Resilient-Optimal Interactive Consistency in Constant Time

Michael Ben-Or *
Institute of mathematics and Computer Science,
The Hebrew University, Jerusalem, Israel
benor@cs.huji.ac.il

Ran El-Yaniv
Department of Computer Science,
Technion - Israel Institute of Technology
Haifa, Israel
rani@cs.technion.ac.il

March 2001

Abstract

For a complete network of n processors within which communication lines are private, we show how to achieve concurrently many *Byzantine Agreements* within constant expected time both on synchronous and asynchronous networks. As an immediate consequence, this provides a solution to the *Interactive Consistency* problem. Our algorithms tolerate up to $(n-1)/3$ faulty processors in both the synchronous and asynchronous cases and are therefore *resilient-optimal*.

In terms of time complexity, our results improve a time bound of $O(\log n)$ (for n concurrent agreements) which is immediately implied by the constant expected time Byzantine Agreement of Feldman and Micali (synchronous systems) and of Canetti and Rabin (asynchronous systems). In terms of resiliency, our results improve the resiliency bound of the constant time, $O(\sqrt[4]{n})$-resilient algorithm of Ben-Or.

*Research supported by Israel Academy of Sciences and by United States – Israel Binational Science Foundation grant BSF-87-00082

An immediate application of our protocols is a constant expected time simulation of simultaneous *broadcast channels* over a network with private lines.
Key words: Distributed systems - Fault tolerance - Byzantine agreement - Interactive consistency - Broadcast channels

1 Introduction

We consider the following *Interactive Consistency (IC)* problem due to Pease, Shostak and Lamport [17].

> **The Interactive Consistency (IC) Problem:** consider a complete network of n processors in which communication lines are private. Among the n processors up to t may fault. Let $p_1, p_2, \ldots, p_n$ denote the processors. Suppose that each processor p_i has some private value of information $V_i \in V, |V| \geq 2$. The question is whether it is possible to devise a protocol that, given $n, t \geq 0$, will allow each nonfaulty processor to compute a vector of values with an element for each of the n processors, such that:
>
> **(i)** The nonfaulty processors compute exactly the same vector;
>
> **(ii)** The element of this vector corresponding to a given nonfaulty processor is the private value of that processor.

We say that an algorithm which solves this problem achieves *interactive consistency*, since it allows the nonfaulty processors, by means of interaction, to come to a consistent view of the values held by all the processors including the faulty ones. The Interactive Consistency Problem has an immediate reduction to the following problem.

> **The Byzantine Agreement (BA) Problem:** each processor p_i in a distributed system has some initial value V_i. For every i, processor p_i has to decide on a single value D_i such that:
>
> **(i) (Consistency)** All nonfaulty processors decide on the same value;
>
> **(ii) (Meaningfulness)** If all nonfaulty processors started with the same value, then they all decide on that value.

In order to solve the original IC problem, every processor p_i will send its value V_i to every other processor and then, using the solution to the BA problem, they will all agree upon a single value related to p_i.

Three common complexity measures are used to evaluate distributed protocols: *local* time (and space), *communication* complexity, and *distributed* time. In this work, our primary interest is in distributed time, defined to be the number of communication *rounds* required for the protocol to terminate. For a synchronous network there is a natural definition for the round term. All communications are sent during time intervals determined by pulses of a global clock accessible to all the processors. The time period between the rth and $(r+1)$st pulses is called the *rth round*. It is guaranteed that every message from a (nonfaulty) processor to another processor which is sent during the rth round will reach its destination in the $(r+1)$st pulse. For an asynchronous network it is less intuitive to define communication rounds but there is still a natural measure based on "rounds of message exchange". We defer the introduction of this measure to Section 3.

We say that a processor is *faulty* in the execution of a protocol if this processor deviates in any way from the protocol. A processor is considered *nonfaulty* if it is not faulty. Many failure models were considered in the literature. In this paper we consider the most general one: the *Byzantine fault model*. In this model a faulty processor might go completely haywire and even send contradictory messages according to an adversarial strategy.

In a totally unreliable system (every processor is faulty), no meaningful task can be achieved. Therefore, we assume that the number of faulty processors is bounded and we denote this bound by t. If a protocol P can tolerate up to t faulty processors we say that P has a *resiliency* t (or that it is *t-resilient*).

Feldman and Micali [14] were the first to show how to achieve one Byzantine Agreement over a synchronous network in constant expected time while maintaining optimal resiliency. Their algorithm can tolerate up to $(n-1)/3$ faulty processors. After the introduction of their solution an immediate question arose: Can their algorithm be exploited to concurrently achieve *many* Byzantine Agreements within (expected) constant time while maintaning optimal resiliency? In this paper we answer this question and present optimally-resilient expected constant time protocols for both synchronous and asynchronous systems.

Fast computation of many concurrent Byzantine Agreements is of importance for many applications. Let us mention a few. Any global synchronization requires all the processors in a distributed system to broadcast their information. Doing this reliably without additional time penalty is exactly what we offer here. Many distributed protocols have to use a network with reliable *broadcast channels* (i.e. any message that is sent by a certain processor over

a broadcast line, is guaranteed to be "heard" by every other processor). Using our results, broadcast channels can be simulated in expected constant time (see Section 4). The protocols of Ben-Or, Goldwasser, and Wigderson [6] and those of Chaum, Crépeau, and Damgård [11] can use our result for handling their Byzantine Agreements in expected constant time. The constant round protocols for secure function evaluation of Bar-Ilan and Beaver [2] and those of Beaver, Micali, and Rogaway [4] can use our method to simulate their broadcast channels. Yet another important application is for generating independent unbiased collective coins in expected constant time. Micali and Rabin [15] can establish this result by combining our protocol with the protocols of [6] and [14].

1.1 On the Problem of Achieving Interactive Consistency Quickly and Related Work

Given an $O(f(n))$-rounds deterministic solution for the BA problem, we can solve the IC problem within $O(f(n))$ rounds by simply running n agreements in parallel. As for randomized solutions, the situation is quite different; we cannot always solve the IC Problem by a straightforward use of fast solutions to the BA problem while keeping the same time bound. The reason for this is that the mathematical expectation of the maximum of n random variables does not necessarily equal the maximum of their expectations. For example, the probabilistic BA algorithm in [14] terminates in k rounds with probability 2^{-k} (the expected termination time is a constant of course). However, running n independent agreements in parallel will give us an expectation of $O(\log n)$ for all of them to terminate. Consider the following observation.

Observation 1 *Let $X_1, X_2, \ldots, X_n$ be independent random variables such that for every $1 \leq i \leq n$, $\Pr[X_i > j] = q^j$ $(0 < q < 1)$. If $Y = \max\{X_i\}$, then $Exp[Y] = O(\log n)$.*

We are familiar with two approaches for handling an (expected) constant time parallel computation of many Byzantine agreements. Given a BA protocol P such that the probability for not terminating after k rounds is less than $O(n^{-k})$, we could compute a polynomial number of Byzantine agreements within a constant expected number of rounds by a straightforward parallel application of P. A good example for using this approach is Ben-Or's solution in [5]; for $t = O(\sqrt[4]{n})$ he presents a BA protocol which terminates within k rounds with a probability greater than $1 - n^{-k}$.

Another possible approach is to use "slower" algorithms such as the one in [14], and to try to overcome the prolonged termination (expected) time by running *several* independent executions for each initial input vector, thus reducing the problem to that of coordinating

the processors to select a unique output vector. Here we use this approach while employing as subroutines known protocols. We need both a fast BA protocol and a fast leader election protocol. For synchronous systems we use the $(n-1)/3$-resilient protocols of Feldman and Micali [14] and for asynchronous systems we use the $(n-1)/3$-resilient protocols Canetti and Rabin [10]. In both the synchronous and asynchronous systems the maximal resiliency one can achieve $(n-1)/3$.[1] Thus, the solutions we provide for the IC problem are optimal both in terms of time (up to a constant factor) and resiliency.

Remark: The BA protocols we use as subroutines ([14] and [10]) are binary (they can only handle a binary set V of initial values). Our (synchronous and asynchronous) solutions rely on multi-valued BA protocols. We therefore use the protocol of Turpin and Coan [18], which provides a reduction of multi-valued BA to a binary BA plus constant number of rounds.[2] The reduction protocol in [18] is designed for a synchronous network and can be extended to asynchronous network using protocols such "gradecast" of [14] or "A-cast" of [7] as its basic message distribution scheme.

