Problem Set 3 - Networked Producer and Consumer

Argamosa, Daniel Cedric (S14)
Donato, Adriel Joseph (S12)

Key Implementation Steps

- **Producer** ran on the user's current unit, while the **Consumer** ran on a VM and were connected through a TCP port using the VM's Ipv4 address and inputted port number
- Validate user inputs such as consumer IP, port number, thread count, and directory paths using loops and conditional checks before properly launching the application
- It packages the **file name length** (4 bytes), the **file name** (in UTF-8), the **file size** (8 bytes), and the **file data** into a single stream before sending them off

```
static void SendFile(string filePath)
{
    try
    {
        string originalFileName = Path.GetFileName(filePath);
        byte[] fileNameBytes = Encoding.UTF8.GetBytes(originalFileName);
        int fileNameLength = fileNameBytes.Length;

        // Read the file data and determine original file size.
        byte[] fileData = File.ReadAllBytes(filePath);
        long fileSize = fileData.Length;

        byte[] fileNameLengthBytes = BitConverter.GetBytes(IPAddress.HostToNetworkOrder(fileNameLength));
        byte[] fileSizeBytes = BitConverter.GetBytes(IPAddress.HostToNetworkOrder(fileSize));

        using (TcpClient client = new TcpClient())
        {
            client.Connect(consumerIP, consumerPort);
            using (NetworkStream ns = client.GetStream())
            {
                // Send header (filename length, filename, file size)
                ns.Write(fileNameLengthBytes, 0, fileNameLengthBytes.Length);
                ns.Write(fileNameBytes, 0, fileNameBytes.Length);
                ns.Write(fileSizeBytes, 0, fileSizeBytes.Length);
                ns.Flush();

                // Send file data
                ns.Write(fileData, 0, fileData.Length);
                ns.Flush();

                // Wait for a response from the consumer
                byte[] responseBuffer = new byte[100];
                int bytesRead = ns.Read(responseBuffer, 0, responseBuffer.Length);
                string response = Encoding.UTF8.GetString(responseBuffer, 0, bytesRead);
                Console.WriteLine($"\nReceived response from consumer: {response}\n");
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error sending file {filePath}: {ex.Message}");
    }
}
```

Key Implementation Steps

- Initializes the main form, load existing videos, and set up the UI components using **ListView and Windows Media Player** for video preview and playback
- Start a **TCP listener on the specified port** to accept incoming uploads and spawn separate worker threads for processing files from different directories
- Launch dedicated threads per directory to read and send video files from the file system, ensuring concurrent file transmission from multiple sources
- Gives feedback to the Producer using messages for various cases, such as acceptance, duplicate checking, and queue dropping

```
public MainForm(int consumerThreadsCount, int queueCapacity, int listeningPort)
{
    this.consumerThreadsCount = consumerThreadsCount;
    this.listeningPort = listeningPort;
    videoQueue = new BoundedQueue<VideoUpload>(queueCapacity);

    InitializeComponent();
    BuildUI();

    // Ensure the folder for uploaded videos exists.
    if (!Directory.Exists("UploadedVideos"))
    {
        Directory.CreateDirectory("UploadedVideos");
    }

    // Ensure the UploadedVideos folder exists and load any existing files.
    LoadExistingVideos();
```

```
// Start the worker threads to process the upload queue.
for (int i = 0; i < consumerThreadsCount; i++)
{
    Thread worker = new Thread(new ThreadStart(ProcessQueue));
    worker.IsBackground = true;
    worker.Start();
}
```

Queueing Details

- Implement a custom **bounded queue**, leaky bucket, that controls video upload buffering and has a fixed capacity set via configuration
- In the queue, the **TryEnqueue** method uses a lock to guarantee thread safety and immediately returns false when the maximum capacity is reached, resulting in file drops
- The **Dequeue** method employs an **AutoResetEvent** to wait until an item becomes available, ensuring that consumer processing occurs only when items are present
- **Duplicate detection** is by using a hash-based system that checks the video itself and does not accept it if it already exists in there.

```
// Bounded queue that drops items if full (leaky bucket)
3 references
public class BoundedQueue<T>
{
    private Queue<T> queue = new Queue<T>();
    private int capacity;
    internal object lockObj = new object();
    private AutoResetEvent itemEnqueued = new AutoResetEvent(false);
```

```
// Try to enqueue an item; return false if the queue is full.
1 reference
public bool TryEnqueue(T item)
{
    lock (lockObj)
    {
        if (queue.Count >= capacity)
        {
            return false; // Drop the item
        }
        queue.Enqueue(item);
        itemEnqueued.Set();
        return true;
    }
}
```

```
// Blocking dequeue (waits until an item is available)
1 reference
public T Dequeue()
{
    while (true)
    {
        lock (lockObj)
        {
            if (queue.Count > 0)
            {
                return queue.Dequeue();
            }
        }
        itemEnqueued.WaitOne(100);
    }
}
```

```
lock (hashLock)
{
    if (knownHashes.Contains(fileHash))
    {
        // Duplicate detected - reject the upload
        responseMsg = $"DUPLICATE_CONTENT: Video {originalFileName} already uploaded under different name";
        byte[] duplicateResponse = Encoding.UTF8.GetBytes(responseMsg);
        ns.Write(duplicateResponse, 0, duplicateResponse.Length);
        ns.Flush();
        return;
    }
}
```

