CPSC 465/565 Theory of Distributed Systems

James Aspnes

2023-08-30

Today's exciting topics

- Course overview
- Distributed computing
- Message passing model
- Safety, liveness, and fairness

Course information

- Instructor: James Aspnes
- Teaching Fellows: John Lazarsfeld and Weijie Wang
- Course notes:
  https://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf
  - Also include lecture schedule, assignments, etc.
  - May change often.
- Coursework:
  - 5 assignments (100% of grade in 465, 85% in 565)
  - Presentation (565 only, 15% of grade)

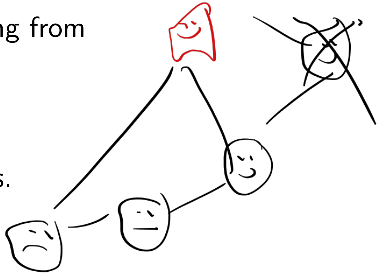
Assignments will mostly involve proving things (it's a theory course). We will not be doing any programming or implementations.

Distributed systems

"A distributed system is one where the failure of a computer you didn't even know existed can render your own computer unusable."
—Leslie Lamport

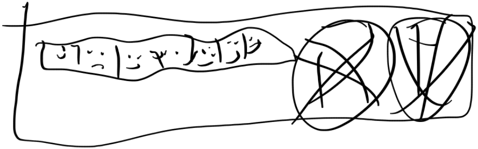
- Many processes
- Lots of **nondeterminism**, arising from
  - Message delays
  - Unpredictable scheduling
  - Failures

Do not confuse with parallel systems.



Parallel systems are friendly

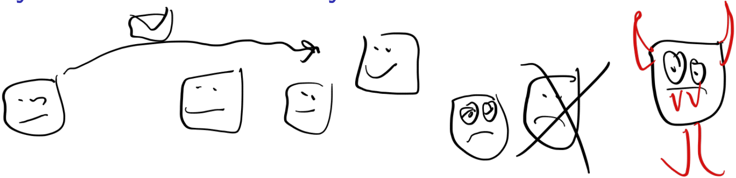
RTX
1200000G



- Example: graphics cards
- Carefully synchronized processing units
- Predictable timing
- No failures

More processors = more power!

Distributed systems are unfriendly



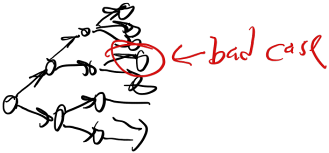
- Example: anything running over a network.
- Uncoordinated pile of machines
- Unpredictable timing
- Failures are normal and expected

More processes = more problems!

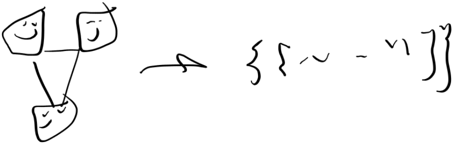
Why theory?

Mathematical modeling + proofs are tools for dealing with nondeterminism.

- Exponentially many possible executions
- Not repeatable
- Bugs show up with low probability

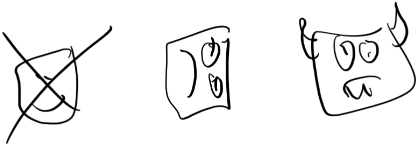


Reasoning about systems

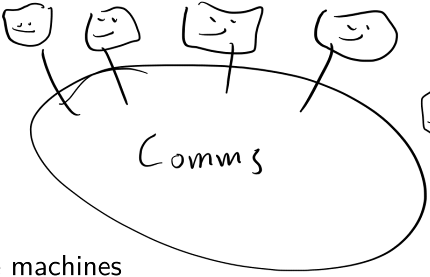


- Model systems as mathematical objects
- Prove correctness
- Or prove impossibility

$\forall$ executions; it works



Models

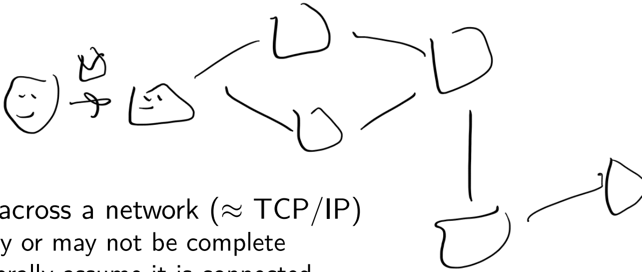


Distributed system =

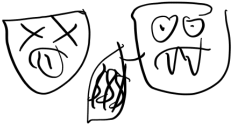
- Collection of state machines
- Communications mechanism
  - Message passing: send packets across a network
  - Shared memory: read and write common address space
  - (or more exotic systems)

We will start by looking at message passing.

Message passing

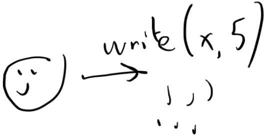


- Send messages across a network (≈ TCP/IP)
  - Network may or may not be complete
  - We will generally assume it is connected
- Nondeterminism:
  - Message delays
  - Message loss
  - Process failures:
    - Crash failures — process stops working
    - Byzantine failures — process turns evil!

