Confidential Gossip[*]

Chryssis Georgiou[†] Seth Gilbert[‡] Dariusz R. Kowalski[§]

Abstract

Epidemic gossip has proven a reliable and efficient technique for sharing information in a distributed network. Much of this reliability and efficiency derives from processes collaborating, sharing the work of distributing information. As a result of this collaboration, processes may receive information that was not originally intended for them. For example, some process may act as an intermediary, aggregating and forwarding messages from some set of sources to some set of destinations.

But what if rumors are *confidential*? In that case, only processes that were originally intended to receive the rumor should be allowed to learn the rumor. This blatantly contradicts the basic premise of epidemic gossip, which assumes that processes can collaborate. In fact, if only processes in a rumor's "destination set" participate in gossiping that rumor, we show that high message complexity is unavoidable. A natural approach is to rely on cryptography, for example, assuming that each process has a well-known public-key that can be used to encrypt the rumor. In a dynamic system, with changing sets of destinations, such a process seems potentially expensive.

In this paper, we propose a scheme in which each rumor is broken into multiple fragments using a very simple coding scheme; any given fragment provides no information about the rumor, while together, the fragments can be reassembled into the original rumor. The processes collaborate in disseminating the rumor fragments in such a way that no process outside of a rumor's destination set ever receives all the fragments of a rumor, while every process in the destination set eventually learns all the fragments. Notably, our solution operates in an environment where rumors are dynamically and continuously injected into the system and processes are subject to crashes and restarts. In addition, the presented scheme can tolerate a moderate amount of collusions among *curious* processes without a substantial increase in cost; curious processes are non-malicious processes that are not in a rumor's destination set, and still want to learn the rumor (that is, collect all fragments of the rumor).

Keywords: Confidentiality, Collusion, Randomized gossip, Fault-tolerance, Dynamic rumor injection, Message complexity.

1 Introduction

Collaboration is as the heart of distributed computing: when a network of devices cooperates to solve a problem, the resulting computation is often more robust and more efficient than if each device had worked independently. A classic example of the benefits of collaboration can be found in the paradigm of epidemic gossip. Consider, for example, a set of n devices that want to share information. If each device communicates independently with the other devices in the network, then the message complexity for the protocol may be $O(n^2)$. By contrast, if the devices collaborate to share the information, each communicating with a small number of random devices in each round, then the message complexity for the protocol can be reduced to $O(n \log n)$ (when message size is unbounded). At the same time, the resulting protocol is quite robust, tolerating a constant fraction of the devices crashing.

Yet there are some drawbacks to collaboration. One significant cost is *privacy*: by collaborating with other devices to solve a problem, it is often the case that private information is divulged. Consider again n devices that want to share information—however the information is potentially *confidential* and should only be shared among specified groups of recipients. For example, a user may want to share an engineering blueprint with her colleagues, but not with her competitors. Or a psychiatrist may want to send an e-mail to a group of patients, but not to everyone. Unfortunately, standard distributed protocols for efficiently sharing information do not satisfy these requirements. For example, if the

[*]A preliminary version of this paper has appeared in the proceedings of ICDCS 2011. This is a pre-print of an article published in the *Distributed Computing* journal. The final authenticated version is available online at `https://link.springer.com/journal/446`. This research was supported by funds for the promotion of research at the University of Cyprus (CG-RA2019), by the Singapore MOE grant MOE2018-T2-1-160, and by the Polish National Science Center (NCN) under Grant No. UMO-2017/25/B/ST6/02553.

[†]Dept. of Computer Science, University of Cyprus, 75 Kallipoleos Str, CY-1678 Nicosia, Cyprus; `chryssis@cs.ucy.ac.cy`.

[‡]Dept. of Computer Science, National University of Singapore, Singapore; `seth.gilbert@comp.nus.edu.sg`.

[§]School of Computer and Cyber Sciences, Augusta University, Georgia, USA, and SWPS University, Poland; `dkowalski@augusta.edu`.

users rely on epidemic gossip to distribute their information, then all confidentiality is lost: every device in the system may learn every piece of information.

In this paper we consider protocols designed to tolerate *honest, but curious* processes. This concept has attracted considerable attention as a model of processes in distributed applications that need limited anonymity and privacy, c.f., [15, 31, 38] (see more in related work below). It is not our protocol's intent to be secure against truly malicious parties: for data that must be kept secure in all circumstances, a more expensive solution is needed. For everyday transactions, however, where privacy is desired, we can ensure that no process ends up in possession of information that it is not intended to learn. Moreover, we can achieve this at relatively limited cost (in terms of message complexity), even if a moderate number of the participants may be colluding (i.e., sharing information).

Results. Thus the question we ask in this paper is whether we can achieve the benefits of collaboration—i.e., robustness and efficiency—without sacrificing confidentiality. We focus our attention on the problem of *Continuous Gossip*, a long-lived version of information sharing (introduced in [13]) that has three notable properties:

(1) any process can inject a *rumor* at any time;

(2) each rumor specifies a set of recipients that should receive the rumor; and

(3) each rumor has a deadline specifying by when it should be received.

In this paper, we present a continuous gossip protocol that guarantees all of the following desirable attributes:

Confidentiality: Only the specified recipients of a rumor learn the contents of the rumor, even if processes outside the specified recipients may collude.

Timeliness: Every rumor is delivered by the deadline.

