How Asynchronous can Federated Learning Be?

Ningxin Su, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—As a practical paradigm designed to involve large numbers of edge devices in distributed training of deep learning models, federated learning has witnessed a significant amount of research attention in the recent years. Yet, most existing mechanisms on federated learning assumed either fully synchronous or asynchronous communication strategies between clients and the federated learning server. Existing designs that were partially asynchronous in their communication were simple heuristics, and were evaluated using the number of communication rounds or updates required for convergence, rather than the wall-clock time in practice.**

In this paper, we seek to explore the entire design space between fully synchronous and asynchronous mechanisms of communication. Based on insights from our exploration, we propose PORT, a new partially asynchronous mechanism designed to allow fast clients to aggregate asynchronously, yet without waiting excessively for the slower ones. In addition, PORT is designed to adjust the aggregation weights based on both the staleness and divergence of model updates, with provable convergence guarantees. We have implemented PORT and its leading competitors in PLATO, an open-source scalable federated learning research framework designed from the ground up to emulate real-world scenarios. With respect to the wall-clock time it takes for converging to the target accuracy, PORT outperformed its closest competitor, *FedBuff*, by up to 40% in our experiments.

I. INTRODUCTION

As one of the emerging distributed learning paradigms, *federated learning* (FL) [1] has witnessed a tremendous amount of research attention in the recent research literature. Compared to conventional distributed machine learning, there are several unique features in federated learning that motivated the recent research attention. *First,* as edge devices (clients) collaboratively train a global model with their locally generated data, it is generally assumed that data is not independent and identically distributed (i.i.d.). *Second,* as there may exist a large number of clients in a federated learning session, only a small subset of clients are selected in each *communication round* between the clients and the FL server. On the other hand, federated learning inherits the same research objective as conventional distributed machine learning: both aim to minimize the amount of time to complete training, measured by the wall-clock time needed to converge to a target accuracy.

It is worth noting that most existing research literature on federated learning, starting from McMahan *et al.*'s seminal work on the federated averaging algorithm [1], assumed that each communication round between the client and the server is fully *synchronous*, in that the server would wait for all the selected clients to complete local training and report their model updates before aggregation takes place. Such a design has been utilized by most existing works as it is simple yet effective, and is similar to the so-called *Bulk Synchronous Parallel* mechanism in the realm of distributed machine learning within the same cluster. However, it is intuitive to observe that the performance — in terms of the wall-clock time to converge to a target accuracy — of such a synchronous communication mechanism may degrade if some clients progresses with a much slower pace than the others in their local training, since the server needs to wait for these *stragglers*, a commonly used term in conventional distributed machine learning.

In this context, it is natural to introduce an *asynchronous* communication paradigm, where the server does not need to wait for all its selected clients to report their model updates, and chooses to proceed with the aggregation process as soon as the model update from one client arrives. Such a design has been first proposed by Xie *et al.* [2], called *FedAsync*, where the server immediately updates the global model whenever it receives a client's model update.

However, in the case where the distribution of client speeds is heavy-tailed, one may easily conceive pathological scenarios where fast clients are rapidly replaced by other fast clients with updated global models, while much slower clients make very little progress based on global models that are becoming out of date. In conventional distributed machine learning with parameter servers, such pathological scenarios were avoided by the introduction of *bounded staleness* in the *stale synchronous parallel* mechanism (SSP) [3]. It is straightforward to design a similar mechanism in federated learning that is staleness-aware, where the server waits for much slower clients that are beyond a pre-determined staleness bound.

We argue, however, that the entire design space between fully synchronous (corresponding to BSP in distributed machine learning) and asynchronous with staleness bounds (corresponding to SSP) has not yet been rigorously explored and experimental evaluated in the existing literature. There exists a number of factors at play in such a design space, reflected in the following basic questions. First, what is the *minimum percentage of clients* the server should wait for before it commences its aggregation process? The more clients being waited for, the more *synchronous* the communication mechanism becomes. Second, what should the *staleness bound* be? The more relaxed the staleness bound is, the more *asynchronous* the design is. Last but not the least, when a server aggregates the model updates it receives so far, which are inherently based on different global models, how should the server allocate the aggregation weights to each client? In extreme cases, the

978-1-6654-6824-4/22/$31.00 The email addresses of the authors are ningxin.su@mail.utoronto.ca and bli@ece.toronto.edu.

server may choose to assign an aggregation weight of zero to extremely slow clients, effectively reducing the relevance of their local training.

Unfortunately, heuristics designed in existing works in the literature reflected point solutions in such a design space, and in several cases failed to motivate their design choices. In particular, most existing works used either the number of gradients, updates, or communication rounds before convergence as their performance metric, which failed to reflect the actual wall-clock time it takes to converge to a target accuracy. This is because each update or communication round may take a substantially different amount of time. As such, it is not clear what the best possible trade-off between conflicting design decisions is, and what the *sweet spot* is in the entire spectrum between synchronous and asynchronous mechanisms.

In this paper, we seek to experimentally explore such a design space in a real-world scalable FL research framework, called PLATO, designed from the ground up to accurate measure the wall-clock time involving all the factors at play. With insights obtained from our experimental evaluations, we propose PORT[1], a new mechanism that navigates the tradeoffs involved in the design space with best practices.

Highlights of our original contributions in this paper are three-fold:

First, in PORT, the server incorporates a *push-pull* mechanism: it allows fast clients to aggressively report their model updates, and aggregates them as soon as a minimum percentage of clients arrive. Yet, it does not need to wait for *stale* clients after the staleness bound has been reached; instead, it aggressively *pull* these stale clients with an *urgent* notification. Clients that receive such urgent notifications are required to report immediately after completing the current training epoch.

Second, inspired by existing adaptive aggregation mechanisms, we propose to assign lower aggregation weights to clients that are more *stale* and more *divergent* in their model updates. The intuition behind this design is that stale clients are based on earlier versions of the global model, and their model updates are therefore of lower quality and less relevant. Theoretically, we show that our mechanism enjoys a convergence guarantee.

Finally, the design of PORT is based on an array of experimental evaluations using our real-world FL framework, with an emphasis on reproducible results when comparing with the state-of-the-art, using the wall-clock time as our performance metric rather than the number of communication rounds. As asynchronous paradigms are inherently designed to minimize the wall-clock time, this is the only suitable way of evaluating competing designs. With a variety of datasets and models, we show that PORT is able to outperform all of its competitors in the literature, and by a margin of up to 40% over its closest state-of-the-art competitor in the literature.

The remainder of this paper is organized as follows. In Section II, we first present necessary preliminaries and highlights of related work. In Section III, we introduce an initial array of experimental evaluations, showing how the design space is to be explored with a range of tuning knobs to be adjusted. In Section IV, inspired by our preliminary experimental results, we present PORT, our proposed mechanism that arbitrates the tradeoff in the design space, enjoys a theoretical convergence guarantee, and achieves the sweet spot in the spectrum between two extremes of communication mechanisms. In Section V, we first introduce new mechanisms we designed and built to simulate wall-clock time and to substantially improve reproducibility in PLATO, and then present an additional set of experimental results to validate the effectiveness of our design as compared to leading competitors such as FedAsync and FedBuff. Finally, we conclude the paper with some further remarks in Section VI.

[1]PORT is a strong, sweet, typically dark red fortified wine, originally from Portugal. It represents a *sweet spot* in our design space.

