FedFetch: Faster Federated Learning with Adaptive Downstream Prefetching

Qifan Yan*, Andrew Liu*, Shiqi He†, Mathias Lécuyer* and Ivan Beschastnikh*
*Department of Computer Science, University of British Columbia, Canada
Email: ericy676@student.ubc.ca, yul02@student.ubc.ca, mathias.lecuyer@ubc.ca, bestchai@cs.ubc.ca
†Computer Science and Engineering, University of Michigan, USA
Email: shiqihe@umich.edu

***Abstract*—Federated learning (FL) is a machine learning paradigm that facilitates massively distributed model training with end-user data on edge devices directed by a central server. However, the large number of heterogeneous clients in FL deployments leads to a communication bottleneck between the server and the clients. This bottleneck is made worse by straggling clients, any one of which will further slow down training. To tackle these challenges, researchers have proposed techniques like client sampling and update compression. These techniques work well in isolation but combine poorly in the downstream, server-to-client direction. This is because unselected clients have outdated local model states and need to synchronize these states with the server first.**

We introduce FedFetch, a strategy to mitigate the download time overhead caused by combining client sampling and compression techniques. FedFetch achieves this with an efficient prefetch schedule for clients to prefetch model states multiple rounds before a stated training round. We empirically show that adding FedFetch to communication efficient FL techniques reduces end-to-end training time by 1.26× and download time by 4.49× across compression techniques with heterogeneous client settings.

I. Introduction

In Federated learning (FL) a set of distributed clients collaboratively train an ML model with the help of a central parameter server [1], [2]. The clients train with their local data which they never share publicly; instead, clients send their local models or the corresponding gradients to the server for aggregation. This feature enables FL to source data from edge clients without needing to pool data into a single location.

Our work focuses on the cross-device FL setting in which a large number of heterogeneous edge clients train a model (e.g., mobile phones, laptops, IoT devices) [2]. Following previous works, we categorize heterogeneity into system and statistical heterogeneity [3], [4]. System heterogeneity refers to the different network bandwidth capacity, compute capacity, and device availability of clients. Statistical heterogeneity focuses on the dissimilarity of training data characteristics, like the number of samples, presence of labels, quality of examples, etc. In general, the client training data is not independently and identically distributed (non-iid).

Due to a large number of heterogeneous clients in cross-device FL, the transfer of updates between the server and clients consumes a significant amount of time and bandwidth, especially in the downstream, server-to-client, direction. For example, Google's production FL system with over 600 clients selected for training every round, with peak server traffic of around 600 MB/s for downstream and 200 MB/s for upstream updates [5]. Furthermore, straggling clients with low bandwidth or compute capacity inflate the training process. We consider two types of approaches to reduce communication costs in terms of both time and bandwidth: client sampling [1], [3], [4], [6]–[11] and update compression. The latter can be further categorized into masking/sparsification [6], [12]–[15], quantization [16]–[22], low-rank decomposition [23], and sketching [24].

To save bandwidth and training time, cross-device FL deployments rely on a combination of client sampling and compression. However, recent work highlighted that the time and bandwidth improvements that client sampling and compression bring diminish significantly when they are combined in the downstream direction [6], [19]. For instance, He et al. [6] found that a naive combination of client sampling with masking is ineffective in the downstream direction. This is because of client model *staleness*, which is when a client model is not up to date with the server's model due to clients not participating in every training round. Model staleness also comes up with non-masking techniques like quantization and low-rank decomposition. Most work on quantizing downstream model updates either assumes full participation to circumvent client model staleness [20] or full model synchronization before every FL round [16], [18], [23]. Consequently, the need to synchronize models slows down FL deployments. Client heterogeneity further exacerbates this issue, as clients with weaker connectivity will inflate the time to download large downstream updates.

We present **FedFetch**, a general FL method to address the time delay related to synchronizing stale client models caused by client sampling and compression. A standard FL system has a single *Train* phase in every round, which includes client synchronizing a server model, performing local training, and sending results for server aggregation. FedFetch introduces two new phases that come before the *Train* phase: *Prepare* and *Prefetch*.

During the *Prepare* phase, the server presamples the clients that will run R rounds in the future. For each sampled client, the server will create a customized download schedule for the client, depending on knowledge about the client's bandwidth profile. In the *Prefetch* phase, the clients download the latest

global model updates according to their schedules.

> In summary, FedFetch shifts client downstream bandwidth usage from the *Train* phase to the *Prefetch* phase. This reduces end-to-end training time.

Overall, we make the following contributions:

- We characterize the deficiencies of naively combining client selection and compression in downstream communication under heterogeneous cross-device FL conditions. We observe that clients need to synchronize a larger update for each round missed due to not being selected.
- We introduce FedFetch, a general prefetching framework for cross-device FL. FedFetch reduces client local model staleness during client sampling to shorten end-to-end and download time in cross-device FL by 1.26× and 4.49× with an 12% extra bandwidth cost.
- We evaluate FedFetch's compatibility and ease of integration with representative client sampling [1], [6] and compression techniques [6], [12], [16], [18]–[20], [23] in environments with system and statistical heterogeneity. We find that FedFetch consistently decreases the downstream state synchronization time for every method.

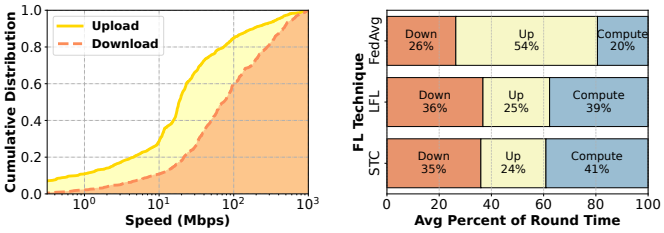
II. Background and Motivation

A. Cross-device FL Characteristics

Cross-device FL is characterized by a high level of system heterogeneity. The differences between clients arise from various sources, such as the type of device, service provider, geographical location etc. In standard FL designs, an FL communication round concludes only when the stragglers (i.e., slowest clients) finish. These stragglers harm performance since client bandwidths vary by orders of magnitude.

To illustrate the effect of system heterogeneity, we plotted the download and upload speeds of edge clients, such as mobile phones and personal computers, in Figure 1a. The figure uses data from Measurement Lab's NDT speed test dataset for N.America in Jan 2024 [25]. Note that roughly 5% of clients have download speeds of less than 4 Mbps, which is about 25 times slower than the median speed of 81.29 Mbps.

Figure 1b shows results from an experiment with the distribution in Figure 1a. It records the breakdown of an FL round in terms of every round's average download, upload, and compute times. We ran these experiments with the FEMNIST dataset using the setup in Section IV-A. This figure shows that communication can become a major bottleneck in cross-device FL settings with FedAvg [1], consuming nearly 80% of the total time in a round. Moreover, upstream communication is the most time-consuming because client upload speeds are typically slower than download speeds (Figure 1a). This communication overhead motivates the need for communication reduction techniques such as masking with STC [12] or quantization with LFL [20]. However, Figure 1b also shows that while these optimizations reduces upstream communication time, they fail to reduce downstream communication. We explore why this is the case in Section II-C.



