Class 23
Today’s plan

• Shared memory vs. networks
• Consensus in asynchronous networks
• Reading:
  – Chapter 17
• Next time:
  – Self-stabilization
  – [Dolev book], Chapter 2
Shared memory vs. Networks

• Simulating shared memory in distributed networks:
  – Popular method for simplifying distributed programming.
  – Distributed shared memory (DSM).
  – Easy if there are no failures.
  – Possible if $n > 2f$; impossible if $n \leq 2f$.

• Simulating networks using shared memory:
  – Easier, because shared memory is “more powerful”.
  – Works for any number of failures.
  – Useful mainly for lower bounds, impossibility results.
    • Carry over impossibility results for shared memory model to network model
    • E.g., for fault-tolerant consensus.
Paxos

- A fault-tolerant consensus algorithm for distributed networks.
- Can use it to implement a fault-tolerant replicated state machine (RSM) in a distributed network.
- Generalizes Lamport’s timestamp-based non-fault-tolerant RSM algorithm.
Simulating networks using shared-memory systems
Simulating networks using shared-memory systems

- Easy transformation from networks to shared-memory, because shared-memory model is more powerful:
  - Has reliable, instantaneously-accessible shared memory.
  - No arbitrary delays as in channels.
- Transformation preserves fault-tolerance, even for \( f \geq n/2 \).
- Assume:
  - Asynchronous network system \( A \), running on undirected graph network \( G \).
  - Failures: \( \text{stop}_i \) event disables \( P_i \) and has no effect on channels.
- Produce:
  - Asynchronous read/write shared-memory system \( B \) simulating \( A \), in the same sense as for atomic objects:
  - For any execution \( \alpha \) of the shared-memory system \( B \times U \), there is an execution \( \alpha' \) of the network system \( A \times U \) such that:
    - \( \alpha \mid U = \alpha' \mid U \) and
    - \( \text{stop}_i \) events occur for the same \( i \) in \( \alpha \) and \( \alpha' \).
    - If \( \alpha \) is fair then \( \alpha' \) is also fair.
Algorithm

- Replace channel $C_{i,j}$ with a 1-writer, 1-reader shared variable $x(i,j)$, writable by $i$, readable 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 end of $x(i,j)$.
    - Does this using a write operation, by remembering what it wrote there earlier.
  - Meanwhile, process $i$ keeps checking its incoming variables $x(j,i)$, looking for new messages.
    - Does this by remembering what it saw there before.
    - When it finds a new message, process $i$ handles it the same way $P_i$ would handle it.
Some pseudocode

- State variables for process $i$
  - $pstate : states(P_i)$
  - $sent(j)$ for each out-neighbor $j$: sequence of $M$, initially empty
  - $rcvd(j), processed(j)$ for each in-neighbor $j$: seq of $M$, initially empty

- Transitions for $i$
  - Internal $send(m,j)_i$:
    - pre: $send(m)_{i,j}$ enabled in $pstate_i$
    - eff: append $m$ to $sent(j)$; $x(i,j) := sent(j)$;
      update $pstate$ as for $send(m)_{i,j}$
  - Internal $receive(m,j)_i$
    - pre: true
    - eff: $rcvd(j) := x(j,i)$;
      update $pstate$ using messages in $rcvd(j) - processed(j)$;
      $processed(j) := rcvd(j)$
  - All others: As for $P_i$, using $pstate$. 
An important corollary

• **Theorem:** This simulation produces an asynchronous shared-memory system B simulating A, in the sense that, for any execution \( \alpha \) of the shared-memory system \( B \times U \), there is an execution \( \alpha' \) of the network system \( A \times U \) such that:
  - \( \alpha \mid U = \alpha' \mid U \).
  - stop\(_i\) events occur for the same \( i \) in \( \alpha \) and \( \alpha' \).
  - If \( \alpha \) is fair then \( \alpha' \) is also fair.

• **Corollary:** 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 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 already know this is impossible [Loui, Abu-Amara].
Another corollary

- **Corollary:** 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.
  - 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:** 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.
  - Process cannot fail in the middle of a broadcast.

- Recall in synchronous model, impossibility results for consensus depended heavily 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.
Simulating shared-memory systems using networks
Simulating shared-memory in distributed networks

- Popular method for simplifying 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
Shared memory in networks

- Assume shared memory system A:
  - Ports 1,...,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 can 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 × U, there is an execution α' of the shared memory system A × U such that:
    - α | U = α' | U and
    - stop_i events occur for the same i in α and α'.
    - If α is fair then α' is also fair (will change for FT case).
Single-copy simulation

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

- $Q_i$ and $R_{x,i}$ interact using invocations and 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 local copy.
- **Non-owners:** Send invocation to owner($x$), await response.
More formally…

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

• Optimization: Avoid busy-waiting on a remote shared variable: Send one request, let owner notify sender when the value of the variable changes, or when some condition on this value becomes true.