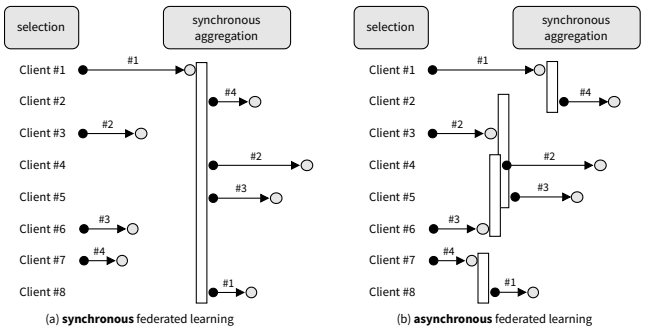


Fig. 1. Synchronous vs. asynchronous mode of operation in federated learning. Naturally, if the server operates in fully asynchronous mode, there is no need to wait for slower clients to report before the aggregation process to commence.

II. PRELIMINARIES AND RELATED WORK

The *synchronous* mode of operation in federated learning (FL) needs no introduction: most existing work in the literature makes such an assumption: the server needs to wait for all the clients it has selected in the current *communication round* (or *round*) to report their model updates, before it proceeds with the aggregation process. Just like *Bulk Synchronous Parallel* parameter servers in the conventional design of distributed machine learning on the same cluster, such synchrony across selected clients in the same round is simple to implement and enjoys proven convergence properties.

In practice, however, given a large number of clients in an FL session, it is natural to assume *client heterogeneity*: different edge devices as clients have a wide variety of computing capabilities, and as such their local training performance varies significantly. In fact, for the same amount of computation, their training times may follow a heavy-tailed distribution (such as Zipf distribution) rather than normal distribution, where a small proportion of clients are much slower than the norm. These slow clients were known as *stragglers* in distributed machine learning, and they motivated operating the training session in *asynchronous* mode.

Asynchronous federated learning. Figure 1 illustrates the intuitive benefits of deploying asynchronous federated

learning. In asynchronous mode, the server only needs to wait for a subset of clients to report before proceeding with aggregation processing. In the extreme case, the server only needs to wait for *one* client — who is the first to complete local training in a round — before proceeding. Such a *fully asynchronous* mode of operation was the essential idea of Asynchronous federated optimization (FedAsync) proposed by Xie *et al.* [2]. It proposed to use a mixing hyperparameter α, which is deployed on the server to determine how much weight should be assigned to the newly arrived model update from the fastest client during the aggregation process.

With such a fully asynchronous mode in operation, the slowest clients are no longer able to slow down the aggregation process in their role as *stragglers* in each round. However, when they eventually finish local training, their model updates may be based on a global model in a much earlier round than faster clients; in other words, these clients may become *stale*. Stale clients are known to affect model convergence in distributed machine learning: the stale synchronous parallel (SSP) mechanism [3] proposed to bound such staleness by asking the server to *wait for* slow clients who exceed a certain staleness threshold before proceeding with processing.

Unlike SSP, FedAsync [2] did not make an attempt to ask the server to wait for slow clients. Instead, it proposed to use a *staleness function* to compute the mixing hyperparameter α using the staleness of clients as input. Intuitively, the more "stale" a client is, the lower weight assigned to its model weights when they are aggregated to the global model. With such a simple design, FedAsync showed that it can solve regularized local problems to guarantee convergence, and has been used in later proposals as a benchmark when studying the performance of asynchronous federated learning.

Alternatives to asynchronous design. There exist several recent proposals in the literature that provided viable design alternatives to the asynchronous mechanism in FedAsync. Similar to such a staleness function in FedAsync, Chen *et al.* [4] proposed that clients that are more stale should be assigned a lower weight when their models are considered during the server aggregation process. FedSA [5] proposed a two-stage FL training process, where a large number of local training epoches, say, 20–50, are used in the initial stage of training for the sake of reducing the number of rounds, and then use a smaller number of local epoches in the convergence stage when the global model is starting to converge. The number of local training epoches is adjusted every round with an elaborate mechanism based on a proprietary definition of client staleness.

Similarly, Chen *et al.* [6] proposed Asynchronous Online Federated Learning (ASO-Fed), which operates in fully asynchronous mode where the server commences its aggregation process as long as it receives the report from the fastest client. It further augmented FedAsync with an elaborate mechanism of feature representation learning on the server, in order to address potentially negative effects on model performance due to asynchronous aggregation.

Semi-asynchronous designs. Wu *et al.* [7] proposed a semi-asynchronous federated learning mechanism, called SAFA, that introduced a hyperparameter called *lag tolerance* to represent the staleness of a client, and discarded local training results for those clients who are considered too stale according to such a lag tolerance setting. In SAFA, the *semi-asynchronous* server waits for a certain pre-specified percentage of clients before aggregation commences, and as such it should not be considered a fully asynchronous mechanism where aggregation starts with only one client reporting. Similarly, FedBuff [8] proposed *buffered updates*, where the server will wait for and buffer the updates from a minimum number of clients before aggregation. As proposed, both SAFA and FedBuff worked as a "hybrid" between fully asynchronous and synchronous modes of operation.

Synchronous designs that accommodated client heterogeneity. FedProx [9] was one of the first proposals that attempted to accommodate client heterogeneity by training for a smaller number of local epoches on slower clients. Though FedProx essentially operates in the synchronous mode, it was shown that such tolerance for partial work on the slower clients did not negatively affect convergence behaviour. A practical challenge, however, was that it was difficult to compute the number of local epoches that should be carried out on a particular client selected by the server, without *a priori* knowledge of the client's computing capabilities. Given the fact that a small subset is chosen from a much larger number of clients in practice, it is unrealistic to assume that each client's computing capabilities can be estimated *a priori* accurately. Choosing to work in the asynchronous mode offers a much simpler design with no such assumptions in place.

Open challenges. Essentially, existing work in the literature on asynchronous federated learning can be considered as *point solutions*, with each heuristic algorithm proposed representing one point of operation in the multi-dimensional design space in which potential designs can operate. The design choices and tradeoffs made in existing works — including hyperparameter settings — may not be well motivated, and the effectiveness of existing design choices were mostly illustrated using empirical evaluation.

Unfortunately, most existing works failed to use the *wall-clock time* elapsed since the start of a federated learning session to evaluate their proposed mechanisms. Instead, the number of communication rounds is most often used. Since asynchronous processing may advance the communication round index whenever a single client reports at the server, it is no longer a viable performance metric for performance evaluations, unlike synchronous federated learning. In rare cases, such as FedFA [5], where the wall-clock time was used for evaluation, it was not clear how client heterogeneity was simulated, how many clients were selected in each round, and what distributions were used for their training performance. As a result, without access to its source code, it would be difficult to *reproduce* its reported results.

III. Design Space and Motivating Observations

Though a more detailed introduction to an open-source implementation of our FL testbed will be deferred to Section V, we first show several motivation experiments to chart the design space for existing and potentially new asynchronous FL mechanisms.

Experimental settings. Our initial set of experiments involves benchmark FL sessions, involving 100 clients that use the `MNIST` dataset to train an `LeNet-5` model. This benchmark model is typically straightforward to train in a centralized training session, but due to non-i.i.d. data distributions in the federated learning context, training can be much slower to converge. In our experiments, the Dirichlet distribution with a concentration parameter of 0.8 is used to simulate the non-i.i.d. distribution.

Naturally, the asynchronous mode of operation is most suitable in the situation where a small number of clients are much slower than usual, forming a heavy-tailed distribution of local training speeds. In our testbed, we simulate such a situation by randomly generating the length of idle periods after each epoch is trained at a client, drawing from the Zipf distribution with parameter $s = 1.7$, and capped at a maximum length of 60 seconds.

