University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Prototypical Implementation of the Node Selection Strategy in Federated learning

Chenfei Ma
Zurich
Student ID: 21-740-816

Supervisor: Alberto Huertas,Chao Feng
Date of Submission: Sept 27, 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

ML (Machine Learning) Technologien entwickeln sich rasch weiter und werden für verschiedene Anwendungen immer breiter angenommen. Anders als beim traditionellen Modelltraining, bei dem alle Benutzerdaten auf zentralisierten Servern gespeichert werden müssen, hat sich FL (Federated Learning) wegen seiner Fähigkeit, Daten verteilt zu speichern, großer Beliebtheit erfreut. Mit FL trainieren dezentrale Geräte ihr Modell lokal mit ihren Datensätzen und aggregieren das lokal trainierte Modell mit dem globalen Modell. Während des FL-Prozesses ist das Modelltraining und die Konvergenzrate eng mit dem Zustand der Geräte verbunden, die am Training teilnehmen. Daher untersucht diese Arbeit Strategien zur Knotenauswahl im FL und konzentriert sich dabei sowohl auf zentrale (CFL) als auch auf dezentrale (DFL) Umgebungen.

In Anlehnung an bestehende Forschungen und die Beiträge von Che[1] und Sultana[2] wird ein neuer Algorithmus vorgestellt. Dieser Algorithmus quantifiziert dynamische Knotenmerkmale wie Rechenleistung, Latenz und Knotenalter und erstellt für jeden Knoten eine spezifische Bewertung. Durch das Zusammenführen dieser Bewertungen mit einem probabilistischen Auswahlverfahren wird jedem Knoten, unabhängig von seinen Merkmalen, eine Chance zur Auswahl garantiert. Eine solche Strategie fördert nicht nur die Gleichstellung der Knoten, sondern beschleunigt auch die Konvergenz im FL."

Machine learning (ML) technologies are rapidly advancing and being widely adopted for various applications. Unlike traditional model training, which needs to store all user data in centralized servers, federated learning (FL) has become popular due to its ability to store data in a distributed manner. With FL, decentralized devices train their model locally over their datasets and aggregate the trained local model with the global model. During federated learning process, the model training and convergence rate is closely related with the devices condition which participant in training. Therefore, this thesis investigates node selection strategies in federated learning, focusing on both centralized (CFL) and decentralized (DFL) environments.

Drawing inspiration from existing research and the contributions of Che[1] and Sultana[2], a novel algorithm is introduced. This algorithm quantifies dynamic node features such as computational power, latency, and node age, producing a specific score for each node. By merging these scores with a probabilistic selection approach, every node, regardless of its attributes, is ensured a chance for selection. Such a strategy not only promotes equity among nodes but also encourages faster convergence in federated learning.

Contents

Chapter 1

Introduction

1.1 Motivation

The rise of Artificial Intelligence (AI) and Machine Learning (ML) is a significant milestone in the development of computing power, affecting various industries. These technologies are increasingly critical in today's big data environment, where the amount of information we generate is doubling roughly every two years. Projections show that data from Internet-of-Things (IoT) devices will significantly increase in the future. Therefore, it is important to effectively utilize this data for machine learning applications. However, there are several challenges involved in this task. The data from IoT devices is often spread across multiple locations or nodes, which complicates matters due to limited communication bandwidth and concerns over data privacy. Additionally, regulations in different countries make it difficult to centralize the data for traditional machine learning approaches.

Traditional machine learning methods usually pull all their data into one central location for easy access during training. But as the world is moving more towards a distributed or decentralized approach, this central way of gathering data is becoming less practical. It's not just costly in terms of communication; It also poses risks such as single points of failure, where the entire system could either fail or experience slowdowns. Plus, this centralized approach often overlooks new, stricter laws about data privacy and where data can be sent or stored across countries. In 2016, a new method called Federated Learning came along to help solve many of these problems[3]. It lets multiple clients or nodes work together to train models without sending all their data to one place. This avoids the issues with centralizing data and makes better use of data that's spread out all over the world, while keeping that data more secure and lowering communication costs. The basic steps of federated learning is shown as Fig.1.3.

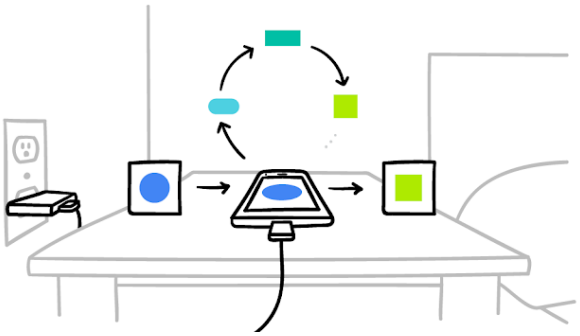

Figure 1.1: Federated learning example

Federated Learning itself has two primary variants—Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL). The example graph is shown as Fig. 1.2. [4] CFL is the more common approach, involves a central server and only for the purpose of model aggregation rather than data collection. While this preserves client data privacy, it it does come with some drawbacks, such as single points of failure and bottlenecks that can adversely affect the scalability and robustness of the federated learning system. Recognizing these challenges, the concept of Decentralized Federated Learning (DFL) emerged in 2018 as an evolution of CFL.[5] DFL eliminates the need for a central server altogether, instead allowing for decentralized aggregation of model parameters directly between neighboring participants [6]. This method uses local computer power more efficiently and makes the whole system more robust by allowing learning to happen locally and through direct exchanges between participants.

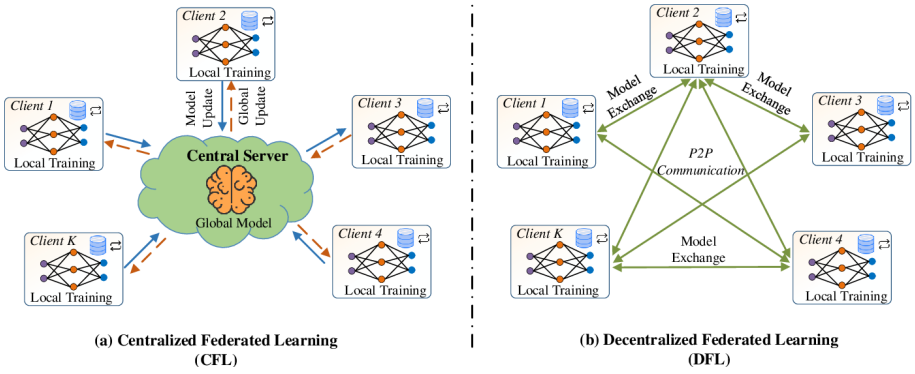

Figure 1.2: Illustration of Structure for CFL and DFL

Despite the advantages offered by DFL over CFL, one big question that hasn't been fully answered is how to best choose the clients that participate in each round of training. Currently, many systems just pick clients randomly or selecting all the clients of participant. These approaches often overlook key factors that could make the learning process more efficient. For example, how much computing power does each client have? Are they reliable? What is the quality of their data? Even scheduled tasks that could interfere with a client's computing abilities or how busy their network is at the moment could make a difference. It is important to note that in a DFL network, it could have a mix of powerful

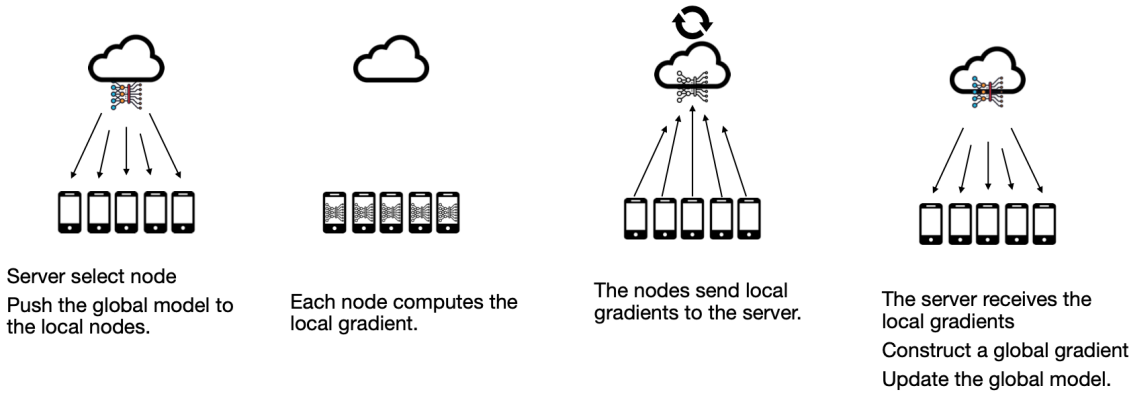

Figure 1.3: Basic steps in federated learning

data centers and smaller, less capable devices like Internet-of-Things (IoT) devices. So, picking the right clients for each round becomes even more critical.

This research gap is the driving force behind this study. This study aim to develop better methods for choosing clients in DFL training rounds, taking into account important factors like computing power, data traffic, and the model performed. The end goal is to make each training round work better, improving both the system's overall performance and its robustness. This study considers that making smarter client selections will speed up DFL and make it more reliable. This will help DFL be more useful in real-world scenarios compare to traditional CFL selection method.

Another key motivators for this research is the need for an empirically-validated approach to node selection in Federated Learning, specifically in both Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL). While various theoretical strategies for client selection have been proposed,[7] [8] there is few in practical applications that rigorously test these strategies. To address this, this study goes beyond theoretical models by incorporating node selection into an framework. This allows us to directly evaluate the effectiveness of different client selection algorithms. Furthermore, we aim to make the selection process more interactive and user-friendly, thus facilitating broader adoption and testing.

In summary, this work seeks to contribute to the field by offering a comprehensive guide and a hands-on tool for smarter, empirically-tested node selection across different Federated Learning approaches and node selection strategy.

1.2 Description of Work

This Master's Thesis focuses on addressing the complexities of CFL and DFL, specifically in node selection. The scope includes multiple related phases.

In the first phase, this work conducted a comprehensive review of the current state-of-the-art in CFL and DFL. This review is essential for gaining a thorough understanding of the existing landscape, including its logic, algorithms, and solutions utilizing CFL and DFL. In addition to comprehending the fundamental mechanisms that facilitate FL, particular emphasis was placed on exploring literature discussing methods for dynamically and flexibly selecting nodes within a federated learning. The primary objective of this initial step is to thoroughly examine existing approaches for node selection in Federated Learning. By identifying factors considered by these approaches and uncovering any unresolved issues or limitations, this study can then set the stage for subsequent stages of the research by determining which aspects need to be addressed.

In the second phase, the work aims to create a comprehensive taxonomy for node selection mechanisms in federated learning (FL). This include both centralized and decentralized methods. While there is existing literature on centralized FL, this work extend those findings to the context of decentralized FL. the goal is not just to adapt existing taxonomies but to develop a new strategy that specifically addresses the conditions and requirements for both CFL and DFL.

After completing the taxonomy, the third phase focuses on designing a solution that effectively monitors and combines selected aspects for node selection in federations. The criteria for selecting these aspects was based on the developed taxonomy. This design process primarily involves making decisions about what aspects to monitor, how to monitor them, and how frequently they should be updated. It is crucial to note that the proposed solution aims to be dynamic and adaptable to changes in both the federation environment and participant behaviors.

Entering the fourth stage, the work focuses on the practical aspects—implementation and deployment. Specifically, the node selection solution designed in the third phase is intended to be integrated into the Fedstellar DFL framework [9]. Furthermore, a suitable use case and dataset are identified to validate the performance and appropriateness of the designed solution. It is crucial for this validation phase to incorporate a dynamic federation scenario where nodes are chosen based on their real-time status. Additionally, this stage involve choosing metrics that are aligned with the goals of the research to compare the proposed solution against existing strategy.

At the end of this project, the study aims to produce a working example that fulfills all the initially set goals. In addition, this study provides detailed documentation that explains each decision made during the design and setup process.

In summary, this Master's Thesis trying to bridge the gaps in existing research on DFL by focusing on optimized node selection. This work aims to contribute to the field of node selection in DFL by conducting a rigorous literature review, creating a taxonomy, designing and implementing solutions, and validating them. Each stage builds on the last to develop a solid understanding of node selection. By so doing, this work aims to make contribution to the field, providing insights and tools that could be invaluable for future research and practical applications.

1.3 Thesis Outline

The thesis consists of five parts: introduction, related work, main chapter, evaluation, and summary/conclusion.

1. Introduction: This section describes the motivation for the study and provides an overview of the thesis structure and work arrangement.

2. Related work: In this section, first thing is to introduce the background about the federate learning and the place of node selection strategy in federate learning. Later, the recent works on CFL, DFL, and node selection strategies in federated learning are introduced. These works focus on different aspects such as effectiveness or adaptiveness of the strategy.

3. Main chapter: The priority selection algorithm is introduced along with its rationale. The integration steps of this strategy into the Fedstellar framework are explained in detail. Additionally, two other selection strategies to be used for later evaluation are listed, along with information about customized configuration and frontend modification implementation.

4. Evaluation: This section explains how decisions were made to build the testing group for evaluation purposes. It also presents a comparison of results from experiments and attempts to explain their significance in demonstrating the effectiveness and efficiency of the proposed strategy.

5. Summary/Conclusion: A comprehensive statement is provided summarizing all completed work and achieved results thus far. Finally, suggestions for future improvements that could enhance this study are presented.

Chapter 2

Related Work

2.1 Background

Machine Learning (ML), a subset of Artificial Intelligence (AI), stands to gain significantly from the data explosion anticipated in the near future. Yet, in many settings like smart city sensor [10] and health monitors [11], those data from countless IoT devices are usually distributed across various locations. Constraints such as limited bandwidth, data privacy concerns, and national laws make it difficult or even counterproductive to gather this data in one central location, as is commonly done in traditional ML methods.

Federated Learning is an innovative approach to machine learning that addresses the challenges of data privacy, distributed storage, and communication overhead inherent in our data world. Emerging as a breakthrough in 2016 [3], Federated Learning enables a network of decentralized nodes to collaboratively train machine learning models without requiring raw data to be sent to a central server. By keeping data localized, Federated Learning not only minimizes data transmission costs but also enhances data security and privacy, complying with increasingly stringent regulations. It allows the network to learn from valuable, possibly sensitive, distributed data, without moving them to a central server.

Federated Learning can be categorized into two main types: Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL). In CFL, a central server is employed for model aggregation. This server initiates the global model, distributes it to all participating nodes, and is in charge of aggregating the updated local models. These local models are derived from data at each node, and their collective knowledge refines the global model on the central server. The centralized approach offers advantages such as streamlined communication and easy management, as well as efficient resource allocation, enhancing computational efficiency. However, it also presents challenges, including a single point of failure and increased vulnerability to cyber-attacks, which could compromise data privacy.

DFL, introduced in 2018, [6], [12] enables decentralized aggregation of model parameters without a central server. In this approach, nodes collaborate through local computation

and by exchanging model parameters among themselves. DFL offers its own set of advantages compared to CFL. Firstly, it is self-scaling; newly joined nodes communicate with their neighbors in the same way as existing nodes, maintaining the network's overall structure. Secondly, the lack of a need for central server infrastructure means that all computations are performed locally on each node, eliminating some of the vulnerabilities associated with CFL.

Despite the promising benefits, Federated Learning still faces numerous challenges such as efficient node selection and ensuring quality of service for ongoing research and development.

In the realm of Federated Learning, there are two primary categories for consideration: Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL). Each has its own approach to node selection, which is a key aspect for determining the quality of the global learning model.

In CFL, a central server takes the lead in selecting nodes for each round of model training. There are several common strategies for doing this. One popular method is random sampling, where the central server selects nodes at random for each training round. This approach is computationally efficient, making it easier to manage from a resources standpoint. However, its downside lies in the potential for creating an imbalanced or less accurate global model because the selection is random. Another method is comprehensive, or 'all' sampling, where every available node is selected for each training round. While this may yield a more balanced and accurate global model, it comes at the cost of using more computational resources and bandwidth. This could result in inefficient data aggregation and increased network overhead.

In DFL, node selection is more complex due to its decentralized nature. In this setup, each node independently selects its neighboring nodes for data aggregation and model training. Because each node has autonomy in making this choice, the criteria for selection can differ from one node to another. This demands a selection strategy that is adaptive to each node's specific circumstances.

Both CFL and DFL face their own set of challenges when it comes to node selection. The central question is how to scale the selection process efficiently, especially as the number of participating nodes increases, without compromising the performance and accuracy of the global model.

To sum up, node selection is a crucial element in both Centralized and Decentralized Federated Learning. It influences not just the computational efficiency of the learning process but also the quality of the global model that is produced. While each has its advantages and disadvantages, choosing the right strategy for node selection can significantly impact the success of a Federated Learning project.

2.2 Node selection in recent works

The subject of node selection in Federated Learning has garnered increasing attention in recent years from 2019. Understanding the role and strategy behind node selection is

crucial for optimizing the Federated Learning system's robustness, performance, and fairness. For instance, robustness ensures the system's resilience against network disruptions, while performance strategies aim at computational efficiency and speed of the learning process. On the other hand, fairness ensures that nodes with varying capabilities have an equitable opportunity to contribute to the global model.

In 2019, Nishio purposed a CFL protocol FedCS [8] that considers the time constraints of the framework. The key idea is to set a certain deadline for clients to communicate with the server and aggregate as many clients as possible in the given time. There is a two-step client selection, the first step is performed by the clients to update the resources information to the server such as bandwidth condition, computational power and data size. The second step is the client selection step which is performed by the server. In the client selection step, the server will estimate the time needed for clients to complete the iteration and determine which clients will be selected to join this iteration. After setting a deadline, the server can optimize the client selection by maximizing the number of selected clients based on resource information while keeping the iteration time within a time constraint.

A similar feature that is closely related to the time constraint is the bandwidth. Tang [13], Amiri [14], Ren [15], Yu [16], Chen [17], Sultana [2], and Tang [18] all consider bandwidth as an optimizing feature in federated learning. Tang2020 introduced an adaptive client selection algorithm that optimizes bandwidth usage. The algorithm's server coordinates client information and training processes, but it does not act as a central server in centralized federated learning that involves parameter aggregation. They use the modified gossip algorithm to generate a gossip matrix based on the communication speed. By only choosing the one peer that has the better bandwidth, the clients only connect with a single peer and save on communication costs. In 2023 Tang [18] purposed a DFL framework that implements the algorithm above. The author state that always choosing the best connection with clients will result in fixed peers. Therefore, randomly selecting another peer with better bandwidth is better for convergence.

Amiri [14] focuses on both the bandwidth and local updates during client selection. Since the bandwidth is limited and more connected clients will result in fewer communication resources. The goal of this paper is to select the clients to reach the best performance within the bandwidth limit. The significance of local updates is measured by the l2-norm. A higher l2-norm indicates a greater importance for labelling the client. They compare four methods based on different priorities: bandwidth only, local updates importance only, bandwidth with higher priority and local updates importance with lower priority, and local updates importance with higher priority and bandwidth with lower priority. The last method gives the best performance during the experiment which means considering both the bandwidth and local updates are necessary and local updates are having a greater impact on the convergence.

Ren [15] purposed a scheduling algorithm which has a similar focus to Amiri. They are focusing on data size, bandwidth, local updates, and unbiasedness with an extra feature which is the probabilistic selection. The importance of local updates is measured as gradient divergence in this paper. They proved that with probabilistic scheduling, the gradient would be unbiased if all the clients have a non-zero chance to be selected. In

their framework, the larger data size and larger gradient divergence in a client mean greater importance of the client to the global model. With a higher probability toward the clients which has larger data size, larger gradient divergence and better bandwidth, a faster convergence rate will achieve.

Cho [19] focuses on local updates and biased selection in a similar approach. Their approach is to select the clients with higher local loss to speed the convergence rate. The selection is in three steps. First, the server samples a subset of all clients with the Power-of-Choice strategy. Then the server sent the global model to clients and gets the local loss back. Last, the server selects the clients with the highest local loss. The paper proved that this selection algorithm increases the convergence rate compared to unbiased client selection. Ribeto [20] also focuses on the local update, while the clients have to choose whether to connect to the server based on the threshold server sent and the local update norm of the client. The server receives updates from clients who choose to send them. For clients that do not send updates, the server relies on the Ornstein-Uhlenbeck process to predict their local updates. By adaptively selecting the threshold, the algorithm can reduce the communication cost during the training period.

Chen [21] and Chen [17] both consider the local update and use a probabilistic approach for the client selection algorithm. Chen [21] takes a similar approach as others'. Using the importance of local updates as the key feature to label the selection probability of the clients and every client are having a non-zero chance to be selected. To further optimize the convergence time, they implement an artificial neural network to estimate the local parameters of those clients which is not being selected. Chen [17] similarly uses the probabilistic selection to prioritize the clients with more contributions to the global model while keeping the communication cost within limits. The nodes first calculates their model update and upload the updates to the central controller. Then the central controller determines the connection probability by updates of the nodes and distance between nodes and controller.

Onoszko [22] focuses on dealing with non-iid(non-independent and identically distributed) data. They purposed an algorithm in DFL called Performance-Based Neighbor Selection(PENS) by considering the data distribution in clients and selecting the peer clients with similar data distribution. The idea of this algorithm is first, the clients randomly communicate in the network. Then the clients will send the model to their peers. By examining the training loss of the sent model from the clients, the clients will be able to find peers with similar data distribution since the peers with similar data distribution will have a lower loss than peers with the dissimilar distribution. After having a group of peers selected from the previous step, the clients will take the top peers to be the selected peers to communicate and learn the model which is more suitable for the local data distribution.

Wu [7] also designed a probabilistic client selection algorithm to improve the convergence rate with non-iid data. They focus on finding clients with a local gradient that helps the convergence of the global gradient and exclude the clients that have an adverse effect on global updates. By iteratively removing the local updates of the clients, the server would identify the clients which adverse local updates and label the clients with a contribution expectation value. Since the expected global gradient would be higher if the server in-

cludes the adverse clients and vice versa. Within the included clients, the algorithm can further adaptively select the clients with higher contributions in each iteration using the probabilistic selection based on the expectation value.

