Distributed Systems — Class 1

cs.nyu.edu/~apanda/classes/sp26

↳ PLEASE SIGN UP FOR CAMPUSWIRE.

Today

- What?
- Class mechanics
- Background.

What?

Sometimes, want to use resources (CPU, memory, disk, GPU, ...) from more than one computer

↳ Fault tolerance

+ ATC (Sift 1978)
+ Netflix
+ ...

→ Geographic reach
- E-mail
- ...

→ Resource limitations on one computer
↳ Memory, disk b/w, GPUs, CPUs, ...

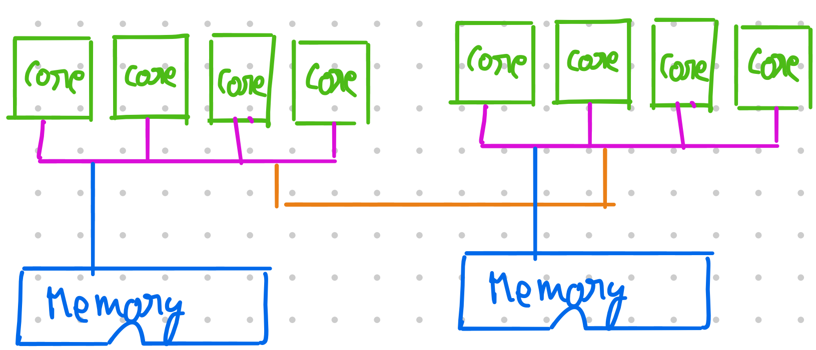
Distributed system

→ Writing & reasoning about programs that run on multiple computers connected by a network

But concurrent programs

- Threads, ...
- Maple
- ...

} → Lots of work
↳ All of you have encountered this (Pre-req)



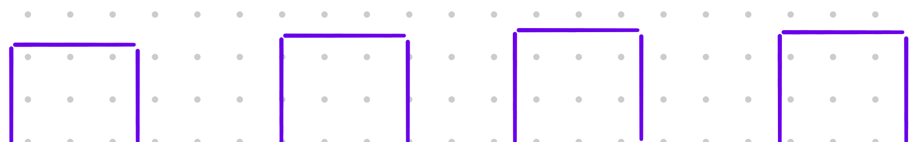
Synchronous Model

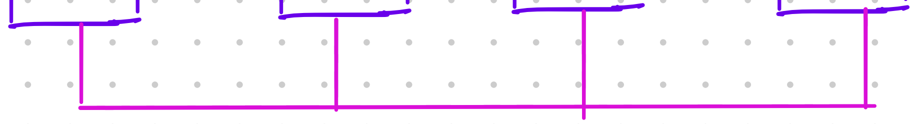
One big difference

- Built so to ensure some timing requirements
  - Inter-core & Inter-node interconnect } Known latency bounds for messages
  - Memory: interconnect } Known lat bounds for access
  - Cores - Known response bounds for messages (Coherence traffic, NMI, IPI, ...)
- Costs money, hard to guarantee these as things scale (geographically, in number of cores, etc.)

Distributed system

→ Writing & reasoning about programs that run on multiple computers connected by a network



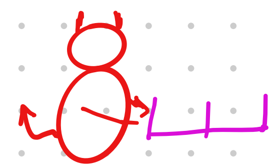


ASYNCHRONOUS MODEL

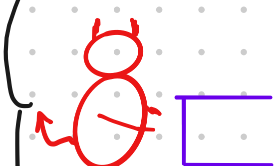
Ideally, no

- Timing assumptions about the network
  + Messages can take arbitrary time
  + Delivery order is arbitrary
  + Fairness guarantee

  } Will make this precise today + next class



- Timing assumptions about the computers (nodes)
  + Processing a message can take arbitrary time, ...
  + Clocks need not run at the same rate.



Asynchronous ⟹ Cannot determine if a computer has failed

↳ Its response might just be delayed

| Cannot distinguish b/w failure & delay |

Concurrent programming primitives
↳ Almost always assume synchrony
→ Almost always assume either
→ No failures
(Probably what you have seen)
→ Can detect failures

Much of our focus will be on algorithms/protocols/programs that can handle failure & make minimal timing assumptions

Common sentiment: writing & testing distributed programs is hard
- Large space of possibilities
  - Messages arrive in diff. orders
  - Diff. machines fail
  - Diff. delays
  - ...

So need tools (pen, paper, mathematical tools, etc.) to reason about programs & protocols:

Our focus
- What does it mean for them to be correct
- Under what assumptions are they correct

Note, a lot of this will involve reading & thinking

Course mechanics

- Course staff: Me
- Material:

  cs.nyu.edu/~apanda/classes/sp26

  Redoing the schedule a bit, currently only covers this class + next
  Complete schedule will be up by next class.

- Communication
  - Campuswire
    - keep your posts about course material public
    - Answer other people's questions
      - ↳ Good way to learn
    - If possible, consider not using anonymous posts.

