6.852: Distributed Algorithms
Fall, 2015

Lecture 23

Today's plan

- Finish Paxos
- Failure detectors
- Readings:
  - [Chandra, Toueg] Unreliable Failure Detectors for Reliable Distributed Systems
  - [Cornejo, Lynch, Sastry] Asynchronous FDs, TR 2013-025
  - [Pike, Song, Sastry] Dining philosophers using FDs
  - [Sastry, Pike, Welch] Weakest FD for Wait-Free Dining Philosophers
  - [Chandra, Hadzilacos, Toueg] Weakest FD for Consensus
  - [Lynch, Sastry] Weakest Asynchronous FD for Consensus
- Next time:
  - Self-stabilization
  - Reading: Dolev book, Chapter 2

Paxos consensus algorithm

- Guarantees agreement, validity in all cases.
- Guarantees termination if the system eventually stabilizes:
  - No more failures, recoveries, message losses.
  - Timing within “normal” bounds.
- Terminates soon after system stabilizes.

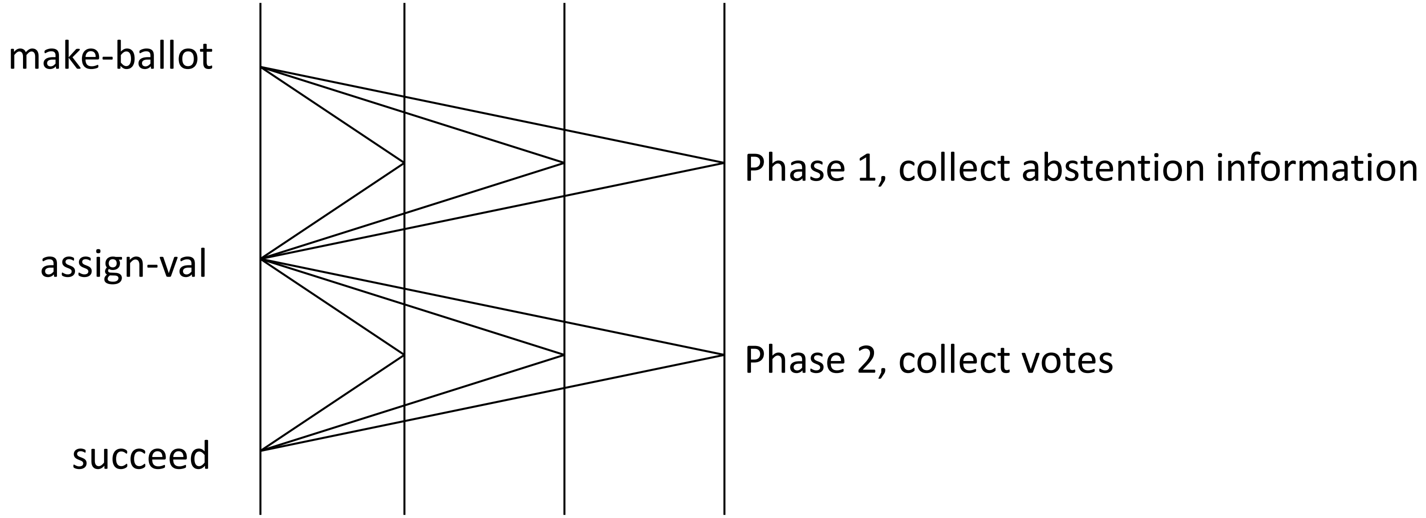
Ballots and Quorums

- Ballot = (identifier, value) pair.
- Ballots get started, get values assigned.
- Processes can vote for, or abstain from, ballots.
- Quorum configuration:
  - A set of read-quorums, finite subsets of process indices.
  - A set of write-quorums, finite subsets of process indices.
  - $R \cap W \neq \varnothing$ 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

- Any process i can create a ballot, at any time.
  - Use a locally-reserved ballot id.
  - Ballot start is triggered by a $BallotTrigger$ service.
- Phase 1:
  - Process i starts a ballot, but doesn't assign a value to it yet.
  - Rather, it first tries to collect enough abstention information for smaller ballots to guarantee a certain Condition (2).
  - If/when it collects that, assigns $val(b)$.
- Phase 2:
  - Tries to get a write quorum of processes to vote for its ballot.

Communication Pattern



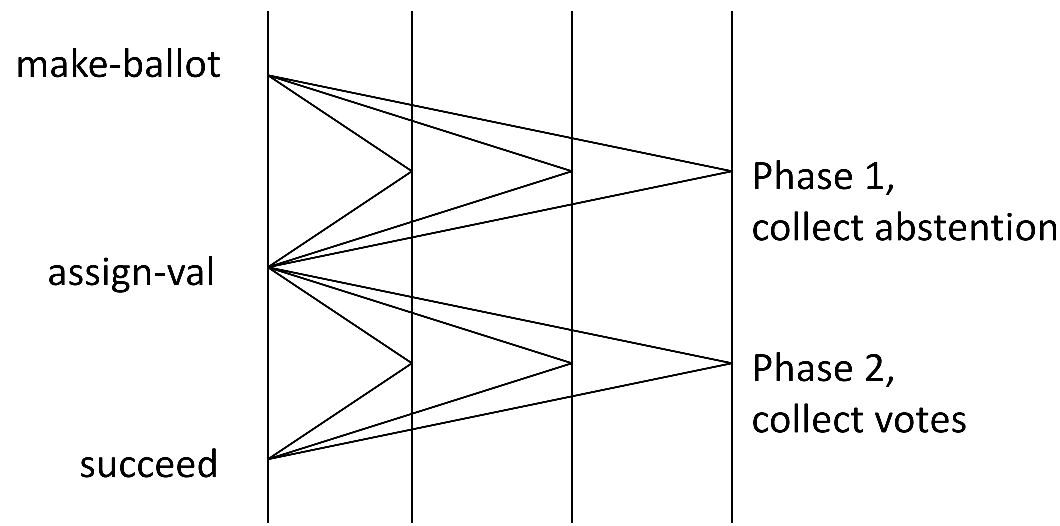
Phase 1

- Process i tells other processes the new ballot number b.
- Each recipient j:
  - Abstains from all smaller ballots it hasn't yet voted for.
  - 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, sends i a message saying that.
- 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.
- Ensures Condition (2): Either every $b' < b$ is dead, or there is some $b' < b$ with $val(b') = v$, such that every b'' with $b' < b'' < b$ is dead.

Phase 2

- 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, may start 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.



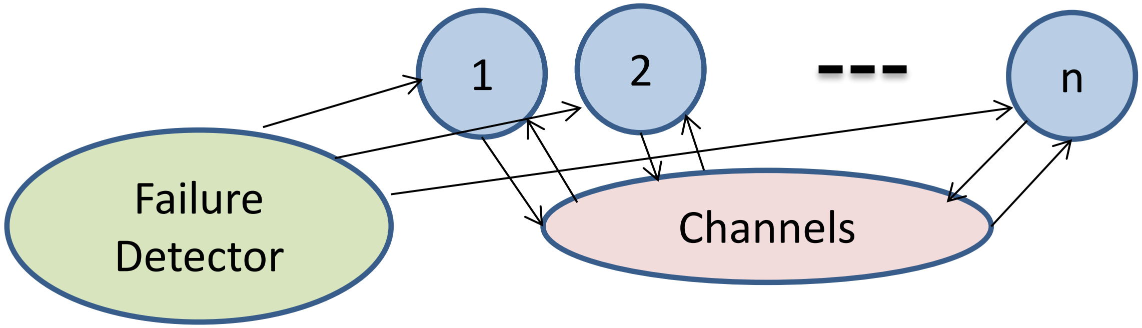
Live version of the algorithm

- To guarantee termination when the algorithm stabilizes, we must restrict its nondeterminism.
- Most importantly, we 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" 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.

Using Paxos to emulate general shared memory in a network

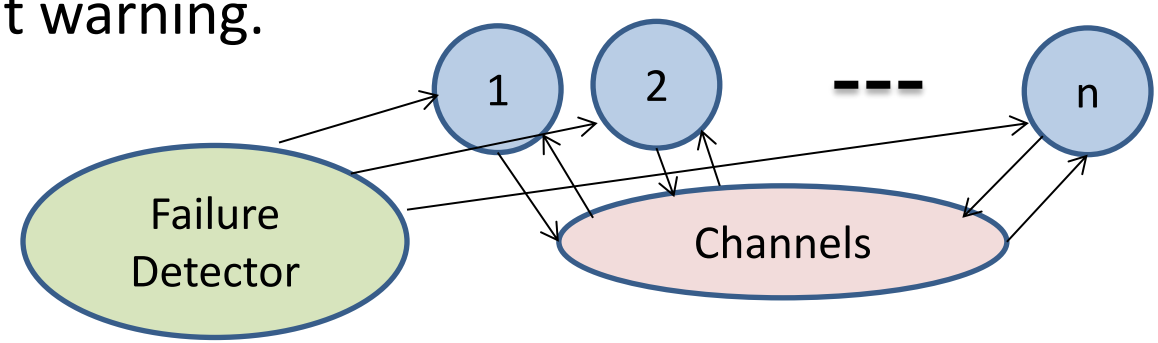
- Paxos paper suggests using the Paxos consensus algorithm repeatedly, to agree on successive operations on a shared data object, of any type.
- Idea is similar to Herlihy's universal construction.
- Uses Replicated State Machines (RSM).
- Emulates shared atomic objects that tolerate stopping failures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

Failure Detectors



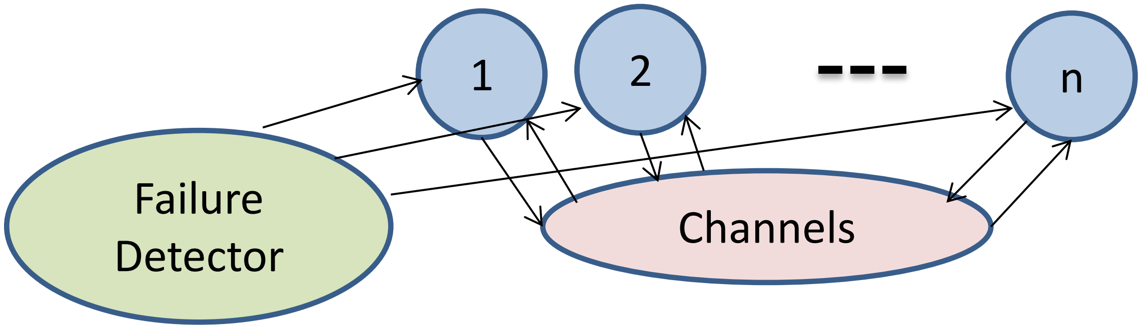
What is a failure detector?