• Q: Where to put the copies?
Multi-copy simulation

- Still not fault-tolerant.
- Just for read/write objects.
- Locate each shared variable $x$ at some known collection of processes, $\text{owners}(x)$.
- Handle each shared variable independently.
- How $P_i$ accesses variable $x$:
  - \text{READ}: Read any copy.
  - \text{WRITE}: Write all copies, asynchronously, in any order.
  - “Read-one, write-all.”
- Can be faster than single-copy, on average, if reading is much more common than writing.
  - E.g., in peer-to-peer systems, sharing files.
- But, without some constraints, we get coherence issues…
Multi-copy simulation: Bad examples

- **Example 1:** Multi-writer, inconsistent order of WRITEs
  - $P_1$ and $P_2$ want to WRITE the same shared variable $x$.
  - $\text{owners}(x) = \{P_3, P_4\}$.
  - $P_1$ and $P_2$ send write request messages to both $P_3$ and $P_4$.
  - $P_3$ and $P_4$ receive the write requests in different orders, so end up with different values.
  - Later READs may get either value, inconsistent.

- **Example 2:** Single-writer, inconsistent READs
  - $\text{owners}(x) = \{P_2, P_3\}$.
  - Writer $P_1$ sends write request messages to $P_2$ and $P_3$.
  - Message arrives at $P_2$, $P_2$ writes its local copy.
  - Then a READ happens 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 $P_3$, $P_3$ writes its local copy.
  - The READs do not overlap, but are concurrent with the WRITE.
  - Out-of-order READ behavior is not allowed by atomic R/W object.
Multi-copy simulation

• So we need some more clever protocols…
• **Idea**: Use atomic transactions:
• E.g., to do a WRITE(x), 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 of x and write them.
  – Phase 2: Release all the locks.
• Must solve problems of deadlock for lock acquisition.
• Works because serialization point for WRITE can be placed at the “lock point”, where all the locks have been acquired.
Majority-voting algorithms

- Still not fault-tolerant.
- Just for read/write objects.
- Locate each shared variable $x$ at some known collection of processes, $\text{owners}(x)$.
- Handle each shared variable independently.
- How $P_i$ accesses variable $x$:
  - READ: Read from a majority of copies.
  - WRITE: Write to a majority of copies.
- Concurrency anomalies suggest that we run each READ or WRITE 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 they are implemented as transactions.
Majority-voting algorithms

• To see that this implements an atomic R/W object for x:
  – 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 R/W operations behave as if they occurred at their transactions’ serialization points:
  – WRITE operations are assigned tags \(1, 2, \ldots\) in order of their transactions’ serialization points.
  – READ or embedded-read obtains the largest tag that has been written by a WRITE operation serialized before it (0 if there are none), together with the associated value for x.
  – These two facts depend, in turn, on the 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.
Some issues

- Still no fault-tolerance:
  - Standard transaction impls 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 is a special case.
Fault-tolerant simulation of shared memory in distributed networks
Fault-tolerant simulation of shared memory in distributed networks

- Tolerates $f$ stopping failures, requires $n > 2f$.
- Assume reliable channels.
- Just for read/write objects, in fact, 1-writer multi-reader objects (exercise: extend to MWMR).

- Modeling failures:
  - Use a $\text{stop}_i$ input at each external port (of the shared-memory system A, or of the network system B).
  - $\text{stop}_i$ 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, requires n > 2f.
- For any execution α of network system B × U, there is an execution α′ of shared-memory system A × U such that:
  - α | U = α′ | 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, requires \( n > 2f \).
• Implement atomic object for each shared variable \( x \), then combine.
• No transactions, no synchronization.
• Each process keeps:
  – \( \text{val} \), a value for \( x \), initially \( v_0 \)
  – \( \text{tag} \), initially 0
• \( P_1 \) does \( \text{WRITE}(v) \):
  – Let \( t \) be the first unused tag (\( P_i \) 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 \( t > \text{current tag} \).
    • In any case, sends ack to \( P_1 \).
  – When \( P_1 \) knows a majority have received \( (v,t) \), returns ack.
[ABD] atomic object algorithm