The rest of this paper is organized as follows. In Section 2 we present our synchronous IC protocol. In Section 3 we extend it to asynchronous networks, and in Section 4 we summerize our conclusions.

2 A Synchronous IC Protocol

2.1 Preliminaries

We consider a synchronous system of n processors with unique identities in which every pair of processors is connected via a private line. To ensure correctness of our protocols we assume worst-case scenario. Thus, we assume an adversary A which, during the execution of our protocols, chooses online a subset of up to t processors to corrupt. When A corrupts a processor p, A completely controls p's behaviour. In addition, during an execution, A always has access to all the previous configurations of the system (that is, all the configurations of previous rounds) but A is oblivious to the current random choices made by every nonfaulty processor. For a complete and formal model of (synchronous and asynchronous) distributed system and a formal definition of the adversary and the byzantine model, the reader is

[1] Karlin and Yao extended (in an unpublished manuscript) the deterministic lower bound in [17] showing that even probabilistic BA protocols cannot tolerate more than $(n-1)/3$ faulty processors.

[2] Essentially, the idea of this reduction protocol is to agree (using the given binary BA protocol) if all nonfaulty processors have the same initial value or not.

referred to [12, 16, 3, 1, 8, 9].

The protocols we present in this section make extensive use of two protocols of Feldman and Micali [14] which we use as black-boxes. The first protocol is a constant time and a reasonably fair random Leader Election protocol. The second, is a Byzantine Agreement (BA) protocol. We now state Feldman and Micali's theorems regarding these Election and BA protocols.

Notation: We adopt the following functional notation to describe a protocol for a distributed system. We specify the common inputs (n and t are always among them), the local inputs for each processor, and the local outputs for each processor as a function of the inputs.

Consider protocol *Byzantine()* in [14]. The common inputs of this protocol are n and $t \leq (n-1)/3$, the local input for every processor p_i, is v_i, and the local output for every nonfaulty processor p_i is d_i, such that the *consistency* and *meaningfulness* conditions of the BA problem definition (Section 1) are satisfied. This protocol and the proof of the following theorem can be found in [14].

Theorem 2 (Feldman and Micali) *There exists an $(n-1)/3$-resilient Byzantine Agreement protocol such that the probability that the protocol will not terminate after k rounds is less than 2^{-k}.*

Note that due to the distributed nature of the problem, the notion of termination time is subjective to every processor. In other words, two processor might terminate their participation in the protocol in two different computational rounds. However, they cannot terminate too far apart.

Observation 3 *Protocol Byzantine() satisfies the following property:*

> **Property 1-Shift:** *If some nonfaulty processor is the first (or among the first) to achieve a Byzantine Agreement in some round r, then every other nonfaulty processor achieves agreement within the next round, $r+1$.*

It is interesting to note that all known randomized Byzantine Agreement protocols satisfy property 1-Shift.

Consider protocol *Election()* in [14]. This is a constant time, $(n-1)/3$-resilient Leader Election protocol with the following specifications. There are no local inputs and the local output for each (nonfaulty) processor is an ID of one processor. The proof of the following theorem can be found in [14].

Theorem 4 (Feldman and Micali) *There exists an* $(n-1)/3$*-resilient protocol, with common inputs* n *and* $t \leq (n-1)/3$ *and with no local inputs, that terminates in a constant number of rounds such that with probability at least 2/3, the local output for every nonfaulty processor is a unique index* i *of a random processor.*

Upon termination of protocol *Election()*, the local output for every nonfaulty processor is the identity of a (random) unique nonfaulty processor with probability at least 2/3. However, there is still a possibility that nonfaulty processors will not agree on the index of the same leader. In our solution we do not use *Election()* for achieving a Byzantine Agreement and therefore we modify protocol *Election()* as follows. Each time we use *Election()* we run *Byzantine()* on the local outputs of *Election()*. This ensures that the identity of the leader is always unique. Altogether, this slight modification of *Election()*, together with Theorem 4, yield the following corollary.

Remark: Although at least 2/3 of the processors are nonfaulty, since we assume an adaptive corruption model (where the adversary can corrupt a processor online) we cannot claim that the chosen index i is that of a nonfaulty processor with probability at least 2/3. Nevertheless, in applications of our leader election protocol we need the property that the chosen leader has been nonfaulty just before the invocation of the leader election protocol. This obviously true as stated in the following corollary.

Corollary 1 *There exists an* $(n-1)/3$*-resilient protocol, with common inputs* n *and* $t \leq (n-1)/3$ *and with no local inputs, that terminates in a constant expected number of rounds such that the local output for every nonfaulty processor is a unique identity* i*. With probability at least* $2/3$*,* i *is the identity of a random processor, which had been nonfaulty just before the application of the protocol.*

2.2 A Synchronous IC Protocol

In this section we present our IC protocol. The protocol produces N Byzantine Agreements in a constant expected number of rounds for all N and therefore, for $N = n$ we have a fast computation of the IC-vector.

Let us first overview the main idea. In the following discussion we informally refer to an entity, say B, as a Byzantine Agreement, meaning that there exists a corresponding vector v_B of length n, with a component for each processor local input. After the system has computed a value (using a BA algorithm with inputs according to v_B) we say that the system has *achieved* a Byzantine Agreement on the vector v_B.

Suppose that $B_1, B_2, \ldots, B_N$ are the agreements to be achieved. For every $1 \leq i \leq N$ we will concurrently run m independent BA applications, each with the same input vector v_{B_i}. Altogether we will concurrently run mN protocols (N "classes" each of multiplicity m). If m is sufficiently large, it is likely that after a small constant number of computational rounds, each nonfaulty processor will "have" at least one terminated protocol (and thus, a corresponding agreement value) for each of the N classes. Note, however, that two different (nonfaulty) processors might have different agreement values for the very same class (obviously, from two different BA applications). The reason is, firstly, that the result of a randomized BA protocol cannot be predicted deterministically (if the nonfaulty processors do not all start with the same input value), and secondly, because two different processors may terminate their participation for the same run in two different rounds.

The idea now is to let the processors choose a leader which will handle a coordination (so that all the nonfaulty processors will choose the same output for each class). Using protocol *Election()* (Corollary 1) with sufficiently high probability, a nonfaulty processor will be chosen as the leader and this leader will coordinate the rest of the nonfaulty processors to a successful termination. In case where the leader p_l is faulty, either the nonfaulty processors will force p_l to coordinate them appropriately (using the power of a Byzantine Agreement), or they will reject p_l and try their luck with a new election.

It turns out that it is sufficient to take $m = \log N$ as the multiplicity for each of the N classes (see Lemma 3). To facilitate the description of our protocols we use the following notations:

- Whenever processor p_i is instructed to *distribute* a value v, this means p_i should send v to every processor including itself. To simplify analyses, any count of how many processors sent a certain message to p_i will include p_i's message.
- When p_i is instructed to perform *Byzantine(v_i)*, it means that p_i should start running protocol *Byzantine()* with private input v_i (and common inputs as in the calling protocol).
- When p_i is instructed to perform *Byzantine()* in exactly r rounds, the private output v for p_i denotes the unique non-null value returned to p_i by *Byzantine()* if agreement was reached within the first r rounds. Otherwise, p_i assigns its private output to null ($\emptyset$).
- Every variable which appears in the protocol of p_i and which is not declared in the header of the protocol is a local variable of p_i.

- When we say that a nonfaulty processor was elected to be a leader we mean that this processor was nonfaulty up until the start of the leader election protocol (see the remark just before Corollary 1).

Consider protocol IC in Figure 1. In the correctness proof and in the analysis of Protocol IC we use the following definitions. Let $\mathcal{E}$ denote the event that a nonfaulty processor was chosen to be a leader (at the end of Epoch 3). Let $\mathcal{D}$ denote the event that for every nonfaulty processor p_i, $S_j \neq \emptyset$ for every j (in Epoch 2). If $\mathcal{D}$ occurred, every nonfaulty processor completed successfully (at least) one Byzantine Agreement for each of the N agreement classes in r rounds.

