What's New in Consensus

Bring: Notes for scribe. Chalk

Today's reading:
*** Lamport, the Part-time Parliament
De Prisco, Lynch,...A Formal Approach (skim)
Guerraoui, Revisiting the relationship between non-blocking atomic commit and consensus
Schiper, Early consensus in an asynchronous system with a weak failure detector.
Hurfin, Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector.
Mostefaoui, Raynal. Leader-based consensus.
*** Mostefaoui, Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors.

Next time:
Aguilera, Toueg. A simple bivalency proof that t-resilient consensus requires t+1 rounds.
*** Keidar, Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial.
*** Gafni, Round-by-round fault detectors: Unifying synchrony and asynchrony
Datta, Guerraoui. The inherent price of indulgence (skim)

1 Introduction

2 Lamport, Paxos algorithm

This paper sat unpublished for 10 years because of its weird style. Lamport refused to rewrite it. It was finally published because many people had recognized the importance of the ideas in it and started using and building on them.

Let's translate, and relate to other things we've been studying.

2.1 The Problem

Legislators are processes.
Have uids.
Fixed, known, finite set of processes (except for a few remarks at the end).
All identical. No client/server division.

Processes have approximately-synchronized logical clocks (*simple hourglass timers*).
Monotone, bounded skew (from each other), bounded drift from real time.

Processes communicate by sending point-to-point messages.
Messages aren't changed, but could be duplicated, lost, reordered.
Arbitrary message delay.

Processes can fail by stopping, and later recover. Any number of times.
Have stable storage (*ledger*) that remains safe when the process stops.
Also volatile memory (*slips of paper*), which might or might not be lost in a crash.

Normal behavior for processes (when they aren't failed): Perform their steps in a small, predictable amount of time.
Normal behavior for messages: Delivered within a fairly small, predictable message delay.

No Byzantine failures.
The problem to be solved:
A totally-ordered broadcast problem.
Given a fixed domain V of values (*decrees*).
$\perp$ (*olive day decree*) is a special element, not in V.

Processes have inputs of the form $propose(v)$, $v \in V$.
For simplicity, assume that the same v is only proposed once anywhere.
Output is a consistent (numbered) sequence of values from $V \cup \{\perp\}$.
The outputs needn't actually be produced in order—just output $(index, value)$ pairs, but can reorder. ???
Any value that appears in the sequence should have been proposed by some process.
??? Are repeats allowed?
The special value $\perp$ may be inserted anywhere.

Liveness: If a process proposes a value v and does not fail, and a "sufficient" interval of time occurs subsequently where no one fails or recovers and a majority of the processes are non-failed, then eventually (1) the value gets assigned a position in the sequence, and (2) the value gets output by every non-failed process.

Thus, Paxos is expressed as a totally-ordered broadcast problem.
Closely related problems:

1. Consensus: A key sub-problem, whose solution is used as a building-block for the Paxos algorithm, is a version of the consensus problem.
2. Sequentially-consistent data: A solution to the t-o broadcast problem can be used to maintain a sequentially-consistent data object, using total replication at all the processes.
   The value set V is just the set of operations that can be performed on the object (with operation uids added for disambiguation).

2.2 Single-Decree Synod

A subproblem, for consensus.
In the general algorithm, this is used to decide on the value to be assigned to a particular position in the sequence.
The same algorithm is used to decide on the value in position 1, position 2, etc.

Inputs: *propose*(v) at each i, $v \in V$.
Now assume at most one input occurs at each participating process.
??? Probably not necessary to restrict this.

Outputs: *decide*(v) at each i.

Correctness:

1. Validity: Any value output should have been input by someone.
2. Agreement: No disagreement on the value output.
3. Termination: If anyone proposes an input and doesn't fail, and an interval of time occurs subsequently where no one fails of recovers and a majority are non-failed, then eventually a value gets decided upon and output by every non-failed process.

Presents the protocol in three stages: "Preliminary protocol" is nondeterministic and satisfies the safety conditions (agreement and validity), but not the liveness condition.
"Basic protocol" is an optimization, still satisfying safety but not liveness.
The general synod protocol satisfies liveness too.

2.2.1 Math

Algorithm uses uniquely identified *ballots*, which are not to be confused with the sequence numbers in the totally-ordered broadcast problem.
These are just for a single position in the t-o sequence.
Ballot numbers are chosen from some totally-ordered set T, to be specific, can use (n, i) pairs, natural number with process identifier to break ties.
Each ballot is used for an attempt by a particular coordinator to reach agreement.

