AFRL-RY-WP-TP-2021-0236

TOWARD SCALING APPLICATIONS ON MODERN AVIONICS

Mitchell Bihn
Resilient & Agile Avionics Branch
Spectrum Warfare Division

SEPTEMBER 2021
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2021	Final	26 August 2021 –26 August 2021

4. TITLE AND SUBTITLE
TOWARD SCALING APPLICATIONS ON MODERN AVIONICS

5a. CONTRACT NUMBER
In-House

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Mitchell Bihn

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate (AFRL/RYWA)
Wright-Patterson AFB, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TP-2021-0236

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
PAO case number AFRL-2021-2868, Clearance Date 26 August 2021. This is a work of the U.S. Government and is not subject to copyright protection in the United States. Report contains color.

14. ABSTRACT

As avionic mission systems become more powerful, the environment in which applications are run starts to resemble that of the modern data center. Efficiently utilizing these resources is challenging to coordinate, but offers many opportunities to deploy onboard applications at scales that have not been feasible in the past. This project demonstrates an approach to transforming future mission applications to support such mission systems. We begin by outlining the design of a hypothetical track generation application (constructed by another intern, Abigail Delnoce), specifically discussing features that ensure the application can scale. Next, we describe the implementation of a local, lightweight Kubernetes environment (spread across two physical nodes) that can support scaling our application. From there, we discuss the deployment of the application into the environment we created. The application's scaling properties are characterized and enumerated, and these results are compared to expected outcomes produced analytically (based on the simulation being run). Finally, we explain some notable challenges with the approaches taken, and describe further application changes and architectural modifications that could increase the efficiency and transparency of mission applications that try to adopt such a pattern of design.

15. SUBJECT TERMS
software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Vahid Rajabian-Schwart
Unclassified	Unclassified	Unclassified	SAR	19	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Toward Scaling Applications on Modern Avionics

Mitchell Bihn
Mentor: Steven Eisemann

Sensors Directorate, Air Force Research Laboratory
Wright-Patterson Air Force Base OH
Summer 2021

ABSTRACT

As avionic mission systems become more powerful, the environment in which applications are run starts to resemble that of the modern data center. Efficiently utilizing these resources is challenging to coordinate, but offers many opportunities to deploy onboard applications at scales that have not been feasible in the past. This project demonstrates an approach to transforming future mission applications to support such mission systems. We begin by outlining the design of a hypothetical track generation application (constructed by another intern, Abigail Delnoce), specifically discussing features that ensure the application can scale. Next, we describe the implementation of a local, lightweight Kubernetes environment (spread across two physical nodes) that can support scaling our application. From there, we discuss the deployment of the application into the environment we created. The application's scaling properties are characterized and enumerated, and these results are compared to expected outcomes produced analytically (based on the simulation being run). Finally, we explain some notable challenges with the approaches taken, and describe further application changes and architectural modifications that could increase the efficiency and transparency of mission applications that try to adopt such a pattern of design.

TABLE OF FIGURES

PROJECT DESIGN

Introduction

Many modern applications are written as collections of simple, atomic applications that can be orchestrated to achieve a complex function. These individual applications are often referred to as *microservices*. Choosing an appropriate scope for each individual microservice represents an essential part of building an application that can efficiently scale across modern infrastructure.

In this project, we simultaneously explored the design and implementation of an application as a series of microservices, as well as the orchestration technologies needed to ensure the application could scale. Specifically, we focused on an approach to taking modern principles of application design as it might exist within a data center, and translating that approach to something better suited to running on an aircraft. Considerations included, for example, the limited size, weight, and power of hardware that can be commonly found on aircraft.

Note that, while this document does include an essential subset of information related to the application itself, detailed information on application design and implementation may be found in a separate report [1]. This report focuses only on the elements of design that were necessary to successfully support orchestration. The specific technology chosen to support orchestration, in this case, is called Kubernetes [2].

Kubernetes and K3s

Kubernetes (often abbreviated as k8s) is an open-source container-orchestration system for automating computer application deployment, scaling, and management originally designed by Google [2]. Individual micro services are deployed via these *containers*. Containers are a unit of software that packages up code and all its dependencies so an application can run reliably from one system to another. This allows for applications to be consistent no matter where they are deployed, and allow for easy replication of the application across clusters. For all these reasons, containers make development, testing, management, and deployment more accessible and straightforward. The specific container technology we chose, in this case, was Docker. Docker provides the ability to run an application in containers, which are lightweight and contain everything needed to run the application [3]. Before we progressed too far into our design based on these choices, however, we wanted to ensure that they could run on the hardware being used.