- Consider an asynchronous distributed system consisting of processes and reliable FIFO channels. Processes may fail by stopping without warning.



- Many problems are unsolvable in this setting:
  - Fault-tolerant consensus, k-consensus, leader election …
  - Fault-tolerant eventual mutual exclusion, Dining Philosophers, …
- A failure detector is a service that interacts with the processes, providing them with some information about process failures.
- This allows some problems to be solved in fault-prone asynchronous systems that would not be solvable otherwise.

Failure Detectors can be Unreliable



- FDs provide information about which processes have failed.
- The information may be unreliable:
  - False negative mistakes: Failure detector might not report that some failed process has failed.
  - False positive mistakes: Might report incorrectly that some non-failed process has failed.
  - Might provide different information to different processes.
  - Might provide different information at different times.
- In spite of this unreliability, FDs can be used to solve many problems in fault-prone systems.

History

- [Chandra, Toueg 96]:
  - Defined failure detector services, for message-passing systems.
  - Gave many examples of failure detectors, with different guarantees.
  - Developed a classification of failure detectors based on relative power, e.g., based on which could be used to implement which others.
  - Gave two algorithms that use imperfect failure detectors to solve consensus.
  - Proved that, for weaker kinds of failure detectors, such algorithms exist only for $f < \frac{n}{2}$.
- [Chandra, Hadzilacos, Toueg 96]:
  - Proved that certain failure detectors are "weakest" to solve consensus, in the sense that any other failure detector that can solve consensus must be capable of "implementing" them.
- [CHT] + [CHJT] won the 2010 Dijkstra Prize

History

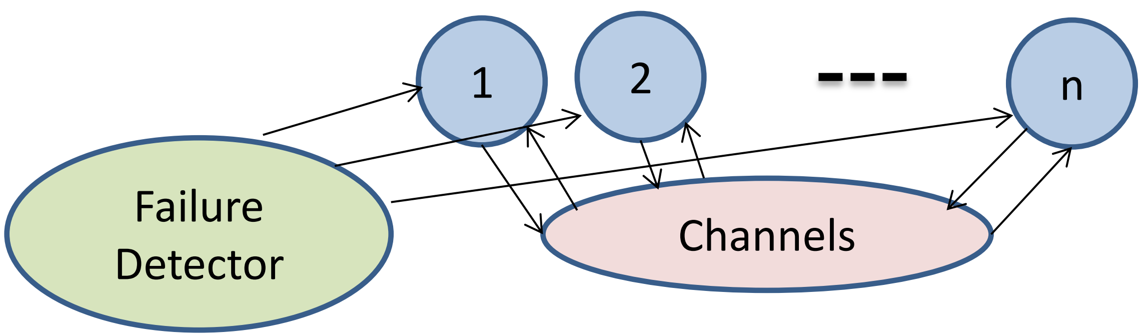
- [CT], [CHT] models include:
  - A mixture of synchronous/timed/asynchronous aspects.
  - A query-response interface.
- Led to oddities, e.g.:
  - Not every FD can "implement" itself.
  - FDs can convey information about aspects of executions other than failures.
- Not based on a general concurrency theory foundation.
- [Cornejo, Lynch, Sastry 13]:
  - Recast the FD definitions purely asynchronously, in terms of I/O automata.
  - FDs send information to processes spontaneously, no queries.
  - Simpler; supports a more complete formal presentation.
  - Removes oddities.
  - Yields some new results about weakest failure detectors.

History, cont'd

- [Lo, Hadzilacos, WDAG 94]
  - Defined failure detectors for shared-memory systems.
  - Gave an algorithm that uses FDs to solve consensus, for any number of failures (wait-free).
  - [CT]'s $f < \frac{n}{2}$ lower bound applies just to distributed networks, not shared-memory systems.
- [Pike, Song, Sastry, ICDCN 08]
  - Wait-free Dining Philosophers algorithm using FDs.
- [Sastry, Pike, Welch, SPAA 09]
  - Weakest FD for wait-free Dining Philosophers.

History

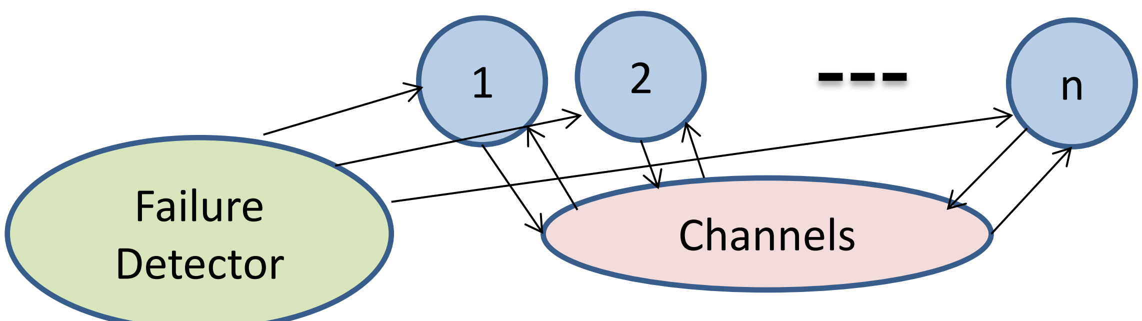
- [Gafni, Kuznetsov], other recent papers:
  - Weakest failure detectors for k −consensus.
- [Lynch, Sastry 14]:
  - Weakest asynchronous failure detector for consensus; similar to [CHT] but with complete I/O automata presentation/proof, gaps filled in.



Overview

1. Definitions: Crash problems, failure detectors
2. Typical FDs: $\diamond P$, $\diamond S$, Ω
3. Definitions: Solving crash problems, comparing FDs
4. Self-implementability of FDs
5. Comparing typical FDs
6. Solving particular problems using FDs
   a) Consensus using Ω
   b) Dining Philosophers using a local version of $\diamond P$
7. Definitions: Weakest FDs to solve problems
8. Weakest FDs for particular problems
   a) Local $\diamond P$ is a weakest FD for wait-free Dining Philosophers
   b) Ω is a weakest FD for fault-tolerant consensus.

1. Definitions:
Crash Problems and Failure Detectors

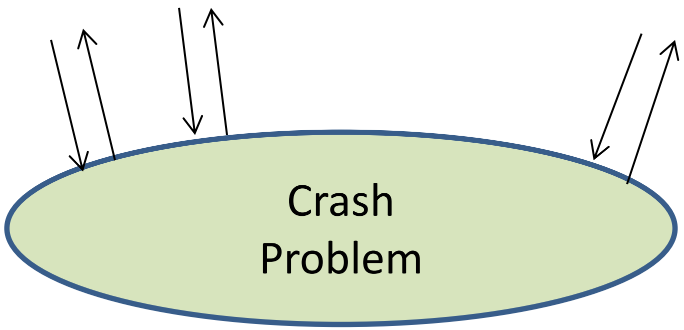


[Chandra, Toueg] definitions

- Time domain $\mathbb{T}$ (integers)
- Fixed, finite set Π of processes.
- Asynchronous message-passing system for inter-process communication.
- Stopping failures: Just stop (crash) without warning. Never recover.
- Failure pattern $F: \mathbb{T} \rightarrow 2^{\Pi}$
  - $F(t)$: The set of processes that have crashed by integer time t
  - These represent the inputs to the failure detector.
- History $H: \Pi \times \mathbb{T} \rightarrow R$, where R is some output domain
  - $H(i, t)$: The output of the FD for process i at time t.
- A failure detector is a mapping that associates, with each failure pattern, a nonempty set of possible histories.
- But we will avoid dividing time into slots…

Crash Problems

- Define a failure detector as a set of allowable traces, consisting of crash inputs and FD outputs.
- A special case of a crash problem.



- Fix finite set Π of process indices.
- Crash problem: (I, O, T), where
  - I is a set of input actions, partitioned into I_i, $i \in \Pi$
  - $crash_i \in I_i$, for every $i \in \Pi$
  - O is a set of output actions, partitioned into O_i, $i \in \Pi$
  - T is a set of (finite and/or infinite) sequences over $I \cup O$.

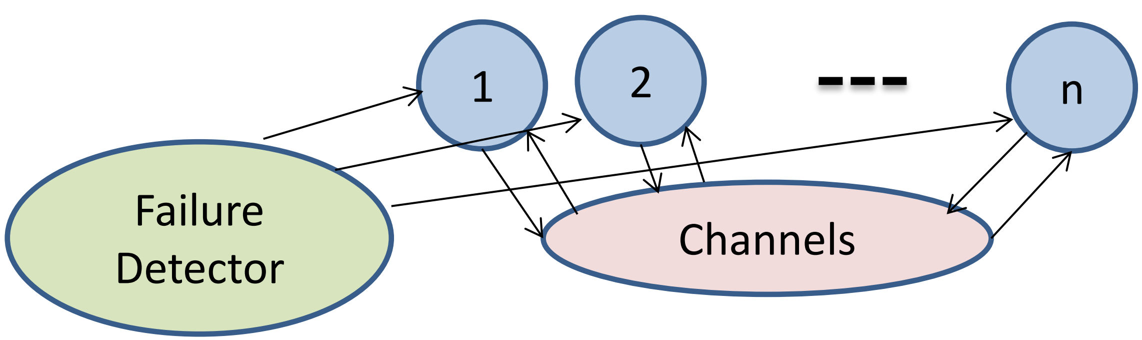
(Asynchronous) Failure Detectors

- A failure detector is a special kind of crash problem (I, O, T).
- Now $I = \{crash_i : i \in \Pi\}$, i.e., crashes are the only FD inputs.
- T satisfies additional properties:
  - Each $t \in T$ is valid:
    - No outputs at location i after $crash_i$.
    - Infinitely many outputs at non-failing locations
  - T is closed under sampling:
    - Take any trace in T,
    - For each faulty location i, delete any suffix of the O_i outputs.
    - The result is also a trace in T.
  - T is closed under constrained reordering:
    - Take any trace in T.
    - Reorder events, but without disturbing the order of FD output events at any location, and without moving crashes later, past any FD output events anywhere.
    - The result is also a trace in T.

Remarks

- Straightforward asynchronous formulation, I/O automata style.
- FD defined simply as a problem, in terms of allowable sequences of input and output events.
- Constraints are simple, and are just what is needed for basic results.
  - These are implicit in the earlier papers.

2. Examples:
Typical Failure Detectors



Failure Detector Examples

- [CT] defines eight failure detectors.
- All have outputs that are subsets of Π, representing the set of processes that are "suspected" to have failed.
- Different reliability guarantees, expressed in terms of:
  - Completeness: Reporting all failures, avoiding false negatives.
  - Accuracy: Not reporting failures incorrectly, avoiding false positives.

