Byzantine Consensus with Few Synchronous Links

Moumen HAMOUMA[1], Achour MOSTEFAOUI[2], and Gilles TRÉDAN[2]

[1] Département d'informatique, Université A. Mira, Béjaia 06000, Algeria
moumen.hamouma@univ-bejaia.dz
[2] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France
{achour|gtredan}@irisa.fr

Abstract. This paper tackles the consensus problem in asynchronous systems prone to byzantine failures. One way to circumvent the FLP impossibility result consists in adding synchrony assumptions (deterministic solution). In the context of crash failures (at most t processes may crash), the weakest partially synchronous system model assumes at least one correct process with outgoing links that eventually permit a bounded transmission delay with at least t neighbors (the set of neighbors may change over time).
Aguilera et al. provided the main result for systems where at most t processes may exhibit a byzantine behavior. They assume a correct process with all its outgoing and incoming links eventually timely. This paper considers a system model with at least one correct process connected with x privileged neighbors with eventually timely outgoing and incoming links. In this system model, a byzantine consensus protocol is proposed. It uses authentication and assumes $x \geq 2t$.

Keywords: Asynchronous distributed system, Byzantine process, Consensus, Distributed algorithm, Eventually timely link, Fault tolerance, Resilience.

1 Introduction

Context and motivation In a distributed system a process is correct if it meets its specification during the whole execution. A process can, however, experience failures for different reasons (hardware, software, intrusion, etc.). The failure could be a simple crash. In this case, it simply stops its execution (fail-stop process). Otherwise a faulty process can exhibit an arbitrary behavior. Such a process is called *Byzantine*. This bad behavior can be intentional (malicious behavior due to intrusion) or simply the result of a transient fault that altered the local state of the process, thereby modifying its behavior in an unpredictable way. We are interested here in solving agreement problems (more precisely, the *Consensus* problem) in asynchronous distributed systems prone to Byzantine process failures whatever their origin.

In the Consensus problem, each process proposes a value, and the non-faulty processes have to eventually decide (termination property) on the same output value (agreement property) that should be a proposed value (validity property). This problem, whose statement is particularly simple, is fundamental in fault-tolerant distributed computing as it abstracts several basic agreement problems. Unfortunately, the Consensus problem has no deterministic solution in asynchronous distributed systems where even a single process can crash [13] (this is known as the FLP impossibility result). So, to

solve Consensus, asynchronous distributed systems have to be enriched with additional power. Synchrony assumptions [12], Common coins [22], randomization [5], and unreliable failure detectors [8] are examples of such additions that make it possible to solve Consensus despite asynchrony and failures. When considering Byzantine processes, the Consensus validity property is stated as: if all correct processes propose the same value v then only v can be decided. Indeed, a Byzantine process may propose a wrong value.

Related work To allow deterministic solutions to the Consensus problem [12], asynchronous systems need to be enriched with additional synchrony assumptions. In the context of crash failures, this approach has been abstracted in the notion of unreliable failure detectors [8]. A failure detector can be seen as a distributed oracle that gives (possibly incorrect) hints about which processes have crashed so far. Nearly all implementations of failure detectors consider that, eventually, the underlying system behaves in a synchronous way. More precisely, they consider the *partially synchronous system* model [8] which is a generalization of the models proposed in [12]. A partially synchronous system assumes there are bounds on process speeds and message transfer delays, but these bounds are not known and hold only after some finite but unknown time (called *Global Stabilization Time*).

The partially synchronous system model is considered by most of the works on Byzantine Consensus [3, 16, 7, 10, 11, 17, 18, 20]. [16] and [10] build a muteness failure detector[3] that is then used to solve the Consensus problem. The Byzantine consensus algorithm proposed in [14] uses directly an eventually perfect muteness failure detector. Paxos-like protocols [6, 20] first look for a stable leader before solving consensus or implementing state machine replication. Finally, [11, 18] establish lower bounds relating resiliency and (very) fast decision. [11] gives a generic algorithm that can be parametrized (w/wo authentication, fast/very fast decision) by taking into account the maximum number of processes that may crash or have malicious behavior. These two papers divide processes into three categories proposers, acceptors and learners (each process can play different roles).

Other system models have been considered like the Trusted Timely Computing Base TTCB [9]. A TTCB is a special communication channel, also nicknamed *wormhole*, that guarantees timely behavior in an otherwise asynchronous byzantine environment. The idea is that this channel is used only by critical aspects of the application (e.g., a consensus protocol), where most of the system uses a standard asynchronous medium. Similarly to the works presented above, it is assumed that the wormhole allows timely communications between any pair of correct processes.

