COMP 4801 Final Year Project

Efficiency and Robustness in Federated Learning

Supervisor	Professor C. L. Wang
Second Examiner	Dr. Heming Cui
Name	Yue Yin, BEng(CS), 3035448512

List of Figures

List of Tables

Contents

1 Introduction

Carrying powerful sensors and being used at high frequency, digital devices like phones and tablets are becoming the primary computing devices for most people nowadays. Such devices have access to an unprecedented amount of private user data, which can be very valuable to machine learning applications. However, because of the privacy concern and high cost of network communication, traditional distributed machine learning scheme becomes infeasible, as it requires uploading user data to some centralized location like a data-center [1].

Federated Learning (FL) [1] has been proposed to address the problem. Federated learning performs machine learning on sensitive data without compromising user privacy, as it trains models without directly accessing private user data. Federated learning brings about new opportunities as well as new challenges, in terms of the efficiency and robustness of the scheme.

The synchronization scheme fundamentally affects the performance of a federated learning algorithm. Different schemes have been proposed, including Bulk Synchronous Parallel (BSP), Asynchronous Parallel (ASP), Stale Synchronous Parallel (SSP), to address different problems. However, most schemes requires the trade-off between convergence speed and model accuracy. This project aims to explore new synchronization scheme variants in federated learning that achieves both simultaneously.

In this project, I focus on the SSP synchronization scheme, the generalized case of BSP and ASP. I explored the *Adaptive-SSP* strategy, which provides faster convergence without sacrificing model accuracy through switch-point detection. I implemented the Adaptive-SSP strategy and evaluated its performance through experiments.

2 Background

2.1 Federated Learning

2.1.1 Motivations

Mobile devices with powerful sensors can access an enormous amount of private user data. Such data are usually sensitive to privacy and uploading such data to the data center for modeling training should be avoided. By decoupling data access from model training, federated learning utilizes private user data without compromising user privacy.

2.1.2 Working mechanisms

A federated learning system is structured similar to the parameter-server model [2], which is proposed to solve the problem of distributed machine learning. In distributed machine learning, the centralized server orchestrates clients. In federated learning, one round of training consists of four steps:

1. Server samples a set of clients. The initial model is downloaded from the server by sampled clients.
2. Each client computes model updates (gradients).
3. Each client uploads its updates to the server.
4. Server aggregates clients' updates to update the global model.

In this paper, this four-step synchronization cycle is referred to as a *clock*.

Federated learning attempts to solve the optimization problem of

$$\min_{\omega} F_i(\omega_i) = \frac{1}{n_i} \sum_{i=1}^{n_i} f(x_j^i; \omega_i),$$

where x_j^i is the j-th data point in the i-th client. At clock t, ω_t can be computed by local gradient descent:

$$\omega_t := \omega_{t-1} - \eta_{t-1} \nabla F(\omega_{t-1}).$$

where η_{t-1} is the learning rate.

2.1.3 Differences from distributed machine learning

A key difference between distributed machine learning and federated learning is that user data are not uploaded to the data center. Instead, private use data remains on the client devices, and the FL server aggregates updates from clients to update the global model.

I summarize other attributes that distinguish federated learning from distributed machine learning:

- **Non-IID Dataset.** In distributed machine learning, each client's dataset is generated by partitioning and shuffling the global dataset, and thus becomes independent and identically distributed (IID). For FL, each training data set is collected independently from the use of each specific device, and any specific user's data may not represent the global data set [1,3], so the data is not Non-IID.
- **Uneven Distribution.** In distributed machine learning, the global dataset is uniformly partitioned into blocks of comparable sizes. In federated learning, the amount of data on each device varies greatly [1].
- **Resource Heterogeneity.** In distributed machine learning, clients are deployed on machines in the same data center with similar resources. For FL, significant variance exists in the characteristics of end devices in terms of CPU, RAM, training participation rate, etc [3]. Stragglers (slower clients) typically exist in the FL setting.
- **Unreliable Communication.** In distributed machine learning, machines are typically inter-connected by cable networks with low-latency and high-bandwidth. For FL, end devices are usually connected by wireless networks, so communication can be slow, unreliable and expensive [1].

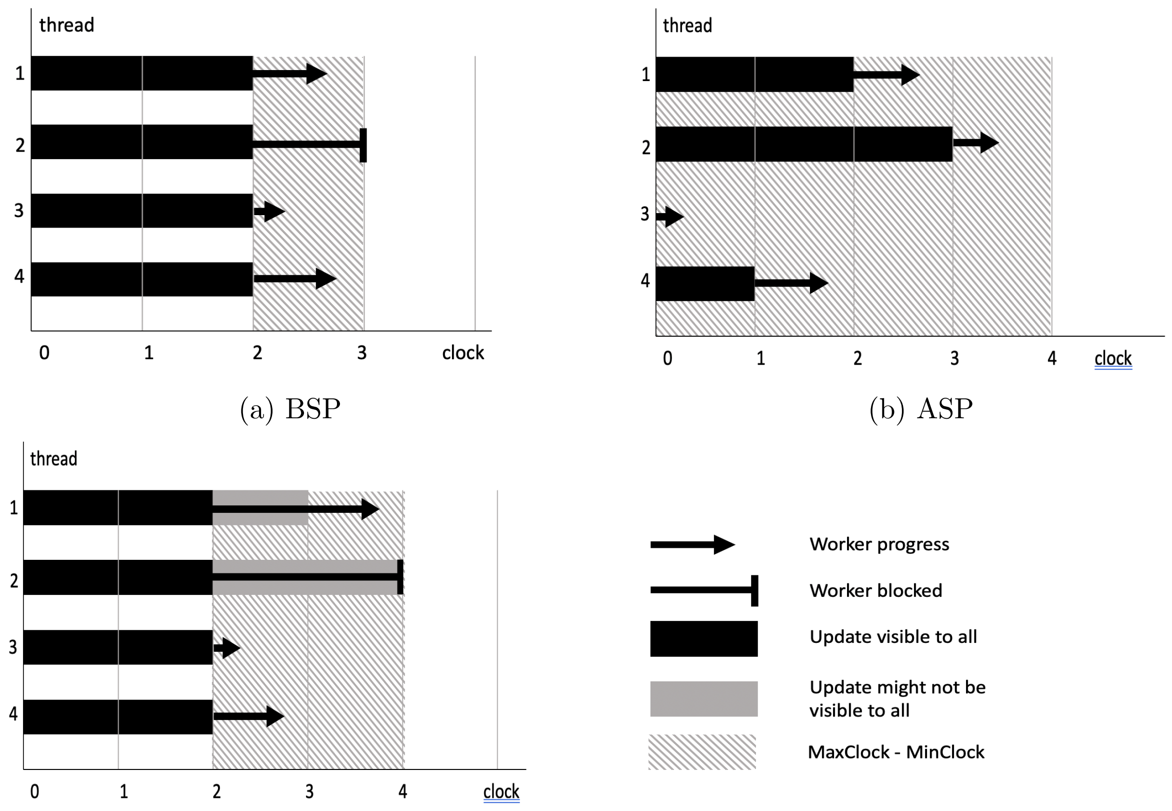
2.2 Synchronization schemes

Three major synchronization schemes have been proposed for distributed machine learning. The three schemes cover most existing federated learning systems.

BSP: Bulk Synchronous Parallel. Systems like FedAvg [1] and FexProx [3] synchronizes clients with the BSP protocol. Under this protocol, one client is blocked in the current clock until the parameter receives updates from all clients in that clock. When stragglers exist, the overall training time is determined by the slowest client.

ASP: Asynchronous Parallel. Systems like [4,5] use an ASP protocol for synchronization. The protocol does not constrain clients' clocks and clients proceed freely without any waiting. ASP systems tend to be faster than BSP systems [4], because of the loose synchronization constraints. However, ASP cannot guarantee convergence in the case of significant stragglers [5].

SSP: Stale Synchronous Parallel. Systems like DSSP [6] and FlexPS [7] employ the SSP strategy [8], which limits the gap between the slowest client's clock and the fastest client's clock with a *staleness bound*, denoted as s. Under this scheme, the fastest worker cannot exceed the slowest one by more than s clocks.



