**This is the author's off-print version of: Distributed Computing Volume 24, Number 6 (2012), 299-321**
Open Access / Official version available at http://dx.doi.org/10.1007/s00446-011-0147-3

# Consensus in the Presence of Mortal Byzantine Faulty Processes

**Josef Widder · Martin Biely · Günther Gridling · Bettina Weiss · Jean-Paul Blanquart**

Published online: 19.11.2011

**Abstract** We consider the problem of reaching agreement in distributed systems in which some processes may deviate from their prescribed behavior before they eventually crash. We call this failure model "mortal Byzantine".

After discussing some application examples where this model is justified, we provide matching upper and lower bounds on the number of faulty processes, and on the required number of rounds in synchronous systems.

We then continue our study by varying different system parameters. On the one hand, we consider the failure model under weaker timing assumptions, namely for partially synchronous systems and asynchronous systems with unreliable failure detectors. On the other hand, we vary the failure model in that we limit the occurrences of faulty steps that actually lead to a crash in synchronous systems.

J. Widder was supported by the Austrian FWF National Research Network RiSE (S11403-N23), by the PROSEED project (proj.no ICT10-050) of the Vienna Science and Technology Fund, by NSF grant 0964696, and by the FWF project THETA (proj.no. P17757). M. Biely was partially supported by the Austrian BM:vit FIT-IT project TRAFT (proj.no. 812205). G. Gridling was supported by the Austrian FWF project SPAWN (proj.no. P18264) and the Austrian BM:vit FIT-IT project FAME (proj.no. 816454). The work of Sect. 4 and parts of Sect. 6 was originally presented at the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007.

Josef Widder
Technische Universität Wien, Formal Methods in Systems Engineering Group E184/4, Austria. widder@forsyte.tuwien.ac.at

Martin Biely
EPFL, Distributed Systems Laboratory (LSR), Lausanne, Switzerland. martin.biely@epfl.ch

Günther Gridling · Bettina Weiss
Technische Universität Wien, Embedded Computing Systems Group E182/2, Austria. {gg,bw}@ecs.tuwien.ac.at

Jean-Paul Blanquart
Astrium Satellites. jean-paul.blanquart@astrium.eads.net

**Keywords** consensus · Byzantine fault · distributed algorithm · fault tolerance · complexity

## 1 Introduction

In a distributed system subject to failures, consensus is the problem of agreeing on a common value out of a set of proposed values. Since consensus is fundamental for building reliable distributed systems, it has been considered under many different failure assumptions. One of the most studied models are crash faults; see [15,20,39] for an overview. However, the applicability of positive results that are related to crashes is limited as crash faults capture only a small number of causes that lead to abnormal behavior, such as power outage or permanent disconnection of a node from the network. A more general process failure model is the Byzantine one, which does not postulate any assumption on the behavior of faulty processes [38]. It captures all causes of failure, ranging from arbitrary bit flips in memory to intentional (malicious) causes like intrusions. However, Byzantine behavior may be overly pessimistic if we consider non-intentional faults, while the required redundancy — less than a third of the processes may be faulty for consensus — may be too expensive for many applications.

Additionally, the classic Byzantine assumption may be too pessimistic in many practical systems where one finds modules that observe the behavior of components and may act on these components — either automatically or by operator decision — in case of anomaly. This results in a modification of the component failure mode as perceived by other components of the system. In such cases, a component may not fail once and for ever according to a single "static" failure mode, but its faulty behavior follows some trajectory in the set of possible failure modes.

In this paper we show that rather than considering the worst case single failure mode exhibited by a component, it can be fruitful to exploit some characteristics of its failure mode trajectory. In particular, we consider failures which eventually end as a crash, after some latency period where a more pessimistic mode (up to arbitrary behavior) may be exhibited. We call this behavior *mortal Byzantine*. Nesterenko and Arora have introduced this kind of behavior in the context of self-stabilizing dining philosophers [43] under the name "malicious crashes".

After some motivation examples in Sect. 2, and general definitions in Sect. 3, we give our technical contribution. The technical part of our paper is divided into three sections. In Sect. 4, we show that in synchronous systems, consensus can be solved with less processes than in the classic Byzantine case [31]: Our algorithm only requires a majority of correct processes. We also show that a majority is necessary, i.e., our algorithm is optimal with respect to the required number of processes. If $t$ is the upper bound on the number of crashes during an execution, it is also known that consensus requires $t+1$ rounds [30]. We show that in the mortal Byzantine case, with just a majority of correct processes, such a bound cannot exist, i.e., no algorithm decides in a bounded number of rounds in every execution. More precisely, we show that in the case of just a majority of correct processes there are executions where processes cannot decide before faulty processes have crashed. So we make explicit that there is a trade-off between system size and liveness guarantees in the case of consensus.

From the case of mortal Byzantine faults in synchronous systems, several generalizations are possible. The two approaches we discuss in the present paper are, on the one hand, weakening the timing assumptions, and on the other hand, parameterizing the failure model.

In Sect. 5, we shall therefore relax the timing assumption. One classic way to solve consensus under relaxed timing behavior is partial synchrony [27]. In Sect. 5.1, we give a lower bound for partially synchronous systems which shows that it is not possible to solve consensus if a third of the processes are faulty. For a matching upper bound, we observe that Dwork, Lynch, and Stockmeyer [27] provided a partially synchronous algorithm solving consensus if less than a third of the processes are Byzantine faulty. As the mortal Byzantine fault model restricts the behavior of faulty processes more than the (classic) Byzantine fault model, this algorithm also works in our case, and provides the matching upper bound. We thus show that in partially synchronous systems nothing can be gained with respect to redundancy if the failure model is restricted from Byzantine to mortal Byzantine.

Another way to solve consensus with more relaxed timing assumptions is the failure detector approach by Chandra and Toueg [13]. With this approach, executions are considered to be asynchronous, but processes have access to oracles that provide information about the liveness of other processes. For classic Byzantine failures, Doudou et al. [25, 26] showed, among other things, that consensus is impossible with classic unreliable failure detectors. (We will discuss their and related approaches to circumvent this more thoroughly in Sect. 7.) For the mortal Byzantine model, we will (in Sect. 5.2) present and prove correct a consensus algorithm based on ideas from [27] and [13]; it requires that less than a third of the processes are faulty. This algorithm is totally asynchronous, which contrasts the literature on failure detector based Byzantine consensus.

The mortal Byzantine model, as described above, implies that if a process commits just one fault (e.g., sends a message with bad content) it crashes eventually. In the context of long-lived applications where the failure model is enforced by monitoring systems, eliminating a process due to one mishap seems to be wasteful. We address this issue by refining the failure model in the third major section, Sect. 6. More specifically, we do not require that every fault committed by a process inevitably leads to a crash. Rather, we count such mishaps and consider them just "potentially lethal." Only when they occur "too often" they are actually lethal and lead to a crash.

In more detail, we consider variations along two lines. On the one hand, we distinguish what kind of faulty behavior is actually potentially lethal, that is, whether all faults may lead to eventual crash or just faults that are perceived as two-faced behavior of the process. On the other hand, we distinguish under what circumstances potentially lethal faults actually lead to crashes: we distinguish when these faults sum up towards some threshold during either the whole execution, or just during a time window bounded in length.

Although these refined failure models are in the same spirit as the mortal Byzantine fault model from the earlier sections, they describe different fault behaviors, for instance, they need not crash if they do not commit too many failures. Hence, most of them are not comparable to the mortal Byzantine assumption.

In the case of all faults being potentially lethal, we show that resilience can be improved compared to mortal Byzantine faults. If only two-faced behavior is potentially lethal, we shall explain that a majority of correct processes is sufficient for consensus. Hence, such models are relevant in applications where one is willing to trade stronger assumptions on failure behavior against reduced system size, such as for instance applications with stringent weight limitation as in the space industry.

Apart from the two generalizations we consider in this paper, other approaches may of course also lead to interesting results in this area. Notably, Bazzi and Herlihy [8] have generalized our previous results to hybrid failure models, that is, for synchronous systems where mortal Byzantine

faults as well as classic Byzantine faults may be present. In addition to consensus, they also studied broadcast, for which they showed that it is solvable for any number of faulty processes. We will review their results in Sect. 7 along with other related work.

## 2 Motivation

Often, space systems have to meet strong requirements in terms of lifetime (without the possibility of repairing or replacing faulty components with new ones) and in terms of mass and volume — limiting the number of available redundant units. Space systems are also characterized for some missions by a strong prevalence of temporary faults; e.g., due to radiation. Therefore, when a faulty behavior is observed, it is usual practice to wait some time and assess more precisely its actual impact and recurrence characteristics, before (or instead of) engaging strong reconfiguration and definitely losing the failed component. In this example, the other components of the system may observe a faulty behavior evolving from maybe fully arbitrary to either correct or crash.

Note that the duration of this process may vary from very short to very long times. On the one hand, short durations occur when an automatic on-board mechanism — e.g., employing execution monitoring, temperature monitoring, or power monitoring — can identify a high severity failure at the first manifestation and switch off the component. While, on the other hand, long durations may be encountered when ground operators are in the loop to analyze trends in the telemetry, before requesting some reconfiguration by tele-commands. Between these two extremes, almost any possible intermediate durations are possible. For instance, it is common practice that an on-board mechanism explicitly counts the number of errors produced by a component in successive computation cycles. When a given threshold is reached — and the fault is considered permanent — the component is then eventually shut down by this on-board mechanism.

The European Automated Transfer Vehicle (ATV) is the unmanned transport spacecraft which is launched towards the manned International Space Station ISS. The ATV computer architecture contains, among other elements, a pool of computing units with distributed voting and agreement mechanisms at pool level. In addition, each computing unit of the pool is provided with a set of self-checking mechanisms (whose main aim is to provide some detection coverage for common mode faults). A computing unit detected as faulty by these self-checking mechanisms is reset into a silent mode, and in particular does not participate in the following votes. As a result of the combination of the vote and agreement mechanisms at computer pool level and of the self-checking mechanisms at the level of each computer unit of the pool, it may happen that a computing unit — as seen from other units — first appears faulty according to an arbitrary failure mode, before appearing crashed.

Applying the *separation of concerns* principle [21], consensus should work independently of the monitoring mechanism (or the human intervention). Consequently, solutions to consensus should rest on an abstract definition of the fault models that have to be tolerated, and this definition should be given independently of the machinery that enforces it. In the examples discussed above, an abstract description of the behavior of faulty processes leads directly to the *mortal Byzantine* failure model discussed in this paper.

## 3 General model and problem statement

Let $\Pi$ be a finite set of processes which constitute the distributed system with $|\Pi| = n$. A distributed algorithm $\mathscr{A}$ is a set of deterministic automata $(A_p)_{p\in\Pi}$. An execution is an infinite sequence of steps of $\mathscr{A}$. In each execution, each correct process $p$ takes an infinite number of steps according to $A_p$. Processes communicate by message passing.

We make the following assumptions for *mortal Byzantine* faults: A process is faulty in an execution if it only takes a finite number of steps in this execution. If $p$ is faulty, its behavior may deviate from its prescribed behavior $A_p$, but $p$ may send at most a finite number of messages. In every execution, at most $t$ processes are faulty.

In the following sections we will consider diverse kinds of models which have additional assumptions. These will be stated in the respective sections. Throughout the paper we will, however, always consider the same variant of the consensus problem. Every correct process has some initial value $v \in \{0,1\}$ represented in its initial state. All states of a correct process $p$ can be partitioned into the three sets in which $p$ has not yet decided, $p$ has decided 0, and $p$ has decided 1. If during an execution a correct process $p$ is in a state where it has decided $w$, we require that in the remainder of the execution, $p$ is only in states in which it has decided $w$, that is, each decision is irrevocable. Within the states there are halting states which ensure that after a correct process $p$ has reached a halting state $s$, each step of $p$ according to $A_p$ results in $p$ being in $s$ and no message is sent by $p$ during this step. We say that $p$ halts if it reaches a halting state. Moreover, each solution to the consensus problem must fulfill the following requirements:

Agreement. No two correct processes decide differently.
Validity. If some correct process decides $v$, then $v$ is the initial value of some correct process.
Decision. Every correct process eventually decides.
Halting. Every correct process eventually halts.

As we consider binary consensus, our validity condition is equivalent to "if all correct processes propose $v$, and a cor-

rect process decides $w$, then $v = w$." Discussions on multi-valued consensus are given in Section 8.

## 4 Synchronous systems

In this section we consider the synchronous model of distributed computation in which the steps of correct processes are organized in rounds. During each round, a correct process sends a message to each process for the current round according to its state, receives messages, and executes a state transition according to its state, the received messages, and its algorithm. A message sent by correct process $p$ to correct process $q$ in round $r$ is received by $q$ in round $r$.

In the following analysis, we will denote by $f_q^r$ the content of correct process $q$'s local variable $f$ (represented in the state of $q$) at the end of round $r$. As part of a message, we denote by "$*$" an open value.

We will present and analyze our algorithm that solves consensus in the presence of up to $t < n/2$ faulty processes in Sect. 4.1. In Sect. 4.2, we will then show that it is optimal.

### 4.1 Algorithm

In the classic lower bound proofs [38,39] for consensus in the presence of Byzantine processes one argues, informally speaking, that for every algorithm that should solve consensus, if there are at least a third of the processes faulty, then there are situations in certain executions where a correct process $p$ "knows" that some other process is faulty, but $p$ cannot decide which one. The main idea of our algorithm is that if $p$ encounters such a case, it waits until the faulty processes crash and the faulty values received can be removed. So this dilemma can be overcome due to the different failure assumption.

