6.852: Distributed Algorithms
Fall, 2009

Class 22

Today's plan

- More on wait-free computability.
- Wait-free vs. f-fault-tolerant computability
- Reading:
  - [Borowsky, Gafni, Lynch, Rajsbaum]
  - [Attiya, Welch, Section 5.3.2]
  - [Attie, Guerraoui, Kouznetsov, Lynch, Rajsbaum]
  - [Chandra, Hadzilacos, Jayanti, Toueg]
- Next time:
  - Shared-memory multiprocessor computation
  - Techniques for implementing concurrent objects:
    - Coarse-grained mutual exclusion
    - Locking techniques
    - Lock-free algorithms
- Reading:
  - [Herlihy, Shavit] Chapter 9

But actually:

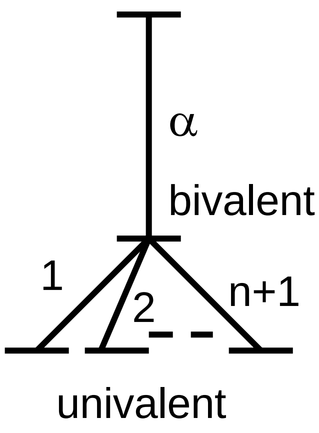
- Next time:
  - Shared memory vs. networks
  - Consensus in asynchronous networks
  - Reading:
    - Chapter 17 of [Lynch book]
    - [Lamport] The Part-Time Parliament (Paxos)

More on wait-free computability

1. n-process consensus objects + registers can't implement (n+1)-process consensus objects [Jayanti, Toueg].
2. Irreducibility theorem [Chandra, Hadzilacos, Jayanti, Toueg].

Consensus objects

- Theorem: n-process consensus objects + registers can't implement (n+1)-process consensus objects.
- Proof:
  - Assume they can.
  - Can find a decider: bivalent, any step produces univalence.
  - At least one is 0-valent, one 1-valent.
  - Let P_0 = processes that produce 0-valence, P_1 = processes that produce 1-valence.
  - Consider any i_0 in P_0, i_1 in P_1.
  - They must access the same object.
    - Else commutativity yields a contradiction.
  - Must be a consensus object.
    - If it's a register, get [Loui, Abu-Amara] contradiction.
  - By considering all i_0 in P_0, i_1 in P_1, can conclude all n+1 processes must access the same consensus object.
  - But it's just an n-process consensus object, contradiction.



Irreducibility Theorem

- [Chandra, Hadzilacos, Jayanti, Toueg]
- Theorem: For every $n \geq 2$ and every set S of types:
  - If there is a wait-free implementation of an n-process consensus object from (n-1)-process consensus objects, objects of types in S plus registers,
  - Then there is a wait-free implementation of n-process consensus from just objects of types in S plus registers.
- That is, the (n-1)-process consensus objects don't contribute anything!
- Proof: An interesting series of constructions, rather complicated, LTTR.

Open question

- Can wait-free 2-process consensus objects plus registers be used to implement a wait-free 3-process queue? (Exercise?)

Wait-free computability vs.
f-fault-tolerant computability

Wait-free computability vs. f-fault-tolerant computability

- We've been considering computability (of atomic objects) when any number of processes can fail (wait-free).
- Now consider a bounded number, f, of failures.
- [Borowsky, Gafni, et al.] transformation converts any n-process, f-fault-tolerant distributed shared r/w memory algorithm to an (f+1)-process f-fault-tolerant (wait-free) shared r/w memory algorithm, that solves a "closely related problem".
- Can derive wait-free algorithms from f-fault-tolerant algorithms.
- Not obvious:
  - E.g., perhaps some shared-memory algorithm depends on having a majority of nonfaulty processes.
  - This says (in a sense) that this can't happen.
- Can infer impossibility results for f-FT shared-memory model from impossibility for wait-free shared-memory model.
  - E.g., impossibility for 2-process wait-free consensus [Herlihy] implies impossibility for 1-FT n-process consensus [Loui, Abu-Amara].

Another consequence: k-consensus

- Theorem: k-consensus is unsolvable for k+1 processes, with wait-free termination.
  - Proved by three teams:
    - [Borowsky, Gafni], [Herlihy, Shavit], [Saks, Zaharoglu]
    - Godel Prize
- [BG] transformation implies impossibility for n-process k-consensus with k failures, $n \geq k+1$.

BG simulation

- Citations:
  - Original ideas presented informally: [Borowsky, Gafni STOC 93]
  - More complete, more formal: [B, G, Lynch, Rajsbaum]

What is a “Problem”?

- Herlihy:
  - Problem = variable type
  - Studies wait-free algorithms that implement an atomic object of a given type.
  - Problems involve ongoing interactions.
- BG:
  - All problems are one-shot:
    - Inputs arrive on some ports, at most one per port.
    - Outputs produced on some of those ports, at most one per port.
  - Problem = decision problem for n processes = set of pairs (I,O), where:
    - I and O are n-vectors over an underlying value domain V, and
    - Each I is paired with at least one O.
- Example: k-consensus
  - I = O = all vectors over V
  - (I,O) $\in$ D if and only if:
    - Every value in O appears somewhere in I, and
    - At most k distinct values appear in O.
  - Consensus: Special case of k-consensus for k = 1.