(a) CDF of edge device download and upload bandwidth distribution for North America in Jan 2024 [25].

(b) Percentage breakdown of an FL round in terms of Download (Down), Upload (Up), and Compute time.

Fig. 1: Cross-device FL Characteristics.

Availability of clients is another key issue in cross-device FL. Clients may spontaneously go offline for various reasons, such as the device running out of power or losing connectivity. In production systems, around 10% of clients may drop out during each round [5], [26]. These device failures slow down model convergence, extend training time, and waste hardware resources. Practical cross-device federated learning systems use an over-commitment (OC) mechanism [5], [6], [27], which selects extra clients to participate in each round to mitigate unavailable and slow clients.

B. Client Sampling

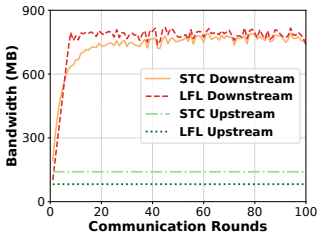
A closely studied approach to reduce communication cost is client sampling, which, in its basic form, uniformly samples a fraction of the total number of clients to participate in each round of FL [1], [6]. Various sampling techniques have been proposed to further improve time/bandwidth-to-accuracy performance [28], ranging from choosing clients with better bandwidth/computation capacity [10] or those more likely to improve convergence [8], [9], [11] or a combination of the two [3], [7]. Sampling reduces downstream (server to client) and upstream (client to server) bandwidth because of fewer participating clients per round.

Unfortunately, client sampling leads to client-side model staleness. Specifically, clients may now have to wait many rounds before being sampled or resampled. For instance, in simple random sampling with uniform probabilities, a client is expected to be resampled every N/K rounds [6]. In effect, client sampling causes the client's local model to become stale relative to the latest server model.

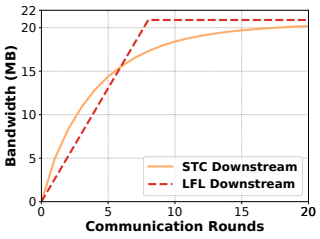
C. Compression

In this paper, we focus on combining client sampling with three representative compression techniques: *masking*, *quantization*, and *low-rank decomposition*.

Masking techniques such as sparsification [6], [12]–[14] and parameter freezing [15] are commonly used for compressing model updates. Specifically, masks are locations of parameters in a model that are transferred along with the parameter value. Usually, the chosen locations correspond to the most useful values in an update. For example, in $TopK$ sparsification,



(a) Download and upload amount per round.



(b) Average download sizes for clients who are resampled after a number of rounds.

Fig. 2: Effect of combining client sampling and compression.

the top $K\%$ parameters, ordered by their absolute value, are transmitted along with their position information [13].

Quantization is another family of techniques to reduce communication costs in FL [16]–[22]. Quantization reduces the encoding precision of the model update to reduce communication, by casting a higher-bit representation of the update into a smaller lower-bit representation.

Low-rank decomposition techniques [23] is the third category of compression techniques we consider in FedFetch. Low-rank methods rely on factorizing an input matrix $M \in \mathbb{R}^{a,b}$ into two low-rank matrices $P \in \mathbb{R}^{a \times r}$ and $Q \in \mathbb{R}^{r \times b}$ where $r \ll \min(a, b)$. The resulting matrices are significantly smaller than the input matrix.

These compression techniques are straightforward to apply in the upstream direction. Yet, the downstream direction is troublesome with client sampling. The reason is that clients selected for training should have the most up-to-date model parameters. In the case of masking, due to the changing server-side mask, an increasing proportion of parameters will have changed with every round passed since the last time a client participated. For clients equipped with a stale local model, synchronization often leads to downloading the entire model at the start of their *Train* phase, instead of a smaller masked update. This nullifies speed and bandwidth improvements in the downstream update synchronization. Downstream compression with quantization, and low-rank decomposition are also difficult when a client's model is out-of-date.

D. Quantifying Staleness

To better understand the staleness problem, we conduct an experiment using simple random client sampling with STC [12], a masking method; and, LFL [20], a quantization technique using default settings from Section IV-A3.

Figure 2a plots communication volume in downstream and upstream directions for FL using STC with a sparsification ratio of 0.2 and LFL with 4 bits. Figure 2a shows that despite the large and consistent savings in the upstream direction, there is little downstream savings past the first 20 rounds. Most clients sampled every round have not participated in training recently. Consequently, these clients must download large updates (possibly the entire model) to catch up.

TABLE I: Summary of notations.

Symbol	Definition
t, T	index and total number of *rounds*
$i, N, \mathcal{N}$	index, total number, set of *clients*
$K, \mathcal{K}_t$	number and set of *clients* sampled at t
$w_t, \hat{w}_t^i$	server and client *model* at the start of t
ν_i	aggregation weight of client i
$\hat{\Delta}_t^i, \Delta$	client i's and aggregated server *update* at t
R	max number of rounds available for prefetch
P_t^i	prefetch schedule for round t
C_{dl}, C_{ul}	downlink and uplink *compressor*
δ_{t_1,t_2}	accumulated server updates from t_1 to t_2
d_t, D_t	true and estimated round duration at t
BW_{dl}^i	downlink bandwidth for i
OC	over-commitment

In Figure 2b, we plot how much content a resampled client needs to download after *not* being selected for a variable number of rounds, for the two methods. The key observation is that with growing local model staleness, a client needs to download an increasingly large update. This update approaches the full model size after 15 rounds for STC and 8 rounds for LFL.

E. Prefetching in FL

Prior research has considered the importance of clients synchronizing the most recent model state [6], [12], [20] or a relatively recent state [19] before receiving compressed downstream updates. With the exception of [19], these methods use simple designs with state synchronization at the start of their training round ($R = 0$) or one prior round ($R = 1$).

DoCoFL [19] assigns clients to start their *Train* phase with a fixed and predetermined time window. For dealing with heterogeneous client bandwidths, they proposed but did not seem to evaluate, two separate time windows: one for strongly-connected clients and another for weakly-connected clients. This approach is unrealistic for cross-device FL environments where round durations vary [5], [27].

In the simple forms of state synchronization above, all clients scheduled for the same round of future training will prefetch a fixed update. However, as we demonstrate in Section II-D, clients with knowledge of the more recent global models can download *smaller* updates. Simple strategies fail to take advantage of this opportunity.

III. FedFetch Design

We introduce prefetching as a new dimension for time-to-accuracy optimization in cross-device FL which we realize in *FedFetch*. We now explain its design.