In more detail, our Algorithm 1 operates in phases consisting of two consecutive rounds. In the first round of each phase, correct processes send their proposed value and their decision value ($\perp$ until processes decide) to all. Every correct process $p$ collects these in the vectors $rcvprop_p$ and $rcvdec_p$. Moreover, $p$ checks whether some process $q$'s message was missing in some round. If this is the case, $q$ is removed from $p$'s estimate of the set of alive processes $\pi_p$, and the number $f_p$ of possibly alive faulty processes is updated. Entries in vectors $rcvdec_p$ or $rcvprop_p$ which correspond to faulty processes are set to $\dagger$ or $\perp$, respectively.

In the second round of a phase, $p$ sends $rcvprop_p$ and $rcvdec_p$ to all. At the end of the second round, $p$ checks whether all vectors it received from processes in $\pi_p$ — that is, processes that are not considered faulty — are equal. In this case $p$ tries to decide: it may decide on a value $w$ if it has received at least $f_p + 1$ messages stating that the sender proposed $w$. Recall that $f_p$ is not constant (as $t \geq f_p$ is the bound on the number of faults) as when a process has been detected as being faulty (if no message was received from this process in some round), $f_p$ is decreased by 1. Additionally, $p$ may only decide $w$ if it (or any other process in $\pi_p$, cf. $rcvdec_p$) did not receive a message in the previous round sent by some still alive process $q$ stating that $q$ has decided $1-w$. Therefore if a faulty (but still alive) process sends that it decided $w$ and another process sends it decided $1-w$, it is impossible to decide in this phase. However, since faulty processes eventually crash, their decision is removed (i.e., overwritten with $\dagger$) and will not be considered anymore; thereby, taking a decision becomes possible again.

At the end of the second round, correct process $p$ halts if all processes are either detected as faulty ($rcvdec_p[i] = \dagger$) or are known to have decided ($rcvdec_p[i] \in \{0,1\}$). Otherwise, a new phase is started.

In the remainder of this section we prove that our algorithm solves consensus by separately proving the required properties. We commence with preliminary definitions, that capture the round $D$ in which the last process decides and the round $H$ the first process halts in. To avoid circularity in the proofs we define $D$ and $H$ to be infinity for executions where such rounds do not exist, that is any property that we show to hold until one of these rounds will hold forever. We show only later in Theorems 3 and 4 that in each execution $D$ and $H$ are in fact finite.

**Definition 1 (Last Decision)** We define $D$ as the round in an execution for which it holds that all correct processes have decided by round $D$, and at least one correct process has not decided in round $D-1$. Further, $d$ is one such correct process.

**Definition 2 (First Halt)** We define $H$ as the maximum round in an execution for which it holds that no correct process has halted in some round less than $H$ in this execution.

We start our analysis with some preliminary lemmas. First, we make some statements about the decision value $decision_p$ which are quite obvious from code inspection:

**Lemma 1 (Decision)** *If $p$ is a correct process, then the following statements hold:*

*(1) $decision_p^0 = \perp$.*
*(2) If $decision_p$ is updated to value $v$, then $v \in \{0,1\}$.*
*(3) Variable $decision_p$ is changed at most once.*

*Proof* Statement (1) follows from the fact that $decision_p$ is initialized to $\perp$ in line 5. For statement (2), we note that in line 41, $decision_p$ gets assigned the minimum value in $W$. Inspection of line 38 reveals that only the values 0 and 1 can be in the set $W$. Statement (3) is a consequence of (2), since the if-condition in line 37, which encloses the only assignment to $decision_p$, can only evaluate to true as long as $decision_p = \perp$. So once line 41 has been executed, the if-condition will evaluate to false in all subsequent rounds. □

**Algorithm 1** Synchronous, Mortal Byzantine Tolerant Consensus

Code for processes $p$ if they are correct:

**Variables**

1: $r_p \leftarrow 0$ // round number
2: $\pi_p \leftarrow \Pi$ // the set of processes that are not detected faulty by $p$
3: $f_p \leftarrow t$ // possible number of not yet detected faulty processes
4: $prop_p \in \{0,1\}$ // proposed value
5: $decision_p \in \{0,1,\perp\} \leftarrow \perp$ // initially $\perp$
6: $rcvprop_p[n] \in \{0,1,\perp\}$ // proposed value received from $i$ in the current round
7: $rcvdec_p[n] \in \{0,1,\perp,\dagger\}$ // decision value received from $i$ in the current round
8: $c\_prop_p[n][n] \in \{0,1,\perp\}$ // the proposed values as seen by other processes
9: $c\_\pi_p[n]$ // the set of alive processes as seen by other processes
10: $c\_dec_p[n][n] \in \{0,1,\perp,\dagger\}$ // the decided values as seen by other processes

11: **repeat**
12: **if** $r_p(\text{mod})2 = 0$ **then**
13: send (INFORM, $prop_p$, $decision_p$) to all
14: receive
15: **for all** $i \in \Pi$ **do**
16: **if** no message from $i$ was received in *some* round **then**
17: $rcvdec_p[i] \leftarrow \dagger$ // $i$ is faulty
18: $rcvprop_p[i] \leftarrow \perp$
19: $\pi_p \leftarrow \pi_p \setminus \{i\}$
20: $f_p \leftarrow t - |\Pi - \pi_p|$
21: **else**
22: **if** received (INFORM, $prop_i$, $decision_i$) from $i$ in *current* round **then**
23: $rcvprop_p[i] \leftarrow prop_i$
24: $rcvdec_p[i] \leftarrow decision_i$
25: **if** $r_p(\text{mod})2 = 1$ **then**
26: send (ECHO, $rcvprop_p$, $\pi_p$, $rcvdec_p$) to all
27: receive
28: **for all** $i \in \Pi$ **do**
29: **if** received (ECHO, $rcvprop_i$, $\pi_i$, $rcvdec_i$) from $i$ **then**
30: $c\_prop_p[i] \leftarrow rcvprop_i$
31: $c\_\pi_p[i] \leftarrow \pi_i$
32: $c\_dec_p[i] \leftarrow rcvdec_i$
33: **else**
34: $c\_prop_p[i] \leftarrow \perp$ // $i$ is faulty
35: $c\_\pi_p[i] \leftarrow \perp$
36: $c\_dec_p[i] \leftarrow \perp$
37: **if** $decision_p = \perp \wedge \forall i,j \in \pi_p : c\_prop_p[i] = c\_prop_p[j] \wedge c\_\pi_p[i] = c\_\pi_p[j] \wedge c\_dec_p[i] = c\_dec_p[j]$ **then**
38: $W \leftarrow \{w : w \in \{0,1\} \wedge |\{j \in \pi_p : rcvprop_p[j] = w\}| \geq f_p + 1 \wedge \forall i \in \Pi : rcvdec_p[i] \neq (1-w)\}$
39: **if** $|W| > 0$ **then**
40: decide $\min W$
41: $decision_p \leftarrow \min W$
42: $r_p \leftarrow r_p + 1$
43: **until** $\forall i \in \Pi : rcvdec_p[i] \neq \perp$
44: **halt**

Now, we consider how a correct process is perceived by another correct process $p$ via the variables $rcvdec_p$ and $\pi_p$.

**Lemma 2 (Perception of Decisions)** *If $p$ and $q$ both are correct processes, then the following statements hold:*

*(1) If $decision_q^r = \perp$, then $rcvdec_p^s[q] = \perp$ in all rounds $s \leq r+1$.*
*(2) if $D$ is finite then $D < H$.*
*(3) If $q$ takes the decision $v$ in round $r$, then $rcvdec_p^s[q] = v$ in all rounds $s$ with $r < s \leq D+1$.*
*(4) If a message from process $i$ does not arrive at $p$ in round $r < H$, then $rcvdec_p^s[i] = \dagger$ and $i \notin \pi_p^s$ in all rounds $s > r$.*

*Proof* The variable $rcvdec_p$ is modified at two places in the code, in line 17 and in line 24. Note that in any round $r$ with $r(\text{mod})2 = 0$, either line 17 or line 24 is executed at each correct process for each process.

As $q$ is correct, $p$ will receive a message from $q$ in every round $r$ until $q$ halts, which cannot happen as long as $decision_q^r = \perp$. Thus, we can limit ourselves to considering line 24, where $p$ takes over the value received from $q$ and $q$ sends $decision_q^r$ in line 13, it follows that $rcvdec_p^s[q] = \perp$ in all rounds $s \leq r$. To prove (1), it thus remains to show that this also holds for $s = r+1$. To see this, consider that $q$ can only assign a new value to $decision_q$ in rounds $r$ where

$r(\mathrm{mod})2 = 1$ (line 41), but $decision_q$ is only sent (and received by $p$) in rounds $r$ with $r(\mathrm{mod})2 = 0$. In other words a change in $decision_q$ can only propagate to $p$ one round later.

To show (2), we observe that correct process $p$ only halts if there is no $\perp$ in the $rcvdec_p$ array, see line 43. Since we get from (1) that $rcvdec_p^r[d] = \perp$ in all rounds $r \leq D$, it follows that no correct process halts before round $H \geq D+1$.

For (3), we again observe that if $q$ has made decision $decision_q = v$ in round $r$ (line 41), then this decision is sent to all in round $r+1$ (line 13) and taken over into $rcvdec_p[q]$ in that same round (line 24). From Lemma 1 (3) we know that once a correct process decides, it sticks to that decision. We also know from (2) that no correct process halts before round $D+1$ such that it sends messages in all rounds until $D+1$ such that line 17 is not executed by a correct process before round $D$. Consequently, $rcvdec_p^s[q] = v$ for all rounds $s$, $r < s \leq D+1$.

Statement (4) is concerned with faulty processes. If $r$ is even, then the if-condition in line 16 will evaluate to true and hence $rcvdec_p^r[i]$ will be set to † in line 17 and $i$ will be removed from $\pi_p$ in line 19. If $r$ is an odd round, then in round $r+1$ the if-condition in line 16 will evaluate to true and $rcvdec_p^{r+1}[i] = \dagger$ will be assigned in line 17 and $i$ will be removed from $\pi_p$ in line 19. Since once the if-condition evaluates to true for process $i$ it does so for the remainder of the execution, we have $rcvdec_p^s[i] = \dagger$ for all rounds $s > r$. Also note that no process is ever added to $\pi_p$, thus that part of the statement follows as well. □

**Lemma 3 (Failure Detection)** *If $p$ and $q$ both are correct processes, then the following statements hold:*

*(1) $q \in \pi_p^r$ in all rounds $r \leq D$.*
*(2) $|\pi_p^r| > t$ in all rounds $r \leq D$.*
*(3) For each round $r \leq D$ no more than $f_p^r$ faulty processes are in $\pi_p^r$.*
*(4) If process $i$ crashes before round $r$, then $i \notin \pi_p^s$ in all rounds $s > r$.*

*Proof* To prove (1), we note that a process $i$ is only removed from $\pi_p$ if $p$ did not receive a message from $i$ in some round (line 16 and line 19). By Lemma 2 (2), $q$ does not halt before round $D$ and by simple code inspection we find that $q$ sends a message in every round until it halts in round $H > D$ or later. Consequently, messages from $q$ must arrive in each round at least until $H$, and thus $q \in \pi_p^r$ for all rounds $r \leq D$.

(2) follows from (1), since in all rounds $r \leq D$ all correct processes are in $\pi_p^r$ of every correct process $p$. By assumption, $|\Pi| \geq 2t+1$. Therefore, $|\pi_p^r| > t$.

To show (3), we note that initially $\pi_p = \Pi$ and $f_p = t$ such that our statement holds by the assumed bound on the number of (faulty) processes. By simple code inspection no process is ever added to $\pi_p$. If a process is removed from $\pi_p$, this is done in line 19 and $f_p$ is set to $t - |\Pi - \pi_p|$ in line 20. Since by (1), up to round $D$ no correct — i.e., only faulty — processes are removed from $\pi_p$, there cannot be more than $f_p^r$ faulty processes remaining in $\pi_p^r$.

Statement (4) follows directly from Lemma 2 (4). □

Equipped with these basic facts, we can now start to consider the consensus properties of the algorithm.

**Theorem 1 (Validity)** *If a correct process decides on some value $v$, then $v$ was proposed by a correct process.*

*Proof* By line 38, correct process $p$ only decides in some round $r$ on a value $v$ that was received by at least $f_p^r + 1$ (INFORM, $v$, $*$) messages from distinct processes in $\pi_p^r$ in round $r-1$. By Lemma 3 (3), there are no more than $f_p^r$ faulty processes in $\pi_p^r$ such that at least 1 correct process must have proposed $v$. □

After validity, we now turn our attention towards agreement, for which we employ the following lemma.

**Lemma 4** *If two correct processes $p$ and $q$ both decide in some round $r$, then both decide on the same value $v$.*

*Proof* By line 38 and line 40, the decision value is a deterministic function (identical at all correct processes) of $rcvprop_s^r$, $\pi_s^r$, $f_s^r$, and $rcvdec_s^r$ for correct process $s \in \{p,q\}$. We thus have to show that these four variables must have identical values at correct processes $p$ and $q$ in some round $r$ in which both decide.

From Lemma 3 (1) it follows that $q \in \pi_p^r$ and $p \in \pi_q^r$ for $r \leq D$. Since both decide in line 40 in round $r$, line 37 must have evaluated to TRUE in this round at both processes. It follows that $rcvprop_p^r = rcvprop_q^r$, $rcvdec_p^r = rcvdec_q^r$, $\pi_p^r = \pi_q^r$, and thus $f_p^r = f_q^r$ by line 20. From the deterministic decision function, our lemma follows. □

**Theorem 2 (Agreement)** *No two correct processes decide differently.*

*Proof* By Lemma 4, two correct processes do not decide differently if they decide in the same round. It remains to show that a correct process $q$ does not decide on a different value from the value another correct process $p$ has decided in some earlier round.

