Consensus with Byzantine Failures and Little System Synchrony

Marcos K. Aguilera
HP Laboratories
Palo Alto, California, USA

Carole Delporte-Gallet
IGM, ESIEE
Paris, France

Hugues Fauconnier
LIAFA, Université Paris 7
Paris, France

Sam Toueg
University of Toronto
Toronto, Ontario, Canada

Abstract

We study consensus in a message-passing system where only some of the n^2 links exhibit some synchrony. This problem was previously studied for systems with process crashes; we now consider byzantine failures. We show that consensus can be solved in a system where there is at least one non-faulty process whose links are eventually timely; all other links can be arbitrarily slow. We also show that, in terms of problem solvability, such a system is strictly weaker than one where all links are eventually timely.

1 Introduction

The consensus problem is at the core of fault-tolerant distributed systems. However, solving consensus is impossible in asynchronous systems subject to process failures [9]. A well-known way to overcome this impossibility is to make partial synchrony assumptions about the system [6, 8]. In particular, from [8], it follows that consensus is possible in a system where the relative speeds of processes are bounded, and *all* links are *eventually timely*, that is, there exists a value Δ and a time T such that messages sent after time T are delayed by at most Δ by a link. This possibility result holds for a system S_{crash} with crash failures and a system S_{byz} with byzantine failures, with a resiliency of $n \geq 2f+1$ and $n \geq 3f+1$, respectively, where n is the number of processes and f is the maximum that can fail.

To solve consensus, is it really necessary that *all* links be eventually timely? What if only *some* links are eventually timely, while other links can be arbitrarily slow; can consensus still be solved? If so, how? Furthermore, do the answers to these questions depend on the type of process failures (crash versus byzantine failures)?

For crash failures, [1, 2] considered systems with a varying number of eventually timely links. [1] shows that consensus is possible in a system where $n \geq 2f+1$ and there is at least one unknown non-faulty process whose $n-1$ *outgoing* directed links are eventually timely (its incoming links can be arbitrarily slow); thus, only $\Theta(n)$ links in the system are eventually timely, and all other $\Theta(n^2)$ links can be arbitrarily slow. Later, [2] has shown that consensus is possible in an even weaker system, where $n \geq 2f+1$ and there is at least one unknown non-faulty process whose f outgoing direct links are eventually timely (and it is not known which f links are those); thus, only $\Theta(f)$ links in the system are eventually timely. For $f = 1$, this result implies that consensus can be solved even if only one unknown directed link in the system is eventually timely.

The above results are for systems with crash failures. In this paper, we consider byzantine failures, in which a failed process may behave arbitrarily. Specifically, we consider a system S'_{byz} where there exists some unknown non-faulty process whose incoming and outgoing links are all eventually timely. We show how to solve consensus in S'_{byz} when $n \geq 3f+1$, which is the maximum possible resiliency.

One may believe that system S'_{byz} (where only the links to and from a correct process are eventually timely) and system S_{byz} (where *all* the links are eventually timely) are equivalent in terms of problem solvability.[1] In fact, it may seem that processes in system S'_{byz} can simulate system S_{byz} by using message flooding: flooding ensures that every message is relayed to its destination by the (unknown) non-faulty process whose incoming and outgoing links are eventually timely, so flooding in S'_{byz} ensures eventually timely communication between *every* pair of processes, like in S_{byz}.

[1]If that were true, the possibility of consensus in S_{byz} shown in [8] would immediately imply the possibility of consensus in S'_{byz}. In other words, [8] would imply the possibility result that we show in this paper.

However, this belief is incorrect. To show this, we exhibit a problem, namely, *eventual timely broadcast*, which can be solved in S_{byz} but not in S'_{byz}. Thus, S'_{byz} is strictly weaker than S_{byz}. The rough intuition is that it is impossible to distinguish in a timely fashion the relaying of a real message by an honest process from the relaying of a fake message by a byzantine process.

This impossibility immediately implies that *timely reliable broadcast* or *timely atomic broadcast* [11] are also impossible in S'_{byz}.