For a system composed of n partially synchronous processes [12] among which at most t may crash, many models [2, 15, 19] try to restrict the eventually synchronous property of communication to only a subset of links in contrast to the related works cited above which assume that the whole system is eventually synchronous. In this setting, a link between two processes is said to be timely at time τ if a message sent at time τ is received not later than $\tau + \delta$. The bound δ in not known and holds only after some finite but unknown time τ_{GST} (called *Global Stabilization Time*). A link is called

[3] A muteness failure detector is an oracle that allows to distinguish between a silent Byzantine process and a correct process that is slow or with which communication is slow

eventually timely if it is timely at all times $\tau \geq \tau_{GST}$. The system model considered in [2] assumes at least one correct process with t outgoing eventually timely links (processes communicate using point-to-point communication primitives). Such a process is called an $\diamond t$-source (eventual t-source). On the other hand, the system model considered in [19] assumes a broadcast communication primitive and at least one correct process with t bidirectional but moving eventually timely links. These two models are not comparable [15]. In such a context, [2] proved that an $\diamond t$-source is necessary (and sufficient) to solve Consensus which means that it is not possible to solve Consensus if the number of eventually timely links is smaller than t or if they are not outgoing links of a same correct process.

In the context where the t faulty processes can exhibit a Byzantine behavior, Aguilera et al. [3] propose a system model with weak synchrony properties that allows to solve the consensus problem. Namely, the model assumes at least one correct process with all its outgoing and incoming links eventually timely (the other links of the system are asynchronous). Such a process is called an eventual bisource ($\diamond$ bisource). This means that the number of eventually timely links could be as low as $2(n-1)$ links. Their protocol does not need authentication but they first build very costly communication procedures on top of point-to-point communication[4]. Their consensus protocol consists of a series of rounds each made up of 10 communication steps and $\Omega(n^3)$ messages.

Contribution This paper first proposes a system model where processes are eventually synchronous and the communication model lies between the asynchronous model and the partially synchronous model. The assumed model considers that only few links are eventually synchronous. If all links are asynchronous the communication model is asynchronous. On the other hand, if all links are eventually synchronous, the system meets the partially synchronous model of [12]. It is thus stronger than the asynchronous model where the Consensus problem cannot be solved and is weaker than the partially synchronous model [12] where Byzantine Consensus can be solved if $t < n/3$ (with or without authentication[5]). The eventually synchronous links have to respect some pattern in order to be able to solve the Byzantine consensus. This pattern is captured by the notion of eventual bisource with a scope x. The eventual bisource assumed by [3] has a maximal scope ($x = n-1$). Informally, an eventual x-bisource is a correct process where the number of privileged neighbors is x instead of $n-1$. In this system model, a byzantine consensus protocol is proposed. It uses authentication and assumes an $\diamond 2t$-bisource. We assume $t < n/3$ meeting the resiliency lower bound byzantine consensus [12]. The proposed protocol enjoys the nice property of being very simple compared to other paxos like algorithms and elegant in its design. Moreover, in good settings, the decision is reached within 5 communication steps whatever is the behavior of byzantine processes. Good settings occur when the first coordinator is a $2t$-bisource.

[4] These communication procedures are similar to the consistent broadcast and the authenticated broadcast procedures [24].

[5] Byzantine Consensus can be solved with $t < n/2$ only if the processes are partially synchronous and communication are synchronous.

This is a very interesting property as under a normal setting, the communication system is mainly synchronous (having an $\diamond 2t$-bisource is very likely to happen) and failures seldom occur. A Consensus algorithm designed for the proposed model terminates as soon as some (even unknown) part of the system enjoys the $2t$-bisource property. Of course, if the system is completely synchronous, the decision can be reached faster. Contrarily, one asynchronous link over the $(n-1)^2$ links of the system can prevent an algorithm designed for a partially synchronous model from terminating.

Paper structure This paper is made up of five parts. Section 2 defines the computation and failure model and the byzantine consensus problem. Section 3 presents the consensus protocol we propose and Section 4 proves its correctness. Section 5 discusses the cost of the protocol and makes a conjecture with the intuition that sustains it. Finally, Section 6 concludes the paper.

2 Computation Model and the Consensus Problem

2.1 Computation Model

The system model is patterned after the partially synchronous system described in [12]. The system is made up of a finite set Π of n $(n > 1)$ fully-connected processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. Moreover, up to t processes can exhibit a *Byzantine* behavior, which means that such a process can behave in an arbitrary manner. This is the most severe process failure model: a Byzantine process can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, send different values to different processes, perform arbitrary state transitions, etc. A process that exhibits a Byzantine behavior is called *faulty*. Otherwise, it is *correct*.

