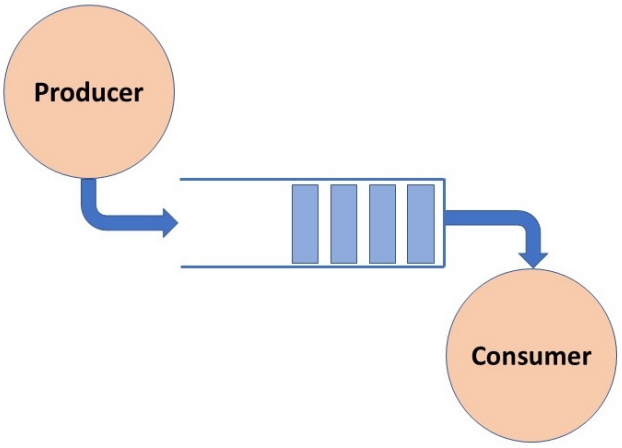
LAB 2

DISTRIBUTED SOFTWARE SYSTEMS

SCENARIO: TWO PRODUCERS - ONE CONSUMER

The producers will generate data, serialize it to JSON, and place it in a shared queue.

The consumer will retrieve items from the queue, deserialize them from JSON, and process them.

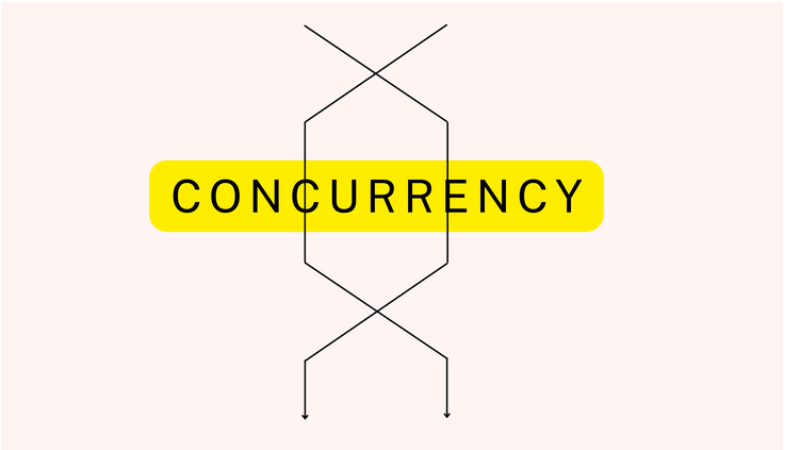


ASPECTS TO TAKE CARE: CONCURRENCY

A producer shouldn't be able to put data in the queue if it is full.

A consumer shouldn't be able to consume data from the queue if it is empty.

If any of those happens it would lead to a concurrency corruption and as a consequence an execution error



SOLUTION: USING LOCKS

```
lock = threading.Lock()
not_full = threading.Condition(lock)
not_empty = threading.Condition(lock)
```

PRODUCER PSEUDOCODE USING LOCKS

```
def producer_structure():
    with not_full:
        #A producer is locked while the queue is full
        while my_queue.full():
            not_full.wait()
        #If is not full, producer pushes serialized data
        my_queue.put("")
        #We unlock the consumer if is waiting for an element in the queue
        not_empty.notify()
```

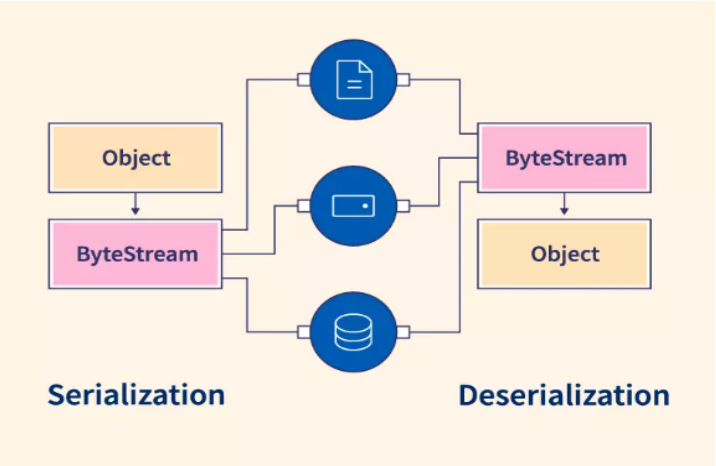
CONSUMER PSEUDOCODE USING LOCKS

```
def consumer_structure():
    with not_empty:
        #A consumer is locked while the queue is empty
        while my_queue.empty():
            not_empty.wait()
        #If is not empty, consumer pops data and desserializes it
        data = my_queue.get()
        #Consumer unlocks producer if is waiting for space in the queue to push data
        not_full.notify()
```

WHY SHOULD WE SERIALIZE DATA

Because the data is converted into a standardized format that can be sent reliably and then reassembled correctly on the receiving side.

