Propius: A Platform for Collaborative Machine Learning across the Edge and the Cloud

Eric Ding

August 26, 2024

Abstract

Collaborative Machine Learning is a paradigm in the field of distributed machine learning, designed to address the challenges of data privacy, communication overhead, and model heterogeneity. There have been significant advancements in optimization and communication algorithm design and ML hardware that enables fair, efficient and secure collaborative ML training. However, less emphasis is put on collaborative ML infrastructure development. Developers and researchers often build server-client systems for a specific collaborative ML use case, which is not scalable and reusable. As the scale of collaborative ML grows, the need for a scalable, efficient, and ideally multi-tenant resource management system becomes more pressing. We propose a novel system, Propius, that can adapt to the heterogeneity of client machines, and efficiently manage and control the computation flow between ML jobs and edge resources in a scalable fashion. Propius is comprised of a control plane and a data plane. The control plane enables efficient resource sharing among multiple collaborative ML jobs and supports various resource sharing policies, while the data plane improves the scalability of collaborative ML model sharing and result collection. Evaluations show that Propius outperforms existing resource management techniques and frameworks in terms of resource utilization (up to 1.88×), throughput (up to 2.76×), and job completion time (up to 1.26×).

1 Introduction

Collaborative Machine Learning is a paradigm in the field of distributed machine learning, designed to address the challenges of data privacy issues. Computation tasks are distributed over client edge devices (mobile phones, IoT devices) and silos (companies, hospitals) for machine learning and data analysis. Collaborative ML can be viewed as a superset of Federated Learning (FL) [5].

Collaborative ML enables privacy-preserving and affordable ML, and unlocks huge amounts of private data for training by moving the models to the client side. These benefits are becoming more prominant in the age of generative AI, as limited public-accessible data are available for generative model training, and concerns about privacy are growing. With collaborative ML, developers can have the access to an increasing amount of private data for building better models in terms of accuracy and customization. As main collaborative ML workload is performed on client devices with client data, it is also natural for clients to continue use trained local model for inference with better customization and privacy.

However, the large volume and heterogeneity nature of the client machines lead to challenges in device resource management. Collaborative ML developers usually need to build custom server-client protocols and softwares to implement custom algorithm to enforce client selection logic, which is time-consuming and cost-prohibitive. If individual job framework requires a fixed set of resource due to the lack of resource management, client resources could be underutilized, as some jobs may not be able to fully utilize the resources allocated to them. Resource sharing could be beneficial in improving resource utilization, but it could also lead to resource contention and thus long training time, as multiple collaborative ML jobs are running on the same client corpus, requesting resources with the best performance and source data quality.

On top of the resource management issues present in collaborative ML, ML plan (weights, training parameters) delivery and result collection (gradients, output of model) across large number of jobs and clients present a scalability challenge. Typical number of simultaneously participating clients required for a single job could be in the order of tens to tens of thousands. Also, low bandwidth network between client edge devices and parameter servers leads to long communication latency, creating model update and compute ramp-up delay.

Existing resource management techniques and frameworks fail to rise up to the challenge. Conventional resources in the cloud (GPUs in data centers, data lakes) often have better characteristics than client machines in terms of availability and capacity. Federated learning frameworks (Flame [1], FedScale [4], etc.) mostly focus on optimizing single-job training performance, and do not consider the resource management and sharing among multiple jobs.

To solve these problems, we propose a novel system, Propius, that can adapt to the heterogeneity of client machines, and efficiently manage and control the computation flow between ML jobs and edge resources in a scalable fashion.

Propius is comprised of a control plane and a data plane. We propose a novel control plane design where the scheduling mechanism is tailored to heterogeneous and transcient resources in collaborative ML setting. Contrary to traditional resource scheduling system where resources are generally static and long-lived, Propius resources, such as edge devices, are short-lived and dynamic. This challenge is addressed by the soft-state design principle of the scheduler, where only the states of clients in a sliding window are maintained in the system. Most of the scheduling states and decisions are mounted on the job framework ends, whose states and scale are more stable and manageable. Specifically, there are two modes of scheduling in Propius scheduler, which scheduling policy plugins can choose from: (1) online scheduling, and (2) small-batch scheduling.

