# Asynchronous Byzantine Consensus with 2f+1 Processes*

Miguel Correia
Universidade de Lisboa
Faculdade de Ciências
Lisboa, Portugal
mpc@di.fc.ul.pt

Giuliana S. Veronese
Universidade de Lisboa
Faculdade de Ciências
Lisboa, Portugal
giuliana@lasige.di.fc.ul.pt

Lau Cheuk Lung
Dep. Informática Estatíst., CT
Univ. Federal Santa Catarina
Florianopolis, Brazil
lau.lung@inf.ufsc.br

## ABSTRACT

Byzantine consensus in asynchronous message-passing systems has been shown to require at least $3f + 1$ processes to be solvable in several system models (e.g., with failure detectors, partial synchrony or randomization). Recently a couple of solutions to implement Byzantine fault-tolerant state-machine replication using only $2f + 1$ replicas have appeared. This reduction from $3f + 1$ to $2f + 1$ is possible with a hybrid system model, i.e., by extending the system model with trusted/trustworthy components that constrain the power of faulty processes to have certain behaviors. Despite these important results, the problem of solving Byzantine consensus with only $2f + 1$ processes is still far from being well understood. In this paper we present a methodology to transform crash consensus algorithms into Byzantine consensus algorithms with different characteristics, with the assistance of a reliable broadcast primitive that requires trusted/trustworthy components to be implemented. We exemplify the methodology with two algorithms, one that uses failure detectors and one that is randomized. We also define a new flavor of consensus and use it to solve atomic broadcast, showing the practical interest of the transformations.

## Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems*

## General Terms

Algorithms, Performance

## Keywords

Distributed algorithms, Consensus, Byzantine fault tolerance

*This work was partially supported by the EC through Alban scholarship E05D057126BR, and by the FCT through the Multiannual and the CMU-Portugal Programmes, the project PTDC/EIA-EIA/100581/2008 (REGENESYS) and the project PTDC/EIA-EIA/100894/2008 (DIVERSE).


*SAC'10* March 22-26, 2010, Sierre, Switzerland.


## 1. INTRODUCTION

*Consensus* is an important distributed computing problem, both in theoretical and practical terms. The problem consists in making a set of processes to agree on one of the values that each one of them proposes. From a theoretical point of view, consensus has been used to show important impossibility and possibility results, the most well-known of which is probably the Fischer-Lynch-Paterson (FLP) impossibility of solving consensus deterministically in an asynchronous system if one process can fail [13]. From a practical point of view, the problem has been shown to be equivalent to several distributing computing problems [17, 8], so an implementation of a consensus algorithm can be used as a building block of distributed systems [16].

Distributed algorithms depend strongly on the system model considered. A realistic model for many of the current large-scale, open distributed systems is the *asynchronous Byzantine message-passing* system model. The time model is asynchronous in the sense that no bounds are assumed for the communication and processing times. The fault model is Byzantine, meaning that some of the processes can fail arbitrarily, even maliciously. The impossibility of solving consensus deterministically in this system model comes trivially from the FLP impossibility result, but the problem is solvable in several variations of this basic system model.

Consensus in the asynchronous Byzantine message-passing model has been shown to require $n \geq 3f+1$ processes, where $f$ is the maximum number of faulty processes, to be solvable in several variations of the basic system model, e.g., with failure detectors [10, 1][1], partial synchrony [12] or randomization [3]. Reducing the ratio $n/f$ is important theoretically, but also in practice as reducing the number of processes/processors has an impact on the cost of a real system.

Recently a few solutions to implement Byzantine fault-tolerant *state-machine replication* using only $n \geq 2f+1$ replicas have appeared [7, 5]. This reduction from $3f+1$ to $2f+1$ is possible with a hybrid system model, i.e., by extending the asynchronous Byzantine system model with components that can not fail in a Byzantine way. These trusted/trustworthy components constrain the power of the adversary in the sense that the services they provide can not be corrupted and become faulty. We call these components simply *wormholes*

[1]Baldoni et al. present a sophisticated algorithm that assumes $f \leq min(\lfloor (n-1)/2 \rfloor, C)$, where $C$ is the maximum number of faulty processes allowed by the certification algorithm [1]. However, as they point out, "known certification techniques assume $n - C = \lceil \frac{2n+1}{3} \rceil$." This means that in practice their algorithm also requires $n \geq 3f + 1$.

using Verissimo's nomenclature [22]. Systems with wormholes are no longer homogeneous but hybrid: most of the system is still asynchronous Byzantine but the wormhole is trusted/trustworthy by construction.

