The Byzantine Agreement Problem: Optimal Early Stopping*

Axel W. Krings
Computer Science Dept.
University of Idaho, USA
krings@cs.uidaho.edu

Thomas Feyer
Computer Science Dept.
BTU-Cottbus, Germany
feyer@informatik.tu-cottbus.de

Abstract

This paper addresses solutions to the problem of reaching agreement in the presence of faults. Whereas the need for agreement has surfaced mainly in fault tolerant real-time applications, agreement can be a useful mechanism in network security to mask intrusions. However, due to the communication overhead involved and the fact that the system is expected to operate without problems most of the time, early stopping algorithms are of special interest. We introduce a non-authenticated early stopping algorithm that is optimal in terms of rounds and the number of processors in the system. The basic idea of the algorithm is closely related to the work in [5]. However, our algorithm is easier to implement because of its algorithmic definition. It is directly derived from the algorithm by Lamport [18], and is based on two simple functions only. We are convinced that the construction of the early stopping algorithm presented here furthermore increases understanding and clarifies underlying problems of early stopping.

1 Introduction

Byzantine Agreement was first introduced by Pease, Shostak and Lamport [20], where the problem was to reach consensus in a network of n processors among which t processors may behave faulty. To reach agreement, a previously designated processor (the initiator) broadcasts a value. Afterwards, all other processors exchange messages until all non-faulty processors agree upon the same value. It is implied that if the initiator is non-faulty, all non-faulty processors agree upon his value. This problem became known as the *Byzantine Agreement Problem.*

The work presented here is motivated by an ongoing project that intends to apply the basic ideas of Byzantine Agreement to network security. The project targets agreement at distributed environments with standard network topologies, rather than replicated broadcast channels [2]. Critical functionalities are replicated on different network nodes in order to increase survivability of the distributed systems in the presence of intrusions. The resulting environment is subjected to communication challenges similar to those addressed in recent literature dealing with reliable ordered delivery protocols and group communication in partitioned networks, e.g. Totem [19] or Transis [14]. However, the project does not address the problem associated with network partitioning, but focuses on malicious behavior as a consequence of system intrusions. Furthermore, the final goal is to apply agreement algorithms under restrictive connectivity assumptions to aid intrusion survivability rather than guarantee fault-tolerance, which is limited for typical topologies. However, rather than restricting the model with respect to connectivity, this paper refocus on traditional Byzantine Agreement at the highest layer of the project's solution space.

An algorithm stated in [18] solves the Byzantine Agreement problem using $n \geq 3t+1$ processors. Questions about bounds in redundancy as well as performance of the algorithm arose. Optimality in terms of redundancy was found in [18, 20]. Specifically, there is no solution for the Byzantine Agreement Problem if there are n processors and at least $n/3$ faulty processors. Among others, performance of Byzantine Agreement algorithms can be measured in terms of the number of *rounds* of message exchanges the processors need in order to achieve agreement. Optimality in terms of rounds was found by Fisher and Lynch [16]. They showed that, given a network with $n \geq 3t+1$ processors containing up to t faulty processors, a non-authenticated Byzantine Agreement algorithm requires at least $t+1$ rounds of message exchange in the worst case. Lamport's algorithm in [18] meets this lower bound.

*This work has been supported in part by the Idaho National Engineering and Environmental Laboratories (INEEL).

Later research focused on the problem that Byzantine Agreement algorithms adapted to systems with up to t faulty processors do not stop earlier if there are only $f \leq t$ faulty processors in the system. It could be shown that an improvement is possible in the case of *eventual* Byzantine Agreement [9, 13], where individual processors may stop in different rounds. Thus, a stopped processor does not know when the agreement among all processors is reached. Dolev and Strong [9] showed that the lower bound on rounds for reaching eventual Byzantine Agreement is $min\{f+2, t+1\}$, if there are $f \leq t$ currently faulty processors in the system. This lower bound was met by an *early stopping algorithm* stated in [5]. A processor *stopped early* when it has decided upon a value, a so-called *agree-value*, and does not send or receive further messages belonging to this particular agreement.

Approaches to solutions of the Byzantine Agreement problem have been studied in various environments, under different assumptions, and have been emphasizing different goals. To better place the research presented here into the proper context, the following brief survey addresses different issues in deterministic agreement. Most of the variants are orthogonal to each other, i.e., algorithms may combine them.

Exact vs. Approximate In an exact agreement all non-faulty processors have to agree exactly on one value, 0 or 1. This condition is relaxed in an approximate agreement. There, non-faulty processors have to decide on values which are approximately equal [1, 11, 21]. In both cases, proven bounds are $n \geq 3t+1$ processors [18] and $t+1$ rounds [16] in non-authenticated protocols. We exclusively concentrate on exact agreement.

Authenticated vs. Non-authenticated In authenticated algorithms, messages are signed. The signatures are assumed to be unforgeable. Therefore, an altered message can be detected, which strongly restricts the ability of faulty processors to lie. Non-authenticated algorithms do not sign messages, and forged messages cannot be detected in general. Contrary to non-authenticated algorithms, Byzantine Agreement can be achieved with an arbitrary number of faulty processors in the authenticated case [12].

Immediate vs. Eventual Immediate algorithms require the non-faulty processors to stop in the same round. By relaxing this assumption, non-faulty processors may stop in an arbitrary round [5, 13, 23]. The resulting algorithms are called eventual.

Fault Models Using fault classification one can improve the fault-tolerance of a system. Faults have been partitioned into benign faults, i.e., globally self-evident faults, and malicious faults, i.e., symmetric or asymmetric faults, in [22].

Network Connectivity The connectivity of the network affects the number of processors and the number of rounds needed to achieve agreement. In general, as connectivity weakens, more processors and more rounds are required. The lower bound for the connectivity is $2t+1$ in the non-authenticated case and $t+1$ in the authenticated case [7, 17].

