Reading: Charline - Toney paper
Ch. 2.2
Next: student presentations - last papers

Last time:
Working on fault-tolerant network computation
Failure detectors, consensus, beast

ED:

Completeness: If step i occurs & step j doesn't, then eventually a final suspect
occurs with no following unsuspect

Accuracy: 4 versions, P, S, and P + S

Focus on S for now:
If not all processes fail, then some (correct) process is never suspected
by anyone.
Solving consensus using failure detectors

Each process accesses FD service, in addition to usual commun. service
(pt-p1 S/R channels between all pairs).

My book, Section 21.4, shows how consensus can be solved using a
perfect failure detector.

Contribution of CT is to show it can be solved even with the
weaker FDs $\diamond f$, $\Diamond p$, and $\Diamond S$.

Solving consensus using $S$: (for any number $f < n$ of failures)

Recall this says some correct guy is never suspected.

Algorithm is lot like the one in my book, 21.4.

Complications arise because:

- Someone can be suspected, unsuspected, suspected, ...
- Correct guys can be suspected (as long as at least
  one isn't).

Each process $i$ accumulates values in a vector $V_i$.

At any time $V_i$ will contain the initial value $v_i$ for some
processes if $i$ will contain "null" for the others.
Initially, contains its own initial value \( v_i \), the rest null.

Also records this info in a \( S_i \) vector - represents it's most recently acquired info.

Then does 2 phases

**Phase 1:**

For each \( n-1 \) rounds. In each round, send \( S_i \) vector (recent info) to everyone.

Want to get this round's info from everyone not currently suspected.

(Technically: Check a sparsification time at some point, say \( vi \) currently not suspected, have received that \( vi \) current round message)

Put any new info (new \( v_i \)'s) you get into your \( V_i \) vector.

Also set the new \( S_i \) vector to contain just the new info.

**Phase 2:** Now send your \( V_i \) to everyone. Wait to receive msg from all not currently suspected.

Then take "intersection" of all the vectors received at phase 2 (not your own)

That is, in any position \( j \), make the value null if any of the vectors has null; else keep the (common) value.

Decide on first non-null value for the intersection vector.

**Corrections:** (of consensus alg)

**Termination:** argue no one gets stuck at any round.

Suppose someone does; consider the first set where anyone gets stuck.

Say i gets stuck there.
What might I be waiting for? Some other j.

1. $j$ eventually fails.
   Then completeness says eventually $i$ suspects it (or keeps suspecting it), which is enough to keep $i$ from waiting.

2. $j$ never fails.
   Then $j$ eventually reaches this round (since it's the earliest rd where anyone gets stuck), sends the required msg.
   So $i$ eventually receives it, which is enough.

We must also argue that after the $m$'th round everyone can pick a value.

requires that at least one value be non-null.

Let $c$ be a non-faulty process that is never suspected (assumed to exist by $\delta$ definition).

Then everyone hears from $c$ at round $1$, which already ensures that all end up with $v_c \neq \text{null}.$

Validity: Obvious - only actual initial values are sent around.

Agreement: Follows if we show all processes that complete rd $m$ have the same $v$ after round $m$.

Let $v_i^n$ denote vector of $i$ after rd $n$.

Hence let $c$ be a process that never fails and is never suspected, as above.

Clain $v_i^{n-1} \leq v_c^{n-1}$ for all $i$ that finish before round $n-1$.

That is, the ordering has the same vectors that are consistent, based on having actual info.
Proof. Consider any position \( j \) in \( V_c^{m-1} \) that's non-null.

So, contains \( v_j \).

If process \( c \) got that value at some round prior to \( m-1 \),
then \( c \) sends it at round \( m-1 \) still in phase 1, and they all receive it.

(Because they never suspect \( c \).)

On the other hand, if \( c \) doesn't receive this value until just at the end of round \( m-1 \) of phase 1, it's too late
for it to send to others.

But we can argue they must already have it:

- Each process \( i \) that has received it at \( m-2 \) (since processes only send out new, incremental info)
  + if received it for some other \( c \) who first received it at
    - \( m-3 \), etc.

Thus, the value \( v_j \) gets relayed through \( m-1 \) distinct processes
other than \( c \) to reach \( c \). That's everyone.

So, they all must have \( v_j \).