Communication network The communication network is reliable in the sense that a message sent by a correct process to another correct process will be received exactly once within a finite time. Messages are not altered by the link and the receiver knows who the sender is. In other words, we are using authenticated asynchronous links. Such a communication network can be built atop of fair lossy links (in fair lossy links, a message can be lost a finite number of times). As advocated in [15], advanced techniques like [1] or [4] could be adopted here. They prove that even a simple retransmission until acknowledgment protocol suffices to implement a reliable link between correct processes. Using these techniques, a message that was initially lossy will eventually be received by its destinator. Note that the simulation preserves the timeliness of the messages sent on timely fair-lossy links.

Synchrony properties and bisource Every process executes an algorithm consisting of atomic computing steps (send a message, receive a message or execute local computation). We assume that processes are partially synchronous, in the sense that every correct process takes at least one step every θ steps of the fastest correct process (θ is unknown). Instead of real-time clocks, time is measured in multiples of the steps of the fastest process like in [12]. In particular, the (unknown) transfer delay bound δ is such that any process can take at most δ steps while a timely message is in transit. Hence,

we can use simple step-counting for timing out messages. Hereafter, we rephrase the definition of [15] to define more formally a timely link and a bisource.

Definition 1. *A link from a process p to any process q is timely at time τ if (1) no message sent by p at time τ is received at q after time $(\tau + \delta)$ or (2) process q is not correct.*

Definition 2. *A process p is an x-bisource at time τ if:*
- (1) p is correct
- (2) There exists a set X of processes of size x, such that: for any process q in X, both links from p to q and from q to p are timely at time τ. The processes of X are said to be privileged neighbors of p.

Definition 3. *A process p is an $\diamond x$-bisource if there is a time τ such that, for all $\tau' \geq \tau$, p is an x-bisource at τ'.*

For the rest of the paper, we consider a partially synchronous system where the only assumed synchrony properties are those needed by the $\diamond x$-bisource. This means that all the links that do not participate in the $\diamond x$-bisource could be asynchronous.

Authentication A process may be Byzantine and disseminate a wrong value (different from the value it would have obtained if it behaved correctly). To prevent such a dissemination, the protocol uses certificates. This implies the use of application level signatures (public key cryptography such as RSA signatures). A straightforward implementation of certificates would consist of including a set of signed messages as a certificate. For example, process p has to relay a value (say v) it has received from process q. Process q signs its message and sends it to p. Process p cannot relay v' if it cannot forge q's signature. Of course p can say that it received no value from q (no one can check whether this is true or not) but if it relays a value from q, it is necessarily the value it actually received from q. This means that in our model we assume that Byzantine processes are not able to subvert the cryptographic primitives. Now, suppose that p has to send to all processes the majority value among all the values it has received. The certificate, will consist of the set of received signed messages (any process can check that the value p has sent is really the majority value).

A certificate for a message m sent by p contains at least $(n - t)$ messages p has received, such that these messages led p to send m according to the protocol. Certificates do not prevent all the bad behaviors of Byzantine processes. As in many asynchronous protocols, during an all-to-all exchange, a process waits for at most $n - t$ messages otherwise it may block forever (of course a process can receive more than $n - t$ messages). In a general case, two different sets of $n - t$ messages can have a different majority value (each of the them can be certified). A Byzantine process that receives more than $(n - t)$ messages can send different certified majority values to different processes (in this case the certificate only means that the sent value is a possible value).

2.2 The Consensus Problem

The Consensus problem has been informally stated in the introduction. This paper considers *multivalued* Consensus (no bound on the cardinality of the set of proposable

values): every process p_i *proposes* a value v_i and all correct processes have to eventually *decide* on some value v in relation with the set of proposed values. Let us observe that, in a byzantine failure context, one must not choose a consensus definition that is too strong. For example, it is not possible to force a faulty process to decide as a correct process, since a byzantine process can decide whatever it wants. Similarly, it is not always possible to decide a proposed value since a faulty process can initially propose different values to distinct processes and consequently the notion of "proposed value" is not defined for byzantine processes. Thus, in such a context, the consensus problem is defined by the following three properties:

- Termination: Every correct process eventually decides.
- Agreement: No two correct processes decide different values.
- Validity: If all the correct processes propose the same value v, then only the value v can be decided.