Lemma 1 *If $\mathcal{E}$ and $\mathcal{D}$ both occur at a certain iteration of Protocol IC, then the protocol successfully terminates at the end of the iteration.*

Proof. Since $\mathcal{E}$ occurred, a nonfaulty processor p_k was chosen for leadership. Since $\mathcal{D}$ occurred, it must be that p_k distributed some nonempty "proposal", $c_{l_1,1}, \ldots, c_{l_N,N}$ at the end of Epoch 2. It follows then that since for every $1 \leq j \leq N$, $c_{l_j,j} \in S_j$ (S_j local to p_k) and since *Byzantine()* satisfies the 1-Shift property, for every nonfaulty processor p_i and for all $1 \leq q \leq N$, $c_q \in \widehat{S}_q$ ($c_q, \widehat{S}_q$ are local to p_i). Therefore, in Epoch 4, every nonfaulty processor p_i, sets $term_i = 1$; therefore, since *Byzantine()* satisfies the meaningfulness condition it must be that $terminate_i = 1$ for every i and all the nonfaulty processors terminate successfully. ∎

Let $\overline{S}_j \leftarrow S_j \cup \widehat{S}_j \cup \widehat{\widehat{S}}_j$, where S_j, $\widehat{S}_j$ and $\widehat{\widehat{S}}_j$ are computed in Epoch 1.

Lemma 2 *A faulty leader (chosen at the end of Epoch 3) cannot cause a collective agreement (of all nonfaulty processors) about a final output vector which was not fully computed by some nonfaulty processor. That is, in any case (even if the leader is faulty), each component $d_{i,j}$, $j = 1, \ldots, N$, of the output vector of a nonfaulty processor p_i is in $\overline{S}_j$.*

Proof. Assume that a faulty processor p_k was chosen. Since in Epoch 4 every nonfaulty processors run *Byzantine()* with p_k's proposal as their private input, in Epoch 4 all nonfaulty processors hold the same vector E. Let us assume that there exist indices i and j such that p_i is nonfaulty and e_j is not in $\overline{S}_j$ (e_j and $\overline{S}_j$ are local to p_i). In this case, for every other nonfaulty processor, p_l, $e'_j \notin S'_j \cup \widehat{S'}_j$ (e'_j and $\widehat{S'}_j$ are local to p_l); otherwise, *Byzantine()* does not satisfy the 1-Shift property. Therefore, all nonfaulty processors will $term_i = 0$ and they all output $terminate_i = 0$ at the end of the epoch so they will all reject the faulty Byzantine proposal. ∎

Common Inputs: n, $t \leq (n-1)/3$, N, r, m
Local Inputs for Processor p_i: $v_{i,1}, \ldots, v_{i,N}$
Local Outputs for Processor p_i: $d_{i,1}, \ldots, d_{i,N}$

Epoch 1:
for $1 \leq l \leq m$, **for** $1 \leq j \leq N$
$u_{l,j} \leftarrow v_{i,j}$
for $1 \leq l \leq m$, **for** $1 \leq j \leq N$
run concurrently Byzantine($u_{l,j}$) exactly $r+2$ rounds in the following way:
Let $c_{l,j}$ denote the outputs after exactly r rounds.
Continue the computation for another one round and let $\widehat{c}_{l,j}$ denote the outputs.
Compute another round and let $\widehat{\widehat{c}}_{l,j}$ denote the outputs.
for $1 \leq j \leq N$
$S_j \leftarrow \{c_{l,j}\}_{l=1}^m$, $\widehat{S}_j \leftarrow \{\widehat{c}_{l,j}\}_{l=1}^m$, $\widehat{\widehat{S}}_j \leftarrow \{\widehat{\widehat{c}}_{l,j}\}_{l=1}^m$

Epoch 2:
if for every $1 \leq j \leq N$, $S_j \neq \emptyset$, **then**
for every $1 \leq j \leq N$, **choose** $c_{l_j,j} \in S_j$ **and distribute** $[c_{l_1,1}, c_{l_2,2}, \ldots, c_{l_N,N}]$
else distribute $\emptyset$

Epoch 3:
run Election(). Let k be the private output, denoting the identity of the leader p_k.

Epoch 4:
Let D denote the value received from p_k in Epoch 2.
Set it to $\emptyset$ if a propper message was not received.
(Note that if $D \neq \emptyset$, then $D = c_1, c_2, \ldots, c_N$ for some c_j.)
run Byzantine(D). Let $E = e_1, \ldots, e_n$ denote the private output.
if for every $1 \leq j \leq N$, $e_j \in S_j \cup \widehat{S}_j$ **then**
set $term_i \leftarrow 1$ **else** set $term_i \leftarrow 0$

Epoch 5:
run Byzantine($term_i$) and let $terminate_i$ denote the private output.
if $terminate_i = 0$ **then**
goto Epoch 1
else
for every $1 \leq j \leq N$ $d_{i,j} \leftarrow c_j$
return $d_{i,1}, \ldots, d_{i,N}$

Figure 1: Protocol IC

Lemma 3 *Let $\{X_{i,j}\}, i = 1, \ldots, N, \;\; j = 1, \ldots, \log N$, be independent random variables such that for every i, j, $X_{i,j} \in \{0, 1\}$ and $Pr[X_{i,j} = 1] \geq \frac{1}{2}$. Let $\{Y_i\}_{i=1}^{N}$, be a set of N independent random variables such that for all i, $Y_i = \sum_{j=1}^{log_2 N} X_j$.*

$$Pr[\forall 1 \leq i \leq N, \;\; Y_i \geq 1] \geq 1/e.$$

Proof. The $X_{i,j}$ are independent and therefore, for every $1 \leq i \leq N$

$$\Pr[Y_i = 0] \;\leq\; (1 - \frac{1}{2})^{\log N} \;=\; \frac{1}{N}.$$

The Y_i are independent; therefore:

$$\Pr[\forall i, \;\; Y_i \geq 1] \;=\; \prod_{i=1}^{N} \Pr[Y_i \geq 1] \;=$$

$$\prod_{i=1}^{N} (1 - \Pr[Y_i = 0]) \;\geq\; (1 - \frac{1}{N})^N \;\cong\; \frac{1}{e}.$$

■

Theorem 5 *For every $N, n, r \geq 1$ and $t < n/3$ Protocol IC produces N Byzantine Agreements in a constant expected number of rounds.*

Proof. Partial correctness follows from Lemma 2. We now prove that protocol IC terminates within constant number of rounds on the average. Lemma 1 says that if both $\mathcal{D}$ and $\mathcal{E}$ occur in a certain iteration, then the protocol will terminate successfully. It remains to show that the probability p that both $\mathcal{D}$ and $\mathcal{E}$ occur (in the same iteration) is sufficiently high. Since we run all parallel executions independently, by using Lemma 3 we easily get that

$$Pr[\mathcal{D} \text{ occurred within } r \text{ rounds}] \geq 1 - (1 - \frac{1}{e})^r.$$

Protocol *Election()* guarantees that for every iteration, $Pr[\mathcal{E} \text{ Occurred}] \geq 2/3$. Therefore,

$$p \geq \frac{2}{3}[1 - (1 - \frac{1}{e})^r].$$

We can now calculate r for which $p \geq 1/2$. We require that

$$\frac{2}{3}[1 - (1 - \frac{1}{e})^r] \geq \frac{1}{2}.$$

Solving for r we get that for any $r \geq 4$, $\mathcal{D}$ and $\mathcal{E}$ occur on the average within two iterations.

In every iteration of protocol IC, the protocol calls twice to *Byzantine()*, once to *Election()* and computes in additional $r+4$ rounds. Therefore, the total expected time $T \leq 2(8+2B+E)$

where B is the expected termination time of *Byzantine()* and E is the expected termination time of *Election()*. ∎

Note: The fast randomized synchronous Byzantine Agreement protocol we use introduces a slack of one round at the end of the protocol. Without any modifications this slack may grow when running a sequence of applications of the protocol. We are aware of the following two methods to resolve this problem.

- The first is to accommodate this increasing slack using an appropriate modification of the above Feldman-Micali protocols.[3] The slightly modified protocol remains correct and the expected slack in our protocol remains known and bounded.
- The second way is to replace the synchronous Feldman-Micali BA protocol with the synchronous application of the *asynchronous* BA protocols of Canetti-Rabin [10] (see Section 3) along with the following simple patch applied to these protocols: Upon agreement, each processor distributes an "End" message that includes the decision value and waits for $n - t$ such messages before starting the next application of the protocol. Any other processor that receives at least $t + 1$ such "End" messages at any stage of the protocol will also decide on the same value and send a similar "End" message. With this patch it is clear that there is at most one slack round between the correct processors, and the slack does not grow.

