Massachusetts Institute of Technology
6.895: Advanced Distributed Algorithms
Professor Nancy Lynch

Lecture 12
October 17, 2002

More Consensus

Bring: Notes for scribe. Chalk

Today's reading:
Some things left over from last time:
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.

New things:
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

Next: Data replication and data consistency:
*** Attiya, Welch
Ladin, Liskov, Shrira, Ghemawat (skim)
*** Fekete, Gupta, Luchangco, Lynch, Shvartsman

1 Consensus Using Failure Detectors

Five little papers about solving consensus using failure detectors.
Have a lot in common. I'll mostly summarize, but will spend a bit more time on the first one by Mostefaoui and Raynal, which I asked you to read.

1.1 Atomic commit and consensus

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

1.1.1 Overview

Non-blocking atomic commit is another agreement problem, slightly different from the usual consensus problem.
Chandra and Toueg showed that ordinary consensus is solvable using the 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 NBAC, 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.

1.1.2 Model

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

Processes also have access to diamond-S failure detector:
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.

1.1.3 Uniform vs. nonuniform consensus

Assume an algorithm A solves ordinary basic consensus using diamond-S.
Suppose for the sake of contradiction that it doesn't solve uniform consensus.
Then in some execution, two processes decide differently (one must be failed).
But the processes have no way of detecting that any process has actually failed; the only information they get is from diamond-S, which may 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.

1.1.4 NBAC problem

NBAC: Participants propose 0 or 1.
Agreement: No two participants (faulty or not) decide differently.
Termination: Every correct participant eventually decides.
Validity:
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.

1.1.5 Impossibility of solving NBAC using diamond-S

Solving NBAC 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 everyone starts with 1, process 1 fails at the beginning, and all other processes are nonfaulty.
Process 2 must eventually decide.
But it can't decide 1, because then it would do the same in the case where process 1's initial value is 0, violating validity condition 1.
And it can't decide 0, because it would do the same in an execution where:
1 doesn't fail, but its messages are delayed past the decision point, and
diamond-S behaves just as it does in R—possibly saying that 1 has failed—this can happen because it can be unreliable for a finite time.

1.1.6 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, replaces 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 add 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.

1.2 Schiper

The remaining four papers 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. Skip.
3. Mostefaoui, Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. DISC 99. Focus on this one.
4. Mostefaoui, Raynal. Leader-based consensus. IRISA TR, 2000.

Schiper uses the same model as Guerraoui: Asynchronous system, processes fail by stopping, reliable channels. Diamond-S.
Gives a uniform consensus algorithm similar to that of Chandra and Toueg, using diamond-S.

Rotating coordinator. Coordinator conducts Paxos-like 2-phase protocol to try to collect enough votes.

He focuses on the best-case behavior: the time to decide in cases where there are no failures and the FD never suspects anyone!
He thinks of this as the "normal case", which should be optimized.

He defines a new "latency measure" to measure the number of message delays.
Namely, for a given computation, define a Lamport-style graph of events and dependencies.
Assign an integer to each event based on the maximum length of a path of messages that lead to that point.

I prefer the more standard time measure as in my book, where we assume an upper bound of d on each message delivery and 0 time for local steps, and measure the latest time that can be assigned to each event.
I think the results come out the same with this measure.

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. Coordinator waits to receive *est* from a majority, then determines its new *est* based on seeing which of the incoming *est*s has been produced in the latest preceding round.

2. Coordinator bcasts its new *est*. Each process waits to receive the new *est* or to suspect the coordinator. If it hears from the coordinator, it adopts the *est* received as its own *est*, else it makes 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 *est*.

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 rotating coordinators.
The description is a little complicated, but it seems to be basically similar to CT, with the first-round optimization as above.
However, Schiper uses all-to-all communication in some situations instead of always communicating back-and-forth with the coordinator.
That saves another round, in the non-failing, non-suspecting case.

1.3 Hurfin, Raynal

1.4 Mostefaoui, Raynal, consensus using diamond-S

Another algorithm for uniform consensus 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.
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 much simpler.

