The Weighted Byzantine Agreement Problem

Nima Dini and Chenguang Liu
Department of Electrical & Computer Engineering
The University of Texas at Austin
Austin, TX, USA
{nima.dini, liuchg}@utexas.edu

***Abstract*—The Byzantine Agreement (BA) Problem [1] is a famous consensus problem in distributed systems [2]. The problem discusses how a set of individual processes in a network can reach an agreement in the presence of failures like crashes, byzantine-faults, etc. In this report, we aim to solve and evaluate the Weighted Byzantine Agreement (WBA) problem [3], which is a variation of the BA. We implement two famous algorithms, Queen and King, which both solve the WBA problem, differing in the ratio of fault they can tolerate. We design our system in *Actor Model Paradigm* [4] using *Scala* programming language [5] and *Akka Actor Library* [6]. We further evaluate our implementations at scale, by designing two experiments: (1) evaluate the functionality of our system for networks containing hundreds of nodes. (2) measuring the effect of different weight assignment and initial value assignment and comparing King and Queen algorithms under two different scenarios.**

***Keywords*—Distributed Consensus, Byzantine Agreement, Fault Tolerance, Actor Model**

I. Introduction

How a set of processes, communicating by message passing, can reach an agreement is a fundamental problem in distributed computing. Fischer, Lynch, and Patterson proved the impossibility of using asynchronous communication model to solve the binary consensus problem even with the presence of one unannounced *fail-stop* process [7]. Since then, a considerable amount of research has focused on solving this problem, under the assumption of synchronous communication. Additionally, many have tried to propose more fault-tolerant algorithms, both in terms of ratio and diversity of failing processes. In this report, we focus on the Byzantine failure model [1].

In the Byzantine failure model, a process may fail with arbitrary behaviors [8], with or without halting. Additionally a process may return unfaithful arbitrary values to disrupt the coordination and foil any protocol. In the classic Byzantine Agreement problem, there are N individual processes in a distributed system and f number of them are traitors, which may present Byzantine faults in an arbitrary fashion. The goal is to make a protocol such that all the *correct* processors can agree on a binary value by following it, regardless of the behavior of the byzantine nodes. Lamport, Shostak, and Pease [1] proved this problem is solvable if and only if more than *2/3* of the generals are correct and follow the protocol. They further proved that the algorithm requires $f + 1$ rounds of communication to terminate. This bound was later proved to be the greatest lower bound by Fischer and Lynch [9].

In 2011, Garg and Bridgman came up with the weighted version of this Byzantine Agreement Problem (WBA) and proposed two algorithms to solve it [3]. The primary difference of this variation from the traditional BGP is that every process in WBA has been assigned a weight $w[i]$ such that $0 \leq w[i] \leq 1$, and $\sum w[i] = 1$. The authors proposed two protocols, and proved theoretically that both are able to achieve lower bounds of the correct processes and provide higher fault tolerance compared to the previously proposed solutions.

In this report, we present our implementation for two weighted Byzantine Algorithms, WBA Queen (WBA_Q) and WBA King (WBA_K). The main goal is to evaluate how these two protocols work in action and to further compare them. Hence, we aim to determine the effectiveness of these two algorithms in various settings, like against systems with hundreds of nodes, different failure and weight assignment models. We used *Scala* programming language [5] and the *state-of-the-art* technique for developing distributed systems *Akka.Actor* [6] for our implementation. To generate test inputs, we implemented a highly configurable input generator, in Python. We used shell scripts to automate the evaluation process.

The rest of this report is organized as follows: First, we discuss some required background knowledge in Section II. Section III presents our implementation in more depth. Section IV conducts a series of experiments to validate the effectiveness of our implementations. Finally, section **??** concludes our project and summarises the key findings of our project.