Other data associated with a ballot:

- *qrm*, subject to the restriction that every pair of *qrm*s, for any pair of ballots, have a nonempty intersection.
  Q: When is this defined? If pre-defined, it means that the ballot is bound to a particular set of participants, which seems restrictive for a fault-tolerant algorithm.
  At least, it has to be defined by the time *val* is defined, see below.

  Note: Later versions of Paxos have allowed two different quorums to be associated with each ballot—one used for collecting values from earlier (smaller id) ballots, and one for voting.
- *dec*, what I would call *val*, the value in V associated with the ballot.
  This is not defined a priori, but at an intermediate stage of the protocol, as we'll see.
- *vot*, the set of participants known to have voted for the ballot.

The goal is to reach a point where $qrm \subseteq vot$, that is, where every quorum member has voted for the ballot.

B3 expresses a key condition preserved by the algorithm. It is expressible in the form of an invariant.

"If $B.val$ is defined, and if any member of $B.qrm$ has voted for any earlier (smaller id) ballot, then $B.val$ is equal to the value associated with the latest of these earlier ballots."

Go over example, p. 6 top. Look at the value for ballot 29.

I'm not sure it helps, but condition B3 is written more formally on p. 7.
I think this rewrite neglects to assume that $B.val$ is defined, though.

This condition isn't an algorithm—it's a property that must be shown for an algorithm.
It's the key to showing agreement, as indicated by the Lemma, p. 7, which expresses another invariant:
Lemma: (In any reachable state) If $B < B'$, $B.val$ and $B'.val$ are both defined, and $B.qrm \subseteq B.vot$ (that is, the earlier ballot had approval from enough voters), then $B'.val = B.val$.

The proof is done carefully, but the intuition isn't easy to see. The idea is simple, though:
Proceed by contradiction, look at any B satisfying $B.qrm \subseteq B.vot$, and consider the earliest $B' > B$ with a different value from B.

$B'.qrm$ intersects $B.qrm$, hence $B'.qrm$ intersects $B.vot$, that is, some process in the quorum for B' has also voted for B.
Then by B3, $B'.val$ is equal to the value of some ballot that is $\geq B$.
But by choice of B', it's the smallest one $> B$ with a value $\neq B.val$.
So that means that $B'.val = B.val$, which is a contradiction.

2.2.2 Preliminary protocol

A distributed algorithm.
Each ballot is initiated by a process, which is the coordinator of the ballot.
Coordinator chooses ballot number, e.g., as (n, i) in increasing order locally.
Also chooses a specific $B.qrm$, right up front (a bit odd).

The key is how to associate a value with a ballot.
Must guarantee B3: "... if any member of $B.qrm$ has voted for any earlier (smaller id) ballot, then $B.val$ is equal to the value associated with the latest of these earlier ballots."

Suggests that the coordinator should collect information about the earlier ballots that everyone in the quorum voted for.
Top p. 10 describes a 2-way interchange by which the coordinator requests this information and gets it.
(This information must be kept stably by the voters.)

But this isn't enough: A race condition arises.
Namely, when a participant sends the coordinator its info about its votes for smaller ballots, that info is correct at the time it sends it. But what makes it correct later?
*** It needs to promise never in the future to vote for any ballots in between B and the latest smaller ballot that it tells the coordinator about.

After hearing from the quorum, the coordinator is finally ready to assign a value: one that satisfies B3.
Then needs to get voters to vote for it. Does this with a second 2-way exchange of messages, sending the ballot value to the quorum and collecting their votes back.
When (if) they all vote for the ballot, the ballot is successful, and the coordinator releases the decision (to its own client, and to the other processes).

The members of the quorum will all vote for the ballot if they can, that is, if this doesn't violate any promise that they have already made because of a higher-numbered ballot.

Any participant can initiate a ballot at any time.
That means that ballots can continually interfere with one another, so progress isn't guaranteed.

2.2.3 Basic protocol

This is a simple optimization of the preliminary protocol, which minimizes the information that needs to be kept and send around.
E.g., don't need to remember all the ballots you've ever voted for—just the largest one.

Also restricts the behavior in certain simple ways, in order to maintain less information.
E.g., restricts so a single process can only be coordinating at most one ballot at a time.
E.g., restricts voter so that it promises never to vote for any ballot less than B, not just any between B and the largest smaller one it already voted for.

