Weighted Byzantine Agreement Implementation

Jason Z. Castillo
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
jzc248@utexas.edu

Tyler Carlson
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
tjcarlson25@utexas.edu

***Abstract*—This paper describes an implementation of the King and Queen algorithm for the Weighted Byzantine Agreement Problem. These algorithms will be evaluated for time and message complexity by adjusting the number of N nodes, L liars, and by using equal or random weights. Integrating weights into both of these multi-phase algorithms allows the algorithm to reduce faulty process weights down to zero, and increase correct process weights.**

***Index Terms*—Weighted Byzantine Agreement (WBA), Process (Node)**

I. Introduction

Although distributed algorithms are usually first formulated to consider cases where everything is optimal, this is not a reality. Failure occurs from hardware, mobile devices dropping in and out, and corrupted data. These issues mainly apply to distributed systems, as centralized systems can observe the fault by a central monitor. With a distributed system, each process must keep faulty or malicious processes from providing erroneous data. Three conditions must exist for a robust and fault-safe design protocol:

- **Agreement**: Two correct processes cannot decide on different values.
- **Validity**: The value decided must be proposed by some correct process.
- **Termination**: All correct processes decide in finite number of steps.

The Byzantine General Agreement (BGA) is the situation of multiple generals surrounding a city with their troops. They all have to decide and agree whether or not to attack or retreat. In a perfect scenario, all generals are loyal and there would be no issue taking over the city if all armies attack at once. Yet, there are treacherous generals that want to foil the attack by sending conflicting messages to disorganize the armies. There has to be an agreement amongst the loyal generals to have a successful attack. One solution to this problem is $f + 1$ rounds of messages through the generals (or processes). This allows for a true value to be agreed upon, in which there is an evaluation between different rounds to determine if there are discrepancies between each round.

The Queen Algorithm for Weight Byzantine Agreements goes through two phases. In the first phase, the processes only knows their own value. Each process broadcasts their value to the other processes and then receives the values from the other processes. From this vector of values from each process, the majority value will be set to the their new value. There is also a rotating main process (or queen) that is chosen for each round that will broadcast the queen value to all processes. In the second phase the process will send out its value and and then receive the queen value. At this point it will evaluate the vector values to determine if $N/2 + f$ copies of the value and it will chose this for its new value, else chooses the queen value. This new value will be the final output of all processes in the distributed system. Note that if a process doesn't receive any value from any process from a certain amount of time, it will consider that process to have failed and will continue on with its evaluation. The King algorithm adds another phase in the evaluation.

By integrating weights to the Queen and King algorithm, we can allow for more than N/3 processes to fail, as long as the total weight of failed processes is less than 1/3. Here each process P_i is assigned a weight $w[i]$ such that $0 \leq w[i] \leq 1$. The system needs to tolerate $\rho = f/N$ such that $0 \leq f < N/3$ in regard to weights of the processes. The notion of an anchor is denoted as α_ρ, which is defined as the least number of processes whose total weight exceeds ρ. The equation representation of the anchor is as follows.

$$\alpha_\rho = \{k | \sum_{i=1}^{i=k} w[i] > \rho\} \tag{1}$$

The Weighted-Queen algorithm can tolerate failures as long as ρ¡1/4 and α_ρ rounds while still remaining to be 2 phases. The Weighted-Queen Algorithm is provided in Figure 1 and still contains all the attributes of the non-weighted version.

The King-Algorithm allows for more failures with at $\rho < 1/3$, with α_ρ rounds and three phases. The Weight-King Algorithm is provided in Figure 2.

