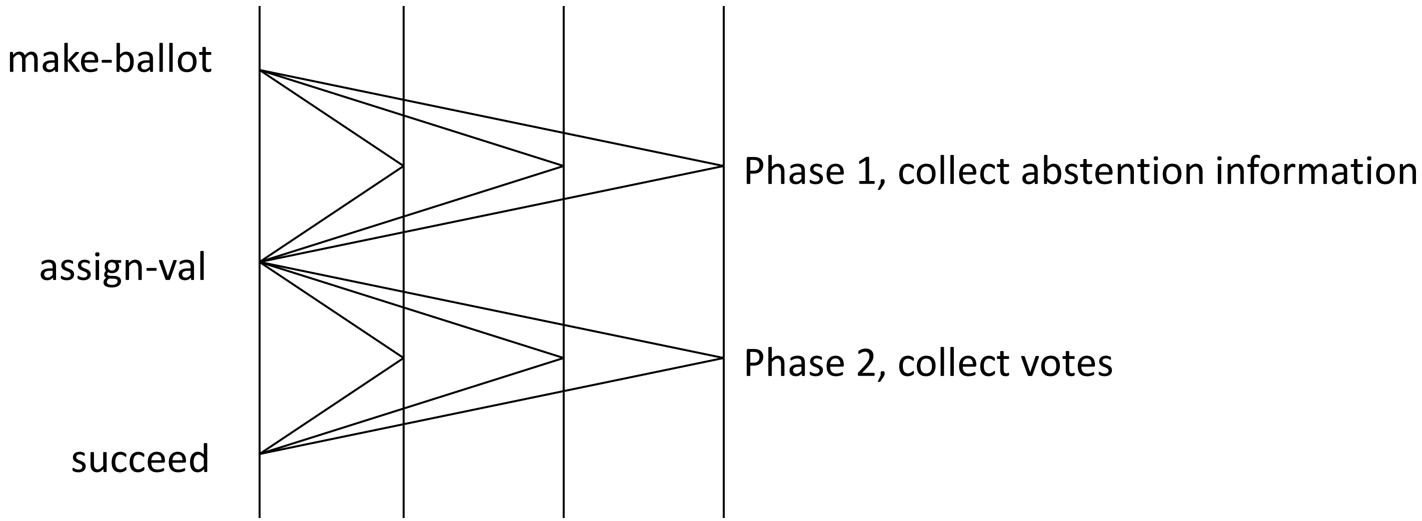
- (2) is easier to ensure.
- (2) implies (1), by an induction on the number of steps in an execution.

Safe algorithm, distributed version

- Any process i can create a ballot, at any time.
  - Use a locally-reserved ballot id.
  - Ballot start is triggered by signal from a separate $BallotTrigger$ service that decides who should start ballots and when, based on monitoring system behavior.
  - Precise choices don't affect the safety properties, so for now, leave them nondeterministic.
- Phase 1:
  - Process i starts a ballot when told to do so by $BallotTrigger$, but doesn't assign a value to it yet.
  - Rather, it first tries to collect enough abstention information for smaller ballots to guarantee condition (2).
  - If/when it collects that, assigns $val(b)$.

Safe algorithm, distributed version

- Phase 2:
  - Tries to get enough other processes to vote for its new ballot.
- Communication pattern:



Ensuring Condition (2)

(2) Either every $b' < b$ is dead, or there exists $b' < b$ with $val(b') = v$, such that every b'' with $b' < b'' < b$ is dead.

- Phase 1:
  - Originator process i tells other processes the new ballot number b.
  - Each recipient j abstains from all smaller ballots it hasn't yet voted for.
  - Each j sends back to i the largest ballot number $< b$ that it has ever voted for, if any, together with that ballot's value.
  - If there is no such ballot, then j sends i a message saying there is none.
  - When process i collects this information from a read-quorum R, it assigns a value v to ballot b:
    - If anyone in R said it voted for a ballot $< b$, then v is the value associated with the largest-numbered of these ballots.
    - If not, then v can be any initial value.
- Claim this choice satisfies (2):