Figure 3 shows how *FedFetch* introduces two new phases to FL: *Prepare* and *Prefetch*. The server-side *Prepare* phase pre-determines which clients will be selected and when they will start prefetching. The client-side *Prefetch* phase features clients prefetching model states before their scheduled training round. Both phases are controlled by the hyperparameter R, the number of rounds a client will be presampled in advance and the maximum number of rounds available for a client to

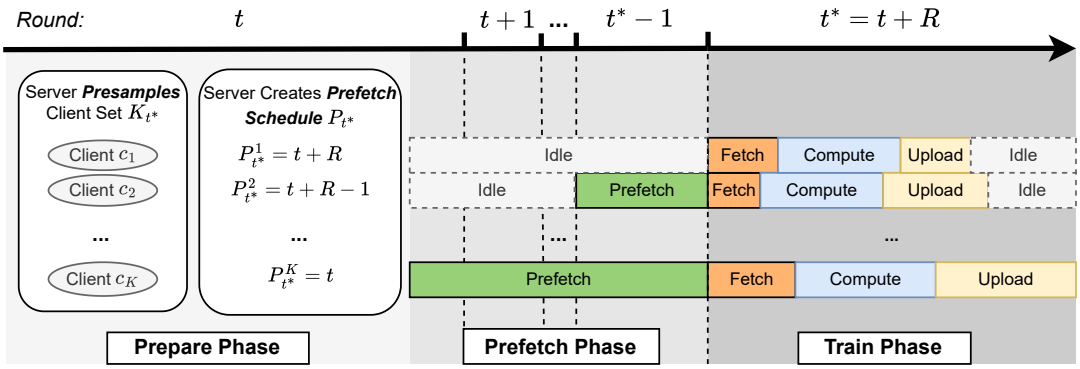


Fig. 3: FedFetch Design. The goal of FedFetch is to minimize the amount of time clients spend on model download during their *Train* phase (*"Fetch"* in orange in the diagram). FedFetch introduces two new phases: *Prepare* and *Prefetch*. During *Prepare*, clients are presampled by the server and provided with a prefetch schedule. During *Prefetch*, each client prefetches model state (*"Prefetch"* in green in the diagram) from the server before their *Train* phase starts.

prefetch. If $R = 0$, then FedFetch is equivalent to the standard synchronous FL algorithm. In FedFetch, we are interested in cases where $R \geq 1$.

A. Prepare phase

The *Prepare* phase has two objectives. First, the server creates a maximum prefetch time budget for clients to prefetch the latest global model before the start of their training round with presampling. Second, the server adjusts the prefetch budget for each client to reduce total downlink update sizes while minimizing download time with prefetch scheduling.

1) Presampling: In presampling, the server samples clients who will participate in a future training round. In general, at round t, the server will generate K_{t^*}, where $t^* = t+R$, the set of K clients scheduled to commence training in round t^*. For simplicity, Algorithm 2 only shows clients that are presampled on round $t = 1$ or later who will start their *Train* phase on round $R+1$ or later. FedFetch conducts no presampling and prefetching for clients with *Train* phase in rounds $1, \ldots, R$.

Critically, presampling is compatible with most client sampling strategies. We need to consider what changes across FL training rounds to understand why this is the case. Typically, the system profile, such as bandwidth and compute capacities, are stable during an FL round, which takes only a few minutes. This means sampling clients a few rounds beforehand based on the client's system profile [10] is sufficient. But, a client's statistical utility is determined through profiling its local data or local model against information on the server [28]. This profiling typically happens when the client last participated in training. The latter approach incurs additional communication overhead associated with synchronizing recent models just for profiling purposes [3]. As clients will take many rounds to be resampled anyway, we argue that predetermining selected clients only a few rounds in advance has little impact on determining a client's statistical utility.

2) Prefetch Scheduling: After the server selects all the clients K_{t^*} to start their *Train* phase in a future round t^*, it will need to construct a prefetch schedule for every client.

To see why FedFetch needs a prefetch scheduler and cannot simply assign a fixed prefetch round for every client, we can consider the following scenario: The system presamples two clients c_1 and c_2. The first client, c_1, is a straggler with low downstream bandwidth and need multiple rounds to finish prefetching all necessary updates. c_2 is a non-straggler with high downstream bandwidth. To optimize the download time in the *Fetch* phase, the server would prefer the client to prefetch as much as possible with a larger time budget in the *Prefetch* phase. Hence, more rounds (up to the maximum of R round) should be allocated to c_1. However, as we increase the prefetch window, we also inevitably increase the overall size of the downstream update (see Section III-A3). This is inefficient for faster clients like c_2 who do not need that much time for prefetching. Therefore, the design goal for the prefetch scheduler is to assign the latest possible prefetch schedule for every client while not negatively impacting the time savings associated with a choice of R.

Algorithm 1 describes FedFetch's prefetch scheduling algorithm. The scheduler follows an iterative process of determining whether for the current prefetch start round t, a client c_i can acquire its *Prefetch* and *Train* phase downstream updates before a time limit T_{limit}. If possible, FedFetch updates the client's prefetch schedule to t. Otherwise, the client keeps its original schedule. In the scenario where all clients can complete within the time limit T_{limit}, the scheduler will advance t^P, the minimum number of rounds required for every client to finish prefetching within the time limit. There are two under-specified elements in this design:

- What should the time limit T_{limit} be?
- How can we determine the time it takes for a client to finish downloading its Prefetch and *Train* phase downstream updates in some future round?

To answer the first question, we define the time limit T_{limit} as the time required for the slowest client to finish acquiring all its downstream updates at the start of the *Fetch* phase. If over-commitment $OC > 1$, then T_{limit} becomes the $1/OC$

Algorithm 1 FedFetch Prefetch Scheduler

1: **procedure** SCHEDULEPREFETCH(Presampled clients set $\mathcal{K}_{t^*}$, current round t^S, *Train* phase round t^*)
2: $t^P \leftarrow t^S$
3: $T_{limit} \leftarrow \infty$
4: **for** $t \leftarrow t^S, \ldots, t^*$ **do**
5: $\mathcal{T}_t \leftarrow \{T_t^i = \text{ESTFETCHTIME}(i) \mid i \in \mathcal{K}_{t^*}\}$
6: ▷ Find clients with *Train* phase fetch time $\leq T_{limit}$
7: $\mathcal{F}_r \leftarrow \{i \mid i \in \mathcal{K}_{t^*}$ and $T_t^i \leq T_{limit}\}$
8: **if** $|\mathcal{F}_r| = |\mathcal{K}_{t+R}|$ **then**
9: $t^P \leftarrow t$
10: $T_{limit} \leftarrow 1/OC$ percentile of $\text{SORTASC}(\mathcal{T}_t)$
11: **end if**
12: $\forall i \in \mathcal{F}_r, P^i \leftarrow t$
13: **end for**
14: **return** $P_{t^*} = \{P^i \mid i \in \mathcal{K}_{t^*}\}$
15: **end procedure**
16:
17: **procedure** ESTFETCHTIME(Client i, prefetch schedule round t, base model round t^P)
18: $D_{avg} \leftarrow \text{EXPWEIGHTEDAVGROUNDDURATION}()$
19: $U \leftarrow \overline{w_t}$ ▷ Accumulated updates to download
20: $B \leftarrow 0$ ▷ Client's prefetch time budget
21: $l \leftarrow t$ ▷ Client's model is in sync with server model w_l once it finishes downloading everything in U
22: **for** $j \leftarrow t, \ldots, t^* - 1$ **do**
23: $B \leftarrow \max(0, B + D_{avg} - U/BW_{dl}^i)$
24: **if** $B > 0$ **then**
25: $U \leftarrow \max(0, \delta_{l,j-1} - B \cdot BW_{dl}^i)$
26: $l \leftarrow j$
27: **else**
28: $U \leftarrow U - D_{avg} \cdot BW_{dl}^i$
29: **end if**
30: **end for**
31: **return** $(U + \delta_{l,t^*-1})/BW_{dl}^i$
32: **end procedure**