To the best of our knowledge, the algorithm presented here is the first to solve consensus with byzantine failures and few eventually timely links. Related work includes [3] and [10], which propose a modular approach to solve consensus in a system with byzantine failures and an oracle that can indicate if a process chooses to deviate from its protocol by refusing to send a message. Such an oracle not only encapsulates the synchrony of the system, but also the expected behavior of a process; it needs to distinguish a byzantine process that refuses to send a message when it should, from a correct process that does not send a message because its protocol does not prescribe to do so. An implementation of this oracle is given for round-based algorithms in system S_{byz}, where all links are eventually timely [7], but not in the model S'_{byz} that we consider, where only few links are eventually timely. The implementation requires correct processes to start a round within bounded time of each other, so that a process can use a timeout from the beginning of its round to accurately suspect processes that refuse to send messages in that round. It is not clear how to change this implementation to work in S'_{byz}. First, by Theorem 24, it is not possible to simulate S_{byz} with S'_{byz}. Second, Theorem 24 also implies that for a round-based algorithm in system S'_{byz} with $n \geq 3$ and $n - f \geq 2$, it is impossible to ensure that correct processes start a round within bounded time of each other—violating a key need of the implementation in [7].[2]

[2] To further illustrate the difficulty in implementing the oracle of [7] in S'_{byz}, consider a correct process p whose incoming and outgoing links are all timely, but other links in the system are not. Then, in a round-based protocol, process p finishes round 1 in a timely fashion as p receives timely round 1 messages from all correct processes. Now suppose that a second correct process q remains stuck in round 1 as its incoming links are slow. Suppose further that all other correct processes finish round 1 and start round 2. Then, it is possible that some third correct process r finishes round 2 by receiving $n - f - 1$ messages from correct process other than q and one message from a byzantine process, while p is still in round 2 since it only receives $n - f - 1$ messages. In fact, p may remain in round 2 for an unbounded amount of time while q is stuck in round 1. In round 3, r does not hear from p and will eventually time out on and suspect p, even though p is a correct process whose incoming and outgoing links are all eventually timely.

2. Model

We consider a message-passing system, where a set $\Pi = \{1, \ldots, n\}$ of processes communicate with each other by sending point-to-point messages over a network.

Each process is an infinite state automaton that computes by taking steps. In each step, a process may perform the following actions in order: (1) it may send a message to one of more processes, or it may issue an external output, (2) it may receive a message from some process, (3) it may receive an external input, and (4) it may change its state.

A process may fail by being *byzantine*, in which case it may behave arbitrarily, rather than according to its automaton. In particular, the process may behave in a way that tries to maxime the damage to the system. A *correct* process is one that is not byzantine. Value f denotes an upper bound on the number of byzantine processes.

Links are unidirectional and there is a link connecting every pair of processes. The link from process p to process q is denoted $p \to q$. Every link is reliable: it does not create, duplicate, or lose messages.

The system is partially synchronous, in that (1) there are unknown bounds on the minimum and maximum delays between two steps of a correct process, (2) some links in the system may be eventually timely. We also assume that processes can measure intervals of real time; this is only to simplify the presentation of our proofs; it is not required for our results. A link $p \to q$ is *eventually timely* if there exists a constant δ and a time T_0 (stabilization time) such that if some correct process p sends m to some correct process q at time t then q receives m by time $\max\{t, T_0\} + \delta$. Intuitively, messages sent after T_0 are received within δ time, while messages sent before T_0 are received by time $T_0 + \delta$.

We do not require every link to be eventually timely; rather, we typically only assume that there exists some correct process s whose incoming and outgoing links are eventually timely. Such a process s is called a *bisource*.

3. Consistent unique broadcast

Our consensus algorithm uses *consistent unique broadcast* as subroutine, which is a broadcast very similar to consistent broadcast [5] and authenticated broadcast [12]. Messages have a tag, and consistent unique broadcast ensures that (1) correct processes deliver the same set of messages, and (2) a correct process delivers at most one message with a given tag. Intuitively, tags are used to ensure that a byzantine process does not broadcast two different messages in the same round.