2.2.4 Full synod protocol

This restricts some more in order to guarantee progress.

Requires participants to perform enabled steps as soon as possible.
Also, resend messages periodically (?)

Key question is who should start ballots and when.
Two issues: Who and when.

Who: Relies on a separate leader-election service.
Condition required is a stability condition (p. 14): If failures and recoveries stop from some point on, then within a bounded amount of time, exactly one (non-failed) participant would consider itself a leader.
Also assumes that no messages are lost. That assumption is hidden in the statement that "no one enters or leaves the chamber"—includes messengers.

How to ensure this? Can implement their own failure detector: Send periodic "I'm alive" messages. Then chose the process with the largest identifier among the processes that are alive.

In terms of Chandra-Toueg's failure detector classification, this is hypothesizing behavior that, at least in a particular run under consideration, is like diamond-P.

But this might not be the best way to look at this.

When: Same coordinator initiates a new ballot when it suspects that its previous ballot might not have succeeded.
That's determined by a timeout based on how long the ballot would take in the absence of errors.

Bottom p. 14: Lamport alludes to the need for an "optimal" leader election strategy. What does this mean? Minimum latency after things stabilize? Is there a good known protocol?
What's so great about this protocol? We know by FLP that we can't get agreement, validity, and termination in the presence of even very simple faults, in the asynchronous model. That extends (see Dolev, Dwork, Stockmeyer) to synchronous model with unknown message delays.
This protocol tolerates a huge variety of practical kinds of failures without every violating agreement or validity.
Termination is guaranteed in good situations—if the algorithm every stabilizes from some point onward.

This description so far is satisfied by Dwork, Lynch, Stockmeyer.
But that paper used the rotating coordinator paradigm, whereas this paper allows concurrent coordinators and depends on a separate leader elector to obtain (eventually) a unique coordinator.

2.3 Multi-Decree Parliament

Now, back to the problem of totally-ordered broadcast.
The Synod protocol just solves ordinary consensus.
The basic idea is to carry out agreement on what is the first message, the second, etc.

2.3.1 The protocol

Elect a single leader, which should take charge of deciding on the messages to be assigned to all indices in the message sequence.

Leader coordinates a logically-separate Synod execution for each message index k.

Optimizations:
Try to combine the mechanisms used for the different consensus executions, for different message indices, as much as possible.
When someone becomes the leader, it attempts to take charge of consensus on all the indices (among those for which it doesn't already know a decision).
Choose a single large-enough ballot number b to cover all these executions.
Then it starts ballots with this number b for all (infinitely many) consensus executions at once.
It can do this by sending a single first-round message, which encodes the first round message for that ballot number b for all the consensus executions.
(This message doesn't contain the actual value to be assigned to that ballot for each consensus execution.)
The responses from the other processes to this request describe their last votes for that ballot number in infinitely many protocol executions.
But, they won't have actually participated in infinitely many protocol executions, so they don't

really have to send an infinite amound of information.

After this exchange, the leader knows enough, and has extracted enough promises, to be able to go ahead and assign values to ballot b for all the consensus executions.
For each such consensus execution, it will only need to carry out the second phase and the dissemination of decisions.

How does it know what values to assign to ballot b in all executions?

1. For ballot b for some of the consensus executions, it has learned from the other processes about some previous votes that determine what value (message) must be assigned to ballot b.

2. For ballot b of the other consensus executions, it can just assign arbitrary messages that haven't already been assigned anywhere.
   To manage this, every process whose client sends a message just sends it to the leader.
   (Since the leader may change, it presumably has to keep track of this and possibly resend it to new leaders. How to manage this?)

   The leader assigns messages to ballot b in the undetermined cases, choosing from among the set of messages it knows about that aren't already assigned to ballot b anywhere.
   Assigns messages in the order received, to the lowest indices for which it's free to choose.

After assigning values, it continues with phase 2 to try to get enough processes to vote for the various messages.

If ballot b fails, the leader can try a new ballot number as usual. Again, it can do the first phase concurrently for all the consensus executions.

Modification:
Leader attaches to the "generic" first round message the largest index n such that it already knows about decisions for all indices up through n.
Then the responders don't need to send back information for indices up through n. Saves communication.
The responders send information they have about their votes, etc., for any index $> n$.
Not just the usual information from the Synod protocol—the largest vote with a smaller ballot number.
Also sends any information it has about indices $> n$.

