CSC 464 – Assignment Two

Parts One and Two

Michael Windels
November 22, 2018
V00854091

For Dr. Yvonne Coady's CSC 464 (Fall 2018) class.
Repository: https://github.com/MWindels/uvic-csc464-a2

0 A Foreword on Running the Tests

To run the tests for the vector clock and Byzantine generals implementations, you must add the path to the directory containing the repository (i.e. the directory containing the `.git` file and the `src` directory) to your GOPATH environment variable. Once this is done, you can then use the `go build` command to compile the tests.

1 Vector Clock

This section describes my implementation of a vector clock. It also describes the tests I wrote to verify the correctness of the implementation, and the results obtained from running those tests.

1.1 Description of the Implementation

To implement a vector clock, I have used a very simple data structure composed of a map and an integer identification field. The map serves as the vector, mapping process identification integers to clock value integers. I have opted to use a map rather than a slice or array, because the map allows elements to be accessed by discontinuous indices. This means that new entries can be added to the clock without needing to re-arrange the whole vector. That task is delegated to the underlying implementation of the map data structure. The vector clock changes state with the `Increment` and `Merge` operations. The `Increment` operation adds one to the element in the vector indexed by the identification field. The `Merge` operation merges the vector clock with another vector clock. Merging a clock with another fills the clock's vector with the maximum values of every entry listed in both clocks' vectors. The implementation is written in Go and can be found at the following link.

https://github.com/MWindels/uvic-csc464-a2/blob/master/src/lib/p1/vector_clock/vector_clock.go

1.2 Description of the Tests

In order to test my vector clock implementation, I have put together a little testing suite which takes an input file and produces logical timestamps. The input file is a JSON file that contains a list of processes. Each process is a list of events. Each event is composed of three fields. The first field is the `wait` field, which denotes how much time an event should take while running. The second field is the `prev` field, which is a list of integers. The third field is the `next` field, which is also a list of integers. To impose a happens before-relationship between two events in different processes, the first event should have some integer *x* in its `next` field, and the second event should have the same integer *x* in its `prev` field. Every unique happens-before relationship should use its own unique integer. A sample input file is provided with the testing file, but you can also write your own. The testing code is written in Go (like the implementation), and can be found at the following link.

https://github.com/MWindels/uvic-csc464-a2/blob/master/src/tst/p1/processes.go

To run the tests yourself, use `go build processes.go` in the `src/tst/p1` directory to compile the tests. Then run the tests using `processes input_file`, where `input_file` is the path to the JSON file you want to use as input.

1.3 Testing Results and Correctness

The `sample.json` file provided in the repository gives us the processes, events, and happens-before relationships shown in Figure 1.1. The processes are denoted by lines. They start on the left, and end on the right. Each node on the process lines represent an event. Arrows between events denote happens-before relationships. After running the tests on `sample.json`, we get the logical timestamps presented in Figure 1.2.

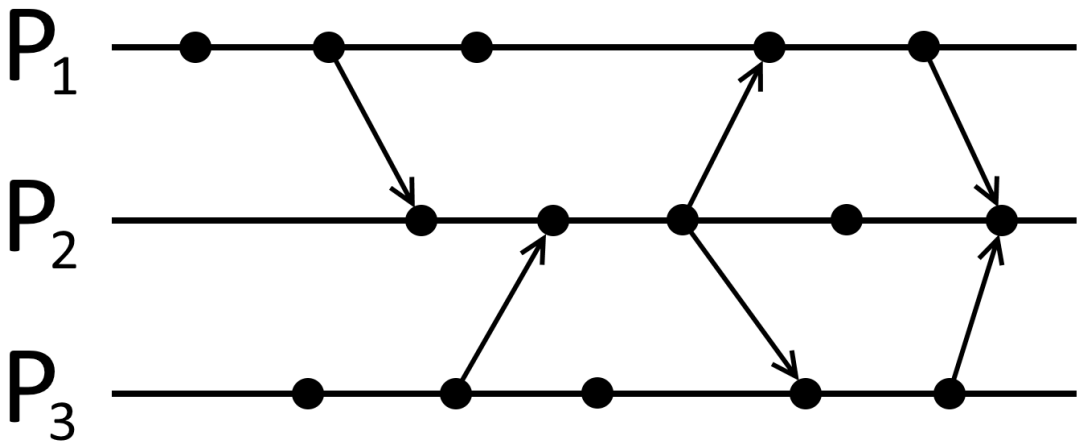

Figure 1.1 – A visualization of `sample.json`.

