An Empirical Study of a Scalable Byzantine Agreement Algorithm

Olumuyiwa Oluwasanmi, Jared Saia
Department of Computer Science,
University of New Mexico,
Albuquerque, NM 87131-1386.
Email: {muyiwa,saia}@cs.unm.edu
This research was partially supported by NSF CAREER Award 0644058, NSF CCR-0313160, and an AFOSR MURI grant.

Valerie King
Department of Computer Science, University of Victoria,
P.O. Box 3055, Victoria, BC, Canada V8W 3P6.
Email:val@cs.uvic.ca
This research was supported by an NSERC grant.

***Abstract*—A recent theoretical result by King and Saia shows that it is possible to solve the Byzantine agreement, leader election and universe reduction problems in the full information model with $\tilde{O}(n^{3/2})$ total bits sent. However, this result, while theoretically interesting, is not practical due to large hidden constants. In this paper, we design a new practical algorithm, based on this theoretical result. For networks containing more than about $1,000$ processors, our new algorithm sends significantly fewer bits than a well-known algorithm due to Cachin, Kursawe and Shoup. To obtain our practical algorithm, we relax the fault model compared to the model of King and Saia by (1) allowing the adversary to control only a $1/8$, and not a $1/3$ fraction of the processors; and (2) assuming the existence of a cryptographic bit commitment primitive. Our algorithm assumes a partially synchronous communication model, where any message sent from one honest player to another honest player needs at most Δ time steps to be received and processed by the recipient for some fixed Δ, and we assume that the clock speeds of the honest players are roughly the same. However, the clocks do not have to be synchronized (i.e., show the same time)**

***Keywords*-Leader Election, Byzantine Agreement, Consensus, Distributed Algorithms, Byzantine, Fault tolerance**

I. Introduction

Increases in speed, frequency and severity of attacks on the Internet have led to a resurgence of interest in traditional problems of robust distributed computing like Byzantine agreement (BA) [2], [11]. Unfortunately, traditional algorithms for solving problems of robust distributed computation typically require each processor to send messages to every other processor in the network, and so simply do not scale to modern network sizes, which may be on the order of hundreds of thousands for peer-to-peer systems, overlay networks and server farms.

In this paper, we seek to redress this issue by designing, implementing and testing an algorithm that solves Byzantine agreement with a total number of bits sent that is $\tilde{O}(n^{3/2})$. This paper focuses on a well-studied message-passing model: n processors are in a fully connected network and a malicious adversary controls a constant fraction of these processors. The contributions of this paper are as follows

- We design a new algorithm which is based on, but more practical than, the consensus algorithm from [8]. Our new algorithm significantly reduces the constants compared to the previous algorithm through use of cryptography.
- We implement and simulate our new algorithm, showing empirically that for large networks, it can achieve consensus with significantly less bandwidth than algorithms that are currently used in practice.

A. Model

We assume a fully connected network of n processors, whose IDs are common knowledge. Each processor has a private coin. Communication channels are authenticated, in the sense that whenever a processor sends a message directly to another, the identity of the sender is known to the recipient. We assume a *non-adaptive* (sometimes called *static*) adversary. That is, the adversary chooses the set of t bad processors at the start of the protocol, where t is a constant fraction, of the number of processors n. The adversary is *malicious*: it chooses the input bits of every processor, bad processors can engage in any kind of deviations from the protocol, including false messages and collusion, or crash failures, while the remaining processors are good and follow the protocol. Bad processors can send *any* number of messages.

We assume a partially synchronous communication model: any message sent from one honest player to another honest player needs at most Δ time steps to be received and processed by the recipient for some fixed Δ, and we assume that the clock speeds of the

honest players are roughly the same. However, the clocks do not have to be synchronised (i.e., show the same time) nor do we require the protocols to run in a synchronous mode (i.e., all players must send their messages at exactly the same time). Our algorithm makes use of a distributed random number generating algorithm from [1] and the algorithm from [3] as a subroutine. Thus, we must make the same assumptions as in those papers. Namely, we assume the existence of 1) public key cryptography (but *not* a public key infrastructure) ; 2) a digital signature scheme; and 3) a bit commitment scheme h, i.e. a function such that $h(x)$ reveals nothing about x.

We overcome the lower bound of [4] by allowing for a small probability of error. In particular, the $\Omega(n^2)$ lower bound on the number of messages to compute Byzantine agreement deterministically implies that any randomized protocol which always uses $o(n^2)$ messages must err with some probability $\rho > 0$, since with probability $\rho > 0$, an adversary can guess the random coinflips and cause the protocol to fail when those coinflips occur. Thus, any randomized algorithm that always achieves $o(n^2)$ messages must necessarily be a Monte Carlo algorithm.

B. Problems

One of the most well studied problems in distributed computing is the *Byzantine agreement* problem. In this problem, each processor begins with either a 0 or 1. An execution of a protocol is *successful* if all processors terminate and, upon termination, agree on a bit held by at least one good processor at the start. The *leader election* problem is the problem of all processors agreeing on a good processor [9]. The *universe reduction* problem [6] is to bring processors to agreement on a small subset of processors with a fraction of bad processors close to the fraction for the whole set. I.e., the protocol terminates and each good processor outputs the same set of processor ID's such that this property holds. For each of these problems, we say the protocol solves the problem with probability ρ if, given any worst case adversary behavior, including choice of initial inputs, the probability of success of any execution over the distribution of private random coin tosses is at least ρ.

Almost everywhere Byzantine agreement, universe reduction, and leader election is the modified version of each problem where instead of bringing all good processors to agreement, a large majority, but not necessarily all, good processors are brought to agreement.

C. Our Results

We show that by making use of the cryptographic assumptions detailed above and by relaxing the fraction of bad processors to $1/8$, we can significantly improve the communication costs and improve the load balancing characteristics while keeping the latency the same or slightly better. Our algorithm works in the synchronous model of communication; although we conjecture that our algorithm could be converted to make it asynchronous. Our research is an extension of the work of two previous papers [9], [10] which introduce the concept of an election graph with groups of processors called committees. The algorithms in these two papers use Feige's protocol described in [5] to elect processors in committees in successive layer's of the election graph. The use of Feige's protocol in the previous algorithms carry a heavy penalty in terms of the message complexity for any reasonable size of networks.

Our new algorithm has two parts. The first part is an almost everywhere Byzantine agreement algorithm similar to [9], [10] except that instead of using Feige's protocol we use a protocol by Awerbuch and Scheideler [1] to elect processors in the committees. This allows us to make the sizes of the committees much smaller, which leads to a significant improvement on the message complexity performance of our algorithms from [9], [10]. Secondly, we implement and make use of a protocol recently described in [8] which allows us to go from almost everywhere Byzantine agreement to everywhere Byzantine agreement using only $\tilde{O}(\sqrt{n})$ messages. In particular, this new algorithm ensures that with high probability all good processors in our new algorithm learn the correct bit unlike the algorithms in [9], [10].

