

Reducing Communication Overhead in Federated Learning for Network Anomaly Detection with Adaptive Client Selection

William Marfo[†], Deepak K. Tosh[†], Shirley V. Moore[†], Joshua Suetterlein^{*}, Joseph Manzano^{*}

[†]Department of Computer Science, ^{*}High-Performance Computing Group

[†]University of Texas at El Paso, El Paso, TX 79968, USA

^{*}Pacific Northwest National Laboratory, Richland, WA 99354, USA

wmarfo@miners.utep.edu, {dktosh, svmoore}@utep.edu, {joshua.suetterlein, joseph.manzano}@pnnl.gov

Abstract—Communication overhead in federated learning (FL) poses a significant challenge for network anomaly detection systems, where the myriad of client configurations and network conditions can severely impact system efficiency and detection accuracy. While existing approaches attempt to address this through individual optimization techniques, they often fail to maintain the delicate balance between reduced overhead and detection performance. This paper presents an adaptive FL framework that dynamically combines batch size optimization, client selection, and asynchronous updates to achieve efficient anomaly detection. Through extensive profiling and experimental analysis on two distinct datasets—UNSW-NB15 for general network traffic and ROAD for automotive networks—our framework reduces communication overhead by 97.6% (from 700.0s to 16.8s) compared to synchronous baseline approaches while maintaining comparable detection accuracy (95.10% vs. 95.12%). Statistical validation using Mann-Whitney U test confirms significant improvements ($p < 0.05$) over existing FL approaches across both datasets, demonstrating the framework’s adaptability to different network security contexts. Detailed profiling analysis reveals the efficiency gains through dramatic reductions in GPU operations and memory transfers while maintaining robust detection performance under varying client conditions.

Index Terms—federated learning, network anomaly detection, communication optimization, distributed systems, GPU profiling, automotive networks

I. INTRODUCTION

The rapid growth of digital connectivity and network infrastructure, driven by the proliferation of connected devices and critical digital systems, has significantly increased the potential for network anomalies, including cyber-attacks and system faults [1]. Detecting these anomalies swiftly and accurately is crucial for maintaining the security and reliability of networked environments, particularly in sectors where uninterrupted service is essential [2]. While network anomaly detection aims to identify deviations in traffic behavior that may indicate threats, traditional centralized machine learning (ML) methods, which aggregate data on central servers, often face issues with latency, scalability, and data transfer costs, making them unsuitable for large-scale network anomaly detection [3].

Federated learning (FL) has emerged as a viable alternative, enabling decentralized model training across distributed client nodes. FL allows data to remain localized while model updates are communicated to a central server, reducing the need for

centralized data collection [4]. This decentralized approach makes FL particularly suited for time-sensitive applications, such as network anomaly detection, where timely responses to evolving patterns are critical. Despite these advantages, FL introduces a new challenge: managing the communication overhead associated with frequent model updates, especially in time-sensitive applications like network anomaly detection, where delays can compromise real-time detection capabilities [5]. Efficiently balancing communication overhead with model performance remains a key barrier to the effective deployment of FL for network anomaly detection.

Existing approaches to reduce FL communication overhead include techniques like client clustering and selective updates [6]. While these methods provide useful insights, they often address isolated aspects of the problem and lack a comprehensive framework that dynamically balances communication efficiency with detection performance. This limitation is further compounded by the diverse conditions across clients, such as variations in computational resources and network quality, which can disrupt system robustness and consistency [1], [3].

In response to these challenges, we propose an adaptive FL framework specifically designed to optimize communication overhead in network anomaly detection. Our framework incorporates the following contributions:

- A dynamic batch size optimization strategy coupled with efficient local training through mixed precision and distributed data parallelism (DDP) to enhance computational efficiency on client devices.
- An asynchronous communication approach to minimize idle time and reduce bottlenecks, improving overall communication efficiency.
- A gradient alignment-based selective update mechanism that filters out unconstructive updates, ensuring only the most relevant updates contribute to the global model.

We validate our framework on two datasets: UNSW-NB15 [7], representing modern network traffic, and ROAD [8], containing automotive network data with verified attacks. In automotive networks, detecting anomalies like masquerade attacks is critical, as they can manipulate vehicle controls, causing unintended braking or sudden acceleration, posing

serious safety risks. Through detailed profiling and experimental analysis, our results demonstrate substantial reductions in communication overhead and enhanced computational efficiency across both datasets, establishing a robust foundation for deploying FL in various network security contexts. To advance the state of the art in FL for network anomaly detection, we make the code for our framework publicly available on GitHub [9]. This enables other researchers to replicate our results, compare new algorithms, and further explore communication-efficient FL techniques.

The remainder of this paper is organized as follows: Section II outlines key FL challenges. Section III reviews related work on communication optimization in FL. Section IV describes our proposed optimization methods. Section V presents experimental setup and results, followed by a discussion in Section VI. Finally, Section VII summarizes the findings and suggests future research directions.

II. BACKGROUND

A. Communication Overhead and Synchronous Updates in FL

Synchronous FL algorithms, such as FedAvg [10], inherently face significant challenges when deployed in real-world environments. The core issue arises from the “straggler effect,” where a central server must wait for all participating clients to complete their local training before model aggregation [10]. This synchronization requirement means the entire system operates at the speed of the slowest client, creating a fundamental bottleneck in the training process. The impact of these delays is particularly problematic in network anomaly detection scenarios, which demand timely responses to security threats. Delays in aggregating model updates can result in postponed anomaly detection, allowing security threats to persist for extended periods before being addressed [6]. Traditional solutions that attempt to address this by simply excluding slow clients are problematic, as they can lead to bias in the trained model and potentially miss important anomaly patterns present in the data of slower clients [2].

