FileSync

Technical Documentation

Author: Antoni Iwan

May 6, 2025

Contents

1 Introduction

1.1 Project Objective

The aim of this project is to develop a file synchronization service that ensures consistency between a client-side and a server-side directory. The system is designed to handle large volumes of files, provide fault tolerance, and support automatic reconnection in case of network failures.

1.2 Scope of the Application

The application focuses on unidirectional synchronization from the client to the server. It is capable of detecting changes in files, managing concurrent transfers using multithreading, and handling disconnections gracefully. The server supports multiple clients; however, only one TCP connection can be active at a time. Other clients must wait in a queue until the connection becomes available.

1.3 Technologies Used

The implementation is written entirely in Python. The application utilizes the `socket` module for network communication, `threading` for parallelism, and `json` for message formatting. Additionally, multicast communication is used for client-server discovery, and file operations rely on the `os` modules.

2 System Overview

2.1 Client-Server Architecture

The system follows a client-server architecture, where the server is responsible for managing synchronization sessions initiated by multiple clients. The server continuously listens for incoming multicast UDP messages on a predefined discovery port. When it receives a `DISCOVER` message from a client, it responds with an `OFFER:<PORT>` message, indicating the TCP port available for synchronization. The client then establishes a TCP connection to that port for further communication. The server handles only one TCP connection at a time; other clients must wait in a queue.

2.2 Multithreading Model

The server employs several threads to manage different aspects of the system:

- A **UDP receiver thread** that waits for discovery messages and responds with TCP port offers.
- A **TCP connection manager thread** that listens on the offered port for incoming client connections.
- A **queue manager thread** that manages the client queue. When a client connects, it is enqueued. If the server is busy, it replies with a `BUSY` message; otherwise, it sends `READY` and starts synchronization.

- For each active synchronization, a **file transfer handler thread** is created to manage the file synchronization process independently.

On the client side, multiple threads are also used:

- A **Multicast discoverer** for sending multicast `DISCOVER` messages.
- A **TCP manager thread** to manage the connection to the server, contais two more threads:
  - A **TCP receiver** to receive data from the server.
  - A **TCP sender** to transmit data to the server.

This architecture ensures that the server can continue receiving discovery requests while managing an active synchronization, and that clients remain responsive and capable of performing background file monitoring.

2.3 Communication Protocol Summary

The communication protocol is split into two stages:

- **Discovery Phase (UDP)**: The client broadcasts a `DISCOVER` message via multicast. The server replies with an `OFFER:<PORT>` to inform the client where to initiate TCP communication.
- **Synchronization Phase (TCP)**:
  1. The client connects to the TCP port and sends identification data.
  2. The server checks whether another synchronization is in progress.
  3. If the server is busy, it sends `BUSY` and the client waits.
  4. If the server is free, it sends `READY`, and a new synchronization thread is launched.
  5. File metadata is exchanged, changes are determined, and files are transferred.

The communication between the client and server follows a custom discovery and synchronization protocol. Initially, the server listens for discovery messages over UDP. Upon receiving a valid discovery request, it responds with an offer message containing the TCP port number for further communication.

Subsequent communication takes place over TCP, managed by a multithreaded connection queue. Clients are added to the queue upon connection and are either served immediately if the server is available or asked to wait with a "busy" message. When ready, the server sends a "ready" message and spawns a dedicated thread to handle file synchronization.

The exact flow of messages and thread management is illustrated in Figure 1.

