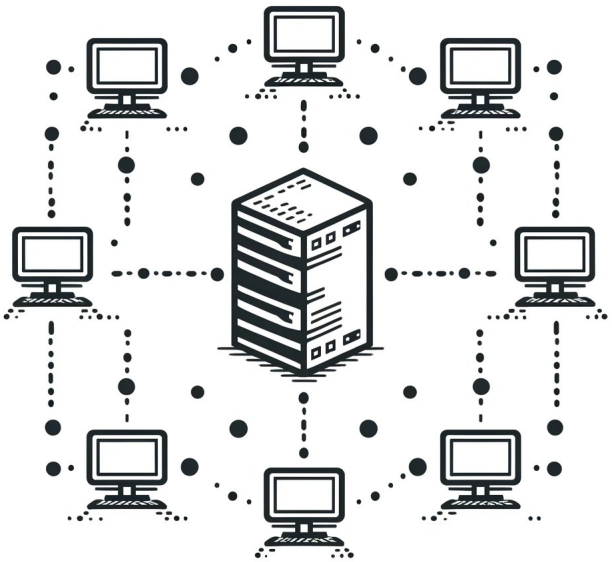


**Autonomous Agents Project**

# F.L.Ex

# Federated Learning EXchange

**A C++ and Python Framework for Federated Learning**

**Ioannis Christofilogiannis 2019030140**

**Goal:**

Train Machine Learning models on-device and securely share their parameters with similar models from other devices to attain better performance, without needing to share sensitive user data.

**Instructions:**

**Install Python requirements:**

```
pip3 install -r requirements.txt
```

Tested on Python 3.11.7, on Ubuntu 22.04 x64 through WSL and on macOS arm64 on this IP and port number combination.

**Compile:**

```
python3 setup.py build_ext --inplace
```

**Run Server:**

```
python3 server.py --p 8080 --c 3 --i 0
```

**Run Clients:**

```
python3 client.py --ip 127.0.0.1 --p 8080
```

For each client instance (on the default config 3 clients are used) Alternatively client2 and client3 files can be used as they are clones of the original file and were created for easier setup.

**How it works:**

In the beginning key exchange happens (more info on Security section) and each client trains a model with their local dataset, also making an initial performance evaluation.

After that, in each client Python notifies the C++ script to send the parameters, which are received by the server which is already waiting for new parameters, to then use the aggregation functions to generate the new params, which contain the knowledge of different datasets because of the nature of Federated Learning.

The aggregated data is returned to the client, which uses a partial fit method to update its model (training with the same dataset with updated theta and variance).

Then this whole process is repeated until convergence, which we decided to base on the f1 score metric and stop when it doesn't change over a certain threshold.

**Reasons for using C++ and Python:**

Python socket programming is known for unreliability, less control by the programmer and worse performance . A language like C, C++ is better suited in order to better control the connection and data transfer process. This is problematic because a language like Python is better suited for training Machine Learning models, so the question is raised: Can we combine the benefits of C++ and Python?

**Python and C++ integration using Cython:**

Cython is a Python module that answers the previous question with a resounding yes.

Using .pyx wrappers it enables Python to interface with C++ with intermediate methods are defined with the Cython syntax resulting in C++ methods being able to be called on Python. For example:

**C++ method:**

```
void participate() {
     std::thread clientThread(&FL_Client::participateThread, this);
     clientThread.detach();
}
```

**Cython Wrapper (Interfacing C++ pointers/methods from Python):**

```
cdef class py_fl_client:
    cdef FL_Client* _client

def participate(self):
    self._client.participate()
```

**Python code (simplified):**

```
def main(ip_address="127.0.0.1", portnum=8080):
    client = py_fl_client(ip_address, portnum,...)
    client.participate()
```

Finally setup.py works like a traditional Makefile does, giving instructions to gcc/clang or any other C++ compiler and integrating the wrappers in the project in a way that the required code is generated and compiled.