Assume by contradiction that a correct process $p$ decides on $w \in \{0,1\}$ in round $r$ and some other correct process $q$ decides $1-w$ in some round $s > r$. By line 38, for all $i$ in $\Pi$, $rcvdec_q^s[i] \neq w$, but by Lemma 2 (3), $rcvdec_q^s[p] = w$, which provides the required contradiction. □

As seen above, agreement and validity can be proven independently of the rounds processes crash in. Unfortunately, this is not true for liveness. However, as we will see in Sect. 4.2 this is inherent to solving consensus in the presence of $t$ mortal Byzantine processes, when $n \leq 3t$. In our

analysis we hence have to use the round in an execution at which the last faulty process crashes in order to show decision (and halting) of our algorithm. More precisely, we prove that if all faulty processes have crashed and their messages are received, the system is in a "clean" state such that the consistency checks in line 37 and line 38 allow every correct process to reach a decision.

**Definition 3 (Last Crash)** We define $C$ as the minimum round such that all processes that crash in an execution are crashed by (i.e., before or in) round $C$ and all messages sent by faulty processes are received by round $C$.

From our system models it follows that in every execution a finite $C$ exists. All rounds $r > C$ are clean rounds, i.e., no messages by faulty processes are received in rounds $r$.

**Theorem 3 (Decision)** *Every correct process eventually decides.*

*Proof* Let $r$ be the minimal round for which it holds that $r > C$ and $r(\mathrm{mod})2 = 0$. If all correct processes decide before round $r$ we are done. So in the remainder of this proof we consider only the case where at least one correct process $p$ does not decide before round $r$.

In the following we will show that every correct process that does not decide before round $r$ decides at the end of round $r+1$ by executing line 40. A correct process executes line 40 only if the statement in line 37 evaluates to TRUE and if $|W| > 0$, where $W$ is computed in line 38. In Lemma 6, Lemma 7, and Lemma 8, we will show that these two requirements are met.

**Lemma 5** *For any two correct processes $p$ and $q$ it holds that $rcvdec_p^{r+1} = rcvdec_q^{r+1}$, $\pi_p^{r+1} = \pi_q^{r+1}$, $f_p^{r+1} = f_q^{r+1}$, and $rcvprop_p^{r+1} = rcvprop_q^{r+1}$.*

*Proof* By Lemma 2 (4) correct processes consistently detect all faulty processes and set the corresponding entries in $rcvdec_p$. By Lemma 2 (1) and (3), all correct processes set the entries corresponding to correct processes in $rcvdec_p$ identically. Thus, there is agreement on the vectors $rcvdec_p$ at all correct processes $p$ at the end of round $r$ (and thus at the beginning of round $r+1$). Similarly, by Lemma 3 (1) and (4), $\pi_p^{r+1}$ and $f_p^{r+1}$ are identical at all correct processes $p$. In round $r$, all correct processes also send their proposed values to all, which are received by all correct processes $p$ who therefore set $rcvprop_p$ consistently. □

**Lemma 6** *For every correct process that does not decide before round $r$, it holds that line 37 evaluates to TRUE in round $r+1$.*

*Proof* In round $r+1$, all correct processes $p$ send (ECHO, $rcvprop_p^{r+1}$, $\pi_p^{r+1}$, $rcvdec_p^{r+1}$) to all. Since by Lemma 5 all $rcvprop_p^{r+1}$, $rcvdec_p^{r+1}$, $\pi_p^{r+1}$, and thus $f_p^{r+1}$ are identical, line 37 evaluates to TRUE at every correct process that does not decide before round $r$. □

So far, we have shown that every correct process $p$ that does not decide before round $r$ reaches line 38 in round $r+1$. In the following, we have to show that after executing line 38 in round $r+1$ it holds that $|W| > 0$ at every $p$ such that $p$ decides. To this end, we distinguish two cases: We consider in Lemma 7 the case where at least one correct process decides in a round before $r+1$, and in Lemma 8 the other case, where no process has decided before.

**Lemma 7** *If at least one correct process decides before round $r$, then for every correct process $p$ that does not decide before round $r$, $|W| > 0$ in round $r+1$ .*

*Proof* By Theorem 2, all correct processes $i$ that already decided in earlier rounds decided on the same value $w$, and thus $rcvdec_p^{r+1}[i] = w$. By Lemma 3 (4), $rcvdec_p^{r+1}[i] = \dagger$ for all faulty processes $i$. By Lemma 2 (1), for correct process $i$ that has not decided yet, $rcvdec_p^{r+1}[i] = \perp$. It follows that $rcvdec_p^{r+1}[i] \neq 1 - w$ for all processes $i$. In order to show that $|W| > 0$, it remains to show that there are more than $f_p^{r+1}$ entries for $w$ in $rcvprop_p^{r+1}[i]$ for processes $i \in \pi_p^{r+1}$, i.e., correct processes $i$.

Assume process $q$ has decided $w$ in some earlier round $s$. Due to line 38, it has received (INFORM, $w$, $*$) from $\ell > f_q^s$ distinct processes in round $s-1$. Out of these messages, $a \leq f_q^s$ are due to faulty senders and $b = \ell - a$ are from correct processes. Note that $b > f_q^s - a$.

For correct process $p$ in round $r$ we know that it has received $b$ messages (INFORM, $w$, $*$) from distinct correct processes (which have sent it via line 13). We also know that all faulty processes are crashed before round $r$, which includes those $a$ faulty processes that helped $q$ to reach its threshold. If follows that $f_p^{r+1} \leq f_q^s - a < b$ and thus sufficiently many (INFORM, $w$, $*$) messages were received. Thus, there are more than $f_p^{r+1}$ entries for $w$ in $rcvprop_p^{r+1}[i]$ for $i \in \pi_p^{r+1}$ and thus by line 38, $|W| > 0$. □

**Lemma 8** *If no correct process decides before round $r$, then for every correct process $p$, $|W| > 0$ in round $r+1$.*

*Proof* For all faulty processes $i$, $rcvdec_p^{r+1}[i] = \dagger$, and for all correct process $j$, $rcvdec_p^{r+1}[j] = \perp$, by Lemma 3 (4) and Lemma 2 (1), respectively. Thus, all entries in $rcvdec_p^{r+1}$ at correct process $p$ are either $\perp$ or $\dagger$, such that $|W| > 0$, if there exists at least one value $w$ for which holds that $|\{j \in \pi_p : rcvprop_p[j] = w\}| \geq f_p^{r+1} + 1$. According to line 20, $f_p = t - |\Pi - \pi_p|$ such that, since $\pi_p \subseteq \Pi$, we get $|\pi_p| = n - t + f_p$ and by the assumption on the number of processes ($n - t > t$) we get $|\pi_p| > t + f_p \geq 2f_p$. It follows from Lemma 1 (2) that there are only two possible decision values such that it follows that at least one must be proposed by more than $f_p^{r+1}$ correct processes and thus $|W| > 0$. □

Lemma 6 implies that every correct process $p$ that does not decide before round $r$ executes line 38 in round $r+1$. By

Lemma 7 and Lemma 8, $p$ executes line 38 in round $r+1$ so that $|W| > 0$ and consequently every $p$ decides in round $r+1$. Thus, every correct process decides at the latest in round $r+1$, and our Theorem 3 follows. □

**Theorem 4 (Halting)** *Every correct process halts.*

*Proof* We have to show that the expression of line 43 eventually evaluates to TRUE at every correct process $p$, i.e., that eventually $\forall i \in \Pi: rcvdec_p[i] \neq \perp$.

By Theorem 3, all correct processes eventually decide. After deciding — but before halting — they send their decision value via (INFORM, $*$, $decision_p$) to all in every even numbered round in line 13. The decisions are then written into $rcvdec_p[i]$ in line 24 such that eventually for all correct processes $i$, $rcvdec_p[i] \neq \perp$ at every correct process $p$. (Note that $rcvdec_p[i]$ is never reset to $\perp$ for correct processes $i$ if it was set to some value once.)

The faulty processes eventually stop sending messages such that missing messages will be detected at every correct process, and in line 17, $rcvdec_p[i] \leftarrow \dagger$ will be set for every faulty process $i$ at every correct process $p$. Consequently, for all processes $i$, eventually $rcvdec_p[i] \neq \perp$ at every correct process $p$ such that our lemma follows. □

**Corollary 1** *Algorithm 1 solves consensus.*

### 4.2 Lower bounds

One property of our algorithm is that it is guaranteed to decide only when all faulty processes have crashed. We show that given the number of processes it is inherently impossible to bound the decision time. Our failure model restricts the behavior more than the classic Byzantine model [38]. Thus, algorithms that solve consensus in the classic model with $n > 3t$ within $t+1$ rounds [38] can be applied to our model as well. In the following, we will show that when one reduces $n$, it is not possible anymore to solve the problem in a fixed number of rounds. We show that the round in which the last correct process decides cannot be constant, but depends on the failure pattern — a mapping of the set of faulty processes to a set of integers representing the round number in which the processes crash [23].

**Theorem 5** *In a system with up to $t$ mortal Byzantine faults, it holds for any integer $c$ and for any deterministic algorithm $\mathscr{A}$ that solves consensus for any $n > 2t$ that there exists at least one execution of $\mathscr{A}$ where the first faulty process crashes in round $c$ and at least one correct process decides in some round $r \geq c$.*

*Proof* Consider by way of contradiction that a consensus algorithm $\mathscr{A}$ exists where in every possible execution of $\mathscr{A}$ in which faults occur, all correct processes decide before the first faulty process crashes. At the time the correct processes decide, the prefixes of these executions can be mapped one-to-one to identical prefixes of executions of $\mathscr{A}$ in the presence of classic Byzantine faults. Thus, $\mathscr{A}$ also solves consensus with classic Byzantine faults contradicting the lower bound by Lamport, Shostak, and Pease [38] of $n > 3t$. □

**Corollary 2** *There is no correct deterministic algorithm that solves consensus in the presence of up to $t \geq n/3$ mortal Byzantine faults in a bounded number $g(t)$ rounds, $g$ being an arbitrary function on the upper bound on the number of faulty processes.*

After considering the decision time, we now show that our algorithm is optimal regarding the number of processes. We show that no algorithm exists for $2t$ processes.

**Theorem 6** *There is no algorithm that solves consensus in the presence of $t$ mortal Byzantine faults if $n = 2t$.*

*Proof* Consider by contradiction that such an algorithm $\mathscr{A}$ exists. And further consider a fault free execution $\mathscr{E}$ of $\mathscr{A}$ in which $t$ correct processes propose 0 and $t$ correct processes propose 1. Let in this execution $p$ be a correct process that proposes $w \in \{0,1\}$ and decides $1-w$, and let it decide in round $r$. Such a process $p$ must exist as both values are proposed by correct processes and only one can be decided upon by the agreement property of consensus.

Now consider executions of $\mathscr{A}$ where $p$ is correct and $p$ is one of $t$ correct processes that propose $w$ and the remaining $t$ faulty processes behave according to the algorithm at least until round $r$ but (wrongly) propose $1-w$. There exists such an execution $\mathscr{E}_t$ that is at least up to round $r$ for correct process $p$ locally indistinguishable from $\mathscr{E}$. Thus $p$ decides on $1-w$ in round $r$. After round $r$, all faulty processes crash such that $\mathscr{E}_t$ is an admissible execution in our model. However, $\mathscr{A}$ violates validity in $\mathscr{E}_t$ as $p$ decides $1-w$ although it was not proposed by any correct process in $\mathscr{E}_t$, which provides the required contradiction to $\mathscr{A}$ solving consensus. □

## 5 Relaxed timing assumptions

### 5.1 Partially synchronous systems

In this section, we employ the *basic round model* introduced in [27]. The model is a generalization of the synchronous model relaxed with respect to reliability of communication. It is assumed that if a correct process receives a message in round $r$ sent from a correct process, the message was sent in round $r$, that is, the model is communication closed [28]. Further, it is assumed that in every execution there exists a round GST (called, global stabilization time) such that for all rounds $r \geq$ GST, a message sent by correct process $p$ to

correct process $q$ in round $r$ is received by $q$ in round $r$. (As a consequence of the definitions, messages sent before GST may be lost.)

Our failure assumption restricts behavior of faulty processes more than the classic Byzantine assumption. Consequently, in partially synchronous systems the algorithms from [27], that solve consensus for $n \geq 3t+1$, solve consensus also for mortal Byzantine faults. Using a standard partitioning argument, we show that this redundancy is optimal for mortal Byzantine faults.

**Theorem 7** *There exists no algorithm that solves consensus among $n = 3t$ processes in the basic round model in the presence of up to $t$ mortal Byzantine faults.*

*Proof* We consider a system of $3t$ processes and separate the processes into three disjoint sets $A$, $B$, and $C$ such that $|A| = |B| = |C| = t$.

Consider by contradiction that an algorithm $\mathscr{A}$ exists that solves consensus in this system.

Let $\mathscr{E}_A$ be an execution of $\mathscr{A}$ where all processes in $C$ are initially crashed and $\mathrm{GST}_A = 0$. Further let all correct processes propose 1 such that there is some round $D_A$ in which the last correct process $p \in A$ decides 1.

Similarly, let $\mathscr{E}_C$ be an execution of $\mathscr{A}$ where all processes in $A$ are initially crashed and $\mathrm{GST}_C = 0$. Further let all correct processes propose 0 such that there is some round $D_C$ in which the last correct process $q \in C$ decides 0.