Gaps: A gap can arise, e.g., if a new leader hears about a value for decree with index k but not index $k-1$. In this case, it should not associate new messages with index $k-1$, but should leave a gap and associate them with larger indices.
The gap gets filled with the special $\perp$ message (olive-day decree).

Why? Can violate causality from the client's viewpoint? It's possible that someone knows about the decision for k and used that to decide to send a message. If that message got assigned to $k-1$ it would be inconsistent with the client's causality relation. ???

2.3.2 Properties of the protocol

Agreement, validity of the Synod sub-protocols imply that everyone gets a consistent assignment of messages to indices.
Termination for Synod implies corresponding (stable case) termination for each message index.

That doesn't say everything we might want.
E.g., does every requested message eventually get assigned to some index? In the stable case, it seems that it does. The stable leader keeps track of all requested messages, and assigns them in order of when the clients have submitted them, to the next available ballots. If there's only one leader, these will succeed.

Can the same message get assigned to two different indices?
Apparently not by the same leader.
But what about by two different leaders?

Ordering property: p. 16 near bottom: I'm restating this from the client's view point, a little weaker:
If a non-$\perp$ message m' is submitted by a client after anyone learns about the index of another non-$\perp$ message m, then any index ever associated with m' has to be larger than the smallest (?) index associated with m.

Why: Before a leader assigns m' to a ballot in a consensus instance, it hears from a quorum about everything they have already voted for.
If anyone has already output an index for m, then a quorum has already voted for this association.
Some process q is in the intersection of these two quorums.
It tells the leader about m.
Then the leader would not try to associate m' with any smaller index.

???
A race condition here? Phase 1, which does the collection, could have happened a long time ago...
Assuming that no one else is a leader and could be causing m to complete while all this is going on. ???

Anyway, what is this restriction needed for?

Another ordering property, looks related, top of p. 19:
"If two messages have indices determined in ballots conducted by different leaders, then one associated its message with its ballot after learning that the other message had been associated with its ballot."

This time, the argument depends on ballot numbers.
Say the two messages are m and m', and their successful ballots are b and b', respectively.
Assume wlog that $b < b'$.
The leader of the ballot for m' must learn about a vote (by some q) for the smaller ballot number b, regardless of the index of the consensus execution.
This seems to follow because phase 1 handles all the consensus executions together.
Process q sends the leader of ballot b' the information about ballot b.

Does this imply that the same message can't get assigned to two different indices in the sequence?

2.3.3 Further developments

Leader election: Brief discussion. I haven't thought through the issues and not much is said here.

Long ledgers:
Discusses condensing the long sequences that are maintained into state + incremental sequence.
This means that he is thinking of the messages in the sequence as operations on a data object.
Given a sequence of operations and a state of the data object, can apply the sequence to get a new state.
The basic idea behind replicated data management. The consistent sequence keeps the data objects consistent.
Here, he notes that the processes can apply some of the operations, summarizing them in an object state, and just keep the recent ones as a sequence.

Bureaucrats
Learning the law
Dishonest legislators, etc.: Briefly discusses other kinds of failures.

Choosing new legislators: This deals with a dynamic version of the algorithm, where a decree (operation) is used to change the *configuration*, which we should think of as the set of members plus sets of quorums.

This is like thinking of the configuration as just a part of the state of the underlying object, and the decree that changes the configuration as an operation that affects that part of the object state.

But this algorithm allows consensus protocols for different indices in the message sequence to be going on at the same time.
That causes a possible problem: the membership could change by a decree with index k, while consensus executions for indices $> k$ could already be going on using the *previous* configuration.

The strategy Lamport proposes is to specify that the configuration used for consensus instance k is always the one that results from applying the decrees up through index $k - 3$.
Anyone initiating configuration k has to wait to hear about all the decrees up through $k - 3$.
Then there's no ambiguity.

What exactly does this guarantee?
Why 3?
Do we need any intersection conditions for successive configurations?

Finally, mentions the dead sailors problem, whereby the new configuration is chosen badly—to be failed processes, e.g.

3 De Prisco,... formal approach to Paxos

This is a note that some of us wrote in order to pin down the consensus part of the Paxos algorithm (synod algorithm) more precisely than what is given in the paper.
Both safety and timing aspects.

For the safety, we proceeded in levels of abstraction, producing:

1. A formal specification of the consensus problem as an abstract state machine.
2. A high-level global automaton, *Global1*, which actually contains most of the interesting ideas of the algorithm.
3. An optimized global specification *Global2*, which looks more like Lamport's protocol.
4. A fully detailed distributed algorithm.