Propius data plane enables scalable collaborative ML plan sharing and result collection for large number of jobs and clients. Optimization algorithms for collaborative ML, such as FedAvg [5], are supported in Propius data plane, as long as they satisfy associativity and communitivity.

In summary, Propius provides several benefits: it

1. Hides the details of heterogeneous client resource management so developers can focus on developing learning and analytics models and algorithms instead.
2. Avoids allocation conflict and maximizes resource utilization by coordinating among multiple collaborative ML jobs.
3. Allows jobs to cherry-pick desired amount of resources in terms of device system attributes.
4. Supports various scheduling algorithms for different performance objectives.

2 Propius Overview

Propius consists of two main components: the control plane and the data plane.

The control plane is responsible for managing the job and client requests, and providing coordination during the resource allocation process.

The data plane is responsible for distributing execution planes from the cloud to geo-distributed clients, collecting updates or outputs from clients, and performing reduction operation during results collection.

Propius is designed to support a wide range of job frameworks, as long as they fall into the category of server-client topologies and adopt round-based communication pattern where there are a one-to-all model plan distribution phase and a all-to-one reduction (aggregation) phase in each round. Propius does not support hybrid topologies where clients communicate with each other [1].

Through Propius data plane, individual job framework has the access to current round results computed by the data plane through a simple API call. Each job framework needs to provide a reducer plugin for the data plane to use during the reduction phase. All the reduction computation from the contributions of all selected clients for the current round and communications are offloaded to the data plane, and job frameworks are only responsible for the model update step at the end of each round for better customization and generalizability of the data plane. However, the convenience comes at a cost. Job frameworks do not have access to individual client constributions for the current round as well as for the past rounds. If the job frameworks require those information, they should turn off the data plane offloading, and use their custom implementation to collect and perform aggregation.

3 Propius Control Plane

Propius controller consists of three layers: 3.1 application layer, 3.2 scheduling layer, and 3.3 binding layer. Propius application layer handles job requests, performs admission control, and manages job state in the application layer. Propius scheduling layer executes a scheduling policy based on real-time information of job states and client resources in the scheduling layer. It groups active jobs, and assigns priority between or within job groups. Propius client manager handles client requests, and allocates available and eligible client resources to outstanding jobs in the binding layer. Multiple client managers can be instantiated to distribute the load from client requests.

3.1 Application Layer

Job manager and job database are located in the application layer, serving job frameworks.

1. *Job framework*, or simply a job, is a specialized workload for collaborative ML that regulates the ML learning process and manages the communication between clients. An example of a job framework is FedScale parameter server [4]. Currently, Propius supports job frameworks that are round-based, with specific demands and constraints on client resources in round granularity. Job constraints are the minimum requirements on client CPU, memory,

Table 1: Endpoints Exposed by Job Manager.

Endpoint	Description	Key Parameters	Returns
JOB_REGIST	Register a new job	total_round est_demand, public_constraint, private_constraint	ack, job_id
JOB_REQUEST	Request a new round	job_id, demand	ack
JOB_END_REQUEST	End a round request	job_id	ack
JOB_FINISH	Finish a job	job_id	ack

Table 2: Key Fields in Job Database Schema.

Field Name	Type	Description
time_stamp	numeric	job registeration time
total_sched	numeric	total time spent in waiting for clients over all rounds
start_sched	numeric	last timestamp when the job starts a new round
job_ip	text	IP address of the job
port	numeric	port number
total_demand	numeric	total number of clients estimated
total_round	numeric	total number of rounds estimated
attained_service	numeric	total number of clients attained
round	numeric	round number
demand	numeric	number of clients requested in this round
amount	numeric	number of clients attained in this round
score	numeric	priority score
public_constraint.[x]	numeric	lower bound for client public attribute value for constraint x
private_constraint.[y]	numeric	lower bound for client private attribute value for constraint y

etc., and demand is the desired number of clients per round. Constraints are specified by individual job frameworks, and can be classified into two types: public constraints and private constraints. Public constraints are enforced by Propius upon client attributes that clients are willing to share (CPU, OS version, etc.), while private constraints are enforced upon attributes that clients are reluctant to share, such as private metadata on datasets. Propius will pass the private constraints to clients for local binding decision.

