A Performance Comparison of Algorithms for Byzantine Agreement in Distributed Systems

Shreya Agrawal
Cheriton School of Computer Science
University of Waterloo
shreya.agrawal@uwaterloo.ca

Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
kdaudjee@uwaterloo.ca

***Abstract*—Reaching agreement in the presence of byzantine processes is an important task in distributed systems. Theoretical analysis of algorithms for Byzantine Agreement can provide insight into their efficiency. However, analysis of algorithms under varying parameters and practical constraints through experimental evaluation can be key to understanding the performance and trade-offs of theoretically well-performing algorithms. We compare the performance of two randomized byzantine agreement algorithms—one using the *pull-push* approach and another using the concept of *quorums*—and a third recent simple deterministic byzantine agreement algorithm. Through implementation on a testbed environment using the metrics of *bit* complexity, *round* complexity and *latency* in the presence of network sizes and faulty processes, we quantify the performance of each algorithm. In terms of bit complexity, we show that for small networks $(n < 32)$ and up to 10% faulty processes, the simple deterministic algorithm performs best, while for larger networks, *pull-push* is the best performing algorithm. The second randomized algorithm performs best in terms of latency.**

***Keywords*-Distributed systems; Performance; Byzantine failures; Fault-tolerant; Consensus; Complexity.**

I. Introduction

The Distributed Consensus problem introduced by Pease et al. [43] is one of the most important and well-studied problems in distributed systems. In essence, the problem deals with multiple processes, each with an initial opinion, cooperating with each other to reach agreement. The motivation for this problem arises from real-world use cases such as database transactions, where a number of systems need to agree on whether to commit or abort a transaction, in aircraft controllers that need to agree on which plane should take-off or land first, or in examples such as the Dragon flight system [1], a reusable aircraft, that is required to be fault-tolerant in cases when it gets close to the International Space Station. In the case of aircraft controllers, where the system is safety-critical, it is essential that they reach an agreement within a bounded period of time and that all the controllers arrive at the same decision, whereas in the case of database transactions, it is only necessary that the system eventually reaches an agreement. Therefore, any consensus reaching algorithm needs to satisfy certain correctness conditions that are formally given as follows [43]:

- *Consistency*: All correct processes agree on the same value and all the decisions are final.
- *Validity*: If all processes start with the same initial value v, then v is the only allowable decision for all processes.
- *Termination*: All correct processes reach a decision.

These correctness conditions also define the *safety* and *liveness* conditions that a distributed consensus algorithm must satisfy. The consistency and validity condition is a safety condition: safety will be violated if any two processes decide on different values. The termination condition specifies the liveness condition. For a system to continue executing correctly until it terminates, all processes must eventually reach the same conclusion.

In real-world applications, it would be unrealistic to assume that the systems involved in solving the problem will continue to work correctly, for the whole duration, as specified by the protocol. There may be cases such as when the communication links between systems break and the system *partitions*, the network becomes congested due to too many requests, or the messages are not delivered in order. To achieve reliability in distributed systems, protocols are needed which enable the system as a whole to continue to function despite the failure of some number of components.

It was shown by Lamport et al. that consensus can never be reached even between two processes if there is a link failure [32]. Under the assumption that the links do not display any kind of fault, one needs to analyze scenarios in the case of process failures. Generally, the kind of failures that a process can encounter is either a *fail-stop* failure or a *Byzantine failure*. Fail-stop failures occur when processes fail by stopping. In the case of a byzantine failure, the processes fail arbitrarily. Depending on the ratio of faulty processes to all the processes, consensus may be impossible even when the system is synchronous. The problem of solving consensus in the presence of byzantine failures is known as the Byzantine Agreement problem [32]. With the increasing number of malicious attacks reported in recent times and software bugs leading to processes behaving arbitrarily, dealing with this problem has become more important than ever before.

A. Need for Experimental Evaluation

In the past, researchers have tried to optimize either the *message* complexity or the *round* complexity of algorithms in the worst case scenarios. Worst-case bounds are insufficient to provide an insight into the trade-offs that exist between

978-1-5090-1582-5/16 $31.00 © 2016 IEEE
DOI 10.1109/EDCC.2016.17

these two types of optimizations under various conditions such as varying network sizes, and percentage of faults. However, experimental evaluation can help greatly by providing detailed results on guarantees of algorithms under such varying conditions. While the literature is rich in theoretical results, there has been a lack of extensive practical results for these algorithms. This can be largely attributed to the fact that simple algorithms are theoretically inefficient while the more complex ones are not easy to implement. Theoretical results give us information about the worst case scenarios but in practice, the system often does not run in the worst case.

In real-world systems, certain constraints—for example, communication bandwidths—may exist which are not considered in theoretical results. Such constraints can reduce the performance—if the available bandwidth is small—or can be used to our advantage by tweaking the algorithm to send multiple packets together instead of in separate rounds. Furthermore, analytical techniques using the Big 'O' notation can hide large constants which are of significance when we look at memory or bandwidth consumption in a real-world system. Many complex algorithms make certain assumptions about systems, for example, the existence of a large fraction of faulty processes. However, in practice, only a small fraction of processes are usually faulty and in such cases the algorithm would be unnecessarily complex and inefficient.

Analysis of algorithms under such varying conditions and practical constraints through experimental evaluation can be key to understanding the performance of theoretically well-performing algorithms. While some applications require best performance for either only number of bits required for communication or for the number of rounds taken to reach consensus, most applications [40] require optimal results for both size and number of faults in the application. A comparison, therefore, of performant algorithms is necessary to understand how they perform with respect to each other. A way to achieve this is by selecting state-of-the-art algorithms having different designs and implementing them on a common platform to compare their performance.

Motivated by these needs, we evaluate the performance of three recently proposed solutions for the Byzantine Agreement problem. Two of these are randomized algorithms—(1) by Ben-Or et al. [7], which we call algorithm *Quorum* throughout the paper, and (2) an almost-everywhere to everywhere algorithm by Braud-Santoni et al. [9] combined with the almost-everywhere algorithm of [28], which we call algorithm *Pull-Push*. An almost-everywhere algorithm reaches consensus among all but a fraction of the processes and an almost-everywhere to everywhere algorithm reaches consensus among all the processes. The third is a deterministic algorithm, due to Kowalski et al. [30], which we refer to as algorithm *EIG* (Exponential Information Gathering).

B. Motivation for Chosen Algorithms

A vast amount of work has been done over the years to solve the Byzantine Agreement problem, and different approaches have been taken for various models of the problem. In a well-known result, Fischer et al. [19] showed that reaching distributed consensus deterministically becomes impossible in an asynchronous system with even just one faulty process. Randomized algorithms allow us to overcome the barrier of this impossibility result in the asynchronous setting. Even in the synchronous setting, they provide major improvements over the deterministic algorithms by a factor polynomial in the size of the network. The probabilistic techniques provide an advantage—the worst-case scenarios can be eliminated by giving them a probability of 0 and a probability distribution is given in the other cases. In cyber-physical systems, logical synchrony is crucial as in the case of a flight system where the supervisory control is a global computation [45]. Many of the reactive systems that are safety-critical require determinism which is the target of synchronous programs [38], [31].

Algorithm *Pull-Push* is one of the most recent and best-known results in terms of communication complexity. The main contribution of this result has been to give an almost-everywhere to everywhere solution, which improves the amortized communication complexity to $\tilde{O}(1)$ per node[1]. The previously best-known result was $\tilde{O}(n^{1/2})$ [25]. The algorithm uses the model previously considered in [25], [28], [7], [26], that of a synchronous, full-information model, with a non-adaptive adversary. They also demonstrate that similar results can be shown even in the case of an asynchronous model. This achievement in reducing the communication cost, however, comes at the cost of a higher round complexity, which is poly-logarithmic in the size of the network.