We now consider the executions of algorithm $\mathscr{A}$ where $\mathrm{GST} > \max\{D_C, D_A\}$, all processes in $A$ and $C$ are correct, and all processes in $B$ are mortal Byzantine faulty and crash only after round GST. We further restrict our analysis to executions where processes within the sets $A$ and $C$ have synchronous communication right from the beginning, but communication between these two sets is asynchronous until GST, i.e., no messages are received from processes in the other set. Further, the faulty processes in $B$ are perceived by all processes in $A \cup C$ as if they would communicate synchronously with them right from the start.

Now consider the execution $\mathscr{E}$ where all processes in $A$ propose 1 and all processes in $C$ propose 0 and where in each round up to GST the processes in $A$ receive the same messages from the processes in $B$ as in $\mathscr{E}_A$ and the processes in $C$ receive the same messages from the processes in $B$ as in $\mathscr{E}_C$. To $A$, $\mathscr{E}$ is indistinguishable from $\mathscr{E}_A$ up to $D_A$ such that they decide 1, while to the processes in $C$, $\mathscr{E}$ is indistinguishable from $\mathscr{E}_C$ such that they decide 0. This behavior violates agreement and thus we have a contradiction to the assumption that $\mathscr{A}$ solves consensus. □

### 5.2 Asynchronous system with failure detector

In this section we consider the asynchronous model of computation augmented with unreliable failure detectors introduced by Chandra and Toueg [13]. We provide here only the details of the model to the extent necessary for our paper; for a more detailed definition, cf. [13].

For the failure detector approach, we assume that there is a discrete global time $\mathbb{N}$. As in the crash case defined in [13], we consider failure patterns as function $F: \mathbb{N} \to 2^{\Pi}$. We only consider failure patterns where $F(t) \subseteq F(t+1)$ and $|\bigcup_{t \in \mathbb{N}} F(t)| < n/3$.

If $p \notin F(t)$ and $p \in F(t+1)$, we say process $p$ crashes at time $t+1$. A process that crashes is mortal Byzantine faulty as defined in Sect. 3, i.e., it is allowed to behave arbitrarily until it crashes, except that it may only send a finite number of messages. A process that never crashes is correct.

Asynchronous computations are modeled as sequences of steps that happen at certain times. During each step a correct process $p$

- delivers at most one message previously sent to $p$,
- queries its local failure detector and receives a response which is defined with respect to the time the step occurs,
- performs a state transition according to its algorithm, the received messages and the response from the failure detector, and
- possibly sends a message.

The system is asynchronous in that we do not assume any bounds on the time between the send step and the receive step of a message (message delays) or the time between two steps (computing speeds), except that all these times have to be finite for correct processes. If some process is in $F(t)$, it does not take a step at time $t$.

We limit ourselves here to failure detectors which respond with a set of process identifiers that are suspected of having crashed. Using this convention one can define the eventually perfect failure detector $\diamond\mathscr{P}$ as in [13] via the following properties:

Strong Completeness. There is a time from which onwards every process that crashes is permanently suspected by every correct process.

Eventual Strong Accuracy. There is a time after which no process is suspected before it crashes.

Aguilera et al. [3] devised an algorithm that solves consensus in the presence of Byzantine faults under weak synchrony assumptions. In their solution, they used *consistent unique broadcast* (based upon [10,46]). Similarly, we use consistent unique broadcast to ensure that faulty processes are not perceived two-faced by correct processes. To ease the presentation of the algorithm, we assume that the system is equipped with *consistent unique broadcast* [3] that has *cubcast* and *cudeliver* as broadcast and delivery primitives, respectively. Under these primitives, messages have the form ($p$, TYPE, $k$, $v$) where

- $p$ is the sender identifier,

- TYPE is a message type identifier (e.g., ACK, NACK),
- $k$ is a phase number, and
- $v$ is a value.

As in the synchronous case, we denote by "$*$" an open value.

**Definition 4 (Consistent Unique Broadcast)** For consistent unique broadcast the message system has to satisfy the following properties:

Totality. If a correct process $p$ invokes *cubcast* with message ($p$, TYPE, $k$, $v$) then all correct processes eventually invoke *cudeliver* for ($p$, TYPE, $k$, $v$).

Unforgeability. If some correct process $p$ does not invoke *cubcast* with message ($p$, TYPE, $k$, $v$) then no correct processes ever invokes *cudeliver* for ($p$, TYPE, $k$, $v$).

Relay. If a correct process invokes *cudeliver* for message ($p$, TYPE, $k$, $v$) then eventually all correct processes invoke *cudeliver* for ($p$, TYPE, $k$, $v$).

Uniqueness. For each $p$, TYPE, and $k$ every correct process invokes *cudeliver* at most once for ($p$, TYPE, $k$, $*$).

Implementations of this broadcasting primitive for asynchronous systems even without failure detectors in the presence of up to $t < n/3$ (classic) Byzantine faults can be found in [10,46]. Consequently, this primitive can also be implemented in our setting.

### *5.2.1 The algorithm*

Algorithm 2 is a variant of the classic Byzantine consensus algorithm that Dwork, Lynch, and Stockmeyer have introduced in their seminal paper on partial synchrony [27]. Moreover, the algorithm uses some mechanisms introduced in the failure detector based (crash) consensus algorithm by Chandra and Toueg [13].

We find it convenient to present the algorithm to consist of two tasks. While the first task loops through phases each consisting of four rounds, the second task performs jobs which are independent of the round the first task is currently executing. (For a complete implementation of the algorithm, additional tasks for implementing the consistent unique broadcast may be required.) We assume that these tasks are scheduled fairly, e.g., in round robin.

The key idea is that in each phase a different process is the coordinator. The coordinator tries to find a decision value that is consistent with the validity requirement of consensus. If such a value is found by a correct coordinator, it sends this value via a DECIDE message. A locking mechanism ensures that no correct process that becomes coordinator in a later phase will find a different value to decide, which ensures agreement. After the system becomes stable (all faulty processes have crashed and the failure detector provides reliable information) every correct coordinator will find a value and send the DECIDE message. These messages are handled in the second task, and eventually there are sufficiently many of these messages in the system such that processes reach a decision and halt. We now give a more detailed description of the separate parts of the algorithm.

Each process $p$ maintains two sets of values. The set $proposed_p$ always keeps the value a correct process $p$ is certain at least one correct process has initially proposed (required for validity). Initially, this set contains just the initial value of the process itself. For the locking mechanism, the second set called $acceptable_p$ is used. It contains only values from $proposed_p$, and in case values are locked by $p$, it contains only those locked by $p$.

During each phase the following happens: In *Round 1*, all correct processes *cubcast* INFORM messages containing their *acceptable* set. The processes wait for delivery of those INFORM messages in *Round 2* where the INFORM messages are collected from all processes that are not suspected — i.e., the processes that are not in the failure detector output $\mathscr{D}_p$. Throughout the algorithm, $\mathscr{M}_p(v,\phi)$ denotes the set of messages ($q$, INFORM, $\phi$, $acceptable_q$) delivered by process $p$ where $v \in acceptable_q$ (this set is maintained in the separate *Task 2*). Note that due to false suspicions and the asynchrony, INFORM messages broadcast in Round 1 of some phase can be delivered in arbitrary rounds (belonging to arbitrary phases). Only when the false suspicions stop occurring and the last messages of all crashed processes are delivered — i.e., in the stable period — it is ensured that the messages are delivered in the round they were broadcast.

In *Round 2* the coordinator tries to lock some value. A value can be locked if at least $n-t$ processes find the value acceptable. If it finds such a value it broadcasts a LOCK message, instructing the other processes that they should set a lock on this value. If the coordinator does not find such a value it broadcasts "I DO NOT KNOW".

In *Round 3*, each correct process waits for the *cudeliver* of either a LOCK message or an I DO NOT KNOW message from the coordinator, as long as the latter is not suspected to have crashed. If a process receives a LOCK message, it checks whether this message is acceptable for at least $n-t$ processes in the current phase. If so, it returns an acknowledgment (ACK) to the coordinator, otherwise a negative acknowledgment is sent. Basically, by sending the ACK message, a correct process $p$ tells the coordinator that it is going to lock the value in the next round.

If a coordinator, during *Round 4*, receives sufficiently many ACK messages, it is certain that sufficiently many correct processes lock the value in the current phase and that therefore no other correct coordinator can find a different value in a later phase. (That is, the current execution is univalent [32].) Therefore, it sends a DECIDE message. The DECIDE messages are used to propagate the decision and for reaching a halting state eventually in Task 2 (see below).

**Algorithm 2** Mortal Byzantine Tolerant Consensus Algorithm

Code for processes $p$ if they are correct:

**Variables**

$lock_p \leftarrow 0$
$r_p \leftarrow 0$
$proposed_p \leftarrow \{v_p\}$
$acceptable_p \leftarrow proposed_p$

**Task 1**

**loop**
  $r_p \leftarrow r_p + 1$
  $\phi_p \leftarrow \lfloor r_p/4 \rfloor$
  $c_p \leftarrow (\phi_p \bmod n) + 1$

  **if** $r_p(\text{mod})4 = 1$ **then**
    *cubcast* ($p$, INFORM, $\phi_p$, $acceptable_p$)

  **else if** $r_p(\text{mod})4 = 2$ **then**
    **wait until** *cudeliver* ($q$, INFORM, $\phi_p$, $acceptable_q$) from all processes $q \notin \mathscr{D}_p$
    **if** $p = c_p$ **then**
      $\mathscr{V}_p \leftarrow \{v : |\mathscr{M}_p(v, \phi_p)| \geq n - t\}$ // Coordinator tries to choose a value.
      **if** $\mathscr{V}_p \neq \emptyset$ **then**
        $w \leftarrow \min\{v : v \in \mathscr{V}_p\}$ // There is a value that can be chosen.
        *cubcast* ($p$, LOCK, $\phi_p$, $w$)
      **else**
        *cubcast* ($p$, I DO NOT KNOW, $\phi_p$)

  **else if** $r_p(\text{mod})4 = 3$ **then**
    **wait until** *cudeliver* ($c_p$, LOCK, $\phi_p$, $w$) **or** ($c_p$, I DO NOT KNOW, $\phi_p$) from $c_p$ **or** $c_p \in \mathscr{D}_p$
    **if** *cudelivered* ($c_p$, LOCK, $\phi_p$, $w$) **and** $|\mathscr{M}_p(w, \phi_p)| \geq n - t$ **then**
      *send* ($p$, ACK, $\phi_p$, $w$) to $c_p$ // The value chosen by the coordinator is consistent with the acceptable values.
    **else**
      *send* ($p$, NACK, $\phi_p$) to $c_p$

  **else if** $r_p(\text{mod})4 = 0$ **then**
    **if** $p = c_p$ **then**
      **wait until** received ($q$, ACK, $\phi_p$, $w$) **or** ($q$, NACK, $\phi_p$) from all processes $q \notin \mathscr{D}_p$ // Coordinator waits for feedback.
      **if** received at least $n - t$ ($q$, ACK, $\phi_p$, $w$) messages from distinct processes $q$ **then**
        *send* ($p$, DECIDE, $w$) to all processes // Sufficiently many correct processes acknowledged, and will lock $w$.

    $proposed_p \leftarrow proposed_p \cup \{v : \exists \phi\, |\mathscr{M}_p(v, \phi)| \geq t + 1\}$ // Update locks (done by all processes)
    $locked_p \leftarrow \{(\phi, w) : \phi \leq \phi_p$ and *cudelivered*$((\phi \bmod n) + 1, \text{LOCK}, \phi, w)$ and $|\mathscr{M}_p(w, \phi)| \geq n - t\}$
    **if** $locked_p \neq \emptyset$ **then**
      $lock_p \leftarrow \phi : (\phi, w) \in locked_p$ and $\forall (\phi', w') \in locked_p : \phi \geq \phi'$
      $acceptable_p \leftarrow \{w\} : (lock_p, w) \in locked_p$
    **else**
      $acceptable_p \leftarrow proposed_p$

**Task 2**

**loop**
  **if** *cudeliver*($q$, INFORM, $\phi_q$, $acceptable_q$) **then**
    $\forall v \in acceptable_q : \mathscr{M}_p(v, \phi_q) \leftarrow \mathscr{M}_p(v, \phi_q) \cup \{(q, \text{INFORM}, \phi_q, acceptable_q)\}$ // Bookkeeping
  **if** $\exists w$: received ($q$, DECIDE, $w$) from $t + 1$ distinct processes **then**
    *send* ($p$, DECIDE, $w$) to all processes
  **if** $\exists w$: received ($q$, DECIDE, $w$) from $n - t$ distinct processes **then**
    *send* ($p$, DECIDE, $w$) to all processes
    decide($w$)
    **halt** // halt all tasks

At the end of Round 4 all processes update their locks. This is necessary due to the inconsistencies that may occur in Round 3 due to unreliable failure detector information. A lock can be set for a value $v$ and a phase $\phi$ only if the coordinator has broadcast LOCK for $v$ and at least $n-t$ processes $q$ found $v$ acceptable at the start of the current phase, i.e., they have sent ($q$, INFORM, $\phi$, $acceptable_q$) messages, with $v \in acceptable_q$. A correct process locks the value for which the required messages were delivered, and the associated phase number is maximal — see line 31 to line 37. In the proofs below, we often state that processes lock some value. By this we mean that they execute line 34 and line 35 such that only one value is acceptable for them, and the variable *lock* is set to the phase number as described above.

Apart from maintaining the sets $\mathscr{M}_p(v,\phi)$, Task 2 plays a central role in the final stage of the algorithm which takes care of deciding and halting. The mechanism used is basically echo broadcasting similar to [46]. More precisely, when a correct process $p$ has received at least $t+1$ DECIDE messages for the same value from distinct processes, it is certain that at least one correct process sent it and it can forward the message to all other processes. Furthermore, when a correct process $p$ receives at least $n-t$ DECIDE messages for the same value from distinct processes, it is certain that (since $n > 3t$) every other correct process will eventually receive at least $t+1$ such messages. Consequently, all correct processes will eventually send this DECIDE message, and thus every correct process will eventually receive at least $n-t$ such messages. Thus, $p$ can decide and halt, as all other processes will eventually also reach this point and will decide and halt. This explains why we can halt all other tasks (including those implementing *cubcast*) at this point: no more messages from process $p$ are required for the other processes to reach a decision.

