Class 22
6.852
Fall '01

bug: chalk
~~schedule of presentations~~
papers for Thursday's presentations

code

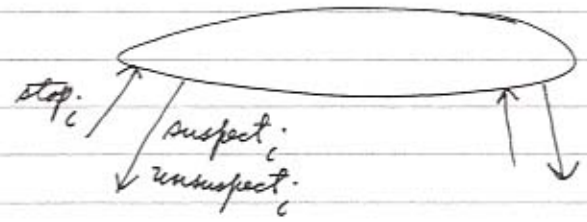
Reading: Chandra – Toueg paper
Ch. 22

Next: student presentations – list papers.

Last time:

Working on fault-tolerant network computation
Failure detectors, consensus, bcast

FD:



Completeness: If $stop_i$ occurs + $stop_j$ doesn't, then eventually a final $suspect(i)_j$ occurs with no following $unsuspect(i)_j$.

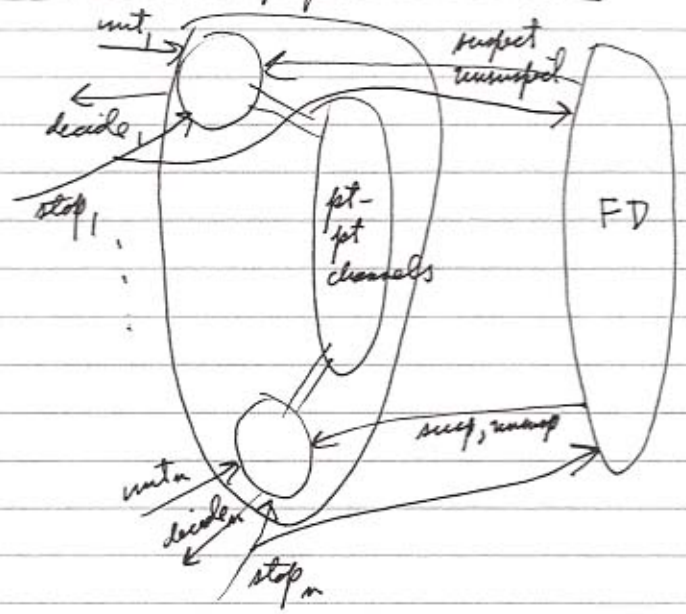
Accuracy: 4 versions, P, S, $\Diamond P$ + $\Diamond S$

Focus on S for now:

If not all procs fail, then some (correct) process is never suspected by anyone.

Solving consensus using failure detectors

Archit:



Each process accesses FD service, in addition to usual comm. service (pt-pt 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 S, $\diamond P$, and $\diamond S$.

E.g. 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 init).

Each process i accumulates processes' init values values in a vector V_i.

At any time, V_i will contain the initial value v_j for some processes j, & null contain "null" for the others.

Initially, contains its own initial value v_i, the rest null.
also records this info in a δ_i vector - represents it's most recently acquired info.

Then does 2 phases

Phase 1:
n-1 rounds. In each round, send δ_i vector (recent info) to everyone.
Want to get this round's info from everyone not currently suspected.
(Formally: Check a precondition true at some point, saying $\forall j$ 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 δ_i vector to contain just the new info.

Phase 2: Now send your entire V_i to everyone. Want to receive msg from all not currently suspected.
Then take "intersection" of all the vectors received at phase 2 (+ 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.

Correctness: (of consensus alg)
Termination: Argue no one gets stuck at any round.
Suppose someone does; consider the first rd where anyone gets stuck.
Say i gets stuck there.

What might i be waiting for? Some other j.

Two cases:

1. j eventually fails.

Then completeness says eventually i suspects it (+ 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 nth round, (Phase 2) 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 1, which already ensures that all end up with $v_c \neq$ null.

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

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

Let ~~V_i^{n-1}~~ ~~the~~

V_i^r denote vector of i after rd r.

~~Claim~~ Let c be a process that never fails + is never suspected, as above.

