DESIGN REPORT

A comprehensive review of the algorithmic and structural design of an automated dispatch system.

Ethan McNamara
ethan.s.mcnamara@gmail.com
GitHub: ethan-mcnamara

Table of Contents

1 Introduction

Regularly taken for granted, emergency dispatchers play a vital role in modern society and the technology with which they work provides a crucial bedrock upon which the whole emergency responders system is built. One such technology is the dispatch system used by a regional fire, police, or EMS district. These systems allow operators to allocate crews and vehicle to events and track the status of the crew throughout the response.

The below document describes in detail the design and architecture of an automated dispatch system. This implementation is designed for a fire district, although that is only true due to the naming of the classes as the foundation of the system could be readily adapted to other varieties of emergency dispatching. To improve performance and fault tolerance of the system, a multithreaded programming model is used, with lock-free techniques used to perform the synchronization. Later sections will explain this synchronization in further detail, in addition to providing a compelling rationale for such a design decision.

2 Problem Definition

Most modern regional fire, police, and EMS districts accept emergency situation details from a 911 call centre and use a desktop application to pick vehicles to respond to the situation and communicate to the crew the situation details. While this system can work very effectively with smaller districts where the volume of events and vehicles is minimal, with larger districts such as major urban centres, this manual process can be slower than ideal and may require constant communication of some sort between large numbers of people. Unfortunately, while the systems used are very sophisticated, there exists no better system as the life and death nature of the job requires an accountable operator.

2.1 System Context

While most dispatch systems provide a graphical user interface to aid operators in assigning vehicles and crew to events, this system does not as while its purpose is consistent with traditional dispatch systems, the method by which it does so bears stark differences from the traditional systems. Primarily, the use of automated dispatching removes the requirement for a pleasant user interface for the operator and as such the tool is run through the command line with basic status updates printed to the console while the dispatching is occurring.

Additionally, the system assumes a static supply of vehicles and fire stations as they must be provided to the system as input in a comma-separated value file at initialization. Most commercial dispatch systems allow for the addition or removal of resources without a necessary system restart but this system operates under the inherent assumption that this process will never be required.

2.2 System Constraints

The primary constraint of a dispatch system is the unpredictable occurrence of events requiring dispatching. These events arrive in spontaneously and have vehicle and crew requirements that are unknown until the moment of dispatching. Thus, it is difficult to allocate resources in an optimized manner as preceding events may unintentionally consume resources better suited for a later event. A fully optimized solution is often unattainable in a real time scheduling system and therefore the decision

parameters must be set such that the decision made meets a certain standard, without the requirement of guaranteeing it is the ideal decision.

2.3 Value of Multithreading

Due to a dispatch system's requirement to perform real-time scheduling for critical situations, performance and fault tolerance are the two quality attributes into which much of the design effort was placed. Multithreading provides a significant benefit to both these attributes as the ability to dispatch crews concurrently and be able to continue execution in the event of a thread failure are significant benefits.

More specifically, since many events require multiple responding vehicles, multiple threads can be valuable as each vehicle could be dispatched by its own corresponding thread, thereby converting a linear-time problem into a constant-time problem. Of course, this assumes the system is operating under very specific circumstances which may not always be true. Additionally, the threat of thread failure is very significant in a single threaded system as such a failure may cause the system to fail which may cause acute real-world problems in a true emergency dispatch system.

3 System Requirements

Like all software projects, there are several requirements in different categories the dispatch system must meet for it to be considered properly implemented.

3.1 Reliability & Fault Tolerance Requirements

Firstly, due to the serious impacts of failing to dispatch first responders to a crisis, the system must have very high reliability and fault tolerance. An exception incurred by one thread, or the complete failure of an individual worker thread, must not have an impact on the correctness of the program. Reductions in performance are to be expected in this scenario, but the priority is ensuring the program logic is intact.