More precisely, consistent unique broadcast is defined by two primitives, *cubcast*(X, k, v) and *cudeliver*(X, k, v, q) where, intuitively, (X, k) is a tag, and v is a value. If a process p invokes *cubcast*(X, k, v) (resp., *cudeliver*(X, k, v, q)), we say that p *cubcasts*(X, k, v) (resp, p *cudelivers*(X, k, v) *from* q). We assume that a correct process *cubcast* at most once for any given X, k; in particular, it does not *cubcast* both (X, k, v) and (X, k, v'). Consistent unique broadcast ensures the following:

- *(Validity)* If a correct process p *cubcasts* (X, k, v) then all correct processes eventually *cudeliver* (X, k, v) from p;
- *(Unforgeability)* If a correct process p does not *cubcast* (X, k, v) then no correct process ever *cudelivers* (X, k, v) from p;
- *(Uniqueness)* For each X, k and q, a correct process *cudelivers* at most one message of form $(X, k, *)$ from process q;
- *(Relay)* If a correct process *cudelivers* (X, k, v) from a process p then all correct processes eventually *cudeliver* (X, k, v) from p.

Consistent unique broadcast can be implemented as described in [5, 12]. For convenience we include the implementation in Figure 1; correctness proofs are in [5, 12].

4. Provable reliable send

Our consensus algorithm also uses a new primitive called *provable reliable send*. Roughly speaking, it can be used for a process p to send a message m to q such that a third party gets a proof that m is in transit. The primitive guarantees that if p is correct then all correct processes r gets the proof that m is in transit, and if a correct process r gets the proof that m is in transit, and q is correct, then q receives m. There are also eventual timeliness properties that guarantee that if the receiver is a bisource then the message cannot be received too much later than the proof.

We now give a precise definition and implement provable reliable send.

4.1. Specification

Provable reliable send is defined by primitives *psend*(m, q), *preceive*(p, m), and *getproof*(p, m, q). If a process p invokes *psend*(m, q) we say that p *psends* m *to* q. If a process q invokes *preceive*(p, m) we say that q *preceives* m *from* p. If a process r invokes *getproof*(m, p, q) we say that r *getsproof of* m *from* p *to* q. Provable reliable send ensures the following:

- *(Integrity)* A correct process q preceives m from a correct process p at most once, and only if p has previously psent m to q;
- *(Validity)* If some correct process p psends m to some correct process q then eventually q preceives m from p;
- *(Proof-Integrity)* If some correct process r getsproof of m from some process p to some correct process q then q preceives m from p;
- *(Proof-Validity)* If some correct process p psends m to some process q then every correct process r getsproof of m from p to q.

We also consider *eventually timely provable reliable send*, which ensures that if process q is a bisource then eventually a message m to q is received within Δ' of a correct process getting proof of m. More precisely, we have the following:

- *(Eventual timeliness)* If process q is a bisource then there exists Δ' and T' such that if some correct process r getsproof of m from some process p to process q at time t then q preceives m from p by time $\max\{t, T'\} + \Delta'$.

Intuitively, if r getsproof of m after time T' then q preceives m within Δ' time, while if r getsproof of m before time T' then q preceives m by time $T' + \Delta'$.

4.2. Implementation

Figure 2 shows an implementation of provable reliable send. To psend a message m to *dst*, a process *src* sends $(\text{PSEND}, m, \textit{src}, \textit{dst})$ to all processes. When a process p receives $(\text{PSEND}, m, \textit{src}, \textit{dst})$ from s, it checks if s is the origin of the message ($s = \textit{src}$) and $p \neq s$. If both conditions are true, p sends $(\text{PSEND}, m, \textit{src}, \textit{dst})$ to all. The reason for checking that $p \neq s$ is to avoid having s send this message to all multiple times. Then p checks that q is the ultimate destination of m ($\textit{dst} = p$) and that q has received $(\text{PSEND}, m, \textit{src}, \textit{dst})$ from at least $f + 1$ processes—and hence from at least one correct process. If so, p preceives m from *src*, if it has not done so already. Then, p checks if it received $(\text{PSEND}, m, \textit{src}, \textit{dst})$ from at least $2f + 1$ processes—and hence from at least $f + 1$ correct processes. If so, p getsproof of m from *src*.