(a) BSP (b) ASP (c) SSP

Figure 1: BSP, ASP, SSP

SSP can be considered as the general model: when $s = 0$, SSP is equivalent to BSP; when $s = \infty$, SSP is equivalent to ASP. Figure 1 visualizes the three schemes. In this project, I use SSP for synchronization in FL.

To distinguish FL schemes from distributed ML schemes using similar protocols, I refer to the three schemes as BSP-FL, SSP-FL, and ASP-FL.

2.3 SSP-FL and staleness bound

In federated learning, stragglers make highly synchronous schemes (like BSP-FL) inefficient, because the overall training time is determined by the slowest worker [9]. On the other hand, non-IID datasets cause the inconsistency between the local optimums and the global optimum, making insufficiently synchronous schemes (like ASP-FL) suffering the problem of incorrect convergence [5].

As an intermediate solution between BSP and ASP, SSP is faster than BSP while achieving an accuracy higher than ASP [6]. However, SSP-FL has no guarantee in either **efficient convergence** or its **robustness against incorrect convergence**, as both of the two crucial aspects of FL depend on the value of the staleness bound, s. SSP uses a fixed value for s throughout the entire training process. However, it is typically difficult to find a single proper value of s. In fact, determining a suitable s typically involves manually searching in a range of integer values through trial-and-errors [6].

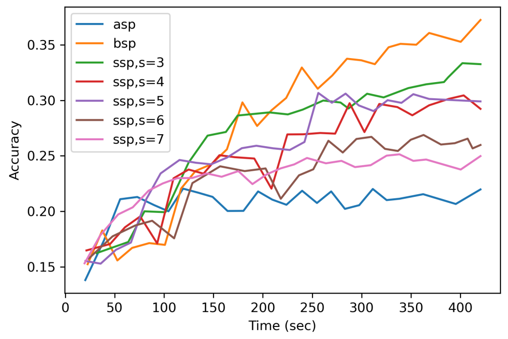
In short, by combining BSP and ASP, SSP can potentially avoid ASP's disadvantage of being prune to incorrect convergence and BSP's inefficiency due to stragglers. However, considering the difficulty of finding a good value of s for SSP-FL, the problem of efficiency (short training time) and robustness (against incorrect convergence) remains to be solved.

3 Methodology

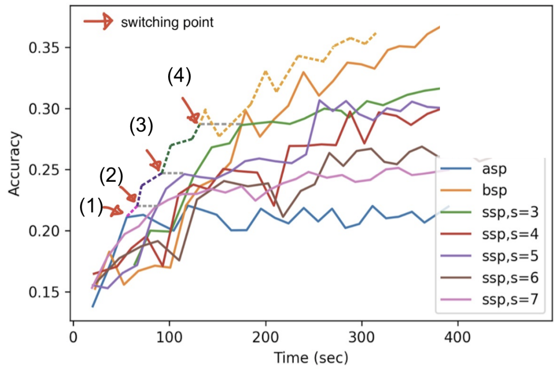
3.1 Staleness Bound

SSP strikes a balance between ASP and BSP through staleness bound s, the parameter that bounds the gap between the fastest client(s) and slowest worker(s) [8]. In distributed machine learning, the parameter is crucial for the performance of SSP-FL, as it affects both the training time and the model accuracy. As s increases, the model converges in a shorter time [10], but the accuracy drops [11].

In federated learning, the impact of staleness bound on convergence time and model accuracy has not been thoroughly examined. Thus, I conducted experiments to explore the impact of staleness bound in federated learning. Our preliminary evaluation result in Figure 2a shows that the effect of s is similar.



(a) SSP, different s



(b) Adaptive s

Figure 2: The effect of staleness bound

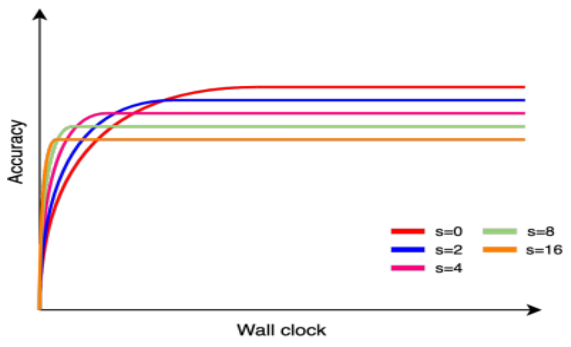
I summarize the relationship between s, model accuracy and training time again:

When s increases, the model converges faster, but the accuracy ceiling is lower; When s decreases, the training time increases, the model takes longer to converge, but the accuracy ceiling is higher.

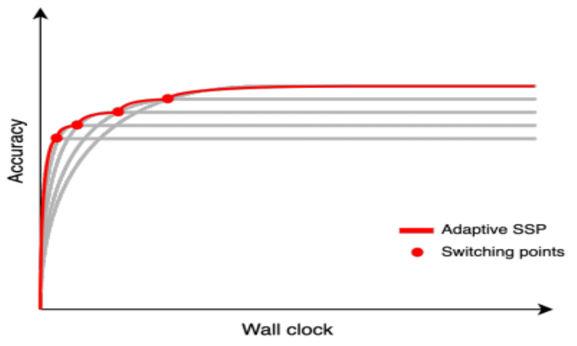
3.2 Adaptive strategy

From Figure 2a, I make the following observation: with a large s, the SSP-FL model reaches lower accuracy quickly; with a small s, the SSP-FL model reaches higher accuracy with longer training time. In addition, it takes longer for a SSP-FL with a larger s to reach a certain accuracy level. The idea is illustrated in Figure 3a.

Thus, a fixed staleness bound s necessarily involves the trade-off between convergence time



(a) Ideal SSP, different s



(b) Adaptive $SSP - FL$, ideal case

Figure 3: The effect of staleness bound

and model accuracy. However, in our Adaptive-SSP, SSP-FL varies s throughout the training process. Intuitively, at the beginning of training, SSP-FL should use a large s to quickly converge to a relatively slower accuracy. Whenever the model converges, i.e., reaching the accuracy ceiling of the current s, SSP-FL can decrease the value of s, allowing SSP-FL to converge to a higher accuracy with longer training time.

The idea of Adaptive-SSP is illustrated in Figure 2b, where time-points to reduce s are marked (referred to as ”switching point”s). Ideally, if the strategy can correctly detect the switching points, the performance of the adaptive strategy (dashed line) will always outperform any SSP-FL strategy with fixed s (including BSP-FL and ASP-FL). Figure 3b visualizes the ideal situation of Adaptive-SSP.

3.3 Switching Point Detection

Switching point detection is crucial to the Adaptive-SSP strategy. This section discusses different schemes to detect switching points in Adaptive-SSP, with different advantages and shortcomings.

3.3.1 Server-driven Adaptive-SSP

Server holds global test dataset. For discussion purposes, I temporarily make the hypothetical assumption that the global test dataset is available to the SSP server. Under this assumption, the global test dataset enables the server to have an accurate estimation of the current model accuracy. Thus, whenever the server receives a parameter update from the client, it can apply the update to the global model, and evaluate it with the global test dataset to obtain the latest model accuracy. By monitoring the global model accuracy, the server can detect model convergence and decrease s. This accuracy tends to be accurate as global test dataset is used.

Server does not hold global test dataset. The previous assumption is clearly unrealistic: FL is designed to protect user privacy and train model without uploading user data to centralized server. Thus, the switching point detection scheme must rely on information from the clients. Instead of returning only parameter updates, the client also returns loss, accuracy, and local dataset size. The server can aggregate such information to estimate the current model accuracy and detect model convergence.

3.3.2 Client-driven Adaptive-SSP

Another possibility for switching point detection is to have clients control their own staleness bounds. Instead of relying on the server, each client monitors the model accuracy on its own dataset and varies its staleness bound independently from others.

3.4 Convergence Detection

In Adaptive-SSP, switching points are detected by model convergence detection. This section discusses possible strategies to detect model convergence.

