Fast Agreement in Networks with Byzantine Nodes

Bogdan S. Chlebus
School of Computer and Cyber Sciences, Augusta University, GA, USA

Dariusz R. Kowalski
School of Computer and Cyber Sciences, Augusta University, GA, USA
SWPS Uniwersytet Humanistycznospołeczny, Warsaw, Poland

Jan Olkowski
Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski, Warsaw, Poland

Abstract

We study Consensus in synchronous networks with arbitrary connected topologies. Nodes may be faulty, in the sense of either Byzantine or proneness to crashing. Let t denote a known upper bound on the number of faulty nodes, and D_s denote a maximum diameter of a network obtained by removing up to s nodes, assuming the network is $(s+1)$-connected. We give an algorithm for Consensus running in time $t + D_{2t}$ with nodes subject to Byzantine faults. We show that, for any algorithm solving Consensus for Byzantine nodes, there is a network G and an execution of the algorithm on this network that takes $\Omega(t + D_{2t})$ rounds. We give an algorithm solving Consensus in $t + D_t$ communication rounds with Byzantine nodes using authenticated messages of polynomial size. We show that for any numbers t and $d > 4$, there exists a network G and an algorithm solving Consensus with Byzantine nodes using authenticated messages in fewer than $t + 3$ rounds on G, but all algorithms solving Consensus without message authentication require at least $t + d$ rounds on G. This separates Consensus with Byzantine nodes from Consensus with Byzantine nodes using message authentication, with respect to asymptotic time performance in networks of arbitrary connected topologies, which is unlike complete networks. Let f denote the number of failures actually occurring in an execution and unknown to the nodes. We develop an algorithm solving Consensus against crash failures and running in time $\mathcal{O}(f + D_f)$, assuming only that nodes know their names and can differentiate among ports; this algorithm is also communication-efficient, by using messages of size $\mathcal{O}(m \log n)$, where n is the number of nodes and m is the number of edges. We give a lower bound $t + D_t - 2$ on the running time of any deterministic solution to Consensus in $(t+1)$-connected networks, if t nodes may crash.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases distributed algorithm, network, Consensus, Byzantine fault, message authentication, node crash, lower bound

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.30

Funding *Dariusz R. Kowalski*: Supported by the Polish National Science Center (NCN) grant UMO-2017/25/B/ST6/02553.

1 Introduction

We consider distributed algorithms solving Consensus in synchronous networks of arbitrary connected topologies. A network has n nodes, each with a unique name. Links connecting nodes are reliable and can transmit messages in both directions. Nodes are prone to failures. We want a distributed algorithm to produce an agreement on a common decision value across the whole network. The feasibility of reaching agreement in a network with faulty nodes depends on its connectivity properties, as showed by Dolev et al. [14]. We consider time performance of distributed and deterministic algorithms solving Consensus in such networks for which Consensus solutions exist. The ultimate goal is to optimize time performance, but next to minimize the initial knowledge of nodes and message size.

A network is $(s+1)$-connected if removing at most s nodes does not break it into multiple connected components. We let D_s denote a maximum diameter of a network obtained by removing up to $s > 0$ nodes, assuming the network is $(s+1)$-connected. An upper bound on the number of faulty nodes is denoted either by t or by f. The difference is that t denotes an upper bound on the number of faulty nodes that is known, in being usable in codes of algorithms, while f is a number of faulty nodes actually occurring in an execution and unknown. This convention extends to network properties determined by the numbers of faults, with respect to either being known or unknown. In particular, properties depending on t, like the magnitude of D_{2t}, are known and can be a part of code, while properties depending on f, like the magnitude of D_f, are not known and cannot be referred to directly.

Bounds on the running time of Consensus solutions have been extensively studied in complete networks. It was shown in [1, 20] that $t+1$ rounds are sufficient and necessary to solve Consensus in the case of node crashes. The number of crashes f actually occurring in an execution could be less than t. This leads to the postulate of early stopping, see [17], which can be interpreted as scaling running time to the number of faults: we want all nodes to decide and halt as early as possible, in a number of rounds that depends on f, and possibly on t as well. Early stopping was studied for complete networks under various models of failures, see [7, 8, 17, 24, 28, 35]; it was established that $\min\{f+2, t+1\}$ rounds are sufficient and necessary for all the nodes to reach agreement and halt. We extend to arbitrary connected topologies the concept of scalability of time of a Consensus algorithm to the number of failures in an execution.

A summary of the results. We give an algorithm solving Consensus with Byzantine modes in $t+D_{2t}$ rounds, which is presented in Section 3. The algorithm works under the assumption that node degrees are greater than $3t$. (In complete graphs, the condition on degrees to be greater than $3t$ is equivalent to having $t < n/3$, which is necessary for solvability of Consensus with Byzantine nodes in such networks.) We show that, for any algorithm solving Consensus for Byzantine nodes, there is a network G and an execution of the algorithm on this network that takes $\Omega(t + D_{2t})$ rounds, which is presented in Section 4. We give a Consensus solution with Byzantine nodes using authentication of messages that runs in $t+D_t$ rounds, in networks with node degrees at least $2t$ and using messages of size polynomial in n, see Section 5. We show that, for any t and $d > 4$, there exists a network G and an algorithm solving Consensus with Byzantine nodes using authenticated messages in fewer than $t+3$ rounds on G, but every algorithm solving Consensus without message authentication requires at least $t+d$ rounds on this network. This separates the model of networks with Byzantine nodes from the model with Byzantine nodes using message authentication, with respect to asymptotic time performance of Consensus solutions in networks of suitably connected topologies; such a difference does not hold for complete networks. We develop an early-stopping algorithm solving Consensus against crash failures and running in time $\mathcal{O}(f + D_f)$, where nodes know their names and can differentiate among ports, see Section 6. This algorithm is communication-efficient, in that nodes use messages of $\mathcal{O}(m \log n)$ size, where n is the number of nodes and m is the number of edges. We give a lower bound $t + D_t - 2$ on the running time of any deterministic solution to Consensus in $(t+1)$-connected networks, if up to t nodes may crash, see Section 7. Our algorithms for Consensus with Byzantine nodes and Consensus with Byzantine nodes along with message authentication rely only on nodes knowing their names and the parameters

t and also either the bound D_t or D_{2t}, unlike previously known solutions in these models, which assumed knowledge of the whole network's topology. The results of this paper, along with some other relevant facts, are summarized in Table 1.