### 5.2.2 Correctness proof

The proof of correctness follows the ideas in [27] and [3]. The proof of decision, however, is more involved than that in [27] due to the asynchrony that eventually stops in the models of [27] but remains in our model during the whole execution. We start by showing simple properties that follow directly from the properties of the broadcasting primitive.

**Lemma 9** *It is impossible for correct processes to lock two distinct values in a phase $\phi$ if it belongs to a correct coordinator $q = (\phi \bmod n) + 1$.*

*Proof* Correct processes only lock a value $w$ when they have added it to $locked_p$ before in line 32, that is, when they invoked *cudeliver* for a message $((\phi \bmod n)+1$, LOCK, $\phi$, $w)$ and they invoked *cudeliver* for ($p$, INFORM, $\phi$, $acceptable_p$) messages broadcast by $n-t$ distinct processes $p$ with $w$ in $acceptable_p$. From code inspection we see that correct coordinators broadcast at most one LOCK message per phase. It follows therefore from the unforgeability property of *cubcast* that if the coordinator is correct, then only one such LOCK message can be delivered per phase; the lemma follows. □

**Lemma 10** *If at least $t+1$ correct processes either halt before reaching phase $\phi+1$ or have a lock on value $w$ in phase $\phi$ with a correct coordinator, no correct process will lock value $v = 1-w$ in any phase $\phi' \geq \phi$.*

*Proof* For phase $\phi' = \phi$ the lemma is true by Lemma 9. Assume by contradiction that there is a phase after $\phi$ where a correct process locks $1-w$ and let $\phi' > \phi$ be the smallest of these phases. By line 32 we must have $|\mathscr{M}_p(v,\phi')| \geq n-t$ which is required to lock a value at any correct process $p$. By our choice of $\phi'$, at the beginning of phase $\phi'$ the $t+1$ processes still have a lock on $w$ such that $v \notin acceptable$ or have already halted. Thus their messages do not show up in $\mathscr{M}_p(v,\phi')$, i.e., a distinct set of processes of cardinality $n-t$ must have sent messages for $v$. Summing up the sizes of these sets we get $(n-t)+(t+1)$. As this is strictly greater than the number of processes $n$, we arrive at the required contradiction. □

**Lemma 11** *If some correct coordinator $c$ sends a message ($c$, DECIDE, $\phi$, $w$) and no other correct process has sent a* DECIDE *message in phases less than $\phi$, then:*

*(1) At least $t+1$ correct processes lock $w$ with phase number $\phi$, or halt before reaching phase $\phi+1$.*
*(2) The correct processes that lock $w$ with phase number $\phi$ will always have a lock on $w$ from then on.*

*Proof* A correct coordinator requires at least $n-t$ acknowledgments to its LOCK message to send ($c$, DECIDE, $r$, $w$), among which are at least $t+1$ sent by correct processes who have received the coordinator's LOCK message; let $S'$ be the set of these processes. From line 41 and line 43 follows that as soon as $c$ has sent ($c$, DECIDE, $\phi$, $w$) to all, correct processes may decide and halt at any time (after enough DECIDE messages have been exchanged what happens concurrently in Task 2). To prove (1) let $S$ be the subset of $S'$ of processes that do not halt before they reach phase $\phi+1$. Since they have sent an acknowledgment, by line 22 it follows for all $p \in S$ that $|\mathscr{M}_p(w,\phi)| \geq n-t$, and (1) follows from line 32.

By Lemma 10, the processes in $S'$ do not lock another value in later rounds such that line 34 and line 35 ensure that $w$ remains locked, which proves (2). □

**Theorem 8 (Agreement)** *No two correct processes decide differently.*

*Proof* Assume by contradiction that two correct processes decide on different values.

From line 41 and line 43 it is evident that in order for a correct process to decide on $v$ at least one correct coordinator must have sent a DECIDE message for that value (in line 30). Thus, in order for processes to decide on different values, there must be two correct coordinators, say $p$ and $q$, which sent in line 30 DECIDE messages for $v$ and $1-v$ in phases $\psi_p$ and $\psi_q$. As each phase belongs to exactly one coordinator, clearly $\psi_p \neq \psi_q$, so w.l.o.g. we assume that $\psi_p < \psi_q$. Now, Lemma 11 tells us that there are at least $t+1$ processes which have a lock on $v$ in phase $\psi_q$, or halt before. That is, they will cubcast $(*, \text{INFORM}, \psi_q, \{v\})$ in the first round of phase $\psi_q$, or halt before. By the properties of cubcast, $q$ can therefore not receive $n-t$ INFORM messages for $1-v$, that is, it cannot broadcast a LOCK message for that value, and thus will not receive $n-t$ ACK messages. Thus it will not send a DECIDE message containing $1-v$; a contradiction. □

**Theorem 9 (Validity)** *If a correct process decides on some value v, then v was proposed by a correct process.*

*Proof* By lines 31-37 it follows obviously that a value must be in the set $acceptable_p$ of a correct process $p$ in order to be decided upon. Initially, only the initial value of $p$ is in $acceptable_p$ and only values broadcast by at least $t+1$ processes, i.e., by at least one correct process, are added. □

**Theorem 10 (Decision)** *Eventually, every correct process decides.*

*Proof* A correct process decides only in line 45, and thus only if it has received $n-t$ DECIDE messages for some $w$. Since $n > 3t$, it follows that at least $t+1$ of these originate from correct processes, and that every correct process will eventually receive these, causing them to send DECIDE messages for the same $w$. Thus, if one process decides, all correct processes will receive $n-t$ DECIDE messages for the same value, i.e., they will all decide. It is thus sufficient to prove that one correct process decides.

To do so, suppose by way of contradiction that no process decides. As a correct processes halts (line 46) only after it has decided (line 45) no correct process ever halts.

**Lemma 12** *If no correct process ever halts, then for any correct process p the phase number $\phi_p$ grows without bound.*

*Proof* Assume by way of contradiction that there is a process and a phase, such that the process remains in this round forever. Let $\phi$ be the minimum among all phases processes remain in forever, and let $p$ be a process which remains in $\phi$ forever.

Inside the loop there are only three lines where processes wait, namely line 12, line 21, and line 28.

If $p$ is blocked in line 12, then there is a process $q$ which is never suspected from which no message is delivered. As $q$ is never suspected, it follows from completeness that $q$ is correct. As by minimality of $\phi$, no correct process is blocked before. Process $q$ thus reaches round $4\phi+1$ and eventually cubcasts the INFORM message in line 10. By totality, $p$ delivers this message eventually. A contradiction.

Therefore, no correct process is blocked in line 12 in phase $\phi$. With similar arguments one proves that $p$ cannot be blocked in line 21 and further that it cannot be blocked in line 28 arriving at the required contradiction. □

Let time $\tau_c$ be the minimal time such that no messages broadcast (or sent) by faulty processes are delivered (or received) after $\tau_c$. Let $\tau_p$ be the minimal time such that all faulty processes are crashed before $\tau_p$, all crashes are detected and no process is suspected before it crashes; in other words, the system is stable from time $\tau_p$ on. As we assume that no correct process decides, and Lemma 12 ensures that phase numbers increase without bound, we are sure that no correct process has halted by time $\max\{\tau_p, \tau_c\}$. Unambiguously, we may thus define $\psi$ to be the maximum phase a correct process is in at time $\max\{\tau_p, \tau_c\}$, and we define the first stable phase $\phi_s = \psi + 1$. Due to the definition of the failure detector and the bounded number of faults, $\phi_s$ is finite in every execution.

**Lemma 13** *If no correct process ever halts, and if there exists an x such that for all correct processes p, we have $x \in acceptable_p$ when they enter phase $\phi \geq \phi_s$, and phase $\phi$ belongs to a correct coordinator $q = (\phi \bmod n) + 1$, then q sends (q, DECIDE, $\phi$, w) for some w to all. Moreover, for all phases $\phi' > \phi$, $acceptable_p = \{w\}$.*

*Proof* All correct processes $p$ invoke *cubcast* for a message $(p, \text{INFORM}, \phi, acceptable_p)$ in line 10 and they invoke *cudeliver* for all inform messages broadcast during phase $\phi$ in line 12 as there are no false suspicions anymore by the choice of $\phi_s$. Since there are at least $n-t$ correct processes $p$ with $x \in acceptable_p$, the set $\mathscr{V}_q$ contains at least the value $x$. Thus, $q$ invokes *cubcast* for a message $(q, \text{LOCK}, \phi, w)$ for some $w$ in line 17 (note that the coordinator need not necessarily propose $x$ to be locked here). As $q$ is not suspected by any correct process $p$, each correct $p$ invokes *cudeliver* for this message and sends the acknowledgment message in line 23. Thus, $q$ receives at least $n-t$ ACK messages and therefore sends a DECIDE message in line 30, which concludes the first part of our lemma.

For the second part, we note that since all correct processes $p$ have sent ACK to message $(q, \text{LOCK}, \phi, w)$, they also lock the value $w$ for round $\phi$, and thus by line 35 all correct processes $p$ set $acceptable_p = \{w\}$ before entering phase $\phi+1$. Therefore, by the choice of $\phi_s$, no other value will ever be broadcast in round 1 in later phases and the second part of our lemma is true. □

Further we show that the assumption of Lemma 13 eventually holds. As with the beginning of phase $\phi_s$ the system is stable, the failure detectors at all correct processes provide the same correct information. By line 12 and line 21, any message broadcast in phases beginning with $\phi_s$ are thus delivered in the phase they are broadcast. However, messages broadcast before phase $\phi_s$ may be delivered at any time. Let $\phi_t$ be the minimal phase such that before the first correct process enters $\phi_t$, all messages broadcast by phase $\phi_s$ are delivered by all correct processes.

**Lemma 14** *If no correct process ever halts, there exists a phase greater than or equal to $\phi_t$ such that in round 1 of this phase, there exists some x such that for any correct process p, the value x is in $acceptable_p$.*

*Proof* We distinguish two cases:

1. There is a correct process $p$, and a phase $\phi$ such that before phase $\phi_t$, process $p$ has delivered a $((\phi \bmod n)+1$, LOCK, $\phi$, $*)$ message by the coordinator as well as the at least $n-t$ messages ($*$, INFORM, $\phi$, $acceptable_*$) broadcast by distinct processes. Let $\phi_\ell$ be the largest phase for which such messages where delivered and let $w$ be the associated value. By relay and the choice of $\phi_t$ all correct processes deliver these messages before phase $\phi_t$ and $(\phi_\ell, w)$ is contained in the *locked* sets of all correct processes. Therefore, by phase $\phi_t$, all processes add $w$ to *acceptable* in line 35 such that all correct processes have the value $w$ in *acceptable* when entering the phase $\phi_t$ and our lemma holds in this case.
2. Otherwise, that is, there is no correct process that delivers a $((\phi \bmod n)+1$, LOCK, $\phi$, $v)$ message by the coordinator and at least $n-t$ messages ($*$, INFORM, $\phi$, $acceptable_*$) broadcast by distinct processes before $\phi_t$. It follows that by the choice of $\phi_t$, no value is ever locked by any correct processes for any phase before $\phi_t$. As $\phi_t$ is a stable phase, every correct process waits in line 12 until it has delivered INFORM message from all correct processes. As there are at least $2t+1$ correct processes, and there are only two possible decision values, there must be some $v \in \{0,1\}$ such that for any correct process $p$, we have $|\mathscr{M}_p(v,\phi_t)| \geq t+1$. It follows that when a correct process $p$ executes line 31 in phase $\phi_t$, then it adds $v$ to *proposed*. As no value has been locked before by the assumption of this case, each correct process $p$ executes line 37 in phase $\phi_t$, and the value $v$ will be in *acceptable* at all correct processes in the next phase. The lemma follows in this case. □

Lemma 14 ensures that there is some phase $\phi \geq \phi_t$ and a value $x$, such that $x$ is in all sets $acceptable_p$ of correct processes $p$ when the phase $\phi$ is started. If phase $\phi \geq \phi_t$ belongs to an already crashed coordinator, we easily observe (as no more "old" messages are delivered) that at each correct process, the value $x$ remains in the set *acceptable*. However, eventually a phase that belongs to a correct coordinator is reached. We may thus apply Lemma 13 which ensures that a correct coordinator $q$ sends ($q$, DECIDE, $*$, $w$) to all, and that $w$ is locked by each correct process forever. Consequently, we may repeatedly apply Lemma 13 for every phase belonging to a correct coordinator after phase $\phi_s$. As there are at least $n-t > 2t+1$ correct processes that will be coordinators, they will all send a DECIDE message (either in line 30, line 42, or line 44) and eventually every correct process receives at least $n-t$ DECIDE messages for the same value and decides in line 45 which provides the required contradiction and thus proves Theorem 10. □

**Theorem 11 (Halting)** *Eventually, every correct process reaches a terminal state.*

*Proof* By Theorems 10, all correct processes eventually decide. By simple code inspection it follows that if a correct process decides it halts. Thus, the theorem follows. □

**Corollary 3** *Algorithm 2 solves consensus.*

In the following section we shortly discuss that the axiomatic failure detector properties can be implemented in partially synchronous systems even in the presence of mortal Byzantine faults.

### *5.2.3 Lower bound*

To show that our algorithm has optimal resilience, it is sufficient to prove that the computational model this algorithm rests on can be implemented under partial synchrony. As a consequence, every lower bound for partially synchronous systems is also valid in the asynchronous model with failure detector, and in particular the lower bound from Sect. 4.2.