Efficiency: The maximum *per-round message complexity*, with high probability, and in the absence of collusion is $O((n^{1+48/\sqrt{dmin}} + n^{1+6/\sqrt[6]{dmin}})\text{polylog}\, n)$, where $dmin$ is the shortest deadline of any *active* rumor (that is, a rumor whose deadline has not expired); note that for rumors with deadline of $\Omega(\log^6 n)$, this results in per-round message complexity of $O(n\text{polylog}\, n)$. When up to τ processes may collude, then we show that the maximum per-round message complexity is, with high probability, increased by a factor τ^2.

Robustness: Processes may crash and restart at any time; there is no bound on the number of crashed processes at any given time. Moreover, failures are ***adaptive***: they may depend on the execution and the random choices made by the individual processes. However, only the specified recipients that were continuously alive (formally defined later) are required to receive the rumor.

Our approach. The major challenge underlying confidential gossip is reconciling the need for collaboration to achieve efficiency, and the inherent loss of confidentiality created by collaboration. At first glance, it seems that only recipients of a rumor can help in its dissemination. Yet, if we limit all information regarding the rumor to its recipients, then it is impossible to achieve good message complexity. As we show in Theorem 1, if we limit messages regarding a rumor ρ to the *destination set* $\rho.D$, then the per-round message complexity is $\Omega(n^{(3/2)-\varepsilon}/dmax)$, for any $\varepsilon > 0$ ($dmax$ is the longest deadline of any rumor). Thus it seems that confidentiality and efficiency are inherently at odds.

We circumvent this seeming impossibility via a simple insight: each rumor can be divided into multiple independent fragments; each fragment provides no information regarding the original rumor, and yet together they can be combined to re-assembled the original rumor. This is the basic idea underlying *cryptographic secret sharing* [34, 36], though we require only the simplest instantiation of this idea. All the processes in the system can now collaborate to distribute the rumor fragments, as long we as ensure that no process collects all the fragments. (In fact, we can rely on existing gossip protocols as a black box, as long we restrict their communication to processes that are allowed to receive the specific message fragments.) In this way, we gain the benefits of collaboration without sacrificing confidentiality.

A second challenge we address is the possibility that failures are not independent and history-oblivious. We assume that processes may crash and restart at any time, and we model failures as being caused by an *adaptive* and omniscient adversary that can fail processes based on the random choices made by the protocol. For example, every time a source sends a rumor (or rumor fragment) to another process, the adversary may choose to immediately crash that recipient, entirely preventing the dissemination of that rumor. We address this challenge by having processes collaborate, exchanging *metadata* that contains no information on rumors. This information allows processes to determine which other processes are failed (or have failed recently), and which processes are being isolated (both in terms of sending and receiving information). Using this metadata, processes can target their messages better, and processes can adjust the number of messages they are sending. By collaborating on metadata, rather than rumors, processes can still overcome an adaptive adversary without giving up confidentiality. (While some information is leaked via the metadata, we discuss in Section 7 how to avoid this problem.)

Alternative approaches. There are two key limitations to our approach which may lead one to prefer a cryptographic approach. First, if a system must be secure against truly malicious parties (i.e., not simply "honest-but-curious" processes), then an alternate approach is needed for guaranteeing confidendtiality.

Second, the basic premise of gossip protocols is that by merging or aggregating rumors into a single message, we can gain efficiency. By collecting several rumors, it may defray some of the overhead of sending a message. Or it may be possible to combine rumors (e.g., if they are both measurements of the same phenomenon). However, if the rumors cannot be merged, then gossip protocols may not be efficient. On the other side, our protocols do introduce overhead to manage the correct delivery of all the rumors. If rumors are large, this seems reasonable. If rumors are small, this overhead may be significant. (It is unclear how the costs compare with cryptographic protocols, which have their own overheads; a more carefully optimized implementation of both approaches would be required to resolve this question.)

There are several possible cryptographic approaches to solving the problem of confidential gossip, many of which exist under the rubric of *multicast security* (e.g., [5, 12, 26, 29, 32, 34]). The basic idea, in many cases, is that each process holds some subset of the cryptographic keys; by encrypting the message with appropriate subsets of the keys, the sender can ensure that the message can only be decrypted by the intended recipients. For example, the processes may be arranged as leaves on a binary tree, where each internal node of the tree contains a cryptographic key; each process is given access to every key found on the root-to-leaf path ending at the leaf owned by the process. When the destination set of a rumor aligns well with the grouping of processes in the tree, such a scheme can be quite efficient; when the destination set contains processes distributed throughout the leafset, then such a scheme can be quite expensive. Many such solutions (e.g., [2, 28, 35, 37]) focus on a single source communicating confidentially with a single group of processes. The source establishes a shared key with the group, and then updates it as the group changes. Often, a tree-like scheme (as above) is used to make the re-keying more efficient.

In general, the cryptographic solutions will be more efficient when the groupings are stable. That is, when some processes want to communicate with a fixed set of destinations, these cryptographic solutions can be made quite efficient by ensuring that the fixed set of processes share a single cryptographic key. Even when there are occasional changes to the destination set, such solutions work reasonably well. However, we are not aware of any sub-quadratic, in terms of message complexity, cryptographic approach to guarantee confidential gossip when the groups are changing rapidly, or when there are no fixed groups, i.e., when each rumor has a different destination set. In many cases, the best solution appears to be encrypting the message individually for each process in the destination set, thereby significantly increasing the amount of data to be sent. Furthermore, there is the question on how efficient secret key maintenance would be in the presence of dynamic crashes and restarts, especially when restarted processes have no memory of the computation prior to restarting (as assumed in our model). As we show, our confidential gossip protocol is efficient even under such dynamic adverse conditions.