In this project Python acts as a data creator, machine learning model trainer and a conductor of all the corresponding operations(notifying the C++ condition variables to wake up), while C++ acts as the network communication manager, handling multiple different socket connections and threads and using condition variables, mutexes and barriers.

**Security:**

Federated Learning is by nature more secure than sharing datasets which may contain sensitive data. However, there are still challenges because data leakage

All the data shared between the server and the clients is end to end encrypted. This is important in order to protect the data from malicious clients or outside attacks. Both symmetric (AES) and asymmetric (RSA) encryption is used accordingly. In the beginning, the server and client exchange RSA public keys with each other.

The server then generates a symmetric key for AES which is encrypted with the server's private key and sent, to be decrypted with the server's public key by the client. After that, every piece of data sent is encrypted with the shared key using AES, also generating an IV which is necessary for the decryption and sent alongside the encrypted data.

The Python Cryptography library was used, which is a robust, secure and simple solution for this problem.

**Experiment:**

This framework has been tested on a diabetes indicators dataset from Kaggle. To demonstrate a realistic use case we split the dataset into smaller ones with ... (3 in this case) in order to simulate a Horizontal Federated Learning setting (for example 3 hospitals in different cities with the same features).

Then a feature selection script that uses Random Forest to focus only on the most relevant features was used. After that each client performed the required training with a different part of the split dataset and Federated Learning was performed until F1 score convergence.

In the results we can see that the performance is averaged between the clients, seeing a more substantial improvements on the ones with the worst starting performance.

Before:

```
Accuracy: 0.75358
Recall: 0.7535871
F1 Score: 0.77416
```

After:

```
Accuracy: 0.807855
Recall: 0.80785504
F1 Score: 0.801486
```

Feature selection Results:

```
Using Random Forest Classifier,
we can sort the features of the dataset by importance:
BMI                   0.182723
Age                   0.122260
Income                0.098446
PhysHlth              0.083586
Education             0.071008
GenHlth               0.068053
MentHlth              0.064441
HighBP                0.042650
Fruits                0.034015
Smoker                0.033774
Sex                   0.028884
Veggies               0.026829
```

Final features were selected based on the random forest and some intuition

Federated Learning Results (Left: Before FL, right after key exchange, Right: After 5 training rounds of FL):