1.4.1 Model and Background

Same model again: Asynchronous system, processes fail by stopping, reliable channels (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.

1.4.2 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 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 $est - from - c$ values 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 easy, assuming majority stays alive (and the FD is complete).

Uniform agreement: First note that the only actual V value that gets sent around during the entire round is the coordinator's actual *est* at the beginning. Therefore, 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 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 sent i the value v (that's all i has seen), k has also sent v to j.
Therefore, the *rec* 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 *est* to v and moves on to round $r + 1$, as needed.

The version of the 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.

1.4.3 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 $2d$?)

1.5 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.
But this cost 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:
Time $3d$ (?) to finish a round in progress. Then done in another time $3d$.

2 Aguilera, Toueg

Now we move to several results dealing with optimal consensus algorithms, with corresponding lower bounds and impossibility results.
Some new results, and some new ways of obtaining old results.
Especially interesting are some newly-observed connections between synchronous and asynchronous models.

2.1 Overview

A really neat 4-page paper!
It gives a very simple proof for the known lower bound of $f+1$ rounds for f stopping failures in the (round-based) synchronous model, by using the technique of FLP for impossibility in the asynchronous case.

Historically, the lower bound on rounds was proved first.
It was proved by constructing a "chain" of executions, where consecutive exections in the chain look the same to some nonfaulty process.
The chain can be constructed to span from a failure-free execution where everyone starts with 0 to a failure-free execution where everyone starts with 1.
Agreement and validity lead to a contradiction.

The first such proof was by Lynch, Fischer, for the Byzantine case.
Around 1979 or 1980.
A more complicated chain construction evolved for stopping failures, not attributed to anyone in particular. Dolev, Strong? Merritt?
A version in my book.
The construction is a little complicated.

The impossibility for consensus in the asynchronous model was proved a little later—the result was actually obtained in 1982.
It introduced a new method based on "bivalence" and "univalence".
Until recently, no one seems to have tried out the new method on the former problem.
It works much better than the old method.

The new proof seems to have been found not just by AT, but also by Moses and Rajsbaum. A similar argument was also used by Bar-Joseph (a student here, before he was here) and Ben-Or, but they applied it to the setting of randomized algorithms.

2.2 Model and problem

Synchronous round model. Known set of processes. In each round, each process sends messages to all other processes, receives messages, changes state.

Stopping failures. When a process stops, it may send any subset of its messages for the round in which it fails.
Furthermore, at most one process fails per round.

(Makes the impossibility result stronger.)

Consensus: Agreement, validity (decision value must be the initial value of some process), termination.
The uniformity of agreement isn't needed.

2.3 The proof

Assumes that $n > f + 1$.

By contradiction. Assume there is such an algorithm A, tolerating f failures and always terminating in exactly f rounds.

Restrict attention to executions with at most f failures.
Define a execution that completes some finite number of rounds to be 0-valent (1-valent) if the only decision that can follow from extensions of the execution is 0 (1).

Univalent: 0-valent or 1-valent.
Bivalent: Both decisions are possible.

Lemma 1: There is some initial state (0-round execution) that is bivalent.

Proof: As in FLP. Assume all are univalent, note that C^0, where everyone starts with 0, is 0-valent and C^1 is 1-valent.
Then there must be two initial configurations with Hamming distance 1 (differ in state of one process) such that one is 0-valent and one is 1-valent.
But this can't be, because if that process fails, no one else can distinguish the two configs.

Lemma 2: There is a bivalent k-round execution for every k, $0 \leq k \leq f - 1$.

Proof: By induction. The base, $k = 0$, follows from lemma 1.
Inductive step: Assume true for k: we have a bivalent k-round bivalent execution r.
Show for $k + 1$, where $0 \leq k \leq f - 2$.
Assume not—then every one-round extension of r is univalent.