The failure detector $\diamond\mathscr{P}$ can be implemented in the basic round model [27] even in the presence of mortal Byzantine faults. Correct processes send heartbeats to all in each round. If a process $p$ does not receive a message from some process $q$ in round $r$, then $p$ suspects $q$, while if a message is received then $p$ does not suspect $q$. After round GST, all messages from correct processes are received in the round they were sent and thus no correct process will be suspected anymore. Since faulty processes can only send a finite number of messages, these messages will eventually all be received. From then on, all faulty processes will be permanently suspected. In other words, Eventual Strong Accuracy and Strong Completeness hold.

It remains to show that the asynchronous model, or more specifically, reliable channels, can be implemented in partial synchrony. This is achieved in the basic round model [27] by retransmission (via piggybacking) of every message that was originally sent in round $r$ in all rounds $r' > r$.

Hence, it is clear that the basic round model is stronger (but not necessarily strictly stronger) than the asynchronous model equipped with $\diamond\mathscr{P}$ such that any lower bound for the former carries over to the latter.

## 6 Refined failure models

We return to the study of synchronous systems, but now consider slightly different failure models. The fault models we consider here are again inspired by the concept of hardware monitors, which observe the behavior of processes, can detect (some) faults, and are able to remove a detected faulty process from the system. In this section we assume that this happens within a known bounded time. We will say a fault is *potentially lethal*, if it is detected by some external means which can cause the associated process to be shut down; a fault is *non-lethal*, if it cannot cause the associated process to be removed no matter how often it occurs.

Further, we distinguish two models of how often potentially lethal faults may occur before the faulty process is shut down. One in which the faults associated to a process accumulate until a given threshold is exceeded, and another one which allows a certain number of faults within a given time period and only removes a faulty process if it creates too many faults too closely together, i.e., within a given time window.

(O) *Bounded Failure Occurrence.* After a process has exhibited potentially lethal faults in a total of $x > 0$ rounds, it fails within $y$ rounds with $0 \leq y < \infty$.

(R) *Bounded Failure Rate.* After a process has exhibited potentially lethal faults within $x > 0$ out of $z \geq x$ consecutive rounds, it fails within $y$ rounds, with $0 \leq y < \infty$.

By choosing $z = \infty$ in (R), we get model (O), and so (O) is a special case of (R). Both models allow a certain number of faults to go unpunished and only remove a faulty process after it exhibits "too many" faults. This allows transient faults, e.g., memory faults, caused for example by single event upsets, to simply run their course; the afflicted process is only removed if it does not recover by itself within a given number of rounds. We do not concern ourselves with the cause of such transient faults, although in practice this might have significant impact on the fault detection and removal logic, for example in case of corrupted messages.

By choosing which faults are detectable and thus potentially lethal, one derives different failure models. We first consider the models (O) and (R) when all faults are potentially lethal in Sect. 6.1. Then in Sections 6.2 we will investigate how to solve consensus if only severe faulty behavior — asymmetric faults — is potentially lethal.

Contrary to the previous sections, we need to restrict the delivery of messages sent by a faulty process $p$ to rounds in which $p$ is still alive.[1] So after the hardware monitor has removed the component (after at most $y$ rounds), no more messages from $p$ may arrive at any correct process.

**Algorithm 3** Fault-Tolerant Full Message Exchange

Code for processes (`ft_fme`$(v_p)$ for process $p$):

**Variables**

1: $k = x + y + 1$ // number of message repetitions
2: $v_p$ // the own data to be sent
3: $rcv_p[n][k]$ // data received from $i$ in $k'$-th send iteration
4: $retval_p[n]$ // data delivered for process $i$, $1 \leq i \leq n$

5: **for** $k' = 1$ to $k$ **do**
6: send $(v_p)$ to all
7: receive
8: **for all** $i \in \Pi$ **do**
9: **if** received $(v_i)$ by $i$ **then**
10: $rcv_p[i][k'] \leftarrow v_i$
11: **else**
12: $rcv_p[i][k'] \leftarrow \perp$
13: **for all** $i \in \Pi$ **do**
14: **if** $\exists w : |\{k' : rcv_p[i][k'] = w\}| = k$ **then**
15: $retval_p[i] = w$
16: **else**
17: $retval_p[i] = \perp$
18: **return** $retval_p$

Based on these assumptions, we can cope with faulty processes by repeating messages for a sufficient number of times (i.e., by time redundancy). This allows us to devise a simulation that makes failures appear in a more restricted failure mode, e.g., as send omission in the following section. Our fault-tolerant full message exchange primitive called `ft_fme`$(v)$ requires $k = x + y + 1$ rounds; see Algorithm 3. The nature of the values exchanged is not specified, it can be single binary values, but it can also be any other data type, e.g., a set, depending on the algorithm using the primitive.

We observe from the code that if `ft_fme`$(v)$ returns $\perp$, in some sense a value fault is converted to an omission. We shall thus use the convention that if `ft_fme`$(v)$ returns $\perp$ we say that no message (or value) has been delivered. In the following section we will show that in fact less severe faults can be simulated by this primitive on top of faults that obey $x$, $y$, and $z$.

### 6.1 When all faults are potentially lethal

In this section, we will consider a system where *every* fault is potentially lethal. This matches the contingency mechanisms discussed in Sect. 2 (e.g., temperature monitoring or power monitoring).

First we show that using `ft_fme`$(v)$ in the models (O) or (R), a faulty process is perceived as *send omission faulty*,

[1] This is similar to the synchronous model of computation in [39]. Contrary, in the previous section we did not have to restrict the timing of messages from and to faulty processes.

i.e., as if it would follow its algorithm as a correct process, but omits to send at least one message during the execution.

**Lemma 15** *In the models (O) and (R), if all communication is done via the* `ft_fme(v)` *primitive, and all faults are potentially lethal, then faulty processes appear omission faulty (or crashed).*

*Proof* Consider one instance of full message exchange, and assume that at the end, correct processes $p$ and $q$ both deliver a value for faulty process $f$, i.e., $retval_p[f] \neq \perp$ and $retval_q[f] \neq \perp$. Since $retval_i[f] \neq \perp$ is equivalent to equation $rcv_i[f][k'] = rcv_i[f][k'']$ for all $k', k'' \in \{1,...,k\}$, it follows that $p$ and $q$ have received $k = x+y+1$ equal values $retval_p[f]$ and $retval_q[f]$, respectively, from $f$. However, $f$ can pollute at most $x$ consecutive rounds, which may be followed by at most $y$ additional (possibly faulty) rounds before $f$ crashes. Thus, during one of the $k = x+y+1$ rounds, process $f$ must have followed the algorithm and has sent correct messages to all processes. Let $v_f$ be this correct value. Hence, there is an $\ell$ such that $rcv_p[f][\ell] = rcv_q[f][\ell] = v_f$ and thus $retval_p[f] = retval_q[f] = v_f$, so both processes $p$ and $q$ deliver the same correct value $v_f$ for faulty process $f$.

Of course, some or all correct processes may not deliver a value at all, for example if the faulty process commits a send omission fault. □

From Lemma 15, we know that a faulty process may appear at most send omission faulty. It is well known that consensus with $t$ send omission faults can be solved with $t+1$ processes [45] in $t+1$ rounds, such that we obtain:

**Theorem 12** *If all faults are potentially lethal, consensus with at most t faulty processes can be solved with $n > t$ processes.*

Apart from achieving consensus with $t < n$ faulty processes, our fault models also allow a bounded algorithm execution time of $k \cdot (t+1) = (x+y+1) \cdot (t+1)$ rounds.

### 6.2 When only asymmetric faults are potentially lethal

In certain applications, the assumption that hardware monitors can detect all kinds of faults (like in the previous sections) may be overly optimistic. In particular, detecting that the content of a message deviates from what should have been sent if the algorithm was followed, requires knowledge of the actual algorithm executed, and which messages are supposed to be sent. Moreover, for instance in the case of control systems, a message's content may depend also on the state of the environment. Hence, the detection of a faulty value may be well beyond the scope of a simple monitor in such cases. We will limit ourselves to broadcast based algorithms. Here two-faced behavior might be detectable in certain systems, and no knowledge about the content of "correct" messages is required.

We will say that a process $a$ commits an *asymmetric* fault in round $r$ if there is a correct process $p$ that receives $m$ from $a$ in round $r$ and there is another correct process $q$ which either receives no message from $a$ in round $r$ or $q$ receives $m' \neq m$ from $a$ in round $r$. We will now consider (O) if only asymmetric faults are potentially lethal.

Note carefully that processes may send erroneous values during the whole execution, and are not required to crash if in each round they send the same value to each correct receiver. Such a faulty behavior we will call symmetric fault in the following. As symmetric faults are non-lethal, a lower bound for symmetric faults is also a lower bound for the fault assumption of this section. As in the proof of Theorem 6 no asymmetric behavior was required to derive the result, with a similar proof we obtain:

**Theorem 13** *There is no algorithm that solves consensus in the presence of $t \geq n/2$ symmetric faulty processes.*

**Corollary 4** *If symmetric faults are non-lethal, consensus with at most t faulty processes requires at least $n > 2t$ processes.*

After this lower bound, we now turn our attention to algorithms. We start with the following preliminary lemma.

**Lemma 16** *Under (O) or (R), if all communication is done via* `ft_fme(v)` *and only asymmetric faults are potentially lethal, then if two correct processes deliver the messages m and $m'$ respectively in any round r, then $m = m'$.*

*Proof* Assume that correct processes $p$ and $q$ both deliver a value for a faulty process $f$. It follows from the code that both $p$ and $q$ have received $k = x+y+1$ equal values. Since asymmetric faults are potentially lethal, the same arguments as those used in the proof of Lemma 15 imply that the received values are the same and consequently we obtain that $retval_p[f] = retval_q[f]$. □

As symmetric faults are non-lethal, the value delivered by the correct processes may now be faulty. Again, some correct processes may not deliver a message sent by a faulty process at all. Conversely, it is easy to show that using the `ft_fme(v)` primitive, the communication between correct processes is reliable. This behavior corresponds to the communication primitive *Crusader agreement* defined in [22]:

**Definition 5 (Crusader Agreement)** Let $c$ be a process that sends a message to all other processes.

- All correct processes that do not explicitly know that $c$ is faulty agree on the same message.
- If $c$ is correct, then all correct processes agree on the message sent by $c$.

In this context a process explicitly knows that $c$ is faulty if `ft_fme(`$v$`)` returns $\perp$. However, we shall show next that asymmetric behavior on the `ft_fme(`$v$`)`-level, requires that the sender commits a potentially lethal fault. To this end, we introduce the notion of an *unclean message exchange*: we say faulty process $p$ causes an unclean message exchange, if a correct process receives a message from $p$ while another one does not receive a message. If this is not the case, the message exchange is clean. Considering `ft_fme(`$v$`)`, each time a faulty process $p$ causes an unclean message exchange, $p$ commits a potentially lethal fault.

### *6.2.1 Bounded failure occurrence*

Under model (O), each time a potentially lethal fault is committed counts towards the $x$ bound. It follows that eventually there are no more asymmetric faults, and that eventually all correct processes will receive the same messages in the same round. That is eventually all message exchanges will be clean. We will use this "eventually consistent" property in Algorithm 4.

**Lemma 17** *In model (O), if a process causes x unclean message exchanges, it will appear crashed in all subsequent full message exchanges.*

*Proof* From Lemma 16 we know that an asymmetric faulty process $f$ can at worst make one correct process $p$ deliver a message $m$ for round $r$ while another correct process $q$ delivers no message at all for $r$. To achieve this asymmetric delivery, at least one of the $k$ messages sent via `ft_fme()` must have been due to a potentially lethal fault, thus $f$ has to commit at least one potentially lethal fault in round $r$ and hence will fail within $y$ rounds after the $x$-th such fault. Since our fault-tolerant full message exchange requires $x+y+1$ rounds, if the $x$-th potentially lethal fault was at the end of full message exchange $\ell$, then process $f$ will send at most $y$ messages in full message exchange $\ell+1$. As the receivers only receive at most $y$ of the $x+y+1$ expected messages, none of the correct processes will deliver a value, and $f$ will appear crashed in this full message exchange and all subsequent ones. □

Our approach to solve consensus is thus to divide computations in macro rounds, where in each macro round each correct process initiates a `ft_fme(`$v$`)`. As a consequence of Lemmas 16 and 17, in order to solve consensus in model (O) it is sufficient to provide a consensus algorithm for the following failure model:

- there are at most $t < n/2$ faulty processes,
- processes communicate through a Crusader Agreement primitive, and
- during an interval of $x+1$ (macro) rounds of the consensus algorithm, faulty processes are perceived symmetrically[2] in at least one round.

**Algorithm 4** Synchronous Consensus under model (O) and non-lethal symmetric faults

Code for processes (for process $p$):

**Variables**

```
1:  v_p ∈ {0,1}                          // current value (initially the proposed value)
2:  r_p ← 0                                                        // current round
3:  rcvprop_p[n] ∈ {0,1,⊥}                // value received from i, 1 ≤ i ≤ n
4:  prune()                                          // a function mapping all values
                                          //  not in {0,1} to ⊥, 0 to 0, and 1 to 1.
5:  repeat
6:      r_p ← r_p + 1
7:      rcvprop_p ← prune(ft_fme(v_p))
8:      if |{i: rcvprop_p[i] = 0}| ≥ |{i: rcvprop_p[i] = 1}| then
9:          v_p = 0
10:     else
11:         v_p = 1
12: until r_p = x·t + 1
13: decide v_p
14: halt
```

In this model, we shall prove that Algorithm 4 achieves consensus with $t < n/2$ faulty processes. The key idea of the algorithm is to calculate a new propose value in each round from the values received from all processes. This is done sufficiently many times so that there is at least one round in which all message exchanges are clean. In this round, all correct processes will calculate the same value, which will henceforth prevail in all remaining rounds. To this end, a correct process $p$ repeatedly sends its own value and collects the values from all processes in set $rcvprop_p$. Process $p$ then sets its new proposed value $v_p$ either to the majority value in $rcvprop_p$, if it exists, or to 1 otherwise. After $x \cdot t + 1$ such iterations, the last calculated proposed value $v_p$ is used as the decision value.