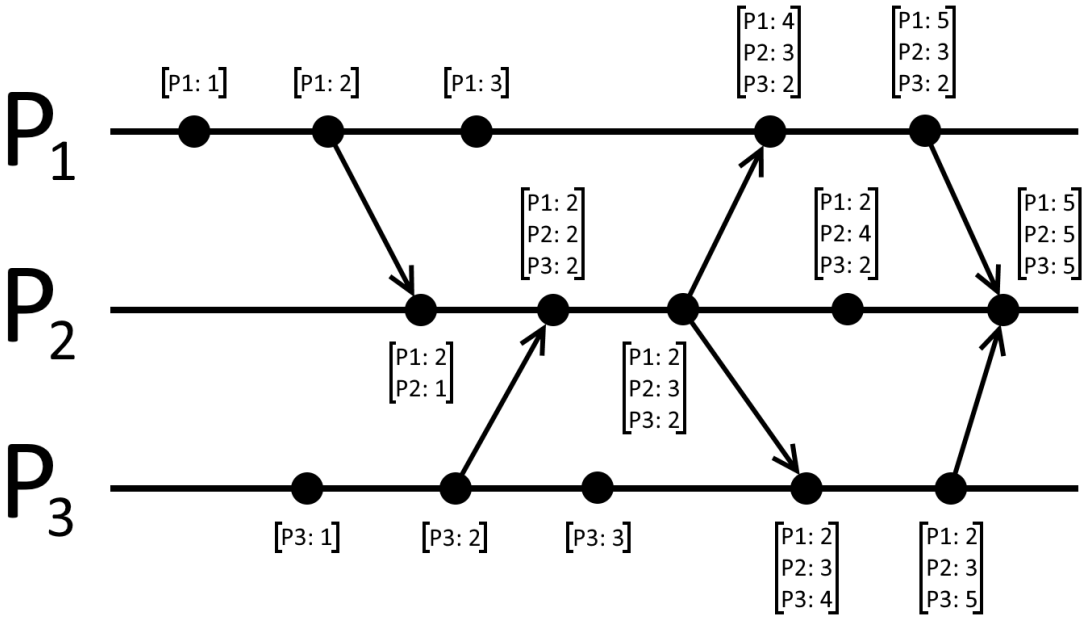

Figure 1.2 – The timestamps generated for every event in `sample.json`.

Evidently these timestamps correctly impose a partial ordering on the events. Figure 1.3 demonstrates which events are well-ordered in relation to the second event of the second process (P_2E_2, the purple node and timestamp), and which events are causally concurrent. The events highlighted in blue happened before P_2E_2, and the events highlighted in red happened after P_2E_2. Every other event is causally concurrent.

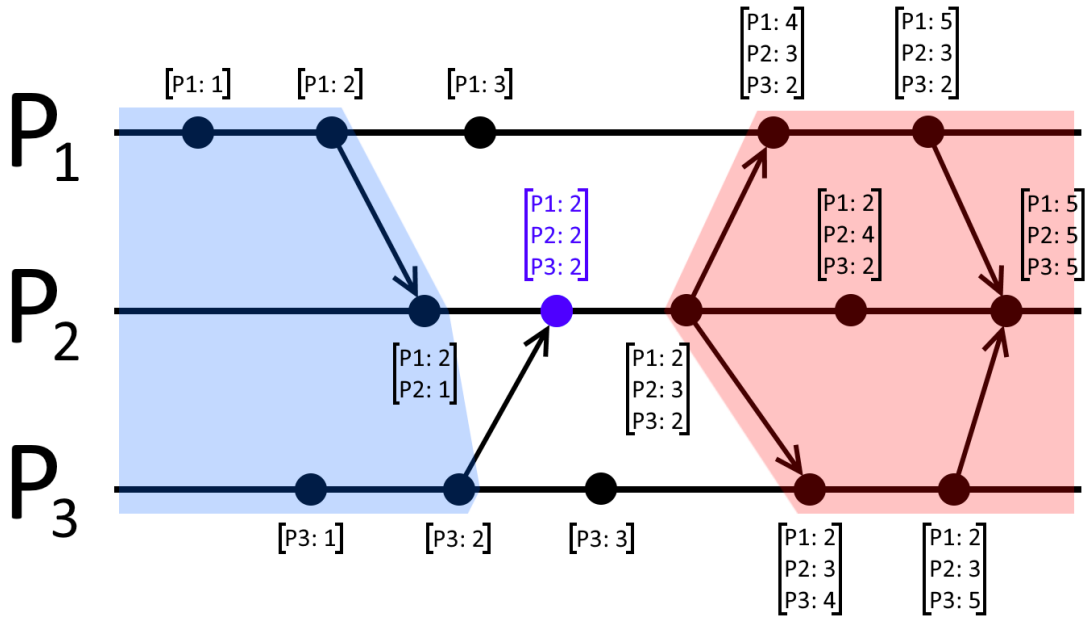

Figure 1.3 – The events ordered in relation to P_2E_2.