D. Related Work

The algorithm presented in this paper, along with the algorithm of [8], use randomization to break through the 1985 $\Omega(n^2)$ barrier [4] for message and bit complexity for Byzantine agreement in the deterministic synchronous model, if we assume the adversary's choice of bad processors is made at the start of the protocol, i.e., independent of processors' private coinflips. As mentioned above, the algorithm in this paper also makes use of algorithms from [9], [10] to solve the almost everywhere Byzantine agreement problem.

In the empirical section of our paper, we compare the resource costs of our algorithm with the Byzantine agreement algorithm proposed by Cachin, Kursawe and Shoup [3]. Their algorithm withstands up to $n/3$ bad processors, runs in constant expected time, and sends $\theta(n^2)$ messages. However, unlike our algorithm, their algorithm requires a trusted dealer to distribute cryptographic keys initially in order to set up a public

key infrastructure. We emphasize that our algorithm does *not* require the establishment of a public key infrastructure. As pointed out in the abstract, the algorithm we describe in this paper is partially synchronous, while the algorithm of Cachin, Kursawe and Shoup is asynchronous.

Organization of the paper In Section II we describe our algorithm. In Section IV we empirically evaluate our algorithm. In section III we describe the algorithmic details and proofs. Finally, in Section V we conclude and describe open problems.

II. Our Algorithm

Our algorithm consists of two parts: 1) a procedure for solving almost-everywhere universe reduction (and BA); and 2) a procedure for going from almost-everywhere universe reduction to everywhere BA. Our procedure for solving almost-everywhere universe reduction is essentially the same as that of [9], with the following two differences: (1) we replace the committee election protocol used in [9], with a new election protocol based on a random number generation protocol by Awerbuch and Scheideler [1]; and (2) we reduce the size of all committees from $O(\log^3 n)$ to $O(\log n)$ and make the appropriate changes in our election graph. This reduction in committee size is made possible by the new election protocol from [1], which makes use of a cryptographic commitment scheme. It is this reduction in committee size that leads to significant savings in bandwidth over the protocol of [9]. We note that we are not particularly dependent on the algorithm from [1]; any robust, distributed random number generating algorithm will work. We simply use this algorithm because it is the current state of the art in terms of resource costs.

Below, we sketch our entire protocol, details are given in section III.

A. Almost Everywhere Universe Reduction and BA

We first describe our protocol to compute almost everywhere universe reduction, based on [9]. The processors are assigned to groups of logarithmic size; each processor is assigned to multiple groups. In parallel, each group then elects a small number of processors from within their group to move on. We then recursively repeat this step on the set of elected processors until the number of processors left is logarithmic. Although this approach is intuitively simple, there are several complications that must be addressed.

(1) The groups must be determined in such a way that the election mechanism cannot be sabotaged by the bad processors.
(2) After each step, each elected processor must determine the identities of certain other elected processors, in order to hold the next election.
(3) Election results must be communicated to the processors.
(4) To ensure load balancing, a processor which wins too many elections in one round cannot be allowed to participate in too many groups in the next round.

We address these problems as follows. Item (1): we use a layered network with extractor-like properties. Every processor is assigned to a specific set of nodes on layer 0 of the network. In order to assign processors to a node A on layer $\ell > 0$, the set of processors assigned to nodes on layer $\ell - 1$ that are connected to A hold an election. In other words, the topology of the network determines how the processors are assigned to groups. By choosing the network to have certain desired properties, we can ensure that the election mechanism is robust against malicious adversaries.

To accomplish item (2), we use *monitoring sets*. Each node A of the layered network is assigned a set of nodes from layer 0, which we denote $m(A)$. The job of the processors from $m(A)$ is simply to know which processors are assigned to node A. Since the processors of $m(A)$ are fixed in advance and known to all processors, any processor that needs to know which processors are assigned to A can simply ask the processors from $m(A)$. (In fact, the querying processor only needs to randomly select a polylogarithmic subset of processors from $m(A)$ in order to learn the identities of the processors in A with high probability. This random sampling will be used to ensure load balancing.)

Since the number of processors that need to know the identities of processors in node A is polylogarithmic, the processors of $m(A)$ will not need to send too many messages, but they need to know which processors need to know so they do not respond to too many bad processors' queries. Hence the monitoring sets need to inform relevant other monitoring sets of this information.

Item (3): We use a *communication tree* connecting monitoring sets of children in the layered networks with monitoring sets of parents to inform the monitoring sets which processors won each of their respective elections and otherwise pass information to and from the individual processors on layer 0.

Item (4) is addressed by having such processors refrain from further participation.

The protocol results in almost everywhere agreement on the subset of nodes rather than everywhere agreement, because the adversary can control a small

fraction of the monitoring sets by corrupting their nodes. Thus communication paths to some of the nodes are controlled by the adversary. We further note that with this protocol, it is trivial to communicate a bit to almost all of the nodes in addition to communicating a small subset. Thus, it solves both almost-everywhere BA and almost everywhere Universe reduction. [1]

B. Almost Everywhere to Everywhere

In this section, we describe the Almost Everywhere to Everywhere protocol of [8]. This is exactly the same protocol as from [8] but we include a description of it here for completeness. In the almost everywhere protocol sketched above all but a $1/\ln n$ fraction of the good processors agree on a small subset of representative processors (and a bit). This result is proven in [8]. Our goal in this section is to improve the fraction of good processors that agree on the bit. The protocol assumes all processors have used the almost-everywhere agreement protocol to ensure that the following precondition holds:

Precondition: We assume there is a subset C of $O(\log^3 n)$ processors, a majority of which are good; and a bit b that is the input bit of at least one good processor. Each processor p starts with an hypothesis of the membership of C, C_p; this hypothesis may or may not be equal to C or may be empty; and each processor p starts with a value b_p that may or may not be equal to b. The following assumption is critical: there is a set S of at least $(1/2+\epsilon)n$ good processors, such that for all $p \in S$, $C_p = C$ and $b_p = b$.