We have the following theorems:

Theorem 1 *Consider a system with byzantine failures and $n \geq 3f+1$. The algorithm in Figure 2 implements provable reliable send.*

Theorem 2 *Consider a system with byzantine failures such that $n \geq 3f + 1$ and there exists at least one correct process whose outgoing and incoming links are eventually timely. The algorithm in Figure 2 implements eventually timely provable reliable send.*

```
Code for each process $p$:

to $cubcast(X, k, v)$ :
    send $(\text{INIT}, X, k, v, p)$ to all processes

upon receive $(\text{INIT}, X, k, v, q)$ from $q$ do
    if no $(\text{ECHO}, X, k, *, q)$ sent before by $p$ then send $(\text{ECHO}, X, k, v, q)$ to all

upon receive $(\text{ECHO}, X, k, v, q)$ from $(n + f)/2$ different processes do
    if no $(\text{ECHO}, X, k, *, q)$ sent before by $p$ then send $(\text{ECHO}, X, k, v, q)$ to all
    if no $(\text{READY}, X, k, *, q)$ sent before by $p$ then send $(\text{READY}, X, k, v, q)$ to all

upon receive $(\text{READY}, X, k, v, q)$ from $f + 1$ different processes do
    if no $(\text{ECHO}, X, k, *, q)$ sent before by $p$ then send $(\text{ECHO}, X, k, v, q)$ to all
    if no $(\text{READY}, X, k, *, q)$ sent before by $p$ then send $(\text{READY}, X, k, v, q)$ to all

upon receive $(\text{READY}, X, k, v, q)$ from $n - f$ different processes do
    if $(X, k, v)$ not already cudelivered from $q$ then $cudeliver(X, k, v, q)$
```

Figure 1. Implementation of consistent unique broadcast in a system with $n \geq 3f + 1$ [5, 12].

```
Code for each process $p$:

To psend $m$ to $q$:
    send $(\text{PSEND}, m, p, q)$ to all processes

upon receive $(\text{PSEND}, m, src, dst)$ from $s$ do
    if $src = s$ and $p \neq s$ then send $(\text{PSEND}, m, s, dst)$ to all processes     (* relay to all *)
    if $dst = p$ and received $(\text{PSEND}, m, src, dst)$ from $f + 1$ processes
            and not already preceive$(src, m)$
    then $preceive(src, m)$
    if received $(\text{PSEND}, m, src, dst)$ from $2f + 1$ processes
    then $getproof(m, src, dst)$     (* know that $f + 1$ processes relay to $dst$ *)
```

Figure 2. Implementation of provable reliable send in a system with $n \geq 3f + 1$.

We now prove the above theorems. Assume that $n \geq 3f+1$.

Lemma 3 *(Integrity) A correct process q preceives m from a correct process p at most once, and only if p has previously psent m to q.*

Proof. The fact that q preceives at most once is because q always checks if it previously preceived before preceiving. Now suppose that q preceives m from p. Then q receives (PSEND, m, p, q) from $f+1$ processes. Thus, q receives (PSEND, m, p, q) from at least one correct process s. As links are reliable and q and s are correct, s sends (PSEND, m, p, q) to q. Thus, either $s = p$ or s receives (PSEND, m, p, q) from p. In the first case, p psent m to q. In the second case, as links are reliable and s and p are correct, p sends (PSEND, m, p, q) to s, and so p psent m to q. $\Box$

Lemma 4 *If some correct process p psends m to some process q then every correct process sends (PSEND, m, p, q) to all processes.*

Proof. Suppose that some correct process p psends m to some process q. Then, p sends (PSEND, m, p, q) to all correct processes. When a correct process $q \neq p$ receives such a message, it also sends (PSEND, m, p, q) to all processes. Therefore all correct processes send (PSEND, m, p, q) to all processes. $\Box$

Lemma 5 *(Validity) If some correct process p psends m to some correct process q then eventually q preceives m from p.*

Proof. Suppose that some correct process p psends m to some correct process q. By Lemma 4, every correct process sends (PSEND, m, p, q) to q. Since there are at least $2f+1$ correct processes, q eventually receives (PSEND, m, p, q) from $f+1$ processes and q preceives m from p. $\Box$

Lemma 6 *(Proof-Integrity) If some correct process r getsproof of m from some process p to some correct process q then q preceives m from p.*

Proof. Suppose that some correct process r getsproof of m from some process p to some correct process q. Then r receives (PSEND, m, p, q) from $2f+1$ processes. As links are reliable and at most f processes are byzantine, we have that at least $f+1$ correct processes sent (PSEND, m, p, q) to r. When sending such a message, each correct process sends to all processes. Therefore, at least $f+1$ correct processes send (PSEND, m, p, q) to q, and so q preceives m from p. $\Box$

Lemma 7 *(Proof-Validity) If some correct process p psends m to some process q then every correct process r getsproof of m from p to q.*