Table 1 A summary of algorithms and lower bounds. The meaning of notations is as follows: letter n denotes the number of nodes, m is the number of links, t denotes a known upper bound on the number of faulty nodes, and f an actual number of node failures in an execution. The asterisk $*$ marks the contributions of this paper.

model	algorithm	time performance	message size	connectivity	remarks	lower bound on time
Byzantine faults	Dolev et al. [14]	$t \cdot D_{2t}$	$\mathcal{O}(n^5)$	$2t+1$	topology is known	-
	FAST-BYZANTINE Section 3 $*$	$t + D_{2t}$ $*$	exponential	$2t+1$	node degrees $\geq 3t$ D_{2t} is known	$\Omega(t + D_{2t})$ Section 4 $*$
Byzantine local broadcast	Khan et al [25]	exponential	exponential	$\lfloor \frac{3}{2}t \rfloor + 1$	node degrees $\geq 2t$ is necessary	-
	Khan et al. [25]	$\mathcal{O}(n)$	exponential	$2t$	-	-
Byzantine message authentication	Bansal et al. [4]	-	-	-	topology is known monitoring model	-
	FAST-AUTHENTICATED Section 5 $*$	$t + D_t$ $*$	polynomial in n $*$	$t+1$	node degrees $\geq 2t$ D_t is known	$t + D_t - 2$ $*$
crash failures	EARLY-STOPPING-CRASHES Section 6 $*$	$\mathcal{O}(f + D_f + 3)$ $*$	$\mathcal{O}(m \log n)$ $*$	$f+1$	f actual number of failures f, D_f not known	$f + D_f - 2$ Section 7 $*$

Previous and related work. Consensus has been among most popular algorithmic problems studied in distributed systems and communication networks, see [3, 23, 27, 30]. The problem of distributed agreement in systems prone to faults was first considered by Pease at al. [29], Dolev [14] and Lamport et al. [26]. Pease at al. [29] proposed an algorithm for Byzantine faults that has nodes share all their information acquired over time, known as "exponential information gathering," with a further modification given by Bar-Noy et al. [5]. This approach to algorithm design requires nodes to send exponentially long messages and process an exponential amount of information, in the number of nodes n. Pease at al. [29] and Lamport et al. [26] considered reaching agreement with Byzantine faults in message-passing systems with authenticated messages. Dolev and Strong [18] gave a simple algorithm for Byzantine nodes with authentication of messages; Sirikanth and Toueg [31] showed how to implement that algorithm in the model of Byzantine faults and local broadcast, in which every node sends identical messages to every neighbor in each round.

A bound $t < n/3$ on the number of Byzantine nodes t was shown to be necessary for solvability of Consensus by Pease at al. [29], Dolev [14] and Lamport et al. [26]. Fisher and Lynch [19] proved a lower bound $t + 1$ on the number of communication rounds, which holds for crashes. Dolev and Reischuk [16] gave a lower bound $\Omega(nt)$ on the number of communication bits necessary to solve Consensus with Byzantine faults, which becomes $\Omega(n^2)$ if the number of Byzantine nodes satisfies $t = \Omega(n)$. Methods to show impossibilities and lower bounds for distributed-computing problems, including reaching distributed agreement, are reviewed in [2]. Garay and Moses [22] showed how to reach agreement with a polynomial number of communication bits and a polynomial local computation, subject only to the bound of $t < n/3$ on the number of Byzantine nodes while staying within $t+1$ communication rounds. Next, we review previous work on distributed agreement solutions that scale well with respect to performance metrics. We consider an algorithm scaling its running time well if it is either fast, by running in time $\mathcal{O}(t+1)$, or early stopping, by running in time $\mathcal{O}(f+1)$. Berman and Garay [6] developed an algorithm for Byzantine faults which uses messages carrying just one input value, so a message is of constant size if the range of input values is of constant size; the algorithm works for $t < n/4$. Galil et al. [21] developed an algorithm for crashes using $\mathcal{O}(n)$ messages, thus showing that this number of messages is optimal; this algorithm runs in over-linear time $\mathcal{O}(n^{1+\varepsilon})$, for a parameter $0 < \varepsilon < 1$; that paper gave an early-stopping algorithm of message complexity $\mathcal{O}(n + fn^{\varepsilon})$, for any $0 < \varepsilon < 1$. Chlebus and Kowalski [9] developed a gossiping algorithm coping with crashes and applied it to develop a fast solution to Consensus which sends $\mathcal{O}(n \log^2 t)$ messages, provided that $n - t = \Omega(n)$. Chlebus and Kowalski [10] developed a deterministic algorithm which is early-stopping and globally scales communication by sending $\mathcal{O}(n \log^5 n)$ messages. Chlebus at al. [12] gave a fast deterministic Consensus algorithm that sends $\mathcal{O}(n \log^4 n)$ bits, and showed that no deterministic Consensus algorithm can be locally scalable with respect to message complexity. Chlebus and Kowalski [11] gave a randomized Consensus solution terminating in $\mathcal{O}(\log n)$ expected time, while the expected number of bits that each process sends and receives against oblivious adversaries is $\mathcal{O}(\log n)$, assuming that a bound t on the number of crashes is a constant fraction of the number of nodes n. Dolev and Lenzen [15] showed that any crash-resilient Consensus algorithm deciding in $f+1$ rounds has worst-case message complexity $\Omega(n^2 f)$. Chlebus et al. [13] gave a scalable quantum algorithm to solve binary Consensus, in a system of n crash-prone quantum processes, which works in $\mathcal{O}(\text{polylog } n)$ time sending $\mathcal{O}(n \text{ polylog } n)$ qubits against the adaptive adversary.

Here is a brief review of previous work on reaching agreement in networks beyond the complete network topologies. The following assumptions about networks and distribution of faults were shown to be necessary and sufficient for solvability of the problem: for Byzantine nodes, networks need to be $(2f+1)$-connected with the number of faulty nodes less than $n/3$, while for Byzantine nodes with authentication of messages and for crash failures they need to be $(f+1)$-connected, see [3, 27]. Dolev [14] showed that solving Consensus with t Byzantine nodes requires at least $3t+1$ nodes in total and network connectivity at least $(2t+1)$; see also [20]; that paper [14] gave an algorithm relying on knowing the network's topology and solving Consensus in any network satisfying these conditions. Khan at al. [25] considered Consensus in networks in the local-broadcast model, in which every node sends the same message to every neighbor in a round, including Byzantine nodes. They showed that in order for Consensus to be solvable in that model, a network needs to be connected and with each node's degree of at least $2t$. Their algorithms rely on knowing a network's topology; one algorithm has an exponential running time, and another has $\mathcal{O}(n)$ running time, in networks that are $2f$-connected. Bansai et al. [4] considered Consensus in arbitrary networks with Byzantine faults such that nodes can authenticate messages, subject to allowing an adversary to monitor up to k nodes to forge their messages; the paper gave tight conditions on network connectivity referring to the values of t and k to make Consensus solvable, assuming additionally that the network's topology is known. Tseng and Vaidya [33] studied solvability of Consensus in networks with Byzantine nodes and uni-directional links; such networks are modeled as directed graphs. Surveys of related work on reaching distributed agreement are given in [32, 34].

2 Preliminaries

A distributed system is modeled as a simple graph, in which vertices represent nodes and edges represent bi-directional links connecting pairs of nodes. We use letter n to denote the number of vertices and letter m for the number of edges. A graph G is $(s+1)$-*connected* if removing up to s nodes from G does not produce a subgraph of G with at least two connected components. For an $(s+1)$-connected graph G, the notation $D(G,s)$, for an integer $s>0$, denotes the s-*diameter of* G, which is a maximum diameter of a graph obtained by removing s nodes from G. If an $(s+1)$-connected graph G is understood from context, then we use the notation D_s for $D(G,s)$.

Networks are synchronous, in that an execution of a communication algorithm is partitioned into global rounds; an execution starts for all nodes at the same round. During a round, a node may send messages to all neighbors, including itself, and receive all messages sent to it in this round. Links are considered fully reliable, in that no messages are lost, duplicated, nor otherwise modified or corrupted.