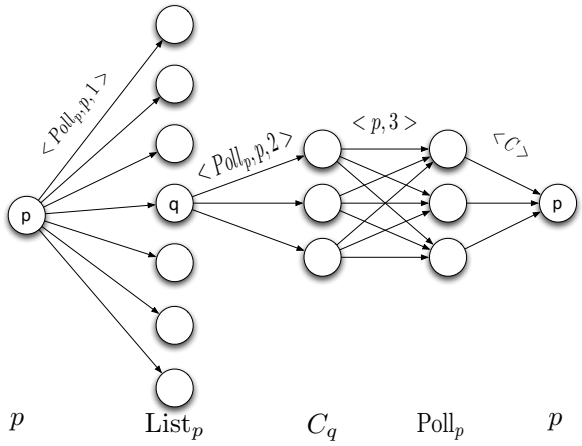

Figure 1: Steps 6-10 of Our Protocol

Algorithm 1 Almost Everywhere to Everywhere

Each processor executes the following steps in any order:

1) Each processor p selects uniformly at random, independently, and with replacement three subsets, $List_p$, $Forward_p$, and $Poll_p$ of processor ID's where: $|List_p| = c\sqrt{n}\log n$; $|Forward_p| = \sqrt{n}$; $|Poll_p| = c\log n$;

Verifying Membership in C:

2) $member_p \leftarrow FALSE$
3) If $p \in C_p$, then p sends a message $<$ *Am I in C?* $>$ to the members of $Poll_p$;
4) If q receives a message $<$ *Am I in C?* $>$ from a processor $p \in C_q$, q sends $< Yes >$ back to the p;
5) If p receives a message $<$ *Yes* $>$ from a majority of members of $Poll_p$ then p sets $member_p \leftarrow TRUE$;

Determing C:

6) p sends a message $< Poll_p, p, 1 >$ (type 1 message) to each processor in $List_p$;
7) For each q: if $< Poll_p, p, 1 >$ is the first type 1 message received from processor p and $p \in Forward_q$, then q sends $< Poll_p, p, 2 >$ (a type 2 message) to every processor in C_q;
8) For each r: if $member_r = TRUE$ then for every processor q, for the first $\sqrt{n}$ type 2 messages of the form $< Poll_p, p, 2 >$ which are received from q, send $< p, 3 >$ (type 3 message) to every processor in $Poll_p$;
9) For each s: for the first $\sqrt{n}\log^2 n$ different type 3 messages of the form $< p, 3 >$ which are each sent by a majority of processors in C_s, send $< C_s, 4 >$ (type 4 message) to p;
10) If s receives the same type 4 message $< C', 4 >$ from a majority of processors in $Poll_s$ then
    a) s sets $C_s \leftarrow C'$; and
    b) s answers any remaining type 3 requests that have come from a majority of the current C_s, i.e. for each such request $< p, 3 >$ s sends $< C_s, 4 >$ to p;

[1] Moreover the processors that correctly learn the subset of nodes will also learn the correct bit for BA.

Overview of Algorithm: The main idea of this protocol is for each processor p to randomly select $c \log n$ processors to poll as to the membership of C. Unfortunately, if these requests are made directly from p, the adversary can flood the network with "fake" requests so that the good processors are forced to send too many responses. Thus, the polling request are made through the set C, which counts the messages received from each processor to enforce that total number of polling requests sent out is not too large.

Unfortunately, this approach introduces a new problem: processor p may have an incorrect guess about the membership of C. We solve this by having p send a (type 1) message containing its poll-list ($Poll_p$) to $List_p$, a set of $c \log n \sqrt{n}$ randomly sampled processors. Processor p hopes that at least one processor in the set $List_p$ will have a correct guess about C and will thus be able to forward a (type 2) message containing $Poll_p$ to C. To prevent these processors $q \in List_p$ from being flooded, each such processor q only forwards a type 2 message from a processor p if p appears in the set $Forward_q$, which is a set of $\sqrt{n}$ processors that are randomly sampled in advance. Upon receiving a $< Poll_p, p >$ (type 2) message from any processor q, a processor in C then sends a (type 3) request with p's ID to each member $s \in Poll_p$. More precisely, a processor in C only processes the first $\sqrt{n}$ such type 2 messages that it receives from any given processor q: this is the crucial filtering that ensures that the total number of requests answered is not too large. Upon receiving a type 3 request, $< p, 3 >$ from a majority of C, s sends C_s to p, a (type 4) message.

There are two remaining technical problems. First, since a confused processor, p, can have a C_p equal to a mostly corrupt set C', C' can overload every confused processor. Hence we require that any processor, p, who receives an overload (more than $\sqrt{n} \log^2 n$) of type 3 requests wait until their own C_p is verified before responding. Second, the processors in C handle many more requests than the other processors. The adversary can conceivably exploit this by bombarding confused processors which think they are in C with type 2 requests. Thus, the algorithm begins with a verification of membership in C. Each processor p sends a request message to a randomly selected sample ($Poll_p$) which is responded to by a polled processor q if and only if $p \in C_q$.

Example: An example run of our algorithm is shown in Figure 1. This figure follows the technically challenging part of our protocol, steps 6-10, which are described in detail in Algorithm 1 listed below. In Figure 1, time increases in the horizontal direction. This figure concerns a fixed processor p that concludes $p \notin C$ in the earlier parts of the algorithm (steps 2-5). For clarity, in this example, only messages that are sent on behalf of p that eventually help p to determine C are shown. Moreover, again for clarity, we show a best case scenario where all nodes in $Poll_p$ are assumed to have received no more than $\sqrt{n} \log^2 n$ type 3 requests. In the first step of this example, p sends the message $< Poll_p, p, 1 >$ to all nodes in $List_p$. The node q is the only node in this set such that $p \in Forward_q$, so q forwards a type 2 message of the form $< Poll_p, p, 2 >$ to all the nodes in C_q. In this example, $C_q = C$. Next all nodes in C_q send the message $< p, 3 >$ to all nodes in $Poll_p$. In this example, all nodes in $Poll_p$ know the set C, so they all send the message $< C >$ to p in the final step.

III. Detailed description of our algorithm and the election graph.

A. Committees

In a network with n processors, a committee is a collection of $O(\ln n)$ processors in the network. First, we choose a set of committees each of size $O(\ln n)$ chosen uniformly at random from the n processors in the network. We call this initial set of processors in committees layer '0' committees.

DEFINITION 1. *A committee is called good if less than a $1/8$ fraction of the processors in the committee have been taken over by the adversary.*

B. Committee Selection

We use the sampler to spread out coalitions of bad processors. Each committee is selected from the processors elected by the ElectHigherComm algorithm from the previous layer so that the fraction of bad committees is bounded by $\epsilon'/2 \ln n$ for some $\epsilon' \in [0, 1]$. The RandomID protocol, which guarantees that the additional fraction of bad processors due to elections from good committees is also bounded by $\epsilon'/2 \ln n$ which then gives a total bound of $\epsilon'/\ln n$ on the growth of the fraction of bad processors appearing in successive layers in the worst case.

C. Election description

The election graph consists of a full tree with the layer '0' committees at the leaf nodes. Initially, there are committees only at the leaf nodes. These committees are created by a uniform sampler that assigns the processors to committees. The processors in non-leaf nodes will be elected by the algorithm described later in this section. This algorithm repeatedly makes use of the

RANDOMID protocol from [1]. This protocol is used to assign random numbers in the range $[0, 1]$. We use these random numbers to select a processor in each committee to advance to subsequent layers. First, we describe the properties of the protocol in [1] by the following Theorem from [1].