Message Complexity Of importance can be the number of messages sent or the size of the messages, i.e., the bit complexity. Message complexity and bounds are addressed in [10, 12, 4]. Stated algorithms are mainly distinguished by the number of messages they require — either exponential or polynomial. In the first case, the algorithms are called *Exponential Information Gathering (EIG)* algorithms. As commonly recognized, there seems to be a tradeoff between the message complexity and the number of rounds.

In this paper, we derive an optimal early stopping algorithm from the algorithm by Lamport [18]. Section 2 introduces the Byzantine Agreement problem and the EIG tree representation. Next, Section 3 gives the motivation for adapting Lamport's algorithm to eventual agreement and furnishes the theoretical basis. In Section 4 the "Ensure" Algorithm is presented which uses two functions to achieve early agreement. Finally, Section 5 concludes the paper with a brief summary and future directions.

2 Background

2.1 The Byzantine Agreement Problem

In the formulation of the Byzantine Generals [18] problem we assume a fully connected reliable network with n processors, t of which may be faulty. There are no restrictions on the behavior of faulty processors, i.e., the faulty processors may send conflicting messages or omit messages altogether. The goal is to mask the behavior of faulty processors to prevent faulty processors from "confusing" the non-faulty processors: The Byzantine Agreement problem is defined as follows: (1) a designated processor, the *initiator*, broadcasts a value to each other processor, (2) by exchanging messages, the so-called *interactive consistency* conditions IC1 and IC2 have to be satisfied [18].

- **(IC1)** All non-faulty processors agree on the same value, and

- **(IC2)** If the initiator is non-faulty, then all non-faulty processors agree on its value.

In order to solve this problem, the following general assumptions (and their justification) are proposed in [18, 11]:

A1. Every message sent is delivered correctly.

A2. The receiver of a message knows who sent it.

A3. The absence of a message can be detected.

We implicitly assume that each processor knows the identity of the sender, the other system participants, and the beginning time of the agreement.

2.2 Issues of Early Stopping

Until now, we only considered algorithms that generally cope with the worst case of t faulty processors. However, the situation in reliable systems normally looks less pessimistic. In a standard scenario, initially all processors are functioning correctly. Later, some processors may fail. Thus in a common case scenario, a reliable system must only cope with $f < t$ faulty processors. As investigated in [13] one could raise the question if an algorithm can take advantage of this by stopping early, i.e., in a round smaller than $t+1$, if there are only f currently faulty processors in the system. The problem is addressed in [5, 8].

Recall that in immediate Byzantine Agreement all processors reach agreement and stop during the same round, whereas in eventual Byzantine Agreement this "synchronization" is not required. In the latter, an early stopped processor cannot determine when all other processors will stop. In the worst case, agreement will be achieved after the maximum number of rounds. Dolev [9] states that immediate Byzantine Agreement cannot be achieved for n processors with at most t faults within t or fewer rounds, provided $n > t+1$. As a result, there is no possibility of early stopping for immediate algorithms. Better results can be achieved, if we eliminate the requirement that all processors have to stop in the same round. A lower bound for eventual agreement of $min\{f+2, t+1\}$ rounds for $f \leq t$ faults was given in [9]. An exponential algorithm that meets this lower bound of $min\{f+2, t+1\}$ rounds was given in [5]. In each round, their algorithm applies a set of rules to the received messages until no more rules are applicable. In Section 4, we will introduce an algorithm that is easier to implement.

2.3 Lamport's Algorithm

To derive the essential ideas of our early stopping algorithm, we first investigate the original algorithm as stated by Lamport [18]. This immediate algorithm $OM(t)$ reaches agreement using $n \geq 3t+1$ processors in the presence of up to t faulty processors in exactly $t+1$ synchronous rounds.

In the algorithm, the function $majority : V^k \rightarrow V$ will be applied to a vector of binary values $\vec{v} = (v_1, v_2, \ldots, v_k)$, $v_i \in V$, $V = \{0,1\}$, $k \in I\!N$, and delivers the value that occurs most often in the vector, or a default value in the case of a tie. Throughout the paper, we use value 0 as a default.

Lamport's Algorithm OM(t)
OM(0)

(1) The initiator sends its value to every processor.

(2) Each processor uses the value it receives from the initiator, or uses the value 0 if it receives no value.

OM(m), m > 0

(1) The initiator sends its value to every other processor.

(2) For each processor p_i, let v_i be the value p_i receives from the initiator, or else be 0 if it receives no value. Processor p_i acts as an initiator in algorithm $OM(m-1)$ to send value v_i to each of the $(n-2)$ other processors.

(3) For each processor p_i, and each $j \neq i$, let v_j be the value processor p_i received from p_j in step (2) (using algorithm $OM(m-1)$), or else 0 if it received no such value. Processor p_i uses the value $majority((v_1, \ldots, v_{n-1}))$.

The recursively defined algorithm consists of two subsequent parts; the first part handles the communication among the processors (the *message exchange*) and the second part the computation of the agree-value (the *voting process*). Both parts consist of exactly $t+1$ rounds. Since computation time is neglectable in comparison with communication time, only the rounds of the message exchange count for performance considerations.

2.4 EIG Tree Representation

To represent the application of the algorithm, we adopt the notion of Exponential Information Gathering trees (EIG tree) [5]. By definition, a *message* $h : v$ consists of a non-empty *history list* h and a value $v \in V$. Thereby, the history list h is a list of processors.

The history list h of a message $h : v$ states which processors broadcasted value v, as well as the broadcasting sequence. For example, message $[p_1, p_7, p_3] : 1$

states that a value was sent by the initiator p_1 to processor p_7, then by p_7 to p_3, and finally by p_3 to the processor that received this message. Nothing is said about the original value. The initiator could have sent 0, and later it could have been altered to 1 by a faulty processor in the history list. Unless specified otherwise, when talking about value v, it is assumed to be in the context of message $h : v$.