This trade-off between communication and round complexity motivated us to pick the algorithm *Quorum* for comparison. This algorithm, due to Ben-Or et al., has a round complexity of $O(\log n)$ although it shows a quasi-polynomial communication complexity of $n^{O(\log n)}$. It uses the same setting as *Pull-Push* and makes use of the well-known Feige's protocol of collective coin-flipping to decide on a small committee with the same fraction of good processes as in the whole network, which then runs a leader election protocol for agreement. A comparison of these two algorithms will allow us to understand how they perform with respect to each other for both communication as well as round complexity.

While both these randomized algorithms attempt to improve upon the metrics for performance comparison, they consider worst-case scenarios when the fraction of adversarial processes is $n/3$ and $n/4$, respectively. However, in a real-world system, this fraction can actually be quite small. This motivated us to choose a deterministic algorithm for comparison which considers this fraction during its execution and thus provides much better results for both communication and round complexity when the number of adversarial nodes to the size of the network is really small. *EIG* is a recent deterministic algorithm, more efficient than the one by Garay et al. [20] that meets the bounds on the optimal range of Byzantine processes and communication rounds. Further, we make modifications to this algorithm by removing the redundant information being

[1] $\tilde{O}$ is same as O up to a poly-logarithmic factor.

sent in every round regarding the list of byzantine nodes to show an improvement in the communication complexity. We call this algorithm *ImprovedEIG*.

We vary two parameters to define the workload of the system—the number of processes, and with that the fraction of faults in the system. For evaluation, we use three metrics: communication complexity and round complexity, which are commonly used for theoretical evaluation, and latency, which is an important measure of efficiency for empirical results. We report on the behavior of the algorithms under varying parameters and analyze their effectiveness using these metrics. These experimental results allow us to determine the best algorithms to pick given the workload and requirements of a system, i.e., the fraction of adversarial nodes, the bandwidth requirements and latency. For example, in certain scenarios with low adversarial nodes, the deterministic algorithm performs better than both the other ones.

Section II gives an introduction to the various models used to solve the distributed consensus problem and provides a summary of related work in this field. In Section III, we elaborate on the three algorithms under comparison. Section IV and V details the implementation and evaluates the results obtained from our experimental evaluation. This is followed by a discussion that includes implementation techniques for better performance, and Section VI concludes the paper.

II. Background

A. Problem Statement

The problem of Byzantine Agreement, in its most basic form is defined as follows [43]:

Definition 1. *Let $\mathcal{P}$ be a protocol among n processes $P = \{P_1, P_2, \ldots, P_n\}$, such that $B \subset P$ processes are byzantine. Each process P_i starts with an input bit b_i, and P_i outputs a bit c_i at the end of the protocol. $\mathcal{P}$ is a Byzantine Agreement Protocol, if the following conditions hold:*

- *Consistency: For any two non-faulty processes P_i and $P_j \in P \backslash B$, $c_i = c_j$.*
- *Validity: If $b_i = b$ for all non-faulty processes $P_i \in P \backslash B$, then $c_i = b$ for all non-faulty processes P_i.*
- *Termination: Protocol $\mathcal{P}$ terminates with probability* 1.

A protocol is said to be k-fault tolerant if it operates correctly as long as no more than k processes fail during execution. The following theorem by [32], [43] shows the impossibility result when $k \geq n/3$.

Theorem 1. *There is a k-fault tolerant synchronous protocol to solve the byzantine agreement problem iff $k < n/3$.* ■

B. Complexity Measures

The practicality of agreement protocols depends heavily on their computational complexity. Theoretically, when talking about complexity measures of algorithms for distributed consensus, one generally uses the following two metrics:

- *Round* complexity - the number of rounds of message exchange before all the non-faulty processes decide.
- *Communication* complexity - the total number of messages sent per process or the total number of bits sent per process.

For empirical analysis, we also consider the following metric:

- *Latency* - the overall CPU time utilization or elapsed real time from start till all the non-faulty processes decide.

C. Previous Work

1) Deterministic Solutions: Fischer and Lynch [18] proved that $k + 1$ is the round complexity in the worst case for a k-fault tolerant synchronous protocol. If the messages were not *authenticated*, the message complexity was initially shown to be exponential in the number of process by Pease et al. [43]. In 1998, Garay and Moses [20], with modifications to the two phase protocol of Bar-Noy et al. [5] using the EIG data structure, improved the message complexity further to polynomial time. If authenticated messages were sent, Dolev and Reischuk [13], proposed an algorithm using $O(n + k^2)$ messages. In an attempt to lower the communication costs, researchers either lowered the fraction of faulty processes to a smaller number [14] or increased the maximum number of rounds needed in the worst case [46]. It was only recently that Kowalski et al. [30] proposed a simple algorithm that holds for the optimal range and optimal number of communication rounds while lowering the communication complexity to $O(n^3 log n)$.

2) Randomized Solutions: Probabilistic solutions were proposed to circumvent the lower bounds on round and message complexity imposed by deterministic settings. They used the idea of a common coin which was seen as 'sufficiently random' by 'sufficiently many' random processes. In the asynchronous setting, using randomized algorithms, Ben-Or [6] showed that if $k < n/5$, then consensus is achievable with probability 1. Rabin [44] showed constant expected round complexity if $k < n/4$. Greatly improved results have been shown in [41], [24], [37] for non-adaptive adversary and in [27], [3] for an adaptive adversary. Assuming that communication channels are *private* between every pair of processes, the algorithm proposed by Patra et al. [42] shows constant expected round complexity and $\tilde{O}(n^2)$ message complexity. These bounds are also applicable to the asynchronous setting. Holtby et al. [22] proved an $\Omega(\sqrt[3]{n})$ lower bound on both message complexity and round complexity for synchronous systems, under restrictive assumptions.

3) Almost-Everywhere Solutions: The almost-everywhere byzantine agreement problem was introduced by Dwork et al. [15]. It is a relaxed version of the byzantine agreement problem and requires all but $O(log^{-1} n)$ fraction of the processes to agree on a common output. For the algorithm by King et al. [28], the round and message complexities are shown to be poly-logarithmic in n. Construction of byzantine agreement from almost-everywhere byzantine agreement, called almost-everywhere reduction, was proposed in [26], [25], using $\tilde{O}(\sqrt{n})$ bits per process and poly-logarithmic number of rounds. Papers such as those by King et al. [25] used push-pull protocols, the complexity of which is dictated by the complexity of the first *push* phase and the size of the

candidate lists, i.e., the number of all possible outputs. Braud-Santoni et al. [9] propose an almost-everywhere to everywhere solution using the almost-everywhere algorithm by King et al. [28].

4) Experimental Evaluations: Many surveys have reported various theoretical results for the Byzantine Agreement problem. In a recent paper by Vavala et al. [47] that implements Bracha's algorithm [8] to bridge the gap between theory and practice, it was reported that the literature is poor in the experimental evaluations of randomized byzantine agreement algorithms. They showed that Bracha's algorithm terminates in constant rounds if only crash failures occur and under normal conditions, whereas theoretically it takes exponential number of rounds to terminate due to the worst-case scenario. They use an averaging method, approximations and stochastic techniques for analysis of the protocol. They ran the experiments for up to 100 processes and reported the round complexity results.

Oluwasanmi et al. [39] improve upon the algorithm by King et al. [26] that was shown to be impractical when implemented due to large hidden constants, although they weaken the control of the adversary to only $1/8$ fraction of the processes. They implement and compare their algorithm with Cachin et al.'s [10] with the size of the network simulated between 10^3 to 4×10^6 processors. They used average number of messages and bits sent per process as well as latency for comparison. Moniz et al. [35] perform experimental evaluations on Bracha's algorithm [8] and Cachin et al.'s [10]. However, significantly better results to both Bracha's and Cachin et al.'s algorithms have been shown by Vavala et al. [47] and Oluwasanmi et al. [39], respectively. Liang et al. [33] implemented and analyzed three different byzantine broadcast algorithms for fault-tolerant state machine protocols (1) the classic solution by Pease et al. [43], (2) a practical BFT protocol by Castro and Liskov [11], and (3) a network coding based BFT that they propose in the paper.

