Problem Set 3

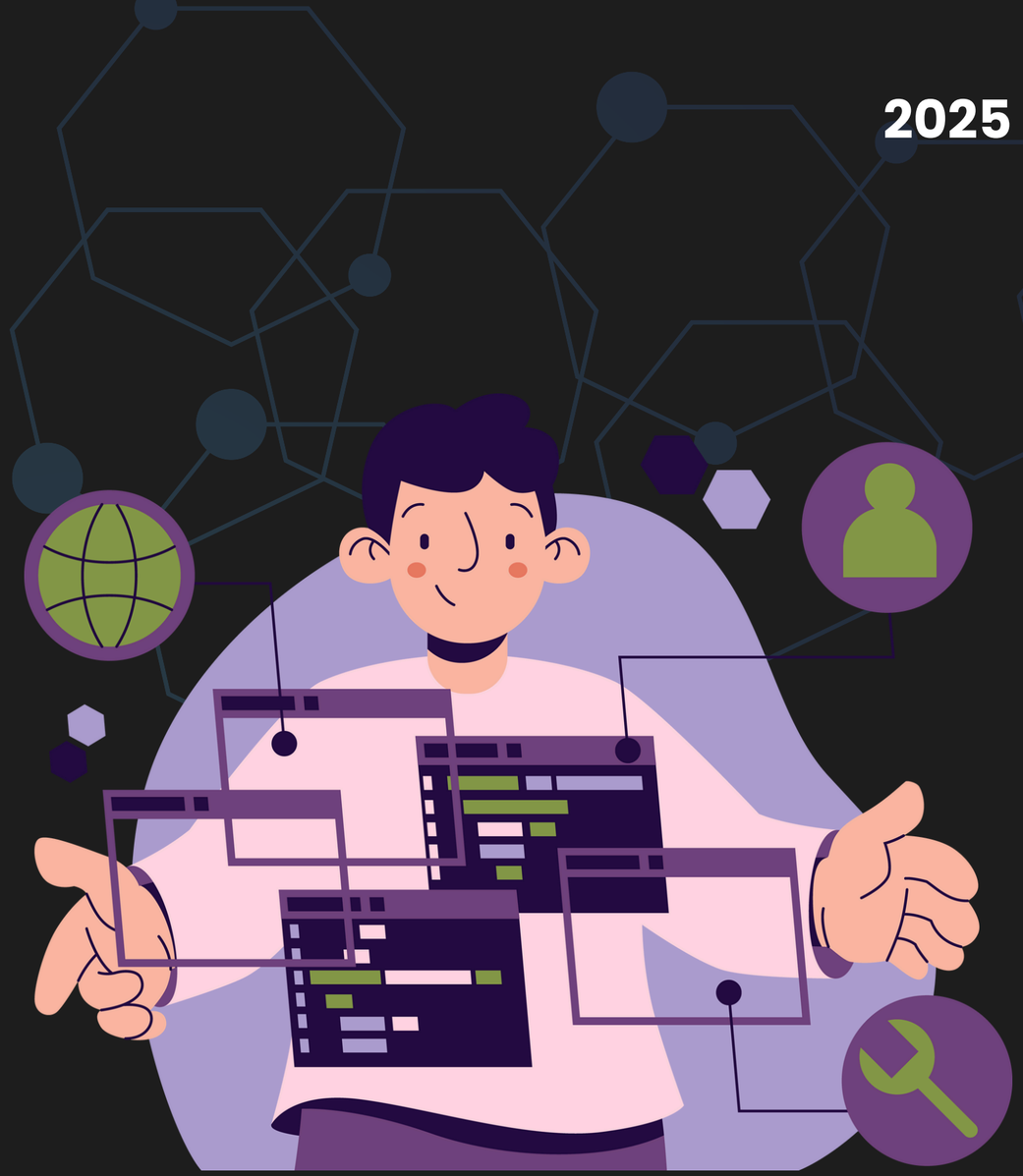
2025

NETWORKED PRODUCER AND CONSUMER

STDISCM S11 GROUP 5

Project Overview

This project, coded in **C#**, simulates a media upload service using a **producer-consumer model** with network communication. The producer and consumer run on separate virtual machines and communicate via **network sockets**, practicing **concurrent programming, file I/O, queuing, and network communication**.