Claim $V_c^{n-1} \leq V_i^{n-1}$ for all i that (finish) ~~finishes~~ round $n-1$

~~That is~~ The ordering here relates vectors that are consistent, based on having (at least as much) ~~more~~ actual info.

Pf: Consider any position j in V_c^{n-1} that's non-null.

So, contains v_j.

If process c got that value at some round prior to $n-1$, then c sends at next rd, still in phase 1, + 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 $n-1$ 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 $n-1$, it gets it from someone else who first received it at $n-2$ (since processes only send out new, incremental info)

+ he received it from someone else who first received it at $n-3$, etc.

Thus, the value v_j got relayed thru $n-1$ distinct procs other than c, to reach c. That's everyone.

So, they all must have v_j.

Claim: $V_c^{n-1} = V_i^n$ for all i that finish rd n

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

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

So V_c^{n-1} is actually equal to the intersection calculated everywhere.

Solving consensus using $\diamond S$

~~(skip)~~

S ~~is~~ makes an artificially strong assumption about one process never getting suspected.

~~They get a recon~~

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

The protocol is quite a bit harder + tolerates $< \frac{n}{2}$ failures.

Turns out, for reasons similar to the impos. pf 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).

Executes asynch ~~rds~~ phases 0, 1, 2, ...

Each ~~round has~~ phase has an "owner" (coordinator), rotating round robin thru processes, mod n.

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

⋮

skip 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 procs + chooses a value consistent with the latest one it hears about.

Intersection of majorities is needed to prevent disagreement.

LTTR. (DLS, Paxos)

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 §25.6.3 of my book. (last §)

Main difference is explicit use of failure detector.

FDs are nice abstractions.

My favorite is $\Diamond P$ – allows talking about arbitrary suspicions up to any point, then see what happens when things "behave well" from some point onward.

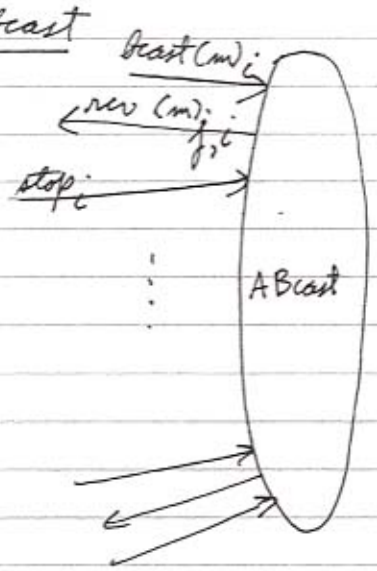
Other contributions of paper:

$\frac{n}{2}$ failure impos. 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 Bcast + its "equivalence" with consensus:

abcast



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

order:
If 2 msgs received at 2 locs, same order at both.

self-delivery:
(In fair exec), any msg bcast 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 nonf. procs.

Easy to guarantee without the order property:
Just make sure ~~to deliver upon~~ that before you deliver any 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-FT solution to ABcast (where "in fair exec" is replaced by "in fair exec with ≤1 failure") would yield a 1-FT consensus alg.~~