percentile of fetch times to account for the fact that only the $K(1/OC)$ client updates are aggregated every round[1].

To answer the second question, we introduce the ESTFETCHTIME function in Algorithm 1 line 17. This function takes in a client c_i's bandwidth profile BW_{dl}^i and some prefetch round t. Similar to [10], the client will immediately provide their bandwidth profile BW_{dl}^i upon being presampled. The function then estimates, for the given client, the amount of time it will take to download all the required downstream updates (see Section III-A3). It does this by simulating FedFetch's *Prefetch* phase (see Section III-B). We choose to estimate the fetch time instead of the total *Train* phase time because, unlike bandwidth, compute speed is harder to profile and would add uncertainty. Moreover, we highlight two differences between time calculations in estimations for prefetch scheduling and in practice during the *Prefetch* phase.

First, we do not know the duration of future rounds during prefetch scheduling. To address this challenge, we estimate the round duration D_t with an exponential weighted moving average of prior round durations with $\alpha = 0.125$. This is the standard approach for estimating round trip times in protocols like TCP [29]. Specifically, the estimated round duration for the current round D_t equals the weighted sum of the true round duration for the previous round d_{t-1} and the past estimated duration D_{t-1}.

$$D_t = \alpha \cdot d_t + (1 - \alpha) \cdot D_{t-1} \tag{1}$$

This better captures trends in round duration throughout the day [5], [27]. In our evaluation we found that the choice of α had little effect on our results, possibly due to the over-commitment mechanism removing extreme stragglers which decreases the variance of round durations.

Second, we do not know the size of various compressed updates exactly for masking techniques because the mask changes unpredictably across rounds. However, the absolute size of the accumulated update after a certain number of rounds is relatively stable (see Section II-D). With this in mind, the server can dynamically profile the size of different rounds of accumulated updates by recording and averaging the sizes of updates sent during the Prefetch and *Train* phases.

These differences between the prefetch scheduling and the actual *Prefetch* phase mean that the scheduler can only approximate the optimal prefetch schedule. Nevertheless, we will show that this approximation leads to significant bandwidth savings in Section IV-B3.

3) Downstream Updates: We now detail the downstream updates prefetched and fetched by clients during the *Prefetch* and *Train* phases. Consider some round t^S where the server presamples clients to start their *Prefetch* phase in round t^P and their *Train* phase in round t^*. It is important to note that individual clients start prefetching at some round $P^i \in \{t^P, \ldots, t^*\}$. The values for P^i and t^P are dynamically decided in Section III-B1. FedFetch requires clients to synchronize a base model w_{t^P} and all the server updates $\Delta_{t^P}, \ldots, \Delta_{t^*-1}$ before the start of t^*. With downstream compression, these server updates will be further compressed with the downstream compressor C_{dl}. For brevity, we represent the sum of these compressed updates from round t_1 to t_2 with $t_1 \leq t_2$ with Equation (2). If $t_1 > t_2$, then $\delta_{t_1,t_2} = 0$.

$$\delta_{t_1,t_2} = \sum_{j=t_1}^{t_2} C_{dl}(\Delta_j) \tag{2}$$

We proceed to summarize the full downstream update below.

$$w_{t^*} = w_{t^P} + \delta_{t^P,t^*-1} \tag{3}$$

In terms of the sizes of each term in Equation (3), w_{t^P} has the same size as a full model. Each server update $C_{dl}(\Delta_j)$ has the minimum possible update size. However, transferring the sum of updates could lead to a smaller size than transferring

[1]We considered adding a factor β, where $1 + \beta < OC$ so that T_{limit} is the $(1 + \beta)/OC$ percentile fetch time. We evaluated different values of β and found no difference in results as long as $1 + \beta$ is not close to OC.

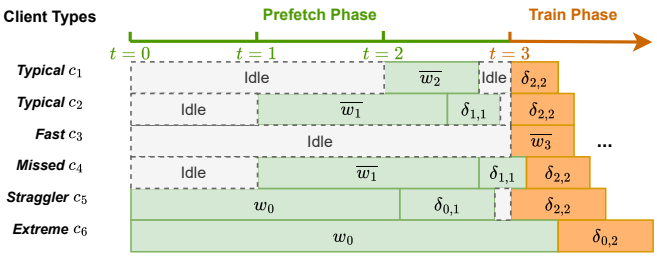


Fig. 4: An example of a prefetch process for six clients and $R = 3$. The blocks represent what each client is currently prefetching (in green) or fetching (in orange) from the server.

each update separately (see Section II-D). For example, $\delta_{1,2} = C_{dl}(\Delta_1) + C_{dl}(\Delta_2)$ could be smaller than transferring $\delta_{1,1} = C_{dl}(\Delta_1)$ and $\delta_{2,2} = C_{dl}(\Delta_2)$ separately if the downstream compressor C_{dl} is a masking method.

Similarly, we can also combine the base model with all the updates until the client-specific prefetch schedule P^i in a single $\overline{w_{P^i}}$ instead of transmitting w_{t^P} and the updates δ_{t_p,P^i-1} . These size reductions resulting from combining updates are what make the prefetch stage efficient. As a result, the prefetched and fetched update for a client is the following.

$$w_{t^*} = \overline{w_{P^i}} + \delta_{P^i,t^*-1} \tag{4}$$

Although written as a single term, the second term in Equation (4) may be separated into multiple updates in both the prefetch scheduling and actual prefetch process to take full advantage of the prefetch budget.

The first term, $\overline{w_{P^i}}$, is unavoidable. So, the additional δ_{P^i,t^*-1} term is the main culprit for bandwidth overhead. Moreover, since the client model at the start of the *Train* phase using FedFetch is equivalent to the client model without prefetch, the convergence behavior of FedFetch should be similar to the non-prefetch case. Therefore, the benefits brought by FedFetch will increase with stronger downstream compressors C_{dl}.