The hardware supporting this project consisted of two physical systems called NUCs. NUCs are small form factor personal computers designed by Intel. These systems have been previously used in drone-based experiments due to their relatively low size, weight, and power (SWaP) profile. While Docker itself seemed adequate to support our microservice-based application, Kubernetes itself seemed relatively heavyweight – its minimum requirements included 2 GB of RAM and 1.5 CPU dedicated to overhead [4]. Given these requirements, we elected to pursue an alternative implementation of Kubernetes that might involve less overhead.

While other implementations were briefly reviewed, K3s seemed like the best option for use here instead of a full distribution of Kubernetes to keep the cluster as lightweight as possible. K3s is a

highly available, certified Kubernetes distribution designed for production workloads in unattended, resource-constrained, remote locations or in IoT appliances [5].

In this deployment, one of the NUCs was designated as the *master* node, and the other as the *agent* node in the cluster configuration. The *master* node was responsible for *scheduling* containers: this process is described in documentation for Kubernetes [6]. Scheduling ensured that containers were distributed fairly across the *master* node as well as all available *agent* nodes. Note that *agent* nodes were entirely passive participants – the *master* node directed them to execute specific tasks, and the *agent* nodes reported on execution status.

Experimental Application

One challenging aspect of building experimental infrastructure is that it is difficult to benchmark without a properly designed application to support it. Thus, another intern designed an experimental application as a collection of containerized microservices. This application was then run within the K3s infrastructure.

A microservice was used to simulate an aircraft flying on a 2-D path: latitude and longitude were considered, but altitude was not in order to simplify the navigation model. This aircraft was equipped with a hypothetical radar system, which is capable of detecting any object (herein referred to as an *entity*) within a user-defined radius of the aircraft's location. Before the simulation began, a list of *entities* of interest was randomly generated and scattered along the flight path. Upon being encountered by the aircraft, each of these entities required a discrete amount of time to process (between 250 milliseconds and 2.5 seconds, in increments of 250 milliseconds). During the simulation, the aircraft tried to process as many of these entities as possible.

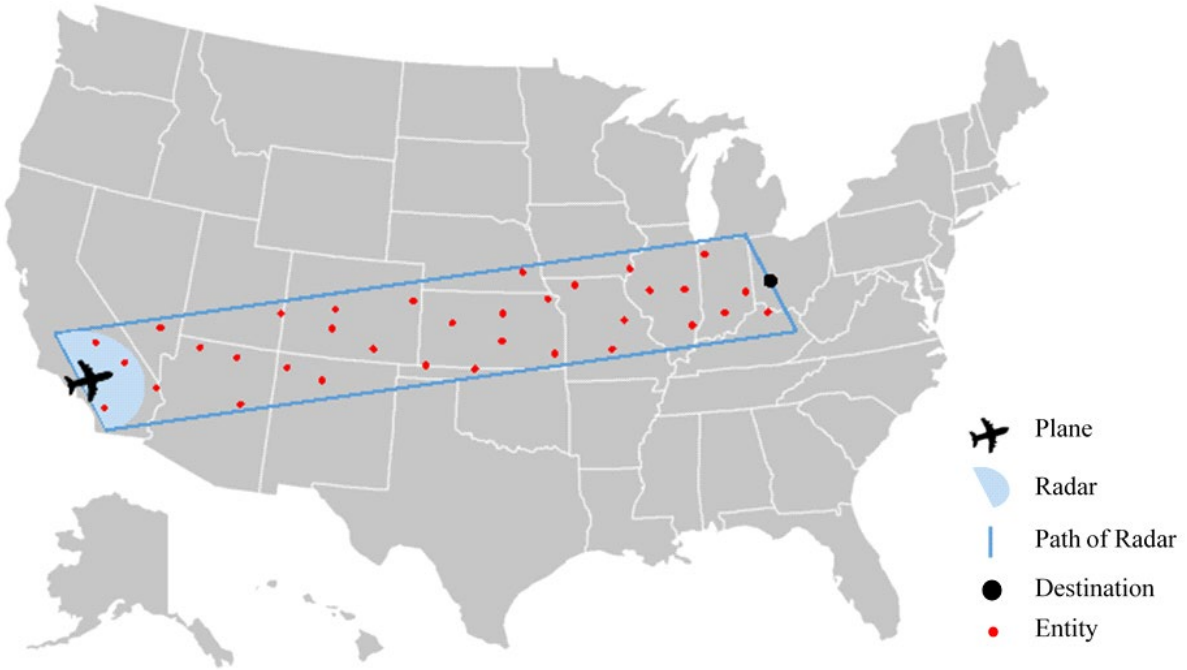

Figure 1 - Overview of Simulation

While the simulation was split into a number of pieces, a relatively large number of entities and the per-entity processing time was chosen to create a processing bottleneck in the system. As such, the entity processing piece (called the *track service*) was the target identified for parallelization. However, supporting parallel operation in a clustered environment required specific design considerations so that the micro services would be able to communicate regardless of where they were deployed.