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 a trace~~

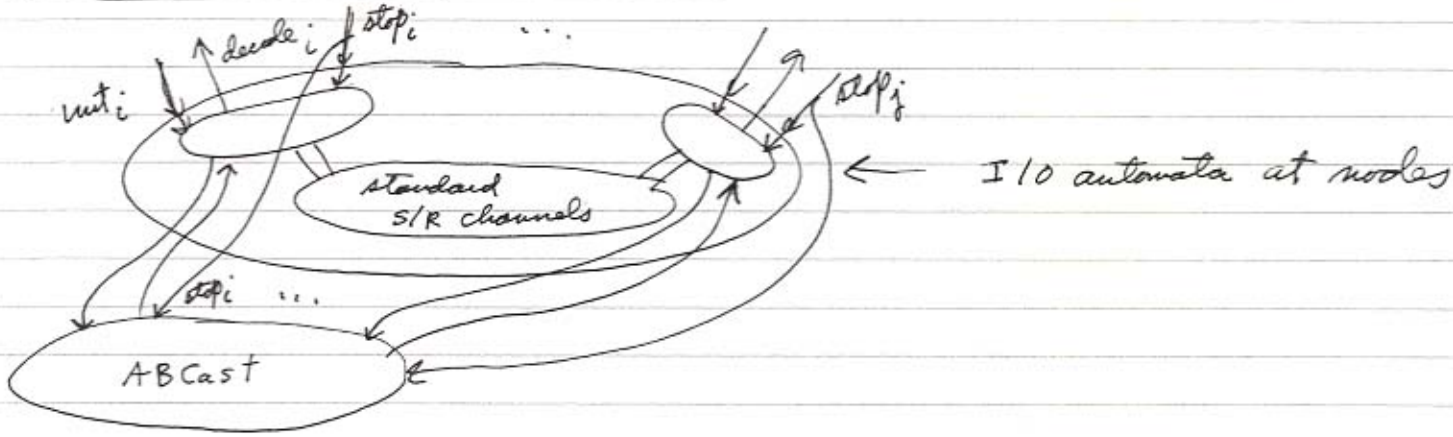
Correctness for an execution α:

$\exists S$, a (finite or infinite) sequence of messages such that:
- S contains only msgs that someone has bcast, 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 α is fair, then every msg bcast on a non-failing port is in S
- " " " " " every msg in S delivered 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 liveness 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 asynch model for $f \geq 1$, this is asserting equivalence between 2 unimplementable problems, tricky.

Solution to f-FT ABCast ⇒ Solution to f-FT consensus



Consensus alg: (Doesn't even use the S/R channels)
Beast 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 every 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.

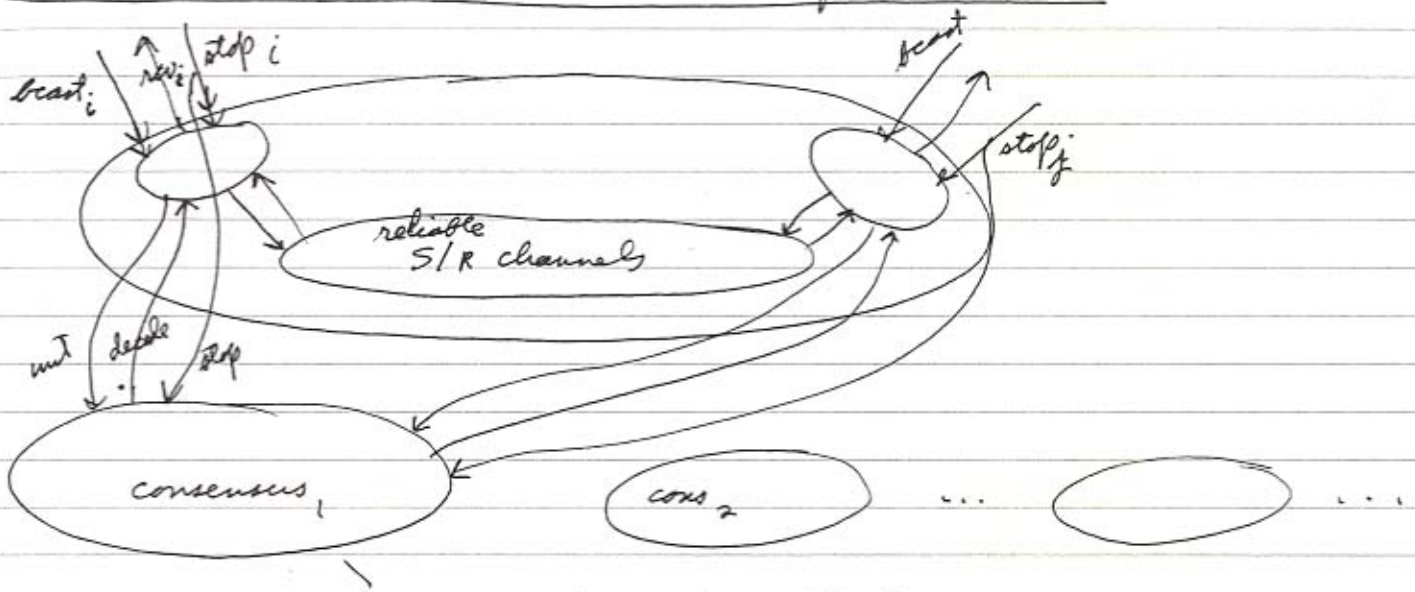
It eventually beasts its value.
Since exec of α has ≤ f failures, the msg ~~gets put in S~~ must be included in S, + must be delivered to i.
So i eventually decides