In the following we will prove that Algorithm 4 solves consensus in the presence of up to $t < n/2$ faulty processes.

**Theorem 14 (Decision & Halting)** *Every correct process decides and halts after algorithm round $x \cdot t + 1$.*

*Proof* This theorem follows from simple code inspection: After round $x \cdot t + 1$, the algorithm halts, and the value of $v_p$ after this round is the decision value. □

**Theorem 15 (Validity)** *If some correct process decides v, then v is proposed by some correct process.*

*Proof* Since we are only considering binary consensus and lines `8-11` clearly ensure that only 0 or 1 can be decided, it suffices to show that when all processes have the same initial value $v$, then $v$ will be decided. Since processes are

[2] That is, either *all* correct processes receive the same message or *no message* of the faulty sender is received by any correct process.

guaranteed to receive at least $n-t$ values from correct processes and $n-t > t$, we know that $|\{i : rcvprop_p[i] = v\}| > |\{i : rcvprop_p[i] = 1-v\}|$ holds in all rounds. Therefore, the check in line 8 will always prefer the correct processes' value. □

**Theorem 16 (Agreement)** *No two correct processes decide differently.*

*Proof* It follows from Lemma 17, that a faulty process can appear asymmetric faulty in up to $x$ rounds before it crashes. Therefore, as there are at most $t$ faulty processes, correct processes can receive inconsistent information in at most $x \cdot t$ rounds. Thus, at least one of the $x \cdot t + 1$ rounds is a clean round. In a clean round, all correct processes have the same set $rcvprop_p[i]$ and will therefore compute the same value, which will henceforth win the majority in every subsequent round and thus will be delivered as the decision value. □

**Corollary 5** *Algorithm 4 solves consensus under (O).*

#### *6.2.2 Bounded failure rate*

In the previous section, we discussed a consensus algorithm which is based on the assumption that processes communicate via Crusader agreement and that faulty processes can be perceived asymmetrically at most for $x$ rounds before they crash. Since in the failure model with bounded failure rate (R), asymmetric behavior leads only to a crash if it happens at a too high density, we know from Lemma 16 that in this section we have to deal with permanent faults, while communication can still be done using Crusader agreement. Consensus actually can be solved under these assumptions with a majority of correct processes: The argument uses results by Fitzi and Maurer [33]. They have shown that consensus can be solved based on a special kind of multi-cast primitive called *two-cast channels*:

**Definition 6 (two-cast channels)** Among any triple of processes and for any process among them, there exists a broadcast channel to the remaining two processes.

Fitzi and Maurer then use this communication primitive to extend their result to primitives with other consistency guarantees, one of which they call *weak broadcast* [33, Definition 4]. As weak broadcast basically is just a reformulation of Crusader agreement, and can thus be implemented with any number of processes under our fault assumption, we obtain that consensus is solvable with a minority of faulty processes under model (R).

### 6.3 Discussion

Table 1 summarizes this section's results with respect to resilience for systems with known lifespan of faults. The table contains pointers to the results in literature which we have shown to apply to our failure models.

| System | all potentially lethal | only asym. potentially lethal |
|---|---|---|
| (O) | $t+1$ [45] | $2t+1$ (Alg. 4) |
| (R) | $t+1$ [45] | $2t+1$ [33] |

**Table 1** Resilience of consensus in different systems

Clearly, systems benefit from a perfect hardware monitor that can detect all faults, as in this case $n > t$ processes are sufficient, at the cost of increased message complexity. If symmetric faults cannot be detected, though, the $n > 2t$ bound of Sect. 4 cannot be improved. Still, even though the bound remains the same, the coverage of systems may be improved: transient failures, which cause the algorithm of Sect. 4 to fail, can now be handled as long as they only occur in moderation.

Our solutions for consensus use the `ft_fme(`$v$`)` primitive to implement macro rounds in which some of the faulty behaviors are masked. In this way, we can solve consensus with simple algorithms based on these macro rounds. A drawback of this approach is that our solutions are quite inefficient with respect to the number of rounds. Whether more efficient algorithms can be found is subject to future work.

The assumptions of certain faults which are non-lethal is not easily comparable to the mortal Byzantine model of Sect. 4.2. On the one hand, faulty processes may be alive forever, given that they do not behave two-faced. On the other hand, after behaving two-faced (too often), a faulty process must crash within a fixed number of rounds, while in Sect. 4.2, a faulty processes was required to crash only eventually.

## 7 Related Work

The seminal work by Pease, Shostak, and Lamport [38,44] considers consensus (or more precisely the closely related Byzantine Generals problem) for synchronous systems with arbitrary faulty processes. This combines, on the one hand, highly optimistic [39, page 5] timing assumptions with, on the other hand, highly pessimistic fault assumptions to result in the well-known $n > 3t$ bound. Naturally, researchers turned to investigating other timing assumptions and fault types. Relaxing the synchronous system assumption led to the well-known impossibility result by Fischer, Lynch, and Paterson [32] for asynchronous systems and subsequently to work on weak synchrony assumptions as the seminal paper by Dwork, Lynch and Stockmeyer [27] and subsequent work as [3,11,17]. Another approach was to augment the asynchronous system with failure detectors [24,25,40]. Other

approaches aim at optimizing normal case behavior [1,18, 41] or consider more elaborate fault models in an effort to improve fault resilience as, for instance, Byzantine faults with recovery [4,5] or hybrid failure models [6,9,47].

Our results can be roughly divided into two domains, namely, synchronous and non-synchronous systems. In the case of synchronous systems, our approach aims at improving resilience by strengthening the fault assumptions. In the other case, we explore in which cases something can be gained from the failure model. We show that in partially synchronous models nothing is gained compared to Byzantine faults. In the asynchronous case we showed that unreliable failure detectors [13] — which were originally introduced for crash failures only — can in fact be used to tolerate non-benign faults as well, if the faulty processes eventually crash. In contrast to most previous work in this area, we consider failure mode trajectories, where components migrate from one (severe) fault type to another (less severe) one, since such models are not just of purely theoretical interest, but also of practical use, as illustrated in the examples from Sect. 2.

In this paper we considered a variant of consensus that is required to halt. As laid out in [15], in general, results on deciding not necessarily carry over to halting, i.e., reaching a terminal state in which no further messages are sent; cf. [23,39]. In our context, the distinction is interesting, as we have shown in Sect. 4.2 that if there is just a majority (i.e., $2t < n \leq 3t$) of correct processes, there is no fixed number of rounds (in the synchronous model) that an algorithm requires to decide in each execution. However, we showed that our algorithms — including the asynchronous one in particular — solve the problem of reaching a terminal state as well. This is an interesting property of our algorithms, which is of practical interest in long running systems where it is advantageous to be able to free the resources of halted processes.

### 7.1 Synchronous Systems

Our Algorithm 1 has some similarities to the EDAC algorithm described in [15] and originally introduced in [29]. EDAC solves the early deciding consensus problem in the presence of crash faults.

We have seen that the number of required processes for synchronous consensus can be reduced to $n \geq 2t+1$, but at the cost that the required decision time (more precisely the number of rounds) cannot be bounded. Due to the latter result, it might appear as if this model cannot be employed if bounded decision time is required. However, following the late binding principle [35] this is only true when the life time of faults (e.g., the delay of the detection mechanism) in the real system is not bounded. As our proofs reveal, if over all executions the time in which faulty processes can pollute the system is bounded, so is the decision time.

### 7.2 Asynchronous system with a failure detector

For crash faults, Chandra and Toueg [13] showed that consensus can be solved in an asynchronous system, if it is enriched with an oracle that provides processes in each step with a set of processes that the oracle assumes to be faulty. In order for this information to be useful to the asynchronous algorithm, from some time on, the oracle has to provide reliable information forever [12,14]. Chandra and Toueg also showed that such oracles can be implemented in a generalization of the partially synchronous systems of [27].

Later, Doudou et al. [25,26] observed that the failure detector approach cannot be extended seamlessly to Byzantine faults. Instead, they proposed "gray-box" modules, which they called muteness detectors, that allow to solve consensus in the presence of Byzantine faults. Gray-box means that information flow is not one-way anymore, as is the case with the Chandra and Toueg solution of consensus where the failure detector outputs the suspicions but takes no input from the consensus algorithm. Rather, gray-box means that the consensus algorithm itself has to provide the failure detector with information on faults (which is impossible for the FD to get without knowledge of the consensus algorithm's internals). Doudou et al. observe that this circularity is inherent to the problem. In the solution they provide, the correctness of both the failure detector and the consensus algorithm depend on the synchrony assumptions of the partially synchronous model.

Most of the existing work on asynchronous failure detector based Byzantine consensus [7,25,26,34,36,37,40] tries to strengthen the system in more than one way. The solution of Doudou et al. adds synchrony assumptions at consensus level as well as bidirectional communication between the consensus and the failure detector module, and so do Friedman et al. [34] who refer to the solution of Doudou et al. [26]. Kihlstrom et al. [37] enrich the system with authentication. Additionally, the failure detector implementation they provide requires the special consensus algorithm proposed in their paper as they observe certain consensus messages; it is assumed that the consensus algorithm is round based. (This is another example of the circular dependency Doudou et al. show to be problem inherent.) Malkhi et al. [40] define quiet processes via behaviors of processes given a certain broadcast primitive that is implementable in asynchronous systems. Implementing the failure detector they define, however, requires synchrony assumptions (partial synchrony) on at least a subset of consensus messages (which may be given in practical applications). None of the discussed consensus algorithms is purely asynchronous.

Our approach is different from the ones that can be found in the mentioned literature: We investigate failure detector based consensus under a stronger fault model, namely the mortal Byzantine model. For this we show that this failure assumption allows to tolerate faults in a modular manner, i.e., the failure detector implementation requires no knowledge on the internals of the consensus algorithm. Moreover, in sharp contrast to the consensus algorithms that were discussed above, our algorithm is purely asynchronous, i.e., there are no synchrony assumptions on consensus messages and the relative execution speed of the processor running the consensus algorithm. These nice properties, however, can of course only be achieved due to our fault semantics being stronger than in the presented related work. We assume that Byzantine processes have to crash eventually, i.e., they only take a finite number of steps. This eliminates the aforementioned circularity.

From a different, perhaps more practical viewpoint, the mortal Byzantine model can be interpreted as a system with Byzantine faults where some (external) part of the system can reliably detect faults and can guarantee that faulty nodes will eventually be silenced or crashed. This latter assumption is the central difference to the failure detector based approaches to tolerate classic Byzantine faults we have discussed above. If this has to be implement within the system, our approach only provides a different kind of layering, which allows an asynchronous consensus algorithm.

There are several attempts to extend the failure detector approach to other fault models, weaker than crash. Most notable Aguilera et al. [2] and Delporte-Gallet et al. [19] consider benign faults, that is, faults that do not lead to corrupted states or messages. Aguilera et al. studied failure detector based consensus in the presence of crashes with recovery. They classified processes into eventually up, eventually down, and unstable processes. Their approach was to find failure detector definitions that are suitable for the crash-recovery failure model; definitions that turned out to be rather intricate. Delporte-Gallet et al. gave general transformations for problem specifications, failure detectors, and algorithms. Hence, they show that crash-tolerant algorithms can be transformed such that they solve transformed problems using transformed failure detectors. Whether such a general transformation is also possible for mortal Byzantine faults is an interesting open question and subject to future work.

The approaches of Aguilera et al. [2] and Delporte-Gallet et al. [19] generalize the liveness aspect of the faulty processes that are originally [13] considered crash faulty only. These approaches deal with failure models that do not ensure that faulty processes terminate as a crash. In our work on asynchronous systems with failure detectors, we still require that processes crash, and our liveness properties are therefore simpler than those in [2] and [19]. However, we focused on "value faults" that can be seen as generalizations of the safety aspect of process behavior.

### 7.3 Mortal Byzantine faults

Bazzi and Herlihy [8] have generalized our results [49] in the synchronous case for hybrid failure models, namely for systems where processes can exhibit both mortal Byzantine and classic Byzantine faults. One could consider that this models systems where only some of the Byzantine failures are detected even if they are perpetually faulty and not just for a few steps as in our models of Sect. 6. In more detail, assuming $m$ faulty processes are mortal and $i$ faulty processes are classic (immortal) Byzantine in a system of $n$ processes, they show that consensus is solvable in synchronous systems if and only if $n > 3i + 2m$. More precisely, they give an algorithm for binary consensus that works with $n > 3i + 2m$, and they show that the Byzantine Generals problem [38] cannot be solved with $n = 3i + 2m$. By setting $i = 0$, one observes that in the more general hybrid setting they extend our results regarding consensus solvability. For partially synchronous or asynchronous systems, however, our lower bound result from Sect. 5.1 suggests that using such a hybrid approach will not lead to consensus algorithms with improved resiliency.

Bazzi and Herlihy [8] also investigated the problem of broadcasting in the mortal Byzantine failure model (in the absence of classic Byzantine faults). They discussed that if all Byzantine processes are eventually detected one can solve a weak form of terminating reliable broadcast if $n > m$. In this weak form, processes are only required to deliver the broadcaster's value if all faulty processes have crashed before the broadcast was initiated. Further they introduced a pipelining mechanism, in which by repeatedly using this weak broadcasting protocol, one can solve terminating reliable broadcast (TRB), that is, the Byzantine Generals problem [38], without any requirement on the number of mortal Byzantine failures.

