# The Producer-Consumer Problem

The purpose of this document is to come upon a concurrency solution for the collection and storage of device data in Chariot. As this is the main purpose of the system and will be expecting data collection from multiple devices, Chariot must utilize concurrency to obtain data in a timely manner. Our major issue with Chariot’s concurrency falls upon the Producer-Consumer problem, in which there is at least one class of threads, producers, add data to a shared collection object, while another class of threads, consumers, retrieve this data. The producers should stop adding data if the collection is filled, and consumers should only retrieve data when the collection is not empty. Now that that short explanation is out of the way I will now go over potential solutions and what I believe to be the best solution for Chariot.

# Python Threading/Multiprocessing Methods and Concurrency Terms

The following section will just go over briefly the various objects and methods that will be used, for full documentation please refer to <https://docs.python.org/3/library/concurrency.html>. This section can be skipped as I try to keep the concepts general enough for those not familiar with Concurrent Programming Paradigms.

## Terms

Scheduler – a program usually associated with an Operating System that determines in what order programs are executed and for how long.

Concurrency – the ability for different parts of a program to execute out-of-order or in partial order without affecting the final outcome

Serial – execution one after the other in order, how most programs work

Parallel – execution of different tasks at the same time, possible with multiple CPU cores

Asynchronous – tasks executed without a predetermined order, jumping to different tasks based on a scheduler

Thread – a sequence of a programed instruction that can be managed independently of a scheduler

Process – a program scheduled by a scheduler to run for the duration of its execution or time slice whichever is shorter.

Deadlock – when two or more threads prevent each other from continuing execution

Starvation – when a thread is prevented from execution, or access to a resource because other threads have access to it. This can be caused by a thread never releasing the resource, or it can just not get access because other threads are faster in obtaining the resource, cut in line.

## Threading

Python’s Thread module, due to the Global Interpreter Lock, these threads work asynchronously, rapidly switching between each other. Similar to Goroutines from Go. Still very efficient as we are mainly I/O bound not CPU bound. Provides synchronization objects like Locks, Semaphores and so on.

Thread.start() – starts a thread’s run() method

Thread.run() – an overridable method in a thread subclass, this is where we will write the code, we want the thread to loop through

Thread.join() – waits until the thread terminates, will stop whatever thread called this from continuing . Don’t use if the thread should be running indefinitely.

## Queue

Python’s synchronized queues. The default class uses a FIFO model, which is the best for our system. Each queue is safe for multi-producer and multi-consumer implementations.

Queue.put() – puts an item into the queue

Queue.get() – retrieves an item from the queue, removing it

Queue.empty()/Queue.full() – Test booleans

## Multiprocessing

This module gets around Python’s GIL issue by spawning new processes. This allows for multiprocessor use, and for networking to other devices to use their processes and threads. Helpful if a user wanted to use a cloud platform. The Process API is purposefully made to copy the API of threads. There are also versions of synchronization objects like Locks and Queues also made in this module. Due to the way processes do not share data, Manager server objects are necessary for the sharing of resources.

# Solutions

## Solution 1: Single Collection

A close up of a map

Description automatically generated

The simplest solution for us would be to use a single shared collection. All IoT device adapters will be separate producer threads all taking the data from their respective devices and appending it to a single collection object. Synchronization and data safety of the collection would be maintained using a Queue object. Saving us from the added effort of creating Locks or other synchronization objects. A single reader thread would be tasked with getting data from this queue and then writing the indices to the database as new records. This can be performed using the queue directly, blocking producers from adding to it, or by making a copy of the queue for use with sending data.

This is the most naïve solution, and as such is the worst solution, only working as a proof of concept with a single producer. Due to the locking situation, devices would block each other constantly. That means data will be lost while waiting for a chance to add to the queue. Furthermore, there is potential for a large enough network to starve adapters access to this queue, preventing the collection of any data from its associated device.

Only use this if you are just testing out how Queues and Threads work in Python, not on Chariot.

## Solution 2: One Queue Per Adapter

A close up of a map

Description automatically generated

This solution creates a queue for every producer thread but will still use a single consumer thread. Due to each device having their own polling rate they will fill the queue at different intervals. If they had to do this following the same time slices as provided by the CPU, one set of adapters would dominate the queue while another will have less data available. With a queue to themselves, it doesn’t matter if there exists a device with a faster polling rate, as only one adapter has access to put data into the queue, allowing the queues to grow at different rates.

The consumer thread will then traverse down a list of these queues, getting data and then passing it on to the database. If a queue is currently empty, it can ignore that thread and move on to the next one, continuing this loop until the DCE terminates.

This solution solves the issue of adapters having data lost, but now brings the issue that a single consumer thread can be overwhelmed by a faster adapter. I purposely did not differentiate whether the consumer would read a queue until it is empty or if it will only get a subset of that queue first. Either way the data will not be saved in real time and can actually cause a significant delay depending on how large the network is. For our testing cases the delay would probably be too insignificant to notice, but if we wait until a queue is empty, a slower queue can get starved for attention by a previous queue being too large. Time is also lost by the writing to a database, allowing more to be made.

## Solution 3: One Queue Per Adapter into a Single Collection

A close up of a map

Description automatically generated

This solution combines solutions 1 and 2 to preserve as much data as possible and allow for asynchronous collection and storage of data. Like in solution 2 we will have every device adapter producer thread put data into a single queue only accessed by it and the consumer thread. The consumer thread will on a timer poll the different device queues. During this polling phase, the consumer will collect all of the data in the queue, and then append it to an existing list, only accessed by the consumer. This allows producers to keep adding to their respective queues, while the consumer can add records to the database in a timely manner.

This is our best solution as it requires the least amount of interaction between the threads, protecting them from starving or deadlocking one another. There is still a risk with this solution as the consumer can build a list of writable data that is so large, that it takes a noticeable amount of time for it to actually write to the database, getting away from our real time goal. In actuality our real time means, that data has a second to pass from a device to the database, which in terms of computer processing is more than enough time to do anything, but just food for thought when it is a large network.

## Optional Additions:

While not full solutions in themselves, there are some additions we can make to improve these solutions, but they also bring upon their own issues.

Currently all solutions utilize a single consumer, while this prevents issues with too many threads accessing data, it does create a bottleneck to the writing of data to the database. One addition for Solution 3 is the splitting of the readers duties to that of two threads. One reads data from adapter queues, and the stores it in another centralized queue. Another DBWriter thread would have the duty of only reading from the centralized queue and writing to the database, allowing for the potential of constant usage of the database, which is a goal for most efficient usage of our databases. The downside is that if the devices are not polling at a high enough rate, either consumer can spend even more time doing nothing, leaving two threads on the CPU with nothing to do in particular.

Another solution instead of two threads doing different consumer tasks, we make multiple threads to read from adapters and write to the database. We will have to create an algorithm for chariot to dynamically determine the most efficient number of consumers to producer ratio. These consumers could either share all adapters, and access them when they are available, or be left to only check on a subset of adapters. These would then have their own lists to write to and then use write records to the database.