- E-mail

  apanda@cs.nyu.edu

- Office hours

  Monday 1-2pm
  405 60FA
  OR e-mail to find alt.

- The work
  - 2ish papers each week
    - ↳ Fine (by me) if you use tools (e.g., NanoBanana Pro) to help make this palatable
    - → Important (for class discussion) that you know what is going on
  - Weekly class
    - ↳ Papers + Outside context
    - → Please interrupt, question, argue
  - Final project (20% of grade)
    - ↳ A bigger part this year
    - → Will have suggestions out by next class
    - → Can do them alone or in a

group of 2 (GROUP PREFERED)

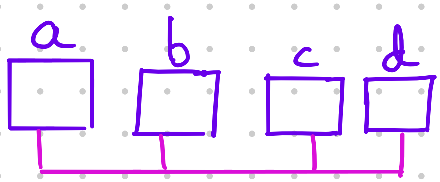
→ IF YOU ARE ALREADY WORKING ON RESEARCH

↳ Reuse that!!!

- 3 coding projects / labs (25%)
  - ↳ In Elixir

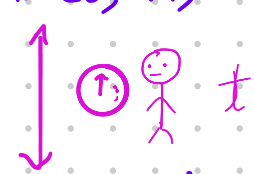
- Midterm (20%) + Final (25%)
  + Open book
  + Final is cummulative

Back to asynchrony



Network → Message passing (What we will be using most often)

a: send (b, m)



b: receive → ⟨a, m⟩

- No bound on t
  - ↳ Can be very small (e.g., 0.0000…1)
  - Can be very large

→ Does not need to be the same across messages

a: send(b, m)

↕ t

b: receive → ⟨a, m⟩

a: send(b, m)

↕ t'

b: receive → ⟨a, m⟩

Cannot predict t' given t

- Implication: Cannot assume an order on how messages are received

a: send(b, m) .. wait(0.5s) .. send(c, m')

b: receive() → ⟨a, m⟩

c: recv() → ⟨a, m'⟩

a: send(b, m) .. wait(0.5s) .. send(c, m')

c: recv() → ⟨a, m'⟩

b: receive() → ⟨a, m⟩

Note, delays are different from the question of whether the network is

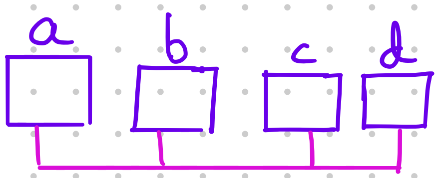
reliable (all messages that are sent are eventually received)

OR unreliable (any received message was

previously sent)

Processing delay is similar.

Fairness:



Will often think of the execution of a distributed protocol/program in terms of a __schedule__ of __events__

__events__

0 a: init
1 b: init
2 c: init
3 d: init

} will omit for simplicity

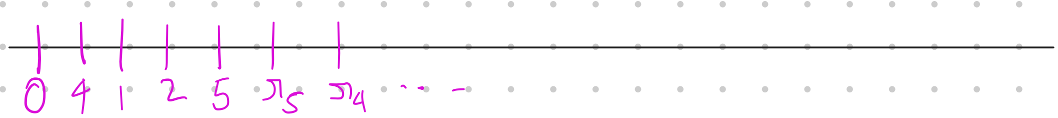
r_4

4 a: send(b, m) — enables — b: receive() → ⟨a, m⟩

5 a: send(c, m) — enables — c: receive() → ⟨c, m⟩

6 b: nop
7 c: nop
8 ⋮
9
⋮

Schedule: the order in which these events appear to occur



As we will see next class –

Reasoning about correctness (safety/liveness)

→ Reasoning about the set of schedules a distributed program can produce

↳ Does each schedule S meet some property P $S \models P$

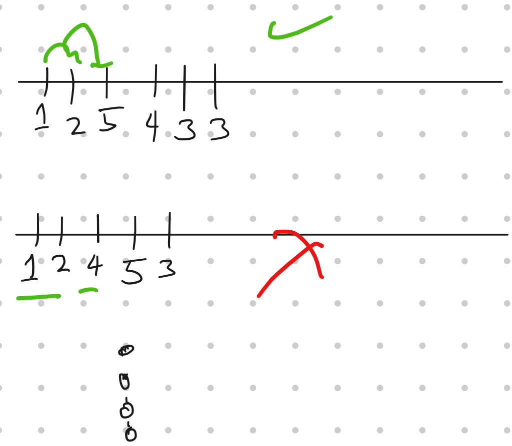
Program specifies what events are Enabled (can occur)

A: on init
  1 send (b,m)
  2 send (c,m)
 on recv ⟨b, ack⟩
  3 send (d, 1)
 on recv ⟨c, ack⟩
  4 send (d, 2)

B,C: on init ___

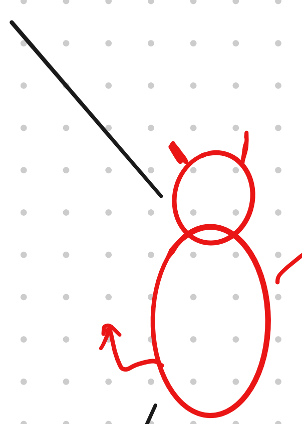
 on recv ⟨a, m⟩
  5 send (a, ack)

Execution environment dictates what schedules are possible



Asynchronous model ⟷ Model of the execution environment

A: on init
  1 send (b,m)
  2 send (c,m)
 on recv ⟨b, ack⟩
  3 send (d, 1)
 on recv ⟨c, ack⟩



When to run (execute) handler for each

1?