Claim: \( V_c^{m-1} = V_c^m \) for all \( i \) that finish rd \( m \).

Anyone who sends a msg at rd \( m \) sends a vector \( V_c^{m-1} \). Thus, \( V_c^{m-1} \) is included in the intersection
anywhere this is computed.

Also \( V_c^{m-1} \) is one of the vectors, so it must be received by
everyone (since no one suspects \( c \)).

So \( V_c^{m-1} \) is actually equal to the intersection calculated everywhere.
Solving consensus using $\diamondsuit S$

S is a make an artificially strong assumption about one process never getting suspected.

They get a reason.

They describe another, quite different style of algorithm, which makes just assuming that some correct processes is eventually not suspected by anyone.$\diamondsuit S$

The protocol is quite a bit harder to tolerate $< \frac{m}{2}$ failures. Turns out, for reasons similar to the imposs. $\land$ for atomic registers, that this failure bound is optimal.

The idea that lets the protocol work in the presence of "eventually good" behavior is to keep trying repeatedly to reach agreement. When (if) an attempt is made after the point where behavior has become good, the attempt succeeds.

The algorithm uses "rotating coordinators", something we saw earlier in the 3-phase commit algorithm (synchronous model).

Each phase

1. Execute any of 0, 1, 2, ...

2. Each phase has an "owner" (coordinator), rotating and rotating them processes, mod $m$.

3. Each phase has $4$ rounds of comm. during which the owner tries to coordinate an agreement.

4. Ship the rest, refer to the paper.

A key idea is that a coordinator must get agreement from a majority of the processes before it actually decides.

The majority plays the usual role of information repository.
When a coordinator picks a value, it first queries a majority of the peers and chooses a value consistent with the latest one it knows about.

Interaction of majorities is needed to prevent disagreement.

Assessment

This alg. relays more interest than the $S$-based alg.

But actually, it's essentially the same as an earlier algorithm by Dwork, Lynch + Stockmeyer, described in §2.5.6.3 of my book. (Last §)

Main difference is explicit use of failure detector.

FDs are nice abstractions.

My favorite is $\Diamond P$ - allows talky about arbitrary suspicions if to any party, then see what happens when things become well from some point onward.

Other contributions of paper:

$m/2$ failure immune result, $\exists$ consensus given $\Diamond P$.

The intuition is that the $\Diamond P$ guarantees don't have to kick in in any given time bond, so can postpone the long enough to reach disagreement between two components in a partitioned network.

Atomic Boost + its equivalence with consensus.
Abstract

integrity:
Only correct msgs delivered, at most once at any location.

order:
If a msg received at 2 locs, same order at both.

self-delivery:
(In fair exec), any msg burst at non-failing port
is eventually delivered at that port.

agreement
(In fair exec), any msg delivered at some process is eventually
delivered at all non-fail. proc.

Easy to guarantee without the order property:
Just make sure to deliver again that before you deliver a msg to the
client, you propagate it (by sending it on reliable low-level
channels) to everyone.
And be sure to deliver own msg.

Adding the order property makes things much harder, in fact,
impossible in the presence of even 1 failure.

To see this, show that a 1-FE solution to Abstract (where
"in fair exec" is replaced by "in fair exec with 1 failure")
would yield a 1-FE consensus algo...
Redo: ABCast, my preferred version

Stronger, in that it is "uniform" w.r.t. failed processes. (the way I stated consensus for stopping failures).

Correctness for an execution $x$:

- Exists a (finite or infinite) sequence of messages such that:
  - $S$ contains only msgs that someone has seen, each at most once
  - $S$ contains all msgs received by anyone
  - (more strongly ordering property) The sequence of msgs received by each process forms a prefix of $S$.
  - If $x$ is fair, then every msg $abc$ in $S$ is in $S$.
  - "..." every msg in $S$ delivered to an on every non-failing port.

Also, f-fault-tolerant version of this problem adds the additional hypothesis "if there are failures on at most $f$ ports" to the above conditions (the last two).

