Gossiping to Reach Consensus

Bogdan S. Chlebus
Computer Science and Engineering
University of Colorado at Denver
and
Instytut Informatyki
Uniwersytet Warszawski
Bogdan.Chlebus@cudenver.edu

Dariusz R. Kowalski
Instytut Informatyki
Uniwersytet Warszawski
and
Département d’Informatique
Université du Québec à Hull
darek@mimuw.edu.pl

1. INTRODUCTION

Many distributed applications are based on collective updating of data that originate at the nodes of a communication network. The problem of gathering distributed information is known as the gossip problem: in the beginning each processor has its distinct piece of information, called a rumor, the goal is to make every processor know all the rumors.

We would like to perform gossiping not only quickly but also with as few node-to-node transmissions as possible. This is especially challenging when the network is prone to failures of components. We consider the case when the nodes may crash. Then it may be not always feasible to collect the input rumor of a node that has failed, since all the nodes that have learned the rumor of v also might have failed in the course of an execution of a gossiping algorithm. We consider the task of gossiping to be completed when every node has learned the following about each node v: either the rumor of v or that v has failed. Note that if a processor fails then some of the nodes may get to know its input rumor but others may only learn that it has failed.

In the consensus problem, the processors need to agree upon a common value, among their input ones, while the processors are subject to failures. This is related to the gossip problem, in that once each processor knows all the input values then the decision can be taken by applying some simple rule. For instance, the processors may want to agree upon the maximum value. However, just running a gossiping algorithm is not enough: it may happen that the maximum value is known only to some of the processors, if the original holder of the value crashed in the course of trying to disseminate it.

We consider a synchronous distributed environment, in which processors communicate among themselves by exchanging messages. The efficiency of algorithms is measured with respect to two performance metrics: time and communication. The communication is gauged as the number of point-to-point messages.

Our results. An important novelty in our approach to the gossip problem is to consider adaptive adversaries that are able to fail processors. In the prior research on the gossip problem in failure-prone environments, either link failures or processor failures controlled by oblivious adversaries have been considered, see the survey by Pelc [33].
Let $n$ be the number of processors and $t$ the number of failures that the algorithm is to tolerate.

I. We develop gossiping algorithms that are efficient with respect to both the time and the number of point-to-point messages sent. One of these algorithms runs in time $O(t^2)$ and sends $O(n \log t)$ point-to-point messages, if only $n - t = \Omega(n/\log n)$. We also give a general theorem that covers stronger adversaries. In particular, if the adversary can fail all but one processor then the time is $O((\log^2 n)$ and communication is $O(n^{1.77})$, which is subquadratic in $n$.

II. We also prove a lower bound on gossiping, to gauge how far from optimality our gossiping algorithm is when both time and communication are taken into account.

We show that, if communication is $O(n \log n)$, then the time to achieve gossiping against $t$ failures has to be at least $\Omega(n/\log n)$. This proves that our gossiping algorithm operating in time $O(\log^2 t)$ misses the time optimality by at most a logarithmic factor, in the class of algorithms that send the number of messages differing from linear by a polylogarithmic factor.

III. Our main result is a consensus algorithm that is able to handle crash failures, and which is optimal with respect to time and efficient with respect to communication. The consensus algorithm uses the gossiping algorithm as a building block.

We present an algorithm tolerating $t$ crashes and solving the consensus problem in the optimal time $O(t)$ and with $O(n \log t)$ messages, if only $n - t$ is in $\Omega(n/\log n)$. The most efficient, with respect to communication, previously known algorithm, among time-optimal ones, used $O(n + t n^3)$ messages. It has also been shown before how to solve consensus with $O(n)$ messages, but the best known algorithms achieving this require time $\Omega(n^{1+\varepsilon})$, for any fixed $\varepsilon > 0$.

Background. One may consider dissemination of information in a network similarly as a spread of rumor or of an infectious disease in a society, which has been studied in applied mathematics [7]. Demers et al. [13] introduced so-called epidemic algorithms for updating data bases, in which a node regularly chooses other nodes at random and transmits the rumors. Such randomized epidemic algorithms have been systematically studied by Karp, Schindelhauer, Shenker and Vöcking [26]. The problem of exchange of information when machines do not initially know each other was considered by Harchol-Balter, Leighton and Lewin [25], in their solution the nodes learn about each other in the course of gossiping. Gossip-style algorithm in which nodes learn about the nearest resource location was given by Kempe, Kleinberg and Demers [27]. Application of gossiping to gathering information about occurrences of failures was done by van Renesse, Minsky and Hayden [37].

The asynchronous version of gossiping, known as collect, was first abstracted by Saks, Shavit, and Woll [35]. The goal is to have all the processes learn all the register values. Computation is asynchronous, with the adversary controlling timing of the processors. Ajtai, Aspnes, Dwork and Waarts [1] showed that the problem can be solved deterministically with work $O(n^{3/2} \log n)$, by an adaptation of the algorithm of Anderson and Woll [3]. Aspnes and Hurwood [6] developed a randomized algorithm achieving work $O(n \log^2 n)$ with high probability. A lower bound of $\Omega(n \log n)$ for this problem is given in [35].

The problem of consensus was introduced by Pease, Shostak and Lamport [32]. They showed [29, 32] that the number $t$ of faulty processors needs to be smaller than $n/3$ for a solution to exist, assuming there is synchrony. The condition $3t < n$ is both sufficient and necessary. The faults required for this result to hold are of the malicious and unrestricted character, called Byzantine [30].