State machine replication consists in replicating a service in a set of $n$ servers, $f$ of which may be faulty. Correia et al. use a wormhole called trusted timely computing base (TTCB) to help define an order for the execution of the clients' requests with only $2f + 1$ servers [7]. The TTCB defines an order for a client's request when $f + 1$ servers show it that they have the request. More recently, Chun et al. used an attested append-only memory (A2M) abstraction (or wormhole) with the same purpose [5]. A2M forces the servers to commit to a monotonically increasing sequence of operations.

Despite these important results, the problem of solving asynchronous Byzantine consensus with only $2f+1$ processes is still far from being well understood. There are several reasons for this: the related works that we cited solve consensus but have the solution for this problem submerged in the complications of a larger problem (state machine replication); they are based on special-purpose components (TTCB, A2M) that researchers are not familiarized with.

The main objective of this paper is to contribute to a better understanding of the problem of consensus with only $2f+1$ processes. To reach this objective, the paper presents *a methodology to transform asynchronous consensus algorithms that tolerate crash faults and require* $2f + 1$ *processes, into similar algorithms that tolerate Byzantine faults also with* $2f + 1$ *processes*. The paper demonstrates the methodology with two previously existing crash fault-tolerant consensus algorithms: an algorithm by Mostefaoui and Raynal that uses failure detectors to circumvent FLP [19] and a probabilistic algorithm by Ben-Or that uses randomization with the same purpose [2]. The methodology to do this modification, which is not necessarily generic, consists in enhancing the algorithm with a set of mechanisms that constrain the power of faulty processes, allowing the algorithm to reach consensus even if there is an adversary that tries to break the algorithm's properties.

The idea of modifying crash into Byzantine fault-tolerant algorithms, or *improving the fault tolerance*, was previously explored by Neiger and Toueg [20], and Coan [6]. The former present two transformations for synchronous systems: one from crash to omission faults and another from omission faults to Byzantine faults. Coan presents a compiler for asynchronous algorithms that transforms crash fault-tolerant algorithms into Byzantine fault-tolerant algorithms, just like our methodology, but considers only the case of approximate agreement and does not provide Byzantine fault-tolerant algorithms for $2f + 1$ processes, which is the main purpose of the present paper.

Like previous works, to solve asynchronous Byzantine consensus with $2f + 1$ processes we need trusted / trustworthy components, or wormholes, that provide certain incorruptible services. However, we use the abstraction provided by the wormhole to obtain a communication primitive commonly used in distributed computing: *reliable broadcast*. This primitive ensures that all processes *(i)* deliver the same messages and *(ii)* deliver all messages sent by correct (i.e., non-faulty) processes. More precisely, we present a reliable broadcast algorithm that imposes no bounds on the number of faulty processes, unlike previous existing algorithms that require $n \geq 3f + 1$ [3].

Using a $2f + 1$ reliable broadcast to solve $2f + 1$ Byzantine consensus is interesting for two reasons. First, it tackles the difficulty of understanding how a wormhole assists in solving a distributed computing problem, by using it to solve a well-known problem, reliable broadcast. Second, it is important to understand that it is possible to solve $2f + 1$ Byzantine consensus by relying only on a $2f + 1$ reliable broadcast and no other "unusual" component (e.g., no other component that needs a wormhole to be implemented).

The transformation methodology also requires a *muteness failure detector*, which detects if a process apparently stopped following the algorithm [11]. The consensus algorithms presented are *indulgent* [15], in the sense that they not violate their safety properties no matter the behavior of the failure detector, which is only needed to ensure termination.

The consensus flavor we solve satisfies a weak validity property. Therefore, we introduce a new flavor of consensus, *endorsement consensus*, and use it to solve atomic broadcast with only $2f + 1$ processes, thus showing that our consensus algorithms are useful to solve a distributed computing problem with practical interest.