So i gets some first message.

2 liveness properties

Corollary: Can't solve 1-FT ABCast in asynch S/R dist. system

Soln to f-FT consensus ⇒ Soln to f-FT ABcast



~~actually not - using 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 new (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 parts.

Then at most f failures for each consensus, so each returns.

~~Consider any msg~~

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

Remains to argue each msg sent on non-f part ~~eventually winds~~ is actually in S, that is, it eventually winds up in some consensus decision set. :

~~Th~~ Say m sent on ABCast 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.

$\boxtimes$

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

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

E.g. FIFO: If process i sends m before m' + ~~both~~ m' appears in S, then so does m, and m precedes m' in S.

E.g. causal: Define causality order based on extend events ~~of~~ :
~~ABCast~~: - all events at sae location
- beast is deliver (anywhere) of sae msg.
- Take ~~tran~~ transitive closure.

Then say if ~~m~~ ~~send~~ beast(m) causally precedes beast(m') ~~then m~~, ~~+ m & m' both in S then~~ m' in S, then so is m, + m precedes m' in S.

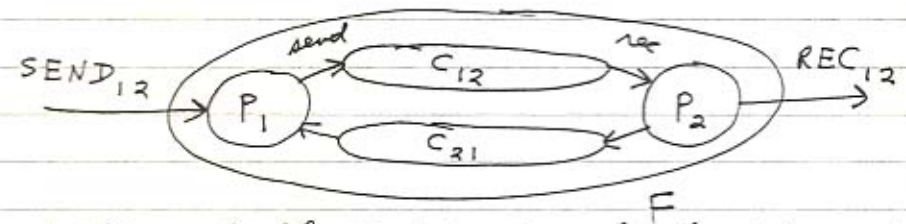
And other variants.

Last topic: Reliable Comm from Unreliable Channels

FIFO reliable channels

less reliable – allow loss, dup, reo
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 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 procs P_1 + P_2.

Want fair traces of composition (hiding low level sends + receives) $\subseteq$ fairtraces(F)

Stenning's Protocol

Assume Cs can lose, dup + reorder

However,

1. Can't lose everything

Specifically, if ∞ many sends then ∞ many receives WLL
(more precisely, receives for ∞ many of the sends).

2. 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 Stenning using bdd tags, in fact, just 0 & 1.

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

Used in formal verif. community, not a practical protocol.

Needs stronger assumptions on C channels, e.g., cap. + reader msgs., finite dup, a WLL
weak loss limitation property
(∞ msg sends $\Rightarrow$ ∞ msg of them are received)

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

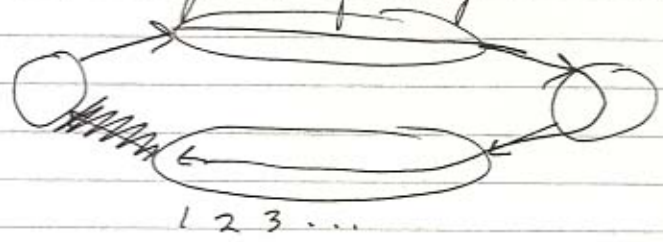
P_2 accepts first msg it receives with tag = 1, then accepts later msgs if tag ≠ tag of previous accepted msg.
Puts acc msg in buffer for delivery to user, delivers, acks with tags.