Our initial set of experiments was conducted on Google Colab Pro+, with NVIDIA Tesla P100 GPU and 54.8 GB of main memory. The server selects 20 clients for training in each round (in the synchronous mode).

Minimum number of clients required. What are the most *influential hyperparameters* in the design space for asynchronous federated learning? The first we wish to experiment with is the *minimum number of clients required* to report before the server starts to aggregate these clients. At one extreme, when the minimum number is 1, the server operates in *fully asynchronous* mode: whenever a client arrives, it will be aggregated immediately without further delays. As examples, FedAsync [2] and ASO-Fed [6] proposed to operate in fully asynchronous mode. At the other extreme, when the minimum number of clients is the same as the number of clients selected in each round, the training session degenerates to the *synchronous* mode of operation in conventional federated averaging [1]: the server waits for all the selected clients to report before aggregation commences.

As we vary the number of minimum clients required before the server starts aggregation, we measure the elapsed wall-clock time, in seconds, as a critical performance metric. As we can immediately observe from our results, shown in Fig. 2a, the fully asynchronous mode of operation, which requires only one client to arrive before aggregation, failed to converge. This is due to the observation that, with only 600 samples utilized at each client, the server would repeatedly aggregate training results that used samples at the fast clients; and when results from much slower clients eventually arrive, they are based on models that were awfully out of date. Our non-i.i.d. data distribution further exacerbates the situation.

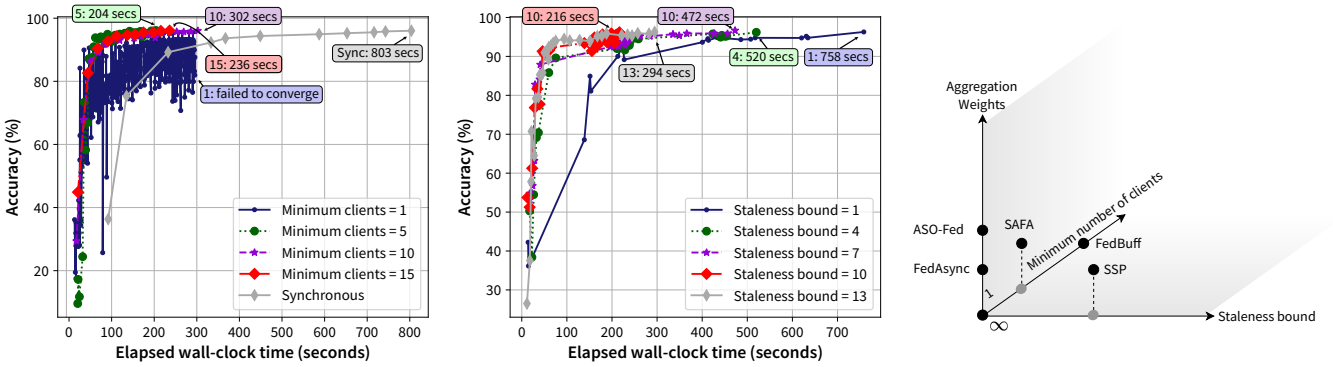
At the other extreme, synchronous FL did manage to converge, but over a much longer period of time (803 seconds). This reflected the well-known *straggler* problem, where the server needs to wait for several much slower clients in each round. It turns out that there was indeed a "sweet spot": in our experiments, aggregating a minimum of 5 clients fared the best: only 204 seconds were needed to converge to our target accuracy of 96%. But even if the server waited for 15 clients before aggregating them, training still converged in 236 seconds — not waiting for the last 5 clients made a world of difference!

The staleness bound. It has been known in the *stale synchronous parallel* mechanism (SSP) [3] that convergence can be guaranteed if stale clients beyond a certain bound are waited for during the aggregation process. However, it is not clear how different bounds of staleness would affect the amount of time it takes to converge. Intuitively, we may not wish to be too aggressive waiting for clients that are only slightly behind; but on the other hand, we may also not wish to accommodate clients that are too stale, such that the models they were based on were severely out of sync with most others. Shown in Fig. 2b, our initial experimental results on the MNIST dataset (with the number of minimum clients set to 5) appeared to have confirmed our intuition, in that there may indeed be a "sweet spot" in the staleness bound, and it was 10 in our initial experiments. The difference in performance is quite substantial: to reach a target accuracy of 96%, 758 seconds were needed with a staleness bound of 1, while only 216 seconds were needed with a staleness bound of 10.

Design space. From our preliminary experiments, it became obvious that multiple dimensions exist in the design space for asynchronous FL. In addition to the minimum number of clients required and the staleness bound, the design of a staleness-aware aggregation algorithm — with weights adjusted according to client staleness — represents another dimension. Conceivably, there exists an *optimal region* in such a multi-dimensional design space that leads to the best possible convergence behaviour in asynchronous FL. Yet, all the existing works only proposed *point solutions* in the design space, without considering how optimal they were. For example, FedAsync [2] did not impose any staleness bounds, and required a minimum of just one client only before aggregation. Our experiments suggested that it may not be operating in the optimal region. Fig. 2c illustrated where the operating points several existing solutions — FedAsync [2], SSP [3], FedBuff [8], ASO-Fed [6], and SAFA [7] — were located in the design space.

IV. Port: Design and Analysis

We are now ready to propose the design of Port, a new algorithm that seeks to operate in the best possible region in the design space for asynchronous federated learning. Upfront, as we have been abundantly clear, Port's design objective is to minimize the wall-clock time for a FL session to converge to a target accuracy, rather than the number of rounds. The upshot in Port's design is its new aggregation algorithm: rather than aggregating clients based on their percentage of samples as in federated averaging, Port's design focuses on



(a) Varying number of clients required before aggregation.

(b) Varying bounds of staleness.

(c) In the design space for asynchronous federated learning, existing mechanisms were point solutions but may not reflect the best possible operating point.

Fig. 2. Asynchronous vs. synchronous federated learning: comparisons and design space.

identifying influential factors that represent the staleness of clients; once they are identified, PORT is designed to discount the aggregation weights of stale clients accordingly.

A. Staleness: Influential Factors

It is natural to assume that much slower clients — who received the global model from the server several rounds ago — would become *stale*, and its model update may not be of high quality during the aggregation process. In fact, such a model update on a stale global model may even *interfere* with the approximate consensus from most other clients, and slow down the convergence process. Intuitively, we should reduce the weight assigned to these *stale* clients during aggregation.

But such an intuition raises an important question: what are the most important influential factors that best represent the *staleness* of a client's model update?

The staleness discount. The *client staleness*, defined as the number of rounds that elapsed since the last time a client received the global model from the server, arises as a natural choice: the more stale a client is, the more discounted its aggregation weight should become. Since PORT synchronously waits for clients that exceed the staleness bound, the client staleness will always be lower than such a bound. More formally, if τ is the current round at the server, and τ^k is the round that a reporting client k last received its model from the server, client k's staleness, S^k, is then $\tau - \tau^k$. We adopt the following *staleness function* that computes the *staleness discount*, which shall be used for discounting the aggregation weights:

$$s_\tau^k = \alpha \cdot \frac{\Omega}{S^k + \Omega} \tag{1}$$

where Ω is the staleness bound, defined formally as:

Definition 1 (Staleness bound)**.** *With the staleness bound of updates, Ω, the staleness $S^k = \tau - \tau^k$ of any reporting client k follows $S^k \leq \Omega$.*

Since S^k is upper bounded by Ω, the staleness discount s_τ^k is lower bounded by 0.5. α is considered a hyperparameter, serving as a tuning knob to control how significant the staleness discount should be in the aggregation process.