The contributions of the paper are mainly the following: from a *theoretical* point of view, it shows that a $2f+1$ reliable broadcast primitive is enough to solve $2f + 1$ asynchronous Byzantine consensus, with the assistance of a muteness failure detector; from a *practical* point of view, it presents a methodology to transform asynchronous crash consensus algorithms into asynchronous Byzantine consensus algorithms with different characteristics keeping the number of processes of $n \geq 2f + 1$; this reduces the number of processes needed to tolerate the same number of faulty ones that is $n = 3f + 1$ in asynchronous Byzantine systems; the paper also presents two $2f + 1$ asynchronous Byzantine consensus algorithms.

# 2. PRELIMINARIES

## 2.1 Asynchronous Byzantine System Model

The system is composed by a set of $n$ processes $\Pi = \{p_1, p_2, ...p_n\}$. A process is said to be *correct* if it follows its algorithm, otherwise it is said to be *faulty*. Faulty processes can deviate from the algorithm arbitrarily, i.e., we assume the existence of *Byzantine faults* [18]. However, no more than $f = \lfloor \frac{n-1}{2} \rfloor$ can be faulty (in the tight case $n = 2f + 1$).

Processes communicate by message-passing. Every pair of processes is linked by an authenticated reliable channel, which does not allow the creation, modification or dropping of messages. In a malicious environment this involves either physically secure communication channels or the use of cryptographic mechanisms, which requires the additional assumption of a computationally bounded adversary.

The system is asynchronous, which means that there are no bounds on the processing times or communication delays. However, we assume the existence of failure detector modules in each of the processes. Failure detectors (FDs) give hints about faulty processes. The original FDs were used to suspect that processes crashed [4]. In this paper we consider *muteness failure detectors*, which suspect that a process is *mute*, either because it crashed or is Byzantine and stopped sending messages according to the algorithm [11]. Unlike crash FDs, muteness FDs depend on the algorithm, $\mathcal{A}$.

We consider a class of muteness FDs that is inspired in Chandra and Toueg's eventually perfect FD, which we call *eventually perfect muteness FD*, $\Diamond\mathcal{MP}_{\mathcal{A}}$. Failure detectors of

this class satisfy the following properties:

- *Mute strong $\mathcal{A}$-completeness.* Eventually every process that is mute to any correct process $p$ is permanently suspected by $p$.
- *Eventual strong $\mathcal{A}$-accuracy.* There is a time after which correct processes are not suspected by any correct process.

This FD is stronger than FDs used in previous Byzantine consensus algorithms [11, 1], but the two properties are satisfied in partition-free partially synchronous systems [12]. Doudou et al. provide an implementation $\mathcal{I}_{\mathcal{D}}$ of the muteness failure detector $\Diamond\mathcal{M}_{\mathcal{A}}$ that they introduce [11]. Although that FD satisfies only eventual weak $\mathcal{A}$-accuracy, it is simple to show that $\mathcal{I}_{\mathcal{D}}$ is also an implementation of $\Diamond\mathcal{MP}_{\mathcal{A}}$ in the same system model.

## 2.2 Reliable Broadcast

The basic system model of the previous section is extended with a reliable broadcast primitive. The reliable broadcast problem consists essentially in making all correct processes *deliver* the same messages [3]. Furthermore, if the process that *broadcasts* the message is correct, then all correct processes deliver the message, and no two messages with the same identifier are delivered by any correct process. Formally, a reliable broadcast algorithm can be defined in terms of the following properties [17] (we consider that the sender also delivers the messages it broadcasts):

- *RB1 Validity.* If a correct process broadcasts a message $m$, then some correct process eventually delivers $m$.
- *RB2 Agreement.* If a correct process delivers a message $m$, then all correct processes eventually deliver $m$.
- *RB3 Integrity.* For any identifier $id$ and sender $p$, every correct process $q$ delivers at most one message $m$ with identifier $id$ from sender $p$, and if $p$ is correct then $m$ was previously broadcast by $p$.

Bracha presented a reliable broadcast algorithm that needs $n \geq 3f+1$ processes [3]. A proof that $3f+1$ is the minimum number of processes was provided by Toueg [21].