Each level implements the next higher in a formal sense, using a forward simulation relation and auxiliary invariants—the main proof technique that works for algorithms like this one.
Toh Ne Win and Nicole Immorlica even pushed the complete proofs through an interactive theorem-prover.

We then refined the distributed algorithm further by adding in the timing constraints.
This allows us to prove time bounds for termination, after things stabilize.
But this part is only sketched out, and not nearly formal enough to push through a theorem-prover.

Novel algorithmic aspects:

1. We don't use majority sets, or general quorums. Rather, we use read and write quorums—read-quorums in the first phase, to determine the latest ballots for which the participants voted, and write-quorums in the second phase, to make sure enough processes have voted for the ballot. The intersection required is that every read-quorum must intersect every write-quorum.
   This follows an observation of De Prisco in his MS thesis.
2. The algorithm descriptions at all levels make explicit the promises that participants make to not vote for a ballot. We say then *abstain* from the ballot.
   These promises take the form of abstaining from entire ranges of ballot identifiers—not just for individual ballots.
   That should make sense—the promises are essentially not to vote for ballots with identifiers smaller than one that is now being attempted.
3. The algorithm descriptions use the abstract notion of a ballot being *dead*, which is essentially due to Butler Lampson.
   It says that some entire read-quorum has abstained from it.
   This is interesting because it means that never in the future can an entire write-quorum vote for it, which is what would be needed for the ballot to succeed.
4. The distributed version of the algorithm lets communication occur in the background, until certain fixed-point conditions hold.

Look at *Global1*, which actually contains the main ideas.
At this level, we don't bother to say who is starting a ballot, who assigns the value, and who determines that a decision can happen.
We do identify the location of initial submissions of values, of announcements of decisions, and we identify who is voting and abstaining.

Basically, a new ballot can be created at any time.
Someone can abstain from a set of ballots provided he hasn't already voted for any of these ballots.
And someone can vote for a ballot provided he hasn't abstained from it.

Somewhat more interestingly, the value v can be assigned to ballot numbered b provided that the ballot doesn't already have a value, v is a proposed value, and—the interesting condition:
For all smaller ballot identifiers b' (not just those of ballots that have actually been created), either there is a ballot that has already been created with that ballot identifier and it has the same value v, or else the ballot is dead, as defined earlier.
Thus, the value has to agree with the values of *all* earlier ballots that might ever succeed.

A ballot can succeed if a write-quorum has voted for it.

Now, this doesn't look exactly like (an abstract version of) Lamport's algorithm: Lamport doesn't actually check all previous ballots.
To capture Lamport's test more accurately (but still with a global description), we developed the next level, *Global2*.
Global2 differs from *Global1* only by the condition under which it assigns a value to a ballot: now it just checks that all the ballots are dead up to some earlier ballot with the same value v. It doesn't check anything earlier than that latest ballot.

It's very easy to show that this test implements the heavier-duty one in *Global1*.
The idea is to use induction on the ballot numbers.

Finally, we give the distributed version.
It's extremely similar to the global version.
Basically, processes use local versions of the global conditions.
E.g., a local version of *dead* says that a read-quorum is *known locally* to have abstained.
(Local knowledge implies the actual condition holds.)

The mapping proof is straightforward.

How long does this take to terminate, after the system stabilizes?
We have only a sketch so far, but it's surprising long, around $10d$ (or maybe it's $12d$), where d is the normal case message delay.

This is because it takes a little while—around $2d$, as we wrote it—for everyone to obtain correct information about who has failed (some time until gossiping, which we called d, plus a message delay).
After that, only the real leader ever starts a ballot.
Then it takes some additional time for everyone to learn about all ballots that were ever started by anyone other than the final leader.

Another $2d$. (?)
After that, any new ballot that's started is guaranteed to complete.

But it might be another $2d$ before the current ballot, which might never finish, gets identified and a new ballot restarted.
(Q: Maybe this needs to be $4d$: if the old ballot has too small an identifier, perhaps it will look OK after the first phase and we will let it proceed to the second phase. But maybe the second phase won't complete, because the processes that we need to vote for it have already abstained from it. ??? Needs to be worked out more carefully.)

Then another $4d$ for the algorithm to complete (everywhere).
(We don't require the leader to tell everyone that the ballot has succeeded—they can make a decision locally, based on the same info that the leader has.)