3 The Byzantine Protocol

The proposed protocol (Figure 1) uses authentication and assumes an $\diamond 2t$-bisource. Each process p_i manages a local variable est_i which contains its current estimate of the decision value. The init phase (lines 1-3) consists of an all-to-all message exchange that allows to initializes the variable est_i to a value it has received at least $(n - 2t)$ times if any[6]. Otherwise, est_i is set v_i the value proposed by p_i. This phase establishes the validity property as if all correct processes propose the same value v, all processes will receive v at least $(n - 2t)$ times and the only value that can be received at least $(n - 2t)$ times is v (in this case, v is the only certified value). From line 5, all messages exchanged during each phase are signed, and include as certificate $(n - t)$ messages the emitting process has received during the previous exchange phase.

> **Message validity** Each process has an underlying daemon that filters the messages it receives. For example, the daemon will discard all duplicate messsages (necessarily sent by byzantine processes as we assume reliable send and receive operations between correct processes). The daemon, will also discard all messages that are not syntactically correct, or that do not comply with the text of the protocol (e.g. a process that sends two different messages with the same type within the same round, a process that sends a QUERY(r,*) message to a process that is not the coordinator of round r, etc.). Of course a message that do not comply with the associated certified is also discarded.

After the init phase, the protocol procedes in consecutive asynchronous rounds. Each process manages a variable r_i (initially set to 0). Each round r is coordinated by a predetermined process p_c (*e.g.*, c can be defined according to the round robin order). So, the protocol uses the well-known *rotating coordinator* paradigm. Each round is composed of four communication phases.

[6] This phase does not use certificates as there is no prior communication.

First phase of a round r (lines 5-7). Each process that starts a round (including its coordinator) first sends its own estimate (with the associated certificate) to the coordinator (p_c) of the current round and sets a timer to ($\Delta_i[c]$). Δ_i is an array of time-outs (one per process) managed by p_i. When the timer times out while waiting the response from a process p_j, $\Delta_i[j]$ is incremented. This allows to eventually reach the bound on the round trip between p_i and p_j if p_i and p_j are privileged neighbors. Moreover, this prevents p_i from blocking while waiting (line 6) for the response of a faulty coordinator. When the coordinator of round r receives a valid QUERY message (perhaps from itself) containing an estimate est for the first time at line 19[7], it sends sends a COORD(r, est) messages to all processes.

The COORD message is sent from another parallel task because the coordinator of round r could be stuck in previous rounds and if it does not respond quickly, the sender on the QUERY message may time out. This is why, whatever is the coordinator doing, as soon as it receives a valid QUERY message for a round it coordinates, it sends the included estimate to all processes (this allows a coordinator to coordinate a round with a certified value it has received even if it is itself lying far behind).

If the current coordinator is a $2t$-bisource it has at least $2t$ privileged neighbors among which at least t are correct process. Consequently, at least $(t + 1)$ correct processes (the t correct neighbors and the coordinator itself) got the value v of the coordinator and thus set their variable aux to v ($\neq \perp$). If the current coordinator is byzantine, it can send nothing to some processes and perhaps send different certified values to different processes (in such a case, necessarily none of these values has been decided in a previous round as we will see later). If the current coordinator is not a $2t$-bisource or if Byzantine, the three next phases allow correct processes to behave in a consistent way. Either none of them decides or if some of them decide a value v, then the only certified value for the next round will be v and thus preventing Byzantine processes from introducing other values.

Second phase of a round r (lines 8-10). This phase aims to extend the scope of the $2t$-bisource. Indeed, if the current coordinator is a $2t$-bisource then at least $(t + 1)$ correct processes set their variable aux_i to the same non-$\perp$ value (say v). During the second phase, all processes relay the value they got from the coordinator (with its certificate) or $\perp$ if they timed out (all-to-all message exchange). Each process collects $(n-t)$ valid messages (the values carried by these messages are stored into a set V_i - of course each value appears at most once in V_i as V_i is a set). If the coordinator is a $2t$-bisource then any correct process will get at least one message from the set of $(t+1)$ correct processes that got the value of the coordinator because $(n - t) + (t + 1) > n$. Otherwise, this phase has no particular effect. The condition $(V_i - \{\perp\} = \{v\})$ of line 10 means that if there is only one non-$\perp$ value v in V_i then this value is kept in aux_i (otherwise, aux_i is set to $\perp$).

[7] For any round, the coordinator will receive at least $(n - t)$ QUERY messages but it will send COORD messages only once and will ignore subsequent QUERY messages related to the same round.

