HIERARCHICAL LOAD BALANCING

ISMAEL ALONSO

THE PROBLEM

- Suppose you have a set of agents (or nodes) capable of processing “Jobs”
- Any one of these agents are assigned jobs externally
  - Clients connect to them and send them work directly
- If a subset of these agents has a bigger load than the rest the response time may degrade
  - There is a caveat here, we’ll get to this later
- How to notice there is a problem and what to do in response?

THE PROBLEM

- We have several options
- Fully centralized load balancing
  - Doesn't scale well
- Fully distributed load balancing
  - Takes some time to reach good decisions
- Can we compromise?

HIERARCHICAL LOAD BALANCING

- Yes, we can!
- We are going to form groups of nodes, each one having a leader
- Any one node may only communicate with a leader or with their children
  - Unless… more on this later.
- Child nodes can be the leaders of their own subgroup
- Effectively, the hierarchy will end up looking like a tree.

BUILDING THE HIERARCHY

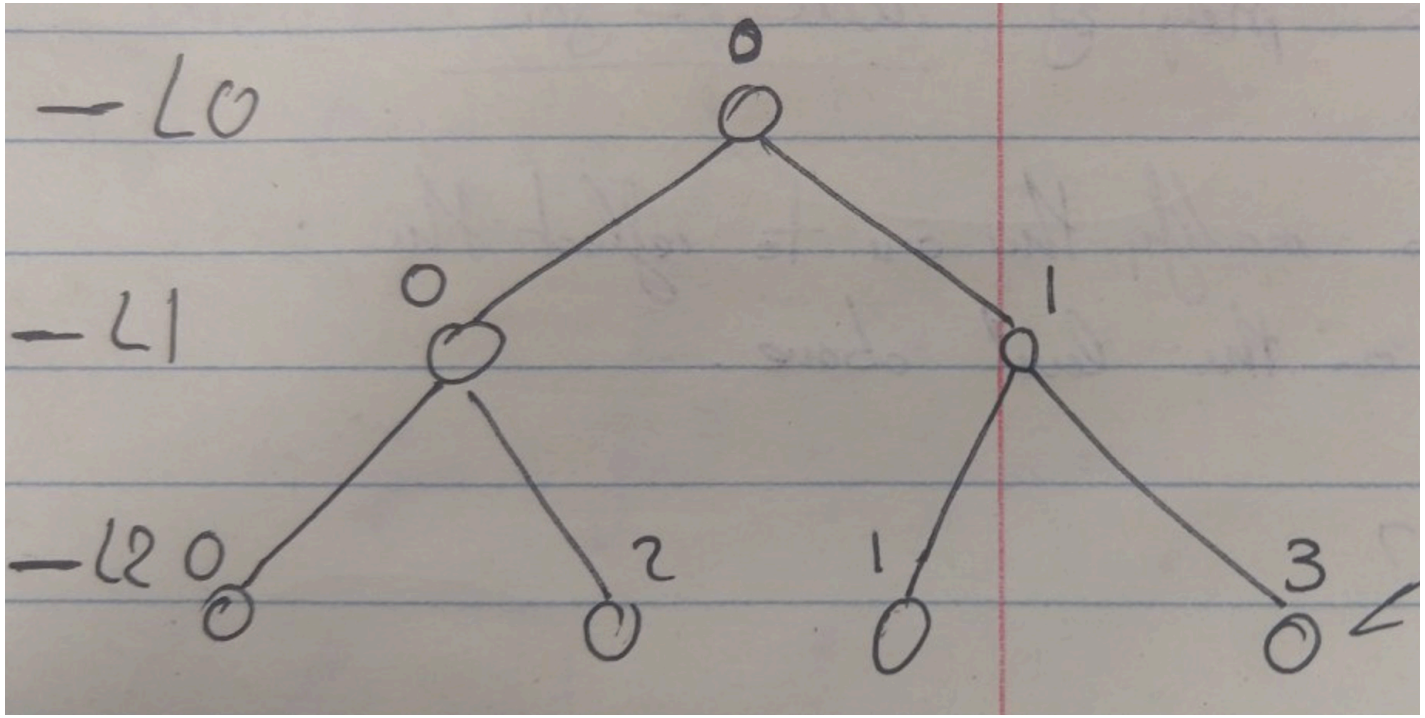
- We now have choices
  - Branching factor (2 in my case)
  - Full tree vs complete tree vs a train wreck of a tree
  - Spatial locality or tunneling may influence tree building
- Truth is, I arranged them as they came
  - Complete tree though, there is some order to my universe
- How?
  - Remember that all nodes need to be leaves independently of whether they are leaders