II. Background

A. The Byzantine Agreement Problem

The BGP is originated from the attack coordination problem of the generals with unreliable communication, near the city of Byzantium in the ancient Roman Empire. According to the anecdote, each of the generals were isolated, but needed to reach a consensus about a plan of action: *attack*, or *retreat*. This simple-looking problem, has some interesting challenges. First, some generals may be treacherous, meaning they know about the protocol, and try to disrupt others to prevent an agreement. They can behave arbitrarily by sending random values to others or not responding to others intentionally. Next, the messenger may be forged which leads to untrusted communication links.

As discussed in the introduction section, there is a classic solution to this problem as long as the number of treacherous generals does not surpass one third of the generals inclusively. The classic solutions are based on 3 more assumptions: (1) every message sent by a non-faulty processor is delivered correctly; (2) a processor can determine the originator of any message that it receives; (3) the absence of a message can be detected by a process. Any system which meets this set of requirements, under the assumption that the correct processes always follow the protocol, can reach an agreement in $f + 1$ rounds [1], where f is the number of faulty processes, and the result would satisfy the following two conditions: (1) All non-faulty processors must use the same input value, to produce the same output. (2) If the input unit is non-faulty, then all non-faulty processes use the value it provides as their input, to produce the correct output.

B. The Weighted version

The WBA problem was introduced by Garg and Bridgman [3]. Instead of limiting the number of faulty processes to f out of N processes, the WBA problem assumes every process is assigned with a weight $w[i]$ and the goal is to make the consensus reachable under the condition that the total weight of faulty processes is not greater than f/N. The solution to this problem is especially useful when a subset of the processes are more *trustful*. In the same paper the authors further proposed two algorithms with different tolerance ratios. The WBA_Q is the generalized version of the algorithm by Berman and Garay [10], which can tolerate any combination of failures as long as the sum of their weights is smaller than $1/4$ ($\rho < 1/4$). In WBA_Q each process takes 2 phases in a round and the consensus will be reached in α_ρ rounds where $\alpha_\rho = min\{k | \sum_{i=1}^{i=k} w[i] > \rho\}$. The parameter α_ρ is called the *anchor* of the system. Similarly we have the WBA_K algorithm [11] in which the tolerance ratio has been further improved to $\rho < 1/3$.

C. Actors, Scala, and Akka

Parallel computing and distributed computing follow the same basic computation model: a set of distributed processes that work concurrently and communicate with one another to do one task as a whole [12]. However, traditionally these two computing models have evolved as separate research disciplines. The main reason is that parallel computing address problems of communication-intensive computation on tightly-coupled processors while distributed computing has been concerned with coordination, availability, etc., of more loosely coupled systems.

Actor model provides a flexible computation paradigm which supports both parallel computing and distributed computing [13]. Actors are objects because they encapsulate data, method, and interface. In addition, actors are autonomous, which means they encapsulate threads of control as well. Actors communicate with other actors using message passing. The messages passing system in actor model is asynchronous and non-blocking. Each actor has a queue, for receiving and temporarily storing messages, which is assigned to its globally unique mailing address. In response to a message, an Actor may send messages, create Actors, or make some changes to its local data.

Scala, stands for SCALAble language, is a general-purpose high-level programming language, which fuses functional and object oriented programming paradigms [14]. Akka is a toolkit and runtime environment symplyfing the construction of concurrent and distributed systems on JVM. Using the actor model, Akka raises the abstraction level and provide a better platform to build scalable, resilient, and responsive applications[1]. To deal with fault-tolerance, Akka adopts the famous "let it crash" model from Erlang [15]. According to this model, processes that cannot perform a task should crash immediately and let another process correct the error. This is exactly against the traditional defensive programming paradigms.

III. Implementation Details

In this section we discuss our implementations of WBA_Q and WBA_K algorithms in three sections: (1) message exchanging, (2) message processing and synchronization, and (3) modeling faulty processes. We first focus on the WBA_Q algorithm to exemplify the key features of our implementation.

A. Message exchanging

To enable message exchanging in a distributed setting, every process should be able to send and receive messages at anytime. Sending messages is a fairly simple task as every programming language provides its point-to-point message passing interface. However, receiving messages is more challenging, because it requires the message queuing, receiving order (e.g., to enforce *first-in-first-out*), and the non-blocking communication.