Consider that there is a set of trusted/trustworthy wormholes $\Upsilon = \{w_1, w_2, ...w_n\}$ and that process $p_j$ has access exclusively to wormhole $w_j$. Each wormhole $w_j$ has a public-private key pair $(K_{uj}, K_{rj})$. The private key $K_{rj}$ is known only by $w_j$ and is used to produce digital signatures. Every correct process knows the correct public key $K_{uj}$ of every wormhole $w_j$. The wormholes provide a single service that can be abstracted as a function that is called by the processes (for wormhole $w_j$): $\sigma \leftarrow sign_j(id, msg)$ . The function takes as parameters a message identifier $id$ and a message $msg$. It returns either the signature $\sigma \in \mathcal{S}$ of $(id, msg)$ or $\perp \notin \mathcal{S}$, where $\mathcal{S}$ is the set of possible signatures. The signature is returned if $id > id'$, where $id'$ is the identifier given as parameter in the previous call to the function; otherwise $\perp$ is returned.

This service is simple but it precludes a faulty process from obtaining two different messages with the same identifier correctly signed. Algorithm 1 uses this service to solve reliable broadcast with any number of faulty processes. The wormhole can be implemented inside a secure coprocessor, a smartcard or another hardware board. Further discussion on the implementation of wormholes can be found in papers on the topic, e.g., [7, 5].

The reliable broadcast algorithm is similar to the classical crash fault-tolerant reliable broadcast algorithm [17]. In relation to Bracha's algorithm, it has one less communication step due to the use of the wormhole and requires no bounds on the number of faulty processes. The algorithm is requested to *broadcast* a message by calling RELIABLE_BROADCAST$(id, msg)$ (first line) and a message is *delivered* when DELIVER$(j, id, msg)$ is called by the algorithm (line 3, 6). Basically the sender sends the message to all processes and all processes send an ECHO message also to all processes. The wormhole is used to prevent a faulty sender from sending two different messages with the same identifier as explained above. Function *verify*$(id, msg, \sigma, K_{uj})$ verifies if the signature $\sigma$ was obtained with message $(id, msg)$ and key $K_{rj}$ (line 4).

**Algorithm 1** Reliable broadcast algorithm (at process $p_i$)

**Function** RELIABLE_BROADCAST$(id, msg)$

**Task T1:**
1: $\sigma \leftarrow sign_i(id, msg)$
2: $\forall j \neq i$ : SEND INITIAL$(i, id, msg)_\sigma$ to $p_j$
3: DELIVER$(i, id, msg)$

**Task T2:** {execute only once per message broadcast}
4: **when** (message INITIAL$(j, id, msg)_\sigma$ or ECHO$(j, id, msg, \sigma)$ is received) and (verify$(id, msg, \sigma, K_{uj})$) **do**
5: $\quad \forall k \neq j$ : SEND ECHO$(j, id, msg, \sigma)$ to $p_k$
6: $\quad$ DELIVER$(j, id, msg)$
7: **end when**

A proof of correctness of the algorithm is provided in the extended version of the paper [9].

# 3. METHODOLOGY AND CONSENSUS

Informally, consensus is the problem of making a set of processes to agree on a value. A process $p$ is said to *propose* a value $v \in \mathcal{V}$ for an execution of the consensus algorithm when it calls *_2FBC_CONSENSUS$(v)$ (with * equal to MR or BO). The process is said to *decide* a value $v$ when the algorithm calls DECIDE$(v)$. There are several definitions of Byzantine consensus in the literature. In this section we consider the following definition, which is the most common for crash consensus [17, 19, 14], and is also much used for Byzantine consensus [10, 1]. *Byzantine asynchronous multi-valued consensus* is defined in terms of the following properties:

- *MVC1 Validity.* If a correct process decides $v$, then $v$ was proposed by some process.
- *MVC2 Agreement.* No two correct processes decide differently.
- *MVC3 Termination.* Every correct process eventually decides.

This section presents the methodology to transform crash consensus algorithms into Byzantine consensus algorithms. The section starts by introducing the methodology with the Mostefaoui and Raynal's crash fault-tolerant consensus algorithm (*MR_Consensus* for short) [19].

