# 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.

# 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.

However, if the consequences are stripped from the problem, an automated system can be developed to replicate the tasks of the operator. With details of simulated emergency events delivered to the system seemingly as they occur and simulated first responders, vehicles, and fire stations an accurate representation of a fire district can be constructed.

## 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.

## 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.

## 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.

# 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.

## 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.

## 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.

## 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.

### 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.

### 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.

# 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.

## 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.

### UML Diagram

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

<INSERT UML DIAGRAM>

## Class Interactions

The classes interact with each other in various means, as is demonstrated below in Figure 2.

<CREATE CLASS INTERACTION DIAGRAM>

# Data Management

In developing multithreaded programs, the management of data is vitally important to avoid the development of race conditions. In order 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.

## 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.

## 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.

### 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 4.

<INSERT HIGH-LEVEL DFD>

### Low-Level Data Flow Diagram (DFD)

To further demonstrate the specific processes and interactions occurring at a finer granularity, Figure 5 shows a low-level DFD.

<INSERT LOW-LEVEL DFD>

# 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.

## 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.

## 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.

## 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.

## 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.

### 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](#_Guarantee_of_Non-Blocking). 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.

### Linearization Point

## Proof of Correctness

(Mathematical proof)

# Conclusion

# References