3.2 Performance Requirements

Again, due to the repercussions of failing to expediently dispatch crews to a situation, high performance is a requirement which must be met. The metric upon which the performance of the system will be measured is the average time from the initial notification of an event to the point at which all required resources have been dispatched to this event. This metric will be calculated for a single-threaded implementation and each subsequent multithreaded implementation must exceed this threshold to the degree required. There will be a maximum number of threads that can provide a performance increase, at which point adding additional threads renders to performance gain. For each thread added below this threshold, the metric should decrease as the performance increases.

3.3 Testing Hardware

To implement a lock-free algorithm, specific atomic instructions are required to guarantee correctness. Due to differences in CPU instruction sets, these atomic instructions may differ from one machine to the next.

3.3.1 Intel® Core™ i7-8650U CPU

The Intel® Core™ i7-8650U CPU is the CPU for which the first version of the program will be built. Due to it being an Intel® product, the atomic instruction available for lock-free synchronization is compare-and-swap (CAS).

This CPU is the primary hardware on which the program will be developed and tested.

3.3.2 Arm® Cortex®-M4 32-bit CPU

The Arm® Cortex®-M4 32-bit CPU is an alternate testing hardware. Arm CPUs offer the load-linked (LL) and store-conditional (SC) atomic instructions for lock-free programming. These different instructions require modifications to the code and thus a version of the dispatch system will also be developed for this CPU, should time permit. These different instructions will provide an interesting opportunity to contrast different lock-free programming techniques.

4 Conceptual Design

This project will be developed in C++ and its design is built according to the features and limitations of the language. Leveraging the C++ language's object-oriented feature set, the program will be built using classes and objects. The procedural (C-style) programming features available in C++ will also be utilized. Specifically, pointers will be used to reduce the cache size of lists of objects when the contents of the objects are not consistently accessed.

4.1 Class Architecture

The class abstractions in this program are consistent with their real-world versions. Classes representing fire stations, vehicles, and events are the primary stores of data and functionality. The characteristics of each fixed fire station, emergency response vehicle, and events, whether crisis events or scheduled maintenance, are stored in class attributes with the methods used by each class also stored within that class.

The vehicle and event classes are implemented as abstract classes. They are extended by specific implementations representative of their real-world counterparts. The abstract vehicle class has two implementations: the fire engine and the fire ladder class. These classes represent the two most used varieties of fire trucks. This program operates under the assumption that these are the only trucks used in the simulated districts, although additional class implementations could be readily added. The abstract event class is implemented by the critical situation and maintenance classes. The critical situation class represents an event requiring an immediate dispatch of first responders. The maintenance event class represents a regularly scheduled maintenance event requiring the vehicle to stay at a fire station for a pre-determined duration of time.

In addition to the three primary classes, there are other classes used to store accompanying program information. The location class stores a two-dimensional location on a Cartesian grid used to approximate GPS location data. The vehicle status class is an enum class representing the possible current statuses of a vehicle. The time class represents an instance in time with a millisecond granularity, used to denote when an event occurs. The district resource class acts as a repository of the vehicles and fire stations available in a district. The final class is the event factory class which holds no data but acts as the mechanism for creating and preparing events for dispatching.

4.1.1 UML Diagram

To demonstrate the available classes and the relations between them, Figure 1 features a UML class diagram.