The interference discount. An important question at this point is: is there another influential factor that represents the staleness of client updates well?

Naturally, the result of server aggregation in the previous round, *i.e.*, $\boldsymbol{w}_\tau - \boldsymbol{w}_{\tau-1}$ where $\boldsymbol{w}_\tau$ denotes the parameters of the global model in the τ-th round, represents the general consensus of selected clients in that round. In the current round τ, consider a client, k, who just reported a weight update Δ_τ^k. Intuitively, if Δ_τ^k *interferes* significantly with the general consensus $\boldsymbol{w}_\tau - \boldsymbol{w}_{\tau-1}$, client k's update may not be of high quality and may need to be discounted during aggregation.

Mathematically, there are two ways of quantitatively evaluating a similarity measure between two vectors. One can compute the *dot product* between Δ_τ^k and $\boldsymbol{w}_\tau - \boldsymbol{w}_{\tau-1}$, which represents both the magnitude and the angle; alternatively, one can compute the *cosine similarity* instead, which represents the angle only.

In PORT, we choose to compute *cosine similarity*, denoted as Θ, to quantitatively evaluate interference. The lower Θ is, the less similar the two vectors are. Since $-1 \leq \Theta \leq 1$, we normalize it to $[0, 1]$ by computing $(\Theta + 1)/2$ instead. The interference discount is therefore defined as:

$$\Theta_\tau^k = \beta \cdot \frac{\Theta\left(\Delta_\tau^k, \boldsymbol{w}_\tau - \boldsymbol{w}_{\tau-1}\right) + 1}{2} \tag{2}$$

where $\Theta(A, B)$ is the cosine similarity between the two vectors A and B. Similar to the staleness discount, we introduce a hyperparameter β, serving as another tuning knob to control how substantial the interference discount should be.

With both influential factors incorporated as discounts, and assuming that each client k performs E training epochs based on its local dataset D^k, the aggregation weight for each client will then be computed as:

$$p_\tau^k = \frac{|D_k|}{|D|}\left(s_\tau^k + \Theta_\tau^k\right) \tag{3}$$

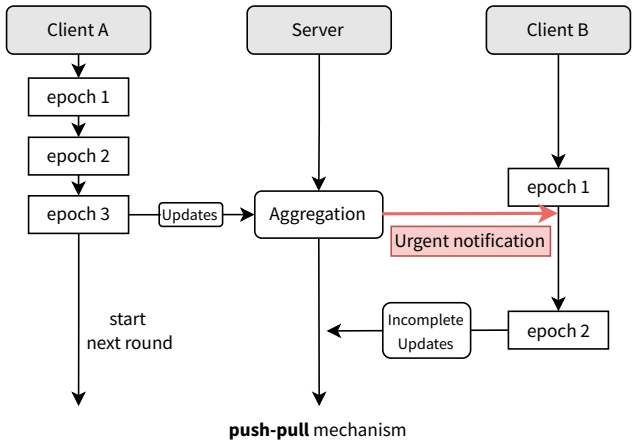


Fig. 3. A fast client, A, reports its updates in time for aggregation, as one of the minimum number of clients that the server waits for in this round. In contrast, a slow client, B, has just exceeded its staleness bound. The server sends an urgent notification to B and waits for its update after it finishes its current epoch of training.

where D is the set of all data samples used in the set of reporting clients, K, in this round. After normalizing all p_τ^k such that their sum becomes 1, the aggregation mechanism at the server can then be formulated as:

$$\boldsymbol{w}_{\tau+1} = \sum_{k \in K} p_\tau^k \boldsymbol{w}_\tau^k \tag{4}$$

B. Urgent Notifications and the Push-Pull Mechanism

In PORT's design, we recognize that the aggregation algorithm alone may not achieve the best possible performance when it comes to the *wall-clock time*, rather than the number of rounds. This is due to the fact that clients exceeding the staleness bound must be waited for, and with a heavy-tailed distribution of local training speeds, a small number of much slower clients may become "stragglers," increasing the amount of time to converge to the target accuracy.

To mitigate the negative effects of these stragglers, the server in PORT sends *urgent notifications* to all clients beyond the staleness bound Ω. Upon receiving such an urgent notification, a client will not proceed to the next epoch of local training; instead, it sends its local model as soon as the current training epoch finishes.

With such a design, in addition to clients *pushing* their models to the server as in conventional FL mechanisms, PORT allows the server to proactively *pull* the current models from the client using urgent notifications. This becomes handy when the server needs to wait for slow clients: rather than waiting for *all* local epoches to finish on a slow client, the server only needs to wait for the current epoch.

Keep in mind that both the minimum number of clients, discussed at length in Section III is still applicable in PORT. The server always waits for a minimum number of clients to report; but with the push-pull mechanism in place, it will also check whether any clients exceed the staleness bound. If so, they will each receive an urgent notification from the server. These urgent notifications introduce one more round-trip between the server and the stale clients: the clients will send their current models after they finish their current epoch of training, and the server waits for all stale clients to report before commencing its aggregation process including all the reporting clients in this round. Fig. 3 illustrates how the server waits for a push from a fast client, A, and pulls the update proactively from a slow client, B.

C. Convergence Analysis

To analyze PORT's convergence behavior, let us consider the following theoretical context. In each round $\tau \in T$ where T denotes the total number of rounds, the server selects K' clients from C clients. Each client k performs E training epochs based on its local dataset D^k and the model, $\boldsymbol{w}_{\tau^k}^k$, that it receives from the server in round τ^k. For any local training epoch $j \in [0, E]$, the local model $\boldsymbol{w}_{\tau^k,j+1}^k$ is obtained through optimizing $\boldsymbol{w}_{\tau^k,j}^k$ by using SGD with a batch size of B and a learning rate of η_l^j. This can be formulated as $\boldsymbol{w}_{\tau^k,j+1}^k = \boldsymbol{w}_{\tau^k,j}^k - \eta_l^j g(\boldsymbol{w}_{\tau^k,j}^k)$ where the gradient $g(\boldsymbol{w}_{\tau^k,j}^k) = \nabla f_k(\boldsymbol{w}_{\tau^k,j}^k, D^k)$. Once K clients have reported, the server commences the aggregation process.

In this context, our convergence analysis is conducted under the following assumptions, which are commonly used in previous works on the analysis of federated learning.

Assumption 1 (Smoothness)**.** *Each objective function f_k of the client k is L-smooth. Thus its derivatives are Lipschitz continuous with constant L, i.e., $\| \nabla f_k(\boldsymbol{w}) - \nabla f_k(\boldsymbol{w}') \| \leq L \| \boldsymbol{w} - \boldsymbol{w}' \|$.*

Assumption 2 (Unbiased local gradient)**.** $E_\xi [f_k(\boldsymbol{w}, \xi)] = \nabla f_k(\boldsymbol{w})$*, where $\boldsymbol{w}$ denotes trainable parameters.*

Assumption 3 (Uniformly bounded local gradient)**.** *The expected squared norm of stochastic gradients is uniformly bounded, i.e., $\mathbb{E} \| \nabla f_k(\boldsymbol{w}, \xi) \|^2 \leq G^2$ for $k = 1, \ldots, K$.*

Assumption 4 (Bounded local gradients)**.** *Let ξ be sampled from the k-th device's local data uniformly at random. The variance of stochastic gradients in each device is bounded as $\mathbb{E}_\xi \| f_k(\boldsymbol{w}, \xi) - f_k(\boldsymbol{w}) \|^2 \leq \sigma_k^2$ for $k = 1, \ldots, K$. Then, we define $\sigma_l^2 := \sum_{k=1}^K \frac{|D^k|}{|D|} \sigma_k^2$.*