Proof. Suppose that some correct process p psends m to some process q, and let r be some correct process. By Lemma 4, every correct process sends (PSEND, m, p, q) to r. As links are reliable and at least $2f+1$ processes are correct, we have that r receives (PSEND, m, p, q) from at least $2f+1$ processes. Therefore, r getsproof of m from p to q. $\Box$

Lemma 8 *(Eventual timeliness) If process q is a bisource then there exists Δ' and T' such that if some correct process r getsproof of m from some process p to process q at time t then q preceives m from p by time $\max\{t, T'\} + \Delta'$.*

Proof. (Similar to proof of Proof-Integrity) Suppose that q is a bisource. Then all incoming and outgoing links of q are eventually timely, and so there exists δ and T_0 such that if some correct process p sends m to q at a time t then q receives m by time $\max\{t, T_0\} + \delta$. Assume that some correct process r getsproof of m from some process p to process q at some time t. Then r receives (PSEND, m, p, q) from $2f+1$ processes. As r is correct and links are reliable, at least $f+1$ correct processes sent (PSEND, m, p, q) to r by time t. When a correct process sends (PSEND, m, p, q) to r, it also sends this message to all processes and in particular to q, and this happens by time t. Since q is a bisource, by time $\max\{t, T_0\} + \delta$, q receives (PSEND, m, p, q) from every correct process, and hence from $f+1$ processes. When this happens, q preceives m from p if it has not done so already. $\Box$

Proof of Theorem 1. Integrity follows from Lemma 3. Validity follows from Lemma 5. Proof-Integrity follows from Lemma 6. Proof-Validity follows from Lemma 7. $\Box$

Proof of Theorem 2. Integrity,Validity, Proof-Integrity and Proof-Validity follow from Theorem 1. Eventual timeliness follows from Lemma 8. $\Box$

5. Consensus

We consider the binary consensus problem, where every correct process proposes some value in $\{0, 1\}$ and must make an irrevocable *decision* on a value such that

- *(Agreement)* No two correct processes decide differently;
- *(Validity)* If some correct process decides v, then v is proposed by some correct process;
- *(Termination)* Every correct process eventually decides some value.

Figure 3 shows an implementation of binary consensus for a system byzantine failures with $n \geq 3f+1$ and at least one bisource. The algorithm was originally inspired by Ben-Or's randomized algorithm [4], with many modifications; it uses as subroutines consistent unique broadcast and eventually timely provable reliable send.

Description. Each process p keeps a current estimate of the decision value, which is initially the value that p proposes to consensus. The algorithm proceeds by rounds, where each round has four phases: certification, reporting, proposing, and consulting the coordinator. In the certification phase, p uses consistent unique broadcast to send its estimate to all processes. Process p considers a value v to be *certified* if p delivers at least $f+1$ certification messages for v. This guarantees that at least one such a message comes from a correct process. Process p waits to deliver certification messages from $n-f$ processes, and then p changes its estimate to the value that was delivered most.

In the reporting phase, p broadcasts its estimate v (we say p *reports* v). Then p waits to deliver $n-f$ messages for values that are certified. As time passes, this phase can complete not because p receives further report messages, but because p receives certification messages that causes a value to become certified. Then p picks the value w that appears in most messages.

In the proposing phase, p checks if all report messages for a certified value is for w and, if so, p broadcasts w (we say p *proposes* w); else p proposes ?. Then p waits for delivery of $n-f$ proposal messages for either w or ? if $1-w$ is certified.

In the consulting phase, p first determines whether to change its estimate and whether to accept the coordinator's value, according to four cases based on how many proposals p delivers for a value $x \neq$?:

- *Case 1: p delivers $n-f$ proposals for x:* In this case, p decides x and does not accept the coordinator's value.
- *Case 2: p delivers $n-2f$ proposals for x:* In this case, p changes its estimate to x and does not accept the coordinator's value.
- *Case 3: p delivers $n-3f$ proposals for x:* In this case, p changes its estimate to x, and later accepts the coordinator's value if p does not time out on the coordinator.
- *Case 4: p delivers less than $n-3f$ proposals for x:* In this case p accepts the coordinator's value if p does not time out on the coordinator.