So serialization becomes essential when you have to transmit information across a network or share it between various parts of a distributed system.

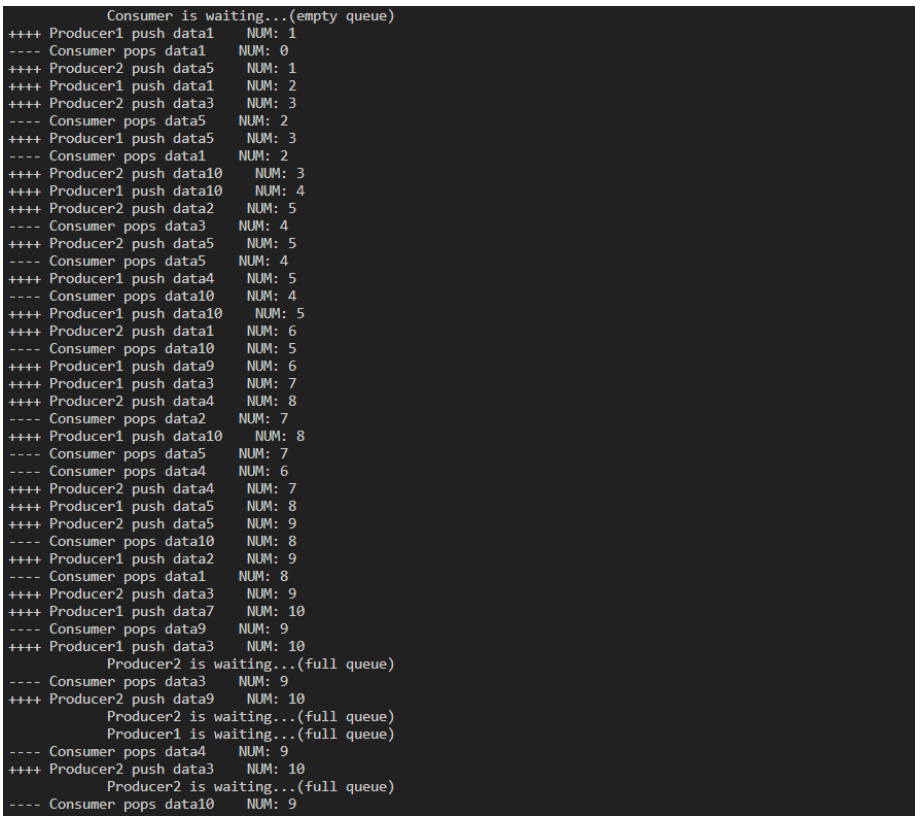


THE CODE

```
c# Import necessary modules
import json         # Module for working with JSON (serialization and deserialization)
import queue        # Module for implementing queues
import random       # Module for generating random numbers
import threading    # Module for working with threads
import time         # Module for working with time

# Create a queue with a maximum capacity of 10 elements
my_queue = queue.Queue(maxsize=10)
# Create a lock object
lock = threading.Lock()
# Create conditions based on the lock to control the full and empty states of the queue
not_full = threading.Condition(lock)
not_empty = threading.Condition(lock)

# Define a function for producer 1
def producer1():
    while True:
        # Create a dictionary with producer information and a random value
        dictionary = {"producer": "Producer1", "value": random.randint(1, 10)}
        # Serialize the data to JSON format
        serialized_data = json.dumps(dictionary)
        with not_full:
            # Wait if the queue is full
            while my_queue.full():
                print("            Producer1 is waiting...(full queue)")
                not_full.wait()
            # Add data to the queue
            my_queue.put(serialized_data)
            print("++++ Producer1 push data{}    NUM: {}".format(dictionary['value'], my_queue.qsize()))
            # Notify the consumer that the queue is not empty
            not_empty.notify()
        # Sleep for a random time before the next iteration
        time.sleep(random.randint(1, 15))

# Define a function for producer 2
def producer2():
    while True:
        dictionary = {"producer": "Producer2", "value": random.randint(1, 10)}
        serialized_data = json.dumps(dictionary)
        with not_full:
            while my_queue.full():
                print("            Producer2 is waiting...(full queue)")
                not_full.wait()
            my_queue.put(serialized_data)
            print("++++ Producer2 push data{}    NUM: {}".format(dictionary['value'], my_queue.qsize()))
            not_empty.notify()
        time.sleep(random.randint(1, 15))

# Define a function for the consumer
def consumer():
    while True:
        with not_empty:
            while my_queue.empty():
                print("            Consumer is waiting...(empty queue)")
                not_empty.wait()
            data = my_queue.get()
            # Deserialize the data from JSON
            desserialized_data = json.loads(data)
            print("---- Consumer pops data{}    NUM: {}".format(desserialized_data['value'],
my_queue.qsize()))
            # Notify the producer that the queue is not full
            not_full.notify()
        time.sleep(random.randint(1, 15))

# Main function
def main():
    # List to store threads
    threads = []

    # Create a thread for the consumer
    c = threading.Thread(target=consumer)
    threads.append(c)

    # Create threads for the producers
    p1 = threading.Thread(target=producer1)
    threads.append(p1)

    p2 = threading.Thread(target=producer2)
    threads.append(p2)

    # Start all threads
    for t in threads:
        t.start()

    # Wait for all threads to finish
    for t in threads:
        t.join()

# Run the main function if the script is executed as the main program
if __name__ == "__main__":
    main()

  return go(f, seed, [])
}
```