1) COMMITTEE ELECTION PROTOCOL

The method used to run elections is a simple adaptation from the random number generation protocol of [1]. This protocol requires a bit commitment scheme h, where $h(x)$ reveals nothing about x. In practice, a cryptographic hash function should be sufficient for h.

Suppose that we have a set P of m players, $p_1, \ldots p_m$, that know each other and their indexing, with any t of them being adversarial for some $t < m/6$. The round-robin random number generator works as follows for some player $p^\star \in P$ initiating it.

1) Each player $p_i \in P$ sets $P_i := P \backslash \{p_i\}$ and waits for $8i$ time steps. Each time it receives an accusation $(p_k)_{p_j}$ from a player $p_j \in P$ it has not received an accusation from yet, it sets $P_i := P \backslash \{p_k\}$. Once the $8i$ steps are over, p_i initiates the next step. p_i terminates after $8(m+1)$ steps.
2) If $|P_i| \geq 2m/3$, then p_i chooses a random $x_i \in \{0,1\}^s$ and sends $(h(x_i), P_i)_{p_i}$ to all players in P_i. Otherwise, p_i aborts the protocol (which will not happen if $t < m/6$).
3) Each player $p_j \in P_i$ receiving a message $(h(x_i), P_i)_{p_i}$ for the first time from p_i with $P_i \geq 2m/3$ chooses a random $x_j \in \{0,1\}^s$ and sends the message $(p_i, h(x_j), P_i)_{p_j}$ to p_i. Otherwise, it does nothing.
4) If all players in P_i reply within 2 time steps, then p_i sends $(\{(p_i, h(x_j), P_i)_{p_j} | p_j \in P_i\})_{p_i}$to all players in P_i. Otherwise, p_i sends an accusation $(p_j)_{p_i}$ for any $p_j \in P_i$ that did not reply correctly or in time to all players in P and stops its attempt of generating a random number.
5) Once $p_j \in P_i$ receives $(\{(p_i, h(x_k), P_i)_{p_k} | p_k \in P_i\})_{p_i}$ from p_i, p_j sends $(x_j)p_j$ to p_i.
6) If p_i gets a correct reply back from all players in P_i within 2 time steps, then it sends $(x_i, \{(x_j)_{p_j} | p_j \in P_i\})_{p_i}$ to all players in P_i and computes $y_i = x_i \oplus \bigoplus_{p_j \in P_i} x_j$ where $\bigoplus$ is the bit-wise XOR operation. Otherwise, p_i sends an accusation $(p_j)_{p_i}$ to all players in P for any $p_j \in P_i$ that did not reply correctly or in time and stops.
7) Once $p_j \in P_i$ receives $(x_i, \{(x_k)_{p_k} | p_k \in P_i\})_{p_i}, p_j$ verifies that all keys are correct. Then p_j computes $y_j^{(i)} = x_i \oplus \bigoplus_{q_k} \in P_i x_k$ and sends the message $(y_j^{(i)})_{p_j}$ to p_i.
8) If p_i receives y_i from at least $2m/3$ players in P within 2 time steps, it accepts the computation and otherwise sends an accusation $(p_j)_{p_i}$ to all players in P for any $p_j \in P_i$ that did not reply correctly or in time.

THEOREM 1. *Suppose that $|P| = m$ and there are $t < 1/6m$ adversarial peers in P. Then the* RANDOMID *protocol generates random keys $y_1, y_2, \ldots y_k \in \{0,1\}^s$ with $m - 2t \leq k \leq m$ and the property that for all subsets $S \subseteq \{0,1\}^s$ with $\sigma = |S|/2^s$, $E[|i\,|y_i \in S|] \in [(m-2t)\sigma, m \cdot \sigma]$*
Further, the worst-case message complexity of the protocol is $O(m^2)$.

ALGORITHMIC DESCRIPTION.: The RANDOMID protocol elects only one processor from each committee to advance to the next layer i of the network and broadcasts a message to its members informing them which processor was elected. This is done using a procedure called ELECTHIGHERCOMM(A) shown in Figure 1. The procedure is called once for each layer $i-1$ to elect the nodes in layer i. As each processor is elected to layer i, the processors in its respective committees in layer $i-1$ are informed via a broadcast to the processors in their committees of the election of these processors. Within the procedure ELECTHIGHER-COMM, the selected processors are assigned committees by the sampler to disperse possible coalitions of bad processors to reduce the probability of having bad processors take over a committee. The sampler selects the processors in each committee such that there are $O(\ln n)$ sized committees with only a small fraction of these committees being bad at each layer i. The processors in each committee A ,learn which processors belong to A by contacting a set of processors called the monitoring set of A which we will refer to as $m(A)$ which is a structure from [9]. The monitoring set of a committee A, is a set of layer '0' committees which know the processors assigned to A, whose identity is fixed in advance and is known to all processors. The processors in committee A, need only randomly sample $O(\ln n)$ processors in $m(A)$ to learn the processors in A with high probability. We will describe later in this section the mechanism by which these monitoring sets learn the processors in each committee. These elections continue until layer l consisting of at least $O(\ln^3 n)$ processors, this algorithm is performed by a procedure called TREESELECT given in figure 2. This procedure is later called by the BA algorithm shown in figure 3. After receiving a set of $O(\ln^3 n)$ processors

from TREESELECT, it runs the Byzantine agreement algorithm in [3] which we will refer to from now on as the RANDOMORACLE on these set of processors. The agreed upon bit by the good processors is then broadcast to all other processors in the network. This is done by traversing the Election graph downward in a breadth first manner and broadcasting the identity of the processor selected to be the leader to each processor in each committee. We note here that this algorithm can be adapted easily to solve the Leader Election problem by using the RANDOMID protocol to select a leader and then broadcasting the identity of this leader downward through the Election graph as mentioned earlier. The monitoring sets are described in [9].

D. ALMOST EVERY WHERE BYZANTINE AGREEMENT PROTOCOL

The almost everywhere Byzantine agreement Protocol, called AEBA, works by calling the RANDOMORACLE algorithm with the group of processors in the last layer of the election graph as input. The RANDOMORACLE algorithm has a message complexity $O(n^2)$ on input of size n. We show that the algorithm AEBA sends no more that $O(\ln^3 n)$ messages per processor in a fully connected communication network. We show this by the following Theorem.

THEOREM 2. *The Byzantine agreement Protocol* AEBA *sends at most* $O(n \ln^3 n)$ *messages in total using a fully connected communication network (complete graph) .*

Algorithm 2 INFORMMONITORINGSET(A)

Input: A the set of processors in a committee in layer i

1: **while** not layer 0 **do**
2: **if** the processors if the processors in the child nodes $C_1, C_2, \ldots C_j$ for A are unknown **then**
3: CALL FINDPROCESSORS(C_j) to learn the processors in each C_j that are children of A.
4: committee A in layer i, sends a message to each of it's children $C_1, C_2, \ldots C_j$ in layer $i-1$
5: CALL INFORMMONITORINGSET on each of $C_1, C_2, \ldots C_j$

Figure 2: Algorithm for informing sending messages to monitoring sets.