B. Prefetch Phase

Once client i from the presampled client set K_{t^*} acquires its prefetch schedule P^i from the *Prepare* phase, the client can start its *Prefetch* phase. Client c_i will attempt to prefetch starting at round P^i until the *Train* phase at round t^*.

1) Prefetch Process: FedFetch employs a greedy prefetch process where the presampled client c always tries to download the most recently available update starting on their scheduled prefetch start round P^c. To minimize the *Train* phase fetch size and time, FedFetch transmits the largest downstream updates at the start and the smallest updates (δ of a single round) at the end of the *Prefetch* phase.

Figure 4 illustrates an example FedFetch run with 6 clients of different categories and $R = 3$. On round 0, FedFetch presamples all 6 clients who will start training on round 3. FedFetch also provides a prefetch schedule for each client which is indicated by the round a client c_i starts prefetching.

Algorithm 2 FedFetch

procedure SERVER
  for $t \leftarrow 1, \ldots, T$ **do**
    ▷ Server: *Prepare* phase
    Presample $\mathcal{K}_{t+R}$ from $\mathcal{N}$
    $P_{t+R} \leftarrow$ SCHEDULEPREFETCH()
    ▷ Clients in $\mathcal{K}_{t+R}$ start *Prefetch* phase
    ▷ Clients in $\mathcal{K}_t$ start *Train* phase
    ▷ Server: Aggregation
    $\Delta_t \leftarrow \sum_{i \in \mathcal{K}_t} \nu_i \hat{\Delta}_t^i$
    $w_{t+1} \leftarrow w_t + C_{dl}(\Delta_t)$
  end for
end procedure

procedure CLIENT i
  ▷ Server notification happens on scheduled round P^i. Clients receive the current round t and train round t^*
  NOTIFIEDBYSERVER()
  ▷ Client: *Prefetch* phase ($t < t^*$)
  ▷ Sync base model
  $\hat{w}_{\ell_i}^i \leftarrow$ DOWNLOAD($\overline{w_{P^i}}$)
  $\ell_i \leftarrow P^i - 1$
  ▷ Sync server updates
  while $t \leftarrow$ QUERYSERVERROUND(), $t < t^*$ **do**
    $\hat{w}_t^i \leftarrow \hat{w}_{\ell_i}^i +$ DOWNLOAD($\delta_{\ell_i,t-1}$)
    $\ell_i \leftarrow t - 1$
  end while
  ▷ Client: *Train* phase ($t = t^*$)
  $\hat{w}_t^i \leftarrow \hat{w}_{\ell_i}^i +$ DOWNLOAD($\delta_{\ell_i,t-1}$)
  $\hat{\Delta}_t^i \leftarrow C_{dl}($LOCALTRAINING$(\hat{w}_t^i) - \hat{w}_t^i)$
  UPLOAD($\hat{\Delta}_t^i$)
end procedure

A typical client like c_1 will start prefetching on round 2 and completes before the *Train* phase in round 3. TRhe update acquired by the client in its *Train* phase is the smallest possible update $\delta_{2,2}$. In contrast, another client, c_2, needs two rounds to prefetch all its updates. Therefore, c_2 prefetches in round 1 and continues to prefetch $\delta_{1,1}$ as its local model is from the start of round 1 and the new update $\delta_{1,1}$ is now available.

For a client with substantial bandwidth, c_3, the server assigns it to start prefetching on round 3 because c_3 can finish downloading the full model $\overline{w_3}$ in its *Train* phase. However, clients like c_4 may not prefetch all their *Prefetch* phase updates before round 3. This forces client c_4 to acquire both pending prefetch and *Train* phase updates, increasing fetch time.

Stragglers like c_5 require the maximum prefetch budget of $R = 3$ rounds to prefetch its updates starting from round 0. Clients with low bandwidth, like c_6, may still fail to finish prefetching within 3 rounds. But, prefetching benefits such clients by reducing the volume they download during the *Train* phase compared to cases with no prefetching.

2) Impact of Client Unavailability: In real deployments of cross-device FL, clients are online at different times. Therefore, client sampling process only samples from the set

of clients that are currently online. So, a method that selects clients earlier than normal may suffer performance drops due to clients going offline. FedFetch addresses this problem with a simple replacement strategy. These replacement clients will immediately start prefetching if there is still a prefetch budget. We observe empirically that this mostly negates the performance issues caused by unavailability (Section IV-B6).

IV. Experimental Evaluation

We evaluate FedFetch along several dimensions and answer the following four questions:

- Q1: How well does FedFetch integrate with existing techniques such as client sampling, masking, quantization, and low-rank decomposition?
- Q2: What impact does FedFetch have on training time, bandwidth usage, and model accuracy?
- Q3: How does FedFetch's hyperparameter impact its performance?
- Q4: How does FedFetch handle settings involving overcommitment and client unavailability?

A. Experimental Setup

1) Environment and Datasets: We run all experiments on the FedScale [27] platform. We use client bandwidth data points from the Measurement Lab's NDT data set [25] (Figure 1a). We rely on FedScale to organize client device hardware data from AI Benchmark [30] and online/offline behaviour traces from FLASH [31] (Section IV-B6).

We use benchmarking datasets provided by FedScale: FEMNIST [32], Google Speech [33], and OpenImage [34]. The FEMNIST and OpenImage datasets are used to train image classification models. The former consists of 640K colored images and 2,800 clients and the latter consists of 1.3M colored images and 10,625 clients. The Google Speech dataset is used to train a speech recognition model and consists of 105K speech samples and 2,066 clients. We train different models — FEMNIST uses ShuffleNet [35], Google Speech uses ResNet-34 [36], and OpenImage uses MobileNet [37]. Similar to recent work [27], we set the target accuracy to be the highest achievable accuracy by tested methods. The number of client results collected per round is $K = 30$ for FEMNIST and Google Speech, and $K = 100$ for OpenImage. We set over-commitment $OC = 1.3$ [5]; the actual number of clients sampled per round will increase by $1.3\times$.

2) Compression strategies: FedFetch works with most existing compression methods applied to FedAvg with simple random client sampling [1]. To answer Q1, we compare the method with FedAvg versus using FedFetch and the method together. We use STC[2] [12], GlueFL[3] [6] as the representatives for masking, QSGD [16], LFL [20], EDEN [18] for quantization, and PowerSGD [23] for matrix decomposition. We also evaluate FedFetch's compatibility with DoCoFL [19], a full model quantization method by replacing DoCoFL's transmission of models from the top of a compressed model queue with FedFetch's prefetch method, which sends more recent compressed server models.

3) Parameters: We use parameter values from previous works which reliably reach the target test accuracy. Clients perform 10 local updates per round with PyTorch's SGD optimizer with a momentum factor of 0.9 on batches of size 20. The initial learning rate is set to 0.01 and decreases by 0.98 every 10 rounds. For STC and GlueFL, we set a compression ratio of $q = 20\%$ for ShuffleNet, $q = 35\%$ for MobileNet, and $q = 30\%$ for ResNet-34. For quantization, we use a bit budget of 4 bits for QSGD, LFL, and EDEN. For PowerSGD, we set the rank to 16 for ShuffleNet and 24 for MobileNet and ResNet-34. Finally, we set DoCoFL's bit-budget, anchor deployment rate, and anchor queue size to 2, 10 and 3 respectively.