Third phase of a round r (lines 11-13). This phase has no particular effect if the coordinator is correct. Its aims is to avoid the situations where the coordinator is Byzantine. Indeed, in such a case two different correct processes may have set their aux_i variables to different values. Phase three is a filter, it ensures that at the end of this phase, at most one non_$\perp$ value can be kept in the aux variables. In other words, if p_i and p_j are correct processes and if $aux_i \neq \perp$ and $aux_j \neq \perp$ then necessarily, $aux_i = aux_j$ whatever is the behavior of the byzantine processes. This phase consists of an all-to-all message exchange. Each process collects $(n - t)$ valid messages the values of which are stored in a set V_i. If all received messages contains the same value v ($V_i = \{v\}$) then v is kept in aux_i otherwise aux_i is set to the default value $\perp$. At the end of this phase, there is at most one (or none) certified value v ($\neq \perp$).

Fourth phase of a round r (lines 14-17). This phase is the decision phase. Its aim is to ensure that the Agreement property will never be violated. This prevention is done in the following way: if a correct process p_i decides v during this round then if some processes progress to the next round, then v is the only certified value. After an all-to-all message exchange, processes collect $(n-t)$ valid messages and stores the values in V_i. If the set V_i of p_i contains a unique non_$\perp$ value v, p_i decides v. Indeed among the $(n-t)$ values v received by p_i, at least $t+1$ have been sent by correct processes. Recall that after phase three, there is at most one certified values. This means that all processes receive at least one value equal to v (the other values could be v or $\perp$). Consequently any set of $(n - t)$ valid signed messages of this phase, will certify a unique value v. If a process p_j has received only $\perp$ values, it is sure that no process decides during this phase and thus it can keep the value it already has stored in est_j (the certificate composed of the $(n - t)$ valid signed messages containing $\perp$ values, allow p_j to keep its previous values).

Before deciding (line 16), a process first sends to all other processes a signed message DEC that contains the decision value (and the associated certificate). This will prevent the processes that progress to the next round from blocking because some correct processes have already decided. When a process p_i receives a valid DEC message at line 20, it first relays is to all other processes and then decides. Indeed, task T_3 is used to implement a reliable broadcast to disseminate the eventual decision value preventing some correct processes from blocking while others decide (not all processes decide necessarily during the same round).

4 Correctness of the protocol

Remark: A message exchange is the combination of a send to all operation and a message collect operation issued by every process. Let us note that, as there are at most t byzantine processes. Each correct process collects $(n - t)$ messages since only the byzantine processes could be silent (only message delivered by the communication daemon described in the previous section are considered).
There are four such exchanges: lines 1-2, lines 8-9, lines 11-12 and lines 14-15.

Function Consensus(v_i)

Init: $r_i \leftarrow 0$; $\Delta_i[1..n] \leftarrow 1$;

Task $T1$: % basic task %

——————————————————— init phase ———————————————————

(1) *send* INIT(r_i, v_i) to all;
(2) **wait until** (INIT$(r_i, *)$ received from at least $(n-t)$ distinct processes);
(3) **if** ($\exists v$: received at least $(n-2t)$ times) **then** $est_i \leftarrow v$ **else** $est_i \leftarrow v_i$ **endif**;

repeat forever
(4) $c \leftarrow (r_i \mod n) + 1$; $r_i \leftarrow r_i + 1$;

——————————————————— round r_i ———————————————————

(5) *send* QUERY(r_i, est_i) to p_c; *set_timer*$(\Delta_i[c])$;
(6) **wait until** (COORD(r_i, est) received from p_c or *time-out*) **store value** in aux_i; % *else* $\perp$ %
(7) **if** (*timer times out*)) **then** $\Delta_i[c] \leftarrow \Delta_i[c] + 1$ **else** *disable_timer* **endif**;

(8) *send* RELAY(r_i, aux_i) to all;
(9) **wait until** (RELAY$(r_i, *)$ received from at least $(n-t)$ distinct processes) **store values** in V_i;
(10) **if** $(V_i - \{\perp\} = \{v\})$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

(11) *send* FILT1(r_i, aux_i) to all;
(12) **wait until** (FILT1$(r_i, *)$ received from at least $(n-t)$ distinct processes) **store values** in V_i;
(13) **if** $(V_i = \{v\})$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

(14) *send* FILT2(r_i, aux_i) to all;
(15) **wait until** (FILT2$(r_i, *)$ received from at least $(n-t)$ distinct processes) **store values** in V_i;
(16) **case** $(V_i = \{v\})$ **then** *send* DEC(v) to all; **return**(v);
(17) $(V_i = \{v, \perp\})$ **then** $est_i \leftarrow v$;
(18) **endcase**;

———————————————————————————————————————

end repeat

Task $T2$: % coordination task %
(19) **upon** *receipt* of QUERY(r, est) for the first time for round r: *send* COORD(r, est) to all;

Task $T3$:
(20) **upon** *receipt* of DEC(est): *send* DEC(est) to all; **return**(est);

Fig. 1. The Byzantine Consensus Protocol (assumes a $2t$-bisource)

Lemma 1. *Let $\mathcal{V}_i$ and $\mathcal{V}_j$ be the sets of messages collected by two correct processes p_i and p_j respectively after a message exchange. We have:*

$$\mathcal{V}_i \cap \mathcal{V}_j \neq \emptyset$$

Proof. The proof is by contradiction. Suppose that $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ and let S be the set of all messages p_i and p_j can receive during the message exchange (i.e. messages sent to p_i and p_j).