## 3.1 Mostefaoui and Raynal's Algorithm

Algorithm 2 is the modified algorithm, MR_2FBC. Like the original algorithm, MR_Consensus [19], it is based on a rotating coordinator. Each round (lines 3-13) one of the processes is selected to be the coordinator (line 4) and tries to impose its estimate as the decision (line 5). Each round has two

phases. In the first (lines 5-7), the coordinator disseminates a PHASE1 message with its estimate of the value to be decided. In the second phase (lines 8-12), each process disseminates a PHASE2 message with the estimate of the coordinator or $\perp \notin \mathcal{V}$. If a correct process receives $n-f$ PHASE2 messages with the same value, it decides this value and disseminates this decision using a DECISION message (line 11).

**Algorithm 2** MR_2FBC Byzantine consensus algorithm (code for process $p_i$)

**Function** MR_2FBC_CONSENSUS($v_i$)

**Task T1:**

1: $r_i \leftarrow 0$ {round number}
2: $est_i \leftarrow v_i$ {current estimate of the value to be decided}
3: **while** true **do**
4: $c_i \leftarrow (r_i \ mod \ n) + 1$; $r_i \leftarrow r_i + 1$ {$c_i$ = coordinator}
{——— phase 1: coordinator to all ———}
5: **if** $(c_i = i)$ **then** RELIABLE_BROADCAST PHASE1$(r_i, est_i)$ **end if**
6: **wait until** (message PHASE1$(r_i, -)$ is received from $p_{c_i}$ or $p_{c_i}$ is suspected by $p_i$'s FD module)
7: **if** (*valid* message PHASE1$(r_i, v)$ received from $p_{c_i}$) **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **end if**
{——— phase 2: all to all ———}
8: RELIABLE_BROADCAST PHASE2$(r_i, aux_i)$
9: **wait until** (*valid* messages PHASE2$(r_i, -)$ are received from at least $n-f$ processes) and ($\forall j$ : *valid* message PHASE2$(r_i, -)$ is received from $p_j$ or $p_j$ is suspected by $p_i$'s FD module)
10: $\forall j$ : **if** (*valid* message PHASE2$(r_i, v)$ received) **then** $R_i[j] \leftarrow v$ **else** $R_i[j] \leftarrow \perp$ **end if**
11: **if** $(\exists v \neq \perp : \#_v(R_i) \geq n-f)$ **then** $est_i \leftarrow v$; $\forall j \neq i$ : SEND DECISION$(r_i, est_i)$ to $p_j$; DECIDE$(est_i)$ **else**
12: **if** $(\exists v \neq \perp : \#_v(R_i) \geq n-2f)$ **then** $est_i \leftarrow v$ **end if**
**end if**
13: **end while**

**Task T2:**

14: **when** *valid* message DECISION$(r, est)$ is received **do** {no need of $f+1$ messages due to the validation mechanism}
15: $\forall j \neq i$ : SEND DECISION$(r, est)$ to $p_j$; DECIDE$(est)$
16: **end when**

Some of the modifications to MR_Consensus are clear: reliable channels are substituted by *authenticated reliable channels* and message disseminations are substituted by the *reliable broadcast primitive* (lines 5, 8). Notice that the identifier (*id*) of a message disseminated is composed by the message type (e.g., PHASE1) and the round number.

Another modification is that we use the *message validation mechanism* introduced by Bracha [3] to prevent some of the attacks that might be done by faulty processes. In several places the algorithm only takes into account messages that are *valid* (lines 7, 9, 10, 14). Informally, a message is said to be valid if it is justified by the messages previously received by the process. For instance, line 14 seems to be wrong since the process $p_i$ should require DECISION messages from $f+1$ processes, for at least one to be sent by a correct process, before deciding in line 15. However, the validation mechanism ensures that the condition in that line is true only if the DECISION message might have been sent by a correct process, i.e., if $p_i$ received $n-f$ PHASE2 messages with the same estimate *est* (lines 9-11).

The definition of *valid* message is identical to Bracha's. Each step $k$ of the algorithm has the following basic format: the process disseminates a message to all other processes, waits for a set $S$ of messages from the other processes, and obtains the content of the next message using a *protocol function* $F(k, S)$. A message that is delivered by the reliable broadcast primitive (messages PHASE1 and PHASE2) or by a reliable channel (messages DECIDE) at step $k$ is called a *k-message*. Each process $p_i$ maintains a set of messages $VALID_i$ such that:

- $VALID_i^1 = \{$ delivered *1-messages* $\}$
- for $k > 1, m_j^k \in VALID_i^k$ if there exist $n-f$ *(k-1)-messages* $m_1, ..., m_{n-f}$ such that $m_j^k = F(k, \{m_1, ...m_{n-f}\})$