- $\diamond P$, Eventually Perfect FD: Each trace $t \in T$ satisfies:
  - Validity: t contains infinitely many outputs at each nonfaulty location, and contains no outputs after a crash at any faulty location.
  - Strong completeness: In some suffix of t, every faulty process appears in every output set.
  - Eventual strong accuracy: In some suffix of t, no nonfaulty process appears in any output set.

Failure Detector Examples

- $\diamond S$, Eventually Strong FD: Each trace $t \in T$ satisfies:
  - Validity: t contains infinitely many outputs at each nonfaulty location, and contains no outputs after a crash at any faulty location.
  - Strong completeness: In some suffix of t, every faulty process appears in every output set.
  - Eventual weak accuracy: In some suffix of t, there is some nonfaulty process that does not appear in any output set.

Failure Detector Examples

- Ω, the Leader Election failure detector [CHT]
- Each FD output is not a subset of Π, representing the processes that are "suspected" to have failed, but rather, an individual element of Π, representing a proposed leader.

- Ω, Leader Election FD: Each trace $t \in T$ satisfies:
  - Validity: t contains infinitely many outputs at each nonfaulty location, and contains no outputs after a crash at any faulty location.
  - Nonfaulty leader: In some suffix of t, there is some nonfaulty process i such that every output is equal to i.

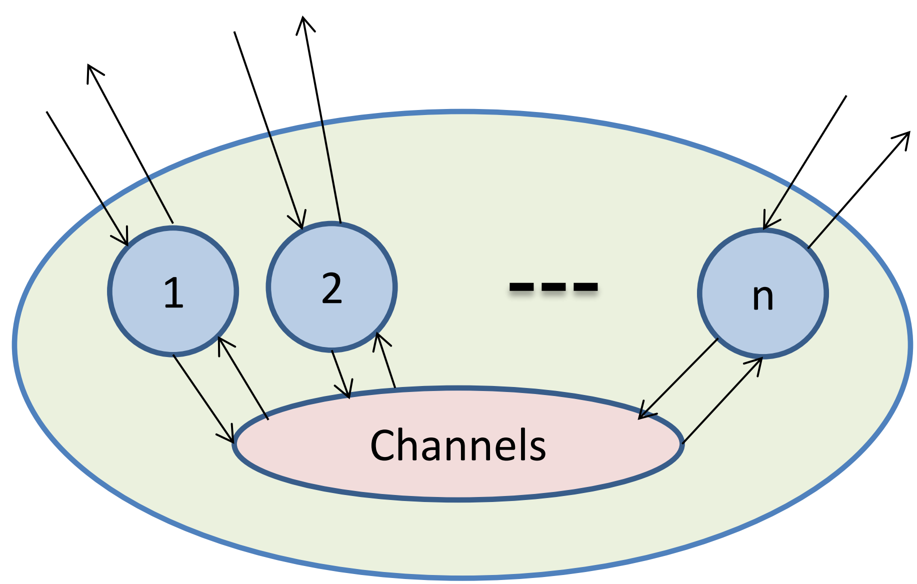
Remarks

- All the FDs in [CT], [CHT], and most others in the literature, satisfy our definition.
- A few do not:
  - Marabout failure detector [Guerraoui IPL 01]
    - It predicts future crashes.
  - D_k failure detectors [Bhatt, Jayanti, DISC 09]
    - They depend on precise times.
- Anomalies.

Implementing Failure Detectors

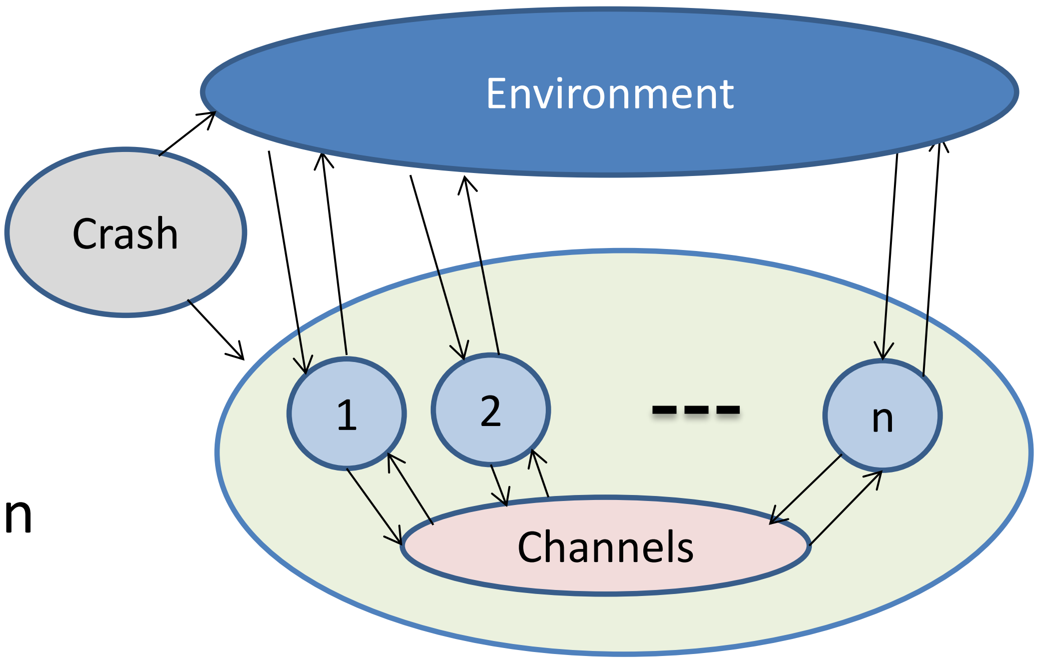
- To implement failure detectors like $\diamond P$, $\diamond S$, Ω, we assume that the underlying system satisfies some synchrony properties, e.g., bounds on message delay and on ratio of process speeds.
- E.g., implement $\diamond P$, assuming known bounds:
  - Periodically, each process sends "I'm alive" messages to all others.
  - Based on the known bounds, process i estimates when it should receive messages from all other processes.
  - If process i doesn't receive process $j's$ message by the estimated time, it adds j to its list of suspects.
  - Process i periodically outputs its list of suspects.
- E.g., implement $\diamond P$, assuming unknown bounds:
  - Similar, but based on guessed estimates for the bounds.
  - Start with default estimates.
  - Now process i could suspect a process j that actually has not crashed.
  - In that case, process i will receive a later message from j.
  - Then process i knows its estimates were too small, so it increases them, and removes j from its list of suspects.

3. Definitions:
Solving Crash Problems and Comparing Failure Detectors



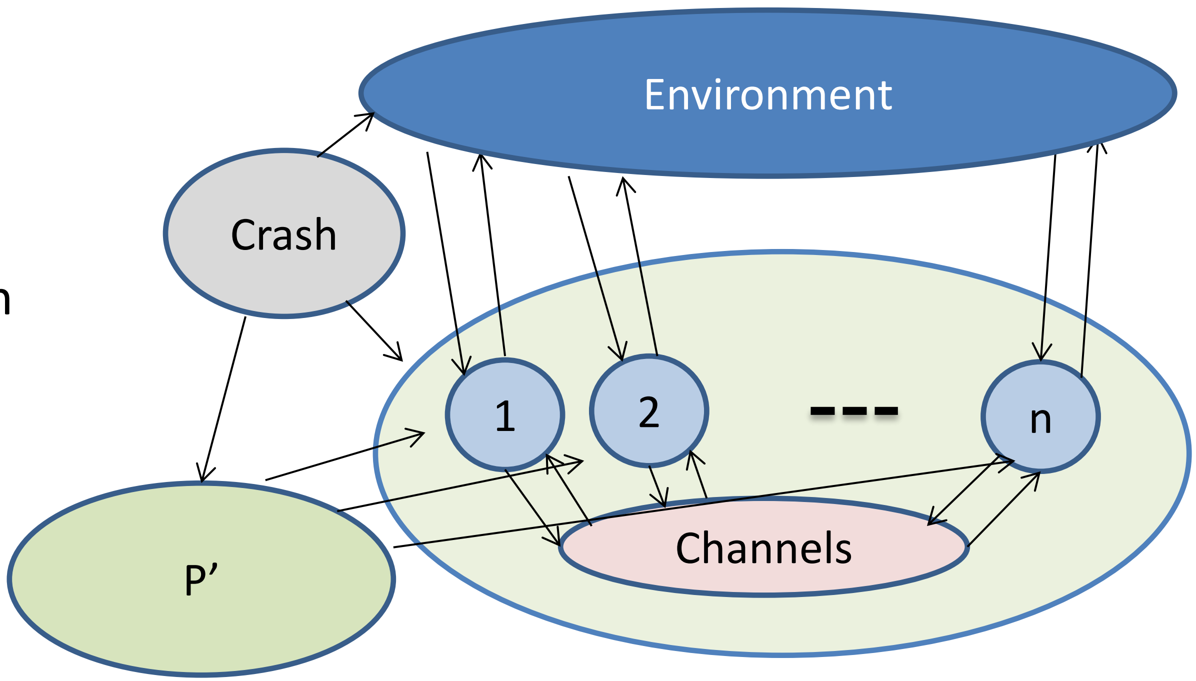
Solving a Crash Problem

- Distributed algorithm A solves crash problem $P = (I, O, T)$ in environment E provided that:
  - The signatures match, e.g., inputs and outputs of A at the external boundary are I and O.
  - When A is composed with the channels, E, and $Crash$, the projection of every fair trace of the composition on the P actions is a sequence in T.



Solving One Crash Problem Using Another

- Distributed algorithm A solves crash problem $P = (I, O, T)$ in environment E using crash problem $P' = (I', O', T')$ provided:
  - The signatures match.
  - When A is composed with channels, E, and $Crash$, for any fair trace t of the composition, if the projection of t on the P' actions is in T', then the projection of t on the P actions is in T.