Chen [23] considers privacy while utilizing probabilistic client selection and prioritizing local updates. The importance of the clients is defined by the norm of local updates which is similar to other studies. After calculating the updates in the clients, clients send the updates' norm to the server. The server calculates the aggregated norm of clients' updates and distributes it to them. Then, based on their local norm and probability matrix (which they calculated), the clients perform the client selection step. In addition, they focus on the privacy of the framework. By using the greedy algorithm from Wangni [24] the norm of the client updates secured aggregated in the server and the server can use only the aggregated norm to keeps the privacy nature of federated learning.

Che [1]purposed a algorithm which uses two opposite selection strategy with the same scoring system. The scoring system determines the honesty level based on the euclidean distance between the local node gradient and peer node gradient. Honest nodes receive a higher score because their gradients are closer, while one honest node and one malicious node receive a lower score that can affect node selection behavior. The selection strategies differ in their robustness. Two strategies are proposed for selecting peer nodes to aggregate the global gradient. The first strategy selects nodes with higher scores, which makes the aggregation process more robust and mitigates the impact of Byzantine attacks. However, this also makes it more difficult to learn the global data. The second strategy selects nodes with lower scores, which allows for faster convergence of global gradients and better understanding of global data. However, this strategy may result in a less robust model that is vulnerable to Byzantine attacks.

Researchers such as Yang [25], Yang [26], and Shi [27] have focused on previously overlooked features including signal-to-noise ratio(SNR), frequency of participation, and latency. Yang [25] conducted an experiment to compare three node selection algorithms: random scheduling, round robin(RR), and proportional fair(PF). The results indicate that the PF algorithm is more efficient when the signal-to-noise ratio threshold is high, while RR performs better under low signal-to-noise ratio conditions. The PF algorithm is achieved by selection nodes with the highest value of instantaneous SNR over average SNR.

Yang [26] focused on the frequency of participation feature in node selection. Their approach was to prioritize nodes that have been recently updated by assigning an "age" value to each node based on its participation frequency. During each iteration, if a node is not selected by the server, its age value increases. The selection algorithm aims to minimize the sum of all age values among nodes under all other constrains like iteration time limit or transmission power.

Shi [27] aims to reduce the training delay by minimizing the latency per iteration in the training process. This is achieved through a scheduling policy that considers both computation and communication latencies. The algorithm involves setting a latency threshold, after which the server adds up node latencies from least to most until the sum reaches the threshold. Nodes whose latencies are included in this process are selected for participation.

In 2020, Sultana [2] proposed the Eiffel algorithm which considers various features such as local updates, data size, computational power and frequency of participation. The algorithm selects nodes based on a specific process. Firstly, the server distributes the global model to all nodes. Secondly, after completing their local training step, each node uploads its resource information and local updates to the server. Thirdly, using this information provided by each node, the server calculates an overall score for every node by considering all aforementioned features. Finally, based on these scores in descending order (highest to lowest), the server determines which nodes will be selected for subsequent training and aggregation iterations. In general terms; higher loss values or larger data sizes along with more computational power and longer "age" are ranked higher in this scoring system.

Fairness plays a significant role in node selection strategies. A fair algorithm should values positive input more than negative inputs and rewards the nodes with positive inputs a more accuracy model. Researchers such as Kang [28], Lyu [29], Rehman [30], and Yu [16] have explored various approaches to address this issue.

In 2019, Kang [28] proposed a node selection algorithm that uses a reputation mechanism. The algorithm starts by calculating the reputation scores for each node based on three factors: interaction frequency, interaction timeline, and interaction effects. Interaction frequency refers to the number of interactions between nodes and the server. Interaction timeline represents how quickly the reputation score declines over time; recent interactions have a higher weight in determining scores. Interaction effects are determined by whether an interaction has a positive effect on the global model; if so, it increases the reputation score. By monitoring these three factors and calculating an overall reputation score, servers can select nodes from highest to lowest rank.

Lyu [29] proposes a different approach to applying the reputation mechanism in node selection algorithms. In this paper, nodes receive distributed models based on their reputation scores, which are calculated as follows: first, the accuracy of each node is determined by calculating its contribution to the global model using uploaded local weight updates. Next, all accuracies are normalized by the server. Finally, a sinh function is applied to these accuracies to obtain the reputation score for each node. The server then distributes more updates to nodes with higher reputation scores and fewer updates to those with lower scores. This ensures that each node receives a unique model and that its accuracy reflects its contribution to the global model.

Rehman [30] employs a distinct punishment approach for low reputation nodes. The method involves identifying adversarial nodes among all the nodes and rejecting updates from them. Adversarial nodes are detected using the local model's accuracy as a feature. Nodes upload their local models to the server, which then calculates their accuracy. After computing the mean and variation of all node accuracies, those with outlier values are excluded from global model aggregation.

The following tables are the summary of recent node selection approach and the corresponding simplified algorithms.

The abbr. are listed here to clarify the equations used in the simplified algorithms.

Recent Node Selection Approaches				
Papers	Year	Area	Features	Focus
Nishio [8]	2019	CFL	Bandwidth, Computational power, Data size, Time constrain	Robust, Performance
Kang [28]	2019	CFL	Reputation, Security	Robust, Fairness
Yang [25]	2019	CFL	Signal to noise ratio	Performance
Yang [26]	2019	CFL	Frequency of participation	Robust
Shi [27]	2019	CFL	Latency	Robust, Performance
Tang [13]	2020	CFL	Bandwidth, Gossip	Robust,Performance
Amiri [14]	2020	CFL	Local updates, Bandwidth	Robust,Performance
Ren [15]	2020	CFL	Local updates Bandwidth Probabilistic, Data size	Robust,Performance
Cho [19]	2020	CFL	Local updates, Biased	Robust,Performance
Ribero [20]	2020	D/CFL	Local updates	Robust,Performance
Lyu [29]	2020	CFL	Reputation	Fairness
Rehman [30]	2020	D/CFL	Local updates	Robust,Fairness
Chen [21]	2021	CFL	Local updates, Probabilistic	Robust, Performance
Chen [17]	2021	CFL	Local updates, Bandwidth Probabilistic	Robust, Performance
Onoszko [22]	2021	DFL	Non-IID, Gossip, Local updates	Robust, Performance
Wu [7]	2022	CFL	Non-IID, Probabilistic	Robust, Performance
Chen [23]	2022	CFL	Local updates, Probabilistic Privacy	Robust, Performance Fairness
Sultana [2]	2022	CFL	Local updates, Data size Computational power Frequency of participant	Robust, Performance Fairness
Che [1]	2022	DFL	Local updates, Security	Robust, Performance
Tang [18]	2023	DFL	Bandwidth, Gossip	Robust, Performance
This work	2023	D/CFL	Local updates,Data size, Latency Computational power Availability, Bytes Frequency of participation	Robust, Performance Fairness

Table 2.1: Recent Node Selection Approaches

Recent Node Selection Approaches			
Papers	Simplified Algorithm	Dynamic	Static
Nishio [8]	$i = \arg\min_{i \in \mathbb{K}} = \frac{1}{cp_i} + \frac{ds}{\mathbf{B}_i}$	cp, B_i	ds
Kang [28]	Frequency+Time+Effect	F,T,E	
Yang [25]	$i = \arg\max \frac{SNR_{inst,i}}{SNR_{ave,i}}, i \in N$	SNR	
Yang [26]	$i = \arg\min \sum_i^{i \in N} age_i$	age	
Shi [27]	$i = \arg_{i \in N} \min\{Lat_i^{comp} + Lat_i^{commu}\}$	Lat	
Tang [13]	$i = \{min(\mathbf{B}_{ij}, \mathbf{B}_{ji}) \vert \mathbf{B}_{ij} > \mathbf{B}_{threshold}\}$	B_i	
Amiri [14]	$1.max \parallel \Delta \mathbf{w}_i \parallel_2$ $2.B_{left} = B_{all} - B_i$	$\mathbf{w}_i$	B_i
Ren [15]	$\mathbb{P}_i = ds_i \parallel \mathsf{g}_i \parallel \frac{1}{Lat_i}$	g_i, Lat_i	ds
Cho [19]	$i = \arg\max_{i \in \mathbb{K}} \ell_i$	ℓ_i	
Ribero [20]	$i = \parallel \Delta \mathbf{w}_i \parallel_2 > \Delta \mathbf{w}_{threshold}$	$\mathbf{w}_i, \mathbf{w}_{threshold}$	
Lyu [29]	$rep = sinh(Acc(\mathbf{w} + \Delta \mathbf{w_i}))$	$\mathbf{w_i}$	
Rehman [30]	$mean, var \leftarrow acc(\mathbf{w_i})$	$\mathbf{w_i}$	
Chen [21]	$\mathbb{P}_i = \frac{\|\Delta \mathbf{w}_i\|}{\sum_{i=1}^N \Delta \mathbf{w}_i}$	$\mathbf{w}_i$	
Chen [17]	$\mathbb{P}_i = \frac{\|\Delta \mathbf{w}_i\|}{\sum_{i=1}^N \Delta \mathbf{w}_i} + \frac{distance_i}{\sum_{i=1}^N distance_i}$	$\mathbf{w}_i$	
Onoszko [22]	1. $\ell_{j,i} = Loss(\mathbf{w}_i, input)$ 2. $i = \arg_i \max(\ell_{j,i})$	$\mathbf{w}_i$	
Wu [7]	1. $i_{remove} = \langle \Delta \mathbf{w}_i, \Delta \mathbf{w}_{global} \rangle < 0, i \in N$ 2. $\mathbb{P}_i = \frac{\|\Delta \mathbf{w}_i\|}{\sum_{i=1}^N \Delta \mathbf{w}_i}$	$\mathbf{w}_i$	
Chen [23]	$\mathbb{P}_i = \parallel \Delta \mathbf{w}_i \parallel / \sum_{i=1}^N \Delta \mathbf{w}_i$	$\mathbf{w}$	
Sultana [2]	$i = \max\{\ell_i + ds_i + cp_i / resource\ demand + age_i\}$	ℓ_i, age_i	cp_i, ds_i
Che [1]	$Score_i = \frac{1}{\|\mathbf{w}_i - \mathbf{w}_j\|_2^2}$	$\mathbf{w}$	
Tang [18]	Similar to Tang [13]	B_i	
This work	$score_i^t = \frac{\|\Delta \mathbf{w}_i^t\|}{\sum_{i=1}^N \Delta \mathbf{w}_i^t}$	$\mathbf{w}_i, cp_i, Lat_i$	ds_i
	$\mathbb{P}_i^t = (\alpha score_i^t + \beta cp_i^t + \gamma ds_i +$ $\delta Lat_i^t + \lambda Bytes_i + \mu age_i^t) aval_i^t$	$age_i, aval_i$ $Bytes_i$	

Table 2.2: Recent Node Selection algorithms

cp = computational power
ds = data size
B = bandwidth
$\mathbf{w}$ = local updates
g = local gradient divergence
Lat = latency
ℓ = local loss
SNR = signal to noise
age = frequency of participation

In summary, the domain of node selection in Federated Learning has seen an increase of robust research since 2019, each contributing unique perspectives and methodologies. These studies can broadly be categorized based on the features they consider essential for

optimizing node selection, such as time constraints, bandwidth, local updates, data size, frequency of participation, and fairness. While some researchers like Nishio and Tang have proposed time-efficient frameworks that optimize the selection based on deadlines or bandwidth, others like Amiri and Ren have focused on the importance of local updates and bandwidth in achieving faster convergence rates. More algorithms proposed by researchers like Wu and Onoszko aim to tackle the challenges posed by non-iid data where non-independent and identically distributed data are more easily slowing down the convergence rate and the algorithm purposed by them could improve such convergence rate. Recent contributions also incorporate critical concerns such as privacy and fairness into the federated learning areas.

2.3 Limitation and challenges

In terms of limitations, existing research on node selection in federated learning often falls short in offering comprehensive models. Many of the current frameworks prioritize one or two performance metrics, such as loss efficiency or bandwidth optimization. While these are important factors, the absence of a more holistic approach limits these models' effectiveness in real-world applications where multiple constraints are at play.

Additionally, a large portion of the existing research has been conducted in controlled environments, making it difficult to generalize these findings to more unpredictable, real-world settings. Devices in real-world settings face variable conditions, such as fluctuating battery life and competing computational tasks. This variability can undermine the efficacy of existing node selection strategies, which are often not designed to adapt to such unpredictability.

There is also an emerging but still insufficient focus on considerations such as fairness and bias in node selection. Most of the current models are not adequately designed to ensure that all participating nodes, particularly those from underrepresented or minority groups, have an equitable influence on the global model.

When we turn the attention to challenges for new node selection strategies, a few key areas emerge. Firstly, there is an immediate need for a more comprehensive approach to optimization. Purposed work should aim to balance a range of variables, from efficiency and bandwidth to fairness and computational power, tailored to the specific needs of the application at hand. Moreover, it is essential for upcoming research to move beyond the scope of well-controlled environments to evaluate the resilience and efficacy of node selection strategies in real-world conditions. Fairness considerations, especially ensuring minimizing bias, should be integrated into the design and evaluation of new node selection algorithms. Scalability is another pressing concern; as federated networks grow in size and complexity, the algorithms guiding node selection must be designed to scale efficiently. Finally, as data privacy concerns continue to escalate, new node selection strategies must navigate the challenging trade-off between optimizing the utility of the global model and preserving the privacy of local data.

In summary, while the current landscape of node selection research has made important groundwork, there is a distinct need for more holistic, real-world applicable strategy that are scalable and adaptively adjust according to surrounding nodes..

Chapter 3

Main

In this main chapter, a detailed explanation of the purposed node selection is present within the context of Federated Learning, focusing on both Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL) modes. The chapter provides a comprehensive overview of the feature selection process for nodes, laying out criteria and methodologies for feature extraction, which serve as essential preliminary steps in design a robust node selection strategy. It further explains the underlying logic that drives node selection in both CFL and DFL modes. As part of this exploration, the following chapter is explain in detail how the proposed node selection strategy can be seamlessly integrated into the Fedstellar framework, [9] a comprehensive platform in federated computing. The integration process is described in detail for both CFL and DFL modes, ensuring that the purposed strategy is not just theoretical but also practically actionable.

3.1 Priority selection

In order to effectively implement a node selection strategy, both centralized federated learning (CFL) servers and nodes in decentralized federated learning (DFL) must ascertain the condition of their neighboring nodes. The process becomes redundant if all nodes exhibit similar conditions; hence, distinct features of each node become vital for a meaningful selection. Furthermore, it is imperative to consider inherent limitations and challenges, such as robustness and efficiency, when devising a node selection strategy.

Robustness, in this context, could be interpreted as the strategy's ability to adapt to the fluctuating conditions of the nodes. Previous research has considered various features to describe node conditions but often neglects the dynamic nature of these features. Given that the condition of edge devices can change over time, an adaptive node selection model should not only focus on convergence rates but also on capturing the fluctuating conditions of each node.

Firstly, computational power stands as a key feature, influencing the training time during learning iterations. The computational capabilities of a node are a cornerstone in determining its efficiency in federated learning tasks. A node with higher computational power

can complete training iterations more rapidly, reducing the overall time required for model convergence. However, computational power is not static; it can fluctuate based on other processes running on the node. Therefore, an adaptive node selection model should constantly evaluate each node's available computational resources to provide a more accurate estimate of iteration time.

Another crucial metric to consider is the communication latency between nodes or between the server and nodes. Higher latency during data transmission extending the time it takes to receive updates from other nodes, consequently extending the iteration period. This can result in inefficient aggregation and distribution of model updates. Therefore, an ideal node selection strategy should prioritize nodes with lower latency to optimize the overall efficiency of the learning process.

The size of data traffic sent and received by a node is a key parameter, particularly during model gossiping and aggregation phases. Nodes that can handle larger data traffic are likely to contribute more meaningful information to the global model, potentially accelerating the rate of model convergence. As a result, the size of data traffic should be factored into the selection process.

The loss metric, which quantifies a model's performance at each iteration, is another feature that can guide the selection strategy. Nodes with higher loss values indicate greater divergence between local and global models, suggesting that their data can significantly influence the global model. Therefore, nodes with higher loss should be given preference in the selection process to expedite convergence.

The volume of local data held by a node is often directly proportional to its contribution to the global model. Larger data sets usually result in more robust learning and faster convergence. Hence, nodes with larger data sets should be given preference during the selection process.

The age of a node in the network, measured by how long it has been since last selected for training, can also be a consideration. Nodes that have not been selected for an extended period, for example due to high latency, might possess unique or diverse data that could be beneficial for the global model. Older, unselected nodes should thus be weighted more heavily in the selection algorithm to ensure their data is also incorporated into the global model.

Last but not least, the availability of a node for training is pivotal. Nodes might pause for various reasons, such as low power or maintenance, which should be taken into account during the selection process. A record of each node's availability status helps in making a more informed selection.

In summary, the process of node selection invovles a multi-aspect approach, taking into account a variety of features such as computational power, communication latency, data traffic size, loss metrics, volume of local data, node age, and availability. These features are not static but rather dynamic, subject to change over time. Therefore, an effective node selection strategy must be inherently adaptive, capable of reassessing and prioritizing nodes based on these fluctuating conditions. The aim is to balance robustness with

efficiency, ensuring not only faster convergence rates but also a more resilient and equitable learning process.

With in mind of the comprehensive features outlined in preceding sections, the subsequent phase is the development of the node selection algorithm designed for both CFL and DFL environments. To effectively implement this mechanism, it is critically important to differentiate between individual nodes based on a evaluative criteria.

The concept of node ranking based on assigned scores were in existing academic literature. Specifically, the work of Che [1] and Sultana[2] provides a valuable theoretical framework, proposing the assignment of numerical scores to each node, based upon their current conditions. These scores serve as indices for a ranking system, which guides the selection process in both Centralized and Decentralized Federated Learning.

Our algorithm not only identifies key features for node evaluation but also quantifies these features to compute unique scores for each individual node. Each node's current operational state is represented as a vector of these features, enabling a multi-dimensional assessment of the node's capabilities and limitations.

We propose an algorithm that builds on Sultana's additive model to aggregate multi-dimensional feature vectors. by having a coefficient in front of each features from the node and a final availability score. the algorithm can compute the score for each of the node and use as a guide for node selection.

Distinct from Sultana's top-down selection methodology, our approach employs probabilistic selection for each node. The probability of selecting a given node is proportional to its score relative to the total score pool. This probabilistic selection serves a key purpose: it allows every node to have non-zero chance to be selected in the selection process thereby facilitating the convergence time[21].

In general, each node is assigned a composite score based on its feature vectors and adjusted by coefficients. And the score times the availability number would be the final score for the node. The probability selection process take the score as the probability and use this to select the nodes. This scoring mechanism ensures that each node's current state is accurately represented. The probability selection mechanism are used to make each node have a chance to be selected. The resulting nodes are the selected nodes, improving the training and convergence process in Federated Learning. The following is the formula 3.1 used to calculate the scores of the node i in iteration t where loss = $loss$, computational power = cp, data size = ds, latency = Lat, Data traffic size = $Bytes$, age = age and availability = $aval$.

$$score_i = (\alpha loss_i^t + \beta cp_i^t + \gamma ds_i + \delta Lat_i^t + \varepsilon Bytes_i + \zeta age_i^t) aval_i^t \tag{3.1}$$

In the context of the coefficients assigned to each feature, these mathematical factors hold significant weight, as they are coupled with the feature values to generate a composite score for each node. The calibration of these coefficients is a non-trivial task and needs careful consideration. Details on the methodology employed to determine these coefficients are discussed in the implementation section of this study. Importantly, these coefficients

are designed to be flexible, providing a framework that can be tailored to accommodate different federated learning scenarios or to optimize various performance metrics in future research endeavors.

$$p_i^t = \frac{score_i^t}{\sum_{i=1}^{N} score_i^t} \tag{3.2}$$

After the scores for each node have been calculated, the algorithm moves into a subsequent phase that involves a probabilistic node selection mechanism. Specifically, this selection process is governed by Equation 3.2, which calculates the probability of selecting each node based on its composite score relative to the sum of all nodes' scores. By integrating these carefully-calibrated coefficients and probabilistic selection methodologies, the algorithm offers a robust and adaptable solution for node selection in both Centralized and Decentralized Federated Learning settings.

3.2 Integration into Fedstellar platform

Fedstellar[9] is a versatile platform for Federated Learning that enables training of Machine Learning models on physical and virtual devices across different federation modes. Designed to operate in centralized, semi-decentralized, and decentralized settings, It effectively addresses challenges in Decentralized Federated Learning, such as communication bottlenecks and heterogeneous network topologies. Empirically, the platform has demonstrated its efficacy through deployments in cyberattack detection and benchmark datasets like MNIST and CIFAR-10. With proven performance and adaptability metrics, Fedstellar presents an excellent framework to extend and test our node selection algorithm. This section dive deep into the mechanics of integrating our node selection algorithm into Fedstellar's architecture. The aim is to enrich Fedstellar's existing capabilities by leveraging our algorithm's dynamic, real-time selection of nodes, thereby potentially improving the platform's efficiency and convergence rates.

Following the introduction to the versatile and robust Fedstellar framework, it is crucial to outline the steps required for the seamless integration of our proposed node selection algorithm. The process involves several pivotal steps, each serving a unique function in ensuring a smooth fusion with Fedstellar's existing architecture.

1. Identification and Extraction of Features: The first crucial step involves the identification and extraction of relevant features from each node within the federation. These features form the foundation for our algorithm's scoring and ranking system. It is essential to work closely with Fedstellar's existing data structures and functionalities to extract these features efficiently without compromising the integrity of the broader system.

2. Algorithmic Implementation: Subsequent to feature extraction, the focus shifts to the actual implementation of our node selection algorithm within the Fedstellar framework. This involves embedding the algorithm in such a manner that it functions coherently

within Fedstellar's existing operational flow. The implementation phase require attention to algorithmic details, including variables, functions and calculation steps, to ensure performance.

3. Integrate the Selection Process with the Original Framework: The final pivotal step is the merging of our node selection algorithm with Fedstellar's existing aggregation mechanisms. This involves establishing a robust connection between the two, ensuring that the newly implemented algorithm not only supplements but helps the efficiency of Fedstellar's original structure.