These algorithms satisfy the following Lemmas(Please see [1] for proofs):

- Lemma 1: There is no protocol to solve the WBA problem for all values of w when $\rho \geq 1/3$
- Lemma 2: Any protocol to solve the WBA problem for a system $\rho < 1$ takes at least α_ρ rounds of messages, in the worse case.
- Lemma 3: (Persistence of Agreement): Assuming $\rho < 1/4$ if all correct process prefer a value v at the beginning

Identify applicable funding agency here. If none, delete this.

```
P_i::
var
 V: {0,1} initially proposed value;
 w: const array[1..N] of weights;
      initially ∀j : w[j] ≥ 0 ∧ (∑_j : w[j] = 1)

 for q := 1 to α_ρ do

   float s0,s1 := 0.0, 0.0;

   first phase :
     if (w[i] > 0) then
       send V to all processes including itself;
     forall j such that w[j] > 0 do
       if 1 received from P_j then
         s1 := s1 + w[j];
       else if
       0 received from P_j or no message from P_j
       then
         s0 := s0 + w[j];
     if (s1 > 1/2) then
       myvalue := 1; myweight := s1;
     else myvalue := 0; myweight := s0;

   second phase:
     if (q = i) then
       send myvalue to all other processes;
     receive queenvalue from P_q;
     if myweight > 3/4 then
       V := myvalue;
     else V := queenvalue;

 endfor;

 output V as the decided value;
```

Fig. 1. Weighted Queen Algorithm.

of a round; then, they continue to do so at the end of the round.

- Lemma 4: There is at least one round in which the queen is correct
- Lemma 5: $\alpha_{f/N} \leq f + 1$ for all w and f.
- Lemma 6: (Persistence of Agreement): Assuming $\rho < 1/3$ if all correct process prefer a value v at the beginning of a round; then, they continue to do so at the end of the round.
- Lemma 7: There is at least one round in which the king is correct
- Lemma 8: In the Weighted-Queen algorithm, a correct process P_i can detect that P_j is faulty if any of the following conditions are met:
  1) If P_j either does not send a message or sends a message with wrong format in any of the round, then P_j is faulty.
  2) If $myweight > 3/4$ in any round and the value sent by the queen in that round is different from myvalue, then the queen is faulty
- Lemma 9: All correct processes with positive weights before the execution of the algorithm have identical w

```
P_i::
var
 V: {0,1,undecided} initially proposed value;
 w: const array[1..N] of weights;
      initially ∀j : w[j] ≥ 0 ∧ (∑_j : w[j] = 1)

 for k := 1 to α_ρ do

   float s0,s1,su := 0.0, 0.0, 0.0;

   first phase :
     if (w[i] > 0) then
       send V to all processes including itself;
     forall j such that w[j] > 0 do
       if 1 received from P_j then
         s1 := s1 + w[j];
       else if 0 received from P_j then
         s0 := s0 + w[j];
     if (s0 ≥ 2/3) then V := 0;
     else if (s1 ≥ 2/3) then V := 1;
     else V = undecided;

   second phase:
     s0,s1,su := 0.0, 0.0, 0.0;
     if (w[i] > 0) then
       send V to all processes including itself;
     forall j such that w[j] > 0 do
       if 1 received from P_j then
         s1 := s1 + w[j];
       else if 0 received from P_j then
         s0 := s0 + w[j];
       else su := su + w[j];
     if (s0 > 1/3) then
       V := 0; myweight := s0;
     else if (s1 > 1/3) then
       V := 1; myweight := s1;
     else if (su > 1/3) then
       V := undecided; myweight := su;

   third phase:
     if (k = i) then send V to all other processes;
     receive kingvalue from P_k;
     if V = undecided or myweight < 2/3 then
       if kingvalue = undecided then V = 1
       else V = kingvalue

 endfor;

 output V as the decided value;
```

Fig. 2. Weighted King Algorithm.

vectors after the execution of the algorithm

The goal of this paper is to implement both the King and Queen algorithms, and then evaluate them for time and message complexity. We will evaluate the following different situations:

- How does the number of nodes affect the time and number of messages sent when all weights are equal?
- How does the number of nodes affect the time and number of messages sent when weights are randomized?
- How does the number of liars affect the time (assuming a preset unequal set of weights)?