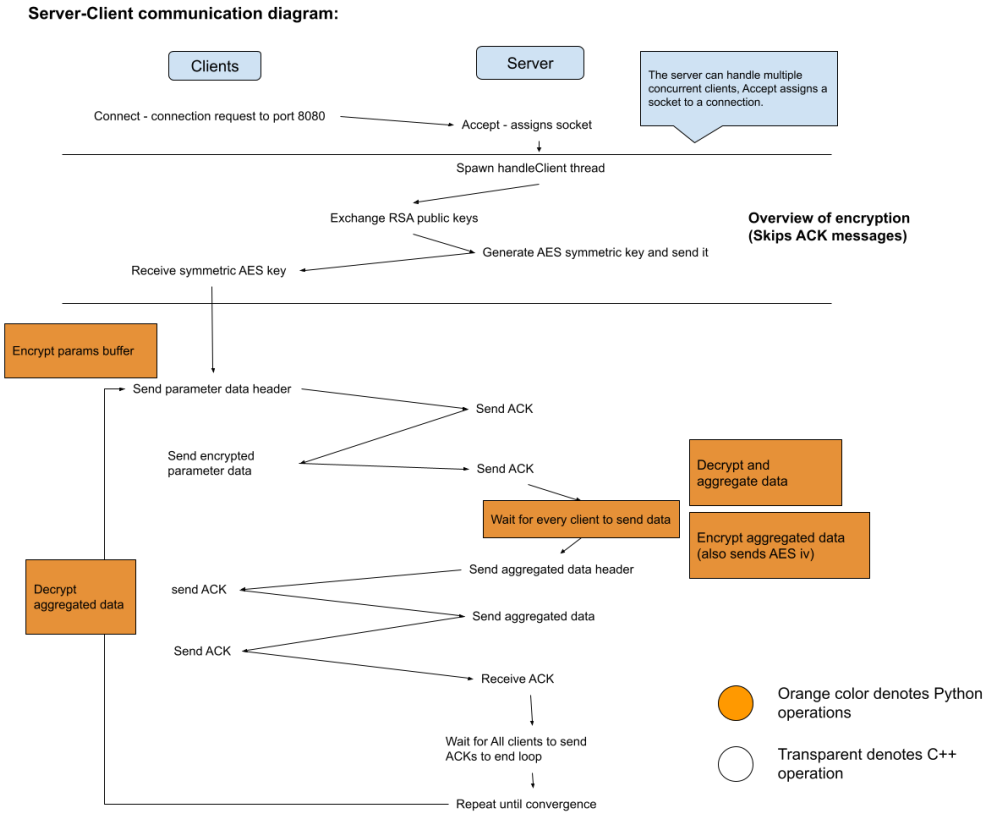
```
Client pbuffer size: 4
Header Sent: MD5:75be2fbcb73cbf391d8bbbdce2ab47c9;ID:00001;SIZE:0000000004
Received header ACK from server, sending file:
Received file ACK from server, File sent successfully.
Aggregated data header: MD5:a581fb46b5c6ab844c822a3fd8a9a698;ID:00001;SIZE:0000000256
Received aggregated data from server, sending ACK.
MD5: d768e668cc7cc5b3f4038146d85d095a
Performance metrics on beginning.
-Recall: 0.8019552191737622
-F1 Score: 0.7983132890803669
Size of params in bytes: 738
MD5: 263a85a86e12ef2ac6e23e30e275cbbe
```

```
Client pbuffer size: 1088
Header Sent: MD5:96effd772761383a1dde15bcc44eaf94;ID:00001;SIZE:0000001088
Received header ACK from server, sending file:
Received file ACK from server, File sent successfully.
Aggregated data header: MD5:16ccfbb600a8be11d7cfb20048c8a189;ID:00001;SIZE:0000001024
Received aggregated data from server, sending ACK.
MD5: 95c6543723ff7115303db021b1036a8a
Round 3: Decision to stop loop:False
Decision to stop: True
-Recall: 0.804478082623778
-F1 Score: 0.8005505551578869
```

## Limitations:

Currently the framework only supports parameter sharing and aggregation of Gaussian Naive Bayes classifiers, although the network communication part is created with the scope of sharing any kind of data (encapsulated in a params dictionary). Any ML model with partial_fit support will be compatible with little to no work and other models could easily be used with modifications.

## Diagram:



**Future work:**

There is an abundance of possible future work on the project.

Firstly changing the convergence condition to stop on a different threshold or performance metric (instead of f1) will drastically change the results.

Using alternative aggregation algorithms on the server, or focusing the learning on the most important features would be a logical first step. After that, testing different models like Neural Networks instead of only using GNB.

This project will continue to grow in the duration of my thesis project.

**Credits:**

This project is part of my diploma thesis which is supervised by professor Ioannidis. Due to sharing a similar subject with Georgios Valavanis this project uses some of his functions, noted in the code (parsing arguments from command line, evaluation model). He also contributed on using the params buffer to send RSA keys and the initial weighted aggregation.

**Sources:**


- GeeksForGeeks TCP server-client implementation
- GeeksForGeeks c++ socket programming
- GeeksForGeeks c++ barrier
- GeeksForGeeks socket programming handling multiple clients
- GeeksForGeeks condition variables
- Jacob Sorber Multithreaded server in C++
- Scikit learn GaussianNB
- Cython documentation - user guide
- Python Cryptography RSA (Assymetric Encryption)
- Python Cryptography AES (Symmetric Encryption)
- ChatGPT for a basic feature selection method using Random Forest

# Server (Python):

## Modules and Libraries

- **Standard Libraries:** `hashlib`, `datetime`, `os`, `sys` for utility functions; `threading`, `queue` for concurrent operations; `dill` for object serialization; `argparse` for command-line argument parsing.
- **Cython Imports:** `FL_cpp_server.py_fl_server`, `PyServerBuffer`, `PyBufferManager` to handle server functionalities and buffer management.
- **Numerical Operations:** `numpy` for numerical computations.
- **Utility Scripts:** `helper` contains utility functions for cryptographic operations and data handling.