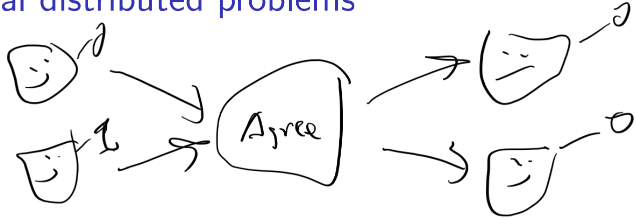


Shared memory

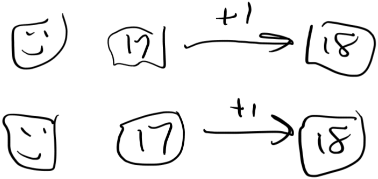
- Read and write shared objects
  - Often read/write registers (atomic registers)
  - Sometimes more powerful objects:
    - Test-and-set
    - Compare-and-swap
- Nondeterminism:
  - Asynchronous operations
  - Process failures (usually just crash failures)



Typical distributed problems



- Agreement
- Replicated state machines
- Simulations, e.g. shared memory from message passing

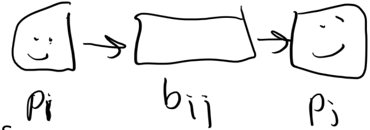


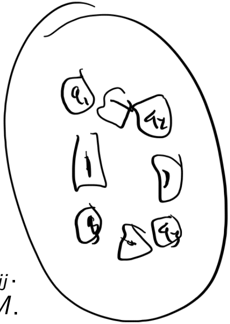
"The standard asynchronous model"

- Many historical candidates (c. 1970-1985 or so)
  - I/O automata
  - Temporal Logic of Actions
  - Communicating Sequential processes
  - π calculus
  - pomsets
  - etc.
- Ultimately pretty much the same
- Compromise position:
  - I say I am using the standard model.
  - You assume I am using your favorite model.

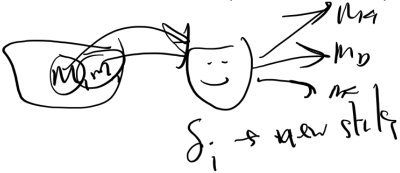
We will use a model adapted from (Attiya and Welch, 2004).

Message passing model

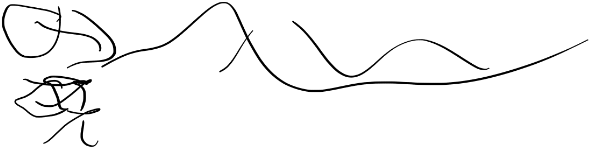




- Processes
  - Each process i has state in some state space Q.
- Buffers
  - Each pair of (connected) processes has buffer b_{ij}.
  - Buffer holds set of undelivered messages from M.
- Configuration = process states + buffer states
  - Formally, element of $Q^n \times P(M)^{n \times n}$.
- Transition function:
  - $\delta_i : Q \times P(M \times [n]) \to Q \times P(M \times [n])$
    - Old state $\times$ messages delivered (with senders)
    - $\to$ new state $\times$ messages sent (with recipients)



Events



- Delivery event $\text{del}(i, S)$ = deliver messages in S to i.
- Removes 0 or more messages from buffers.
- Applies transition function δ:
  - Updates state of recipient
  - Adds 0 or more messages to recipient's outgoing buffers

Execution

$$C_0 \xrightarrow{\mathrm{del}(i,\emptyset)} C_1 \xrightarrow{\mathrm{del}} C_2 \rightarrow$$

A **execution** is an alternating sequence of configurations C_t and events α_t:

$$C_0 \alpha_1 C_1 \alpha_2 C_2 \alpha_3 C_3$$

- May be finite (ending in a configuration) or infinite.
- Constraints:
  - Each α_{t+1} is **enabled** in C_t:
    - Message passing: $\mathrm{del}(i, S)$ enabled if $S \subseteq \cup_j b_{tj}$.
    - Can only deliver messages that exist.
    - Other models will have different conditions.
  - C_{t+1} is the result of applying α_{t+1} to C_t

$$C\alpha\beta\gamma = C'$$

$$C_t \alpha_{t+1} = C_{t+1}$$

Implicit assumptions in the message-passing model

- Only point-to-point messages
  - I can send messages to multiple processes at the same time.
  - But no guarantee they are delivered at the same time.
- Worse: "same time" only makes sense at one process:

$$C_0 \text{del}(p_1, S_1) C_1 \text{del}(p_2, S_2) C_2$$

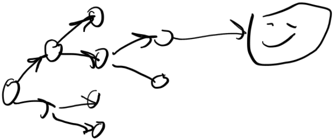
- Impossible to represent simultaneous delivery events directly!
- (But not necessary.)

Nondeterminism

A configuration C might have many enabled actions. Which to do?

- *All of them!*
- More specifically $\forall$ possible executions, algorithm works.
- Universal quantifier anthropomorphized as the **adversary**.
- Adversary chooses which enabled action to do next.

Safety and liveness



Want to prove an algorithm works no matter what the adversary does.