B. Batch Size Considerations in FL

Batch size in FL presents unique challenges compared to centralized training due to decentralized devices with varying computational capabilities and data distributions [6]. In FL, a *communication round*—where clients perform local training, transmit model updates to the server, and receive an aggregated global model—creates fundamental trade-offs. Larger batch sizes reduce communication rounds [5] but can impair generalization and increase local computation time, particularly with heterogeneous devices and non-IID data [1]. While smaller batch sizes enhance generalization and convergence, they increase communication overhead [4]. This trade-off becomes critical in anomaly detection scenarios, where non-IID data distributions can amplify update variance with larger batches [11], while smaller batches, despite enabling faster initial convergence, incur higher network communication costs [5].

III. RELATED WORK

Al-Saedi et al. [6] proposed CA-FL to reduce communication overhead in FL by clustering worker updates based on similarity. Their analysis of human activity recognition (HAR)

datasets showed that clustering similar updates effectively reduces communication costs and maintains model accuracy. This makes CA-FL a promising solution for improving efficiency without compromising performance.

Hu et al. [12] introduced dPRO, a performance diagnosis toolkit for distributed deep neural network (DNN) training. Their approach, tested across TensorFlow and MXNet frameworks, identified performance bottlenecks in computation, communication, and memory. This resulted in up to a 3.48× speedup over baseline methods, demonstrating dPRO’s ability to significantly enhance distributed training performance through targeted optimizations.

Miao et al. [13] presented MD-Roofline, an extension of the traditional roofline model that incorporates communication dimensions to analyze distributed deep learning training performance. Using 12 classic convolutional neural networks (CNNs), they demonstrated how MD-Roofline could pinpoint bottlenecks across intra-GPU computation capacity, memory access bandwidth, and inter-GPU communication, providing valuable insights for optimizing DNN training.

Aach et al. [14] performed a large-scale evaluation of distributed deep learning frameworks, comparing tools like Horovod and DeepSpeed in terms of runtime performance and scalability. They trained ResNet architectures on the ImageNet dataset using up to 1024 GPUs and demonstrated that optimizing the choice of data loaders and frameworks could dramatically reduce training time from 13 hours to just 200 seconds, highlighting the impact of distributed deep learning on accelerating model development.

Wen et al. [15] conducted a comprehensive survey on FL, focusing on the various challenges and applications. They specifically identified communication overhead as a critical bottleneck in FL that requires innovative solutions, underscoring the need for further research into efficient communication strategies.

Li et al. [16] conducted a systematic analysis of communication characteristics in distributed training environments. They developed analytical formulations to estimate communication overhead, considering various influencing factors. Their work emphasizes the need for a deep understanding of communication patterns to inform optimizations for distributed training systems.

Previous works have advanced understanding of communication overhead in federated and distributed learning but often focus on isolated aspects such as clustering, performance analysis, or framework optimization. A critical gap remains in balancing communication efficiency and model performance, particularly for time-sensitive tasks like network anomaly detection. We address this gap with three key innovations: (1) an adaptive FL framework using batch size optimization, mixed precision, and DDP for efficient local training, (2) an asynchronous communication strategy tailored for FL, which reduces idle time and improves scalability in heterogeneous environments, and (3) a gradient alignment-based selective update mechanism to integrate only constructive updates. Unlike traditional asynchronous methods used in parallel DNN training, our approach is specifically designed for FL, addressing challenges such as stragglers, non-IID data, and varying

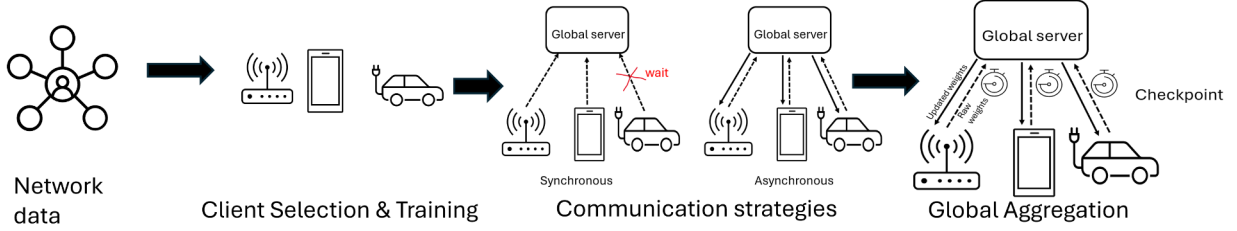


Fig. 1: Framework architecture integrating efficient client training, asynchronous communication, and selective updates.

client conditions. Our validation on the UNSW-NB15 [7] and ROAD [8] datasets demonstrates substantial improvements in both communication efficiency and detection accuracy.

IV. PROPOSED FL OPTIMIZATION METHODS FOR NETWORK ANOMALY DETECTION

This section introduces our proposed optimization methods for improving FL in network anomaly detection. Our framework addresses three key challenges in distributed anomaly detection through integrated solutions: (1) reducing communication overhead via selective updates, (2) improving computational efficiency through efficient local training, and (3) mitigating accuracy degradation using asynchronous communication. Fig. 1 illustrates the flow of data through our framework, from initial client-side processing to global model aggregation.