Other related work. The gossip problem has frequently been considered in relation to random, epidemic communication (e.g., [10, 19–21]). In this context, the problem is also know as *rumor spreading* and the protocols usually use a simple epidemic approach: each process periodically sends its rumor—along with any new rumors it has learned—to another randomly selected process. This approach can lead to efficient rumor dissemination while tolerating benign failures (e.g., [19]).

The gossip problem has been considered in a variety of fault-prone environments, ranging from crash failures to malicious/Byzantine ones (e.g., [6, 14, 17, 23, 25, 27]). Notably, solutions in Byzantine networks tend to focus on distributing rumors along disjoint paths to ensure that they have not been corrupted by Byzantine processes along the way. The survey by Pelc [33] together with the book by Hromkovic et al. [18] overview solutions for the gossip problem in fault-prone distributed networks.

Another line of work related to ours is the one considering the problem of constructing scalable overlays of topic-based Publication/Subscribe systems (e.g., [1,7,8,30]). The aim is to design an overlay network for each pub/sub topic, so that for each topic, the subgraph induced by the nodes interested in the topic will be connected; such overlays are called *topic-connected*. New events for each topic can then be routed from publishers only to interested subscribers using such topic-connected overlays. If destination sets are viewed as topics, then a topic-connected overlay could provide a confidential way of distributing a rumor to its destination set. Unfortunately, we don't know how to maintain topic-connected overlays in a dynamic setting (i.e., for changing destination sets), and even the static case is NP-complete [8, 30]. Theorem 1 (Section 3) effectively implies that topic-connected overlays cannot be used to support efficient confidential gossip.

The *honest-but-curious* model, also refered as the *semi-honest* model [4] is a standard cryptographic adversarial model [15]. This model has been widely considered in the problem of multi-party privacy-preserving computation of some function [15, 39]. The usual demand is for the function to be computed collectively by the computing entities without leaking any information about the entities' inputs, except that revealed by the algorithm's output. Various computations have been considered in the context of computations on sets, such as set union, intersection, element reduction (c.f., [22]), on graphs (c.f., [4]), in the are of data mining (c.f., [3, 24]), majority voting (c.f., [31]), in Cloud computing (c.f., [38]), as well as private predicate computation in mobile population protocols [9].

Our solution to the confidential gossip problem can be viewed as a tool in the process of computing these functions when the privacy of inputs (in the form of rumors) could be kept within groups of processes (i.e., certain rumors would have as their destination set a specific group). For example, a number of group of social networking websites, wishing to efficiently calculate aggregate statistics such as degrees of seperation and average number of acquaintances without compromising the in-group privacy, could use as a building block our confidential gossip algorithm.

In line with the above discussion, the work in [11] considers a variant of gossip, termed *social gossip*, which is motivated by the sharing of recommendations in a social network. The processes (users of the social network) require a rumor to be substantiated by multiple, independent sources in order to adopt it. Specifically, a user accepts a gossip message (recommendation) only if it has been received over $f+1$ disjoint gossip paths, for some parameter threshold f. The purpose of this requirement is to protect the network from spammers. Since a message needs to be carried over $f+1$ non-overlapping simple paths to be validated, then recommendations from bad sources are presumed impossible to reinforce. So, this work shares in spirit our approach of using different paths to disseminate information over the network. However, our objectives are somewhat different, as we spread over the network different pieces of a rumor, to prevent unauthorized processes to collect the whole rumor, while using these processes as relays for faster dissemination.

Paper organization. In Section 2 we present the model of computation and the confidential continuous gossip problem. In Section 3 we show that if only the processes of a rumor's destination set collaborate in disseminating the rumor, then high message complexity is unavoidable. In Section 4 we present and analyze an efficient randomized algorithm for confidential continuous gossip assuming no collusion and in Section 5 we provide its analysis. In Section 6 we show the effect of collusion on the problem under consideration and we modify our algorithm to tolerate collusions. We conclude in Section 7.

2 Model and Definitions

Processes. We consider a distributed system consisting of n synchronous processes that can communicate via message-passing over a reliable network, where each process can communicate directly with each other process. Message are not lost or corrupted in transit. Processes have unique ids from the set $[n] = \{1, \ldots, n\}$.

Synchronous communication. The computation proceeds in synchronous rounds. In each round, each process can: (i) send point-to-point messages to selected processes, (ii) receive a set of point-to-point messages sent in the current round, and (iii) perform some local computation (if necessary). We assume that processes have access to a global clock, that is, rounds are globally numbered.

Crash and restart. Processes may crash and restart dynamically as an execution proceeds. Each process is in one of two states: either `alive` or `crashed`. When a process is crashed, it does not perform any computation, nor does it send or receive any messages. We assume that processes have no durable storage, and thus when a process restarts, it is reset to a default initial state consisting only of the algorithm to execute and $[n]$. Each process can only crash or restart once per round. We denote by $crash(p, t)$ the event in which process p crashes in round t. The event $restart(p, t)$ is defined similarly. We say that a process p is ***continuously alive*** in the period $[t_a,\ t_b]$ if: (a) process p is `alive` at the beginning of round t_a and at the end of round t_b, and (b) for every $t \in [t_a,\ t_b]$, there are no $crash(p, t, \cdot)$ events.

When a process p crashes in round t, some of the messages sent by p in round t may be delivered, and some may be lost. Similarly, when a process p restarts in round t, some of the messages sent to p may be delivered and some may be lost.