2. *Job manager* interfaces with jobs over WAN, maintains job states, collects job metadata (number of rounds, IP and port, etc.), constraints and demands on client resources in the job database. Additionally, the job manager performs admission control, rejecting job requests if their demand exceeds allocation limitation during initial registration. It forwards successful job requests to the scheduling layer via RPC calls, which initiates scheduling, allocation and binding process. The endpoints the job manager exposes are listed in Table 1.

3. *Job database* keeps track of job states, and maintains job metadata. It is a key-value datastore indexed by job ID, accessbile to the job manager, the scheduler, and the client manager in the control plane. It also serves as the means of communication between the three control plane layers, where scheduling decisions are stored by the scheduler, and retrieved by the client manager. Key fields in the job database schema are shown in Table 2.

3.2 Scheduling Layer

On a high level, Propius employs a two-level scheduling process: global scheduling, and client binding. Propius scheduler is responsible for making centralized global scheduling decisions on coarse client resource allocation. Clients can further perform fine-grained binding decisions based on local, private attributes and the constraints of tasks offered by Propius.

Propius should be highly scalable, capable of handling large number of clients and jobs, potentially in the order of millions of concurrent client tasks. It will be expensive to keep track of every client status in the system. On top of

Table 3: Key API for Scheduling.

API	Description
job_db_portal.exist(job_id)	Query whether a job exists in the system
job_db_portal.get_job_size()	Get the number of jobs in the system
job_db_portal.get_field(job_id, "[field_name]")	Get the field value of a job
job_db_portal.query(query_str)	Get a list of jobs that satisfy query string
job_db_portal.set_score(score, job_id)	Set priority score for a job
client_db_portal.get_client_size()	Get the number of clients within the system window
client_db_portal. get_client_proportion(public_constraint)	Get the proportion of clients within the system window of which attributes satisfy the public constraint
client_db_portal.get_client_subset_size(query_str)	Get the number of clients within the system window of which satisfy the query string

the scalability challenge, the transient and dynamic nature of clients in collaborative ML settings makes it difficult to maintain correct client states. To ensure state correctness, the system needs to communicate with clients frequently, which will incur huge communication traffic. These problems motivate us to move away from traditional stateful design, such as Slurm [8], to a soft-state design.

Propius scheduler can be configured with two modes of scheduling: (1) online scheduling, and (2) small-batch scheduling. In online scheduling, Propius does not track client states at all. Instead, scheduling policy plugins pre-compute a priority score for every job, and store the scores in the shared job database. This action can be triggered by job registeration or job request event, or asynchronously, which is defined by scheduling policies. Policy plugins can compute the score from job metadata, and global request and demand statistics by calling the set of APIs exposed by Propius scheduler service (Table 3). After the scores are stored in the shared job database, client manager will pair newly-arrived clients to requesting jobs with the highest scores while ensuring the job public constraints are satisfied, and send job private constraints to the clients for client-side binding.

Online scheduling mode binds available clients to outstanding jobs as soon as their requests arrive. It requires a priority order amoung requesting jobs enforced by scheduling policy plugins so that the binding layer can make fast decisions for hot requests from clients. This, however, could lead to head-of-line (HOL) issues, and inflexible policy design due to the ordering requirement. For example, it will be difficult to implement fair-sharing scheduling in online scheduling, as it requires the plugins to re-compute the entire score-based priority ordering based on real-time global allocation status at a high frequency.