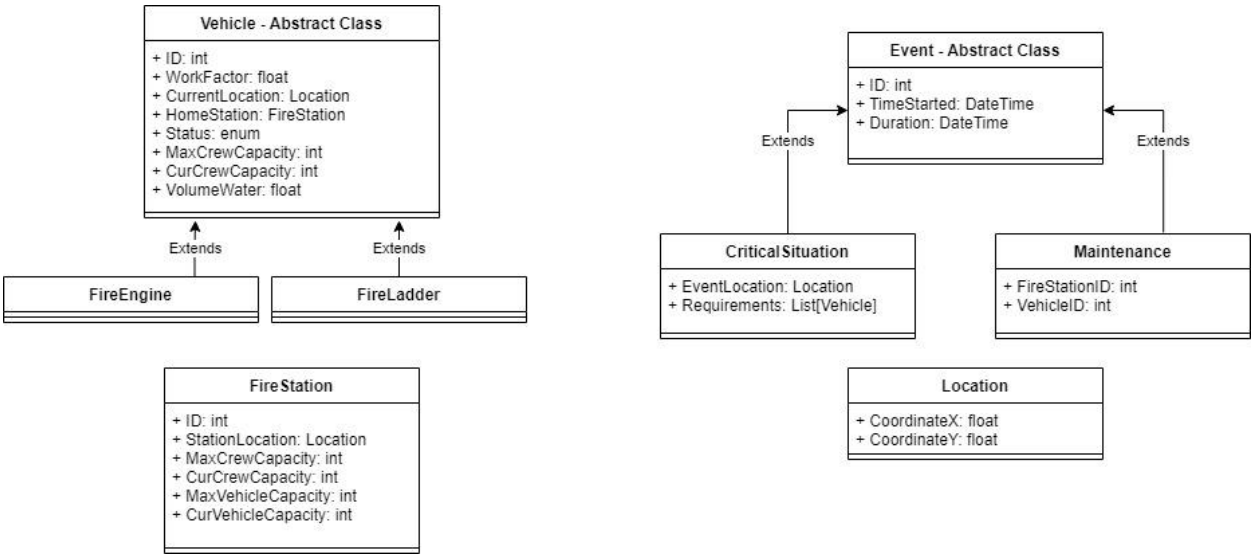

Figure 1: *A UML diagram of the automated dispatch system*

5 Data Management

In developing multithreaded programs, the management of data is vitally important to avoid the development of race conditions. To do so properly, an explicit definition of the shared mutable must be made to show the data which may be edited concurrently by separate threads so the proper safeguards may be applied.

5.1 Shared Mutable Data

The data element regularly modified by separate worker threads is the vehicle status attribute of the vehicle objects. This status attribute defines whether a vehicle can be dispatched to an event or whether it is unavailable at the current time to be dispatched. In dispatching vehicles to an event, this status is read and the determination of whether this vehicle will be dispatched will depend on the value read. Should the vehicle be available, the worker thread will write to the status attribute with a value indicating the change in availability. It is this writing which is the primary point of contention as it may be possible for two threads to read a vehicle's status as available and both attempt to write the value simultaneously, which would have negative program logic outcomes unless properly mitigated as an individual vehicle can not be assigned to two events at the same time. While this data element is not the sole mutable data element potentially accessed simultaneously by multiple threads, it is the only data element whose value written is dependent on the existing value. All other instances of mutable data are modified based on external factors and should not experience race conditions.

5.2 Data Flow Design

Recognizing the importance of data management in a multithreaded program, the design of the flow of data between classes is vital and as such models have been created to show the processes involved in moving data at two different granularities.

5.2.1 High-Level Data Flow Diagram (DFD)

The higher-level DFD models the primary interactions between classes, the processes involved, and the data elements transmitted. This diagram can be seen in Figure 2.