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

Code p. 698-699

Pf: Safety: Relate formally to Stenning
To do this, consider Stenning with the stronger FIFO duplicating channels.
For this case, get new inv about the order in which tags appear in the channels + processes:

Let T = sequence of integers consisting of $queue_{21}$, tag_2, $queue_{12}$, tag_1



Then T is nondecreasing + difference ≤ 1.

Sim reln from ABP to Stenning: (this new S.)
Everything same, except tag in ABP is low-order bit of tag in Stenning
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$(m, b)_{1,2}$, where m is accepted by P_2
Then $b \neq$ ABP.tag_2
Means in Stenning, $(k \neq$ Sten.$tag_2)$ mod 2, by corresp.
k: integer tag of incoming msg

To show accepted in Sten, need $k =$ Sten.$tag_2 + 1$

Use Lemma (inv) above: Incoming tag has different parity. with inv, can only be $tag_2 + 1$.

Similar args used for $\begin{cases} m \text{ not accepted by } P_2 \\ + \text{ for } P_1 \end{cases}$

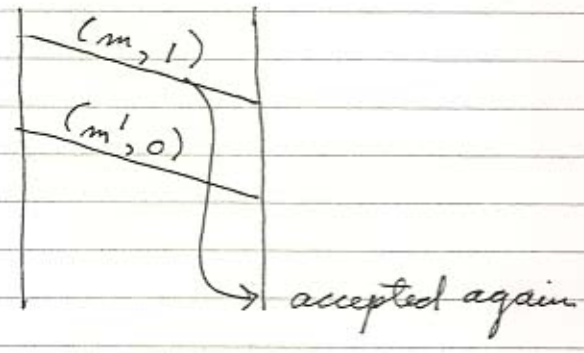
<u>Liveness</u>: LTTR. Can show directly, or via a kind of liveness-preserving sim reln.
(based on the strong 1-1 corresp between actions)

<u>Moral</u>: With unbdd tags, can tolerate all failure types
With bdd tags, can tolerate loss + dup, not reo.

Finish by considering <u>protocols that tolerate reordering</u>.

<u>Q</u>: What exactly ~~goes~~ goes wrong with ABP when used with reordering channels?

<u>A</u>: P_2 can be fooled into accepting an <u>old high level msg m</u> that happens to arrive tagged with same bit as the one currently expected
Can cause duplicate delivery.



<u>Q</u>: Might there be other <u>bdd-tag protocols</u> that do tolerate reordering?

<u>A</u>: <u>3 results</u>:

1. <u>Nonexistence</u> of bdd-tag protocols tolerating <u>reo + dup</u>
2. Bdd tag protocol tolerating <u>loss + reo, not dup</u>
   (not practical, high complexity)
3. <u>Nonexistence of efficient protocols</u> tolerating <u>loss + reo</u>.
   in terms of # of low-level msgs.

Thus, the high complexity is unavoidable.

<u>Bounded tag</u>: Formalize by saying M + M' <u>both finite</u>
hi level msg alph — low level

1. Impossibility for reo + dup

Assume there is a protocol (P_1, P_2) using bdd tags (finite M') to implement F (FIFO reliable) over channels Q_{12} + Q_{21}.

Qs defined to admit $\begin{cases} \text{arbitrary reo} \\ \text{no losses} \\ \text{finite (unbdd) dup} \end{cases}$

Show contradiction:

Run the system until P_1 sends a low-level msg m_1

Continue until P_1 " " different low-level msg m_2

⋮

continue until no longer possible.

Doing this may involve SEND inputs; include them as needed.

Get α_1.

