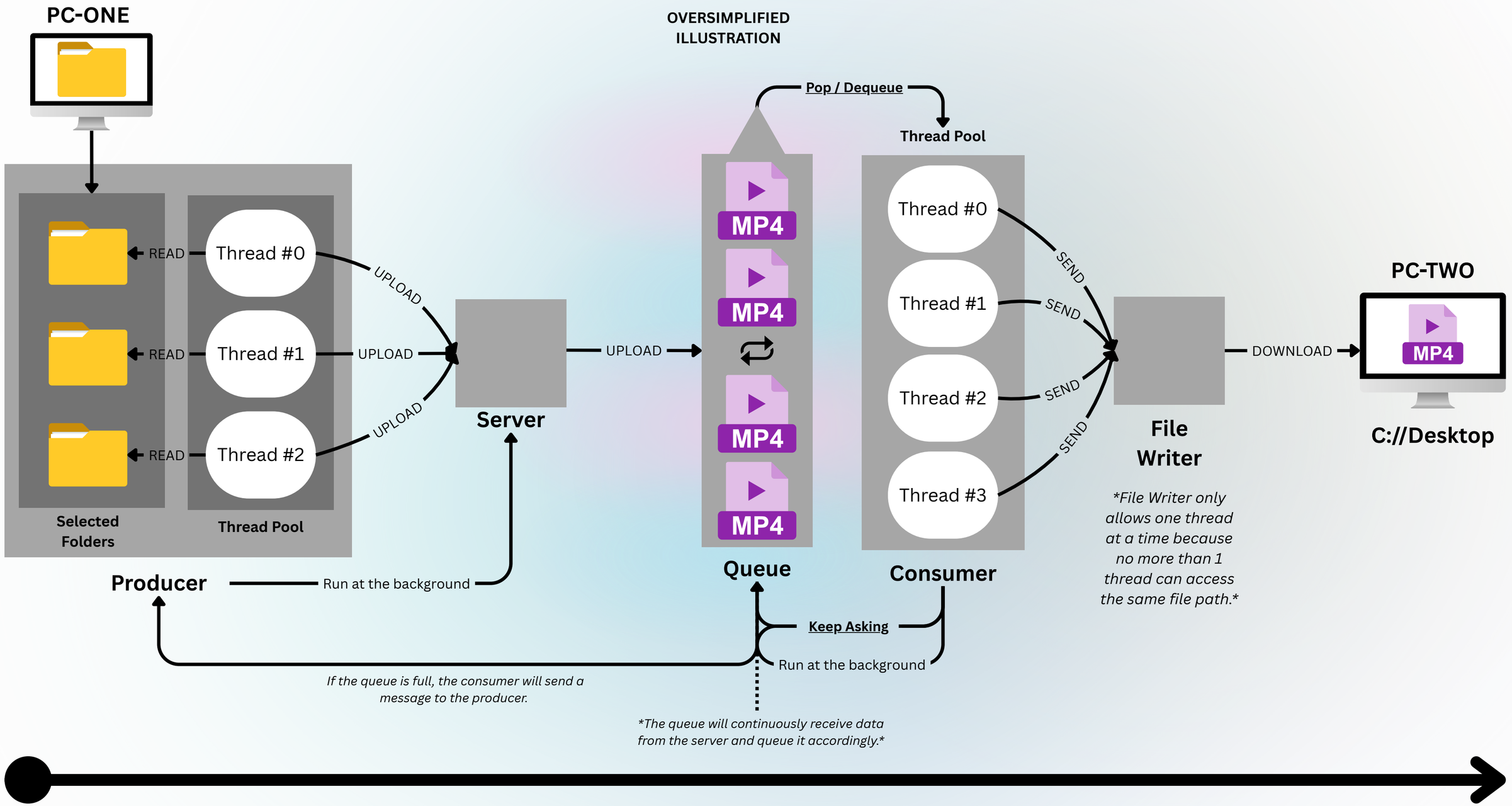
NETWORKED PRODUCER AND CONSUMER

Lopez and Romblon

WHOLE ALGORITHM

OVERSIMPLIFIED ILLUSTRATION



KEY IMPLEMENTATION STEPS

Producer Side

1. Opens the server and acts as the host. (i.e. Peer-to-Peer Connection)
2. Server uses the IPv4 address of the PC rather than the MAC address to host.
3. Runs a background thread to receive messages from the consumer (for the 'queue is full' message).
4. Uses a thread pool to reuse threads for reading different folders. These threads run concurrently with proper synchronization through the use of semaphores to limit the number of threads used.
5. Threads will take turns sending their video files to the buffer/queue on the consumer side (1 video per thread, then switch to another thread, and so on).