Solving a Problem

- An n-process shared-variable system solves an n-decision problem D, tolerating f failures, if all its executions satisfy:
  - Well-formedness: Produces answers only on ports where inputs are received, no more than once each.
  - Correct answers: If inputs occur on all ports, forming a vector I, then the outputs that are produced could be completed to a vector O such that $(I,O) \in D$.
  - f-failure termination: If inputs occur on all ports and at most f stop events occur, then an output occurs on each nonfailing port.
- Same style as our earlier definitions for consensus.

Relating two problems

- The BG simulation:
  - Takes a system that solves an n′-process decision problem D′, tolerating f failures.
  - Produces a system that solves an n-process decision problem D, also with f failures.
    - The n-process system simulates the n’-process system.
  - Special case where n = f+1 yields wait-freedom.
- D and D′ are not the same decision problem---e.g., they use different numbers of ports.
- But they must be related in some way.
- For some problems, the relationship is “obvious”:
  - Consensus, k-consensus defined by the same correctness conditions for n ports and n′ ports.
- In general, we need translation rules; express by:
  - A mapping G for input vectors, mapping n-vectors to n′-vectors.
  - A mapping H for output vectors, mapping n′-vectors to n-vectors.

Input translation G

- g_i :
  - For each i, $1 \leq i \leq n$, define a function g_i that maps an element of V (process i's input) to an n'-vector of V (proposed inputs for the simulated processes).
- G:
  - Mix and match, nondeterministically assigning each position in the final n'-vector a value from any of the vectors produced by the g_i functions.
- Example: k-consensus
  - $g_i(v) = (v, v, \ldots, v)$, n' entries
  - E.g., for $k = 2$, $n = 3$, $n' = 5$:
    - G(0, 0, 0) consists of (0,0,0,0,0) only.
    - G(0, 1, 1) consists of all vectors of 0s and 1s.

Output translation H

- h_i :
  - For each i, $1 \leq i \leq n$, define a function h_i that maps any "reduced" n'-vector of V (an n'-vector of V with up to f values replaced by $\perp$) to a value in V.
  - Represents process i's output, calculated from the output it sees from the simulated n'-process algorithm (may be missing up to f positions, because of failures).
- H:
  - Uses h_i to compute i's entry in the final n-vector.
- Example: k-consensus, $k > f$
  - h_i picks the first non-$\perp$ element of the given reduced vector.

Combining the pieces

- What we need:
  - If we combine G and H with the relation D′ (problem specification for the simulated algorithm), we should satisfy the relation D (problem specification for the simulating algorithm).
- More precisely:
  - Take any input n-vector I.
  - Apply individual mappings g_i and combine nondeterministically using G to get an input n′-vector I′ for D′.
  - Choose any output vector O′ such that $(I', O') \in D'$.
  - For each i separately:
    - Reduce O′ by setting up to f positions (any positions) to $\perp$.
    - Apply h_i to the reduced vector.
  - Assemble n-vector O from all the h_i outputs.
  - Then (I,O) should satisfy D.
- Example: Works for consensus, k-consensus, where D and D′ are the "same problem".

The BG construction

- Given: A system P′, with n′ processes, solving D′, tolerating f failures.
- Assumptions about P′:
  - P′ uses wait-free snapshot shared memory.
  - One shared snapshot variable, mem′.
  - Each P′ process is deterministic:
    - Unique start state.
    - In any state, at most one non-input action is enabled.
    - Any (old state, action) has at most one new state.
- Produce: A system P, with n processes, solving D, also tolerating f failures.
- Assumptions about P:
  - P uses wait-free snapshot shared memory.
  - One shared snapshot variable, mem.
- Do this by allowing the processes of P to simulate the processes of P′.

The BG construction

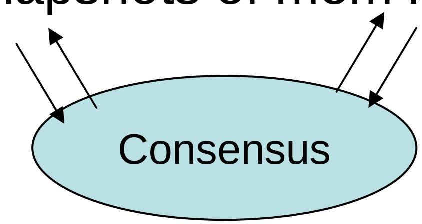
- Given: A system P′, with n′ processes, solving D′, tolerating f failures.
- Assumptions about P′:
  - P′ uses wait-free snapshot shared memory.
  - Each P′ process is “deterministic”:
- Produce: A system P, with n processes, solving D, also tolerating f failures.
- Assumptions about P:
  - P uses wait-free snapshot shared memory.

- Read/write shared memory instead of snapshot memory:
  - Same construction works if the two systems use read/write memory, but the proof is harder.
  - Alternatively, result carries over to the read/write case, using the fact that wait-free snapshots can be implemented from wait-free read/write registers.

- Q (for snapshot memory): How can the processes of P simulate an execution of P′?