Now we can argue an "equivalence" between $f$-fault-tolerant consensus + $f$-fault-tolerant ABCast & having an oracle for either yields a solution to the other. But since $f$-FT consensus isn't implementable in asynchronous model for $f \geq 1$, this is asserting equivalence between 2 unimplementable problems, tricky.
Solution to $f$-FT $ABCast \Rightarrow f$-FT consensus

Consensus algo: $\xi$ (Doesn't even use the S/R channels)
Best your init value on the $ABCast$ service
Decide on the first value you receive.

agreement: easy because of prefix property
validity: only actual initial values are sent on $ABCast$, so only these are ever delivered

f-termination: if at most $f$ failures on outer parts, also at most $f$ on inner parts. Consider any non-failing process $i$ who initiates consensus, show it must decide.

$\xi$ eventually beats its value.
Since every of $x$ has $\leq f$ failures, the msg gets sent in $S$ must be included in $S$. So $i$ must be delivered to $i$.

So $i$ eventually decides.

Corollary: Can't solve $1$-FT $ABCast$ in any $S/R$ dist. system
Solve to f-FT consensus \Rightarrow solve to f-FT AB cast

Using multiple instances of consensus.
Model as composition with multiple consensus services.

When msgs arrive from client, ship them off to everyone on SIR channels.
Collect all msgs you know about in a set candidates.
Submit your candidates set as input to first consensus, wait for decision. When decision comes, take the resulting set of msgs, order them according to some commonly-known default ordering, & deliver to client.
Then remove them from the candidates set.
Continuing: submit new (reduced) candidate set to next consensus, etc.

Conclusions:
The common order S is the sequence of sets that result from consensus, each in the default order.
This S clearly yields the safety properties of ABCast.
For the f-FT properties, assume at most f failures for the ABCast part.
Then at most 1 failure for each consensus, so each returns.

Each non-failures process delivers each msg in S, therefore.

Remains to argue each msg sent on non-fail process essentially arrives in actual in S, that is, it eventually arrives up in some consensus decision set.

ABCast

Say m sent on part i. Then it sends on S/R channels + eventually arrives everywhere.

So after some pt, m is in every the init set of every participant. So by validity, it will be in the decision set.

A lot of other communication services have been studied. Different guarantees about ordering which can be expressed in terms of different conditions + reliability.

See Hadzilacos + Toueg papers. (though they don't express things in the same style)

E.g. FIFO: If process i sends m before m' + m appears in S, then m precedes m' in S.

E.g. causal: Define causality order based on external events:

ABCast: all events at same location

- beast to deliver (anywhere) of same msg.

- Take defeasible transitive closure.

Then say if beast m) causally precedes beast(m') then m in S, then m precedes m' in S,

And other variants.
Last topic: Reliable Comm from Unreliable Channels

FIFO reliable channels
Less reliable - allow loss, dup, res
Don't consider manufacture of completely bogus msgs (integrity)

Chapter also considers crashes that lose info, but I won't get to this.
Focus on simple link between 2 nodes

Implement the reliable FIFO channel (with state containing a queue)
using $C_{12}, C_{21}$, channels with some weaker guarantees, composed
with proc $P_1 + P_2$.

Want fair traces of composition (hiding low level sends vs. receives) ≤ fairness ($F$)

Steiner's Protocol
Assume C's can lose, dup + reorder

However
1. Can't lose everything
   Specifically, if as many sends then as many receives
   (more precisely, receives for as many of the sends)
2. Can't duplicate too much
   Each msg dup only fin. many times
   (can reorder arbitrarily, though.)
Protocol

P₁ remembers high-level msgs in buffer, tagging with integers, starting with 1.

P₁ repeatedly sends first msg in buffer, with tag.

P₂ accepts first msg tagged with 1 that it receives.

P₂ accepts each subsequent msg tagged with tag = previous + 1 exactly if

tag = previous are accepted + 1.

P₂ accepts msg in buffer for delivery to user

acks hi-level msgs by sending tag back (repeatedly).

When P₁ receives ack for current tag, moves on to begin processing

next hi-level msg.

Code p. 684-5

Proof: LTTT

ABP:

Variation on stemming using bold tags, in fact, just 0 & 1.

Regard as optimized version of stemming, with integers replaced by

low-order bits.

Used in final nonl. comm., not a practical protocol.

Needs stronger assumptions on C channels, e.g.,

Model msgs. finite dup., a WLL

read loss limitation, perfect

(0 msg sent, 0 msg of them)

are received

Protocol: P₁ puts msg in buffer, tags with 0 or 1, alternately starting

with 1, send 0 (repeat) wth tag.