In our implementation, a process is modeled as an instance of a class, extended from *Akka.actor*. This inheritance allows us to utilize the following two types of message passing:

$actorRef$! $message$ // tell

$actorRef$? $message$ // ask

! is a method call for sending a message asynchronously and return immediately. In other words, it is fired in a non-blocking fashion, so the Akka document calls it the *fire-and-forget* method[2]. For instance, a process can send out its proposed value to all processes, as needed in the first phase of both algorithms, by using this one line command:

$generals\ foreach$ (_ $\Rightarrow$ _ ! phaseOneValue(V))

Alternatively, an actor can send an asynchronous message and wait for the response using the ? (*ask*) method. This method returns a *Future* object to represent a possible action when the response is back. Specifically, this *Future* object forks a new thread within the process and waits for an expected result. An actor uses ask method is explicitly say it waits for a definite time for the response to arrive. For example, to advance a process into the next phase when the process has received a certain message from the queen process, we used the following code snippet:

implicit val $timeout$ = Timeout($delay$1 milliseconds)

val f: Future[Value] = (queen ? value).mapTo[Value]

$f\ onSuccess\ \{\ //decide\ on\ a\ value\ \}$

For the message delivery reliability our code follows the *exactly-once* rule in akka library as it guarantees for each message handed to the mechanism exactly one delivery is made to the recipient (as opposed to the two other options: *at-most-once* and *at-least-once*). Note that we override the behaviors of the Byzantine processes to simulate the failures, we will get to that later in this section. The messages are delivered in order per sender-receiver pair.

B. Message Processing and Synchronization

Scala provides strong abstraction of concurrent operations. As of message processing we tried to utilize one of the common features - the *Future* data structure. As discussed in the previous sub-section, the messages between a pair of sender-recipient are delivered in order, with the support of this *Future* Objects we can control the individual executions synchronously regardless of the underlying network layer. The *Future* construct needs an *ExecutionContext* plus a concurrent operation to behave as a placeholder, triggering the logic as a callback if the response is back.

In the example discussed in the last sub-section, we use a Future to wait for a *QueenValue* from the queen process in a specific round. Only when this *Future[QueenValue]* is received or the timeout specified with this object has passed, this process shall advance to its next round. This guarantees that there is no

[1] http://doc.akka.io/docs/akka/current/AkkaJava.pdf

[2] http://doc.akka.io/docs/akka/snapshot/scala/actors.html

process in the entire system which can advance itself to the next round, prior to a *correct* queen process.

C. Modeling Faulty Processes

The key issue in solving BGP and WBA problems is to tolerate the Byzantine faulty processes in the system. This is also the primary factor in validating the effectiveness of the proposed protocols. Therefore in our implementation we have delicately modeled the adversaries. As shown in figure 1, in the traditional BGP, we have three types of generals: normal general, traitorous general, and crashed general. The normal generals are the correct processes in a system. Traitorous generals are the ones which send arbitrary values to other processes. Finally, the crashed generals suffer from omission failures and then stop responding. We have implemented all the three of these processes. Through a configuration file, the system can be initialized with any portion of either number of faulty processes or a combination of them.

To simulate the adversaries more realistically, we considered two modes for the Byzantine processes: normal and *extreme*. In this normal mode the Byzantine processes generate arbitrary values and send them out to the other processes. Because all processes in the system know the default value $V_\perp$, in the *extreme* mode the adversarial processes participate in a conspiracy such that they all disseminate the value other than $V_\perp$, i.e., if $V_\perp = 1$ then the adversarial processes always return 0. In addition, we create as many as correct processes to disseminate the same adversarial value (0 in the example), under the condition that the remaining correct processes can still have their cumulative weights $(\sum w[i] \geq (1-\rho))$.