Rumors. Rumors are dynamically injected into the system as the execution proceeds. A rumor ρ consists of a triplet $\langle z,\ d,\ D\rangle$, where z is the data to be disseminated, $D \subseteq [n]$ is the set of processes that z must be sent (destination set), and d is the deadline duration by which the rumor must be delivered. For ease of notation, we will be referring to $\rho.z$, $\rho.d$ and $\rho.D$ to the corresponding part of rumor $\rho = \langle z,\ d,\ D\rangle$. We denote by $Inj(\rho, t, p)$ the event in which rumor ρ is injected at process p in round t. We will be referring to p as the *source* process of rumor ρ. We assume that at most one rumor is injected at each process per round. We say that rumor ρ is ***active*** in round t if it was injected no later than round t and has a deadline after or during round t.

Adversary. We model crash/restarts and rumor injection via a ***Crash-and-Restart-Rumor-Injection adversary***, or $\boldsymbol{CRRI}$ adversary for short. In each round, the adversary determines which processes to fail, which processes to restart, and which rumors to inject. The adversary is ***adaptive*** in the sense that it can make decisions in a round t based on the events in all prior rounds before t, as well as the random choices being made in round t itself. We refer to an *adversarial pattern* $\mathcal{A} \in CRRI$ as a set of crash, restart and injection events caused by adversary $CRRI$.

Delivery guarantees. Ideally, we would like every rumor ρ injected into the system to be learned by all processes in $\rho.D$; moreover it should be delivered before the deadline. However, as argued in [13], this is not always possible: for example, a process $q \in \rho.D$ may be crashed throughout the duration of a rumor's lifetime. Following [13], we consider admissible rumors: we say that a rumor $\rho = \langle z,\ d,\ D\rangle$ injected at process p in round t is ***admissible*** for $q \in \rho.D$ if both p and q are continuously alive in the period $[t,\ t + d]$. This leads to the following definition of Quality of Delivery (introduced in [13]) that essentially requires admissible rumors to be delivered.

Definition 1 (Quality of Delivery) *We say that a gossip protocol guarantees* quality of delivery *if every rumor ρ injected in round t at a process p is delivered no later than round $t + \rho.d$ to every process in $\rho.D$ that is continuously alive for $[t,\ t + d]$, if p is also continuously alive for $[t,\ t + d]$.*

Confidentiality. We now formalize the notion of confidentiality, a property that was not considered in [13].

Definition 2 (Confidentiality) *We say that a gossip protocol is* confidential *if every rumor ρ is delivered only to processes in $\rho.D$, in every execution of the protocol.*

Message complexity. Typically, message complexity accounts for the total number of point-to-point messages sent during a given computation. However, in this work we allow for computations to have unbounded duration, and rumors may be injected into the system over an unbounded time period; thus counting the total number of messages sent in the entire computation is not meaningful. Instead, following [13], we focus on the number of messages sent *per round*.

Definition 3 (Per-round Message Complexity) *We say that a* randomized *algorithm Rand operating under adversary $CRRI$ has* per-round message complexity *at most $M(Rand)$, if for every round t, for every $\mathcal{A} \in CRRI$, the following holds with high probability: the number of messages sent $M_t(Rand, \mathcal{A})$ by Rand in round t, is at most $M(Rand)$.*

We allow messages to be of arbitrary size. Gossip protocols gain their efficiency by combining many rumors in a single message. In many cases, rumors can be merged or aggregated efficiently (e.g., when they are measurements of the same phenomenon). However, in other cases, if rumors are large and cannot be merged, this may result in large messages. At the same time, there is additional protocol overhead.

Essentially, for randomized algorithms our goal is to deterministically guarantee Quality of Delivery, that is, admissible rumors are delivered with probability 1, and confidentiality; we will guarantee the per-round message complexity with high probability. More on the rationale of Quality of Delivery and in general on the continuous gossip problem can be found in [13].

3 The Limitations of Strong Confidentiality

We say that a gossip protocol is *strongly confidential* if for every rumor, no message causally dependent on that rumor is ever sent to a process that is not in the destination set of that rumor. This essentially implies that only the processes in the destination set of a rumor can collaborate for that rumor's dissemination. As the following theorem states, such collaboration incurs high per-round message complexity, even against an *oblivious adversary* that can only arrange the rumors destination sets prior to the start of the computation.

Theorem 1 *For any constant $\varepsilon > 0$, every randomized strongly confidential gossip algorithm has a maximum per-round message-complexity of at least $\Omega(n^{(3/2)-\varepsilon}/dmax)$, with probability 1, even against an oblivious adversary, where $dmax$ is the longest deadline of the injected rumors.*

Proof: We may assume that n is sufficiently large (in fact, $n \geq 8$ is sufficient). Let c be a constant and x be a parameter, to be specified later. Suppose that only rumors with uniform deadlines $dmax$ are injected. Suppose that each process receives one rumor with a random set of destinations in the beginning of the computation, defined independently over processes and in such a way that for each process it is decided independently with probability x/n whether it belongs to this destination set (or not, otherwise).

We argue that under this scenario where all the rumors are injected simultaneously, with probability at least $1 - x^{2c+2}/n^{c-1}$, no message can carry more than c rumors. The intuition here (to be formalized later) is that a message from a process p_i to a process p_j can only contain a rumor that has both p_i and p_j in its destination set: otherwise, confidentiality has been violated. However, we can construct a set of rumors to inject so that no subset of $c+1$ rumors contains two (or more) nodes in the intersection of their destination sets. Hence, there is no pair of nodes p_i and p_j where p_i could send a message to p_j that contains that set of $c+1$ messages without violating confidentiality.