We have $|S| = |\mathcal{V}_i| + |\mathcal{V}_j|$. Thus, $|S| = 2 \times (n-t)$ as each process waits for $(n-t)$ messages during the collect phase of an exchange.

Moreover, let f be the actual number of byzantine processes ($f \leq t$). Since, the $(n-f)$ correct processes send (according to the protocol) the same message to both processes and the f byzantine processes can send a different message to them, we have $|S| \leq 1 \times (n-f) + 2 \times f = (n+f)$ and hence, $|S| \leq (n+t)$ as $f \leq t$.

We have, $|S| \geq 2 \times (n-t)$ and $|S| \leq (n+t)$. This leads to $(n+t) \geq 2 \times (n-t)$ i.e. $n \leq 3t$ a contradiction as we assume $n > 3t$ □

Lemma 2. *After the message exchange lines 11-12, at most one non-$\perp$ value can be certified.*

Proof. Let us consider a run where a process p_i collects only messages carrying values v. Process p_i keeps the value v (the collected messages constitute the certificate of v). By Lemma 1, no other process p_j can exhibit a set of $(n-t)$ that all carry w values as the two sets need to intersect and hence, no certificate can be exhibited for another value. □

Corollary 1. *If a process decides a certified value v during a round, then only v can be decided in the same or in the next rounds (no other value than v can no more be certified).*

Proof. Let us consider the first message exchange that led a process p_i to decide a certified value v (p_i received only v values during the exchange). As p_i received a certified value v then, by Lemma 2 v is the only certified value. Thus all valid messages either carry v or $\perp$. By Lemma 1, we have: $\forall j, \mathcal{V}_i \cap \mathcal{V}_j \neq \emptyset$. As p_i received only v values, all possible sets of messages of size $(n-t)$ (i.e. certificates for the next round) include at least one one message carrying the value v. If a process decides, it decides v. If it does not decide, it has to set its local variable est_j to v for the next round (v will be the only certified value as even $\perp$ is not certified). □

Theorem 1 (agreement). *No two correct processes decide differently.*

Proof. If a correct process decides at line 20, it decides a certified value decided by another process. Let us consider the first round where a process decides at line 16. By Corollary 1, if a process decides a certified value during the same round, it decides the same value. If a process decides after receiving a DEC message at line 20 it decides the same value. Any process that starts the next round with its local variable $est_i \neq v$ will see its messages rejected (no value different from v could be certified). □

Lemma 3. *If no process decides a certified value during $r' \leq r$, then all correct processes start $r + 1$.*

Proof. Let us first note that a correct process cannot be blocked forever in the init phase. Moreover, it cannot be blocked at line 6 because of the time-out.

The proof is by contradiction. Suppose that no process has decided a certified value during a round $r' \leq r$, where r is the smallest round number in which a correct process p_i blocks forever. So, p_i is blocked at lines 9, 12 or 15.

Let us first examine the case where p_i blocks at line 9. In that case, as r is the smallest round number in which a correct process p_i blocks forever, and as line 9 is the first statement of round r where a process can block forever this means that all correct processes (they are at least $(n - t)$) eventually execute line 8. Consequently as communication is reliable between correct processes the messages sent by correct processes will eventually arrive and p_i that blocks forever at line 9. It follows that if p_i does not decide, it will proceed to the next round. A contradiction. □

Theorem 2 (termination). *If there is a $\diamond 2t$-bisource in the system, then all correct processes decide eventually.*

Proof. If a correct process decides then, due to the sending of DEC messages at line 16, any correct process will receive such a message and decide accordingly (line 20).

So, suppose that no process decides. The proof is by contradiction. By hypothesis, there is a time τ after which there is a process p_x that is a $2t$-bisource. Let p_j be a correct process and one of the $2t$ privileged neighbors of p_x. As no process decides, the time-out on the round-trip delay (from p_j to p_x plus the local computation time on p_x plus the transmission delay back to p_j) as computed by p_j will continuously increase (line 7) until it bypasses the bound imposed by the system model. Consequently, there is a time τ' after which the respective timers of all the privileged neighbors of p_x will never expire. Let r be the first round that starts after τ' and that is coordinated by p_x. As by assumption no process decides, due to Lemma 3, all the correct processes eventually start round r.