4) Metrics: For Q2 and Q3, we measure the end-to-end time, download-specific training time, and bandwidth usage. Since the straggler client will determine the total time duration of an FL round, we use the sum of the stragglers' download times to represent the total fetch time (FT) and similarly for compute and upload time. This way, the total FL training time (TT) is equal to the sum of the fetch (FT), compute, and upload times. For bandwidth usage, the total transmission volume (TV) represents the transmission volume used by every client, including overcommitted clients. The fetch volume (FV) represents the downstream bandwidth associated with the fetch operation for all clients participating in the *Train* phase. For Q4, we report the time and bandwidth when the average of the previous 5 testing rounds reaches the target accuracy listed in Table II. This is similar to prior work [3], [6].

B. Results

1) Main Performance Results (Q1–Q2): Using settings from the previous section, we experiment with each compression method with and without FedFetch. Therefore, FedFetch's baselines are the compression methods without FedFetch. After reaching the target accuracy in Table II, we record the time-to-accuracy and bandwidth-to-accuracy performance.

Table II shows that FedFetch consistently saves fetch (download) and end-to-end training time at little extra bandwidth overhead for all tested compression algorithms and across different models and datasets. On average, we see a $4.49\times$ reduction in fetch time. This translates into a mean end-to-end training time speedup of $1.26\times$.

The time savings are significant for masking techniques like STC, where FedFetch speeds up training time by $1.49\times$. FedFetch is less effective at improving GlueFL ($1.09\times$ end-to-end speedup on average). However, this is expected because GlueFL already employs sticky sampling, a client sampling technique that favors clients who recently participated in training. For quantization methods like QSGD, LFL, and EDEN, FedFetch achieves an average end-to-end time speedup of $1.29\times$. We further show that FedFetch reduces PowerSGD's end-to-end time by $1.28\times$. The total time savings ($1.07\times$ speedup) are less noticeable for full model quantization

[2]STC is a hybrid method featuring both masking and quantization which are orthogonal techniques, we exclusively evaluate STC's masking strategy

[3]We replace simple random sampling with the sticky client sampling

TABLE II: Main performance results for three model-dataset pairs. Metrics **FT** and **TT** represent **F**etch **T**ime and **T**otal Training **T**ime, in hours. **FV** and **TV** represent **F**etch **V**olume and **T**otal Transmission **V**olume, in $\times 10^2$ GB. Results are recorded when the target accuracy (*Trg*) is reached[a]. We run each setting at least 3 times and report the mean.

[a] PowerSGD achieves a maximum test accuracy of 61% on the OpenImage dataset, we underline these results

		FEMNIST Trg 75%				*Google Speech Trg 61%*				*OpenImage Trg 68%*			
		FT	**TT**	**FV**	**TV**	**FT**	**TT**	**FV**	**TV**	**FT**	**TT**	**FV**	**TV**
Baseline	FedAvg	0.64	2.56	1.56	2.76	5.54	21.7	12.2	21.6	1.14	5.4	10.8	19.1
Masking	STC	0.85	2.46	2.58	**3.07**	9.96	25.0	13.7	**18.0**	0.97	4.47	11.5	**15.0**
	FedFetch + STC	**0.21**	**1.67**	**1.31**	3.14	**2.71**	**13.4**	**10.4**	21.6	**0.44**	**4.00**	**8.85**	18.1
	GlueFL	0.30	2.11	2.58	**3.32**	2.28	16.7	12.9	**18.9**	0.38	2.55	14.0	**20.7**
	FedFetch + GlueFL	**0.18**	**1.84**	**1.49**	4.01	**1.26**	**14.3**	**11.2**	25.0	**0.23**	**2.36**	**8.84**	24.6
Quantization	QSGD	0.46	1.35	1.56	**1.73**	3.41	8.62	10.49	**11.6**	0.72	3.68	10.8	**12.0**
	FedFetch + QSGD	**0.06**	**0.90**	**0.58**	1.83	**0.84**	**6.26**	**4.87**	14.3	**0.13**	**3.07**	**4.45**	13.7
	LFL	0.43	1.27	1.45	**1.60**	3.03	10.9	10.78	**11.9**	0.67	3.55	10.4	**11.6**
	FedFetch + LFL	**0.06**	**0.90**	**0.58**	1.83	**0.58**	**8.88**	**4.7**	14.7	**0.12**	**3.09**	**4.34**	13.7
	EDEN	0.42	1.26	1.43	**1.60**	4.02	12.7	11.7	**13.7**	0.74	3.54	10.2	**11.7**
	FedFetch + EDEN	**0.06**	**0.89**	**0.60**	1.75	**1.12**	**10.1**	**6.92**	16.5	**0.16**	**3.12**	**5.48**	14.1
Low-rank	POWERSGD	1.49	4.23	5.25	**5.62**	6.87	26.7	28.1	**29.8**	0.58	2.81	8.01	**9.24**
	FedFetch + POWERSGD	**0.14**	**3.02**	**1.47**	6.54	**0.67**	**21.0**	**6.89**	35.5	**0.11**	**2.43**	**4.28**	11.0
Quantization (Full model)	DoCoFL	0.04	0.96	**0.13**	0.72	**0.48**	8.79	**1.45**	6.97	0.12	3.26	**1.15**	5.52
	FedFetch + DoCoFL	0.04	**0.93**	0.28	**0.56**	0.50	**8.47**	2.95	**6.02**	0.12	**2.87**	2.50	**5.38**

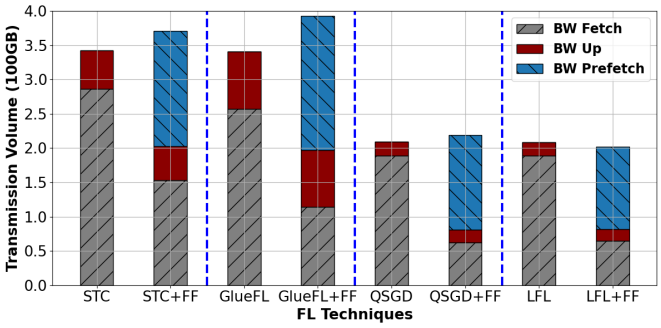
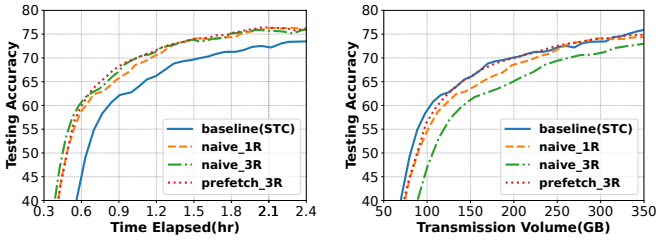
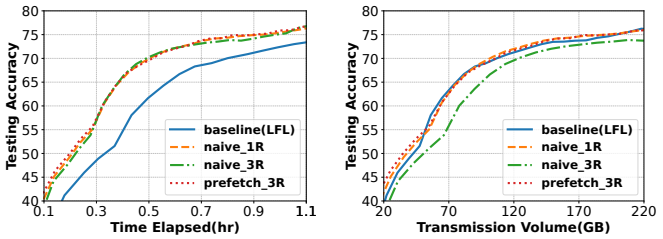


Fig. 5: Bandwidth usage of select FL techniques with and without FedFetch. Techniques ending with "+FF" apply FedFetch. Each bar is divided into Fetch, Up(load), and Prefetch.



