J Braz Comput Soc (2012) 18:29–42
DOI 10.1007/s13173-012-0058-6

LADC 2011

Timing analysis of leader-based and decentralized Byzantine consensus algorithms

Fatemeh Borran · Martin Hutle · André Schiper

Received: 4 November 2011 / Accepted: 13 January 2012 / Published online: 4 February 2012
© The Brazilian Computer Society 2012

Abstract We consider the Byzantine consensus problem in a partially synchronous system with strong validity. For this problem, two main algorithms—with different resilience—are described in the literature. These two algorithms assume a leader process. A decentralized variant (variant without leader) of these two algorithms has also been given in a previous paper. Here, we compare analytically, in a round-based model, the leader-based variant of these algorithms with the decentralized variant. We show that, in most cases, the decentralized variant of the algorithm has a better worst-case execution time. Moreover, for the practically relevant case $t \leq 2$ (where t is the maximum number of Byzantine processes), this worst-case execution time is even at least as good as the execution time of the leader-based algorithms in fault-free runs.

Keywords Distributed algorithms · Fault tolerance · Byzantine consensus · Timing analysis

The work was done while M. Hutle was at EPFL.

F. Borran (✉) · A. Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
e-mail: fatemeh.borran@epfl.ch

A. Schiper
e-mail: andre.schiper@epfl.ch

M. Hutle
Fraunhofer AISEC, Parkring 4, 85748 Garching near Munich, Germany
e-mail: martin.hutle@aisec.fraunhofer.de

1 Introduction

Consensus is a fundamental building block for fault-tolerant distributed systems. Algorithms for solving the consensus problem can be classified into two broad categories: *leader-based* algorithms that use the notion of a (changing) leader (a process with some specific role), and *decentralized* algorithms, where no such dedicated process is used. Most of the consensus algorithms proposed in early 80s, for both synchronous and asynchronous systems,[1] are decentralized (e.g., [2, 11, 14, 15]). Later, a leader (also called coordinator) was introduced, in order to reduce the message complexity and/or improve the best-case performance (e.g., [5, 7, 10]).

Obviously, there is a trade-off between the best-case performance and the worst-case performance of leader-based algorithms. For instance, a leader-based algorithm that requires α rounds in the best case (α is usually a constant), would typically require $\alpha(t+1)$ rounds in the worst case (where t is the maximum number of faulty processes). The first question we address, is whether the decentralized version of the same algorithm requires less than $\alpha(t+1)$ rounds or not? If it requires less, since the best case for a leader-based algorithm is expected to have better performance than the best case for its decentralized version, there is an interesting trade-off to analyze. The second question is to analyze the worst-case performance of the leader-based algorithm and the decentralized algorithm in terms of (i) number of rounds and (ii) in terms of execution time. The last question we address is whether the performance in terms of number of rounds allows us to predict the performance in terms of execution time.

[1] In asynchronous systems, using randomization to solve probabilistic consensus.

This work is motivated by the results of Amir et al. [1] and Clement et al. [6]. These two papers have pointed out that the leader-based PBFT Byzantine consensus algorithm [4] is vulnerable to performance degradation. According to these two papers, a malicious leader can introduce latency into the global communication path simply by delaying the message that it has to send. Moreover, a malicious leader can manipulate the protocol timeout and slow down the system throughput without being detected. This has motivated the development of decentralized Byzantine consensus algorithms [3]. The next step, addressed here, is to compare analytically the execution time of decentralized and leader-based consensus algorithms. We study the question analytically in the model considered in [4] for PBFT, namely a partially synchronous system in which the end-to-end messages transmission delay δ is unknown.

Our paper analyzes two Byzantine consensus algorithms that ensure strong validity, each one with a decentralized and a leader-based variant.[2] One of these two algorithms is inspired by *Fast Byzantine* (*FaB*) *Paxos* [12], the other by *PBFT* [4]. Our analysis shows that there is a significant trade-off between the leader-based and the decentralized variants. Mainly, it shows the superiority of the decentralized variants over the leader-based variants in different cases: First, the analysis shows that for the decentralized variants the worst-case performance and the fault-free case performance overlap, which is not the case for the leader-based variants. Second, it shows that, in most cases, the worst case of the decentralized variant of our two algorithms is better than the worst case of its leader-based variant. Third, for $t \leq 2$, it shows that the worst-case execution time of our decentralized variant is never worse than the execution time of the leader-based variant in fault-free runs.

Finally, our detailed timing analysis confirms that the number of rounds in an algorithm is not necessarily a good predictor for the performance of the algorithm.

Roadmap In the next section, we give the system model for our analysis and introduce the round model that we use for the description of our algorithms. Section 3 presents in a modular way the consensus algorithms under consideration. In Sect. 4, we give the implementation of the round model. Section 5 contains our main contribution, the analysis and comparison of the algorithms. Section 6 discusses about the hybrid variants. Finally, we conclude the paper in Sect. 7.

[2] A similar study could be done for the consensus algorithms that ensure only weak validity, such as FaB Paxos and PBFT. The results and conclusion would be similar.

2 Definitions and system model

2.1 System model

We consider a set Π of n processes, among which at most t can be faulty. A faulty process behaves arbitrarily. Nonfaulty processes are called *correct* processes, and $\mathcal{C}$ denotes the set of correct processes.

Processes communicate through message passing, and the system is partially synchronous [7]. Instead of separate bounds on the process speeds and the transmission delay, we assume that in every run there is a bound δ, unknown to processes, on the *end-to-end transmission delay* between correct processes, that is, the time between the sending of a message and the time where this message is *actually received* (this incorporates the time for the transmission of the message and of possibly several steps until the process makes a receive step that includes this message). This is the same model considered in [4] for PBFT. We do not make use of digital signatures. However, the communication channels are authenticated, i.e., the receiver of a message knows the identity of the sender. In addition, we assume that processes have access to a local nonsynchronized clock; for simplicity we assume that this clock is drift-free.

2.2 Round model

As in [7], we consider rounds on top of the system model. This improves the clarity of the algorithms, makes it simpler to change implementation options, and makes the timing analysis easier to understand. In the round model, processing is divided into rounds of message exchange.

In each round r, a process p sends a message according to a sending function S_p^r to a subset of processes and, at the end of this round, computes a new state according to a transition function T_p^r, based on the messages it received and its current state. Note that this implies that a message sent in round r can only be received in round r (rounds are *communication closed*). The message sent by a correct process p in round r is denoted by σ_p^r; messages received by process p in round r are denoted by $\boldsymbol{\mu}_p^r$ ($\boldsymbol{\mu}_p^r$ is a vector, with one entry per process; $\boldsymbol{\mu}_p^r[q] = \bot$ means that p received no message from q in round r). In all rounds, we assume the following *integrity* predicate $\mathcal{P}_{\text{int}}(r)$, which states that if a correct process p receives a non-$\bot$ message from a correct process q in round r, then this message was sent by q in round r:

$$\mathcal{P}_{\text{int}}(r) \equiv \forall p, q \in \mathcal{C} : \ \boldsymbol{\mu}_p^r[q] \in \{\bot, \sigma_q^r\}.$$

In a partially synchronous system it is possible to ensure the following property: there exists some round *GSR* (*Global Stabilization Round*) such that for all rounds $r \geq$ *GSR*, the message sent in round r by a correct process q

to a correct process p is received by p in round r. This is expressed by $\forall r \geq GSR : \mathcal{P}_{\text{sync}}(r)$, where

$$\mathcal{P}_{\text{sync}}(r) \equiv \forall p, q \in \mathcal{C} : \ \boldsymbol{\mu}_p^r[q] = \sigma_q^r.$$

We say that such a round r is *synchronous*. We further need the definition of a *consistent* round. In such a round, correct processes receive the same set of messages:

$$\mathcal{P}_{\text{cons}}(r) \equiv \forall p, q \in \mathcal{C} : \ \boldsymbol{\mu}_p^r = \boldsymbol{\mu}_q^r.$$

Consensus algorithms consist of a sequence of phases, where each phase consists of one or more rounds. For our consensus algorithms, we need eventually a phase where all rounds are synchronous, and the first round is consistent. A round in which $\mathcal{P}_{\text{cons}}$ eventually holds will be called a *WIC round* (Weak Interactive Consistency, defined in [13]).

2.3 Byzantine consensus

In the consensus problem each process has an initial value, and processes must decide on one value that satisfies the following properties:

- *Strong validity*: If all correct processes have the same initial value, this is the only possible decision value.
- *Agreement*: No two correct processes decide differently.
- *Termination*: All correct processes eventually decide.

In the paper we analyze a sequence of consensus instances.