For state replication protocols, it is important for processes to agree upon an order to process the requests. In this experimental evaluation, the authors concentrated on the implementation and analysis of the byzantine broadcast part of the algorithm that is used to reach consensus on the order of requests to be processed by the state machines. They reported the latency when the batch size of the requests to 4 servers is varied. Several other experimental evaluations for state machine replication methods use byzantine agreement protocols but mostly in the asynchronous settings and some protocols even require a trusted subsystem [2], [12], [48], [23], [29]. Other works such as those by Moniz et al. [34], [36] consider wireless and asynchronous settings, respectively, which is outside the scope of this paper.

Most of these experimental evaluations have used either a synchronous model or a partially synchronous model for simplicity. The algorithms we have chosen allow us to differentiate and compare different randomized as well as deterministic algorithms which makes it necessary to use a synchronous setting due to the impossibility results for the asynchronous case [19].

III. Algorithms

In this section, we describe the model of the network, adversarial conditions and the algorithms of [7], [9] and [30], i.e., algorithm *Quorum*, *Pull-Push* and *EIG*, respectively, that are to be evaluated under these conditions. Table 1 compares key properties of these algorithms. Note that the algorithms, theorems and protocols presented in this paper have been reproduced or condensed from works by Ben-Or et al. [7], Braud-Santoni et al. [9] and Kowalski et al. [30].

A. Model

Our model is a *fully-connected* network of n processes, with authenticated communication channels—the identity of the sender is known to the recipient and authentication is not required during the execution of the protocol. We require the network to be *reliable*, i.e., a message sent (to a non-faulty process) will eventually be delivered and in order. Per Section II, we consider only the *synchronous* model of communication. In this setting, the communication proceeds in rounds and all processes have synchronized clocks; a process moves on to the next round after all the processes have completed the previous round.

B. Adversary

The adversary controls a fraction of the processes—a maximum of k processes, which are *Byzantine* processes. Such processes deviate arbitrarily from the algorithm by crash failures or sending false messages. The assumption is that the adversary is *non-adaptive* for all three algorithms, that is, the adversary chooses the set of byzantine processes at the start of the protocol. The adversary is computationally unbounded and has *full information* about the state of all processes and the network, and the communication between any pair of them. Another characteristic is that the adversary is *rushing*. This means that, in a round, the adversary knows all the messages sent by the good processes in all the previous rounds and the current round before choosing which messages to send in that round. This is also known as the *full information* model [21].

C. Input String

Both the algorithms *Quorum* and *Pull-Push* make use of a string of bits in the final decision. However, algorithm *EIG* uses a single bit as input and output decision value.

D. Algorithms

1) Algorithm Quorum[7]: The main result of this algorithm, due to Ben-Or et al., is as follows:

Theorem 2. *For any constant $\epsilon > 0$, there exists a protocol that reaches Byzantine Agreement in a synchronous full-information network tolerating $k < (1/4-\epsilon)n$ non-adaptive Byzantine faults, and runs for expected $O(\log n/\epsilon^2)$ rounds.* ■

The algorithm makes use of a weaker version of broadcast known as *Graded Broadcast* or *Gradecast* [17], with a designated authority called the 'dealer' that wants to broadcast a value v. At the end of the protocol, every process outputs a

TABLE I
A SUMMARY OF FEATURES OF THE ALGORITHMS UNDER EVALUATION

Algorithm	Type	n	Rounds	Bit Complexity	Decision value	Communicating nodes	Remarks
Ben-Or, Pavlov, Vaikuntanathan [7] (*Quorum*)	Randomized	$4k+1$	$O(\log n)$	$n^{O(\log n)}$	String of $O(\log n)$ bits	All-to-all communication and within quorums of size $O(\log n)$	Everywhere byzantine agreement
Braud-Santoni et al. [9] (*Pull-Push*)	Randomized	$3k+1$	$O(\frac{\log n}{\log \log n})$	$\tilde{o}(n)$	String of $O(\log n)$ bits	With samplers of size $O(\log n)$	Almost-everywhere to everywhere
Kowalski and Mostefaoui [30] (*EIG*)	Deterministic	$3k+1$	$k+1$	$O(n^3 \log n)$	Single bit	All-to-all communication	Uses EIG data structure

value and a number that denotes the confidence of the process in that value. If the dealer is honest, every honest process outputs the same value with full confidence.

The quorum protocol proceeds in stages. It makes use of Feige's protocol [16] for collective coin-flipping which works as follows: in the first round, all the processes throw a ball at random into one of $O(n/\log\ n)$ bins. The processes which throw their ball into the *lightest bin* (bin with least number of balls) survive. The protocol is invoked recursively on the $O(\log\ n)$ processes in the lightest bin. Agreement on the lightest bin is achieved by running *sub-protocols* among every subset of processes of size $\frac{3}{4}\log n$. These sub-protocols can be executed in parallel since the decision of one does not affect the decision of the other.

The key idea is that honest processes are unbiased and the resulting bin will contain a large fraction of honest processes. After $\log^* n$ invocations of the process, a leader is elected. The leader then flips a coin, and broadcasts it, which is the agreed value. The challenge is that dishonest processes will exhibit byzantine behavior when throwing their ball into one of the bins by sending conflicting values to different processes. This is overcome by using the *gradecast* protocol described earlier.

2) Algorithm Pull-Push [9]: This algorithm by Braud-Santoni et al. [9] is the first probabilistic Byzantine Agreement algorithm whose communication and round complexities are poly-logarithmic. The authors use the almost-everywhere algorithm by King et al. [28] and extend the protocol to Byzantine agreement in the complete network. This protocol uses the *pull-push* communication model. The following theorem from [9] states the main result:

Theorem 3. *For n processes in an asynchronous full-information message passing model with a non-adaptive Byzantine adversary which controls less than a $1/3-\epsilon$ fraction of the processes, if more than $3/4$ of the processes know a string $g_{\texttt{string}}$ (random enough), there is an algorithm such that with high probability:*

- *At the end of the algorithm, each correct process knows $g_{\texttt{string}}$.*
- *The algorithm takes $O(\frac{\log n}{\log\ \log n})$ rounds and $\tilde{O}(n)$ messages are exchanged in total.* ■

The algorithm makes use of *quorums* to filter requests or messages sent by other processes. The choice of quorums used by processes is directed by both deterministically-known information (like the identity of the process), and random sources (randomly chosen initial string). Such quorums are called *Samplers*.

Each process starts with a candidate string (the string to be agreed upon). The assumption is that more than half of the processes are both correct and have the same candidate string. The algorithm proceeds in two phases. In each of the phases, messages are sent to or received from only selectively chosen processes by sampler functions.

Push Phase: In the first phase, each process starts to *diffuse* (send) its candidate string $g_{\texttt{string}}$. A push occurs when a process receives information about their candidate string from other processes without asking for it. To each pair of string s and process x, the push quorum $I(x,s)$ assigns a set of $O(\log\ n)$ processes. x may receive pushes for s from processes in $I(x,s)$ only. If more than half of the processes in $I(x,s)$ push for s, s is added to x's candidate list L_x. We refer the reader to Algorithm 1 in the Appendix for details.

Pull Phase: In the second phase, called the pull phase, the bogus strings are discarded so that each process keeps only the correct string. This is done by each process requesting the strings it received in the push phase to be verified by some other processes. A *pull* query is sent out to receive information about each string as a consequence. Checking a string s involves a Poll List $J(x,r_x)$ and a Pull Quorum $H(x,s)$, where r_x is chosen at random. Algorithms 2, 3 and 4 in the Appendix give a detailed implementation of the sending, routing and answering of pull requests.