Each of these steps is expounded upon in the subsequent sections, offering details for the algorithm's integration into the Fedstellar framework. This multi-step procedure aims to improve Fedstellar's node selection capabilities substantially, thereby contributing to the optimization of both Centralized and Decentralized Federated Learning paradigms within the framework.

3.2.1 Feature extraction

The Fedstellar framework employs Docker containers to simulate the nodes in the federated learning network, a design choice that considerably eases the process of feature extraction. Within this context, node features can be broadly divided into two categories: Physical Conditions and Learning Conditions. For the extraction of features related to Physical Conditions—such as computational power and data traffic size—the Python package 'psutil' proves to be invaluable.The Python package 'psutil' is an incredibly versatile utility that offers an expansive suite of methods for system monitoring. Designed to be cross-platform, it allows for the acquisition of a wide range of system statistics and metrics, including, but not limited to, CPU statistics, memory usage, disk activity, network status, and even details about individual running processes[31]. This comprehensive set of capabilities makes 'psutil' a highly attractive choice for feature extraction, especially within the scope of this study where precise metrics are necessary for informed node selection.

By incorporating the 'psutil' package into each Docker-simulated virtual node, we can readily obtain crucial metrics like computational power and data traffic size. In the context of federated learning, the selection of appropriate metrics for computational power is important for the effective functioning of the algorithm. After careful consideration, CPU utilization, represented by the 'cpu_percent' attribute in 'psutil,' was chosen as the indicator for computational power for several reasons. Firstly, CPU utilization serves as a proxy for the computational burden being placed on a given node at any given time, offering real-time insights into the system's current capabilities and workload. Secondly, it is a readily available metric, thereby minimizing the computational overhead associated with the feature extraction process itself. Lastly, CPU utilization is a well-understood and widely-used metric, making it a reliable and interpretable choice for other researchers. By using 'cpu_percent' as the computational power indicator, we aim to capture the instantaneous computational status of each node, thereby enabling a more nuanced and dynamic node selection process within the federated learning environment. For monitoring data traffic size, the "psutil" package conveniently offers a ready-to-use attribute called

"net_io_counters." Listing 3.1 provides the code snippet employed within the simulated virtual node to capture both computational power and data traffic size.

Listing 3.1: physical feature extraction

```
self.cpu_percent = psutil.cpu_percent()
net_io_counters = psutil.net_io_counters()
self.bytes_received = net_io_counters.bytes_recv
self.bytes_send = net_io_counters.bytes_sent
```

Before diving into the specifics of learning feature extraction, it is essential to introduce the PyTorch Lightning package, a significant part in the fedstellar framework's machine learning process. PyTorch Lightning is a lightweight PyTorch wrapper designed to decouple the science code from engineering code. It enables more organized, modular, and reusable code and simplifies complex network training tasks into straightforward, high-level abstractions. For example, with PyTorch Lightning, training loops, and callbacks are abstracted in a way that allows us to focus on the model logic rather than plain code. It not only brings structure to our code but also makes it easier to extract performance metrics, such as loss.

Regarding learning conditions, two features must be extracted: loss and data size. Thanks to PyTorch Lightning, these metrics can be easily acquired. The loss is accessible through the 'callback_metrics' method of the 'LightningLearner''s trainer module, as illustrated by the code listing 3.2. In each training process, the trainer execute this callback method, obtain the training loss from the trainer, and save it to the node for later selection. Similarly, data size is straightforward to extract from the 'LightningLearner''s data module. The code snippet `self.data\_size = len(self.data.train\_dataloader().dataset)` retrieves the length of the dataset, giving us an easy way to determine data size.

Listing 3.2: partial feature extraction

```
self.loss = float(self.__trainer.callback_metrics['Train/Loss'])
self.data_size = len(self.data.train_dataloader().dataset)
```

In this implementation of node selection, the metrics of availability, latency, and age are extracted using diverse methods tailored to each. Starting with 'availability,' for the purpose of simulation, we uniformly set this metric to 1 across all nodes. This serves as a general, neutral coefficient that affects all other attributes of the nodes. This standard setting helps maintain a controlled environment for testing. However, when moving from the simulated world into real-world applications, availability would likely be variable and could be linked to several factors like battery status, network availability, or even geographic location. Each of these factors can play a critical role in a node's ability to participate in Federated Learning tasks. As for 'latency,' the methodology is a bit more involved. Instead of each client node measuring its own latency, this task falls on the receiving node. In a Centralized Federated Learning (CFL) scenario, the central server is the receiver. It collects data from various client nodes and, in the process, calculates the latency for each one. On the other hand, in a Decentralized Federated Learning (DFL) system, each node acts as an aggregator and calculates latency based on interactions with its neighbors. This information is stored and used for node selection, as detailed in Code

Listing 3.3. Here, whether in a CFL or DFL setup, the receiving entities regularly obtain 'feature messages' from their neighbors and employ specific algorithms to compute the latency between them and their corresponding neighbors. Lastly is the 'age,' a less straightforward but equally important metric. Unlike availability and latency, age is not directly extracted from the nodes themselves. Instead, it is managed at the level of the 'selector,' which exists either on the central server in CFL or individually within each node in a DFL context. Initially, when federated learning kicks off, every neighboring node is assigned an age value of 1. As learning progresses and nodes participate in different training rounds, their age metric evolves. Specifically, if a neighboring node isn't chosen for a particular training iteration, its age increases by 1. Conversely, the age of a selected node remains constant, offering a dynamic yet consistent means of evaluating long-term participation and reliability.

Listing 3.3: partial feature extraction

```
def get_latency(self, h, p):

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    start = time.time()
    s.connect((h, p))
    s.close()
    latency = ((time.time()-start)*1000)

    return latency
```

3.2.2 Feature messaging

The feature messaging have a template to follow which is the heartbeat module in the framework. The Heartbeater module in the Fedstellar platform serves as a real-time monitor and communicator between nodes involved in Federated Learning. It has four main functions, the first one is to send heartbeats. It sends out regular "heartbeat" messages to check if neighboring nodes are active. Secondly it maintaining node list. It keeps track of active and non-responsive nodes, removing any that don't respond within a certain timeframe. Third it Periodically updates both the node's role and its status to a centralized controller for oversight. the Heartbeater acts as the pulse monitor and communicator for nodes, ensuring everything is functioning as expected. By leveraging this pre-existing structure, we can easily adapt the Heartbeater module to include triggering feature messaging. This extension would involve periodic notifications to nodes for sharing feature data, ensuring smooth information flow and enhancing node selection effectiveness in Federated Learning scenarios. The implementation of this can be seen in code listing 3.4, which includes an extra method to adjust the time interval between each triggering event. This allows for a customize frequent occurrence, in this case a less frequent event than the default frequency, reducing network overhead caused by feature messaging events.

Listing 3.4: Heartbeater triggering feature events

```
self.__send_event_with_interval(
    Events.SEND_FEATURES_EVENT, self.__count, beat_interval=8)

def __send_event_with_interval(
        self,event, count ,beat_interval,obj=None):

        if count % beat_interval == 0:
            self.notify(event, obj)
```

Following the initiation of the "SEND FEATURE EVENT," each node activates its corresponding feature-sending method as shown in code listing 3.5. This activation entails a multi-step process. First, the node extracts the features according to the procedures outlined in the prior section, encompassing elements such as computational power, data traffic size, availability, and learning conditions like loss and data size.

The next step is to build the feature message demonstrated in code listing 3.6, a crucial task that adheres to the existing communication protocol within the Fedstellar framework. This message comprises several parts. The initial segment specifies the message type, which, in this context, is "FEATURE." Subsequent sections of the message are dedicated to conveying the actual feature data. Notably, the feature message includes the name of the node which sending the message, along with other extracted features, omitting only the "age" and "latency" attribute.

Upon successfully building the message, the node proceeds to broadcast this feature message to its neighboring nodes. This is accomplished by leveraging a pre-existing broadcasting method within the framework. The robustness of this setup ensures not only the coherent assembly of feature information but also its efficient and reliable broadcasting among nodes.

Listing 3.5: Sending feature events

```
elif event == Events.SEND_FEATURES_EVENT:
    self.__feature_extraction()
    self.broadcast(CommunicationProtocol.build_feature_msg(
            self.get_name(),
            self.loss,
            self.cpu_percent,
            self.data_size,
            self.bytes_received,
            self.bytes_send,
            self.latency,
            self.availability
            ))
```

Listing 3.6: Building feature message

```
@staticmethod
    def build_feature_msg(node, loss, cpu_percent,
        data_size, bytes_received, bytes_send, latency, avaliability):
```

```
        """
        Static method that builds a feature message.
        CommunicationProtocol.FEATURE + node + feature
        Returns:
            An encoded feature message.
        """

        return (CommunicationProtocol.FEATURES
                + " "
                + node
                + " "
                + str(loss)
                + " "
                + str(cpu_percent)
                + " "
                + str(data_size)
                + " "
                + str(bytes_received)
                + " "
                + str(bytes_send)
                + " "
                + str(avaliability)
                + "\n").encode("utf-8")
```

The sending mechanism is just one half of the equation; the receiving end plays an equally important role in making the feature messaging system robust and functional. To facilitate this, a specific message-receiving routine is integrated into the existing communication protocol of the Fedstellar framework. In technical terms, this is achieved by extending the communication protocol with an additional 'elif' condition. This conditional statement serves as the framework's way to recognize incoming messages that carry the "FEATURES" header. Upon such identification, the protocol then delegates the body of the message to a designated command method. The simplified code snippet ins shown in code listing 3.7. This method, in turn, triggers a "FEATURE RECEIVED EVENT" notification for the receiving node while also forwarding the feature data encapsulated in the message.

Listing 3.7: Identifying feature message

```
elif message[0] == CommunicationProtocol.FEATURES:
        ......
            if self.__exec(
                                CommunicationProtocol.FEATURES,
                                message[1],
                                float(message[2]),
                                float(message[3]),
                                float(message[4]),
                                float(message[5]),
```

```
                                    float(message[6]),
                                    float(message[7]),
                                    ):
                                        message = message[8:]
    ......
```

What happens after the receiving node is notified of a new feature message? The node's next steps are determined by the logic scripted under the "FEATURE RECEIVED EVENT", which is also showed in code listing 3.8. firstly, the node fetches the address of the sending node from the received message. Following this, it calculates the latency associated with that particular node, employing methods previously mentioned. Last, these freshly received feature data are stored within the node's selector component for future utilization, be it for node selection or other operational considerations.

Listing 3.8: Receiving feature message

```
elif event == Events.FEATURES_RECEIVED_EVENT:
            h, p = obj[0].split(":")
            lat = self.get_latency(str(h), int(p))
            self.selector.add_node_features(obj[0], obj[1:], lat)
```

By carefully coordinating the sending and receiving mechanisms for feature messages, the system ensures a seamless and efficient exchange of critical information between nodes. This functionality is essential for optimizing Federated Learning processes by enabling better node selection and enhancing overall network efficiency.

3.2.3 Algorithm implementation

Upon successfully establishing a feature messaging mechanism, where sender and receiver nodes can effectively exchange features, the next critical phase emerges: the implementation of the node selection algorithm. This step make use of these features. Thus, the transition from feature extraction and message communication to algorithmic execution for node selection is both a logical and necessary progression in the node selection development.

To utilize these features and manage the age of nodes, one approach is to create a new module within the node called "Selector". The Selector module handle all tasks related to node selection and management of variables associated with node selection. The implementation of the Selector can be seen in code listing 3.9, which begins by initializing a list of neighboring nodes, an empty dictionary for storing features, and an empty dictionary for storing neighbors' ages. When triggered by the "FEATURE RECEIVED EVENT", the Selector uses the method "add_node_features" to store seven features into a dictionary using the node's name as key and feature content as value. The other methods within the Selector class offer additional utility: 'set_neighbors' and 'add_neighbor' allow dynamic modification of the neighbors list, while 'node_selection' serves as a placeholder for the actual node selection algorithm. Furthermore, the 'clear_selector_features' method offers the ability to reset the features dictionary, providing a clean slate for updated feature data.

Another noteworthy method is 'init_age', which initializes the age of each neighboring node to 1, providing a foundational step in node age management.

Listing 3.9: Selector module in Node

```
class Selector():

    def __init__(self, node_name="unknown", config=None):
        self.node_name = node_name
        self.config = config
        self.role = self.config.participant["device_args"]["role"]
        self.neighbors_list = []
        self.features = {}
        self.age_list = {}

    def add_node_features(self, node, features, latency):
        self.features[node] = {}
        self.features[node]["loss"] = features[0]
        self.features[node]["cpu_percent"] = features[1]
        self.features[node]["data_size"] = features[2]
        self.features[node]["bytes_received"] = features[3]
        self.features[node]["bytes_send"] = features[4]
        self.features[node]["availability"] = features[5]
        self.features[node]["latency"] = latency

    def set_neighbors(self, neighbors_list):
        self.neighbors_list = neighbors_list

    def get_neighbors(self):
        return self.neighbors_list

    def add_neighbor(self, neighbor):
        self.neighbors_list.append(neighbor)

    def node_selection(self):
        pass

    def clear_selector_features(self):
        self.features = {}

    def init_age(self):
        for i in self.neighbors_list:
            self.age_list[i] = 1
```

Building upon the foundational elements of the Selector module, a Priority Selection Algorithm can be seamlessly integrated, primarily by overriding the ‘node_selection‘ method. This algorithm functions by employing various node features to generate a list of nodes

selected for model aggregation or model training, among the neighboring nodes. the detailed code implementation is shown in the code listing 3.10 and the following is the step-wise explanation for building and implementing the algorithm.

To commence this process, the algorithm starts by creating a duplicate list of all neighboring nodes. This ensures that any changes made during selection do not interfere with the original list. A crucial parameter set at this stage is the range for the number of nodes to be selected. Given that selecting zero nodes could lead to potential errors in subsequent operations, the minimum number of nodes to be selected is set to one. Meanwhile, the maximum is established at 80% of the total number of neighboring nodes, rounded down to the nearest integer. For instance, if there are eight neighboring nodes, the maximum number of nodes selected under this criterion would be six. Importantly, this ceiling can be adjusted depending on the condition of the nodes—should a large number of nodes be in poor condition, fewer may be selected to maintain system performance. Next, a multi-dimensional feature array is constructed, where each column corresponds to a specific neighboring node and each row represents a particular feature (e.g., 'loss,' 'cpu_percent,' etc.). This array is populated using the feature values stored in the Selector's 'features' dictionary. One critical consideration is the timing of the first node selection relative to the start of training among the neighboring nodes.

The 'loss' value can serve as a useful indicator; however, there are instances when this value may not be accurately reported, such as when a node has not yet started training. In such scenarios, the algorithm assigns a 'loss' value of 100 as a placeholder. This high value automatically positions such nodes as less favorable for selection, ensuring that nodes which have begun training are prioritized. Additionally, some feature values need to be transformed before they can be used for selection. Specifically, the 'cpu_percent' and 'latency' features require computing their reciprocal numbers. A higher CPU utilization percentage usually signifies a more efficient system under the same workload, thus justifying its inversion. Similarly, lower latency is generally preferable, making its reciprocal a suitable measure for selection.

Listing 3.10: Priority Selector algorithm implementation

```
def node_selection(self, node):
    neighbors = self.neighbors_list.copy()
    if len(neighbors) == 0:
        return node
    num_selected = max(1,int(len(neighbors)*0.8//1))
    availabililty = []
    feature_array = np.empty((7, 0))

    for node in neighbors:

        feature_list = list((self.features[node]["loss"],
                             self.features[node]["cpu_percent"],
                             self.features[node]["data_size"],
                             self.features[node]["bytes_received"],
                             self.features[node]["bytes_send"],
```

```
                                self.features[node]["latency"],
                                self.age_list[node]))
            # if loss not available, set loss to 100
            if feature_list[0] == -1:
                feature_list[0] = 100

            availabililty.append(self.features[node]["availability"])

            feature = np.array(feature_list).reshape(-1, 1)
                            .astype(np.float64)
            feature_array = np.append(feature_array, feature, axis=1)

        # 1 / cpu_percent
        feature_array[1, :] = 1/feature_array[1, :]
        # 1 / latency
        feature_array[5, :] = 1/feature_array[5, :]

        # Normalized features
        feature_array_normed = normalize(feature_array, axis=1, norm='l1')

        # Add weight to features
        # Loss, cpu, data size, data rec, data send, latency, age
        weight = [10.0, 1.0, 1.0, 0.5, 0.5, 10.0, 3.0]
        weight = np.array(weight).reshape(-1, 1)
        feature_array_weighted = np.multiply(feature_array_normed, weight)

        # Before availability
        scores = np.sum(feature_array_weighted, axis=0)

        # Add availability
        final_scores = np.multiply(scores, np.array(availabililty))

        # Probability selection
        p = normalize([final_scores], axis=1, norm='l1')
        selected_nodes = np.random.choice(
            neighbors, num_selected, replace=False, p=p[0]).tolist()

        selected_nodes.append(self.node_name)

        # Update age dict
        for node in neighbors:
            if node not in selected_nodes:
                self.age_list[node] = self.age_list[node] + 1

        return selected_nodes
```

After ensuring that all feature values are accurately placed within the multi-dimensional

feature array, the algorithm proceeds to normalize these values for each feature. This normalization is a preparatory step for subsequent computations. Once normalized, each feature is then multiplied by a specific coefficient to adjust its weight in the final selection process. For example, if the user considers the 'loss' feature to be of critical importance compared to all other features, they could set its coefficient to 100 while leaving the coefficients for other features at 1.

Subsequent to this weighting, an aggregate score for each node is calculated by summing up the weighted feature values. This aggregate score is further refined by multiplying it with an 'availability' value, thereby producing a final score for each neighboring node. It is important to note that the algorithm employs Python's NumPy library, specifically the 'random choice' module, to generate a list of node indices based on the final scores' probabilities.

Furthermore, the selector is configured for self-selection by default. The rationale behind this is that the node's own data best reflects local conditions and is exempt from network latency or other types of interference. This makes self-selection a reliable default behavior in the implemented algorithm.

The last operational step updates the age list. The algorithm iterates through the keys in the age dictionary. If a key corresponds to a node in the selected list, its age value remains the same. Otherwise, the age value is incremented by 1. This feature allows the algorithm to dynamically adapt to changes in the network's conditions over time.

Ultimately, the algorithm outputs a list of selected neighboring nodes.

3.2.4 Integration into Fedstellar workflow

After successfully developing the Selector module, integrating it seamlessly into the existing FedStellar framework is the next step. But before the proceeding with this, it is imperative to understand the default workflow of the FedStellar framework to ensure compatibility and functional coherence.

In the default workflow, the process is initiated at the framework's controller level. The controller receives a trigger from the web interface to launch the learning process and starts a Docker image for each participating node. Each node subsequently initializes its core communication modules: the gossip and heartbeater modules. These modules establish basic network functionality and facilitate communication with neighboring nodes.

After initialization, each node enters the learning phase, starting by identifying its neighbors and initializing its learners. Within this phase, the role each node plays has a significant impact on its logic.

In a CFL setup, nodes assume one of two roles: either as a trainer or as a server. Trainers are primarily responsible for receiving model updates from server nodes and submitting their own trained models back to these servers. On the other hand, server nodes act as aggregation points, receiving models from all connected trainers. They aggregate these models and disseminate the updated model back to the trainer nodes. In contrast, in a

DFL architecture, each node functions as an aggregator. Here, every node is responsible for training its own model and sharing it with neighboring nodes via a gossip mechanism. Once a node receives models from its neighbors, it aggregates them to compute an average model, which then becomes its new updated model.

To integrate the nodes selection into the existing framework, this study began by focusing on the common elements shared between CFL and DFL. One of the most critical components for the Selector module is the initialization of an age list, which serves as a record to monitor one of the features of each neighboring node. The age list is initialized at the beginning of the federated learning process during the "start learning" phase. This ensures that all neighboring nodes are properly recorded before the learning iteration begins. By integrating this step early on, the Selector module can immediately access information about neighboring nodes, which is crucial for subsequent node selection algorithms.

Listing 3.11: Age Initiation

```
for n in self.__initial_neighbors:
    if n.get_name() not in self.selector.get_neighbors():
        self.selector.add_neighbor(n.get_name())
# Add self if not already in the neighbors list
self.selector.add_neighbor(self.get_name()) if self.get_name()
    not in self.selector.get_neighbors() else None
# Initialize the age list
self.selector.init_age()
# Update the Selector's neighbors list for consistency
self.selector.set_neighbors(self.get_neighbors_names())
```

In the code snippet provided in code listing 3.11, the Selector module fetches information about the node's initial set of neighbors to populate the age list. It iterates through each neighbor in self._initial_neighbors. If a neighbor is not already in the Selector's neighbor list, it is added using self.selector.add_neighbor(n.get_name()). Additionally, the node adds itself to the neighbor list if it is not already present. After populating the neighbors, the method self.selector.init_age() is called to initialize the age attributes of all listed nodes to a default value (typically set to 1). Finally, the method self.selector.set_neighbors(self.get_neighbors_names()) is set to ensure that the Selector's internal list of neighbors is synchronized with the node's list of neighbors.

CFL integration

The next step is to integrate node selection into the training loop. However, it is worth noting that CFL's training loop behaves differently from DFL. Therefore, the immediate focus is on implementing node selection within the server-side logic of the CFL framework. By default, the server in a CFL environment follows a specific sequence of actions: it initiates an evaluation phase, proceeds to aggregate models received from the trainer nodes, and then gossip the updated global model through gossip communication. The loop then resets for the next iteration.

The inclusion of node selection can potentially facilitate this process in multiple ways. One immediate strategy is to place the node selection algorithm right before the model aggregation step. This would allow the server to consider only high score models for aggregation, which can accelerate convergence. However, implementing node selection solely at the server level would not mitigate resource utilization at the trainer nodes. Each trainer node would still execute its training cycle and communicate its local model to the server at every iteration, irrespective of its selection status. This leads to unnecessary computational resource consumption and potentially prolongs the time to convergence. Therefore, a more comprehensive strategy is proposed: node selection occurs both at the server and the trainer nodes. Here's how it works. The server, after running its node selection algorithm, sends a list of selected nodes to all trainer nodes. A trainer node, upon receiving this list, evaluates its own inclusion. If it is part of the selected nodes, it proceeds with its training cycle; otherwise, it skips training for that iteration. This approach provides a dual benefit. First, it allows for resource conservation at the trainer nodes by making the training conditional based on the selection. Second, it reduces the computational burden on the server by narrowing down the pool of models it needs to aggregate.