To support this objective, each service adopted a publish / subscribe model of communication. Specific messages were *published* through an ActiveMQ message broker. Services indicated (*subscribed* to) specific message types that they are interested in. Each time the broker received a published message, the message was forwarded to all active subscribers. This allowed microservices to communicate with each other as long as they had access to the broker.

Such a design allowed the deployment of multiple *instances* (or copies) of the track service. Each instance would individually subscribe to receive aircraft location updates, and could process those updates to determine which entities were near enough to be processed. However, there were some additional considerations related to trying to ensure that each service would process a different set of entities – this was called *load balancing*.

For the sake of simplicity, this project adopted a *static* approach to load balancing. In a *static* approach, the number of instances would be set at the beginning of the simulation, and would remain constant throughout its execution. Supporting a more dynamic approach to load-balancing would have involved the creation of an additional service, as well as substantial logic to support safe on-demand modification to active deployments. This task was determined to be beyond the scope of the task for the summer.

Static load-balancing was also convenient to support relatively deterministic experimentation. A dynamic scheme would have resulted in changing numbers of instances during execution (with corresponding startup and shutdown delay). Therefore, such a scheme would have made it difficult to precisely profile the way that performance scaled per-instance.

The *cluster* configuration can be seen in Figure 2. It consisted of two Intel NUCs (described previously) connected through a 100 Mbps switch. One NUC was identified as the master, and the other NUC was identified as an agent. Note that only one agent was deployed in this case because no additional hardware was available to support additional agents. Deploying an extra agent requires little effort, as K3s makes the process as simple as installing K3s on the new node, and executing a join command with the token from the master node.

Upon each run of the simulation, if a new build of the service is available, the images are rebuilt and pushed into the *registry*. This is a private image registry configured on the master system that has 10 GB partitioned for the images of the containers. Each microservice in the application's images are stored in this registry that can be pushed to and pulled from, to either update the build or retrieve the build upon deployment.

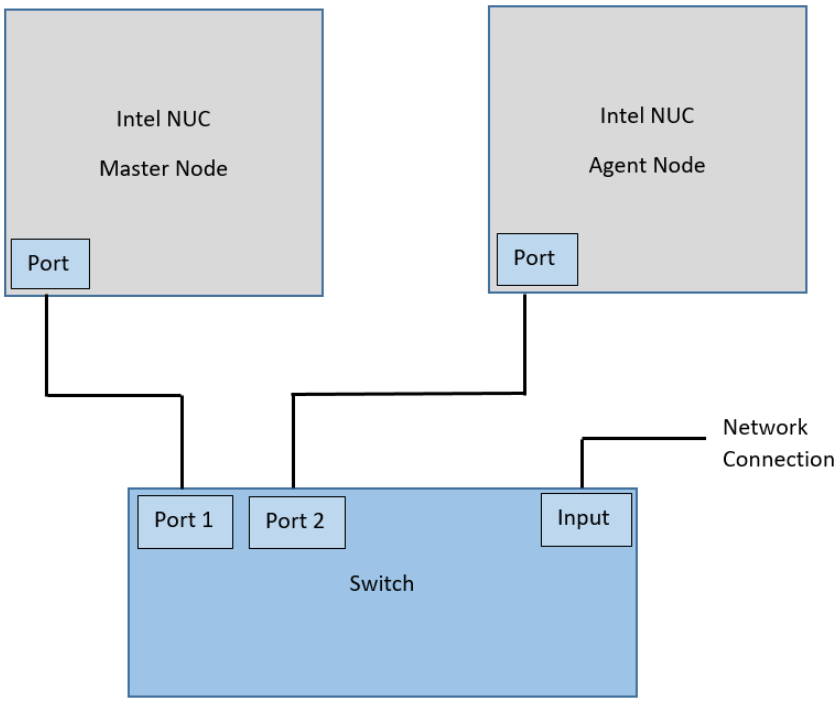

Figure 2 - Cluster Configuration

As mentioned before, an *entity list* has the locations of all the randomly generated entities in the path, and is input into the application. An example of the way these entries are formatted in the list can be seen in Appendix A. From there the services are deployed, and the output is put into log files, which report the total unique entities seen as well as the overall success percentage which is a percentage of the number of entities seen over the total simulated entities.

Scaling the Track Service

Per [1], the *track service* accepted a list of entities as input. Every time the *location service* sent an update, the *track service* would update its own location and begin processing all entities within range. While it was simple to deploy multiple instances of the track service, this would not have supported scaling in and of itself: every track service would have had an identical entity list, and thus all instances would have all processed entities in the same order. This approach would not have led to performance improvements.