(a) Comparison with naive prefetching on time-to-accuracy performance for STC.

(b) Comparison with naive prefetching on bandwidth-to-accuracy performance for STC.



(c) Comparison with naive prefetching on time-to-accuracy performance for LFL.

(d) Comparison with naive prefetching on bandwidth-to-accuracy performance for LFL.

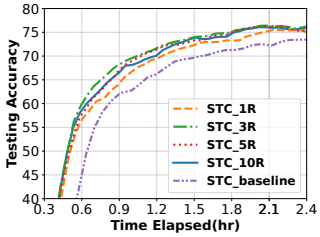
Fig. 6: Comparison with naive forms of prefetching for masking with STC and quantization with LFL.

methods like DoCoFL. This is expected because DoCoFL already leverages a simple form of prefetching which we optimize further with FedFetch. Specifically, through prefetch scheduling, FedFetch allows DoCoFL to safely use more recently compressed base models without impacting round duration whereas DoCoFL uses a more stale model from the top of its compressed model queue.

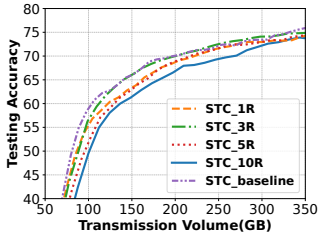
Crucially, FedFetch achieves the above speedups with a 13% average increase in bandwidth. FedFetch fulfills its primary goal of speeding up downstream compression communication in cross-device FL for a broad range of compression techniques in a bandwidth-efficient manner.

2) Bandwidth Breakdown (Q2): We examine the effect of FedFetch on bandwidth usage in Figure 5, which plots the total volume breakdown between prefetch, fetch, and upstream for STC, GlueFL, LFL, QSGD methods with and without FedFetch after they reach the target accuracy for FEMNIST. We see a shift in the distribution from fetch to prefetch when FedFetch is used. With FedFetch, the fetch bandwidth is, on average, 59% lower. This suggests that FedFetch achieves the goal of shifting downstream bandwidth from fetch to prefetch.

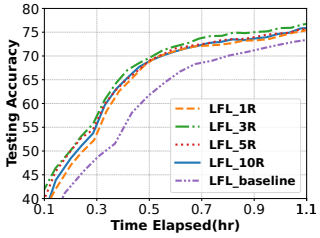
3) Comparison with Naive Forms of Prefetching (Q2): We compare FedFetch with two simple prefetching methods on FEMNIST. In the first, every presampled client shares a fixed prefetch schedule of 1 round, while the second method uses 3 rounds. We plot the time and total bandwidth versus



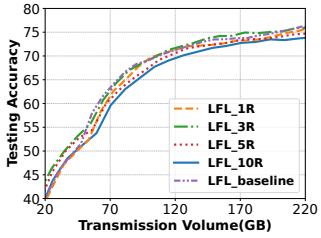
(a) Effect of R on time-to-accuracy performance for STC.



(b) Effect of R on bandwidth-to-accuracy performance for STC.



(c) Effect of R on time-to-accuracy performance for LFL.

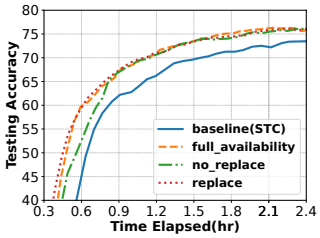


(d) Effect of R on bandwidth-to-accuracy performance for LFL.

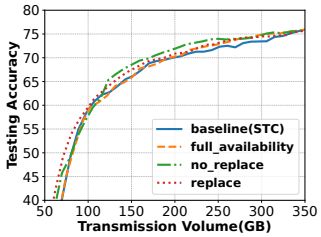
Fig. 7: Sensitivity analysis for the max number of prefetch rounds R for masking with STC and quantization with LFL.

TABLE III: Impact of overcommitment OC on the FEMNIST dataset (Trg 75%). See Table II for column definitions.

	STC				STC + FedFetch			
OC	**FT**	**TT**	**FV**	**TV**	**FT**	**TT**	**FV**	**TV**
1.0	10.97	51.33	1.97	**2.45**	**5.5**	**48.39**	**1.90**	2.63
1.1	3.03	8.12	1.77	**2.37**	**1.16**	**7.34**	**1.50**	2.68
1.2	1.54	3.5	2.09	**2.53**	**0.59**	**2.75**	**1.44**	3.09
1.3	0.85	2.46	2.58	**3.07**	**0.21**	**1.67**	**1.31**	3.14
1.4	0.57	2.04	2.78	3.28	**0.11**	**1.37**	**1.23**	**3.22**



(a) Impact of client availability on time-to-accuracy performance.



(b) Impact of client availability on bandwidth-to-accuracy performance.

Fig. 8: Impact of Client Availability for FedFetch + STC.

accuracy performance for the base compression method (STC or LFL), FedFetch with $R = 3$, and the two simple prefetching methods in Figure 6. These plots show that FedFetch brings as much time-to-accuracy performance as using a fixed 3-round prefetch schedule. But, FedFetch uses the same or less amount of extra bandwidth as the fixed 1-round prefetching, despite a prefetch budget of 3 rounds. FedFetch achieves the maximum speedup for a given R without extra bandwidth usage.

4) Sensitivity Analysis of R (Q3): We evaluate the impact of the max number of prefetch rounds, R, on performance. We apply FedFetch to STC and LFL on FEMNIST. We vary R, choosing values of 1, 3, 5, and 10. For any choice of R, adding FedFetch to a compression method shifts the corresponding time-to-accuracy curve to the left, as seen in Figure 7a. This indicates that adding FedFetch can consistently reduce training time. Among these, the curve associated with $R = 1$ has the worst time-to-accuracy performance. This is expected because the slowest selected clients may require more than 1 round to complete prefetching. For $R > 1$, the difference in time-to-accuracy and bandwidth-to-accuracy performance is smaller because stragglers needing multiple rounds of prefetching participate less frequently in training. Nevertheless, Figure 7b and Figure 7d show that the bandwidth consumption of FedFetch only increases slightly for larger values of R. This indicates that FedFetch can minimize the extra bandwidth overhead associated with prefetching.