II. Implementation

The implementation of both algorithms are done in Java. The King and Queen algorithm have similar variables and construction. The real differences come to light when we look at the way each node's values is determined. Because of these similarities, we incorporated a few instances of polymorphism, which can be seen in Figure 3.

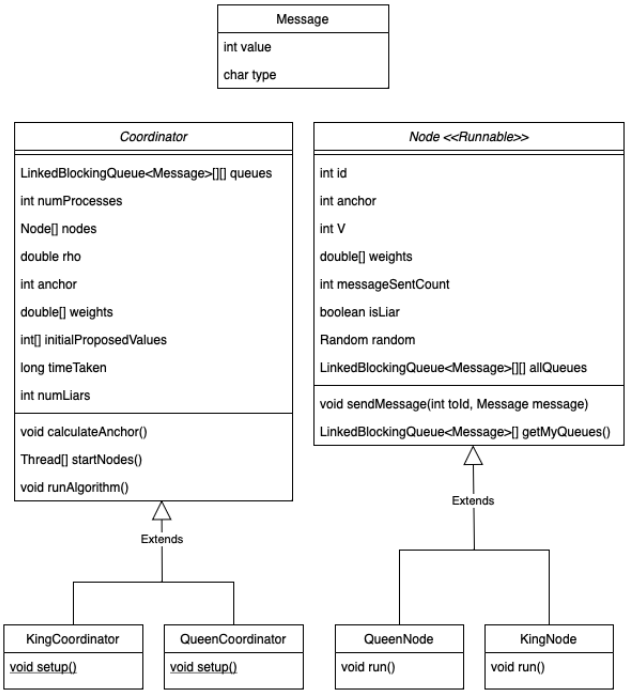

Fig. 3. Java UML Diagram

There is a common *Coordinator* and *Node* class, which each algorithm has its own implementation of. We decided to *mimic* a distributed system with threads, which was much easier to get up and running. For our case, a node is equivalent to a process, which extends the *Runnable* interface. Each Node's ID is just an increment starting from 0 (i.e. the first Node's ID is 0). The Coordinator sets up all nodes, queues, weights, initial proposed values, ρ value, α_ρ value, and other metadata. When the coordinator method *runAlgorithm()* is called, each thread (Node) is started and the algorithm is run with the set values. Once the algorithm is finished, the metadata, such as time taken and messages sent, can be extracted from the coordinator.

The only differences between a King and Queen Coordinator is that of the ρ value set and how many phases they go through. The Queen's ρ is set to $1/4$, while the King's ρ is set to $1/3$.

The construction of the Coordinators and Nodes was fairly easy. However, passing messages between Nodes was a little more complex. Because we are using threads to mimic a distributed system, we needed a protocol for inter-thread communication. We accomplished this by utilizing a 2D array of *LinkedBlockingQueue*, of type *Message*. A Message is a wrapper class that holds an integer value and the type of message (i.e. 'Q' for queen message or 'V' for a normal value message). The type of message isn't used in the program, but rather to allow for easier debugging. The 2-dimensional array is setup as follows.

Each first array index (i) corresponds to the Node i's array of queues. Each second array index (j) corresponds to the specific queue in Node i's array of queues, which is the queue to hold all messages received from j. For example, in a system of 3 nodes, we will have the 2D array look like this: $[[q0, q1, q2], [q0, q1, q2], [q0, q1, q2]]$. If Node 0 wants to send a message to Node 1, then we would add a Message to the following queue like this: $queues[1][0].add(message)$. This is encapsulated in the *Node* abstract class method $sendMessage(toId, Messagemessage)$.

Another key was to set the timeout of receiving a message. We didn't want it to be too long, or the algorithm would take too long for failed messages. We also didn't want it to be too short, or the receiving process could assume the node failed, even though the message may still be in transit. Because we are using threads, communication time is very quick. This allowed us to set a timeout of $250ms$.