3 Consensus algorithms

In this section, we first present two consensus algorithms, namely MA and CL, both from [3, 13], that we use for our analysis. Both algorithms require a WIC round. Then we give two implementations of WIC rounds, one leader-based (L), the other decentralized (D).

Figure 1 presents an overview of the consensus algorithms presented in the paper. The upper layer is the round-based Byzantine consensus algorithm (MA or CL) and is discussed in Sect. 3.1. Our consensus algorithms require eventually a phase where all rounds are *synchronous*, and the first round is *consistent*. Eventually synchronous rounds are provided by the implementation of the round model, which is discussed in Sect. 4 (third layer from top in Fig. 1). Ensuring eventually consistent rounds can be done in a leader-based (L) or decentralized (D) way, and is discussed in Sect. 3.2 (second layer from top in Fig. 1). By combining two consensus algorithms with two WIC round implementations we get four algorithms that will be analyzed in Sect. 5.

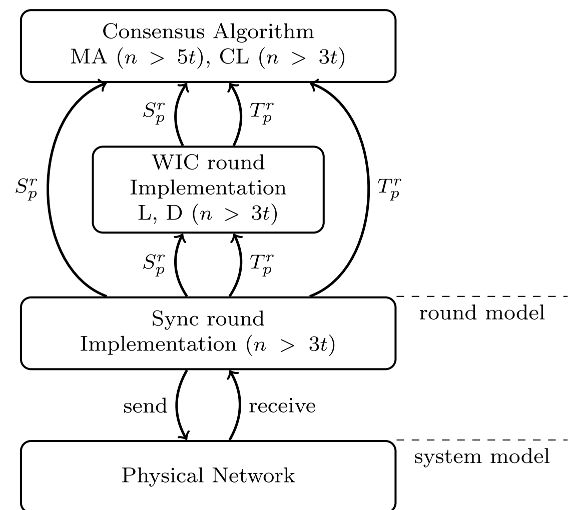

Fig. 1 Overview of the Byzantine consensus algorithms (*the arrows* represent function calls; S_p^r and T_p^r are the sending and transition functions introduced in Sect. 2.2)

3.1 Consensus algorithms with WIC rounds

3.1.1 The MA algorithm

The MA algorithm [3, 13] (Algorithm 1) is inspired by the FaB Paxos algorithm proposed by Martin and Alvisi [12].[3] A phase ϕ of Algorithm 1 consists of two rounds: $2\phi - 1$ and 2ϕ. Each process p has a *state* consisting of its current estimate x_p, initially its initial value, and its decision $decision_p$, initially $\perp$. In round $2\phi - 1$, processes first exchange their estimate (line 6) and if they receive at least $n - t$ messages (line 8), then they adopt the smallest most often value received (line 9). In round 2ϕ, processes exchange their new estimate (line 12) and can decide if at least $n - t$ messages are the same (lines 14–15). The algorithm is safe with $t < n/5$. For termination, there must be a phase, where both rounds are eventually synchronous and the first round is a WIC round.

Agreement follows from the fact that once a process decided, at least $n - 2t$ correct processes have the same estimate x, and thus no other value will ever be adopted in line 9. A similar argument is used for validity. Termination follows from the fact that in a round $2\phi - 1 \geq GSR$ with a consistent reception vector $\boldsymbol{\mu}^r$ all correct processes adopt the same value in line 9, and thus decide on this value in round 2ϕ.

[3] FaB Paxos is expressed using “proposers”, “acceptors” and “learners.” MA is expressed without these roles. Moreover, FaB Paxos solves consensus with *weak validity*, while MA solves consensus with *strong validity*. In addition, MA is expressed using rounds.

Algorithm 1 The MA algorithm with $n > 5t$ (code of process p) [3, 13]

```
1:  State:
2:    x_p ∈ V                         /* V is the set of initial values */
3:    decision_p ∈ V

4:  Round r = 2φ − 1 :                                 /* WIC round */
5:    S_p^r:
6:      send ⟨x_p⟩ to all processes
7:    T_p^r:
8:      if number of non-⊥ elements in μ_p^r ≥ n − t then
9:        x_p ← smallest most frequent non-⊥ element in μ_p^r

10: Round r = 2φ :
11:   S_p^r:
12:     send ⟨x_p⟩ to all processes
13:   T_p^r:
14:     if n − t elements in μ_p^r are equal to v ≠ ⊥ then
15:       decision_p ← v
```

3.1.2 The CL algorithm

The CL algorithm [3, 13] (Algorithm 2) is inspired by the PBFT algorithm proposed by Castro and Liskov [4], expressed using rounds, including one WIC round.[4]

Algorithm 2 consists of a sequence of phases ϕ, where each phase has three rounds: $3\phi - 2$, $3\phi - 1$, 3ϕ. Each process p has an estimate x_p, a vote value $vote_p$ (initially ?), a timestamp ts_p attached to $vote_p$ (initially 0), and a set $pre\text{-}vote_p$ of valid pairs $\langle vote, ts\rangle$ (initially $\emptyset$). Processes exchange some of their state variables in every round. The structure of the algorithm is as follows:

- If a correct process p receives the same estimate v in round $3\phi - 2$ from $n - t$ processes by line 14, then it accepts v as a valid vote and puts $\langle v, \phi\rangle$ in $pre\text{-}vote_p$ set by line 15. The prevote set is used later to detect an invalid vote (lines 28–30).
- If a correct process p receives the same pre-vote $\langle v, \phi\rangle$ in round $3\phi - 1$ from $n - t$ processes by line 20, then it votes v (i.e., $vote_p \leftarrow v$) and updates its timestamp to ϕ (i.e., $ts_p \leftarrow \phi$) by line 21.
- If a correct process p receives the same vote v with the same timestamp ϕ in round 3ϕ from $2t + 1$ processes by line 26, it decides v in line 27.

The algorithm guarantees that (i) two correct processes do not vote for different values in the same phase ϕ; and (ii) once $t + 1$ correct processes have the same vote v and the same timestamp ϕ, no other value can be voted in the following phases.

The algorithm is safe with $t < n/3$. For termination, the three rounds of a phase must eventually be synchronous and the first round must be a WIC round.[5]

[4] PBFT solves a sequence of consensus instances with *weak validity*, while CL solves consensus with *strong validity*.

[5] The proofs are given in [3].

Algorithm 2 The CL algorithm with $n > 3t$ (code of process p) [3, 13]

```
1:  State:
2:    x_p ∈ V                         /* V is the set of initial values */
3:    pre-vote_p ← ∅
4:    vote_p ∈ V ∪ {?}, initially ?
5:    ts_p ← 0
6:    decision_p ∈ V

7:  Round r = 3φ − 2 :                                 /* WIC round */
8:    S_p^r:
9:      send ⟨x_p, vote_p⟩ to all processes
10:   T_p^r:
11:     if at least n − t elements in μ_p^r are equal to ⟨−, ?⟩ then
12:       x_p ← smallest most frequent element ⟨x, −⟩ in μ_p^r
13:       pre-vote_p ← pre-vote_p ∪ {⟨x_p, φ⟩}
14:     if at least n − t elements in μ_p^r are equal to ⟨v, −⟩ then
15:       pre-vote_p ← pre-vote_p ∪ {⟨v, φ⟩}

16: Round r = 3φ − 1 :
17:   S_p^r:
18:     send ⟨v | ⟨v, φ⟩ ∈ pre-vote_p⟩ to all processes
19:   T_p^r:
20:     if at least n − t elements in μ_p^r are equal to ⟨v⟩ then
21:       vote_p ← v; ts_p ← φ; x_p ← v

22: Round r = 3φ :
23:   S_p^r:
24:     send ⟨vote_p, ts_p, pre-vote_p⟩ to all processes
25:   T_p^r:
26:     if at least 2t + 1 elements in μ_p^r are equal to ⟨v ≠ ?, φ, −⟩
        then
27:       decision_p ← v
28:     if exists ⟨v ≠ ?, ts, −⟩ in μ_p^r s.t. vote_p ≠ v and ts > ts_p then
29:       if exists t+1 elements ⟨−, −, pre-vote⟩ in μ_p^r s.t. ⟨v, ts'⟩ ∈
          pre-vote and ts' ≥ ts then
30:         vote_p ← ?; ts_p ← 0; x_p ← v
31:     if vote_p ≠ ? then x_p ← vote_p
```

3.2 Implementation of a WIC round

We consider two implementations for a WIC round: one leader-based and one decentralized. The implementations are also expressed using rounds. In order to distinguish them from the "normal" rounds, we use ρ to denote these rounds. The implementation has to be understood as follows. Let r be a WIC round, e.g., round $r = 2\phi - 1$ of Algorithm 1. The messages sent in round $r = 2\phi - 1$ are used as the input variable m_p in the WIC implementation (see Algorithm 3). The resulting vector provided by the WIC implementation, denoted by $\vec{M}_p$ (Algorithm 3) is then passed to the transition function of round r as the reception vector $\boldsymbol{\mu}_p^r$.