Assumption 5 (Bounded gradient divergence)**.** *For any client k and the parameter $\boldsymbol{w}$, we define δ_k as an upper bound of $\| f_k(\boldsymbol{w}) - f(\boldsymbol{w}) \|^2$, i.e., $\| f_k(\boldsymbol{w}) - f(\boldsymbol{w}) \|^2 \leq \delta_k^2$. Then, we define $\delta_g^2 := \sum_{k=1}^K \frac{|D^k|}{|D|} \delta_k^2$.*

In essence, we are solving a generic optimization problem in federated learning, but the updates in the server aggregation process contain various gradient delays, *i.e.*, Eq. (4). PORT can be formulated as an asynchronous aggregation problem with buffered updates, which was previously discussed in FedBuff [8]. PORT's push-pull mechanism with urgent notifications also guarantees a staleness bound — as defined in Definition 1 — to client updates. In addition, we mathematically introduce a *staleness discount* by utilizing a client's

staleness to modulate its weights on a per-gradient basis. With the *interference discount*, we have Lemma 1 on the weight for each gradient.

Lemma 1. *Given the hyperparameters of staleness and interference discounts, α and β, the aggregation weight p_τ^k for each gradient can be bounded by $p_\tau^k \in \left[\frac{\alpha}{2}d_k, (\alpha+\beta)\,d_k\right]$ where $d_k = \frac{|D^k|}{|D|}$.*

Without loss of generality, we ignore the denominator term in Lemma 1 because it does not affect the proof of convergence. We can obtain PORT's convergence rate as follows:

Theorem 1 (Convergence rate)**.** *Following Assumptions 1 to 4 and Lemma 1,* PORT*'s convergence rate is formulated as:*

$$\begin{aligned}\frac{1}{T}\sum_{\tau=0}^{T-1}\mathbb{E}\parallel\triangledown f(\boldsymbol{w}_\tau)\parallel^2 \leq\; & 2\frac{(f(\boldsymbol{w}_0)-f(\boldsymbol{w}^*))}{\phi(E)TK} \\ & + 6K(\alpha+\beta)^2\lambda(d)L^2E\psi(E)\left(K^2\Omega^2+1\right)\sigma^2 \\ & + L\frac{\psi(E)}{K\phi(E)}(\alpha+\beta)\sigma_l^2\end{aligned} \tag{5}$$

where $\lambda(d) = \sum_{i=1}^K d_i^2$, $\phi(E) = \sum_{j=1}^E \eta_l^j$, $\psi(E) = \sum_{j=1}^E \left(\eta_l^j\right)^2$, and $\sigma^2 = (\alpha+\beta)\sigma_l^2 + (\alpha+\beta)\delta_g^2 + G^2$.

Also, to achieve the convergence upper bound, the relations of K and η_l should follow:

$$\frac{4(\alpha+\beta)}{\alpha^2\lambda(d)}K\eta_l^j \leq \frac{1}{L} \tag{6}$$

Proof. Following the commonly used convergence proof procedure in federated learning methods, such as [8] with the non-convex objective function, our proof begins by exploiting the smoothness assumption 1, thereby setting the upper bound of $f(\boldsymbol{w}_{\tau+1})$ to:

$$\begin{aligned}f(\boldsymbol{w}_{\tau+1}) \leq & f(\boldsymbol{w}_\tau) - \sum_{k\in K} p_\tau^k\langle\triangledown f(\boldsymbol{w}_\tau), \triangle_{\tau^k}\rangle \\ & + \frac{L}{2}\parallel\sum_{k\in K}p_\tau^k\triangle_{\tau^k}\parallel^2\end{aligned} \tag{7}$$

where $\triangle_{\tau^k} = \sum_{j=1}^E \eta_l^j \triangledown f_k\left(w_{\tau^k,j}^k\right)$.

Then, as presented in Eq. (4), PORT computes the staleness discount for each gradient. It introduces a new aggregation algorithm that represents a more general case than FedBuff [8], which utilized equal weights in the aggregation. To be more specific, we show a sketch of our proof in three parts.

First, we obtain the upper bound for three important components. Based on Assumptions 3, 4, 5 and Lemma 1, we bound the expectation of stochastic gradient $\mathbb{E}\parallel\triangledown f_k\left(w_{\tau^k,j}^k, D^k\right)\parallel$ of client k by $\sigma^2 = (\alpha+\beta)\sigma_l^2 + (\alpha+\beta)\delta_g^2 + G^2$. Then, by using Assumption 1 and adding a zero term in the decomposition, we prove that the upper bound for staleness-aware gradient divergence $\mathbb{E}\parallel\sum_{k=1}^K p_\tau^k\left(\triangledown f_k(w_\tau) - f_k\left(w_{\tau^k}^k\right)\right)\parallel^2$ is $6K\sum_{k=1}^K\left(p_\tau^k\right)^2\sum_{k=1}^K L^2Q\psi(E)\left(K^2\Omega^2+1\right)\sigma^2$. Finally, we bound the $\mathbb{E}\parallel\sum_{k\in K}p_\tau^k\triangle_{\tau^k}\parallel^2$ based on the Lemma 1 and Assumption 5.

Second, after introducing these derived components to Eq. (7), we reorganize the equation to get the specific upper bound for $\mathbb{E}[f(\boldsymbol{w}^\tau)]$. To remove the term containing $\mathbb{E}\parallel\triangledown f_k\left(\boldsymbol{w}_{\tau^k}^k\right)\parallel^2$, we tend to make the upper bound of its coefficient to 0, i.e., $-\frac{K}{2}\left(\sum_{k=1}^K\left(p_\tau^k\right)^2\right) + \frac{LK^2E\left(\eta_l^j\right)^2}{2}p_\tau^k \leq 0$. Thus, based on Lemma 1, we obtain Eq. (6).

Finally, with the simplified R.H.S. in Eq. (7), we sum up τ from 1 to T and rearrange the equation to obtain Eq. (5). □

Based on Theorem 1, we have Corollary 1:

Corollary 1. *Following the convergence rate in Theorem 1, we set the learning rate η_l to be a constant value that satisfies the condition in Eq.* (6)*, specifically $\eta_l = \frac{1}{\sqrt{TKE}}$. Then, for a sufficiently large T, we obtain:*

$$\begin{aligned}\frac{1}{T}\sum_{\tau=0}^{T-1}\mathbb{E}\parallel\triangledown f(\boldsymbol{w}_\tau)\parallel^2 \leq\; & \mathcal{O}\left(\frac{(f(\boldsymbol{w}_0)-f(\boldsymbol{w}^*))}{\sqrt{TKE}}\right) \\ & + \mathcal{O}\left(\frac{EK^2\Omega^2\sigma^2}{T}\right) + \mathcal{O}\left(\frac{E\sigma^2}{T}\right) \\ & + \mathcal{O}\left(\frac{\sigma_l^2}{K\sqrt{TKE}}\right)\end{aligned} \tag{8}$$

where $\sigma^2 = (\alpha+\beta)\sigma_l^2 + (\alpha+\beta)\delta_g^2 + G^2$.

The proof of this corollary is omitted due to space constraints. From our theoretical analysis, we can make several noteworthy observations on the influential factors over convergence.

The staleness bound Ω. Based on the second term in Eq. (8), the impact of the staleness bound Ω on the convergence dissipates at a rate of $1/T$. A large staleness bound is not desirable, as it dominates the increase of the second term. However, we can adjust the minimum number of clients to counteract its influence on the convergence rate.

