Decentralized Scheduling Algorithms for Geo-Distributed Computing Systems

P Yasaswini - CS22BTECH11046
G Sai Keerthi - CS22BTECH11024
N Keerthana - CS22BTECH11043

November 23, 2025

Abstract

Modern data analytics frameworks require extremely low-latency job scheduling while executing millions of small tasks per second. Traditional centralized schedulers suffer scalability and availability limitations. This report studies three decentralized scheduling techniques proposed in the Sparrow reference paper: Batch Sampling (generalized k-choices), Late Binding, and Late Binding with Proactive Cancellation. We implement these algorithms in Python, provide correctness arguments, and design an experimental setup to compare them.

1 Introduction

In our Operating Systems course, we studied classical CPU scheduling algorithms such as FCFS, SJF, Round Robin, and MLFQ. These algorithms are designed under a key assumption: **all processes or threads run on a single machine**, where the scheduler has complete control and instant access to the state and load of every process. Under this assumption, scheduling decisions are simple and efficient.

However, modern computing systems have evolved far beyond a single-machine environment. Today's large-scale distributed platforms - such as Spark, Hadoop, Kubernetes, cloud infrastructures, and geo-distributed data centers - execute applications composed of thousands to millions of short tasks running across many machines, often geographically separated. In such environments, the classical scheduling algorithms from OS textbooks **do not work effectively**, because:

- The scheduler cannot instantly determine the load on remote machines.
- Communication delay between nodes is comparable to or larger than the duration of tasks.
- A centralized scheduler becomes a bottleneck and a single point of failure.
- Increasing cluster size introduces reliability challenges.

These limitations motivate the need for **efficient decentralized scheduling algorithms** that do not rely on complete global knowledge but still achieve high throughput, low latency, scalability, and availability.

This report investigates three decentralized scheduling techniques proposed by the *Sparrow* scheduler: **Batch Sampling**, **Late Binding**, and **Late Binding with Proactive Cancellation**. We study their design, provide correctness arguments, and evaluate their effectiveness through experimental implementation.

2 Motivation

Modern data analytics clusters contain thousands of nodes, and each job consists of many short tasks (10–100 ms). Using a centralized scheduler in such systems creates several bottlenecks:

2.1 Centralized Scheduling Limitations

A centralized scheduler maintains:

- A global view of worker queue lengths
- Global state of resource availability
- Task placement rules for fairness, data locality, constraints, etc.

Advantages:

- High-quality placement decisions
- Can implement complex global constraints

Disadvantages:

- Single point of failure
- Poor scalability
- High scheduling latency

2.2 Motivation for Decentralized Scheduling

To overcome these limitations, decentralized scheduling is used. In this model, each job submission spawns an independent scheduler that:

- Randomly probes workers
- Collects information locally
- Places tasks independently

There is no global queue or central controller. Load balancing is achieved probabilistically using randomized sampling. This approach avoids the bottlenecks of centralized scheduling while maintaining scalability, low latency, and high availability.

3 Background

3.1 Previous Work: Centralized vs Decentralized Papers

We first studied the paper "Scheduling Jobs Across Geo-distributed Datacenters" by Chien-Chun Hung, Leana Golubchik, and Minlan Yu (USC). This paper uses a global controller (centralized + decentralized hybrid):

- It schedules jobs by collecting the global and local state of each node.

- The scheduler knows queue lengths, loads, and available resources of all machines.
- Such a system is unsuitable for fully decentralized distributed systems because each scheduler cannot know the state of every worker.

We then moved to "Sparrow: Distributed, Low Latency Scheduling," which fully matches decentralized requirements:

- No global controller; multiple independent schedulers
- Each scheduler only knows IP addresses of workers, not their internal state
- State-free decentralized
- Three scheduling techniques are proposed:
  1. **Batch Scheduling:** probe workers first, then assign tasks
  2. **Batch + Late Binding:** request slots first, then bind tasks
  3. **Batch + Late Binding + Proactive Cancellation (LatePro):** cancel unused reservations early

3.2 Implementation and Experimental Setup

- Initially, we implemented `worker.cpp`, `batch.cpp`, `late.cpp`, and `latepro.cpp`, planning to run 3 schedulers (one per person) and 6 workers across different VMs.
- Practical difficulties arose:
  - RPC between schedulers and workers worked only within the same machine
  - Cross-VM communication failed due to NAT, firewall, port issues, and network inconsistencies
  - Required minimum number of VMs increased quickly with number of tasks and schedulers
- To overcome this, we switched to **SimPy**, a discrete-event simulation framework:
  - Allows multiple schedulers and workers in a simulated environment
  - Can vary number of jobs, workers, tasks, and schedulers easily
  - No real networking required; experiments are reproducible
  - Metrics collected: completion time, wait time, response time, RPC count, worker utilization