A *value tree* is a tree that represents a set M of messages $h : v$. Its leaf nodes are labeled by the values v of the messages in M. Its edges are labeled by processor names, so that the path from the root node to a leaf node v represents the history list h that corresponds to value v. If two messages $h \circ h_1 : v_1,\ h \circ h_2 : v_2 \in M$ start with the same processor list h, the corresponding path h in the tree will be shared by both messages. Thereby, "$h_1 \circ h_2$" represents the concatenation of the history lists h_1 and h_2.

EIG trees are value trees that represent the set of received values. Since our goal is to derive an early stopping algorithm, we intend to observe the set of received values after each round of the message exchange. To indicate the observed round r, we speak of round-r-trees instead of general EIG trees. Thus, we define a *round-r-tree*, $r = 0, 1, \ldots, t$, as a value tree that represents the set of values a processor received until the end of round r of the message exchange.

At this point it is helpful to introduce an example of a Byzantine Agreement with $n = 7$ processors $(p_1, p_2, \ldots, p_7)$ containing $t = 2$ asymmetric faulty processors p_1 and p_7. The initiator is faulty processor p_1. We have to apply algorithm $OM(2)$:

The Message Exchange

- Round 0:
  - *Broadcast phase:* Faulty initiator p_1 sends value [] : 1 to processors $p_1, \ldots, p_5$ and value [] : 0 to processors p_6 and p_7.
  - *Receive phase:* Non-faulty processors $p_2, \ldots, p_5$ receive the round-0-tree shown in Figure 1a. Non-faulty processor p_6 receives the round-0-tree shown in Figure 1b.
- Round 1:
  - *Broadcast phase:* All non-faulty processors $p_2, \ldots, p_6$ broadcast their received value $[p_1]$: v to processors $p_2, \ldots, p_7$ correctly. The faulty processor p_7 sends value $[p_1]$: 1 to p_2, p_3, and p_4 and value $[p_1]$: 0 to p_5, p_6, and p_7.
  - *Receive phase:* Non-faulty processors p_2, p_3, and p_4 receive the round-1-tree shown in Figure 2a. Non-faulty processors p_5 and p_6 receive round-1-tree shown in Figure 2b.
- Round 2:
  - *Broadcast phase:* All non-faulty processors $p_2, \ldots, p_6$ broadcast the values they received in round 1. The faulty processor p_7 exclusively broadcasts 0 values.
  - *Receive phase:* Figures 3 and 4 show the round-2-trees received by processors p_3 and p_6 respectively. Note that some non-bottom level nodes within a round-r-tree carry values, whereas others do not, e.g., see value $[p_1, p_3]$: 1 within the round-2-tree in Figure 3). These are values that are re-broadcasted to the transmitter itself, since they will not be re-broadcasted further.

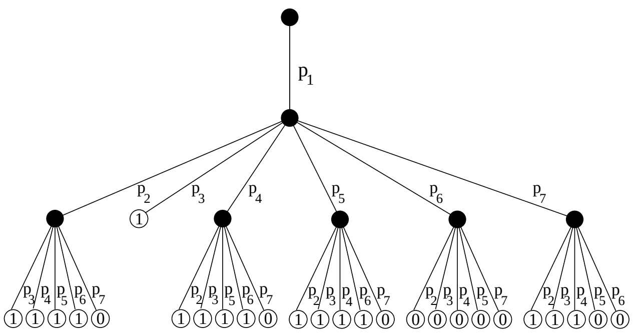


Figure 3: Round-2-tree received by p_3

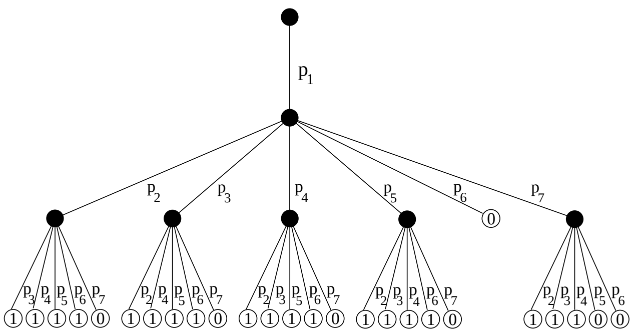


Figure 4: Round-2-tree received by p_6

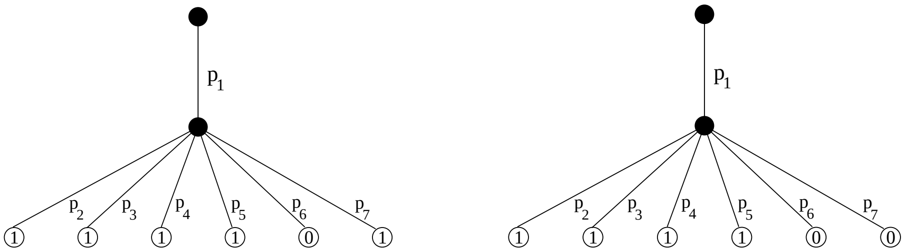
The Voting Process:

- Round 2:

  All non-faulty processors use their received round-2-tree to compute the round-1-tree shown in Figure 5a. Thereby, function *majority* is applied to

a) Received by $p_2, \ldots, p_5$ b) Received by p_6

Figure 1: Round-0-trees



a) Received by p_2, p_3, p_4 b) Received by p_5 and p_6

Figure 2: Round-1-trees

the grouped leafs in the bottom layer of the round-2-tree.

- Round 1: All non-faulty processors use their computed round-1-tree to compute the round-0-tree shown in Figure 5b. Thereby, function *majority* is applied to the remaining value vector $(1,1,1,1,0,1)$ in the bottom layer of the round-1-tree. The agree-value is 1.

Observations

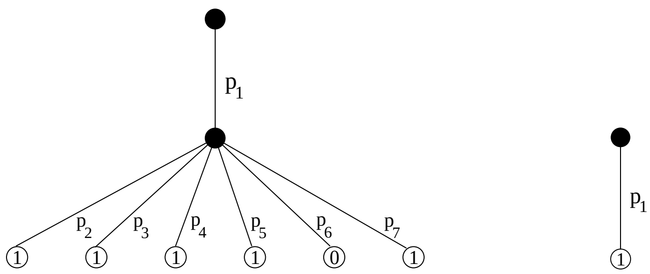
As demonstrated in the example, each processor does not only observe round-r-trees during the message exchange but also during the voting process. To distinguish between these trees, we introduce the notion of *received* and *computed* round-r-trees. If we do not want to specify whether a round-r-tree is received (during the message exchange) or computed (during the voting process), we call it *obtained* round-r-tree.