Fisher, Lynch and Paterson [19] showed that the problem is unsolvable in an asynchronous setting, even with only one crash failure. Randomization can make solutions possible in an asynchronous setting, as was observed by Ben-Or [9] and Rabin [34]. The issue of the use of randomization in consensus protocols has been intensively studied, in particular by Aspnes [4, 5], Bar-Joseph and Ben-Or [8] and Goldreich and Petrank [22].

For the relevance of the consensus problem to fault-tolerant broadcast and other communication problems see the paper by Hadzilacos and Toueg [24]. Fisher and Lynch [18] showed that a (synchronous) solution requires $t + 1$ rounds, when $t$ is the tolerable number of failures. Garay and Moses [21] developed an algorithm with polynomial-size messages for $n > 3t$ processors in $t + 1$ rounds. The message complexity of consensus when no failures actually happen was studied by Amdur, Weber, and Hadzilacos [2] and Hadzilacos and Halpern [23].

Dolev and Reischuk [15] studied the message complexity of the consensus problem in the case of malicious faults. They distinguished between the general case of Byzantine faults and the less demanding situation when some (cryptographic) authentication mechanism is available, which makes forging signatures of forwarded messages impossible. They showed a lower bound $\Omega(n^6)$ on the number of signatures for any algorithm using authentication, which is also a lower bound on the total number of messages for any protocol without authentication. They also showed that any algorithm with authentication needs to send $\Omega(n^{3/2})$ messages, and that achieving these many messages is possible. This shows that in the case of malicious adversaries the required number of messages can be as high as quadratic in $n$.

For crash failures, which are benign compared to Byzantine, the issue of message complexity of consensus is radically different. For a long time only the trivial linear lower bound $\Omega(n)$ on the number of messages was known, and the issue of its optimality was open. The first solutions, like the one given by Lamport and Fisher [28], had $O(n^3)$ message complexity. Bracha [10] showed how to achieve a subquadratic message complexity. Dwork, Halpern and Waarts [16] found a solution with $O(n \log n)$ messages but with exponential time. Finally, Galil, Mayer and Yung [20] developed an algorithm with $O(n)$ messages, thus showing that this amount of messages is optimal. The drawback of their solution is that it runs in overlinear time $O(n^{1+\varepsilon})$, for any $0 < \varepsilon < 1$.

A solution to consensus has the early-stopping property if the running time is $O(t + 1)$ where $t$ is the actual number of failures in an execution, this time $O(t + 1)$ is optimal. Lamport and Fisher [28] gave an early stopping algorithm with $O((t + 1)^2)$ messages. Chandra and Toueg [11] improved this to $O(t + 1)n$). Galil, Mayer and Yung [20] found an early-stopping solution with $O(n + t n^3)$ communication complexity, for any $0 < \varepsilon < 1$.

Upfal [36] showed how an almost everywhere agreement can be achieved with a linear number of faults in networks
versaries are adaptive, in that they make decisions which in the framework of a suitable adversarial model. Our ad-

Consider two measures of performance: time and communica-

tion, where communication means the total number of point-
to-point messages sent by termination.

Failures. Processors may fail by crashing. We do not assume failures to be clean: if a processor fails, while attempting to multicast a message, then some of the recipients may receive the message and some not. Except for that the communication is reliable, and messages are not lost or corrupted, while in transit. Failure patterns are considered in the framework of a suitable adversarial model. Our adversaries are adaptive, in that they make decisions which processors to fail on-line, usually subject to some quantitative restrictions. An adversary is size bounded if there is an upper bound on the number of failures it is allowed to incur; usually we denote this bound by \( t \); then the adversary is called \( t \)-bounded. An adversary is linearly-bounded if it is \( cn \)-bounded, for a constant \( c \). The unbounded adversary can fail up to \( n - 1 \) processors. An algorithm is resilient against a given adversary if it is correct if run against it. It is \( t \)-resilient if it is correct against a \( t \)-bounded adversary, and in the case of \( t = n - 1 \) such an algorithm is simply called resilient.

Graphs. Graphs \( G = (V, E) \) used in this paper are always undirected. The set \( V \) of nodes is a subset of all the processors. The edges in \( E \) are used to determine pairs of nodes that communicate directly. The \( k \)th power \( G^k \) of \( G \) is defined as follows: it has \( V \) as its set of nodes, two nodes are connected by an edge in \( G^k \) iff there is a path between them in \( G \) of length at most \( k \). We use special graphs \( G + k \) to have communication patterns that allow to save on the number of messages. The underlying graph \( G \) is a constructive expander, as defined by Lubotzky, Phillips and Sarnak [30]. For a positive integer \( m \), the network \( G(m) \) has \( m \) nodes and it simulates such an expander graph of \( \Theta(m) \) nodes (see [12, 30]). The idea of using powers of expanders to amplify their expansion was first suggested by Margulis [31], and used for communication purposes by Diks and Pelc [14].

The graph \( G(m) \) has a constant bound on the node degrees, we denote it by \( \Delta_0 \), and its diameter is \( O(\log m) \). We will use the following property of graphs \( G(m) \), we call it property \( R(m, t) \), where \( t < m \leq 2t \):

There is a function \( P \), which, for each subgraph \( R \subseteq G(m) \) of size at least \( m - t \), defines a subgraph \( P(R) \subseteq G(n) \) such that the following hold:

1. \( P(R) \subseteq R \).
2. \( |P(R)| = |R|/7 \).
3. \( \text{Diameter}(P(R)) \leq 30 \log m + 2 \).
4. If \( R_1 \subseteq R_2 \) then \( P(R_1) \subseteq P(R_2) \).

