Last time:

Various consensus problems:
- voting as arbitrary BA
- connectivity loss for BA
- $k$-agreement
- weak agreement

Today:

One more consensus problem: distributed commit.

Then: asynchronous systems, starting with models.
Skip approx. agreement this time

Summary: Get different kinds of results if we don't require exact agreement, but only approximate.

Each process starts with real number, must decide on real, subject to agreement: values within ε decision.

validity: decisions in range of central values of approx. terms.

can be approx. range (or could be approx. range).

Can send reals in msgs.

Can solve, both for stopping + Byz. case.

Can do by solving exactly.

But this problem admits different strategies also, using rounds where results "converge" (get closer + closer the more rounds that are executed).

- Can be simpler, cheaper than exact agreement.
- Extends to async setting whereas, as we'll soon see, exact agreement strategies do not.

Motivation: Clock synchronization, in presence of failures.
The commit problem:

I'll describe some basic, standard algorithms for the problem of distributed database transaction commit.

Situation: all transactions at multiple sites in a distributed system, and at each node, wind up with a "commit" or "abort" decision.

Based on whether there are no failures the work for that transaction has been successfully completed at that site. (results made stable)

Assume 3 processes failings only:

- stopping only
- n-nodes undirected, complete graph

Agreement: no 2 processes decide differently (uniformity)

Validity: 1. if any process starts with 0 then 0 is the only possible decision value

2. if all start with 1 and no failures then 1 is only possible decision value.

Asymmetry: to commit, everyone needs to say things are OK. Only guarantee must commit if all want to commit initially + maybe all get them (best case; second case)

Termination:

- weak version: if there are no failures then all processes eventually decide
- strong version: (non-blocking cond) all non-faulty processes eventually decide
2P commit

Standard algorithm, blocking (weak termination only).
Assumes distinguished p₁ acts as “coordinator”

 dân ia

Decides consistently with the validity rules

\begin{align*}
\text{List (Serial Alg.)} \\
\text{When can they decide?} \\
\text{Anyone whose initial value is } 0 \text{ decides } 0 \text{ at some later time.} \\
p₁ \text{ decides } 1 \text{ after getting } \text{all } \text{ of } 1 \text{ msg(s) consistent with validity cond.} \\
\text{Others decide} \\
\text{if sees a } 0 \text{ or doesn't receive a msg from someone, decides } 0\text{, else } 1.
\end{align*}

Everyone else decides after not 2, based on what they hear from p₁.

Agreement (because decision is centralized)

Validity: because of how the coordinator decides

Weak termination: \checkmark

But not strong termination: if p₁ fails before sending all its not 2 msg(s), others won't terminate.
In practice, add termination protocol, where when p1 fails & others detect & try to finish agreeing on their own.

But this can't always work:

\[ \text{Ex: if } 0 \quad \text{init values} \]

+ p1 fails at beg. validity says others should decide 0.

But what if p1 actually received all the 1s, had a 1 itself, & decided 1 just before failing?

Then 0 would be a disagreement (violates uniform agreement).

Complexity: 2 ids

\[ \leq 2m \text{ msg} \]

3-phase commit

Beef up 2PC to get strong termination.

Such is to introduce an extra stage of agreement, before actually deciding.

2PC: process states classified into 4 categories:

- **dec0**: decided 0
- **dec1**: decided 1
- **ready**: ready to decide 1 but haven't yet
- **uncertain**

Again, same graph (initial conditions...
rd 1: All other processes send their init values to \( f_1 \).
   All with init value 0 decide 0 \( \rightarrow \text{dec} 0 \).
   If \( f_1 \) gets all 1's + has a 1 itself, then becomes ready.
   If sees 0 or doesn't hear from someone, \( f_1 \) decides 0 \( \rightarrow \text{dec} 0 \).

rd 2: If \( f_1 \) has decided 0, \text{bests } \text{"decide(0)"}.
   Else \text{bests } \text{"ready"}.
   Anyone else who receives \text{dec} 0 decides 0 \( \rightarrow \text{dec} 0 \).
   Anyone else who receives \text{ready} becomes \text{ready} \( \rightarrow \text{ready} \).
   After this rd, \( f_1 \) decides 1 if it hasn't already decided.

rd 3: If \( f_1 \) has decided 1, \text{bests } \text{"decide 1"}.
   Any other process that receives \text{dec} 1 decides 1.