Having this, observe that the number of $(source\_process, destination\_process)$ pairs is at least $nx/2$, with probability at least $1-e^{-nx/8} \geq 1-1/e$, by a Chernoff bound. It follows that the total number of rumor copies carried by messages is at least $nx/2$ (e.g., count the rumors received by the destination processes). Therefore, the total number of messages needed to deliver all these rumors is at least $\frac{nx}{2c} = \Omega(nx)$, with probability at least $1-1/e-x^{2c+2}/n^{c-1}$.

Setting x to $n^{1/2-2/c}$, the above probability is at least $1-1/e-n^{-2} \geq 1-2/e > 0$. For any given ε, we can set $c = \lceil 2/\varepsilon \rceil$, which implies that the total number of messages needed to deliver the rumors is $\Omega(nx) \supseteq \Omega(n^{(3/2)-\varepsilon})$, with a positive probability. It follows, by applying the probabilistic method, that there is a configuration of destination sets of size x such that $\Omega(n^{(3/2)-\varepsilon})$ messages is necessary, and this holds in any fixed execution of any algorithm under such scenario. Hence, even randomized algorithms require total message complexity of $\Omega(n^{(3/2)-\varepsilon})$ with probability 1. In view of the deadline $dmax$, this implies a $\Omega(n^{(3/2)-\varepsilon}/dmax)$ per-round message complexity with probability 1.

Note that the suitable destination sets can be computed by the adversary prior to the computation, the injection round is the same for all the rumors, and no crashes/restarts are needed to enforce this lower bound — hence the adversary is oblivious.

It remains to prove that with a positive probability, no message can carry more than c rumors, for some constant c, in the scenario of random destination sets, and to define parameters x and c.

Consider any $c+1$ processes. The probability that there are at least two processes in the intersection of the destination sets of their rumors is at most

$$n^2 \cdot (x/n)^{c+1} \cdot (x/n)^{c+1} = x^{2c+2}/n^{2c} .$$

Here we used the union bound, where n^2 is the upper bound on the number of pairs in the destination set for the given set of $c+1$ rumors and $(x/n)^{c+1} \cdot (x/n)^{c+1}$ is the probability that for a given such pair of different processes they are both in all destination sets. Therefore, with this probability, the rumors of these processes cannot be sent together in one message (as there is no other process in the common destination for these rumors). The number of such $(c+1)$-tuples of processes is at most n^{c+1}. Consequently, by the union bound, we get that with probability of at most

$$x^{2c+2}/n^{2c} \cdot n^{c+1} = x^{2c+2}/n^{c-1}$$

there are $c+1$ processes with at least two processes in the intersection of the destination sets of their rumors. Recall that this event is a superset of the event that more than $c+1$ rumors can be carried out by some message in the execution. □

In view of the upper bound $O(n^{1+6/\sqrt[3]{dmin}}\text{polylog}\, n)$ on continuous gossip without confidentiality assumptions [13] (against an adaptive adversary), we obtain a polynomial, in n, *price of strong confidentiality*, in terms of per-round message complexity (for sufficiently long deadlines, i.e., with minimum deadline $dmin > 24$). The above result has motivated our study of the weaker version of confidential gossip that allows processes outside a rumor's destination set to receive a message related to this rumor, as long as the rumor datum is not revealed.

4 Gossiping Continuously and Confidentially

In this section we present a continuous gossip algorithm, called CONGOS, that guarantees that the content of rumors remains confidential under adversary $CRRI$. For simplicity, we assume no collusion. In Section 6, we show how to extend the approach here when processes collude. In this section, we describe the algorithm, and in Section 5, we show its correctness and analyze its performance.

In describing the algorithm, for clarity of presentation, we focus on how the algorithm operates on a collection of rumors that were injected in the same round and have the same deadline. In the beginning of Section 4.2, we explain how to partition the active rumors into $O(\text{polylog}\, n)$ sets where all the rumors in a set satisfy this criterion (by trimming the deadlines in a way that does not impact the asymptotic performance).

4.1 Overview of Algorithm CONGOS

In a nutshell, when a rumor is injected at a process p_i, the algorithm executes the following procedure (several times, in parallel):

Step 1: Process p_i splits the rumor into two fragments such that only a process with both fragments can reconstruct the rumor. (In Section 6.2, to cope with processes that collude, we will split each rumor into more than two fragments.) The processes are partitioned (deterministically) into two equal-sized groups.

Step 2: Since process p_i itself belongs to one of the two groups, it uses a black-box continuous gossip service to share one of the half rumors with its own group. It uses a Proxy Service to distribute the other half rumor to the other group, with which it cannot gossip directly. At the end of the second step, each non-failed process has received one of the two half rumors.

Step 3: The rumor fragments are sent to their appropriate final destinations using the GroupDistribution service. That is, the fragments for rumor ρ are sent to processes in the destination set $\rho.D$.

In more detail, rumors are injected in the ConfidentialGossip service. In order to ensure confidentiality, each rumor ρ is divided into two fragments ρ_0 and ρ_1. Both fragments maintain certain *metadata*, such as the rumor's destination set, but each fragment on its own provides no information as to the original rumor datum $\rho.z$; yet together, they allow the original rumor to be reconstructed.

There are a variety of simple schemes for accomplishing this. For example, let $\rho_0.z$ be a random binary string, and let $\rho_1.z = (\rho.z \textbf{ xor } \rho_0.z)$. It is a simple observation that any process that learns ρ_0 or ρ_1 (but not both) cannot deduce anything about the contents of the original rumor.