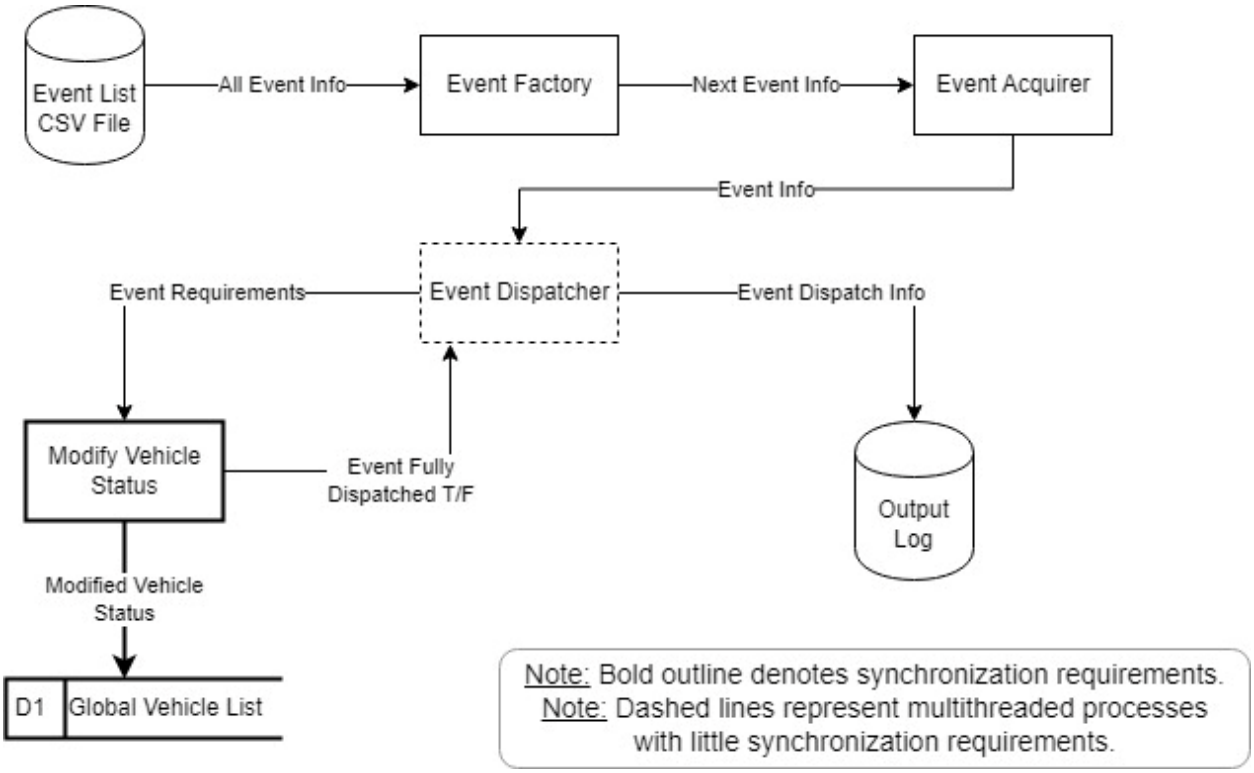

Figure 2: *A high-level data flow diagram of the automated dispatch system*

5.2.2 Low-Level Data Flow Diagram (DFD)

To further demonstrate the specific processes and interactions occurring at a finer granularity, Figure 3 shows a low-level DFD.

