Report for option1

Hengrui Liu

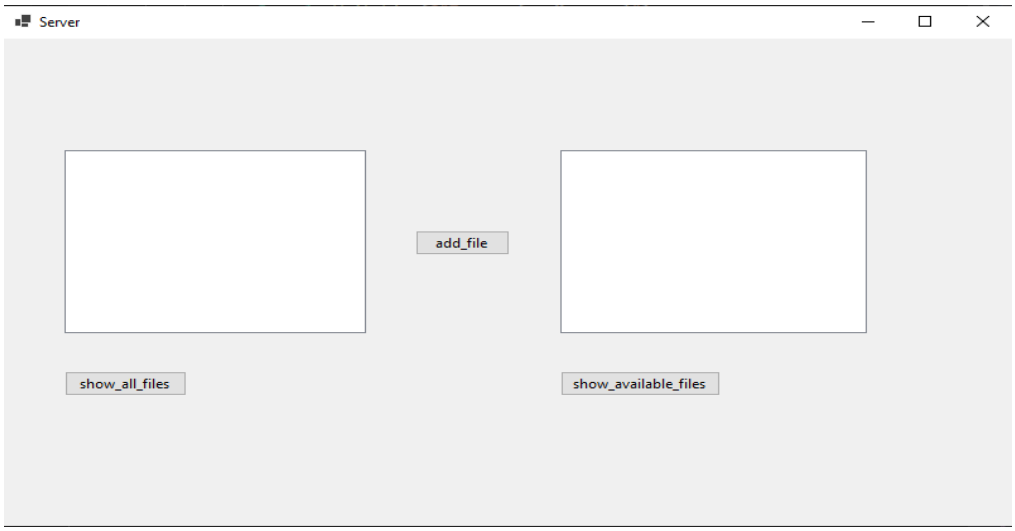
hliu825

(a) (b) I implemented everything from option1 and have tested it on a lab machine

(c)How to run: Start in the order of server-cache-client. The Server folder is server, the WinFormApp1 folder is cache, and the client folder is client. After starting, click the corresponding button to realize the corresponding function.

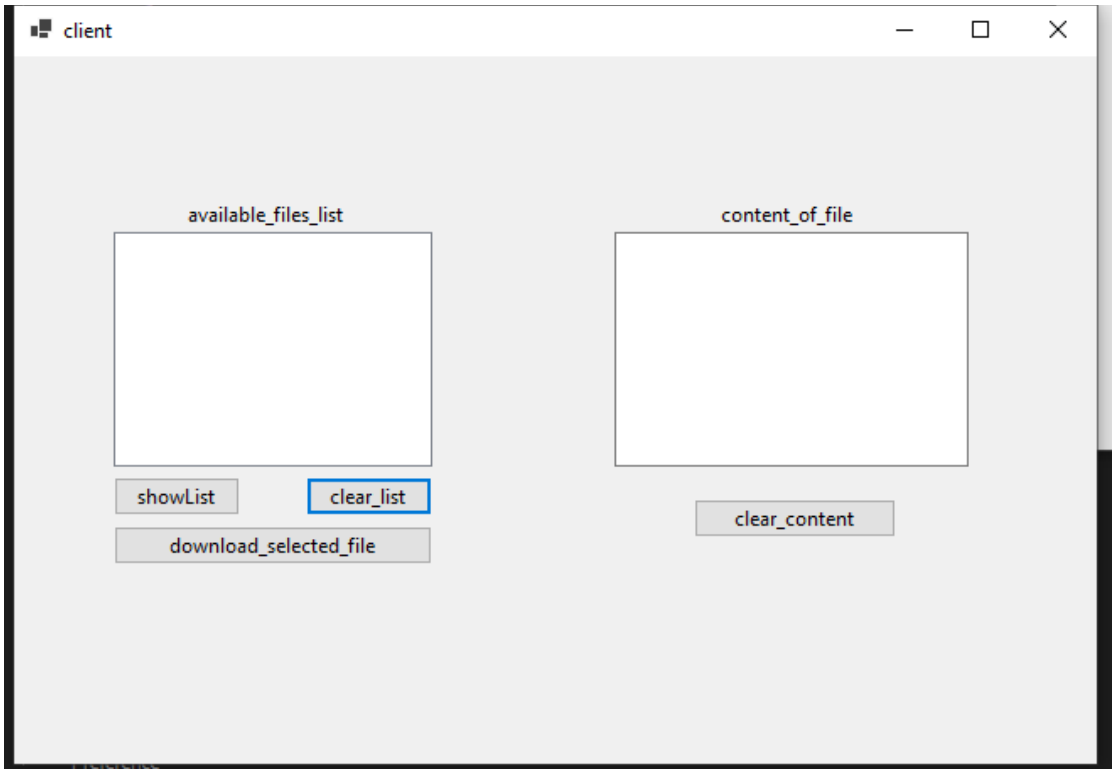
Server and cache port is 8081,8082.