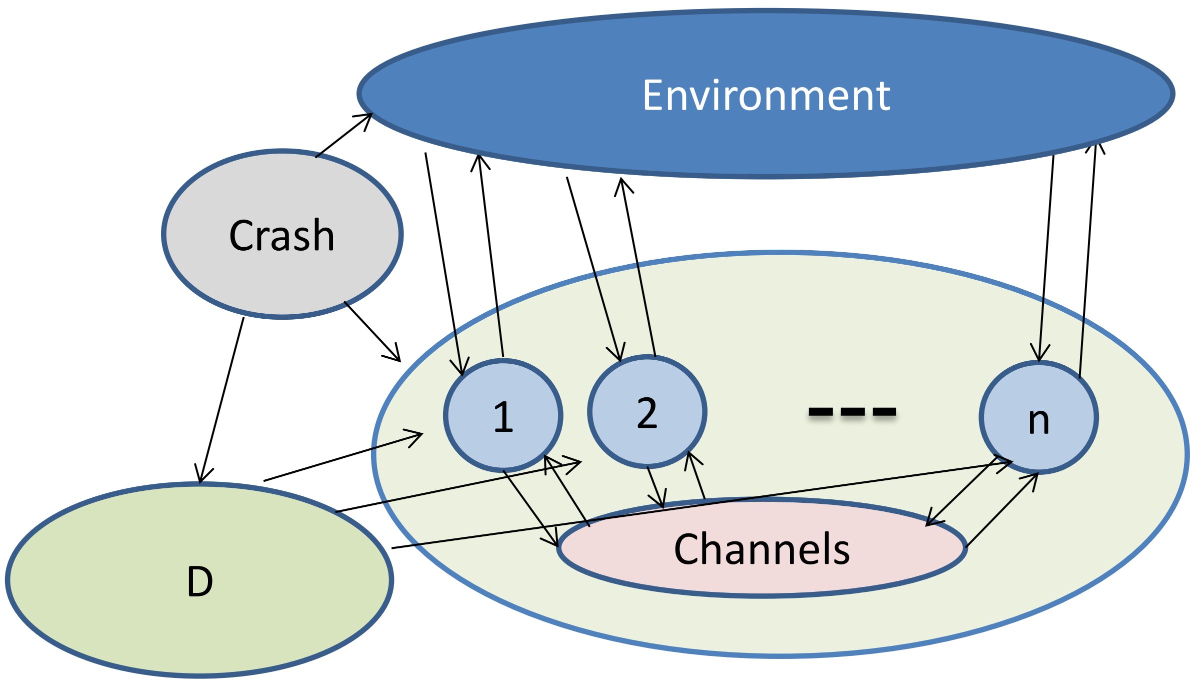
Specializing to Failure Detectors

- Since FDs are now just special cases of crash problems, our definitions specialize to define:
  - Distributed algorithm A solves (implements) FD D.
  - A solves crash problem P in environment E using failure detector D'.
  - A solves failure detector D using crash problem P'.
  - A solves failure detector D using failure detector D'.

Comparing FDs, and other crash problems

- If P and P' are crash problems, then define $P \leq_E P'$ if there is some distributed algorithm A that solves P in environment E using P'.
- If D and D' are failure detectors, then define $D \leq D'$ if there is some distributed algorithm A that solves D using D'.
- Transitivity results (complicated a bit by the need for renaming).

4. Self-Implementability



Self-Implementability

- A basic property of failure detectors should be that any FD should be able to "implement" itself, i.e., for any FD D, there should be a distributed algorithm A that implements D using D itself.
- Using the [CT] definitions, this isn't always true!
- Instantaneously Perfect failure detector $\mathcal{P}^+$
  - [Charron-Bost et al., 2010]
  - At each time t, it outputs the exact set of processes that have crashed by that time.
  - $\mathcal{P}^+$ is not self-implementable (using an asynchronous distributed algorithm).
- Q: So how could we rule out such cases?
- Q: How should a self-implementing algorithm work?

A Simple Algorithm

- Design a distributed algorithm A that implements failure detector D using D itself.
- Formally, we must rename the actions of D, i.e., A implements D using D', which is a renamed version of D.

- Algorithm A, process i:
  - Queue up inputs that arrive from D', in order.
  - Output them in the same order.

- Algorithm A doesn't work for $\mathcal{P}^+$, because the processes convey out-of-date information.
- Q: What failure detectors does it work for?
- A: All FDs that satisfy the new (asynchronous) definition.

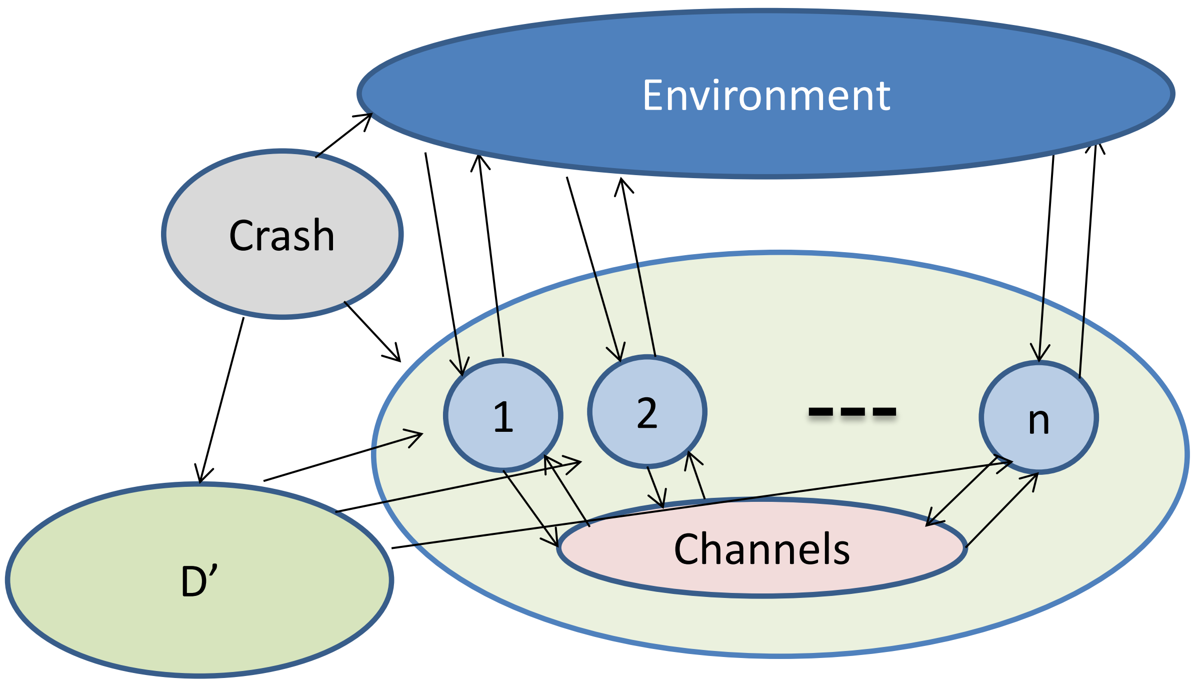
Why the Algorithm Works

- Since D' is a failure detector, we have by the definition of an FD:
  - Each $t \in T'$ is valid.
    - No outputs at already-crashed locations.
    - Infinitely many outputs for nonfaulty processes.
  - T' is closed under sampling.
    - Can remove any suffix of outputs at faulty processes.
  - T' is closed under constrained reordering.
    - Can delay outputs, if we don't disturb the order of output events at each location, or move crashes later, past outputs.
- In any execution, the external trace t' produced by the algorithm is a constrained reordering of a sampling of the trace of D'.
- This is because the only changes the algorithm makes to the output sequence are:
  - Remove a suffix of outputs of a crashed process (if the process crashes with a nonempty queue).
  - Move some FD outputs later (because of the delay in the queue).

Remarks

- We obtained self-implementability by adding some constraints to the FDs.
- Constraints are minimal, satisfied by nearly all the interesting FDs in the literature.
- But not satisfied by some “oddities”.

5. Examples: Comparing Typical FDs



Typical FDs

- $\diamond P$, Eventually Perfect FD
- $\diamond S$, Eventually Strong FD
- Ω, Leader Election FD
- Many others; see, e.g., [CT]
- Compare based on implementability, recall:
  - If D and D' are failure detectors, then $D \leq D'$ if there is a distributed algorithm A that solves D using D'.
- Claim 1: $\diamond S \leq \diamond P$
- Proof:
  - $\diamond P$ imposes stronger constraints than $\diamond S$.
  - So we can just use the self-implementation algorithm.

$\Omega \leq \diamond P$

- Claim 2: $\Omega \leq \diamond P$
- Proof:
  - Algorithm (for process i):
    - Upon receiving a set $susp$ of suspected processes from the $\diamond P$ service, choose the smallest id that is NOT in the set $susp$.
    - Put that in an output queue, perform outputs from the output queue.
  - Eventually, $\diamond P$ outputs exactly the set of faulty processes (from some point on, everywhere).
  - So eventually, each nonfaulty process will output the smallest id of a process that isn't in that set, i.e., the smallest id of a nonfaulty process.
  - That satisfies the requirement for Ω.

$\diamond S \leq \Omega$

- Claim 3: $\diamond S \leq \Omega$
- Proof:
  - Algorithm (for process i):
    - Upon receiving an id $leader$ from the Ω service, construct the set of all ids except for $leader$, that is, the set $\Pi - \{leader\}$.
    - Put an entry containing that set at the end of the output queue.
    - Perform outputs from the output queue.
  - Eventually, Ω outputs the same, nonfaulty process (from some point on, forever).
  - Check key requirements for $\diamond S$:
  - Strong completeness: In some suffix of t, every faulty process appears in every output set.
    - Yes, because only one, nonfaulty process is left out of the final set.
  - Eventual weak accuracy: In some suffix of t, there is some nonfaulty process that does not appear in any output set.
    - Yes, the final leader process.

Summary

$$\boxed{\Diamond P} \geq \boxed{\Omega} \geq \boxed{\Diamond S}$$

- Q: What about the other relationships?

