# CS131, Spring 2018
# Project Report

## 1 Abstract

An application server herd is an architecture in which multiple application servers communicate with each other as well as a database server. The communication propagates rapidly evolving data so that each server could handle appropriate tasks to improve parallelism, concurrency, and user response. For example, users at different geo-locations may query local endpoints and caches instead of slow read-writes to the database.

The architecture is inherently asynchronous, so libraries such as Python's `asyncio` or Node.js's `async` functions are candidates for implementation. Realistically, a Python backend analogous to Node.js would be Django or Flask, which handle asynchrony separately from `asyncio`. For our purposes here forth, we will discuss `asyncio`.

## 2 Overview

Both Python and Node.js implementations operate by using an event loop, which is a queue of references to functions to run whenever the system is idle. This does not necessarily require multi-threading. However, it limits blocking of any particular operation, so it increases efficiency and responsiveness.

Before version 7.6[1], Node.js relied on Javascript `Promise` to handle this task, which has syntax quite different than the current iteration of `asyncio`. In fact, it is closer to `Task` before Python version 3.4[2]. A Promise in Javascript is an object that ensures execution of its executor function before passing either a `resolve` or `reject` function. This function may be another `Promise`, allowing it to be chainable[3]. Analogously, this is Python's `create_task()` and `ensure_future()`.

The new syntax simplifies it somewhat by making the event loop more transparent. In Python, `async` coroutines return objects which could be resolved at a later time with the `await` keyword. Co-routines must be called within the event loop, or added explicitly to the event loop[2]. Node.js also introduce keywords `async` and `await` as wrappers that convert a function return to `Promise` behind the scenes[1]. The newer syntax is similar on both sides, and although the older syntax is different, their implementations are similar[4].

## 3 Application Herd Implementation

Implementation of the application server herd using either `asyncio` or Node.js' equivalent is straight forward. The asynchrony uses minimal boilerplate to create the event loop, and schedule initial entry point of the application. This is done in the `Places Server` object constructor in the prototype. A list of server names, port numbers, and their communication rules are stored inside a `ServerConfig` object, accessed through a dictionary. Similarly, user data are stored in `User` objects and accessed through a dictionary. The prototype uses string and regular expressions libraries to parse TCP inputs, with a basic test for correctness. *IAMAT* commands are then propagated to appropriate servers via flooding, when the user location, coordinates, and access time do not match the entries in the dictionary. This ensures that only updates are propagated and prevents an infinite loop of flooding. In the event that communication fails because a server is down, the message and destination are stored in a queue to be run before the next iteration. Logs are made for debugging and oversight purposes. Examples of the logs are contained in the appendix.

The *IAMAT* command establishes user presence. The same user may log in to multiple locations and coordinates at different times, so, given that each command by the user occur more than nanoseconds apart, subsequent commands are updated. Since *IAMAT* is propagated through the server herd, the *WHATSAT* command to any given server queries Google Place API for a Nearby Search of radius and items specified by the user. This is achieved using `aiohttp` request, which returns a JSON (Javascript object notation) object, a dictionary with string keys, manipulated by the `json` Python library. Since the response is just a dictionary containing a list of results, it can be subset as usual.

There were no particular problems during implementation of the prototype. Learning curve was gentle and documentation were a smooth read.

## 4 Threading

Aside from occasional memory bugs, `asyncio` is not known to have any memory issues. This is, in fact, one advantage of the event loop model over traditional multi-threading. The event loop causes the context to be switched at specifically defined entry points rather than nondeterministic intervals between threads[5]. This saves memory and improves resource overhead.

Highly parallel tasks could still benefit from multi-threading, since parallel calculations could be performed over distinct cores. The asynchronous model is still single-threaded, for the most part. However, it is much simpler in implementation, and avoids the common issues of race conditions and deadlocks altogether.

Python still has multi-threading support, of course. The `threading` library provides traditional locks and semaphore support, while the `multiprocessing` library provides a way to take advantage of multiple threads while abstracting away some of these difficulties. For example, the `Pool` object parallelizes execution of a function as a map across an input list[6]. There is also the `ThreadPoolExecutor` that uses pools to schedule calls asynchronously. However, one experiment found that `asyncio` is better in terms of overall performance and memory footprint[7]. The disadvantage, it notes, is that while `asyncio` is simple in concept, it does require pre-existing code base to be converted to use `async` co-routines, while it is relatively painless to add thread support to isolated points using `ThreadPoolExecutor`.

On the other hand, Node.js, being based on Javascript, is inherently single-threaded – a limitation due to the V8 Javascript engine. Despite the name, Javascript is a completely different language than Java, more functional than it's non-web counterpart. It is possible to use Java for the backend, but not as Node.js. `java.util.concurrency` provides direct access to locks and semaphores objects, and higher level thread pool executors, like Python. While it is possible to implement a simple event loop with the reactor pattern[8], there is no native support in Java.

## 5 Type Checking

While for smaller applications, Python dynamic typing decreases prototyping time and boilerplate, it may increase debugging time for larger applications due to hard-to-find run-time bugs.

Fortunately, unit testing can be conceived as a must for defensive programming, and Python has a natively supported `unittest` module for test cases and fixtures, and `pydoc` module for documentation generation. It is also easy to test for variable types using `int()`, `float()`, and `str()` constructors, since it doesn't have any confusing wrapper classes for primitive types like Java, or pointers, references, or templates like C languages.