P₂ accepts first msg it receives with tag = 1, then accepts later

msgs if tag = tag of previously accepted msg.

P₂ puts msg in buffer for delivery to user, delivers, acks with tags.

P₁ receives ack for current tag, moves on to next msg.

Code p. 698-99
If: Safety! Relate formally to blocking, i.e., duplicate.
So do this, consider blocking with the stronger FIFO channels.
For this case, get new limit about the order in which tags appear in
the channels & processes:

Let $T$ = sequence of integers consisting of queue, tag, queue, tag,

Then $T$ is nondecreasing & difference $\leq 1$.

Sim relm from ABP to Blocking: (this new 8.)
Everything same, except tag in ABP is low-order bit of tag in Blocking
show as usual
stay correspondence; same actions
Any steps of the proof are the low-level receive steps, where we
must argue that the same decision to accept or reject the msg
is made in both algorithms.

E.g.: receive $(m, 8)$, where $m$ is accepted by $P_2$

$\text{Step: } 8 \neq \text{ABP: tag}$

Means in Blocking $(k \neq \text{tag } 2)$

integer tag of
incoming msg

so show accepted in
$\text{need } k = \text{tag } 2 + 1$

Use lemma (hint) above: incoming tag has different parity,
with hint, can only be tag $+ 1$.

Similar args used for $8m$ not accepted by $P_2$

[Notes: some handwriting is unclear and may require further clarification.]
Reissues: LTTR. Can show directly, or via a kind of
reissues - preserving six reln.
(based on the strong 1-1 correspondence between actions)

Moral: With unbold tags, can tolerate all failure types
With bold tags, can tolerate loss & dup, not res.

Finish by considering protocols that tolerate reordering.

Q: What exactly goes wrong with ABP when used with
reordering channels?
A: P_2 can be fooled into accepting an old high-level msg
that happens to arrive tagged with same bit as the one
currently expected.
Can cause duplicate delivery.

Q: Might there be other bold-tag protocols that do tolerate reordering?
A: 3 results:
1. Nonexistence of bold-tag protocols tolerating res + dup
2. Bold tag protocol tolerating loss + res, not dup
   (not practical, high complexity)
3. Nonexistence of efficient protocols tolerating loss + res
   in terms of # of low-level msgs.

Thus, the high complexity is unavoidable.

Bounded tag: Somewhat by saying M & M both finite
both high msg depth - low level
1. Impossibility for rec + dup

Assume there is a protocol \( (P_1, P_2) \) using odd tags (finite \( M' \))

... to implement \( F \) (=IP0 reliable) over channels \( Q_{12} \) and \( Q_{21} \).

As defined to admit

\[ \{ \begin{align*}
&\text{no loss} \\
&\text{no offered dup (finite \( M' \))}
\end{align*} \]

Show contradiction:

Run the system until \( P_1 \) sends a low-level msg \( m \).

Continue until \( P_1 \) sends a different low-level msg \( m_2 \).

Continue until no longer possible. Doing this may involve SEND inputs; include them as needed.

Set \( \alpha_1 \).

... etc.

Let \( n = \# \) of SEND events in \( \alpha_1 \).

Now extend \( \alpha_1 \) with one more SEND and continue safely, with no further SEND inputs.

Set \( \alpha_2 \).

By def of \( F \), \( \alpha_2 \) contains exactly \( n+1 \) RECEIVE events.

Set \( \alpha_3 = \) part of \( \alpha_2 \) up to last REC.

By \( F \), \( \alpha_3 \) contains exactly \( n+1 \) RECEIVE events.

Now construct alternative exec \( \alpha_4 \) extending \( \alpha_1 \):

- \( \alpha_4 \) looks like \( \alpha_1 \) to \( P_1 \).
- \( \alpha_4 \) looks like \( \alpha_3 \) to \( P_2 \).

That would give \( n \) SENDS + \( n+1 \) RECS, contradiction.
How to construct:

Stop all events of $P_1$ after $\alpha_1$ and let $P_2$ proceed as in $\alpha_3$.

Why is it possible for $P_2$ to do all the same things as in $\alpha_3$?

Only issue is low-level receive - how do we know we can get these without $P_1$'s help?