Finally, we get to the actual algorithm implementation, which was fairly easy. Because each Node is also a thread, we can simply put the algorithm code in the $run()$ method. After each Node finishes, they will be collected in the main method which started them.

III. Evaluation

For the first two tests, we need to evaluate how the number of nodes affects the message and time complexity for both equal and randomized weights. For the third test, we need a way to tell a node to lie.

For the randomized weights, we simply generated n random doubles between 0 and 1, and then normalized them by dividing each by n. This got us n weights that sum to 1.

We should also note that the initial proposed values are split 50/50. Half of the nodes start with a proposed value of 1, while the other half starts with 0.

For liars, we tell the coordinator how many liars we want for our current test. Each node also has a $isLiar$ boolean variable that will help us later. When the coordinator creates each node, it tells the node if it is a liar or not. It determines if each node is a liar based on if the node's ID is less than the number of liars requested. On the node's side of things, if it is a liar, then anytime it needs to send a message to another node, it will instead send the opposite of its actual value.

When performing the liar test, we also did *not* use randomized weights. Instead we created our own distribution of weights to be consistent. The first half of the weights sum to 0.25 (equal distribution), a quarter of the weights sum to 0.5, and the last quarter sum to 0.25. This gives us many small weights at the beginning so we can properly see if more lying nodes can be tolerated.

For evaluation, there is a *Main* class, which creates different coordinators based on these different testing scenarios. The program takes approximately 10 minutes to run. The first 2 tests were to see how the number of nodes affects the time and message complexity when weights are all equal and then randomized. This is done starting from 20 nodes, all the way up to 760 nodes (in increments of 20). After running the king and queen algorithm for each of these scenarios, 4 CSV files are built for store results. The same process is completed for the final test, which is seeing how the number of liars affects the runtime of each algorithm when weights are preset with the values discussed above. 4 CSV files are created for these results. In total we have the following output files: *EqualWeights-King.csv, EqualWeights-Queen.csv, RandomWeights-King.csv, RandomWeights-Queen.csv, Liars-EqualWeights-King.csv, Liars-EqualWeights-Queen.csv, Liars-UnequalWeights-King.csv, Liars-UnequalWeights-Queen.csv*

IV. Results

Three graphs are produced based on the CSV output files. Figure 4 shows the time taken (in milliseconds) based on the number of nodes used for the King and Queen algorithm, each with Equal and Random weights. Figure 5 shows the number of messages sent (in millions) based on the number of nodes used for the King and Queen algorithm, each with Equal and Random weights. Finally, Figure 6 shows the time taken (in milliseconds) based on the number of Liars present in the King and Queen algorithm, where weights are equal.

We can see that the time and message complexity of each algorithm are less when randomized weights are used, instead of equal weights. Also, the queen algorithm generally showed better performance than the king algorithm.

When looking at the number of liars, there is surprisingly not much effect on the time complexity. The queen algorithm performed better than the king, but both times stayed pretty constant as the number of liars increased.

Lastly, Figure 7 shows a comparison of both algorithms and how many liars can be supported while still coming to an agreement. Clearly, we can see that when unequal weights are used, both algorithms can support many more liars if their sum of weights is less than ρ. However, the King algorithm can support more liars than the queen algorithm.