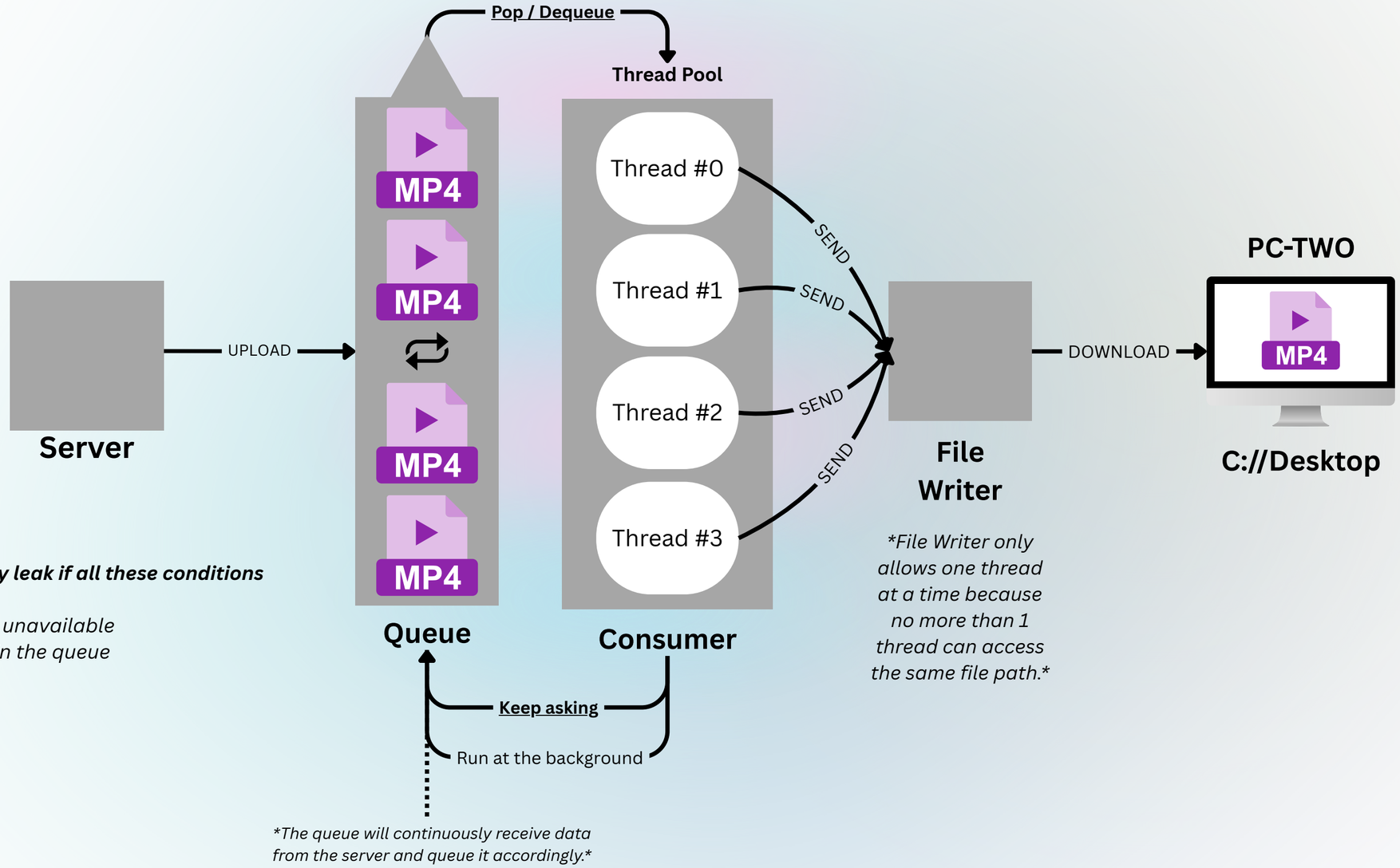
KEY IMPLEMENTATION STEPS

Consumer Side

1. Runs a background thread for:
    a. Queue/Buffer – to continuously receive data from server
    b. User Interface – to continuously refresh the list of thumbnails of the uploaded video(s).
2. Thread pool is used to manage thread memory efficiently for writing to a single folder with proper synchronization (e.g., locks).
3. Queue continuously receives data from the producer and manages it accordingly (e.g. add video to the queue).
4. Queue uses the Leaky Bucket Algorithm
5. Uses FFmpeg to get the first frame from an uploaded video.
6. Uses WindowsMediaPlayer to preview the uploaded video.