Corollary 2 *For a synchronous system of n processors, Protocol IC applied with $N = n$ and $m = \log n$ solves asynchronous Interactive Consistency within constant expected number of rounds and achieves optimal resiliency of $(n-1)/3$.*

3 An Asynchronous IC Protocol

In the synchronous model there is a natural identification of *absolute* time, as measured by a global clock; that is, the rth round occurs between the $(r-1)$st and the rth global clock pulses. In asynchronous systems, no such identification is possible. However, there are various meaningful ways to define asynchronous time. We use the one based upon *rounds of message exchanges*. An *asynchronous round* for a nonfaulty processor p_i is the time interval between p_i's request of information from other processors and the receipt of that information. (The computation time required by p_i once the information is received is also included in the

[3] It is easy to check that the Feldman-Micali BA protocol can operate if it starts with a known upper bound on the slack.

round.) We assume that an adversary schedules the arrival time of every message sent from every processor to every nonfaulty processor. It is only guaranteed that a message sent by a nonfaulty processor will *eventually* arrive upon its destination. Due to this limitation no processor can wait for a message sent by a specific processor. The reason is that one cannot tell the difference between a dead (faulty) processor and a very slow (nonfaulty) processor. As a consequence, given that every processor is supposed to distribute a certain message in a certain round, any processor cannot expect more than $n - t$ of those messages to arrive in the following round.

In this section we introduce an asynchronous IC protocol with optimal resiliency of $(n-1)/3$. We make use of the optimally-resilient asynchronous variants of *Election()* and *Byzantine()* of Canetti and Rabin [10]. [4] Denote by *A-Election()* and *A-Byzantine()* these protocols, respectively. Each of these protocols is $(n-1)/3$-resilient and satisfies all the properties mentioned in Theorem 2 and Corollary 1, respectively.

The presentation of our asynchronous IC solution is organized as follows. In Section 3.1 we introduce basic asynchronous broadcast and message dispersion protocols that will be used in our asynchronous protocols. The asynchronous IC protocol we design is based on a reduction to a new type of consensus protocol called *Selection*, which essentially, is stronger than a standard Byzantine agreement in the sense that the nonfaulty processors can guarantee an agreement on a non-default (non-null) value. In Section 3.2 we define this new type of "selection" consensus and construct a selection protocol that runs in constant expected number of rounds. This protocol has optimal resiliency of $(n-1)/3$. Then, in Section 3.3, we show how to reduce the IC problem to selection and obtain an optimally-resilient asynchronous IC protocol that runs in constant expected number of rounds.

3.1 Message Broadcast and Amplification Primitives

In this section we introduce two broadcast protocols that will be used in our asynchronous IC solutions. The basic protocol we employ is the known *asynchronous broadcast* protocol which was (to the best of our knowledge) first defined and constructed by Bracha [7]. Following Feldman [13] we call this protocol "A-cast". Using a slightly modified version of A-cast, which we call A-cast$^+$, we then construct a simultaneous broadcast "amplification" protocol called "*Spread* " that allows the set of nonfaulty processors to disperse their initial values so that there is a subset of at least $t+1$ nonfaulty processors that received a common set of initial values from at least $2t+1$ processors. Protocols A-cast and *Spread* are later

[4] In [10] these protocols are called *common-coin* and *BA*, respectively.

Common Inputs: $n, t \leq (n-1)/3$; k (the identity of the transmitter)
Local Inputs for Processor p_i: none
Additional Local Input for p_k: V
Local Output for Processor p_i: $Value_i$

Epoch 1: (p_k Only)
  distribute ["init",p_k,V]

Epoch 2: (Every Processor p_i)
  upon receiving ["init",p_k,v]-message for some v,
    distribute ["echo",v]

Epoch 3: (Every Processor p_i)
  upon receiving $2t+1$ ["echo",v]-messages for some v
    distribute ["ready",v]

Epoch 4: (Every Processor p_i)
  upon receiving $t+1$ ["ready",v]-messages for some v
    distribute ["ready",v]

Epoch 5: (Every Processor p_i)
  upon receiving $2t+1$ ["ready",v]-messages for some v
    $Value_i \leftarrow v$
    return $Value_i$

Figure 2: Protocol A-cast

used as primitives in the following sections. We use A-cast as the basic message distribution primitive and *Spread* to disperse the initial values. Protocol A-cast$^+$ is used as a subprotocol in *Spread* .

3.1.1 Protocol *A*-cast

Definition 1 *Let P be a protocol which includes as a common input k - the identity of a distinguished processor p_k (the* transmitter*). The transmitter has a private input value, V. The private output of every nonfaulty processor is $Value_i$. We say that P is an* asynchronous broadcast *protocol if the following hold:*

1. *If p_i and p_j terminate, then $Value_i = Value_j$.*
2. *If any nonfaulty processor terminates, then every other nonfaulty processor terminates.*
3. *If the transmitter is nonfaulty, then every nonfaulty processor p_i terminates with $Value_i = V$.*

Theorem 6 (Bracha) *Protocol A-Cast (Figure 2) is a constant time* $(n-1)/3$*-resilient asynchronous broadcast protocol.*

Proof. Let $n = 3t + 1$. We show that the three properties in Definition 1 are satisfied. Suppose p_i terminated after receiving $2t + 1$ ["ready",v]-messages in Epoch 5. At least $t + 1$ of these messages arrived from nonfaulty processors. Each of these processors has not sent a "ready" message with other values v', and therefore no processor will pass the threshold of $n - t = 2t + 1$ "ready" messages (in Epoch 3) for a value $v' \neq v$. Therefore, property 1 is satisfied. Since $t + 1$ of those "ready"-messages are from nonfaulty processors, every nonfaulty processor p_j will eventually receive these messages and the condition in Epoch 4 will be eventually satisfied. Therefore, every nonfaulty processor p_j will eventually distribute a "ready" message for v (if it has not done so before) in Epoch 4 and the condition in Epoch 5 will eventually be satisfied. Therefore, property 2 holds. It is easy to see, by inspection of the protocol, that property 3 holds. ∎

We say that a nonfaulty processor p_i *accepts* an *A-Cast* for some value V if it terminates executing protocol *A-Cast* with $Value_i = V$.

Perhaps the most important property of protocol A-cast is that if some nonfaulty processor accepts an *A-Cast* broadcast with some value V, it is guaranteed that all nonfaulty processors will eventually terminate the execution of A-cast accepting V. However, if a faulty processor is the transmitter, then A-cast may never terminate. Nevertheless, the use of A-cast is for message distribution so when we instruct every processor to A-cast some value we never expect a particular processor to accept more than $2t + 1$ of these A-casts.

3.1.2 Protocol *A*-cast$^+$

In this section we extend protocol A-cast so as to force a faulty processor p (who wishes to broadcast a message) to "commit" on its message in the sense that no future broadcasts can be completed succefully (i.e. accepted by a nonfaulty processor) by p if it later attempts to broadcast a contradictory message. The extended protocol, called A-cast$^+$, is used as a subprotocol of protocol *Spread* of Section 3.1.3. This commitment property achieved by A-cast$^+$ (and *Spread*) will later be needed in protocol *Select* of Section 3.2, which uses *Spread* as a subprotocol. Specifically, the special property achieved by this protocol (Lemma 5 below) is applied in the proof of Theorem 8.

Here is the idea. Consider an A-cast from a transmitter p_k for a value v. We add one additional epoch to protocol A-cast where we instruct each processor, before termination, to *accept* $2t + 1$ A-cast "commitment" message as a condition for termination. Each termi-

nated A-cast requires $2t+1$ "[ready,v]"-messages, at least $t+1$ of which are from nonfaulty processors. Therefore, if nonfaulty processors are instructed to avoid participating in contradictory A-casts of the transmitter p_k, no such contradictory A-casts can ever be accepted by nonfaulty processors.

Consider protocol A-cast$^+$ in Figure 3. This protocol is different from protocol A-cast in two points. First, in Epoch 5 processors are instructed to A-cast a "commit" message for the transmitter's value. The protocol terminates in Epoch 6 only after accepting $2t+1$ A-casts. Second, the protocol returns, in addition to the transmitter's value, a set of $2t+1$ processor indices called *Relay*. This set includes indices of processors which were committed to the value of the transnitter.