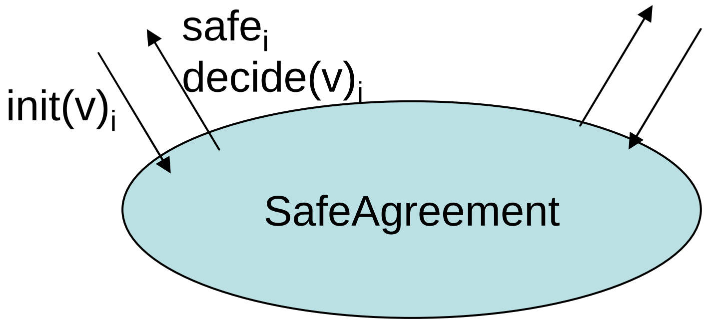
How P simulates P′

- Each P process simulates an execution of entire P′ system.
- We would like all of them to simulate the same execution.
- Since the P′ processes are assumed to be deterministic, many of the steps are determined, and can be simulated consistently by the P processes on their own.
- However, P processes must do something to agree on:
  - The P′ processes' initial inputs.
  - What the P′ processes see whenever they take snapshots of mem′.
- How? Use a consensus service?
  - Well-formedness, agreement, strong validity.
  - What termination guarantee?
  - Need f-failure termination, since f processes of P can fail.
  - But not implementable from snapshot memory [Loui, Abu-Amara].
- So we are forced to use something weaker…

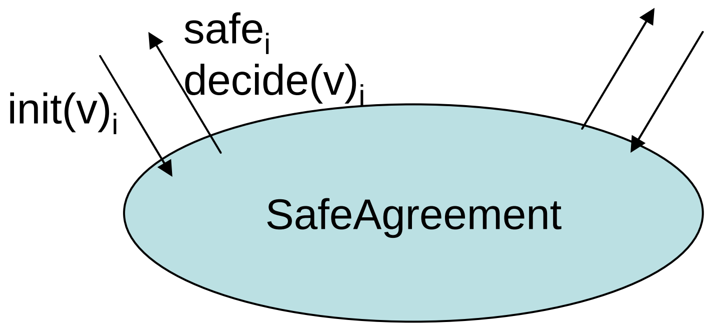


Safe Agreement

- A new kind of consensus service.
- Guarantees agreement, strong validity, failure-free termination, as usual.
- But now, susceptibility to failure on each port is limited to a designated "unsafe" part of the consensus execution.
- New interface:
  - Add safe outputs.
  - $safe_i$ anounces to user at port i that the "unsafe" part of the execution at i has completed.
  - $decide(v)_i$ provides the final decision, as usual.
- Well-formedness:
  - For each i, $init()_i$, $safe_i$, $decide()_i$ occur in order.
  - Component must preserve well-formedness.

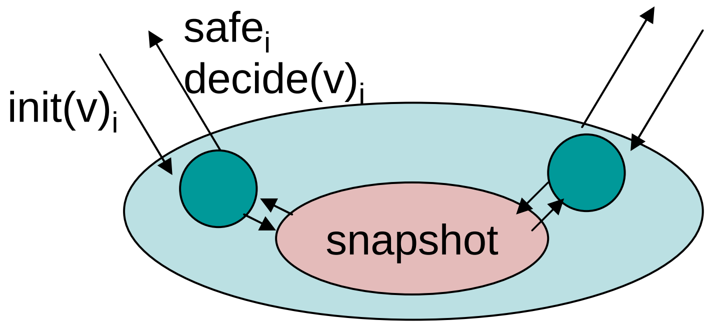


Safe Agreement



- Well-formedness
- Wait-free safe announcements:
  - In any fair execution, for every i, if an init$_i$ occurs and stop$_i$ does not occur, then safe$_i$ eventually occurs.
  - That is, any process that initiates and does not fail eventually gets a safe response---it can't be blocked by other processes.
- Safe termination:
  - In any fair execution, either:
    - For every i, if an init$_i$ occurs and stop$_i$ does not occur, then a decide$_i$ eventually occurs, or
    - There is some i such that init$_i$ occurs and safe$_i$ does not occur.
  - That is, the component acts like a wait-free implementation, unless someone fails in the unsafe part of its execution.
- Separating the termination guarantees in this way leads to an implementable specification, using snapshot or read/write shared memory.

Safe consensus implementation



- [BGLR, p. 133-134].
- Snapshot memory, component i:
  - val(i), in $V \cup \{ \perp \}$, initially $\perp$
  - level(i), in { 0, 1, 2 }, initially 0
- Process i:
  - When init(v)$_i$ occurs, set val(i) := v, level(i) := 1.
  - Perform one snapshot, determining everyone else's levels.
  - If anyone has level = 2, reset level(i) := 0, else set level(i) := 2.
  - In either case, move on, become safe, output safe$_i$.
  - Next, take repeated snapshots until you see no one with level = 1.
  - At this point (can show that) someone has level = 2.
  - Decide on v = val(j), where j is the min index for which level(j) = 2, output decide(v)$_i$.

Correctness

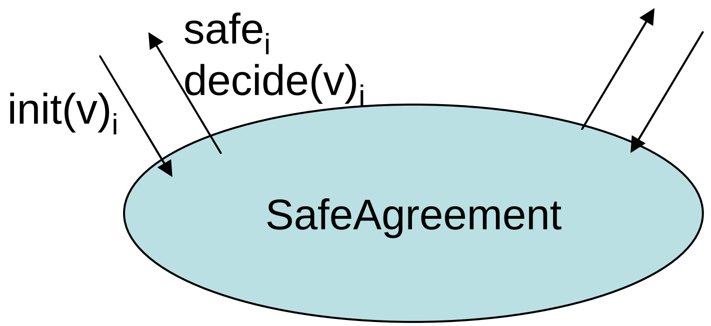
- Well-formedness, strong validity: Obvious.
- Agreement:
  - Suppose process i is first to take a deciding snapshot.
  - Say it decides on value v obtained from process k.
  - At the point of i's deciding snapshot, i sees level $\neq$ 1 for every process, and k is the min index with level = 2.
  - Claim: Subsequently, no process changes its level to 2.
  - Why:
    - Suppose some process j does so.
    - At the point of i's deciding snapshot, level(j) = 0 (can't = 1).
    - So j must first raise level(j) from 0 to 1, and then perform its initial snap.
    - But then it would see level(k) = 2 in its initial snap, reset level(j) to 0, and never reach level 2.
  - So, any process that takes its deciding snapshot after i does also sees k as the min index with level = 2, so decides on k's value v.