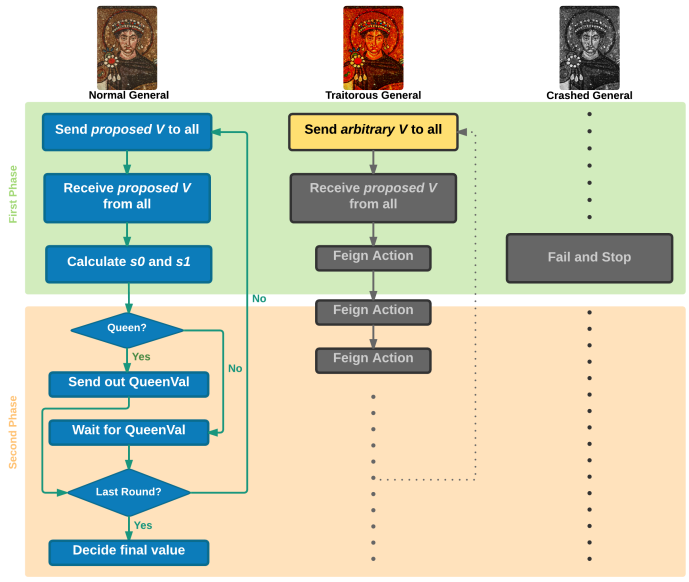

Fig. 1: General types used in $WBA_{Q/K}$

IV. Evaluation

Evaluating a distributed system is not a trivial task [16]. To show the correctness of our implementations we designed two main experiments: First, we tested each implementation separately against different auto-generated inputs. Next, we evaluated the effect of different weight assignments for each algorithm separately and next compared them together. In this section, we show our two key experiments in detail and discuss the obtained results [3].

[3]The code and data collected are available at https://github.com/nimadini/DistributedSystems/tree/master/WeightedByzantineAgreementProblem

A. Experiment I

Experiment I simply focuses on testing the functionality of our implementations. We used a python script to automate the process of test input generation. The script takes 5 inputs: (1) number of nodes, (2) WBA version, i.e., queen or king, (3) number of byzantine general types, in our case only 2, (4) default agreement value, (5) timeout for each phase.

Given the input arguments, our script produces an output containing three parts: (1) the analysis of the auto-generated input to help us in writing this section. (2) a valid problem input to feed into our WBA implementations. (3) an Akka configuration to specify the resources of our distributed system.

The weight and type of each process is selected by a random number generator with uniform distribution. The algorithm tries to greedily maximize the overall weight of the failing processes while respecting the threshold required by the WBA algorithms. In our experiment, we chose 0 as the default consensus binary value and 1000 milliseconds as the default delay for our synchronous communication.

Figure 2 shows a sample auto-generated configuration file for WBA_Q. The comments at the beginning of the file shows an overview of the generated test, which we used in writing this section. In this configuration file, $Akka$ is the main element and it has two children, $root$ and $actor$. root contains is the main input to our WBA problem algorithm and actor specifies the resources for Akka under which we ran our WBA implementations.

```
# ro (byz weight): 0.1783439
# number of correct generals: 2
# number of byz type1: 0
# number of byz type2: 1
# expected output cannot be 1

akka {
  root {
    initialVals = [0, 0, 0],
    procTypes = [0, 0, 2],
    Weights = [0.4267515923566879,
        0.39490445859872614,
        0.17834394904458598],
    defaultVal = 0,
    delay = 1000
  },
   actor {
      default-dispatcher {
         throughput = 10
         type = Dispatcher
         executor = "fork-join-executor"
         fork-join-executor {
            parallelism-min = 2
            parallelism-factor = 2.0
            parallelism-max = 40
         }
         throughput = 10
      }
   }
}
```