Thus, a separate load balancing service (called the *entity list* service) was created to split the total list of entities into pieces, and to provide each instance of the track service with its own slice of that list. This service was written on top of Flask, a minimalist web framework for Python [7]. Flask allows for rapid and concise web service development: the total size of the *entitiy list* service backend is approximately 100 lines of code.

To utilize this service, the track service container was modified to include a start script. This script would run when the container was deployed, and would make an HTTP GET request to

the entity list service. The entity list service would return a small piece of the total list as a response, which was saved within the container as the entities.txt file that the *track service* used to determine which entities it was responsible for processing. An example of this process based on division of the list across four active track service instances is described in Figure 3 (below).

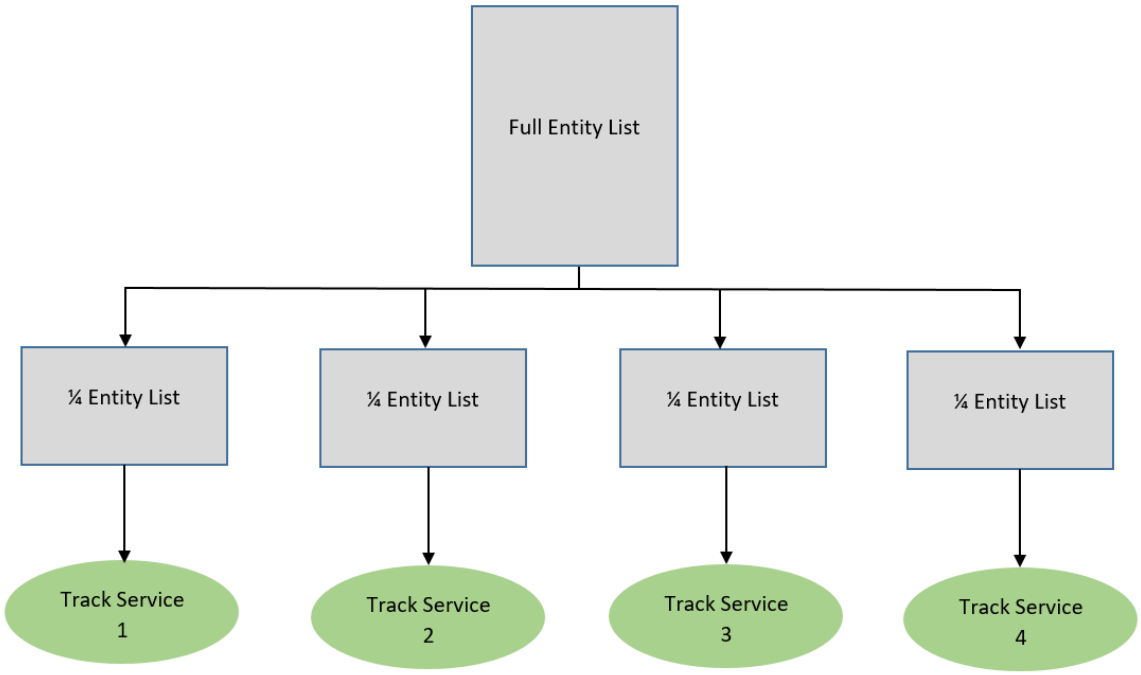

Figure 3 - Entity List Service Diagram

Application Deployment

At the beginning of this process, all services were deployed manually. This process proved to be labor-intensive and error-prone. With this in mind, this process was identified as a target for automation. After some review, we identified and successfully tested a Python library that could interact directly with Kubernetes [8]. From this library, we implemented a Python script to manage all service deployments. An example of running this script is shown in Figure 4 below:

```
$ sudo python3 ScaledDeployment.py 10
```

Figure 4 - Running the Deployment Script

The argument to this script ("10" in the example above) specified the number of track service instances that would be deployed and executed. The script first used this number to generate a K3s deployment configuration file for the track service. The track service's deployment configurations (as well as all others) were formatted in YAML: a complete example of such a configuration file can be found in Appendix B. Since other services' configurations did not change in response to adjusting the number of track service instances, no other configuration

files were generated at this point. Once an appropriate K3s deployment was generated, the script performed a number of other tasks.