All correct processes (and possibly some byzantine processes) p_i start round r and send a QUERY message to p_x (line 5). When the coordinator p_x of round r receives the first QUERY message (line 19), it sends a COORD message to all processes. If we consider any privileged neighbor p_i of p_x the COORD message will be sent by p_x at the latest when the QUERY message from p_i is received by p_x. This means that no one of the correct privileged neighbors of p_x will time-out. They all will receive the COORD message.

In the worst case, there are t byzantine processes among the $2t+1$ privileged neighbors of p_x. A byzantine process can either relay the value of p_x or relay $\perp$ during the next phase (these are only two certified values). This allows to conclude that the value v sent by p_x is relayed (line 8) at least by the $t + 1$ correct privileged neighbors of p_x (the only other possible value is $\perp$). Since each process collects at least $(n - t)$ RELAY messages we can conclude that *all* processes will get at least one message RELAY containing the value v of p_x. It is important to notice that even byzantine processes cannot

lie about the fact they received p_x's value at line 10 as any set of $(n - t)$ messages contains at least one value v and possibly $\perp$ values.

During the third phase (lines 11-13), as the value v of p_x is the only certified value, *all* the processes that emit a certified message (byzantine processes can stay mute) emit v. This allows to conclude that all processes will have to set their aux value to v value line 13. By the same way, all processes that emit certified messages will emit v at line 14. From there we can conclude that correct processes will all decide at line 16, which proves the theorem. □

Theorem 3 (validity). *If all correct processes propose v, then only v could be decided.*

Proof. Let v the only proposed value by correct processes. Since all correct processes propose v, v is sent at least $(n - t)$ times at line 1. Since processes discard at most t messages, we can conclude that at line 3 any process will receive at least $(n - 2t)$ times the value v. Moreover, any value proposed by byzantine processes will be received at most t times. As $n > 3t$, we have $t < n - 2t$. Consequently, the only certified value is v. □

5 Discussion

5.1 On the efficiency of the protocol

The number of rounds executed by the protocol is unbounded but finite. Each round in composed four all-to-all message exchanges. Each message exchange needs $\Omega(n^2)$ messages if the links are reliable (we do not include messages sent by the byzantine processes as they can sent any number of messages).

If we consider synchronous runs (we assume accurate values for the time-outs) and no process exhibits a malicious behavior, the protocol terminates after the first round (four communication steps) and the init phase (one communication step). The protocol thus terminates in 5 communication steps.

Let us now consider synchronous links and $f \leq t$ processes exhibit malicious behavior. In the worst cat case, the first f coordinators are Byzantine. This means that the protocol will terminate at the latest after round $f + 1$ (and the init phase). The total number of communication steps is thus $(4f + 5)$. Which is the worst case.

5.2 On the minimality of the $\diamond 2t$-bisource

If we consider the partially synchronous model we defined in Section 2 (extension of [12]), we conjecture that an $\diamond 2t$-bisource in the weakest timing assumption that allows to solve the Byzantine Consensus problem if at most t processes can exhibit a Byzantine behavior.

The intuition that underlies this conjecture is the following. The $\diamond 2t$-bisource and its privileged neighbors can be seen as a cluster. Inside this cluster, communication is eventually synchronous as the bisource (1) is a correct process, (2) has timely links with all other processes of the cluster and thus can serve as a router between processes that will provide eventually timely communication between any pair of processes. We

suppose that processes are partially synchronous. Moreover, if we assume that communication is synchronous then it has been proved in [12] that the size of the cluster needs to be at least $2t + 1$ if authentication is used to be able to solve synchronous Byzantine Consensus inside the cluster. In our case communication is only partially synchronous but we still only need a size of $2t + 1$ for the cluster for the following reason.

The whole set of processes can be used for agreement preserving will trying all possible clusters. Let us imagine a protocol that executes a series of rounds each coordinated by a preselected cluster (a rotating coordination among all possible clusters of size $2t + 1$). Necessarily, it will select the good cluster infinitely often if the algorithm executes an infinite number of rounds. During the first phase of a round, the processes of the selected cluster execute a limited scope synchronous Byzantine Consensus and each of them broadcasts its decision value to the whole set of processes of the system. Indeed, if the communication between the processes of the cluster is synchronous, the correct processes that compose it will all decide the same value otherwise the processes of the cluster will terminate the synchronous Byzantine consensus with different values. In the latter case, the whole processes of the system need to execute all-to-all message exchanges to preserve the overall agreement property (at most one value). In the case where the correct process of the cluster have all decided the same value, in order to be able to extend the agreement among the processes of the "good" cluster to the whole system, it is necessary for the cluster to be enough big. This minimal size is also $2t + 1$ as this implies that there are at least $t + 1$ correct processes of the cluster that will send the decided value and hence any process from outside the cluster that collects messages from the cluster will get at least one response from a correct process.