Correctness:

Key is that certain states cannot coexist.

Invariant: (true after \( 0, 1, 2 \) or 3 rd)

1. If any process is in \text{ready} or \text{dec} 1, then all processes' init values are 1.

2. If any process is in \text{dec} 0, then no process is in \text{dec} 1
   or non-failed process is ready.

3. If any process is in \text{dec} 1, then no process is in \text{dec} 0
   or non-failed process is uncertain.

Correctness possibilities:

\text{dec} 0 \rightarrow \text{unc} \rightarrow \text{ready} \rightarrow \text{dec} 1

But fewer possibilities for non-failed.
Pf: LTTR

Step 1: When \( p_1 \) decides 1, check 3rd cond. holds. It's the end of rd 2. \( p_1 \) knows that every other process has either received "ready" or has failed.

(Synchronization critical here - else would have to wait, etc.)

Now agreement + validity hold w.r.t. these 3 roles. But strong termination doesn't yet follow.

E.g. if \( p_1 \) fails, can leave others with initial val = 1 undecided after rd 3.

For this need to add more rounds, usually called "termination protocol"

Amounts to giving others a chance to act as coordinator.

E.g., first \( p_2 \), then \( p_3 \), ... \( p_n \)

rd 4: All send their current status \((D_0, D_1, R, V)\) to \( p_2 \)

If \( p_2 \) sees any \( D_0 \) values it hasn't already decided, decides 0.

If all \( D_1 \) values, \( p_2 \) decides 1.

If all the known values are uncertain, then \( p_2 \) decides 0.

Otherwise (all known values in \( E \), \( \text{ready} = 3 \)) + at least one = \( \text{ready} \), \( p_2 \) becomes \( \text{ready} \), but doesn't decide yet.
Rds 5+6: Are like Rds 2+3, but with $p_2$ acting as the coordinator.

Rd 5: If $p_2$ has (ever) decided, lets decide ($p_2$ may
whatever it is
else lets ready.

Any undecided process that receives decide ($p_2$ may decides accordingly.

Any process that receives ready becomes ready.

Then $p_2$ decides if hasn't already decided.

Rd 6: As for rd 3.

Can show its still preserved.

So got agreement & validity
But now also strong termination.

Complexity: 
Rds: high, 3n worst case
msgs: $O(n^2)$ worst case

(only 3 n) "normal case"
(0cn) normal

Practical issues with 3PC:

Depends on strong assumptions, which are hard to guarantee in practice.

async model: But OK since could emulate with timeout and timeouts
reliable msg delivery: Can retrofit, though must be careful.

if it takes too long, sync model not accurately simulated.

accurate failure diagnosis: Algorithm lets process conclude
that, if they haven't heard from someone, it must have failed.
But hard to guarantee in practice.

Other consensus algo can be used for commit - ones that aren't so dependent on reliability & timing guarantees.

Paxos consensus algorithm is one.

Tolerates many kinds of failures.

Another algorithm where all processes try to act as coordinator (sort out conflicts using leader election subalgo).

But now a decision of 0 or 1 depends on active support by a majority of the processes

rather than just assuming something about what must be true because you didn't hear from someone.
Async Systems

Now switch to async model
Process no longer proceed in rounds, but can go at arbitrary relative speeds

Will consider two kinds of async models in this course:
- async networks: consisting of nodes + channels
- async store in memory: where processes communicate via shared variables or shared objects

Usually I've started with the ASM.
But this time I'll try something new, going directly to async networks.

Corresponds closer to the sync networks we've been doing so far.
We'll come back to ASM systems.
(They form useful abstractions for programming networks.)