Server



Cache

Client



(d)

The technology used in Option2

For option2, I thought about the required technology. I divided the data in the local folder in the server into chunks when starting the server. Using the rolling Robin algorithm, set the window size to 3bytes, then loop through the file content, and calculate the corresponding hash value for each window. Take the hash value %2048 calculated for each window as the result. This allows the size of each hash to be distributed between 0-2048. Then when the result is equal to the preset hash value, the file is sliced. Since the hash algorithm is random, this can ensure that the probability of each hash value division is 1/2048. Therefore, the size of each divided file block can be maintained at about 2KB. Below is the segmentation algorithm I am trying to implement. In the following code, the given file is divided into blocks, and the block size is about 2KB.

For the cache, when the cache receives a GET_CONTENT request from the server, it first reads the file name requested by the client, and uses the file name as the name of the folder to create a folder under the path ../../../, For example, if the client requests a test.tmp or test.txt file, a folder is created with the path ../../../test. Then forward the request to the server. After receiving the data stream sent by the server, the data stream first sends the number of file blocks, and then the server traverses each file block to send the operation code (0 or 1) and the corresponding hash value or the length and content of the file block. The cache first reads the number of file blocks, and then loops for the corresponding number of times. In the loop, it first reads the opcode of the bytes type, 0 or 1. If the opcode

is 0, then the hash value of the bytes type will be received next. Read the hash value, convert the hash value into a string, and then use it as the name of the file to traverse in the ../../../block folder. After finding the file block named by the hash value, Copy this file to the folder previously created with the requested file name, and name it with the requested file name plus the file block index, for example, the first cycle is test_1, the second cycle is test_2, and then placed in the test file in the middle. If the opcode is 1, then read the length of the file block next, and then read the content of the file block according to the length, use the SHA256 algorithm to calculate the hash value for the content of the file block, and use the calculated hash value as the file name Create a file in block. Then create another file with the content of the file and store it in a folder named after the requested file name, such as the test folder. The name of this file is the requested file name plus the file block index, such as test_1, test_2. Just read the stream in a loop. After that, the files are spliced in the cache, and then the content is returned to the client.

The following is part of the implementation code

First call in the function SplitAllFilesAndSaveBlocks() in startserver()

```
private void SplitAllFilesAndSaveBlocks()
    {
        string currentFolderPath = Path.GetDirectoryName(Application.ExecutablePath);
        string allFilesFolderPath = Path.Combine(currentFolderPath, "../../../all_files");
        string[] allFiles = Directory.GetFiles(allFilesFolderPath);

        foreach (string file in allFiles)
        {
            SplitFileAndSaveBlocks(file);
        }
    }



private void SplitFileAndSaveBlocks(string inputFilePath)
    {
        string blockBaseFolderPath = "../../../block";
        int fixedHashValue = 0;

        // Create a new folder named after the input file
```