## Constants

- None explicitly defined, but server settings such as port numbers and number of clients are configured through command-line arguments or function parameters.

## Global Variables

- **Buffer Management:** Handles data sent to and from clients.
- **Security Keys:** Public and private keys for RSA, and symmetric keys for AES encryption.
- **Performance Tracking:** Variables to track scores from clients to determine the convergence of the learning process.

## Main Functions

- **`read_params`**:
  - Retrieves and decrypts parameters sent by clients.
  - Can optionally read encryption keys if specified.
- **`aggregate_model`**:
  - Aggregates parameters received from multiple clients.
  - Calculates the new average performance and determines if the learning process should continue based on a threshold comparison.
- **`send_public_key`** and **`send_symmetric_keys`**:
  - Distributes the server's public key and symmetric keys to clients, ensuring subsequent communications are encrypted.
- **`checkScoreDiff`**:
  - Compares the difference between current and previous performance scores to decide if further iterations are needed.

## Workflow Functions

- **`main`**:
  - Initializes and configures the server.

- Manages the key exchange and data aggregation in a loop until the performance criterion stops the rounds or the maximum number of iterations is reached.

### Command-Line Interface (CLI)

- Configures the server through command-line arguments such as port number, number of clients, and number of iterations.

---

# Client (Python):

### Modules and Libraries

- **Standard Libraries:** `time`, `io.BytesIO` for stream operations.
- **Cython Imports:** `FL_cpp_client.py_fl_client`, `PyClientBuffer` for handling low-level operations and buffer management with Cython.
- **Machine Learning Libraries:** `sklearn` for model training and evaluation, `joblib` for model serialization.
- **Utility Scripts:** `helper` contains utility functions for operations like checksum calculation and encryption.

### Constants

- `DATA_PATH`: Path to the dataset directory.

### Global Variables

- Model instance and training data variables.
- Buffers for managing data aggregation and parameter sharing.
- Keys for encryption and decryption.
- Control variables for the learning rounds.

### Main Functions

- **`load_data`**:
  - Loads data from a CSV file.
  - Splits the data into training and test datasets.
- **`train_model`**:
  - Trains the Gaussian Naive Bayes model on the provided dataset.
  - Returns the training score (F1 score).
- **`readParams`** and **`readSymmetricKeys`**:
  - Reads and decrypts parameters from the aggregation buffer.
  - Handles secure transmission of symmetric keys and parameters.
- **`updateModel`**:
  - Updates the model parameters from the received values and recalculates its performance.
- **`writeParamsBuffer`** and **`writeKeyToBuffer`**:
  - Serializes and writes new parameters or keys to the buffer.

- `getModelPerformance`:
  - Evaluates the model's performance on the test set using metrics like recall and F1 score.
- `main`:
  - Initializes the client with server connection details.
  - Manages the flow of model training, key exchange, and parameter updates in a federated learning cycle.
- **CLI Handling**:
  - Parses command-line arguments for IP address, port number, and dataset ID.

---

# Helper (Python):

### Modules and Libraries

- **Standard Libraries:** `hashlib`, `datetime`, `os`, `sys`, `time` for general programming utilities; `threading` for multithreaded operations; `io.BytesIO` for in-memory binary streams; `base64` for encoding binary data.
- **Data Handling Libraries:** `numpy` and `pandas` for data manipulation.
- **Cryptography Libraries:** `cryptography.hazmat` for cryptographic functions including RSA and AES encryption/decryption, key generation, and serialization.