Algorithm 3 FINDPROCESSORS(A)

Input: A the set of processors in a committee in layer i

1: **for** each processor $p \in A$ **do**
2: sample uniformly at random a set S of $O(\ln n)$ processors in $m(A)$
3: poll the processors in S for the identities of the processors in A
4: accept via majority filtering the identities of the processors in A from S

Figure 3: Algorithm for learning the identity of the processors its committee.

Algorithm 4 ELECTHIGHERCOMM(A)

Input: A the set of processors in layer i of the election graph
Output: D the set of processors, elected to layer $i+1$ in the election graph

1: **for** each committee C in layer i **do**
2: **for** each processor p_i in C **do**
3: FINDPROCESSORS(C)
4: C selects the processor p with $p \leftarrow$ RANDOMID(C)
5: add p to D
{use the sampler to spread out the bad processors in committees}
6: $D = F(D)$
7: **for** each committee C_i in D **do**
8: CALL INFORMMONITORINGSET(C_i).
9: **return** D

Figure 4: Algorithm for electing the next layer of processors.

Algorithm 5 TREESELECT(A)

Input: A the set of n processors
Output: P a set of $\ln^3 n$ processors .

1: $Com_0 \leftarrow$ A
2: **for** i=0 to l and $|Com_{i+1}| > \ln^3 n$ **do**
3: for each processor in layer i {elect the processors to layer $i+1$ }
4: $Com_{i+1} \leftarrow$ ELECTHIGHERCOMM(Com_i)
5: **return** P

Figure 5: Algorithm for electing a set of $\ln^3 n$ processors

Algorithm 6 AEBA(A)

Input: A the set of n processors

Output: Most good processors agree on the value of a bit.

1: $A_0 \leftarrow$ A
2: $A_1 \leftarrow$ TREEELECT(A_0)
3: RANDOMORACLE(A_1) { Run Byzantine agreement protocol on A_1 $O(ln^3 n)$ subset of A } {inform the processors in successive layers the of the value of the bit}
4: **for** i=l to 1 **do**
5: for each node(committee) in layer i
6: Broadcast to the processors in child nodes in layer i-1 the bit agreed upon.
7: The processors in layer i-1 use majority filtering to accept the bit agreed upon.

Figure 6: Algorithm for Byzantine agreement.

E. SAMPLERS,COMMUNICATION TREE AND MONITORING SETS

1) SAMPLERS

Our protocols makes use of samplers, which are a special family of bipartite graphs that define subsets of elements such that all but a small number of the selected subsets contain close to the fraction of bad elements in the whole set. We use the definition of samplers found in [7] and equivalent to the one defined in [14].

DEFINITION 2. *Let [r] denote the set 1,... r and $[s]^d$ be subsets of $[s]$ size d. Let F be a function $[r] \rightarrow [s]^d$ that assigns the elements to subsets of size d. Then F is a $(\theta, \epsilon, \beta)$ sampler if $\forall S \subset [s]$ with $|S| > \beta s$,and at most an ϵ fraction of the inputs x have $\frac{|F(x) \cap S|}{d} > \frac{|S|}{S} + \theta$*

LEMMA 3.1. *For every $r, s, d, \theta, \epsilon, \beta > 0$ such that $(\log_2 e)(d\theta^2 \beta r)/3 > s/\epsilon$, there exists a $(\theta, \epsilon, \beta)$ sampler $[r] \rightarrow [s]^d$.*

2) COMMUNICATION TREE AND MONITORING SETS

When processors advance in the election graph, processors within a committee do not know the identity of the other peers within the committee. The monitoring sets provide this information to the members of each committee. The monitoring set of a committee A , m(A) is the set of layer '0' committees in the election graph known to everyone in advance, they are children of committee A in layer '0' in the election graph. It is convenient to assume that the committee A, in layer i is the root of some tree with it's children being the committees linked to it via the processors elected from layer $i-1$. We will from now on refer to this tree as the communication tree. This communication tree is rooted at the set of $O(\ln^3 n)$ processors in layer '0' of the election graph referred to in Lemma III-F. The children of some node i in layer l in the communication tree is the set of $O(\ln n)$ committees in layer $l-1$, that is committees numbered $i \cdot \gamma, i \cdot \gamma + 1, \cdots, (i+1) \cdot \gamma - 1$ from layer $l-1$ where $\gamma = r/s$ from the sampler properties. The leaf nodes of the communication tree are the layer '0' committees in the election graph. This scheme embeds the communication tree completely in the election graph. We assume sending messages from node A to node B to simply mean every processor in node A sending messages to every processor in node B with each processor deciding by majority filtering on the messages it received. The identity of the processors in committee A, are sent to $m(A)$ after the election of each processor to a committee using the procedure called INFORMMONITORINGSET. The procedure sends the identity of the elected processors in A to $m(A)$ by recursively sending messages through the children of node A in the communication tree.. The procedure FINDPROCESSORS is called by each processor in a committee to learn the identities of the processors in that particular committee. It does this by sampling uniformly at random $O(\ln n)$ processors in $m(A)$ to learn the identity of the processors in A. This guarantees that with high probability, these nodes know the processors in A.

F. DETAILED PROOFS.

LEMMA 3.2. *In the election graph, if the number of processors in layer $i-1$ is no less than $\ln^3 n$ and the fraction of bad processors is no more than $1/8 - \epsilon_0$ then, the fraction of bad committees in layer i is no more than $\epsilon'/2 \ln n$.*

Proof: Using the uniform $(\theta, \epsilon, \beta)$ sampler with $\theta = \epsilon_0, \beta = 1/8 - \epsilon_0, \gamma = s/r = 1/c_1, c_1 > 1, \epsilon = \epsilon'/2 \ln n$ and $(\log_2 e)(d\epsilon_0^2(1/8 - \epsilon_0))/3 > 2 \ln n / c_1 \epsilon' \Rightarrow d > C \ln n$ for some constant C. We see that the bounds are automatically satisfied from the properties of the sampler. ■

LEMMA 3.3. *Let G be the set of processors elected from good committees at layer i with at least $\ln^3 n$ processors in layer i, and the fraction of bad processors in layer i, is no more than f_i. Then with high probability, the fraction of bad processors in G is no more than $f_i + \epsilon'/2 \ln n$.*

Proof:

Let X_j be the random variable assigned to jth processor elected at layer $i-1$, such that $X_j = 1$ if the jth processor is bad and 0 otherwise with

$X = \sum_{j=1}^{i=l} X_j$ where l is the number of processors in layer i, then these random variables are independent. Using Chernoff bounds $Pr(|X - \mu| \geq \delta\mu) \leq 2e^{-\delta^2\mu/3}$ with $\mu = E[X] = \beta l, \delta = \epsilon'/2\ln n$ we get $Pr(|X - \mu| \geq \beta l\epsilon'/2\ln n) \leq 2e^{-\beta l\epsilon'^2/12\ln^2 n} = 1/n^c$ for some constant c, if $l \geq \ln^3 n$. ■

LEMMA 3.4. *With high probability, at layer i, with the number of processors in layer $i-1$ being no less than $\ln^3 n$, the fraction of bad processors in layer i is no more than $f_i = f_0 + i\epsilon'/\ln n$ for some constant ϵ'.*

Proof: The proof is by induction on the layer i.