Note that \( P(R) \) is connected, even if \( R \) is not, since its diameter is finite. Upfal [36] proved that if \( m - t \geq 2m \) then graph \( G(m) \) has the property \( R(m, t) \). This was later extended by Chlebus, Gasieniec, Kowalski and Shvartsman [12], who showed that, for any parameter \( t \) such that \( t < m \leq 2t \), a graph obtained from \( G(m) \) by taking a sufficiently large power has property \( R(m, t) \). This result is used in this paper in the following form:

**Fact 1** ([12]). For every \( t < m \leq 2t \) there exists a positive integer \( t \) such that graph \( G(m)^t \) has property \( R(m, t) \). Moreover, the maximum degree \( \Delta \) of a graph \( G(m)^t \) is bounded by \( O\left( \frac{m}{m-t} \right)^{2\log_\lambda \Delta_0} \), for constants \( \Delta_0 \) and \( \lambda \), where \( \lambda > 1 \) depends on the expansion of \( G(m) \).

**Fact 2** ([36, 14, 12]). The following bound holds:

\[
(1 + 2\log_\lambda \Delta_0)^{-1} > 0.23.
\]

From now on we consider the specific value \( m = \min(n, 2t) \). We define graph \( G(n, t) \) as follows. All the processors serve as vertices. Edges are of two kinds. First we take the edges of a graph isomorphic with \( G(m)^t \), whose nodes are the processors with ids in the range \( 1 \) through \( m \), and \( t \) is such that \( G(m)^t \) has property \( R(m, t) \), as guaranteed by Fact 1. Secondly, if \( m < n \) then we add all connections of the form \((p, q)\), where \( p \leq m \) and \( q \equiv p \mod m \). Graph \( G(n, t) \) is called the communication graph. This graph may evolve in the course of an execution of a gossiping algorithm: if a processor fails then it is no longer a node.

3. **GOSSIPING ALGORITHM**

In this section we describe a gossiping algorithm called **Algorithm Gossip**. The following parameters are hardwired into the algorithm: the number \( n \) of processors, the maximum number \( t \) of failures, the graph \( G(n, t) \), an integer termination threshold \( T \), and a family \( P = \{\pi_p\}_{1 \leq p \leq m} \) of local permutations. Only the parameters \( T \) and \( P \) have not been presented yet, their role will be described in the specification of algorithm, and their properties related to the efficiency of the algorithm will be discussed in Section 4.

We say that a processor \( p \) heard about processor \( q \) if either \( p \) knows the original input rumor of \( q \) or it knows that \( q \) has failed.

Categories of processors. We partition the set of all the processors into \( m \) groups, where \( m = \min(n, 2t) \). Group \( D(i) \), for \( 1 \leq i \leq m \), is defined to consist of processors \( q \) such that \( q \equiv i \mod m \). Groups are of balanced size, each has either \( m/n \) or \( n/m + 1 \) processors. The processor with the smallest id in \( D(i) \) is called its leader. The subgraph of \( G(n, t) \) induced by the leaders is isomorphic to \( G(m)^t \).

Each leader starts an execution as a collector. Collectors actively seek information about the rumors of the other processors, by sending direct inquiries to them. After a collector has heard about all the processors then it becomes a disseminator. These processors disseminate their knowledge by sending out their local state. The processors that are not leaders are called ordinary. The ordinary processors are most passive among all the categories of processors. They do not seek information on their own, relying instead on the initiative of the leaders. An ordinary processor just waits till messages come to it. If such a message is an inquiry from a collector then the ordinary processor replies to
Receiving step: Receive messages

Computation step:

a. Update the local arrays
b. If p is a collector that has heard about all the processors then become a disseminator

sending step:

a. Send a graph message to each neighbor q of p in the communication graph that is active according to p
b. Send a reply message to each collector q from which a range message was received in the receiving step of this phase

c. If p is a collector and q is in the first chunk with a processor about which p has not heard yet then send a range message to q
d. If p is a disseminator and q is in the first chunk with a processor that needs to be notified by p then send a range message to q

Figure 1: Regular phase of Algorithm_Gossip. Code for processor p. The ordering of chunks that processor p uses if it is a leader, is determined by the permutation πp.

it by sending its local state. If the message is a summary of the knowledge of a disseminator, then it makes the ordinary processor sufficiently informed.

Local state. Initially each processor p knows only its id p and its input rumorp. To store incoming information, processor p maintains the following arrays: Rumorsp, Activep, and Pendingp, each of size n. All these arrays are initialized to zeros. If an entry in the array is not equal to the initial zero then it is some information with a special meaning that has been stored. Given an array Xp of processor p, we denote by Xp[q] the contents of its qth location.

The array Rumor is used to store all the rumors that a processor knows. In the very beginning p sets Rumorsp[p] to its own input rumorp. After processor p learns some rumorp it immediately sets Rumorsp[p] to this value.

The array Active is used to store the set of all the crashed nodes that a processor knows about. Once processor p learns that processor q has failed it immediately sets Activep[q] to 0. Notice that processor p has heard about processor q if either Rumorsp[q] or Activep[q] is not equal to zero.

The purpose of using the array Pending is to facilitate the job of dissemination. If processor p learns that some other processor q is fully informed, that is, it is either a disseminator itself or has been already notified by a disseminator, then it marks this information in Pendingp[q]. The array Pendingp is useful even if p is an ordinary processor, because it may relay its local knowledge to a collector by replying to an inquiry. If p is a leader then array Pendingp may be also used to send dissemination messages in a systematic way: it is sufficient to scan Pendingp to find those processors that possibly still have not heard about some processor.