QUEUEING

OVERSIMPLIFIED ILLUSTRATION



The queue will only leak if all these conditions are met:

1. *All threads are unavailable*
2. *No free space in the queue*

File Writer only allows one thread at a time because no more than 1 thread can access the same file path.

The queue will continuously receive data from the server and queue it accordingly.

QUEUEING DETAILS

A. Algorithm (Queueing of Videos)

1. As the consumer program starts, a background thread will be assigned to the function responsible for receiving data and queueing videos.

In the function:

1. Receive the file size sent by the producer and assign it to the size of a byte array (byte[]).
2. Receive all the bytes of a video and assign it to the initialized byte array.
3. Lock the queue (This is necessary because the threads in the thread pool also uses this variable.)
4. If there is a space in the queue, queue the video (bytes of the video). Otherwise, skip it and send a message to the producer that the queue is full.
5. Unlock the queue

QUEUEING DETAILS

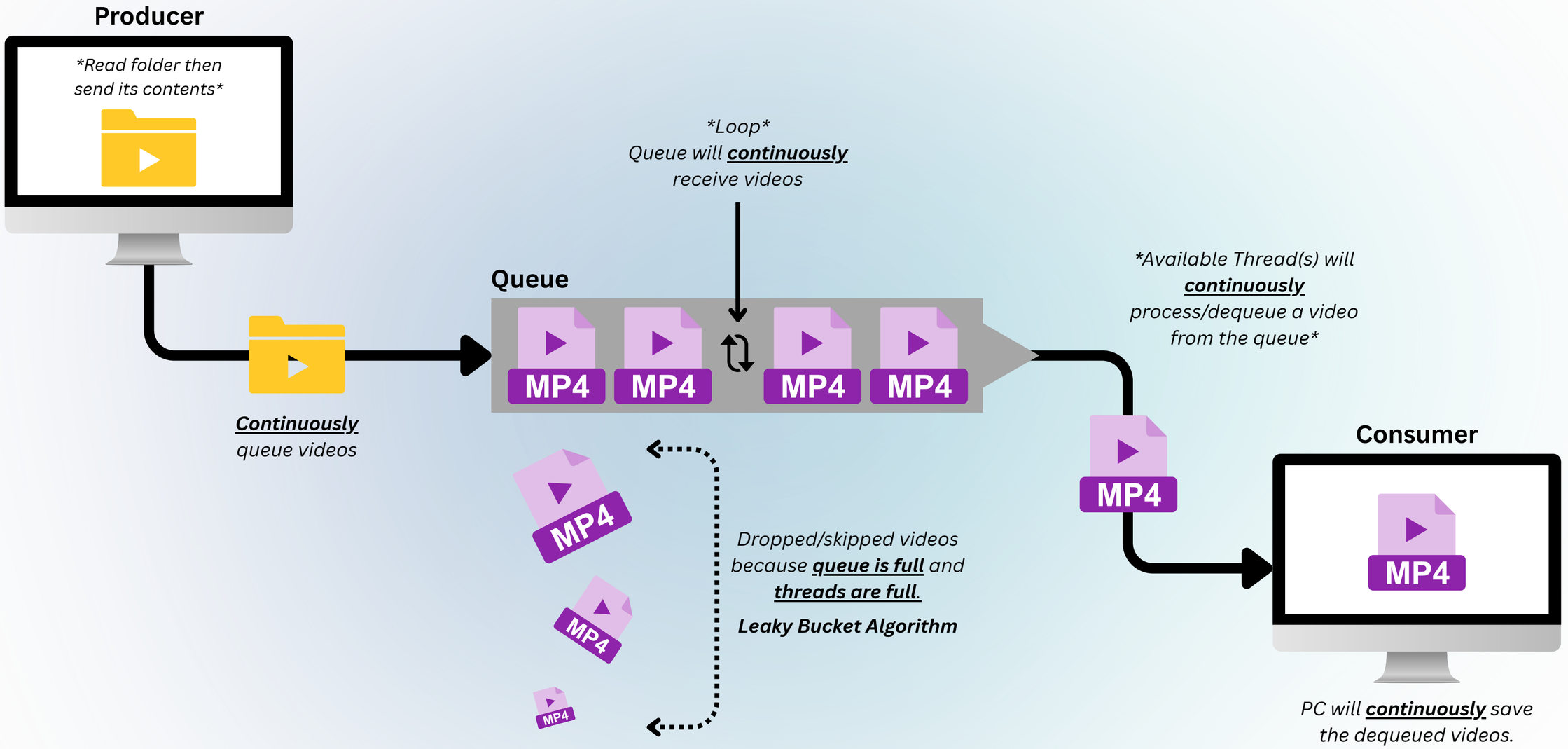
B. Algorithm (<u>Retrieving video from the Queue</u>)

1. N number of consumer threads (in the thread pool) will concurrently run a function responsible for retrieving videos from the queue.
    a. These threads are in a while loop. Thus, it will not end unless there is nothing in the queue.