~~Say the a~~

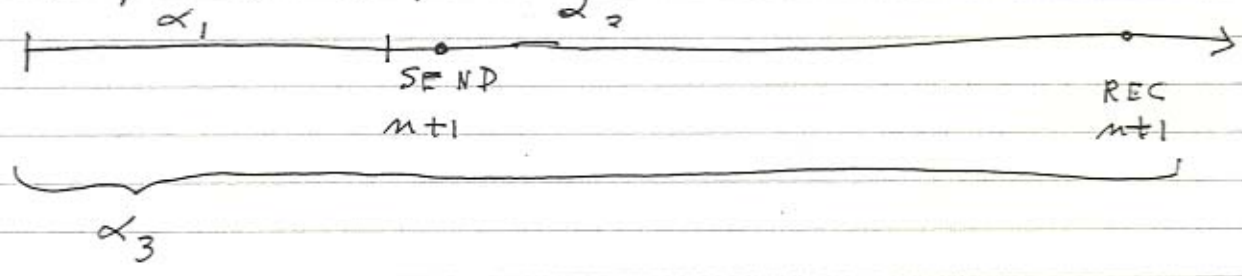
Let n = # of SEND events in α_1.

Now extend α_1 with one more SEND + continue fairly, with no further SEND inputs.

Get α_2.

By def of F, α_2 contains exactly $n+1$ RECEIVE events

Let α_3 = part of α_2 up to last REC.



Now construct alternative exec α_4 $\begin{cases} \text{extending } \alpha_1 \\ \text{looks like } \alpha_1 \text{ to } P_1 \\ \text{looks like } \alpha_3 \text{ to } P_2 \end{cases}$

That would give n SENDs + $n+1$ RECs, contrad.

<u>How to construct</u>:

Stop all events of P_1 after α_1 + let P_2 proceed as in α_3.

<u>Why is it possible</u> for P_2 to do all the same things as in α_3?

Only issue is <u>low-level receives</u> – how do we know we can get these without P_1's help?

Possible because all low-level msgs P_1 <u>sends after α_1</u> contain ~~msg~~ ms that P_1 has already <u>sent in α_α</u>

So can use dup + reo to allow P_2 to receive them again.

☒

2. Bdd tag alg tolerating loss + reo

Now assume no duplicates

Possible to implement F, with loss + reo, with bdd tags.

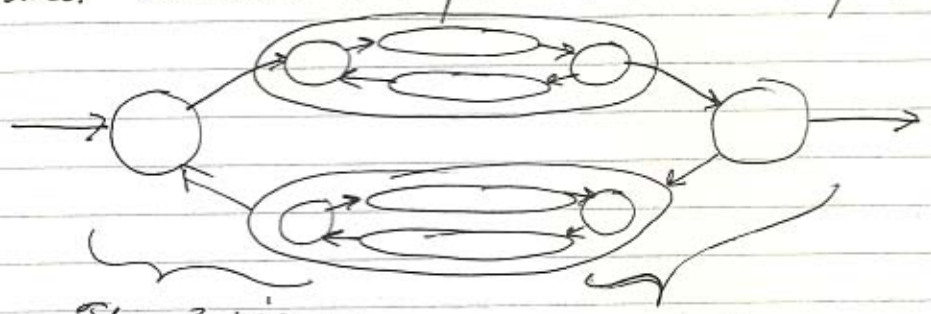
Probe Alg:

Not practical – arose as a counterexample to an impossibility result.

2 layers:

1. Use given channels to implement channels that don't reorder, but can lose + dup (WLL)
2. Use these interwed channels to implement F (using ABP)

Compose layers:



These 3 pieces compose to give P_1 of final algorithm.

node P_2

Multiplex channels: Need to be fair to both, so actually need a stronger loss-limit property

E.g: "send m ∞ many times ⇒ deliver m ∞ many times, for each m"

Layer 1: Impl. non-reo (but can loss – WLL – or dup) using given non-dup (but can lose ~~SL actually~~ WLL enough here – or dup)

Receiver-driven:

P_1 sends msg only in response to explicit probe msg from P_2.