- Safety properties: nothing bad happens
- Liveness properties: something good happens eventually

Typically both are proved by induction over executions.

Liveness will require some constraints on adversary we'll discuss later.

Example: flooding

Goal: Deliver a message to every process in an incomplete but connected network.

```
initially do
    if pid = root then
        received ← m
        send m to all neighbors
    else
        received ← ⊥

upon receiving m do
    if received = ⊥ then
        received ← m
        send m to all neighbors
```

Proving correctness

Claim: Algorithm delivers m and only m to all processes.

Split into

1. In any reachable configuration, $\text{received}_i \in \{\perp, m\}$.
2. Eventually $\text{received}_i \neq \perp$ forever.

where both claims hold for all processes i.

Note: (1) is a safety property, (2) is a liveness property.

$\exists$ future win

Eventually: 0 0 0 0 0 0

Forever: $\underbrace{0\ 0\ 0\ 0\ 0}_{\text{always win.}}$

Ev P: $\Diamond P$

For P: $\Box P$

(2): $\Diamond \Box\ \text{received}_i \neq \perp$

$0\ 0\ 0\ 0\ 0\ \underbrace{1\ 1\ 1\ 1\ 1}_{\text{always keep winning.}}$

Proving a safety property

Show P is an **invariant**:

- $P(C_0)$ holds.
- If $P(C)$ holds and α is enabled in C, then $P(C\alpha)$ holds.

Since in any execution $C_{t+1} = C_t\alpha$, this gives

- $P(C_0)$
- $P(C_t) \Rightarrow P(C_{t+1})$

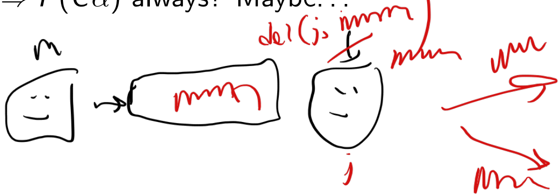
So induction gives $P(C_t)$ for all $t \in \mathbb{N}$.

Safety of flooding

Recall the claim:

In any reachable configuration, $\text{received}_i \in \{\perp, m\}$.

- Is it true in C_0? Yes, $\text{received}_i = m$ for initiator and $\perp$ for everybody else.
- Does $P(C) \Rightarrow P(C\alpha)$ always? Maybe. . .



Using a stronger invariant

Let's try:

1. For all i, $\text{received}_i \in \{\perp, m\}$.
2. For all ij, b_{ij} contains only m.

Is this an invariant?

1. Is true in C_0. If α changes received_i to m', then m' was in some b_{ji} in a configuration satisfying (2). So $m' = m$.
2. Is true in C_0. If a new message m' is added to b_{ij}, it's because i received it from some b_{ki} in a configuration satisfying (2). So again $m' = m$.

(1+2) hold in init config.

(1+2) preserved by del(i, m)

C

α

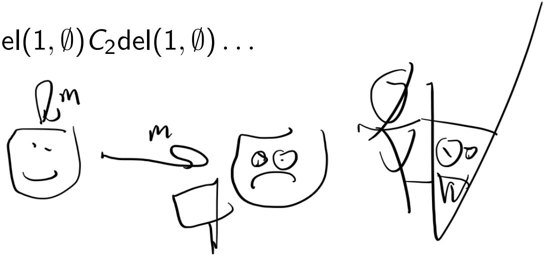
Proving liveness

Eventually, $\text{received}_i \neq \perp$ forever.

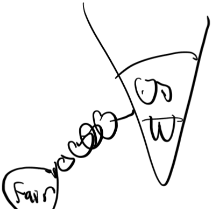
Since nothing in the code replaces not-$\perp$ with $\perp$, it's enough to show that eventually $\text{received}_i \neq \perp$ in some configuration.

But here is an execution in which this is not true:

$C_0 \text{del}(1, \emptyset) C_1 \text{del}(1, \emptyset) C_2 \text{del}(1, \emptyset) \dots$



Fairness



To prove liveness, we must have **fairness**.

Define a **fair** execution by the rule

- Every message that is sent is eventually delivered.

(Details may change in other models.)

Now insist only that fair executions satisfy liveness.

$\forall\ \Xi$, works

$\forall$ fair Ξ works

Liveness of flooding with fairness

Expand

- For all i, eventually $\text{received}_i \neq \bot$

to an induction hypothesis

- For all d and all i at distance d from the initiator, eventually i recieves m.

Now do induction on d:

- Base case: $d = 0$. The initiator receives m.
- Induction step: Suppose i is at distance $d + 1$. Then i has at least one neighbor j at distance d. When j eventually receives m for the first time, it sends m to $i \Rightarrow$ eventually i receives m.

How much does this cost?

Some complexity measures are straightforward:

- Local computation: How much work did each process do?
  - Don't care. Pretend it's free.
- Message complexity: How many messages were sent?
  - Flooding sends $2|E|$ messages.
- Bit complexity: What was the total size of those messages?
  - Flooding sends $2|E||m|$ bits.

Some are not:

- Time complexity: How much time until algorithm finishes?
  - Maybe a long time with unbounded message delays!
  - We'll revisit this next time.