In the function:

1. Lock the queue
2. Pop or dequeue a video from the queue
3. Unlock the queue
4. Lock file writer (This is necessary since file writer only allows one thread to write in the folder)
5. Save the video in the folder (This is necessary because MD5 needs to open the video before it can generate)
6. Generate an unique MD5 Hash Code of the video
7. If the MD5 hash code is not listed in the collection of videos, generate a thumbnail and add it to the collection. Otherwise, delete the saved video from the folder and do not generate a thumbnail.
8. Unlock file writer

BREAKDOWN OF PRODUCER AND CONSUMER CONCEPTS APPLIED



BREAKDOWN OF PRODUCER AND CONSUMER CONCEPTS APPLIED

Since the queue uses the leaky bucket algorithm, it requires a constant rate of input, which in this case is the continuous flow of videos from the folder.

If the queue is full and all threads are active, the remaining videos uploaded by the Producer will be dropped or skipped and will not be processed by the Consumer.

As videos are being queued, they are also being dequeued simultaneously by the consumer threads.

With the help of thread pools, locks, semaphores, queuing, and the leaky bucket algorithm, solving the Producer–Consumer problem is possible.

SYNCHRONIZATION MECHANISMS USED TO SOLVE THE PROBLEM

Producer Side

- **Thread Pool**
  - To concurrently read the videos inside separate folders without threads being destroyed.
- **Semaphores**
  - Since C# aims to utilize all available hardware resources, it often uses more threads than ThreadPool. SetMaxThreads() if there are more tasks in its queue. By using a semaphore, the number of active threads in the ThreadPool can be limited to the specified number of semaphores.
- **Locks**
  - Most used synchronization mechanism.
  - The most crucial lock is in SendVideoToClient because it must ensure that only one video is sent at a time. Without a lock, bytes from other videos could also be read by File.ReadAllBytes. This can happen because the SendVideoToClient function is used by multiple threads.

```
semaphore = new Semaphore(ConfigParameter.nProducerThreads, ConfigParameter.nProducerThreads);

while (ConfigFolders.foldersFilePath.Count != 0)
{
    // Upload folder
    string folderPath = ConfigFolders.foldersFilePath[0];
    ConfigFolders.foldersFilePath.RemoveAt(0); // Remove the first element

    ThreadPool.QueueUserWorkItem(state =>
    {
        // Wait for an available slot in the semaphore (blocking)
        semaphore.WaitOne();

        try
        {
            Interlocked.Increment(ref ConfigParameter.taskCounter);
            UploadFolder(folderPath);
        }
        finally
        {
            semaphore.Release();
            if (progressBar.InvokeRequired)
            {
                progressBar.Invoke(new Action(() =>
                {
                    progressBar.Value += 1;
                }));
            }
        }
    });
}
```

```
public static void SendVideoToClient(string filePath, Producer producer)
{
    lock (socketLock)  // Locking the socket communication
    {
        try
        {
            // 1. Send the file size
            FileInfo fileInfo = new FileInfo(filePath);
            long fileSize = fileInfo.Length;
            byte[] fileSizeBytes = BitConverter.GetBytes(fileSize);
            ClientSettings.selectedSocket.Send(fileSizeBytes);

            // 2. Send the video file
            byte[] fileData = File.ReadAllBytes(filePath);
            if (ClientSettings.selectedSocket != null && ClientSettings.selectedSocket.Connected)
            {
                //producer.LogMessage("[SYSTEM]: Uploading " + filePath + "...");
                ClientSettings.selectedSocket.Send(fileData);
                producer.LogMessage("[SYSTEM]: Video has been successfully uploaded!");
            }
            else
            {
                producer.LogMessage("[ERROR]: Socket is not connected.");
            }
        }
        catch (Exception ex)
        {
            producer.LogMessage($"[ERROR]: Exception occurred: {ex.Message}");
        }
    }
}
```

SYNCHRONIZATION MECHANISMS USED TO SOLVE THE PROBLEM

Consumer Side

- **Thread Pool**
  - To dequeue videos from the queue and write/download them in a single folder.
- **Locks**
  - Most used synchronization mechanism.
  - Used for the Queue variable as this variable is widely used around functions.
  - Used for the file writer since it only allows one thread to write video bytes to the folder at a time. Without this lock, an error will occur due to multiple threads trying to access the file writer simultaneously.