3.2.1 Leader-based implementation

Algorithm 3, which appears in [13], implements WIC rounds using a leader. It implements a WIC round if eventually the coordinator is correct and all three rounds are synchronous. If a correct process is the coordinator, and round

Algorithm 3 Leader-based implementation of a WIC round with $n > 3t$ (code of process p) [13]

```
1:  Initialization:
2:    ∀q ∈ Π : received_p[q] ← ⊥

3:  Round ρ = 1 :
4:    S_p^ρ:
5:      send ⟨m_p⟩ to all processes
6:    T_p^ρ:
7:      received_p ← μ_p^ρ

8:  Round ρ = 2 :
9:    S_p^ρ:
10:     send ⟨received_p⟩ to coord_p
11:   T_p^ρ:
12:     if p = coord_p then
13:       for all q ∈ Π do
14:         if |{q' ∈ Π : μ_p^ρ[q'][q] = received_p[q]}| < 2t + 1 then
15:           received_p[q] ← ⊥

16: Round ρ = 3 :
17:   S_p^ρ:
18:     send ⟨received_p⟩ to all processes
19:   T_p^ρ:
20:     for all q ∈ Π do
21:       if (μ_p^ρ[coord_p][q] ≠ ⊥) ∧ |{i ∈ Π : μ_p^ρ[i][q] =
          μ_p^ρ[coord_p][q]}| ≥ t + 1 then
22:         M⃗_p[q] ← μ_p^ρ[coord_p][q]
23:       else
24:         M⃗_p[q] ← ⊥
```

$\rho = 3$ is synchronous, all processes receive the same set of messages from this process in round $\rho = 3$.

In round $\rho = 2$, the coordinator compares the value received from some process p with the value indirectly received from other processes. If at least $2t + 1$ same values have been received, the coordinator keeps that value, otherwise it sets the value to $\perp$. This guarantees that if the coordinator keeps v, at least $t + 1$ correct processes have received v from p in round $\rho = 1$. Finally, in round $\rho = 3$ every process sends values received in round $\rho = 1$ or $\perp$ to all. Each process verifies whether at least $t + 1$ processes validate the value that it has received from the coordinator in round $\rho = 3$. Rounds $\rho = 1$ and $\rho = 3$ are used to verify that a faulty leader cannot forge the message from another process (integrity).

Since a WIC round can be ensured only with a correct coordinator, we need to ensure that the coordinator is eventually correct. In Sect. 4 we do so by using a *rotating coordinator*. A WIC round using this leader-based implementation needs three "normal" rounds.

3.2.2 Decentralized implementation

Algorithm 4 is a decentralized (without leader) implementation of a WIC round [3]. It implements a WIC round if eventually all $t + 1$ rounds are synchronous. It is based on the *Exponential Information Gathering* (EIG) algorithm for synchronous systems proposed by Pease, Shostak and Lamport [14]. Initially, process p has its initial value m_p given by the first round of the consensus algorithm (e.g., round $2\phi - 1$ in Algorithm 1). Throughout the execution, processes learn about the initial values of other processes. The information can be organized inside a tree. Each node of the tree constructed by process p has a label and a value (represented as a pair $\langle lable, value \rangle$). The root has an empty label λ and a value m_p. Process p maintains the tree using a set W_p. When p receives a message $\langle \beta, v \rangle$ from q adds $\langle \beta q, v \rangle$ to W_p, otherwise it adds $\langle \beta q, \perp \rangle$. After $t + 1$ rounds, badly-formatted messages in W_p are dropped, and all correct processes have the same values in W_p.

Algorithm 4 Decentralized implementation of a WIC round with $n > 3t$ (code of process p) [3]

```
1:  Initialization:
2:    W_p ← {⟨λ, m_p⟩}

3:  Round ρ, 1 ≤ ρ ≤ t + 1 :
4:    S_p^ρ:
5:      send {⟨α, v⟩ ∈ W_p : |α| = ρ − 1 ∧ p ∉ α ∧ v ≠ ⊥} to all
        processes
6:    T_p^ρ:
7:      for all {q | ⟨α, v⟩ ∈ W_p ∧ |α| = ρ − 1 ∧ q ∈ Π ∧ q ∉ α} do
8:        if ⟨β, v⟩ is received from process q then
9:          W_p ← W_p ∪ {⟨βq, v⟩}
10:       else
11:         W_p ← W_p ∪ {⟨βq, ⊥⟩}
12:     if ρ = t + 1 then
13:       for all ⟨α, v⟩ ∈ W_p from |α| = t to |α| = 1 do
14:         W_p ← W_p \ ⟨α, v⟩
15:         if ∃v' s.t. |⟨αq, v'⟩ ∈ W_p| ≥ n − |α| − t then
16:           W_p ← W_p ∪ ⟨α, v'⟩
17:         else
18:           W_p ← W_p ∪ ⟨α, ⊥⟩
19:       for all q ∈ Π do
20:         M_p[q] ← v s.t. ⟨q, v⟩ ∈ W_p
```

Similarly to the leader-based implementation, Algorithm 4 requires $n > 3t$. On the other hand, a WIC round using this decentralized implementation needs $t + 1$ "normal" rounds.

3.3 The four combinations

Combining the two WIC based algorithms, namely MA and CL, with the two implementations of WIC rounds, namely leader-based (L) and decentralized (D), we get four algorithms, denoted by MA-L, MA-D, CL-L and CL-D. Phases have the following lengths: four rounds for MA-L, $t + 2$ rounds for MA-D, five rounds for CL-L and $t + 3$ rounds for CL-D. Table 1 shows the number of rounds for different algorithms in the best and worst cases. However, as we will see in Sect. 5, these numbers are not good predictors of the execution time of the algorithms.

Table 1 Performance of algorithms in terms of number of rounds

	Best case	Worst case
MA-D	$t+2$	$t+2$
MA-L	4	$4(t+1)$
CL-D	$t+3$	$t+3$
CL-L	5	$5(t+1)$

4 Round implementation

As already mentioned in Sect. 2.1, we consider a partially synchronous system with an unknown bound δ on the end-to-end transmission delay between correct processes. The main technique to find the unknown δ in the literature is using an adaptive timeout, i.e., starting the first phase of an algorithm with a small timeout Γ_0 and increasing it from time to time. The timeout required for an algorithm can be calculated based on the bound δ and the number of rounds needed by one phase of the algorithm. The approach proposed in [7] is to increase the timeout linearly, while recent works, e.g., PBFT [4], increase the timeout exponentially.

The main question is when to increase the timeout. Increasing the timeout in every phase provides a simple solution, in which all processes adopt the same timeout for a given phase. However, this is not an efficient solution, since processes might increase the timeout unnecessarily. An efficient solution is increasing the timeout when a correct process requires that. This occurs typically when a correct process is unable to terminate the algorithm (i.e., decide) with the current timeout. The problem with this solution is that different processes might increase the timeout at different points in time. Therefore, an additional synchronization mechanism is needed in this case.

For leader-based algorithms, a related question is the relationship between leader change and timeout change. Most of the existing protocols apply both timeout and leader modifications at the same time [1, 4, 6, 7, 9, 12]. Our round implementation allows decoupling timeout modification and leader change. We show that such a strategy performs better than the traditional strategies in the worst case.

4.1 The algorithm

Algorithm 5 describes the round implementation. The main idea of the algorithm is to synchronize processes to the same round (round synchronization). The algorithm also requires view synchronization (eventually processes are in the same view) in addition to the round synchronization. This is because processes might increase the timeout at different rounds. The view number is thus used to synchronize the processes' timeout.