TIME TO PUT THE NODES TO WORK

- Originally three problems:
  - Fibo: linear
  - Sieve of Eratosthenes: n log(log(n))
  - Square sum (n^2)
- This turned out to be a problem to test the system
  - We'll see why in a minute

WHAT THEN?

- Now we need to know the load of the nodes in the hierarchy
- What do you report though?
- Remember this mess? → → →
  - 1 cannot report its own to 0
  - 0 cannot have a combined report at L0
- Merge up whatever you have
  - Its good enough, trust me 👌

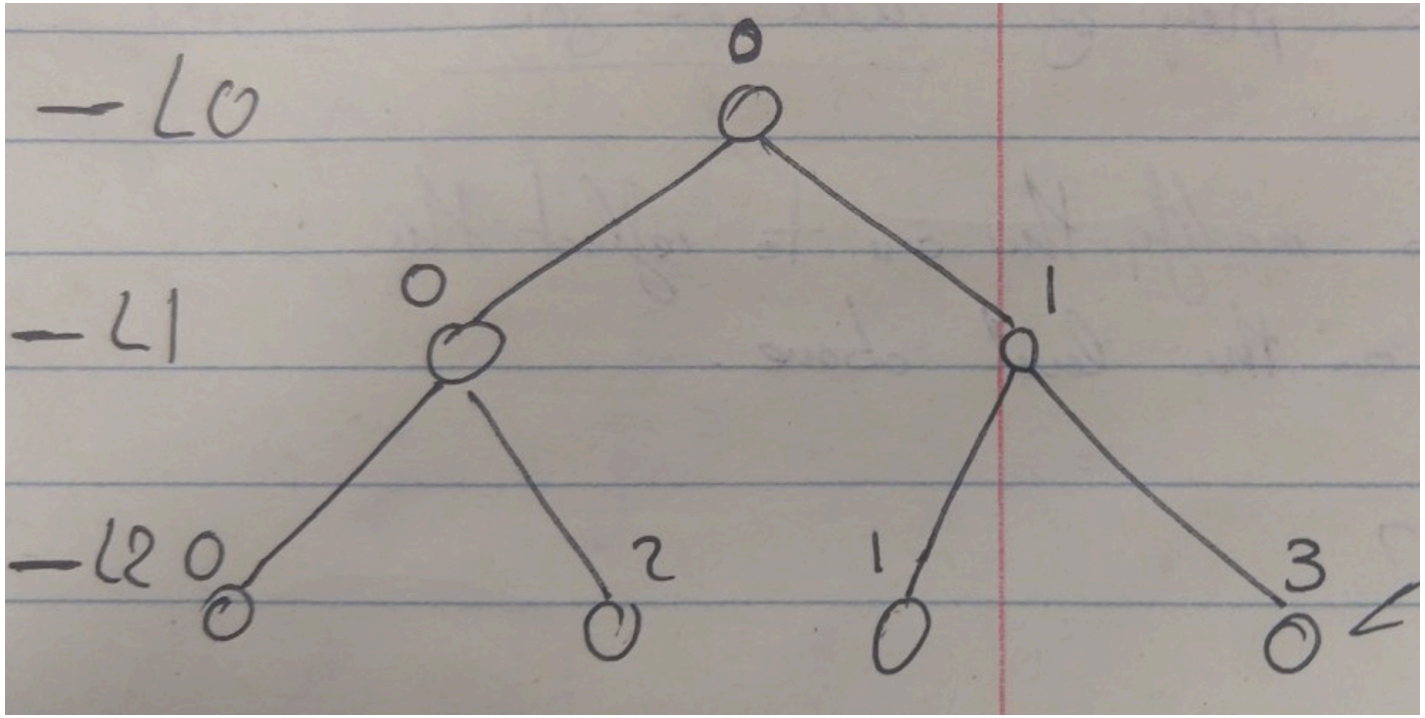


YOU GOT MAIL!

- An imbalance has been found!
  - A node is 10% above or below average load
- Freeze!
  - Or don't, but you run the risk of balancing jobs that were completed at balance time
- Collect all the jobs, aggregate them and pass them up
  - JK, just job metadata: estimated length and owner
- Each leader defers decisions until its own leader has made decisions
- Leaders can choose to further aggregate jobs by combining them (this is an optimization)

MORE ON COLLECTION

- I bet you still remember these shenanigans → → →
- Aggregating the jobs is not easy, requires synchronization
  - Like a lot of it
- Cannot pass anything up until you have a complete picture



PROBLEM #1

- Unless a node is over capacity, any utilization metric is worthless
- Suppose you got a node at 20% capacity and another one at 70% capacity
  - Congrats, it's load balancing time!
  - Wait, there are no jobs in the queues… wat?
- To be more specific, hierarchical load balancing in this system is only good if two conditions are met simultaneously:
  - One node is grossly overutilized
  - One node is at least somewhat underutilized

PROBLEM #2

- Okay, so, to test this thing we'll skew the job allocation distribution.
- Sure, let's set up a thread that sends jobs to certain nodes with a frequency and an input size.
- K cool:
  - $n=1000^2$ and T=50ms: load is $1.4x10^{-4}$% (hmmm… right…)
  - $n=2500^2$ and T=40ms: load is 0.01% (whalp, k)
  - $n=6000^2$ and T=40ms: int overflow for estimated job size (!!). Also load ~0.2 (better, but not good enough)
- Ended up putting nodes to sleep while counting it as working for jobs
  - They got off easy, I know…