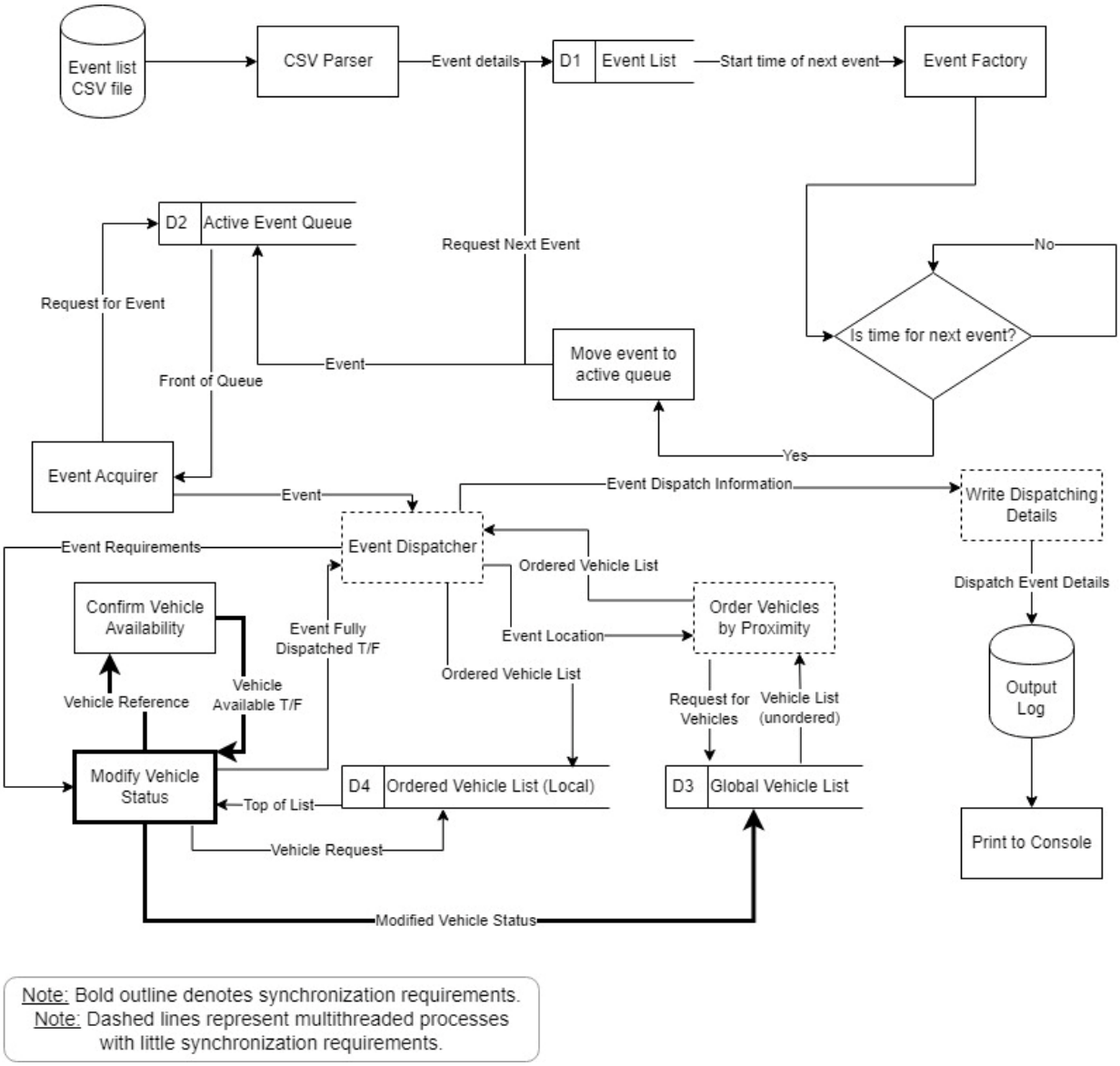

Figure 3: *A low-level data flow diagram of the automated dispatch system*

6 Lock-Free Concurrency

While lock-free programming techniques may be used in several interactions in the system, the primary point at which the lock-free techniques will be the reading and potential writing of the vehicle status attribute. This process will be described in further detail in this section to demonstrate the capability of the lock-free programming techniques.

6.1 Shared Data Structure

The vehicle status data member is a C++ enum class with the following statuses: (a) Available, (b) Responding, (c) Returning, (d) Maintenance, (e) Unavailable. The status property is a private variable of the vehicle class accessible using public getter and setter functions.

6.2 Lock-Free Techniques

The primary lock-free technique used to guarantee synchronization is the compare-and-swap (CAS) loop. The CAS loop relies on the CAS instruction which atomically compares a value parameter with the value stored in a memory address parameter and if they are identical the second value parameter is written to the memory address. This instruction is used in a loop by storing the variable whose value is intended to be modified in a temporary variable and storing the modified value in a second temporary variable. The unmodified temporary variable is then passed to the CAS function and its value is compared to the value in the memory address. If they are equal, no other threads have modified this variable in the interim since the loop began and the modification logic can be ensured. Should they not be equal, concurrent access has occurred and the value in memory must be read again. The CAS function is executed in the conditional section of the loop and its Boolean return value determines whether the loop iterates.

Alternatively, on different testing hardware on whom the CAS function is not available, the store-conditional (SC) and load-linked (LL) atomic instructions are used to ensure lock-free synchronization. These operations work in tandem by relying on a tag placed on a cache line that remains raised until the data stored in memory represented by the cache line is modified. For example, if two cores have the same cache line in their level one cache and this cache line is modified by one core, the flag would be automatically lowered on the other core due to the CPU's cache coherency protocol. The LL operation reads a variable from memory and adds the flag to that variable's cache line. The SC operation stores a modified value to memory if the flag is still raised and returns a Boolean value stating the success of the write.