Then p sends its current estimate to the coordinator using provable reliable send. The coordinator rotates with each round: for round k, the coordinator is process $k \bmod n$. When p knows that at least $n-f$ processes have sent their estimates to the coordinator, p starts a timer. If p receives the coordinator's value within a timeout period, and p previously decided to accept its value, then p changes its estimate to the received value. On the other hand, if p times out on the coordinator, p increases the timeout for the future, in case it was a premature timeout.

The coordinator waits to receive $n-f$ estimates from processes, picks the value that occurs most, and sends back this value to processes.

Intuition. The algorithm uses various mechanisms to solve consensus:

- Like Ben-Or's algorithm, the reporting and proposing phases ensure that at most one non-? can be sent in the propose phase by correct processes. This ensures that, in each round, processes attempt to decide on only one value, which is important for agreement.
- Unlike Ben-Or's algorithm, we use an extra certification phase to ensure that if all processes start with the same value z, then reports for $1-z$ (from byzantine processes) are ignored.
- We use consistent unique broadcast to ensure that byzantine processes cannot propose two different values in the same phase.
- We replace the random coin tosses of Ben-Or's algorithm with a coordinator in the consulting phase.
- In the consulting phase, processes can fall in one of the four cases described before, according to how many times they see a non-? value in the proposing phase. Ben-Or's algorithm only has three cases. With only three cases, our algorithm would not work.
- Finally, because coordinators can be byzantine, we have the following problem: (a) a byzantine coordinator may pretend that it never receives messages, and processes waiting for the coordinator must eventually time out, (b) however, a correct process p cannot start the timeout timer as soon as p asks help to the coordinator, because the coordinator may be correct but other processes may be lagging behind in previous rounds, so that the coordinator will not get enough estimates to respond; so, p must wait until p knows that the other correct processes have also sent their estimate to the coordinator. To do that, processes could try to broadcast their estimates to the coordinator; then p can start its timer when p delivers $n-f$ such broadcasts. But this idea requires the broadcast to be timely: the coordinator also needs to receive the $n-f$ messages in a bounded amount of time. As we show, such timely reliable broadcast cannot be implemented in a system with only one bisource. We solve this problem with eventually timely provable reliable send, which allows processes to know when $n-f$ messages are on their way to the coordinator in a timely fashion when the coordinator is a bisource.

```
Code for process $p$:

  Initialization:
    *Timeout* $\leftarrow 1$

  **function** *certified*$(k)$
1   **return** $\{w$ : *cudelivered* $($CERTIFY$, k, w)$ from at least $f + 1$ processes $\}$

  To propose$(v)$:
2   $k \leftarrow 0$
3   **while** true **do**
4     $k \leftarrow k + 1$
        (* phase 0: certification *)
5     *cubcast*$($CERTIFY$, k, v)$
6     **wait until** *cudeliver* $($CERTIFY$, k, *)$ from $n - f$ processes

        (* phase 1: reporting estimates *)
7     $v \leftarrow$ value *cudelivered* most in $($CERTIFY$, k, *)$ messages
8     *cubcast*$($REPORT$, k, v)$
9     **wait until** *cudeliver* $($REPORT$, k, *)$ from $n - f$ processes with $* \in$ *certified*$(k)$

        (* phase 2: proposing the most common estimate *)
10    $w \leftarrow$ value *cudelivered* most in $($REPORT$, k, *)$ messages
11    **if** all $($REPORT$, k, *)$ with $* \in$ *certified*$(k)$ are for $w$
12    **then** *cubcast*$($PROPOSE$, k, w)$
13    **else** *cubcast*$($PROPOSE$, k, ?)$
14    **wait until** *cudeliver* $($PROPOSE$, k, *)$ from $n - f$ processes with $* = w$
        or $(* = ?$ and $1 - w \in$ *certified*$(k))$

        (* phase 3: consulting coordinator *)
15    *accept_coord_new_estimate* $\leftarrow$ *true*
16    **if** *cudelivered* $($PROPOSE$, k, x)$ with $x \neq ?$ from $n - f$ processes **then**
17      **decide** $x$
18      $v \leftarrow x$
19      *accept_coord_new_estimate* $\leftarrow$ *false*
20    **else if** *cudelivered* $($PROPOSE$, k, x)$ with $x \neq ?$ from $n - 2f$ processes **then**
21      $v \leftarrow x$
22      *accept_coord_new_estimate* $\leftarrow$ *false*
23    **else if** *cudelivered* $($PROPOSE$, k, x)$ with $x \neq ?$ from $n - 3f$ processes **then**
24      $v \leftarrow x$
25    *psend* $($HELP-REQ$, k, v)$ to $k \bmod n$
26    **wait until** *getproof* of $($HELP-REQ$, k, *)$ from $n - f$ processes to $k \bmod n$
27    *start_time* $\leftarrow$ *clock*$()$
28    **wait until** received $($HELP-RESP$, k, y)$ from $k \bmod n$ or *clock*$()$ $-$ *start_time* $>$ *Timeout*
29    **if** received $($HELP-RESP$, k, y)$ from $k \bmod n$ **then**
30      **if** *accept_coord_new_estimate* **then** $v \leftarrow y$
31    **else** *Timeout* $\leftarrow$ *Timeout* $+ 1$

    (* coordinator's help *)
  **upon** *preceive* $($HELP-REQ$, k, *)$ from $n - f$ processes **do**
32  $z \leftarrow$ value that occurs most in $($HELP-REQ$, k, *)$ messages
33  send $($HELP-RESP$, k, z)$ to all
```