In small-batch scheduling mode, Propius binding layer stores temporary states of available clients in a data cache, from which requesting jobs can select eligible clients through querying. Requesting jobs can be grouped into multiple job groups by scheduling policy plugins. Each job group can specify a set of constraints on client resources through query strings. Priority order could also be maintained within each job group by plugins. The cache can be viewed as a sliding window of client states, where only the most recent available client states are stored. Once the client is selected by a job, it is removed from the cache, and the cache is updated with new client states. This prevents requesting jobs being paired with stale clients. Through specifying querying strings for each job group, the policy plugin can partition the resources among job groups, avoiding HOL blocking as well as enforcing job constraints on resources at the same time. This follows the partition scheduling paradigm introduced in POP [6], as the scheduling problem we are solving is a granular allocation problem, where each job requests a small fraction of resources, and resources are fungible.

The job group partitioning has other benefits. It enables more general scheduling policies by removing the necessity of pre-computing priority scores, and reduces overall reduced scheduling runtime. The flexiblity derives from the fact that a priority order between any two jobs is not required, and the policy plugin can arrange these jobs in different job groups, selecting resources from different client partitions. By breaking the scheduling problem into sub-problems, a super-linear runtime speedup can be achieved, and these sub-problems can be executed in parallel [6]. The downside of small-batch scheduling is pro-longed waiting time for avaialble clients, leading to higher probability of task failure as clients may become offline during the waiting.

Note that both the scheduling order and the job constraints are considered in these two modes. Propius scheduler service will enforce the job constraints automatically without the need for explicit binding rules for policy plugins to specify. Policy plugins are only responsible for computing the priority order of jobs, or partitioning the resources among job groups. Preemption is not supported in the current design, as the resources are fungible, and the communication overhead (between the cloud and edge) is high. Only one task will be placed onto one client resource at a time, as the resource is typically weak in computation power. Propius employs a two-level scheduling process. Contrary to

Mesos [2] two-level scheduling, where the scheduler assigns resources to jobs frameworks and asks jobs to reject the resources if they do not meet the constraints, Propius let individual client resources to select suitable jobs based on private attributes and job constraints, empowering clients in the binding process for collaborative ML tasks and reducing round-trip traffics between individual job frameworks and clients.

3.3 Binding Layer

In binding layer, Propius client manager conducts the first level scheduling, binding tasks from job frameworks to client resources, based on the priority score (online scheduling) or group partitioning (small-batch scheduling) computed by the scheduling layer. Each active and free clients, recieves metadata for one or more framework tasks if eligible, and decides which task to accept based on the task constraints on private attributes. Propius offers a client library that can be integrated into client-side ML application, and handles the binding process and communication with the Propius binding layer. Client-side application can specify the decision logic in a plugin function that is called by the Propius client library. After determining which task to accept, the client sends the information back to the binding layer. The binding layer updates the job database with the binding result, incrementing the allocation counter for the job, until the allocation count reaches the requested amount of resources by the job framework at the begining of the round.

The client manager stores client metadata in shared client database (accessible by job manager and scheduler) using a sliding window scheme, such as public attributes (CPU, OS version, etc.) and availability timestamps. Global statistics on client resources can be captured by the scheduling layer to compute the priority score for each job, and can be queried by individual job frameworks for dynamic task planning. Schedulers or job frameworks can call $client\_db\_portal.get\_client\_proportion(attributes\_threshold)$ to get an idea of the global client resource distribution and eligible client subset size. The metadata store is also useful for general monitoring and logging purposes. In addition to the client database, the client manager caches the most recently available client information in small-batch scheduling mode. As mentioned in §3.2, scheduler plugins can provide a query for each job group for client resource partitioning. The client manager performs the query on the client cache, and assigns job partitions to queried clients.

To handle the large number of client requests, Propius binding layer can be horizontally scaled to distribute the load among multiple client managers. A load balancer is placed in front of the binding layer. The database and cache is sharded along with every instance of the client manager.

4 Propius Data Plane

To enhance the performance and efficiency of content delivery for collaborative ML, we propose a content delivery policy that can be integrated into existing content delivery networks (CDNs). Our approach aims to optimize the distribution and retrieval of content, particularly in scenarios requiring dynamic updates and low-latency interactions, such as federated learning models or large-scale data processing tasks.