The minimum number of clients K. As shown by the first term of Eq. (8) that presents the distance to optimal loss, increasing the minimum number of updates K induces a faster loss drop. However, the corresponding impact from variance σ^2 can be enlarged, thus improving the gradient drift in training. Also, once there is a large staleness bound Ω, waiting for more clients in the server aggregation introduces more serious out-of-date updates, negatively affect convergence. Therefore, we expect $K \in (1, M]$ in which M cannot be too large.

Compared to relate work such as FedBuff [8], PORT represents a more general case of semi-synchronous aggregation methods with buffered updates. Specifically, setting consistent weights $p_\tau^k = \frac{1}{K}$ for gradients in the server aggregation process, PORT's convergence rate naturally degenerates into Theorem 1 in [8].

V. PORT: IMPLEMENTATION AND EVALUATION

A. Simulating the wall-clock time within PLATO

In reviewing the literature in asynchronous federated learning, it is hard to overlook the fact that there does not exist a

federated learning framework that is the *de facto* standard on evaluating the performance of newly proposed mechanisms. Each proposed work used its own proprietary testbed for the purpose of performance evaluations, and no existing works shared their testbed implementation as open source, making it almost impossible to reproduced their experimental results. But such a glaring lack of *reproducibility* is not the only issue outstanding in the existing works: as a global model is trained in an FL session, they also resorted to gradients (*e.g.* [2]), the number of rounds (*e.g.* [6]), or the number of updates (*e.g.* [8]) till convergence to a target accuracy as the performance metrics to be evaluated.

Wall-clock time. While it is acceptable for synchronous FL, the number of rounds is no longer a suitable performance metric in the context of asynchronous FL, since different rounds would take substantially different durations in *wall-clock time*. Naturally, the wall-clock time elapsed before converging to a target accuracy is the most suitable — and arguably the most important — performance metric that should be evaluated when evaluating new asynchronous mechanisms. However, as we investigate the feasibility of measuring the wall-clock time in existing open-source federated learning frameworks, such as FedML [10], none of them offered this feature.

Conceptually, measuring the elapsed wall-clock time is straightforward: one would only need to watch the clock! As we look into this problem with greater depth, however, there is one major roadblock: there exists a very tight bound with respect to the number of clients training simultaneously, due to limitations in GPU memory in our experimental testbed (such as a GPU-powered virtual machine). For example, if one modern GPU is supported on the virtual machine (such as Tesla V100), we have 16GB of GPU memory, which can maximally train no more than 16 `LeNet-5` models concurrently, not to mention more complex models.

Why should we simulate the wall-clock time? In order to scale up to a number of clients that is greater than what GPU hardware can accommodate, we will have no choice but to train them in batches, one batch at a time. When clients in the same round are trained in batches, however, it is imperative to *simulate* the wall-clock time on the server. In synchronous mode, the server only needs to advance the wall-clock time by an amount equivalent to the training time of the slowest client in one round. Yet, simulating wall-clock time becomes quite a significant challenge when asynchronous federated learning is used.

Consider the case where 10 clients are selected in a particular round by the server. In practice when operating in fully asynchronous mode, the server would start the aggregation process when the fastest client reports its model update, and another client would be selected immediately after aggregation. Yet, if these clients train in batches — say two clients at a time — the server will have no choice but to wait for *all* 10 clients to arrive before it recognizes which client is the fastest. But when all the clients, including the slowest ones, have already finished training and reported to the server, how can the server select a new client in the next round immediately after the fastest client reports? What if the newly selected client in the second round quickly reports back again, earlier in wall-clock time than the remaining 9 clients in the first round?

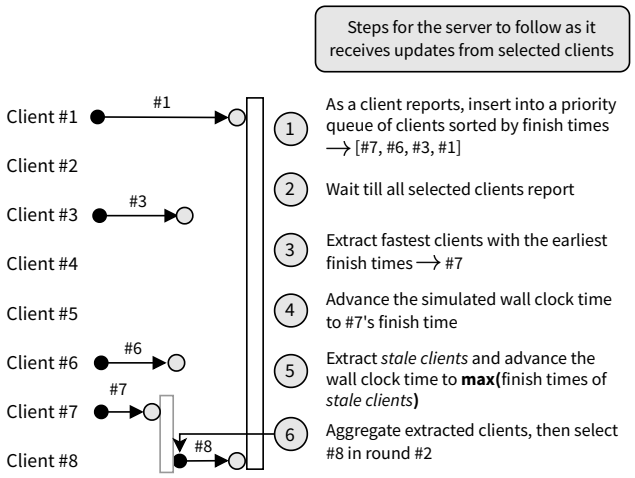


Fig. 4. When simulating the wall-clock time, the server needs to wait for all selected clients to report, store these reports in a list sorted by their finish times, and advance the simulated wall-clock time only when a client's update needs to be aggregated and replacement clients selected for the next round.

If we were to use the wall-clock time (rather than the number of rounds) as our performance metric, these are real and pressing research questions we are presented with when evaluating asynchronous FL mechanisms. Yet, no existing research software frameworks or papers on asynchronous federated learning provided any potential solutions, let alone satisfactory ones.

PLATO: an open-source research framework from scratch. As no existing works on asynchronous FL shared their implementations as open-source, and existing open-source FL frameworks fell short on evaluating asynchronous FL mechanisms appropriately, we implemented PLATO, an open-source research framework for scalable federated learning research from scratch. Development on PLATO started in November 2020, and so far required around 20 person-month of research and development time. PLATO is designed and built with several key objectives: it is *scalable* to a large number of clients; *extensible* to accommodate a wide variety of datasets, models, and FL algorithms; and *agnostic* to deep learning frameworks such as TensorFlow and PyTorch. In PLATO, clients communicate with the server over industry-standard WebSockets, while the server may either run in the same GPU-enabled physical machine as its clients — suitable for an emulation research testbed — or deployed in a cloud datacenter.

Simulating the wall-clock time. As a complete narrative on PLATO is beyond the scope of this paper, we only illustrate one novel design closely related to our performance evaluation: how is the wall-clock time simulated on a server operating in asynchronous mode?

Fig. 4 illustrates the steps that the server follows as it receives updates from its selected clients in the same round. Our overarching design assumption is that the server depends on accurate reports from the clients on the wall-clock times needed for their local training. As their reports are received, clients are locally managed by a priority queue, sorted by their finish times. Since the server has no *a priori* knowledge on the client's training speeds, it waits for all the clients selected in the same round to report before it starts processing them. As the server commences its processing, it would extract a number of fastest clients from the priority queue, and advance its wall-clock time accordingly.

In addition, in order to simulate a staleness bound where stale clients are to be waited for, the server would scan the remaining clients and extract the stale candidates, again advance its wall-clock time as clients that finish later are extracted. PORT's push-pull mechanism and urgent notifications can also be implemented in this context, by sending a message to each client beyond the staleness bound with a requested wall-clock time, and requesting them to send a local per-epoch checkpoint correspondingly. Finally, all extracted clients are aggregated to produce a new global model, which is used for the next round of client selection.

It may occur that faster clients in the next round finish even sooner than slower clients in the previous rounds that have not yet been extracted and processed. This can be handled correctly in our wall-clock time simulation, since all the clients who reported in real-time but are still considered training in simulated time would still be in the priority queue, sorted by their finish times. When the fastest clients are extracted and processed, they can belong to either the current round or any of the preceding rounds.