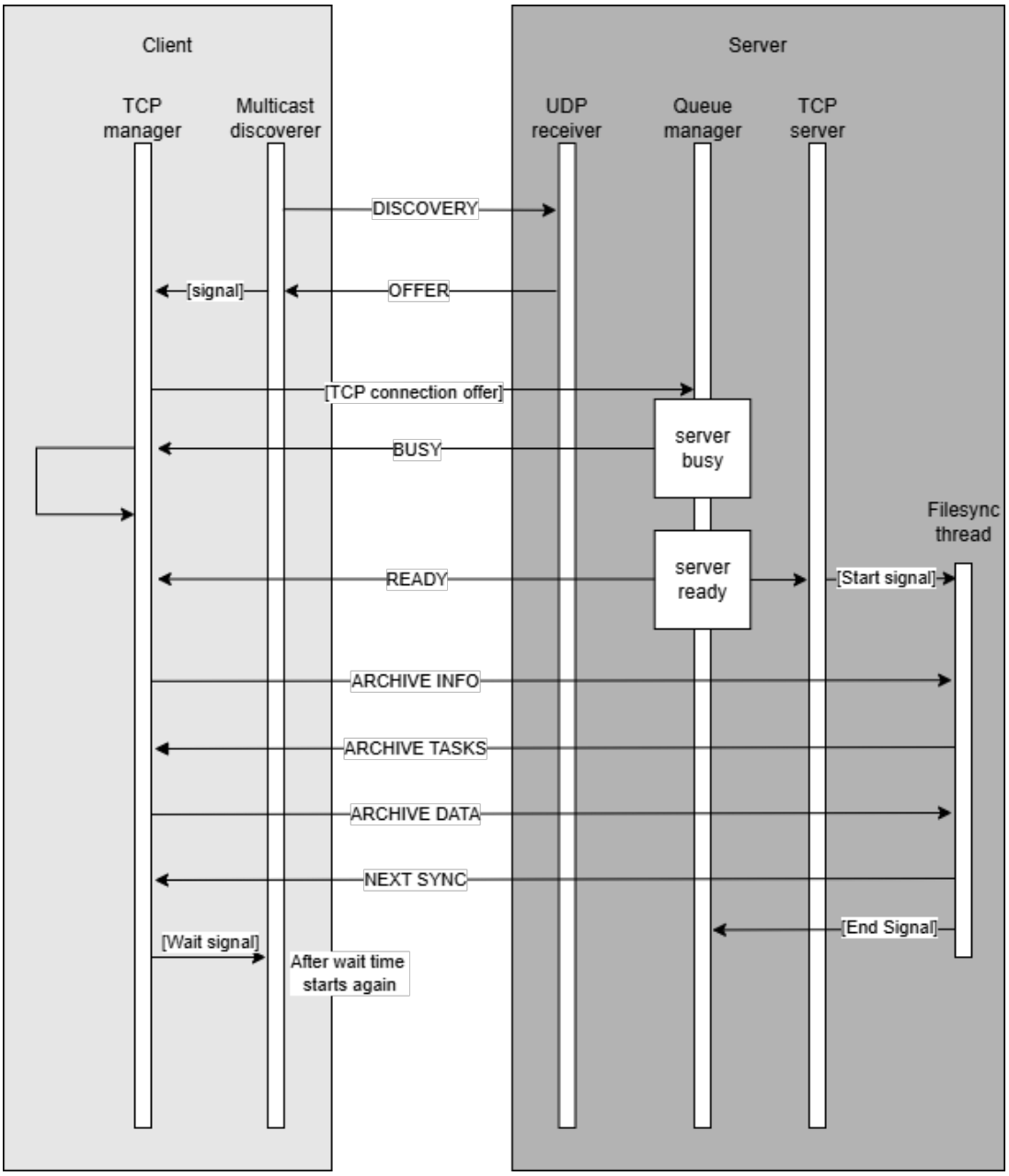

Figure 1: Client-Server communication process with threads and synchronization stages

3 Client Application

The client application is responsible for initiating communication with the server, discovering its availability, and managing the file synchronization process. It supports customizable startup parameters, uses multicast for initial discovery, and handles data transfer in a multithreaded, queued system.

3.1 Startup Parameters

The client prompts the user for several configuration options at startup, including:

- **Archive Directory:** The directory where synchronized files will be stored.

- **Unique Client ID:** An identifier for the client, used to uniquely recognize the client during communication with the server.

3.2 Multicast Discovery Mechanism

At startup, the client sends a discovery packet via UDP to a predefined multicast address and port. This packet allows the client to locate the server dynamically on the network. The server, upon receiving the discovery packet, responds with an offer containing the TCP port number where synchronization can occur. This approach enables the client to locate and communicate with the server without requiring prior configuration.

3.3 TCP Management

The TCP management system is responsible for establishing and maintaining the connection between the client and the server. It handles the full lifecycle of the TCP connection, including establishing the connection, sending and receiving data, and managing synchronization tasks.

3.3.1 TCP Connection Lifecycle

The TCP manager is responsible for waiting for a signal from the UDP part of the system that indicates a connection should be established. Once the connection signal is received, it attempts to connect to the server using the TCP protocol. Upon a successful connection, the manager starts two threads: one for receiving data **TCP receiver** and one for sending data **TCP sender**.