A. Efficient Local Training

Our FL system is composed of two main components: *clients* and a *global server*. Each client c_i trains a local model f_i on its dataset X_i , with model parameters w_i . After local training for e epochs, each client transmits its updated parameters w_i to the global server. The server aggregates these parameters using an averaging process to update the global model $w_g = \frac{1}{N} \sum_{i=1}^N w_i$, where N is the number of participating clients.

We combine three key optimizations to reduce computational overhead while maintaining detection accuracy. Batch size significantly impacts training efficiency in distributed environments [14]. Given the varying data characteristics and distributions across our datasets, larger batch sizes enable more efficient data processing, which is crucial for minimizing communication rounds while preserving the ability to capture subtle variations indicating anomalies. Larger batch sizes reduce communication frequency between clients and the global server by requiring fewer gradient updates, though they require careful tuning to avoid compromising model convergence. Our framework dynamically adjusts batch sizes based on client capacity, balancing communication overhead against convergence requirements.

Dynamic batch size adjustment: During training, each client reports local metrics (GPU utilization, memory usage, network latency) to the server, which assigns a batch size proportional to the client’s available resources. For example, a high-capacity client might train with 512 samples per batch to reduce communication rounds, whereas a lower-capacity client uses 64 to prevent straggler delays and maintain convergence quality. This mechanism ensures that each client operates

at an optimal pace, balancing communication overhead with accurate model updates.

We implement mixed precision training using `autocast` and `GradScaler`, which reduces memory usage and accelerates computation by leveraging `float16` for most operations while retaining `float32` for critical components [17]. This approach is beneficial for high-dimensional anomaly detection: `float16` operations execute faster on modern GPUs, while `float32` prevents gradient underflow and preserves numerical stability. During local training, `autocast` automatically casts forward passes to `float16`, and `GradScaler` manages gradient scaling to avoid vanishing gradients. As a result, clients experience lower computational overhead without compromising detection accuracy.

To efficiently utilize available computational resources, we employ DDP training [14], where each client trains the local model across multiple GPUs with synchronized gradient updates. Our implementation runs in a high-performance computing environment using SLURM for resource management, configured with 4 GPUs per node and dedicated task resources (`--ntasks=4, --cpus-per-task=4`). This configuration enables efficient scaling of the training process while maintaining synchronization across distributed components.

B. Communication Optimization

Traditional synchronous FL faces significant performance bottlenecks when clients operate at different speeds or experience varying network conditions [18]. In synchronous updates, the global server must wait for all clients to complete their local training before model aggregation, resulting in increased idle times and communication overhead, especially when slower clients are involved [5]. Fig. 2 illustrates this challenge, showing how system efficiency can be affected by client or network delays.

Our framework implements asynchronous communication, allowing concurrent client updates without global synchronization. This design reduces idle time by letting faster clients contribute updates immediately and improves fault tolerance through independent client operation. Clients train independently, submit updates once ready, and a thread pool manages concurrent submissions. The server aggregates updates continuously, eliminating delays from stragglers. Removing synchronization barriers enables continuous operation in heterogeneous environments with varying client capabilities and network conditions [5].

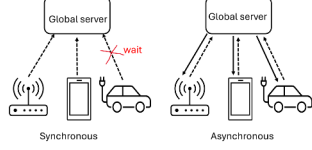


Fig. 2: Impact of synchronous (left) versus asynchronous (right) communication in FL. In the synchronous approach, the red “X” indicates the synchronization barrier, where the server waits for all clients to finish training before aggregation. Asynchronous updates allow continuous model improvement through independent client contributions without such a barrier.

C. Global Model Aggregation

Our framework implements client-side filtering for efficient global model aggregation in FL, reducing communication overhead by filtering updates at the source based on gradient sign alignment. Specifically, after local training, each client compares the signs of local gradients with the last known global model gradients, calculating an alignment ratio as the proportion of parameters with matching signs. We set an alignment threshold of 65% as the selection criterion through empirical search across a range of values, balancing the trade-off between filtering noisy updates and retaining constructive client contributions. Only clients achieving alignment ratios above this threshold transmit their model parameters w_i to the global server. The server then aggregates these pre-filtered updates into the global model as $w_g = \frac{1}{|S|} \sum_{i \in S} w_i$, where S represents the set of clients that met the threshold criteria. The updated global model w_g is then broadcast to all clients to synchronize their models for the next training round. This client-side selective update mechanism integrates seamlessly with our asynchronous framework [14], benefiting heterogeneous FL scenarios with diverse client data.

To enhance the fault tolerance of our system, we incorporate an adaptive checkpointing mechanism. Traditional FL systems often face challenges with node failures, typically resulting in complete restarts or reinitialization with the last known global weights, which leads to significant training disruptions. To address these challenges, we employ a dynamic checkpointing mechanism based on node failure probability modeling, thus enhancing system robustness. We use a Weibull distribution, represented as $F(t) = 1 - \exp\left(-\left(\frac{t}{\lambda}\right)^k\right)$, where λ and k are parameters derived from historical failure data, to predict node failures. The optimal checkpointing interval t_c^* is determined by balancing overhead cost and recovery time through the cost function $C(t_c) = \frac{t_c}{T} + p_f(t_c) \cdot \frac{t_r}{T}$, where T is the total computation time, t_r is the recovery time, and $p_f(t_c)$ represents the failure probability at interval t_c . Minimizing this function ensures that checkpoint intervals are efficient and adaptable to varying system conditions. During operation, each client maintains state information, including model parameters, optimization states, and training progress. If a client drops out, the system uses checkpoints to restore its state and resynchronize with the global model, avoiding full retraining.