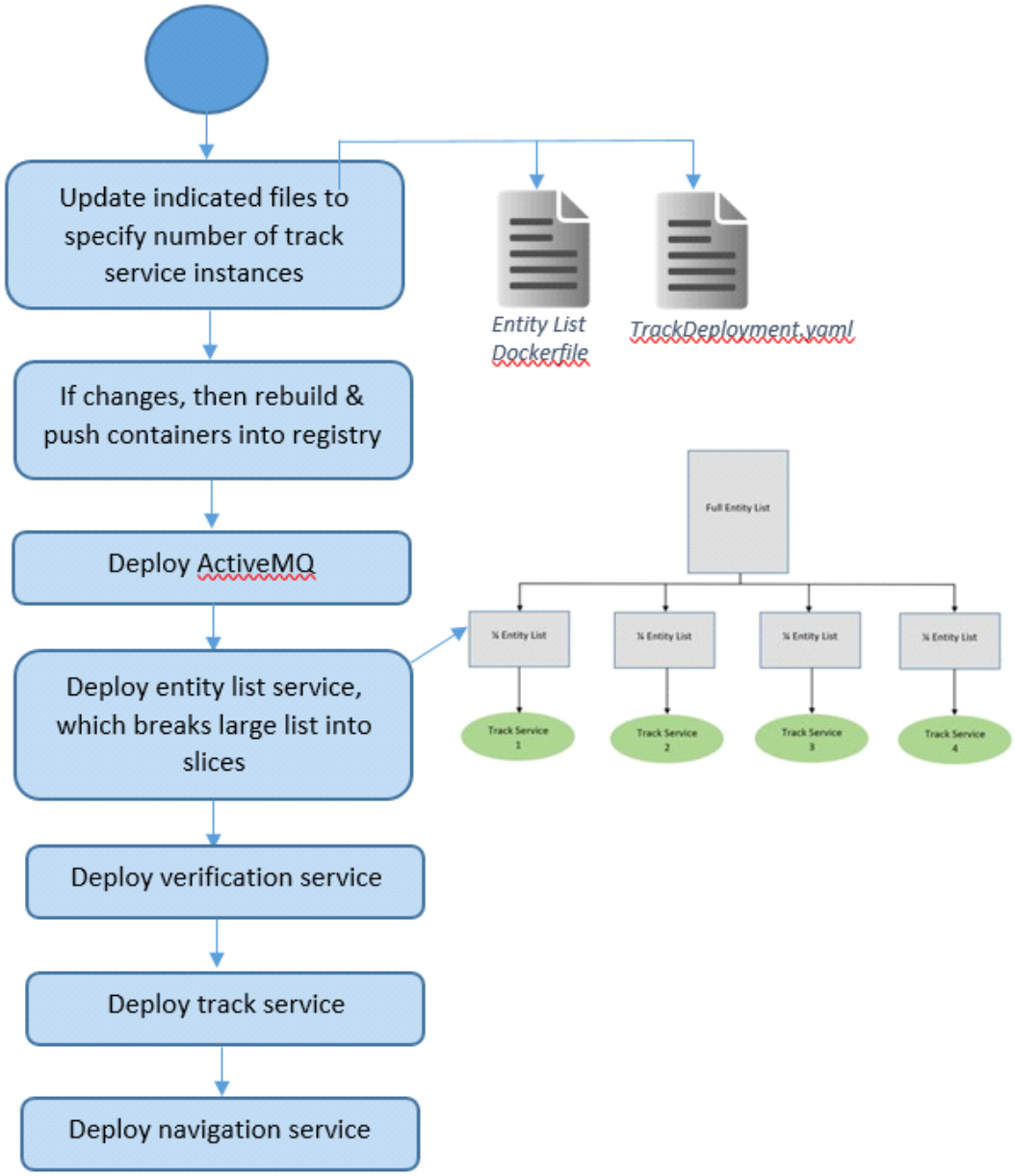

Figure 5 - Process Diagram

It configured the entity list service to support the specified number of track service instances. The script also rebuilt all microservice containers. Once the build process was complete, the script pushed them to a registry. Deploying the container images to the registry allowed all nodes of the K3s cluster to access them. The services themselves had internal dependencies on one another.

Since services began running as soon as they were scheduled, the services had to be deployed in a certain order in order to function properly:

- ActiveMQ
- The entity list service
- The verification service
- All instances of the track service, and finally
- The navigation service.

Upon completion of the startup process, the standard `kubectl` application could be used to monitor output and log messages generated by each instance. The verification service, specifically, was extremely useful during testing because it output statistics related to the total number of entities seen, as well as the total number of unique entities and the relative detection rate of an execution. This output allowed us to establish a relationship between the number of track services running and the number of entities that could be successfully detected.

TESTING AND RESULTS

In order to support reproducible tests, we generated a fixed simulation configuration and list of 1000 entities. In this case, the simulation specified a flight that began in Dayton, OH and ended in Columbus, OH. The aircraft moved at 1500 km / hr, and supported a radar range of 100 km.