3) Algorithm EIG [30]: This protocol, similar to the classic protocol by Bar-Noy et al. [4], has two phases. In the first phase, each process communicates with every other process for $k+1$ rounds and stores the collected information in each round at a corresponding level in a tree-like data structure. In the second phase, a bottom-up evaluation is done on each of the trees at each process. The fundamental difference between this protocol and past EIG solutions is that after a couple of rounds, instead of storing and sending proposed values at each level of the tree, only an array of suspected byzantine processes is sent. This array is updated after each round using the *confirmation mechanism* that works as follows:

- A process p_i sends the *main information* to every other process in round r.
- At round $r+1$, process p_j *echoes* information received from p_i in the previous round to every other process.
- If the main information received from p_i in round r is echoed by at least $(n-k)$ processes in round $r+1$, then process p_j is said to *confirm* the information received from p_i in round r.

The rounds 1 to $k+1$ follow this protocol:

Round 1 Each process sends its proposed value to all other processes. `val(x)` is set to the received value v from process x where `val` is a variable of nodes at level 1 of the EIG tree.

Round 2 Each process echoes the messages received in the first round. The received messages are set as the value of variable `cval` of nodes at level 2. For a node $x = jk$, $\texttt{cval}(\texttt{x}) = v$ where v is the value that process p_k reports that it received from p_j in the previous round. The confirmation mechanism is applied to the received echoed messages by each process p_i. The process whose main information is not confirmed in this round is added to p_i's byzantine list.

Round 3 The echo messages received in round 2 are sent as echo messages again. The suspected byzantine list is sent as the main information. Confirmation mechanism is applied to the echoed information and the byzantine list is updated in each round from here on.

Round r ($4 \leq r \leq k+1$) From the 4th round onwards, the suspected byzantine list of a process is sent as its main information and byzantine lists received in the previous round are echoed. Nodes at corresponding levels of the EIG tree from round 3 onwards, for each process p_i, are given values for `val` and `cval` as follows:

- For node $x = ykl$, $\texttt{val}(\texttt{x}) = \top$, if p_l never reported to p_i by round k that it suspects p_k, else $\texttt{val}(\texttt{x}) = \perp$.
- For node $x = yjkl$, $\texttt{cval}(\texttt{x}) = \top$, if p_l never reported to p_i by round $k+1$ that p_k suspects p_j, else $\texttt{cval}(\texttt{x}) = \perp$.
- For leaf nodes $x = ykl$, $\texttt{val}(\texttt{x}) = \top$, if p_l did not report to p_i that it suspects p_k, else $\texttt{val}(\texttt{x}) = \perp$.

where y is a string of ids (possibly empty), and j, k, l are ids of three different processes.

Importantly, note here that in every round even though the number of nodes in the EIG tree increases by factor n, the new information received remains quadratic which allows us to use arrays to store this information. Even the size of the echoed messages remains quadratic in every round.

Extracting the final information:
Starting from the leaves, the nodes of the EIG tree are evaluated bottom-up as follows:

- if x is a leaf, $\texttt{newval}(x) \leftarrow \texttt{val}(x)$.
- if x is root, $\texttt{newval}(x) \leftarrow v$ such that strict majority of new values of its children are set to v, otherwise it is set to default value v_0.
- otherwise, $\texttt{newval}(x) \leftarrow v$ if for $T = \{y \mid y \text{ child of } x \wedge \texttt{newval}(y) = \top\}$, $\mid T \mid \geq (n-t-l)$, and a strict majority of nodes in T have `cval` set to v.

4) Algorithm ImprovedEIG: We modify the *EIG* protocol slightly to require that instead of sending the complete byzantine list every time, from round 4 onwards only the changes to this list be sent in every round. We are motivated to do so since sending information about the existence of a process in the suspected byzantine list of another process in every round is redundant. This does not change the correctness of the algorithm since all the good processes send the same changes to every other process in a round and in the next round the confirmation mechanism would confirm these updated lists. The rest of the algorithm is identical to algorithm *EIG*. This means that the number of rounds required to reach consensus remains unchanged as the information every process has about the suspected byzantine list of another process at the end of a round is the same as in algorithm *EIG*. This implies that the round complexity is unchanged, and therefore the latency of the algorithm is unaffected.

IV. Implementation

The implementation of the three algorithms to obtain performance results was done using the C++ programming language along with the Message Passing Interface standard (MPI). The experiments were run on a 16 node cluster of 64-bit Xeon machines using 4 cores on each node. Each machine in the cluster runs the Linux operating system CentOS $6.x$. The machines are connected to each other through a 10Gbps local area network. The complete implementation is approximately 2K lines of code and can be found at https://github.com/shreya-68/Consensus/tree/master/MPI. A basic synchronous peer-to-peer network framework was set up on top of which the algorithms were implemented.

A. Input

The input into the system is size of the network, i.e., total number of processes, along with the input binary value chosen from a uniform distribution for each of the processes for achieving consensus. The ratio of the byzantine processes to the good processes is provided along with a specified byzantine behavior (see Section IV-H). This behavior allows the byzantine processes to decide which values to send at every round of the protocol.

B. Output

The output is the number of rounds carried out, latency in terms of CPU time utilization and elapsed real time, number of messages/bits sent per process, and the final decision value of each process.

C. Testing Parameters

For testing the three algorithms, the number of processes (n) lies in the range 4 to 64 on a cluster of 1 to 16 machines. The number of failures (f) lies in the range $[0, n/3)$ for algorithms *Pull-Push* and *EIG*, and in the range $[0, n/4)$ for algorithm *Quorum*. The algorithms have been shown to be correct only in these respective ranges [9], [30], [7].

D. Authentication of Processes

It was assumed that the channels are authenticated and each receiving entity knows the identity of the sender. The communication primitives provided by the MPI library implicitly provide this functionality and hence, a byzantine process cannot spoof their identity or hide it.

E. Connection

Each process in the peer-to-peer network had full knowledge about the network connection of other processes, i.e., the IP address and port number of the hosts it was connected to. Using the MPI library this is done by giving a unique ID to each process from $[0, n)$. The MPI point-to-point operations were used for sending and receiving messages between (any)

two processes. The connection provides the same reliability and ordering guarantees as a TCP connection.

F. Synchronization

To implement a synchronous model of communication communicators have been used. They are essentially a group of processes. `MPI_Barrier`, a synchronization operation provided by the MPI library, allows one to specify a group to wait on. A process blocks on this call until all processes in the specified group reach this call. To keep all the processes in the network synchronized after each phase, this routine is used whose group is set to the active processes in the network.

G. Parallelization

To allow programs to execute parts of the protocol that are independent of each other in parallel, MPI thread level support has been used. To further improve performance, non-blocking calls have been used in some places where, for example, the execution of the next few instructions does not depend on the completion of a send call. The MPI implementation creates a system buffer to typically hold data in transit. This is useful when a process wants to receive messages sent to it by multiple other processes at a later time. This improves performance by allowing send-receive operations to be asynchronous. The non-blocking calls were used such that execution within a round became asynchronous but not between rounds so as to ensure correctness. To ensure that messages are not lost by the overflow of a system buffer, each message was given a unique tag number which corresponded to a unique system buffer.

H. Types of Byzantine Failures

A process can display either of the following failures:
Active failures:

- Processes send conflicting data to different processes in the same round. Any data that is not the same or is null is said to be conflicting.
- Processes send arbitrary content or more messages than necessary. Byzantine processes may try to do so to choke the communication and flood the network or overload the requests that need to be answered.

For analysis of active failures, we inject both of the above mentioned failures, i.e., processes send conflicting data as well as attempt to flood the network.
Inactive failures:

- Crash failure: Processes may fail by stopping. Any good process that fails to send messages or aborts its execution is considered byzantine. This is a worst-case assumption to ensure correctness even if for any unknown reason a good process fails.
- Denial of service: Byzantine processes may deny responding to requests from good processes or may not forward messages as required.