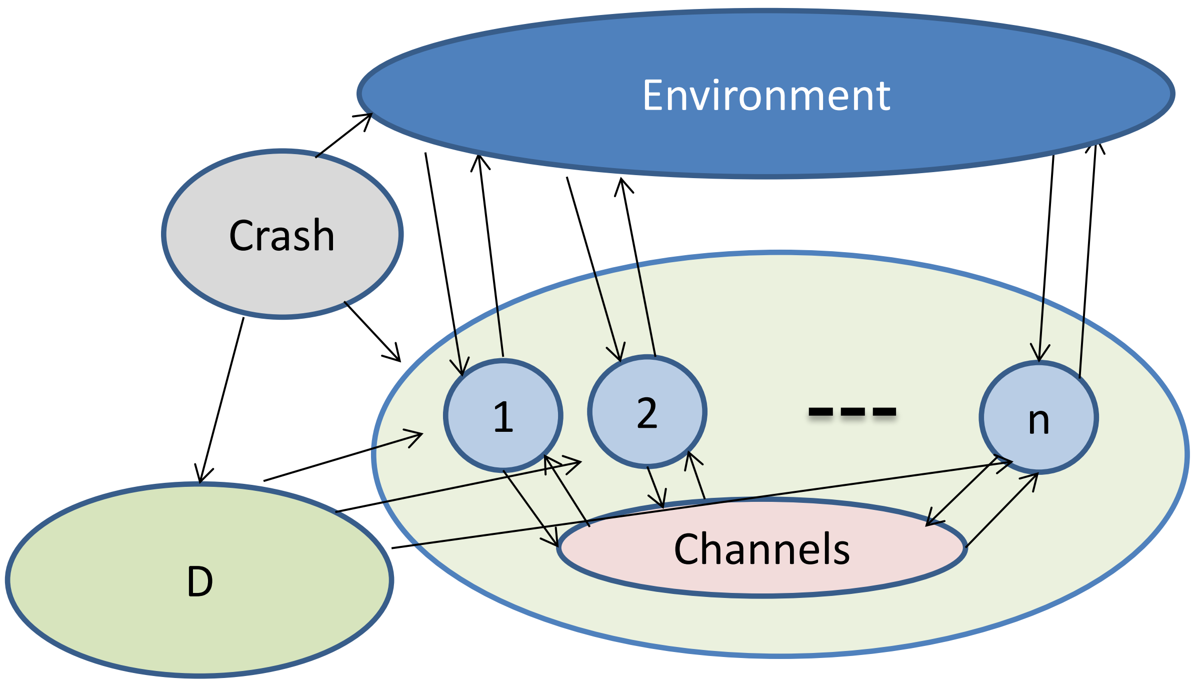
Other reductions

- The reductions so far have been simple and local.
- Some others in [CT] require significant distributed processing.
- These involve FDs with a weaker completeness condition:
- Weak completeness: For every process i that is faulty in t, there is some nonfaulty process $k(i)$ such that: In some suffix of t, i appears in $k(i)$'s output set.
- Compare with:
- Strong completeness: In some suffix of t, every faulty process appears in every output set.
- The reductions involve distributed algorithms by which processes exchange and update their failure information, LTTR.

Overview

1. Definitions: Crash problems, failure detectors
2. Typical FDs: $\diamond P$, $\diamond S$, Ω
3. Definitions: Solving crash problems, comparing FDs
4. Self-implementability of FDs
5. Comparing typical FDs
6. Solving particular problems using FDs
   a) Consensus using Ω
   b) Dining Philosophers using a local version of $\diamond P$
7. Definitions: Weakest FDs to solve problems
8. Weakest FDs for particular problems
   a) Local $\diamond P$ is a weakest FD for wait-free Dining Philosophers
   b) Ω is a weakest FD for fault-tolerant consensus.

6. Solving Problems Using FDs: Consensus and Dining Phillosophers



Fault-Tolerant Consensus Using FDs

- Original motivation for FDs: Give a simple, practical abstract service that we can add to a fault-prone asynchronous distributed system in order to solve consensus .
- [CT] describe an algorithm using $\diamond S$ that solves consensus for $f < \frac{n}{2}$.
- Complicated…and looks a lot like Paxos.

- So instead, let’s try using Paxos + Ω.
- Basic idea: Use Ω to select leaders, who are the processes that start ballots.
- Use simple majorities rather than general quorums.

Consensus Algorithm Using Ω

- Paxos-style algorithm using Ω:
  - Process i may start a new ballot only when the latest local output from Ω is i, that is, when process i thinks that it is the leader.
  - A process abstains from a ballot only if it has heard of a ballot with a larger identifier.
  - All processes respond to all phase 1 and phase 2 messages.
  - In phase 2, a process either sends a positive ack containing a vote for the ballot, or a negative ack saying that it has abstained from the ballot (because it has heard of a larger one).
  - The leader waits for a majority of responses for each phase.
  - If these yield enough votes to decide, the leader does so. If not, then it abandons the ballot and starts a new one with a larger ballot id.
  - A leader should not abandon old ballots and start new ones "unnecessarily".
- Details LTTR.

Fault-Tolerant Consensus Using FDs

- So, we can solve fault-tolerant consensus for $f < \frac{n}{2}$, provided we have an eventually-stable, nonfaulty leader.

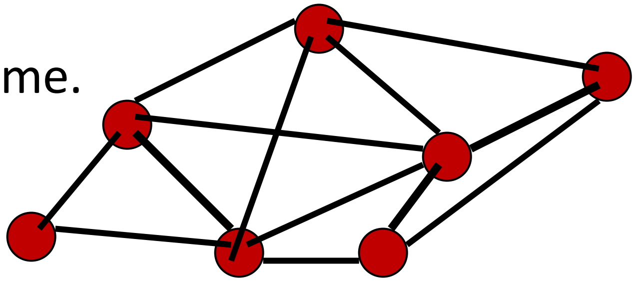
- [CT] also prove a lower bound saying that, even with $\diamond P$, it's impossible to solve consensus with $f \geq \frac{n}{2}$ ...

Lower Bound for Consensus Using FDs

- Theorem: Even with $\diamond P$, it's impossible to solve fault-tolerant consensus with $f \geq \frac{n}{2}$.
- Proof:
  - Assume $f \geq \frac{n}{2}$ and get a contradiction.
  - Uses a partitioning argument.
  - Divide the processes into two groups of size between 1 and f, one with inputs = 0 and one with inputs = 1.
  - Each group suspects the other, receives no messages from the other group (delayed), finishes on its own.
  - $\diamond P$ doesn't help, because its guarantees are required to hold only eventually. In the short term, it can give the processes information consistent with their own suspicions.
  - Paste the two executions together and get the usual sort of contradiction.

Wait-Free Dining Philosophers Using FDs

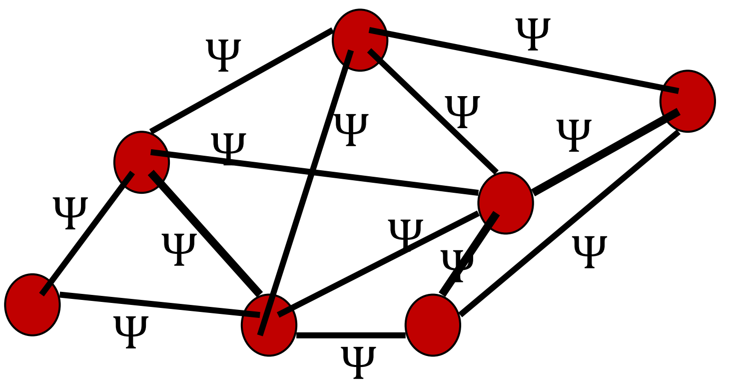
- [Pike, Song, and Sastry, ICDCN 08]
- An algorithm that solves the wait-free eventual Dining Philosophers problem, using $\diamond P$.
- Q: What is that?
- DP without failures :
  - Processes at the nodes of an undirected exclusion graph:
  - Trying, critical, exit, remainder regions.
  - Neighbors should not be critical at the same time.
  - Trying process should eventually go critical.
  - Solve this using a message-passing model.



- DP with process stopping failures:
  - Eventual exclusion: In some suffix, no two non-failed neighbors are simultaneously critical.
  - Wait-freedom: Every nonfaulty trying process eventually goes critical, even if other processes fail.

DP Algorithm Using $\diamond P$

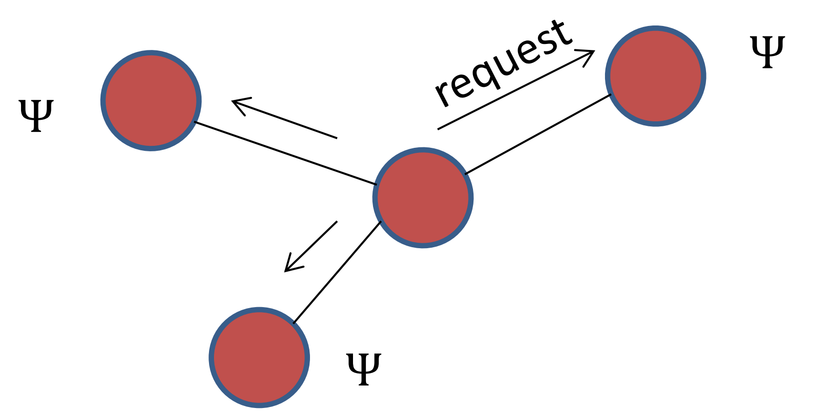
- Two primary mechanisms, forks and priorities.
- Forks:
  - Associate a fork with each edge in the graph.
  - Represented explicitly, by a token that may reside at the process at either end of the edge, or be "in transit" in one of the channels between them.



- Priorities:
  - Each process maintains its own priority value, made unique by using process ids as tiebreakers.

Basic fork collection scheme, no failures

- Trying process i tries to collect the forks for all incident edges.
- Requests all missing forks.
- Neighbors might or might not send the requested forks:
  - Processes in the remainder region, and lower-priority trying processes, always honor fork requests.
  - Processes in the critical (or exit) region, and higher-priority trying processes, always defer fork requests (waiting for conditions to change).
- When process i has all forks, it enters the critical region.



Basic fork collection scheme, no failures

- Trying process i tries to collect the forks for all incident edges.
- Requests all missing forks.
- Neighbors might or might not send the requested forks:
  - Processes in the remainder region, and lower-priority trying processes, honor fork requests.
  - Processes in the critical (or exit) region, and higher-priority trying processes, defer fork requests.
- When process i has all forks, it enters the critical region.
- Upon exiting the critical region, process i reduces its priority below those of all its neighbors and honors all deferred fork requests.
- To keep track of priorities, each process sends its latest priority on every message.

- This clearly guarantees exclusion.
- In the absence of failures, it also guarantees lockout-freedom.

What about failures?

- Now consider process stopping failures.
- We want wait-freedom: Every nonfaulty trying process eventually goes critical, even if other processes fail.
- In the current scheme, if a neighbor has crashed, it never sends the fork, which means that its neighbors may be stalled, and so their neighbors may be stalled,…
- So, modify the rule for entering the critical region, using $\diamond P$:
  - Trying process i may enter the critical region if for every incident edge (i, j), either process i has the fork, or process i believes that process j has failed, because j is included in the most recent output set of $\diamond P$ at location i.
- And modify the rule for exiting the critical region:
  - Reduce the priority, but now we can't be sure this will be less than that of all neighbors.
  - Honor all deferred fork requests anyway.

Guarantees

- Eventual exclusion: In some suffix, no two non-failed neighbors are simultaneously critical.
- Wait-freedom: Every nonfaulty trying process eventually goes critical, even if other processes fail.