Fig. 2: A sample auto-generated configuration for WBA_Q

We used the python script to generate various inputs for our implementations. Table I and II show the result of our experiments for WBA_Q and WBA_K respectively. We evaluated our implementations for 7 inputs as discussed in the tables. Each

table shows 7 different values per run: (1) $nodes$ shows the number of processes (generals) in the system (2) $correct$ counts number of correct processes (3) byz_1 is the number of byzantine processes of type 1, which never respond to any request message (4) byz_1 stands for the number of byzantine generals of type 2, which return an arbitrary binary value for each request (5) ρ means the overall ratio of failing processes (6) α_ρ indicates the number of rounds the algorithm will converge (7) $agreed$ is the final consensus value

run	$nodes$	$correct$	byz_1	byz_2	ρ	α_ρ	$agreed$
1	3	2	0	1	0.1783	1	0
2	5	4	1	0	0.2212	1	0
3	10	7	1	2	0.2413	2	1
4	25	17	5	3	0.2462	4	1
5	50	35	11	4	0.2491	8	0
6	100	75	13	12	0.2497	16	0
7	1000	741	118	141	0.2499	150	0

TABLE I: WBA_Q execution results

run	$nodes$	$correct$	byz_1	byz_2	ρ	α_ρ	$agreed$
1	3	2	0	1	0.3058	1	1
2	5	3	1	1	0.3267	2	1
3	10	6	2	2	0.3172	3	0
4	25	16	4	5	0.3314	5	0
5	50	34	8	8	0.3320	11	0
6	100	61	20	19	0.3321	21	0
7	1000	665	176	159	0.3331	202	0

TABLE II: WBA_K execution results

B. Experiment II

We designed this experiment to evaluate the effect of changing Initial Consensus Ratio (ICR) of correct processes. This metric shows what percent of correct processes share the same initial binary value. A higher ICR value for correct processes indicates that the initial values of the majority of such processes are more *desirable*. The other key properties of this experiment are as follows:

- the distributed system has 100 nodes
- maximum allowed weight for byzantine generals is considered, i.e., $1/4 - \epsilon$ and $1/3 - \epsilon$ for WBA_Q and WBA_K respectively where ϵ is equal to 0.0001
- the default value $V_\perp$ for both of the algorithms is 0

Algorithm 1 shows the pseudocode of our automated script. It generates 11 different valid inputs for different ICRs, ranging from 0% to 100% (with 10% increase at a time), and execute each one 10 times to determine the cumulative frequency of $0s$ and $1s$ per each ICR. Additionally, we updated our python script to ensure the above properties hold.

For each of the WBA_Q and WBA_K, we considered 2 variations as follows:

1. The Extreme Scenario:

- all the byzantine generals pick 0 as the initial value due to a collusion

```
for i = 0 : 10 do
    config := python_script(
        algo = Queen/King,
        size = 100,
        byz_general_types = 2,
        default_val = 0,
        ICR_for_Correct_Processes = i,
        delay = 1000
    )
    for i = 0 : 9 do
        run WBA_{Q/K}(config)
    end
end
```

Algorithm 1: Experiment II Automation Script Pseudocode

- $(100 - ICR)\%$ of correct processes pick 0 as the consensinitialus value by coincidence

2. The Normal Scenario:

- the byzantine generals select a random binary value with uniform distribution as the initial value
- $(100 - ICR)\%$ of correct processes select a uniformly distributed random binary as the inital value

Figure 2 shows the result of 4×110 (440 total) executions on a distributed system of 100 nodes. As we expected, the extreme variations are more agnostic to the increase of x-axis value, while the normal ones react sooner to the same increase. Additionally, WBA_Q in both variations converges faster towards the dominant consensus value as the x-axis increases, compared to WBA_K. Finally, the values of all the 4 variations at the left and right boundary of x-axis is 0 and 1 respectively. This value is expected for any valid implementation of WBA algorithm.