6.3 Guarantee of Non-Blocking Execution

Lock-free algorithms differ from blocking multithreaded algorithms in that there does not exist the possibility of no threads making progress. For an algorithm to be considered lock-free, at least one thread must always make progress. With both the CAS approach and the LL & SC approach, this is true as the loops will only iterate when a concurrent modification occurs. When multiple threads simultaneously attempt to modify a shared data element, one thread will always succeed, and the others will perform their modifications in a sequential manner after iterating their respective loops.

6.4 Requirements for Correctness

In scientific literature regarding lock-free algorithms, two aspects of an algorithm are proven to show the correctness of an algorithm: safety and liveness. In layman's terms, safety is the guarantee "bad things" will not occur while liveness is the guarantee "good things" will occur. This is proven by explaining how an algorithm will eventually execute the correct instructions without executing instructions that will result in negative program outcomes.

6.4.1 Safety & Liveness

The safety aspect of this algorithm speaks to the correct dispatching of resources to an event without a deadlock situation occurring. The later portion of that statement is proven by the lock-free mechanisms

used in the algorithm's implementation, as discussed in Section 6.3. The goal of this algorithm is to accurately and expediently dispatch vehicles and crews to an event, with the specific vehicles dispatched based on the location and requirements of the event. Each emergency event will have a list of vehicular, personal, and water volume requirements that must be met for the event to be considered correctly dispatched. Additionally, assuming there exists a surplus of vehicles meeting the situation's requirements, vehicles should be considered for dispatch based on proximity, with the nearest vehicles considered for dispatching before those further away.

Safety is proven in this algorithm by the fact that there does not exist the possibility for race conditions, due to the lock-free mechanisms used. With no potential for race conditions, and with proper implementation of the algorithm, actions that negatively impact the program logic should not occur.

6.4.2 Linearization Point

In scholarly discussions of lock-free programming, the linearization point is the aspect of an algorithm most important for proving the correctness of a lock-free algorithm. The linearization point represents the single discrete point at which a function of a multithreaded algorithm is said to execute and can be compared to every other function in the algorithm as either occurring before or afterwards. This is vital to proving the correctness of a multithreaded lock-free algorithm as to do so one must prove that the execution history of an algorithm, that is the history of all functions executed by the algorithm, can be ordered sequentially, with no two functions occurring exactly simultaneously.

While explained in further detail in Section 6.6, the linearization point of this algorithm is the modification of the vehicle status attribute of a Vehicle object in the global shared vehicle list.

6.5 Deterministic Algorithm Execution

To measure the performance of lock-free concurrency, the algorithm requires deterministic execution such that it may be effectively compared to a single-threaded or blocking implementation. This is important as to compare the running of an algorithm with one technique implemented with another running of the same algorithm with a different implemented technique it must be shown that these individual runs are consistent with the standard execution of the algorithm. An algorithm whose execution can vary widely from one run to the next cannot have its performance effectively compared.

Within the context of this algorithm, this property is implemented by guaranteeing that the vehicles dispatched to each event remain consistent when the synchronization technique remains constant.

6.6 Vehicle Status Finite State Diagram

To aid the proof of correctness in the next section, and considering the vehicle status attribute is the modified data at the linearization point, a finite state diagram representing the possible state transitions is shown in Figure X. Each state represents a vehicle status possibility and the arrows between the states represent the allowable state transitions in the system. Note the Unavailable state does not support any transitions in our out of this state as this is an exceptional state whose behaviour is only required to handle unforeseen situations or errors.