For analysis of inactive failures, crashed processes remain silent and stop sending messages to emulate node crash.

I. Fault Tolerance Against Crash Failures

We assume that once a process suffers a crash failure, it cannot recover from it. An advantage of using TCP connections is that it allowed failure discovery in the case of a crash failure. If a process failed before the start of a phase, no other process would be able to establish a connection with it, leading to failure discovery. If a process crashed within a phase, the wait groups helped in detecting the failure by using a timeout mechanism. Each phase had to be completed within a certain amount of time and if all the processes did not reach the synchronization operation within that allotted time, it indicated a process failure and the other processes proceeded with the next phase. By running the system multiple times we could determine a timeout value that allowed alive processes to finish the phase with high probability. An alternative to this method was sending an acknowledgement back for each received message, but we refrained from using this method since that meant dealing with twice the number of messages which added unnecessary overhead for our testing purposes.

J. Message Format

Each message sent over the network was an array of bytes of the following form: `LENGTH`, `PHASE`, `MESSAGE`

The `LENGTH` parameter indicated how many bytes of the message to parse. The `PHASE` parameter indicated which phase the message was sent during and also what its purpose in that phase is. The format of the `MESSAGE` varied for each of the algorithms.

V. Results

A. Bit Complexity

For the *EIG* algorithm, we can see from Figure 1 that as the network size increases, the bit complexity increases as a cubic function in accordance with the theoretical Big 'O' complexity of $O(n^3 \log n)$, where n is the size of the network. This is a major improvement from classic deterministic algorithms that have very high polynomial growth of $O(n^9)$ [20]. On a linear scale, we note that the fault ratio affects the bit complexity by a factor polynomial in n. This is because with a higher ratio, faulty processes try to dominate the number of bits a good process receives by claiming all processes are byzantine in every round. They send n^2 bits of information in every round to every other process, whereas good processes send only $n*k$ bits of information. As the number of processes increases, the trends for each of the fault ratios becomes similar on the $\log$ scale. However, for small networks, the bit complexity on the $\log$ scale is quite diverse for different fault ratios. Overall, we observe that increasing the fault ratio will affect the bandwidth consumption linearly for all network sizes.

For the *Pull-Push* algorithm, as the network size increases the bit complexity displays a poly-logarithmic growth. This can be seen in Figure 2. The growth trend is similar for all fault ratios for small and larger networks. The increase in number of failures has an effect on the number of bits sent per process but only by a constant factor. This can be attributed to the fact that in the protocol, even if byzantine processes try to send

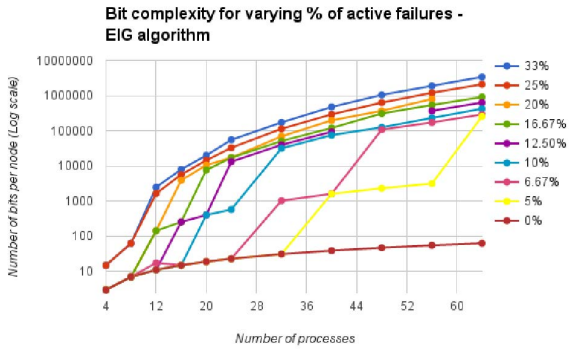

Fig. 1. Performance results for EIG algorithm (Log Scale)

conflicting values, and a large number of them, to samplers for the purpose of flooding, the samplers do not receive enough of them to forward these messages further in the protocol. Also note that for networks with the same value of $\log n$ the bits sent per process is the same, and it only changes at network sizes that are powers of 2. This is because good processes communicate only with samplers of size $\log n$. Hence, for larger networks, increasing the size of the network by less than 100% will not affect the bandwidth consumption.

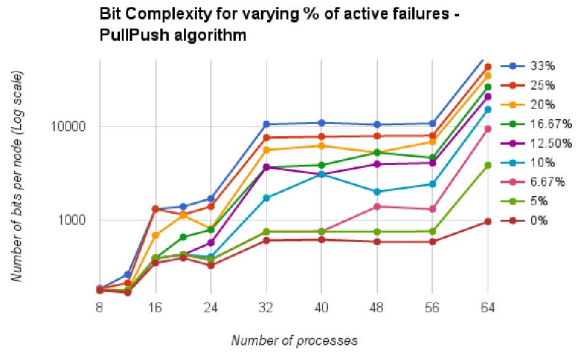

Fig. 2. Performance results for Pull-Push algorithm (Log scale)

From Figure 3, algorithm *Quorum* shows very high number of bits per process communication even for small networks. It increases rapidly as the network size increases. The reason for this is that *Graded broadcast* requires all-to-all communication between processes and the protocol requires processes to run a deterministic byzantine agreement protocol for every subset of processes with their ball in the same bin in Stage 2 of the protocol. The fault ratios do not have much of an effect on the communication costs. Byzantine processes are unable to increase bits on the network by participating in more sub-protocols than the algorithm requires since membership in a sub-protocol is dictated by their ID, which is fixed. Hence, increasing the number of byzantine processes does not influence the (already high) communication cost greatly.

Each of our algorithms for every configuration was run multiple times. In the analysis, each data point is an average over 5 independent runs and we obtained confidence intervals for each of them. The confidence interval for algorithm *Pull-Push* was $\pm 1\%$, $\pm 0.1\%$ for algorithm *EIG*, and $\pm 0.5\%$ for

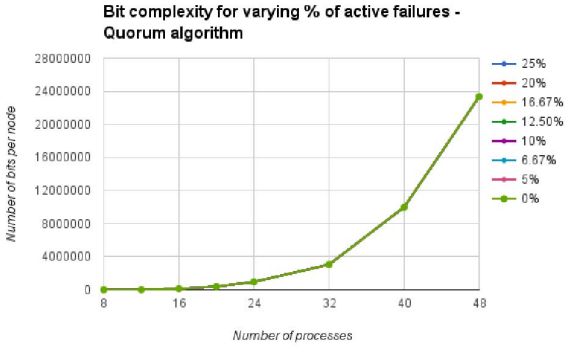

Fig. 3. Performance results for Quorum algorithm (Log scale)

algorithm *Quorum*.

B. Comparison of Bit Complexity

For large networks ($n = 64$), we vary the ratio of failures to the number of processes (f/n), which lies in the range $[0, 1/3)$ for algorithms *EIG* and *Pull-Push*, and in the range $[0, 1/4)$ for algorithm *Quorum*. As can be seen from Figure 4, the algorithm *Pull-Push* performs much better than the other algorithms for any fault percentage.

Next, we evaluate the modified algorithm *ImprovedEIG* introduced in Section III-D4. Even though this does not change the communication complexity in the worst case, it overall reduces the communication bits as good processes would send a bit for every suspected byzantine process only in any one of the rounds instead of every round. This modified version of the algorithm performs much better. For small ratios, since the number of rounds is small, the number of bits sent per process remains the same. But, as the ratio increases, the performance varies greatly from that of algorithm *EIG*.

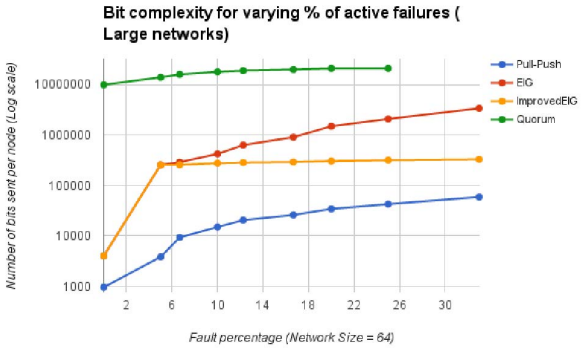

Fig. 4. Comparison for large networks

For small networks ($n < 40$), if the fault ratio is small and in the range $[0, 1/15)$, Figure 5 shows that algorithm *EIG* performs much better than any of the other algorithms. This is because only the first three rounds of this algorithm will be executed and since the ratio is small, a simple broadcast is sufficient to gather all the information. *Pull-Push* and *Quorum*, which are more complex algorithms, perform worse in such scenarios. Figure 6 shows that for higher fault ratios and any network size, algorithm *Pull-Push* performs better.