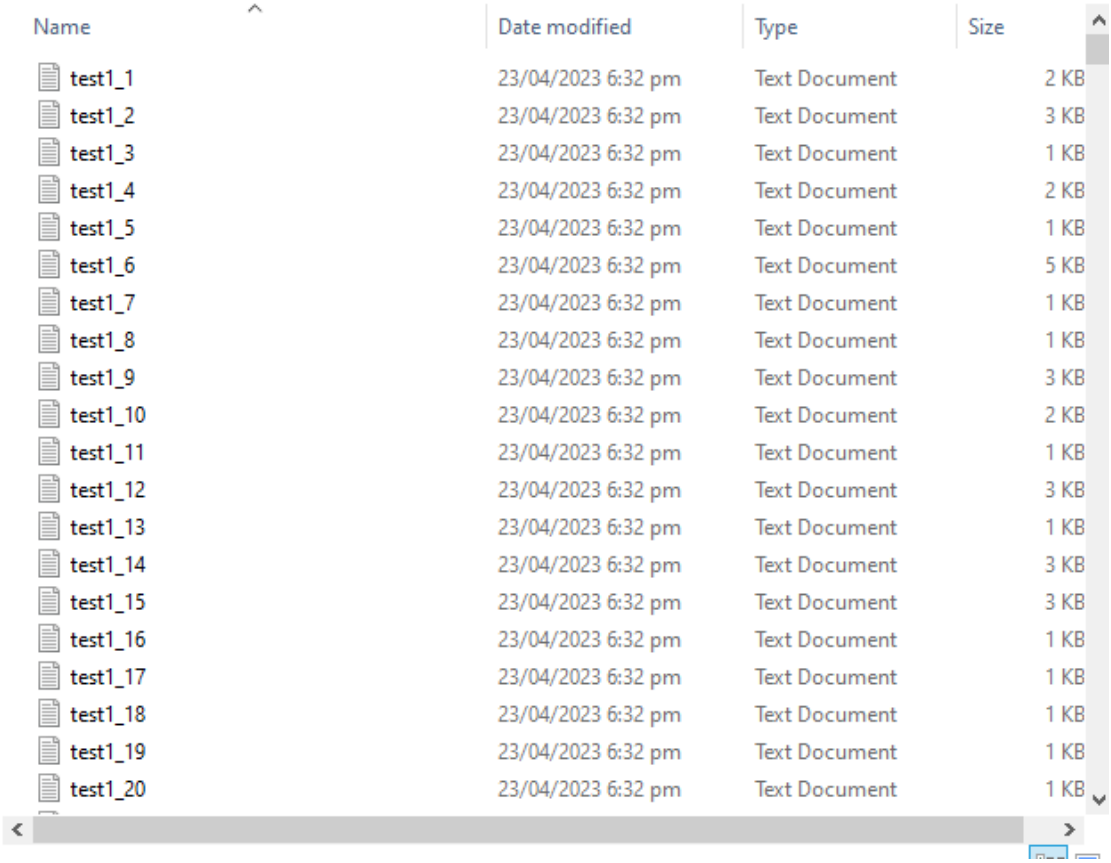
```
        string fileName = Path.GetFileNameWithoutExtension(inputFilePath);
        string blockFolderPath = Path.Combine(blockBaseFolderPath, fileName);
        Directory.CreateDirectory(blockFolderPath);

        List<byte[]> blocks = SplitFileIntoBlocks(inputFilePath, blockFolderPath,
fixedHashValue);

        Console.WriteLine($"File {fileName} has been successfully split into blocks.");
    }



public static List<byte[]> SplitFileIntoBlocks(string filePath, string blockFolderPath, int
fixedHashValue)
    {
        byte[] fileContent = File.ReadAllBytes(filePath);
        List<byte[]> blocks = new List<byte[]>();

        int blockStart = 0;
        int blockNumber = 1;
        int windowSize = 3; // Increase the window size to provide a better distribution of
hash values

        for (int i = 0; i <= fileContent.Length - windowSize; i++)
        {
            byte[] windowData = new byte[windowSize];
            Array.Copy(fileContent, i, windowData, 0, windowSize);
            int hashValue = CalculateHash(windowData);

            if (hashValue == fixedHashValue)
            {
                int blockSize = i - blockStart + windowSize;
```