Key Implementation Steps

Producer Implementation

- User drops folders into the app.
  - Each folder is validated for video content.
  - Adds accepted folders to folderPaths (HashSet) and FoldersToUpload (UI binding)
- When “Upload” is clicked
  - The app connects to the Consumer
  - File compression (user selection)
  - Splits work across threads to upload videos using TCP sockets
  - Files can be skipped (if duplicates) or added to the retry list (queue full)

```
private bool IsValidVideoFolder(string directoryPath)
{
    if (Directory.EnumerateDirectories(directoryPath).Any())
    {
        return false;
    }

    try
    {
        var files = Directory.EnumerateFiles(directoryPath, "*.*", SearchOption.AllDirectories);

        bool allAreVideos = files.All(file => allowedExtensions.Contains(Path.GetExtension(file)));
        return allAreVideos;

    }
    catch (UnauthorizedAccessException uaEx)
    {
        return false;
    }
    catch (Exception ex)
    {
        return false;
    }
}
```

```
public void HandleDrop(string[] paths)
{
    var acceptedFoldersForMessage = new List<string>();
    var rejectedFoldersForMessage = new List<string>();
    var duplicateFolders = new List<string>();
    foreach (string path in paths)
    {
        if (Directory.Exists(path))
        {
            if (IsValidVideoFolder(path))
            {
                if (folderPaths.Add(path))
                {
                    acceptedFoldersForMessage.Add(Path.GetFileName(path));
                    FoldersToUpload.Add(new FolderItem { FolderPath = path });
                }
                else
                {
                    duplicateFolders.Add(Path.GetFileName(path));
                }
            }
            else
            {
                rejectedFoldersForMessage.Add(Path.GetFileName(path));
            }
        }
    }
    if (rejectedFoldersForMessage.Count > 0)
    {
        MessageBox.Show($"Only folders that are empty or contain exclusively supported video file
                        "Invalid Folder Content", MessageBoxButton.OK, MessageBoxImage.Warning);
    }
}
```

Key Implementation Steps

Producer Implementation

- UploadVideoAsync()
  - Validates the IP address of the Consumer
  - Check if consumer is online
  - Divide folders among the ThreadCount threads

```
[RelayCommand(CanExecute = nameof(CanUpload))]
2 references
private async Task UploadVideosAsync()
{
    if (!System.Net.IPAddress.TryParse(ConsumerIP, out _))
    {
        System.Windows.MessageBox.Show("Please enter a valid IP address.", "Invalid IP", System.Windo
        return;
    }

    if (!CheckOnline(ConsumerIP, 5001))
    {
        MessageBox.Show("Failed to connect to server.",
                        "Connection Failed", MessageBoxButton.OK, MessageBoxImage.Warning);
        return;
    }

    isUploading = true;
    UploadVideosCommand.NotifyCanExecuteChanged();

    var folderList = folderPaths.ToList();
    int total = folderList.Count;
    int perThread = total / ThreadCount;
    int remainder = total % ThreadCount;

    int startIndex = 0;
    List<Thread> threads = [];

    for (int i = 0; i < ThreadCount; i++)
    {
        int count = perThread + (i < remainder ? 1 : 0);
        var subset = folderList.GetRange(startIndex, count);
        startIndex += count;

        var videoList = getVideoList(subset);

        Thread thread = new(() => UploadVideo(videoList));
        thread.Start();
        threads.Add(thread);
    }

    await Task.Run(() => threads.ForEach(t => t.Join()));

    isUploading = false;
    UploadVideosCommand.NotifyCanExecuteChanged();

    System.Windows.Application.Current.Dispatcher.Invoke(() =>
    {
        folderPaths.Clear();
        FoldersToUpload.Clear();
    });

    Logger.Log("[Producer] Upload complete.");
}
```

Key Implementation Steps

Producer Implementation

- UploadVideo(List<string> videoSubset)
  - Connects to the Consumer via TCP
  - Sends how many files will be uploaded
  - For each file:
    - Video Compression
    - Sends filename and hash (skips duplicates)
    - Waits for Consumer's response
      - 0 → already exists → skip
      - 2 → queue full → retry later
    - Sends file

```
private void UploadVideo(List<string> videoSubset)
{
    List<string> retryList = new();

    try
    {
        using TcpClient client = new();
        Logger.Log($"[Producer] Connecting to consumer at: {ConsumerIP}:5001");
        client.Connect(ConsumerIP, 5001);
        using NetworkStream stream = client.GetStream();
        using BinaryWriter writer = new(stream);
        using BinaryReader reader = new(stream);

        writer.Write(videoSubset.Count);

        foreach (var path in videoSubset)
        {
            string compressedPath;
            try
            {
                compressedPath = CompressVideo(path);
            }
            catch (Exception ex)
            {
                Logger.Log($"[Producer] Compression failed for {Path.GetFileName(path)}: {ex.Message}");
                continue;
            }

            string fileName = Path.GetFileName(compressedPath);
            byte[] fileNameBytes = Encoding.UTF8.GetBytes(fileName);
            writer.Write(fileNameBytes.Length);
            writer.Write(fileNameBytes);

            int serverResponse = reader.ReadInt32();
            if (serverResponse == 0)
            {
                Logger.Log($"[Producer] Skipped (duplicate): {fileName}");
                continue;
            }
            else if (serverResponse == 2)
            {
                Logger.Log($"[Producer] Rejected (queue full): {fileName}");
                retryList.Add(path);
                continue;
            }

            writer.Write(fileData.Length);
            stream.Write(fileData, 0, fileData.Length);

            Logger.Log($"[Producer] Uploaded: {fileName}");
        }
    }
    catch (Exception ex)
    {
        Logger.Log($"[Producer] Upload error (batch): {ex.Message}");
    }

    if (retryList.Count > 0)
    {
        Logger.Log($"[Producer] Retrying {retryList.Count} rejected files after delay...");
        Thread.Sleep(3000);
        UploadVideo(retryList);
    }
}
```

Key Implementation Steps

Consumer Implementation

- StartServer()
  - Listens on port 5001.
  - Accepts incoming producer connections.
  - For each connection, starts HandleProducer().