1) Base case $i = 0$.
   The fraction of bad processors initially at layer 0 is f_0 .
2) Inductive Step.
   We assume the statement is true for layer i, so using the Inductive Hypothesis, $f_i = f_0 + i\epsilon'/\ln n$. For layer $i+1$, using Lemma 1.3 the additional fraction of bad processors elected to layer $i+1$ due to bad committees is at most $\epsilon'/2\ln n$ applying Lemma 1.4 the additional fraction of bad processors elected from good committees is at most $\epsilon'/2\ln n$. So the total fraction of bad processors at layer $i+1$ is $f_i+\epsilon'/2\ln n+\epsilon'/2\ln n = f_i+\epsilon'/\ln n$ which is $f_0 + (i+1)\epsilon'/\ln n$.

■

THEOREM 3. *The Byzantine agreement Protocol* BA *sends at most $O(n\ln^3 n)$ messages in total using a fully connected communication network (complete graph) .*

Proof: The number of committees in layer i of the election graph is n/γ^{i+1}. The number of processors in layer i is $Cn\ln n/\gamma^{i+1}$. The election graph is of height $l = \ln nC - 2\ln\ln n - \ln\gamma/\ln\gamma$ and $C\ln n$ is the size of a committee, where $C > 0$ is some constant defined by the properties of the sampler in Lemma 2.2.

- Communication costs for informing monitoring sets for processors in layer i:
  - Each node in the communication tree needs only to learn the identity of it's immediate children in layer $i-1$ once. The identity of the processors in child nodes in the lower layers of the communication tree are already known.
  - The cost of learning the identity of a committee by sampling $10\ln n$ processors is $10C^2\ln^3 n$.
  - The cost of learning the processors for all the immediate children of the nodes in layer i is $10nC^2\ln^3 n/\gamma^i$.
  - The cost of learning all the processors in the communication tree of height l is then $10C\ln^3 n\left(nC - (\gamma\ln n)^2\right)/\gamma(\gamma-1)$.
  - The cost of sending messages down the tree for all layers $1\cdots l$ is: $C\ln^2 n(nC\ln\left(nC/\ln^2 n\right)(\gamma-1) + \gamma\ln\gamma\left(nC - \ln^2 n\right))/\ln\gamma(\gamma-1)^2$.
  - So the total cost of informing the monitoring sets is: $C\ln^2 n(nC\ln\left(nC/\ln^2 n\right)(\gamma-1) + \gamma\ln\gamma\left(nC - \ln^2 n\right))/\ln\gamma(\gamma-1)^2 + 10C\ln^3 n\left(nC - (\gamma\ln n)^2\right)/\gamma(\gamma-1)$. Which is $O(n\ln^3 n)$.
- Next, we calculate the message complexity for the election process:
  - The cost of learning the processors for all committees in all the layers of the election graph is $10C\ln^3 n\left(nC - (\gamma\ln n)^2\right)/\gamma(\gamma-1)$.
  - The cost of running the RANDOMID algorithm for all layers of the algorithm is $10C^2\ln^2 n\left(nC - \gamma\ln^2 n\right)/(\gamma-1)$
  - The total cost of the election process is $O(n\ln^3 n)$.
- The cost of the Byzantine Agreement Algorithm run on the processors in the last layer is $O(\ln^6 n)$.
- The cost of sending the agreed bit down to the processors in layer '0' from the $O(\ln^3 n)$ processors in layer l is $C\ln^2 n\left(nC - \gamma\ln^2 n\right)/(\gamma-1)$.
- The total cost of this process is $O(n\ln^3 n)$.

■

The proofs of the correctness of the algorithm to inform all the confused processors can be found in King and Saia [8].

1) PROOF OF MESSAGE COMPLEXITY OF ALMOST EVERYWHERE BYZANTINE AGREEMENT ALGORITHM

- Communication costs for informing monitoring sets for processors in layer i:
  - Each node in the communication tree needs only to learn the identity of it's immediate children in layer $i-1$ once. The identity of the processors in child nodes in the lower layers of the communication tree are already known.
  - The cost of learning the identity of a committee by sampling $10\ln n$ processors is $10C^2\ln^3 n$.
  - The cost of learning the processors for all the immediate children of the nodes in layer i is $10nC^2\ln^3 n/\gamma^i$.
  - The cost of learning all the processors in the communication tree of height l is then $\sum_{2\leq i\leq l} 10nC^2\ln^3 n/\gamma^i = 10nC^2\ln^3 n\left(1-\gamma^{-l+1}\right)/\gamma(\gamma-1)$. For the value of l defined above we have this cost

to be $10C\ln^3 n\left(nC-(\gamma\ln n)^2\right)/\gamma(\gamma-1)$.
  - The cost of sending messages down the communication tree from layer h to inform the monitoring sets in layer '0' is: $\sum_{1\le i\le l} nC^2\ln^2 n/\gamma^i = nC^2\ln^2 n\left(1-\gamma^{-h}\right)/(\gamma-1)$.
  - The cost of sending messages down the tree for all layers $1\cdots l$ is: $\sum_{1\le h\le l} nC^2\ln^2 n\left(1-\gamma^{-h}\right)/(\gamma-1)$ $= nC^2\ln^2 n\left(\gamma^{-l}+l(\gamma-1)-1\right)/(\gamma-1)^2$. For the value of l defined above we have this cost to be: $C\ln^2 n(nC\ln\left(nC/\ln^2 n\right)(\gamma-1) + \gamma\ln\gamma\left(nC-\ln^2 n\right))/\ln\gamma(\gamma-1)^2$.
  - So the total cost of informing the monitoring sets is: $C\ln^2 n(nC\ln\left(nC/\ln^2 n\right)(\gamma-1)+\gamma\ln\gamma\left(nC-\ln^2 n\right))/\ln\gamma(\gamma-1)^2 + 10C\ln^3 n\left(nC-(\gamma\ln n)^2\right)/\gamma(\gamma-1)$. Which is $O(n\ln^3 n)$.