The core of our policy revolves around leveraging the inherent capabilities of CDNs while introducing specific mechanisms tailored to the needs of collaborative ML. By doing so, we ensure that content delivery not only meets the demands of high availability and reliability but also aligns with the specialized requirements of collaborative ML.

4.1 Data Plane Protocol

The data plane protocol for collaborative ML involves several key steps to ensure efficient content delivery and processing. The protocol is designed to facilitate the distribution of model execution plans and aggregation of results while minimizing latency and optimizing resource usage. The steps are as follows:

1. **Client Request:** A client initiates a request for an execution plan from the nearest leaf server.
2. **Model Check:** The leaf server verifies if it has the requested model or execution plan in its cache.
3. **Pull Caching:** If the model is not available in the leaf server, it sends a request to its parent server to retrieve the model.
4. **Plan Distribution:** Upon obtaining the model, the leaf server sends the execution plan to the client and caches the plan for a specified time-to-live (TTL).
5. **Result Upload:** The client processes the execution plan and uploads the results to the leaf server.
6. **Result Aggregation:** The leaf server aggregates the results received from clients.
7. **Periodic Update:** The leaf server periodically sends partially-aggregated results to its parent server to maintain updated information across the network.

The caching strategy within this protocol involves decisions on the content to cache, the duration of caching (TTL), and eviction policies. This ensures that frequently accessed models and execution plans are readily available while managing storage resources efficiently.

Support for aggregation algorithms such as Federated Averaging (FedAvg) [5] is integral to this protocol. The protocol supports any aggregation algorithm that satisfies associativity and commutativity, allowing for flexibility in optimization methods. Optimization algorithms will be implemented by individual job frameworks, ensuring compatibility with various task requirements.

4.2 Integration with Existing CDNs

The proposed data plane protocol is designed to integrate with existing CDN infrastructures. It leverages the established server selection algorithms and load balancing mechanisms inherent in CDNs to achieve cost savings and enhance performance. Specifically:

1. **Server Selection:** The protocol utilizes existing server selection algorithms to route client requests to the most appropriate leaf server. This ensures efficient handling of requests and minimizes latency.
2. **Load Balancing:** By relying on the CDN's built-in load balancing capabilities, the protocol ensures that the load is distributed evenly across the network, optimizing resource utilization and avoiding overloading individual servers.
3. **Cost Efficiency:** The integration with CDNs enables cost-effective content delivery by taking advantage of the CDN's infrastructure and pricing models. This approach reduces the need for additional infrastructure investments while maintaining high performance.

In summary, our data plane protocol enhances the content delivery process for collaborative ML by leveraging existing CDN infrastructure and introducing specific mechanisms to address the unique requirements of collaborative ML. This integration ensures improved performance, cost efficiency, and adaptability to various content delivery needs.

5 Implementation

Propius is implemented in Python based on a microservice architecture. We use gRPC for communication between services, and Redis database for state management.

6 Evaluation

We evaluate the effectiveness of Propius control plane on various metrics using simulated client traces from [4] and multi-tier jobs, with variance in demands and resource requirements. We choose two baseline implementations of collaborative ML resource selection: static partitioning, and pure random selection. We compare our system with these two baselines in terms of throughput, resource utilization, and job completion time.

In addition, We study the performance of different scheduling algorithms in terms of throughput, resource utilization, job completion time, request completion time, execution completion time, and failure rate.

6.1 Methodology

Static partitioning: Every client is assigned to a fixed parameter server based on its attributes and server constraints. The number of assigned clients is proportional to the server's demand. There is no resource sharing among jobs since the partitioning is static.