The received and the computed round-r-tree of a specific processor are equal in their structure but might differ in their values in the bottom layer. For example, processor p_5 receives the round-1-tree shown in Figure 2b and computes the round-1-tree shown in Figure 5a. Both round-1-trees differ in the single value on path $[p_1, p_7]$.

3 From Immediate to Eventual Agreement

To achieve early stopping using Lamport's algorithm, a processor has to investigate the received tree after *each* particular round of the message exchange. If it can infer the agree-value early, in a round $r < t$, it decides on this value and stops. Early stopping of a processor causes missing messages for other processors. In some scenarios the processors might decide on a different agree-value than they would using Lamport's algorithm without early stopping. To avoid this effect, we apply a straightforward replacement strategy for missing values which prevents the processors from deciding on an altered agree-value. Thus our algorithm will compute the same agree-value as Lamport's algorithm in each possible scenario, i.e., it *conforms* to Lamport's algorithm at least in the agree-value.

A promising approach for investigating whether a received round-r-tree "reveals" the agree-value early can be performed by a special voting process, as will be discussed in the next section. For readability reasons, we first introduce a slightly modified version of Lamport's algorithm. Thereby, a processor executes the whole voting process not only once after round t, but after *each* particular round r of the message exchange, i.e., the whole voting process is applied to each received round-r-tree. In every round r, $r = 0, 1, \ldots, t$, this process results in a single value (the computed root



a) Round-1-tree

b) Round-0-tree

Figure 5: Trees computed by all non-faulty processors

value), either 0 or 1. Nonetheless, these computed root values do not reveal whether the agree-value will be 0 or 1. However, comparing values as well as trees that occur in this adapted algorithm leads to our optimal early stopping algorithm.

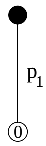


Figure 6: *Received* round-0-tree

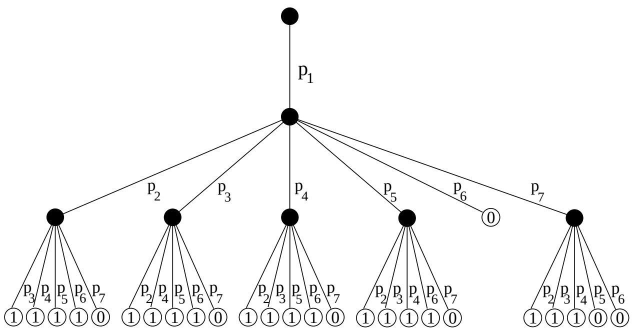


Figure 8: *Received* round-2-tree

An example of this adapted algorithm is demonstrated in Figures 6 to 9, using the example of Section 2. During the algorithm, each processor, in general, obtains one received round-r-tree and more than one computed round-r-tree. In the example, processor p_6 receives one round-0-tree (Figure 6) and computes *two* round-0-trees (Figures 7 right and 9 right). To distinguish between this sequence of trees, we attach the round number in which they are obtained. We use *round-*$r^{(r+k)}$*-tree* for the round-r-tree a processor obtains in round $(r+k)$, $(r+k) = r, r+1, \ldots, t$. The values in these trees are denoted similarly, i.e., value $h : v^{(r+k)}$ indicates the value v a processor obtains on the path h in round $(r+k)$. If we do not want to indicate the specific round $(r+k)$, $k \geq 0$, in which the round-$r^{(r+k)}$-tree was obtained, we will use *(obtained) round-*r*-tree* as a synonym for each round-r-tree.

Our intention is to compare all round-r-trees observed as a sequence. In particular, we want to compare the round-$0^{(0)}$-tree, the round-$0^{(1)}$-tree, ..., and the round-$0^{(t)}$-tree with one another. All these trees carry the root values computed in each round. By definition, the root value computed in round t, i.e., the single value in the round-$0^{(t)}$-tree, is taken as the agree-value in Lamport's algorithm. In the example, the round-$0^{(2)}$-tree in Figure 9 (right) has agree-value 1.

Again, we want to apply our adapted algorithm to the example scenario of Section 2. Processor p_6 obtains the sequence of root values $0, 1, 1$ as demonstrated in Figures 6, 7 (right) and 9 (right). In contrast, applying the adapted algorithm to the processors p_2, ..., p_5, they obtain the sequence of root values $1, 1, 1$. Using this strategy, processors $p_2, \ldots, p_5$ might possibly stop early after round 0 and processor p_6 might possibly stop early after round 1 since they correctly computed the agree-value 1. This example motivates the existence of a round after which this sequence of root values always possesses the correct agree-value. As a result of these observations, we pose two essential questions: (1) By which round will the sequence of root values always possess the correct agree-value (observed by a specific processor)? We will call it the *early stopping round* (of this processor). (2) How can a processor detect this round?

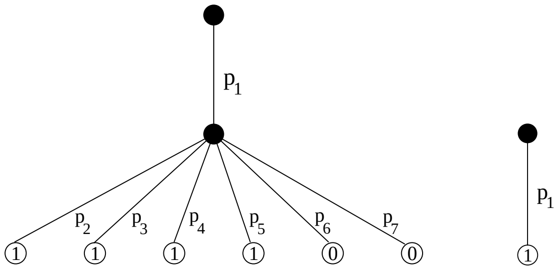


Figure 7: *Received* round-1-tree and *Computed* round-$0^{(1)}$-tree

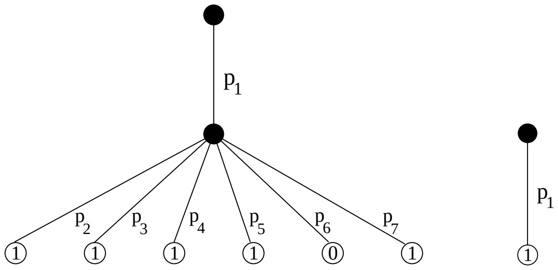


Figure 9: *Computed* round-$1^{(2)}$-tree and round-$0^{(2)}$-tree

With respect to the first question, in the worst case this round is round t, i.e., early stopping is not possible. As demonstrated in the example, the adapted version of Lamport's algorithm delivers a sequence of root values. Our idea is to superimpose the root values. Thus, if there exists an early stopping round, we can say the root value is *fixed* in that round. In the example above, the root values of processors $p_2, \ldots, p_5$ are fixed to value 1 in round 0, whereas the root value of processor p_6 is fixed to value 1 in round 1.