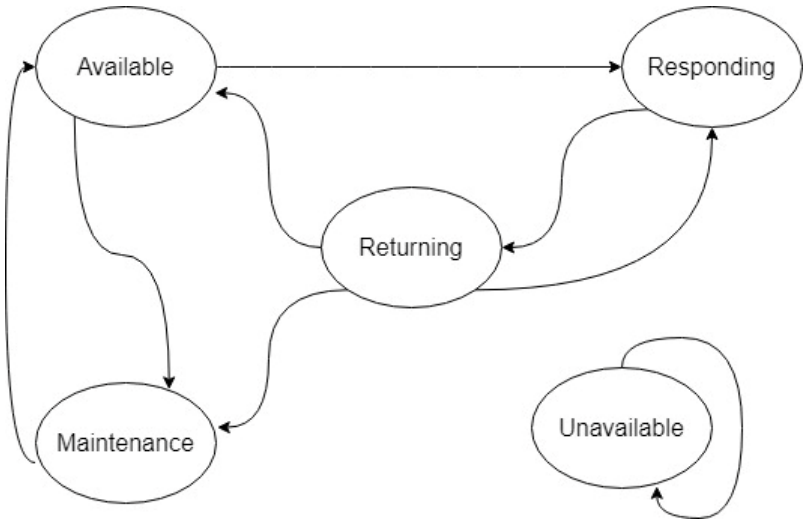

Figure 4: *A finite state diagram of the possible vehicle statuses*

6.7 Proof of Correctness

The below proof demonstrates the correctness of the algorithm used by the dispatching system.

6.7.1 Algorithm Pseudocode

Leveraging the classes and data flows demonstrated in above sections, the dispatch system's algorithm will resemble the below pseudocode.

```
A1:    bool modifyVehicleStatus(Vehicle* curVehicle)
A2:    {
A3:        if (curVehicle.Status != 'Available' || curVehicle.Status != 'Returning'):
A4:            return false
A5:
A6:        curVehicleStatus = curVehicle.Status
A7:        newVehicleStatus = 'Responding'
A8:        return CAS(curVehicle.Status, curVehicleStatus, newVehicleStatus)
A9:    }


B1:    void selectVehicles(List<Vehicle> vehicleReqts, List<Vehicle> orderedVehicles)
B2:    {
B3:        numVehiclesNeeded = vehicleReqts.size
B4:        for (vehicle1 in vehicleReqts):
B5:            listSubset =
orderedVehicles.thatMeet(vehicleReqts)[0:numVehiclesNeeded*2]
B6:            avgWorkFactor = AVG(listSubset.elements.WorkFactor)
B7:            for (vehicle2 in listSubset):
B8:                if (vehicle2.WorkFactor > avgWorkFactor):
B9:                    if ( modifyVehicleStatus(&vehicle) ):
B10:                       orderedVehicles.remove(vehicle2)
B11:                       vehicleReqts.remove(vehicle1)
B11:                       --numVehiclesNeeded
B12:                       break inner loop && continue outer loop
B13:           for (int i = 0 -> orderedVehicles.size):
B14:               if ( modifyVehicleStatus(&orderedVehicles[i]) ):
B15:                   break
```

6.7.1.1 *modifyVehicleStatus*

The *modifyVehicleStatus* function is the function in which the status of the vehicle attempting to be dispatched is modified, with the lock-free synchronization incorporated. As seen on line A3, the vehicle's status is first checked to confirm whether it can be dispatched. To be assigned to an event, it must be either available or returning from another event. Thus, if it's status does not meet these conditions, it can be quickly discarded from consideration. A copy of the status is taken on line A6 and the desired new value (Responding) is written to a temporary variable on line A7. On line A8, the CAS function is used to attempt to modify the vehicle's status attribute. Should the CAS function succeed, true will be returned indicating the status has been successfully modified. Should false be returned, a different thread has already dispatched this vehicle, therefore the attempt to modify the status by the current thread has failed and a negative Boolean value is returned from the *modifyVehicleStatus* function.

6.7.1.2 *selectVehicles*

The *selectVehicles* function includes the logic for selecting the vehicles which will be dispatched to the provided event. The two factors affecting whether a vehicle will be dispatched to an event are its proximity to the event and its work factor, a floating-point value ranging from zero to one with a higher value representing a more efficient response. The function will also consider the requirements of the event and will only consider those vehicles who meet the requirements.

In considering the work factor, the *selectVehicles* function will consider twice as many vehicles as it currently needs and select those with a work factor rating above an average value of all vehicles considered. As vehicles are dispatched to an event, the number of outstanding vehicles on the requirement list will decrease, with the number of vehicles under consideration decreasing proportionally.

Should the list of considered vehicles be exhausted without effectively dispatching a vehicle with an above-average work factor rating, the function will revert to a purely proximity-ordered list and attempt to assign the nearest vehicle, as seen on lines B13-B15.