Nodes are prone to failures. A node *crashes* at a round when it stops all activity beyond this round and never resumes it again in an execution; some messages sent by a node in a round it crashes may be delivered. A node fault is *arbitrary* or *Byzantine* if the node may undergo arbitrary state transitions in an execution.

We say that a network is equipped with a mechanism to *authenticate messages* if a copy of a message received by a node can be embedded in its future messages such that the authenticity of the contents of the included copy can be verified beyond doubt. We abstract from a mechanism to implement authentication of messages, simply assuming that it is available to every node in a network. If Byzantine faults of nodes are combined with authentication of messages, this is understood such that faulty nodes cannot forge messages, which imposes a restriction on how "arbitrary" their state transitions can be.

In the problem of *Consensus*, each node p is given an *input value* denoted $\texttt{input}_p$. We say that a node *decides on a value* x if it sets a dedicated private variable to this value x; such a decision is considered irrevocable in an execution. Informally, the goal for all nodes is to eventually decide on one common value. We use standard specifications of Consensus expressed as agreement, validity, and termination, depending on the kind of faults, which are either Byzantine or crashes; see [3, 27, 30].

Our algorithms are deterministic and their performance is measured in the worst-case sense. One performance metric is *time*, understood as a number of communication rounds until termination. Another performance metric is *message size*, meant to be an upper bound on the number of bits a node transmits to a neighbor in a round.

We say that some aspect of a distributed system is *known* if nodes can use it in the code of an algorithm. Each node is equipped with a unique *name*, which it knows. We assume a name can be encoded by $\mathcal{O}(\log n)$ bits if transmitted in messages. A node communicates with its neighbors by transmitting messages via ports, one port per neighbor. Ports at a node are distinguishable in the following sense: if a node wants to send a message then it can specify by which port the message is to be transmitted, and if a node receives a message then the node can identify which port delivered the message.

Letter t denotes a known upper bound on the number of faulty nodes; if an algorithm refers to t then its correctness needs to hold only in executions in which up to t nodes are faulty. Letter f denotes a number of faults actually occurring in an execution; this parameter f is not known. By analogy, if we refer to network parameters like D_t or D_f then D_t is assumed to be known but D_f is not assumed to be known. If nodes only know their names and can distinguish ports then this is the model of *minimal knowledge*. We say that a node *knows its neighbors* if it can map ports on the names of nodes that receive messages transmitted via each of these ports. If neighbors are not initially known, then this can be discovered by having every node send its name to every neighbor, which takes one round but contributes $\mathcal{O}(m \log n)$ bits to total communication.

3 Fast Byzantine Consensus

We present a distributed algorithm Fast-Byzantine solving Consensus in arbitrary networks with Byzantine node faults in asymptotically-optimal time. Consensus in arbitrary networks with bi-directional links was already studied by Lamport et al. [26] and Dolev in [14]. These papers concentrated on solvability of Consensus, as determined by network's connectivity, and did not attempt to optimize time and communication performance; most importantly, they gave algorithms relying on knowing network's topology. We propose an approach to solve Consensus in a deterministic and distributed manner that does not require knowing network's topology, and works assuming sufficiently strong connectivity, while each node's degree is at least $3t$. We use a paradigm to solve Consensus in complete networks with Byzantine nodes, which is known as "exponential information gathering," see [5, 29]. An execution is partitioned into two parts. In the first part, which takes $t+1$ rounds, information is exchanged among the nodes. In a round i, nodes store the information gained so far in a a tree of height i. Leaves store the input value that passes trough subsequent nodes with names belonging to the path from the root to a respective leaf. When the first part is over, the trees have height $t+1$. In the second part, the tree is evaluated from the lowest level to the highest one by a local computation in each node such that the value at the root represents a decision. We divide executions of our algorithm into four stages, described next, referring to notations used in the pseudocode of the algorithm Fast-Byzantine given in Algorithm 1. All non-faulty nodes communicate by sending messages of the format $(\overline{p_1 p_2 \dots p_i}, W)$, which is an ordered pair such that $p_1, p_2, \dots, p_i$ is a sequence of names of nodes that forwarded W from p_1 through p_i. A message of this format is *well-formed* if all node names in the sequence $p_1, p_2, \dots, p_i$ are distinct.

The initialization stage. This stage initializes variables $\texttt{Paths}[i]$ representing sets and is performed in the very beginning of the first round. Each set $\texttt{Paths}[i]$ at p, for $2 \le i \le t+1$, will store well-formed pairs $(\overline{s_1 s_2 \dots s_{i-1} s_i}, W)$, where $s_i = p$. The set $\texttt{Paths}[1]$ at node p is initialized to one ordered pair, which has a single-vertex path p as the first component and the input value $\texttt{input}_p$ of node p as the second component. The sets $\texttt{Paths}[i]$, for $2 \le i \le t+1$, are initialized to an empty set each.

The local authorization stage. In this stage, nodes work to deliver their input values to other nodes at distance at most t. A node p considers only messages that passed through $t+1$ different nodes, including itself at the end: these are pairs of the form $(\overline{s_1 s_2 \dots s_{t+1}}, W)$ with all distinct vertices on the path. To collect such pairs, each node tries to propagate its input value along paths of length $t+1$: each node forwards all the received messages it considers legitimate to its neighbors during t consecutive rounds. More precisely, whenever a node p receives a pair $(\overline{s_1 s_2 \dots s_{i-1}}, W)$ at round $i-1$, then it checks two conditions: (1) did the last node on the path s_{i-1} send the message, and (2) is the path $\overline{s_1 s_2 \dots s_{i-1}, p}$ well-formed. If both conditions are met, node p forwards the pair $(\overline{s_1 s_2 \dots s_{i-1}, p}, W)$ to its neighbors in the next round. An upper bound $3t$ on a node's degree ensures that each non-faulty node can propagate its input value in a verifiable manner to other nodes via sufficiently many well-formed sequences of nodes.

Algorithm 1 A pseudocode of algorithm Fast-Byzantine for a node p, structured into four stages. A pseudocode of procedure Deliver is in Algorithm 2. Variable `Leaves`$[q]$ stores paths of nodes, each path starting with node q.

`algorithm` Fast-Byzantine

— initialization —

1. initialize `Paths`$[1] = \{(\bar{p}, \texttt{input})\}$
2. initialize `Paths`$[2], \ldots,$ `Paths`$[t+1]$ as empty sets

— local authorization —

3. **for** $i \leftarrow 1$ **to** t **do**
  a. send `Paths`$[i]$ to each neighbor
  b. **foreach** neighbor q **do**
    i. receive `Paths`$_q[i]$ from q
    ii. **foreach** $(\overline{s_1 s_2 \ldots s_i}, W)$ in `Paths`$_q[i]$ **do**
      if $s_i = q$ and $(\overline{s_1 s_2 \ldots s_i p}, W)$ is well-formed   \\\\ *q is just before p on the path*
        add $(\overline{s_1 s_2 \ldots s_i p}, W)$ to `Paths`$[i+1]$

— global communication —

4. $(\texttt{Nodes}, M_1, \ldots, M_{|\texttt{Nodes}|}) \leftarrow$ Deliver(`Paths`$[t+1]$)
5. **foreach** q in `Nodes` **do**
  a. construct set `Leaves`$[q]$ based on sets of paths $M_1, \ldots, M_{|\texttt{Nodes}|}$.