Each process p keeps a round number r_p and a view number v_p, initially equal to 1. While the round number corresponds also to the round number of the consensus algorithm, the view number increases only upon reconfiguration. Thus, the leader and the timeout are functions of the view number. The leader changes whenever the view changes,

Algorithm 5 A round implementation for Byzantine faults with $n > 3t$ (code of process p)

```
1:  r_p ← 1; next_r_p ← 1                                /* round number */
2:  Rcv_p ← ∅                                 /* set of received messages */
3:  ∀i ∈ ℕ : state_p[i] ← ⊥                         /* state of instance i */
4:  ∀i ∈ ℕ : start_p[i] ← 0            /* starting round for instance i */
5:  v_p ← 1; next_v_p ← 1                                  /* view number */

6:  while true do
7:    coord_p ← p_{(v_p−1 mod n)+1}
    -- input & send --
8:    I ← input()
9:    for all ⟨i, v⟩ ∈ I do
10:     state_p[i] ← init(v)
                        /* initialization of state with initial value v */
11:     start_p[i] ← r_p
12:   for all i : state_p[i] ≠ ⊥ do
13:     msgs[i] ← S_p^{r_p}(state_p[i], coord_p)
14:   for all q ∈ Π do
15:     M_q ← {⟨i, msgs[i][q]⟩ : state_p[i] ≠ ⊥ }
16:     send(START, M_q, v_p, r_p, p) to q
    -- receive --
17:   timeout_p ← current_time + Γ(v_p)
18:   while next_v_p = v_p and next_r_p = r_p do
19:     if current_time ≥ timeout_p then
20:       send(INIT, v_p, r_p + 1, p) to all
21:     receive(M)
22:     Rcv_p ← Rcv_p ∪ M
23:     if ∃r and t + 1 q s.t. ⟨INIT, v_p, r + 1, q⟩ ∈ Rcv_p then
24:       let r_0 be the largest such r
25:       if r_0 ≥ r_p then
26:         next_r_p ← r_0
27:         send(INIT, v_p, r_0 + 1, p) to all
28:     if ∃v and t + 1 q s.t. ⟨INIT, v + 1, −, q⟩ ∈ Rcv_p then
29:       let v_0 be the largest such v
30:       if v_0 ≥ v_p then
31:         next_v_p ← v_0
32:         send(INIT, v_0 + 1, r_p, p) to all
33:     if ∃2t + 1 q s.t. ⟨INIT, v_p, r_p + 1, q⟩ ∈ Rcv_p then
34:       next_r_p ← max{r_p + 1, next_r_p}
35:     if ∃2t + 1 q s.t. ⟨INIT, v_p + 1, −, q⟩ ∈ Rcv_p then
36:       next_v_p ← max{v_p + 1, next_v_p}
    -- comp. & output --
37:   O ← ∅
38:   for all i : state_p[i] ≠ ⊥ do
39:     for all r ∈ [r_p, next_r_p − 1] do
40:       ∀q ∈ Π : M_r[q] ← m
            if ∃M ⟨START, M, v_p, r, q⟩ ∈ Rcv_p ∧ ⟨i, m⟩ ∈ M
            else M_r[q] ← ⊥
41:       state_p[i] ← T_p^r(M_r, state_p[i], coord_p)
42:     if ∃v s.t. decision(state_p[i]) = v the first time then
43:       O ← O ∪ ⟨i, v⟩          /* v is the decision of instance i */
44:   output(O)
45:   if v_p = next_v_p ∧ next_r_p mod α = 1 then
46:     if ∃i : start_p[i] ≤ next_r_p − α ∧ decision(state_p[i]) = ⊥
        then
47:       send(INIT, v_p + 1, next_r_p, p) to all
48:   r_p ← next_r_p
49:   v_p ← next_v_p
```

based on the rotating leader paradigm (line 7). Note that the value of $coord_p$ is ignored in decentralized algorithms. The timeout does not necessarily change whenever the view changes. After line 7, a process starts the *input & send* part, in which it queries the input queue for new proposals (using a function *input*(), line 8), initializes new slots on the *state* vector for each new proposal (line 10), calls the send function of all active consensus instances (line 13), and sends the resulting messages (line 16). The process then sets a timeout for the current round using a deterministic function Γ based on its view number v_p (line 17), and starts the *receive* part, where it collects messages (line 22). Basically, this part uses an init/echo message scheme for round synchronization based on ideas that appear already in [7, 8, 16]. The receive part is described later. Next, in the *comp. & output* part, the process calls the state transition function of each active instance (line 41), and outputs any new decisions (line 44) using the function *output*(). Finally, a check is done at the end of each phase, i.e., only if $next_r_p \bmod \alpha = 1$ (line 45), where α represents the number of rounds in a phase. The check may lead to request a view change, therefore, the check is skipped if $v_p \neq next_v_p$ (the view changes anyway). The check is whether all instances, started at the beginning of the phase, have decided (lines 45–46). If not, the process concludes that the current view was not successful (either the current timeout was small or the coordinator was faulty), and it expresses its intention to start the next view by sending an INIT message for view $v_p + 1$ (line 47).

The function *init*(v) (line 10) gives the initial state for initial value v of the consensus algorithm; respectively, *decision*(*state*) (line 42) gives the decision value of the current state of the consensus algorithm, or $\perp$ if the process has not yet decided.

Receive part To prevent a Byzantine process from increasing the round number and view number unnecessarily, the algorithm uses two different type of messages, INIT messages and START messages. Process p expresses the intention to enter a new round r or new view v by sending an INIT message. For instance, when the timeout for the current round expires, the process—instead of starting immediately the next round—sends an INIT message (line 20) and waits that enough processes timeout. If process p in round r_p and view v_p receives at least $2t + 1$ INIT messages for round $r_p + 1$ (line 33), resp. view $v_p + 1$ (line 35), it advances to round $r_p + 1$, resp. to view $v_p + 1$, and sends an START message with current round and view (line 16). If the process receives $t + 1$ INIT messages for round $r + 1$ with $r \geq r_p$, it enters immediately round r (line 23), and sends an INIT message for round $r + 1$. In a similar way, if the process receives $t + 1$ INIT messages for view $v + 1$ with $v \geq v_p$, it enters immediately view v (line 28), and sends an INIT message for view $v + 1$.

Properties of Algorithm 5 The correctness proofs of Algorithm 5 are given in Sect. 4.4. Here, we give the main properties of the algorithm:

1. If one correct process starts round r (resp. view v), then there is at least one correct process that wants to start round r (resp. view v). This is because at most t processes are faulty (see Lemma 1).
2. If all correct processes want to start round $r + 1$ (resp. view $v + 1$), then all correct processes eventually start round $r + 1$ (resp. view $v + 1$). This is because $n - t \geq 2t + 1$ and lines 33–36 (see Lemma 2).
3. If one correct process starts round r (resp. view v), then all correct processes eventually start round r (resp. view v). This is because a correct process starts round r (resp. view v) if it receives $2t + 1$ INIT messages for round r (resp. view v). Any other correct process in round $r' < r$ (resp. view $v' < v$) will receive at least $t + 1$ INIT messages for round r (resp. view v). By lines 23 to 26, these correct processes will start round $r - 1$ (resp. view $v - 1$) and will send an INIT message for round r (resp. view v), see line 27. From item 2, all correct processes eventually start round r (resp. view v). The complete proof is given by Lemmas 3–5.

4.2 Timing properties of Algorithm 5

Algorithm 5 ensures the following timing properties:

1. *If process p starts round r (resp. view v) at time τ, all correct processes will start round r* (resp. view v) *by time* $\tau + 2\delta$. This is because p has received $2t + 1$ INIT messages for round r (resp. view v), at time τ. All correct processes receive at least $t + 1$ INIT messages by time $\tau + \delta$, start round $r - 1$ (resp. view $v - 1$) and send an INIT message for round r (resp. view v). This message takes at most δ time to be received by all correct processes. Therefore, all correct processes receive at least $2t + 1$ INIT messages by time $\tau + 2\delta$, and start round r (resp. view v). The complete proof is given by Lemma 5.
2. *If a correct process p starts round r (view v) at time τ, it will start round $r + 1$ the latest by time $\tau + 3\delta + \Gamma(v)$.* By item 1, all correct processes start round r, by time $\tau + 2\delta$. Then they wait for the timeout of round r, which is $\Gamma(v)$. Therefore, by time $\tau + 2\delta + \Gamma(v)$ all correct processes timeout for round r, and send an INIT message for round $r + 1$, which takes δ time to be received by all correct processes. Finally, the latest by time $\tau + 3\delta + \Gamma(v)$, process p receives $2t + 1$ INIT messages for round $r + 1$ and starts round $r + 1$. The complete proof is given by Lemma 6.

We can make the following additional observation:

(3) *A timeout $\Gamma(v) \geq 3\delta$ for round r (view v) ensures that if a correct process starts round r at time τ, it receives*

Table 2 Summary of different strategies

Strategy	A	B	C
$\Gamma(v)$	$v\Gamma_0$	$2^{v-1}\Gamma_0$	$2^{\lfloor \frac{v-1}{t+1} \rfloor}\Gamma_0$

all round r messages from all correct processes before the expiration of the timeout (*at time* $\tau + 3\delta$). By item 1, all correct processes start round r, by time $\tau + 2\delta$. The message of round r takes an additional δ time. Therefore, a timeout of at least 3δ ensures the stated property. The complete proof is given by Lemma 7.