The threads are responsible for handling incoming and outgoing messages, ensuring the synchronization process continues efficiently. After both threads finish, the connection is closed, and the manager clears the active connection signal, preparing the system for the next synchronization task.

3.3.2 Sending Data (TCP Sender)

The **TCP sender** thread is responsible for sending data to the server once a connection is established. It waits until there is data to send, which is signaled by the dedicated data communication node. The type of message to send is determined by the protocol received from the server. The sender prepares the appropriate data, such as the client's archive information or file synchronization tasks, based on the protocol type. If the connection is closed or an error occurs during sending, the thread will terminate gracefully.

3.3.3 Receiving Data (TCP Receiver)

The **TCP receiver** thread is responsible for receiving data from the server. It waits for incoming data from the server and processes it accordingly. When data is received, the protocol type is determined, and the corresponding protocol handler is activated. If the received protocol is not BUSY, the data are stored and processed accordingly. For example, if the protocol is NEXT SYNC, the system will adjust the synchronization time. After processing the received data, the receiver signals the sender thread using dedicated data communication node signal to indicate that new data is available.

3.4 File Collection and Transmission

The client maintains an archive of synchronized files, and the **TCP sender** prepares the necessary files for transmission based on the protocol received. The special function is used to gather metadata about the files in the archive, including filenames, relative paths, and the last modified time.

Files that need to be sent to the server are determined based on the synchronization task received from the server. The system then sends the file data over the TCP connection in a reliable, thread-safe manner.

3.5 Synchronization and Error Handling

The TCP connection is synchronized with the system's state. If an error occurs, such as the server closing the connection or a protocol mismatch, the system handles these errors gracefully. It ensures that any ongoing synchronization tasks are either completed or interrupted cleanly.

The client also manages synchronization times with the server, ensuring that data transfers are performed at the correct intervals.

4 Server Application

The server application is responsible for managing client connections, synchronizing file transfer tasks, and handling data exchanges with clients over TCP and UDP. It listens for incoming client requests, manages the session queue, and initiates file synchronization tasks based on the received protocols.

4.1 Startup Configuration

Upon starting the server application, the user is prompted to configure the synchronization rate (in seconds) and the TCP port (between 1024 and 65535). The application validates these inputs and then starts the necessary threads for both the TCP server and the UDP receiver, using the provided configuration values.

- **Sync Rate:** The server sync rate determines how often synchronization tasks are triggered. It is used to set up a synchronization schedule with connected clients.
- **TCP Port:** This port is used by the server to listen for incoming TCP connections from clients.

4.2 Multicast Listener Thread

The UDP receiver listens for multicast signals from the client-side application. Upon receiving these signals, it triggers the TCP connection establishment process. The server listens on a specific UDP port to capture these signals, which indicate when the server should initiate a session with a client.

This functionality is essential for establishing initial communication between clients and the server, enabling synchronization tasks and data exchange over TCP.

4.3 TCP Listening and Session Queue

The TCP server listens for incoming client connections on a specified port. When a connection request is received, the server checks whether there is an active session being processed. If another session is ongoing, the server sends a "BUSY" message to the new client and places the connection request in the session queue.

4.3.1 Session Queue

The session queue is managed using a thread-safe queue (CLIENT QUEUE). The server processes each connection in sequence, ensuring that only one client is served at a time. The connection request is placed in the queue if the server is busy.

4.4 Client Session Workflow

Each client session is processed by the handle USP service function. The workflow is as follows:

1. **Connection Establishment:** When a client connection is accepted, the server sends a "READY" message to confirm the connection.
2. **Receiving Data:** The server listens for incoming data from the client. It processes the data according to the protocol type. If the data pertains to file synchronization (e.g., ARCHIVE INFO or ARCHIVE DATA), the server responds accordingly.
3. **Archive Handling:** When the client sends archive information (ARCHIVE INFO), the server processes this information and sends the list of files that need to be synchronized (ARCHIVE TASKS). For archive data (ARCHIVE DATA), the server processes the received files and responds with the next synchronization time (NEXT SYNC).
4. **Session Closure:** Once the synchronization tasks are complete, the server closes the connection with the client, releases the session lock, and proceeds to handle the next client in the queue.

4.5 Archival Task Generation

Archival tasks are generated based on the information received from the client. When the client sends an archive info message (ARCHIVE INFO), the server processes this data and prepares a list of files that need to be synchronized.