We use the following terminology based on the current contents of the arrays Active and Pending. Processor q is said to be active according to p if p has not yet received any information implying that q crashed, which is the same as having a zero in Activep[q]. Processor q is said to need to be notified by p if it is active according to p and Pendingp[q] is equal to zero.

Categories of messages. A processor p may send a message to its neighbor in the graph G(n, t), provided that it is still active according to p, such a message is called a graph one. These messages are not sufficient since the communication graph may become disconnected due to failures. Hence other messages are also sent, to cover all the processors in a systematic way, they are called range ones. To define how this part of communication works, the processors are partitioned into w chunks. The chunk number w is equal to ⌈n/∆⌉ if m = n, otherwise w = m. In any case w ≤ m. Chunk C(i), for 1 ≤ i ≤ w, is defined to consist of processors q such that q ≡ i mod w. Chunks are of balanced size, each of ∆ or ∆ − 1 elements for m = n, and either ⌈n/m⌉ or ⌈n/m⌉ + 1 processors otherwise. Either the equality C(i) = D(i) holds for m < n or sets D(i) are singletons, then C(i) are of size ∆ or ∆ − 1. The local permutations are of the integers in the range [1..w]. Each leader p considers the chunks as ordered according to its local permutation πp, that is, it considers them in the order πp(1), πp(2), . . . , πp(w).

A collector p sends a range message to the first chunk with a processor about which p has not heard yet, and to all such processors in the chunk simultaneously. Each recipient of such a message sends back a reply message. Disseminators send range messages also to subsets of chunks. The target chunk selected by disseminator p is the first one containing a processor that still needs to be notified by p, and the message goes to all the processors in the chunk that share this property. Disseminator-range messages need not be replied to: the sender knows already the rumors of all the processors that are active according to it, and the purpose of the message is to disseminate this knowledge.

Messages of the kinds discussed so far are sent during T rounds of communication, where T is called the termination threshold. After that the processors that have not heard about someone yet send direct inquiries to these processors. Such messages are called last-resort ones, and are replied to by the non-faulty recipients in the next step.

Updating local state. A message sent by a processor carries its current local knowledge. More precisely, a message

Figure 2: Ending phase of Algorithm_Gossip. Code for processor p.

received from processor $p$ brings the following: the id $p$, the arrays $\text{Rumors}_p$, $\text{Active}_p$, and $\text{Pending}_p$, and a label to notify the recipient about the character of the message. The label is selected from the following: \text{graph}, \text{message}, \text{range}, \text{from}, \text{collector}, \text{range}, \text{from}, \text{disseminator}, \text{this}, \text{was}, \text{reply}, \text{last}, \text{resort};$ their meaning is self-explanatory.

A processor $p$ scans a newly received message from $q$ to learn about rumors, failures, and the current status of other processors. It copies each rumor from the received copy of $\text{Rumors}_q$ into $\text{Rumors}_p$, unless it is already there. It sets $\text{Active}_p[r]$ to failed if this value is at $\text{Active}_q[r]$. It sets $\text{Pending}_p[q]$ to done if this value is at $\text{Pending}_q[r]$. It sets $\text{Pending}_p[q]$ to done if $q$ is a disseminator and the received message is a range one.

If $p$ is itself a disseminator then it sets $\text{Pending}_p[q]$ to done immediately after sending a range message to $q$. If a processor $p$ expects a message to come from processor $q$, for instance a graph one from a neighbor in the communication graph, or a reply one, and the message does not arrive, then $p$ knows that processor $q$ has failed, and it immediately sets $\text{Active}_p[q]$ to failed.

The algorithm. Algorithm_Gossip is a loop in which phases are iterated. Each phase consists of three steps: receiving messages, local computation, and multicasting messages. Phases are of two kinds: either regular, its pseudocode is in Figure 1, or ending, its pseudocode is in Figure 2. For $t < \log n$ an additional before-ending module is executed between the regular an ending phases. This module consists of four phases, see Figure 3 for its pseudocode. The algorithm starts with all the processors initializing their local objects.

1. Initialize the local objects
2. Perform the regular phase $T$ times
3. If $t < \log n$ then perform the before-ending module
4. Perform the ending phase three times

Figure 4: Algorithm_Gossip.

A processor $p$ initializes the list $\text{Rumors}_p$ to zeroes at all the locations, except for the $p$th one, which is set equal to $\text{rumor}_p$. If processor $p$ is a leader then it starts as a collector. A pseudocode of the algorithm is in Figure 4.

Lemma 1. For any termination threshold $T > 0$ and any family of local permutations $\Pi$, Algorithm_Gossip is a resilient gossiping algorithm.

4. ANALYSIS AND VARIANTS OF THE GOSSIPING ALGORITHM

We start with a randomized version of Algorithm_Gossip. It does not have a family of permutations as a part of its code, instead each leader $p$ generates a random permutation of the integers in $[1..n]$ at the start of the algorithm. This permutation is denoted also by $\pi_p$, since its role is exactly the same as if it were a part of the code. The generation is done uniformly, with respect to all the permutations, and independently, over all the processors. The randomized version of Algorithm_Gossip is called Randomized_Gossip.