— local computation —

6. **foreach** q in `Nodes` **do**
  a. construct `Tree`$[q]$ based on the set `Leaves`$[q]$
  b. **for** $i \leftarrow t$ **to** 1 **do**   \\\\ *evaluation of tree for node q*
    i. **foreach** vertex $\overline{s_1 s_2 \ldots s_i}$ of `Tree`$[q]$ **do**
      A. **if** $\overline{s_1 s_2 \ldots s_i}$ is active `then`
        set `resolve`$(\overline{s_1 s_2 \ldots s_i})$ to majority among values `resolve`$(\overline{s_1 s_2 \ldots s_i s_{i+1}})$
        such that $\overline{s_1 s_2 \ldots s_i s_{i+1}}$ is an active vertex of `Tree`$[q]$
      B. **else** `resolve`$(\overline{s_1 s_2 \ldots s_i}) \leftarrow \perp$
7. `decision` $\leftarrow$ the majority among values `resolve`(`Tree`$[q]$`.root`) for q in `Nodes`

The global communication stage. In this stage, each node p tries to propagate all legitimate messages it has accumulated in the previous stage to all other nodes in the network. To this end, node p invokes procedure Deliver, which returns a set `Nodes` of nodes that p heard from along with sets $M_1, \ldots, M_{|\texttt{Nodes}|}$ that node p could validate as information they wanted to propagate. Node p discovers what a node q in `Nodes` wants to send by working with a set of paths of nodes traversed by messages that start at node q and arrive at p after suitable forwards. This stage begins by invoking procedure Deliver, which has its pseudocode in Algorithm 2. It takes a node's name and the information to be sent as parameters, and returns a set `Nodes` of names of nodes along with the information that these nodes wanted to propagate. In order to achieve this, nodes work to propagate messages between any two non-faulty nodes trough all possible paths of length D_{2t}: in consecutive D_{2t} rounds, each node propagates all received valid paths, by first appending its name at the end of a received path to form a new path. The assumed connectivity $2t + 1$ ensures that at least $t + 1$ paths

Algorithm 2 A pseudocode of procedure DELIVER for a node p. Parameter I denotes information to be sent. Variable M_q stores paths of nodes, each ending with node q. Variable `Relay` denotes a set of pairs received from neighbors to be forwarded. An element $(\overline{s_1 s_2 \ldots q}, W)$ in `Relay` is considered *confirmed* if there are at least $t+1$ received pairs such that all the pairs carry value W and the paths by which they reached node p are disjoint, except for the endpoints s_1 and p.

`procedure` DELIVER (I)

---

1. initialize sets `Relay` $\leftarrow \{(\overline{p}, I)\}$; `Nodes` $\leftarrow \emptyset$
2. **for** $i \leftarrow 1$ **to** D_{2t} **do**
   a. **foreach** x in `Relay` **do** send x to each neighbor unless x was already sent
   b. **foreach** neighbor q **do**
      i. **foreach** $(\overline{s_1 s_2 \ldots s_i}, W)$ received from q **do**
         A. **if** $s_i = q$ and $(\overline{s_1 s_2 \ldots s_i p}, W)$ is well-formed \\ *q is last node on received path*
            then add $(\overline{s_1 s_2 \ldots s_i p}, W)$ to `Relay`
3. **foreach** confirmed $(\overline{s_1 s_2 \ldots s_k}, W)$ in `Relay` **do**
   add s_1 to set `Nodes` ; assign $\mathtt{M}_{s_1} \leftarrow W$
4. `return` $(\mathtt{Nodes}, M_1, \ldots, M_{|\mathtt{Nodes}|})$

---

are free of faulty nodes. Thus, the information the other node q wanted to propagate could be computed by node p by considering the maximum set of received disjoint paths starting at q with the same propagated value, with at least $t+1$ such disjoint paths. A node p that received $(\overline{s_1 s_2 \ldots q}, W)$ from a neighbor considers W a *confirmed value from* s_1 if there are at least $t+1$ pairs received from neighbors such that all the pairs carry value W and the paths by which they reached node p are disjoint, except for the endpoints s_1 and p. When the global communication stage is over, then each node p wants to store a record of the preceding communication about all nodes q in the set `Leaves`$[q]$, such that the input value of q could be retrieved from this set. This is indeed doable in non-faulty nodes q.

The local computation stage. For a node q in `Nodes`, we treat paths in the set `Leaves`$[q]$ as leaves of a tree. More precisely, these are paths of the form $\overline{s_1 s_2 \ldots s_{t+1}}$ such that a pair $(\overline{s_1 s_2 \ldots s_{t+1}}, ?)$ is in `Leaves`$[q]$, where notation ? is a wildcard character. The tree for a node q in `Nodes` is denoted `Tree`$[q]$ and referred to as the *tree for* q. If a sequence x is a prefix of a sequence y then we say that a vertex x is a ancestor of a vertex y and y is a descendant of x; immediate ancestors and descendants are parents and children. Vertex $\overline{q}$ is a root of tree `Tree`$[q]$, denoted `Tree`$[q]$`.root`. The property to be an *active vertex* is defined recursively as follows: each leaf is active and a vertex with at least $t+1$ active children is such as well. Node p associates a value `resolve`$(\overline{s_1 s_2 \ldots s_k})$ with each vertex $\overline{s_1 s_2 \ldots s_k}$ of tree `Tree`$[q]$. This is a unique value such that node p believes node s_k received it from node s_{k-1}, who received this value from node s_{k-2}, who in turn received the value from node s_{k-3}, and so on. A systematic approach to compute `resolve` values is as follows. If $\overline{s_1 s_2 \ldots s_{t+1}}$ is a leaf then `resolve`$(\overline{s_1 s_2 \ldots s_{t+1}})$ equals W taken from the pair $(\overline{s_1 s_2 \ldots s_{t+1}}, W)$ in the set `Leaves`$[q]$. The value of `resolve` for an inner node is the majority of values `resolve` of its children. Finally, a node determines the input value of a node q to `resolve`(`Tree`$[q]$`.root`). A node decides on the majority of the input values of all nodes q in `Nodes`.

▶ **Theorem 1.** *Algorithm* FAST-BYZANTINE *solves Byzantine Agreement in* $t + D_{2t}$ *communication rounds, provided* $n > 3t$.

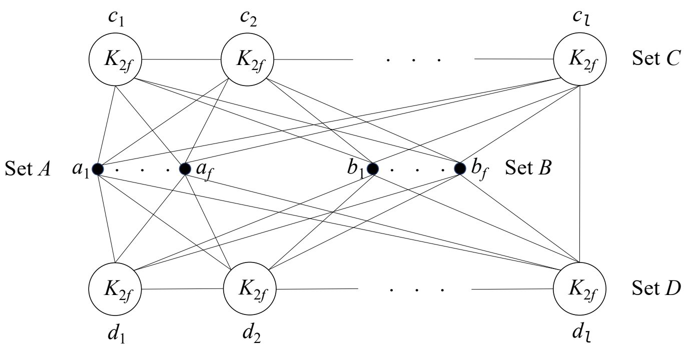

Figure 1 A visualization of graph $G_{t,\ell}$. The parameter t is an upper bound on the number of faulty nodes, while the parameter ℓ denotes a number of cliques in the sets C and D. An edge between either two cliques or a node and a clique indicates that every node at one end is connected with all nodes at the other end.