As indicated earlier, our adapted version of Lamport's algorithm does not only deliver a sequence of root values, but also a sequence of round-1-trees, round-2-trees, etc. Thus, each processor can also compare its obtained sequences of round-r-trees. We want to extend the notion of fixed values to arbitrary, received values. Specifically, a value $h : v$ a processor received in round r is called *fixed (in round* $(r+k)$, $(r+k) = r$, $r+1$, $\ldots$, t*)*, if for the values $h : v^{(r+k)}$, $h : v^{(r+k+1)}$, $\ldots$, $h : v^{(t)}$,

$$v^{(r+k)} = v^{(r+k+1)} = \ldots = v^{(t)}.$$

We say, value $h : v$ *does not change in later rounds* (after round $(r+k)$). If value $h : v$ is already fixed in round r in which it was received, we say, $h : v$ is *fixed immediately*.

Consider the values processor p_6 receives in round 1 (Figure 7 left). When comparing this tree with the round-1-tree (Figure 9 left) p_6 computes in round 2 (Figure 8), we recognize the following: the four 1 values received on the paths $[p_1, p_2]$, $[p_1, p_3]$, $[p_1, p_4]$, $[p_1, p_5]$ are fixed immediately. The 0 value received on the path $[p_1, p_6]$ is also fixed immediately. Finally, the value received on the path $[p_1, p_7]$ is fixed to 1 only in round 2 since it differs from its corresponding value computed in round 2. (Note that *all* values are fixed in the last round t since the sequence of obtained values ends by this round).

We now need to refine our view of a path with respect to its last processor. A path $h \circ [p_c]$ that starts with an arbitrary path h and ends with a non-faulty processor p_c, we call a *correct path*. A path $h \circ [p_f]$ that starts with an arbitrary path h and ends with a faulty processor p_f, we call *faulty path*. Values a processor obtains on a correct or a faulty path, we call *correct value* or *faulty value* respectively. We are now ready to state our first lemma.

Lemma 1 *Each value v received on a correct path, i.e. a path $h \circ [p_c]$ which ends with a non-faulty processor p_c, is fixed immediately.*

Proof: See [15, Lemma 3.2b]. $\square$

When speaking of *descendants* of a value, we indicate all values that occur on descendant paths in the next round(s). For example, the value received on the path $[p_1]$ by processor p_6 (Figure 6) possesses the descendant values $(1,1,1,1,0,0)$ received in round 1 (Figure 7 left). On the other hand, the values $(1,1,1,1,0,1)$ computed in round 2 (Figure 9 left) are also descendants of the value in Figure 6. By this general definition, the descendant values are obviously ambiguous. However, when considering Lemma 1, at least the correct descendant values are equal to one another since they are fixed.

Note that as a consequence of Lemma 1, if the initiator is a non-faulty processor, its broadcasted value is fixed immediately for all other processors. In the case of a faulty initiator, the sequence of obtained root values might alternate from round to round between 0 and 1. For this case, the following lemma answers the question by which particular round the sequence of root values always possesses the correct agree-value.

Lemma 2 *A faulty value v received on a non-empty path h_f is fixed no later than in round $f \leq t$, if there are f currently faulty processors in the system. Thereby, h_f denotes a path that exclusively contains faulty processors.*

Proof: See [15, Lemma 3.2c]. □

The proof uses the assumption that if all direct descendants of the considered faulty value are fixed, the faulty value itself is fixed. This assumption holds since the faulty value is computed by applying a *majority* function to its direct descendants. The correct descendants are fixed by Lemma 1. Only the faulty descendants have to be investigated. Thus, the same discussion again starts for the faulty descendant values. However, only f faulty processors are in the system. Therefore, the longest possible path which exclusively contains faulty processors has length f. Thereby, all its descendants are correct values which are fixed immediately. Thus, the considered faulty value is fixed at the latest in round f.

To apply this lemma to the root value, we just set $h_f := [Initiator]$, where "*Initiator*" denotes the initiating processor. In our example $h_f := [p_1]$. To summarize the result, we can answer the question by which round the root values have the correct agree-value by: round 0, if the initiator is non-faulty, and round f, if the initiator is faulty.

In order to answer the question how a processor can detect the early stopping round, we pose a more specific question: How can a processor detect if a value is fixed? A motivation for the answer is given by the following lemma:

Lemma 3 *A value on an arbitrary non-empty path h is fixed to 1 (0) if more than (at least) one half of its direct descendants are fixed to 1 (0).*

Proof: The proof follows directly from the definition of function *majority*. □

A combination of Lemma 3 and Lemma 1 implicates the possibility of detecting fixed values. We will demonstrate this by using our example. There, processor p_2 receives the values $(1,1,1,1,0,1)$ (Figure 2a) in round 1. The processor p_2 assumes that the initiator p_1 is faulty — which is true in this scenario. (We postpone the reasons for this assumption to the next section.) Then, at most one faulty value might be in the received value vector $(1,1,1,1,0,1)$. Processor p_2 does not know which one it is. However, it knows that all correct values are fixed immediately (by Lemma 1). Thus, at least $5-1=4$ values are fixed to 1 immediately. Applying function *majority* results in the agree-value 1 independently if the single faulty value will change later. Processor p_2 could decide on value 1 early and stop.