4.3 Parameterizations of Algorithm 5

We now discuss different adaptive strategies for the timeout value $\Gamma(v_p)$. First, we consider the approach of [7]: increasing the timeout linearly (whenever the view changes). We will refer to this parameterizations by A. Then we consider the approach used by PBFT [4]: increasing the timeout exponentially (whenever the view changes). We will refer to this parameterization by B. Finally, we propose another strategy, which consists of increasing the timeout exponentially every $t + 1$ views. In the context of leader-based algorithms, this strategy ensures that, if the timeout is large enough to terminate the started consensus instances, then a Byzantine leader will not be able to force correct processes to increase the timeout. We will refer to this last parameterizations by C. These three strategies are summarized in Table 2, where v represents the view number and Γ_0 denotes the initial timeout.

4.4 Correctness proofs of Algorithm 5

In the sequel, let τ_G denote the first time that the actual end-to-end transmission delay δ is reached. All messages sent before τ_G are received the latest by time $\tau_G + \delta$. Let v_0 denote the largest view number such that no correct process has sent a START message for view v_0 by time τ_G, but some correct process has sent a START message for view $v_0 - 1$. Let r_0 denote the largest round number such that no correct process has sent a START message for round r_0 by time τ_G, but some correct process has sent a START message for round $r_0 - 1$. We prove the results related to the view number, similar results hold for round numbers:

Lemma 1 *Let p be a correct process that sends message* $\langle \text{START}, -, v, -, p\rangle$ *at some time* τ_0, *then at least one correct process q has sent message* $\langle \text{INIT}, v, -, q\rangle$ *at time* $\tau \le \tau_0$.

Proof Assume by contradiction that no correct process q has sent message $\langle \text{INIT}, v, -, q\rangle$. This means that a correct process can receive at most t messages $\langle \text{INIT}, v, -, -\rangle$ in line 28. Therefore, no correct process executes line 32, and no correct process starts view v because of line 35, which is a contradiction. □

Lemma 2 *Let all correct processes p send message* $\langle \text{INIT}, v, -, p\rangle$ *at some time* τ_0, *then all correct processes p will send message* $\langle \text{START}, -, v, -, p\rangle$ *by time* $max\{\tau_0, \tau_G\} + \delta$.

Proof If all correct processes p send message $\langle \text{INIT}, v, -, p\rangle$ at some time τ_0, then all correct processes are in view $v - 1$ at time τ_0 by lines 45–47. A correct process q in view $v - 1$, receives at least $n - t \ge 2t + 1$ messages $\langle \text{INIT}, v, -, p\rangle$ by time $\tau_0 + \delta$ if $\tau_0 \ge \tau_G$, or by time $\tau_G + \delta$ if $\tau_0 < \tau_G$. From lines 35 and 36, q starts view v by time $max\{\tau_0, \tau_G\} + \delta$. □

Lemma 3 *Every correct process p sends message* $\langle \text{START}, -, v_0 - 1, -, p\rangle$ *by time* $\tau_G + 2\delta$.

Proof We assume that there is a correct process p with $v_p = v_0 - 1$ at time τ_G. This means that p has received at least $2t+1$ messages $\langle \text{INIT}, v_0 - 1, -, -\rangle$ (line 35). Or at least $t + 1$ correct processes are in view $v_0 - 2$ and have sent a message $\langle \text{INIT}, v_0 - 1, -, -\rangle$. These messages will be received by all correct processes the latest by time $\tau_G + \delta$. Therefore, all correct processes in view $< v_0 - 1$ receive at least $t + 1$ messages $\langle \text{INIT}, v_0 - 1, -, -\rangle$ by time $\tau_G + \delta$, start view $v_0 - 2$ (line 31) and send a message $\langle \text{INIT}, v_0 - 1, -, -\rangle$ (line 32). These messages are received by all correct processes by time $\tau_G + 2\delta$. Because $n - t > 2t$, all correct processes receive at least $2t + 1$ messages $\langle \text{INIT}, v_0 - 1, -, -\rangle$ by time $\tau_G + 2\delta$ (line 35), start view $v_0 - 1$ (line 36), and send a message $\langle \text{START}, -, v_0 - 1, -, -\rangle$ (line 16). □

Lemma 4 *Let p be the first* (*not necessarily unique*) *correct process that sends message* $\langle \text{START}, -, v, r, p\rangle$ *with* $v \ge v_0$ *at some time* $\tau \ge \tau_G$. *Then no correct process sends message* $\langle \text{START}, -, v + 1, -, -\rangle$ *before time* $\tau + \Gamma(v)$. *Moreover, no correct process sends message* $\langle \text{INIT}, v + 2, -, -\rangle$ *before time* $\tau + \Gamma(v)$.

Proof For the START message, assume by contradiction that process q is the first correct process that sends message $\langle \text{START}, -, v + 1, 1, q\rangle$ before time $\tau + \Gamma(v)$. Process q can send this message only if it receives $2t + 1$ messages $\langle \text{INIT}, v + 1, -, -\rangle$ (line 35), This means that at least $t + 1$ correct processes are in view v and have sent $\langle \text{INIT}, v + 1, -, -\rangle$. In order to send $\langle \text{INIT}, v + 1, -, -\rangle$, a correct process takes at least $\Gamma(v)$ time in view v (line 19). So message $\langle \text{START}, -, v + 1, -, q\rangle$ is sent by correct process q at the earliest by time $\tau + \Gamma(v)$. A contradiction.

For the INIT message, since no correct process starts view $v + 1$ before time $\tau + \Gamma(v)$, no correct process sends message $\langle \text{INIT}, v + 2, -, q\rangle$ before time $\tau + \Gamma(v)$. □

Lemma 5 *Let p be the first (not necessarily unique) correct process that sends message $\langle \text{START}, -, v, -, p\rangle$ with $v \geq v_0$ at some time $\tau \geq \tau_G$. Then every correct process q sends message $\langle \text{START}, -, v, -, q\rangle$ by time $\tau + 2\delta$.*

Proof Note that by the assumption, all view $v \geq v_0$ messages are sent at or after τ_G, and thus they are received by all correct processes δ time later. By Lemma 4, there is no message $\langle \text{START}, -, v', -, -\rangle$ with $v' > v$ in the system before $\tau + \Gamma(v)$. Process p sends message $\langle \text{START}, -, v, -, p\rangle$ if it receives $2t + 1$ messages $\langle \text{INIT}, v, -, -\rangle$ (line 35). This means that at least $t + 1$ correct processes are in view $v - 1$ and have sent message $\langle \text{INIT}, v, -, -\rangle$, the latest by time τ. All correct processes in view $<v$ receive at least $t + 1$ messages $\langle \text{INIT}, v, -, -\rangle$ the latest by time $\tau + \delta$, start view $v - 1$ (line 31) and send $\langle \text{INIT}, v, -, -\rangle$ (line 32) which is received at most δ time later. Because $n - t > 2t$, every correct process q receives at least $2t + 1$ messages $\langle \text{INIT}, v, -, -\rangle$ by time $\tau + 2\delta$ (line 35), start view v (line 36), and send message $\langle \text{START}, -, v, -, q\rangle$ (line 16). □

Following two lemmas hold for round numbers.

Lemma 6 *If a correct process p sends message $\langle \text{START}, -, v, r, p\rangle$ at time $\tau > \tau_G$, it will send message $\langle \text{START}, -, v, r + 1, p\rangle$ the latest by time $\tau + 3\delta + \Gamma(v)$.*

Proof From Lemma 5 (similar result for round number), all correct processes q send message $\langle \text{START}, -, v, r, q\rangle$ the latest by time $\tau + 2\delta$. Then they wait for the timeout of round r which is $\Gamma(v)$ (lines 17 and 19). Therefore, by time $\tau + 2\delta + \Gamma(v)$ all correct processes timeout for round r, and send $\langle \text{INIT}, v, r + 1, q\rangle$ message to all (line 20), which takes δ time to be received by all correct processes. Finally, the latest by time $\tau + 3\delta + \Gamma(v)$, process p receives $n - t \geq 2t + 1$ messages $\langle \text{INIT}, v, r + 1, -\rangle$ and starts round $r + 1$ (line 36). □

Lemma 7 *A timeout $\Gamma(v) \geq 3\delta$ for round r ensures that if a correct process p sends message $\langle \text{START}, -, v, r, p\rangle$ to all at time $\tau \geq \tau_G$, it will receive all round messages $\langle \text{START}, -, v, r, q\rangle$ from all correct processes q, before the expiration of the timeout (at time $\tau + 3\delta$).*

Proof From Lemma 5 (similar result for round number), all correct processes q send message $\langle \text{START}, -, v, r, q\rangle$ to all the latest by time $\tau + 2\delta$. The message of round r takes an additional δ time. Therefore a timeout of at least 3δ ensures the stated property. □

Therefore, we have the following theorem.