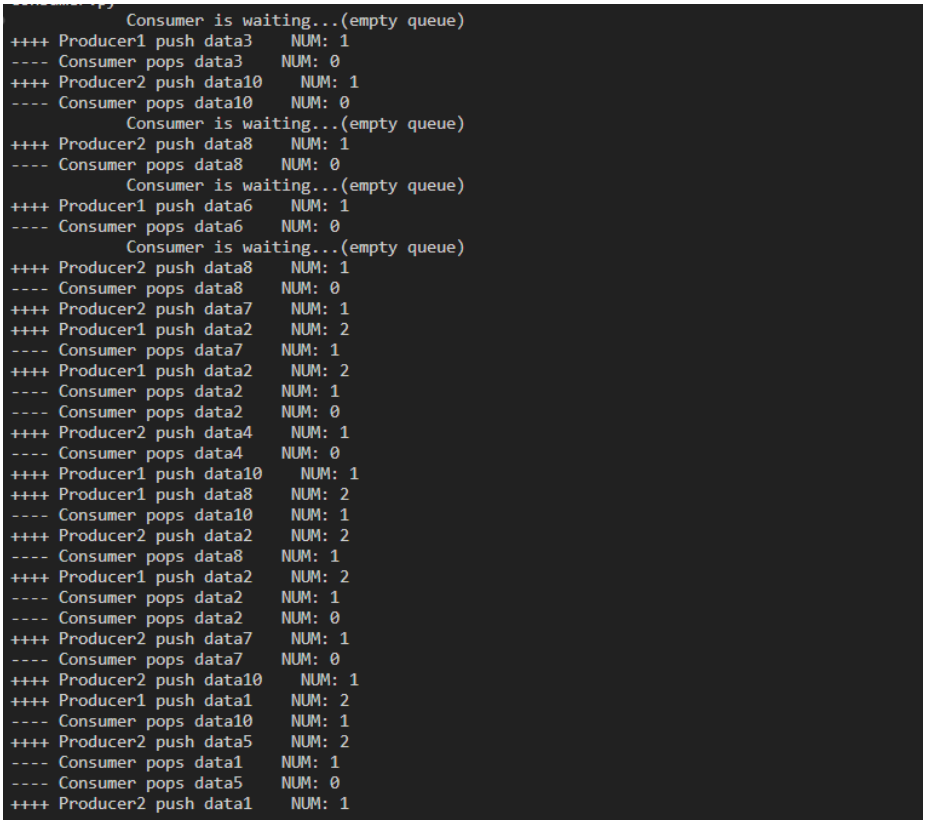
INFINITE EXECUTION

(1)IF THE CONSUMER HAS LESS SLEEP TIME



```
           Consumer is waiting...(empty queue)
++++ Producer1 push data1    NUM: 1
---- Consumer pops data1    NUM: 0
++++ Producer2 push data5    NUM: 1
++++ Producer1 push data1    NUM: 2
++++ Producer2 push data3    NUM: 3
---- Consumer pops data5    NUM: 2
++++ Producer1 push data5    NUM: 3
---- Consumer pops data1    NUM: 2
++++ Producer2 push data10    NUM: 3
++++ Producer1 push data10    NUM: 4
++++ Producer2 push data2    NUM: 5
---- Consumer pops data3    NUM: 4
++++ Producer2 push data5    NUM: 5
---- Consumer pops data5    NUM: 4
++++ Producer1 push data4    NUM: 5
---- Consumer pops data10    NUM: 4
++++ Producer1 push data10    NUM: 5
++++ Producer2 push data1    NUM: 6
---- Consumer pops data10    NUM: 5
++++ Producer1 push data9    NUM: 6
++++ Producer1 push data3    NUM: 7
++++ Producer2 push data4    NUM: 8
---- Consumer pops data2    NUM: 7
++++ Producer1 push data10    NUM: 8
---- Consumer pops data5    NUM: 7
---- Consumer pops data4    NUM: 6
++++ Producer2 push data4    NUM: 7
++++ Producer1 push data5    NUM: 8
++++ Producer2 push data5    NUM: 9
---- Consumer pops data10    NUM: 8
++++ Producer1 push data2    NUM: 9
---- Consumer pops data1    NUM: 8
++++ Producer2 push data3    NUM: 9
++++ Producer1 push data7    NUM: 10
---- Consumer pops data9    NUM: 9
++++ Producer1 push data3    NUM: 10
          Producer2 is waiting...(full queue)
---- Consumer pops data3    NUM: 9
++++ Producer2 push data9    NUM: 10
          Producer2 is waiting...(full queue)
          Producer1 is waiting...(full queue)
---- Consumer pops data4    NUM: 9
++++ Producer2 push data3    NUM: 10
          Producer2 is waiting...(full queue)
---- Consumer pops data10    NUM: 9
```

(2)IF THE CONSUMER HAS LESS SLEEP TIME