Common Inputs: $n, t \leq (n-1)/3$; k (the identity of the transmitter)
Local Inputs for Processor p_i**:** none
Additional Local Input for p_k**:** V
Local Output for Processor p_i**:** $Value_i$, $Relay_i$

Epoch 1: (p_k Only)
  distribute ["init",p_k,V]

Epoch 2: (Every Processor p_i)
  upon receiving ["init",p_k,v]-message for some v,
    distribute ["echo",p_k,v]

Epoch 3: (Every Processor p_i)
  upon receiving $2t+1$ ["echo",p_k,v]-messages for some v
    distribute ["ready",p_k,v]

Epoch 4: (Every Processor p_i)
  upon receiving $t+1$ ["ready",p_k,v]-messages for some v
    distribute ["ready",p_k,v]

Epoch 5: (Every Processor p_i)
  upon receiving $2t+1$ ["ready",p_k,v]-messages for some v
    A-cast ["commit",p_k,v]
Epoch 6: (Every Processor p_i)
  upon accepting $2t+1$ ["commit",p_k,v] A-casts
  from a set S of processors,
    $Value_i \leftarrow v$
    $Relay_i \leftarrow S$
    return $Value_i$, $Relay_i$

Figure 3: Protocol A-cast$^+$

It is easy to see, by inspection of the protocol, that protocol A-cast$^+$ is an asynchronous broadcast protocol. We omit the proof of the following lemma.

Lemma 4 *Protocol* A*-cast*$^+$ *(Figure 3) is a constant time* $(n-1)/3$*-resilient asynchronous*

broadcast protocol.

Lemma 5 *Let p_k be the transmitter of a message V in protocol* A-*cast*$^+$*. Upon termination of the protocol by one nonfaulty processor p_i, for each processor p in $Relay_i$, there exists a subset $C_{k,v}$ of (at least) $t+1$ nonfaulty processors all of which received a ["commit",p_k,v]-message from p.*

Proof. Assume that processor p_i terminated protocol A-cast$^+$. According to the protocol, p_i accepted the "commit" A-casts the $2t+1$ processors in $Relay_i$. By inspection of protocol A-cast, in order to accept an A-cast from a processor $p \in Relay_i$, p_i must receive "commit" messages from $2t+1$ processors. $C_{k,v}$ is the subset of (at least) $t+1$ nonfaulty processors among them. ∎

3.1.3 Protocol *Spread* - A Dispersion Primitive

In our asynchronous IC solution the basic "distribute" operation is replaced by the stronger A-cast asynchronous primitive. However, in order to distribute messages and achieve a large dispersion level (of common messages) a straightforward application of A-cast by itself is not sufficient. Consider a scenario where every nonfaulty processor has some *initial value* which is to be sent to every other nonfaulty processor. The maximal (and ideal) dispersion is the one in which every nonfaulty processor receives every nonfaulty initial value. In a synchronous network this goal is trivially achieved if every processor distributes its initial value. In the asynchronous network such an ideal dispersion is not always possible. Let $n = 3t+1$. Every nonfaulty processor can wait for no more than $n - t = 2t+1$ messages (A-casts) and t of these messages can be faulty. If p_i and p_j are nonfaulty, the adversary can cause p_i and p_j to receive the messages from different subsets of processors each of size $2t+1$, so the number of common messages they receive is at most $t+1$. Similarly, for 3 nonfaulty processors the number of common messages can be no more than 1. In general, if $n = kt+1$, for some positive integer k, the number of common messages received by d processors may be not more than $\max\{0, n-dt\}$. This means that unless $n = O(t^2)$, we cannot hope, using this straightforward approach, for a set of $O(t)$ nonfaulty processors which received $O(t)$ common messages.

In this section we construct a simple message dispersion protocol, based on A-cast (and A-cast$^+$). The new protocol, called *Spread* , can guarantee upon termination (within constant number of rounds) that a subset of $t+1$ nonfaulty processors received the messages of a subset of $2t+1$ processors (up to t of which my be faulty). The basic idea is as follows. Let $n = 3t+1$. Every processor p_i will A-cast its own initial value V_i. After *accepting*

$2t+1$ A-casts $V_{i_1}, \ldots, V_{i_{2t+1}}$ from $2t+1$ processors, $p_{i_1}, \ldots, p_{i_{2t+1}}$, processor p_i will A-cast the vector:

$$\langle (p_{i_1}, V_{i_1}), \ldots, (p_{i_{2t+1}}, V_{i_{2t+1}}) \rangle.$$

After accepting $t+1$ such vectors in which for every component (p_l, V_l), V_l was accepted (by p_i) from p_l, the protocol terminates. As we later show (Lemma 7) this is sufficient for generating the desired level of dispersion.

This is the basic idea. However, instead of using A-cast in the first round we use A-cast$^+$. This will ensure a commitment of processors to the value they want to distribute (and will be used later in protocol *Select* of Section 3.2).

Protocol *Spread* is given Figure 4.

Common Inputs: $n, t \leq (n-1)/3$
Local Inputs for Processor p_i**:** V_i
Local Output for Processor p_i**:** $\langle X_1, X_2, \ldots, X_n \rangle$

Epoch 1:
  A-cast$^+$ V_i

Epoch 2:
  wait until $2t+1$ values $V_{i_1}, \ldots, V_{i_{2t+1}}$ accepted (from $p_{i_1}, \ldots, p_{i_{2t+1}}$)
  A-Cast $\langle (p_{i_1}, V_{i_1}), \ldots, (p_{i_{2t+1}}, V_{i_{2t+1}}) \rangle$
    Participate in the A-Cast of a vector $u = \langle (p_{j_1}, U_{j_1}), \ldots, (p_{j_{2t+1}}, U_{j_{2t+1}}) \rangle$
    only if for every $1 \leq k \leq 2t+1$, U_{j_k} accepted from p_{j_k}.

Epoch 3:
  wait until $t+1$ vectors $u_{l_1}, \ldots, u_{l_{2t+1}}$ accepted (from $p_{l_1}, \ldots, p_{l_{t+1}}$) such that
    for every $1 \leq k \leq t+1$ if $u_{l_k} = \langle (p_{k_1}, U_{k_1}), \ldots, (p_{k_{2t+1}}, U_{k_{2t+1}}) \rangle$,
    and for every $1 \leq r \leq 2t+1, U_{k_r}$ accepted from p_{k_r}'s A-cast$^+$.
    for every $1 \leq l \leq n$
      if a value V was accepted from p_l via A-cast$^+$ **then**
        $X_l \leftarrow (V, Relay_{i,l})$
        (where $(V, Relay_{i,l})$ is the output of A-cast$^+$)
      else
        $X_l := \emptyset$
    return $\langle X_1, \ldots, X_n \rangle$

Figure 4: Protocol *Spread*

Lemma 6 *Protocol Spread terminates in a constant number of rounds.*

Proof. Clearly, X_l in Epoch 3 is well defined. The reason is that V was *accepted* from p_l's A-cast$^+$ but no two different values could be accepted from p_l (even if p_l was faulty). We claim that no deadlock can occur in Epoch 3. This is true because there exists a set S of

at least $2t+1$ nonfaulty processors such that every processor p_i in S will eventually A-cast a vector in which for every component (p, V), p_i *accepted* the A-cast$^+$ of V from p. If a nonfaulty processor *accepted* some A-cast$^+$, then it is guaranteed that every other nonfaulty processor will eventually *accept* the same A-cast$^+$. This means that every nonfaulty processor will eventually *accept* $t+1$ vectors (in Epoch 3) and there is no possibility of a deadlock. By inspection of the protocol it can be seen that it terminates in constant number of rounds. ∎

Lemma 7 *Let p_i be the first nonfaulty processor which has just terminated the protocol. At this stage, there exists a subset of $t+1$ nonfaulty processors all of which accepted the same $2t+1$ A-cast$^+$s of initial values.*

Proof. p_i is instructed to accept $t+1$ vectors. One of these vectors is surely from a nonfaulty processor p_j. Since p_i accepted u_j (via A-cast) from p_j, it follows that there are $2t+1$ processors which participated in this broadcast. Since every nonfaulty processor is instructed (in Epoch 2) to participate only in A-casts of vectors in which every component is *accepted*, it follows that there are $t+1$ nonfaulty processors which accepted the same $2t+1$ *A-Cast*s. ∎

3.2 A Selection Protocol

In this section we define and construct a new type of consensus protocol called "selection". Later, in the next section, we will provide an IC protocol by reducing the IC problem to selection and Byzantine agreements.