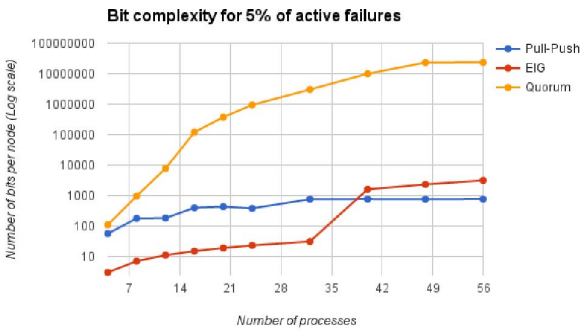

Fig. 5. Comparison for low % of failures

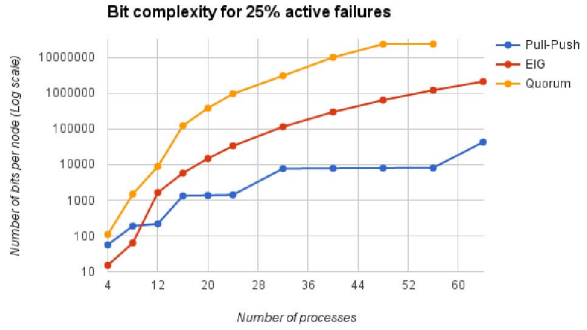

Fig. 6. Comparison for high % of failures

Depending on the system requirements, such as how fast we want the information to reach the destination, bandwidth, and so on, there were two mechanisms we used to send the messages. In an algorithm like *Pull-Push*, where multiple message packets may be sent to the same process by a process in the same round, one could marshal all the packets into one packet and then send it. This decreases the number of messages sent per process while the number of bits sent remains the same. However, it comes at the cost of parallelization as one has to first wait to receive all the messages from the previous round, perform operations and then send out messages all at once.

C. Round Complexity

Round complexity is the number of phases that have to be executed sequentially by a process and cannot be parallelized. For example, in algorithm *EIG*, a round consists of the broadcast of messages at the same level i in the *EIG* tree of each process. For algorithm *Quorum*, a round consists of the broadcast of messages during a particular sub-protocol. There are two ways to analyze the round complexity of this algorithm. The first is that each of the sub-protocols S_i^j could be executed in parallel since none of these sub-protocols are dependent on each other. Hence, all of them together make one round. But, if parallel execution of these sub-protocols is not possible due to resource constraints then each of them is an independent round. We implemented the algorithm to run the sub-protocols in parallel since that improved the round complexity. For algorithm *Pull-Push*, the push phase makes one round and sending, routing and answering in the pull phase each form independent rounds.

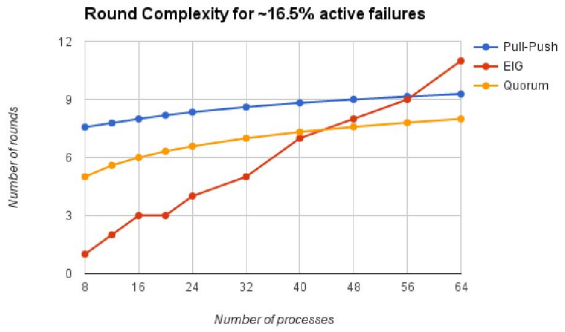

Fig. 7. Round Complexity comparison

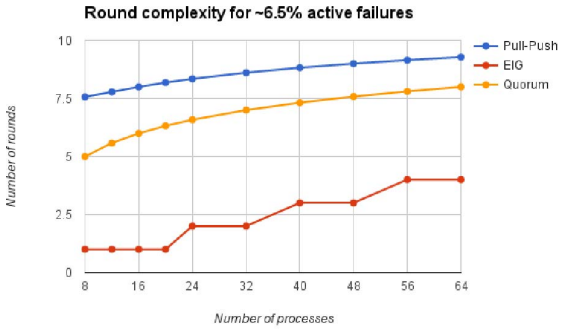

Fig. 8. Round Complexity comparison

A comparison of the round complexities can be seen in Figs. 7 and 8 for fault percentage $\sim 16.5\%$ and $\sim 6.5\%$, respectively. We can see that for the small ratio of $\sim 6.5\%$ the round complexity of *EIG* is best compared to *Quorum* as well as *Pull-Push* for all network sizes. However, for larger ratios *EIG* performs the best for small network sizes among the three algorithms. It performs worse than the other two algorithms for larger networks while *Quorum* performs the best here. For the fault percentage of $\sim 16.5\%$ we can see that for larger networks $(n > 40)$, *Quorum* starts performing better from among the three algorithms.

D. Latency

For performance comparison of latency, we compare the total CPU times and elapsed real time. The total CPU time is the sum of CPU time consumed by all of the CPUs utilized by an execution of an algorithm. If a program has parallel tasks, the total CPU time takes into account the time taken by each of the tasks. Elapsed real time is simply the time taken from the start of a computer program until it terminates as measured by an ordinary clock. From Figures 9 and 10, with increasing network size, we see that CPU time utilization and elapsed real time of *EIG* increases rapidly, and it is significantly more compared to the other two algorithms. If we look at the elapsed real time, we can see that algorithm *Quorum* remains the fastest throughout. This is because *Quorum* is highly parallel.

The trade-off here is that the CPU time utilization of *Quorum* is a lot more than that of *Pull-Push*. This trend is maintained for all fault ratios.

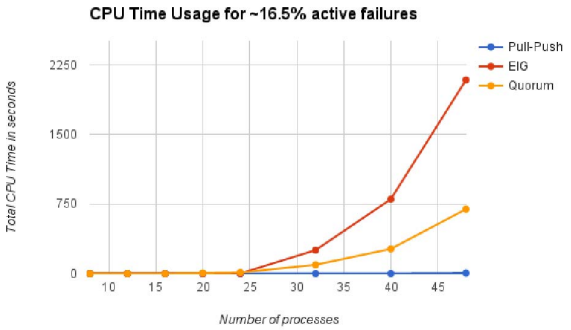

Fig. 9. CPU time utilization comparison

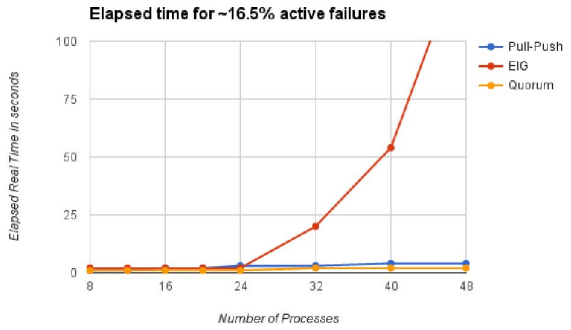

Fig. 10. Elapsed real time comparison

E. Discussion

As can be seen from the results above, depending on the requirements and resources available, each algorithm performs differently. Before implementation, it is crucial to consider what resources are available and how a system can be designed to obtain optimal results for those available resources. Implementing the testbed framework on a cluster allowed us to design, study and test the algorithms under realistic scenarios. We considered different byzantine behaviors for each metric analysed. To report on the number of bits sent per process in the worst case, we considered byzantine processes that try to flood the network. In this scenario, among the three algorithms we considered, algorithm *Pull-Push* performs the best generally. However, in the case of a small network, with the number of processes less than 32 and a low fault percentage—(approximately less than 10%), algorithm *EIG* performs better since it considers the number of faults in its protocol.