4 A Lower Bound for Byzantine Faults

In this section, we present a lower bound on time performance of algorithms solving Consensus with Byzantine nodes. The performance of algorithm FAST-BYZANTINE from Section 3 matches this lower bound. This shows that the parameter $D(G, 2t)$ of a network G captures time-optimality of algorithms solving Consensus with Byzantine faults of nodes in networks of general topologies. Let $G_{t,\ell}$ denote a graph with topology as depicted in Figure 1. Its nodes are partitioned into four sets A, B, C, D. Each of the two sets A and B has t elements: $A = \{a_1, \ldots, a_t\}$ and $B = \{b_1, \ldots, b_t\}$. The set C is partitioned into ℓ disjoint subsets, denoted by $c_1, \ldots, c_\ell$, each of $2t$ nodes, and analogously, the set D is also partitioned into ℓ disjoint subsets, denoted by $d_1, \ldots, d_\ell$, each of $2t$ nodes. The edges of the graph are defined as follows:

- for every i such that $1 \le i \le \ell$, all pairs of nodes in c_i are connected to produce a clique, and all nodes in d_i are connected to produce a clique;
- for every i such that $1 \le i \le \ell - 1$, every node in c_i is connected to all nodes in c_{i+1}, and every node in d_i are connected with all nodes in d_{i+1};
- every node in clique c_ℓ is connected with all nodes in clique d_ℓ;
- for every i such that $1 \le i \le \ell$, node a_i and node b_i is connected with all nodes in $C \cup D$.

Graph $G_{t,\ell}$ is $2t + 1$ connected, the parameter $D(G_{t,\ell}, t)$ of such a graph equals 2, regardless of t and ℓ, while $D(G_{t,\ell}, 2t)$ equals 2ℓ, regardless of t.

▶ **Lemma 2.** *For every deterministic algorithm solving Consensus with at most t Byzantine nodes and for every $\ell \ge t$, there exists an execution of the algorithm on some graph $G_{t,\ell}$ with t faulty nodes that takes more than ℓ rounds.*

▶ **Theorem 3.** *For every $t \ge 1$ and $\ell \ge 1$ and any algorithm $\mathcal{A}$ solving Consensus in networks with Byzantine faults there is a network G such that $\ell \ge D(G, 2t)/2$ and an execution of algorithm $\mathcal{A}$ on this network G, with some t faulty nodes, that takes more than $(t + \ell)/2$ rounds to terminate.*

Proof. We consider two cases: either $t \ge \ell$ or $t < \ell$. If $t > \ell$, then we take a a clique of $3t + 1$ nodes as G. A graph obtained from G by removing some $2t$ nodes has diameter 1, so that $D(G, 2t) = 1 \le \ell$. Any algorithm requires at least $t + 1$ rounds to solve Consensus in this G with t faulty nodes, see [19]. Observe that $(t + \ell)/2 \le t < t + 1$. If $t \le \ell$ then we take graph $G_{t,\ell}$ as G. We rely on the property $D(G, 2t) = 2\ell$, which follows by inspection of the topology of G. By Lemma 2, some execution of algorithm $\mathcal{A}$ takes at least $\ell + 1$ rounds. Observe that $(t + \ell)/2 \leq \ell < \ell + 1$. ◀

5 Fast Authenticated Consensus

We show a separation of the Consensus problem with Byzantine nodes from its version with the same model augmented by authentication of messages. The last part of this section discusses a class of graphs in which time performance of algorithm solving Consensus can be greater, by an arbitrarily large amount, than time performance of an algorithm solving Consensus when supported by authentication of messages, subject to a given number of Byzantine faults.

We use the concept ot authentication as originally proposed by Pease et al. [29]. It assumes the existence of *authenticators* and an oracle creating such authenticators. The authenticator of data d calculated at a node p will be denoted $A_p[d]$. It is considered infeasible for a node q different from p to be able to forge $A_p[d]$. At the same time, each node should be able to check if $d = A_p[d]$ for any other node p. Algorithm Fast-Authenticated has its pseudocode in Algorithm 3. In local authorizing stage the node p tries to make exchange of input values dependable. An input value becomes reliable if passes through a well-formed path of length exactly $f+1$. In order to accomplish this, nodes authenticate received messages and send it to all neighbors in the next round. To avoid exponential message complexity, nodes keep at most one message from every non-faulty source and at most two messages from every node that decided to equivocate. Then the node p analyses the knowledge that it receives in the previous stage and determines a decision.

The algorithm works in four stages. All messages sent by non-faulty nodes are of the following format: $(p_i, a_i, (p_{i-1}, a_{i-1}, \ldots (p_1, a_1, v) \ldots))$, where the node p_1 is the original source of the transmitted value v. The nodes $p_i, \ldots, p_1$ are the ones that propagated the message in consecutive rounds. The value a_j is the authenticator of the message $(p_{j-1}, a_{j-1}, (p_{j-2}, a_{j-2}, \ldots (p_1, a_1, v) \ldots))$, for $2 \leq j \leq i$, while the value a_1 is the authenticator of value v. This format ensures, that consecutive nodes that propagated the message could not fabricate value v. If all nodes $p_1 \ldots p_i$ are different, then the message $(p_i, a_i, (p_{i-1}, a_{i-1}, \ldots (p_1, a_1, v) \ldots))$ is *well-formed*. Non-faulty nodes in a network need to exchange messages. If messages can be authenticated, then network connectivity is the main constraint. Assuming $(f + 1)$-connectivity allows for all non-faulty nodes to communicate in principle without faulty nodes interfering. Procedure Send-To-All implements such communication, its pseudocode is in Algorithm 4. We assume an authentication mechanism with an overhead of a polynomial number of bits per authenticated message. This allows achieving $\mathcal{O}(|M|n^2 \log n)$ message-size complexity, which is polynomial in n, where $|M|$ denotes an upper bound on the number of bits in an authenticated input value.

The local authorizing stage. The nodes work to disseminate the inputs dependably. The mechanism of authentication prevents forging messages, but this does not provide consistency of knowledge about inputs of faulty nodes. To handle this, nodes do not disseminate their input values directly, but instead work also to validate knowledge of their inputs. A validation is provided by passing information through a path of $f + 1$ different nodes. - Such a path contains at least one non-faulty node, which guarantees that later the node spreads the value reliably among other non-faulty nodes. We require each node on this path to confirm forwarding the message.