- Next, we calculate the message complexity for the election process:
  - The communication cost for learning the processors in layer $i\ge 1$ is $10nC^2\ln^3 n/\gamma^{i+1}$.
  - The cost of learning the processors for all committees in all the layers of the election graph is $\sum_{1\le i\le l-1} 10nC^2\ln^3 n/\gamma^{i+1} = 10nC^2\ln^3 n\left(1-\gamma^{-l+1}\right)/\gamma(\gamma-1)$. The actual cost substituting for the value of l is $10C\ln^3 n\left(nC-(\gamma\ln n)^2\right)/\gamma(\gamma-1)$.
  - The cost of running the RANDOMID algorithm per committee election is $10C^2\ln^2 n$, the cost of running the algorithm for layer i is $nC^2\ln^2 n/\gamma^{i+1}$. The cost for all layers of the algorithm is $\sum_{0\le i\le l-1} 10nC^2\ln^2 n/\gamma^{i+1} = 10C\ln^2 n\left(nC-\gamma\ln^2 n\right)/(\gamma-1)$. The actual cost substituting the value of l is $10*C\ln^2 n\left(nC-\gamma\ln^2 n\right)/(\gamma-1)$
  - The total cost of the election process is $O(n\ln^3 n)$.

- The cost of the Byzantine Agreement Algorithm run on the processors in the last layer is $4\ln^6 n$.
- The cost of sending the agreed bit down to the processors in layer '0' from the $O(\ln^3 n)$ in layer l is $\sum_{1\le i\le l}\left(nC^2\ln^2 n\right)/\gamma^i = C^2 n\ln^2 n\left(1-\gamma^{-l}\right)/(\gamma-1)$ substituting the value of l into this we get $C\ln^2 n\left(nC-\gamma\ln^2 n\right)/(\gamma-1)$.
- The total cost of this process is $O(n\ln^3 n)$.

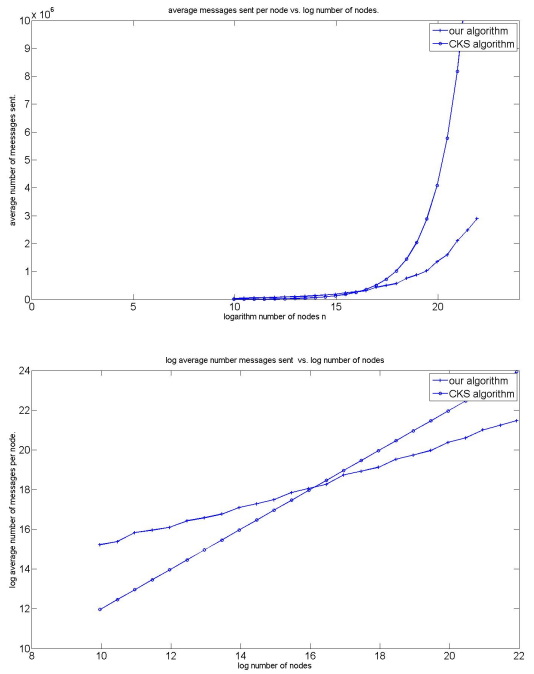

Figure 7: Top: Log of number of nodes vs. average number of messages; Bottom: Log of number of nodes vs log of average number of messages

IV. Experimental Results

A. Experimental Setup

We ran our simulations using the BEA 64 bit Java 6.0 virtual machine JRockit on a machine with 8G of memory. The size of the network simulated was between 1,000 to 4,000,000 processors. In our algorithm, the parameters for the sampler were $\gamma = r/s = 60, \beta = \epsilon_0 = 1/12$ making $d \ge 50\ln n$. We used the latest draft standards for hash functions FIPS 180-3 [12] and the latest draft standards for digital signatures [13] in our measure of the actual bit complexity of our algorithm. We used hash functions of size 512 bits, and 2048 bits for digital signatures.

We simulated two algorithm in the experiments: our algorithm which has been described previously; and the algorithm from [3] which we will refer to as the CKS algorithm. We simulated our algorithm with parameters set so that it can tolerate a $1/8$ fraction of bad processors. Our choice of the CKS algorithm was motivated by the fact that if seems to have the smallest message complexity of Byzantine agreement algorithms described in the literature. We compared these two algorithms along three metrics: number of messages sent, number of bits sent, and latency. The CKS algorithm requires each node to send to every

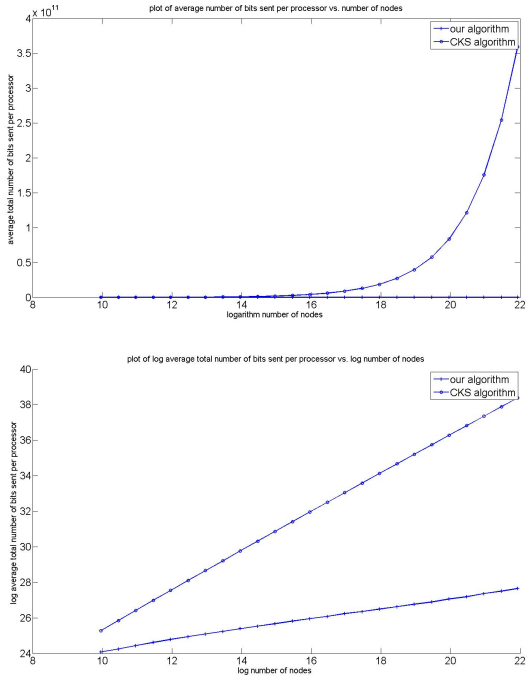

Figure 8: Top: Log of number of nodes vs. average number of bits sent; Bottom: Log of number of nodes vs log of average number of bits sent

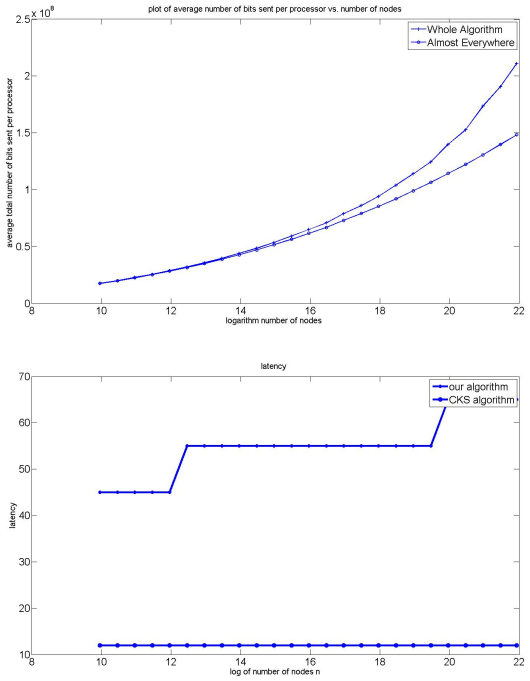

Figure 9: Top: Proportion of bandwidth used by the almost everywhere part of our algorithm. Bottom: Latency vs. the logarithm of the number of nodes

other node in the network, so the asymptotic number of messages sent per node is $O(n)$. This is in contrast to $\tilde{O}(\sqrt{n})$ for the same metric for our algorithm. The latency for the CKS algorithm is a constant in contrast to the latency for our algorithm which is $O(\log n)$. The CKS algorithm can tolerate a $1/3$ fraction of faulty processors. We emphasize that this is larger than the fraction of bad processors that can be tolerated by our algorithm as simulated here. However, our interest in scalable communication costs inclines us to consider tradeoffs of fault tolerance for scalability.

