Reading: Charles - Toney paper
Ch. 22
Next: Student presentation - last papers

Last time:
Working on fault-tolerant network computation
Fault detectors, consensus, honest

FD:

Completeness: If step 1 occurs & step 2 doesn't, then eventually a final suspect occurs with no following unsuspect (i.e.,)

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

Focus on S for now:
If not all flaws fail, then some (correct) process is never suspected by anyone.
Each process accesses FD service in addition to usual comm. service (pt-to-sk channel 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 $\diamond$, $\Box$, $\Diamond$, and $\Diamond S$.

Solving consensus using $S$: (for any number, $f < n$ of failures)
Recall this says some correct guy is never suspected.

Algorithm a 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).
- Process injects

Each process $i$ accumulates values in a vector $V_i$.
At any time, $V_i$ will contain the initial value $v_i$ for some process $j$, $i$ will contain "null" for the others.
Initially, contains its own initial value $v_i$; the rest null.
Also record this info in a $S_i$ vector - represents it's most recently acquired info.

Then does 2 phases

Phase 1:
$n-1$ rounds. In each round, send $S_i$ vector (recent info) to everyone.

Wait to get this round's info from everyone not currently suspected.
(Formally: Check a precondition true at some point, saying $v_j$ currently not suspected, have received that $j$'s current round message)

Put any new info (new $v_j$'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 (in your own).
That is, in any position $j$, write the value null if $v_j$ of the vectors has null; else keep the (common) value.

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

Correctness: (of consensus alg)
Termination: argue no one gets stuck at any round.
Suppose someone does; consider the first set where anyone gets stuck since $v_i$ gets stuck there.
What might I be waiting for? Some other i.

Two cases:
1. i eventually fails.
   Then completeness says eventually i suspects it (+ keeps suspecting it?), which is enough to keep i from waiting.
2. i never fails.
   Then i 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 nonfaulty process that is never suspected (assumed to exist by s definition).

Then everyone hears from c at round i, which already ensures that all end up with v = null.

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

Agreement: Follows if we show all procs that complete rd m have the same V after round m.

Let

\[ V^i_c \] denote vector of i after rd m.

Claim \[ V^m_{c'} \leq V^m_i \] for all i that finish.

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

So, contains \( V_j \).

If process \( c \) got that value at some round prior to \( m-1 \),
then \( c \) sends at most \( r_d \) 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:

Since \( c \) gets it at phase \( m-1 \), it gets it from someone else who just received it at \( m-2 \) (since processes only send out new, incremental info).

The receive it from someone else who just received it at \( m-3 \), etc.

Thus, the value \( V_j \) got 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 \( rd \) \( m \)

Argue who gets a msg at \( rd \) \( m \) sends a vector \( \overrightarrow{V_c^{m-1}} \).

Thus, \( V_c^m \) 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 we are suspect \( c \)).

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

5 is making an artificially strong assumption about one process never getting suspected.

They get a paper.

They describe another, quite different style of algorithm, which makes just assuming that some correct process is eventually not suspected by anyone.

The protocol is quite a bit harder and tolerates \( \frac{n}{2} \) failures. Turns out, for reasons similar to the impossibility 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 coordinates", something we saw earlier in the 3-phase commit algorithm (synchronous model).

Each phase has an "owner" (coordinateator), rotating round robin through processes, \( 1 \leq i \leq n \).

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

(End the rest, refer to the paper.)

The 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.

LTTR. (DLS, Paris)

Assessment

This alg. seems more interesting than the S-based alg.

But actually, it's essentially the same as an earlier algorithm by Dwork, Lynch + Stockmeyer, described in 5.25.6.3 of

my book. (last 5)

The

main difference is explicit use of failure detector.

FDs are nice abstractions.

My favorite is $\Diamond P$ - allows talking about arbitrary suspensions up to a point, then see what happens when things "become well" from some point onward.

Other contributions of paper:

$n/2$ failure implies result for consensus given $\Diamond P$.