Liveness properties

- Wait-free safe announcements:
  - Obvious. No delays.
- Safe termination:
  - Suppose there is no process j for which $init_j$ occurs and $safe_j$ doesn’t (no one fails in the unsafe portion of the algorithm).
  - Then there is no process j whose level remains 1 forever.
  - So, eventually every process’ level stabilizes at 0 or 2.
  - Thereafter, any non-failing process will succeed in any subsequent snapshot, and decide.

Back to the BG simulation

- Each P process simulates an execution of entire P′ system.
- All of them should simulate the same execution.
- Since P′ processes are deterministic, many of the steps are determined, can be simulated by the P processes on their own.
- However, P processes must do something to agree on:
  - The P′ processes' initial inputs.
  - What the P′ processes see whenever they take snapshots of mem′.
- Can't use consensus.
- So, use safe-agreement.



Where are we?

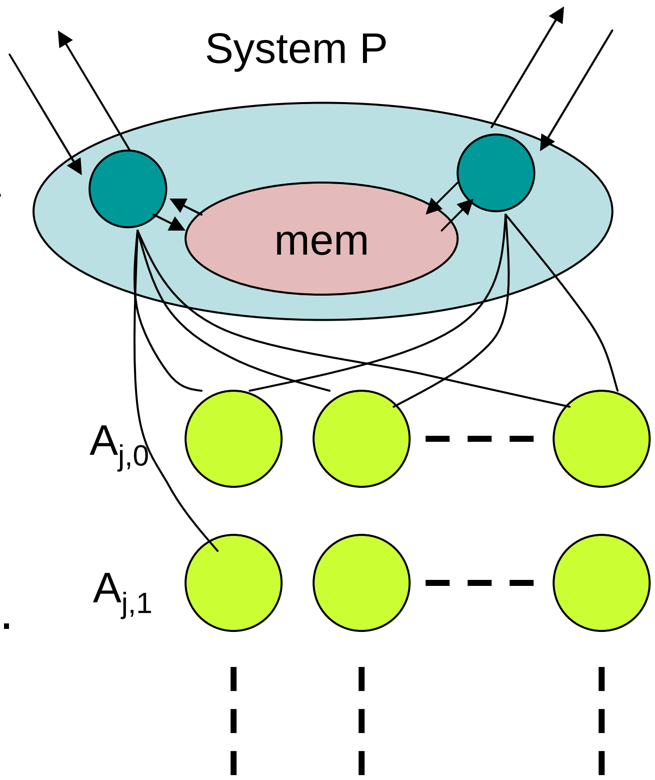
- We have produced a safe-agreement algorithm:
  - Agreement, strong validity, failure-free termination.
  - Well-formedness.
  - Wait-free safe announcements.
  - Safe termination.
- Now back to the main BG simulation algorithm.
- Uses (many) safe-agreement services.

BG simulation

- Processes of system P use (countably many) safe-agreement services to help them to agree on initial values and snapshot results, for P′ processes.
- Follow a discipline whereby each P process is in the unsafe part of at most one safe-agreement at a time.
- So if a P process fails, it "kills" at most one safe-agreement service, and so, kills at most one simulated P′ process.
  - The one for which the safe-agreement service is trying to decide on an initial value or snapshot result.
- So, f failures among P processes cause at most f failures of P′ processes.
- So we get the f-fault-tolerance guarantees of system P′, which imply that the nonfaulty P processes terminate.

The main construction

- [BGLR, Section 5]
- P has n processes.
- Shared memory:
  - mem, a single snapshot shared variable, with a component mem(i) for each i:
    - mem(i).sim-mem
    - mem(i).sim-steps
- Safe agreement modules:
  - $A_{j,l}$, $1 \leq j \leq n'$, l any nonnegative integer
  - Infinitely many safe-agreement modules for each process j of P′.
  - $A_{j,0}$: Used to agree on initial value for process j.
  - $A_{j,l}$, $l \geq 1$: Agree on the l^{th} simulated snapshot result obtained by process j.
- Other steps simulated locally, don't need consensus.
- In final algorithm, the $A_{j,l}$ modules are replaced by safe-agreement implementations.



The main construction

- Code, p. 135-136 of [BGLR].
- Process i of P simulates all processes of P′.
- Simulates steps of each j of P′ sequentially.
- Works concurrently on different j.
- Simulates deterministic steps locally, uses safe-agreement for inputs and snapshot results.
- Ensures that it is in unsafe portion of its execution for at most one simulated process j at a time.
- Locally, process i keeps track of where it is up to in simulating each process j of P′.
- In shared memory mem, process i records:
  - mem(i).sim-mem: The latest value i knows for the snapshot variable mem′ of P′ (from i's progress in the overall simulation).
  - mem(i).sim-steps, a vector giving the number of steps that i has simulated for each process j of P′, up to and including the latest step at which process j updated mem′(j).