Response always has value of latest hi-level msg received from U, (latest) (OK to lose some)

(unanswered keeps track of probes to answer)

P_2 sends probes continually, counting in pending the total number it's ever 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's recent – a copy was sent by P_1 after last delivery.

$\boxed{\text{code p. 708-709}}$

can use to show no res occurs.

Liveness:

Needs WLL – if ∞ many SENDs, then ∞ many of these have corresp. RECs.

Suppose ∞ many SENDs – then P_2 keeps probing, P_1 keeps answering, channels keep delivering something.

So ∞ many RECs.

Corresp. to ∞ many SENDs?

Because each msg corresponds to latest, at some point after previous REC, + inf many SENDs occur

Then the RECs must corresp to ∞ many SENDs.

Complete protocol: Combine as above

(Need SLL 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?

Liveness: Needs WLL - if ∞ many SENDs, then ∞ many of these have corresp. RECs:

Suppose ∞ many SENDs - then P_2 keeps probing, P_1 keeps answering, channels keep delivering (something - WLL).

so inf. many RECs.

Corresp. to ∞ many SENDs?

Because each msg corresponds to latest, at some point after previous REC, + inf many SENDs, get corresp with ∞ many different SENDs.

Complete protocol: Combine as above
(Need SLL 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 Tolerating Loss + Reo.

Assume underlying channels $\left\{\begin{array}{l}\text{don't duplicate}\\ \text{can lose, subject to SLL}\\ \text{can reorder}\end{array}\right\}$ same situation as just considering

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

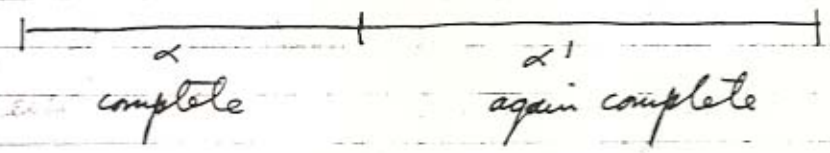
Q: How to measure cost?

A: Paraphrase: protocol is k-msg-bdd if, from any point in the exec, it's possible to deliver a new msg with $\leq k$ rec_{12} events.

More precisely:

"From any point": Consider points after "complete" finite execs, where # SENDs = # RECs.

"It's possible to deliver": $\exists \alpha'$

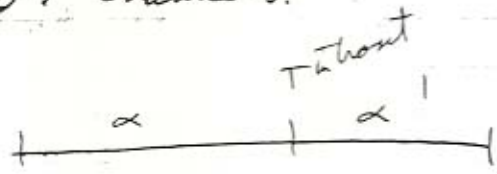


- The portion of α' after α has 1 more SEND + 1 more REC (exactly)
- # of rec_{12} events in α' after α is $\leq k$
- ~~[illegible]~~ All low-level msgs rec. by P_2 in α' after α are also sent after α.

(technical)

(Best-case $\leq k$)

<u>Theorem</u>: $\nexists$ k-bdd protocol (P_1, P_2) (for any k) implementing F using lossy (SLL) reo (non-dup) channels.



<u>Pf</u>: Suppose $\exists$, fix such k.
Suppose we could produce $\begin{cases} \text{multiset } T \text{ of elements of } M', \\ \text{complete exec. } \alpha \\ + \text{ k-extension } \alpha' \end{cases}$ satisfying:

1. <u>All msgs in T are "in-transit"</u> (sent but not received) from P_1 to P_2 after α.
2. <u>Multiset of low-level msgs received by P_2 in α' after α is ~~submultiset of~~ submultiset of T.</u>

<u>Then get contrad</u>: Produce alternative exec. α, extending α, looks like $\begin{cases} \alpha \text{ to } P_1 \\ \alpha' \text{ to } P_2 \end{cases}$

P_2 can receive needed new msgs from those already in transit. Has more RECs than SENDs, contrad.