2 Byzantine Generals Problem

This section describes my implementation of Leslie Lamport's Oral Messages (OM) algorithms to solve the Byzantine generals problem. This section also covers the tests I have written to verify the correctness of the implementation, and the results obtained from those tests.

2.1 Description of the Implementation

My implementation of the OM algorithms (as described in Lamport's paper) is written in Go. To solve the problem, I have put together a data structure called a `ConsensusTree`. A `ConsensusTree` is a tree in which every node represents an invocation of OM(min{m, n} – k), where k is the depth down the tree at which the node sits (beginning at zero, and ending at the smaller of m or n). Note that when constructing a `ConsensusTree`, m is an integer equal to or greater than the number of traitorous generals, and n is the number of total generals.

Each node has a commander who will distribute their order, and a list of lieutenants who will receive the commander's order. The list of lieutenants is actually a map of lieutenant IDs to messenger channels on which orders will be passed. Note that because the tree uses channels, each general should run as a separate goroutine. Each node also has a number of children equal to the number of

lieutenants listed in the node. Each child of a node uses one of the lieutenants listed in its parent node as its commander, and all the remaining lieutenants continue to serve as lieutenants in the child. The commander in the root node is always the zeroth general.

Generals reach consensus with the tree by traversing it and communicating orders they have received to other generals using the channels at every node. A general does not need to traverse the entire tree. If a general reaches a node in which they are the commander, they do not proceed to any of the node's children, since they are not included in the child's list of lieutenants. At a leaf node, a general simply receives a value from their commander and passes it back up the tree. On the next node up, the general accumulates orders from child nodes where they were not the commander. Then the general takes the `majority` of the values accumulated at each child, and the value they received at the node they are currently in. The result is then passed up and the process repeats until the root is reached. It is worth noting that this implementation also allows the commander (the general with ID zero) to send out bogus values if they are a traitor. This is done in order to more faithfully conform to Lamport's description of the problem.

Figure 2.1 is a visualization of a `ConsensusTree` for four generals and one traitor. Each node in the tree corresponds to a round of the OM(i) algorithm with some commander sending their order to the listed lieutenants. Each level contains all of the OM(i) rounds.

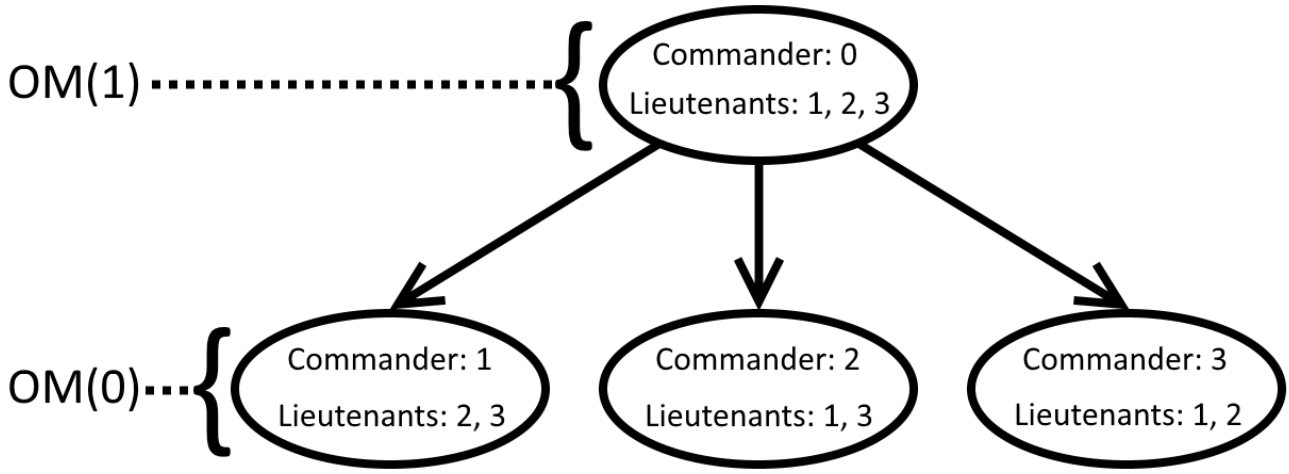

Figure 2.1 – A `ConsensusTree` for four generals and one traitor.