Determining “latest” value for mem′

- Different P processes can get out of synch in their simulations, making different amounts of progress in simulating different P′ processes.
- Thus, different mem(i)s can reflect different stages of the simulation of P′.
- Function latest combines information in the various mem(i)s, to give the maximum progress for each j of P′.
  - Returns a single vector of values, one value per process j of P′, giving the latest value written by j to mem′ in anyone’s simulation.
  - Determined by, for each j, choosing the sim-mem(j) associated with highest sim-steps(j).

Simulating snapshots

- When P_i simulates a snapshot step of P'_j:
  - P_i takes a snapshot of mem, thus determining what all processes of P are up to in their simulations of P′.
  - Uses latest function to obtain a candidate value for the simulated memory mem′.
  - However, P_i doesn't just use that candidate mem′ for the simulated snapshot response.
  - Instead, it submits the candidate mem′ to the designated safe-agreement module.
  - This ensures that everyone will use the same candidate mem′ snapshot value when they simulate this snapshot step of j.

The code

- $init(v)_i$: Just record your own input.
- $propose(v)_{j,i,0}$:
  - Compute (using g_i) candidate input value for process j of P′.
  - Initiate safe-agreement.
  - Don’t start safe-agreement while you’re in unsafe part of any other safe-agreement.
- $agree(v)_{j,i,0}$: Gets agreement on j’s initial value.
- Then starts simulating locally.
- $snap_{j,i}$: When up to a snap step of j, do an actual snapshot from mem and compute a candidate snapshot result.
- $propose(w)_{j,l,l,}$ $l \geq 1$:
  - Proposes candidate snapshot result to next safe-agreement for j.
  - Don’t start safe-agreement while you’re in unsafe part of any other safe-agreement.
- $agree(w)_{i,j,l}$, $l \geq 1$: Gets agreement on j’s l^{th} snapshot result.

A code bug

- Paper has a code bug, involving liveness.
- As written, this code doesn't guarantee fair turns to each j:
  - When process i is about to propose an initial value or snapshot result for j to a safe-agreement module, it checks that no other simulated process is unsafe.
  - It's possible that, every time i gives j a turn, someone else might be in the unsafe region, thereby stalling j forever.
- Solution: Add a priority mechanism, e.g.:
  - When there's a choice, favor the j for which i has simulated the fewest snapshot steps so far.
  - [Attiya, Welch] use a round-robin discipline, LTTR.

The code, continued

- Other simulated steps are easier:
- sim-update$_{j,i}$:
  - Deterministic.
  - Process i determines j's update value locally.
  - Writed it to the actual snapshot memory, mem:
    - mem(i).sim-mem, mem(i).sim-steps
- sim-local$_{j,i}$: Does this locally.
- sim-decide$_{j,i}$: Computes a decision value for j, locally.
- decide(v)$_i$:
  - Process i computes its actual decision, using h_i.
  - Outputs the decision.

Correctness proof

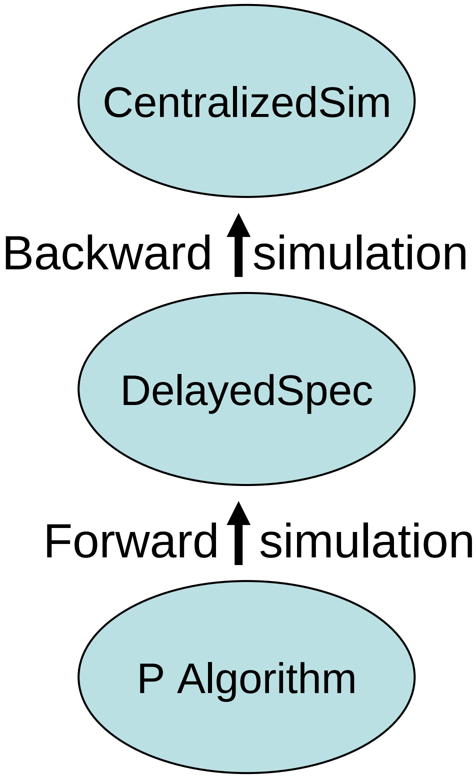
- f-failure termination:
  - Assume at most f failures in P.
  - With the added priority mechanism, P emulates a fair execution of P′ with at most f failures.
  - There are at most f failures in the simulated execution of P′, because each failed process in P can kill at most one safe-agreement, hence at most one process of P′.
  - By f-failure termination of P′, the non-failed processes of P′ eventually decide, yielding enough decisions to allow all non-failed processes of P to decide.

Correct emulation of P′

- Key idea: The distributed system P emulates a centralized simulation of P′.
  - mem′, the simulated memory of P′ in the centralized simulation, is determined by the latest information any of the P processes have about mem′.
  - Likewise for simulated states of P′ processes.
  - Initial value of process j of P′ is the value determined by safe-agreement $A_{j,0}$; the $init_j$ is deemed to occur when the first agree step of $A_{j,0}$ occurs.
  - Result of the l^{th} snapshot by j is the value determined by safe-agreement $A_{j,l}$; the $snap_j$ is deemed to occur when the candidate snapshot that eventually wins is first defined (as part of a snapshot in P).
- Formalize all this using simulation relations.