The main difference from MR_2FBC to MR_Consensus is line 9. In MR_Consensus, processes *wait until* they receive messages from $n-f$ processes (line 8 of Fig. 1 at [14]). Clearly it is not possible to block waiting for more messages as $f$ processes can be faulty. However, there is an important difference between the crash and the Byzantine fault models: while in the crash fault model (thus in MR_Consensus) all of those $n-f$ messages are sent by processes that follow the algorithm, in the Byzantine fault model (thus in MR_2FBC) $f$ of those messages can be sent by faulty processes. In the worst case, with $n = 2f+1$ and $f$ Byzantine processes, in every round that set of $n-f$ messages contains $f+1$ messages, $f$ of which sent by Byzantine processes[2]. The behavior of these $f$ faulty processes is constrained by the message validation mechanism, but they can do a simple attack that is undistinguishable from correct behavior: to send always $\perp$ as their estimate, pretending that their FD modules suspect of the coordinator (lines 6-8).

To deal with this problem, line 9 must "know about" *all* processes before continuing. More precisely, line 9 waits for messages from $n-f$ processes, but also either for messages or to suspect of the rest of the processes. This ensures that eventually $p_i$ receives messages from all correct processes, as there is a time after which correct processes are not suspected by any correct process (eventual strong $\mathcal{A}$-accuracy). This is also the reason why we need a stronger FD than previous Byzantine consensus algorithms, that require only eventual *weak* $\mathcal{A}$-accuracy [11, 1]. While those algorithms require only that the coordinator is eventually not suspected, MR_2FBC requires that eventually no correct process is suspected, i.e., eventual *strong* $\mathcal{A}$-accuracy.

Recall that the protocol function $F(k, S)$ is the function used to obtain the next message to be sent. The function comes trivially from the algorithm pseudo-code. The notion of step used to define the protocol function in this case is a phase. The formal specification of the function can be found in the pseudo-code itself. For instance, to obtain a PHASE2 message, if $p_i$ received a valid PHASE1$(r_i, v)$ message from the coordinator of round $r$ then the function returns $v$, otherwise it returns $\perp$.

A proof of correctness of the algorithm is provided in the extended version of the paper [9].

## 3.2 The Methodology

The transformation of Mostefaoui and Raynal's algorithm into a $2f+1$ Byzantine consensus algorithm illustrates the application of the methodology for increasing the fault tolerance of crash algorithms. The methodology consists in doing the following modifications, then prove the correctness of the resulting algorithm:

*Communication channels:* communication channels are substituted by *authenticated reliable channels.* These channels

[2]This is not the case with $3f+1$ Byzantine consensus algorithms as they wait for $2f+1$ messages, a majority of which must come from correct processes.

constrain the power of the adversary in the network in the sense that these channels do not allow the creation, modification or dropping of messages.

*Broadcast communication:* broadcasts are substituted by *reliable broadcasts.* This mechanism constrains the power of the adversary by preventing it from delivering different messages with the same identifiers to different processes. This requires the reliable broadcast algorithm of Section 2.2.

*Message validation:* message receptions are enhanced with message validations, i.e., when a message is received it is only considered if it is valid as defined in Section 3.1. The objective is to force the adversary to conform to the algorithm.

*Reception quorum:* receptions of messages from quorums of $n - f$ processes are substituted by: reception of messages from at least $n-f$ processes plus the FD suspicion of all other processes (line 9 in Algorithm 2). This requires an eventually perfect muteness FD, $\Diamond\mathcal{MP}_{\mathcal{A}}$.

The process of applying the methodology is straightforward as shown in the next section.

## 3.3 Ben-Or's Algorithm

The objective of this section is to show how to apply the methodology, now that it was introduced. This section presents a transformation of Ben-Or's asynchronous crash-tolerant binary randomized consensus algorithm [2] into an asynchronous Byzantine binary randomized consensus algorithm, BO_2FBC.

The application of the methodology consisted of picking the original algorithm and modifying it following the list above. Then, we simply proved that the resulting algorithm satisfies the properties of Byzantine consensus.