I also skimmed the paper by Boichat, Dutta, et al. on "Deconstructing Paxos". It presents Paxos in a modular style, using a *round-based register* abstraction.
This is similar to the more recent paper by Chockler and Malkhi that uses a *ranked register*.
Since the latter abstraction seems to be claimed to supersede the earlier one, I didn't think we needed to go over this one in detail.

4 Atomic commit and consensus

Guerraoui. Revisiting the relationship betwen non-blocking atomic commitment and consensus. WDAG (precursor to DISC), 1995.

Switching gears a bit...
This is the first of five little papers I read, from researchers in Lausanne and Paris, all involving aspects of consensus with failure detectors.
The papers have a lot in common, so I'll try to summarize the highlights.
I chose one—by Mostefaoui and Raynal—as a sample for you to read, so will spend more time on that one.

Non-blocking atomic commitment is another consensus problem, a little different from the usual consensus problem.
Since Chandra and Toueg showed that ordinary consensus is solvable using unreliable failure detector diamond-S, Guerraoui considers whether NBAC is also solvable using diamond-S.
He discovers that it isn't (in the presence of a single crash failure).

So, he defines a weaker version of the problem, which is a little unusual in that it is defined relative to the failure detector.
He shows that this *can* be solved with diamond-S.

As a sidelight, he shows that any algorithm that solves consensus with diamond-S also solves "uniform consensus" with diamond-S.
Uniform means that even the failed processes have to agree.
(I think that's the more interesting version anyway—in my book, I consider only uniform consensus problems.)

It's pretty obvious that, in the presence of diamond-S, this equivalence holds: Suppose the algorithm ever allows any two processes, failed or not, to decide differently.
The processes have no way of detecting that a process has actually failed—the only information they get is from diamond-S, which can be unreliable in any finite prefix!
So we can modify the "conflicting" execution to one in which neither process is actually failed, which violates the weaker nonuniform agreement property.

4.1 Model

Processes: Asynchronous, fail by crashing.
Channels: Reliable, eventual delivery, no time bound.

Processes also have access to a diamond-S failure detector. Recall diamond-S:
Strong completeness: After some particular time, every failed process is permanently suspected by everyone.
Eventual weak accuracy: After some particular time, some correct process is never suspected by any correct process.

4.2 Impossibility of solving NBAC using diamond-S

NBAC: Participants propose 0 or 1.
Agreement: No two participants (faulty or not) decide differently.
Termination: Every correct participant eventually decides.
Validity: Two conditions:
1. If decision is 1 then all inputs are 1.
2. If all inputs are 1 and no processes fail then the only possible decision is 1.

Now, the possibility of solving this with diamond-S (or S, or diamond-P) seems problematic, because condition 2 is based on actual failure, whereas the failure detectors can make mistakes.
Thus, it's impossible to detect actual failures reliably.

The impossibility proof proceeds by contradiction. Suppose an algorithm exists.
Consider an execution R in which eveyrone starts with 1, process p_1 fails at the beginning, and all other processes are nonfaulty.
Process p_2 must eventually decide.
It can't decide 1, because then it would do the same in the case where p_1's initial value is 0, violating validity condition 1.
It can't decide 0 because it would do the same in an execution where p_1 didn't fail, but its messages are delayed past the decision point. Also, the unreliable failure detector behaves just as it does in R—possibly saying that p_1 has failed—this can happen because it's allowed to be unreliable for a finite time.

4.3 Weak NBAC and how to solve it using diamond-S

Guerraoui's solution is to weaken property 2 so that it mentions the failure detector output rather than the actual failures.

That is, replace 2 with:
2'. If all inputs are 1 and no participant is ever suspected, then the only possible decision value is 1.

This problem (perhaps not surprisingly) is solvable using diamond-S.
Guerraoui shows this by using an ordinary uniform consensus sub-service.
Previous results of CT, as described by Tina, show that ordinary uniform consensus is solvable using diamond-S.
To turn this into a weak NBAC algorithm, all he does is attach an extra message exchange round at the beginning:

Everyone sends their values to everyone, including themselves.
Wait until, for every process j, you've either received a value or else suspected j.
If receive 1 from everyone, start uniform consensus subroutine with input 1.
If receive any 0 or suspect anyone, then start uniform consensus with 0.
In either case, return the value returned by uniform consensus.