Theorem 1 *Algorithm 5 with $n > 3t$ ensures the existence of round r_0 such that $\forall r \geq r_0 : \mathcal{P}_{\text{sync}}(r)$.*

Table 3 Parameters for algorithms MA and CL

	Fault-free case		Worst case	
	α	β	α	β
MA-D	$t+2$	0	$t+2$	0
MA-L	4	0	4	t
CL-D	$t+3$	0	$t+3$	0
CL-L	5	0	5	t

5 Timing analysis

In this section, we analyze the impact of the strategies A, B and C on our four consensus algorithms. We start with the analysis of the round implementation. Then we use these results to compute the execution time of k consecutive instances of consensus using the four algorithms MA-L, MA-D, CL-L, and CL-D.

First, for each strategy A, B, C, we compute the best case and worst-case execution time of k instances of repeated consensus, based on two parameters α and β: The parameter α is the one used in Algorithm 5. It denotes the number of rounds per phase of an algorithm, i.e., the number of rounds needed to decide in the best case. Thus, α gives also the length of a view in case a process does not decide. The parameter β denotes the number of consecutive views in which a process might not decide although the timeout is already set to the correct value. This might happen when a faulty process is the leader. Table 3 shows the values for α and β for our algorithms.

5.1 Best case analysis

In the best case we have $\Gamma_0 = \delta$ and there are no faults. Processes start a round at the same time and a round takes 2δ (δ for the timeout and δ for the INIT messages), and processes decide at the end of each phase ($= \alpha$ rounds). Therefore, the decision for k consecutive instances of consensus occurs at time $2\delta\alpha k$. Obviously, the algorithm with the smallest α (that is, the leader-based or the decentralized with $t \leq 2$) performs in this case the best.

5.2 Worst case analysis

We compute now $\tau_X(k, \alpha, \beta)$, the worst-case execution time until the kth decision when using the strategy $X \in \{A, B, C\}$. Based on item 3 in Sect. 4.2 (and Lemma 7), the first decision does not occur until the round timeout is larger or equal to 3δ. We denote below with v_0 the view that corresponds to the first decision ($k = 1$).

5.2.1 Strategy A

With strategy A, the timeout is increased in each new view by Γ_0 until $v\Gamma_0 \geq 3\delta$, i.e., until $v = \lceil 3\delta/\Gamma_0 \rceil$. Then the timeout is increased for the next β views. Therefore, we have $v_0 = \lceil 3\delta/\Gamma_0 \rceil + \beta$. To compute the time until a decision, observe that a view v lasts $\Gamma(v)$ (timeout for view v) plus the time until all INIT messages are received. It can be shown that the latter takes at most 3δ (see item 2 in Sect. 4.2). Therefore, we have for the worst case:

$$\begin{aligned}\tau_A(1,\alpha,\beta) &= \sum_{v=1}^{v_0} \alpha\big(\Gamma(v) + 3\delta\big) \\ &= \alpha \sum_{v=1}^{v_0} (v\Gamma_0 + 3\delta) = \alpha\left(\frac{v_0(v_0+1)}{2}\Gamma_0 + 3\delta v_0\right) \\ &= \alpha\left(\frac{\Gamma_0}{2}\big(\lceil 3\delta/\Gamma_0\rceil + \beta\big)\big(\lceil 3\delta/\Gamma_0\rceil + \beta + 1\big) \right. \\ &\quad \left. + 3\delta\big(\lceil 3\delta/\Gamma_0\rceil + \beta\big)\right) \qquad (1)\end{aligned}$$

and for $k > 1$,

$$\begin{aligned}\tau_A(k,\alpha,\beta) &= \tau_A(k-1,\alpha,\beta) + \alpha(v_0\Gamma_0 + 3\delta) \\ &= \tau_A(k-1,\alpha,\beta) \\ &\quad + \alpha\big(\lceil 3\delta/\Gamma_0\rceil\Gamma_0 + \beta\Gamma_0 + 3\delta\big). \qquad (2)\end{aligned}$$

5.2.2 Strategy B

With strategy B, the timeout doubles in each new view until $2^{v-1}\Gamma_0 \geq 3\delta$. In other words, the timeout doubles until reaching view $v = \lceil \log_2 \frac{6\delta}{\Gamma_0} \rceil$. Including β, we have $v_0 = \lceil \log_2 \frac{6\delta}{\Gamma_0} \rceil + \beta$, and:

$$\begin{aligned}\tau_B(1,\alpha,\beta) &= \sum_{v=1}^{v_0} \alpha\big(\Gamma(v) + 3\delta\big) \\ &= \alpha \sum_{v=1}^{v_0} \big(2^{v-1}\Gamma_0 + 3\delta\big) \\ &= \alpha\big(\big(2^{v_0} - 1\big)\Gamma_0 + 3\delta v_0\big) \\ &= \alpha\left(\big(2^{\lceil \log_2 \frac{6\delta}{\Gamma_0}\rceil + \beta} - 1\big)\Gamma_0 \right. \\ &\quad \left. + 3\delta\left(\left\lceil \log_2 \frac{6\delta}{\Gamma_0}\right\rceil + \beta\right)\right) \\ &= \alpha\left(2^{\lceil \log_2 \frac{3\delta}{\Gamma_0}\rceil} 2^{\beta+1}\Gamma_0 - \Gamma_0 \right. \\ &\quad \left. + 3\delta\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil + 3\delta + 3\delta\beta\right) \qquad (3)\end{aligned}$$

and for $k > 1$,

$$\begin{aligned}\tau_B(k,\alpha,\beta) &= \tau_B(k-1,\alpha,\beta) + \alpha\big(2^{v_0-1}\Gamma_0 + 3\delta\big) \\ &= \tau_B(k-1,\alpha,\beta) + \alpha\big(2^{\lceil \log_2 \frac{3\delta}{\Gamma_0}\rceil} 2^{\beta}\Gamma_0 + 3\delta\big). \qquad (4)\end{aligned}$$

5.2.3 Strategy C

Finally, for strategy C, the timeout doubles in each new view until $2^{\frac{v-1}{t+1}}\Gamma_0 \geq 3\delta$. In other words, the timeout doubles until reaching view $v = 1 + (t+1)\lceil \log_2 \frac{3\delta}{\Gamma_0} \rceil$; then it remains the same for the next β views. Therefore, we have and: $v_0 = (t+1)\lceil \log_2 \frac{3\delta}{\Gamma_0} \rceil + \beta + 1$, and:[6]

$$\begin{aligned}&\tau_C(1,\alpha,\beta) \\ &= \alpha\left((t+1)\sum_{l=0}^{\frac{v-1}{t+1}-1} \big(2^l\Gamma_0 + 3\delta\big) + (\beta+1)\big(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\big)\right) \\ &= \alpha(t+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 - \Gamma_0 + 3\delta\frac{v-1}{t+1}\right) \\ &\quad + \alpha(\beta+1)\big(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\big) \\ &= \alpha(t+1)\left(2^{\lceil \log_2 \frac{3\delta}{\Gamma_0}\rceil}\Gamma_0 - \Gamma_0 + 3\delta\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil\right) \\ &\quad + \alpha(\beta+1)\big(2^{\lceil \log_2 \frac{3\delta}{\Gamma_0}\rceil}\Gamma_0 + 3\delta\big) \qquad (5)\end{aligned}$$

and for $k > 1$,

$$\begin{aligned}\tau_C(k,\alpha,\beta) &= \tau_C(k-1,\alpha,\beta) + \alpha\big(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\big)(\beta+1) \\ &= \tau_C(k-1,\alpha,\beta) \\ &\quad + \alpha\big(2^{\lceil \log_2 \frac{3\delta}{\Gamma_0}\rceil}\Gamma_0 + 3\delta\big)(\beta+1). \qquad (6)\end{aligned}$$

Note that strategy C makes sense only for leader-based algorithms.

5.2.4 Comparison

Table 3 gives α and β for all algorithms we discussed. For the worst case analysis, we distinguish two cases: the worst *fault-free case*, which is the worst case in terms of the timing for a run without faulty process; and the general *worst-case* that gives the values for a run in which t processes are faulty.

We compare our results graphically in Figs. 2–4 for algorithms MA and CL. The execution time for each algorithm and strategy is a function of k, t, and the ratio δ/Γ_0. In the sequel, we fix two of these variables and vary the third.

We first focus on the first instance of consensus, that is, we fix $k = 1$ and assume $\delta = 10\Gamma_0$ which gives

[6] Note that from $v = 1 + (t+1)\lceil \log_2 \frac{3\delta}{\Gamma_0} \rceil$ it follows that $\frac{v-1}{t+1}$ is an integer.