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

Atomic Beast + its equivalence with consensus:
Abstract

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

order:
If a msg received at a loc, same order at both.

self-delivery:
In fair exe, any msg at non-failing port is eventually delivered at that port.

agreement
In fair exe, any msg delivered at some proc 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 DFT solution to Abstract (where "in fair exe" is replaced by "in fair exe with 1 failure") would yield a DFT consensus.
Redo: ABCast, my preferred version

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

Correctives for an execution $x$:

1. $S$, 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 beast by a process is in $S$
   - "... every msg in $S$ delivered to an every non-failing part."
Solution to 1-FT ABCast \Rightarrow 1-FT consensus

Consensus alg: \( \Phi \) (Doesn't even use the S/R channels)
Beast your int 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 those are ever delivered

1-FT termination: if at most 1 failure in outer parts, also at most 1 on inner parts. Consider any non-failing process \( i \) who initiates consensus, show it must decide.
It eventually beats its value.
Since \( \text{exec of } x \) has \( \leq f \) failures, the msg gets from \( x \) must be included in \( S \), \( x \) must be delivered to \( i \).
So \( i \) eventually decides

So \( i \) gets some first message.

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

- Acturally not doing multiple
- Using multiple instances of consensus
- Model as composition with multiple consensus services

When msgs arrive from client, ship them off to everyone on S/R 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 now (reduced) candidate set to next consensus, etc.

Conclusion: 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-failed process delivers each msg in S, therefore.

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

Say \( m \) sent on port \( i \). Then it sends on SIR channels + eventually arrives everywhere.

So after some pt, \( m \) is in every the limit 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 on S.

See Hadjicostas & Soney paper. (though they don't express things in the same style)

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

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

- All events at one location
- \( m \) precedes \( m' \) delivery (causally) of same msg.
- Take abstraction change.

Say if \( m \) causes \( m' \) causally precedes \( m' \) in S, then so is \( m \) + \( m' \) in S, and other variants.
Last topic: Reliable Comm from Unreliable Channels

FIFO reliable channels

can't consider - allow loss, dup, reordering

Chapter also considers crashes that lose info, but I won't get to this.
Focus on single 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 transfer of composition (hiding low level sends & receives) & fairness (P)

Stenning's Protocol
Assume (s can lose, dup & reorder)

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

$P_1$ remembers high-level msgs in buffer, tagging with integers, starting with 1.

$P_1$ repeatedly sends first msg in buffer, with tag.

$P_2$ accepts first msg tagged with 1 that it receives.

accepts each subsequent msg by sending tag back exactly if tag = previous one accepted + 1.

puts accepted msgs in buffer for delivery to user.

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

When $P_1$ receives ack for current tag, moves on to begin processing next hi-level msg.

Code p. 694-5

Proof: $LTTR$

ABP:

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

Regard as optimized version of Steenig, with integers replaced by low-order bits.

Used in formal env, not a practical protocol.

Needs stronger assumptions on C channels, e.g., can't read in msg, finite drop, a WLL

weak loss limitation property

(no msg sends $\rightarrow$ as many of them are received)

Protocol: $P_1$ puts msg in buffer, tags with 0 or 1 alternately starting with 1. Sends 1st repeatedly, with tag.

$P_2$ accepts first msg it receives with tag = 1, then accepts later msgs if tag $\neq$ tag of previous accepted msg.

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

$P_1$ receives ack for current tag, moves on to next msg.

Code p. 698-699
If: Safety: Relate formally to Steenig (duplicacy)
To do this, consider Steenig with the stronger FIFO channels.
For this case, get new insight about the order in which tags appear in the channels & processes.

Let $T = \text{sequence of integers consisting of } q_1, q_2, \ldots, q_{12}, \text{tag}_1$

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

Simply from ABP to Steenig: (this new S.)

Everything same, except tag in ABP is low-order bit of tag in Steenig

Show as usual.

Step correspondence: same actions

Key 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 $(m, k)$, where $m$ is accepted by $P_2$

Then $b \neq k \implies \text{ABP, tag}_2$

Means in Steenig: $(k \implies k \neq \text{tag}_2 \mod 2)$

Integer tag of incoming thing

By concept.

To show accepted in $P_1 \implies k = \text{tag}_2 + 1$

Use Lemma (w.r.t.) above: In-cmig tag has different parity, with w.r.t. can only be $\text{tag}_2 + 1.$

Similar args used for $\{ \text{m not accepted by } P_2 \} + \text{for } P_1.$
Liveness: L TTR. Can show directly, or via a kind of
liveness-preserving simulation
(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
rendering channels?
A: P2 can be fooled into accepting an old high level msg
that happens to arrive tagged with time bit as the one
currently expected
Can cause duplicate delivery.

\[
\begin{array}{c}
(m, 1) \\
\downarrow \\
(m', 0)
\end{array}
\]
accepted again

Q: Might there be other bold-tag protocols that do tolerate rendering?
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\textsuperscript{'} both finite
the local msg alphabet low level
1. Impossibility for rec + dup

Assume there is a protocol $(P_1, P_2)$ using odd tags (finite $M'$) to implement $F$ (IFO reliable) over channels $Q_{12}$ and $Q_{21}$.