- **Gradients monitoring.** The strategy monitors gradients computed by the clients. For example, by computing the sum of squares of gradient entries, the strategy can estimate the amount of change of each gradient. Whenever receiving a series of gradients with the sum of squares smaller than a certain threshold, the strategy reports model convergence.
- **Accuracy monitoring.** The strategy monitors the model accuracy and reports model convergence when the accuracy changes are smaller than a certain threshold. For example, the strategy can report model convergence when the standard variance of consecutive accuracy is smaller than a certain threshold.
- **Gradient products monitoring.** This strategy performs a statistical test based on a variant of stochastic approximation [12, 13]. When the model has not converged yet, due to the exponential decrease of the error, gradients are likely to point in the similar direction, making the inner product of consecutive gradients positive. However, when the model is near convergence, negative products will accumulate. When the negative products have reached a certain threshold, the strategy can report model convergence.

4 Implementations

I implemented Adaptive-SSP and different switching-point detection strategies based on an open-source federated learning framework, Flower [14]. In this section, I examine the architecture of Flower, and explain how to adapt the framework to support Adaptive-SSP and different switching-point strategies.

4.1 Flower Framework

4.1.1 Framework Architecture

Flower is an open-source federated learning framework agnostic to heterogeneous client conditions and can scale to different numbers of clients [14], provided with well-written documentation. Designed to help make the transition from existing machine learning code to federated learning simple, Flower is mostly built upon common ML libraries such as Tensorflow [15] and PyTorch [16]. Such implementation makes Flower clearly structured and grants room for extension, especially for the synchronization strategy. The architecture of the framework is visualized in Figure 4.

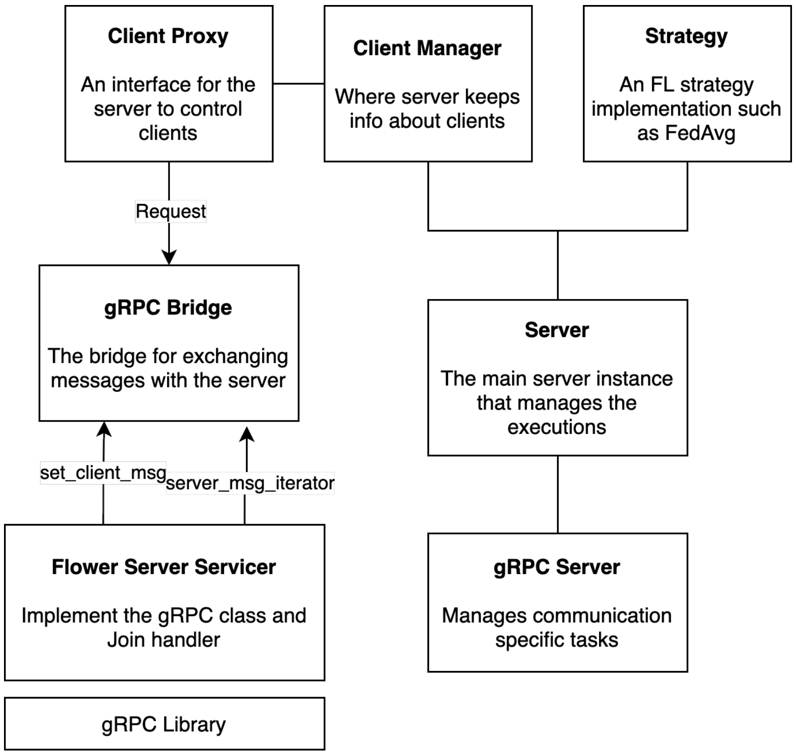

Client Proxy
An interface for the server to control clients
Client Manager
Where server keeps info about clients
Strategy
An FL strategy implementation such as FedAvg
Request
gRPC Bridge
The bridge for exchanging messages with the server
Server
The main server instance that manages the executions
set_client_msg
server_msg_iterator
Flower Server Servicer
Implement the gRPC class and Join handler
gRPC Server
Manages communication specific tasks
gRPC Library

Figure 4: The structure of Flower Framework

Network communication. The Flower framework provides an abstraction over the communication layer and hides the network details with `GRPCBridge` and `ClientProxy`, accommodating different network conditions like 2G/3G mobile network, cable network, WiFi, etc. Flower relies on gRPC, which provides efficient serialization in binary format.

Server. On the server side, the `ClientManager` class is responsible for client registrations at the start of FL, and creates a `ClientProxy` for each client when successfully registered. The `ClientManager` maintains the list of clients and their states, to provide an abstraction for the server. The `ClientProxy` is the interface through which the FL server controls clients, and exposes APIs like `fit`, `evaluate`, etc. The server orchestrates the clients, and implements two methods in the abstract base class `ServerBase`: `fit` and `evaluate`.

Client. On the client side, the base class `Client` implements three methods: `get_parameters`, `fit`, and `evaluate`. The `start_client` function repeatedly reads message from the gRPC connection, de-serializes the binary message, and calls `handle` to dispatch to corresponding methods. The `NumPyClient` inherits from `Client`, and returns NumPy result for `fit`, `get_paramters`, and `evaluate`. The execution is driven by the wrapper class `NumPySSPClientWrapper`, which implements the same set of wrapper functions of the same name. This is visualized in Figure 5.

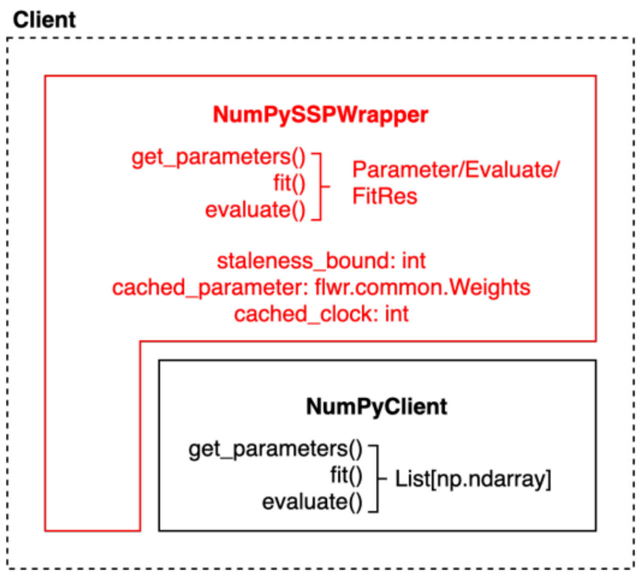


Figure 5: Flower client

Client-Server interaction. When the server starts, it waits for enough clients to register. After that, it calls `fit` to execute logic for federated training. After `fit` finishes, it calls `execute` to evaluate the trained model, and record metrics like loss, accuracy, training time, etc. In `fit`, the server sends messages like `fit` and `get_parameters` to clients. When the server invokes a method on a `ClientProxy` to send a message to the client, the message is serialized with the Google Protocol Buffers protocol, and pushed into the `GRPCBridge` (implemented as a FIFO queue). Each client continuously monitors the `GRPCBridge` and dispatches the message to the handlers accordingly. When the client finishes execution for the current message, it sends the reply message back to the server by pushing to the

`GRPCBridge` in a similar manner.

4.1.2 Framework Adaption

The following characteristics make it easy to implement variants and extensions on Flower:

Useful abstractions. The Flower framework provides nice abstractions for the network communication layer, which greatly reduces the work required compared with implementing a new FL framework from scratch. Also, by providing base classes like `ServerBase` and `Client` with well-defined methods, the framework clearly separates the details internal to the framework from aspects external to the framework, like aggregation strategy, client models, etc.

ML-framework agnostic. The architecture of Flower makes it agnostic to the execution logic of the server and clients, as long as they implement the required methods. The important implication is that it supports model implementation in any ML frameworks, or even no framework at all. This is superior to other frameworks like TensorFlow Federated (tightly coupled with TensorFlow), PySyft (supporting only PyTorch and Tensorflow), and LEAF (depending on TensorFlow).

