ABSTRACT

This report contains detailed information about the SIMPLE FILE TRANSFER PROTOCOL Version 1.0 protocol and the coding process involved in file transfer using this protocol. The report begins with the Request for Comments (RFC) created to introduce the protocol and ends with a detailed analysis of the coding process for file transfer.

TABLE OF CONTENT

REQUEST FOR COMMENTS(RFC)

SIMPLE FILE TRANSFER PROTOCOL(SFTP) Version 1.0

Introduction

The Simple File Transfer Protocol is an internet protocol that enables file transfer from one user to another. To control the file transfer, a server program is utilized between the two users. This server handles the communication and ensures the integrity and successful delivery of the files. Additionally, the protocol is designed to be straightforward and easy to implement, making it a suitable choice for basic file transfer needs.

Protocol Overview

The protocol defines a file transfer process initiated by sending side. Sending side divides the file into 256-byte packets and sends them to server. Server stores these packets in a buffer and then forwards them to receiving side. The protocol uses a producer-consumer model, meaning that sending side cannot send new data until the data sent by server is consumed.

The protocol aims to ensure complete and error-free file transfer from one user to another. To achieve this, a 16-byte header is used along with each packet during file transmission. This protocol is used exclusively for transferring files larger than 100kB and files with extensions .exe or .bin.

SFTP/1.0 uses TCP for reliable data transfer and port numbers for service identification. The default port number for SFTP/1.0 is 12345-TCP.

Protocol Design

Header Structure

1 byte	1 byte	Bytes
CheckSum(4 bytes)		1 byte
		1 byte
StatusCode(1 byte)	Reserved(1 byte)	1 byte
FileSize(4 bytes)		1 byte
		1 byte
CurrentFileSize(4 bytes)		1 byte
		1 byte
PacketNumber(2 bytes)		1 byte

Figure1 General Representation of Header Format

SFTP needs an 16-byte header and maximum 256-bytes body due to the Body Length.

PacketNumber : Specifies the number of how many 256-byte packets the file to be sent can be divided into.

CurrentFileSize: Indicates the packet number of the ongoing transfer process.

FileSize: Specifies the size of the file to be sent in bytes.

Reserved: This section is reserved for updates.

StatusCode: Indicates the result of the transfer after the packet transfer.

00000001b : Buffer is full

00000111b : Buffer is empty

00000010b : Checksum Error

00000100b : Checksum Correct

CheckSum: Indicates the checksum values corresponding to the packets.

Message Body

The body, which is 256 bytes long, is used for transferring file data. It is transferred as a byte array.

Protocol Operation

The protocol supports file transfer from one user to another. The operation proceeds as follows:

The sending side establishes a connection with the server. The sending side divides the file to be sent into 256-byte packets and sends them to the server. The server stores the received packets in its buffer. If any packet is corrupted, the server discards all packets currently in its buffer and sends statusCode 00000010b to the sending side. As a result, the sender retransmits the packets. When the buffer is full and all packets have been received without errors, the server sends statusCode 00000001b to the sending side, and packet transmission stops. The server sequentially forwards the received packets to the receiving side. If any packet is corrupted, the sending side sends statusCode 00000010b to the server, and the server retransmits the packets. If the packets are received correctly, the sending side sends statusCode 00000100b to the server. As a result, the server's buffer is emptied. The server sends statusCode 00000111b to the sending side, and packet transmission resumes. This process continues until all packets have been sent.

Examples

The headers of the first and last packets sent from TCPClient1 to TCPServer are given as examples respectively. It is assumed that the file to be sent is 100KB.

Size of the last packet: 100000%256 = 160 bytes

Total number of packets: 100000/256 + 1 (last packet) = 391

Header of the first packet sent:

PacketNumber: 391

CurrentFileSize: 160

FileSize: 100000

Reserved: 0

StatusCode: 0

Checksum: checksum value of packet

Header of the last packet sent:

PacketNumber: 1

CurrentFileSize: 100000

FileSize: 100000

Reserved: 0

StatusCode: 0

Checksum: checksum value of packet

Conclusion

The Simple File Transfer Protocol (SFTP) Version 1.0 provides a reliable and straightforward method for transferring large files between users, using a server to manage the process. By employing a producer-consumer model and a robust header structure, SFTP ensures data integrity and efficient communication. The protocol's design, which includes error checking through status codes and checksum verification, guarantees error-free file transfers. Its simplicity and effectiveness make SFTP a suitable choice for basic file transfer needs over TCP, particularly for files larger than 100kB and with .exe or .bin extensions.

CODE EXPLENATION

Introduction

There are three main programs for file transfer using the SFTP protocol. These are TCPClient1, TCPServer, and TCPClient2 programs. The file transfer takes place between TCPClient1 and TCPClient2. TCPServer controls the file transfer process. TCPClient1 can be referred to as the sending side, TCPClient2 as the receiving side, and TCPServer as the server.

The programs are written in Java using the IntelliJ IDEA environment.