WE HAVE JOB INFO AT THE TOP NOW

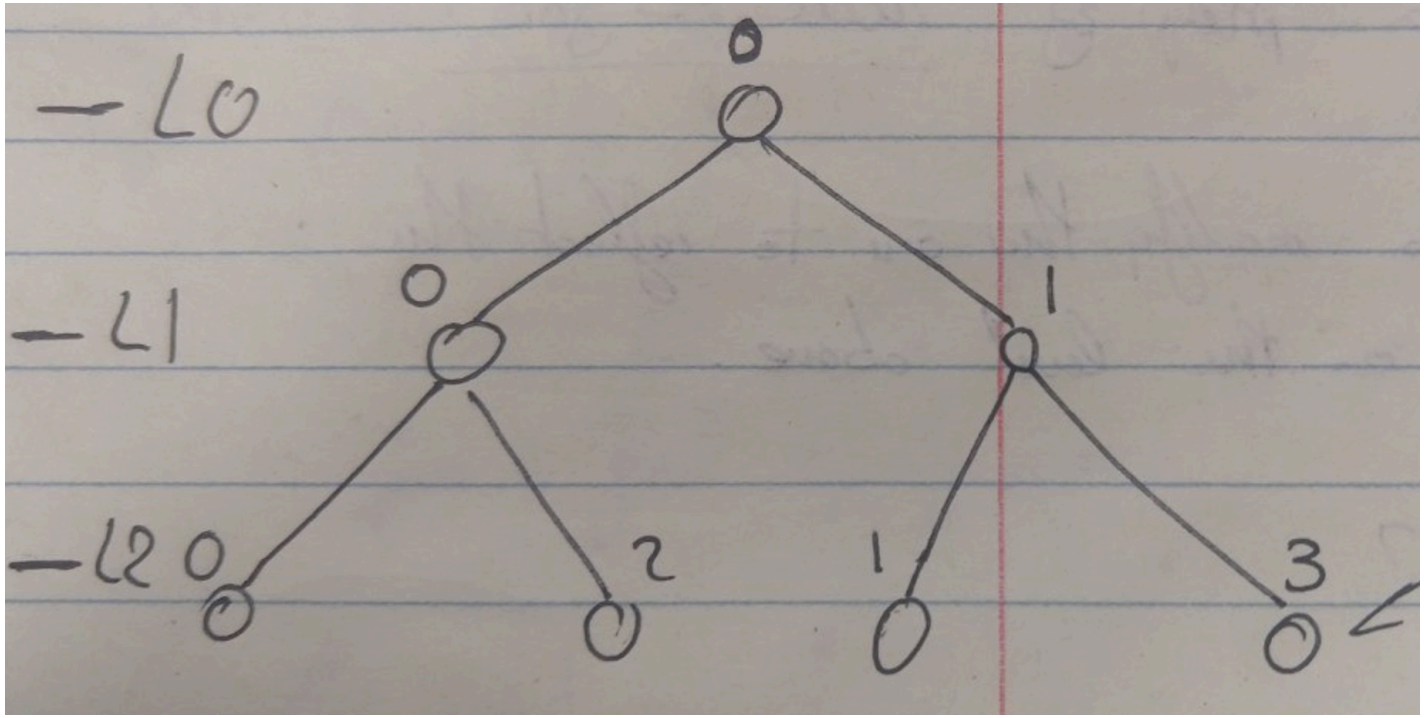
- RefineLB? What's that?
  - As it turns out this is a Charm specific thing and I wanted to build my own
- Each processor has a set of jobs it owns, ordered by size
- The processors are also ordered in decreasing order of load (work allocated)
- Jobs over the average load are donors
- Jobs under the average load are recipients