4.2 Stale Synchronous Parallel

This section explains the steps taken to adapt Flower to support Stale Synchronous Parallel (SSP). In 4.2.1, I will review SSP-FL algorithm. In 4.2.2, I will explain how to adapt Flower to support asynchronous training. In 4.2.3, I will explain how I change message formats of `FitIns` and add additional message types (`ReadIns` and `ReadRes`) to support efficient model pulling.

4.2.1 SSP in FL

SSP-FL has the following characteristics [8]:

- Each client participates in training in an asynchronous manner. The global model is immediately updated whenever the server receives a training result from a client;
- If the cached model is not s-old, the client continues training with the cached model;
- If the cached model is at least s-old, the client downloads a fresh model from the server;
- If the clock of the fastest client is greater than that of the slowest client by s, the fastest client will be blocked until the gap drops below s.

The parameter s is the *staleness bound* in the SSP model. When receiving a global model from the server, the client updates its cached global model and the clock.

4.2.2 Asynchronous Training

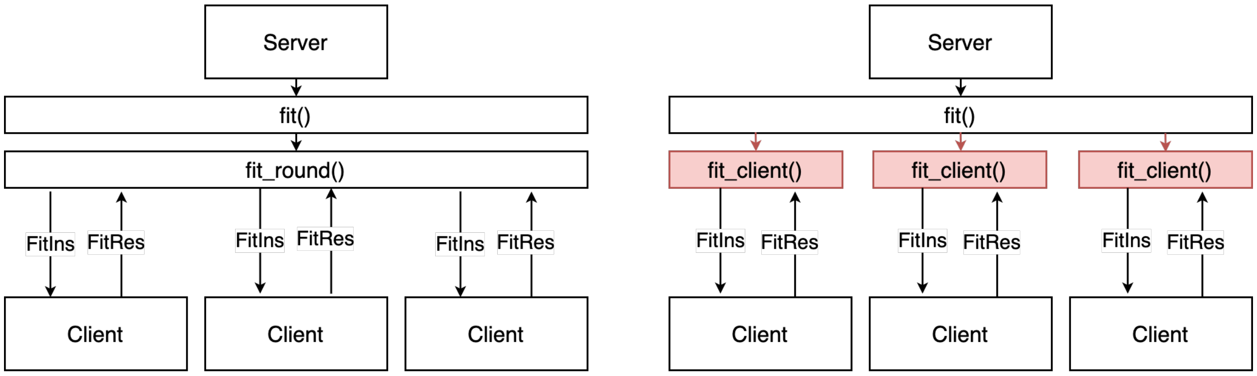


Figure 6: Original training structure (*Left*) and modified training structure (*Right*)

Flower is originally designed to execute BSP-FL (like FedAvg [1]). Flower server notifies the clients to start training with the `FitIns` message, representing a Fit Instruction. The `FitIns` includes the global model at the server side. Flower clients receive the global model, fit the model using the local dataset, and send back `FitRes`, representing Fit result, to upload the gradients to the global model. In each round, the Flower server sends `FitIns` to all registered clients and wait to receive all their `FitRes` before proceeding to the next round. This synchronous training approach is visualized in Figure 6 (Left).

To support asynchronous training in SSP, I implemented multi-threading for FL server. One thread is created for each client, enabling the FL server to serve clients concurrently and asynchronously. The new approach is visualized in Figure 6 (Right).

Adding new message types to Flower requires the following steps:

- Edit the Interface Definition Language (IDL) file to add definitions for the new message types `ReadIns` and `ReadRes`.
- Run gRPC compiler with Dropbox mypy-protobuf extension to generate code.
- Update Protocol Buffer serialization and de-serialization functions to encode/decode the message properly.
- Add logic to client and server implementation to interpret the message correctly.

4.2.3 Pull model from server

Flower originally supports BSP-FL, and every `FitIns` carries the full model, as shown in Figure 7.

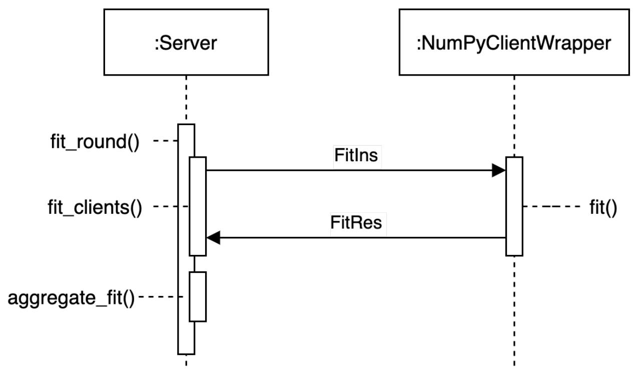


Figure 7: Original Flower implementation

However, in SSP-FL, the clients use models with bounded staleness and do not require fresh model downloading for each training round. Thus, I implemented two new message types, `ReadIns` and `ReadRes`, representing Read Instruction and Read Result respectively, to implement SSP on top of Flower. `ReadIns` is sent from a client to the server when its cached clock is less than its current clock by the staleness bound s. This is visualized in Figure 8.

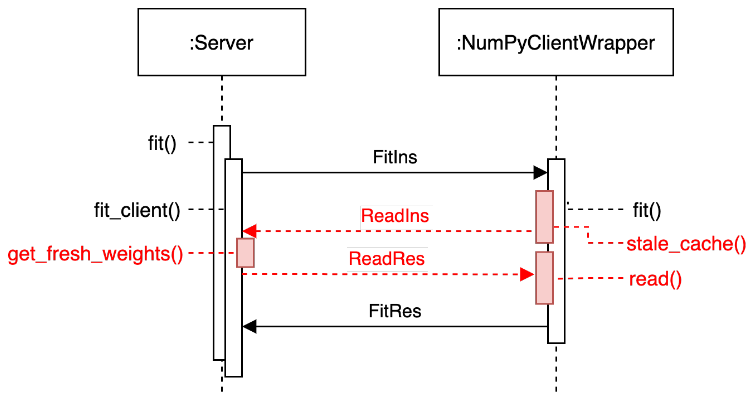


Figure 8: Our implementation

4.2.4 Client Implementation

I implemented `NumPySSPClientWrapper`, adopting from provided `Client`, `NumPyClient` and `NumPyClientWrapper` classes. Compared with `NumPyClientWrapper`, `NumPySSPClientWrapper` accepts takes one additional parameter: `staleness_bound`, which is provided as command line arguments.

In addition, `NumPySSPClientWrapper` defines two additional instance variables: `cache_parameter` (referring to the possibly stale model cached by the client, initialized to `None`) and `cache_lock` (referring to the clock of `cache_parameter`, initialized to `0`).

4.2.5 Server Implementation

In `fit`, the server creates one thread for each registered and sampled client, and runs the `fit_client` method to handle all interactions with that client (as explained in 4.2.2).

4.2.6 SSP execution logic

The complete workflow of our design is shown as follows:

1. The server is created and starts waiting for incoming connections.
2. All clients are created. Each client connects to the server and is registered in the client manager.
3. The server initializes a global model with random weights.
4. The server starts executing the `fit` function, launching threads to run `fit_client` for all clients concurrently.
5. `fit_client` sends a `FitIns` to the client, informing it to start training and the current clock.
6. When receiving a `FitIns`, the client checks if the cache is stale by checking if the clock in `FitIns` is greater than that of the cached model by more than s. If so, the client requests to download the model from the server by sending the `ReadIns` message. In `ReadIns` the client also includes the `min_clock`, representing the expected minimum *global clock*. The `min_clock` is the minimum clock across all clients, maintained by the server. If the global clock does not reach `min_clock`, the client will be blocked until the global clock reaches `min_clock`. `min_clock` is calculated by `current_clock`$-s$. If the cache is fresh, the client will proceed to training in step 8. In the first round, clients notice that they do not hold any local model, so they consider their cache stale and send a `ReadIns`.
7. The server receives `ReadIns` and `min_clock` of the client. The server calls the `get_fresh_weights` function to check if `min_clock` is ahead of global clock by more than s. If so, `get_fresh_weights` is blocked on a conditional variable and waits to be notified by other client-handling threads when the global clock is updated. Otherwise, it sends back its current global model by a `ReadRes` immediately. The client's `read` method will be invoked to read the fresh model.