Possible because all low-level msgs $P_1$ sends after $\alpha_1$ contain ms that $P_1$ has already sent in $\alpha_3$.

So can use dup + reo to allow $P_2$ to receive them again.
2. Bold tag alg tolerating loss + res

Now assume no duplicates
Possible to implement F, with loss + res, with bold tags.

Probe alg:
Not practical - more as a counterexample to an impossibility result.

2 layers:
1. Use given channels to implement channels that don't reorder, but can lose & dup (well)
2. Use these interior channels to implement F (using ABP)

Compare layers:

Multiply channels: Need to be fair to both, so actually need a stronger loss-bound property.
E.g., "send $m$ exactly $m$ times $\rightarrow$ deliver $m$ exactly $m$ times, for each $m$"

Layer 1: Impl. non-res (but can lose - well - or dup)
using given non-dup (but can lose - well enough here - or dup)

Receiver's view:
P$_1$ sends msg only in response to explicit probe msg from P$_2$.
Response always has value of latest di-level msg received from U.
(latest) (OK to lose some)
P_2 sends probes continually, counting in pending the total number it's sent.

Counts (in count(m)) number of copies of each hi-level msg m received since it last delivered a msg to U_2.

Whenever delivers msg to U_2, P_2 sets old := pending, remembering total # of probes sent before this delivery.

When count(m) > old for some particular m, P_2 can deliver m. Knows it is recent - a copy was sent by P_1 after last delivery.

can use to show no reo occurs.

Reasons:
Needs N_{REO} if oo many SENDS, then oo many of these have concept REC.
Suppose oo many SENDS - then P_2 keeps pushing, P_1 keeps answering, channels keep delivering something.
So oo many REC.

Concept: is oo many SENDS?
Because each msg corresponds to latest, at some point after previous REC, and in msg SENDS occur.
Then the REC's must come up to oo many SENDS.

Complete protocol: Combine as above.
(need all to be fair to both channels.)

Complexity: More and more msgs needed to deliver last + last hi-level msgs (esp. if low-level msgs lost)

Q: Possible to avoid cost?
Given: Needs \( \text{WLL} \) - if \( \infty \) may SENDS, then \( \infty \) may of those have corresp. REC's.

Suppose \( \infty \) many SENDS - then \( P_2 \) keeps probing, \( P_1 \) keeps announcing, channels keep delivering (something - WLL).

\( \infty \) inf. may REC's.

Corresp. to \( \infty \) many SENDS?

Because each msg corresponds to latest, at some point after previous REC. If \( \infty \) may SENDS, get corresp with \( \infty \) many different SENDS.

**Complete protocol:** Combine as above
\( (\text{Need SSL to be fair to both channels.}) \)

**Complexity:** Bad - more \& more msgs needed to deliver later \& later hi-level msgs (esp. if low-level msgs lost)

Q: Possible to avoid cost?

3. **Nonexistence of Low-Complexity Protocols (Iterating Losses)**

Assume underlying channels don't duplicate
\( \{ \text{can lose, subject to SSL can resend} \} \)

Assume alg. \((P_1, P_2)\) implements \( F \), show "high cost."

Q: How to measure cost?

A: Paraphrase: protocol is \( k\)-seq-bkd if, from any point in the error, it's possible to deliver a new msg with \( k \) rec events.

More precisely:
"from any point" - Consider points after "complete" finite errors, where
\[ \# \text{SENDs} = \# \text{REC's}. \]
It's possible to deliver: \( \exists \alpha \)

- The portion of \( \alpha' \) after \( \alpha \) has \( 1 \) more SEND + \( 1 \) more REC (exactly)
- \# of rec events in \( \alpha' \) after \( \alpha \) is \( \leq k \)
- All low-level msgs rec. by \( P_2 \) in \( \alpha' \) after \( \alpha \) are also sent after \( \alpha \).

(Best-case \( \leq k \))

Theorem: \( \exists k \)-fold protocol \((P_1, P_2)\) (for any \( k \))
implementing \( F \) using lossy rec (non-dup) channels.

Proof: Suppose \( \exists \alpha \) for such \( k \).
Suppose we could produce (multiset \( T \) of elements of \( M \))
complete exec. \( \alpha \)
and extension \( \alpha' \) satisfying:

1. All msgs in \( T \) are "in-transit" (sent but not received)
from \( P_1 \) to \( P_2 \) after \( \alpha \).
2. Multiset of low-level msgs received by \( P_2 \) in \( \alpha' \) after \( \alpha \)
is submultiset of submultiset of \( T \).