5 Consensus Using Failure Detectors

Now, four papers that deal with solving the basic uniform consensus problem, three using diamond-S and one a leader-oriented fd:

1. Schiper. Early consensus in an asynchronous system with a weak failure detector. Distributed Computing 1996.
2. Hurfin, Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. Distributed Computing 1999.
3. Mostefaoui, Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. DISC 99.
4. Mostefaoui, Raynal. Leader-based consensus. IRISA TR, 2000.

Focus on the third of these, just summarize the results in the others.

5.1 Schiper

Same model as Guerraoui: Asynchronous system, processes fail by stopping, reliable channels. Diamond-S.

Schiper gives a uniform consensus algorithm similar to that of Chandra and Toueg, using diamond-S.
Rotating coordinator. Coordinator engages in a Paxos-like 2-phase protocol to try to collect enough votes.

What he is interested in here is the best-case behavior: the time it takes to decide in cases where there are no failures and the FD never suspects anyone!
The intuition is that this is the "normal case", so should be optimized.
They don't analyze the behavior of their algorithm in other situations.

He defines a new "latency measure" to measure the number of message delays. Basically, look at a Lamport-style graph representing a computation, and assign an integer to each point based on the maximum length of a path of messages that precede that point.

I would prefer to think of the more standard kind of time measure, where we put an upper bound of d on each message and 0 time for local processes, and measure the latest time that can be assigned to each point.
I think we get similar results in that case.

Recall the CT algorithm. Uses asynchronous rounds, rotating coordinator, $r \bmod n$. Everyone maintains an "estimate" est.
Each round has four steps:

1. Every process sends its latest est to the coordinator. Coord waits to receive est from a majority, and determines its new est based on seeing which of the incoming ests has been produced in the latest preceding round.
2. Coord bcasts its new est. Each process waits to receive the new estimate from the coordinator or to suspect the coordinator. If it hears from the coordinator, it adopts the estimate received, else no change.
3. Each process that heard from the coordinator sends back an ack; others send nack. Coordinator waits to hear (ack or nack) from majority. If ack from majority, decides on the current estimate.
4. If coordinator decided, bcast decision.

Latency measure yields 4 for the best case.
Can be reduced to 3 by a trivial optimization: In the first round, omit step 1. The coordinator can simply use its own initial value as the first proposal, in step 2.

Schiper's new algorithm has latency 2 in the best case.
Again, he uses the rotating coordinator strategy like CT (with the first-round optimization described above).
The description is a little complicated, but it seems to be basically very similar to that of CT, except that it makes greater use of all-to-all communication instead of always communicating back-and-forth with the coordinator.
That turns out to save another round, in the non-failing, non-suspecting case.

The essence of this work seems to be the 2-round termination in the non-failing case.
Will see the same idea, in a simpler algorithm, in the work of Mostefaoui and Raynal.

5.2 Hurfin, Raynal

5.3 Mostefaoui, Raynal, consensus using diamond-S

They give an algorithm for the uniform consensus problem using diamond-S.
Also give a version that uses S and requires fewer processes to tolerate the same number of failures.
They feature a "generic" presentation, wherein they present the two versions together, in the same

code.
This highlights the similarity.
The algorithm is still nice and simple.

In the non-failing, non-suspecting case, with diamond-S, this yields 2 rounds, as does Schiper. But this one seems to be much simpler.

5.3.1 Model and Background

Processes: Asynchronous, fail by stopping.
Channels: Reliable, eventual delivery.
Access to diamond-S (or S).

Consensus problem: The usual. Uniform version.

Review results, top of p. 4:
1. CT, algorithm for consensus using S, any number of faults.
2. CT, algorithm for consensus using diamond-S, tolerates less than half faulty. Also show that's optimal—need that many processes.
3. diamond-S is, in a sense, the weakest FD that allows solving consensus.
4. Guerraoui, Any algorithm that solves consensus with an unreliable FD also solves uniform consensus.

5.3.2 The algorithm

Describe the diamond-S version.
Same general style as CT, Schiper, Hurfin-Raynal.
Each process keeps local variable est, set initially to its initial value.
Asynchronous rounds, rotating coordinator.