Wait-Freedom

- Wait-freedom: Every nonfaulty trying process eventually goes critical, even if other processes fail.
- Proof sketch:
  - Eventually, all crashes have happened.
  - By strong completeness, from some point on, $\diamond P$ always reports the failure of all faulty processes.
  - So a trying process i will not be blocked forever by a failed neighbor.
  - Process i must still obtain all forks from nonfaulty neighbors.
  - This depends on careful management of the priorities.
  - Inconsistent views of priorities could result in deadlock: If each of two neighbors thinks it has higher priority than the other, then neither might send a requested fork.

Wait-Freedom

- Wait-freedom: Every nonfaulty trying process eventually goes critical, even if other processes fail.
- Proof sketch, cont'd:
  - One priority scheme that works:
    - Priorities are of the form (Integer, process ID)
    - Reduce priority by some arbitrary amount (not necessarily lower than neighbors) when leaving the critical region.
    - Change priority only when leaving the critical region.
  - Highest priority nonfaulty trying process in the entire network gets all its needed forks, enters the critical region, and lowers its priority.
  - So eventually, every nonfaulty trying processes becomes the highest priority nonfaulty trying process and enters the critical region.

Eventual Exclusion

- Eventual exclusion: In some suffix, no two non-failed neighbors are simultaneously critical.
- Proof:
  - Eventually, all crashes have happened.
  - By eventual strong accuracy, eventually $\diamond P$ stops reporting that correct processes have crashed, i.e., it reports only actual crashes.
  - Let π be a point in the execution after all this has happened.
  - Consider what happens after point π, and after the effects of old errors disappear.
  - Consider any two neighbors i and j that both start trying after point π.
  - In order to enter the critical region, each must actually obtain the fork corresponding to the edge between them, and will keep it during its time in the critical region.
  - So they can't end up in the critical region at the same time.

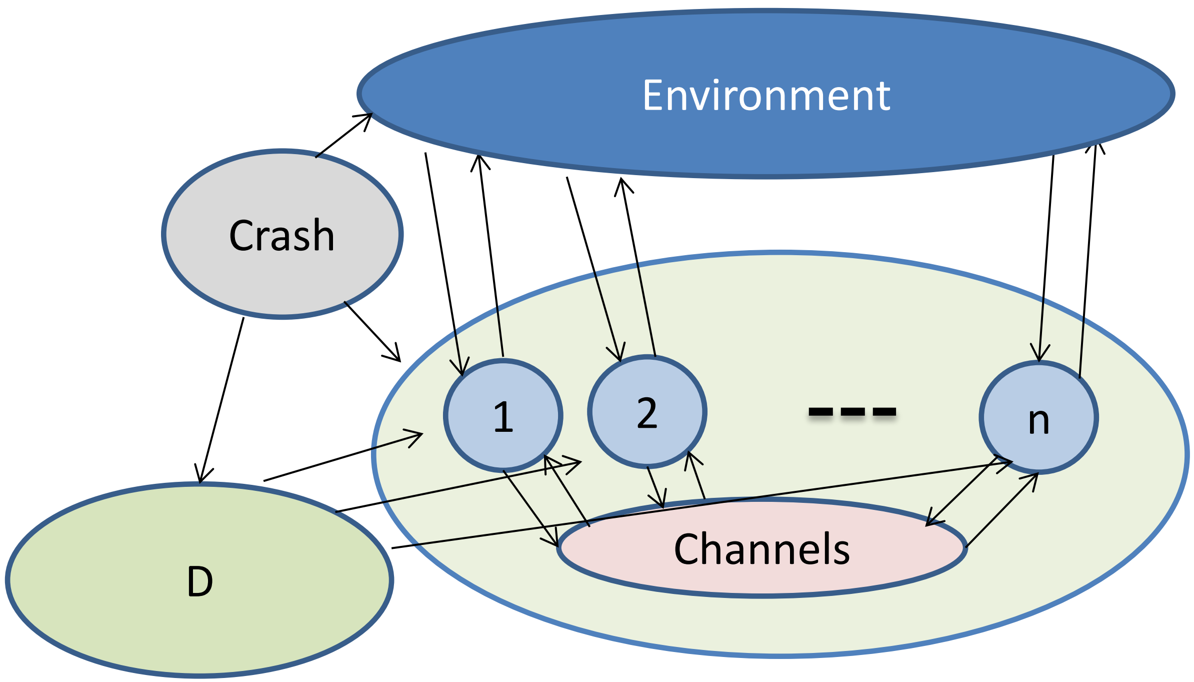
Remarks

- Power of FDs:
  - These results show that FDs can help in solving other problems besides fault-tolerant consensus.
- Local $\diamond P$:
  - Here, $\diamond P$ could be weakened to a “local” version, where the FD at each location reports only about the failure status of neighboring locations.
- Behaving correctly for “sufficiently long”:
  - Eventual FDs like $\diamond P$ seem strong, because they must behave correctly from some point on, forever.
  - In most cases, behaving correctly for “sufficiently long” is good enough.

Overview

1. Definitions: Crash problems, failure detectors
2. Typical FDs: $\diamond P$, $\diamond S$, Ω
3. Definitions: Solving crash problems, comparing FDs
4. Self-implementability of FDs
5. Comparing typical FDs
6. Solving particular problems using FDs
   a) Consensus using Ω
   b) Dining Philosophers using a local version of $\diamond P$
7. Definitions: Weakest FDs to solve problems
8. Weakest FDs for particular problems
   a) Local $\diamond P$ is a weakest FD for wait-free Dining Philosophers
   b) Ω is a weakest FD for fault-tolerant consensus.

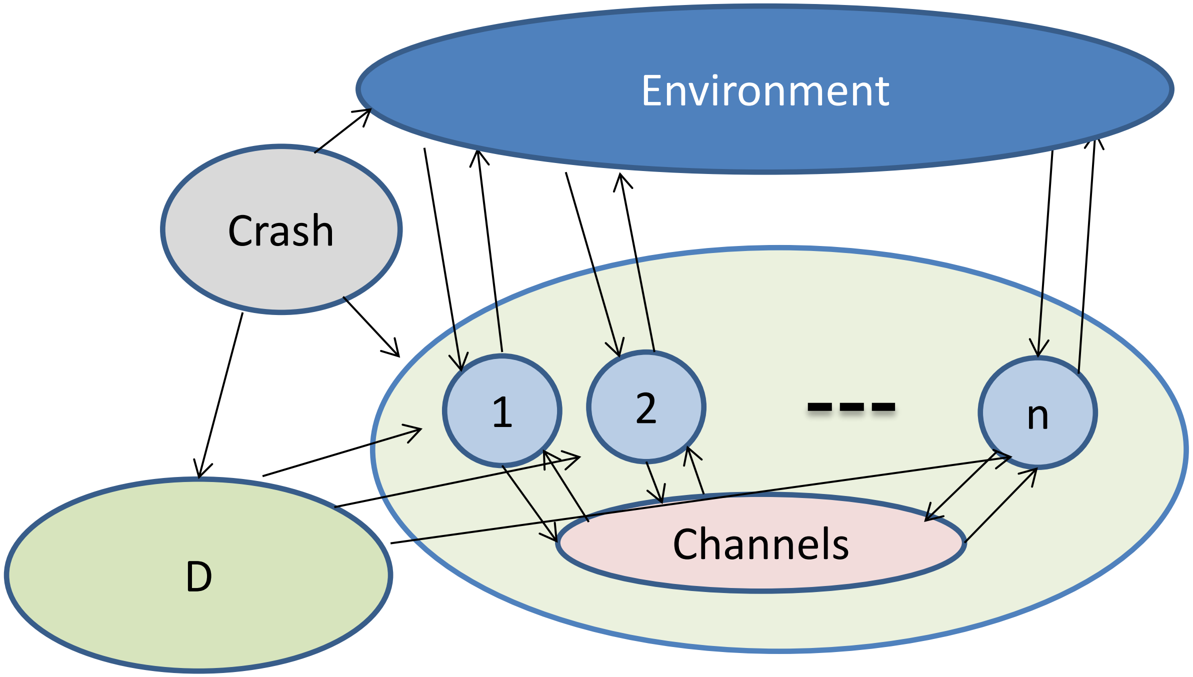
7. Weakest Failure Detector definitions



Weakest Failure Detectors

- Failure detector D is a weakest FD (WFD) for problem P in environment E iff:
  - $P \leq_E D$, that is:
    - D is sufficient to solve P in environment E, i.e.,
    - There is a distributed algorithm A that solves P in environment E using D.
  - For any failure detector D' that is sufficient to solve P in environment E, $D \leq D'$, that is:
    - D' is sufficient to implement D, i.e.,
    - There is a distributed algorithm A that implements D using D'.
- Failure detector D is a weakest FD for problem P in a class of environments if it is a weakest FD for P in every environment in the class.

8. Weakest Failure Detectors for Dining Philosophers and Consensus



A Weakest FD for Dining Philosophers

- [Sastry, Pike, Welch, SPAA 09]
- We showed that $\diamond P$ solves wait-free Dining Philosophers.
- A local version of $\diamond P$ suffices, where each location reports only about the failure status of its neighboring locations.
- This local version of $\diamond P$ is a weakest failure detector for wait-free Dining Philosophers.

Definition of Local $\diamond P$

- Outputs at each location are a subset of the set of neighboring locations.
- Each trace $t \in T$ satisfies:
  - Validity
  - Strong completeness: In some suffix of t, every faulty neighbor appears in every output set.
  - Eventual strong accuracy: In some suffix of t, no nonfaulty neighbor appears in any output set.

Weakest FD for Dining Philosophers

- Theorem: Local $\diamond P$ is a weakest FD for wait-free Dining Philosophers.
- More strongly, it is a representative FD: there is a distributed algorithm that implements Local $\diamond P$ using finitely many instances of DP.
- Proof:
  - An interesting technical construction.
  - See Fall, 2014 course slides, or the [SPW] paper.

A Weakest FD for Consensus

- [Chandra, Hadzilacos, Toueg], [Lynch, Sastry]
- Theorem: Assume $f < n/2$. Then Ω_f is a weakest FD for f-fault-tolerant consensus.
- Here, Ω_f is assumed to behave like Ω, in executions with at most f failures.

Proof Strategy

- Start with any distributed algorithm A and failure detector D such that A uses D to solve f-fault-tolerant consensus.
- Use A and D to construct a new distributed algorithm A_Ω that implements Ω_f.
- Part 1 of the proof: Analyze the structure of executions of A and D that allows them to solve consensus.
  - Define an execution tree of A with D.
  - Show that the execution tree contains a decision gadget.
  - A decision gadget has a critical process, which must be nonfaulty.
- Part 2: Devise a distributed algorithm that, using the actual FD D, emulates algorithm A running with D, and uses the emulation results to extract such a nonfaulty process. This yields the properties required for Ω_f.