The basic structure will consist of processes communicating via channels.

\[ \text{E.g.} \]

Mathematically, what are the ps and the Cs?

Reactive components, modules that interact with their environments
via input and output actions

\[ \text{E.g.} \] \text{invocations + responses, sends + receives} \]

[Diagram of process interaction with channels]

[Diagram of process interaction with channels]
In any real memory systems, we'll also see reactive components, e.g.

![Diagram]

Pens have client I/O, plus use + response on objects.

Can share some (perhaps with restrictions).

In both cases, we need to talk about reactive components of systems.

Let's start with a general model for reactive components:

**π/0 Automata**

A very general math model for reactive components.

Has very little structure.

Must specialize it, add structure for particular types of reactive systems - networks, I/O systems, etc.

Designed for describing systems in a "structured" way:

- Allows description of system components and how they compose to yield a larger system.
- Also allows description of systems at different "levels of abstraction," e.g., detailed implementation, more abstract algorithm description,
  - Unoptimized form
  - Optimized algorithm is more efficient, simpler, etc. (as you've seen before)
Supports good proof methods: invariants, compositional methods, simulation relations (like running algo side-by-side).

8.1 I/O Automata

A consists of:

- $\text{sig}(A)$: a signature, which specifies the input, output, and internal actions ($\text{acts}(A)$, etc.)
- $\text{states}(A)$: not necessarily finite
- $\text{start}(A)$: a states($A$): (arbitrary)
- $\text{trans}(A) \subseteq \text{states}(A) \times \text{acts}(A) \times \text{states}(A)$
- $\text{tasks}(A)$, an equiv. rel on local($A$) with at most countably many equivalence classes

One reduction:

Requires input-enabling: any input enabled in any state.

Remarks:

- $\text{sig}(A)$: Clarification. I/O/Int explain intent

I/O external
out & int locally controlled (under control of component)
input under control of environment, could happen at any time

- External actions used

External actions used as basis for specifying externally-visible behavior of system components.

Input-enabling: Discuss

Not too strong for basic model.

If we want restrictions, can model environment (as another automaton) + put restrictions on it.

Component might sometimes be required to detect bad inputs, or
might exhibit unconstrained behavior in face of inputs - can describe either.

states: Not like FSM, can be infinite (just as in synchronous model)
trans: labeled by actions
tasks: Think of these as threads of control.

Group actions together that are part of the same thread.

Useful for modeling components that perform >1 job.

In the formal theory, need to talk about fairness (give four turns to all tasks, to take steps).

Needed for stating liveness, progress properties; reasoning about whether a system satisfies them.

Examples:

Channel automaton

Reliable FIFO channel, unidirectional, between 2 processes

(only one type, could model any type also)

\[ \text{send}(m) \xrightarrow{} C \xrightarrow{} \text{recv}(m) \]

3 \in M, msg alphabet.

signature: input: \{ send(m): m \in M \}
outputs: \{ recv(m): m \in M \}

states: Describe using rules, as before.

Here, queue is a FIFO queue of M, init \lambda.

trans: Described by transition rules - little code fragments.

Each rule describes a set of transitions for designated actions (group by type, usually)
sent(m) 
\[ \text{effect: add m to queue}\] 
\[ \text{rev(m)}\] 
\[ \text{pre: m = head(queue)}\] 
\[ \text{effect: remove head of queue}\] 

\[ \text{tasks: put all rev actions into 1 task}\] 

Another ex given in Ch. 8, for a process automaton for some kind of consensus system: \[\text{(non-FT)}\] 

\[\text{init} \rightarrow \text{decide}\] 
\[\text{send} \rightarrow \text{receive}\] 

\[\text{As seen: collects all values - its own by init + others by msgs.}\] 
\[\text{When has a value for all, outputs a value obtained by applying a standard function f.}\] 
\[\text{(Can get new inputs, can change value, can send & output repeatedly)}\] 
\[\text{Tasks: separate ones for sending to each node for deciding}\] 

How does an FTA execute? 
Describe with a formal notion of an execution. 
No history for the moment, so ignore tasks.

\[\text{execution = } s_0, t_1, s_1, \ldots\] 
\[\text{finite or infinite}\] 
\[\text{Each } (s_c, t_{c+1}, s_{c+1})\] 
\[\text{is a transition}\]
reachable states: state occurring in some exec.
(equiv. at end of some finite exec)

exec fragment: same but so needn't be a start state.