Following the broadcasting of the selected nodes list, the server undertakes a second round of selection to identify which received models should be integrated into the aggregated global model. This two-step selection process serves dual purposes: it minimizes computational resource expenditure and expedites the overall convergence rate of the federated learning model.

Listing 3.12: Selected Node Broadcasting

```
#broadcast the selected node list
self.selected_nodes = self.selector.node_selection(
                    self.get_name())
    self.__train_set = self.selected_nodes
    self.broadcast(
        CommunicationProtocol.build_select_node_msg(
            str("-".join(self.selected_nodes))))

#build select node msg
@staticmethod
def build_select_node_msg(nodes):
    msg = (CommunicationProtocol.SELECTED_NODES
        + " "
        + nodes
        + "\n").encode("utf-8")
    return msg


 # SELECTED_RECEIVED_EVENT
 elif event == Events.SELECTED_RECEIVED_EVENT:
            selected_nodes_list = obj.split("-")
            self.check_in_selected_list(selected_nodes_list)
```

To implement the outlined approach, additional functionalities need to be added within the framework. The main task is to inform neighboring trainer nodes about the selected nodes. This can be done by adopting an approach similar to that used for broadcasting feature events. As shown in code listing 3.12, the server first calls the node_selection() method from the Selector class to generate a list of selected nodes. Subsequently, this list is transformed into a delimited string, making it more manageable for the subsequent broadcasting stage.

This string is then broadcast to all neighboring nodes using the build_select_node_msg methods, part of the CommunicationProtocol class, that constructs the appropriate message format.Upon receiving the message at the other end, the communication protocol identifies it as a SELECTED NODES message. This triggers the SELECTED RECEIVED EVENT event, which prompts the Trainer node to split the incoming selected node string into a list using the ‘split(”-”)‘ function. The Trainer node then executes a subsequent check called check_in_selected_list to determine if it exists in this newly-formed list of selected nodes.

Before starting the training phase, the Trainer checks if it is included in the list of selected nodes. If it is, then it proceeds with the training process. Otherwise, it skips the training stage and moves on to the next steps in the workflow. This optimizes resource usage and reduces unnecessary computational loads, which aligns with the initial goals of node selection.

DFL integration

After carefully outlining the integration of the node selection mechanism into the CFL framework workflow, it is important to consider its applicability in a broader context. The complexity of ensuring efficient convergence and resource utilization in CFL raises similar concerns for DFL. In a DFL setting, where all nodes serve as aggregators, introducing node selection could offer benefits such as optimized resource utilization and faster model convergence. However, it is clearly noted that DFL operates differently from CFL, with every node acting as an aggregator. Therefore, integrating node selection into the DFL framework requires some adaptations in the workflow. Now the next step is to explore how the integration of node selection into the DFL framework using principles established in the CFL context.

Building on the insights from the CFL setting of the framework, integrating node selection into the DFL setting comes with its own set of challenges and benefits. However, since age initiation, node selection algorithms, and the broadcasting of selected nodes have already been developed, the integration into DFL is somewhat convenient. The DFL setting of the framework applies just one role—Aggregator—which simplifies the implementation.

Similar to the approach in CFL, the node selection process takes place after the model evaluation phase. The major distinction lies in the decision-making process for the Aggregator regarding training commencement. In a DFL framework, each Aggregator receives multiple selected node lists from its neighboring nodes, complicating the decision on whether to proceed with training. To resolve this, a specific method, captured in the code under the function _selected_decision, is required.

Listing 3.13: Selection decision

```
def __selected_decision(self):

    waiting_time = 0
    time.sleep(2)

    if self.config.participant["device_args"]["role"]
            == Role.AGGREGATOR:
        while self.total_selected_msg_received
                == 0 and waiting_time < 10 :
            time.sleep(2)
            waiting_time += 2

        if self.total_selected_msg_received == 0:
            return True

    if self.my_node_selected_msg_received /
        self.total_selected_msg_received >= 0.5:
            return True

    else:
        return False
```

The _selected_decision function in code listing 3.13 serves the purpose of making the training decision for an Aggregator. The function begins by introducing a waiting_time variable, initiating it at zero, and allowing a brief sleep time of 2 seconds. This sleep period is likely necessary to ensure that all messages from neighboring nodes have been received. It then checks if the role is that of an Aggregator. If no selected node messages have been received within a 10-second window, the function returns 'True', allowing the Aggregator to proceed with training, on the assumption that it has been selected by default. This is set to account for situations with poor connections between nodes, where one node may become disconnected from the network. By defaulting the value to true, it allows training to continue even when a node is isolated.

Subsequently, another decision is made. If the proportion of messages received that include the Aggregator in the selected node list is greater than or equal to 0.5, the function also returns 'True', authorizing training. Otherwise, the function returns 'False', suggesting that the Aggregator should avoid from training in that iteration.

This _selected_decision function thus serves as a critical component in the DFL node selection, ensuring not only efficient utilization of resources but also more smooth and effective training processes. This fills the gap in DFL by offering a method to decide dynamically whether an Aggregator should engage in the training for a given iteration or not, based on the received selected node lists.

In summary, the comprehensive workflow of the framework after the integration of the node selection feature is depicted in Fig. 3.1. The process initiates with the node ex-

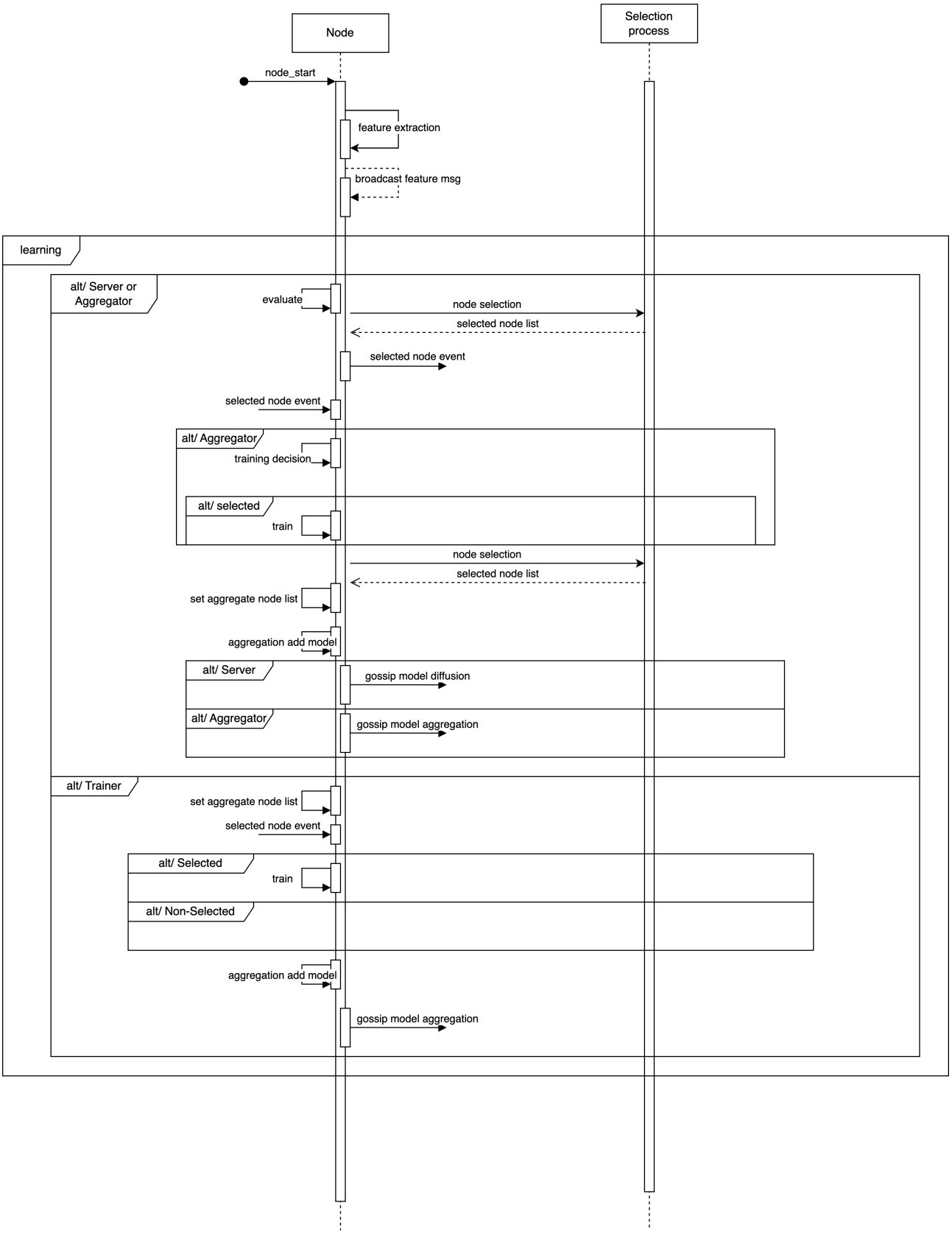

Figure 3.1: Sequence diagram of node selection

tracting features from its local virtual environment. This node then spontaneously sends feature messages to neighboring nodes, using the default broadcasting method established within the framework. These preliminary steps serve as the groundwork for the node selection mechanism, as the heartbeater module continually notifies the nodes to update features and share them with their neighbors.

Subsequent to this initial phase, the framework steps to the core learning section, the behavior of which varies depending on both the federation settings and the specific role of each node.

In the CFL setting, a node can serve as either a Trainer or a Server. If acting as a Trainer, the node initially sets the aggregated node list to solely include the Server at the beginning of each iteration. It then awaits the list of selected nodes from the Server, which the Server broadcasts after completing the first round of node selection. At this point, the Trainer faces two scenarios: if selected by the Server, it proceeds to the training phase, utilizing computational resources to develop a locally-trained model. Conversely, if the Trainer is not selected, it skips the training phase altogether. Regardless of the selection outcome, all Trainers then involved in the model aggregation stage, integrating their local models into a global model that is subsequently disseminated across the network.

Conversely, if the node is a Server within the CFL framework, it stars with an evaluation phase consistent with the default workflow. Following evaluation, the Server conducts the first round of node selection, based on the features obtained from neighboring nodes. This results in a list of selected nodes, which the Server then broadcasts to the Trainers. Since the Server is not involved in training in the CFL setup, it moves directly to the second round of selection, identifying nodes whose models contributes to the ensuing aggregation.Then, the Server performs the aggregation using the models from the selected Trainer and output an aggregated global model. Finally, the newly aggregated global model is diffused to neighboring Trainer nodes, following standard process within the framework.

In the DFL framework, nodes have only one role: Aggregator. Mirroring the duties of the Server in the CFL setup, an Aggregator initiates the learning process with a model evaluation, followed by a first-round node selection. After selection, the Aggregator broadcasts the list of chosen nodes to its neighbors. Simultaneously, it receives selection lists from neighboring Aggregators, storing these for subsequent decision-making. Based on these aggregated lists, the node decides whether to proceed with training. If the decision is affirmative, the Aggregator continues on the training phase, contrasting with the non-training role of the Server in CFL. If the decision is negative, the Aggregator will skip training and move on to the next stage. Similar to the CFL Server, the Aggregator then engages in a second round of node selection, determining which nodes' models will be involved in the model aggregation. After this, the Aggregator will perform the aggregation calculation and have a aggregated model as output. Finally, this aggregated model is distributed among the network's other Aggregators.

The flow diagram in Figure 3.2 presents a simplified outline of all the key steps in the learning section, following the framework's integration with the node selection strategy. At the top of the flow diagram, there are two options: CFL and DFL. In CFL, the primary distinction lies in the role of the node, which can either be a Server or a Trainer.

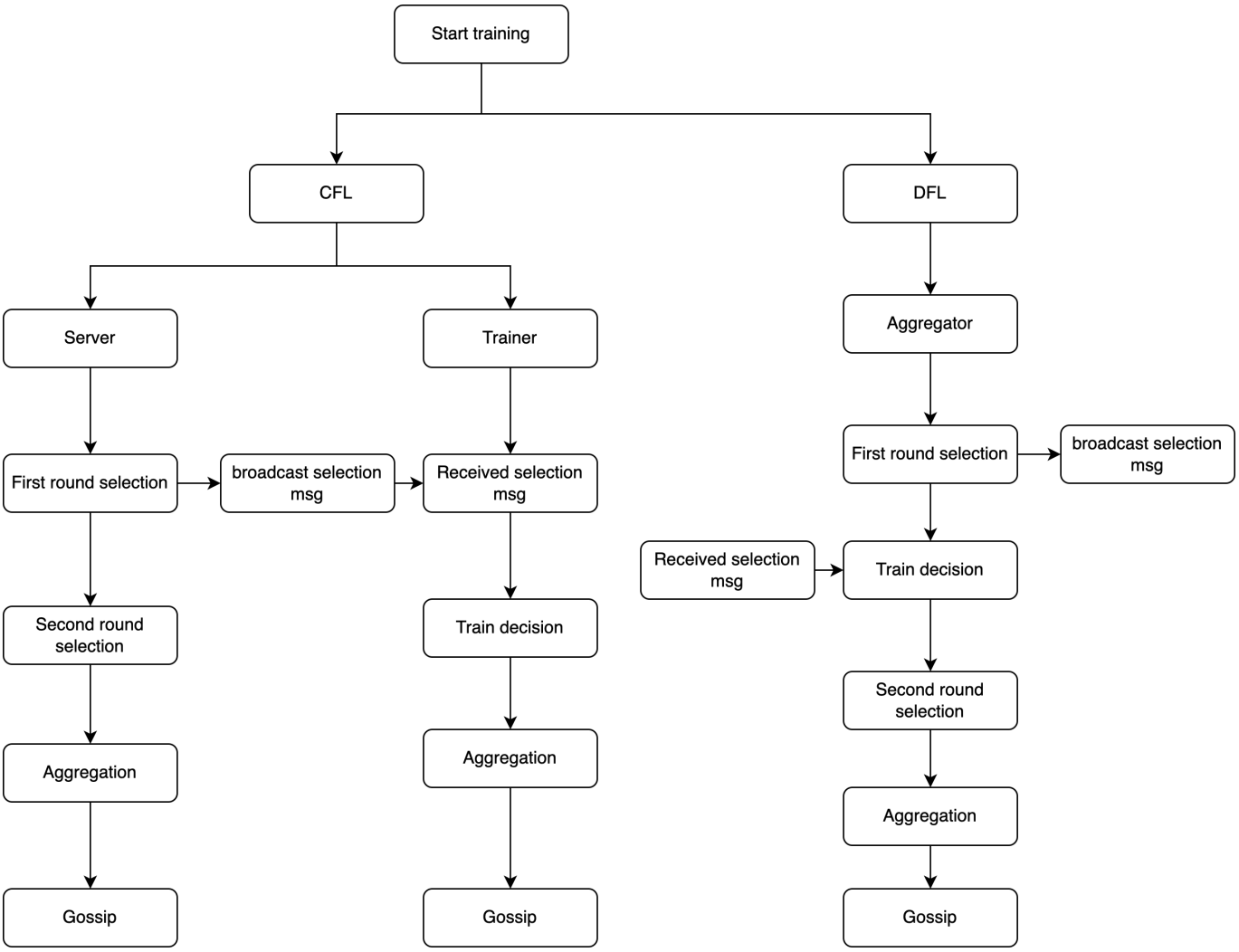

Figure 3.2: Flow diagram of node selection

The Server performs the first-round selection and broadcasts this selection message to the Trainers, followed by a second-round selection. After these two steps, the round concludes with model aggregation, and the aggregated model is disseminated to neighboring nodes. The Trainer, on the other hand, first receives the node selection message and makes a training decision based on it. Then it proceeds with local model aggregation and shares this model with the Server. In DFL, all nodes start without any role options. The Aggregator is the only role here. They all perform a first-round selection, broadcast the selection message, and receive selection messages from other nodes. Once all the selection messages are received, each node makes a training decision, opting to either proceed or skip the training section. Following the training, all nodes engage in a second-round selection to determine which nodes' models is included in the aggregation process. Finally, they perform the aggregation and gossip the models throughout the network.

3.3 Random selection and default selection

To transition to the next section, it is important to expand the investigation into node selection algorithms. While the preceding section focused on a specialized, feature-based approach for selecting nodes, this section now dives into two other algorithms: the "All Selection" and the "Default Selection." The All Selection algorithm is fairly straightforward, opting to include all neighboring nodes in the aggregation process. On the other hand, the Default Selection utilizes the same algorithm that the original framework employs, serving as a control strategy for comparative purposes. The rationale behind implementing these two additional algorithms is to establish baseline metrics. These baselines facilitates a more comprehensive evaluation in later sections, allowing us to measure the effectiveness and efficiency of the feature-based node selection strategy in a more convincing manner.

The first alternative algorithm introduced is the "Random Selection." Same as its name, this algorithm takes a straightforward approach by randomly choosing neighboring nodes to participate in both the aggregation process and the training sessions. Building upon the foundational methods already available in the Selector class, the implementation of the random selection becomes relatively simple, essentially presenting a simplified version of the priority selection mechanism. Unlike the priority-based strategy, this algorithm doesn't rely on the reception of features from nodes. Instead, it simply requires a list of neighboring nodes. From this list, it selects a subset randomly and returns the result. The design choice mandates that the selection encompasss approximately 80% of the total neighboring nodes, with a strict minimum of one node. This minimum is set to prevent potential errors from arising due to empty selection.

The detailed code is shown as code listing 3.14. The RandomSelector class extends the primary Selector class. Upon initialization, it configures itself based on the provided settings, capturing the role of the node within the context. The node_selection method first creates a copy of the current neighbors list. If this list is empty, it returns 'None' indicating no selection. However, in cases with available neighbors, it determines the number of nodes to be selected, ensuring that it is at least one or 80% of the total neighbors. By using the numpy library, it randomly chooses the requisite nodes without

repetition. To this list of selected nodes, the name of the current node is also appended. The final list is then returned, completing the selection process.

Listing 3.14: Random selection

```
class RandomSelector(Selector):
    def __init__(self, node_name="unknown", config=None):
        super().__init__(node_name, config)
        self.config = config
        self.role = self.config.participant["device_args"]["role"]

    def node_selection(self, node):
        neighbors = self.neighbors_list.copy()
        if len(neighbors) == 0:
            return None
        num_selected = max(1,int(len(neighbors)*0.8//1))

        selected_nodes = np.random.choice(
            neighbors, num_selected, replace=False).tolist()
        selected_nodes.append(self.node_name)

        return selected_nodes
```

Another strategy to add in the framework is the "Default Selection". The "Default Selection" serves a control group to the other algorithms. This selection strategy is specifically designed to provide a baseline, facilitating comparison of experimental results obtained before and after the integration of the priority selection. Its main objective is to emulate the original framework's behavior as though no modifications were ever made, thereby conserving computational resources and reducing network overhead.

To ensure this default behavior, the default selection strategically skips certain methods and events that are initiated by the selector module. Drawing from the workflow detailed in the priority selection implementation, one of the first aspects that need to bypass is the "send feature event." Within the Heartbeater module, bypassing this event is achieved by introducing a conditional check. This check examines the node's configurations using the config.participant attributes. If the selection_algorithm attribute is set to "no," the Heartbeater avoids triggering the "send feature event" to prevent unnecessary network communication.

Goes into the framework's workflow, the node selection integrated Node module has certain steps which are add-ons compared to the original framework. One such step is the initialization of 'age' at the beginning of the learning section. To avoid this, a conditional check is placed before the initialization process. The initialization only proceeds if a selection algorithm is active. This selective initiation logic is repeated in both the CFL and DFL settings during the first and second round selections. Regardless of whether a node is functioning as a Server or an Aggregator, the selection process is omitted unless both 'round' and 'selector' are defined. For nodes operating as Aggregators, the default selection also need to skip the training decision. The original framework's behavior is

simulated, which aways results in model training. The final adjustment is to set the list of nodes that contribute to aggregation. In a default scenario, the node incorporates all its neighbors for model aggregation. If a selection strategy is in play, only the selected nodes are considered for aggregation.

Listing 3.15: Send feature event skipping

```
if self.config.participant["scenario_args"]
                ["selection_algorithm"] != "no":
    self.__send_event_with_interval(
            Events.SEND_FEATURES_EVENT,
            self.__count,
            beat_interval=8)
```

Listing 3.16: Age initialization skipping

```
# age initialization skipping
if self.selector is not None:
    for n in self.__initial_neighbors:
        if n.get_name() not in self.selector.get_neighbors():
            self.selector.add_neighbor(n.get_name())
    self.selector.add_neighbor(self.get_name()) if self.get_name(
    ) not in self.selector.get_neighbors() else None
    self.selector.init_age()
    self.selector.set_neighbors(self.get_neighbors_names())
```

Here shows the detailed implementation in the code. To skipping the send feature event, unless the selection algorithm is set to "no," the node broadcasts its features at regular intervals as code listing 3.15.

In code listing 3.16, age initialization is only undertaken if a selection algorithm is active, allowing nodes to maintain their neighbor list and age metadata.

Listing 3.17: Node selection skipping

```
# node selection skipping
if self.round is not None and self.selector is not None:
    self.selected_nodes = self.selector.node_selection(
        self.get_name())
    self.__train_set = self.selected_nodes
    self.broadcast(
        CommunicationProtocol.build_select_node_msg(
                str("-".join(self.selected_nodes))))

# training decision skipping
if self.selector is not None:
    if self.__selected_decision():
```

```
            self.__train()
else:
    self.__train()
```

In code listing 3.17, node selection only execute if the current round and selector are defined. Selected nodes are then broadcasted to the network. And the training always takes place under default settings. If a selector exists, it checks the training decision before initiating training.