As defined to admit

\[ \begin{cases} &\text{rec} \\
&\text{no loss} \\
&\text{finite (un)dup} \end{cases} \]

Show contradiction:

Run the system until $P_1$ sends a low-level msg $m_1$.
Continue until $P_1$ sends different low-level msg $m_2$.

Continue until no longer possible.
Doing this may involve SEND inputs; include them as needed.
Let $\alpha_1$.

Let $n = \# \text{ of SEND events in } \alpha_1$.

Now extend $\alpha_1$ with one more SEND and continue exactly, with no further SEND inputs.
Let $\alpha_2$.
By def of $\alpha_2$, $\alpha_2$ contains exactly $n+1$ RECEIVE events.
Let $\alpha_3$ = part of $\alpha_2$ up to last REC.

Now construct alternative exec $\alpha_4$ extending $\alpha_1$:

\[ \begin{cases} &\text{extending } \alpha_1 \\
&\text{looks like } \alpha_1 \text{ to } P_1 \\
&\text{looks like } \alpha_3 \text{ to } P_2 \end{cases} \]

That would give $n$ SENDS + $n+1$ RECs, contradiction.
How to construct:
Stop all events of $P_1$ after $x_1$ and let $P_2$ proceed as in $x_3$.

Why is it possible for $P_2$ to do all the same things as in $x_3$?

Only issue is low-level receives — how do we know we can get those without $P_1$’s help?

Possible because all low-level msg $P_1$ sends after $x_1$ contain msgs that $P_1$ has already sent in $x_3$.

So can use dup+reco to allow $P_2$ to receive them again.
2. Bold tag alg tolerating loss + reo
Now assume no duplicates
Possible to implement F , with loss + reo , 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 reo, but can lose + dup (WLL)
2. Use these internal channels to implement F (using ABP)

Compact layers:

3 pieces
compare to give P
node P2

Multiple channels: Need to be fair to both, so actually need a stronger loss-limit property.
E.g.: "send m in oo many times \implies" deliver m in oo many times, for each m.

Layer 1: Simpl. non-reo (but can lose - WLL - or dup)
          using given non-dup (but can lose indefinitely)
          WLL example above - or dup

Receiver-driven:
P1 sends msg only in response to explicit probe msg from P2.
response always has mode of latest bi-level msg received from U,
      (latest) (OK to lose one)
(unanswered keeps track of probes to answer)

P₂ 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 occurred since it last delivered a msg to U₂.

Whenever delivers msg to U₂, P₂ sets old := pending, remembering total # of probes sent before this delivery.

When count (m) > old for some particular m, P₂ can deliver m.

Since it is recent - a copy was sent by P₁ after last delivery.

Code p. 705-709

\[
\text{can use to show no race occurs.}
\]

Reasons:

Needs all - if oo many SENDS, then oo many of these have crosso. RECs.

Suppose oo many SENDS - then P₂ keeps probing. P₁ keeps answering.

Channels keep delivering something.

So oo many RECs.

Corresp. to oo many SENDS?

Because each msg corresponds to latest, at some point after previous RECS + if msg SENDS occur

Then the RECs must correspond to oo many SENDS.

Complete protocol: Combine as above

(need SSL to be fair to both channels)

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

Q: Possible to avoid cost?
3. Necessity of SENDS

- Needs SENDS - if no SENDS, then no many of those done correct.
- Suppose no SENDS - then P2 keeps probing, P1 keeps answering.
- Channels keep delivering (something - WLC).
- So inf. may RECs.

Corresp. to no many SENDS?

Because each msg corresponds to defect, at some point after previous REC, inf. may SENDS, get correct with no many different SENDS.

Complete protocol: Combine as above
(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?


Assume underlying channels don't duplicate
- can lose, subject to SSL
- can retransmit as part retransmitting

Assume alg. (P1, P2) implements F, show "high cost".

Q: How to measure cost?

A: Paraphrase: protocol is k-msg-bred if, from any point in the race, it's possible to deliver a non msg with k rec events.