Definition 2 *Let P be a protocol. The common inputs of P are n and t; the local inputs for each processor p_i are V_i, an arbitrary value, and $Pred_i(x)$, a predicate; the local output is D_i. We say that P is a* Selection *protocol if the following holds:*
If for every nonfaulty p_i and p_j, $\models$[5] $Pred_j(V_i)$ and $V_i \neq \emptyset$, then:

1. *(Consistency) Upon termination, for every nonfaulty p_i and p_j, $D_i = D_j \neq \emptyset$.*
2. *(Selection) Upon termination there exists a nonfaulty processor p_k such that $\models Pred_k(D_k)$.*
3. *(Termination) The protocol terminates with probability* 1.

Essentially, a selection protocol is a Byzantine agreement protocol in which the "meaningfulness" condition (where, if all processors start with the same initial value, they terminate

[5]The notation "$\models P$" is the standard Mathematical Logic notation for "P is valid".

with that value) is replaced with the stronger "selection" condition in which if all nonfaulty processors start with initial values that satisfy their predicates, then the terminating value satisfies at least one nonfaulty predicate. To see that a selection protocol can achieve more than a standard Byzantine agreement, consider a scenario where each nonfaulty processor p_i starts the protocol with the predicate $Pred_i(x) \equiv [m \leq x \leq M]$. Suppose that all the nonfaulty processors start the protocol with different initial values that are all in the interval $[m, M]$. In this setting, when using a standard Byzantine agreement protocol, the nonfaulty processors will terminate with a null (default) value. However, when using a selection protocol it is guaranteed that the nonfaulty processors will terminate with agreement on a number in the prescribed interval as specified by the predicate. Note that when using a selection protocol the predicates (and initial values) can be computed online just before the invocation of the selection protocol.

In our analysis we will use the following simple observation.

Observation 7 *Protocol* A-Byzantine() *satisfies the following property: The only possible outcomes for nonfaulty processors are either null or private inputs of nonfaulty processors.*

It is worth noting that all known solutions to the Byzantine Agreement problem satisfy the above property. [6]

In the following presentation of protocol "Select" we made no effort to optimize the number of rounds in the presented selection. On the contrary, for the sake of clarity, we expanded it by a few more rounds than needed (e.g. the A-casts of Epochs 4 and 5 could be done concurrently). To facilitate the explanation we use the following notations and definitions.

- We say that a value (that a processor p_i attributes to some processor p_k) is *inconsistent* with another value (which p_i also attributes to p_k) if the values are both non-null and different.
- When instructing a processor to participate only in *admissible* broadcasts we mean that the processor should not participate in broadcasts of messages that contain information that is *inconsistent* with its database of all previously received values.
- In order to relate to the kth component of a vector v, we use standard array notation (i.e. $v[k]$).

[6] The authors have a simple method to extend every BA protocol to satisfy the above property. If the BA protocol terminates (on the average) within T rounds , then this extension terminates within $3T + 1$ rounds.

Common Inputs: n, $t \leq (n-1)/3$
Local Inputs for Processor p_i: V_i - an arbitrary value; $Pred_i(x)$ - a predicate
Local Outputs for Processor p_i: D_i

Epoch 1:
$U_i \leftarrow V_i$
run Spread$(U_i, Pred_i(x))$ and let $View_i$ denote the local output.

Epoch 2:
run A-Election(). Let k be the local output denoting the identity of the leader, p_k.

Epoch 3:
$U \leftarrow View_i[k]$ (i.e. the value' "related" to p_k upon termination of *Spread*).
if U was *accepted* during the run of Spread **and** $\models Pred_i(U)$ **then**
$R_i \leftarrow U$
$U_i \leftarrow$ ["deflected", $U, Relay_i$]
where $Relay_i$ is obtained by *Spread* (via A-cast$^+$) for the accepted broadcast of the leader p_k
else $R_i \leftarrow \emptyset$

run A-Byzantine(R_i). Let D_i denote the local output.
if $D_i \neq \emptyset$ **then**
return D_i

Epoch 4:
A-Cast $View_i$ ($View_i$ defined in Epoch 1).
Participate in *admissible* A-Casts only.

Epoch 5:
wait until $2t+1$ views, $View_{i_1}, \ldots, View_{i_{2t+1}}$ accepted.
A-Cast U_i

Epoch 6:
wait until $2t+1$ values accepted.
if a ["deflected",U',$Relay$]-message was accepted
and $p_k \in Relay$ (p_k is the leader from Epoch 2)
and there are $t+1$ processors in $Relay$ which have a non-null
value related to p_k in their accepted Views with
the same value U'
Comment: If several such messages were accepted with different
U' values take an arbitrary one
then $R_i \leftarrow U'$
else $R_i := \emptyset$

Epoch 7:
run A-Byzantine(R_i) and let D_i denote the local output.
if $D_i \neq \emptyset$ **then return** D_i
else
$U_i \leftarrow V_i$
goto Epoch 1

Figure 5: Protocol Select

Consider protocol "Select" in Figure 5. The protocol consists of 7 epochs. In Epoch 1 the processors broadcast their initial values using protocol Spread of Section 3.1.3. The properties of *Spread* guarantee that upon termination there is a core subset P_1 of nonfaulty processors all of which accepted a common set of broadcasts which were initiated by $2t+1$ processors (up to t of which can be faulty). This subset of $2t+1$ processors is denoted by P_2. In Epoch 2 the system runs the election protocol to choose a random leader, denoted p_k. As we later show, with sufficiently high probability the leader is nonfaulty and moreover, is in P_2, which means that its message has been succesully accepted by at least $t+1$ nonfaulty processors. In Epoch 3 the processors run a Byzantine agreement on the values they relate to the leader. If they agree on a non-null value (which means they agree on one of the nonfaulty initial values), they terminate. Otherwise, they continue the computation and in Epoch 4 they broadcast (A-cast) the individual views they each have on the vector of initial values (including the value they attribute to the leader). In Epoch 5, after waiting for acceptance of $2t+1$ views, they broadcast a list of processors which participated in the first leader broadcast (via *Spread*) in Epoch 1. This list contains $2t+1$ processors which were in fact commited to the leader's initial value in the sense that no such processor can participate in A-casts claiming a different value for the leader. In Epoch 6, after the processors receive $2t+1$ such lists they assign their revised estimate for the leader's initial value and (as we later show) this time it must be that if the leader is indeed nonfaulty and in P_2, then each nonfaulty processor gets to know the true initial value of the leader. Finally, in Epoch 7, the processors run a Byzantine agreement on their revised estimate and if they terminate with a non-null value, the protocols ends. Otherwise, the protocol is restarted.

Remark 1 *In Protocol* Select *of Figure 5 we use in Epoch 1 protocol* Spread *of Section 3.1.3. Since in a selection consensus our goal is generate terminal values that satisfy nonfaulty predicates, when we apply protocol* Spread *here, we assume that each processor p_i is instructed to participate only in* A*-casts which satisfy its own predicate $Pred_i$.*

Theorem 8 *Let* A-Byzantine() *be an asynchronous constant expected time $(n-1)/3$-resilient Byzantine Agreement protocol. Let* A-Election() *be an asynchronous constant expected time $(n-1)/3$-resilient Leader Election protocol. Protocol Select is an asynchronous constant expected time $(n-1)/3)$-resilient Selection protocol.*

Proof. Let $n = 3t+1$. We first prove partial correctness. Given that the preconditions hold (i.e. for every nonfaulty p_i and p_j, $\models Pred_j(V_i)$ and $V_i \neq \emptyset$), we first show that if protocol *Select* terminates, then the "Consistency" and "Selection" conditions (as in Definition 2) are satisfied.

Let us assume that protocol *Select* has just terminated. The Consistency condition holds since the only two possibilities to terminate (Epochs 3 and 7) are on outputs of *A-Byzantine()* and, as instructed by protocol, there is no termination on null values. Assuming termination and denoting the (unique) local outputs (from the call to *A-Byzantine()*) of the nonfaulty processors by D, we prove that the Selection condition holds.