The outermost loop (line B4) will continue to iterate until all requirements for the event have been met.

6.7.2 Formal Proof

This proof will be organized as follows: Section 6.7.2.1 will provide a set of cases in which the shared data structure element can be modified and the affect this change will have on the other threads; Section 6.7.2.2 will prove the linearizability of the algorithm; while Section 6.7.2.3 will prove the algorithm's lock-freedom.

6.7.2.1 *Shared Data Modification Cases*

Case 1: Vehicle Status: Available, Two Events Require this Vehicle

In this case, a single vehicle, denoted Vehicle A, is under consideration. In the *selectVehicles* function of the threads working to dispatch each event, Vehicle A is the nearest vehicle with the highest work factor rating, and is thus the vehicle both events wish to acquire first. Assuming these events are occurring exactly or nearly simultaneously, and that only a single worker thread is working to dispatch each event, two calls to the *modifyVehicleStatus* function will be made in short succession. Due to the use of the atomic CAS function on line A8 of the *modifyVehicleStatus* function, only the thread who calls this function first will succeed and therefore only one call to *modifyVehicleStatus* will return true while the

other will return false as the status of Vehicle A will have been modified before the call to CAS. Thus, the program logic is maintained and the thread with the failing *modifyVehicleStatus* call will continue its execution and find another vehicle to meet the event requirements.

Case 2: Vehicle Status: Responding, Event Completes, Different Event Requires this Vehicle

While responding to an event, a vehicle's status will be 'Responding'. After the pre-determined duration of time required to respond to the event, the vehicle's status will be changed from 'Responding' to 'Returning'. Assume Vehicle A is currently responding to Event 1 and is nearly finished responding, and Event 2 is near Event 1 but with a start time shortly after the completion time of Event 1. If the thread handling Event 2 attempts to dispatch Vehicle A before it has completed Event 1, the *modifyVehicleStatus* will return false on line A4 as the vehicle's status is not 'Available' or 'Responding'. As soon as Event 1 ends, the worker thread handling that portion of the work related to Event 1 will write 'Responding' to the status of Vehicle A. This write will not be done using a CAS function as this write does not require synchronization as only one thread is attempting to do this work with no other competition for the shared data. If the thread responsible for Event 2 then attempts to call *modifyVehicleStatus* on Vehicle A, it will be able to do so successfully as the conditional statement into which the earlier thread fell into will no longer be true.

6.7.2.2 Linearizability

The linearization point of the vehicle dispatch process is the modification of the vehicle status attribute, done with the CAS function on line A8. Both the *selectVehicles* function and the *modifyVehicleStatus* have the same linearization point as they both contribute to the same process and the *selectVehicles* function makes no modification to the shared data structure until its call to the *modifyVehicleStatus* function. Line A8 is the point at which the shared data structure, the vehicle objects' status attribute, is modified and it is this point at which the vehicle is considered dispatched to an event.

6.7.2.3 Lock-Freedom

It is simple to show the lock-freedom of the algorithm as the lack of locking structures can be seen from the pseudocode, but it is important to examine the algorithm flow carefully to ensure there does not exist the possibility of a locking scenario or a deadlock. To show the lack of a lacking scenario, consider the situation in which two threads are attempting to dispatch the same vehicle to different events. Should that occur, only one thread will succeed while the other will continue to iterate the loop (line B7 or B13) within *selectVehicles* and will not block. Additionally, due this quick failure and iteration of an unsuccessful vehicle status modification, there exists no possibility for two threads to force each other into a deadlock.

7 Conclusion

With the proper synchronization mechanisms in place, a multithreaded lock-free emergency dispatch system can provide an effective and interesting simulation of its real-world counterpart. As seen in this document, the flow of data between members and the respective interactions must be appropriately considered to guarantee proper algorithm logic but if done correctly should result in a system able to outperform a manual, single-threaded, or blocking implementation.

8 References

[1] D. Hendler, N. Shavit, and L. Yerushalmi, “A scalable lock-free stack algorithm,” *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 1–12, Oct. 2009.