```
           Consumer is waiting...(empty queue)
++++ Producer1 push data3    NUM: 1
---- Consumer pops data3    NUM: 0
++++ Producer2 push data10    NUM: 1
---- Consumer pops data10    NUM: 0
           Consumer is waiting...(empty queue)
++++ Producer2 push data8    NUM: 1
---- Consumer pops data8    NUM: 0
           Consumer is waiting...(empty queue)
++++ Producer1 push data6    NUM: 1
---- Consumer pops data6    NUM: 0
           Consumer is waiting...(empty queue)
++++ Producer2 push data8    NUM: 1
---- Consumer pops data8    NUM: 0
++++ Producer2 push data7    NUM: 1
++++ Producer1 push data2    NUM: 2
---- Consumer pops data7    NUM: 1
++++ Producer1 push data2    NUM: 2
---- Consumer pops data2    NUM: 1
---- Consumer pops data2    NUM: 0
++++ Producer2 push data4    NUM: 1
---- Consumer pops data4    NUM: 0
++++ Producer1 push data10    NUM: 1
++++ Producer1 push data8    NUM: 2
---- Consumer pops data10    NUM: 1
++++ Producer2 push data2    NUM: 2
---- Consumer pops data8    NUM: 1
++++ Producer1 push data2    NUM: 2
---- Consumer pops data2    NUM: 1
---- Consumer pops data2    NUM: 0
++++ Producer2 push data7    NUM: 1
---- Consumer pops data7    NUM: 0
++++ Producer2 push data10    NUM: 1
++++ Producer1 push data1    NUM: 2
---- Consumer pops data10    NUM: 1
++++ Producer2 push data5    NUM: 2
---- Consumer pops data1    NUM: 1
---- Consumer pops data5    NUM: 0
++++ Producer2 push data1    NUM: 1
```

PRODUCERS-CONSUMERS IN A DISTRIBUTED SYSTEM

Distributed Social Network: **INSTAGRAM**

The working of a social network requires of different servers that manage user authentication, content publishing and notification delivery for example.

THE “PRODUCERS”(Content publishing nodes)

Each user on the social network is a potential producer, generating content such as posts, photos or videos.