From these parameters, we derived an expected running time for the application. First, we determined that the distance between the start location in Dayton and the end location in Columbus was approximately 104 km. Based on a speed of 1500 km / h:

$$Time = \frac{distance}{speed} \qquad (1)$$

We calculated that the approximate time our simulation would run to be around 249.6 seconds. Processing times for individual entities ranged between 250 milliseconds and 2.5 seconds. To arrive at a specific processing delay, each entity had a *priority* assigned to it within the entities.txt file. This priority was multiplied by a base value (called a *dwell time*) to arrive at total processing delay. Possible delays are enumerated within Figure 6 (below).

		Priority									
		1	2	3	4	5	6	7	8	9	10
Dwell Time	250	0.25	0.5	0.75	1	1.25	1.5	1.75	2	2.25	2.5

Figure 6 - Processing Times per Entity

Given that the distribution of priorities assigned to individual entities was uniform, we were able to compute a theoretical mean by dividing the sum of the delays by the number of delays – this worked out to be 1.375 seconds (represented as E[T]). T amount of possible entities seen can be calculated. With one track service, doing 249.6 seconds divided by 1.375 seconds on average to

scan an entity, results in an expected 181.5 entities seen. This calculation then scales as the amount of radars increases:

$$entities = \frac{time * count}{E[T]} \quad (2)$$

For example, with two track service instances, we would expect to see 181.5 * 2 = 363 entities. With five instances, we would expect to see 5 * 181.5 = 907.5 entities. This results in linear behavior for total entities seen in Figure 8.

With theoretical expectations prepared, the next step was to compare expected values to empirical data. Figure 7 describes the result of running tests and comparing the total number of entities processed to the theoretical values. This empirical data is addressed in Figure 8.

Given that there are only 1000 *unique* entities available to identify, once the track service instances are able to achieve that number, there is little point to scaling further. This target number of track service identifies something of a point of diminishing return – additional computational support provided by additional instances cannot be observed by evaluating the number of unique entities. This yields an asymptote at 1000 entities, which is highlighted by the orange line below.

Track Service Instances	Entities Expected	Unique Entities Seen	Total Entities Seen(Duplicates Included)	Success Percentage
1	181	189	189	18.9%
2	363	377	377	37.7%
3	545	542	542	54.2%
4	726	723	723	72.3%
5	908	884	902	88.4%
6	1089	951	1085	95.1%
7	1270	969	1279	96.9%
8	1452	979	1447	97.9%
9	1633	990	1638	99%
10	1815	994	1825	99.4%
11	1997	995	2014	99.5%
12	2178	999	2186	99.9%
13	2360	999	2352	99.9%
14	2541	1000	2547	100%
15	2722	1000	2733	100%
16	2904	1000	2893	100%
17	3085	1000	3082	100%
18	3267	1000	3287	100%
19	3449	1000	3471	100%
20	3631	1000	3644	100%

Figure 7 - Number of Track Service Instances vs. Total Entities Seen

To verify that processing capabilities scaled linearly beyond that point (14 services per Figure 7 above), we added a capability for the verification service to record cases where the same entity was processed more than once. This allowed the service to report the total number of entities

processed (the blue line in Figure 8), and demonstrates that the service continues to scale linearly up to 20 instances.