Property $R(m, t)$ motivates the following terminology. A subgraph $K \subseteq G(m)^t$ is compact if the size of $K$ is at least $(m-t)/T$ and the diameter of $K$ is at most $30 \log m + 2$. If $K$ is a subgraph of $G(m)^t$ and $p \in K$ then by the neighborhood of node $v$ in $K$ we mean the subgraph containing each node in $K$ whose distance from $p$ in $K$ is at most $30 \log m + 2$. A node $p \in K$ is said to be compact in $K$ if its neighborhood in $K$ has size at least $(m-t)/T$. Notice that each node in a compact subgraph is itself compact.

We partition the computation into consecutive stages: one stage comprises $30 \log m + 2$ consecutive phases. One stage is sufficient for all the processors in a compact subgraph to communicate. When we refer to a processor during a stage then we mean a processor that does not fail until the end of this stage.

We consider two conceptual lists $R_p$ and $S_p$, for each leader $p$. The list $R_p$ is obtained by ordering all the chunks of processors according to $\pi_p$, and then removing each chunk with the property that $p$ has already heard about all the elements in this chunk. The list $S_p$ is obtained by ordering all the chunks of processors according to $\pi_p$, and then removing each chunk without elements that processor $p$ still needs to notify. Both the lists $R_p$ and $S_p$ are dynamic, and shrink in the course of an execution. If $p$ is a leader then it becomes a disseminator exactly when the list $R_p$ disappears altogether. If $p$ is a collector then it sends range messages to all the processors in the first chunk of $R_p$ that it still has not heard about. If $p$ is a disseminator then it sends range messages to all the processors in the first chunk of $S_p$ that it still needs to notify.
We use the following notations when referring to the behaviour of the two algorithms Algorithm\_Gossip and Randomized\_Gossip. The notation \( r_i(p) \) means the size of list \( R_p \) at the end of the \( i \)th stage. Let \( K \) be a subgraph of \( G(m)^t \). The notation \( r_i(K) \), or \( r_i \) if graph \( K \) is indicated by the context, means the size of the set of all the chunks which are on any of the lists \( R_p \) at the end of stage \( i \), for \( p \in K \). More precisely, \( r_i(K) \) denotes the number \( \left\{ \sum_{p \in K} R_p \right\} \), where \( R_p \) are as at the end of stage \( i \). Similarly, we define \( s_i(p) \) and \( s_i \) by considering the list \( S_p \) instead of \( R_p \).

**Lemma 2.** Let \( K \) be a fixed subgraph of \( G(m)^t \) of diameter at most \( 30 \log m + 2 \) at the end of stage \( i+1 \) of Randomized\_Gossip. If \( r_{i-1} > \log w \) then the inequality \( r_{i+1} \leq \max\{r_{i-1} - 3|K|\log m, (2/3)r_{i-1}\} \) holds and otherwise \( r_{i+1} = r_i = 0 \), with the probability at least \( 1 - \exp(-\gamma|K|\log m) \), for some positive constant \( \gamma \). The same estimates hold for \( s_i \).

**Lemma 3.** Assume \( n - t > \log n \). There is a constant \( \alpha > 0 \) such that if \( T \geq \alpha \log w \log m \) then with the probability at least \( 1 - \exp(-\Omega((m-t)\log m)) \) by phase \( \alpha \log w \log m \) of Algorithm Randomized\_Gossip the gossiping has been completed. Moreover, the communication by this phase is
\[
O\left(\frac{n}{n-t}\right)^{2\log \lambda \Delta_0} n \log w \log m).
\]

**Theorem 1.** If \( n - t = \Omega(n/\log^n n) \), for a positive constant \( \alpha \), then Algorithm\_Gossip can be implemented to run in time \( O((\log^2 t)^t) \) and send \( O(n \log^{4.35} t) \) messages.

**Proof.** We use the probabilistic method, relying on Lemma 3. Let us set \( T = \alpha \log w \log m \) in Algorithm\_Gossip, where the constant \( \alpha \) is as in Lemma 3. Let \( X \) denote a random variable equal to the amount of communication performed during an execution of algorithm Randomized\_Gossip.

Suppose first that \( t > \log n \), then also \( m > \log n \). The event
\[
X = O\left(\frac{n}{n-t}\right)^{2\log \lambda \Delta_0} n \log w \log m)
\]
holds with the probability at least \( 1 - \exp(-\beta(m-t)\log m) \), for some positive constant \( \beta \), by Lemma 3, as no communication is performed during the ending phases. The event \( X = O(n^2 \cdot \log w \log m) \) holds always, since \( O(n^2) \) point-to-point messages are sent at every step of Algorithm\_Gossip. Hence we can bound the expected value of \( X \) by:
\[
O\left(\frac{n}{n-t}\right)^{2\log \lambda \Delta_0} n \log w \log m)
\]
where we used the estimate \( (m-t)\log m = \omega(\log n) \). It follows that there exists a family of permutations \( \Pi \) such that if it is used in the code of Algorithm\_Gossip then the amount of its communication is at most \( E[X] \). Now we use the estimate of the number \( (1 + 2\log \lambda \Delta_0)^{-1} \) given in Fact 2, and the estimates \( m, w = O(t) \) and \( \frac{n}{n-t} = O(\log^6 n) = O(\log^4 t) \). We obtain the final bounds: \( T = \alpha \log w \log m = O(\log^2 t) \) and \( O\left(\frac{n}{n-t}\right)^{2\log \lambda \Delta_0} n \log^2 t) \) is an estimate on the amount of communication.

The analysis of case \( t < \log n \) is similar, it relies on the before-ending module instead of the ending phases.

**Figure 5:** Algorithm Balanced\_Gossip. The constant \( \alpha \) is as in Lemma 3. If \( t \geq t_0 \) then \( G(n, t_0) \) is the communication graph, where \( t_0 \) is given by equation (1).