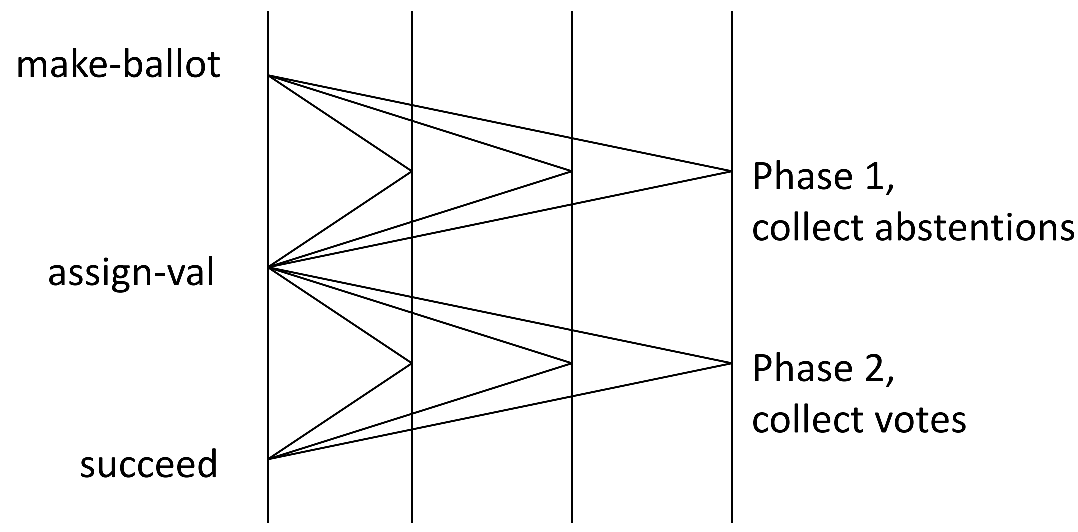
Ensuring Condition (2)

- (2) Either every $b' < b$ is dead, or there exists $b' < b$ with $val(b') = v$, such that every b'' with $b' < b'' < b$ is dead.
- Claim this choice satisfies (2):
- Case 1: Someone in R says it voted for some ballot $< b$.
  - Say b' is the largest such ballot number.
  - Then everyone in R has abstained from all ballots between b' and b.
  - So all ballots properly between b' and b are dead.
  - So, choosing $v = val(b')$ ensures the second clause of (2).
- Case 2: Everyone in R says it did not vote for any ballot $< b$.
  - Then everyone in R has abstained from all ballots $< b$.
  - So all ballots $< b$ are dead.
  - Satisfies the first clause of (2).

Safe algorithm, distributed version, cont'd

- After assigning $val(b) = v$, originator i sends Phase 2 messages asking processes to vote for b.
- If i collects such votes from a write-quorum W, it can successfully complete ballot b, decide v, and inform others.

- Note:
  - Originator i, or others, may start up new ballots at any time.
  - (2) guarantees that all successful ballots will have the same value v.
  - Arbitrary concurrent attempts to conduct ballots are OK, at least with respect to safety.



Live version of the algorithm

- To guarantee termination when the algorithm stabilizes, we must restrict its nondeterminism.
- Most importantly, we must restrict $BallotTrigger$ so that, after stabilization:
  - It asks only one process to start ballots (a leader).
  - It doesn't tell the leader to start new ballots too often---allows enough time for ballots to complete.
- E.g., $BallotTrigger$ might:
  - Use knowledge of "normal case" time bounds to try to detect who has failed.
  - Choose the smallest-index non-failed process as leader (refresh periodically).
  - Tell the leader to try a new ballot every so often---allowing enough "normal case" message delays to finish the protocol.
- Notice that $BallotTrigger$ uses time information---not purely asynchronous.
- We know we can't solve the problem otherwise.
- Algorithm tolerates inaccuracies in $BallotTrigger$: If it "guesses wrong" about failures or delays, termination may be delayed, but safety properties are still guaranteed.

Using Paxos to emulate general shared memory in a network

- Paxos paper suggests using the Paxos consensus algorithm repeatedly, to agree on successive operations on a shared data object.
- Idea is similar to Herlihy's universal construction.
- Uses Replicated State Machines (RSM).
- Emulates shared atomic objects that tolerate stopping failures and recoveries, message loss and duplication.
- Paper also includes various optimizations, LTTR.
- Considerable follow-on work, engineering Paxos to work for maintaining real data efficiently.
  - Disk Paxos
  - HP, Microsoft, Google,…

Next time

- Failure detectors
- Readings:
  - [Chandra, Toueg] Unreliable FDs for reliable distributed systems
  - [Cornejo, Lynch, Sastry] Asynchronous FDs
  - [Pike, Song, Sastry] FDs for Dining Philosophers
  - [Sastry, Pike, Welch] Weakest FD for Wait-Free Dining Philosophers
  - [Chandra, Hadzilacos, Toueg] Weakest FD for Consensus
  - [Lynch, Sastry] Weakest Asynchronous FD for Consensus