If we were to consider a network with its size showing high variance over time, algorithm *Pull-Push* has the advantage that the number of bits sent per node remains the same if the increase in size is within the next power of 2. For networks with strict bandwidth constraints, this allows high flexibility in changing the size of the network. On the other hand, if we were to look at networks which did not change in size but had varying fault ratios, algorithm *Quorum* inhibited a varying number of byzantine processes from changing the communication complexity. This, however, is restricted to networks with high bandwidth in the first place. Our modified version of algorithm *EIG*, i.e. *ImprovedEIG* showed this characteristic as well. The number of bits sent per node increased very minimally on increasing the fault ratio and also performed much better when compared to *Quorum* or *EIG* as can be seen in Figure 4.

Furthermore, we demonstrated a trade-off between the number of communication bits and the number of rounds: algorithm *Quorum* terminated in lower number of rounds than *Pull-Push* for all network sizes and fault ratios. This can be attributed to its highly parallel protocol. Algorithm *EIG* displayed growth proportional to the size of the network, and performed well only for very low fault ratios of $< 10\%$. This trend is reflected in the latency results as well when we consider the elapsed real time. However, the total CPU time utilization increased rapidly for *Quorum* due to the exponential increase in the number of sub-protocols executed in parallel as the network size increased.

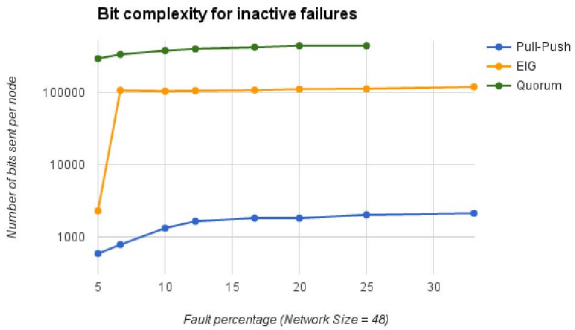

Fig. 11. Comparison for crash failures

In the case of a denial of service attack, an increasing ratio of faulty processes reduced the communication overhead instead of increasing it. Similar results were also observed for crash failures. To evaluate the effect of crashes, we tested the system such that crash failures occurred on a uniform distribution over time. As can be seen in Fig. 11, an increasing fault ratio keeps the number of bits sent almost the same. When the faulty processes sent out arbitrary messages only, the algorithms executed correctly without hampering the communication cost much even if the ratio of byzantine processes was increased.

To further optimize the performance, one can improve certain tasks such as sending a larger message instead of many smaller messages or executing parallel tasks on multi-core machines. Another optimization that would have improved the running time of the system would be to use UDP instead of TCP since they take up less resources and provide lower overhead. Even though UDP does not guarantee message delivery, for a small system it could still be considered highly reliable. The rationale behind using TCP connections for our

testbed framework was to model and understand implementation issues for more realistic distributed systems, which could have peers separated by geographical distance, requiring greater reliability.

An analysis of the three algorithms and their performance for a wide range of number of processes and faults shows that communication of each process with fewer number of processes yields good results instead of all-to-all communication. This inhibits byzantine processes from influencing values of too many good processes. It is also important that requests from byzantine processes be throttled at an early stage. The good performance of the deterministic algorithm for small fault ratios shows that it is important to consider this factor when designing an algorithm. Communication between multiple sets of quorums allows parallel tasks to be executed and gives good latency results. The combined use of these techniques would help design improved algorithms to solve the problem of distributed consensus.

VI. Conclusion

In real-world systems, achieving distributed consensus can be critically important. Consensus algorithms are used frequently in systems that rely on protocols such as those used in state machine replication and distributed databases. Thus, understanding the performance of algorithms for various scenarios occurring in real-time is essential to the overall performance of such systems. We show that there is a need to consider implementation issues that come along with any of these algorithms and not only their theoretical results.

In this paper, we focused on implementation and analysis of three recently proposed algorithms with best results for their respective agendas. An Exponential Information Gathering protocol for consensus [30] (algorithm *EIG*) showed that deterministic algorithms have come a long way since the early results. We further improved upon the results obtained for this algorithm. In general, the randomized algorithm *Pull-Push* [9], performed better than the other two in terms of communication complexity. When latency was considered, as can be seen in Fig. 10, algorithm *Quorum* [7] performed better. In real-time situations, as the number of processes in a network increase the probability of having faulty processes naturally increases. Hence, even though algorithm *EIG* shows better performance when the fault ratio is small, under high fault ratio its performance degrades. Quantifying the performance of the algorithms empirically provides a practical understanding of how the different algorithms perform under different conditions to achieve consensus in distributed systems.

References

[1] http://lwn.net/Articles/540368/.

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, pages 59–74, 2005.

[3] D. Alistarh, J. Aspnes, V. King, and J. Saia. Communication-efficient randomized consensus. In *DISC*, 2014.

[4] A. Bar-Noy and D. Dolev. Consensus algorithms with one-bit messages. *Distributed Computing*, 4, 1991.

[5] A. Bar-Noy, D. Dolev, C. Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite byzantine agreement. In *PODC*, 1987.

[6] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. PODC '83, 1983.

[7] M. Ben-Or, E. Pavlov, and V. Vaikuntanathan. Byzantine agreement in the full-information model in o(log n) rounds. In *STOC*, 2006.

[8] G. Bracha. An asynchronous (n-1)/3-resilient consensus protocol. In *PODC*, 1984.

[9] N. Braud-Santoni, R. Guerraoui, and F. Huc. Fast byzantine agreement. In *PODC*, 2013.

[10] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology*, 18, 2005.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *TOCS*, 20, 2002.

[12] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, November 2006.

[13] D. Dolev and R. Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32, 1985.

[14] D. Dolev, R. Reischuk, and H. Raymond Strong. Early stopping in byzantine agreement. *J. ACM*, 37, 1990.

[15] C. Dwork, D. Peleg, N. Pippenger, and E. Upfal. Fault tolerance in networks of bounded degree. *SIAM J. Comput.*, 17, 1988.

[16] U. Feige. Noncryptographic selection protocols. In *FOCS*, 1999.

[17] P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26, 1997.

[18] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14, 1981.

[19] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *PODS*, 1983.

[20] J. A. Garay and Y. Moses. Fully polynomial byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM J. Comput.*, 27, 1998.

[21] O. Goldreich, S. Goldwasser, and N. Linial. Fault-tolerant computation in the full information model. *SIAM J. Comput.*, 27, 1998.

[22] D. Holtby, B. M. Kapron, and V. King. Lower bound for scalable byzantine agreement. *Distributed Computing*, 21, 2008.

[23] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[24] B. M. Kapron, D. Kempe, V. King, J. Saia, and V. Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. In *SODA*, 2008.

[25] V. King, S. Lonargan, J. Saia, and A. Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. In *ICDCN*, 2011.

[26] V. King and J. Saia. From almost everywhere to everywhere: Byzantine agreement with $\tilde{o}(n^{3/2})$ bits. In *DISC*, 2009.

[27] V. King and J. Saia. Byzantine agreement in polynomial expected time: [extended abstract]. In *STOC*, 2013.

[28] V. King, J. Saia, V. Sanwalani, and E. Vee. Towards secure and scalable computation in peer-to-peer networks. In *FOCS*, 2006.

[29] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[30] D. R. Kowalski and A. Mostéfaoui. Synchronous byzantine agreement with nearly a cubic number of communication bits. In *PODC '13*, 2013.

[31] J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications.

[32] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4, 1982.

[33] G. Liang, B. Sommer, and N. H. Vaidya. Experimental performance comparison of byzantine fault-tolerant protocols for data centers. In *INFOCOM*, 2012.

[34] H. Moniz, N. Ferreira Neves, and M. Correia. Turquois: Byzantine fault-tolerant consensus in wireless ad hoc networks. In *IEEE Trans.Mob.Comput.*, 2013.

[35] H. Moniz, N. Ferreira Neves, M. Correia, and P. Veríssimo. Experimental comparison of local and shared coin randomized consensus protocols. In *SRDS*, 2006.