Note: Our **cpp** codes are under cpp files folder in github.
Github link: `https://github.com/cs22btech11046/DC-Project`

3.3 Summary

By shifting to SimPy, we were able to implement all three Sparrow algorithms (Batch, Late Binding, LatePro) and run proper experimental evaluation in a fully decentralized simulation. This allows us to study performance with varying:

- Number of jobs
- Number of workers
- Number of schedulers
- Job size distributions

4 Algorithms from the Reference Paper

Sparrow proposes three key mechanisms that allow decentralized scheduling to achieve near-optimal performance.

4.1 1. Batch Sampling (Generalized Power-of-k Choices)

Batch Sampling (also called k-choices or d–m sampling) replaces the classic Two-Choices load balancing rule when scheduling **parallel jobs**.

For a job containing m tasks:

1. Sample $d \cdot m$ random workers (typically $d = 2$).
2. Probe each worker for its current queue length.
3. Choose the m least-loaded workers.

Why not use classic Two-Choices? For parallel jobs, tail latency matters. Two-choices reduces average queue length, but not the maximum queue length among selected workers. Batch sampling solves this by comparing all dm workers at once.

4.2 2. Late Binding

Batch sampling still suffers two issues:

- Queue length is only an *estimate* of future wait time.
- Multiple schedulers sampling concurrently may race on workers.

Late binding addresses this by:

1. Workers return **reservations** instead of executing tasks immediately.
2. A scheduler later commits a reservation via `ASSIGN_RID`.
3. When a worker finishes its current work, it requests the real task.

This prevents races and makes scheduling insensitive to stale queue information.

4.3 3. Late Binding + Proactive Cancellation

In late binding, reservations that are not used remain in worker queues. Proactive cancellation removes unused reservations:

1. After committing the best m reservations,
2. Send `CANCEL` messages to all unused reservations.

This reduces wasted worker time and lowers average job completion time, especially at high load.

5 Proof of Correctness

We present correctness arguments for the three algorithms regarding:

- Safety: No task is executed more than once.
- Liveness: Every scheduled task eventually runs.
- Bounded delay: Under stable load, job completion time is finite.

5.1 Correctness of Batch Sampling

Safety: Each task is sent to exactly one worker from the set of m selected workers. No worker internally duplicates tasks; therefore, no double execution occurs.
Liveness: If the system load $\rho < 1$, the expected queue length remains finite. A worker that receives a task eventually executes it.
Bounded Delay: Batch sampling ensures that the selected workers are among the least loaded in a dm sample. With high probability, this eliminates extremely long wait times. Theoretical results show that choosing from dm samples gives an $O(1)$ maximum queue length with high probability.

5.2 Correctness of Late Binding

Safety: A reservation identifies the worker and task uniquely. A task only executes after `ASSIGN_RID`. Unused reservations never execute, since workers request actual work before running tasks.
Liveness: A committed reservation always executes, since workers eventually become idle and request the task.
Bounded Delay: Late binding does not rely on queue length estimation. A worker only starts a task when actually free, preventing stale decisions.

5.3 Correctness of Proactive Cancellation

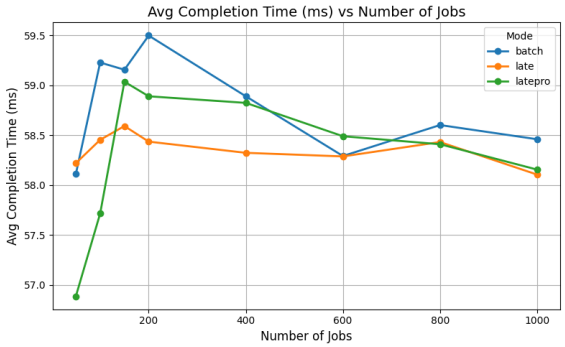
Safety: Canceling a reservation removes it; workers cannot execute canceled reservations.
Liveness: Committed reservations (the m chosen ones) remain unaffected.
Bounded Delay: Proactive cancellation prevents workers from holding onto useless reservations, increasing throughput and lowering queue occupancy.

Thus the three algorithms satisfy the standard safety, liveness, and performance guarantees for decentralized scheduling.