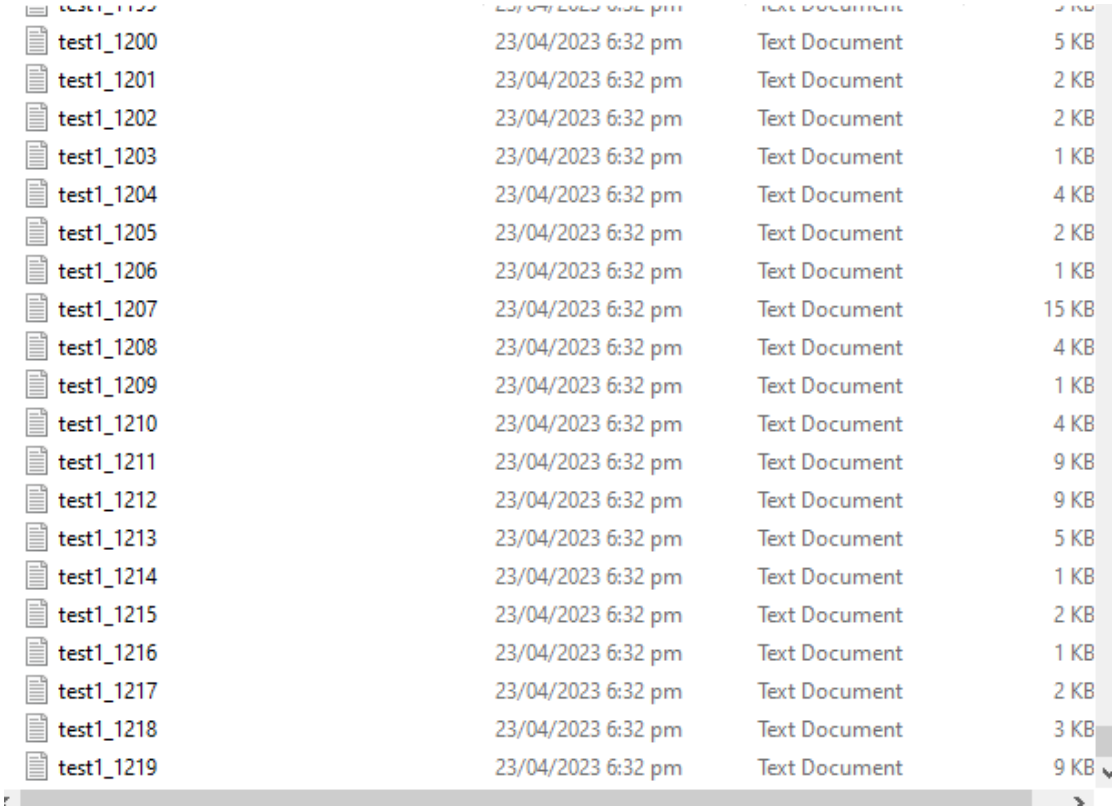
```
            byte[] block = new byte[blockSize];
            Array.Copy(fileContent, blockStart, block, 0, blockSize);
            blocks.Add(block);

            // Save block to file
            string folderName = Path.GetFileName(blockFolderPath);
            string blockFileName = $"{folderName}_{blockNumber}.txt";
            string blockFilePath = Path.Combine(blockFolderPath, blockFileName);
            File.WriteAllBytes(blockFilePath, block);

            blockStart = i + windowSize;
            blockNumber++;
            i += windowSize - 1; // Move the index to the end of the window
        }
    }

    if (blockStart < fileContent.Length)
    {
        byte[] block = new byte[fileContent.Length - blockStart];
        Array.Copy(fileContent, blockStart, block, 0, block.Length);
        blocks.Add(block);

        // Save block to file
        string folderName = Path.GetFileName(blockFolderPath);
        string blockFileName = $"{folderName}_{blockNumber}.txt";
        string blockFilePath = Path.Combine(blockFolderPath, blockFileName);
        File.WriteAllBytes(blockFilePath, block);
    }

    return blocks;
}
```

```
private static int CalculateHash(byte[] data)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] hashBytes = sha256.ComputeHash(data);
            int hash = BitConverter.ToInt32(hashBytes, 0);
            return Math.Abs(hash) % 2048;
        }
    }



    private string Calculate(byte[] data)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] hashBytes = sha256.ComputeHash(data);
            string hashString = BitConverter.ToString(hashBytes).Replace("-",
"").ToLowerInvariant();


            string currentFolderPath = Path.GetDirectoryName(Application.ExecutablePath);
            string hashFilePath = Path.Combine(currentFolderPath, "../../../hash.txt");


            // If the hash.txt file doesn't exist, create it
            if (!File.Exists(hashFilePath))
            {
                File.Create(hashFilePath).Dispose();
            }
```

```
        // Append the hash value to the file
        using (StreamWriter writer = File.AppendText(hashFilePath))
        {
            writer.WriteLine(hashString);
        }

        return hashString;
    }
}
```