8. All clients acquire a fresh model before starting the training process. They start training and return the new weights by sending back `FitRes` to the server.

9. When the thread executing `fit_client` receives the `FitRes` message, it updates the global model according to the response. Then it checks the clocks of all the clients to see if the global clock can be updated. If the server advances the global clock, it will notify all client-handling threads that are being blocked on the conditional variable. Then `fit_client`, executing in a loop, sends out new `FitIns` message with an incremented clock. The client handles the instruction as in step 6.

10. After all clients finish their rounds, i.e., all `fit_client` threads complete execution, the server evaluates the model by aggregating the loss from test dataset of clients. Eventually, the server sends shutdown signals to all clients and all processes exit.

Server-side implementation in pseudo code:

```
Algorithm 1: fit
initialize weights by asking one client to return its model;
sample clients;
start one thread per client to run fit_client;
```

```
Algorithm 2: fit_client
client          : ClientProxy that provides client interface for server
num_rounds      : number of rounds to run for the client
staleness_bound: staleness bound

for $i \leftarrow 1$ to num_rounds do
    reply = send FitIns to client;
    if reply is ReadIns then
        weights = get_fresh_weights(reply);
        reply = send ReadRes(weights) to client;
    end
    fit_res = reply;
    update global clock;
    updated_model = update model with fit_res.update;
    loss, acc = evaluate(updated_model);
end
```

Client-side implementation in pseudo-code:

Algorithm 3: `fit`

```
params = self.cached_paramter;
if self._stale_cache(ins.clock) then
    return ReadIns(ins.clock - self.staleness_bound);
end
updated_params, num_datapoints, loss, acc = self.numpy_client.fit(params);
self.cached_parameter = params;
return FitRes(updated_params)
```

Major differences between our implementation and the original Flower implementation:

- **The client is aware of its current round.** In the original implementation of Flower, the clients are stateless: they simply perform training and are unaware of the training process. This is the design decision made by Flower to simplify the structure of the clients. In our implementation, the staleness bound information is required for the client to decide the staleness of its cached model.
- **`FitIns` does not include the global model.** Flower originally implements BSP-FL, so the client gets a copy of the global model at the beginning of every training round in `FitIns`. In our implementation, clients download the global model from the server using `ReadIns` when its cache is stale, so it does not make sense to include the global model in `FitIns`. Weights are included in `ReadRes` only.
- **The server maintains a global clock to track the training progress.** In the original implementation of Flower, all clients are at the same clock because of the BSP-FL setting. In our implementation, a global clock is maintained to track progress of the training process, as the client needs to cache this clock to calculate the difference between local and global clock.

4.3 Adaptive-SSP

In this section, I explain the steps taken to implement Adaptive-SSP, based on our SSP implementation in 4.2, with different strategies 3.3.

4.3.1 Switching point strategy

I define the base class `SPStrategy` (where `SP` stands for "switching-point"), with one instance variable `data`, as an internal list of metrics (loss/accuracy) collected. The `SPStrategy` defines two methods:

- `feed(data)`: appends one more entry to the internal list, where `data` can be any metrics useful for switching-point detection strategy.
- `should_switch()`: returns a boolean value to indicate whether a switching point is detected.

I implemented three switching point strategies based on `SPStrategy`:

- `AccuracyVariance` represents a strategy that detects switching-points by measuring the variance of consecutive model accuracy levels. An `AccuracyVariance` has three parameters: `last_k_data`, `var_threshold`, and `clear_on_switch`. The parameter `last_k_data` specifies how many accuracy levels are considered when the strategy determines whether a switching-point is found. The parameter `var_threshold` specifies the variance threshold: when the variance of `last_k_data` drops below `var_threshold`, the strategy reports a switching point. The parameter `clear_on_switch` specifies whether to clear the list when a switching point is detected. I present the pseudo-code of `should_switch` for `AccuracyVariance`.

  Algorithm 4: `AccuracyVariance::should_switch`

  ```
  if len(self.data) <self.last_k_data then
      return False;
  else
      variance = Variance(self.data[-self.k:]);
      if variance <self.var_threshold then
          if self.clear_on_switch then
              self.data.clear();
          end
          return True;
      end
      return False;
  end
  ```

  Note that `AccuracyVariance` can be easily extended to detect switching-points using loss.

- `ProductSigns` represents a strategy that detects switching-points by monitoring the product signs of consecutive gradients updates. It takes two parameters: `threshold` and `clear_on_switch` and `feed` should accept gradient update as argument. It has an additional instance variable `negative_count` to keep track of the number of negative product signs detected. I present the pseudo-code of `should_switch` for `ProductSigns`.

Algorithm 5: `ProductSigns::should_switch`

```
if len(self.data) < 2 then
    return False;
end
gradient1 = self.data[-1];
gradient2 = self.data[-2];
if InnerProduct(gradient1, gradient2) < 0 then
    self.negative_count += 1;
end
if self.negative_count > self.threshold then
    if self.clear_on_switch then
        self.data.clear();
    end
    return True;
end
return False;
```

4.3.2 Client-driven Adaptive-SSP

Each client varies its staleness bound independently, based on its local training metrics. The algorithm is presented in Algorithm 6.

Algorithm 6: `fit`

```
params = self.cached_paramter;
if self._stale_cache(ins.clock) then
    return ReadIns(ins.clock - self.staleness_bound);
end
updated_params, num_datapoints, loss, acc = self.numpy_client.fit(params);
self.sp_strategy.feed(acc);
if self.sp_strategy.should_switch() AND self.staleness_bound > 1 then
    self.staleness_bound -= 1;
end
self.cached_parameter = params;
return FitRes(updated_params)
```

In this scheme, the server are unaware of the staleness bound. The process is visualized in Figure 9.

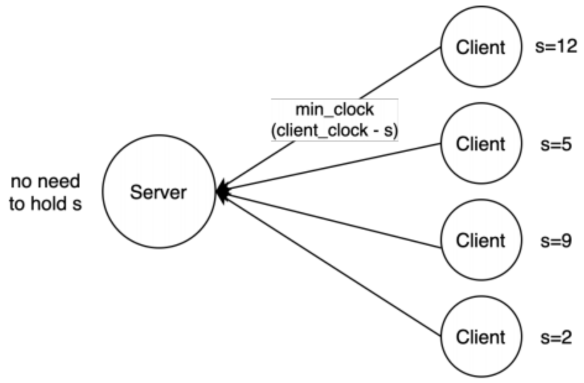


Figure 9: Client-driven Adaptive-SSP

4.3.3 Server-driven Adaptive-SSP

No matter if the server detects the switching-point by evaluating the model on the global test dataset or by aggregating information sent from the clients, the server is responsible for detecting the switching-point, updating the staleness bound, and notify the clients about the change. Thus, a new type of message is needed between the server and clients, `UpdateSIns`, a one-way message to notify the client about the change in staleness bound. The client-side message handler of `UpdateSIns` calls the `update_staleness_bound` method of the client interface to update the client's staleness bound. This process is visualized in Figure 10.