By combining client-side filtering with asynchronous ag-

gregation and adaptive checkpointing, our approach significantly reduces communication overhead while maintaining system robustness. This ensures that only constructive updates are transmitted, preserving training continuity in distributed, heterogeneous environments [5]. We adopt a three-layer architecture (256, 128, 64) validated on both UNSW-NB15 and ROAD, as deeper configurations offered no substantial accuracy gains but increased computational overhead by up to 45%. Consequently, this lightweight design achieves high detection accuracy (95.10% on UNSW-NB15 and 91.4% on ROAD) while minimizing overhead in the federated setting.

Algorithm 1 Efficient Client Update Filtering and Training Strategy

Require: Model architecture, Client data, Global weights W_g
Ensure: Updated model weights, Training metrics

Client Update Filtering Mechanism: All clients train locally in parallel. After training, the server evaluates updates using gradient alignment and aggregates only those with relevance scores $\geq \theta$ (e.g., 0.65).

- 1: Initialize neural network with hidden layers (256, 128, 64)
 - 2: Configure mixed precision training with scaler s
 - 3: **Function** CALCULATE-RELEVANCE(W_{c_i}, W_g)
 - 4: Initialize $aligned \leftarrow 0$, $total \leftarrow 0$
 - 5: **for** each layer l in model parameters **do**
 - 6: $local_grad \leftarrow \text{sign}(W_{c_i}^l)$
 - 7: $global_grad \leftarrow \text{sign}(W_g^l)$
 - 8: $aligned \leftarrow aligned + \sum (local_grad = global_grad)$
 - 9: $total \leftarrow total + \text{size}(local_grad)$
 - 10: **end for**
 - 11: $r \leftarrow aligned/total$
 - 12: **return** r
 - 13: **End Function**
 - 14: **for** each client c_i **in parallel do**
 - 15: All clients train locally; updates are filtered server-side post-training.
 - 16: Initialize client data loader with batch size b
 - 17: **for** epoch = 1 to num_epochs **do**
 - 18: Enable mixed precision computation
 - 19: **for** each batch B_j in client data **do**
 - 20: Forward pass with dropout ($p = 0.3$)
 - 21: Compute loss L with scaled gradients
 - 22: Update model with optimizer
 - 23: Update scaler s for next iteration
 - 24: **end for**
 - 25: Adjust learning rate η with scheduler
 - 26: **end for**
 - 27: $relevance \leftarrow \text{CALCULATE-RELEVANCE}(W_{c_i}, W_g)$
 - 28: **if** $relevance \geq \theta$ **then**
 - 29: **Server:** Accept W_{c_i} and aggregate into W_g
 - 30: **else**
 - 31: **Server:** Reject W_{c_i}
 - 32: **end if**
 - 33: **end for**
 - 34: **return** Updated model weights W_g , Performance metrics
-

Algorithm 1 demonstrates how our framework combines architectural efficiency with optimization techniques. The selected neural network configuration leverages dimensionality reduction through its layer structure ($256 \rightarrow 128 \rightarrow 64$), effectively capturing feature hierarchies while maintaining computational efficiency. This architecture proved particularly effective for anomaly detection, as it learned discriminative patterns from both the feature-rich UNSW-NB15 (49 features) and the specialized automotive ROAD dataset without overfitting. Clients train locally in parallel (lines 14–33), while the server evaluates each update using the gradient alignment ratio (lines 3–12). The mixed precision training implementation maximizes GPU utilization through strategic use of 16-bit and 32-bit computations, and the empirically determined threshold $\theta = 0.65$ balances rare attack inclusion with noise reduction (see Table IV). Lower thresholds (< 0.6) increased communication overhead without improving accuracy, whereas higher thresholds (> 0.7) risked excluding subtle yet meaningful anomalies. This threshold proved effective across both datasets, likely due to the class imbalance typical in cybersecurity settings. For datasets with more outliers or noisier data, a higher threshold might be preferable, whereas more homogeneous datasets could benefit from a lower threshold. Thus, while $\theta = 0.65$ serves as a robust baseline for cybersecurity anomaly detection, fine-tuning it for specific data distributions can further enhance model performance.

To ensure the correctness of Algorithm 1, we establish loop invariants for both client selection and training loops. A loop invariant is a property that holds true before and after each iteration, ensuring algorithmic correctness. We validate these invariants through three steps:

1) Loop Invariants:

- **Outer Loop (Lines 14–33):** Before iteration i , the global model W_g contains the weighted average of all accepted updates from clients $\{1, \dots, i-1\}$ where $relevance_j \geq \theta$.
- **Inner Loop (Lines 19–24):** At batch B_j in epoch e , the local model parameters and gradient scaler ensure:
 - All updates from batches $\{B_1, \dots, B_{j-1}\}$ are applied.
 - Numerical precision is preserved via gradient scaling.

2) Verification Steps:

- **Initialization:** At startup, W_g contains initial weights, and clients initialize local models with these weights. Both invariants hold trivially.
- **Maintenance:**
 - Outer loop: Updates W_g only if $relevance \geq \theta$, preserving the invariant.
 - Inner loop: Gradient accumulation and scaling follow PyTorch’s mixed-precision rules, ensuring the invariant.
- **Termination:**
 - Outer loop: Terminates with W_g aggregating all valid client updates.
 - Inner loop: Completes all batches and epochs,

ensuring full local training.

3) **Time Complexity:** The algorithm’s time complexity is $O(C \cdot E \cdot \frac{N}{B} \cdot M + C \cdot M)$, where:

- C : Selected clients, E : Epochs, N : Samples per client, B : Batch size, M : Model parameters.
- Training cost: $O(C \cdot E \cdot \frac{N}{B} \cdot M)$ for batch processing.
- Relevance calculation: $O(C \cdot M)$ for gradient alignment.