On the other hand, the processors p_5 and p_6 cannot infer such a statement. In round 1, they receive the values $(1,1,1,1,0,0)$ (Figure 2b). Thus at least $4-1=3$ values are fixed to 1 immediately. The result of function *majority* is not yet predictable. In this case, our early stopping algorithm (introduced in Section 4), will compute a value X, since it cannot derive a save statement.

Now, we are able to answer the question how a processor can detect the round in which the sequence of computed root values will always get the same value. It is the round in which the processor detects that the root value is fixed. By Lemma 3, the root value is fixed if more than half of its direct descendants are fixed to the same value. Otherwise, the processor has to investigate their descendants, etc. Up to this point, we only know that the root value indeed is fixed at the latest in round f by Lemma 1 and Lemma 2. Whether the processors are able to detect this, we will discuss in the following section.

Besides detecting fixed values we want to indicate another interesting fact we will exploit in the early stopping algorithm. Not all values a processor receives in Lamport's algorithm affect the agree-value. In other words, the received trees contain *non-interesting* values which might be altered to 0 or 1 without having any influence on the computed root value. We define these two distinguishing classes of values as follows, where the length of a path h, i.e., the number of processors in h, is denoted by $|h|$. Specifically, all paths $h_f \circ h_c$, $|h_f \circ h_c| \geq 1$, we call *interesting paths.* All other paths, we call *non-interesting paths.* Thereby, the path h_f may only contain faulty processors, and the path h_c may only contain non-faulty processors. Furthermore, we speak of *(non-) interesting values* if these values are on (non-)interesting paths. Thus interesting paths are (1) non-empty paths h_c, (2) non-empty paths h_f, and (3) paths $h_f \circ h_c$, where h_f and h_c are non-empty. The following corollary states our above motivated observation:

Lemma 4 *Non-interesting values do not affect the agree-value.*

Proof: See [15, Corollary 3.1]. □

Consider the root value $[Initiator] : v$ itself. Regardless of whether the initiator is faulty or non-faulty, the root value is on an interesting path by definition. If the initiator is non-faulty, we can show that its faulty descendants on a path $[Initiator, p_f]$, where p_f is faulty, do not affect the root value. The reason is that all correct descendants on a path $[Initiator, p_c]$, where p_c is non-faulty, are equal and fixed by Lemma 1. Since there are more correct than faulty descendants, we can apply Lemma 3, so that the root value is fixed independent of the faulty descendants. In the case of a faulty initiator, all direct descendants of the root value are interesting. Then, the argument can again be applied to the correct descendants.

4 Optimal Early Stopping

4.1 An Optimal Early Stopping Algorithm

The early stopping algorithm derived from Lamport's algorithm takes advantage of two ideas. First, it tries to detect fixed values (before round t), and second it uses the knowledge that the modified Lamport algorithm may behave incorrectly on non-interesting paths. Therefore, the algorithm starts a special voting process after each particular round r of the message exchange. When receiving the round-r-tree in round r, the algorithm first tries to detect (or to "ensure") fixed values. Values, which could not be ensured, get the special value X. Values X are assumed to have any value, i.e., either 0 or 1, because they might change in later rounds. In contrast, the detected fixed values cannot change in later rounds. The algorithm can "rely" on them. After this voting step, the received round-r-tree shrinks to a round-$(r-1)$-tree that may contain X values. Then, the usual voting process will be applied to the remaining round-$(r-1)$-tree, which includes X values. Thereby, during the voting process, fixed values "ripple" through the tree. If a processor could detect sufficiently many fixed values so its voting process ends with a root value $[Initiator] : v^{(r)}$ unequal to X, it can decide on this value v as the agree-value and can stop. Otherwise, if the voting process ends with value X, the processor needs to run the next round of the message exchange.

The Ensure Algorithm

After each round r of the message exchange, $r = 1, \ldots, t$, a voting process is applied to the received round-r-tree. This voting process consists of two parts. The first part executes the first voting step, i.e., it computes a round-$(r-1)$-tree from the received round-r-tree. It exclusively applies a function $majority_{Ens}$ to the value vectors in the bottom layer of the received round-r-tree. The second part consists of the execution of all following voting steps until the root value has been computed. It exclusively applies function $majority_X$ to the value vectors in the bottom layer of the stepwise computed round-r_i-trees, $r_i = r-1, r-2, \ldots, 1$. If a processor finishes this voting process with a root value v unequal to X, it decides on v and stops. Otherwise, it joins the next round of message exchange. The following functions are defined:

$$majority_{Ens}((v_1, v_2, \ldots, v_{n-r})) :=$$

$$\begin{cases} 1 & \text{if } |(v_1, v_2, \ldots, v_{n-r})|_1 - (t-r) > \frac{n-r}{2}, \\ 0 & \text{if } |(v_1, v_2, \ldots, v_{n-r})|_0 - (t-r) \geq \frac{n-r}{2}, \\ X & \text{otherwise.} \end{cases}$$

where $|\vec{v}|_x$ counts the occurrences of given value x within value vector $\vec{v}$, and

$$majority_X((v_1, v_2, \ldots, v_{n-r_i})) :=$$

$$\begin{cases} 1 & \text{if } |(v_1, v_2, \ldots, v_{n-r_i})|_1 > \frac{n-r_i}{2}, \\ 0 & \text{if } |(v_1, v_2, \ldots, v_{n-r_i})|_0 \geq \frac{n-r_i}{2}, \\ X & \text{otherwise.} \end{cases}$$

where r_i, $r_i = r-1, \ldots, 1$, indicates the round-r_i-tree, function $majority_X$ is applied to.

Early stopped processors don't send messages. Thus, if a processor obtains an omissive fault on a path $h \circ [p]$ with a non-empty path h, caused by processor p, it replaces this value by

$$h \circ [p] : v_r \ := \ h : v\,.$$

This notation indicates that the value v_r on path $h \circ [p]$ will be replaced by value v. Thereby, the path $h \circ [p]$ remains unchanged. Thus a missing value will be replaced by the value received on the predecessor path. Omissive faults caused by the initiator will be replaced by default value 0.