- HandleProducer()
  - Receives metadata and video bytes.
  - Checks for duplicates via hash.
    - Compressed videos are not considered duplicates
  - Enqueues accepted videos.
  - Responds to producer with status.

```
private void HandleProducer(TcpClient client)
{
    try
    {
        using NetworkStream stream = client.GetStream();
        using BinaryReader reader = new(stream);
        using BinaryWriter writer = new(stream);

        int fileCount = reader.ReadInt32();

        for (int i = 0; i < fileCount; i++)
        {
            int fileNameLen = reader.ReadInt32();
            string fileName = Encoding.UTF8.GetString(reader.ReadBytes(fileNameLen));

            int hashLen = reader.ReadInt32();
            byte[] hash = reader.ReadBytes(hashLen);

            // Only check first
            if (ConsumerQueue.IsDuplicate(hash))
            {
                Logger.Log($"[Server] Rejected (duplicate): {fileName}");
                writer.Write(0); // Duplicate
                continue;
            }

            if (!ConsumerQueue.GetSlot())
            {
                Logger.Log($"[Server] Rejected (queue full): {fileName}");
                writer.Write(2);
                continue;
            }

            writer.Write(1);

            int fileSize = reader.ReadInt32();
            byte[] fileData = reader.ReadBytes(fileSize);

            ConsumerQueue.FinishEnqueue(fileName, fileData, hash);
            Logger.Log($"[Server] Enqueued: {fileName}");
        }
    }
    catch (Exception ex)
    {
        Logger.Log($"[Server] Upload error: {ex.Message}");
    }
    finally
    {
        client.Close();
    }
}
```

Key Implementation Steps

Consumer Implementation

- StartConsumerThreads()
  - Starts consumer threads.

- DownloadConsumer()
  - Waits for video in queue.
  - Saves video to disk.
  - Adds it to the UI gallery.
  - Requeues if saving fails.

```
private void DownloadConsumer()
{
    Thread consumerThread = new Thread(() =>
    {
        while (isConsuming)
        {
            if (P3___Networked_Consumer.ConsumerQueue.TryDequeue(out var item))
            {
                string savePath = Path.Combine(saveFolder, item.FileName);
                bool saved = false;

                try
                {
                    File.WriteAllBytes(savePath, item.FileData);
                    saved = true;

                    Dispatcher.Invoke(() =>
                    {
                        AddVideoToGallery(savePath);
                    });
                    Logger.Log($"[Consumer] downloaded + displayed: {item.FileName}");
                }
                catch (Exception ex)
                {
                    Debug.Write($"[Consumer] Failed to save {item.FileName}: {ex.Message}");
                }

                if(!saved)
                {
                    P3___Networked_Consumer.ConsumerQueue.TryRequeue(item);
                    Thread.Sleep(1000);
                }
            }
            else
            {
                Thread.Sleep(1000);
            }
        }
    });

    consumerThread.IsBackground = true;
    consumerThread.Start();
}
```

Queuing System

ConsumerQueue

- ConcurrentQueue<VideoFileItem> uploadQueue
  - core thread-safe queue where video uploads are stored.
  - It ensures multiple threads can enqueue and dequeue items safely.
- HashSet<string> videoHashes
  - Tracks hashes of already uploaded videos to detect duplicates.
  - Uses lock(videoHashes) to avoid race conditions during check/add.
- Semaphore queueLock
  - Enforces the queue size limit (q)
  - The semaphore starts with q permits. Each enqueue consumes one; each dequeue releases one.
  - This is key to rejecting uploads when the queue is full.

```
0 references
public static EnqueueStatus TryEnqueue(string fileName, byte[] fileData, byte[] hash)
{
    string hashString = Convert.ToHexString(hash);

    if (IsDuplicate(hash))
        return EnqueueStatus.Duplicate;

    if (!queueLock.WaitOne(0))
    {
        Logger.Log($"[Queue] Rejected (FULL): {fileName}");
        return EnqueueStatus.Full;
    }

    uploadQueue.Enqueue(new VideoFileItem
    {
        FileName = fileName,
        FileData = fileData,
        Hash = hashString
    });

    lock (videoHashes)
        videoHashes.Add(hashString);

    Logger.Log($"[Queue] Enqueued: {fileName} - Queue size: {uploadQueue.Count}");
    return EnqueueStatus.Success;
}
```

Producer and Consumer Concepts Applied

01

Producer

Spawns threads to send videos

02

Consumer

Accepts or rejects files

03

Queue + Retry Mechanism

Prevents flooding the Consumer

04

Concurrency

Multiple threads upload in parallel

05

Handshake protocol

File upload only proceeds after Consumer approval

06

Deduplication

File hash is checked before sending

Synchronization Mechanisms

ConcurrentQueue<T>

Allows multiple threads to safely enqueue and dequeue without manual locks.

Semaphore

Prevents the queue from overflowing when many producers are enqueueing at the same time.

Locks

Protect againsts race conditions and ensures atomic updates,