4 send (d, 2)

B, C: on init ___

on recv <a, m>

5 send (a, ack)

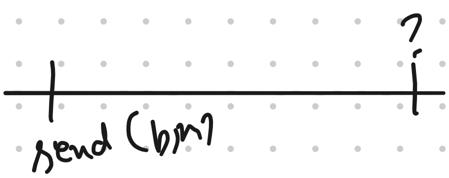
each enabled event

3?

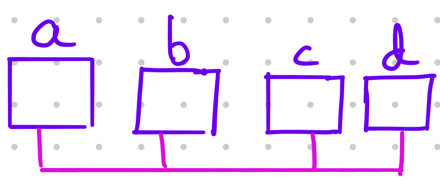
- Remember, 3 can be executed arbitrarily after 1
  - ↳  choose to never execute enabled e?

Fairness — rule to avoid this

↳ In this class we use STRONG FAIRNES



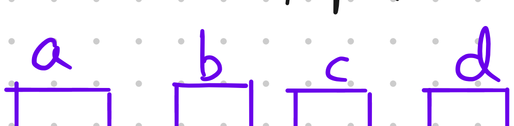
If event e is enabled infinitely often then e [handler] is executed infinitely often.

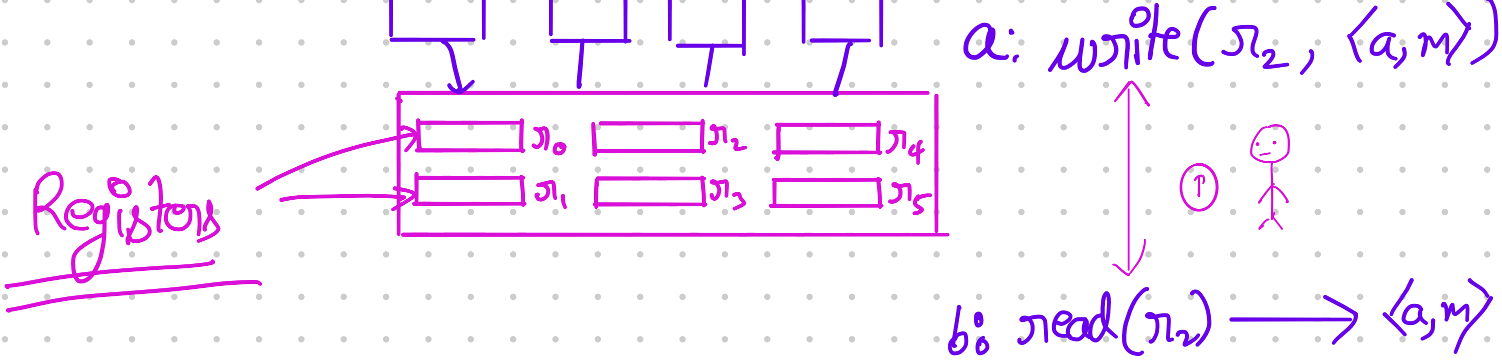


→ Message passing (What we will be using most often)

→ Shared memory

- Won't be using this for the most part
  But... shows up in the linearizability paper





Usually, registers are linearizable ⟷ Don't worry about this for now

Assert: equivalent to message passing

Processes & failure model

- Remember — we care about protocols & programs that function despite failures
  - ↳ Impossible under arbitrary failures

  n machines —n fail→ ??

  → Many things are impossible (unsolvable) even if we assume at most n-1 failures

  → As we will see, some (important) things impossible even with 1 failure

- So common for the things we study to restrict what can fail and how
  - ↳ FAILURE MODEL

→ # of computers that fail

→ How does failure manifest

↳ FAIL STOP: A failed computer does not do anything: no sending messages, no processing, ...

← COMMON (for this class)

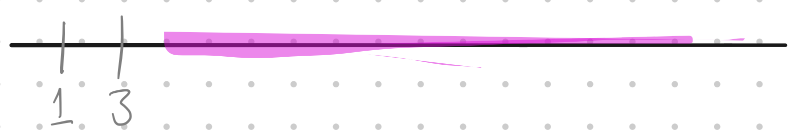
→ FAIL RECOVER: fail stop but eventually recovers, maybe with old state

→ Byzantine: Failed computers can behave arbitrarily

I/O Automata ⟷ Our model of process execution

on init
    send (b, m) 1
    send (c, m) 2
on recv ⟨b, ack⟩
    send (d, 1) 3
on recv ⟨c, ack⟩
    send (d, 2) 4

Sched/trace



send(a, 2)