Compared to Javascript-based Node.js, it is similar, since Javascript also has dynamic types. However, while currying, prototypal inheritance, binding and the such are powerful features, they might throw a curve ball in terms of ease of implementation. It is easier to promote good practices with Python than Javascript.

## 6 Recommendations

There are many good language choices for an application server herd. Python is one such choice. Positively, it has wide native support for a variety of operations, excellent asynchronous module and multi-threading choices. Dynamic typing is a probable downside. It is also slower than a native Java application, since it is interpreted; however, this should never be the bottleneck in a server, which will probably be the database.

Node.js shares many of these downsides, and more. Although it is possible to remedy dynamic types with Typescript, which compiles down to Javascript.

Java is statically typed, but has much more boilerplate, and lower level implementations, and lack of a native event loop makes it harder to use for this purpose.

Therefore, Python is recommended.

## 7 References

[1] Nemeth, Gergely. "Node.js Async Function Best Practices". *Nemeth Gergely*, 23 Oct 2017. https://nemethgergely.com/async-function-best-practices/. Accessed 8 Jun 2018.

[2] Poirier, Dan. "Asyncio". *Dan's Cheat Sheets*, 2017. http://cheat.readthedocs.io/en/latest/python/asyncio.html. Accessed 8 Jun 2018.

[3] Behrens, Matt. "Javascript Promises – How They Work". *Atomic Object*, 16 Feb 2016. https://spin.atomicobject.com/2016/02/16/how-javascript-promises-work/. Accessed 8 June, 2018.

[4] Pradet, Quentin. "Javascript Promises are equivalent to Python's asyncio". *Quentin Pradet*, 2017. https://quentin.pradet.me/blog/javascript-promises-are-equivalent-to-pythons-asyncio.html. Accessed 8 Jun 2018.

[5] Humrich, Nick. "Asynchronous Python: Await the Future". *Hackernoon*, 23 Sep 2016. https://hackernoon.com/asynchronous-python-45df84b82434. Accessed 8 Jun 2018.

[6] "17.2 multiprocessing – Process-based parallelism". *The Python Standard Library*, 8 Jun 2018. https://docs.python.org/3.6/library/multiprocessing.html. Accessed 8 Jun 2018.

[7] Wolf, Jakub. "Memory efficiency of parallel IO operations in Python". *code.kiwi.com*, 13 Mar 2018. https://code.kiwi.com/memory-efficiency-of-parallel-io-operations-in-python-6e7d6c51905d. Accessed 8 Jun 2018.

[8] Salerno, Rafael. "Understanding Reactor Pattern: Thread-Based and Event-Driven". *DZone / Java Zone*, 31 Aug 2016. https://dzone.com/articles/understanding-reactor-pattern-thread-based-and-eve. Accessed 8 Jun 2018.

## 8 Appendix A: Logs

The logs are recorded in the following conditions. Golomon, Hands, Holiday, and Wilkes are started, with Welsh turned off to simulate a server down. The two commands in the specification are given (note that the time differences are expectedly large), and a final invalid command. The commands are

| Command | Destination |
|---|---|
| `IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997` | Golomon |
| `WHATSAT kiwi.cs.ucla.edu 10 5` | Holiday |
| `WHERESAT kiwi.cs.ucla.edu invalid foo` | Holiday |

**Golomon.log**

```
[Incoming] IAMAT kiwi.cs.ucla.edu +34.068930-118.445127
1520023934.918963997
[Hands] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Holiday] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Wilkes] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Client] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Incoming] AT Hands +8482426.891814947 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Incoming] AT Holiday +8482426.893899918 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Incoming] AT Wilkes +8482426.896380663 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
```

**Hands.log**

```
[Incoming] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Golomon] AT Hands +8482426.891814947 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Wilkes] AT Hands +8482426.891814947 kiwi.cs.ucla.edu +34.068930-
118.445127 1520023934.918963909
[Incoming] AT Wilkes +8482426.896380663 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
```

**Holiday.log**

```
[Incoming] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Golomon] AT Holiday +8482426.893899918 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Welsh] Timeout.
[Wilkes] AT Holiday +8482426.893899918 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Welsh] Timeout.
[Incoming] AT Wilkes +8482426.896380663 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
[Welsh] Timeout.
[Incoming] WHATSAT kiwi.cs.ucla.edu 10 5
[Google] https://maps.googleapis.com/maps/api/place/nearbysearch/
json?location=+34.068930,-118.445127&radius=10&key=[redacted]
[Client] AT Holiday +8482426.893899918 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963909
...
[Welsh] Timeout.
[Incoming] WHERESAT kiwi.cs.ucla.edu invalid foo
[Client] ? WHERESAT kiwi.cs.ucla.edu invalid foo
```

**Hands.log**

```
[Incoming] AT Golomon +8482426.885822058 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963909
[Incoming] AT Hands +8482426.891814947 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963909
[Incoming] AT Holiday +8482426.893899918 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963909
[Golomon] AT Wilkes +8482426.896380663 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963909
[Hands] AT Wilkes +8482426.896380663 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963909
[Holiday] AT Wilkes +8482426.896380663 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963909
```