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

STATE VARIABLES per process
val: V, initially \( v_0 \)
tag: \( \mathbb{N} \), initially 0
readtag: \( \mathbb{N} \), initially 0
lots of “bookkeeping” variables

READERS
on read
readtag := readtag+1
send “read(readtag)” to all other processes
- wait for ack from majority
let t be largest tag received
if t > tag then (val,tag) := (v,t)
where v is value received with t
send “propagate(val,tag,readtag)” to all readers
- wait for ack from majority
return val

ALL PROCESSES
on receiving “read(rt)” from j
send “read-ack(val,tag,rt)” to j

READERS
on receiving “write(v,t)” from writer
if t > tag then
  (val,tag) := (v,t)
send “write-ack(t)” to writer

WRITER
on write(v)
  (val,tag) := (v,tag+1)
send “write(val,tag)” to all readers
  - wait for ack from majority
return ack
Correctness of [ABD] atomic object algorithm

- Well-formedness √
- f-failure termination, for n > 2f √
- Atomicity:
  - Algorithm is similar to [Vitanyi, Awerbuch], so use similar proof, based on partial order lemma.
  - Here, define the partial order by:
    - Order WRITEs by tags.
    - Order READ right after WRITE whose value it gets.
  - **Key: Condition 2:** If operation \( \pi \) finishes before operation \( \varphi \) starts, then \( \varphi \) is not ordered before \( \pi \).
  - Consider cases, based on operation types.
  - Case 1:
    - Because majorities intersect, \( \varphi \) gets a tag \( \geq \) the tag written by \( \pi \).
    - So \( \varphi \) is ordered after \( \pi \).
Correctness of [ABD] atomic object algorithm

- linearization point of write with tag t
  - when majority of processes have tag $\geq t$
  - may linearize multiple writes at same point

- linearization point of read returning value associated with tag t
  - immediately after linearization point of write with tag t, or
  - immediately after invocation of read, (why do we need this?)
  - whichever is later
Atomicity, cont’d

– Partial order:
  • Order WRITEs by tags.
  • Order READ right after WRITE whose value it gets.

– Condition 2: If operation $\pi$ finishes before operation $\varphi$ starts, then $\varphi$ is not ordered before $\pi$.

– Case 2:

Then $\varphi$ gets a tag $\geq$ the tag obtained by $\pi$, because of propagation and majority intersection.

– So $\varphi$ is not ordered before $\pi$.

– Other cases: Simpler, LTTR.
Now use [ABD] atomic object algorithm to construct a distributed simulation of any fault-tolerant shared-memory algorithm $A$ that uses 1-writer multi-reader shared vars:

- Simply replace shared variables by [ABD] atomic object implementations.

**Guarantees:**

- For any execution $\alpha$ of network system $B \times U$, there is an execution $\alpha'$ of shared-memory system $A \times U$ such that:
  - $\alpha \upharpoonright U = \alpha' \upharpoonright U$ and
  - stop$_i$ events occur for the same $i$ in $\alpha$ and $\alpha'$.
- Moreover, if $\alpha$ is fair and contains stop$_i$ events for at most $f (< n/2)$ different ports, then $\alpha'$ is also fair.

That is, we have a correct simulation, provided that there are at most $f$ failures in the network system $B$. 

[ABD] Simulation Corollaries

• Guarantees:
  – For any execution $\alpha$ of network system $B \times U$, there is an execution $\alpha'$ 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 $\alpha$ and $\alpha'$.
    • If $\alpha$ is fair and contains stop$_i$ events for at most $f$ different ports, then $\alpha'$ is also fair.

• Corollary: Wait-free atomic snapshot algorithm using 1WmR registers (Chapter 13) can be transformed, using [ABD], to a distributed network memory-snapshot algorithm.

• Corollary: [Vitanyi, Awerbuch] wait-free mWmR register implementation using 1WmR registers can be transformed, using [ABD], to a distributed network register implementation.

• But 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.
Some issues

• Can generalize majorities to quorum configuration:
  – Set of read-quorums, set of write-quorums.
  – $R \cap W \neq \emptyset$ for every read-quorum $R$, write-quorum $W$.

• Then
  – READ operation accesses both a read-quorum and a write-quorum.
  – WRITE operation accesses just a write-quorum.

• So, we cannot improve READ performance by using smaller read-quorums!