### Helper Functions

- `get_md5_checksum`:
  - Computes the MD5 checksum for the contents of a specified file.
- `get_md5_checksum_from_bytesio`:
  - Calculates the MD5 checksum for data stored in a `BytesIO` object, ensuring data integrity.
- `parse_header`:
  - Extracts and decodes metadata from a received header, typically used to store and verify data transmission details such as MD5 checksums and machine identifiers.
- `generate_rsa_keys`:
  - Generates a pair of RSA private and public keys used for asymmetric encryption, with best practices for key size and public exponent.
- `generate_aes_key`:
  - Creates a 256-bit symmetric key for AES encryption, providing strong security for data encryption.
- `encrypt_message_rsa` and `decrypt_message_rsa`:
  - Encrypts and decrypts messages using RSA public and private keys respectively, employing OAEP padding and SHA-256 for enhanced security.
- `encrypt_and_prepare` and `receive_and_decrypt`:
  - Functions for AES encryption and decryption that handle initialization vector (IV) generation, data padding, and base64 encoding/decoding to facilitate secure data storage and transmission.

# FL Server-Client Module (C++)

## Net_lib - Common Helper Methods and Constants

Constants:

- `const char* ackMessage = "ACK";`
  - **Description**: A predefined acknowledgment message used in server-client communication.

Methods:

`int stringToInt(const std::string& str)`

- **Purpose**: Parses an integer from a String
- **Parameters**:
  - `str` : Reference to the string
- **Returns**: The parsed integer

`std::string setupHeader(const std::string md5_checksum, int machine_id, int64_t file_size)`

- **Purpose**: Constructs a header string with MD5 checksum, machine ID, and file size.
- **Parameters**:
  - `md5_checksum` : The MD5 checksum of the file.
  - `machine_id` : The identifier for the machine.
  - `file_size` : The size of the file in bytes.
- **Returns**: A formatted header string.

`void parseHeader(const std::string& header, std::string& receivedMD5, std::string& machineID, int64_t& fileSize)`

- **Purpose**: Parses the header string into individual components.
- **Parameters**:
  - `header` : The received header string.
  - `receivedMD5` : Extracted MD5 checksum.
  - `machineID` : Extracted machine ID.
  - `fileSize` : Extracted file size.

`void printError(int sock, const std::string& errorMessage)`

- **Purpose**: Prints an error message and closes the socket.
- **Parameters**:
  - `sock` : The socket descriptor.
  - `errorMessage` : The error message to be printed.

`std::string sanitizeMID(const std::string& input)`

- **Purpose**: Removes newline characters from the machine ID.
- **Parameters**:
  - `input` : The machine ID string.
- **Returns**: The sanitized machine ID.

`int getFileSize(const std::string filename)`

- **Purpose**: Determines the size of a file.
- **Parameters**:
  - `filename`: The name of the file.
- **Returns**: The size of the file in bytes.

`void receiveFileSock(int socket, std::string filename, int64_t fileSize)`

- **Purpose**: Receives a file from a socket and writes it to disk.
- **Parameters**:
  - `socket`: The socket descriptor.
  - `filename`: The name of the file to write to.
  - `fileSize`: The expected size of the file.

`void sendFileSock(int socket, std::string filename, int64_t fileSize)`

- **Purpose**: Reads file from disk and sends it over a socket.
- **Parameters**:
  - `socket`: The socket descriptor.
  - `filename`: The name of the file to send.
  - `fileSize`: The size of the file.

`void sendBufferSock(int socket, char* dataBuffer, int64_t bufferSize)`

- **Purpose**: Sends data from a buffer over a socket.
- **Parameters**:
  - `socket`: The socket descriptor.
  - `dataBuffer`: The data buffer to send.
  - `bufferSize`: The size of the buffer.