There are two cases; either the protocol terminated at the end of Epoch 3, or at the end of Epoch 7. Consider the first case. Clearly, as the protocol satisfies Consistency, $D \neq \emptyset$. By Observation 7, *A-Byzantine()* generates terminal values which were inputs of nonfaulty processors. Therefore, D was the initial value of one of the nonfaulty processors, say p_i (so, according to the protocol, $D = R_i$, and R_i is local to p_i). But $R_i \neq \emptyset$ only if $\models Pred_i(R_i)$ (Epoch 3). Now consider the case where the protocol terminated at the end of Epoch 7. Clearly, $D = R_i$ for some nonfaulty p_i. Therefore, the value of R_i is a result of some assignment $R_i := U'$ (Epoch 6) and the condition in Epoch 6 guarantees that there exist at least $t + 1$ processors which have accepted U' as their *View* to the leader's value. This means that at least one nonfaulty processor p_j accepted U' and since the *View*s were obtained using protocol *Spread*, it must be that $\models Pred_j(U')$ (see Remark 1).

We now prove that if the preconditions hold (for every nonfaulty p_i and p_j, $\models Pred_i(V_j)$ and $V_i \neq \emptyset$), then the protocol is well defined and that it terminates (i.e. there are no deadlocks). First note that there cannot be deadlocks in Epochs 1-5 and in Epoch 7. In all these Epochs the processors run the protocols *Spread* , *A*-cast, *A-Election()* and *A-Byzantine()* which are deadlock free. We will show that there cannot be a deadlock in Epoch 6.

Let NF denote the subset of all nonfaulty processors. Let p_i be the the first nonfaulty processor terminated *Spread* (in Epoch 1), by Lemma 7 there exists a subset P_1 of NF, of size $\geq t + 1$, such that all members of P_1 accepted the same $2t + 1$ *A*-casts of initial values from (the same) $2t + 1$ processors. Let P_2 denote this subset of $2t + 1$ processors.

Let $\mathcal{E}$ denote the event where the leader chosen (in Epoch 2), p_k, is in $P_2 \cap NF$. Note that $|P_2 \cap NF| \geq t + 1$. Thus, if the event $\mathcal{E}$ occurred, the leader p_k is nonfaulty and at least $t + 1$ nonfaulty processors accepted the *A*-cast of the initial value of the leader. Let U denote the initial value of the leader. We will show that if $\mathcal{E}$ occurred then the protocol terminates.

Assume that $\mathcal{E}$ occurred. Without loss of generality set $P_1 = \{p_1, \ldots, p_{t+1}\}$. According to the protocol, in Epoch 3 every nonfaulty processor p_g which accepted a non-null value from p_k, assigns $R_g := U$. Every other nonfaulty processor p_o assigns R_o to null. This means that the nonfaulty processors start the *A-Byzantine()* of Epoch 3 with these two values alone, but

since protocol *A-Byzantine()* satisfies the property of Observation7, the only possibilities for a nonfaulty processor's local output (from *A-Byzantine()*) are $\emptyset$ and U. By our assumption (that $\mathcal{E}$ occurred) $U \neq \emptyset$ and satisfies all nonfaulty predicates and therefore, if U is the local output of the nonfaulty processors (which obtained using *A-Byzantine()*) then the protocol will terminate.

However, even if the event $\mathcal{E}$ occurred there is a possibility that no more than $t+1$ nonfaulty processors start the *A-Byzantine()* of Epoch 3 with U, and therefore, an outcome of $\emptyset$ is also possible. We now show that if this is the case, the computation performed during Epochs 4-6 guarantees that all nonfaulty processors start the Byzantine Agreement of Epoch 7 with the value U. That is, assuming that $\mathcal{E}$ occurred we now prove that for any nonfaulty p_i, R_i is set to U in Epoch 6. Recall that since $\mathcal{E}$ occurred, the leader $p_k \in P_2 \cap NF$. Consider a processor $p_l \in P_1$. p_l accepted U from p_k (via *Spread* in Epoch 1), and since protocol *Spread* uses A-cast$^+$ to distribute initial values, according to Lemma 5, for every member p_{l_j} of $Relay_l$, $1 \leq j \leq 2t+1$, there exists a subset W_{l_j} of NF containing $t+1$ nonfaulty "witnesses" each of which *accepted* a ["commit",p_k, U]-message (via A-cast) from p_{l_j}. Every nonfaulty processor is instructed to participate only in *admissible* A-Casts (of views). Therefore, all $t+1$ members of W_{l_j} will not cooperate with A-casts of ["deflected",U',$\cdot$]-messages with $U' \neq U$, whenever such A-casts are initiated by processors in $Relay_l$. Therefore such messages cannot be accepted (via A-cast) by any nonfaulty processors because the termination condition of A-cast requires $2t+1$ participants (so even if t faulty processors support such A-casts the avoidance of the $t+1$ members of W_{l_j} will prevent termination of such A-casts). Therefore any nonfaulty p_i can only assign R_i to U. On the other hand, among the members of $Relay_l$ there are at least $t+1$ nonfaulty processors which initiated an A-cast of a ["deflected",U,$\cdot$]-message, which will be accepted by all nonfaulty processors. Therefore, since in Epoch 6 p_i waits to accepts A-casts of $2t+1$ "deflect"-messages, at least one such A-cast must be from a member of W_{l_j}. This ensures that the "if" condition in Epoch 6 can be satisfied.

By inspection of the protocol it is clear that even if the event $\mathcal{E}$ did not occur, there will be no deadlock.

Finally, we prove that protocol *Select* terminates in constant expected number of rounds. Since *A-Election()* generates the each leader independently of previous iiterations, in every iteration $|P_2 \cap NF| \geq t+1$. Therefore, $\mathcal{E}$ occurs with probability at least 1/3. Protocol *Select* makes one call to *Spread* and two calls tp *A-Cast* which are both constant time protocols. In addition, *Select* makes one call to *A-Election* and one call to *A-Byzantine* and both are constant expected time protocols. Therefore, Protocol *Select()* terminates within constant expected number of rounds. ∎

3.3 Asynchronous IC Protocol

In this section we present our Asynchronous IC protocol. In this Protocol we still use the basic idea of running multiple instances for each of the n agreements (as in the synchronous case). However, in the asynchronous protocol we employ the selection protocol of Section 3.2. We use protocol *Select* twice. The first use of *Select* is for synchronization and the second is for selecting a unique and proper output vector.

Consider protocol Asynchronous IC in Figure 6. This protocol consists of 3 epochs. In Epoch 1 the processors initiate concurrent runs of mN *A-Byzantine()* instances (m for each class, as in the synchronous case). Immediately after they have one terminated run of *A-Byzantine()* for each class they stop this Epoch. In Epoch 2 we employ protocol *Select* to synchronize the processors on a round number in which each all nonfaulty processor have computed appropriate representatives for each of the classes (see details below). In Epoch 3 we again employ protocol *Select* to choose one appropriate vector of final values.

Theorem 9 *Let Select() be an asynchronous* $(n-1)/3$*-resilient constant expected time Selection protocol. For every* $N \geq 1$*, Protocol Asynchronous IC produces* N *Byzantine Agreements within a constant expected number of rounds and is* $(n-1)/3$*-resilient.*

Proof. We first prove that for every terminated run of Asynchronous IC, the following hold: (i) $R^\star$ is a round number in which every nonfaulty processor already computed a proposal for the output vector. (ii) V_P is a proper output vector (i.e. was properly computed by every nonfaulty processor).

To prove (i) we argue as follows. $R^\star$ is the output of protocol *Select* applied in Epoch 2. By definition, if the nonfaulty procesors start protocol *Select* while the precondition of this protocol holds (i.e. all nonfaulty predicates are satisfied by all nonfaulty initial values), then protocol *Select* generates a value which satisfies at least one nonfaulty predicate. Let us prove first that the precondition hold. Consider R_i in Epoch 1. For a nonfaulty p_i, R_i is the minimum round number where at least one Byzantine agreement is achieved for each of the n classes. Once this is achieved by one nonfaulty processor, say p_i, the rest of the nonfaulty processors achieve it after at most one round (recall property 1-Shift). Therefore, for all nonfaulty p_i and p_j, $|R_i - R_j| \leq 1$. This means that a nonfaulty predicate $Pred_j(X) \equiv [X$ is a round number and $|X - R'_j| \leq 1]$, of any nonfaulty p_j is satisfied by any R'_k value of any non-faulty p_k. (Note that $R'_i = R_i + 2$.) Therefore, the precondition of protocol *Select* holds.