• Q: So how can we get faster READ performance?
• A: Optimize to eliminate “most” propagation phases.
  – When a WRITE with tag $t$ completes, or a READ completes propagation of tag $t$, then tag $t$ doesn’t require further propagation.
  – So, an operation that completes $t$ can send messages to everyone saying that $t$ is complete; everyone who receives such a message marks $t$ 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: hardly anything interesting can be computed with $\geq n/2$ failures.
- Contrast with shared-memory model: 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: (Same structure as for other proofs showing impossibility of n/2-fault-tolerance.)
  - 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 $\leq f$:
    - $G_1 = \{1,\ldots,f\}$, $G_2 = \{f+1,\ldots,n\}$.
  - By $f$-fault-tolerance, even if one entire group fails, the other group must still give correct atomic register responses.
Impossibility of n/2-fault-tolerance

- **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 \( G_1 = \{1,\ldots,f\} \), \( G_2 = \{f+1,\ldots,n\} \).
  - If one group fails, the other group must still give correct atomic register responses.
  - **Execution \( \alpha_1 \):**
    - \( G_2 \) processes fail initially.
    - \( P_1 \) invokes WRITE(1).
    - WRITE must eventually terminate with ack.
    - Let \( \alpha_1' \) be the portion of \( \alpha_1 \) up to the ack.
  - **Execution \( \alpha_2 \):**
    - \( G_1 \) processes fail initially.
    - \( P_n \) invokes READ.
    - READ must eventually terminate with response 0.
    - Let \( \alpha_2' \) be the portion of \( \alpha_2 \) up to the response.
Proof, cont’d

- **Execution $\alpha_1$:**
  - $G_2$ processes fail initially.
  - $P_1$ invokes WRITE(1).
  - WRITE must eventually terminate with ack.
  - Let $\alpha_1'$ be the portion of $\alpha_1$ up to the ack.

- **Execution $\alpha_2$:**
  - $G_1$ processes fail initially.
  - $P_n$ invokes READ.
  - READ must eventually terminate with response 0.
  - Let $\alpha_2'$ be the portion of $\alpha_2$ up to the response.

- **Execution $\alpha_3$: Paste...**
  - Don’t fail anyone.
  - Do all the steps of $\alpha_1'$ first, including the ack.
  - Then do all the steps of $\alpha_2'$, including the response of 0.
  - Meanwhile, delay all messages between $G_1$ and $G_2$.

- Activity in $\alpha_1'$ and $\alpha_2'$ is independent, so $\alpha_3$ is an execution.
- But not correct for an atomic register, since the WRITE(1) completes before the start of the READ that returns 0.
- Contradiction.
An implication

- This theorem implies that there is no general simulation of shared-memory systems by networks, preserving f-fault-tolerance, for \( f \geq n/2 \).
  - See book, p. 567, for a definition of f-simulation, which formalizes “preserving f-fault-tolerance”.
  - It’s essentially the overall guarantee we gave earlier for [ABD].
- Because 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 algorithm exists, neither does such a general simulation.
Fault-Tolerant Agreement in Asynchronous Networks: The Paxos Algorithm
Agreement in asynchronous networks

• It’s impossible to reach agreement in asynchronous networks, even if we know that at most one failure will occur.

• But what if we really need to?
  – For transaction commit.
  – For agreeing on the order in which to perform operations.
  – ...

• Some possibilities:
  – Randomized algorithm (Ben-Or), terminates with high probability.
  – Approximate agreement.
  – Use a failure detector service, implemented by timeouts.
Best approach

- 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 when system is stable.
- Actually, stable behavior need not continue forever, just long enough for computation to terminate.
Eventually stable approach: Some history

- [Dwork, Lynch, Stockmeyer] first presented a consensus algorithm with these properties (2007 Dijkstra Prize)
- [Cristian] used 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 more kinds of failures.
    - Basic strategy for assuring safety similar to [DLS].
  - Background:
    - Paper unpublished for 10 years because of nonstandard style.
    - Eventually published “as is”, because others began 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 crashes, where volatile memory is lost in a crash (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 this to get termination soon after stabilization.
The nondeterministic “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 $V \cup \{\bot\}$, where $V$ is the consensus domain.

• Somehow, ballots get started, and get values assigned to them.

• Processes can **vote for**, or **abstain from**, particular ballots.
  – Abstention from a ballot is a promise never to vote for it.
The safe algorithm: 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.

• Generalization of majorities.

• 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

- Anyone can create a new ballot with Bid b:
  - `make-ballot(b)`
  - Provided no ballot with Bid b has yet been created.
  - `val(b)` is set to ⊥.