Using the scenario from Section 2, after the message exchange of round 1, processors p_2, p_3 and p_4 have received the round-1-tree shown in Figure 2a. Processors p_2, p_3 and p_4 compute a round-$0^{(1)}$-tree identical to Figure 1a by applying function $majority_{Ens}$. Because they obtain five 1 values in their round-1-tree, function $majority_{Ens}$ computes 1 since $|\vec{v}|_1 - (t-r) = |(1,1,1,1,0,1)|_1 - (2-1) = 5 - 1 > \frac{7-1}{2} = \frac{n-r}{2}$. They decide on value 1 and stop since their computed root values $[Initiator] : v^{(1)}$ are 1 (unequal to X).

In contrast, processors p_5 and p_6 receive the round-1-tree shown in Figure 10a and compute the round-$0^{(1)}$-tree shown in Figure 10b. Therefore, only p_5, p_6 and p_7 execute round 2. After the message exchange of round 2, the necessary replacements caused by the stopped processors take place. Figure 11 shows the round-2-tree received by processor p_6, where replaced values are indicated by dashed circles.

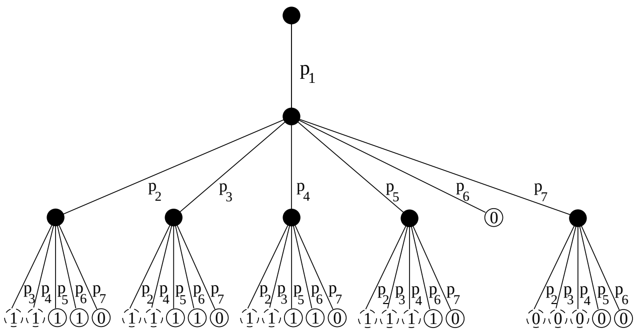


Figure 11: Round-2-tree received in round 2 by processor p_6

Processors p_5 and p_6 start the voting process for the second time. The first voting step computes round-$1^{(2)}$-tree shown in Figure 12a by applying function $majority_{Ens}$. By applying function $majority_X$, the second voting step computes the round-$0^{(2)}$-tree shown in Figure 12b. Therefore, p_5 and p_6 also decide on agree-value 1 and stop.

In order to prove that the Ensure algorithm is a Byzantine Agreement algorithm, we will show that its computed agree-value is equal to the agree-value computed by Lamport's algorithm in the same scenario. Therefore, we want to compare *every* value a processor obtains in the Ensure algorithm with its corresponding value in Lamport's algorithm. Instead of the original Lamport algorithm, we want to consider our modified version of Lamport's algorithm (page 5) which executes the whole voting process after each round of the message exchange. Obviously, this modified version computes the same agree-value as Lamport's algorithm (page 3). Then, each value a processor obtains in the Ensure algorithm possesses a corresponding value obtained in our version of Lamport's algorithm. We need to compare values in the Ensure algorithm with their corresponding values in our version of Lamport's algorithm. We say value $h : v'$ *conforms* with Lamport's algorithm, if a value $h : v$ obtained by processor p in Lamport's algorithm and value $h : v'$ obtained by the same processor p within the same scenario in the Ensure algorithm are equal. Because of Lemma 4, it is sufficient to investigate interesting values. As a matter of fact, non-interesting values computed by the Ensure algorithm do not always conform with Lamport's algorithm.

The proof of the Ensure algorithms is too lengthy to present in this paper. The interested reader is referred to [15] for details. Essentially, the proof incorporates three steps that show: (1) All interesting values computed unequal to X conform with Lamport's algorithm as long as no processor stops broadcasting. In particular, the computed root value conforms with Lamport's algorithm, if unequal to X. (2) The maximum number of rounds a non-faulty processor needs to run is $min\{f+2, t+1\}$ in the case of f actually faulty processors. (3) Replacing missing values does not affect the computation of the root value.

Theorem 1 *The Ensure algorithm is a Byzantine Agreement algorithm that stops not later than in round* $min\{f+1, t\}$.

Proof: See [15, Theorem 3.1]. □

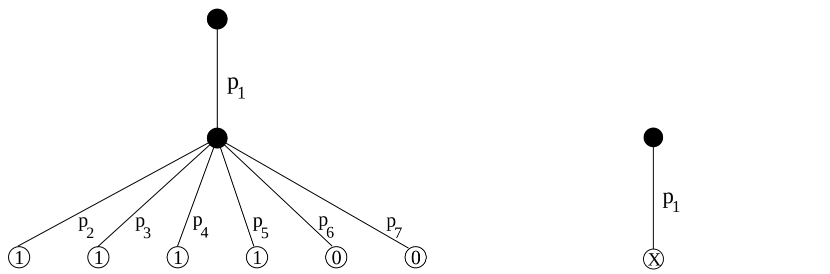
4.2 Behavior of Early Stopping Algorithm

In the case of a non-faulty initiator, the algorithm usually shows a better performance than with a faulty initiator:

Corollary 1.1 *If the initiator is non-faulty, then the following holds: (i) in the case of up to* $\lceil t/2 \rceil$ *faulty processors, all non-faulty processors stop after 2 rounds, (ii) in the case of* $(t-k)$ *faulty processors,* $k = 0, 1, \ldots, \lfloor t/2 \rfloor - 1$, *all non-faulty processors stop after at most* $t + 1 - 2k$ *rounds.*

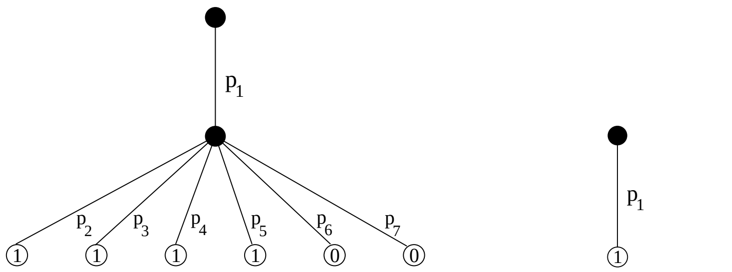
Proof: See [15, Corollary 3.3]. □

Figure 13 illustrates the statement of Corollary 1.1 by demonstrating the difference in the number of



a) Received round-1-tree b) Computed round-$0^{(1)}$-tree

Figure 10: Trees of processors p_5 and p_6 in round 1



a)Round-$1^{(2)}$-tree b) Round-$0^{(2)}$-tree

Figure 12: Computed trees of p_5 and p_6 in round 2

rounds if the initiator is non-faulty or faulty. The number of rounds indicates the maximum number of rounds a non-faulty processor needs to execute in the given scenario. The number of faults represents the number of faulty processors in the system.