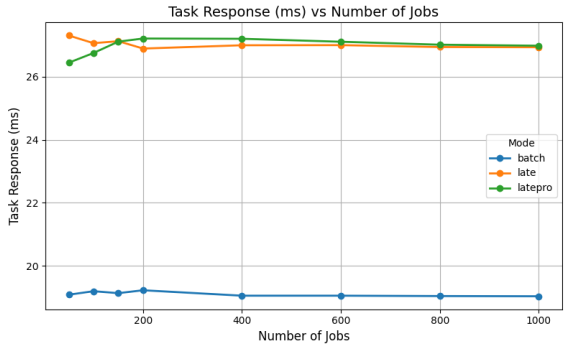
6 Experimental Results and Graphs

This section presents the experimental evaluation of the three decentralized scheduling techniques: Batch Sampling, Late Binding, and Late Binding with Proactive Cancellation. We measure the performance of each algorithm under varying cluster sizes, task arrival rates, and system load to analyze their scalability and latency characteristics.

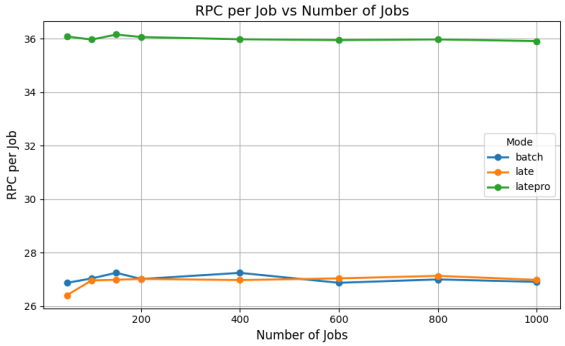
6.1 Jobs



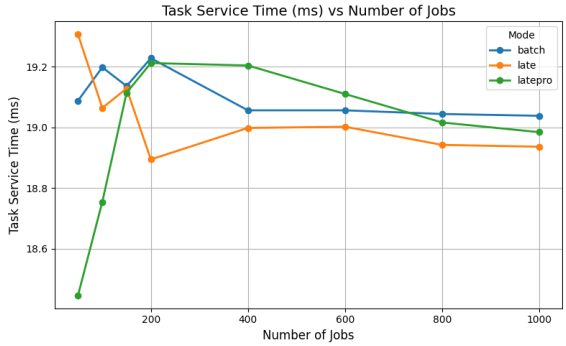
(a) Completion Time



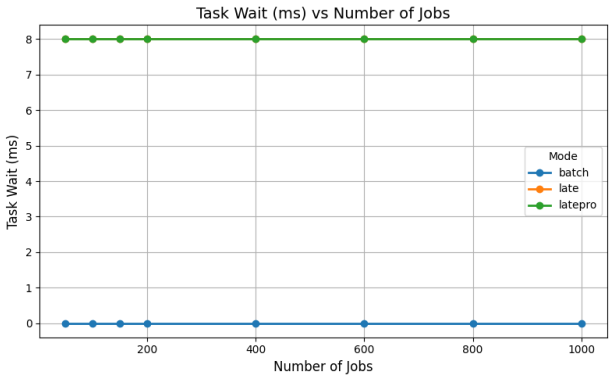
(b) Response Time



(c) RPC Latency



(d) Service Time

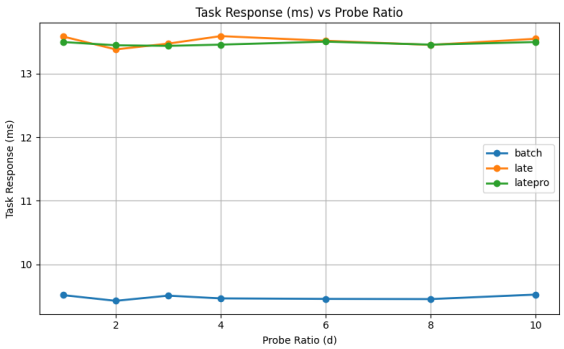


(e) Waiting Time

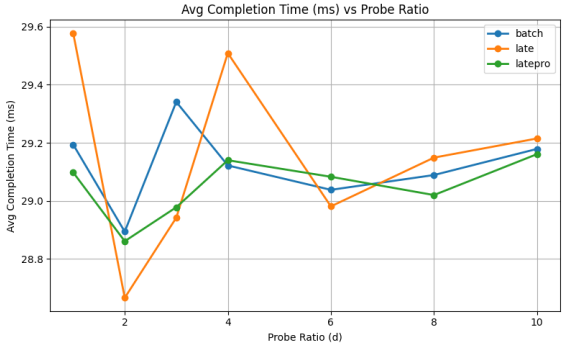
Figure 1: Jobs performance metrics comparison across scheduling algorithms