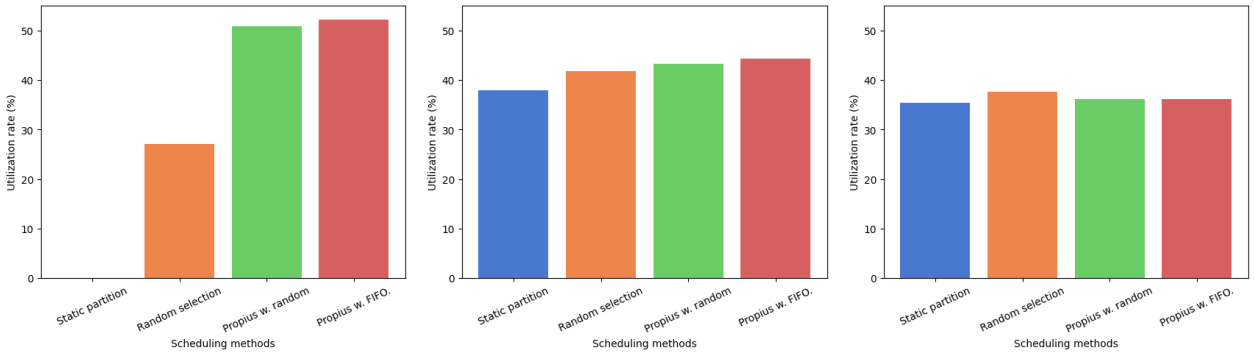
Pure random: Every clients select a random parameter server to bind to. Parameter server conducts client selection based on its resource constraints, including both public and private constraints. Clients do not know which job is actively selecting clients prior to making binding request. This is the resource selection method implemented by Apple [7] and Meta [3].

Metrics: Throughput, resource utilization, job completion time

6.2 Results

Resource utilization: Through resource sharing, random selection and Propius selection outperforms static partition in terms of resource utilization, especially when available resources are scarce (Fig. 1). When there are 1500 active clients, random selection achieves a 1.1× improvement in resource utilization compared to static partition, while

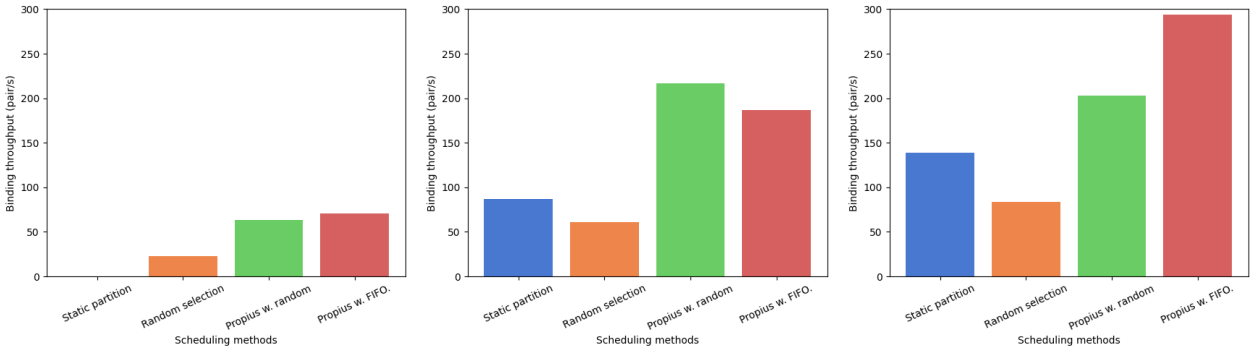
Propius selection with random policy achieves a 1.14× improvement. Propius's additional contribution to resource utilization can be attributed to its centrally managed pairing process of jobs with resource constraints and clients, as well as dynamically adjusting client allocation based on job demand. The resource-sharing benefit is more pronounced when the available clients are scarce. With 1200 active clients, static partition selection cannot meet the demand of individual jobs, resulting in failed runs. Propius's efficient management of resources allows it to improve client resource utilization by 1.88× compared to random selection under pure random scheme. As the number of clients increases, every selection method achieves relatively similar resource utilization rate.



(a) Client number: 1200 (b) Client number: 1500 (c) Client number: 1800

Figure 1: Resource utilization of different scheduling method. Resource utilization is defined as the average client partitipation time (contributing to a job) divided by the overall job completion time.