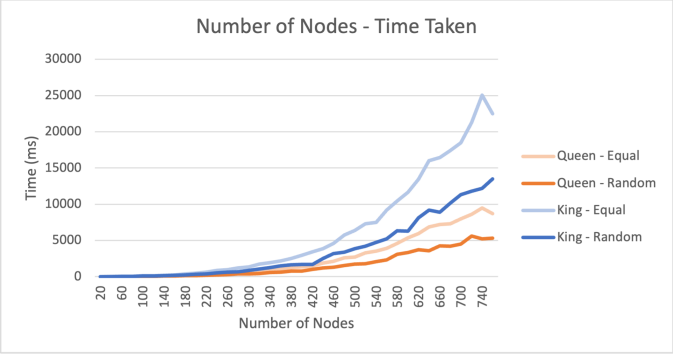

Fig. 4. Time Taken vs Number of Nodes

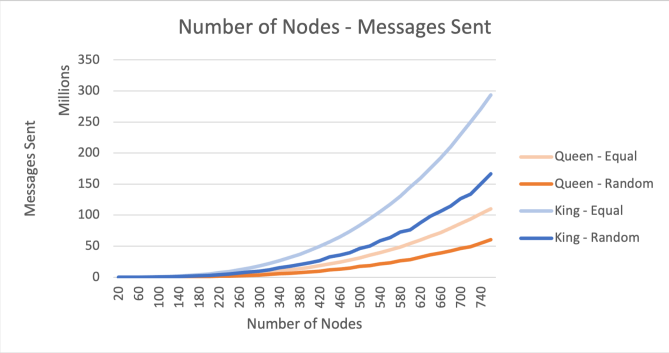

Fig. 5. Number of Messages Sent vs Number of Nodes

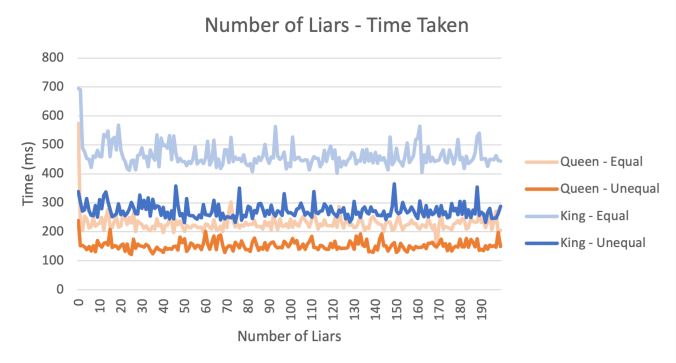

Fig. 6. Time Taken vs Number of Liars

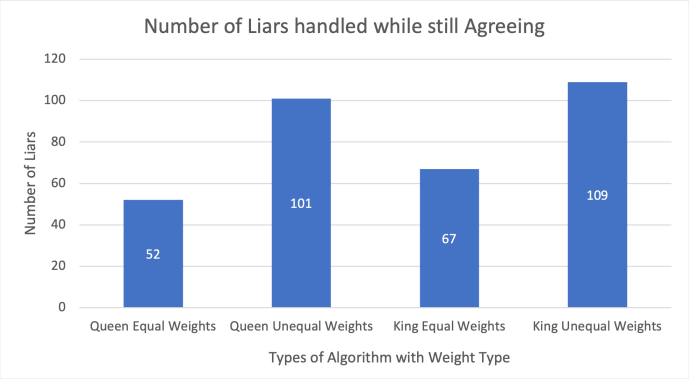

Fig. 7. Comparison of number of liars supported while algorithm is still in agreement

V. Conclusion

After evaluating of the weighted Queen and King algorithms, it is clear that the Queen has a better runtime than the King. This makes sense when taking a look at the extra phase in the King algorithm. As the time grows exponentially, we can confirm that although this is a good algorithm for coming to agreement with liars/failures, its application would be limited to a smaller set of nodes. It was observed, as expected, that if too many faulty nodes exist, the system will fail to come to agreement. The system coming to agreement is based on the

initial assigned weights. If all assigned weights are distributed equally, this will mimic the original non-weighted version of the King and Queen algorithm. The weighted King and Queen algorithm allows for a good improvement to the original solution to the byzantine agreement problem, and should be used for a reliable distributed system.

REFERENCES

[1] Vijay K. Garg, and John Bridgman, “The Weighted Byzantine Agreement Problem,” The University of Texas at Austin

[2] Vijay K. Garg, ”Agreement,” chapter 14 A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.