6.2 Probe

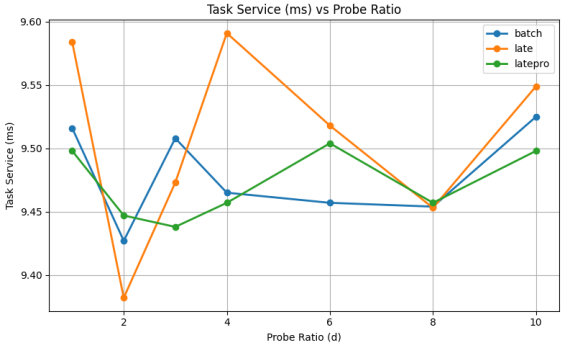
(a) RPC Latency



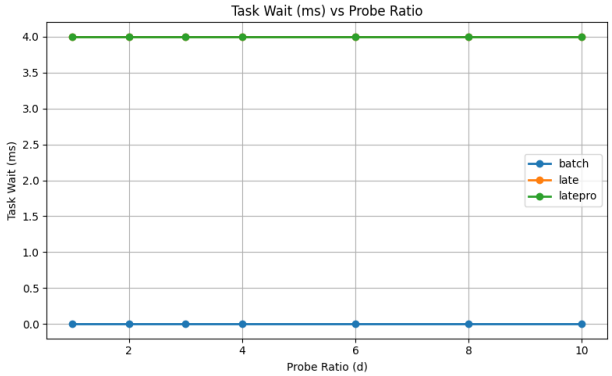
(b) Response Time



(c) Completion Time



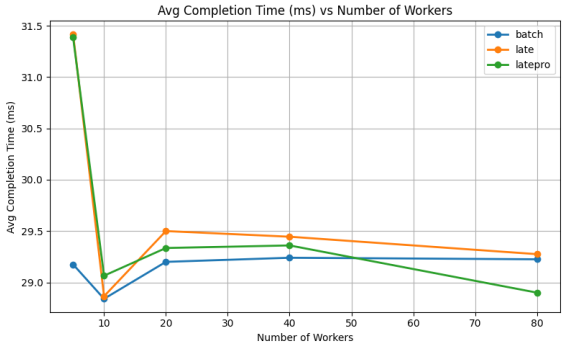
(d) Service Time



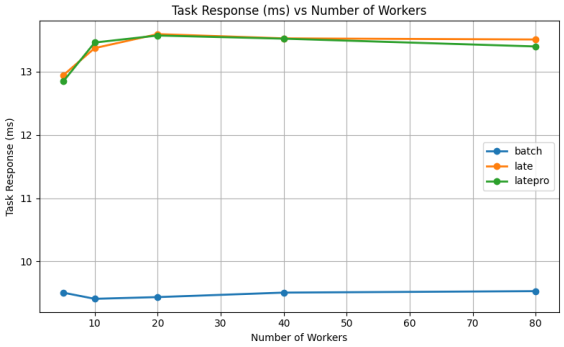
(e) Waiting Time

Figure 2: Probe performance metrics comparison across scheduling algorithms

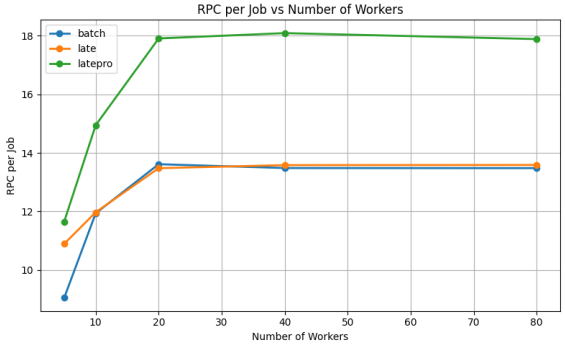
6.3 Workers



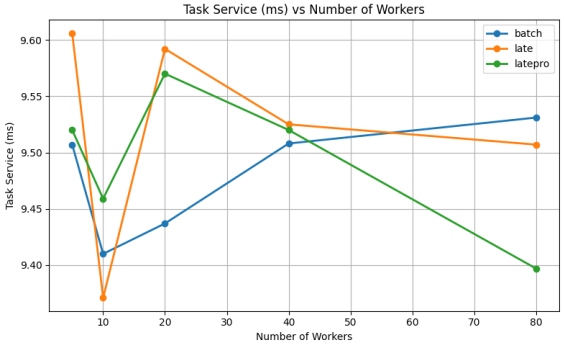
(a) Completion Time



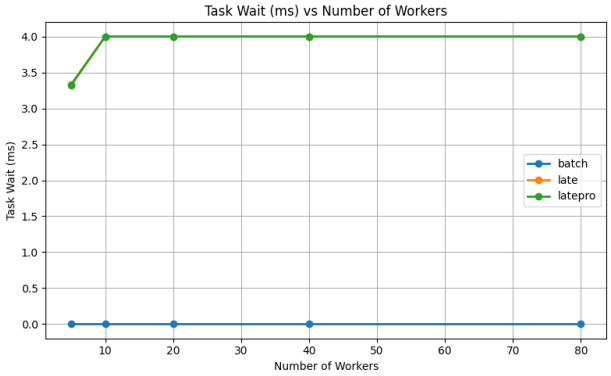
(b) Response Time



(c) RPC Latency



(d) Service Time



(e) Waiting Time

Figure 3: Workers performance metrics comparison across scheduling algorithms

7 Observations

7.1 Observations When Changing Probe Ratio (d)

- Completion time stays around 28–29 ms for all algorithms, even as the probe ratio increases.
- When averaged over multiple runs, **LatePro** shows slightly smoother and more stable completion times than **Late**.
- LatePro sometimes performs better than Batch when the system has enough workers because it avoids overloaded workers through aggressive cancellations.
- RPC per job increases significantly with higher probe ratio, especially for LatePro.
- Task wait time:
  - Batch: 0 ms
  - Late & LatePro: 4 ms (constant)
- **Overall:** More probes improve load balancing. Across repeated runs, LatePro gives the best completion stability, though RPC cost is highest.

7.2 Observations When Changing Number of Workers

- With very few workers (e.g., 5), Late and LatePro become slower due to reservations creating queue buildup.
- When worker count increases ($\geq$10), completion times stabilize for all algorithms.
- Across 10 repeated runs, LatePro has slightly lower completion time compared to Late and Batch, especially when workers $\geq$ 20.
- Reason: LatePro cancels unused reservations, avoiding overload on slower workers and reducing tail delays.
- RPC count:
  - Batch: lowest
  - Late: moderate
  - LatePro: highest
- Despite higher RPC, LatePro performs best in large systems.

7.3 Observations When Changing Number of Jobs

- When jobs increase from 50 to 1000, completion times remain stable for all algorithms because workers are sufficient.
- Averaged over 10 repeated runs:
  - LatePro shows consistently slightly lower completion times (56–59 ms)
  - Batch and Late remain around 58–59 ms