More precisely:

"from any point": Consider points after "complete" finite races, where

# SENDS = # RECs.
It is possible to deliver: \( \exists x \) such that:

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

(best case \( \leq k \))

**Theorem**: \( k \)-order protocol \( (P_1, P_2) \) for any \( k \)
Implementing \( F \) using lossy rays (non-dup) channels.

**Proof**: Suppose \( \exists \) such \( k \).
Suppose we could produce \( s \) multiset \( T \) of elements of \( M \)
complete exe. \( \alpha \)
\( +k \)-extension \( x' \) 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 \( x' \) after \( \alpha \)
is submultiset of submultiset of \( T \).

Then get contradiction: Produce alternate exe. \( \alpha' \) extending \( x' \)
looks like \[ \alpha' \to P_1 \]
\[ \alpha' \to P_2 \]
\( P_2 \) can receive needed new msgs from that already in transit.
Yes more RECs than SENDs, contradiction.
Se: want to produce this bad situation

Claim: If a multiset $T$ of low-level merges 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 $\mathbf{\exists} \mathbf{\exists}$ is complete exec.

$T$ a multiset of low-level merges in transit $1 \rightarrow 2$ after $\alpha_1$

where $T$ contains $\leq k$ copies of $\alpha_1$ element

Then at least one of these holds:

1. $\exists k$-exec $\alpha'$ of $\alpha$ s.t. the multiset of low-level merges $\rightarrow_2 \alpha_1$ after $\alpha$ is a submultiset of $T$

or

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

new multiset $T'$ of low-level merges in transit after $\alpha'$

where $T'$ contains $\leq k$ copies of each elt,

$\{T \subseteq T'\}$

proper submultiset.

If claim true: Create the bad situation by working inductively

$\alpha_0, \alpha_1, \ldots$ complete execs

$T_0, T_1, \ldots$ multises in transit

$\alpha_0$ initial state, $T_0$ empty

If Case 1 holds - done

else Case 2 - produce $\alpha_1, T_1$, and continue.

If claim true, Case 1 ever holds done

Else go on forever.

But impossible, since \{ fin many merges in $M$ \}

at most $k$ copies allowed, of each
Prop. Claim
Given $\alpha$, produce $k$-ext $\alpha'$ of $\alpha$ (say for $m \in M$)

\[ \alpha \xrightarrow{T} \alpha' \]

$T$ in transit

If set of msys received by $P_2$ in $\alpha'$ after $\alpha \leq T$ then Case 1 satisfied.

So assume not. Then $\exists p \in M$ s.t. # of new rec $(p)$ events $> \#

\text{number of copies of } p \text{ in } T.$

Define $T' = T U \{ p \}$

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

(20) # of rec $(p)$ events $\leq k$

Must get complete set of $\alpha$ having $T'$ in transit

(Not in transit after $\alpha'$ since received.)

the new $p$ is

Technical: $\exists$ send $(p)$ in $\alpha'$ after $\alpha$, because msys need after $\alpha$, assumed sent after $\alpha$.

$\alpha' = part \wedge \alpha$, ending with first such send $(p)$

$T'$ in transit after $\alpha'$

We need $\alpha'$ to be complete, though.

Recall $\alpha'$ after $\alpha$ contains SEND + REC

If $\alpha'$ includes both a neither, then $\alpha'$ satisfies Case 2.

Rem case is where $\alpha_1$ contains the SEND but not the REC:

\[ \alpha \xrightarrow{T} \alpha_1 \xrightarrow{T'} \]

$\alpha_1 \cup T'$
Then add some stuff to \( x_1 \) to deliver \( m \) (REC) but without delivering the low-level msg up to \( p_2 \).

That is, "lose" \( p_\) then run fairly, with no new SENDS,
+ all \( REC \) to \( p_2 \)'s caused by new \( send_{12} \) s ("lose" all previous msgs)

(Can run any IOA fairly from any point.)

Then by corr. cond., eventually \( REC \) or \( m \) occurs. Stop there.

---

**Tolerating Crashes**

Solved all questions of impl. rel. using unixd. channels, if wires are reliable.

Don't consider stopping, Byz - these are permanent, & in setting of only 2 wires, that's pretty meaningless.

Instead, consider crash-recover failures

Lost 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
(\( \Rightarrow \) default state of volatile)

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

---

For now consider the commands:

**Results:**

1. Impos of F using crashing forces
2. Weaken problem regs, still impossible
3. A practical protocol - just mention, LTR

---

Special case:

Assume crashes lose all info, \( \Rightarrow \) recovery \( \Rightarrow \) arbitrary start state