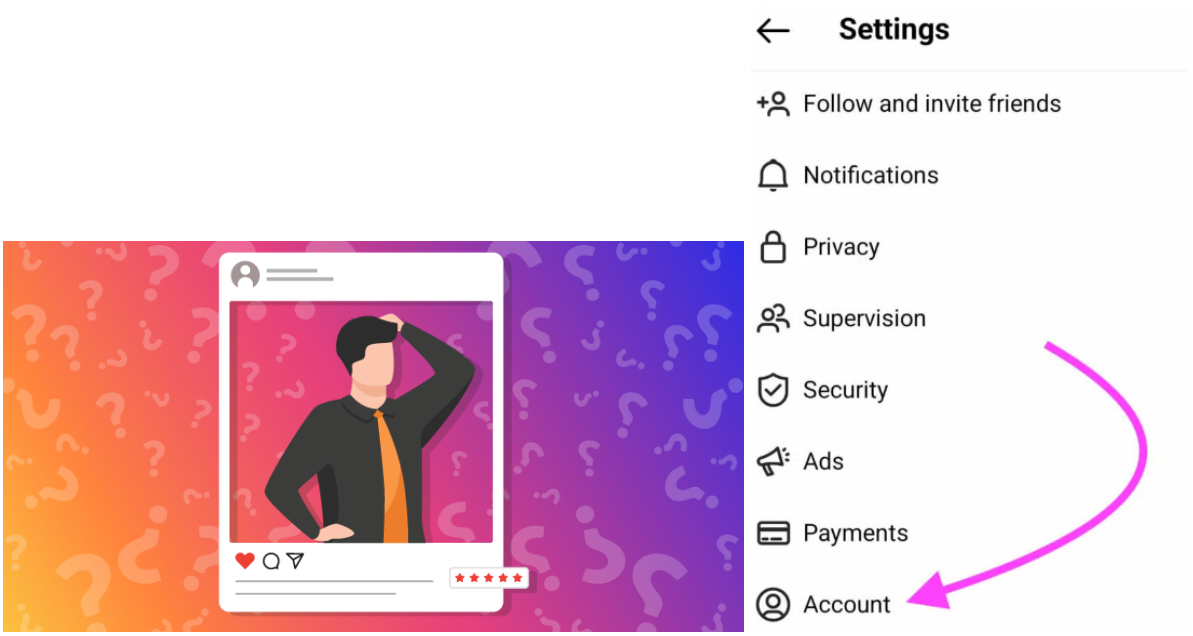
Content publishing nodes generate new publication data and send it to the distributed system.

So content publishing nodes act as producers that generate data in the form of new posts and changes to user data.

THE “QUEUE”(posts and user data)

There is a publication queue and a data user queue that act as a **shared buffer.**

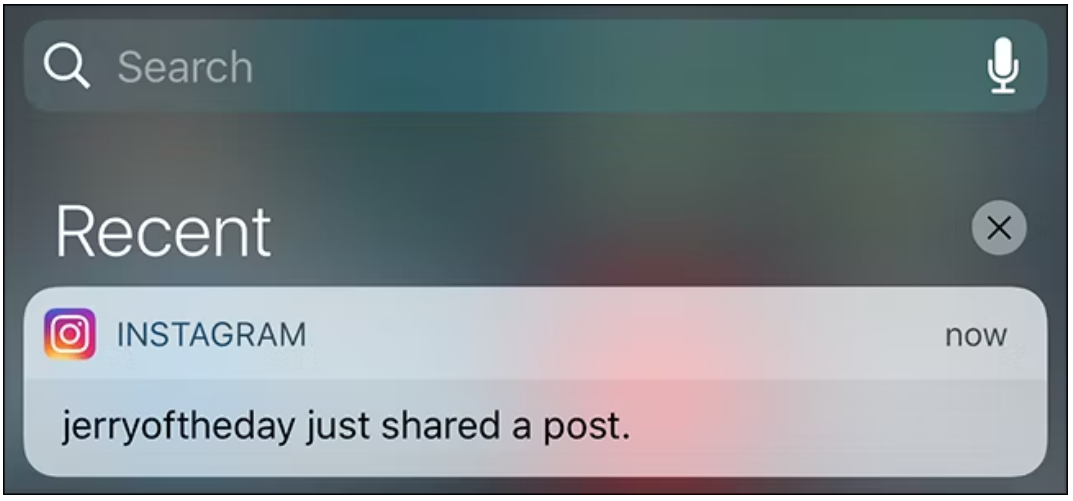
The data from new posts and user data (such as changes to profile settings) are temporarily stored in these queues before being processed by consumers.



THE “CONSUMERS” (Notification and Authentication Processing Nodes)

Centralized processing nodes receive data from new posts and user data.

They process new posts to ensure their visibility according to privacy settings and generate notifications for the followers, so they act as the consumers.



SUMMARY

So as we see we can apply the producers-consumers problem in the real life as we have seen with the social network distributed system in which users are the producers, shared buffer work as the queue and the processing data nodes as the consumers, involving the generation and processing of data in a distributed system, where effective coordination is essential for optimal performance.