The communication of Algorithm\_Gossip is large for the number of failures \( t \) close to \( n \). This is because the communication graph \( G(n, t) \) has a large degree for large \( t \). To improve upon the communication for large \( t \) we could introduce the following two modifications to the algorithm:

I: Use some fixed graph \( G(n, t_0) \) for communication, for \( t \geq t_0 \), where \( t_0 \) is some threshold value.

II: Stop performing regular phases at some earlier suitable moment, and finish with some direct-communication pattern instead.

The idea is to balance the communication during the two parts of the algorithm. The threshold \( t_0 \) can be estimated as follows. During the ending phases the communication is \( O(n(n-t_0)) \), also \( m = n \) in this case. Select \( t_0 \) so that the contributions to the amount of communication from the regular and ending phases are the same:
\[
n(n-t_0) = \left(\frac{n}{n-t_0}\right)^{2\log \lambda \Delta_0} n \log^2 n.
\]

Solving for \( t_0 \) gives:
\[
t_0 = n - n^\frac{1}{\log^2(\log^{-2}\log^2\lambda \Delta_0} n^\frac{1}{\log^2(\log^{-2}\log^2\lambda \Delta_0} n.
\]

The resulting new algorithm is called Balanced\_Gossip. It has regular phases executed \( T = \alpha \log^2 t \) times. If \( t < t_0 \) then graph \( G(n, t) \) is used as the communication graph, otherwise graph \( G(n, t_0) \) serves this purpose. A pseudocode of Algorithm Balanced\_Gossip is given in Figure 5. We also consider a randomized version of Balanced\_Gossip in which the leaders select their permutations in \( \Pi \) randomly at the start of the algorithm, rather than having them as part of their code. This algorithm is called Randomized\_Balanced\_Gossip, its useful performance characteristic is given in Lemma 4.

**Lemma 4.** The number of point-to-point messages sent by algorithm Randomized\_Balanced\_Gossip is
\[
O\left(\frac{n}{n-t}\right)^{2\log \lambda \Delta_0} n \log^2 t)
\]
provided the inequality \( t \leq t_0 \) holds, and otherwise it is
\[
O\left(n^2 \cdot \log^4 \lambda \Delta_0 \log^6 n \right)
\]
with the probability at least \( 1 - \exp(-\Omega((m-t)\log m)) \).
Proof. First consider the case \( t \leq t_0 \). The communication during the regular phases is
\[
O\left( n^2 \log n / \log \left( \frac{n}{t} \right) \right),
\]
by Lemma 3 and the estimates \( m, w = O(t) \). It also follows that with the probability \( 1 - \exp(-\Omega((m-t) \log m)) \) all the processors have heard about each other, hence during the ending phases no communication is needed.

Next consider the case \( t > t_0 \). If during \( T \) steps there have been at most \( t_0 \) failures then no communication is needed during the ending phases, with the probability at least \( 1 - \exp(-\Omega((m-t) \log m)) \). The amount of communication during the regular phases can be estimated as in the first case for \( t = t_0 \), and is equal to \( O\left( n^2 \log \left( \frac{n}{t} \right) \right) \).

If the number of failures is greater than \( t_0 \) then the communication during the first \( T \) phases is the same, and during the three ending phases the communication is
\[
O((n-t_0)n) = O\left(n^2 \log \left( \frac{n}{t} \right) \right),
\]
by the definition of \( t_0 \).

Using the probabilistic method, based on Lemma 4, we obtain the following result:

**Theorem 2.** Algorithm Balanced_Gossip can be implemented in such a way that:
\[ n - t = \Omega(n^{0.77} \log^{0.46} n) \)
then its time is \( O(\log^2 t) \) and communication is \( O(\left( n^{1.35} \log^2 t \right) \)
otherwise its time is \( O(\log^2 n) \) and communication is \( O(n^{1.77}) \).

5. LOWER BOUND FOR GOSSIPING

Suppose that a \( t \)-resilient gossiping algorithm \( A \) has at most \( \beta n \log \alpha \) \( n \) messages sent in each execution. Our goal in this section is to show that algorithm \( A \) must perform \( O(\log n / (\log(\log n) - \log t)) \) steps.

Let us fix \( n \) and \( t < n \). For each sequence of input rumors \( I = (\text{rumor}_1, \ldots, \text{rumor}_n) \) we construct a strategy \( F_I \) of a \( t \)-bounded adversary such that after \( b \log n / (\log(\log n) - \log t) \) steps of algorithm \( A \) there is a processor \( p_I \) which has not heard about some other processor \( q_I \).

We will determine a strategy of an adversary to keep at least \( \max(n-t, n/2) \) processors nonfaulty. Let \( n_I \) denote the number of nonfaulty processors after the step \( i \), and let \( x > 0 \) be a parameter. By the bound on the message complexity, we have that in each step of algorithm \( A \) at most \( \beta n \log \alpha \) \( n \) point-to-point messages are sent. Observe that during the \((i+1)\)th step of communication at most \( n_I / (2x) \) processors receive messages from more than \( x \cdot \beta n \log \alpha \) processors. The adversary’s strategy is to fail these processors in step \( i \). The numbers \( n_I \) satisfy the inequality \( n_I \geq n_{i-1} - 1 \), and each nonfaulty processor has heard about rumors of at most \( (1 + x \cdot \beta n \log \alpha)^i \) processors by step \( i \).