So, want to produce this bad situation

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

More formally:

Claim: Suppose $\begin{cases} \alpha \text{ is complete exec.} \\ T \text{ mset of low-level msgs in transit } 1 \to 2 \text{ after } \alpha, \\ \quad \text{where } T \text{ contains} \le k \text{ copies of each element} \end{cases}$

Then at least one of these holds:

1. $\exists$ k-ext α' of α s.t. the mset of low-level msgs rec by P_2 in α' after α is a submset of T

or 2. $\exists$ complete extension α' of α, new mset T' of low-level msgs in transit after α', where T' contains $\le k$ copies of each elt,

+ $T \subset T'$

proper submset.

If claim true: Create the bad situation by working inductively

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

$T_0, T_1, \ldots$ msets in transit

α_0 initial states, T_0 empty

If Case 1 holds - done

else Case 2 - produce α_1, T_1 + continue. 1+1

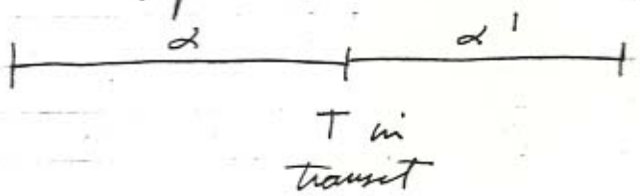
- If ~~claim true~~ Case 1 ever holds, done

Else go on forever.

But impossible, since $\begin{cases} \text{fin many msgs in } M' \\ \text{at most } k \text{ copies allowed, of each} \end{cases}$

Prove Claim

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



If set of msgs received by P_2 in α' after α $\subseteq T$ then Case 1 satisfied.

So assume not. Then $\exists\, p \in M'$ s.t. # of new $rcv(p)_{12}$ events > number of copies of p in T.

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

Then T' still has $\leq k$ copies of each (since α' is a k-extension so # of $rcv(p)$ events $\leq k$)

Must get complete ext of α having T' in transit.

(Not in transit after α', since the new p is received.)

Technical: $\exists$ send(p) in α' after α, because msgs rcvd after α assumed sent after α.

α_1 = part of α' ending with first such send(p)

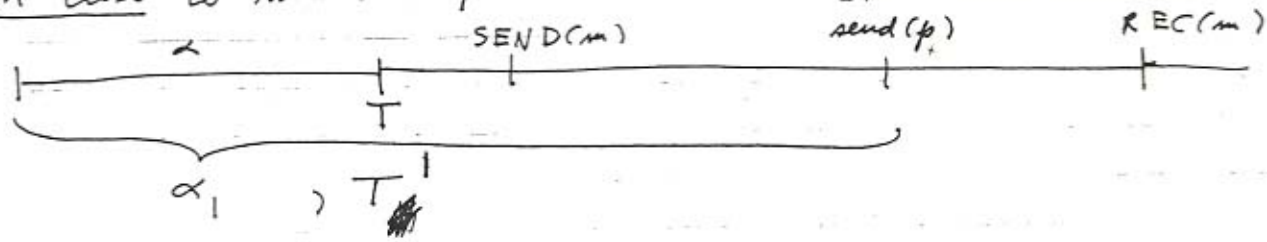
T' in transit after α_1

We need α_1 to be complete, though.

Recall α' after α contains SEND + REC

If α_1 includes both or neither, then α_1 satisfies Case 2.

Rem case is where α_1 contains the SEND but not the REC:



Then add some stuff to α_1 to deliver m (REC) but without delivering the low-level msg p to P_2

That is, "lose" p, then run fairly, with no new SENDs, + all rec_{12}'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(m) occurs. Stop there.

Tolerating Crashes

Settled all questions of impl. rel. using unrel. channels, if procs are reliable.

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

Instead, consider crash-recover 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.

~~So now, consider the case where~~

Results: 1. Impossible of F using crashing procs
2. Weaken problem reqs, still impossible
3. a practical protocol – just mention, LTTR

special case: assume crashes lose all info, recovery → arbitrary start state