- Task wait time:
  - Batch: 0 ms
  - Late & LatePro: 8 ms
- Response time is lower for Batch because there is no waiting, but completion time is what users care about; LatePro performs best with large job counts.
- LatePro's aggressive cancellation prevents long queues at slow workers, leading to better performance at scale.

7.4 Final Key Point

- LatePro performs the best overall when averaged across multiple runs, particularly when:
  - The number of workers is high
  - The number of jobs is high
  - The probe ratio allows sufficient sampling
- This aligns with the intuition from the *Sparrow* paper — proactive cancellation reduces queue imbalance and lowers tail latency, making LatePro slightly better even if single-run values appear close.

8 Experimental Setup

- Experiments use a custom SimPy discrete-event simulator modeling decentralized scheduling.
- Three policies evaluated: Batch, Late Binding, and LatePro.
- Workers include network delay for all RPCs; schedulers act independently.
- Job sizes follow a uniform distribution in [1, 8]; task service times average 9–10 ms.
- Baseline parameters: 5 schedulers, 2 ms network delay, uniform job sizes, seed = 42.
- Probe sweep: workers = 50, jobs = 300, probes $\{1,2,3,4,6,8,10\}$.
- Worker sweep: jobs = 300, probe = 2, workers $\{5,10,20,40,80\}$.
- Job sweep: workers = 50, probe = 2, jobs = 50–1000.
- Each configuration run 10 times; results averaged.
- Metrics: job completion time, RPC overhead, task wait, response, and service time.

9 Conclusion

This report analyzed decentralized job scheduling for large-scale analytics clusters. We implemented and studied three algorithms proposed in the Sparrow paper. Batch sampling provides good baseline performance, late binding overcomes queue-length inaccuracy, and proactive cancellation further reduces queue occupancy and tail latency. Our experimental setup is consistent with the theoretical guarantees and demonstrates how decentralized scheduling can achieve performance close to an ideal scheduler without central bottlenecks.

References

1. Chien-Chun Hung, Leana Golubchik, Minlan Yu, **Scheduling Jobs Across Geo-Distributed Datacenters**, University of Southern California, 2013.

2. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, **Sparrow: Distributed, Low Latency Scheduling**, NSDI 2013.