Functions

isFileSizeLessThan100KB (String filePath): This function checks if the size of a given file is less than 100 KB. It achieves this by first creating a File object using the provided file path. Then, it retrieves the file size in bytes using the length() method of the File class. The file size in bytes is then converted to kilobytes by dividing the byte size by 1024. Finally, the function returns true if the resulting file size in kilobytes is less than 100, otherwise it returns false. Located on TCPClient1.

checkFileExistence (String filePath): This function verifies the existence of a file at a specified path and ensures that it is a file (and not a directory). It creates a File object with the given file path and then checks if the file exists using the exists() method. Additionally, it checks if the path corresponds to a file using the isFile() method. If both conditions are satisfied, the function returns true, indicating that the file exists and is indeed a file; otherwise, it returns false. Located on TCPClient1.

combineHeader (int reserved, int statusCode, int fileSize, int currentFileSize, int packetnumber, int checksum): This function constructs a header for a packet by combining several fields into a byte array. It uses a ByteBuffer to sequentially place each field into the buffer. The fields include a 12-bit packet number, current file size, total file size, a reserved byte, a 3-bit status code, and a checksum. The ByteBuffer is allocated with enough space to hold all these fields (16 bytes). After placing all the values into the buffer, the function returns the byte array representation of the header. Located on TCPClient1, TCPServer and TCPClient2.

combineHeaderAndData (byte[] header, byte[] body): This function merges a header byte array and a data byte array into a single byte array. It first creates a new byte array that is large enough to hold both the header and the data. Using the System.arraycopy method, it copies the header array into the beginning of the new array, followed by copying the data array into the subsequent positions. The resulting combined byte array, which contains both the header and the data, is then returned. Located on TCPClient1, TCPServer and TCPClient2.

parseHeader (byte[] header): This function extracts and returns the fields from a header byte array as a string. It wraps the header array in a ByteBuffer to facilitate the extraction of individual fields. The function reads the packet number, current file size, total file size, reserved byte, status code, and checksum from the buffer. Each of these values is then formatted into a string and returned, allowing for easy inspection of the header's contents. Located on TCPClient1, TCPServer and TCPClient2.

calculateChecksum (File file): This function computes the checksum of a file by reading its contents into a byte array and then performing an XOR operation on each byte. The file is read into a byte array using a FileInputStream, and the readFileToByteArray helper function. Once the file data is in a byte array, the function iterates over each byte, applying an XOR operation cumulatively to calculate the checksum. The resulting checksum value is then returned. Located on TCPClient1.

readFileToByteArray (File file): This helper function converts the contents of a file into a byte array. It initializes a FileInputStream to read the file's data and allocates a byte array with a size equal to the file length. The file data is read into this array. After reading, the FileInputStream is closed to release system resources. The byte array containing the file's data is then returned. Located on TCPClient1.

getSpecificPart (File inputFile, int chunkSize, int partNumber): This function retrieves a specific part of a file by dividing it into chunks and returning the specified chunk. It opens a FileInputStream to read the file and uses a buffer of the specified chunk size. The function reads the file in chunks, keeping track of the current chunk number. When the desired chunk is reached, it closes the input stream and returns the chunk as a byte array. If the chunk size is larger than the remaining file data, it adjusts the buffer size to match the actual data length. Located on TCPClient1.

calculateChecksum(byte[] data): This function calculates the checksum of a given byte array using an XOR operation. It initializes a checksum variable to zero and iterates

over each byte in the array, applying an XOR operation cumulatively to each byte. The final checksum value, which represents the result of the XOR operations, is then returned. This function is useful for validating data integrity by comparing checksums before and after data transmission or storage. Located on TCPClient1, TCPServer and TCPClient2.

compareChecksum(byte[] data, long receivedChecksum): This function compares the checksum of a given byte array to a received checksum to verify data integrity. It first calculates the checksum of the provided data array by calling the calculateChecksum method. This method iterates over each byte in the array and performs an XOR operation to produce the checksum. Once the calculated checksum is obtained, the function compares it to the received checksum, which is passed as a parameter. If the calculated checksum matches the received checksum, the function returns true, indicating that the data has not been altered and its integrity is intact. Otherwise, it returns false, indicating a potential discrepancy in the data.

verifyFileSize(byte[] receivedFileBytes, String filePath): This function verifies that the size of a received file matches the size of the actual file on disk. It begins by creating a File object using the provided file path. Then, it retrieves the size of the actual file in bytes using the length() method of the File class. Simultaneously, it determines the size of the received file by checking the length of the receivedFileBytes array. The function compares the size of the received file to the actual file size. If both sizes are equal, it returns true, indicating that the file sizes match and the received file is complete. Otherwise, it returns false, indicating a size mismatch which may suggest an incomplete or corrupted file transfer.

Process

The file transfer process occurs as described in the Protocol Operation section. To perform any file transfer, the following steps should be followed:

- Start TCPServer.
- Start TCPClient1. Copy the path of the file to be sent and proceed.(ex. C:\Users\onatb\OneDrive\Masaüstü\hw5.txt)
- Start TCPClient2. Once the file transfer is complete, a message will appear on the console. Based on the message, it can be determined whether the file transfer

was successful or not. The time required to complete the process may vary depending on the size of the file to be sent.