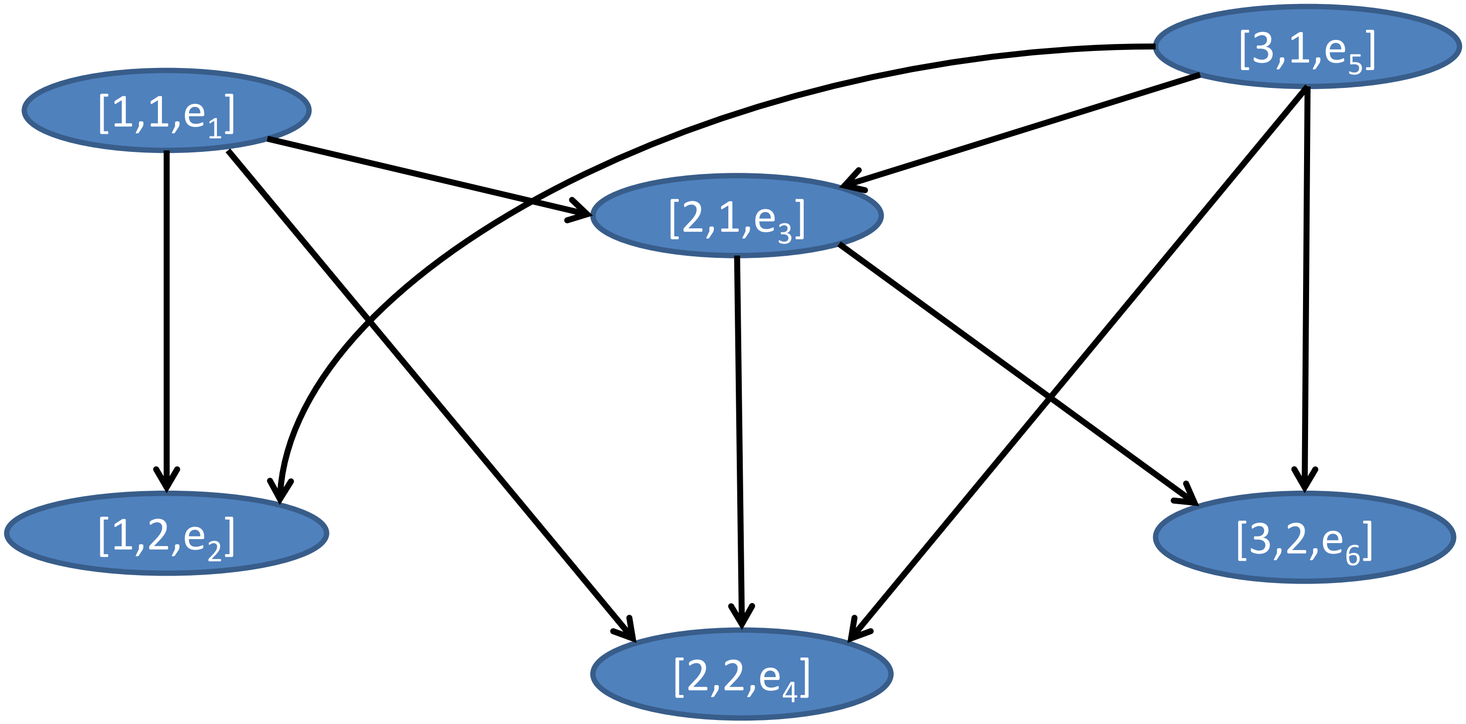
Part 1

- Analyze the structure of executions of A with D that allows them to solve consensus.
- Do this in several stages:
  - Define observation DAGs; each observation DAG G represents some possible outputs of D, plus some ordering relationships among these outputs.
  - If G is consistent with the outputs in some actual execution of D, then it's viable.
  - Define the execution tree of A for any particular observation DAG G.
  - Show that, if G is viable, the resulting execution tree contains a decision gadget for solving consensus.
  - Such a decision gadget has a critical process, which must be nonfaulty.

Observation DAGs

- Consider a particular failure detector $D = (I, O, T)$.
- An observation DAG G for D has vertices of the form $[i, k, e]$, where i is a location, k is a positive integer, and e is an output in O.
  - $[i, k, e]$ means that D's k^{th} output at location i is e.
  - For each i and k, at most one triple.
  - For each i, the values of k form a prefix of the positive integers.
- Location i is live in G if G contains infinitely many vertices for i.
- G has edges representing some ordering relationships between the vertices.
  - Order the triples for the same i, according to values of k.
  - Transitively closed.
  - For every vertex $[i, k, e]$, and every j that is live in G, there is an edge from $[i, k, e]$ to some vertex for j.

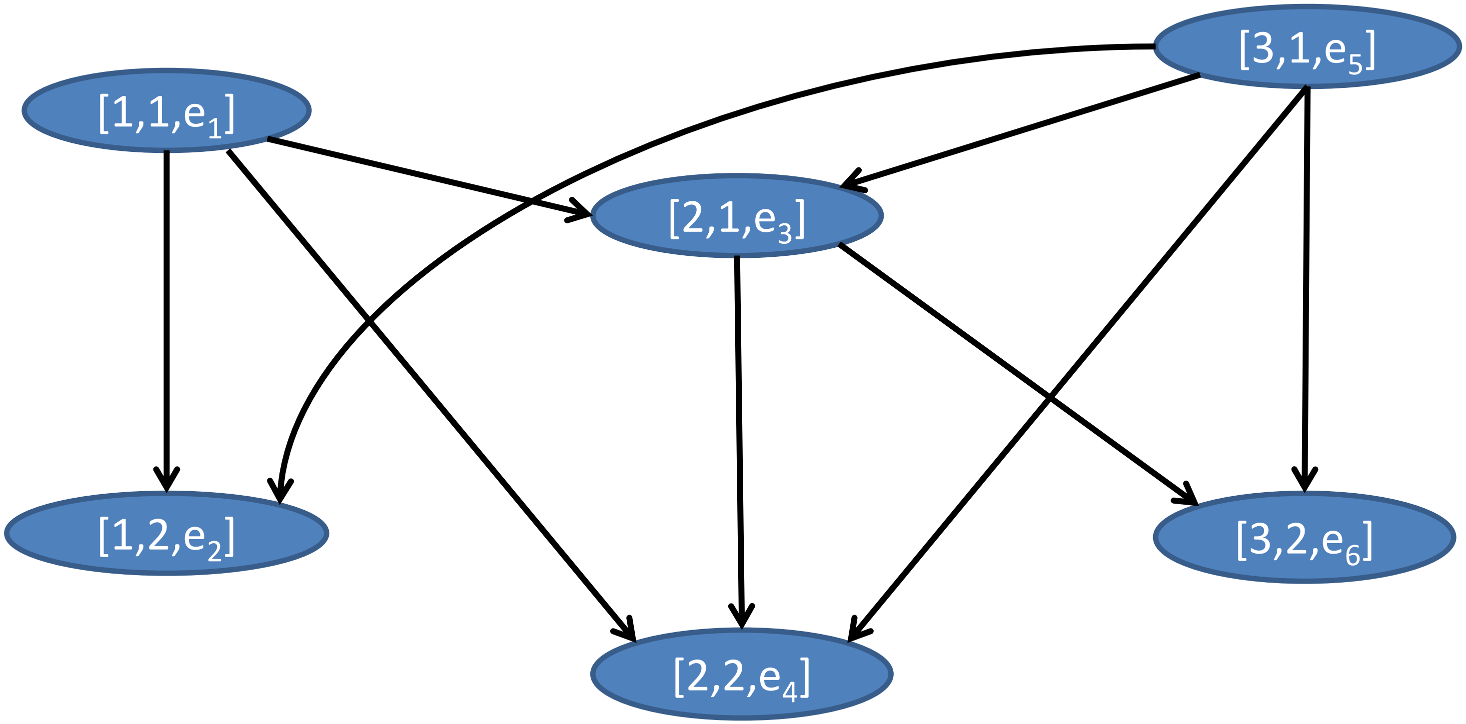
An observation DAG



Viable observations DAGs

- So far, the outputs e in the observation DAG could be any values from O, not related to the actual behavior of $D = (I, O, T)$.
- Define G to be viable for D provided that there is a topological ordering of the vertices of G whose event sequence is exactly the output subsequence of some $t \in T$.

A viable observation DAG



The total ordering might be [1,1,e1], [3,1,e5], [1,2, e2],[2,1,e3],[2,2,e4],[3,2,e6].
The full (valid) trace in T might be e1, e5, e2, crash1, e3, e4, crash2, e6, crash3.

Why observation DAGs?

- Observation DAGs are used in the emulation algorithm in Part 2, which is used to implement Ω_f.
- Processes simulate many executions of A with D, in parallel.
- Processes have access to the actual FD D.
- They receive local outputs from D, and communicate them to other nodes.
- They can't determine the actual total ordering of D's outputs, just a partial ordering based on causality.
- Represent this information by an observation DAG:
  - Vertices correspond to the FD outputs.
  - Edges defined based on communication.
- Processes simulate executions of A using FD outputs given by various paths through the observation DAG.

Execution tree for a DAG G

- Similar to the execution trees used for FLP.
- Now our system consists of:
  - Processes,
  - (FIFO reliable) Channels,
  - the Environment (divided into one piece per location), and
  - an FD.
- Each tree edge is labeled with one of:
  - A process task (assume one task per process),
  - A channel task (one task per channel),
  - An environment task, or
  - FD_i for some i.
- From an internal node N of the execution tree, we have:
  - An edge for each process, channel, and environment task.
  - An edge labeled FD_i for each vertex $[i, k, e]$ that appears in G and is ordered strictly after all vertices that already appear in the path in the tree leading to node N; if there are none, then just one FD_i edge.

Execution tree for a DAG G

- Each tree edge is labeled with one of:
  - A process task, channel task, environment task, or
  - FD_i for some i.
- Then tag the nodes and edges of the tree with system states and actions, in the natural way.
  - Tag each FD_i edge with the corresponding $[i, k, e]$ vertex from G; if none, then tag with ⊥.
  - Tag a $Proc_i$ edge from node N with action a if and only if:
    - a is enabled from the state associated with node N, and
    - N has an outgoing non-⊥ FD_i edge.
  - Assume "enough" determinism so these actions are uniquely determined.
  - Likewise for Env_i edges.

Execution tree for a DAG G

- The tree represents all executions (fair and unfair) of the system in which the FD output sequence corresponds to a path in the observation DAG G.
- A fair branch of the execution tree is one in which:
  - Each process, channel, and environment task appears infinitely often.
  - For each i that is live in G, FD_i labels occur infinitely often.