Algorithm 3 A pseudocode of algorithm Fast-Authenticated for a node p structured into four stages. Procedure Send-To-All has its pseudocode in Algorithm 4.

```
algorithm FAST-AUTHENTICATED

——— initialization ———
1. initialize set ReceivedMessages[1] = {(p, input, A_p[input])}
2. initialize ReceivedMessages[2], ..., ReceivedMessages[f + 1] to empty sets

——— local authorization ———
3. for i ← 1 to f do
   a. initialize mapping Inputs to empty
   b. send ReceivedMessages[i] to each neighbor
   c. foreach neighbor q do
      i. Let ReceivedMessages[i]_q be the set of messages received from the node q
      ii. foreach well-formed message (p_i, a_i, (p_{i-1}, a_{i-2}, ... (p_1, a_1, v) ...)) received from q
          and such that p ∉ {p_1, ..., p_i} do
           if a_1 = A_{p_1}[v] and a_j = A_{p_j}[(p_{j-1}, a_{j-1}, ... (p_1, a_1, v) ...)]  for all 2 ≤ j ≤ i
             encrypted-message ←
             (p, (p_i, a_i, (p_{i-1}, a_{i-1}, ... (p_1, a_1, v) ...)), A_p[(p_i, a_i, (p_{i-1}, a_{i-1}, ... (p_1, a_1, v) ...))])
             if Inputs[p_1] = ∅                \\ this is the first message from the node p_k
               ReceivedMessages[i + 1] ← encrypted-message, Inputs[p_1] ← v
             elseif Inputs[p_1] ≠ v            \\ message does not match the previous one
               ReceivedMessages[i + 1] ← encrypted-message, Inputs[p_1] ←⊥

——— global communication ———
4. {Nodes, M_1, ..., M_|Nodes|} ← SEND-TO-ALL(p, ReceivedMessage[f + 1])

——— local computation ———
5. set map Inputs to empty ; set Inputs[p] ← input
6. foreach (p_{f+1}, a_{f+1}, (p_f, a_f, ... (p_1, a_1, v) ...)) ∈ ⋓_{i∈Nodes} M_i do
     if a_1 = A_{p_1}[v] and a_i = A_{p_i}[(p_{i-1}, a_{i-1}, ... (p_1, a_1, v) ...)] : 2 ≤ i ≤ f + 1
       if Inputs[p_1] = ∅ then Inputs[p_1] ← v
       elseif Inputs[p_1] ≠ v then Inputs[p_1] ←⊥
7. decide on the majority of {Inputs[q] : Inputs[q] ≠⊥}
```

The mechanism given above takes f rounds. A node p maintains sets $\texttt{ReceivedMessages}[i]$, for $1 \leq i \leq f+1$, with the messages received after rounds $0, 1, \cdots, f$ respectively. In a round i, the node p sends the set $\texttt{ReceivedMessages}[i]$ to its neighbors. Consider a genuine message $(p_i, a_i, (p_{i-1}, a_{i-2}, \ldots (p_1, a_1, v)\ldots))$ received by a node p in round i. The message is processed as follows: if node p has marked the node p_1 as faulty, it skips the message; if in *this* round the node p has not received yet a message carrying the p_1's input value, it adds the message to the set $\texttt{ReceivedMessages}[i+1]$; if the node p has received a message carrying the p_1's input value before, but the new value does not match the old one, it still adds the message to the set $\texttt{ReceivedMessages}[i+1]$, but also marks the node p_1 as a faulty one (indicated by symbol $\perp$ in the pseudocodes); otherwise node p omitts the message. In case of adding the message to the set $\texttt{ReceivedMessages}[i+1]$, node p confirms the authenticity of communication by adding its name to the list of nodes traversed by the forwarded message

Algorithm 4 A pseudocode of procedure SEND-TO-ALL for a node p. The variable m denotes the information to be sent. The node p checks the genuine of an message by verifying the authenticator of the sender.

`procedure` SEND-TO-ALL(p,m)

---

1. initialize set `ReceivedMessages` $\leftarrow \{m\}$
2. `for` $i \leftarrow 1$ `to` D_f `do`
   - **a.** `EncryptedMessages` $\leftarrow \{(p, A_p[v], v) : v \in$ `ReceivedMessages`$\}$
   - **b.** send `EncryptedMessages` to each neighbor
   - **c.** `foreach` neighbor q `do`
     - `foreach` $(p_1, a_1, (p_2, a_2, \ldots (p_k, a_k, v) \ldots))$ received from q `do`
       - \\\\checking if the message is genuine
       - `if` $a_k = A_{p_k}[v]$ `and` $a_i = A_{p_i}[(p_{i+1}, a_{i+1}, \ldots (p_k, a_k, v) \ldots))] : 1 \leq i \leq k-1$ `and` $p_k \notin$ `Nodes` `then`
         - **A.** add $(p_1, a_1, (p_2, a_2, \ldots (p_k, a_k, v) \ldots)))$ to `ReceivedMessages`
         - **B.** add p_k to `Nodes`
         - **C.** $M_{p_k} \leftarrow v$
3. `return` (`Nodes`$, M_1, \ldots, M_{|\texttt{Nodes}|}$)

---

and authenticating it. If a node p adheres to this scheme of communication, then it stores in each round at most one message from a non-faulty node and at most two messages from a faulty node. This gives a polynomial bits complexity for this stage.

The global communication stage. Once nonfaulty nodes deliver their input values dependably to all other nodes, by executing the previous stage, the messages containing input values are scattered among the network nodes. To distribute the information among all nodes, the nodes perform an all-to-all communication procedure called SEND-TO-ALL. It takes a node's name and the information to be sent as parameters, and returns the set `Nodes` of the names of the other nodes together with the information the nodes from set `Nodes` wanted to propagate; in the case of Byzantine nodes, such a message may be missing. During the following D_f rounds, each node sends its entire knowledge to all the neighbors, using messages that can be verified for their authenticity. A node that obtains a message from a neighbor, verifies if the message is authentic. If this is the case, then the recipient learns new knowledge by adding it to its private repository. Such knowledge includes, for each node that has been learned about, the input value of each node along with the path that this piece of knowledge traversed from its originator node.

The local computation stage. Once the global communication stage is over, the information about input values is distributed among the non-faulty nodes. Each non faulty node authenticate all received messages. Based on all received authenticated messages, a node p discovers input values of another node q. This information is stored array $\texttt{Input}_p[q]$. It may occur that two different input values of the same node q appear even after authenticating messages. Each node that sent such ambiguous information is considered as faulty by the node p. Once this is detected, the corresponding value in the set $\texttt{Input}_p[q]$ is set to a special symbol $\perp$. The decision at node p is made on a majority from the values in the array $\texttt{Inputs}_p$ that are different from $\perp$.

▶ **Theorem 4.** *Algorithm* Fast-Authenticated *solves Consensus with authenticated messages in $f + D_f$ communication rounds while using messages with a polynomial number of authenticators and a polynomial number of auxiliary bits.*

The following corollary combines Theorem 3 from Section 4 with properties of algorithm Fast-Authenticated to establish a separation between Byzantine Agreement and Authenticated Byzantine Agreement based on time performance.

▶ **Corollary 5.** *For every $d \geq f \geq 4$ there exists a $(2f+1)$-connected graph G on which solving Consensus in the presence of at most f Byzantine nodes requires at least d rounds, while solving Consensus with authenticated messages in the presence of at most f Byzantine nodes is possible within $f + 2$ rounds.*

6 Early Stopping for Node Crashes

We propose an algorithm that is more efficient in terms of message size than algorithm Fast-Authenticated, assuming nodes are only prone to crashes. We give a Consensus algorithm that operates in time proportional to $f + D_f$, it uses messages of size $\mathcal{O}(m \log n)$, and relies on limited initial knowledge of nodes. Each node knows only its own name and could locally distinguish ports; in particular, knowing either $f + D_f$ or n is not assumed.