The implementation can be found at the following link.

https://github.com/MWindels/uvic-csc464-a2/blob/master/src/lib/p2/byzantine/byzantine.go

2.2 Description of the Tests

To test my implementation of the OM algorithms, I have written a testing suite that enables the user to test for correctness (i.e. consensus) by brute-force. The testing suite takes some number of generals (n) and some initial order from the user, and tests for consensus in the presence of zero up to $\left\lfloor \frac{n-1}{3} \right\rfloor$ traitors. For every number of traitors (m, from zero to $\left\lfloor \frac{n-1}{3} \right\rfloor$), the testing suite tests every

permutation of the generals where the IDs of the *m* traitors are permuted amongst the loyal generals. It is relevant to test this way due to the assignment specification. It specifies that all traitorous generals should send out the order they received if the recipient has an odd ID, and they should send out the opposite order if the recipient has an even ID. This is a deterministic behaviour, so its correctness can be tested by brute force. However, it is only feasible to test for small numbers of generals (i.e. no larger than sixteen or so). Otherwise the sheer number of permutations coupled with the running time of the OM algorithm (it is $O(n^m)$ to traverse the whole tree) will cause the testing to take a very long time to complete. The testing code is also written in Go, and can be found at the following link.

https://github.com/MWindels/uvic-csc464-a2/blob/master/src/tst/p2/consensus_tester.go

To run the tests yourself, use `go build consensus_tester.go` in the `src/tst/p2` directory to compile them. Then run the tests using `consensus_tester number_of_generals order`, where `number_of_generals` is the number of generals you want to test with, and `order` is the initial order you want to give (either attack or retreat).

2.3 Testing Results and Correctness

For all numbers of generals from two up to sixteen (the largest number of generals I tested with), the brute-force testing always succeeds. It is not exactly feasible to test with values larger than sixteen, because the running time of the test algorithm is $O\left(\sum_{i=0}^{m}\left(\frac{n!}{i!\,n-i!}n^i\right)\right)$ where *n* is the number of generals and *m* is the number of traitors. The factorial part is the number of ways to permute *i* (where $0 \leq i \leq m$) traitors amongst *n* total generals. The exponential part is the amount of time each general (running in a separate thread) takes to traverse a `ConsensusTree` with *i* levels. Some typical output of the tests (for a small number of generals) is demonstrated in Figure 2.2.

```
C:\University\Fourth Year\CSC 464\assignment_2\uvic-csc464-a2\src\tst\p2>consensus_tester 7 attack
Test (n = 7, m = 0, permute = .......) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = ......T) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = .....T.) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = ....T..) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = ...T...) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = ..T....) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = .T.....) passed.  Consensus: Attack.
Test (n = 7, m = 1, permute = T......) passed.  Consensus: Retreat.
Test (n = 7, m = 2, permute = .....TT) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ....T.T) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ....TT.) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ...T..T) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ...T.T.) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ...TT..) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ..T...T) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ..T..T.) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ..T.T..) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = ..TT...) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = .T....T) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = .T...T.) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = .T..T..) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = .T.T...) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = .TT....) passed.  Consensus: Attack.
Test (n = 7, m = 2, permute = T.....T) passed.  Consensus: Retreat.
Test (n = 7, m = 2, permute = T....T.) passed.  Consensus: Retreat.
Test (n = 7, m = 2, permute = T...T..) passed.  Consensus: Retreat.
Test (n = 7, m = 2, permute = T..T...) passed.  Consensus: Retreat.
Test (n = 7, m = 2, permute = T.T....) passed.  Consensus: Retreat.
Test (n = 7, m = 2, permute = TT.....) passed.  Consensus: Retreat.

All tests passed!
```

Figure 2.2 – Brute-force testing with seven generals.

Notice that while the original order was to attack, in all tests where the commander was a traitor (the tests highlighted in yellow) the consensus reached by the loyal generals was to retreat. This is correct behaviour. Recall that section 2.1 claimed that the commander could send out bogus orders if they were a traitor. In each of the highlighted cases in Figure 2.2, the commander has sent out inconsistent values to their lieutenants, but the loyal generals still share a common strategy: to retreat. While the strategy the loyal generals have decided upon is not what I (the user, not the commander) ordered them to do, it is still consensus. Therefore, these test cases are correct.

Note that the loyal generals do not always decide to do the opposite of what the original order was if the commander is a traitor. It just so happens that for seven generals, they decided to retreat every time. Figure 2.3 demonstrates a case where the loyal generals decided on different strategies for different permutations (when the commander was a traitor). What is important is that the loyal generals all agree on a shared strategy.

```
Test (n = 12, m = 3, permute = T...T......T) passed.  Consensus: Retreat.
Test (n = 12, m = 3, permute = T...T.....T.) passed.  Consensus: Attack.
Test (n = 12, m = 3, permute = T...T....T..) passed.  Consensus: Retreat.
Test (n = 12, m = 3, permute = T...T...T...) passed.  Consensus: Attack.
```

Figure 2.3 – Differing consensus for different permutations.

We always get consensus results thanks to the `ConsensusTree` data structure, which helps faithfully implement Lamport's OM algorithms.