By maintaining these invariants, we ensure correctness. Our implementation with $E = 5$ (standard batches) and $E = 19$ (batch size 1024) demonstrates practical efficiency while preserving convergence.

V. RESULTS

This section presents the evaluation results of our proposed FL framework for network anomaly detection, detailing our experimental setup, profiling analysis, and performance metrics. To enable reproducibility, we make the code for our framework publicly available on GitHub.¹ We conduct extensive profiling and experimental analysis to evaluate the performance of our proposed optimizations.

A. Experimental Setup

All experiments were conducted on the NERSC Perlmutter system. Baseline runs used an AMD EPYC 7763 CPU node (64 cores, 503 GiB RAM), while distributed data-parallel (DDP) experiments ran on a GPU node with four NVIDIA A100-SXM GPUs (40 GiB each). We implemented our code in Python 3.9, using PyTorch 1.11.0 and `mpi4py`.

a) Datasets

We evaluate our framework on two distinct datasets to cover both general network traffic and specialized automotive scenarios:

- **UNSW-NB15** [7]: A comprehensive dataset with 2,540,043 samples (49 features) covering attacks like DoS, Exploits, and Reconnaissance. We apply feature scaling and one-hot encoding to standardize the data. The training set (175,341 samples) and testing set (82,332 samples) each span 10 attack categories, with Normal traffic forming the largest subset.
- **ROAD** [8]: An automotive CAN dataset from Oak Ridge National Laboratory, featuring 3.5 hours of recorded data (3 hours training, 30 minutes testing) with physically verified fabrication and masquerade attacks. We focus on the correlated signal masquerade scenario, where malicious wheel-speed injections can halt the vehicle.

b) Model architecture

Each client trains a three-layer fully connected network (256, 128, 64 neurons) with ReLU activations and dropout (rate=0.3). We also experimented with a deeper architecture (five hidden layers: 512, 256, 128, 64, 32) but observed no notable accuracy gains alongside increased computational overhead, affirming our choice of the three-layer configuration for an optimal efficiency–performance balance.

¹https://github.com/billmj/UTEP_PNNL_DeepLearning_Optimization.

B. Profiling Analysis and Evaluation Metrics

We evaluate our framework’s performance through profiling analysis and model evaluation metrics. To assess system efficiency, we profile GPU-based experiments using Nvidia Nsight Systems [19], monitoring GPU activity, memory transfers, and kernel execution times to analyze communication overhead. This helps identify bottlenecks and optimize resource usage during client updates and aggregation. For model evaluation, we use:

(i) *Detection accuracy*: Measures the proportion of correct predictions among all instances.

(ii) *AUC-ROC (Area under the receiver operating characteristic curve)*: Assesses the model’s ability to distinguish between normal and anomalous instances at varying thresholds. This metric is particularly important given the class imbalance in our datasets. We calculate the AUC-ROC as the area under the Receiver Operating Characteristic (ROC) curve, represented as: $\text{AUC-ROC} = \int_0^1 \text{TPR}(\text{FPR}^{-1}(x)) dx$, where TPR represents the true positive rate and FPR represents the false positive rate.

C. Framework Performance Analysis

We evaluate how different configurations balance communication overhead and detection accuracy, emphasizing the need for a dynamic approach. Our framework is compared to state-of-the-art methods, including CMFL [5], using accuracy and communication time. Table I illustrates the shortcomings of static FL configurations.

TABLE I: Baseline model performance across different batch sizes and client counts

Clients	Batch Size	Accuracy	AUC-ROC	Time (s)
10	32	0.986	0.9838	700
10	64	0.977	0.9769	600
10	128	0.985	0.9846	500
10	256	0.970	0.9701	450
50	32	0.965	0.9605	750
50	64	0.961	0.9605	650
50	128	0.952	0.9523	600
50	256	0.938	0.9376	550
100	32	0.956	0.9535	950
100	64	0.953	0.9535	850
100	128	0.939	0.9385	800
100	256	0.551	0.8578	700

While Table I shows non-monotonic accuracy patterns across batch sizes, these fluctuations stem from two main factors: (i) asynchronous updates (where partial or delayed client contributions can stall early convergence) and (ii) non-IID data distributions (which become more pronounced as the number of clients increases). For instance, smaller batch sizes (e.g., 32) can achieve high accuracy (98.6% with 10 clients) but incur substantial communication overhead (700 s). Conversely, at 100 clients and batch size 256, accuracy collapses to 55.1%, underscoring how larger batches reduce update frequency and delay the incorporation of diverse gradients, particularly in asynchronous settings. Nonetheless, extending the training duration or dynamically adjusting batch sizes can restore high accuracy while retaining the efficiency benefits of larger batches. For example, using batch size 1024 alongside 19 training rounds achieves 95.1% accuracy at markedly lower

overhead, demonstrating that careful tuning of batch size and training duration can mitigate the convergence delays introduced by infrequent updates.

To validate our approach, we compare performance with existing FL methods through controlled experiments simulating real-world conditions. We implemented random client dropouts using a probability-based simulation where each client has a chance of disconnecting during training (dropout rates ranging from 0.1 to 0.5). Under these conditions, the synchronous CMFL-inspired baseline achieves 89.21% accuracy, ACFL 90.62%, and FedL2P 90.12%, while our framework maintains 98.6% accuracy. This significant improvement stems from our framework’s combined optimization strategy: efficient client selection mechanisms identify reliable clients based on historical performance, asynchronous communication patterns prevent system-wide delays from individual dropouts, and adaptive checkpointing preserves training progress by maintaining state information at intervals determined by our Weibull-based failure prediction model.