Round r consists of two steps:

1. Current coordinator sends its current est to everyone.
   Everyone waits to either receive an est from coordinator or suspect the coordinator. They record either the received value or $\perp$ (in case the coordinator is suspected), in est-from-c.
2. Everyone sends their est-from-c value to everyone.
   Wait to receive est-from-c values from a majority of processes.
   Let rec be the set of values received in this step, from everyone in the majority. Three possibilities:
   (a) $\{v\}$ for some particular v (the coordinator's actual est at the beinning of this round).
   In this case, decide on v.
   (b) $\{v, \perp\}$.
   Then don't decide, but change your est to v.
   (c) $\{\perp\}$.
   Don't decide, keep your previous est.

See code, p. 6.

This differs from CT, Paxos, in that the boundary of the rounds has shifted: The new estimate is computed in the last step of each round, not the first.
So we don't have to "optimize" by eliminating the first step of the first round—that is eliminated automatically.

Correctness: Validity is obvious. Termination isn't hard, assuming that a majority stays alive (and the FD is complete).

Uniform agreement: First note that the only actual V values that get sent around during the entire round are the coordinator's actual est at the beginning. Therefor, if two processes decide at the same round, they must decide on the same value.

Now suppose that r is the first round at which someone decides, and say it's process i and it decides on v.
Claim that processes that decide at later rounds will also decide on v.
This will follow if we can show that v is the only possible est that anyone can have at the beginning of round $r + 1$.

So, let j be any other process.
Both i and j see majorities at step 2 of round r.
These majorities intersect, say in process k.
Since k has send i the value v (that's all i has seen), k has also sent v to j.
Therefore, the set seen by j is either $\{v\}$ or $\{v, \perp\}$.
In the first case, j would also decide at this round and we are done.
In the second case, j changes its estimate to v and moves on to round $r + 1$, as needed.

The version of othe protocol using S instead of diamond-S is almost the same. The only difference is in the definition of the quorums for which the processes wait in the second step. For diamond-S, we used majorities. For S, they use the set of all processes that aren't suspected. That works for any number of failures.
The reason this works with S is that some process is *never* suspected. Therefore, it is in the intersection of all the quorums. Then the same argument holds as above.

5.3.3 Analysis

In the absence of failures and suspicions, this takes time at most $2d$ (one round, that is, both steps).

If there are failures and/or false suspicions, then consider the time from when failures and false suspicions stop (GST), until every non-failed process decides.
Let f be upper bound on number of failures that actually occur during the particular execution.
Up to f rounds can be "wasted" until a correct process is the coordinator. The last round takes 2 steps, which is time at most 2.
Claim that the earlier rounds take only time 1. That's because the coordinator of each of the unsuccessful rounds is suspected at the beginning, which means that the processes aren't going to wait to hear from it in step 1. So, they go right to step 2 and only time 1 is used.

Total is $(f + 2)d$.

(Actually, may have to add something to finish a round in progress when GST occurs. Another $2d$?)

5.4 Mostefaoui, Raynal, consensus using a leader

In the algorithm of the previous paper, the time to finish after GST can be as bad as $(f + 2)d$, where f is the number of actual failures.
Recall that $f + 1$ total rounds are inherent for consensus (and, as we will see next time, $f + 2$ are inherent for uniform consensus with early stopping).
But, note that the f failures have all already occurred by GST.
It's not clear that we should have to incur the f time delays after that point.

This seems to be an inherent feature of rotating coordinator algorithms.

In this new paper, MR replace the rotating coordinator strategy with a leader election strategy.
Like Paxos.

Same computation model, asynchronous processes, crash failures, reliable channels.
Uniform consensus problem.

Leader oracle: Keeps reporting a leader, subject to: There is a point in the execution and a correct process p such that, after that point, every correct process has leader $= p$.

Can implement this from diamond-S. In fact, CT show the two are "equivalent".
Assumes majority non-failed.

Asynchronous rounds as before, now with three steps.

1. Everyone sends everyone their est. Then waits until it receives an est message for this round from the process it considers to be the leader. (This may change while the process is waiting.) At that point, resets its own est to the value received.
2. Bcast this new est. Wait until step 2 messages received from all but f processes. Set a new variable aux: if a majority of the processes agree on some v, then aux $= v$, else $\perp$.
3. Bcast aux. Wait until step 3 messages received from all but f processes. If any of these messages has a value v then reset your est to v, else unchanged. If $f + 1$ of these messages are non-$\perp$, then decide on your latest est.

Familiar style...proofs LTTR.

Time in case where no failures occur and a single correct leader exists from the beginning: 3.
Time in executions in which, from some point on (GST), no failures occur and there is a single correct leader:
Count time ($3d$?) to finish a round in progress. Then done in another time $3d$.