In this way, we have reduced the problem of confidentiality to the problem of ensuring that no process, except those in the destination set, learn both fragments of the rumor. In Lemma 3, we show that this guarantee is maintained, and hence that we achieve confidentiality.

A key to this guarantee is that all the processes in the system are partitioned into two components, and one rumor fragment is distributed to each half. It is not sufficient, however, to carry out this splitting-and-partitioning process only once: the adversary, being adaptive, may kill all the processes in one of the groups in the partition. We thus construct $\log n$ different partitions. For each partition ℓ, we construct a different (ρ_0, ρ_1) pair and send one fragment to each half of partition ℓ. We show in Lemma 5 that if at least two processors remain alive, then there is some partition that separates them into to groups. This fact implies that at least one partition will survive and work properly if the sender and target of a rumor are both still alive, i.e., the rumor is admissible.

The algorithm guarantees that no process outside a rumor's destination set gets both fragments of the rumor (for any partition), while all processes in the rumor's destination set (for which the rumor is admissible) deliver the rumor by the specified deadline.

4.2 High-level Construction

We now describe the overall structure of the algorithm, including the management of deadlines, the partitions, and the basic modules out of which the algorithm is constructed.

At any given time, each process executes many instances of the protocol, each designed for rumors injected in a specific round and with the same deadline. First, we trim the deadlines that are unnecessarily big. When a rumor has a deadline bigger than $c \log^6 n$ for some constant $c > 0$, we truncate the deadline to $c \log^6 n$. This does not increase the asymptotic message complexity of the protocol. Thus, at any given time, the active rumors were injected in at most $O(\log^6 n)$ different rounds. Next, we further truncate each deadline by rounding down to the nearest power of 2, i.e., a rumor with deadline $dline$ has its deadline reduced to $2^{\lfloor \log n \rfloor}$. Again, this does not increase the asymptotic message complexity of the protocol. Therefore, for rumors injected in the same round, there are at most $O(\log \log n)$ different possible deadlines. So in order to support all possible rumors, we run $\Theta(\log \log n \log^6 n)$ instances of the protocol in parallel. From now on, we can focus on rumors that were injected in the same round with the same deadline. This increases the per round message complexity by at most an $O(\text{polylog } n)$ factor.

Now, for each instance of the protocol, we construct $\log n$ partitions. Each partition is based on a specified bit in the binary representation of a process's identifier. Let $p_j[\ell]$ be the ℓ^{th} bit in p_j's binary representation. Then partition ℓ is defined by the two sets $P_{0,\ell} = \{p_j : p_j[\ell] = 0\}$ and $P_{1,\ell} = \{p_j : p_j[\ell] = 1\}$.

Each instance of the protocol consists of three basic modules, which will be described in further detail in the rest of the section:

- ConfidentialGossip: This is the main protocol that coordinates the distribution of messages. When a rumor is injected, this module divides it into fragments and initiates its distribution via the other services. Further details are given in Section 4.3.

- Proxy[ℓ]: The proxy safely hands messages from one group in partition ℓ to another, whereupon they can be distributed using the GroupDistirbution[ℓ] service. Further details are given in Section 4.4.

- GroupDistribution[ℓ]: This component distributes the rumor fragments, for partition ℓ, to their final destination set. Further details are given in Section 4.5.

The algorithm CONGOS also relies on a collection of pre-existing distributed services. We will treat these existing distributed protocols as a block box, without delving into the underlying implementation details. We assume that the system consists of the following services:

- Network: We model the communication network as one such distributed service, with local input port *send* and a local output port *receive* at each process. As already described in Section 2, any pair of processes can communicate and communication is reliable (i.e., messages are not lost) and synchronous.

- GroupGossip[ℓ]: We assume the availability of an existing *Continuous Gossip service*, albeit, one that does not guarantee confidentiality. It does ensure, however, that every admissible rumor is delivered (with probability 1) by the specified deadline, and it bounds the per-round message complexity by $O(n^{1+6/\sqrt[3]{dmin}} \text{polylog } n)$, where $dmin$ is the shortest deadline of any active rumor.

  This service can be realized by the deterministic gossip algorithm developed for continuous gossip in [13]. This algorithm begins with a classical gossip approach and derandomizes it by replacing random choices with carefully chosen expander graphs. In order to achieve good per-round efficiency, it relies on two mechanisms that execute concurrently: the first mechanism discovers other collaborators, i.e., processes with active rumors (rumors whose deadline has not expired); the second mechanism distributes the rumors to their destination sets (for which the rumor is admissible). At any given time, any (or all) of the collaborating processes may crash, and the remaining processes must finish the job. As the details are a bit involved, and to avoid a restatement of the results, we refer the reader to [13] for details. For our purposes is enough to treat this service as a black box.

  We assume there are $\log n$ instances of this continuous gossip service, GroupGossip[ℓ] for $\ell \in \{1, \ldots, \log n\}$. The instance GroupGossip[ℓ] is associated with partition ℓ of the network which divides the network into two parts. Every message sent by GroupGossip[ℓ] is *filtered* before being sent over the network: if a process p_i is a member of some group P' in partition ℓ, then every message sent by GroupGossip[ℓ] at process p_i to a process *not* in P' is dropped; every message sent by GroupGossip[ℓ] at process p_i to a process in P' is relayed to the Network and sent. From the perspective of the instance GroupGossip[ℓ], the processes that cannot be reached due to the filter are effectively failed. (The continuous gossip service can tolerate arbitrary failures.)