`void receiveBufferSock(int socket, char* dataBuffer, int64_t bufferSize)`

- **Purpose**: Receives data from a socket into a buffer.
- **Parameters**:
  - `socket`: The socket descriptor.
  - `dataBuffer`: The buffer to store the received data.
  - `bufferSize`: The size of the buffer.

`std::string receiveHeaderSTR(int sock)`

- **Purpose**: Receives a header string from a socket.
- **Parameters**:
  - `sock`: The socket descriptor.
- **Returns**: The received header string.

## Shared Buffer ( Shared Buffer class in shared_buffer.h, shared_buffer.cpp )

Methods:

`void write(const char* data, size_t length)`

- **Purpose**: Appends data to the end of the internal buffer.
- **Parameters**:
  - `data`: Pointer to the data to be written.
  - `length`: The number of bytes to write.
- **Behavior**: Method to append `length` bytes from `data` to the buffer.

`void read(char* output, size_t length, size_t offset)`

- **Purpose**: Reads data from the buffer into the provided output array.
- **Parameters**:
  - `output`: Pointer to the output buffer where the data will be copied.
  - `length`: Number of bytes to read.
  - `offset`: Starting position in the buffer from where to begin reading.
- **Behavior**: Method to copy `length` bytes from the buffer to `output`, starting at `offset`. Prints an error if the read operation exceeds buffer bounds.

`size_t size() const`

- **Purpose**: Returns the current size of the buffer.
- **Returns**: The number of bytes currently stored in the buffer.
- **Behavior**: Method to obtain the buffer's size.

`char* getBufferPtr()`

- **Purpose**: Provides direct access to the internal buffer.
- **Returns**: A pointer to the beginning of the internal buffer.
- **Behavior**: Method to obtain a raw pointer to the buffer data.

`void clear()`

- **Purpose**: Clears the buffer, removing all data.
- **Behavior**: Method to erase all contents of the buffer, resetting its size to zero.

`void setComplete()`

- **Purpose**: Sets writing complete condition to true, signifying that the data is ready to be accessed
- **Behavior**: Method to set a bool flag to true

`void checkComplete()`

- **Purpose**: Checks writing complete condition to see if the data is ready to be accessed
- **Behavior**: Read a boolean

`void resetComplete()`

- **Purpose**: Resets writing complete condition to false, signifying that the buffer has not yet received the expected data
- **Behavior**: Method to set a bool flag to false

`void setMD5()`

- **Purpose**: Stores MD5 checksum of written data to a public variable

- **Behavior**: Method to set the MD5 checksum buffer variable
- **Parameters**:
  - `data` : The given md5 checksum

## Buffer Manager ( Buffer Manager class in shared_buffer.h, shared_buffer.cpp )

Manages SharedBuffer objects, used for server side data storage

Constructor: `BufferManager()`

- **Purpose**: Initializes a new instance of the `BufferManager` class.
- **Behavior**: Sets up the internal structures required to manage multiple `SharedBuffer` objects.

Methods:

`void createBuffers(int count)`

- **Purpose**: Allocates and initializes a specified number of `SharedBuffer` objects.
- **Parameters**:
  - `count` : The number of `SharedBuffer` instances to create.
- **Behavior**: Dynamically creates `count` instances of `SharedBuffer` and stores them for later access.

`SharedBuffer* getBuffer(int index)`

- **Purpose**: Retrieves a pointer to a `SharedBuffer` object at a specified index.
- **Parameters**:
  - `index` : The index of the buffer to retrieve.
- **Returns**: A pointer to the `SharedBuffer` object at the specified index, or `nullptr` if the index is out of bounds.
- **Behavior**: Provides access to a specific `SharedBuffer` managed by the `BufferManager`, ensuring thread safety during access.

`size_t size() const`

- **Purpose**: Returns the number of `SharedBuffer` objects managed by the `BufferManager`.
- **Returns**: The total count of `SharedBuffer` instances currently managed.
- **Behavior**: Allows querying the number of buffers currently available without modifying any internal state.