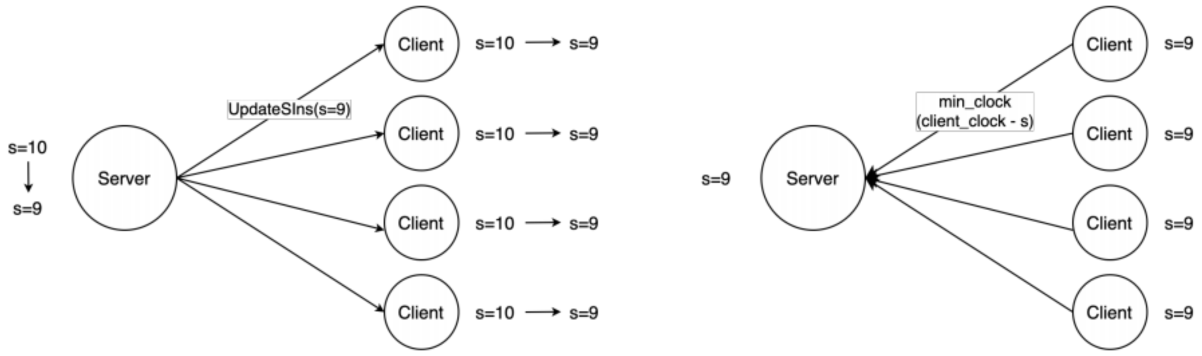


Figure 10: Sever-driven Adaptive-SSP with `UpdateSIns`

Server holds global test dataset. Server evaluates on the global test dataset after receiving each gradient and checks for switching-point. The server-side logic is presented in

Algorithm 7.

```
Algorithm 7: fit_client
client          : ClientProxy that provides client interface for server
num_rounds      : number of rounds to run for the client
staleness_bound: staleness bound

for $i \leftarrow 1$ to num_rounds do
    reply = send FitIns to client;
    if reply is ReadIns then
        weights = get_fresh_weights(reply);
        reply = send ReadRes(weights) to client;
    end
    fit_res = reply;
    update global clock;
    updated_model = update model with fit_res.update;
    loss, acc = evaluate(updated_model);
    self.sp_strategy.feed(acc);
    if self.sp_strategy.should_switch() AND self.staleness_bound >1 then
        self.staleness_bound -= 1;
        client.update_staleness(self.staleness_bound);
    end
end
```

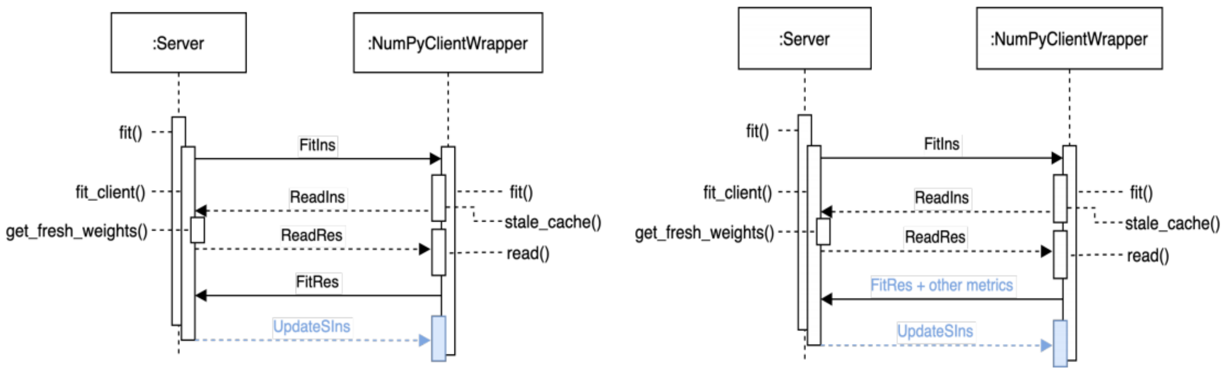
Server does not hold global test dataset. Server cannot evaluate using its dataset and relies on metrics from the client. The server-side logic in presented in Algorithm 8.

Algorithm 8: `fit_client`

```
client          : ClientProxy that provides client interface for server
num_rounds      : number of rounds to run for the client
staleness_bound: staleness bound

for i ← 1 to num_rounds do
    reply = send FitIns to client;
    if reply is ReadIns then
        weights = get_fresh_weights(reply);
        reply = send ReadRes(weights) to client;
    end
    fit_res = reply;
    update global clock;
    updated_model = update model with fit_res.update;
    self.sp_strategy.feed(fit_res.loss, fit_res.acc, fit_res.num_datapoints);
    if self.sp_strategy.should_switch() AND self.staleness_bound >1 then
        self.staleness_bound -= 1;
        client.update_staleness(self.staleness_bound);
    end
end
```

When the server does not hold the global test dataset, clients send back additional metrics in `FitRes` to the server for switching-point detection. This difference is visualized in Figure 11.



(a) Using global dataset (b) Using metrics from client

Figure 11: Difference between two server-driven Adaptive-SSP schemes

5 Experiments and Results

The performance of Adaptive-SSP is empirically evaluated in experiments. This section describes the experiment settings and results.

5.1 Experiment Implementations

This section explains how federated learning environments are simulated to produce realistic results.

5.1.1 Straggler simulation

In the experiment, random sleeps are inserted into client execution path to simulate the presence of stragglers. In each experiment, a configurable fraction of clients is chosen to be stragglers (`straggler_fraction`), with configurable upper-bounds and lower-bounds for sleep duration (`min_delay` and `max_delay`). When a client finishes training, it will sleep for a random duration between `min_delay` and `max_delay` if it is chosen as a straggler. The values of `min_delay` and `max_delay` should be properly set to correlate the normal training time. In the experiments, `min_delay` is set to be 0.5x of training time, and `max_delay` is set to be 1x of the training time.

5.1.2 Execution environment

Instead of running server and client processes together on the same host without isolation, I use the combination of Docker [17] and Kubernetes [18]. Docker containers provide execution environment isolation, and Kubernetes provides cluster management ability and makes it easy to create a large number of containers.

5.1.3 Non-IID dataset

The original implementation does not provide non-IID dataset suitable for federated learning experiments. In our experiment, I use the following two Non-IID dataset:

- **Non-IID CIFAR10 [19].** I manually perform data transformation to simulate the non-IID condition. I performed the non-IID transformation in the following steps: (1) read full dataset from disk; (2) split the full dataset into training set and testing set; (3) break training set into classes based on the label; (4) split training set unevenly into buckets according to classes, so that each bucket has few overlapping classes; (5)

shuffle data within each bucket; (6) perform (4) - (5) for testing set; (7) each client picks a bucket based on its index assigned at creation.

- **LEAF dataset.** LEAF is a benchmarking framework for federated learning and provides non-IID dataset [20]. It provides datasets and models (implemented in Tensorflow) for different categories of machine learning applications, including image classification, sentiment analysis, text generation, etc. In our experiment, I used the Non-IID text dataset of Shakespeare Dialogues.

5.1.4 Machine Learning Applications & Models

I evaluated Adaptive-SSP using two categories of machine learning applications:

- **Image classification.** I implemented image classification application based on the Non-IID CIFAR10 dataset. I experimented on different models:
  - `SimpleCNN` is a simple convolutional neural network (CNN) model.
  - `LeNet` is implemented based on LeNet [21].
  - `ResNet` is implemented based on ResNet [22].
  - `VGG` is implemented based on VGG [23].
- **Text prediction.** I implemented next-character prediction application using Recurrent Neural Network (RNN) [24]. In particular, I implemented:
  - `LSTM` is implemented based on Long-Short-Term-Memory (LSTM) [25].
  - `GRU` is implemented based on Gated Recurrent Unit (GRU) [26].

5.1.5 Accuracy collection

Accuracy levels of the models are used to evaluate the performance of different federated learning schemes. In our experiments, accuracy metrics are collected by the server, evaluated using the global test dataset. As explained in 3.3.1, it is unrealistic to assume that the server will hold the global dataset in actual federated setting. However, it is acceptable to make the server evaluate the model using global dataset only for the purpose of metrics collection.

The accuracy collection process should impose minimum effect on the FL scheme itself. Throughout experimenting, I explored three different approaches to evaluate the global model:

- **Evaluate global model only when `min_clock` is updated at server.** In a FL experiment with n rounds, the server will evaluate the global model n times. The advantage of this approach is that this imposes little effect on the FL scheme. However, the problem with this approach also comes from its low evaluation latency. Consider an SSP-FL experiment with large s and severe stragglers. In this case, `min_clock` is not updated from 0 to 1 until it accumulates many updates from non-stragglers but only one update from the slowest straggler. Also, towards the end of training when all non-stragglers have finished training, the model at clock $t+1$ only integrates few updates from stragglers, compared with model at t. Such cases make this evaluation scheme undesirable and not a good indication of the training progress.