In this sketch, we can see that $2t + 1$ is used twice. The first time to reach among the processes of the cluster (the minimal size is $2t + 1$) and the second time, the cluster needs to be large enough in order to be able to extend the decision to the whole system such that any process is sure to hear from at least one correct process of the cluster.

6 Conclusion

This paper has presented a protocol for solving Consensus in distributed systems prone to Byzantine failures. The protocol assumes a relaxed partially synchronous distributed system but where at least $4t$ communication links are eventually synchronous. These links connect the same process ($2t$ incoming links and $2t$ outgoing links). The proposed protocol has very simple design principles. In favorable setting, it can reach decision in only 5 communication steps and needs only $\Omega(n^2)$ messages in each step. Of course this protocol uses authentication.

The major contribution of this paper is to show that Byzantine Consensus is possible with very fewtimelyy links ($4t$ eventually synchronous links) versus $2n$ links for the best known protocol. Moreover, we conjecture that this is a lower bound.

Acknowledgments

The authors would like to thank Corentin Travers for the fruitful discussions on Byzantine Consensus.

References

1. Aguilera M.K., Chen W., and Toueg S., Heartbeat: a timeout-free failure detector for quiescent reliable communication. *Proc Workshop on Distributed Algorithms (WDAG'97)*, pages 126-140, 1997.
2. Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S, Communication-efficient leader election and consensus with limited link synchrony. *Proc. 23nd ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, 2004.
3. Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Consensus with byzantine failures and little system synchrony. *Proc. International Conference on Dependable Systems and Networks (DSN'06)*, Philadelphia, 2006.
4. Basu A., Charron-Bost B., and Toueg T., Crash failures vs. crash + link failures. *Proc 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, Philadelphia, Pennsylvania, 1996.
5. Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30, 1983.
6. Boichat B., Dutta P., Frölund S., and Guerraoui G., Deconstructing paxos. SIGACT News in Distributed Computing, 34(1):47-67, 2003.
7. Castro, M. and Liskov, B., Practical Byzantine fault tolerance. *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, USA, February 1999.
8. Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
9. Correia M., Neves N.F., Lung L.C. and Verissimo P., Low Complexity Byzantine-Resilient Consensus. *Distributed Computing*, Volume 17, 13 pages, 2004.
10. Doudou A., Garbinato B. and Guerraoui R., Encapsulating Failure Detection: from Crash to Byzantine Failures. *Proc. International Conference on Reliable Software Technologies*, Vienna (Austria), 2002.
11. Dutta P., Guerraoui R., and Vukolic M., Best-case complexity of asynchronous byzantine consensus. *Technical Report EPFL/IC/200499*, EPFL, Feb. 2005.
12. Dwork C., Lynch N.A. and Stockmeyer L., Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288-323, 1988.
13. Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
14. Friedman R., Mostefaoui A. and 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.
15. Hutle M., Malkhi D., Schmid U., and Zhou L., Chasing the Weakest System Model for Implementing Omega and Consensus. *Research Report 74/2005, Technische Universität Wien, Institut für Technische Informatik*, July, 2006.
16. Kihlstrom K.P., Moser L.E., and Melliar-Smith P.M., Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector, *Proc. of the Int. Conference on Principles of Distributed Systems (OPODIS)*, pp. 61-75, 1997.
17. Kursawe K., Optimistic Byzantine agreement. *Proc. of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02 Workshops)*, 2002.
18. Lamport, L., Lower bounds for asynchronous consensus. *Distributed Computing*, vol 19(2):104-125, 2006.
19. Malkhi D., Oprea F., and Zhou L., Ω meets paxos: Leader election and stability without eventual timely links. *Proc. 19th International Conference on Distributed Computing (DISC'05)*, Cracow, Poland, pp. 26-29, 2005,

20. Martin J.P., and Alvisi L., Fast Byzantine paxos. *Proc. International Conference on Dependable Systems and Networks (DSN'05)*, Yokohama, Japan, pp. 402-411, 2005.
21. Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228-234, 1980.
22. Rabin M., Randomized Byzantine Generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pp. 403-409, 1983.
23. Schneider F.B., Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299-319, 1990.
24. Srikanth T.K. and Toueg S., Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):380-394, 1987.