Since $R^\star$ is the outcome of *Select*, there exists a nonfaulty processor p_i such that $\models Pred_i(R^\star)$ ($Pred_i()$ is the predicate of p_i as defined in Epoch 2). This means that $R^\star$ is a round number and $|R^\star - R'| \leq 1$ where R' is local to p_i and defined in Epoch 2. Therefore,

Common Inputs: n, $t \leq (n-1)/3$, N, m
Local Inputs for Processor p_i: $v_{i,1}, \ldots, v_{i,N}$
Local Outputs for Processor p_i: $d_{i,1}, \ldots, d_{i,N} \in V$

Epoch 1:
for $1 \leq l \leq m$, **for** $1 \leq j \leq N$
$u_{l,j} \leftarrow v_{i,j}$
for $1 \leq l \leq m$, **for** $1 \leq j \leq N$
run concurrently A-Byzantine($u_{l,j}$)
$c^r_{l,j} \leftarrow$ the local outputs of A-Byzantine($u_{l,j}$) in round r
$c^*_{l,j} \leftarrow \bigcup_r \{c^r_{l,j}\}$
when for every $1 \leq j \leq N$ there exists $1 \leq l \leq logN$ such that $c^*_{l,j} \neq \emptyset$:
$R_i \leftarrow$ the current round number
goto Epoch 2 (but continue every run of A-Byzantine concurrently).

Epoch 2:
$R'_i \leftarrow R_i + 2$
$Pred_i(X) \leftarrow [X$ is a round number and $|X - R'_i| \leq 1]$
run Select($R', Pred_i(X)$) and let R^* denote the local output.
if the current round number is greater than or equal to $R^* + 2$ **then**
terminate every run of A-Byzantine.
goto Epoch 3
else
continue every run of A-Byzantine
until round No. $R^* + 2$ **and goto** Epoch 3

Epoch 3:
$c_{l,j} \leftarrow \bigcup_{i=1}^{R^*} \{c^i_{l,j}\}$, $\widehat{c}_{l,j} \leftarrow \bigcup_{i=1}^{R^*+1} \{c^i_{l,j}\}$, $\widehat{\widehat{c}}_{l,j} \leftarrow \bigcup_{i=1}^{R^*+2} \{c^i_{l,j}\}$ }
(Note that $c_{l,j} \subseteq \widehat{c}_{l,j} \subseteq \widehat{\widehat{c}}_{l,j}$.)
for every $1 \leq j \leq N$
$S_j \leftarrow \{c_{l,j}\}_{l=1}^m$, $\widehat{S}_j \leftarrow \{\widehat{c}_{l,j}\}_{l=1}^m$, $\widehat{\widehat{S}}_j \leftarrow \{\widehat{\widehat{c}}_{l,j}\}_{l=1}^m$
for all $1 \leq j \leq N$
pick a value $c_{l_j,j} \in S_j$
$V_i \leftarrow \langle c_{l_1,1}, c_{l_2,2}, \ldots, c_{l_N,N} \rangle$
$Pred_i(X) \leftarrow [X =< c_1, c_2, \ldots, c_N >$ (i.e. X is an N-ary vector) and for every $1 \leq j \leq N$, $c_j \in S_j \cup \widehat{S}_j]$
run Select($V_i, Pred_i(X)$) and let V_P denote the local output.
(i.e. $V_P = \langle b_1, b_2, \ldots, b_N \rangle$ for some b_i.)
for all $1 \leq j \leq N$ $d_{i,j} \leftarrow b_j$
return $d_{i,1}, \ldots, d_{i,N}$

Figure 6: Protocol Asynchronous IC

since *A-Byzantine()* satisfies the 1-Shift property, it is guaranteed that every other nonfaulty processor has computed a proper output vector until a round number not exceeding $R' - 1$. But $R^\star \geq R' - 1$ so (1) has been proved.

We now consider the application of protocol *Select* in Epoch 3. We first prove that the precondition hold. The N components $c_{l_j,j}$ of the vector V_i (computed by p_i) are picked from sets S_j, $j = 1, \ldots, N$, which include results of Byzantine agreements up until round $R^\star$. We already know that $|R^\star - R'_i| \leq 1$, which implies that $R_i + 2 - R^\star \leq 1$ or, in other words, that $R^\star \geq R_i + 1$. Therefore all the $c_{l_j,j}$ values picked by p_i were computed by all other processors (property 1-Shift) and the precondition holds.

Since V_P is the outcome of protocol *Select()*, there exists some nonfaulty processor p_i such that $\models Pred_i(V_P)$. Hence, V_P is an N-ary vector such that for every $1 \leq j \leq N$, $b_j \in S_j \cup \widehat{S}_j$. This means that V_P was fully computed by p_i until round $R^\star + 1$, but since *A-Byzantine()* satisfies the 1-Shift property and since every other nonfaulty processor computed until round number $R^\star + 2$, it is guaranteed that V_P was fully computed by every nonfaulty processor. This proves (ii).

Next we argue that Protocol Asynchronous IC terminates in constant expected time. By claims (i) and (ii) above, and by the properties of protocol *Select*, both invocations of *Select* terminate within expected constant number of rounds. The now proof easily follows from Lemma 3. ∎

Corollary 3 *For an asynchronous system of n processors, Protocol Asynchrounous IC applied with $N = n$ and $m = \log n$ solves asynchronous Interactive Consistency within constant expected number of rounds and achieves optimal resiliency of $(n-1)/3$.*

4 Conclusions

We presented optimally-resilient IC protocols for both synchronous and asynchronous distributed systems. Both protocols terminate in constant expected number of rounds. We also defined and constructed a new type of consensus protocol called "Selection". A selection protocol is stronger than a Byzantine agreement in the sense that it allows for agreements on non-default values. This protocol can be of independent interest for coordination and synchronization purposes.

A simple (but important) application of our protocols is a constant expected time simulation of *broadcast channel*s. If a processor p transmits a message m over a broadcast channel,

then every other nonfaulty processor receives m (and the identity of p). Using our protocols it is trivial to simulate n concurrent transmissions over broadcast channels. One broadcast is exactly equivalent to one Byzantine Agreement. The only necessary assumptions are that the communication lines are private and that an appropriate bound on the number of faulty processors holds.

Corollary 4 *For any n and $t \leq (n-1)/3$, a synchronous or asynchronous complete network of n processors with private lines can simulate broadcast channels in constant expected time.*

To the best of our knowledge this is the only known method to simulate broadcast channels in constant expected time, with high resiliency and with no cryptographic assumptions.

Acknowledgements

We thank the anonymous referees for their valuable comments that greatly improved the content and presentation. We also wish to thank Ran Canetti and Juan Garay for helpful discussions and comments.

References

[1] H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.

[2] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In *PODC*, pages 201–209, 1989.

[3] D. Beaver. Secure multi-party protocols and zero-knowledge proof systems tolerating a faulty minority. *J. of Cryptology*, 4:75–122, 1991.

[4] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, pages 503–513, 1990.

[5] M. Ben-Or. Fast asynchronous Byzantine agreement. In *PODC*, pages 149–151, 1985.

[6] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, 1988.

[7] G. Bracha. An asynchronous $(n-1)/3$-resilient consensus protocol. In *PODC*, pages 154–162, 1984.

[8] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. of Cryptology*, 13(1):143–202, 1991. online version at http://philby.ucsd.edu/cryptolib/1998/98-18.html.

[9] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[10] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *STOC*, pages 42–51, 1993. Updated version: http://www.research.ibm.com/security/cr-ba.ps.

[11] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19, 1988.

[12] B. Chor and C. Dwork. Randomization in Byzantine agreement. In S. Micali, editor, *Advances in Computing Research Vol. 5: Randomness and Computation*, pages 443–497. 1989.

[13] P. Feldman. *Optimal Algorithms for Byzantine Agreements*. PhD thesis, MIT, 1988.

[14] P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997. A preliminary version of this paper appeared at STOC 1988.

[15] S. Micali and T. Rabin. Collective coin tossing without assumptions nor broadcasting. In *CRYPTO*, pages 253–266, 1990.

[16] S. Micali and P. Rogaway. Secure computation. In *CRYPTO*, pages 392–404, 1991.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):191–205, 1980.

[18] R. Turpin and B.A. Coan. extending binary byzantine agreement to multivalued byzantine agreement. *Information Processing Letters*, 18:73–76, 1984.