[36] H. Moniz, N. Ferreira Neves, M. Correia, and P. Veríssimo. RITAS: services for randomized intrusion tolerance. *IEEE Trans. Dependable Sec. Comput.*, 8, 2011.
[37] A. Mostéfaoui, M. Hamouma, and M. Raynal. Signature-free asynchronous byzantine consensus with t 2<n/3 and o(n^2) messages. In *PODC*, 2014.
[38] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000), 2-6 April 2000, Eilat, Israel*, page 52, 2000.
[39] O. Oluwasanmi, J. Saia, and V. King. An empirical study of a scalable byzantine agreement algorithm. In *IPDPS workshop*, 2010.
[40] W. p. Ken Yiu, Xing Jin, and S. h. Gary Chan. Vmesh: Distributed segment storage for peer-to-peer interactive video streaming.
[41] A. Patra, A. Choudhury, and C. Pandu Rangan. Asynchronous byzantine agreement with optimal resilience. *Distributed Computing*, 27, 2014.
[42] A. Patra and C. Pandu Rangan. Brief announcement: communication efficient asynchronous byzantine agreement. In *PODC*, 2010.
[43] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27, 1980.
[44] M. O. Rabin. Randomized byzantine generals. In *FOCS*, 1983.
[45] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Olveczky. Pals: Physically asynchronous logically synchronous systems. http://hdl.handle.net/2142/11897.
[46] S. Toueg, K. J. Perry, and T. K. Srikanth. Fast distributed agreement. *SIAM J. Comput.*, 16, 1987.
[47] B. Vavala and N. Neves. Robust and speculative byzantine randomized consensus with constant time complexity in normal conditions. In *SRDS*, 2012.
[48] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, 2009.

APPENDIX

A. Algorithm Quorum[7]

Definition 2. *A protocol P is said to achieve graded broadcast if, at the beginning of the protocol the dealer D holds a value v, and at the end of the protocol, every process P_i outputs a pair (v_i, c_i) where $c_i \in \{0, 1, 2\}$ denotes the confidence of the process in value v_i. With that, the following properties should hold:*

1) *If D is honest, then $v_i = v$ and $c_i = 2$ for every honest process P_i.*
2) *For any two honest processes P_i and P_j, $\mid c_i - c_j \mid \leq 1$.*
3) *(Consistency) For any two honest processes P_i and P_j, if $c_i > 0$ and $c_j > 0$, then $v_i = v_j$.*

The graded broadcast algorithm is described in detail as follows:

Input to the Dealer D**:** A value v
Output of process P_i**:** A pair (v_i, c_i)
Step 1 The dealer D distributes v to all the processes.
Step 2 (For every process P_i) Send v_i, the received value from the dealer, to all other processes.
Step 3 (For every process P_j) Let v_i^j denote the message from process P_i in Step 2. If there is a value μ such that $\geq n - k$ of the v_i^j's are equal to μ, then send μ to all the processes. Else, send $\perp$.
Step 4 (For every process P_i) Let $\texttt{num}_\mu$ denote the number of players that sent μ to P_i in Step 3.

- If $\texttt{num}_\mu \geq 2k + 1$ for some μ, output $(\mu, 2)$.
- If $2k \geq \texttt{num}_\mu \geq k + 1$ for some μ, output $(\mu, 1)$.
- If $\texttt{num}_\mu \leq k$ for all μ, output $(\perp, 0)$.

B. Algorithm Pull-Push [9]

Algorithm 1: Push phase

Input: Process P_i with a random string g_{string}, a list of all possible strings C_{string}
Output: Each node creates a candidate strings list L_{P_i}

$g_{string} \leftarrow$ createRandString();
broadcast(g_{string});
$L_{P_i} \leftarrow g_{string}$;
foreach $str \in C_{string}$ **do**
  $I_{str} \leftarrow$ getPushQuorum(str, P_i);
  foreach $p \in I_{str}$ **do**
    if $recv(I_{str}) == str$ **then**
      $num++$;
  if num $> len(I_{str})/2$ **then**
    $L_{P_i} \leftarrow L_{P_i} \cup str$;

Algorithm 2: Sending Pull Request

Input: L_{P_i}, list of candidates for node P_i
Output: String agreed upon

foreach $s \in L_{P_i}$ **do**
  $r_{P_i,s} \leftarrow$ generateRand();
  $J_{r,s} \leftarrow$ getPollList($r_{P_i,s}, P_i$);
  $H_s \leftarrow$ getPullQuorum(s, P_i);
  send($POLL, s, r_{P_i,s}, J_{r,s}$);
  send($PULL, s, r_{P_i,s}, H_s$);
  Upon event: recv($ANSWER, s, r$) $\Leftarrow w$;
  if $w \in J_{r,s}$ **then**
    $\texttt{count}_s++$;
    if $\texttt{count}_s > \frac{1}{2}|J_{r,s}|$ **then**
      has_decided $\leftarrow$ true;
      $\texttt{final}_s \leftarrow s$;
      return s;

Algorithm 3: Routing Pull Request

Upon event: recv($PULL, s, r_{x,s}, H_s$) $\Leftarrow x$;
  if ($g_{string} == s$) *and* ($P_i \in H_s$) **then**
    $J_{x,r_{x,s}} \leftarrow$ getPollList($r_{x,s}, x$);
    foreach $w \in J_{x,r_{x,s}}$ **do**
      $H_{w,s} \leftarrow$ getPullQuorum(s, w);
      send($ROUTE, x, s, r_{x,s}, w$) $\Rightarrow H_{w,s}$;
Upon event: recv($ROUTE, x, s, r_{x,s}, w$) $\Leftarrow P_j$;
  $H_{x,s} \leftarrow$ getPullQuorum(s, x);
  $J_{x,r_{x,s}} \leftarrow$ getPollList($r_{x,s}, x$);
  if ($g_{string} == s$) *and* ($P_j \in H_{x,s}$) *and* ($w \in J_{x,r_{x,s}}$) **then**
    $\texttt{fw\_count}_{s,x}++$;
    if $\texttt{fw\_count}_{s,x} > \frac{1}{2}|H_{x,s}|$ **then**
      send($FORWARD, x, s, r_{x,s}$) $\Rightarrow w$;
      $\texttt{fw\_count}_{s,x} \leftarrow \infty$;

Algorithm 4: Answering Pull Request

Upon event: recv($ROUTE, x, s, r_{x,s}$) $\Leftarrow z$;
  if $\texttt{count}_s > \texttt{log}^2 n$ **then**
    Wait for has_decided;
  $J_{x,r_{x,s}} \leftarrow$ getPollList($r_{x,s}, x$);
  $H_{P_i,s} \leftarrow$ getPullQuorum(s, P_i);
  if ($g_{string} == s$) *and* ($P_i \in J_{x,r_{x,s}}$) *and* ($z \in H(P_i, s)$) **then**
    $\texttt{fw\_count}_{s,x}++$;
    if ($\texttt{fw\_count}_{s,x} > \frac{1}{2}|H_{x,s}|$) *and* ($(x, s) \in$ Polled) **then**
      $\texttt{count}_s++$;
      send($ANSWER, s, r_{x,s}$) $\Rightarrow x$;
Upon event: recv($POLL, s, r_{x,s}$) $\Leftarrow x$;
  $J_{x,r_{x,s}} \leftarrow$ getPollList($r_{x,s}, x$);
  if $P_i \in J_{x,r_{x,s}}$ **then**
    Polled $\leftarrow$ Polled $\cup (x, s)$;
    if $\texttt{fw\_count}_{s,x} > \frac{1}{2}|H_{x,s}|$ **then**
      $\texttt{count}_s++$;
      send($ANSWER, s, r_{x,s}$) $\Rightarrow x$;
      $\texttt{fw\_count}_{s,x} \leftarrow \infty$;