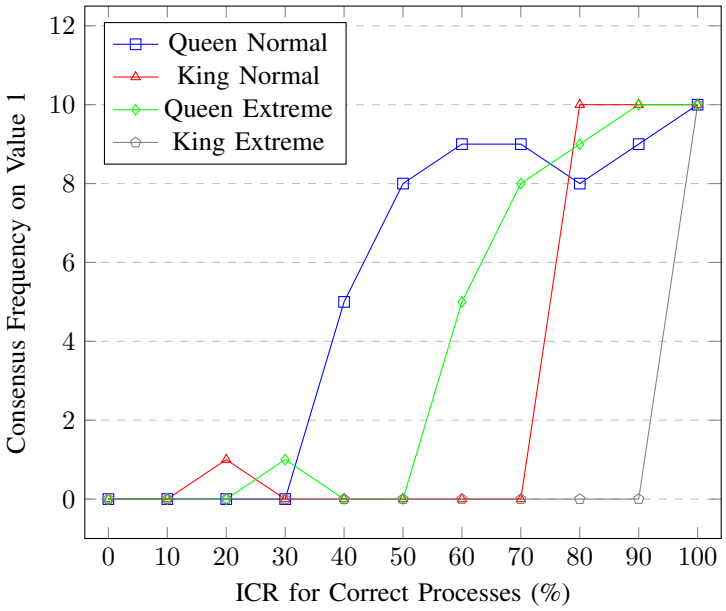

Fig. 3: Experiment II Result

V. Conclusions and Future Work

In this project, we described our implementation of the WBA_Q and the WBA_K algorithms to solve the *WBA* problem.

We started by discussing the backgrounds of *BGA* and *WBA* and continued by presenting the key design details of our implementation. Next, in this report we evaluated two interesting experiments to evaluate the effectiveness of the algorithms and the performance of the system.

The first experiment focused on the behavior of the system under different networks. Using $Scala$ and $Akka$, we were able to create realistic distributed systems, varying in size from 3 to 1,000 nodes. In the second experiment, we evaluated the ratio of correct processes, that should initially agree on a desirable value, so that the consensus happens on that desirable value. As discussed, we considered 2 different adversaries, the normal scenario with arbitrary behavior of byzantine generals, and the extreme scenario, in which the byzantine generals make a conspiracy to prevent the desired consensus. In total, we ran the two algorithms 440 total times over a network of 100 nodes for 11 configuration files per each ICR of correct processes (ranging from 0% to 100%). The extreme scenario pushes the boundaries, in the sense that byzantine processes make a collusion to prevent the desirable consensus. Our experiments show that WBA_Q acts better to agree on a desirable value, comparing to WBA_K. This finding match what we expected, in the sense that there are more byzantine generals in our WBA_K experiment. Our experiments further indicate that the extreme scenario for both of the algorithms, i.e., when byzantine processes form a collusion, requires higher number of correct processes to initially agree on the desirable value, comparing to the normal scenario.

One future work direction might be to apply our implementation in a mobile or web-based application. It may be very useful for implementing a coordination module in such applications. Another direction may be integrating more functionality (e.g., feedback adaption) with the current design of our system and make it highly customizable for other systems written by the Java/Scala languages.

REFERENCES

[1] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[2] EA Akkoyunlu, K Ekanadham, and RV Huber. Some constraints and tradeoffs in the design of network communications. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 67–74. ACM, 1975.

[3] Vijay K Garg and John Bridgman. The weighted byzantine agreement problem. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 524–531. IEEE, 2011.

[4] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[5] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[6] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[7] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[8] Vijay K Garg. *Elements of distributed computing*. John Wiley & Sons, 2002.

[9] Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. Technical report, DTIC Document, 1981.

[10] Piotr Berman and Juan A Garay. *Asymptotically optimal distributed consensus*. Springer, 1989.

[11] Piotr Berman, Juan Garay, Kenneth J Perry, et al. Towards optimal distributed consensus. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 410–415. IEEE, 1989.

[12] Gul A. Agha and WooYoung Kim. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45, 1999.

[13] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[14] Martin Odersky and al. An overview of the scala programming language. Technical report, EPFL Lausanne, Switzerland, 2004.

[15] Joe Armstrong. Erlang. *Commun. ACM*, 53:68–75, 2010.

[16] Samira Tasharofi, Milos Gligoric, Darko Marinov, and Ralph Johnson. Setac: A framework for phased deterministic testing of scala actor programs. In *The Second Scala Workshop*, 2011.