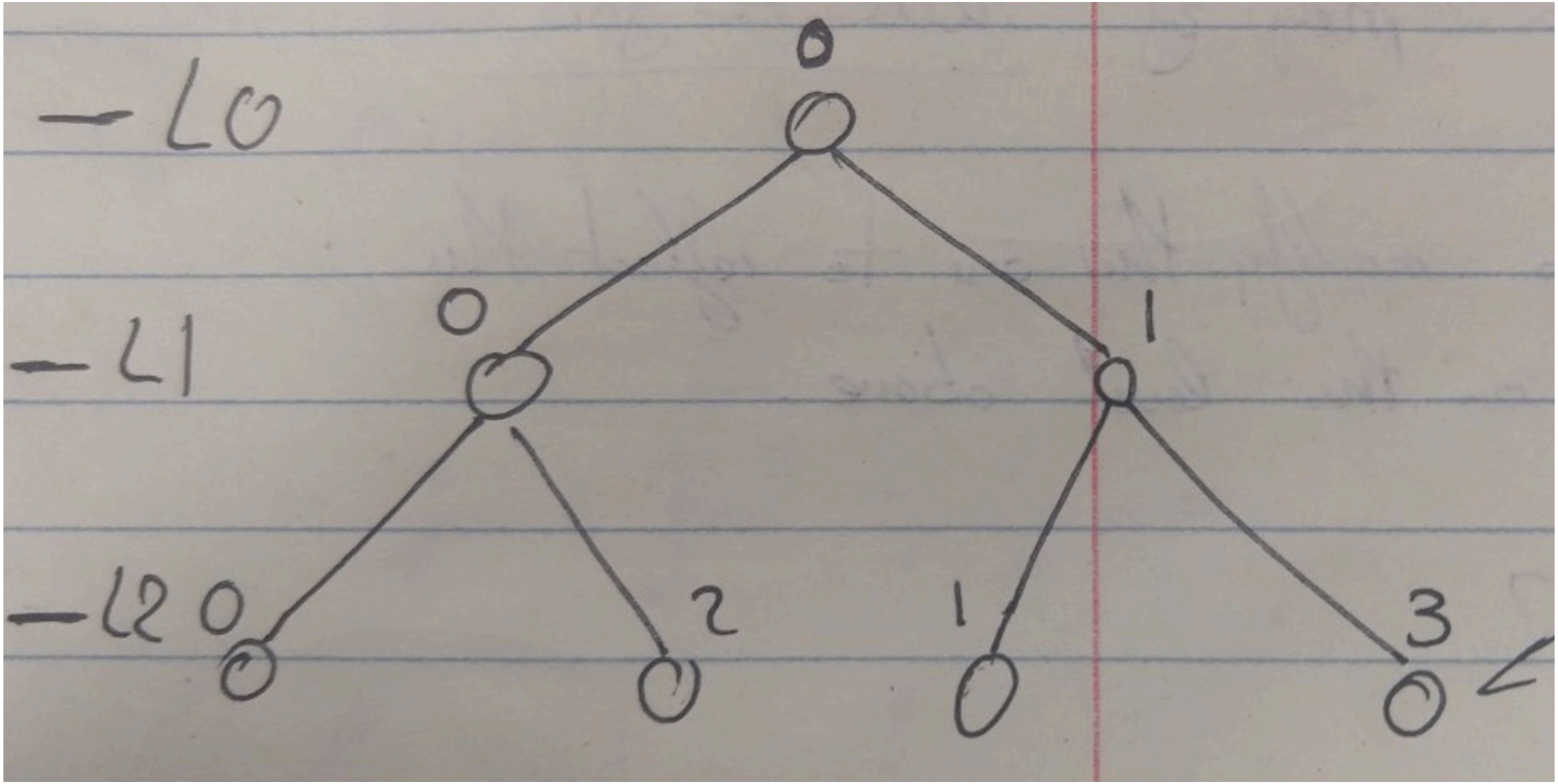
IZZY'S LAW OF LOADS

- A node that's already overloaded will tend to stay loaded
  - Don't know if this is a thing, just a suspicion
- Donors will donate until they're under the average
- Recipients will receive until they're above the average
- Repeat until there's not longer both donors and recipients
- Selected job is one that's just under the potential room of load a recipient can take to go above
  - Or the one immediately above if none below

PROPAGATING THE CHANGES DOWN THE PIPE

- A leader will communicate with all its children about the decisions made
- The first level is easy, nothing external to the process really needs to be considered
- I'm sure at this point we're drinking buddies with this bad boy → → →
  - Whalp, there's a few edge cases…




L0
L1
L2
0
0
1
0
2
1
3

SO… AGREEMENT?

- Once everyone is on the same page about what jobs we need to steal from other nodes…
- We just ask for them… nicely.
- Once we get what we want we resume operations and wait some time to start checking whether there is an imbalance again.

MY SYSTEM

- A little overengineered, perhaps
- Controllers spawn nodes in a single machine
- Nodes do the work
  - Controllers spawn them as processes, so getting info out of them isn't trivial
- We got a dedicated logging node, started directly from a terminal we have access to
  - Fixed location
- There is a client, just runs the show

MY SYSTEM

- At some point late last week realized I didn't have time to prep it to run in multiple machines
- Single local controller spawns as many nodes as we need
  - Remember they don't do real work anymore, so we can spawn virtually an unlimited amount of them
- Demo?

ANALYSIS

- Load information is readily available and cheap to send/aggregate
- To get job info, as many RTTs as levels the tree has are required
- Propagating the load balancing decisions takes a variable amount of time
  - The big fish executes the process for as many levels as the tree has, but at the end of the process its info is final
  - Nodes closer to leaves must wait until the chain of command has decided what to do (up to log n RTTs)
- Fetching your jobs takes one RTT

QUESTIONS?