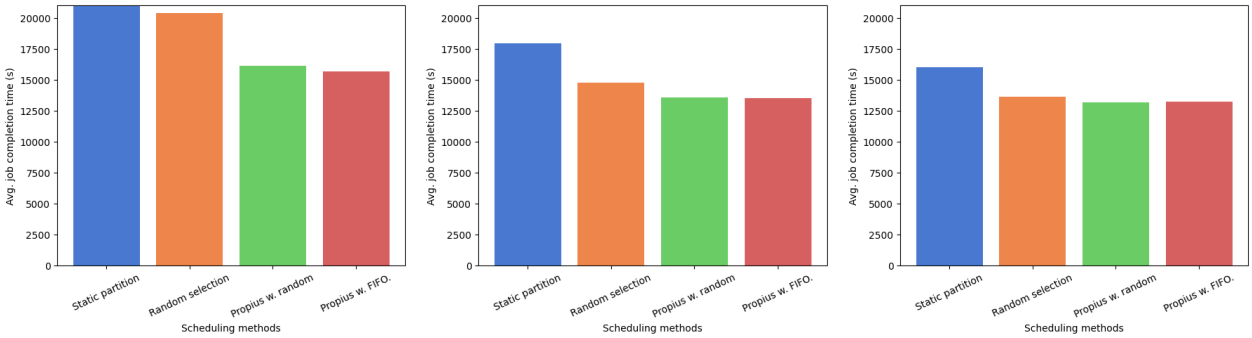
Throughput: When there are ample resources available for each job, static partition method outperforms random selection in terms of binding throughput as each client is dedicated to a specific workload, removing the pairing handshake process in random selection. Yet Propius performs the best, as is shown in Fig. 2. Under random policy, Propius achieves a 2.76× improvement in throughput compared to random selection at best. This is because Propius's central scheduling scheme could efficiently maps available clients to active jobs in time, while clients have to execute multiple handshakes to find suitable jobs in random selection. Also, Fig. 2 also shows the robustness of Propius under varying loads.



(a) Client number: 1200 (b) Client number: 1500 (c) Client number: 1800

Figure 2: Binding throughput of different scheduling methods. Throughput is defined as the number of client-job binding per second.

Job completion time (JCT): With the benefit of resource sharing and efficient pairing, Propius reduces the average JCT by at most 1.264×. It is also interesting to notice that random selection outperforms static partition method in JCT, even though it has a lower binding throughput. This can be explained by the fact that resource sharing prevents client resources with better performance being monopolized by certain jobs, and poor-performing clients being stragglers for the entire duration of a job workload.



(a) Client number: 1200 (b) Client number: 1500 (c) Client number: 1800

Figure 3: Average job completion time of different scheduling methods

7 Conclusion

In this paper, we propose a novel collaborative ML system, Propius, that can adapt to the heterogeneity of client machines, and efficiently manage and control the computation flow between ML jobs and edge resources in a scalable fashion. Propius is comprised of a control plane and a data plane. The control plane enables efficient resource sharing among multiple collaborative ML jobs and supports various resource sharing policies, while the data plane improves the scalability of collaborative ML model sharing and result collection. Evaluations show that Propius outperforms existing resource management techniques and frameworks in terms of resource utilization (up to 1.88×), throughput (up to 2.76×), and job completion time (up to 1.26×).

References

[1] Harshit Daga, Jaemin Shin, Dhruv Garg, Ada Gavrilovska, Myungjin Lee, and Ramana Rao Kompella. Flame: Simplifying topology extension in federated learning, 2024.

[2] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[3] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, Kaikai Wang, Anthony Shoumikhin, Jesik Min, and Mani Malek. Papaya: Practical, private, and scalable federated learning. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 814–832, 2022.

[4] Fan Lai, Yinwei Dai, Sanjay S. Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Fedscale: Benchmarking model and system performance of federated learning at scale. In *International Conference on Machine Learning (ICML)*, 2022.

[5] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[6] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[7] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, Sudeep Agarwal, Julien Freudiger, Andrew Byde, Abhishek Bhowmick, Gaurav Kapoor, Si Beaumont, Áine Cahill, Dominic Hughes, Omid Javidbakht, Fei Dong, Rehan Rishi, and Stanley Hung. Federated evaluation and tuning for on-device personalization: System design & applications, 2022.

[8] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.