The algorithm is structured into three stages: discovery, testing and deciding. An execution starts with discovering the neighbors in one round. It is followed by the stage of testing, which is the main part of the algorithm, and completed by the stage of deciding. The algorithm is called Early-Stopping-Crashes, its pseudocode is given in Algorithm 5.

Algorithm 5 A pseudocode of algorithms Early-Stopping-Crashes for a node p . The pseudocode of procedure Send-And-Receive is in Algorithm 6. The main while loop implements testing and deciding. The variable `Grasp` represents the state of a node. The variable `Inputs` is the set of input values known to the node, and max(`Inputs`) is the maximum value in this set. The variable `Faulty` stores the set of crashed nodes known to p.

`algorithm` Early-Stopping-Crashes

---

1. `tentative` $\leftarrow$ `null`, `Inputs` $\leftarrow \{$`input`$_p\}$, `Faulty` $\leftarrow \emptyset$, `Grasp` $\leftarrow \emptyset$, $i \leftarrow 1$
2. `for` each port `do`
  send message with `name`$_p$ to each neighbor
  `if` `name`$_q$ received through this port `then` assign `name`$_q$ to this port
3. `while tentative` $=$ `null do`
  a. $j \leftarrow 1$, `checkpoint` $\leftarrow i$, `previousInputs` $\leftarrow$ `Inputs`, `previousFaulty` $\leftarrow$ `Faulty`
  b. `while` (`tentative` $=$ `null`) and ($j < 2($`checkpoint`$+1)$)
   and ($|$`Faulty` $\setminus$ `PreviousFaulty`$| <$ `checkpoint`)
   and (for every `input`$_q \in$ `Input` $\setminus$ `Faulty` : $(q, ?, ?, j) \in$ `Grasp`) do
    set (i, `Inputs`, `Faulty`, `Grasp`, `tentative`) to the output returned by
      Send-And-Receive(v, i, `Inputs`, `Faulty`, `Grasp`, `tentative`)
    $j \leftarrow j + 1$
  c. if $j = 2($`checkpoint`$+1)$ and (`tentative` $=$ `null`) then `tentative` $\leftarrow$ max(`Inputs`)
4. send `tentative` to each neighbor as the decision
5. `decide` on `tentative`

In the beginning, each node initiates its variables by instruction (1.) in the pseudocode. The variable `Inputs` stores the set of the known input values represented as pairs of a node's name and the input value of this node. Initially it contains the input value of the node. If a node knows an input value of some other node, then this is a correct value, because nodes are prone to crashes only. The variable `Faulty` stores nodes that are known to have crashed. These are nodes from whom some of their neighbors failed to receive a message in some round, and this information has been forwarded to other nodes in the network. A set `Grasp` is a digest of current knowledge. It is exchanged among neighbors until it stabilizes. More precisely, the set `Grasp` at node p consists of tuples $(q, \texttt{Inputs}, \texttt{Faulty}, i)$, each interpreted such that p has learned the set of input values `Inputs` and crashed nodes `Faulty` as known by q at the end of round i. Here q is a node's name, which could be also p, while `Inputs` and `Faulty` contain the content of these variables taken at the end of round i.

The stage of discovery is implemented by instruction (2.) of the pseudocode given in Algorithm 5. It consists of sending a node's name to all neighbors and collecting their names in return to assign neighbors' names to ports. Testing occurs in instruction (3.) and is structured as a loop. The stage of deciding begins in instruction (3c.) and continues through the last two lines of the pseudocode in Algorithm 5.

Algorithm 6 A pseudocode of procedure Send-And-Receive for a node p. It implements communicating the essential components of its state in a round to all the neighbors and updating the state by collecting similar information from the neighbors. Letter i denotes the current round number.

`procedure` Send-And-Receive $(p, i, \texttt{Inputs}, \texttt{Faulty}, \texttt{Grasp}, \texttt{decision})$

---

1. $i \leftarrow i + 1$; add a tuple $(p, \texttt{Inputs}, \texttt{Faulty}, i - 1)$ to set `Grasp`
2. send message with `Grasp` to each neighbor of p
3. for each neighbor q of p do
  a. if p received message $\texttt{Grasp}_q$ from q then
    for each tuple $(r, \texttt{Inputs}_r, \texttt{Faulty}_r, i_r)$ from $\texttt{Grasp}_q$ do
      $\texttt{Inputs} \leftarrow \texttt{Inputs} \cup \texttt{Inputs}_r$
      $\texttt{Faulty} \leftarrow \texttt{Faulty} \cup \texttt{Faulty}_r$
    add each tuple in $\texttt{Grasp}_q$ to `Grasp`
  b. if p received message with $\texttt{decision}_q$ from q then
    $\texttt{decision} \leftarrow \texttt{decision}_q$
  c. if p did not received anything from q then
    add node q to set `Faulty`
4. return (i, `Inputs`, `Faulty`, `Grasp`, `decision`)

The conditions controlling the testing loop (3.) allow for the next iteration if we have not heard from a neighbor about its decision yet and if the set `Inputs` has not just been updated and if the current estimate of the number of crashes is less than `checkpoint`. When a new testing phase begins at a round `checkpoint`, a node starts monitoring the changes of its `Grasp` in the subsequent period of $2(\texttt{checkpoint} + 1)$ rounds spent on executing the inner loop (3b.). If it finds too many changes in `Grasp` in the course of this testing period, it aborts this testing phase and starts a new one with an updated `Grasp`. Otherwise, if a node considers `Grasp` stable enough at this point, it decides on the maximum of the known input values, broadcasts its decision value to its neighbors and halts, which is what makes the stage of deciding. Similarly, if a node receives a decision value, it decides on it, broadcasts it to its neighbors and halts.

What a node tests is if its set `Grasp` is stable enough to make a decision. It does this in subsequent testing phases. Suppose that a node starts a new testing phase in round `checkpoint`. Then, in the course of $2(\texttt{checkpoint} + 1)$ following rounds, a node observes

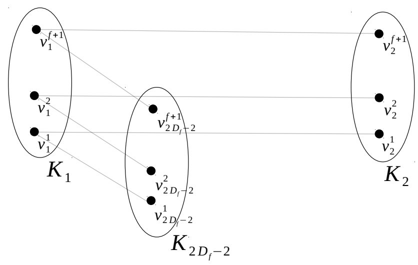

Figure 2 A visualization of graph G_{f,D_f}, for given values of f and D_f. It consists of $2(D_f-1)$ cliques of size $f+1$ each, denoted $K_1, K_2, \ldots, K_{2(D_f-1)}$, with additional edges between corresponding nodes of subsequent cliques.

but also transmits further the set `Grasp` it has received. A node must receive consistent information about the `Grasp` of each round of every non-faulty node it learned so far, up to this round – not present in the current set `Faulty` but present in some pair in `Inputs`. If any of the received sets `Grasp` provides information about a new node in the network or increases the number of known crashes learned by the node during this testing phase beyond t, this is interpreted that `Grasp` is not stable enough. This results in aborting this testing phase and a new one starts with respect to the current `Grasp`. If such an event does not occur, a node proceeds to decide at the end of round $2(\texttt{checkpoint}+1)$ in the phase. The rules to update `Grasp` at a node p at the end of round i are as follows. Initially, a tuple $(p, \texttt{Inputs}, \texttt{Faulty}, i-1)$ is created. Corresponding to the digest of the state of node p at the end of round $i-1$. The current sets `Inputs` and `Faulty` are updated by incorporating the contents of the sets `Inputs` and `Faulty` in the sets `Grasp` relayed by the received messages. Each neighbor from which a message has not been received at a round is deemed crashed and added to the set `Faulty`.