We are constrained by the following inequalities:
\[
\begin{align*}
  n \cdot (1 - 1 / (2x))^i & \geq \max(n-t, n/2) ; \quad (2) \\
  (1 + x \cdot \beta n \log \alpha)^i & \leq n/2 . \quad (3)
\end{align*}
\]

There are two cases. If \( n-t \leq n/2 \) then the following estimates hold: \( 1/x = O(1) \) and \( i(\log x + \log \log n) = O(\log n) \), which imply \( i = O(\log n / \log \log n) \). If \( n-t > n/2 \) then \( 1 - i/(2x) \geq 1 - t/n \) and \( i(\log x + \log \log n) = O(\log n) \), hence \( i = O(\log(n \log n) / \log \log n) \).

One can verify directly by simple algebra that there is a constant \( M \) such that \( i = b \log \left( n / (\log(n \log n) - \log t) \right) \) and \( x = \log(n \log n) - \log t \) satisfy inequalities (2) and (3). We conclude that after \( b \log n / (\log(n \log n) - \log t) \) steps there are at least \( n-t \) non-faulty processors and each of them has not heard about some other non-faulty processor.

Let the notation \( p_I \) and \( q_I \), for the input rumors \( I \), denote processors such that \( p_I \) has not heard about \( q_I \), and both of them are nonfaulty. We still need to show that processor \( p_I \) could not decide correctly on the rumor of processor \( q_I \) in spite of the lack of communication. We assume that rumors are from an infinite domain, for instance they can be arbitrary positive integers. Denote by \( M_I \) the message pattern defined to be the sequence of sets of links used in consecutive steps of an execution of algorithm \( A \). The numbers of different message patterns \( M_I \) and of failure patterns \( F_I \) are finite, as well as the number of pairs of processors \( p_I \) and \( q_I \), but the number of possible initial configurations of rumors \( I \) is infinite. It follows, by the pigeonhole principle, that there are two different inputs \( I_1 \) and \( I_2 \) such that the rumor of processor \( p_{I_1} \) is distinct from the rumor of processor \( p_{I_2} \), while the following are equal: \( q_{I_1} = q_{I_2} \), \( F_{I_1} = F_{I_2} \) and \( M_{I_1} = M_{I_2} \). Hence, if processor \( p_{I_1} \) terminated by step \( b \log n / (\log(n \log n) - \log t) \), then it would have to decide what the rumor of processor \( q_{I_1} \) is, and that decision would have to be the same that \( p_{I_2} \), made in exactly the same time for the input \( I_2 \), which is impossible.

This completes the proof of the following lower bound:

**Theorem 3.** If rumors are from an infinite domain and a \( t \)-resilient algorithm generates \( O(n \log^2 n) \) messages, for some positive constant \( a \), then it requires \( \Omega(\log n / (\log(n \log n) - \log t)) \) steps to complete gossiping.

6. CONSENSUS ALGORITHMS

Each processor \( p \) has a positive integer, from a range of a constant size, as its input \( \text{value}_p \). We want the processors to agree on one such value, it will be the maximum one known by the operational processors.

Our consensus algorithms are embedded in the gossiping algorithm Balanced_Gossip. The notations and operations of this algorithm are used throughout this section with the same meaning. During the first run, the value of rumor of processor \( p \) is initialized to its input. The notation \( \text{rumor}_p \) means now a local variable of processor \( p \), this is to maintain the consistency of notifications, it is initialized to \( \text{value}_p \).

If \( t \leq n^{-\epsilon} \) then algorithm GMTI generates \( O(\log n / (\log(n \log n) - \log t)) \) messages, hence we may concentrate on the case \( t > n^{-\epsilon} \).

Algorithm Consensus uses phases of a new kind, they are called spreading. A pseudo code for spreading phase is in Figure 6. Messages sent out during such phases are also called spreading, each such a message carries the current value of \( \text{rumor}_p \) of its disseminator \( p \).

A pseudo code of Algorithm Consensus is in Figure 7. Array Rums is initialized in the beginning of each execution of the gossiping algorithm, but using the current value of the variable \( \text{rumor}_p \). The value of variable \( \text{rumor}_p \) is kept equal to the current maximum value either in the array Rums or heard from other processors, unless it is set explicitly.
Receiving Step: Receive spreading messages

Computation Step: If a value of some variable rumor was received in the previous step, such that it is larger than the current value of rumor_p then set rumor_p to the largest among such values

Sending Step: If the variable rumor changed its value in the previous step then send a spreading message to each neighbor, in the communication graph G(n, t), that is active according to p

Figure 6: Spreading phase of Algorithm Consensus. Code for processor p.

to the input value_p. The number \( \lceil 7m/(m-t) \rceil \) of iterations is at least as large, by property \( R(m, t) \), as the number of connected components in the communication graph that contain any compact neighborhood at any moment of an execution. We say that processor p believes to be in a compact neighborhood if it is compact in the subgraph induced by the processors that are still active according to p. Processor p can check this property using its array active_p and the topology of graph G(n, t).

**Lemma 5.** Let G be a connected subgraph of G(m)^t, after parts c and d of Step 3 at Algorithm Consensus have been performed. Then all the processors p in G store the same value at rumor_p.

**Lemma 6.** Algorithm Consensus is a t-resilient algorithm solving consensus.

**Proof.** The value agreed upon is among the initial input ones because in every step each rumor_p stores some among the initial input values.