Figure 3. Implementation of binary consensus in a system with $n \geq 3f + 1$ and at least one bisource.

We now state some key properties for the correctness of the algorithm. Detailed proofs are omitted because of space limitations; they will be included in the full version of the paper.

Lemma 9 *If all correct processes start round k with the same value v then they all decide v in round k.*

Lemma 10 *In round k, if two correct processes propose $v_1 \neq ?$ and $v_2 \neq ?$, respectively, then $v_1 = v_2$.*

Corollary 11 *For each round k, there exists a non-? value v_k such that if a correct process delivers $(\text{PROPOSE}, k, v)$ from a correct process then $v = v_k$ or $v = ?$.*

Lemma 12 *The value w that a correct process p chooses in line 10 is in certified(k) when p executes line 10.*

Lemma 13 *If a correct process p proposes $?$ in round k then at the time of the broadcast, $\{0, 1\} \subseteq$ certified(k) at p.*

Lemma 14 *If $z \in$ certified(k) at some correct process p then eventually $z \in$ certified(k) at every correct process.*

Lemma 15 *In every round k, correct processes do not get stuck in phases 0, 1, 2 or 3.*

As previously observed, a correct process p can be in four cases regarding how it executes the consulting phase: (Case 1) p delivers $n - f$ proposals for a non-? value and executes lines 17–19, (Case 2) p delivers $n - 2f$ proposals for a non-? value and executes lines 21–22, (Case 3) p delivers $n - 3f$ proposals for a non-? value and executes line 24 or (Case 4) p delivers less than $n - 3f$ proposals for a non-? value. We now show that in any given round, correct processes always fall into two consecutive cases.

Lemma 16 *In any round k, all correct processes fall in Cases 1 or 2, or they all fall in Cases 2 or 3, or they all fall in Cases 3 or 4.*

Lemma 17 (Termination) *Every correct process eventually decides some value.*

Lemma 18 *If in round k correct processes p and p' decide x and x', respectively, then $x = x'$.*

Lemma 19 *If a correct process p decides x in round k then all correct processes start round $k + 1$ with their estimates set to x.*

Corollary 20 *If a correct process p decides x in round k then all correct processes decide x in round $k + 1$.*

Lemma 21 (Agreement) *No two correct processes decide differently.*

Corollary 22 (Validity) *If some correct process decides v, then v is proposed by some correct process.*

Theorem 23 *Consider a system with byzantine failures such that $n \geq 3f + 1$ and there exists at least one correct process whose outgoing and incoming links are eventually timely. The algorithm in Figure 3 solves consensus.*

Proof. Agreement follows from Lemma 21. Validity follows from Corollary 22. Termination follows from Lemma 17. □

6. Weakness of having one bisource

With byzantine failures, we now show that a system S'_{byz} with at least one bisource is strictly weaker than a system S_{byz} where all links are eventually timely, in terms of problem solvability. To do so, we show that a type of broadcast, called *eventual timely broadcast*, can be implemented in S_{byz}, but it cannot be implemented in S'_{byz}. This impossibility result also implies that timely reliable broadcast [11] and timely atomic broadcast [11] cannot be implemented in S'_{byz}.

In contrast, with crash failures, *eventual timely broadcast* can be implemented in both a system S_{crash} where all links are eventually timely *and* a system S'_{crash} with at least one bisource.