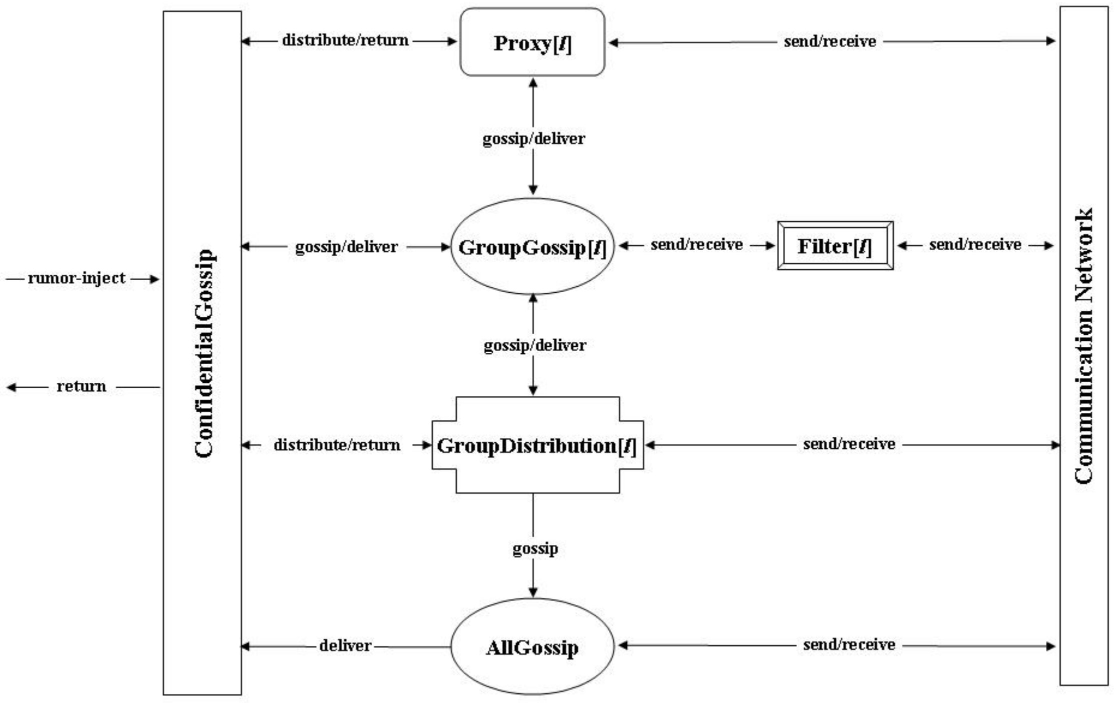

Figure 1: The interaction between the services at process i, for a partition ℓ.

- AllGossip: We assume a single continuous gossip service, AllGossip, that is not filtered. That is, it is allowed to communicate with all processes in the system.

Figure 1 depicts the interaction of the various services at a process i for a partition ℓ.

4.3 Main Protocol

We now describe the procedure taken when a rumor is injected in the ConfidentialGossip service. Figure 2 provides a high-level outline of the ConfidentialGossip service at a process p_i. (A more detailed pseudocode is given in Figure 8 in the Appendix.)

The protocol consists of four main parts, and the timing of the process is as follows: Time is divided into blocks of $dline/4$ rounds. A rumor injected during some block B is split into fragments during block B (step 1); the fragments are distributed to their respective groups during block $B + 1$ (step 2); the fragments are reassembled in block $B + 2$ (step 3 and 4); and the source verifies that its rumor has been delivered during block $B + 3$. If it cannot verify that its rumor has been delivered before the deadline expires, then it simply sends its rumor directly (last bullet).

A notable aspect of the above protocol is that a process cannot directly distribute both rumors. If a process p_i is in group $P_{0,\ell}$, it cannot directly participate in gossip with group $P_{1,\ell}$; if it did, it might risk learning rumor fragments associated with the other group. The Proxy service is designed to circumvent this problem.

Another notable aspect occurs at the end of the protocol, when a process attempts to confirm that its rumors have been delivered. Each process, as part of the GroupDistribution service (see below), initiates a gossip (via AllGossip) indicating which rumor fragments have been distributed to which processes. Of course, a process cannot divulge the *contents* of the rumor that have been distributed; however, it can safely indicate a unique identifier that was appended by the source, when the rumor was split. In this way, the source can ensure that, for at least one of the fragment pairs, both rumor fragments were successfully delivered. (Note that it would not be sufficient for recipients to send an acknowledgment to the sender, as the source does not know which processes have remained alive throughout the interval.) This fallback delivery mechanism is used in the proof of Lemma 4 where we show that every admissible rumor is delivered within the deadline.

Outline of ConfidentialGossip **service at** p_i**:**

- Do in parallel for each $\ell = 1, \ldots, \log n$:
  1. Split rumor ρ into a pair $\langle \rho_{0,\ell}, \rho_{1,\ell} \rangle$.
  2. If p_i is in group $b \in \{0, 1\}$ of partition ℓ, inject $\rho_{b,\ell}$ into GroupGossip$[\ell]$, and inject $\rho_{1-b,\ell}$ into Proxy$[\ell]$. Together, these two services ensure that each rumor fragment is delivered to every non-failed process in the appropriate group of the partition.
  3. For each rumor fragment received from GroupGossip$[\ell]$ or Proxy$[\ell]$, inject the fragment into GroupDistribution$[\ell]$.
  4. Save every fragment received from GroupDistribution$[\ell]$, and reassemble and deliver rumors as fragments become available.
- Whenever a message from AllGossip confirms that, for some partition ℓ, both fragments of a rumor ρ, initiated at p_i, have been sent to every destination in $\rho.D$, confirm that ρ has been delivered.
- Whenever a deadline is about to expire for some rumor ρ initiated at p_i, and there is no confirmation that ρ has been delivered, send ρ directly to every process in $\rho.D$. (A simple optimization would be to only send rumors to destinations for which no confirmation was received. Since this is a low probability event, it has little impact on performance.)