Simulation relation proof

- Simulation proof is done in two stages, using an intermediate "DelayedSpec".
- DelayedSpec does all the candidate snapshots, then later, in a separate step, chooses the winner.
- DelayedSpec maps to the centralized simulation.
  - Uses a "backward simulation".
  - Needed because we don't know that a particular candidate corresponds to the snapshot in CentralizedSim at the point where the candidate is first defined.
  - We learn this only later, when the winner is chosen.
- P maps to the DelayedSpec
  - Ordinary forward simulation.



BG for read/write memory

- Same result holds if P and P′ use read/write memory instead of snapshot memory.
- Can see this by implementing P's snapshots using read/write registers, as in [Afek, et al.]
- Can avoid the overhead of implementing snapshots by:
  - Defining a modified version of the BG construction for read/write memory, and arguing that it still works.
  - Harder proof, see [BGLR].
  - Uses an argument like that we used earlier, to show correctness of a simple implementation of a read/increment atomic object.

Recap: [BGLR]

- Theorem (paraphrase): For any n, $n' \geq f$:
  - If there is an n' -process, f-fault-tolerant read/write shared memory algorithm A' solving a problem D',
  - then there is an n-process, f-fault-tolerant read/write shared memory algorithm A solving a “closely related” problem D.
- Proof involves simulating steps of A one-by-one, rather than using D as a “black box” object.
- [Chandra, Hadzilacos, Jayanti, Toueg] sketch a similar result, allowing other types of shared memory.

A Non-Boosting Result
[Attie, Guerraoui, Kouznetsov, Lynch, Rajsbaum]

Non-boosting result

- Q: Can some set of f-fault-tolerant objects, plus reliable registers, be used to implement an n-process (f+1)-fault-tolerant consensus object?
- Now consider black-box implementations.

- We already know:
  - Wait-free (f+1)-process consensus + registers cannot implement wait-free (f+2)-process consensus.
  - [BGLR], [CHJT]: There are close relationships between n-process, (f+1)-fault-tolerant algorithms and wait-free (f+2)-process algorithms.
- So we might expect the answer to be no.
- Here is a simple, direct impossibility proof.

f-resilient atomic objects

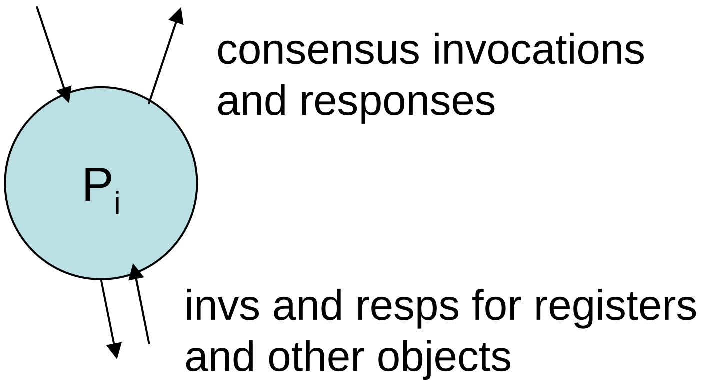
- Model f-resilient atomic objects as canonical f-resilient atomic object automata.
- State variables:
  - val, copy of the variable
  - inv-buffer, resp-buffer for each port, FIFO queues
    - Expect at most one active invocation at a time, on each port.
  - failed, subset of ports
- Tasks:
  - For every port i, one i-perform task, one i-output task.
- Explicitly program fault-tolerance:
  - Keep track of which ports have failed.
  - When $> f$ failures have occurred, the object need not respond to anyone (but it might).
  - When $\leq f$ failures have occurred, the object must respond to every invocation on a non-failing port.
  - Convention: Each i-task includes a dummy action that's enabled after failures (either of i itself, or of $> f$ ports overall).

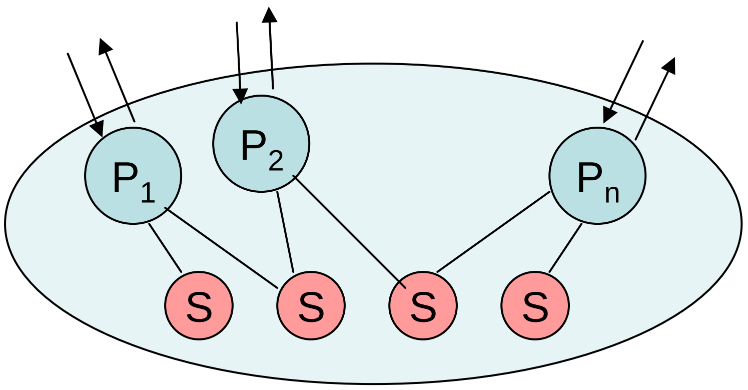
Concurrent invocations

- Since f-fault-tolerant objects can die, a nonfaulty process i might invoke an operation on a dead object and get no response.
- If process i accesses objects sequentially, this would block it forever.
- Avoid this anomaly by allowing a process to issue current accesses on different objects.
- Issue doesn't arise in the wait-free case.

System Model

- Consists of:
  - Processes P_i, $i \in I$
  - f-resilient services S_k, $k \in K$
  - Reliable registers S_r, $r \in R$
- Process P_i:
  - Automaton with one task.
- f-resilient service S_k:
  - Canonical f-resilient atomic object of some type, with some number of ports.
- Register S_r:
  - Wait-free atomic read/write object.
- Complete system:
  - Compose everything, arbitrary connection pattern between processes and services/registers.
  - Tasks: 1 for each process, 2 for each port in each service/register.