- A process i can abstain, in one step, from an entire set of ballots:
  - `abstain(B,i), B ⊆ Bid`
  - Provided i has not previously voted for any ballot in B.
  - We allow B to be any set of Bids, not necessarily associated with already-created ballots.
    - For example, B = all Bids in some range \([b_{min}, b_{max}]\).
    - This is important…
Safe algorithm, centralized version

• Anyone can assign a value \( v \) to a ballot id \( b \), \texttt{assign-val}(b,v), provided:
  - A ballot with id = \( b \) has been created.
  - \( \text{val}(b) \) is undefined.
  - \( v \) is someone’s consensus input.
  - (***) For every \( b' \in \text{Bid} \), \( b' < b \), either \( \text{val}(b') = v \) or \( b' \) is dead.

• Notes on (***):
  - Recall: \( b' \) dead means some read-quorum has abstained from \( b' \).
  - (***) Refers to every \( b' \in \text{Bid} \), not just created ones.
    - Relies on “set abstentions”.

• Thus, we can assign a value to a ballot \( b \) only if we know it won’t make \( b \) conflict with lower-numbered ballots \( b' \).

• Motivation:
  - Several ballots can be created, can collect votes.
  - More than one might succeed in collecting write-quorum of votes.
  - But we don’t want successful ballots to conflict.
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 a 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:
  – Because of the careful way we avoid assigning different values to ballots that might succeed.
  – **Key Invariant:** If \( \text{val}(b) \neq \bot \), \( b' \in \text{Bid} \), and \( b' < b \), then either \( \text{val}(b') = \text{val}(b) \) or \( b' \) is dead.
  – Implies that all successful ballots have the same value.
Modifying the ** condition for assigning ballot values

• Instead of checking:
  
  (***) For every $b' \in Bid$, $b' < b$, either $val(b') = v$ or $b'$ is dead.

• Check the apparently-weaker condition:
  
  (***) 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.

• (***) is easier to check in a distributed algorithm (will show how).

• And (***) implies (**), by easy induction on the number of steps in an execution.
Safe algorithm, distributed version

• Any process $i$ can create a ballot, at any time.
  – Use locally-reserved ballot id $b$.
  – 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, first tries to collect enough abstention information for smaller ballots to guarantee (***)
  – If/when it collects that, assigns $\text{val}(b)$. 
Safe algorithm, distributed version

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

- **Communication pattern:**

```
make-ballot  |  Phase 1, collect abstention information
assign-val   |
 succeed     |  Phase 2, collect votes
```
Ensuring (***)

(***) Either every \( b' < b \) is dead, or there exists \( b' < b \) with \( \text{val}(b') = v \), 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-numbered 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.
    • Else (if no such ballot), sends 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 \( R \) says it voted for a ballot \( < b \), then \( v = \) the value associated with the largest-numbered of these ballots.
    • If not, then \( v = \) any initial value.

• Claim this choice satisfies (***):
Ensuring (***)

• (***): Either every $b' < b$ is dead, or there exists $b' < b$ with $\text{val}(b') = v$, such that every $b''$ with $b' < b'' < b$ is dead.

• Why does this choice satisfy (***)?

• **Case 1:** Someone in R says it voted for a 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 between $b'$ and $b$ are dead.
  – So, choosing $v = \text{val}(b')$ ensures the second clause of (***).

• **Case 2:** Everyone in R says it did not vote for a ballot $< b$.
  – Then everyone in R has abstained from all ballots $< b$.
  – So all ballots $< b$ are dead.
  – Satisfies the first clause of (***).
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 and decide v.

- Note:
  - Originator i, or others, could start up new ballots at any time.
  - (***) 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.
Liveness

• To guarantee termination when the algorithm stabilizes, we must restrict its nondeterminism.
• Most importantly, must restrict BallotTrigger so that, after stabilization:
  – It asks only one process to start ballots (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 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.
Replicated state machines (RSMs)

- Paper also deals with repeated consensus, in particular, on a sequence of operations for an RSM.
- Yields an RSM that tolerates stopping failures/recoveries, message loss/duplication.
- Strategy:
  - Use infinitely many instances of Paxos to agree on first operation, second, third,…
  - Similar to Herlihy’s universal construction, which uses repeated consensus to decide on successive operations for an atomic object.
- Lamport’s paper also includes various optimizations, LTTR.
- Considerable follow-on work, engineering Paxos to work for maintaining real data.
  - Disk Paxos
  - HP, Microsoft, Google,…
Next time

• Self-stabilization
• [Dolev book], Chapter 2