The figure below shows the result after segmentation.





test1_1200	23/04/2023 6:32 pm	Text Document	5 KB
test1_1201	23/04/2023 6:32 pm	Text Document	2 KB
test1_1202	23/04/2023 6:32 pm	Text Document	2 KB
test1_1203	23/04/2023 6:32 pm	Text Document	1 KB
test1_1204	23/04/2023 6:32 pm	Text Document	4 KB
test1_1205	23/04/2023 6:32 pm	Text Document	2 KB
test1_1206	23/04/2023 6:32 pm	Text Document	1 KB
test1_1207	23/04/2023 6:32 pm	Text Document	15 KB
test1_1208	23/04/2023 6:32 pm	Text Document	4 KB
test1_1209	23/04/2023 6:32 pm	Text Document	1 KB
test1_1210	23/04/2023 6:32 pm	Text Document	4 KB
test1_1211	23/04/2023 6:32 pm	Text Document	9 KB
test1_1212	23/04/2023 6:32 pm	Text Document	9 KB
test1_1213	23/04/2023 6:32 pm	Text Document	5 KB
test1_1214	23/04/2023 6:32 pm	Text Document	1 KB
test1_1215	23/04/2023 6:32 pm	Text Document	2 KB
test1_1216	23/04/2023 6:32 pm	Text Document	1 KB
test1_1217	23/04/2023 6:32 pm	Text Document	2 KB
test1_1218	23/04/2023 6:32 pm	Text Document	3 KB
test1_1219	23/04/2023 6:32 pm	Text Document	9 KB

Comparation:

Option1 can only reuse files with the same name and the same content. For large files, even if only a small part of the content changes, it cannot be reused. In Option2, a large file is divided into many small file blocks. As long as the content of the file blocks is the same, they will have the same hash value, so the server can detect and complete the reuse.

The option1 cache only needs to calculate and judge whether the file name already exists in the cache folder, but the server of option1 does not need to judge. The server in Option2 needs to find the corresponding folder after receiving the requested file name, calculate the hash value for each file block content, and then determine whether the file block is stored in the cache. Option2 server has more calculation than option1

The response time of option1 is very fast when the file has not changed, because the request does not need to be sent to the server, and it can be processed directly in the cache. Option2 needs to cut the file into pieces during initialization, which requires a certain response time. But in most cases, option2 saves a lot of bandwidth than option1