Listing 3.18: Set model aggregation node

```
# set model aggregation node
if self.selector is not None:
    self.__train_set = self.selected_nodes
    self.aggregator.set_nodes_to_aggregate(self.selected_nodes)
else:
    self.aggregator.set_nodes_to_aggregate(self.__train_set)
```

In code listing 3.18, the final aggregator setting determines which nodes are included in the aggregation process. By default, all neighbors are considered, but if a selection strategy is active, only the selected nodes are taken into account.

3.4 Random virtual constrains

As progress, all the comparison algorithms are ready for use. However, a important consideration arises regarding the physical conditions of the nodes. In this federated learning framework, nodes are simulated using Docker containers. By default, these nodes have similar conditions due to uniform Docker configurations. Docker containers are essentially isolated environments that can run software independently. While this isolation provides consistency and reliability, it also means that, under default Docker configurations, these nodes exhibit similar conditions in terms of computational resources, memory allocation, and network condition. Such uniformity presents a challenge for node selection. When the node selection strategy is implemented, it considers various node features such as computational power, available memory, network latency, and other performance metrics. In a simulated environment where these metrics are nearly identical for each node, distinguishing between them becomes difficult. In real-world applications, nodes represent physical devices, each with its unique set of conditions and attributes. Such diversity allows the node selection algorithm to function more effectively. To recreate this realistic scenario within our virtual framework, it becomes inevitable to simulate various node conditions using virtual constraints. By doing so, the framework can emulate the variability of real-world physical devices, enhancing the robustness and validity of the comparisons.

In the federated learning framework, during the feature extraction phase, there have identified a set of seven key features that play important roles in the node performance and node-selecting process. These features were chosen based on a combination of their representing conditions to real-world scenarios and their potential impact on learning outcomes. While all seven hold significance, for the purpose of simulating virtual constraints,

two features particularly stand out due to their direct correlation with a node's physical condition: computational consumption and latency.

Computational consumption is primarily a reflection of the CPU's workload. It is represented in terms of the percentage of CPU usage. In real-world scenarios, this consumption can vary widely based on the CPU model. For instance, a high-end device running minimal applications has lower computational consumption compared to an older device with multiple background processes. On the other hand, latency is an indicator of the delay in data transmission, primarily affected by the network's condition. Various real-world factors can lead to different latency experiences. For example, a device connected to a high-speed Wi-Fi has lower latency compared to a device relying on a weaker cellular signal or one faced with physical obstructions or network congestion. In a federated learning environment, varied latency among nodes can introduce challenges.

To simulate the real-world condition of nodes in this virtual environment, choosing the two features could be a good representation of the random device condition. Simulating these constraints ensures that the framework is robust and adaptive, capable of handling various physical condition of the node device. By focusing on computational consumption and latency, the random virtual constrains aim to mimic the node conditions as seen in real-world framework, thereby testing node selection strategies under realistic conditions.

When the framework is initiated through the frontend interface, it primarily uses the controller module to retrieve and process configurations. These configurations are then used to initiate nodes. The framework utilizes Docker Compose to arrange and run these nodes concurrently with similar configurations.

Docker Compose is a tool in the Docker ecosystem that simplifies the creation and management of multi-container applications. In this framework, the multi-container is used as the nodes in the federated learning network. Instead of manually launching individual containers, users can define their application setup using a docker-compose.yml file. This file specifies the services, networks, and volumes needed for the application, allowing for easy management of complex setups with interdependent containers. It is especially useful in scenarios like this framework where simultaneous operations are required.

The default setup of the docker compose file is shown in code listing 3.19. The image tag specifies the Docker image to be used, in this case, 'fedstellar'. The restart policy is set to 'always', ensuring that the container restarts automatically if it stops. The volumes maps the specified directory to the '/fedstellar' path inside the container. With extra_hosts, an entry for 'host.docker.internal' is added to the container's /etc/hosts file. The ipc configuration set to 'host' allows processes within the container to communicate. The privileged status provides the container with elevated privileges, allowing it more access to the host system.

The command section defines the series of commands to be executed when the container starts. This includes checking the container's network configuration, adding an entry to the host file, and then starting the node with the python script node_start.py. Furthermore, they all connect on a specified network 'fedstellar-net', with a fixed 'ipv4_address'.

Listing 3.19: Docker compose default setup

```
image: fedstellar
        restart: always
        volumes:
            - {}:/fedstellar
        extra_hosts:
            - "host.docker.internal:host-gateway"
        ipc: host
        privileged: true
        command:
            - /bin/bash
            - -c
            - |
            ifconfig && echo '{} host.docker.internal' >>
                /etc/hosts && python3.8
                /fedstellar/fedstellar/node_start.py {}
        depends_on:
            - participant{}
        networks:
            fedstellar-net:
                ipv4_address: {}

participant_template.format(index,
                            os.environ["FEDSTELLAR_ROOT"],
                            self.network_gateway,
                            path,
                            idx_start_node,
                            node['network_args']['ip']
                            )
```

To set the constrains to the computational consumption, a straightforward way is to set the percentage of the CPU that can be used in the Docker container. By default, Docker containers have unlimited access to the host CPU. However, in federated learning frameworks where different physical node conditions need to be mimicked, it becomes necessary to limit this access. Docker Compose simplifies this control by allowing users to specify settings under the deploy section of the compose file. One powerful attribute provided by Docker Compose is the resources attribute. This attribute enables users to set physical resource constraints for containers operating on the host machine. For example, using the cpus field under resources, users can determine how much of the available CPU cores a container can use. Setting a value of '0.5' for cpus means that only half a core of the host machine's CPU is used by that container.

Listing 3.20: CPU constraints

```
# CPU constraints
```

```
deploy:
    resources:
        limits:
            cpus: '{}'
```

Introducing latency between different nodes is a bit more complicated compared to CPU constraint configuration. One effective solution to this is the utilization of a package known as tcconfig.[32] This Python package is capable of configuring network traffic control settings such as bandwidth, latency, and more.

To integrate tcconfig within the Docker environment, it needs to take several steps. Firstly, Install the iproute2 package by adding the line "RUN apt-get install -y iproute2" to the Docker build file (Dockerfile). This ensures that all dependencies for tcconfig are in place. Subsequently, adding tcconfig to the requirements.txt file ensures that when the Docker image is being built, the package is also installed. Upon successful installation of tcconfig, setting up the desired latency becomes relatively straightforward. As illustrated in code listing 3.21, the command "tcset eth0 –delay " can be executed. Here, the placeholder ‘‘ is substituted with the desired latency value in milliseconds. By executing this command, the specified latency is introduced to the network interface, allowing for a more realistic simulation of various node conditions in federated learning scenarios.

Listing 3.21: Latency constraints

```
# Latency constraints
command:
        - /bin/bash
        - -c
        - |
        tcset eth0 --delay {} && ifconfig &&
            echo '{} host.docker.internal' >>
            /etc/hosts && python3.8
            /fedstellar/fedstellar/node_start.py {}
```

3.5 Front-end implementation

The framework has almost completed the implementation of virtual constraints for nodes. This allows the framework to limit CPU resources and introduce latency between nodes, simulating complex device and network conditions in real-world scenarios. However, manually adjusting constraints for each node is time consuming and not representative of actual deployment scenarios, especially when there are more than ten nodes in a network during testing. Additionally, determining ideal constraint values requires careful consideration to ensure they align with the intended simulation conditions. In this work, we plan a random virtual constraint in a certain scale for setting CPU and latency constraints. CPU constraints is ranged from 0.3 to 1 core, simulating environments ranging from low-end computational devices with slower processing times during training to more

efficient systems that don't affect computational speed. Latency varies between 1ms and 50ms, replicating optimal network conditions (near-zero latency at 1ms) as well as common but less ideal conditions where data transmission delays occur at 50ms. By making these adjustments, the framework aims to create a realistic simulation environment that accurately reflects real-world scenarios.

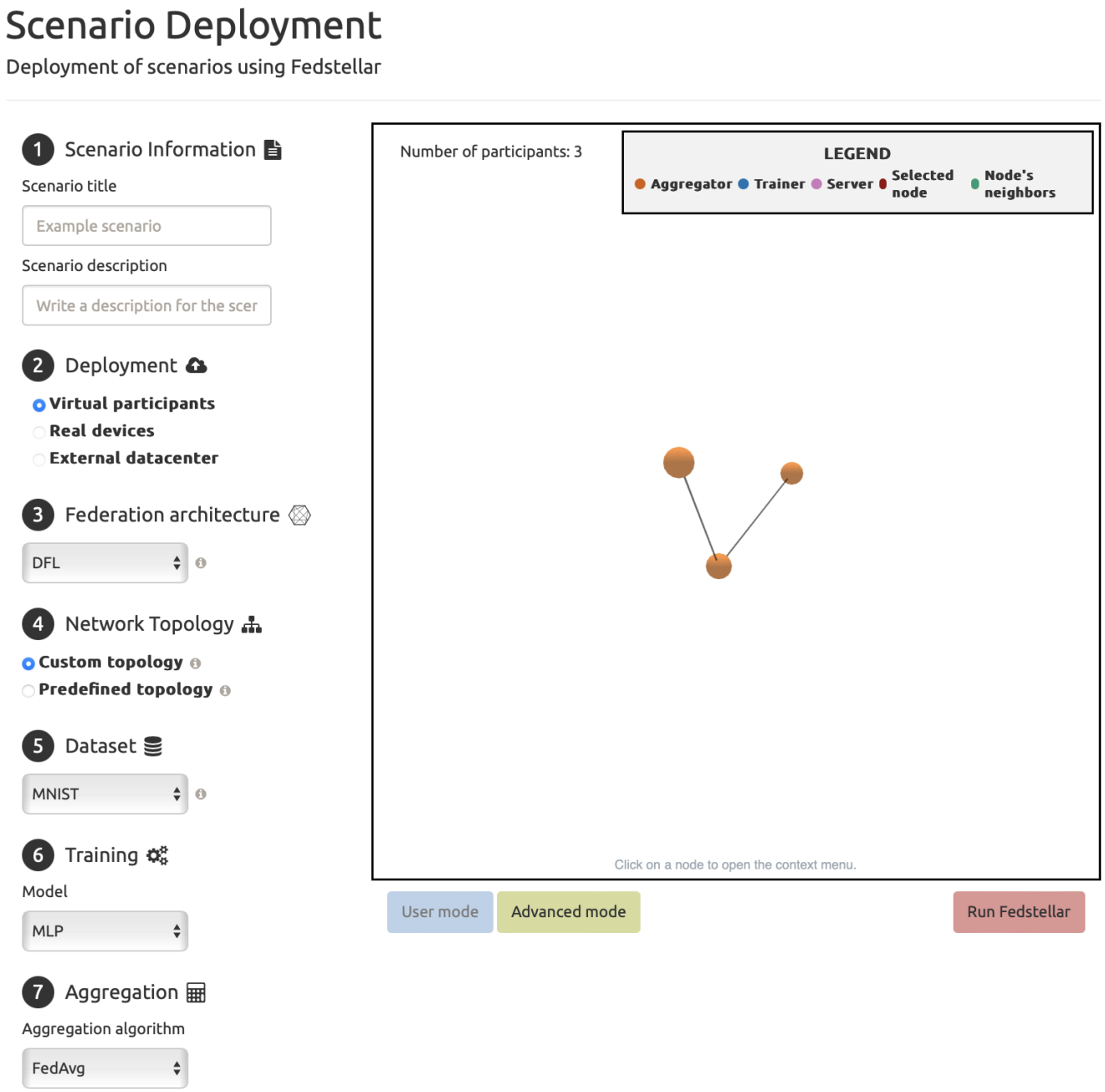

Figure 3.3: The default deployment page

To achieve the generation of random value into the constraints during network deployment, the next step is directed towards refining the frontend configuration. This refinement includes several modifications. First, incorporating constraints for specific features. Second

is the random value generation. Third is the node selection strategy option.

In the default setting, the left panel is for all the configurations, and the right panel shows a visual representation of the network topology (Fig. 3.3). Clicking on the 'advanced' button reveals additional configurations, which are shown in Fig. 3.4. Notably, section number 8 provides detailed information about each node's IP address and port number. Given this existing framework, the plan is to add in additional functionalities. This integration ensures that users have all the tools and access to not only the newly implemented node selection strategy but also the simulation of real-world scenario during experiment.

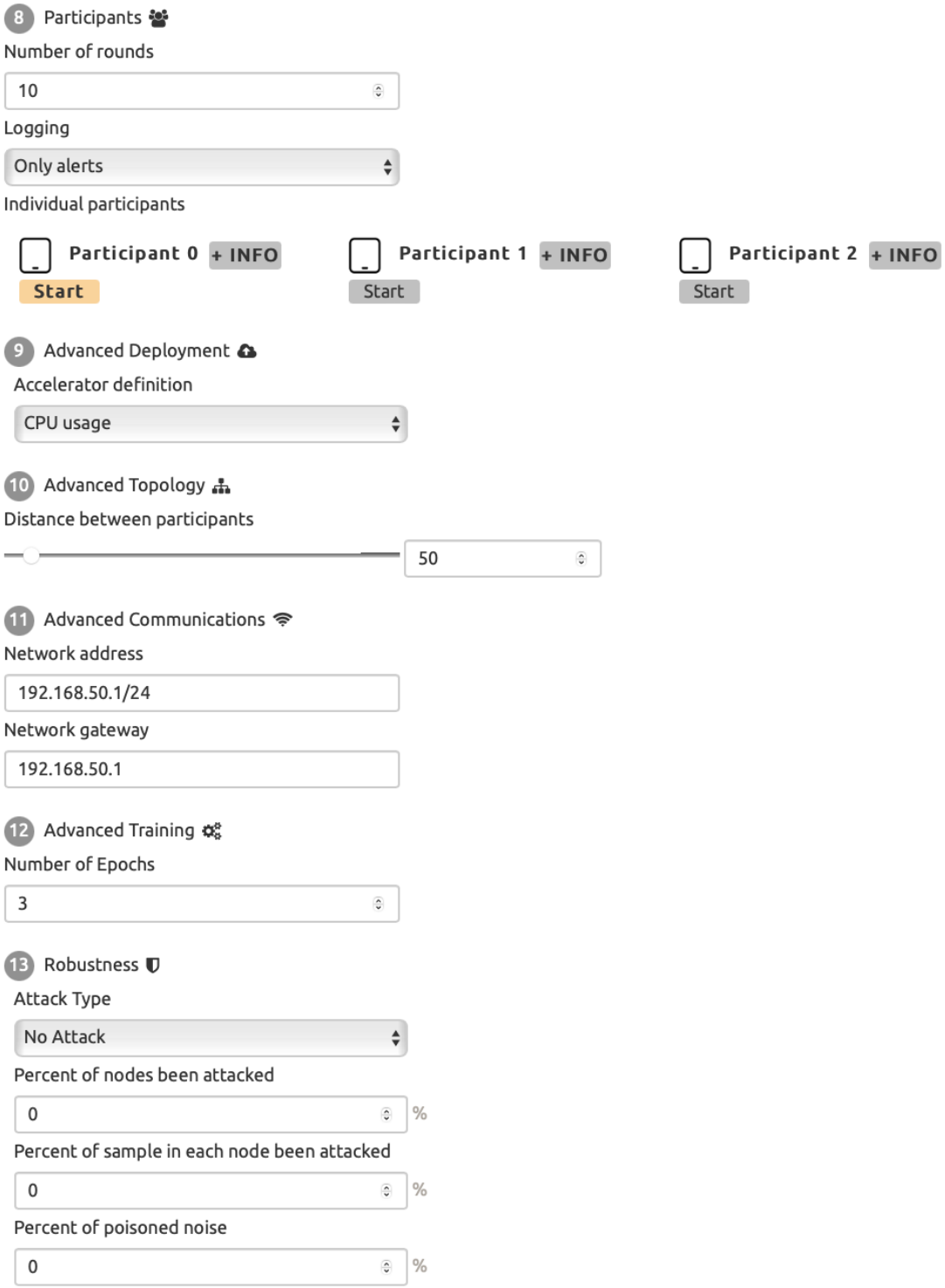

Figure 3.4: The advanced setting at deployment page before modification

Starting from the easy task. The first step is to provide users with options for node selection strategies. This allows them to choose a strategy that best aligns with their experimental requirements. To achieve this, the solution involves using a simple code snippet that incorporates an HTML div element designed for form input as shown in code

listing 3.22. This helps the user to select the desired node selection strategy by adding an drop-down menu.

Listing 3.22: Selection Option

```
<div class="form-group row">
    <h4...>Node Selector<.../h4>
    <div ...>
        <select class="form-control" id="selector" ...>
            <option value="new">Priority Selection</option>
            <option value="random">Random Selection</option>
            <option value="all">All Selection</option>
            <option value="no">Default Selection</option>
        </select>
    </div>
</div>
```

After implementing such modification, the dropdown menu looks like the Fig 3.5. Users can choose from three selection strategies: priority selection, random selection, and default selection.

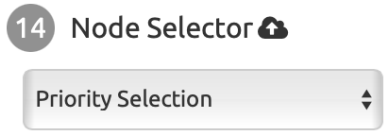

Figure 3.5: Node selection option

Taking the implementation forward, the objective was to incorporate the virtual constraints directly into each node's detailed information panel. Within the default interface, as can be seen in Figure 3.6, pressing the "+INFO" button next to a participant shows a detailed information panel of that specific node. Here, users can easily find parameters like IP address and port number, both of which can be customized as needed. According to previous planned, the virtual constraints value such as the CPU limit and latency for individual nodes is placed at the same section here. To make this modification, two new input fields were introduced: one for setting the CPU limit and another for defining the latency.

Figure 3.6: The network detail of participant before modification

In the code listing 3.23, a label with the text ”CPU limit” is created. This is followed by an input field where users can enter specific values. A text explanation is provided to clarify the meaning of the CPU limit and guide users when filling in the values. And a similar pattern is followed for the latency configuration, where a label and input field are created. Both the ”CPU limit” and ”Latency” configurations are then added to a modal, which serves as the detailed information panel for participants. After this, a default value for cpu limit equals 0.5 and latency value equals 1ms is added.

Listing 3.23: Virtual constraints

```
const cpulimitLabel = document.createElement("label");
cpulimitLabel.setAttribute("for", "participant-" + i + "-cpu_limit");
cpulimitLabel.innerHTML = "CPU limit";
const cpulimitInput = document.createElement("input");
cpulimitInput.id = "participant-" + i + "-cpu_limit";
cpulimitInput.type = "text";
cpulimitInput.defaultValue = participant.cpu_limit;

const infoText = document.createElement("small");
infoText.textContent = "(limit for number of cores of this participant can use.)";
document.getElementById("participant-modal-content").appendChild(cpulimitLabel);
document.getElementById("participant-modal-content").appendChild(infoText);
document.getElementById("participant-modal-content").appendChild(document.createElement("br"));
document.getElementById("participant-modal-content").appendChild(cpulimitInput);
document.getElementById("participant-modal-content").appendChild(document.createElement("br"));


const latencyLabel = document.createElement("label");
latencyLabel.setAttribute("for", "participant-" + i + "-latency");
latencyLabel.innerHTML = "Latency";
const latencyInput = document.createElement("input");
latencyInput.id = "participant-" + i + "-latency";
latencyInput.type = "text";
latencyInput.defaultValue = participant.latency;
document.getElementById("participant-modal-content").appendChild(latencyLabel);
document.getElementById("participant-modal-content").appendChild(document.createElement("br"));
document.getElementById("participant-modal-content").appendChild(latencyInput);
document.getElementById("participant-modal-content").appendChild(document.createElement("br"));
```

With these in place, users can now manually define how much computational power each node can utilize and what sort of network delay it would experience just like the Fig.3.6 .

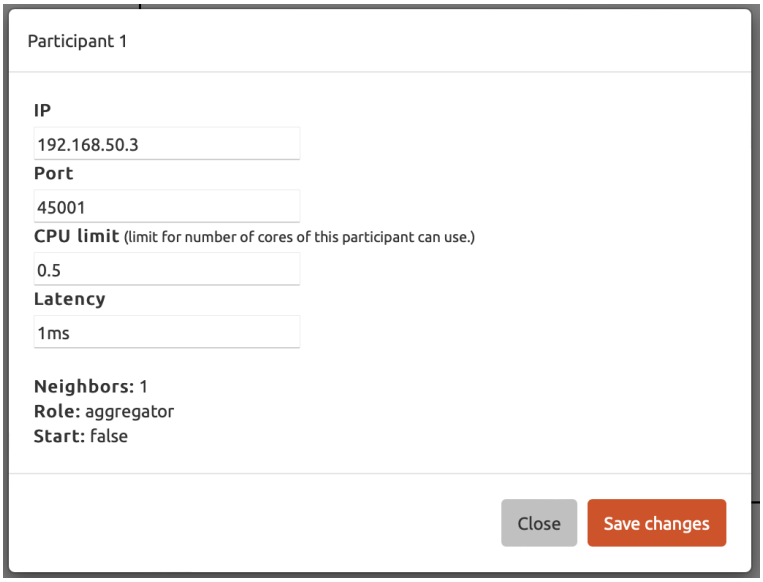

Figure 3.7: The network detail of participant after modification

The final step in the frontend implementation involves integrating a feature that auto-generates random values for both CPU limit and latency for each node within previously defined scale. . As illustrated in code listing 3.24, the function is implemented within a JavaScript script tag. A brief description for users is provided in the form of a popover. This popover explains the constraints for random generation, specifying that the CPU limit values is ranged between 0.3 to 1 and latency values between 1ms to 50ms. The code initiates by setting up the 'popover' interface through the jQuery function and this popover is triggered on hover and shows a brief overview of the constraints for the CPU limit and latency. Next, two specific functions are set to generate these random values. The generateRandomCpuLimit function is designed to output values within the 0.3 to 1 range for the CPU limit, ensuring the result is a floating-point number rounded to one decimal place. Similarly, the generateRandomLatency function is straightforward; it returns an integer value between 1ms to 50ms. The event listener attached to the "random-constrains" button element. Upon activation the function iterates through all nodes and employs the earlier defined functions to produce random CPU limit and latency values for each node in the network.