B. Experimental Results

The outcomes of our experiments are shown in Figures 7, 8 and 9. We note that, in our experiments, the measured message complexity for the CKS algorithm varies predictably for different network sizes. This is true since the CKS algorithm requires every node to send messages to every other node in the network a fixed number of times and then always stops. In contrast, the number of messages that a given node sends in our algorithm is less predictable. All data points shown in all of our plots are the average over at least 5 trials.

Figure 7 (top) shows the log of the network size vs. average number of messages sent. This plot shows that our algorithm begins to display better performance at about 65,000 processors on this metric, and for networks much larger than this size, exhibits significant improvement over the CKS algorithm. Figure 7 (bottom) shows the log of the network size vs. log of the average number of messages sent. Since this is a log-log plot, the slopes of the two lines fitting the data points give a good approximation to the exponents of n in the function giving the average message cost. Thus, as expected, in this plot the slope for the line for the CKS algorithm is approximately 1. Moreover, as expected, the slope for our algorithm is about $1/2$, since the almost everywhere to everywhere part of the algorithm requires each node to send $\tilde{O}(n^{1/2})$ messages.

Figure 8 (top) shows the log of the network size vs the average number of bits sent. For this metric, our algorithm performs better than the CKS algorithm for all networks of size greater than about $1,000$. This is due to the larger message sizes of the CKS algorithm because of its extensive use of cryptography. The bit complexity barely registers on the graph because of the resolution and since it is at most of the order of 10^8 bits. Figure 7 (bottom) shows the log of the network size vs. log of the average number of messages sent. Again the CKS algorithm displays linear slope for this plot. However,

the slope for our algorithm is about $1/4$, which much less than the $1/2$ expected. We believe this discrepancy is due to the fact that the "almost everywhere" stage of our algorithm dominates in terms of the number of bits sent for network sizes we tested, and that this stage has an asymptotic cost less than $\tilde{O}(n^{1/2})$. The dominance of the almost everywhere stage is likely due to the fact that it is the only part of our algorithm that uses cryptography. To verify our conjecture, we separated out the bit cost for the almost everywhere stage in the plot shown in Figure 9 (top). As can be seen in this figure, for larger values of n the dominance of the almost everywhere stage becomes less pronounced, and so we expect that for very large values of n, the slope in the log-log plot will approach $1/2$.

Figure 9 (bottom) shows the log of the network size vs latency. The latency for our algorithm is a step function since many values of n map to the same election graphs, and the latency of our algorithm is dominated by the diameter of the election graph.

V. Future work and Conclusion

We have described in this paper an algorithm for solving the Byzantine agreement problem using $\tilde{O}(n^{1/2})$ average messages per node. We simulated this algorithm and ran extensive experiments suggesting that for large networks, it requires significantly less bandwidth than the CKS algorithm from [3], which seems to be one of the more bandwidth efficient Byzantine agreement algorithms in the literature. For all networks our algorithm required fewer total bits sent than the CKS algorithm, and for networks of size larger than about $65,000$, our algorithm also required fewer messages sent. Our results suggest that our algorithm might be a significant step toward developing Byzantine agreement algorithm for large networks.

Several open problems remain including the following. First, the algorithm of Awerbuch and Scheideler [1] is a significant bottleneck for reducing bit cost in our algorithm. Can we devise a more efficient subroutine for choosing random numbers in our committees? We believe that this might be possible, by careful recursive use of our algorithm, coupled with use of the algorithm from Feige [5]. Second, we are interested in designing scalable algorithms for other fault-tolerant distributed computing problems, most generally, secure multi-party computation. A final goal would to implement the algorithm on a cluster of computers to do an actual distributed run of the algorithm on multiple processors.

References

[1] Baruch Awerbuch and Christian Scheideler. Robust random number generation for peer-to-peer systems. In *OPODIS*, pages 275–289, 2006.

[2] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 145–154, New York, NY, USA, 2008. ACM.

[3] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, New York, NY, USA, 2000. ACM.

[4] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.

[5] Uriel Feige. Noncryptographic selection protocols. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 142, Washington, DC, USA, 1999. IEEE Computer Society.

[6] Ronen Gradwohl, Salil P. Vadhan, and David Zuckerman. Random selection with an adversarial majority. In *CRYPTO*, pages 409–426, 2006.

[7] Bruce Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. In *SODA '08: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, New York, NY, USA, 2008. ACM Press.

[8] Valerie King and Jared Saia. From almost everywhere to everywhere: Byzantine agreement with $\tilde{O}(n^{3/2})$ bits. In *To appear in Proceedings of DISC 2009:23rd International Symposium on Distributed Computing. Elche/Elx, Spain, September 23-25, 2009*, 2009.

[9] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 990–999, New York, NY, USA, 2006. ACM Press.

[10] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 87–98, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[12] National Institutes of Standards and Technology. Federal information processing standards publication 180-3:secure hash standard. World Wide Web electronic publication, 2007.

[13] National Institutes of Standards and Technology. Federal information processing standards publication 186-3: Digital signature standard. World Wide Web electronic publication, 2007.

[14] David Zuckerman. Randomness-optimal oblivious sampling. In *Proceedings of the workshop on Randomized algorithms and computation*, pages 345–367, New York, NY, USA, 1997. John Wiley & Sons, Inc.

VI. Biographies

Olumuyiwa Oluwasanmi is a PhD candidate is the Department of Computer Science at the University of New Mexico. His interest are in designing provably secure distributed algorithms, randomized algorithms, optimisation,numerical analysis, PDE's and Applied Mathematics in general.

Jared Saia obtained his PhD in Computer Science at the University of Washington in 2002 and is now an Associate Professor at the University of New Mexico. His broad research interests are in theory and algorithms with specific interests in probability, randomized and distributed algorithms, graph theory, and spectral methods. A strong current interest is designing randomized algorithms that are provably robust against a computationally unbounded adversary. He is the recipient of several grants and awards including the NSF CAREER Award and the School of Engineering Junior Faculty Research Excellence Award.

Valerie King obtained her PhD in Computer Science at the University of California Berkeley in 1988 and is now a Professor at the University of Victoria. Her interests are in Randomized algorithms, data structures, distributed computing,lower bounds, applications to computational biology and networks.