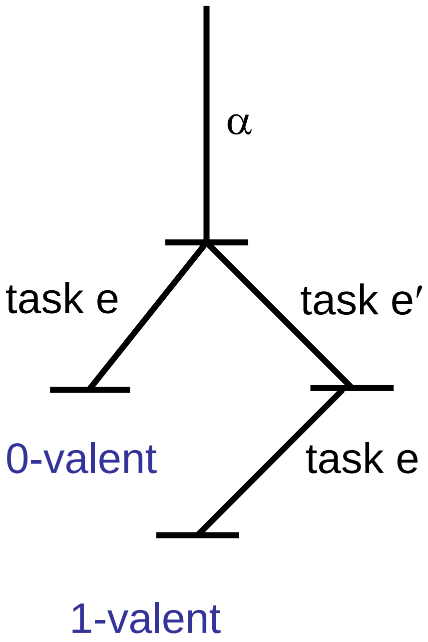
Boosting Impossibility Result

- Theorem: Suppose $n \geq 2$, $f \geq 0$. Then there is no (f+1)-resilient n-process implementation of consensus from f-resilient services (of any types) and reliable registers.

- Proof:
  - Depends on the delays within the services.
  - By contradiction, assume an algorithm.
  - Determinism:
    - WLOG, assume processes are deterministic:
      - One task.
      - From each state, exactly one action enabled, leading to exactly one new state.
    - WLOG, variable types are deterministic.
    - Tasks determine execution.
  - As usual, get a bivalent initialization (inputs for all processes).
  - From there, construct a “decider”:

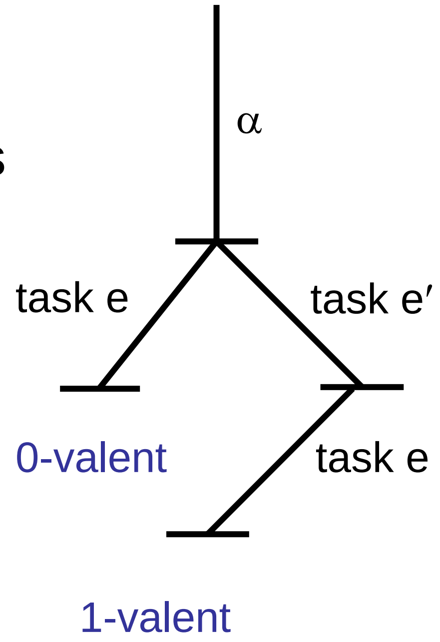
A Decider

- Tasks e and e′ are both applicable after α, and e and e′ e yield opposite valence.
- Clearly, e and e′ are different tasks.
- Claim: The step of e after α and the step of e′ after α must involve a common process, service, or register.
- Proof: If not, we get commutativity, contradiction.
- Three cases:
  - Steps involve a common process P_i.
  - Steps involve a common f-resilient service S_k.
  - Steps involve a common reliable register S_r.



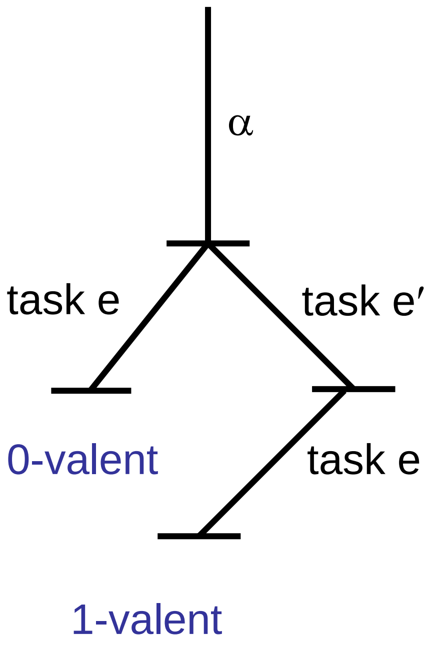
Case 1: Common process P_i

- The step of task e after α and the step of task e′ after α must involve only P_i, plus (possibly) inv-buffer$_i$ and resp-buffer$_i$ within some services and registers.
- So the step of e after α e′ also involves only P_i and its buffers.
- Then α e and α e′ e can differ only in the state of P_i and contents of its buffers within services and registers.
- Now fail i after α e and α e′ e:
  - Let the other processes run fairly, with i taking no further steps.
  - No i-perform or i-output task occurs in any service or register.
  - Failing i allows services/registers to stop performing work on behalf of i.
- These two executions look the same to the others, decide the same, contradiction.



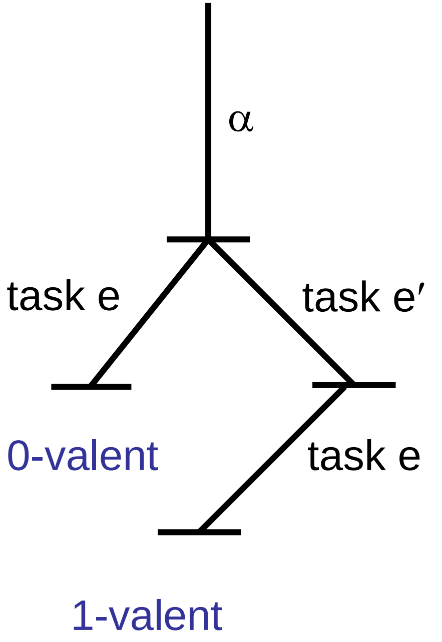
Case 2: Common f-resilient service S_k

- By Case 1, can assume no common process.
- If e after α involves S_k and P_i, and e′ after α involves just S_k (i.e., is a perform inside S_k):
  - Then commute, contradiction.
- If e after α involves just S_k, and e′ after α involves S_k and P_i.
  - Then commute, contradiction.
- If e after α involves S_k and P_i, and e′ after α involves S_k and P_j:
  - Then $i \neq j$ by assumption of no common process.
  - Commute, contradiction.
- Remaining case: e after α and e′ after α both involve just S_k :