▶ **Theorem 6.** *Algorithm* Early-Stopping-Crashes *solves Consensus in $\mathcal{O}(f+D_f)$ rounds.*

7 A Lower Bound for Node Crashes

We present a lower bound that applies to node crashes. This bound matches the performance of algorithm `Early-Stopping-Crashes` given in Section 6 and algorithm `Fast-Authenticated` given in Section 5. For any values of $f \geq 3$ and $D_f \geq 4$, we define a corresponding $(f+1)$-connected graph G_{f,D_f}, which consist of $2(D_f-1)$ cliques of size $f+1$ each, in which additionally the ith node in clique j is connected to the respective ith nodes in cliques $j-1$ and $j+1$ taken modulo $2(D_f-1)$, for any $1 \leq i \leq f+1$ and $1 \leq j \leq 2(D_f-1)$; see Figure 2 for an illustration. The value of D_f for graph G_{f,D_f} is exactly D_f. Additionally, for each node p, if some other f nodes get removed, there is still a node of distance at least D_f-1 from p in the remaining graph.

▶ **Theorem 7.** *For every deterministic distributed algorithm solving Consensus there is an execution in which this algorithm terminates after least $f+D_f-2$ rounds.*

Proof. Suppose that input values are binary: 0 and 1, to simplify the exposition. For every graph G and every deterministic algorithm $\mathcal{A}$ there exists a bivalent initial configuration, see [3]. There is an execution $\mathcal{E}$ of $f-1$ rounds that ends in a bivalent configuration, see [3]. If there is an extension of $\mathcal{E}$ to some execution $\mathcal{E}'$ of f rounds, ending in a bivalent configuration, then take $\mathcal{E}'$ and continue similarly through rounds $f+1, f+2, \ldots$, until reaching an execution $\mathcal{E}''$ of $r \geq f-1$ rounds such that each of its possible extensions leads to a univalent configuration. Such $\mathcal{E}''$ exists because otherwise agreement would not be achieved for some unbounded extension of $\mathcal{E}$, contradicting the fact that $\mathcal{A}$ solves Consensus. Let β be an extension of $\mathcal{E}''$ by one round such that there is no crash in round $r+1$. Since β is univalent, it determines a decision value v_β. By the choice of $\mathcal{E}''$, there is another extension of it to some execution γ of $r+1$ rounds resulting in a different decision $1-v_\beta$. Since $\gamma \neq \beta$, there must be a crash in round $r+1$ of γ; denote a crashed node by p. There is at least one node q of distance at least $D_f - 1$ from p in a subgraph of G_{f,D_f} induced by the nodes that are non-faulty at round $r+1$. Let β_1, respectively γ_1, be an extension of β, respectively γ, to the following $D_f - 3$ rounds, with no crashes occurring. The relation $\beta_1 \overset{q}{\sim} \gamma_1$ holds. Indeed, the only difference between these two executions occurs at round $r+1$ and takes place in the non-faulty neighbors of node p in graph G_{f,D_f}. Since p and q are of distance $D_f - 1$, the neighbors of p are of distance at least $D_f - 2$ from q. The information about the only difference between these two executions could be recorded in a state of q at round $r+1+D_f-2$. Both executions take $r+1+D_f-3$ rounds, therefore $\beta_1 \overset{p}{\sim} \gamma_1$. Configurations β_1 and γ_1 are univalent and result in different decisions, therefore node q cannot decide by round $r+1+D_f-3 \geq f+D_f-3$. So at least $f+D_f-2$ rounds are needed for algorithm $\mathcal{A}$ to terminate. ◀

References

1 Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t+1$ rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.

2 Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.

3 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, Second edition, 2004.

4 Piyush Bansal, Prasant Gopal, Anuj Gupta, Kannan Srinathan, and Pranav K. Vasishta. Byzantine agreement using partial authentication. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, volume 6950 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 2011.

5 Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, 1992.

6 Piotr Berman and Juan A. Garay. Cloture votes: $n/4$-resilient distributed consensus in $t+1$ rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993.

7 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Optimal early stopping in distributed consensus (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 1992.

8 Armando Castañeda, Yoram Moses, Michel Raynal, and Matthieu Roy. Early decision and stopping in synchronous consensus: A predicate-based guided tour. In *Proceedings of 5th International Conference on Networked Systems (NETYS)*, volume 10299 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2017.

9 Bogdan S. Chlebus and Dariusz R. Kowalski. Robust gossiping with an application to consensus. *Journal of Computer System and Sciences*, 72(8):1262–1281, 2006.

10 Bogdan S. Chlebus and Dariusz R. Kowalski. Time and communication efficient consensus for crash failures. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2006.

11 Bogdan S. Chlebus and Dariusz R. Kowalski. Locally scalable randomized consensus for synchronous crash failures. In *Proceedings of the* 21*st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 290–299. ACM, 2009.

12 Bogdan S. Chlebus, Dariusz R. Kowalski, and Michał Strojnowski. Fast scalable deterministic consensus for crash failures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 111–120. ACM, 2009.

13 Bogdan S. Chlebus, Dariusz R. Kowalski, and Michał Strojnowski. Scalable quantum consensus for crash failures. In *Proceedings of the* 24*th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

14 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

15 Danny Dolev and Christoph Lenzen. Early-deciding consensus is expensive. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279. ACM, 2013.

16 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

17 Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

18 Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

19 Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

20 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

21 Zvi Galil, Alain J. Mayer, and Moti Yung. Resolving message complexity of Byzantine agreement and beyond. In *Proceedings of the* 36*th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 724–733. IEEE, 1995.

22 Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

23 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

24 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, 2003.

25 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact Byzantine consensus on undirected graphs under local broadcast model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 327–336. ACM, 2019.

26 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

27 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

28 Philippe Raipin Parvédy and Michel Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 302–310. ACM, 2004.

29 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

30 Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

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

32 Lewis Tseng. Recent results on fault-tolerant consensus in message-passing networks. In *Proceedings of the* 23*rd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 9988 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2016.

33 Lewis Tseng and Nitin H. Vaidya. Fault-tolerant consensus in directed graphs. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 451–460. ACM, 2015.
34 Lewis Tseng and Nitin H. Vaidya. A note on fault-tolerant consensus in directed networks. *SIGACT News*, 47(3):70–91, 2016.
35 Xianbing Wang, Yong Meng Teo, and Jiannong Cao. A bivalency proof of the lower bound for uniform consensus. *Information Processing Letters*, 96(5):167–174, 2005.