Ex: channel

\lambda, \text{send}(a), a, \text{send}(b), ab, \text{rev}(a), b, \text{rev}(b), \lambda

states

Any prefix, e.g., stop

Stall an exec. no fairness req

traces: Often interested in external only

Esp. important when building up system using components.

\text{trace}(a) = \text{restriction of } a \text{ to external acts only}

(no state) no internal acts

Ex: \text{send}(a), \text{send}(b), \text{rev}(a), \text{rev}(b)
8.2 Operations on Automata

To build system out of components, model needs formal "building" ops.

Composition (hiding, renaming)

Composition: Simply match up same action in different components.
All components having that action perform steps involving that action at the same time. (synchronize on actions)

Compatibility: Composing \( A_i \mid i \in I \) is countable (finite a ref.)

Need \( \text{int}(A_i) \cap \text{acts}(A_j) = \emptyset \)

Internal acts not seen or shared by anyone else.

Also, \( \text{out}(A_i) \cap \text{out}(A_j) = \emptyset \)

Only one out. controls each output action.

But can have output of one = input of 1 (or more) others.

Allow as many to be composed.

Model used for abstract, logical systems, not just physical systems.
Including dynamically-created components (e.g. from systems).

But this necessitates another compatibility restriction:

No action is contained in as many sets acts \( (A_i) \)

Now define composition \( A = \Pi A_i \) assuming the \( A_i \) are compatible.

\[ \text{deg}(A) : \text{out}(A) = \bigcup_{i \in I} \text{out}(A_i) \]

\[ \text{int}(A) = \bigcup_{i \in I} \text{int}(A_i) \]

\[ \text{in}(A) = \bigcup_{i \in I} \text{in}(A_i) - \text{out}(A) \]
state \((A) = \prod_{i \in I} \text{state} \((A_i)\) \quad \text{Cartesian proc.} \)

\[\text{start} \((A) = \prod_{i \in I} \text{start} \((A_i)\)\]

\[\text{trans} \((A) : (s, \tau, s') \in \text{trans} \((A_i)\) \quad \text{such that for all } i, \text{ if } \tau \in \text{act} \((A_i)\) \]

\[\quad \Rightarrow (s_i, \tau, s_i') \in \text{trans} \((A_i)\) \quad \text{projection} \]

\[\quad \text{else } s_i = s_i'\]

\[\text{tasks} \((A) = \bigcup_{i \in I} \text{tasks} \((A_i)\) \quad \text{Maintain all the separate, again classes (threads)}\]

\[\text{later, when we consider fairness/determinacy, we'll see this gives the right meaning: concurrent components in a system all keep taking steps.}\]

\textbf{ex:} Composing channels & concurrent processes, as above.

\[\text{require first }\]

\[\text{must make sure to disambiguate actions, e.g. } \text{send} \((m)\) \quad \text{by i, j}\]

\[\text{Subscripts on processes, channels, actions.}\]

\[\text{(Also, C, S, M must include) }\]

\[\text{also the lanes P, }\]

\[\text{ends.}\]

\[\text{Then get a single I/OA representing the composed system (figs 1 of 3) }\]

\[\text{inputs: } \text{init} \((v)\) \quad \text{for all } i\]

\[\text{outputs: } \text{decode} \((v)\) \quad \text{if finally, still outputs, in} \]

\[\text{send} \((v)\) \quad \text{combined system} \]

\[\text{recv} \((v)\) \quad \text{internal: none}\]