Then get contd: Produce alternative exec. \( \alpha' \), extending \( \alpha' \),
looks like \( \{ \alpha \to P_1, \alpha' \to P_2 \} \).

\( P_2 \) can receive needed new msgs from those already in transit.
More recs than sends, contd.
So want to produce the bad situation

Claim: If a multiset $T$ of low-level mgs is in transit $1 \rightarrow 2$, then either the bad situation already exists, or we can augment $T$ to a larger multiset in transit.

More formally:

Claim: Suppose $\exists x$ is complete exec.

Then at least one of these holds:

1. $\exists x' \subseteq x$ s.t. the multiset of low-level mgs in $1 \rightarrow 2$ after $x'
\quad$ where $T$ contains $\leq k$ copies of each element

2. $\exists x' \subseteq x$ is a subset of $T$

or

2. $\exists$ complete extension $x'$ of $x$

new multiset $T'$ of low-level mgs in transit after $x'$, where $T'$ contains $\leq k$ copies of each elt.

If claim true: Create the bad situation by working inductively:

- $\exists x_0, x_1, \ldots$ complete execs
- $\exists T_0, T_1, \ldots$ multisets in transit

If $\exists x_0$ initial states, $T_0$ empty

If Case 1 holds - done
- else Case 2 - produce $x_1, T_1$ & continue.

If $\exists x_1$ ever holds, done
- Else go on forever.

But impossible, since $\exists$ at most $k$ copies allowed, of each
Proof Claim

Given \( k \), produce \( k \)-ext \( x' \) of \( x \) (say for \( m \in M \))

\[ x \xrightarrow{\text{transit}} x' \]

If \( m \) msg received by \( P_2 \) in \( x' \) after \( x \leq T \) then Case 1 satisfied.

So assume not. Then \( \exists \ p \in M \) s.t. \( \# \text{ of new rec}(p) \) events > number of copies of \( p \) in \( T \).

Define \( T' = T \cup fp \)

Then \( T' \) still has \( \leq k \) copies of each \( p \) (since \( x' \) is a \( k \)-extension)

so \( \# \text{ of rec}(p) \) events \( \leq k \)

must get complete set of \( x \) leaving \( T' \) in transit.

(Not in transit after \( x' \), since received.)

the new \( p \) is

Technical: \( \exists \text{ send}(p) \) in \( x' \) after \( x \); because \( m \) msg received after \( x \); assumed sent after \( x \).

\( x' = \text{part of} \ x \) ending with first such \( \text{send}(p) \)

\( T' \) in transit after \( x' \); we need \( x_1 \) to be complete, though.

Recall \( x' \) after \( x \) contains \( \text{SEND} + \text{REC} \)

If \( x_1 \) includes both a neither, then \( x_1 \) satisfies Case 2.

Rem case is where \( x_1 \) contains the \( \text{SEND} \) but not the \( \text{REC} \):

\[ \xrightarrow{\text{SEND}(m)} \ x_1 \xrightarrow{\text{send}(p)} \ x_1 \xrightarrow{\text{REC}(m)} \]
Then add some stuff to x, to deliver m (REC) but without delivering the low-level msg up to P2

That is, "lose" all then run fairly, with no new SENDs,
and all recs caused by new send is ("lose" all previous msg)

(Can run any IOA fairly from any point.)
Then by corr. cond., eventually REC(m) occurs. Stop there.

---

Tolerating Crashes

Solved all questions of impl. rel. using unl. channels, if pros are reliable.

Don't consider stopping, by - there are permanent, & in setting of only 2 pros, that's pretty meaningless.

Instead, consider crash-recovery failures

Crash loses info. (If not, just like going slow.)

Models physical processors with volatile memory
or combination of stable/volatile memory.

In crash, volatile memory lost.

Recovery involves resuming from previous state of stable DEFAULT state of volatile

In reality, recovery protocol would run, but we'll model this as a single recovery step.

For now, consider the case when:

Results:
1. Impos of F using crashing proces
2. Recover problem regs, still impossible
3. A practical protocol - just mention, LTTR

---