Breakdown of Producer and Consumer Concepts Applied

- The **Producer** reads video files from user-specified directories and, if a file exceeds 10 MB, optionally compresses it using **FFmpeg** via a separate function that reduces quality for a smaller size without changing the file name.
- After reading the file and header information, the Producer sends the header (filename length, filename, file size) and the file data sequentially over a TCP client and waits for a final response message from the Consumer.

```
if (fileSize > threshold)
{
    // Call the separate video compression function
    if (CompressVideo(filePath, out byte[] newFileData, out long newSize))
    {
        fileData = newFileData;
        fileSize = newSize;
        compressed = true;
        Console.WriteLine($"\nVideo above 20MB, file {originalFileName} was compressed.
    }
    else
    {
        Console.WriteLine($"\nCompression failed for file {originalFileName}. Proceedin
    }
}
```

```
using (TcpClient client = new TcpClient())
{
    client.Connect(consumerIP, consumerPort);
    using (NetworkStream ns = client.GetStream())
    {
        // Send header
        ns.Write(fileNameLengthBytes, 0, fileNameLengthBytes.Length);
        ns.Write(fileNameBytes, 0, fileNameBytes.Length);
        ns.Write(fileSizeBytes, 0, fileSizeBytes.Length);

        // Send file data immediately
        ns.Write(fileData, 0, fileData.Length);
        ns.Flush();

        // Then wait for a response (e.g., OK or QUEUE_FULL)
        byte[] responseBuffer = new byte[100];
        int bytesRead = ns.Read(responseBuffer, 0, responseBuffer.Length);
        string response = Encoding.UTF8.GetString(responseBuffer, 0, bytesRead);
        Console.WriteLine($"\nReceived response from consumer: {response}\n");
    }
}
```

Breakdown of Producer and Consumer Concepts Applied

- The **Consumer**, on the receiving end, uses a TcpListener to handle each incoming connection, reads the file header, file name, and file data, and then checks for duplicates using the SHA-256 hash.
- Based on the state of the bounded queue and duplicate detection, the Consumer sends back a final response (O**K, QUEUE_FULL, or DUPLICATE_CONTENT**) to inform the Producer whether the file was accepted, dropped, and compressed

```
private void HandleClient(TcpClient client)
{
    try
    {
        using (var ns = client.GetStream())
        {
            // Read file name length (4 bytes)
            byte[] intBuffer = ReadExact(ns, 4);
            if (intBuffer == null) return;
            int fileNameLength = IPAddress.NetworkToHostOrder(BitConverter.ToInt32(intBuffer, 0));

            // Read the file name
            byte[] nameBuffer = ReadExact(ns, fileNameLength);
            if (nameBuffer == null) return;
            string originalFileName = Encoding.UTF8.GetString(nameBuffer);

            // Read file size (8 bytes)
            byte[] longBuffer = ReadExact(ns, 8);
            if (longBuffer == null) return;
            long fileSize = IPAddress.NetworkToHostOrder(BitConverter.ToInt64(longBuffer, 0));

            // Now, read the file data.
            byte[] fileData = ReadExact(ns, (int)fileSize);
            if (fileData == null) return;
```

```
VideoUpload vu = new VideoUpload { FileName = finalFileName, Data = fileData };
bool enqueued = videoQueue.TryEnqueue(vu);

if (!enqueued)
{
    responseMsg = $"QUEUE_FULL: Dropping file: {originalFileName}";
}
else
{
    // Only store the hash after successful enqueue
    lock (hashLock)
    {
        knownHashes.Add(fileHash);
    }

    responseMsg = $"OK: File accepted: {originalFileName}{compressionMsg}";
}
```

Synchronization Mechanisms Used to Solve the Problem

- Use of the lock statement in both the **TryEnqueue** and **Dequeue** methods to protect access to the shared queue, ensuring that concurrent threads do not modify the queue simultaneously
- Implementation of **AutoResetEvent** to signal waiting threads immediately when new items are enqueued, allowing the consumer worker threads to wake up and process files
- Multithreading is applied throughout the system: separate **Producer** threads handle file reading and sending, while **Consumer** threads handle network connections and queue processing concurrently
- An artificial delay using **Thread.Sleep**, is introduced in the queue processing method to simulate slow processing and test the queue's full behavior, ensuring that the drop logic and response messages are triggered as expected

```
/*
* Processes the video queue by saving videos to disk and updating the UI
* Continuously dequeues video data from the shared queue,
* writes it to the "UploadedVideos" directory, and updates the ListView
* on the UI thread to display newly saved videos.
*/
1 reference
private void ProcessQueue()
{
    while (true)
    {
        VideoUpload vu = videoQueue.Dequeue();
        // Save the file to disk
        string filePath = Path.Combine("UploadedVideos", vu.FileName);
        File.WriteAllBytes(filePath, vu.Data);
        Console.WriteLine("Saved file: " + vu.FileName);

        // Introduce an artificial delay because otherwise it would process too fast
        Thread.Sleep(5000);

        // Update the ListView on the UI thread.
        this.Invoke((MethodInvoker)delegate {
            if (!listViewVideos.Items.ContainsKey(vu.FileName))
            {
                ListViewItem item = new ListViewItem(vu.FileName);
                item.Name = vu.FileName;
                listViewVideos.Items.Add(item);
            }
        });
    }
}
```