5) Impact of Over-commitment (Q4): In Table III, as over-commitment OC increases, more clients with weaker connectivity are removed. This leads to faster training time at the cost of higher data transmission volume, with this effect the most significant for lower values of OC. FedFetch consistently decreases FT, TT, and FV across OC values. Note that the fetch volume decreases with OC values for STC+FedFetch instead of increasing as in STC. This is because clients with average bandwidth are more numerous and are likely to become stragglers under higher OC settings. FedFetch's adaptively encourages prefetch schedules with more prefetch rounds for these clients. This shifts bandwidth consumption from fetch to prefetch.

6) Impact of Client Availability (Q4): Figure 8 compares two versions of FedFetch applied to STC. One version is without modifications (*no_replace*), and the second performs client replacement (*replace*) by including a random client from the set of all clients currently online. We also plot the baseline STC technique under the same availability and FedFetch STC when clients have perfect availability. With clients going offline, the time-to-accuracy performance of FedFetch diminishes as compared to full availability. However, adding the simple replacement mechanism allows FedFetch to mitigate most of the effect of offline clients. This is indicated by the *replace* line being further to the left than the *no_replace* line in Figure 8a.

V. Conclusion

We introduced FedFetch, a strategy to address the communication bottleneck in federated learning (FL) caused by the combination of client sampling and update compression techniques. FedFetch efficiently schedules model state prefetching for clients, significantly reducing download and overall training times. Our evaluation demonstrates that incorporating FedFetch into communication-efficient FL methods can decrease end-to-end training time by 1.26× and download time by 4.49× across a variety of compression techniques.

REFERENCES

[1] 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.

[2] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings *et al.*, "Advances and open problems in federated learning," *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[3] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Efficient federated learning via guided participant selection," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[4] B. Luo, W. Xiao, S. Wang, J. Huang, and L. Tassiulas, "Tackling system and statistical heterogeneity for federated learning with adaptive client sampling," in *IEEE INFOCOM 2022-IEEE conference on computer communications*. IEEE, 2022, pp. 1739–1748.

[5] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan *et al.*, "Towards federated learning at scale: System design," *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[6] S. He, Q. Yan, F. Wu, L. Wang, M. Lécuyer, and I. Beschastnikh, "Gluefl: Reconciling client sampling and model masking for bandwidth efficient federated learning," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 695–707, 2023.

[7] C. Li, X. Zeng, M. Zhang, and Z. Cao, "Pyramidfl: a fine-grained client selection framework for efficient federated learning," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, ser. MobiCom '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 158–171.

[8] M. Chen, N. Shlezinger, H. V. Poor, Y. C. Eldar, and S. Cui, "Communication-efficient federated learning," *Proceedings of the National Academy of Sciences*, vol. 118, no. 17, p. e2024789118, 2021.

[9] W. Chen, S. Horvath, and P. Richtarik, "Optimal client sampling for federated learning," *Transactions on Machine Learning Research*, 2022.

[10] 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.

[11] F. Wu, S. Guo, Z. Qu, S. He, Z. Liu, and J. Gao, "Anchor sampling for federated learning with partial client participation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 37 379–37 416.

[12] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Robust and communication-efficient federated learning from non-iid data," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 9, pp. 3400–3413, 2019.

[13] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified sgd with memory," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.

[14] F. Wu, X. Wang, Y. Wang, T. Liu, L. Su, and J. Gao, "Fiarse: Model-heterogeneous federated learning via importance-aware submodel extraction," *Advances in Neural Information Processing Systems*, 2024.

[15] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, and G. Zhang, "Communication-efficient federated learning with adaptive parameter freezing," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 1–11.

[16] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[17] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Fifteenth annual conference of the international speech communication association*, 2014.

[18] S. Vargaftik, R. B. Basat, A. Portnoy, G. Mendelson, Y. B. Itzhak, and M. Mitzenmacher, "EDEN: Communication-efficient and robust distributed mean estimation for federated learning," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 21 984–22 014.

[19] R. Dorfman, S. Vargaftik, Y. Ben-Itzhak, and K. Y. Levy, "DoCoFL: Downlink compression for cross-device federated learning," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 8356–8388.

[20] M. M. Amiri, D. Gunduz, S. R. Kulkarni, and H. V. Poor, "Federated learning with quantized global model updates," 2020.

[21] S. Zheng, C. Shen, and X. Chen, "Design and analysis of uplink and downlink communications for federated learning," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 7, pp. 2150–2167, 2020.

[22] F. Wu, S. He, S. Guo, Z. Qu, H. Wang, W. Zhuang, and J. Zhang, "Sign bit is enough: A learning synchronization framework for multi-hop all-reduce with ultimate compression," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 193–198.

[23] T. Vogels, S. P. Karimireddy, and M. Jaggi, "Powersgd: Practical low-rank gradient compression for distributed optimization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[24] D. Rothchild, A. Panda, E. Ullah, N. Ivkin, I. Stoica, V. Braverman, J. Gonzalez, and R. Arora, "Fetchsgd: Communication-efficient federated learning with sketching," in *International Conference on Machine Learning*. PMLR, 2020, pp. 8253–8265.

[25] Measurement Lab, "The M-Lab NDT data set," (2024-01-01 – 2024-01-31).

[26] C. Yang, M. Xu, Q. Wang, Z. Chen, K. Huang, Y. Ma, K. Bian, G. Huang, Y. Liu, X. Jin, and X. Liu, "Flash: Heterogeneity-aware federated learning at scale," *IEEE Transactions on Mobile Computing*, vol. 23, no. 1, pp. 483–500, 2024.

[27] F. Lai, Y. Dai, S. S. Singapuram, J. Liu, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "FedScale: Benchmarking model and system performance of federated learning at scale," in *International Conference on Machine Learning (ICML)*, 2022.

[28] L. Fu, H. Zhang, G. Gao, M. Zhang, and X. Liu, "Client selection in federated learning: Principles, challenges, and opportunities," *IEEE Internet of Things Journal*, 2023.

[29] M. Sargent, J. Chu, D. V. Paxson, and M. Allman, "Computing TCP's Retransmission Timer," RFC 6298, Jun. 2011.

[30] A. Ignatov, "Ai benchmark performance ranking," https://ai-benchmark.com/ranking.html, 2023.

[31] C. Yang, Q. Wang, M. Xu, Z. Chen, K. Bian, Y. Liu, and X. Liu, "Characterizing impacts of heterogeneity in federated learning upon large-scale smartphone data," in *Proceedings of the Web Conference 2021*, 2021, pp. 935–946.

[32] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "Leaf: A benchmark for federated settings," 2019.

[33] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[34] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, "The Open Images Dataset V4: Unified Image Classification, Object Detection, and Visual Relationship Detection at Scale," *IJCV*, 2020.

[35] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[37] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.