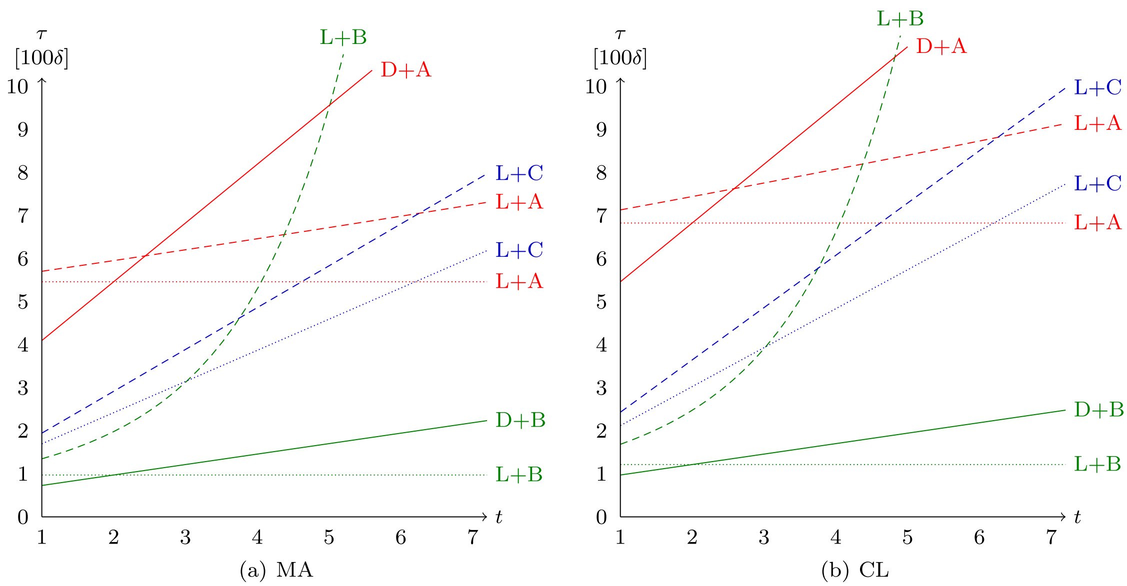

Fig. 2 Comparison for $k = 1$. *The dotted curve* represents the fault-free case and *the dashed curve* represents the worst case. *The filled curve* represents both the fault-free case and the worst case

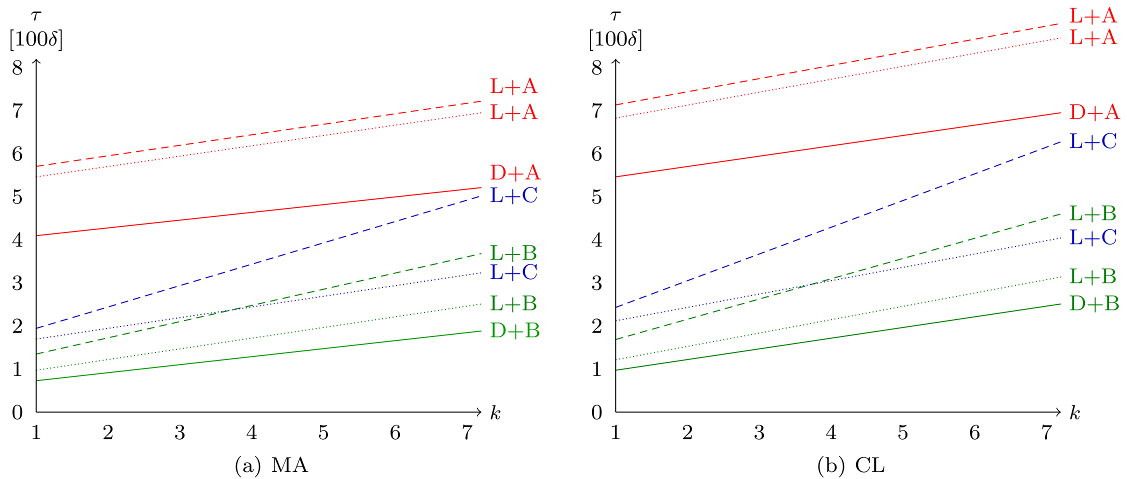

Fig. 3 Comparison for $t = 1$. *The dotted curve* represents the fault-free case and *the dashed curve* represents the worst case. *The filled curve* represents both the fault-free case and the worst case

$\lceil \log_2(3\delta/\Gamma_0)\rceil = 5$, i.e., the transmission delay is estimated correctly after five times doubling the timeout. The result is depicted in Fig. 2. We first observe, as expected, that the fault-free case and the worst-case are the same for the decentralized versions (curves $D + A$ and $D + B$). For the—in real systems relevant—cases $t < 3$, for each strategy, the decentralized algorithm decides even faster in the worst-case than the leader-based version of the same algorithm in the fault-free case. For larger t, the leader-based algorithms with strategy B, are faster in the fault-free case ($L + B$ dotted curves), but less performant in the worst-case ($L + B$ dashed curves). In the worst case, the execution time of leader-based algorithms with strategy B grows exponentially with the number of faults. This shows the interest for strategy C ($L + C$ dashed curves) in the worst case.

One can also observe that the behavior of different algorithms shown in Fig. 2 could not be derived from Table 1, although the main results match. This means that the performance of different algorithms in terms of the number of rounds does not completely predict the performance of those algorithms in terms of execution time. Even the same algorithm with different timeout strategies has different performance. This confirms the need for a detailed timing analysis.

Next, we look how the algorithms perform for multiple instances of consensus. To this end, we depict the total time until k consecutive instances decide in Fig. 3, for

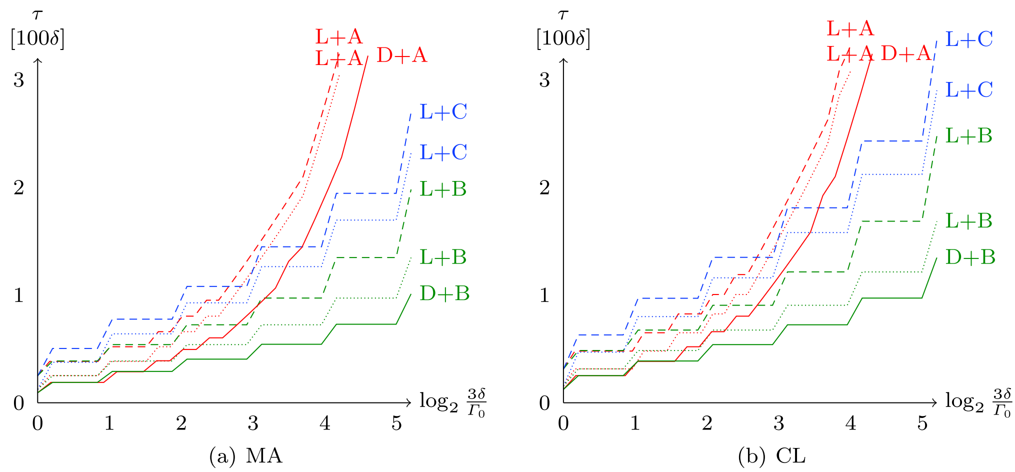

Fig. 4 Comparison of different strategies with $k = 1$ and $t = 1$. *The dotted curve* represents the fault-free case and *the dashed curve* represents the worst case. *The filled curve* represents both the fault-free case and the worst case

the most relevant case $t = 1$. Again, we assume $\delta = 10\Gamma_0$. Here, the decentralized algorithm is always superior to the leader-based variant using the same strategy, in the sense that even in the worst case it is faster than the corresponding algorithm in the fault-free case. In absolute terms, the decentralized algorithms with strategy B perform the best.

Finally, we analyze the impact of the choice of Γ_0 on the execution time (Fig. 4). This is relevant only for the first decision, i.e., $k = 1$. We look at the case $t = 1$ and vary $\log_2 \frac{3\delta}{\Gamma_0}$. Again, the decentralized version is superior for each strategy. However, it can be seen that strategy A is not a good choice, neither with a decentralized nor with a leader-based algorithm, if $\log_2 \frac{3\delta}{\Gamma_0}$ is too large. From this perspective, strategy B is the best.

In all graphs, algorithm MA performs better than algorithm CL, since it requires less number of rounds, as shown in Table 1. But both algorithms have similar behaviors.

6 Discussion

There are two important additional issues that we would like to emphasize before concluding the paper: the choice of the partial synchronous system model and the possibility to get a hybrid algorithm.