The result can be found in Algorithm 3. Notice that Ben-Or's consensus is a binary consensus, i.e., $\mathcal{V} = \{0, 1\}$. Notice also that the consensus is randomized so there is a random action (line 13). The definition provided for multi-valued consensus in Section 3 is still valid, except for the termination that becomes probabilistic with probability 1.

The BO_2FBC algorithm is straightforward and uses a notation similar to the previous one (although we retained some of Ben-Or's notation also), so we skip a detailed textual description of how it works. The original algorithm was randomized so it had no failure detectors, but BO_2FBC uses $\Diamond\mathcal{MP}_{\mathcal{A}}$ due to the methodology of transformation (lines 4 and 10). A proof of correctness of the algorithm is provided in the extended version of the paper [9].

**Algorithm 3** BO_2FBC Byzantine consensus algorithm (code for process $p_i$)

**Function** BO_2FBC_CONSENSUS($v_i$)

1: $est_i \leftarrow v_i$ {current estimate of the value to be decided}
2: **step 0:** $r_i \leftarrow 1$ {round number}
3: **step 1:** RELIABLE_BROADCAST PHASE1$(r_i, est_i)$
4: **step 2:** **wait until** (*valid* messages PHASE1$(r_i, -)$ are received from at least $n-f$ processes) and ($\forall j$ : *valid* message PHASE1$(r_i, -)$ is received from $p_j$ or $p_j$ is suspected by $p_i$'s FD module)
5: **if** (more than $n/2$ messages have the same value $v$) **then**
6: RELIABLE_BROADCAST PHASE2$(r_i, v, decision)$
7: **else**
8: RELIABLE_BROADCAST PHASE2$(r_i, \perp)$
9: **end if**
10: **step 3:** **wait until** (*valid* messages PHASE2$(r_i, -)$ are received from at least $n-f$ processes) and ($\forall j$ : *valid* message PHASE2$(r_i, -)$ is received from $p_j$ or $p_j$ is suspected by $p_i$'s FD module)
11: **if** (there is one *decision* message PHASE2$(r_i, v, decision)$) **then** $est_i \leftarrow v$
12: **else if** (there are $n - f$ *decision* messages PHASE2$(r_i, v, decision)$) **then** DECIDE$(v)$
13: **else** $est_i \leftarrow 1$ or 0 each with probability 1/2
14: **step 4:** $r_i \leftarrow r_i + 1$; **go to step 1**

# 4. $2F + 1$ ATOMIC BROADCAST

The previous section shows that it is possible to solve asynchronous Byzantine consensus problems with $2f+1$ processes, using a reliable broadcast algorithm that needs $2f+1$ (or less) processes and an eventually perfect muteness FD ($\Diamond\mathcal{MP}_{\mathcal{A}}$). The flavor of multi-valued consensus solved in Section 3.1 is often used to show that a certain combination of mechanisms can be used to solve consensus, but it is particularly weak and not very useful to solve other distributed computing problems. This section introduces a novel flavor of consensus that can be solved with $2f + 1$ processes, and shows how it can be used to solve atomic broadcast.

The Validity property MVC1 states that the value that is decided is one of the values proposed (Section 3). However, it does not say if that process is correct. Suppose we want to solve some problem that involves solving several multi-valued consensuses: it is perfectly possible that *all* values decided in those consensus are proposed by faulty processes, turning them useless. The solution to this difficulty is to use another flavor of consensus with a different *validity property* (the other properties, MVC2/MVC3 remain the same), so we introduce a new form of consensus, *endorsement consensus*, and present an algorithm that solves the problem using only $2f + 1$ processes. The idea behind endorsement consensus is to consider that a correct process can have a notion about which values are adequate decisions for the consensus. More formally, each process $p_i$ has a set $E_i$ of values that it *endorses*, i.e., that it considers to be adequate decisions for the consensus. The problem is defined in terms of MVC2, MVC3 and the following property:

- *EC1 Validity.* If a correct process decides $v$, then $v$ was endorsed by some correct process.