It remains to show that all the processors agree upon one common value. For each processor p and any moment of computation, the value at rumor_p is at least as large as value_p. Let k_i be the total number of processors in the connected components in G(m)^t that contain a compact processor at the end of part b of the i-th execution of Step 3, where \( i = 1, \ldots, \lceil 7m/(m-t) \rceil + 1 \). We may assume that \( k_0 = m \). The inequality

\[
k_i \geq \frac{(m-t)/7}{m}
\]

holds by property \( R(m, t) \). It follows, by induction, that after the i-th iteration of Step 3 the following invariants hold:

**Invariant 1.** All the processors p, that believe to be compact and are in the same connected component, have the same value of rumor_p.

**Invariant 2.** This common value is equal to at least the maximum of value_p taken over all the processors that have not failed.

**Invariant 3.** At the end of part b of iteration (i+1), either:

(A) all the processors p that believe to be compact have the same value rumor_p, or

(B) the inequality \( k_i \leq k_{i-1} - (m-t)/7 \) holds.

---

Figure 7: Algorithm Consensus. Code for processor p.

By gossiping in Step 4, all the processors receive a common value of processors believing to be compact, and this value, by Invariant 2, is maximal among all the rumors. By property \( R(m, t) \), there is at least one processor believing to be compact at the end of the computation, and all the processors decide the same maximum value in (5).

Let \( T(n, t) \) denote the time and \( C(n, t) \) the amount of communication of Balanced_Gossip. In the next lemma we bound the time and communication complexity of Algorithm Consensus.

**Lemma 7.** Algorithm Consensus terminates in time \( O(t + T(n, t) \cdot n/m) \) and sends \( O((n \cdot \Delta \cdot \log n + C(n, t)) \cdot n/m) \) messages.

**Proof.** If \( t \leq n^{1-\epsilon} \) then the time is \( O(t) \) and the number of point-to-point messages is \( O(n) \) by the performance of algorithm GMY.

Suppose \( t > n^{1-\epsilon} \). Observe that \( \frac{m}{n^t} = O(\frac{1}{n^t}) \), for all n and t. Hence there are \( O(\frac{1}{n^t}) \) executions of Step 3 in the algorithm, see Figure 7. We estimate the time of one execution of Step 3: part a takes \( T(n, t) \) steps, part d takes \( O(t) \) steps, the propagating part e takes \( O(\log m) \) steps, the remaining parts take \( O(1) \) steps. Hence the total duration of Step 3 is

\[
O \left( \left( t + \log m + T(n, t) \cdot \frac{n}{n-t} \right) \right) = O \left( \left( t + T(n, t) \right) \cdot \frac{n}{n-t} \right),
\]

since \( \log m = O(t) \) for \( t > n^{1-\epsilon} \). The gossiping procedure in Step 4 takes \( O(T(n, t)) \), which is dominated by the contribution from Step 3.

The number of messages sent during one execution of a spreading phase in parts c or d of Step 3 is at most \( O(n \Delta) \). Each processor changes its rumor at most some
constant number of times, equal to the number of initial values. Hence, for each link in the communication graph, only a constant number of point-to-point messages are sent during one execution of the spreading phases in c or d. The total number of links in the communication graph is \( O(n) \). The amount of communication during e is \( O(n \Delta \log m) \), and during a is \( O(C(n, t)) \). We obtain \( O(n \Delta \log m + C(n, t)) \) as a bound on the number of point-to-point messages sent during one execution of Step 3, hence the total is \( O(n \Delta \log n + C(n, t)) \). The amount of communication during Step 4 is \( O(C(n, t)) \) which is dominated by the contribution from Step 3.

Lemma 7 and Theorem 2 imply the following:

**Theorem 4.** Algorithm \( \text{Consensus} \) runs in time \( O(n) \) and sends \( O(n \log^2 n) \) messages against any linearly-bounded adversary.

Next we consider another consensus algorithm, called Balanced\( \_ \)Consensus, which is a modification of Algorithm \( \text{Consensus} \). To describe it, we change the meaning of a compact neighborhood as follows. A node \( v \) believes to be in a \( \frac{r}{m} \) compact neighborhood if its neighborhood in the subgraph induced by the processors that are still active according to \( p \) has size at least \( \frac{r}{m} \). Instead of executing \( \lceil 7m/(m - t) \rceil \) times Step 3 in Algorithm \( \text{Consensus} \), we iterate 14 times only its parts a-f in the stages \( i = 1, \ldots, 1 + \log [m/(m - t)] \). During stage \( i \), we use the notion of a \( \frac{r}{2} \)-compact neighborhood instead of a compact neighborhood, and iterate the spreading phase \( \frac{r}{2^i} \) times only, in part d of Step 3.

**Lemma 8.** Algorithm \( \text{Balanced} \_ \text{Consensus} \) is a t-resilient consensus algorithm that runs in time \( O(t + T(n, t) \log t) \) and with \( O(C(n, t) \log t + n \Delta \log^2 t) \) messages.

Combining Lemma 8 with Theorem 1 yields the following result:

**Theorem 5.** If \( n - t = \Omega(n/\log^a n) \), where \( a \) is a positive constant, then there is a t-resilient algorithm solving the consensus problem in time \( O(t) \) and with \( O(n \log^3 4.35a n) \) messages.

7. DISCUSSION

We considered the problem of gossiping and consensus in a synchronous setting with crash failures. We developed algorithms that are efficient with respect to both time and communication. The performance of our solutions depends on the power of adversary, defined in terms of the number of crashes it may incur. Our solutions have performance characteristics close to the optimum ones within polylogarithmic factors, only the number of processors that remain operational is \( \Omega(n/\text{polylog } n) \). The problem of developing solutions of a similar performance, or showing limitations to such solutions, for the unbounded adversary, that may fail all but one processor, remains an open problem.

8. REFERENCES