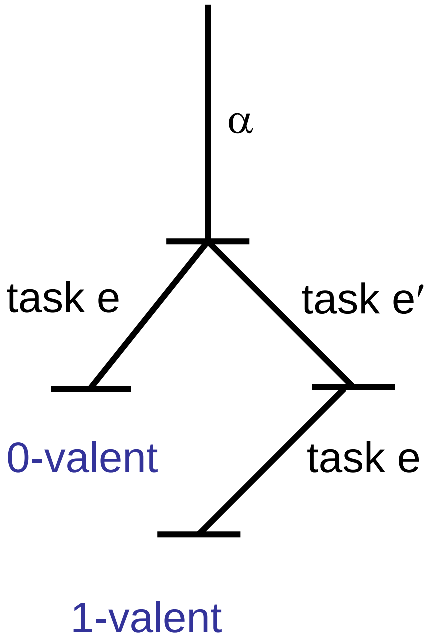
Case 2: Common f-resilient service S_k, cont'd

- If e after α and e′ after α involve just S_k:
  - Then both are performs.
  - Might not commute!
  - But only service S_k can tell the difference.
- Fail f+1 processes connected to S_k, after α e and α e′ e:
  - If fewer processes are connected to S_k, fail all processes connected to S_k.
  - Fails service S_k, allows it to stop taking steps.
  - Run the rest of the system with S_k failed, after α e and α e′ e.
  - Behaves the same, contradiction.



Case 3: Common register object S_r

- Argument is the same as for Case 2, until the last step.
- Again, we get 2 perform steps, don't commute.
- But now we can't fail the register by failing f+1 processes, since it's assumed to be reliable (wait-free).
- Instead, we rely on the [Loui, Abu-Amara] arguments for registers.
- Again, a contradiction.



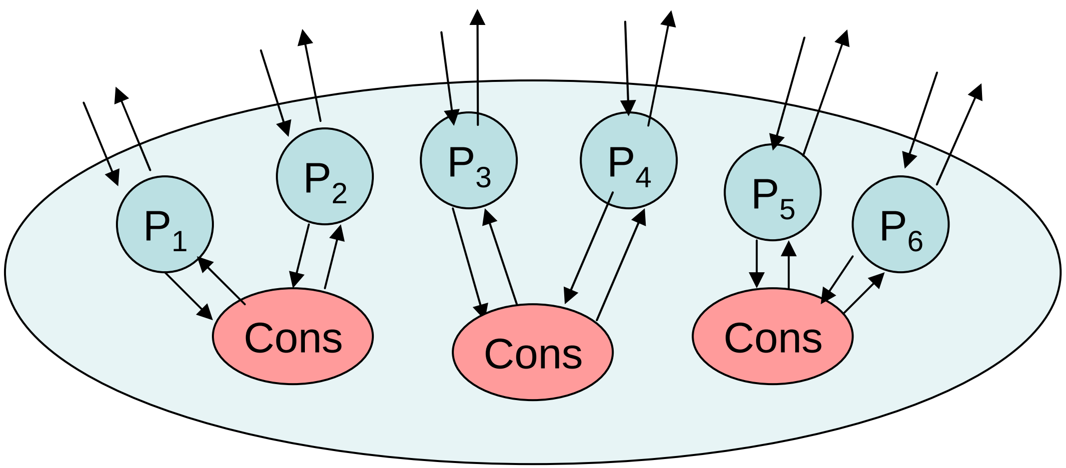
Recap: [AGKLR]

- Theorem: Suppose $n \geq 2$, $f \geq 0$. Then there is no (f+1)-resilient n-process implementation of consensus from f-resilient services (of any types) and reliable registers.

In contrast…

- Theorem: There is no (f+1)-resilient n-process implementation of consensus from f-resilient services and reliable registers.

- Example: Can sometimes boost resiliency
  - Can build a wait-free (5-resilient) 6-process, 3-consensus object from three 2-process wait-free (1-resilient) consensus services.
  - Each process P_i submits its initial value to its own consensus service.
  - The service responds, since it's wait-free.
  - Then P_i outputs the result.



Where are we?

- General goals:
  - Classify atomic object types: Which types can be used to implement which others, for which numbers of processes and failures?
  - A theory of relative computability, for objects in distributed systems.
- What we have so far:
  - Herlihy's classification based on solving consensus (wait-free), for different numbers of processes.
  - General transformation showing close relationship between (f+1)-process f-failure (wait-free) computability and n-process f-failure computability.
  - Non-boosting result for number of failures, for consensus.
- Much more work remains.

Next time…

- Shared memory vs. networks
- Consensus in asynchronous networks
- Reading:
  - Chapter 17 of [Lynch book]
  - [Lamport] The Part-Time Parliament (Paxos)

6.852J/18.437J Distributed Algorithms
Fall 2009