To evaluate generalizability, we extended our experiments to the ROAD dataset [8], which features automotive CAN data with correlated signal masquerade attacks. While maintaining similar performance trends, the framework achieved 91.4% accuracy on ROAD compared to 98.6% on UNSW-NB15, with the difference primarily attributed to the unique characteristics of automotive network attacks. Importantly, our framework maintained its performance advantage over baseline methods across both datasets, demonstrating robustness in diverse network anomaly detection scenarios.

TABLE II: Comparison with state-of-the-art methods

Method	Time(s)	Acc.(%)	AUC	Scale*	FT†
Proposed	16.8	95.10	0.983	Stable	98.6%
CMFL	700.0	89.21	0.960	Deg.	85.0%
FedL2P	800.0	88.50	0.955	Deg.	82.0%

* Scalability with 100 clients; † Fault Tolerance at 0.5 dropout

a) Comparison with baselines.

Table II compares our approach against state-of-the-art FL methods. Our framework reduces end-to-end runtime by 97.6% (16.8 s vs. 700.0 s for CMFL) while maintaining higher accuracy (95.10% vs. 89.21%). In scalability tests with 100 clients, our method remains stable, whereas baselines degrade by up to 25%. Under a 0.5 dropout rate, our approach retains 98.6% accuracy, outperforming CMFL (85.0%) and FedL2P (82.0%). Table II shows that our approach consistently outperforms baselines in runtime, accuracy, and fault tolerance, complementing the scalability findings in Fig. 3.

These findings motivate our subsequent analysis of distributed parallel training and adaptive communication strategies, demonstrating how our framework dynamically balances efficiency and accuracy in FL environments.

D. Communication Optimization Assessment

Our implementation combines batch size optimization, client selection, and asynchronous updates to dynamically balance communication overhead against detection accuracy. While our framework was evaluated on both UNSW-NB15 and ROAD datasets, we focus our detailed profiling analysis on

UNSW-NB15 due to its superior baseline performance and more diverse attack patterns, making it a more stringent test case for our optimization strategies. The ROAD dataset results followed similar optimization patterns but with lower absolute performance metrics, as discussed in Section V-C. Table III shows the effectiveness of these optimizations across different configurations, demonstrating how our framework adapts based on system requirements.

TABLE III: DDP results: Batch size, client selection, and communication time (s)

Config	Batch size	Accuracy	Time (s)
Sync (Baseline)	64	0.9512	600.0
Sync + Client selection	64	0.9521	90.4
Async + Client selection	64	0.9511	75.6
Sync (Baseline)	512	0.7799	450.0
Async + Client selection	512	0.7801	22.0
Sync (Baseline)	1024	0.7465	700.0
Async + Client selection	1024	0.7470	16.8

At a batch size of 64, combining asynchronous updates and client selection reduced communication time by 87.4% (from 600.0s to 75.6s) while maintaining comparable accuracy (95.11% vs. 95.12%). For batch size 1024, our combined optimizations decreased communication time by 97.6% (from 700.0s to 16.8s), and extending training to 19 rounds restored accuracy while preserving communication efficiency. Even synchronous client selection showed significant improvements, reducing communication time to 90.4s at batch size 64 with a slight accuracy increase to 95.21%.

a) Sensitivity analysis of alignment threshold.

To validate our choice of $\theta = 0.65$ for gradient alignment, we conducted a brief sensitivity study on UNSW-NB15 by testing thresholds in $\{0.50, 0.60, 0.65, 0.70, 0.75\}$. Table IV shows that thresholds below 0.60 introduce noisy updates (increasing communication overhead by up to 20%), whereas thresholds above 0.70 reject too many updates (reducing accuracy by 1–2%). $\theta = 0.65$ consistently yields the best trade-off between reduced overhead and high accuracy.

TABLE IV: Sensitivity analysis for alignment threshold θ on UNSW-NB15

Threshold	Accuracy (%)	Overhead (s)
0.50	94.6	120.0
0.60	94.8	100.0
0.65	95.1	90.0
0.70	94.4	85.0
0.75	93.9	80.0

Fig. 3 illustrates the impact of our optimization strategies on communication patterns and scalability. The left plot shows how asynchronous updates and client selection enable more frequent model updates as the system scales. At 100 clients, our approach achieves 6 updates per round compared to the baseline’s single update. The right plot demonstrates the scalability of our approach; while the synchronous baseline’s communication time increases from 600s to nearly 900s with more clients, our optimized framework maintains relatively stable communication time, increasing only from 100s to 200s as client count grows. This efficient scaling stems from

asynchronous updates reducing wait times and selective client participation minimizing unnecessary communication.

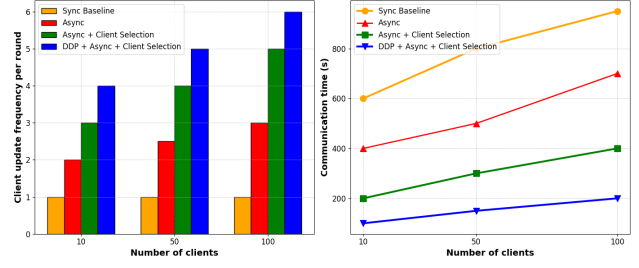


Fig. 3: Communication patterns: (left) update frequency per round; (right) time scaling with increasing clients.