- **Evaluate global model whenever receiving `FitRes` from client.** In a FL experiment with n rounds and m clients, the server will evaluate the global model $m \times n$ times. The advantage of such high-frequency evaluation is that it closely measures the model updates throughout the training process, and avoids the problem in the previous approach. However, the obvious limitation of this approach is that such frequent evaluation hinders the normal execution flow of the FL scheme. From our experiment, such frequent evaluations drastically slow down the training process, and is therefore undesirable.
- **Evaluate global model periodically and at `min_clock` update.** As an optimization of the previous two approaches, this approach aims to monitor the model accuracy without significantly affect the execution flow of FL-SSP. At creation, the server accepts an argument of `eval_interval` representing the evaluation interval. The server maintains a `prev_eval` variable as the timestamp of previous evaluation. The server's behavior under two cases are listed as follows:
  - When server receives an update and does not update global clock, it checks `pre_eval` to see if it has evaluated the model within past `eval_interval`. If so, it updates `prev_eval` to be current timestamp and evaluates the model. Otherwise, the server does not evaluate the model.
  - When server receives an update and update global clock, it updates `prev_eval` to be current timestamp and evaluates the model.

5.1.6 Automated experiments

In this project, I continuously revised our Adaptive-SSP implementations and evaluated different strategies. Thus, automated experiments with configurable parameters are especially important to improve efficiency and reduce manual efforts. I fully automate the experiments in the following ways:

- `updateImage.sh` rebuilds the Docker image and upload to local registry;
- `deployCluster.sh` removes any existing deployment and redeploys Docker containers on the Kubernetes cluster;

- `one_test.sh` obtains pods from the current deployment, and use `kubectl exec` command to start the server and clients;
- `collect_log.sh` collects log output from different contains for evaluation purpose;
- `many_tests.py` combines all scripts above to run different experiments automatically. It takes the following arguments:

Argument name	Meaning
`--containers`	total number of contains, including server and clients
`--staleness`	staleness bound
`--rounds`	total number of rounds
`--max_delay`	upper bound of random delay
`--min_delay`	lower bound of random delay
`--script`	name of script to run one experiment
`--name`	naming for collected logs

5.2 Experiment Results

I conducted the following experiments to evaluate the performance of Adaptive-SSP.

Model	Staleness	Containers	Rounds	Dataset	Straggler Ratio	Random Delay
GRU	1, 2, 4, 8, 16, 25, *sd*, *cd*	32	30	LEAF Shakespeare	0.5	[100, 200]
LSTM	1, 2, 4, 8, 16, 25, *sd*, *cd*	32	30	LEAF Shakespeare	0.5	[100, 200]
VGG-11	1, 2, 5, 10, *sd*, *cd*	8	50	Non-IID CIFAR10	0.5	[40, 80]
LeNet	1, 2, 5, 10, *sd*, *cd*	64	50	Non-IID CIFAR10	0.5	[40, 80]

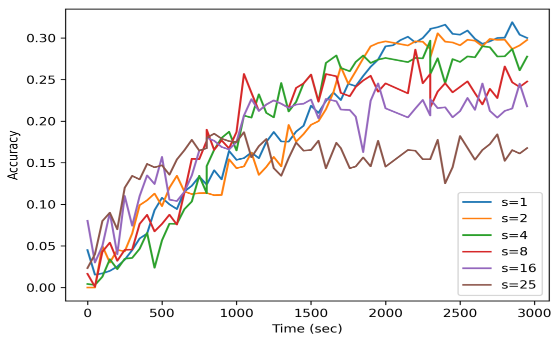
Table 1: Evaluation settings

In the table above, *sd* means server-driven Adaptive-SSP, and *cd* means client-driven Adaptive-SSP. All experiments are conducted using `AccuracyVariance`. For VGG-11, only 8 containers are used as it reaches the memory limit of the Kubernetes cluster. For more light-weighted models like LeNet, LSTM and GRU, I use a larger number of clients to simulate federated learning. For different models, I use different random delays comparable to 1x training clock time to simulate stragglers.

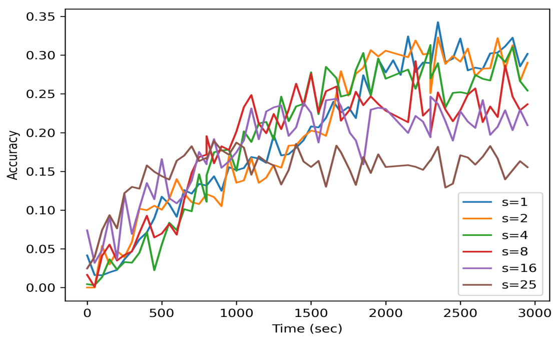
5.2.1 Baselines

As baselines, I evaluated SSP-FL with different staleness bounds, for VGG-11, LeNet, LSTM and GRU. The result is shown in Figure 12.

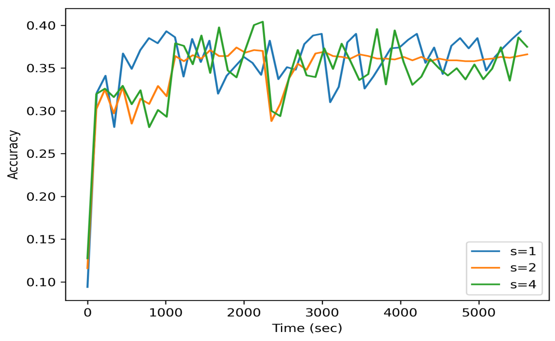
From Figure 12a and 12b, I can observe the following properties:



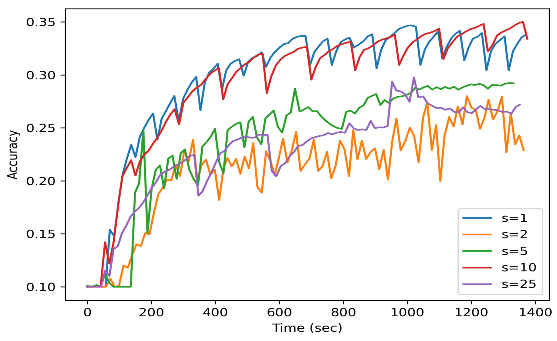
(a) LSTM Baseline



(b) GRU Baseline



(c) VGG Baseline



(d) LeNet Baseline

Figure 12: SSP Baselines

- With larger s, the model converges in a shorter time, with lower accuracy;
- With smaller s, the model converges in a longer time, with higher accuracy.
- For two SSP-FLs with different staleness bounds s_1 and s_2 ($s_1 < s_2$) that can both achieve accuracy level β, SSP-FL with staleness bound s_1 reaches β earlier than s_2.

This identifies with previous discussion in 3 about the value of staleness bound s and the performance of SSP-FL.

However, in the experiment of VGG-11 and LeNet, the result does not strictly conform to our expectations. I analyze the possible reasons in the remaining section.

In Figure 12c, the result does not expose any impacts by the varying value of staleness bound, and the model converges mostly quickly when $s = 1$. One possible reason is that VGG-11 model is more complex than other models (LeNet, LSTM, or GRU) and achieves high accuracy within the first round of training. Thus, the impact by different staleness bounds is diluted in this case.

In Figure 12d, the result also seems contradictory with our expectations. A larger s reaches a higher accuracy than smaller s. Also, a smaller s converges faster than a larger s. I investigated this abnormal case by monitoring the accuracy changes of both global model at the server and the local model cached at each client. I discovered that this case is caused mainly by Non-IID data partitioning. Though I split the dataset in Non-IID way, a very small fraction of clients (1 or 2 in our case) is assigned a subset of data representative of the global dataset. It also happened that these small number of clients were not chosen to be stragglers in our experiment framework. I refer to them as ”lucky clients”. The outcome is that, with a large staleness bound, these lucky clients pushed a large number of gradient updates to the global model at the early stage of training, and the global model quickly reaches high accuracy. However, this situation does not appear with smaller staleness bounds. Why this only happen to LeNet but not VGG-11? I believe this anomaly is not caused by the difference in model, but rather the Non-IID data partitioning process: I used 8 clients for VGG-11, but 64 clients for LeNet.