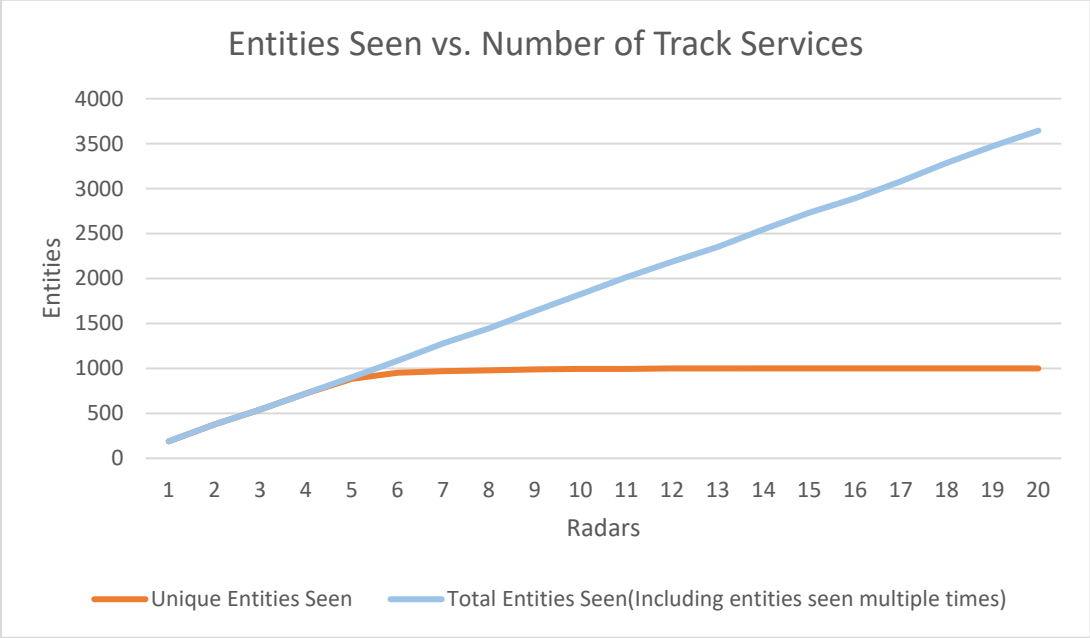

Figure 8 - Entities Seen vs. Track Services Instances

We did not record test results beyond 20 instances because we began to encounter limitations of the hardware – the NUCs would, for example, run out of memory during the simulation when executed with large numbers of services. Further, running these simulations for extended periods of time (between three and four hours) would lead the systems to begin to overheat. When this occurred, the CPUs throttled themselves (a reduction from 3 to 3.5 GHz down to 1 and 1.5 GHz) to reduce their heat output. This resulted in a large number of missed entities, and thus prevented us from running experiments when the CPUs were in a throttled state.

CONCLUSION

Impact

This project developed an infrastructure to support a microservice-based application. Working with the application developer, we were able to identify a task that would be a bottleneck, and identified changes that were necessary to alleviate this bottleneck. We successfully designed and implemented custom infrastructure (running on top of K3s). When run on our infrastructure, we demonstrated linear performance scaling up to the limits of the hardware. Further, we demonstrated a point of diminishing return, or a state in which adding additional services offered no real mission impact despite the additional compute capability they provided.

This point of diminishing returns represents a target for systems engineering as it relates to infrastructure design: once that point is identified, the system can be designed to achieve that

specific target. As such, these kinds of benchmarks and configurations can be useful during the early stages of a project.

Additionally, this project demonstrated how parts of a software development and deployment pipeline might be automated with the assistance of Kubernetes. Lessons learned throughout this project can, for example, be used when designing applications that can be deployed onto future infrastructure and systems.

Future Work

Given additional time, there are a few areas that we would hope to address. First, project did not address security considerations associated with deploying and executing applications in a Kubernetes environment. Future efforts would likely benefit from considering secure registries and certificate design / implementation. Secondly, we would hope to demonstrate how Kubernetes supported the addition and use of more than two physical nodes. Finally, we would hope to add support for a dynamic load balancing scheme. A dynamic load balancing approach would allow services to start and to stop based on actual demand, which would allow systems to more effectively adapt to unexpected workloads and / or circumstances.

REFERENCES

- Delnoce, Abigail. “Open API Mission Microservices,” (2021).
- Kubernetes, “Production-Grade Container Orchestration.” https://kubernetes.io/, (2021)
- Docker, “Docker overview,” https://docs.docker.com/get-started/overview/, (2021)
- Kublr, “Kubernetes Cluster Hardware Recommendations,” https://docs.kublr.com/installation/hardware-recommendation/, (2021)
- K3s, “Lightweight Kubernetes,” https://k3s.io, (2021)
- Kubernetes Scheduler, “Kubernetes Scheduler”, https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, (2021)
- Flask, “What Is Flask?” https://realpython.com/tutorials/flask/, (2021)
- Kubernetes API for Python, “kubernetes-client/python,” https://github.com/kubernetes-client/python/, (2021)

APPENDIX A – ENTITY LIST FORMAT EXAMPLE

```
39.76728119248441,-84.05126723245719,2,0
39.07459834938423,-83.01379626171507,2,1
39.31851801369255,-83.53994234442838,10,2
39.91432289053531,-83.33637472446185,8,3
40.11610957620486,-83.18192980842073,4,4
39.51863707018105,-83.08177414817324,1,5
39.33684953818491,-83.08900629995914,1,6
40.531736057147135,-83.53294452636305,10,7
39.61879868478499,-83.78445409464614,10,8
40.44163504413345,-84.10923657004922,2,9
39.70887986798583,-83.94799019436992,10,10
39.72634254107229,-83.41036983885178,10,11
40.15900182622297,-83.67622810835094,1,12
39.08966555430029,-83.88824262322449,1,13
39.93057304307898,-83.56920421415575,2,14
39.35425627188024,-83.58442139782126,3,15
40.23368530257015,-83.81916873526744,7,16
39.79544037627297,-83.57302730685302,8,17
40.55670887779244,-83.78754310777758,3,18
39.91801267235132,-84.15085979274654,9,19
39.752106403072624,-83.13925150398447,3,20
40.028482177518285,-84.1269894863844,3,21
40.64083450774442,-83.26355393150979,7,22
39.85556987192312,-83.11271760458511,5,23
39.91374972998053,-84.05674851652469,8,24
39.97406839481599,-83.32969828733819,2,25
39.959977811448525,-83.04741980825555,6,26
39.86459201220851,-83.59199517061705,10,27
39.12237519151439,-83.07816541581138,5,28
40.24707309977438,-83.08089274251512,2,29
40.66452606189176,-84.00699188343442,3,30
39.36886898054906,-83.75028584435147,8,31
39.727532349929945,-83.80264651879929,1,32
39.40586921780418,-83.7754770687091,6,33
39.02286306399716,-83.34260355320627,4,34
```