- Theorem, paraphrased: Each fair branch of the execution tree corresponds to a fair execution of algorithm A in which the FD events form a trace in T.
- Theorem, a bit more carefully: Let $D = (I, O, T)$ be a strong-sampling FD, G a viable observation for D. For every fair branch b of the tree, there is a fair execution α of A with D such that :
  - $exe(b)$ is the same as α with the crashes removed, and
  - α restricted to the FD events is in T.

Execution tree for DAG G

- Theorem: Let $D = (I, O, T)$ be a strong-sampling FD, G a viable observation for D. For every fair branch b of the tree, there is a fair execution α of A with D such that :
  - $exe(b)$ is α with the crashes removed, and
  - α restricted to the FD events is in T.
- Proof idea:
  - Obtain α by inserting crashes into $exe(b)$, as follows.
  - Since G is viable, we can identify a topological ordering of all the vertices of G whose event sequence is exactly the output subsequence of some trace $t \in T$. This t includes crashes.
  - Obtain a strong sampling t' of t whose FD output events are exactly those in $exe(b)$. By closure under strong sampling, we also have $t' \in T$. Note that t' includes crashes.
  - Insert crashes into $exe(b)$ in the positions at which they occur in t', to get α.

Execution tree for DAG G

- Theorem: Let $D = (I, O, T)$ be a strong-sampling FD, G a viable observation for D. For every fair branch b of the tree, there is a fair execution α of A with D such that :
  - $exe(b)$ is α with the crashes removed, and
  - α restricted to the FD events is in T.

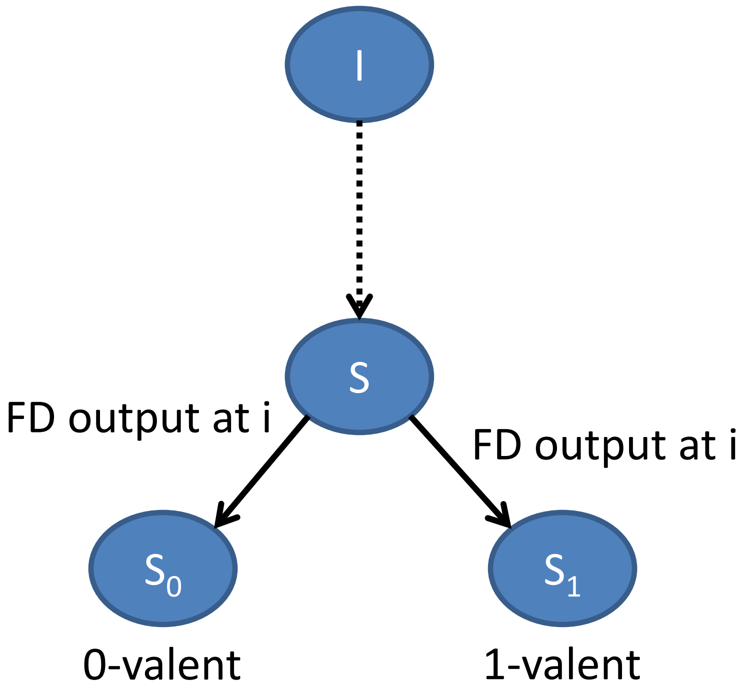
- Summary: The execution tree describes fair executions in which A solves consensus using certain traces of $D -$ those compatible with a particular observation DAG.

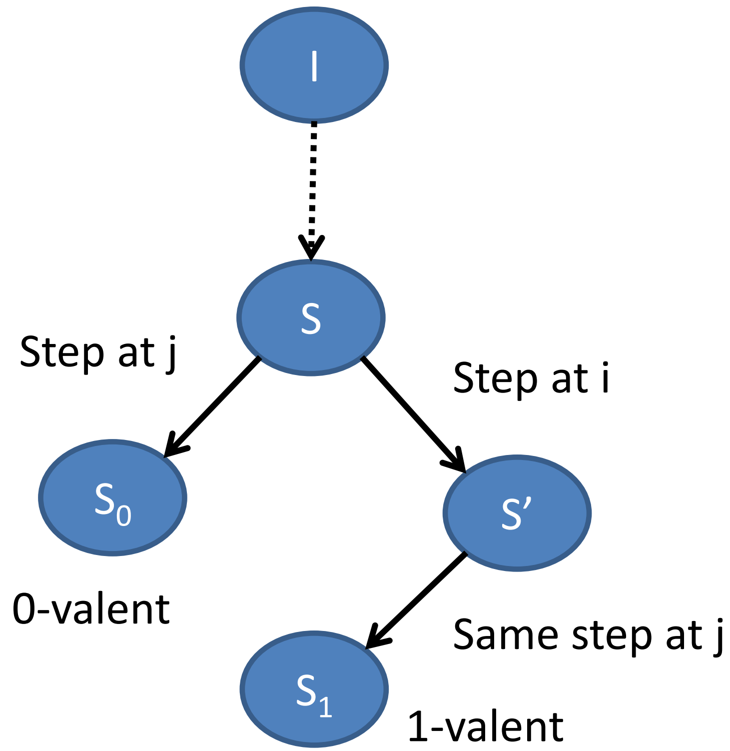
- Q: How does the decision get made?

Decider gadgets

- The transition from a bivalent configuration to a univalent configuration must happen as a result of a “decider” gadget, which in this case can be either a “fork” or a “hook”:

Forks and Hooks

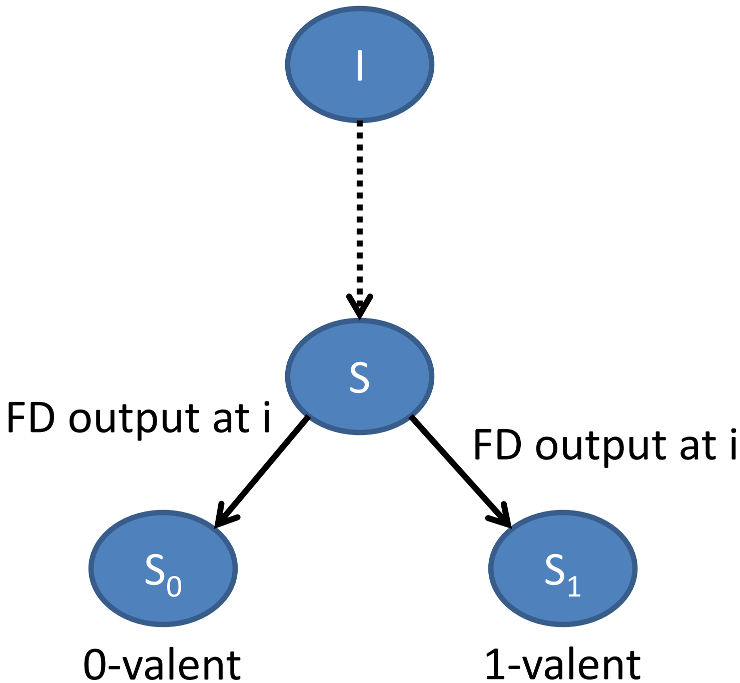


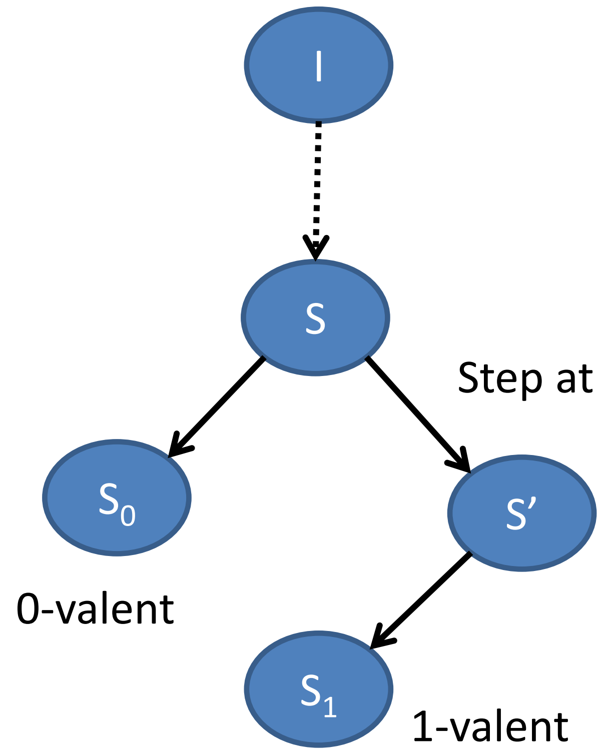


- S_0 and S_1 result from different FD outputs at the same process i.
- Process i is the deciding process of the fork.

- S_0 and S_1 result from the same step at some process j, performed before or after a step at some process i.
- Process i is the deciding process of the hook.

And Furthermore:





- The deciding process i of any fork or hook must be nonfaulty.
  - Arguments are like the FLP decider case analyses.
- We can identify a "smallest" hook/fork in the tree (not easy).

Part 1: Recap

- We analyzed the structure of executions of A with D that allows them to solve consensus:
  - Defined observation DAGs; each observation DAG G represents some possible outputs of D, plus some ordering relationships among these outputs.
  - If G is consistent with the outputs in some actual execution of D, then it's viable.
  - Defined the execution tree of A for any particular observation DAG G.
  - Claimed that, if G is viable, then the execution tree contains a decision gadget for solving consensus.
  - Such a decision gadget has a critical process, which must be nonfaulty.
  - We can identify a "smallest" decision gadget in the tree.

Part 2: Distributed Algorithm using D to implement Ω_f

- Processes continually exchange their local FD outputs.
- Algorithm for process i: Periodically do:
  - Build an observation DAG based on the FD outputs received so far and the known temporal orderings between them (determined by Lamport causality).
  - Construct the execution tree based on the current (finite) observation DAG.
  - If this tree contains a decision gadget, then:
    - Determine the "smallest" decision gadget.
    - Output the id of the critical process of this decision gadget.

Correctness (sketch)

- In the limit:
  - The observation DAGs at all nonfaulty processes converge to the same (infinite) observation DAG, G^{∞}, and
  - The execution trees at all nonfaulty processes converge to the same execution tree, $R(G^{\infty})$.
- The limiting execution tree $R(G^{\infty})$ must have a decision gadget; let Gad be the smallest one, and let $crit$ be its (nonfaulty) critical process.
- Eventually, Gad is the "smallest" decision gadget in the simulated trees at all nonfaulty processes.
- So eventually, all nonfaulty processes output the same process id, $crit$, forever.
- Process $crit$ is nonfaulty.
- So this implements Ω_f.

Next time

- Self-stabilization
- Reading:
  - [Dolev, Chapter 2]

Presentation Day

- Friday, December 11, 10AM until done.
- 15 minute presentations
- Lunch!