The problem *assumes* that the endorsement sets satisfy the following properties: *Initial endorsement.* For any correct process $p_i$, $E_i$ contains always at least the value proposed by $p_i$ ($v_{0_i}$). *Increasing endorsement.* For any correct process $p_i$, for any two instants of time during the execution of the algorithm $t_1, t_2$, if $t_2 > t_1$ then $E_{i_2} \supseteq E_{i_1}$, where $E_{i_1}$ and $E_{i_2}$ are the values of $E_i$ at the two instants. *Eventual endorsement.* For any pair of correct processes $p_i$ and $p_j$, eventually $v_{0_i} \in E_j$. This last property means that all proposals of correct processes are eventually endorsed by all correct processes. Notice that this is not something that is guaranteed by an algorithm that solves the problem, but something that has to be satisfied for the algorithm to solve the problem, i.e., an assumption. This is precisely what happens with the atomic broadcast algorithm that we see next.

An algorithm that solves this flavor of consensus is Algorithm 2 with a single modification, substituting line 7 by:

7: **if** (valid PHASE1$(r_i, v)$ received from $p_{c_i}$ $\boxed{\text{and } v \in E_i}$) **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **end if**

We call this algorithm MR_2FBEC. The idea is that correct processes $p_i$ only disseminate a message PHASE2 if the estimate of the coordinator ($v$) is in $E_i$, besides the message being valid as already in Algorithm 2. A proof of correctness of the algorithm is provided in the extended version of the paper [9].

*Byzantine atomic broadcast* can be defined similarly to reliable broadcast (properties RB1-3 above) plus an additional order property:

- *AB4 Total order:* If two correct processes deliver two messages $M_1$ and $M_2$ then both processes deliver the two messages in the same order.

This problem has been shown to be equivalent to consensus in several system models [17, 8]. A transformation from consensus into Byzantine atomic broadcast with only $2f + 1$ processes, which we designate by HT_2FBAB, is provided by a combination of: the transformation of consensus and reliable broadcast into atomic broadcast presented in [17]; algorithm MR_2FBEC, which substitutes the consensus used in the transformation; in every process $p_i$, the set $E_i \equiv R\_delivered$.

The transformation provided in [17] is for crash faults. The idea is the following. When the algorithm is requested to do atomic broadcast of a message, it does reliable broadcast of that message. When a process receives such a message, it inserts it in the *R_delivered* set. When there are messages in that set that have still not been ordered, each process proposes the set of those messages to a consensus. Consensuses are done in an ordered fashion and messages within a consensus can be trivially ordered (e.g., in lexicographical order), so this provides a total order of messages.

In Hadzilacos and Toueg's transformation the consensus always decides a value proposed by a process that is not malicious, because there are no malicious processes in the crash system model. In our case we have to deal with that case, which is where the endorsement consensus comes in: it only lets values endorsed by correct processes to be decided by the consensus, thus ordered by the atomic broadcast algorithm. More precisely, given a set $S_i$ of messages pending to be ordered, proposed by some process $p_i$, this set is only decided if at least once correct process endorses this set, i.e., if it receives these messages from the reliable broadcast algorithm (remember that $E_i \equiv R\_delivered$). We skip the proof of correctness of HT_2FBAB.

# 5. REFERENCES


[1] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, 2003.

[2] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, Aug. 1983.

[3] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, Aug. 1984.

[4] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 189–204, October 2007.

[6] B. A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, Dec. 1988.

[7] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183, Oct. 2004.

[8] M. Correia, N. F. Neves, and P. Verissimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Computer Journal*, 41(1):82–96, Jan. 2006.

[9] M. Correia, G. T. Veronese, and L. C. Lung. Asynchronous Byzantine consensus with 2f+1 processes (extended version). DI/FCUL TR 09–17, Department of Informatics, University of Lisbon, November 2009.

[10] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash-stop to Byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50, May 2002.

[11] A. Doudou, B. Garbinato, and R. Guerraoui. Tolerating arbitrary failures with state machine replication. In H. B. Diab and A. Y. Zomaya, editors, *Dependable Computing Systems Paradigms, Performance Issues, and Applications*, chapter 2, pages 27–56. Wiley, 2005.

[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

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

[14] R. Friedman, A. Mostefaoui, and M. Raynal. Asynchronous bounded lifetime failure detectors. *Information Processing Letters*, 94(3):85–91, 2005.

[15] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 289–298, July 2000.

[16] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, Jan. 2001.

[17] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[18] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[19] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 49–63, 1999.

[20] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 248–262, Aug. 1988.

[21] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, Aug. 1984.

[22] P. Verissimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.