B. PORT *vs. State-of-the-Art: Experimental Evaluation*

Hyperparameter sweep with improved reproducibility. Our new mechanism of simulating the wall-clock time is indispensable when PORT is to be evaluated experimentally with respect to the elapsed wall-clock time, and should not be dismissed as an implementation detail. Before we evaluate PORT experimentally, however, we will need to perform a hyperparameter sweep for PORT's hyperparameters: α for computing the staleness discount in Eq. (1) and β for the interference discount in Eq. (2).

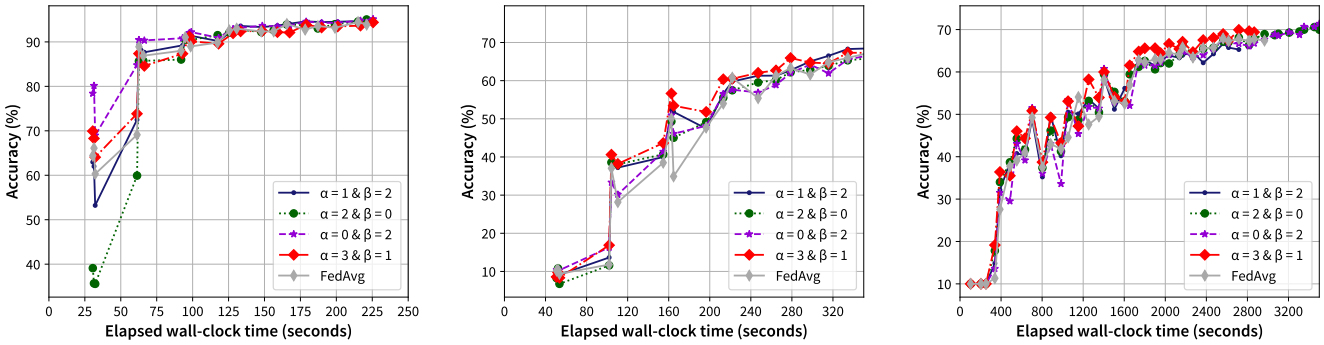
It turns out that, in order to compare the convergence performance across-the-board as α and β vary in our hyperparameter sweep, we will need to run a large number of FL training sessions. Yet, each of these sessions involves a substantial amount of randomization: the server randomly selects a number of clients, each client randomly sample the dataset with an i.i.d. or non-i.i.d. probability distribution, and the simulated client speed needs to be randomly sampled from a distribution as well. In PLATO, we support specifying random seeds for random number generators, and using `random.getstate()` and `random.setstate()` to protect random number generation from third-party frameworks. In addition, we preferred using Pareto distribution rather than Zipf for heavy-tailed client speed simulation, as Zipf is a discrete distribution and clients with the same speed may be processed differently across different runs. Finally, rather than measuring the actual wall-clock time of local training loops on each client, we use a pre-specified constant duration instead to ensure that the same set of clients would be selected in each round across different algorithms and runs.

With these mechanisms in place, PLATO is able to support a *reproducible* mode, which enables fully reproducible experiments where the same set of clients and data samples are selected across runs. Throughout this section, all our parameter settings have been released as open-source with PLATO to guarantee reproducibility (as the focus of our work), and all our experimental results were obtained on Compute Canada's `narval` supercomputer, with NVIDIA A100 GPUs and 128GB of memory. 100 clients participated in all our experiments, and up to 20 (for the synchronous mode) of which are randomly selected in each round. All non-i.i.d. data distributions are sampled with the Dirichlet distribution with a concentration parameter of 5, leading to a mild non-i.i.d. distribution.

For our purpose of hyperparameter sweep, a large batch of training experiments have been conducted, exploring values from 0 to 40 for both α and β. Fig. 5 presents a comparison across several representative pairs of values for α and β, in comparison with FedAvg and over three image classification tasks: MNIST and Extended MNIST (EMNIST) with the LeNet-5 model, as well as CIFAR-10 with the ResNet-18 model. Overall, $\alpha = 3$ and $\beta = 1$ offered a slight performance advantage against other value pairs. FedAvg also performed exceptionally well as this is a fully asynchronous incarnation of FedAvg, with the same minimum number of clients required before aggregation and the same staleness bound as its PORT counterparts. This shows convincing evidence that, even with PORT's aggregation algorithm only, it can decidedly outperform federated averaging with all other settings left alone.

PORT without urgent notifications vs. its competitors: experimenting with the staleness bound. In the next batch of experiments, we wish to further explore PORT's performance in the context of its three leading competitors: FedAvg [1], FedAsync [2], and FedBuff [8]. In addition to EMNIST and CIFAR-10, we added the CINIC-10 dataset and the VGG-16 model as a new benchmarking task, with 90000 training data samples and a randomly sampled 5000 validation data samples. Since our focus in these experiments was on a thorough evaluation of the effectiveness of applying a staleness bound, we decided to continue to maintain our reproducible mode where local training times and all random seeds are pre-specified and comparable across-the-board. PORT's push-pull mechanism with urgent notifications, on the other hand, was not activated as it required access to the actual training times in each epoch. Fig. 6 showed our results, comparing PORT (without urgent notifications) with its competitors.

Right off the bat, we can observe that FedAsync completely failed to converge. In fact, with additional experiments, we concluded that it failed to converge regardless of its choice of staleness functions. This is partly due to its aggressive

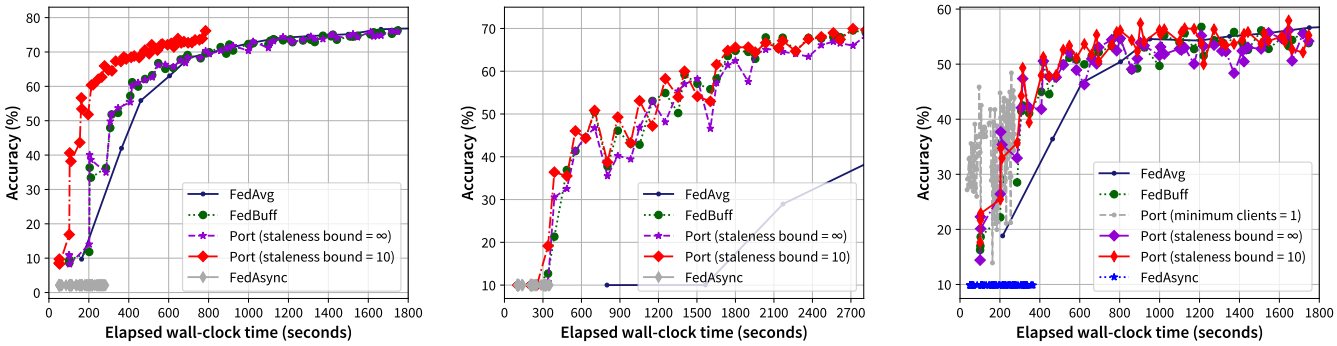


(a) Evaluating the effects of hyperparameters α and β with the MNIST dataset.

(b) Evaluating the effects of hyperparameters α and β with the EMNIST dataset.

(c) Evaluating the effects of hyperparameters α and β with the CIFAR10 dataset.

Fig. 5. Evaluating the effects of hyperparameters α and β with comparisons across-the-board in reproducible experiments.



(a) PORT vs. its competitors with EMNIST.

(b) PORT vs. its competitors with CIFAR10.

(c) PORT vs. its competitors with CINIC10.

Fig. 6. PORT (*without* urgent notifications) vs. FedBuff, FedAsync and federated averaging: a performance comparison with a focus on the staleness bound.