APPENDIX B – YAML CONFIGURATION FILE EXAMPLE (TRACK SERVICE)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: track-deployment
  labels:
    app: track
spec:
  replicas: 20
  selector:
    matchLabels:
      app: track
  template:
    metadata:
      labels:
        app: track
    spec:
      containers:
      - name: track
        image: 10.43.121.51:5000/track-service:latest
        ports:
        - containerPort: 80
```

APPENDIX C – EXAMPLE VERIFICATION OUTPUT

```
{2021-07-23 09:44:03,619 [Thread-1] DEBUG}: Success Percentage: 17.599999999999998%
{2021-07-23 09:44:05,884 [Thread-1] DEBUG}: Entities Seen: 178 Total Entites: 1000
{2021-07-23 09:44:05,884 [Thread-1] DEBUG}: Entities Seen with Duplicates: 178
{2021-07-23 09:44:05,884 [Thread-1] DEBUG}: Success Percentage: 17.8%
{2021-07-23 09:44:08,139 [Thread-1] DEBUG}: Entities Seen: 180 Total Entites: 1000
{2021-07-23 09:44:08,139 [Thread-1] DEBUG}: Entities Seen with Duplicates: 180
{2021-07-23 09:44:08,139 [Thread-1] DEBUG}: Success Percentage: 18.0%
{2021-07-23 09:44:11,171 [Thread-1] DEBUG}: Entities Seen: 183 Total Entites: 1000
{2021-07-23 09:44:11,171 [Thread-1] DEBUG}: Entities Seen with Duplicates: 183
{2021-07-23 09:44:11,171 [Thread-1] DEBUG}: Success Percentage: 18.3%
{2021-07-23 09:44:13,676 [Thread-1] DEBUG}: Entities Seen: 185 Total Entites: 1000
{2021-07-23 09:44:13,676 [Thread-1] DEBUG}: Entities Seen with Duplicates: 185
{2021-07-23 09:44:13,676 [Thread-1] DEBUG}: Success Percentage: 18.5%
{2021-07-23 09:44:16,218 [Thread-1] DEBUG}: Entities Seen: 188 Total Entites: 1000
{2021-07-23 09:44:16,218 [Thread-1] DEBUG}: Entities Seen with Duplicates: 188
{2021-07-23 09:44:16,218 [Thread-1] DEBUG}: Success Percentage: 18.8%
{2021-07-23 09:44:18,478 [Thread-1] DEBUG}: Entities Seen: 189 Total Entites: 1000
{2021-07-23 09:44:18,478 [Thread-1] DEBUG}: Entities Seen with Duplicates: 189
{2021-07-23 09:44:18,478 [Thread-1] DEBUG}: Success Percentage: 18.9%
{2021-07-23 09:44:18,494 [Thread-1] DEBUG}: They've arrived!  Go them!
{2021-07-23 09:44:18,494 [Thread-1] DEBUG}: Shutting down session and connection.
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Results (Entity Number:  Number of times seen)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 1 with coordinates (39.07459834938423,-83.01379626171507)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 57 with coordinates (40.811601610781096,-83.27413886135726)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 61 with coordinates (38.97277880485434,-83.96605823111216)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 67 with coordinates (40.78590133808858,-83.28266828255839)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 87 with coordinates (38.95570345455213,-83.82826298664305)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 96 with coordinates (38.98210110482364,-83.89644539116985)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 122 with coordinates (40.566908950436385,-84.07597099271572)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 140 with coordinates (39.00544033414125,-83.81296008705469)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 142 with coordinates (39.35603557131686,-83.93062088330504)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 149 with coordinates (38.99049028037316,-83.73740781504134)
{2021-07-23 09:44:18,509 [Thread-1] DEBUG}: Missed entity 154 with coordinates (39.26085891588616,-83.21773977209847)
```