5.2.2 Adaptive-SSP

On the basis of baselines, I evaluated Adaptive-SSP on VGG-11, LeNet, LSTM and GRU, with both server-driven strategy and client-driven strategy. The staleness bound is initialized to be `num_total_rounds / 2`. The result is shown in Figure 13.

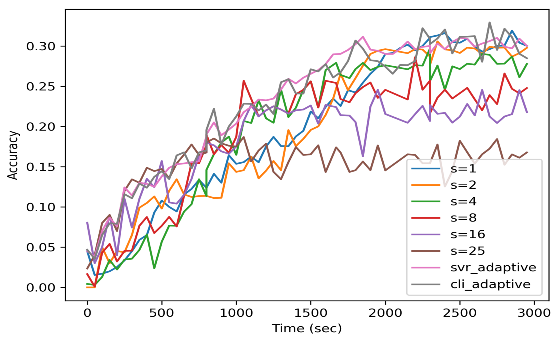
From Figure 13a and 13b, I make the following observations:

- Adaptive-SSP reaches accuracy comparable to SSP-FL with $s = 1$ and $s = 2$ (around 0.3), but converges more quickly (In LSTM example, SSL-FL with $s = 1$ converges at $t = 2400$, while `svr_adaptive` converges at $t = 2000$ and `cli_adaptive` converges at $t = 2300$).
- For each accuracy level β, Adaptive-SSP reaches it no later than any SSP-FL with fixed staleness bound. Visually speaking, Adaptive-SSP FL appears as the left-most curve at each accuracy level in the graph.

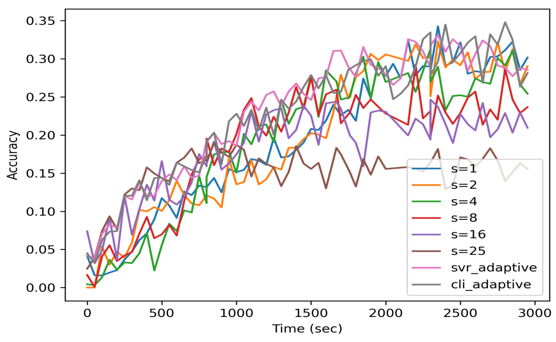
This identifies with our expectation: by starting with a large staleness bound and continuously decreasing its value at model convergence, Adaptive-SSP achieves accuracy comparable to SSP-FL with small staleness bound, but with a shorter time.

For Figure 13c, as explained in 5.2.1, I cannot clearly observe the impact of different staleness bounds, thus the impact of Adaptive-SSP is also not observable.

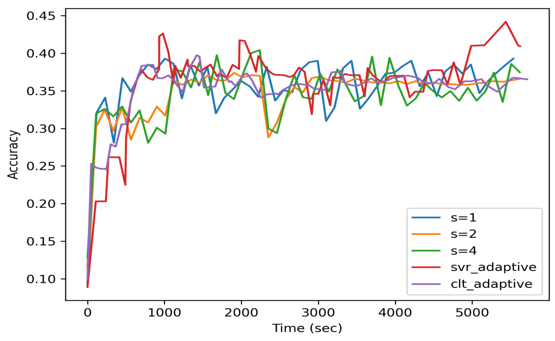
For Figure 13d, despite the anomalies analyzed in 5.2.1, I observe that server-driven Adaptive-SSP achieves comparable accuracy level as SSP-FL with small staleness bounds, but with converge earlier. It can be clearly observed that server-driven Adaptive-SSP and client-driven Adaptive-SSP starts out with similar progress, and start to converge at round $t = 400$. However, it is interesting to know why the two Adaptive-SSP schemes diverges after that,



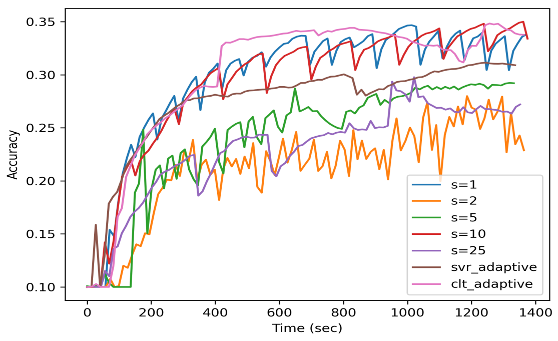
(a) LSTM Adaptive-SSP



(b) GRU Adaptive-SSP



(c) VGG Adaptive-SSP



(d) LeNet Adaptive-SSP

Figure 13: Adaptive-SSP results

and client-driven Adaptive-SSP out-performs server-driven Adaptive-SSP. This is a problem worthy of further investigation.

6 Conclusion and Future Works

This report investigated the problem of federated learning. In FL, SSP-FL is widely used as a generalization of BSP-FL and ASP-FL. The performance of SSP-FL primarily depends on the value of staleness bound. This project investigates the performace of SSP-FL with respect to the value of staless bound, and makes the following observation: when s is large, SSP-FL converges quickly to a lower accuracy level; when s is small, SSP-FL converges less quickly but achieves higher accuracy. In this project, I design and implement the Adaptive-SSP strategy as an extension to SSP-FL. Adaptive-SSP is similar to SSP-FL, but supports varying staleness bound during the training. It starts the training with a relatively large staleness bound, and continuously detects switching-points to decrease the value of staleness bound. I evaluated the performance of Adaptive-SSP by experiments. The experiment result demonstrates the effectiveness of Adaptive-SSP of achieving quick convergence and high accuracy simultaneously, though not in all test cases. The experiment result is properly analyzed.

More works can be done in the future on Adaptive-SSP:

More ML applications. Currently, the experiment only evaluates the performance of Adaptive-SSP on two ML applications (image classification, text prediction) and four models (VGG-11, LeNet, LSTM, GRU). More experiments can be performed to evaluate the effectiveness of Adpative-SSP under different use cases, like voice recognition, behavior prediction, etc.

The comparison between client-driven and server-driven Adaptive-SSP. In 5.2.2 I observe that server-driven Adaptive-SSP outperforms client-driven Adaptive-SSP in LSTM and GRU, while client-driven Adaptive-SSP greatly outperforms server-driven Adaptive-SSP in LeNet. The relationship between the two strategies remains to be investigated.

The combination of client-driven and server-driven Adaptive-SSP. I implemented and evaluated the performance of client-driven and server-driven Adaptive-SSP separately in this project. An interesting problem is the feasibility of the combination of two strategies. This can be implemented and tested in the future.

References

[1] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2017.

[2] Mu Li. Scaling distributed machine learning with the parameter server. In *Proceedings of the 2014 International Conference on Big Data Science and Computing*, BigDataScience '14, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks, 2020.

[4] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization, 2019.

[5] Ming Chen, Bingcheng Mao, and Tianyi Ma. Efficient and robust asynchronous federated learning with stragglers. In *Submitted to International Conference on Learning Representations*, 2019.

[6] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. Dynamic stale synchronous parallel distributed training for deep learning, 2019.

[7] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. Flexps: Flexible parallelism control in parameter server architecture. *Proceedings of the VLDB Endowment*, 11(5), 2018.

[8] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[9] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 463–478, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems*, 26:1223–1231, 2013.

[11] Hanpeng Hu, Dan Wang, and Chuan Wu. Distributed machine learning through heterogeneous edge systems, 2019.

[12] Georg Ch Pflug. Non-asymptotic confidence bounds for stochastic approximation algorithms with constant step size. *Monatshefte für Mathematik*, 110(3):297–314, 1990.

[13] Serge Kas Hanna, Rawad Bitar, Parimal Parag, Venkat Dasari, and Salim El Rouayheb. Adaptive distributed stochastic gradient descent for minimizing delay in the presence of stragglers, 2020.

[14] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. Flower: A friendly federated learning research framework, 2020.

[15] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[17] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[18] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[19] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[20] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H. Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings, 2019.

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[24] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, Mar 2020.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[26] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.