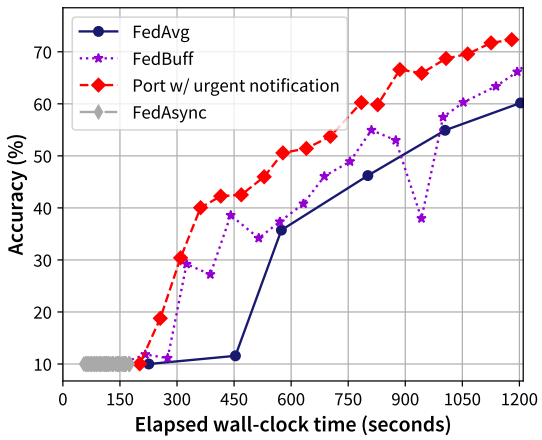
behaviour of aggregating the fastest client immediately as it arrives, and partly due to the design of its own aggregation algorithm. In fact, in Fig. 6c, we showed the results of using PORT with a minimum number of clients of 1, and though it didn't perform well, it outperformed FedAsync substantially.

Though it is not a surprise that PORT outperformed synchronous federated averaging in terms of wall-clock times across all three datasets, the question whether it outperforms FedBuff — with the same minimum number of clients — needs a more detailed explanation. As FedBuff does not wait for clients beyond a staleness bound, it effectively has a staleness bound of ∞. The question becomes, therefore, whether having a staleness bound of ∞ performs better than having a limited staleness bound of, say, 10 (which is the "sweet spot" we identified in Section III). For this reason, we experimented with all three cases: FedBuff, PORT with a staleness bound of ∞, and PORT with a staleness bound of 10.

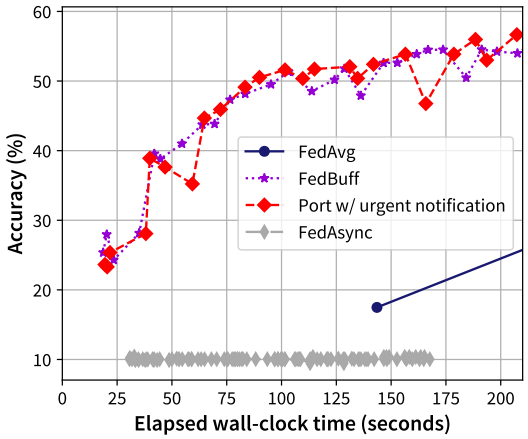
The verdict is in: with a staleness bound of ∞, PORT performed very similarly as compared to FedBuff, mostly due to the fact that most of the client updates aggregated are not stale. Both, however, suffered from the phenomenon that when stale clients eventually arrive, they may affect the accuracy of the model negatively for a few rounds. With a staleness bound of 10, PORT performed quite visibly better with all three datasets, and especially with EMNIST. With this round of experiments, we are able to make the counter-intuitive observation that, with respect to the wall-clock time, it is worth imposing a reasonable staleness bound, even when we need to wait for the slow clients to arrive.

PORT with urgent notifications vs. its competitors: the finale. With all the results focused squarely on reproducibility and fair comparisons across-the-board, we are now ready to start measuring the actual training times, and turn on the push-pull mechanism and urgent notifications in PORT. Our results are shown in Fig. 7. With the CIFAR-10 dataset, PORT used a low staleness bound of 3 so that urgent notifications can be sent to a client when the staleness bound is reached. In this scenario, PORT has clearly shown its forte: it took only 578 seconds to reach 50%, and 1125 seconds to reach 70%. In comparison, its closest competitor, FedBuff, took 811 seconds to reach 50% and 1311 seconds to reach 70%. This shows that PORT enjoyed a performance margin of up to 40% over FedBuff.

What about a higher staleness bound? With the CINIC-10 dataset, PORT applied a staleness bound of 13, and we observed that it progresses in a lock-step manner with FedBuff, and only started to show a marginal advantage towards the end of the convergence. As each client used only 3% of the total samples for its training with the CINIC-10 dataset — as



(a) PORT (*with* urgent notifications and a staleness bound of 3) vs. its competitors: CIFAR-10.



(b) PORT (*with* urgent notifications and a staleness bound of 13) vs. its competitors: CINIC-10.

Fig. 7. PORT (*with* urgent notifications vs. FedBuff, FedAsync and federated averaging with the CINIC-10 and CIFAR-10, and a mild non-i.i.d. Dirichlet data distribution.

opposed to 10% with CIFAR-10 — this phenomenon can be attributed to the observation that a staleness bound of ∞ may not be detrimental at all if local training completes quickly and the turnover rate to new clients is high. In these situations, PORT's performance advantage over FedBuff may diminish, as the effects of urgent notifications to stale clients take a less significant role. It is also worth noting that due to the randomness of measured training times, without activating the reproducibility mode in PLATO, comparisons between close competitors such as PORT and FedBuff can vary over different datasets and runs. As compared to FedAsync (which failed to converge) and federated averaging, however, it goes without saying that PORT's performance advantage is substantial with both datasets.

Last but not the least, it is worth noting that, in contrast to operating in a fully asynchronous mode — having a staleness bound of ∞ — in FedBuff, a *finite* staleness bound provides a well-known and attractive theoretical property that training will be guaranteed to converge [3]. Although FedBuff always converged in our experiments, having a theoretical guarantee offers additional peace of mind, as it is unlikely to experiment with all potential combinations of parameter settings.

VI. CONCLUDING REMARKS

How asynchronous can federated learning be? Towards providing a convincing and reproducible answer, we made several original contributions in this paper. We first explored the entire design space of asynchronous FL involving multiple parameterized dimensions, and argued that an optimal region of operation should involve proper design choices along each of the dimensions. We then presented PORT, our proposed aggregation algorithm that utilizes well-chosen operating points in each of the dimensions in our design space, and with proven convergence guarantees.

One of the highlights in our contributions is the ability of simulating wall-clock times and improving reproducibility in our experiments, thanks to our new open-source FL framework, PLATO, developed from scratch. We are able to take advantage of such improved reproducibility during PORT's hyperparameter sweep, and to present PORT's superior performance in wall-clock times compared to FedBuff, FedAsync, and federated averaging. While we are confident on PORT's performance across new datasets and models, the open-source and reproducible nature of PLATO opens up new opportunities for pushing the performance envelope further with new and improved designs in the future with convincing and reproducible results.

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 *Proc. Artificial Intelligence and Statistics (AISTATS)*, 2017.

[2] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous Federated Optimization," in *Proc. NeurIPS Workshop on Optimization for Machine Learning (OPT)*, 2020.

[3] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, vol. 26, 2013.

[4] Y. Chen, X. Sun, and Y. Jin, "Communication-Efficient Federated Deep Learning with Layerwise Asynchronous Model Update and Temporally Weighted Aggregation," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 10, pp. 4229–4238, 2020.

[5] M. Chen, B. Mao, and T. Ma, "FedSA: a Staleness-Aware Asynchronous Federated Learning Algorithm with Non-IID Data," *Future Generation Computer Systems*, vol. 120, pp. 1–12, 2021.

[6] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous Online Federated Learning for Edge Devices with Non-IID Data," in *Proc. IEEE International Conference on Big Data*, 2020, pp. 15–24.

[7] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis, "SAFA: A Semi-Asynchronous Protocol for Fast Federated Learning with Low Overhead," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 655–668, 2021.

[8] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated Learning with Buffered Asynchronous Aggregation," in *Proc. International Conference on Machine Learning (ICML)*, 2021.

[9] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated Optimization in Heterogeneous Networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 429–450, 2020.

[10] C. He and *et al.*, "FedML: A Research Library and Benchmark for Federated Machine Learning," in *Proc. NeurIPS Federated Learning Workshop*, 2020.