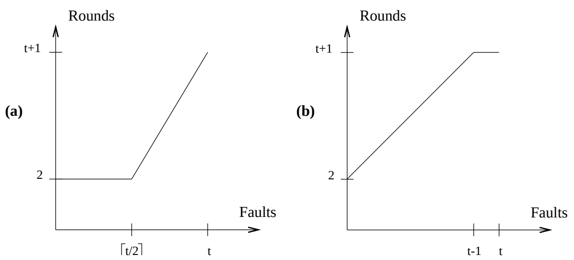


Figure 13: Rounds vs. Faults for a) non-faulty and b) faulty initiator

5 Summary

This paper presented a new early stopping algorithm as a solution to non-authenticated Byzantine Agreement. The algorithm introduced consists of the application of two simple functions, requires simple message replacement rather than message reconstruction as used in [5], is optimal in terms of rounds and most of all has an algorithmic definition. It is derived from the agreement algorithm by Lamport [18].

Current research focuses on adapting the early stopping algorithm to more realistic and refined fault-models, thus taking advantage of fault taxonomies. This hybrid approach is expected to improve early stopping as well as achieving higher reliability.

References

[1] Azadmanesh, M.H., and Krings, A.W., "Analysis of a Realistic Fault Model for Large Distributed Systems", *Systems Science Journal*, Vol. 22 (3), pp. 29-51, 1996.

[2] Babaoglu, O., and R. Drummond, "Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, pp. 546-554, June 1985.

[3] Bar-Noy, A., Dolev, D., "Consensus Algorithms with One-Bit Messages", *Distributed Computing*, Vol. 4, No. 3, pp. 105-110, 1991.

[4] Bar-Noy, Amotz, and Danny Dolev, "Shifting Grear: Changing Algorithms on the Fly to Expedite Byzantine Agreement", *Information and Computation*, Vol. 97, pp. 205-233, 1992.

[5] Berman, P., Garay, J.A., Perry, K.J., "Optimal Early Stopping in Distributed Consensus", *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG '92)*, LNCS 647, pp. 221-237, Springer-Verlag, Nov. 1992.

[6] Coan, B., "Efficient Agreement Using Fault Diagnosis", *Proceedings of the 26th Allerton Conference on Communication, Control and Computing*, University of Illinois, Urbana, Illinois, pp. 663-672, 1988.

[7] Dolev, D., "The Byzantine Generals Strike Again", *Journal of Algorithms*, Vol. 3, No. 1, pp. 14-30, Mar. 1982.

[8] Dolev, D., et. al., "Eventual Is Earlier than Immediate", *23rd Annual Symposium on Foundations of Computer Science*, pp. 196-203, Chicago, Nov. 1982.

[9] Dolev, D., Strong, H., R., "Requirements for Agreement in a Distributed System", *Proc. 2nd International Symposium on Distributed Data Bases*, Berlin, Sep. 1982.

[10] Dolev, D., Reischuk, R., "Bounds on Information Exchange for Byzantine Agreement", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 132-140, Ottawa, Canada, Aug. 1982.

[11] Dolev, D., et. al., "Reaching Approximate Agreement in the Presence of Faults", *Proc. 3^{rd} Symposium on Reliability in Distributed Systems*, pp. 145-154, Oct. 1983.

[12] Dolev, D., Strong, H.R., "Authenticated Algorithms for Byzantine Agreement", *SIAM Journal on Computing*, Vol. 12, No. 4, pp. 656-666, Nov. 1983.

[13] Dolev, D., Reischuk, R., Strong, H.R., "Early Stopping in Byzantine Agreement", *Journal of the ACM*, Vol. 37, No. 4, pp. 720-741, Mar. 1990.

[14] Dolev, D., and Dalia Malki, "The Transis Approach to High Availability Cluster Communication", *Communication of the ACM*, Vol. 39, No. 4, pp. 64-70, April 1996.

[15] Feyer, T., and Krings, A.W., "Optimal Early Stopping in Byzantine Agreement", University of Idaho, Computer Science Dept., *Technical Report LAL-98-05*, 1998.

[16] Fischer, M., Lynch, N., "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, Vol. 14, No. 4, pp. 183-186, June 1982.

[17] Fischer, M.J., Nancy, A.L., Merritt, M., "Easy Impossibility Proofs for Distributed Consensus Problems", *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pp. 59-70, Minaki, Ontario, Canada, Aug. 1985.

[18] Lamport, L., et. al., "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382-401, July 1982.

[19] Moser, L.E., P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System,", *Communications of the ACM*, April 1996.

[20] Pease, M., Shostak, R., and Lamport L., "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, Vol. 27, No. 2, pp. 228-234, Apr. 1980.

[21] Schneider, F., "Understanding Protocols for Byzantine Clock Synchronization", Department of Computer Science, Cornell University, Report 87-859, August 1987.

[22] Thambidurai, P., Park, Y.-K., "Interactive Consistency with Multiple Failure Modes", *7th Symp. on Reliable Distributed Systems*, pp. 93-100, OH, Oct. 1988.

[23] Toueg, S., Perry, K., Srikanth, T., "Fast Distributed Agreement", *SIAM Journal of Computing*, Vol. 16, No. 3, pp. 445-457, June 1987.