Listing 3.24: Default selection

```
<script>
    randonconstrainHelp = '<ul>\n' +
        '<li>CPU limit: generated between 0.3 - 1</li>\n' +
        '<li>Latency: generated between 1ms - 50ms</li>\n' +
        '</ul>'
    $('#randomconstrainHelpIcon').popover({
        title: 'Generate random constrain values for the virtual participant',
        html: true,
        placement: 'right',
        content: randonconstrainHelp,
        trigger: 'hover',
        container: 'body',
```

```
        delay: {show: 500, hide: 100},
    });

    //generate cpu limit
    function generateRandomCpuLimit(min, max) {
            const randomValue = Math.random() * (max - min) + min;
            return parseFloat(randomValue.toFixed(1));
        }
    //generate latency between 1 and 50 ms
    function generateRandomLatency(min, max) {
            return Math.floor(Math.random() * (50 - 1 + 1)) + 1 + "ms";
        }

    document.getElementById("random-constrains")
        .addEventListener("click", function () {

        for (let i = 0; i < getNumberOfNodes(); i++) {

            const participant = Graph.graphData().nodes[i];
            participant.cpu_limit = generateRandomCpuLimit(0.3, 1);
            participant.latency = generateRandomLatency();
            //Graph.graphData(Graph.graphData());
        }

    })
</script>
```

The finished frontend is displayed as shown in Figure 3.7. It includes a button located at the end of section 8. The info icon at the right side of the button displays the explanation text for its functionality. When the button is clicked, both the CPU limit and latency values for each participant is randomly generated within a predefined range.

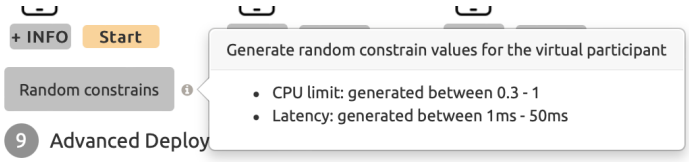

Figure 3.8: The random constraints button

Once the frontend adjustments are complete, corresponding modifications need to be made to the backend. Specifically, after users set their configurations on the deployment page, this data should be reflected and stored in each participant's configuration file. This file serves as the source of configurations for each node and is created from a pre-defined template.

To begin, it needs to update the configuration template to include new attributes. Entries like "selection_algorithm":"new", "cpu_limit":0.5", and "latency": "1ms" should be inserted as placeholders for real values that users may input. This ensures a default setup and

guarantees that the frontend's configuration decisions are recorded in the backend configuration files.

The next important step is capturing this data as it is passed from the frontend to the backend. Code listing 3.25 shows how the web server's receiving end intercepts these configuration values and updates the associated participant's configuration file with them. Upon making configurations on the frontend, the settings are communicated to the backend in the form of JSON data. The backend then processes this data, diving into each specific node from the received data set. For every node, a default configuration file is loaded and then updated based on the frontend settings. After these updates are made, the configurations are saved and can be used later.

Listing 3.25: Update configration

```
data = request.get_json()
nodes = data['nodes']
...
for node in nodes:
     node_config = nodes[node]

    participant_file = os.path.join('....json')
    with open(participant_file) as f:
                    participant_config = json.load(f)
    ...
    participant_config["device_args"]["cpu_limit"]
        = node_config["cpu_limit"]
    participant_config["network_args"]["latency"]
        = node_config["latency"]
    participant_config['scenario_args']['selection_algorithm']
        = data['selector']

     with open(participant_file, 'w') as f:
                    json.dump(..)
```

In summary, there are series of modifications made to the frontend, the user interface was significantly enhanced to improve usability and accommodate the new functions. Initially, a dropdown menu was incorporated to provide users with a selection of node strategies, enhancing their control over the learning process. After that, the detailed information panel introduced manual configuration capabilities for CPU limits and latency for each node, giving users the control over virtual constraints. To enhance the experience, a random value generator was integrated. This generator, at the click of a button, randomly assigns CPU limits and latency values within pre-defined scales to each node, mimicking the various conditions of a real-world scenario. The final product of the deployment page is shown as the Fig. 3.8.

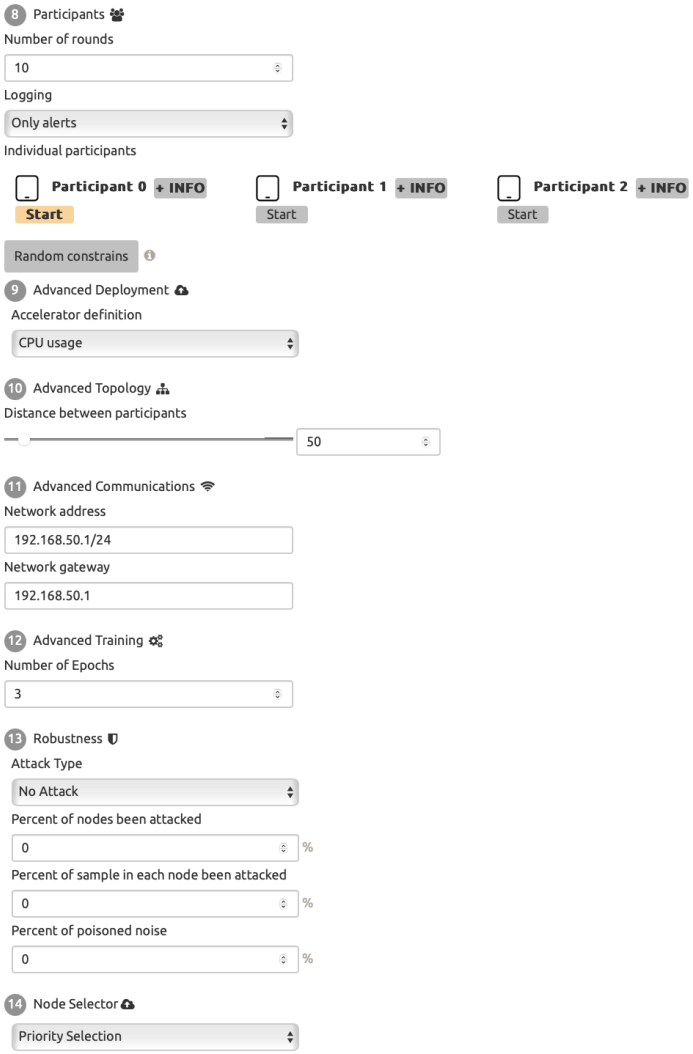

Figure 3.9: Advanced setting after modification

Chapter 4

Evaluation

4.1 Scenario deployment

After modifying the framework, it is time to run the experiment and evaluate the results. This involves measuring the effectiveness and efficiency of the newly implemented node selection strategy and functionalities using metrics, methodologies, and scenarios.

To begin, we need to go through the deployment process and determine which configurations will be used in the experiment. After logging into the framework, users can access the scenario management page (Fig. 4.1) where they can click on "Deploy a scenario" to reach the scenario settings page (Fig. 3.3).

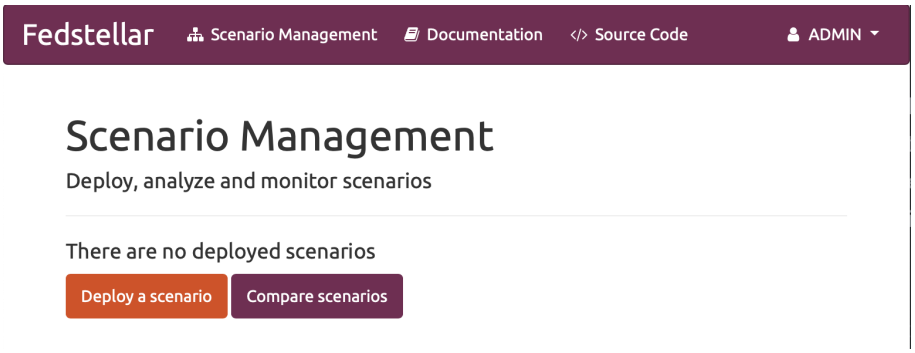

Figure 4.1: Scenario management page

In this deployment page, several configurations must be considered for conducting the experiment: federation architecture, dataset, training model, virtual constraints, and node

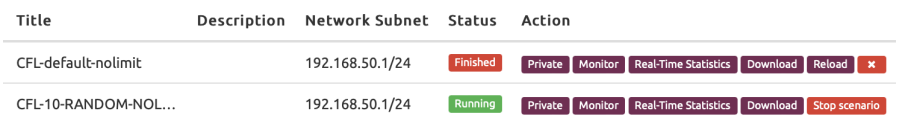

Figure 4.2: Scenario list

selection strategy. The following section will analyze how different configurations impact the experiment and evaluation setup.

The real-time statistics are shown in Figures 4.3 and 4.4. These figures display key features and metrics of resources, as well as important results such as accuracy and F1 score during testing, training, and validation.

In Figure 4.3, the plots include traffic size received and sent, CPU percentage, RAM percentage of the node over time. Figure 4.4 displays metrics such as accuracy, F1 score, loss precision, and recall for nodes in the train, test or validation sets. These metrics are crucial for comparing different node selection strategies.

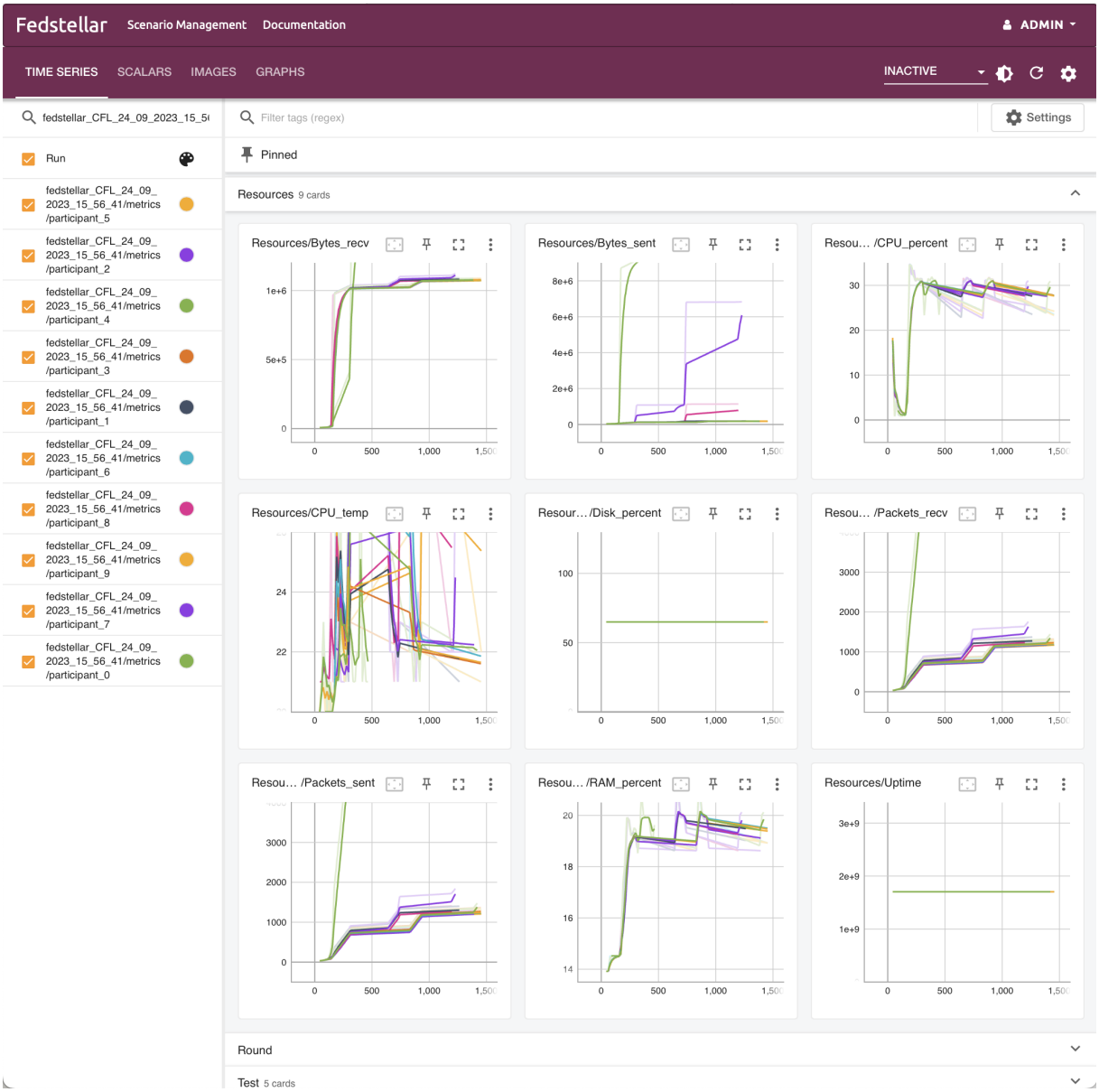

Figure 4.3: Resources metrics

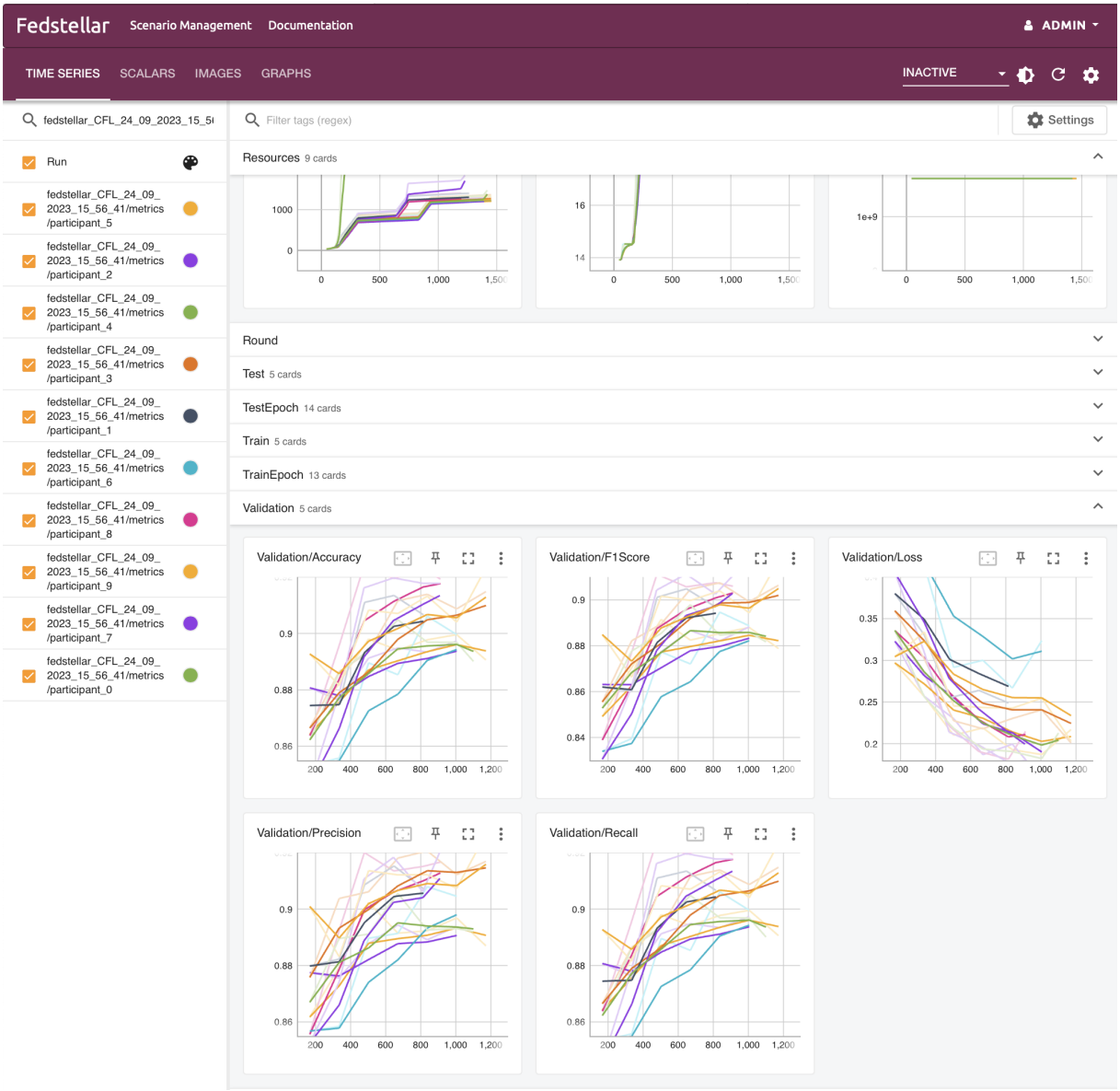

Figure 4.4: Validation metrics

To effectively evaluate different node selection strategies in federated learning, it is important to analyze specific metrics that capture the essence of a model's performance, with accuracy being one of the most crucial ones. Accuracy serves as a direct measure of how well a model performs against a given dataset. In general, a higher accuracy means that a model is more adept at fitting the dataset. In federated learning, comparing two models shows that the one with higher accuracy is more robust and has a better understanding of the data distribution across the network.

However, solely focusing on accuracy may not provide a complete understanding of the model's efficiency. Time is an important factor to consider in this context. When designing our node selection strategy, we aimed for adaptability, robustness, and efficiency. Adaptability is achieved by continuously monitoring node conditions. Robustness refers to the model's ability to achieve high validation accuracy

Efficiency in federated learning is not only determined by training speed. It also includes the framework's capability to achieve comparable or better accuracy levels within a shorter timeframe compared to other frameworks. If two strategies yield similar accuracies, but one achieves it faster, it demonstrates greater efficiency. Therefore, when evaluating our node selection strategy, an important measure of success will be comparing the time taken by different strategies to reach specific accuracy benchmarks.

In summary, the evaluation rationale for node selection strategy is based on optimizing the balance between accuracy and time. We analyze these parameters to determine the effectiveness of a node selection strategy in a federated learning environment.

4.2 Experiment setup

To design an impactful experiment, it is crucial to carefully identify the dependent outcomes and manipulate the independent variables. In this investigation, we are primarily interested in examining how federated learning performs in terms of time and accuracy under specific network configurations.

A critical determinant in federated learning is the dataset. The framework provides three options: MNIST, CIFAR10, and SYSCALL. However, practical trials exposed certain issues. The SYSCALL dataset encounters errors related to its settings and downloading process while running the framework. Since it is not commonly used in other federated learning studies, we have excluded it from our experiment. When using the CIFAR10 dataset with a CNN model, most nodes perform well. However, some nodes experience spontaneous socket connection drops during training. This issue does not occur consistently across all nodes or specific ones. Since the error is not happening in every node and can not be precisely located, the CIFAR10 dataset is also exclude for the experiment. Fortunately, the MNIST dataset with an MLP model runs smoothly without any problems encountered by the other two datasets. Therefore, for our experiment purposes, we have chosen to use the MNIST dataset with an MLP model as it provides reliable performance without any known issues.

With the dataset chosen, the focus shifted to the node selection strategy. The main objective of this study is to compare a newly proposed priority selection strategy with the default strategy. To provide a comprehensive comparison, we also included a random selection strategy. Since the priority selection algorithm picks 80% of neighboring nodes using a unique algorithm, contrasting it with random selection helps highlight the effectiveness of the priority algorithm without considering the less number of node selected.

Additionally, the experimental design considered both constrained and unconstrained conditions. This distinction is important as it demonstrates the adaptability of the priority selection in realistic scenarios where random constraints may be present. By testing in both states, we can evaluate the priority selection strategy comprehensively.

Furthermore, last configuration in the framework would be the federated mode. It is intuitive that both CFL and DFL setting need to be included in the experiment for a more comprehensive result. We also took into account the number of devices involved

in this exploration and set a threshold at ten devices for CFL and 5 devices for DFL. This number places a balance between computational power for the host machine and providing more comprehensive results compared to just three or four nodes.

Having determined all the configurations, the experiment's design systematically pairs each configuration option with every other option. This approach can be visualized as a matrix where each configuration intersects with every other, ensuring comprehensive coverage of potential settings. This extensive pairing allows us to understand how different configurations might interact and influence outcomes in federated learning scenarios. This study have tabulated these pairings in Table 4.1 for a clearer representation of the combinations. In total, there are 12 distinct experimental groups that will be evaluated.

In the CFL mode with no constraints, groups 1 to 3 distinct in different selector modes. Group 1, which uses the default selection, establishes a baseline by capturing performance metrics in the default configuration. Comparing results between groups 1 and 3 will highlight the effectiveness of the priority selection strategy. By comparing groups 2 and 3, we can determine how much impact the priority selection strategy has on overall performance compared to when no specific strategy is used.

When introducing random constraints in the CFL mode, we have groups 4 through 6. Here, group 4, employing the default selector under constraints, offers insights into how the system performs when faced with variable conditions but without any particular node selection strategy. Comparing the outcomes of groups 5 and 6 can provide insights into the robustness and efficiency of the priority selection strategy, particularly in an uncertain environments similar to real-world scenario.

Transitioning to the DFL mode without constraints, groups 7 to 9 provide a perspective on a fully connected topology. Group 7, once again, serves as our benchmark, giving us the basic DFL behavior. Differences between groups 8 and 9 will indicate the efficacy of the priority selection strategy within a DFL setting, ensuring that the observed trends in CFL are not mode-specific.

Finally, in groups 10 to 12, we will introduce random constraints within the DFL mode. The analyses in this section will serve two purposes: first, to examine the performance of a fully connected network under resources constraints; and second, to assess the value added by the priority selection strategy in these constrained conditions.

To summarize, this comprehensive set of experiments ensures a broad coverage across different selector strategy, federate mode, and constraints. Through a systematic comparison, this work aim to provide a comprehensive result for the performance of purposed node selection strategy for federated learning.