Let r^* be the one-round extension of r in which no new failures occur in round $k + 1$.
This is univalent by assumption, wlog 1-valent.
Since r is bivalent, there must be another 1-round extension of r, call it r^0, that is 0-valent.
In r^0, some single process p fails in round $k + 1$, and it fails by sending to some subset of the processes.
Let $q_1, \ldots, q_m$ be the processes to which p fails to send a message in round $k + 1$.

Now define a little chain of runs $r^1, r^2, \ldots, r^m$.
Each r^i is the same as r^0 except that p sends a message to $q_1, \ldots, q_i$ in addition to the others that it sends to in r^0.
That is, this chain represents adding in one message from p at a time.
In all these executions, p still fails at round $k + 1$; however, in the case of r^m, it fails (think of a special "fail" event occurring) after sending all its messages.

Each r^i, $0 \le i \le m$, is univalent, by assumption.
Since r^0 is assumed to be 0-valent, there are two possibilities:

1. At least one is 1-valent.
   Then there must some j such that r^{j-1} is 0-valent and r^j is 1-valent.
   But these two differ after round $k+1$ only in the state of process q_j.
   This can't be possible, because we can extend both executions by simply failing q_j at the beginning of round $k+2$ (recall we have $k+2 \le f$), and no one who is left could tell the difference.

2. All of them are 0-valent.
   Then compare r^m, in which p sends all its messages at round $k+1$ and then fails, with r^*, in which p sends all its messages at round $k+1$ and does not fail.
   r^m is 0-valent and r^* is 1-valent.
   But we can fail p at the beginning of round $k+2$ in r^*, and then no one else can tell the difference, again a contradiction.

So, we can keep the execution bivalent through $f-1$ rounds, while expending at most $f-1$ failures.

Now, what about the final round? The argument is essentially the same as the inductive argument. But instead of arguing that there is a bivalent f-round execution (which doesn't exactly make sense, since we've required that everyone decide by the end of f rounds), we argue that there is an f-round execution in which two nonfaulty processes decide differently:

Lemma 3: There is an f-round execution in which two nonfaulty processes decide differently (which is a contradiction to the problem definition).

Proof: Use Lemma 2 to get a bivalent $f-1$-round execution r (with at most $f-1$ failures).
We know that in every one-round extension of r, everyone who hasn't failed decides (and agrees).

Let r^* be the one-round extension of r in which no new failures occur in round f.
Everyone decides in this extension.
If two decide differently we are done, so assume they all decide the same. Wlog they decide 1.
Since r is bivalent, there must be another 1-round extension of r, call it r^0, in which some (nonfaulty) process q decides 0.

In r^0, some single process p fails in round f.
And p must fail to sends a message to q in round f (otherwise the execution would look to q just like r^* and it would decide 1).

Now define a very small chain: an "intermediate" execution $r^{0,1}$ that is identical to r^0 but in which p sends its message to q.
Then $r^{0,1}$ looks to q like r^*, so it decides 1.
But $r^{0,1}$ looks to every other nonfaulty process like r^0, so they decide 0.
Thus, we have disagreement, as needed.

2.4 Discussion

The final round argument is really like the earlier round arguments. There should be an even shorter proof that treats this as a special case.

Note the slight trickiness that arises in these proofs: Every failure is apportioned to some round. There is an ambiguity when someone fails "between" rounds—can be assigned to either round. Has to be handled carefully in the proofs.

The key seems to be what happens in a round that happens right after a bivalent execution.
If a single failure can occur at the next round, then the execution can be forced to remain bivalent.
And if the decision must be made after that additional round, then disagreement is possible.

What can we say about the situation where the maximum number of failures is some value f, but the actual number that occur in some particular execution may be smaller?

Note that this proof assumed right at the beginning that everyone decides right at the end.
We could generalize this to allow processes to decide earlier if possible.
There are "early stopping" algorithms in the literature that terminate early in better cases, where fewer failures occur.

This lower bound doesn't show that more than f rounds are needed in every execution—just that this is necessary in the worst case. The proof assumes we are free to expend the maximum number of failures f.
The next paper considers what is possible for early stopping algorithms.