# CSC 464 Assignment 2

Sterling Laird - V00834995

November 22th 2018

## 1 Introduction

This report covers the basic implementation of vector clocks and the byzantine generals problem, as well as outlines the steps taken to test my implementations for verification of the algorithms.

All code in this assignment was written by me, Sterling Laird with all my understanding of the problems coming from *The Byzantine Generals Problem* (Lamport et al.,1982) and from various online sources outlining the algorithms used to solve these problems. All implementations can be found on my Github page (https://github.com/sterlinglaird/CSC-464).

## 2 Vector Clocks

For this assignment I implemented a framework for using vector clocks across any number of threads to enable synchronization/orderer arbitrary events among one another. I choose to create my implementation in Go as Go makes it easy to communicate across concurrent threads so I would not have to worry about creating my own message passing system for this problem.

My implementation of vector clocks allows the user of the framework to create new instances of vector clocks for each thread, provided they supply the identifiers of the threads and a list of the communication channels. The framework provides the functions Inc(), Send(), Recieve() which can be used to synchronize between the threads and to increment the local clock of a thread. All synchronization is expected to be done by the user of the API as the framework only provides the *means* of synchronization, not the act.

To test my implementation I created the program test_vector_clock.go to run a selection of examples to see if the implementation produces the correct final clock vectors for each thread. Each test uses a fixed number of threads and each thread is given a list of the events that will be used to order the threads. In a real system, each thread could be doing any arbitrary amount of work in between these events and the system would still be ordered correctly, but this in unneeded for these tests. Once all the threads have completed running their events we compare the final vector clocks with the expected results that are already know, if they all match then the test was a success. Of the two tests in my implementation, one is an example from the wikipedia page for vector clocks, and the others is an example developed by me. The correct values for the vector clocks were calculated by hand.

## 3 Byzantine Generals

In my implementation of the byzantine generals problem using the oral messages algorithm, I created a framework that can be used to simulate a collection of generals with arbitrary numbers of traitorous/loyal generals, and generates the order that each general will take using the oral messages algorithm.

My implementation of byzantine generals creates a message passing tree where each node is a commander for OM(d) where d is the depth at that point in the tree and the children of the node are the lieutenants for that commander. This process continues until the height of the tree equals the number of traitors, because as outlined by Lamport, this is all that is needed. Once this tree has been constructed, any node's final result can be queried to calculate the result. This process happens recursively by calculating the majority of the final results of the nodes that send messages to our queried node one level down the tree.

To test my implementation of the Byzantine Generals problem I created two programs which are outlined in the following:

1. **test_byzantine_generals.go** tests the framework by simulating the problem with different numbers of generals, and with different ratios of traitors to loyal generals, as well as with an initial order of both ATTACK and RETREAT. Generals are randomly assigned to be traitorous and the commander can also be traitorous. We only test ratio of traitors to loyal generals that we expect to work (number of generals > 3 * number of traitors) because otherwise we cannot tell if the algorithm works correctly since we cannot expect the correct algorithm to achieve or fail to achieve consensus. The numbers of generals that are currently tested are 4, 7, 10, 13, 16 which results in a total of 40 tests.

2. **run_byzantine_generals.go** runs the simulation on an input set of generals. This program is used to satisfy the assignment requirement to have command line input to run the algorithm. Parameter information can be found by the -h flag and an example simulation is "test_byzantine_generals.go -g C:L,L1:L,L2:L,L3:T -o ATTACK".