Group	Federate mode	Topology	Devices	Selector mode	Artificial constrains	Dataset
1	CFL	Star	10	Default	No constrains	MNIST
2	CFL	Star	10	Random	No constrains	MNIST
3	CFL	Star	10	Priority	No constrains	MNIST
4	CFL	Star	10	Default	Random constrains	MNIST
5	CFL	Star	10	Random	Random constrains	MNIST
6	CFL	Star	10	Priority	Random constrains	MNIST
7	DFL	Fully	10	Default	No constrains	MNIST
8	DFL	Fully	10	Random	No constrains	MNIST
9	DFL	Fully	10	Priority	No constrains	MNIST
10	DFL	Fully	10	Default	Random constrains	MNIST
11	DFL	Fully	10	Random	Random constrains	MNIST
12	DFL	Fully	10	Priority	Random constrains	MNIST

Table 4.1: Experiment group set up

4.3 Result comparison

Within each experimental scenario, there are ten participants. To simplify the results and facilitate clear comparative analysis, the displayed results will extract data from three nodes in each group.

In the comparative visuals presented later, the result will depict a total of six nodes - three from each group being compared. To enhance clarity and ease of interpretation, nodes within the same group will share the same color coding.

The x-axis of these visuals represents time during the federated learning process, while the y-axis shows node accuracy at any given time. It is important to note that data collection spans six minutes for each. The reason chose this timeframe is because after six minutes, accuracy values show minimal fluctuations. The initial six minutes are considered the most dynamic and revealing phase of the federated learning process and therefore most relevant to the analysis.

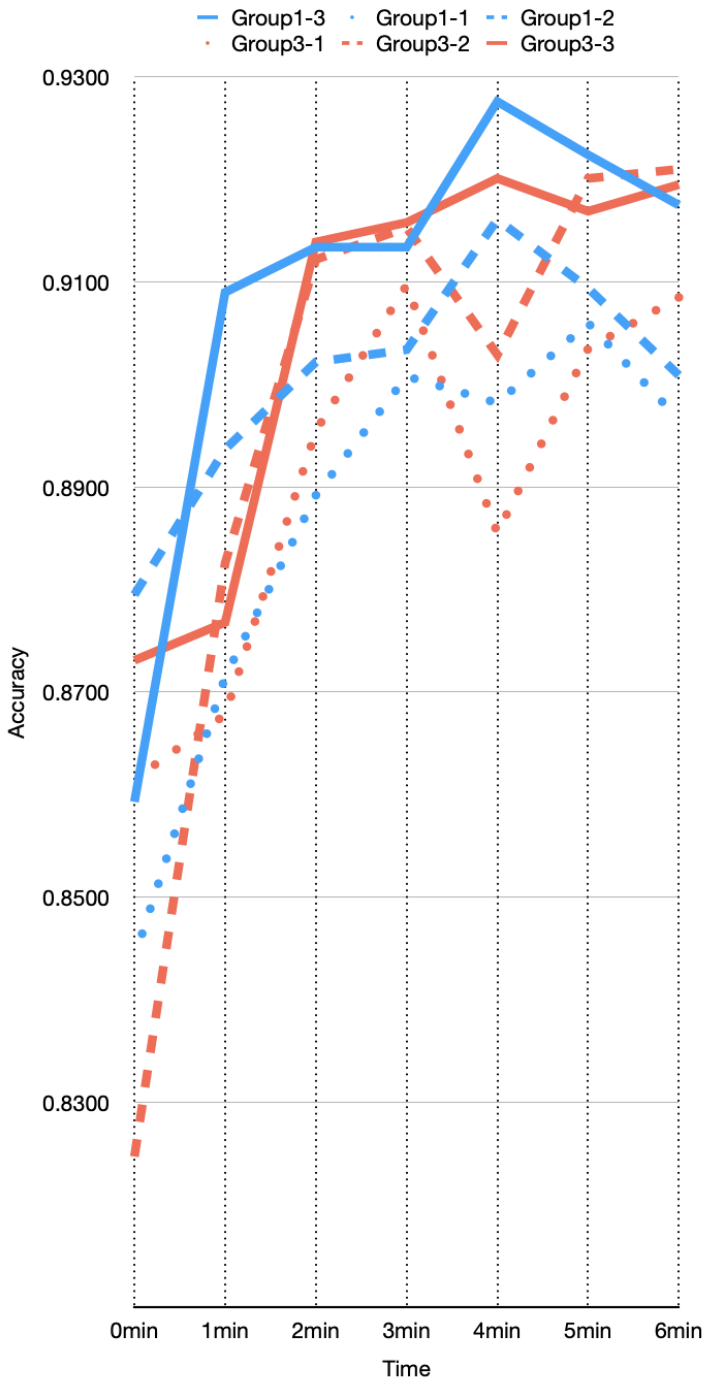

Figure 4.5: Accuracy result from group 1 and group 3

Upon reflecting on the experiments conducted, it is important to highlight certain challenges and considerations. One key point is the nature of using the MNIST dataset with the MLP model. This combination tends to converge rapidly compared to other dataset-model pairings. While fast convergence is generally advantageous, it presents challenges for our experiments. The experiment have to use the MNIST dataset due to unexpected technical difficulties integrating node selection strategy and federated learning with other datasets. As a result, MNIST became the only viable option for experimentation. How-

ever, this decision led to a brief training duration of just 6 minutes. This limited timeframe may not be sufficient to fully explore and demonstrate differences in convergence efficiencies across various node selection strategies. Additionally, the simplicity of the MNIST dataset poses limitations as well. It is primarily designed for handwritten digit recognition and already starts with high accuracy levels during training. This initial advantage leaves little room for improvement or refinement by the federated learning model. Consequently, the dataset's simplicity might not fully capture how different node selection strategies could benefit more complex scenarios.

The first set of results compares group 1 and group 3 in the CFL setting with a total of 10 participants. Group 1 represents the scenario without artificial constraints using the default selection strategy, while Group 3 represents the scenario with no artificial constraints using the priority selection strategy. In Figure 4.6, it is shown that the priority selection strategy performs similarly to the default settings in CFL. There are several possible reasons for these results: 1. The priority selection strategy may not have any significant positive or negative effect on CFL efficiency. 2. The computational overhead and network overhead associated with implementing the priority selection strategy could diminish its benefits within the framework. 3. Although node selection strategies can bring benefits to centralized federated learning, limitations in dataset and timeframe may prevent these benefits from being demonstrated in this particular result. 4. It is possible that both reasons two and three contribute to this result, resulting in similar performance between the priority selection strategy and default selection strategy.

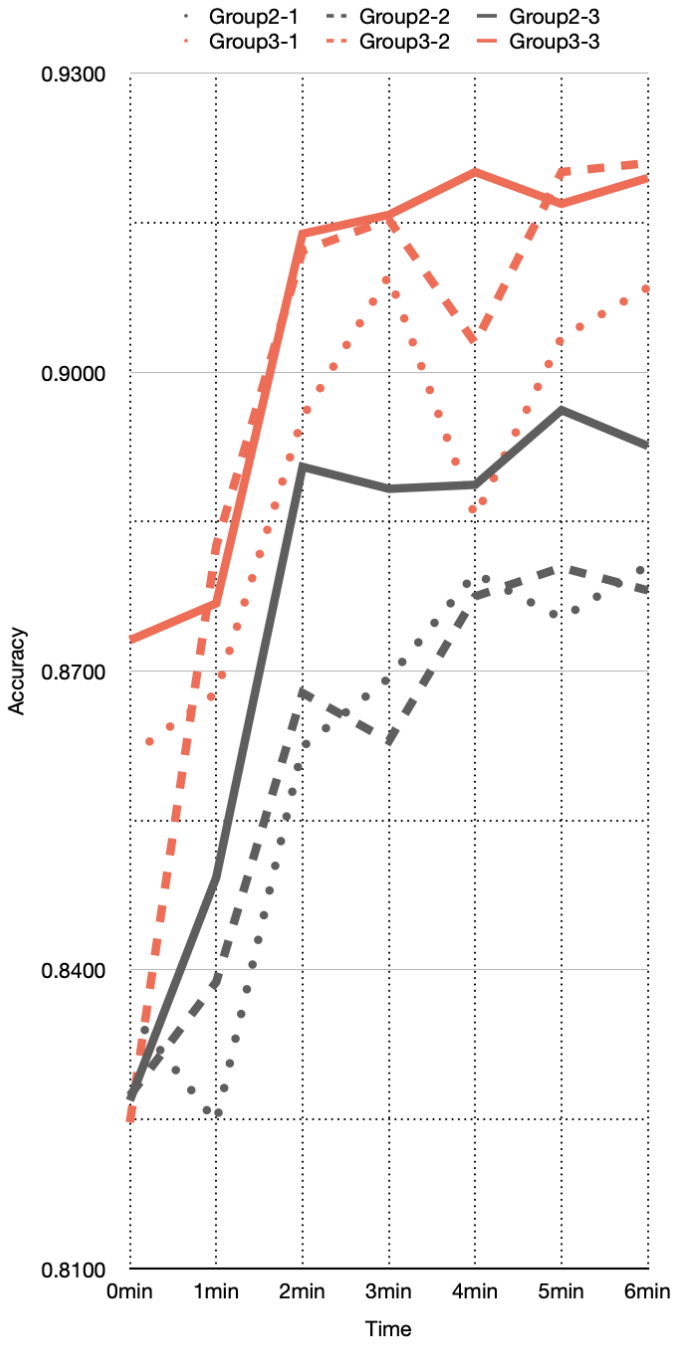

Figure 4.6: accuracy result from group 2 and group 3

To examine the hypothesis of similar performance as shown in Figure 4.6, we can use the results from Figures 4.7 for group 2 and group 3. Group 2 represents an experiment where the random selection strategy is used, while all other variables remain the same compared to groups 1 and 3. In Figure 4.7, it is evident that the priority selection strategy performs better than the random selection strategy in terms of accuracy at equivalent timepoints. This suggests that priority selection has a higher convergence rate compared to random selection. Since both group 2 and group 3 have identical implementations in

terms of framework, computational overhead, and network overhead differences are not factors influencing these results. Therefore, it can be concluded that priority selection does have a positive effect on convergence rate. These findings also provide explanations for some previous hypotheses. Firstly, this result contradicts the first hypothesis for the first comparison because there is a difference in efficiency between priority selection and random selection strategies in federated learning; with priority selection showing superior performance. Secondly, it supports the possibility of the second hypothesis in the first comparison since both group 2 and group 3 have additional implementation steps compared to group 1 which lacks this implementation step. Thus, overhead differences could be one reason why priority selection did not show performance differences when compared to default settings. Lastly, considering that there are subtle differences observed between comparisons, it is plausible that various factors influence these slight variations without any evidence suggesting that dataset limits or timeframe limits do not affect convergence rates.

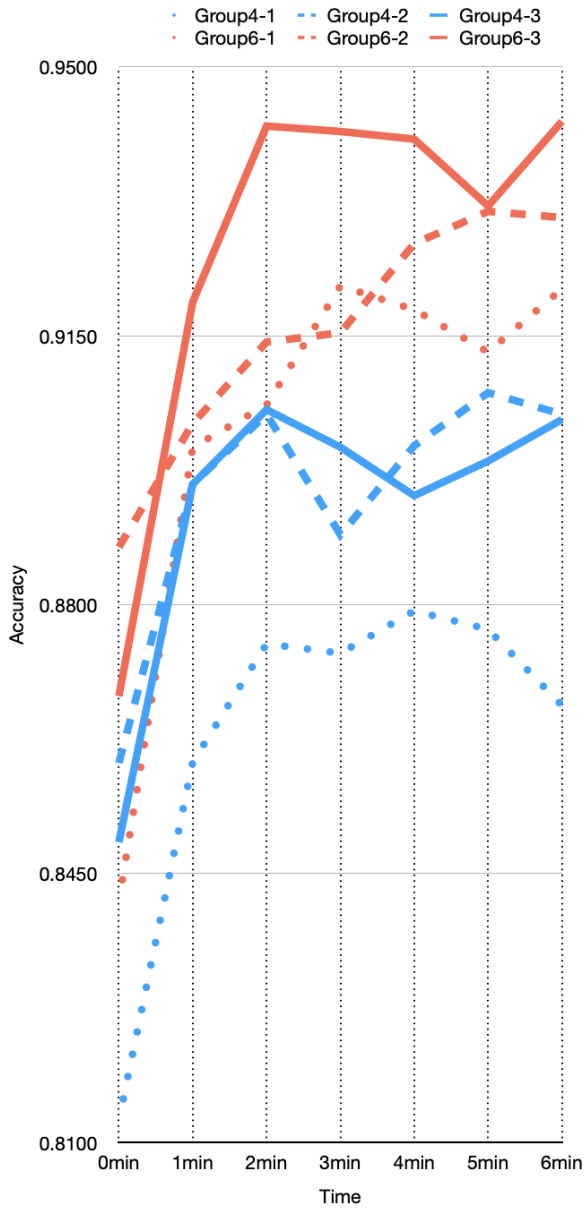

Figure 4.7: accuracy result from group 4 and group 6

The second set of results compared the performance between Group 4 and Group 6. Both groups had 10 participants and were tested in a CFL setting. Group 4 used the default selection strategy with random artificial constraints, while Group 6 also had the same random constraints but switched to a priority selection strategy. The results showed that Group 4 performed slightly better than Group 6. There are several explanations for this result. Firstly, using the priority strategy with random constraints may have made a difference as it can outperform the default selection strategy. Since both groups had random artificial constraints, the node conditions may not have been as favorable as in the first set of experiments. Therefore, the benefits of choosing priority selection were more evident in this setting. Secondly, during these random constraint settings, the default selection strategy performed poorly due to limitations in computational power

and latency which slowed down convergence speed. Additionally, including all neighboring nodes in the default selection strategy further exacerbated issues related to latency and computational resources. It is possible that these factors combined contributed to the performance difference observed between Groups 4 and 6. In simpler terms, when node conditions are complex and non-uniform (closer to real-world scenarios), the selective nature of priority selection may lead to better performance compared to default selection strategies.

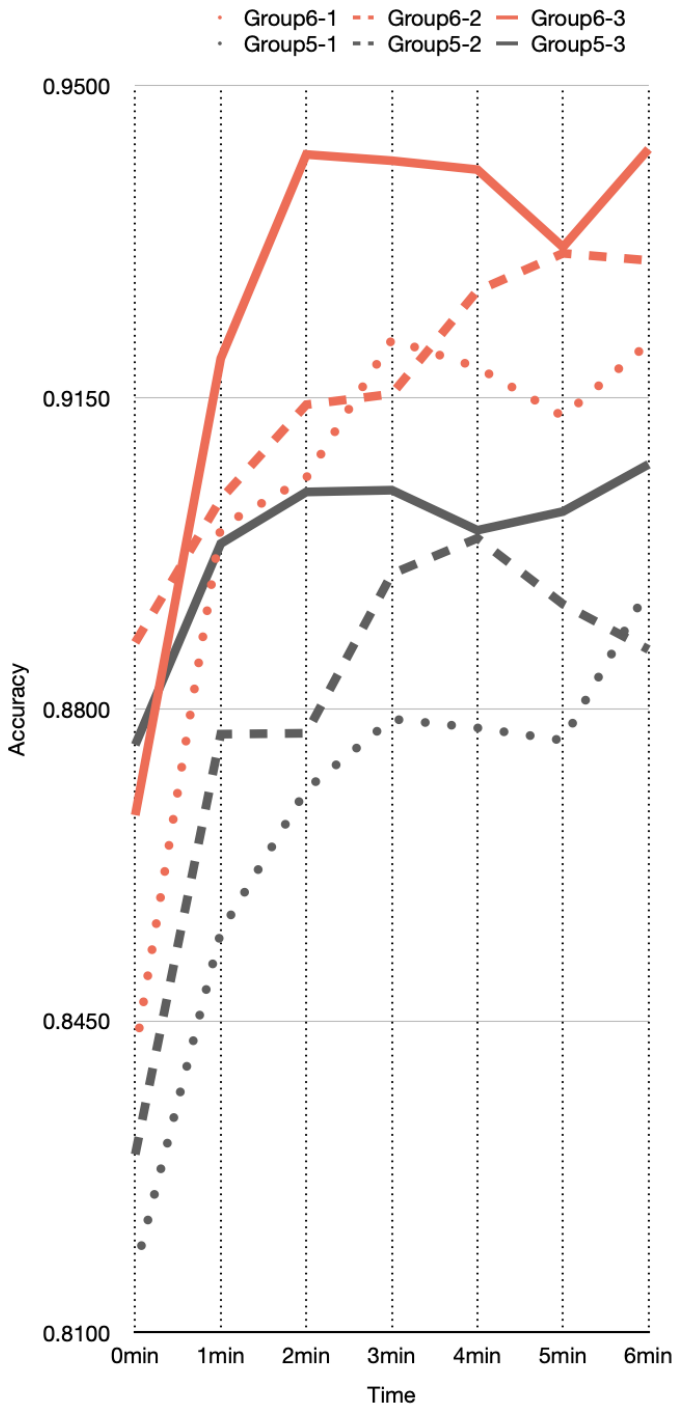

Figure 4.8: accuracy result from group 5 and group 6

Now, when shifting the attention to the results from Groups 5 and 6, an obvious pattern emerges. The priority selection in Group 6 manages to surpass the random selection in Group 5, despite both groups dealing with the same random constraints. This suggests that the priority selection does possess selection advantages. Not only is it more efficient

with quicker convergence rates, but it also shows a strategic edge by cherry-picking nodes that helps convergence times. This is in contrast to the random selection strategy, which relies on only limiting the selected node number but have no effect on the convergence performance rather than intentionally selecting the nodes which helps in speed up the convergence time.

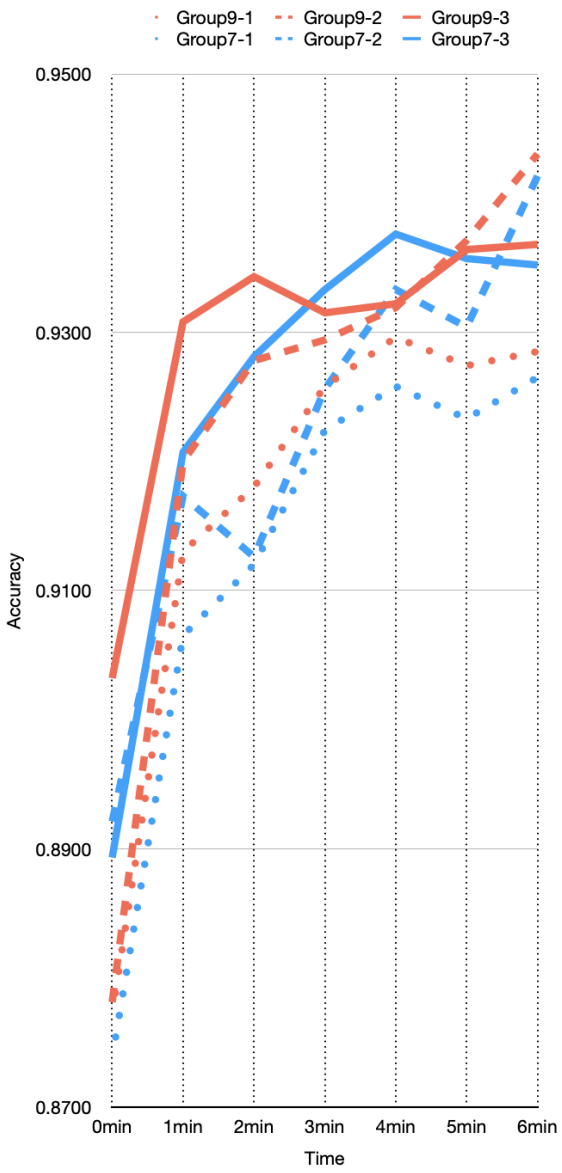

Figure 4.9: accuracy result from group 7 and group 9

The third set of results compares Group 7 and Group 9 in the DFL setting with a total of 10 participants. Group 7 represents the scenario without artificial constraints using the default selection strategy, while Group 9 represents the scenario with no artificial constraints using the priority selection strategy. In Figure 4.10, it is shown that the priority selection strategy performs almost the same as the default node selection strategy

in DFL. Similar to the result in CFL setting, there are also several possible reasons for it:

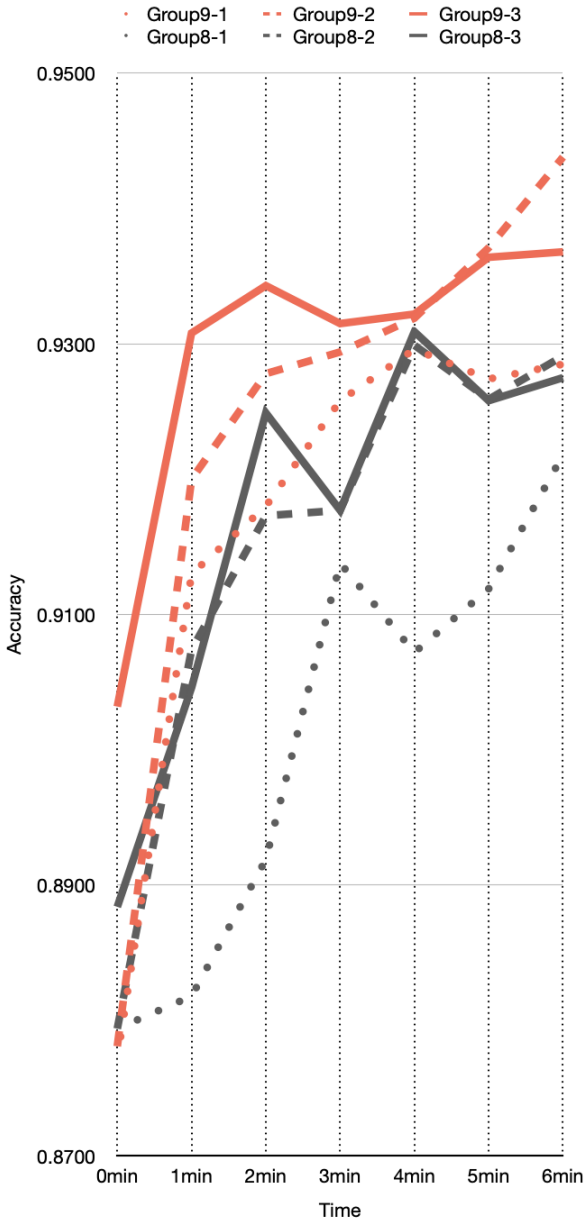

Figure 4.10: accuracy result from group 8 and group 9

1. The priority selection strategy did not improve federated learning efficiency in the DFL setting. 2. In DFL, every node acts as an Aggregator and performs node selection, which leads to higher computational consumption compared to CFL. The improved efficiency due to the priority selection is compromised by the extra computational consumption in DFL. Therefore, the convergence rate did not shown much difference between priority selection and default selection strategy. 3. Similar to CFL, decentralized federated learning is limited by dataset and timeframe when it comes to benefiting from priority node selection strategies. 4. It is possible that both reasons two and three contribute to this

result, resulting in similar performance between the priority selection strategy and default selection strategy.