- **File Handling:** The server extracts the file metadata, such as filenames, paths, and modification timestamps, from the archive info and prepares a list of files for synchronization.
- **Synchronization:** Once the server has the list of files, it sends the ARCHIVE TASKS message to the client, instructing the client on what files to send for synchronization.
- **Synchronization Feedback:** After receiving the archive data from the client, the server processes the files and sends the NEXT SYNC message, indicating the next synchronization time.

5 Communication Protocol

This section describes the communication protocol used between the client and server. The protocol is based on JSON-encoded messages, which define various actions and information exchanges. Each message type is represented by a unique protocol identifier, ensuring that both parties can understand the message content and respond accordingly.

5.1 JSON Message Format

All messages in the protocol are encoded in JSON format, ensuring a human-readable and easily parsed structure. Each message contains a "type" field that identifies the protocol message type. Depending on the message type, additional fields may be included, such as port numbers, client information, file data, or synchronization times.

5.2 DISCOVER / OFFER

DISCOVER: This message is sent by a client to discover available servers in the network. It is the first step in establishing a connection. Example message:

```
{
    "type": "DISCOVER"
}
```

OFFER: In response to a DISCOVER message, the server sends an OFFER message, indicating that it is available and providing the port number where the client can connect. Example message:

```
{
    "type": "OFFER",
    "port": 12345
}
```

The port field in the OFFER message allows the client to establish a TCP connection to the server for further communication.

5.3 BUSY / READY

BUSY: This message is sent by the server when it is currently unavailable to handle new connections. It informs the client that the server is occupied and may not be able to process requests at the moment. Example message:

```
{
    "type": "BUSY"
}
```

READY: Once the server is ready to begin data exchange, it sends the READY message to the client. This signals that the server is ready to receive further requests or initiate synchronization. Example message:

```
{
    "type": "READY"
}
```

These messages ensure that clients know whether the server is available and ready for communication, preventing unnecessary retries or conflicts.

5.4 Client Archive Summary

The server and client exchange archive-related information through several protocol messages, including ARCHIVE INFO, ARCHIVE TASKS, and ARCHIVE DATA.

ARCHIVE INFO: Sent by the client to the server to provide a summary of the archives to be backed up, including the client ID and a list of files. This is the first step in archiving. Example message:

```
{
    "type": "ARCHIVE_INFO",
    "client_id": clientID,
    "files": [
        {
        "filename": "file1.txt",
        "relative_path": "archive/file1.txt" ,
        "last_modified": "2023-05-06T15:25:12"
        },
        {
        "filename": "file2.jpg",
        "relative_path": "archive/file2.jpg"
        "last_modified": "2023-06-06T15:25:12"
        }
    ]
}
```

ARCHIVE TASKS: After receiving the archive information, the server sends a list of files that are to be archived, including metadata such as filenames and file paths. Example message:

```
{
    "type": "ARCHIVE_TASKS",
    "files": [
        {
        "filename": "file1.txt",
        "file_path": "folder/file1.txt"
        },
        {
        "filename": "file2.jpg",
        "file_path": "folder/file2.jpg"
        }
    ]
}
```

This message enables the client to know exactly which files are being considered for backup and synchronization.

ARCHIVE DATA: This message contains the actual data for the archives, including base64-encoded file contents. It is sent from the client to the server after the server requests the archive data. The files are encoded and timestamped to ensure proper synchronization and integrity.

```
{
    "type": "ARCHIVE_DATA",
    "client_id": "clientID",
    "files_data":
    [
        {
            "filename": "file1.txt",
            "file_path": "folder/file1.txt",
            "last_modified": "2025-05-06T15:30:00",
            "content": "base64encodedstring..."
        },
        {
            "filename": "file2.jpg",
            "file_path": "folder/file2.jpg",
            "last_modified": "2025-05-06T15:30:00",
            "content": "base64encodedstring..."
        }
    ]
}
```

This message is crucial for performing actual data transfers, including any necessary metadata like the last modified timestamps for accurate synchronization.

5.5 Synchronization Time Message

NEXT SYNC: This message provides the client with the scheduled time for the next synchronization. It ensures that both the server and client are in sync regarding when future data exchanges should occur. Example message:

```
{
    "type": "NEXT_SYNC",
    "next_sync_time": 1700012345
}
```

The next sync time field contains a timestamp in Unix format, indicating when the next synchronization should occur.