6.1 System model issue

The first issue is related to the round implementation. As we already mentioned, we consider a partially synchronous system where the end-to-end transmission delay is unknown. There are two variants in this model: (i) $GST = 0$, or (ii) $GST > 0$, where GST refers to the *Global Stabilization Time* after which the bounds on the message transmission delay and process speed hold. In the first case, there

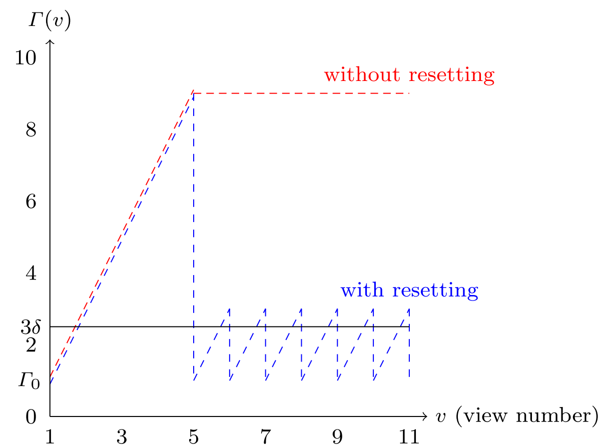

Fig. 5 Comparing different mechanisms for timeout

is no message loss, while in the second case there might be message loss before GST. Our round implementation (Algorithm 5) is correct in both system models. However, it would not be efficient in the second model, since the timeout is increased before GST, and is never decreased. For the timing analysis in Sect. 5 we have considered the first model. To obtain a more efficient round implementation in the second model, we suggest the following modifications:

1. Each correct process increases its timeout according to the timeout strategy until its first consensus decision.
2. Then the process asks to reset the timeout to Γ_0 by sending a *reset* message.
3. If a correct process receives $2t + 1$ *reset* messages, it resets the timeout to the initial timeout, i.e., Γ_0.
4. If a correct process receives $t + 1$ *reset* messages, it sends a *reset* message.

Using this protocol, the timeout is increased just enough to decide for the consensus instance. However, each consen-

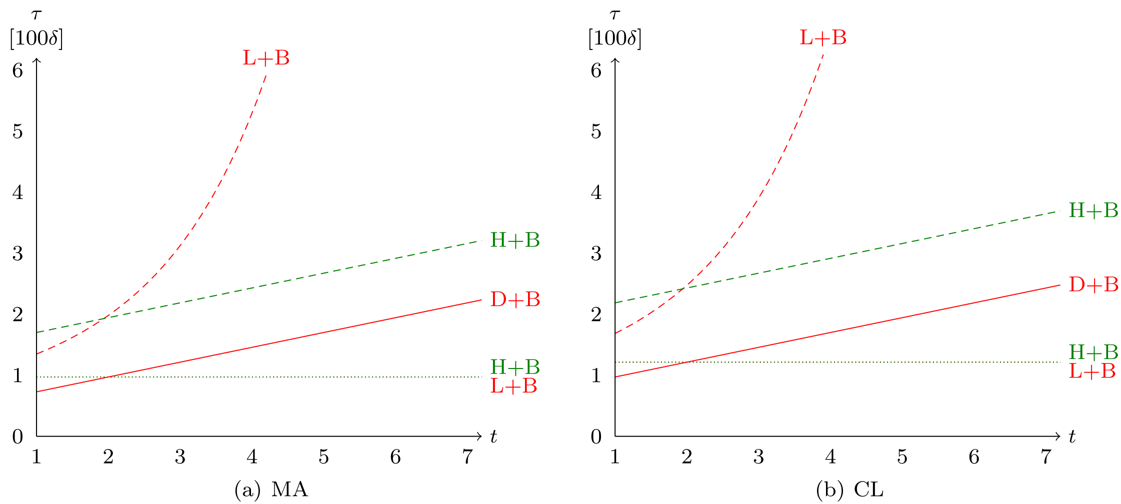

Fig. 6 Comparison of hybrid algorithm for $k = 1$ and strategy B. *The dotted curve* represents the fault-free case and *the dashed curve* represents the worst case. *The filled curve* represents both the fault-free case and the worst case

Table 4 Parameters for the hybrid algorithms

	Fault-free case		Worst case	
	α	β	α	β
MA-H	4	0	$t+6$	0
CL-H	5	0	$t+8$	0

sus instance will require the same time as the first instance. In other words, we have the following formula for the worst-case execution time until the kth instance:

$$\tau_X(k, \alpha, \beta) = k \cdot \tau_X(1, \alpha, \beta),$$

where $\tau_X(1, \alpha, \beta)$ is given by the same formulas as in Sect. 5.2.

Figure 5 compares the previous timeout mechanism (without resetting) with the mechanism presented in this section (with resetting). Assuming that *GST* holds at view number 5, the former keeps a larger timeout comparing to the latter.

6.2 Hybrid algorithm issue

The second issue is related to the leader-based versus decentralized WIC round implementation. The leader-based version has better performance in the best case, while the decentralized version performs better in the worst case. By combining two approaches, we can obtain an algorithm that performs good in both cases. The idea is the following: in the first phase (or view) run the leader-based algorithm, i.e., MA-L or CL-L. If the first view is not successful, i.e., if there is a view change, then switch to the corresponding decentralized algorithm, i.e., MA-D or CL-D.

Table 4 shows the parameters for the hybrid algorithm (H refers to the hybrid algorithms).

Figure 6 illustrates the results of the hybrid algorithms for strategy B, and compares them with the leader-based and decentralized algorithms. The hybrid algorithms are as good as the leader-based algorithms in the best case. In the worst case, the hybrid algorithms are much more efficient than the leader-based algorithm (for $t \geq 2$), but not as good as the decentralized algorithms.

7 Conclusion

We compared the leader-based and the decentralized variant of two typical Byzantine consensus algorithms with strong validity in an analytical way using the same round implementation.

Our analysis allows us to better understand the trade-off between the leader-based and the decentralized variants of an algorithm. The results show a surprisingly clear preference for the decentralized version. The decentralized variant of algorithms has a better worst-case performance for the best strategy. Moreover, for the practically relevant cases $t \leq 2$, the decentralized variant is at least as good as the fault-free case of the leader-based variant. Finally, in the best case, for $t \leq 2$, the decentralized variant is at least as good as the leader-based variant.

The results of our detailed timing analysis confirm the fact that the number of rounds is not necessarily a good estimation of the performance of a consensus algorithm.

References

1. Amir Y, Coan B, Kirsch J, Lane J (2008) Byzantine replication under attack. In: DSN'08, pp 197–206

2. Ben-Or M (1983) Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: PODC'83. ACM, New York, pp 27–30. doi:10.1145/800221.806707
3. Borran F, Schiper A (2010) A leader-free byzantine consensus algorithm. In: ICDCN. Lecture notes in computer science (LNCS). Springer, Berlin, pp 67–78
4. Castro M, Liskov B (2002) Practical Byzantine fault tolerance and proactive recovery. ACM Trans Comput Syst 20(4):398–461
5. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267
6. Clement A, Wong E, Alvisi L, Dahlin M, Marchetti M (2009) Making Byzantine fault tolerant systems tolerate Byzantine faults. In: NSDI'09. USENIX Association, Berkeley, pp 153–168
7. Dwork C, Lynch N, Stockmeyer L (1988) Consensus in the presence of partial synchrony. J ACM 35(2):288–323
8. Hutle M, Schiper A (2007) Communication predicates: a high-level abstraction for coping with transient and dynamic faults. In: Dependable systems and networks (DSN 2007). IEEE Press, New York, pp 92–100
9. Kotla R, Alvisi L, Dahlin M, Clement A, Wong E (2007) Zyzzyva: speculative byzantine fault tolerance. Oper Syst Rev 41(6):45–58. doi:10.1145/1323293.1294267
10. Lamport L (1998) The part-time parliament. ACMTCS 16(2):133–169
11. Lamport L, Shostak R, Pease M (1982) The byzantine generals problem. ACM Trans Program Lang Syst 4(3):382–401. doi:10.1145/357172.357176
12. Martin JP, Alvisi L (2006) Fast Byzantine consensus. IEEE Trans Dependable Secure Comput 3(3):202–215. doi:10.1109/TDSC.2006.35
13. Milosevic Z, Hutle M, Schiper A (2009) Unifying byzantine consensus algorithms with weak interactive consistency. In: OPODIS, pp 300–314
14. Pease M, Shostak R, Lamport L (1980) Reaching agreement in the presence of faults. J ACM 27(2):228–234. doi:10.1145/322186.322188
15. Rabin M (1983) Randomized Byzantine generals. In: Proc symposium on foundations of computer science, pp 403–409
16. Srikanth TK, Toueg S (1987) Optimal clock synchronization. J ACM 34(3):626–645. doi:10.1145/28869.28876