In summary, although DFL provides a distributed approach to federated learning, the decentralization also brings computational demands that can limit the benefits of strategies like priority selection. This situation involves various factors that interact and affect the outcome.

An analysis of the results comparing Group 8 with Group 9 provides insights into the result in DFL setting. Group 8 used a random selection strategy and the result comparing Group 8 and 9 showed that implementing the algorithm increased computational load, leading to a subtle decrease in convergence rate. However, when compared to the priority selection strategy of Group 9, the difference was not significant. Several factors contribute to this outcome.

One key factor is the all-connected topology inherent in DFL, where every Aggregator utilizes a global model derived from neighboring nodes instead of working in CFL setting. This enables faster convergence compared to CFL settings where nodes rely on a centralized server. The collaborative approach promotes better accuracy achievement. Additionally, considering the MNIST dataset for its quick convergence properties. When deploying it in a DFL scenario with an all-connected structure, it is already an efficient environment for convergence.

Consequently, there is minimal room for improvement regardless of whether node selection algorithms are random or priority-based; They have limited potential to enhance already speed up convergence further. This likely explains why efficiency levels across Groups 7, 8, and 9 appear closely aligned. The combination of DFL environment and MNIST dataset creates a ceiling effect where differences between selection strategies are barely noticeable.

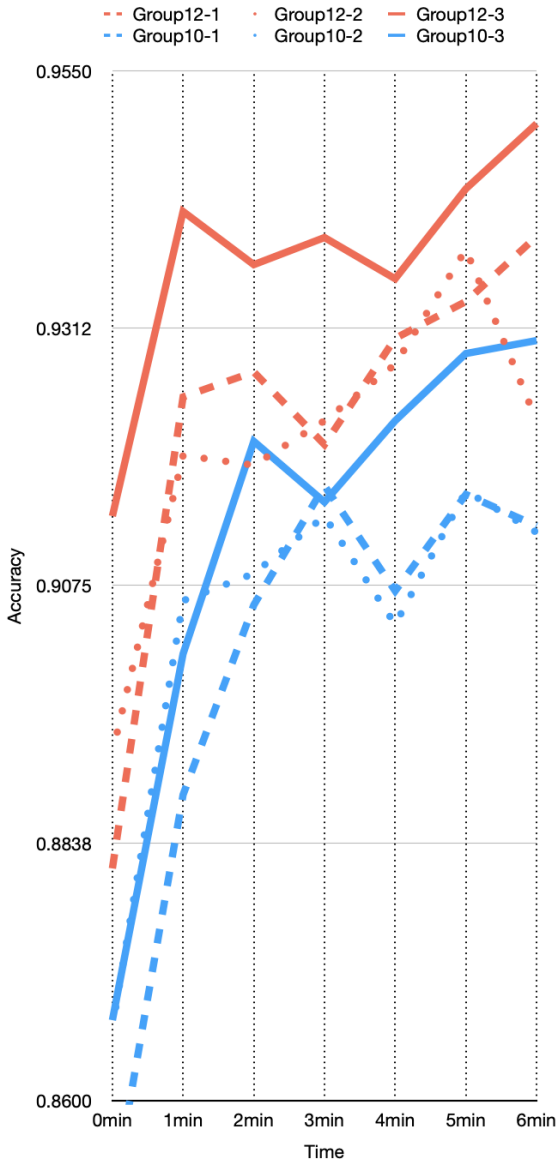

Figure 4.11: accuracy result from group 10 and group 12

The last set of results compares Group 10 and Group 12 in the DFL setting with a total of 10 participants. Group 10 represents the scenario with random artificial constraints using the default selection strategy, while Group 12 represents the scenario with random artificial constraints using the priority selection strategy. Figure 4.11 shows that the priority selection strategy outperforms the default node selection strategy.

This result indicates that, during the DFL setting, random artificial constraints can limit the performance of the default selection strategy and negatively impact convergence rate due to non-ideal device conditions. By choosing the priority selection strategy under such non-ideal conditions, participants can exclude neighbors that would greatly affect convergence rate and improve efficiency in federated learning.

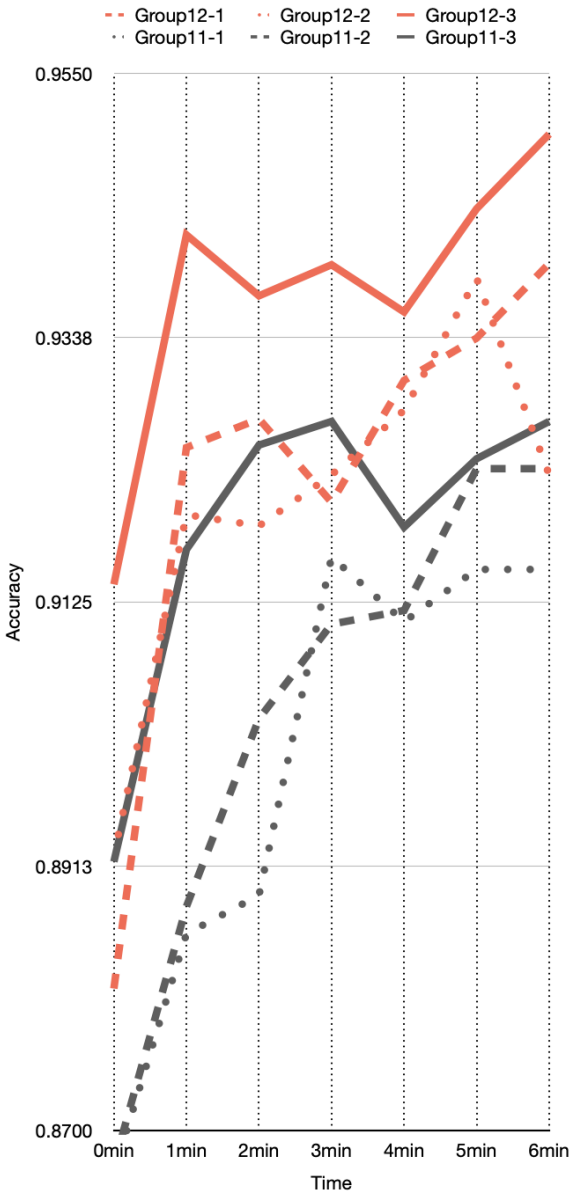

Figure 4.12: accuracy result from group 11 and group 12

Comparing this result to when Group 10 is compared to Group 11, it is observed that there is no significant increase in convergence rate when using a random selection strategy compared to using a default selection strategy. This similarity between results from both strategies further supports that the priority selection strategy has a positive effect on efficiency in DFL.

In summary, the evaluation part is aimed to investigate how federated learning performs under different network conditions with particular node selection strategy. We initially considered three datasets but ultimately chose the MNIST dataset paired with the MLP model due to its reliability. To provide a comprehensive comparison, we evaluated it against both the default and random selection methods.

In the initial findings using centralized federated learning (CFL), the result showed that priority selection performed similarly to the default method. It appeared that the additional computations required by the priority method might be offsetting its potential benefits. However, when introducing random variables or challenges, the priority method demonstrated its advantage in the conditions which similar to real-world settings. Transitioning to decentralized federated learning (DFL), the result is not the same as the CFL setting. Decentralization is more computationally demanding by nature.The result comparison showed the priority method maintained its advantage, particularly when faced with random constraints. Comparing priority selection to random selection in DFL revealed the efficiency but not significant differences. The fully connected topology configuration of DFL and quick convergence capabilities of MNIST limited the performance variations among node selection strategy. In summary, the findings indicate that while the priority selection method has some benefits, its efficiency depends on specific environmental factors. The complex scenarios may benefit from this method as an advantageous tool; however controlled environments may not yield as pronounced advantages.

Chapter 5

Summary and Future work

5.1 Conclusion

This thesis provides a comprehensive review of federated learning, with a specific focus on node selection strategies. After evaluating the current node selection strategy and identifying its limitations, this study proposes a novel priority selection. The priority selection approach utilizes the features of the node, which are continuously monitors. These features are then employed to determine the priority probability for choosing the node for model aggregation. The proposed strategy is implemented in the Fedstellar framework, which is then evaluated against default and random selection strategies.

The evaluation section begins by discussing the choice of dataset for experiments. Initially considering MNIST, CIFAR10, and SYSCALL datasets, we ultimately settled on using MNIST paired with an MLP model due to stability issues and consistency requirements.

The main investigation compares the performance of the priority selection strategy with default and random strategies in CFL environments. Initially, our results showed that the priority method performed similarly to the default method but raised concerns about computational overheads and real-world applicability. However, as the experiment introduced constraints and random variables into our experiments, such as limiting the CPU computational power and adding latency between nodes for simulating a bad network environment, it became evident that the priority selection method exhibited greater resilience and adaptability under changing conditions.

Transitioning to DFL setups where each node communicates with every other presented new challenges. In ideal conditions without any constraints applied to devices, there was no clear benefit observed from choosing the priority selection strategy. However, when applying random values for CPU limit and latency as constraints to devices during experimentation, slight improvements were seen in efficiency and convergence rates compared to other methods used.

To conclude the findings, when devices are in non-uniform conditions and their physical condition is not ideal for federated learning, priority selection yields better results compared to when devices are in ideal and identical conditions.

5.2 Future work

While this thesis provides some insights into federated learning and node selection strategies, there is still much more to explore. Additionally, upon reflecting on the thesis, there are some areas that can be enhanced and refined in future studies. Here are some potential directions for future research:

The first one is to explore other datasets. While MNIST with the MLP model was the final choice, it would be insightful to rectify the issues faced with CIFAR10 and SYSCALL and use them in similar experiments. The convergence rate of the CIFAR10 and SYSCALL datasets is slower compared to the MNIST dataset. As a result, running federated learning with these datasets can lead to more significant changes in accuracy. Additionally, it may be necessary to extend the timeframe in order to accurately reflect the impact of changing node selection strategies.

One improvement is to enhance the expandability and configurability of the feature extraction and selection module. Currently, this module is inflexible in terms of adding new features. When additional features need to be extracted, modifications must be made across various aspects such as feature transmission, storage, and probability calculation in the selection algorithm. To address this issue, it would be beneficial to refine the code so that the workflow can easily adapt to a wider range of features.

Besides the refinement in implementation part of the work, there are some aspect on the algorithm which could be explored more in the future works. The current study uses priority selection based on seven device features. However, this is just the beginning of exploring feature-based node selection. To fully utilize this approach, a more thorough exploration of features is necessary.

The works in the future could expand this to a more variety aspect of features of devices. For example, data distribution patterns can provide valuable insights into how data is spread across devices and how it affects the efficiency of the learning process. The size of RAM can have a significant impact on a device's ability to handle larger models or datasets. Device temperature, which often reflects computational load, can be used as an indicator of a device's current operational state. Additionally, factors like battery life, network strength, and geographical location may introduce interesting dimensions that could influence priority decision-making in during node selection.

The second aspect is the weight that is added to the features when calculating the selection probability could be studied more extensively. Since using static or arbitrary weights can be limiting, and the weight of the feature will greatly affects priority selection behavior, so it is important to carefully design its value. For example, if the network is highly sensitive to latency and even a slight increase in latency negatively impacts convergence rate, the weight assigned to latency should be increased. Future research could also focus on adaptive weighting mechanisms, such as employing an algorithm to adjust weights based on real-time performance feedback. This dynamic recalibration allows the system to continuously refine its strategy and optimize for current conditions.

In conclusion, this thesis provides a foundational understanding of federated learning. However, there is still much to explore in the area of node selection based on device

features. The proposed directions aim to deepen our understanding of selection strategy in federated learning and the future works could help maximize the benefits of the priority selection.

Bibliography

[1] C. Che, X. Li, C. Chen, X. He, and Z. Zheng, "A decentralized federated learning framework via committee mechanism with convergence guarantee", *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4783–4800, Dec. 1, 2022, ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2022.3202887. [Online]. Available: https://ieeexplore.ieee.org/document/9870745/ (visited on 05/12/2023).

[2] A. Sultana, M. M. Haque, L. Chen, F. Xu, and X. Yuan, "*Eiffel* : Efficient and fair scheduling in adaptive federated learning", *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4282–4294, Dec. 1, 2022, ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2022.3187365. [Online]. Available: https://ieeexplore.ieee.org/document/9810502/ (visited on 05/11/2023).

[3] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, *Federated optimization: Distributed machine learning for on-device intelligence*, Oct. 8, 2016. arXiv: 1610.02527[cs]. [Online]. Available: http://arxiv.org/abs/1610.02527 (visited on 05/11/2023).

[4] D. C. Nguyen, M. Ding, P. N. Pathirana, A. Seneviratne, J. Li, and H. Vincent Poor, "Federated learning for internet of things: A comprehensive survey", *IEEE Communications Surveys & Tutorials*, vol. 23, no. 3, pp. 1622–1658, 2021, ISSN: 1553-877X, 2373-745X. DOI: 10.1109/COMST.2021.3075439. [Online]. Available: https://ieeexplore.ieee.org/document/9415623/ (visited on 09/25/2023).

[5] T. Wang, Y. Liu, X. Zheng, H.-N. Dai, W. Jia, and M. Xie, "Edge-based communication optimization for distributed federated learning", *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 4, pp. 2015–2024, Jul. 1, 2022, ISSN: 2327-4697, 2334-329X. DOI: 10.1109/TNSE.2021.3083263. [Online]. Available: https://ieeexplore.ieee.org/document/9446648/ (visited on 03/23/2023).

[6] E. T. M. Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, *Decentralized federated learning: Fundamentals, state-of-the-art, frameworks, trends, and challenges*, Apr. 24, 2023. arXiv: 2211.08413[cs]. [Online]. Available: http://arxiv.org/abs/2211.08413 (visited on 05/11/2023).

[7] H. Wu and P. Wang, *Node selection toward faster convergence for federated learning on non-IID data*, Feb. 2, 2022. arXiv: 2105.07066[cs]. [Online]. Available: http://arxiv.org/abs/2105.07066 (visited on 04/09/2023).

[8] T. Nishio and R. Yonetani, *Client selection for federated learning with heterogeneous resources in mobile edge*, May 2019. DOI: 10.1109/ICC.2019.8761315. arXiv: 1804.08333[cs]. [Online]. Available: http://arxiv.org/abs/1804.08333 (visited on 04/23/2023).

[9] E. T. M. Beltrán, Á. L. P. Gómez, C. Feng, *et al.*, "Fedstellar: A platform for decentralized federated learning", 2023, Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2306.09750. [Online]. Available: https://arxiv.org/abs/2306.09750 (visited on 09/10/2023).

[10] G. S. Ramachandran, R. Radhakrishnan, and B. Krishnamachari, "Towards a decentralized data marketplace for smart cities", in *2018 IEEE International Smart Cities Conference (ISC2)*, Kansas City, MO, USA: IEEE, Sep. 2018, pp. 1–8, ISBN: 978-1-5386-5959-5. DOI: 10.1109/ISC2.2018.8656952. [Online]. Available: https://ieeexplore.ieee.org/document/8656952/ (visited on 09/11/2023).

[11] Z. Lian, Q. Yang, W. Wang, *et al.*, "DEEP-FEL: Decentralized, efficient and privacy-enhanced federated edge learning for healthcare cyber physical systems", *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 5, pp. 3558–3569, Sep. 1, 2022, ISSN: 2327-4697, 2334-329X. DOI: 10.1109/TNSE.2022.3175945. [Online]. Available: https://ieeexplore.ieee.org/document/9779505/ (visited on 09/11/2023).

[12] L. He, A. Bian, and M. Jaggi, "COLA: Decentralized linear learning",

[13] Z. Tang, S. Shi, and X. Chu, *Communication-efficient decentralized learning with sparsification and adaptive peer selection*, Feb. 22, 2020. arXiv: 2002.09692[cs, stat]. [Online]. Available: http://arxiv.org/abs/2002.09692 (visited on 05/11/2023).

[14] M. M. Amiri, D. Gunduz, S. R. Kulkarni, and H. V. Poor, *Convergence of update aware device scheduling for federated learning at the wireless edge*, May 8, 2020. arXiv: 2001.10402[cs,math]. [Online]. Available: http://arxiv.org/abs/2001.10402 (visited on 04/23/2023).

[15] J. Ren, Y. He, D. Wen, G. Yu, K. Huang, and D. Guo, *Scheduling for cellular federated edge learning with importance and channel awareness*, Jun. 23, 2020. arXiv: 2004.00490[cs,math]. [Online]. Available: http://arxiv.org/abs/2004.00490 (visited on 04/23/2023).

[16] H. Yu, Z. Liu, Y. Liu, *et al.*, "A fairness-aware incentive scheme for federated learning", in *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, New York NY USA: ACM, Feb. 7, 2020, pp. 393–399, ISBN: 978-1-4503-7110-0. DOI: 10.1145/3375627.3375840. [Online]. Available: https://dl.acm.org/doi/10.1145/3375627.3375840 (visited on 05/12/2023).

[17] C. Chen, H. Xu, W. Wang, *et al.*, "Communication-efficient federated learning with adaptive parameter freezing", in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, DC, USA: IEEE, Jul. 2021, pp. 1–11, ISBN: 978-1-66544-513-9. DOI: 10.1109/ICDCS51616.2021.00010. [Online]. Available: https://ieeexplore.ieee.org/document/9546506/ (visited on 03/09/2023).

[18] Z. Tang, S. Shi, B. Li, and X. Chu, "GossipFL: A decentralized federated learning framework with sparsified and adaptive communication", *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 909–922, Mar. 1, 2023, ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2022.3230938. [Online]. Available: https://ieeexplore.ieee.org/document/9996127/ (visited on 04/08/2023).

[19] Y. J. Cho, J. Wang, and G. Joshi, *Client selection in federated learning: Convergence analysis and power-of-choice selection strategies*, Oct. 2, 2020. arXiv: 2010.01243[cs,stat]. [Online]. Available: http://arxiv.org/abs/2010.01243 (visited on 04/23/2023).

[20] M. Ribero and H. Vikalo, *Communication-efficient federated learning via optimal client sampling*, Oct. 14, 2020. arXiv: 2007.15197[cs,stat]. [Online]. Available: http://arxiv.org/abs/2007.15197 (visited on 05/07/2023).

[21] M. Chen, H. V. Poor, W. Saad, and S. Cui, *Convergence time optimization for federated learning over wireless networks*, Mar. 26, 2021. arXiv: 2001.07845[cs,math,stat]. [Online]. Available: http://arxiv.org/abs/2001.07845 (visited on 04/23/2023).

[22] N. Onoszko, G. Karlsson, O. Mogren, and E. L. Zec, *Decentralized federated learning of deep neural networks on non-iid data*, Jul. 20, 2021. arXiv: 2107.08517[cs]. [Online]. Available: http://arxiv.org/abs/2107.08517 (visited on 05/12/2023).

[23] W. Chen, S. Horvath, and P. Richtarik, *Optimal client sampling for federated learning*, Aug. 22, 2022. arXiv: 2010.13723[cs]. [Online]. Available: http://arxiv.org/abs/2010.13723 (visited on 04/23/2023).

[24] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization", 2017, Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1710.09854. [Online]. Available: https://arxiv.org/abs/1710.09854 (visited on 05/31/2023).

[25] H. H. Yang, Z. Liu, T. Q. S. Quek, and H. V. Poor, *Scheduling policies for federated learning in wireless networks*, Oct. 9, 2019. arXiv: 1908.06287[cs,eess,math]. [Online]. Available: http://arxiv.org/abs/1908.06287 (visited on 05/07/2023).

[26] H. H. Yang, A. Arafa, T. Q. S. Quek, and H. V. Poor, *Age-based scheduling policy for federated learning in mobile edge networks*, Oct. 31, 2019. arXiv: 1910.14648[cs,eess,math]. [Online]. Available: http://arxiv.org/abs/1910.14648 (visited on 05/11/2023).

[27] W. Shi, S. Zhou, and Z. Niu, *Device scheduling with fast convergence for wireless federated learning*, Nov. 3, 2019. arXiv: 1911.00856[cs,math]. [Online]. Available: http://arxiv.org/abs/1911.00856 (visited on 05/11/2023).

[28] J. Kang, Z. Xiong, D. Niyato, Y. Zou, Y. Zhang, and M. Guizani, *Reliable federated learning for mobile networks*, Oct. 14, 2019. arXiv: 1910.06837[cs]. [Online]. Available: http://arxiv.org/abs/1910.06837 (visited on 04/09/2023).

[29] L. Lyu, X. Xu, and Q. Wang, *Collaborative fairness in federated learning*, Aug. 27, 2020. arXiv: 2008.12161[cs,stat]. [Online]. Available: http://arxiv.org/abs/2008.12161 (visited on 05/12/2023).

[30] M. Habib Ur Rehman, A. Mukhtar Dirir, K. Salah, and D. Svetinovic, "FairFed: Cross-device fair federated learning", in *2020 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, Washington DC, DC, USA: IEEE, Oct. 13, 2020, pp. 1–7, ISBN: 978-1-72818-243-8. DOI: 10.1109/AIPR50011.2020.9425266. [Online]. Available: https://ieeexplore.ieee.org/document/9425266/ (visited on 05/12/2023).

[31] Giampaolo Rodola, *Psutil*, version 5.9.5. [Online]. Available: https://github.com/giampaolo/psutil.

[32] thombashi, *Tcconfig*, version 0.28.0. [Online]. Available: https://github.com/thombashi/tcconfig.

List of Figures

Listings

List of Tables