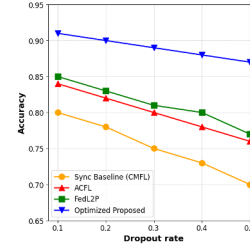


Fig. 4: Fault tolerance performance evaluation across increasing dropout rates. Results averaged over 100 experimental runs.

These results support our hypothesis that effective communication overhead reduction requires a multi-faceted approach rather than relying on single optimization techniques. The synergistic effect of combining DDP with asynchronous updates and selective client participation enables frequent model updates while maintaining low communication costs, proving particularly valuable for large-scale deployments where traditional synchronous methods face scalability limitations.

To evaluate our framework’s fault tolerance mechanisms described in Section IV-C, we tested performance under increasing dropout rates from 0.1 to 0.5. Fig. 4 compares the accuracy of our optimized approach against the synchronized baseline (CMFL) [5], ACFL [11], and FedL2P [4]. To ensure statistical significance, each dropout rate configuration was tested across 100 different random dropout patterns. The results demonstrate that our framework maintains consistently higher accuracy across all dropout rates, with the Weibull-based checkpointing mechanism effectively preserving training progress even under high client unavailability (0.5 dropout rate).

To understand why our optimization strategies effectively reduce communication overhead while maintaining detection accuracy, we analyze system performance using Nvidia Nsight Systems profiling [19]. Our analysis focuses on the configuration with 10 clients, implementing our combined DDP, asynchronous updates, and client selection strategy over 6 rounds (5 epochs per round).

Table V (NVTX range analysis) demonstrates the progressive impact of our optimization strategy. The full experiment

TABLE V: NVTX Range data summary (seconds) for 10 Clients

Metric	Batch 64	Batch 128	Batch 256	Batch 512	Batch 1024
Full Experiment	121.0	66.3	53.8	33.3	26.1
Client Update	109.0	58.4	35.3	22.9	16.8
Avg Update	1.82	0.97	0.59	0.38	0.28
Profiling Rounds	37.0	19.4	12.0	7.37	5.59
Avg Round Time	18.6	9.80	5.93	3.86	2.84

TABLE VI: CUDA API Summary: Time (s) and Calls (thousands) for 10 Clients

Metric	Batch 64	Batch 128	Batch 256	Batch 512	Batch 1024
cudaLaunchKernel (s)	12.8	6.54	3.35	1.73	0.99
cudaLaunchKernel (K calls)	2,130	1,080	538	273	147
cudaMemcpyAsync (s)	2.55	1.44	0.75	0.40	0.25
cudaMemcpyAsync (K calls)	371	187	93.8	47.0	25.4
cudaStreamSync (s)	1.01	0.53	0.29	0.17	0.12
cudaStreamSync (K calls)	124	62.5	31.3	15.7	8.48

time reduces by 78.4% from batch size 64 to 1024, with average client update time improving by 84.6%. Although this significant reduction in processing time initially affected model accuracy, our extended training (19 rounds at batch size 1024) preserved detection performance and efficiency gains.

Table VI (CUDA API analysis) shows how our combined optimization approach drastically reduces GPU operations and synchronization overhead. With batch size 1024, we achieve dramatic reductions across all operations: kernel launches decrease by 92.3% (from 2.13M to 147K calls), memory transfers reduce by 90.2%, and synchronization overhead drops by 88.2%. These specific reductions in GPU operations, combined with our asynchronous updates and client selection mechanisms, prevent idle time and enable our framework to achieve both efficient communication and accurate detection.

Note that due to the fine-grained nature of Nvidia Nsight Systems profiling visualizations, we make the full resolution profiling data and reports available at our repository [9]. Our profiling reveals systematic performance improvements with increasing batch sizes, showing clear reductions in operation density and execution time across configurations. These profiling results align with the quantitative improvements shown in Tables V and VI.

E. Statistical Validation

We performed a Mann-Whitney U [20] test to assess AUC-ROC differences between our optimized approach and three baselines (CMFL, ACFL, FedL2P) on UNSW-NB15 and ROAD. This non-parametric test does not assume normality, making it suitable for our dataset distributions. The null hypothesis (H_0) is that there is no significant difference in performance, while the alternative (H_1) is that our method outperforms the baselines. At $\alpha = 0.05$, Table VII shows p -values < 0.05 for all comparisons, allowing us to reject H_0 and confirm our method’s significant improvement.

Notably, while our method demonstrates stronger performance on UNSW-NB15, it also achieves significant gains on ROAD, confirming that our combined optimization strategy not only reduces overhead but also yields statistically significant improvements across diverse network anomaly detection scenarios.

TABLE VII: Mann-Whitney U test results showing the statistical significance of the optimized approach compared to the baseline (CMFL), ACFL, and FedL2P across UNSW-NB15 and ROAD datasets.

Comparison	Dataset	U Statistic	p -value
Optimized vs. CMFL	UNSW-NB15	11856.0	4.67e-17
Optimized vs. ACFL	UNSW-NB15	12012.0	3.15e-17
Optimized vs. FedL2P	UNSW-NB15	11934.0	3.98e-16
Optimized vs. CMFL	ROAD	9785.0	1.02e-08
Optimized vs. ACFL	ROAD	9901.0	5.67e-09
Optimized vs. FedL2P	ROAD	9823.0	8.45e-08

VI. DISCUSSION

Our findings confirm that balancing batch size optimization, asynchronous updates, and selective client filtering can reduce communication overhead by over 97% while preserving high detection accuracy across UNSW-NB15 and ROAD. Profiling analysis shows that larger batches (e.g., 1024) significantly reduce GPU operations and memory transfers, although they require extended training to maintain accuracy.