Eventual timely broadcast is defined by two primitives, *broadcast*(m) and *deliver*(m, p). If p invokes *broadcast*(m), we say that p *broadcasts* m. If q invokes *deliver*(m, p), we say that q *delivers* m *from* p. Eventual timely broadcast ensures the following:

- *(Eventual timely validity)* There exists a time T and a value γ such that, if a correct process p broadcasts m at time $t > T$, then every correct process delivers m from p before time $t + \gamma$;
- *(Unforgeability)* If a correct process p does not broadcast m then no corrrect process delivers m from p.

Theorem 24 *Consider a system with byzantine failures where $n \geq 3$, $n - f \geq 2$, and some correct process has incoming and outgoing links that are eventually timely. There is no implementation of eventual timely broadcast.*

Proof. By way of contradiction, assume there is such an implementation. Construct a run R as follows. Pick some process s to be a bisource, by choosing some δ such as $\delta = 2n$ and making every message sent by s at some time t be received by time $t + \delta$. Pick some process $p \neq s$ to be correct, and let p use the implementation of eventual timely broadcast to broadcast distinct messages infinitely often. Let every message sent from p to a process different from s at some time t be received after time $2t$. Since the implementation of eventual timely broadcast is correct, there exists some time t_0 and a value γ such that every message broadcast by p after time t_0 is delivered within γ time units. Let q be a correct process different from s and p. Since p broadcasts infinitely often, then p broadcasts a message m at some time $t_1 > \max\{t_0, \gamma\}$. Such a message is delivered by q at some time $t_2 \leq t_1 + \gamma$. Moreover, every message sent by p to any process different from s between times t_1 and t_2 is received after time t_2: this is because such messages are received after time $2t_1$ and $2t_1 > t_1 + \gamma \geq t_2$.

Now consider a run R' that is identical to R up to time t_1, but (a) at time t_1, p does not broadcast m, (2) s in R' is a byzantine process (3) all processes except p have the same behavior as in run R up to time t_2, (4) after time t_2, messages to and from p are delayed by at most t_2 time units, so that p is a bisource. For process q, runs R and R' are indistinguishable until time t_2. Thus, q delivers m from p at time t_2, contradicting the Unforgeability property. $\Box$

Since both timely reliable broadcast [11] and timely atomic broadcast [11] can be used to implement eventual timely broadcast, we have the following:

Corollary 25 *Consider a system with byzantine failures where $n \geq 3$, $n - f \geq 2$, and some correct process has incoming and outgoing links that are eventually timely. There is no implementation of timely reliable broadcast or of timely atomic broadcast.*

If all links are eventually timely, then it is trivial to implement eventual timely broadcast: to broadcast m, process p sends (ETB, m, p) to all, and every q delivers (m) when q receives (ETB, m, p) from p. We thus have the following:

Theorem 26 *In a system with byzantine failures where all links are eventually timely, there is an implementation of eventual timely broadcast.*

7 Conclusion

Algorithms that work with general failures and weak assumptions have better coverage than algorithms that work with restricted failures and strong assumptions. In this paper, we studied the implementability of consensus with byzantine failures and with weak assumptions on synchrony. We have shown that consensus is possible in system S'_{byz}, in which there exists some unknown non-faulty process whose incoming and outgoing links are all eventually timely. We have also shown that S'_{byz} is less powerful than system S_{byz}, in which *all* the links are eventually timely. Some open problems regarding consensus remain in systems with byzantine failures: can consensus be solved if there exists at least one non-faulty process s whose $n - 1$ *outgoing* links are all eventually timely? How about if s has only f outgoing links that are eventually timely? (Recall that with *crash failures*, this condition is sufficient to solve consensus.) We conjecture that the answer is negative in both cases.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 306–314, July 2003.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23nd ACM Symposium on Principles of Distributed Computing*, July 2004.

[3] R. Baldoni, J.-M. Hélary, M. Raynal, and L. Tangui. Consensus in byzantine asynchronous systems. *J. of Discrete Algorithms*, 1(2):185–210, 2003.

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

[5] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, Oct. 1985.

[6] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

[7] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *EDCC-3: Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, pages 71–87, London, UK, 1999. Springer-Verlag.

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

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

[10] R. Friedman, A. Mostéfaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, 2005.

[11] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornelpublisl University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.

[12] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.