For classic Byzantine failures, the TRB and consensus problems are usually considered to be equivalent, thus their result seems to conflict with our $2t + 1$ lower bound on the number of processes. In fact it does not, as these two results together give a very precise characterization on which safety properties is harder to maintain in presence of mortal Byzantine failures: recall that we only used the validity property of consensus in the proof of Theorem 6; a property that requires that in case all correct processes initially agree, their value is the only legal decision value. It is in some sense evident that this property requires a majority of correct processes. In contrast, the validity property of TRB only requires a process to decide on the value sent by one process (or to deliver no value, in the case of a faulty broadcaster).

It is also worth mentioning, that this result shows that consensus is harder than TRB with respect to resilience to mortal Byzantine failures. As is observed in [13], the opposite is true with respect to failure detectors (in the presence of benign failures).

Nesterenko and Arora study malicious crashes [43]. The assumptions on malicious crashes are basically the same as for mortal Byzantine failures. However, their work is in a different context. On the one hand, they consider shared registers as communication medium which contrasts our message passing models. On the other hand, they study a different problem, namely, dining philosophers. Their solution uses techniques from self-stabilization to ensure that the system is able to recover or at least contain the disorder caused by the malicious behavior and eventual crash of faulty processes. The maximal distance from a faulty process to a process that is affected by the fault is called the failure locality. Clearly, ensuring total recovery of the system after all (mortal Byzantine) faulty processes have crashed would amount to guaranteeing failure locality 0, which is impossible to achieve in asynchronous systems as the minimal failure locality of any dining philosophers algorithm is known to be 2 [16].

Similarly, [50] studies self-stabilizing leader election in rings in the presence of mortal Byzantine failures. In their solution the failure locality depends on the number of arbitrary state changes a process may perform.

## 8 Discussions

We studied a fault model that seems to accurately describe faulty behavior as experienced in certain practical applications. Our motivating examples were from the space domain, but our fault model may also be used to describe human-managed systems, where upon observing a deviation from normal behavior by some computer, the operator shuts it down. The model lies between crash model and Byzantine model as faulty processes are allowed to behave Byzantine until they eventually crash.

The notion of failure mode trajectory seems to be an interesting issue for future work. We considered only the extreme case where faulty processes may start to exhibit the most general failure mode (Byzantine) and eventually converge to a very benign fault, i.e., they crash. In the synchronous case, we showed that $n > 2t$ is necessary and sufficient to solve consensus, and it is thus possible to tolerate more faults than in the (classic) Byzantine case. In general, it should be possible to devise more efficient solutions (e.g., regarding synchrony assumptions or required number of processes) by replacing a static fault model with a model where the behavior of faulty processes converges towards some benign fault.

When considering failure mode trajectories, algorithms typically have to be safe in the presence of the most severe failure mode within the trajectory. Our synchronous algorithm and our asynchronous algorithm are safe even in the presence of (classic) Byzantine faults. In order to guarantee decision, processes have to crash, however.

In the synchronous case, we also considered more refined models where processes may exhibit a bounded number of faults without ever crashing, but if they do exceed this bound, they crash within a known number of rounds. These models allow to deal with certain transient faults.

In this paper we considered only binary consensus, that is, the set of possible decision values is $\{0,1\}$. For the lower bounds in this paper, this is a safe choice as considering binary consensus gives stronger results than in the multi-valued case.

Considering upper bounds, our asynchronous algorithm can easily be adapted to solve multi-valued consensus by changing the way the proper set is handled similar to [27]. There are also general techniques to extend algorithms for binary consensus to solve the multi-valued version of the problem. Most notably, for the Byzantine case there is the synchronous algorithm by Turpin and Coan [48], which requires $n > 3t$. As we only have $n > 2t$ in the synchronous case, the results by Turpin and Coan do not directly imply that our binary consensus algorithms also provide an upper bound result for multi-valued consensus. However, the following simple arguments close this gap.

As mentioned in Section 3, an alternative to our Validity condition that is also suitable for multi-valued consensus would be to require "if all correct processes propose $v$, and a correct process decides $w$, then $v = w$." We will now explain how to achieve consensus with this Validity property when $v$ can be from a some set $V$ with $|V| > 2$, based on our algorithms. Processes start one instance of binary consensus for each $v \in V$, and use 1 as initial value if $v$ is their initial value, otherwise they use 0 for the instance corresponding to $v$. After a correct process has decided in all $|V|$ instances, it decides on the minimal $v$ such that the binary consensus instance corresponding to $v$ decided on 1, or some default value if all instances decided on 0. Agreement, Decision and Halting of this simple protocol follow directly from the corresponding properties of the binary protocol. Moreover, when all processes propose $v$ then all correct processes propose 1 for the instance corresponding to $v$, and 0 in all other instances. According to the Validity property, our algorithms guarantee the decision value of each instance is the initial value of some correct process for that instance. Consequently, the processes decide on 1 for the instance corresponding to $v$, and 0 for all other instances. Clearly the decision value of our reduction will be $v$, thus satisfying the Validity property given above.

This simple reduction needs $|V|$ instances when the initial values are taken from the set $V$. An alternative approach would be to instead reach agreement over every bit of the initial values, thus reducing the number of instances to $\log |V|$. Similar bit-per-bit approaches that work only for benign failures are used in the reduction algorithms in [42,51].

Authentication can be used "in practice" to reduce the required number of processes [38]. In this paper, we have chosen not to consider authenticated algorithms, because in addition to the lack of a formal definition of authentication in the presence of Byzantine faults, and the disadvantage of computational and communication overhead, there is also the possibility that authentication can be broken.

## Acknowledgments

We are grateful to Danny Dolev for pointing out [33] to us, and for enlightening us about the relation between Crusader agreement and consensus. We thank Rida Bazzi and Maurice Herlihy for valuable discussions on their results [8]. We also thank the anonymous reviewers whose constructive comments helped to improve the organization of the paper significantly.

## References

1. Abd-El-Malek, M., Granger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. In: 20th ACM Symposium on Operating Systems Principles (SOSP'05), pp. 59–74 (2005)
2. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. Distributed Computing **13**(2), 99–125 (2000)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with Byzantine failures and little system synchrony. In: DSN '06: Proceedings of the International Conference on Dependable Systems and Networks, pp. 147–155. IEEE Computer Society, Washington, DC, USA (2006). DOI 10.1109/DSN.2006.22
4. Anceaume, E., Delporte-Gallet, C., Fauconnier, H., Hurfin, M., Le Lann, G.: Designing modular services in the scattered Byzantine failure model. In: 3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004), pp. 262–269. IEEE Computer Society (2004)
5. Anceaume, E., Delporte-Gallet, C., Fauconnier, H., Hurfin, M., Widder, J.: Clock synchronization in the Byzantine-recovery failure model. In: International Conference On Principles Of DIstributed Systems OPODIS 2007, LNCS, pp. 90–104. Springer Verlag, Guadeloupe, French West Indies (2007)
6. Azadmanesh, M.H., Kieckhafer, R.M.: New hybrid fault models for asynchronous approximate agreement. IEEE Transactions on Computers **45**(4), 439–449 (1996)
7. Baldoni, R., Hélary, J.M., Raynal, M., Tangui, L.: Consensus in Byzantine asynchronous systems. Journal of Discrete Algorithms **1**(2), 185–210 (2003)
8. Bazzi, R.A., Herlihy, M.: Enhanced fault-tolerance through Byzantine failure detection. In: 13th International Conference on Principles of Distributed Systems (OPODIS), *LNCS*, vol. 5923, pp. 129–143. Springer (2009)
9. Biely, M.: An optimal Byzantine agreement algorithm with arbitrary node and link failures. In: Proc. 15th Annual IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'03), pp. 146–151. Marina Del Rey, USA (2003)
10. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM **32**(4), 824–840 (1985)
11. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: 3rd Symposium on Operating Systems Design and Implementation (1999)
12. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43**(4), 685–722 (1996)
13. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2), 225–267 (1996)
14. Charron-Bost, B., Hutle, M., Widder, J.: In search of lost time. Information Processing Letters **110**(21), 928–933 (2010)
15. Charron-Bost, B., Schiper, A.: Uniform consensus is harder than consensus. J. Algorithms **51**(1), 15–37 (2004)
16. Choy, M., Singh, A.K.: Efficient fault tolerant algorithms for resource allocation in distributed systems. In: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, STOC '92, pp. 593–602. ACM, New York, USA (1992)
17. Correia, M., Neves, N.F., Lung, L.C., Veríssimo, P.: Low complexity Byzantine-resilient consensus. Distributed Computing **17**, 237–249 (2005)
18. Correia, M., Neves, N.F., Veríssimo, P.: From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. The Computer Journal **49**(1), 82–96 (2006)
19. Delporte-Gallet, C., Fauconnier, H., Freiling, F.C., Penso, L.D., Tielmann, A.: From crash-stop to permanent omission: Automatic transformation and weakest failure detectors. In: 21st International Symposium on Distributed Computing (DISC), *LNCS*, vol. 4731, pp. 165–178. Springer Verlag (2007)
20. Delporte-Gallet, C., Fauconnier, H., Horn, S.L., Toueg, S.: Fast fault-tolerant agreement algorithms. In: Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05), pp. 169–178. ACM Press, New York, USA (2005)
21. Dijkstra, E.W.: On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer-Verlag (1982). EWD 447
22. Dolev, D.: The Byzantine generals strike again. Journal of Algorithms **3**(1), 14–30 (1982)
23. Dolev, D., Reischuk, R., Strong, H.R.: Early stopping in Byzantine agreement. Journal of the ACM **37**(4), 720–741 (1990)
24. Doudou, A., Garbinato, B., Guerraoui, R.: Encapsulating failure detection: From crash to Byzantine failures. In: Reliable Software Technologies - Ada-Europe 2002, LNCS 2361, pp. 24–50. Springer, Vienna, Austria (2002)
25. Doudou, A., Garbinato, B., Guerraoui, R., Schiper, A.: Muteness failure detectors: Specification and implementation. In: Proceedings 3rd European Dependable Computing Conference (EDCC-3), *LNCS 1667*, vol. 1667, pp. 71–87. Springer, Prague, Czech Republic (1999)
26. Doudou, A., Schiper, A.: Muteness detectors for consensus with Byzantine processes. In: Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17). Puerto Vallarta, Mexico (1998)
27. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2), 288–323 (1988)
28. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Science of Computer Programming **2**(3), 155–173 (1982)
29. Fischer, M., Lamport, L.: Byzantine generals and transaction commit protocols. Technical Report 62, SRI International (1982)
30. Fischer, M.J., Lynch, N.: A lower bound for the time to assure interactive consistancy. Information Processing Letters **14**(4), 198–202 (1982)

31. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. In: Proceedings of the fourth annual ACM symposium on Principles of distributed computing, PODC '85, pp. 59–70. ACM, New York, USA (1985)
32. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2), 374–382 (1985)
33. Fitzi, M., Maurer, U.M.: From partial consistency to global broadcast. In: STOC, pp. 494–503 (2000)
34. Friedman, R., Mostéfaoui, A., Raynal, M.: Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. IEEE Transactions on Dependable and Secure Computing **2**(1), 46–56 (2005)
35. Hermant, J.F., Le Lann, G.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51**(8), 931–944 (2002)
36. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Solving consensus in a Byzantine environment using an unreliable fault detector. In: Proceedings of the International Conference on Principles of Distributed Systems (OPODIS), pp. 61–75. Chantilly, France (1997)
37. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine fault detectors for solving consensus. The Computer Journal **46**(1), 16–35 (2003)
38. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (1982)
39. Lynch, N.: Distributed Algorithms. Morgan Kaufman Publishers, Inc., San Francisco, USA (1996)
40. Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Proceedings of the 10th Computer Security Foundations Workshop (CSFW97), pp. 116–124. Rockport, MA, USA (1997)
41. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Transactions on Dependable and Secure Computing **3**(3), 202–215 (2006)
42. Mostefaoui, A., Raynal, M., Tronel, F.: From binary consensus to multivalued consensus in asynchronous message-passing systems. Information Processing Letters **73**(5-6), 207 – 212 (2000)
43. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), pp. 191–198. Vienna, Austria (2002)
44. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM **27**(2), 228–234 (1980)
45. Perry, K.J., Toueg, S.: Distributed agreement in the presence of processor and communication faults. IEEE Transactions on Software Engineering **SE-12**(3), 477–482 (1986)
46. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing **2**, 80–94 (1987)
47. Thambidurai, P.M., Park, Y.K.: Interactive consistency with multiple failure modes. In: Proceedings 7th Symposium on Reliable Distributed Systems, pp. 93–100 (1988)
48. Turpin, R., Coan, A.B.: Extending binary Byzantine agreement to multivalued Byzantine agreement. Information Processing Letters **18**(2), 73–6 (1984)
49. Widder, J., Gridling, G., Weiss, B., Blanquart, J.P.: Synchronous consensus with mortal Byzantines. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN'07), pp. 102–111. Edinburgh, UK (2007)
50. Yamauchi, Y., Masuzawa, T., Bein, D.: Adaptive containment of time-bounded Byzantine faults. In: 12th International Symposium Stabilization, Safety, and Security of Distributed Systems (SSS 2010), *LNCS*, vol. 6366, pp. 126–140. Springer Verlag (2010)
51. Zhang, J., Chen, W.: Bounded cost algorithms for multivalued consensus using binary consensus instances. Information Processing Letters **109**(17), 1005 – 1009 (2009)