Trade-offs and alternatives: Larger batch sizes boost efficiency but need careful tuning to avoid accuracy drops. Asynchronous updates scale reliably to 100 clients—crucial for real-world anomaly detection. While we optimize client-server interactions, compression (e.g., gradient quantization) remains a complementary option for bandwidth-constrained scenarios.

Limitations and future Work: We plan to explore real-time hyperparameter tuning and resource-aware scheduling, as well as hybrid strategies (e.g., compression + client filtering) to further reduce overhead in edge environments.

VII. CONCLUSION

This paper presents an adaptive FL framework that dynamically balances communication efficiency and detection accuracy for network anomaly detection. By integrating batch size optimization, asynchronous updates, and gradient-aligned client filtering, our approach reduces communication overhead by 97.6% (16.8s vs. 700.0s) while achieving comparable accuracy to centralized methods (95.10%). The framework’s scalability and fault tolerance are validated on both general (UNSW-NB15) and domain-specific (ROAD) datasets, demonstrating broad applicability. Future work will explore real-time

adaptation mechanisms and integration with data compression techniques to further enhance edge deployment capabilities.

ACKNOWLEDGMENTS

This research is supported by the United States Department of Energy (DOE) ASCR grant DE-SC0024352. This research used compute resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported under Contract No. DE-AC02-05CH11231, using NERSC award ASCR-ERCAP0030084. This research is also partially funded by DOE award DE-FE0032089 and supported by DOE's Office of Advanced Scientific Computing Research under award 81585, in collaboration with Pacific Northwest National Laboratory, operated by Battelle under Contract DE-AC05-76RL01830.

REFERENCES

- [1] W. Marfo, D. K. Tosh, and S. V. Moore, "Network anomaly detection using federated learning," in *MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM)*. IEEE, 2022, pp. 484–489.
- [2] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *ICC 2019-2019 IEEE international conference on communications (ICC)*. IEEE, 2019, pp. 1–7.
- [3] W. Marfo, D. K. Tosh, and S. V. Moore, "Enhancing network anomaly detection using graph neural networks," in *2024 22nd Mediterranean Communication and Computer Networking Conference (MedComNet)*. IEEE, 2024, pp. 1–10.
- [4] R. Lee, M. Kim, D. Li, X. Qiu, T. Hospedales, F. Huszár, and N. Lane, "Fed12p: Federated learning to personalize," *Advances in Neural Information Processing Systems*, 2024.
- [5] W. Luping, W. Wei, and L. Bo, "Cmfl: Mitigating communication overhead for federated learning," in *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*. IEEE, 2019, pp. 954–964.
- [6] A. A. Al-Saedi, V. Boeva, and E. Casalicchio, "Reducing communication overhead of federated learning through clustering analysis," in *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2021, pp. 1–7.
- [7] M. Nour and S. Jill, "Unsw-nb15: a comprehensive data set for network intrusion detection systems," in *Proceedings of the 2015 Military Communications and Information Systems Conference (MilCIS)*, 2016, pp. 1–6.
- [8] M. E. Verma, R. A. Bridges, M. D. Iannacone, S. C. Hollifield, P. Moriano, S. C. Hespeler *et al.*, "A comprehensive guide to can ids data and introduction of the road dataset," *PLoS One*, vol. 19, no. 1, p. e0296879, 2024.
- [9] W. Marfo, "Federated Learning Framework and Nsight Profiling Visualizations," 2024, GitHub repository: https://github.com/billmj/UTEP_PNNL_DeepLearning_Optimization/tree/main.
- [10] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.
- [11] G. Yan, H. Wang, X. Yuan, and J. Li, "Criticalfl: A critical learning periods augmented client selection framework for efficient federated learning," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.
- [12] H. Hu, C. Jiang, Y. Zhong, Y. Peng, C. Wu, Y. Zhu, H. Lin, and C. Guo, "dpro: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 623–637, 2022.
- [13] T. Miao, Q. Wu, T. Liu, P. Cui, R. Ren, Z. Li, and G. Xie, "Md-roofline: A training performance analysis model for distributed deep learning," in *2022 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2022, pp. 1–8.
- [14] M. Aach, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann, "Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks," *Journal of Big Data*, vol. 10, no. 1, p. 96, 2023.
- [15] J. Wen, Z. Zhang, Y. Lan, Z. Cui, J. Cai, and W. Zhang, "A survey on federated learning: challenges and applications," *International Journal of Machine Learning and Cybernetics*, vol. 14, no. 2, pp. 513–535, 2023.
- [16] W. Li, X. Liu, Y. Li, Y. Jin, H. Tian, Z. Zhong, G. Liu, Y. Zhang, and K. Chen, "Understanding communication characteristics of distributed training," in *Proceedings of the 8th Asia-Pacific Workshop on Networking*, 2024, pp. 1–8.
- [17] NVIDIA Developer, "Automatic mixed precision for deep learning," <https://developer.nvidia.com/automatic-mixed-precision>, 2023, accessed: 2024-11-09.
- [18] C. Xu, Y. Qu, Y. Xiang, and L. Gao, "Asynchronous federated learning on heterogeneous devices: A survey," *Computer Science Review*, vol. 50, p. 100595, 2023.
- [19] NVIDIA Corporation, "Nvidia nsight systems," <https://developer.nvidia.com/nsight-systems>, 2024. [Online]. Available: <https://developer.nvidia.com/nsight-systems>
- [20] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.