Destructor: `~BufferManager()`

- **Purpose**: Cleans up allocated resources upon the destruction of the `BufferManager` instance.
- **Behavior**: Iterates through all managed `SharedBuffer` objects, safely deletes them, and clears the internal container to prevent memory leaks.

## Barrier (class in barrier.h, barrier.cpp )

A class that emulates c++20 barrier for c++11, used in the server for thread synchronisation. Specifically, the barrier awaits for all the client connections (handleClient thread) to reach a specific point, in order to either start the aggregation, when all client data is received or reset the aggregation condition for the training loop to restart.

Constructor: `Barrier(std::size_t count, std::function<void()> completionFunction)`

- **Purpose**: Initializes a new instance of the `Barrier` class.
- **Behavior**: Sets up a barrier,  meaning a specific point in which the threads wait until a certain number of them arrive.
- **Parameters**:
  - `count` : The number of threads required for the barrier to continue.
  - `completionFunction` : A function given as void pointer argument, that runs after the barrier is reached by `count` number of threads, before continuing.

Methods:

`void wait(int count)`

- **Purpose**: Waits for a certain number of threads, calls completion function once, continues code execution
- **Parameters**:
  - `count` : The number of threads required for the barrier to continue.
- **Behavior**: Use a condition variable on the generation variable to proceed only when generation increases, meaning that the barrier has been passed.

## Server (`FL_Server` class in `server.cpp`)

Constructor: `FL_Server(int port, int numOfClients, BufferManager* agg_bm, BufferManager* clientBufferManager)`

- **Purpose**: Initializes the FL server with specified port, number of clients, aggregation buffer, and client buffer manager.
- **Parameters**:
  - `port` : Port number for the server to listen on.
  - `numOfClients` : Maximum number of clients the server will handle.
  - `agg_bm` : Pointer to a `BufferManager` for aggregated data buffer manager.
  - `clientBufferManager` : Pointer to a `BufferManager` managing buffers for each client's data.

Methods:

`void run()`

- **Purpose**: Starts the server to listen for incoming connections and handles them.

- **Behavior**: The server listens on the specified port and accepts incoming client connections up to the specified maximum number of clients. Each client connection is managed by a separate thread.

`void handleClientConnection(int clientSocket, int cCount)`

- **Purpose**: Manages individual client connections in separate threads.
- **Parameters**:
  - `clientSocket`: Socket descriptor for the connected client.
  - `cCount`: Count of the client being handled.
- **Behavior**: Receives data from the client, including a header and file data, processes it, and sends an acknowledgment back to the client. Waits for aggregation to complete before sending aggregated data back to the client.

`void aggregationDone()`

- **Purpose**: Notifies all waiting threads that aggregation is complete.
- **Behavior**: Sets the aggregation flag to true and wakes up all threads waiting on this condition.

Destructor: `~FL_Server()`

- **Purpose**: Cleans up resources upon server destruction.
- **Behavior**: Closes the server and client sockets, ensuring a clean shutdown.

## Client (`FL_Client` class in `client.cpp`)

Constructor: `TCPClient(const char* server_ip, int server_port, SharedBuffer* paramsBuffer, SharedBuffer* aggregationBuffer)`

- **Purpose**: Initializes the FL client with the server's IP address and port.
- **Parameters**:
  - `server_ip`: IP address of the server to connect to.
  - `server_port`: Port number of the server.
  - `paramsBuffer`: Buffer that contains the params to send
  - `aggregationBuffer`: Buffer that contains the aggregated data received from the server
- **Behavior**: Sets up the client socket and connects to the server.

Method: `void participate()`

- **Purpose**: Sends a file to the server along with its metadata.
- **Behavior**: Prepares and sends a header with data metadata, waits for acknowledgment, sends the data, and handles the server's response including receiving aggregated data. Repeat this process until convergeance.

Destructor: `~FL_Client()`

- **Purpose**: Ensures proper cleanup when the client object is destroyed.
- **Behavior**: Closes the client socket to ensure a clean shutdown.