Figure 2: Outline of ConfidentialGossip service at p_i.

4.4 Proxy Service

The goal of the proxy service is to deliver rumor fragments safely across group boundaries. Essentially, the proxy service for partition ℓ at process p_i repeatedly samples processes from the other group $P_{1-p_i[\ell],\ell}$ (i.e., the group that p_i does not belong to), requesting that these processes act as *proxy* for p_i in distributing its rumor fragments. The potential proxies then participate in GroupGossip$[\ell]$, attempting to distribute the rumor fragments, as requested. If they succeed, they send an acknowledgment to p_i. Otherwise, process p_i needs to try again.

The challenge, here, is that the adversary may (adaptively) crash processes as soon as they receive proxy requests. (In fact, for some partitions, the adversary may crash all the members of a given group.) Even worse, at any given time, most of the members of a group may be failed, requiring p_i to send a very large number of queries to find a proxy. To avoid this problem, the processes in the same group collaborate on finding proxies. At the same time, p_i does not share any information on the fragments it is attempting to distribute in the *other group* $P_{1-p_i[\ell],\ell}$ with processes in the *same group* $P_{p_i[\ell],\ell}$ with which it is collaborating. The Proxy service for partition ℓ is outlined in Figure 3 (a detailed pseudocode is given in Figure 9 in the Appendix).

For correctness, there is one key guarantee that the Proxy service provides:

> [PROXY:CONFIDENTIAL]: The Proxy service ensures confidentiality, i.e., a process p_i never sends a fragment $\rho_{0,\ell}$ to a process in group $P_{1,\ell}$, and never sends a fragment $\rho_{1,\ell}$ to a process in group $P_{0,\ell}$. This is immediately true by construction, and is used in Lemma 3.

For message complexity there are two key guarantees that the Proxy service provides:

> [PROXY:MESSAGES]: The Proxy service does not send too many messages, i.e., it sends at most $O(n^{1+48/\sqrt{dline}} \log n)$ messages per round, by *all* the nodes collectively (excluding messages sent by GroupGossip). We show this in Lemma 7, and it follows immediately by construction because each collaborator sends messages to at most $O(n^{1+48/\sqrt{dline}} \log n / |collaborators|)$).

> [PROXY:DELIVERY]: The Proxy service guarantees that each admissible rumor fragments received by the service is distributed to every process in the opposite group that has been alive throughout the relevant period within $dline/4$ time, with high probability. This is proven in Lemma 8. The key difficulty in the proof is showing that the processes in one group successfully collaborate to find a non-failed proxy in the other group, which can then

Outline of Proxy[ℓ] **at** p_i**:**

- Time is divided into blocks of length $dline/4$.
- At the beginning of a block, if has been alive for at least $dline/4$ rounds, then collect all the fragments that have been injected since the last block began, and if there is at least one such fragment, then set $status$ to active.
- Each block is divided into *iterations* of $\sqrt{dline}+2$ rounds. In each iteration, we maintain a set $collaborators$ of the active processes (i.e., processes with $status$ active) in the same group as p_i, i.e., $P_{p_i[\ell],\ell}$. We also keep track of *failed-proxies*, i.e., those that we have already learned (in previous iterations) have failed in this block. For each iteration, repeat (the following are executed if $status$ = active):
  - Round 1: send every rumor fragment associated with the other group (i.e., rumor fragment $\rho_{1-p_i[\ell],\ell}$) to $\Theta(n^{1+48/\sqrt{dline}}\log n/|collaborators|)$ processes chosen uniformly at random from group $P_{1-p_i[\ell],\ell}$, excluding processes in $failed\text{-}proxies$. (Notice that as long as the set $collaborators$ is a good estimate of the set of collaborators, this ensures a good bound on the message complexity of this step.) Every process that receives a request to be a proxy for the other group caches the received rumor fragments.
  - Rounds $2,\ldots,\sqrt{dline}+1$: initiate a GroupGossip[ℓ] in which processes in $P_{p_i[\ell],\ell}$ share the set of $failed\text{-}proxies$, as well as establish the set of $collaborators$, i.e., members of the group that still have $status$ active. Processes also share all the rumor fragments received from group $P_{1-p_i[\ell],\ell}$. (The deadline for rumors in GroupGossip[ℓ] here is $\sqrt{dline}$.)
  - Round $\sqrt{dline}+2$: Any process that was asked to be a proxy for the other group sends an acknowledgment that proxying was successful. Any process that sent a request, and does not receive an acknowledgment, adds the non-acknowledging processes to the set of $failed\text{-}proxies$.
- Upon recovering from a failure, obtain the round number from the global clock, set $status =$ idle and wait until a new block begins.

Figure 3: Outline of Proxy[ℓ] at p_i

distribute the message using the GroupGossip service (as a black box). The basic idea is that in every round of collaboration, either a large fraction of processes in one of the two groups fail, or there is a good probability of finding a non-failed proxy; hence by repeating a small number of times, the protocol will succeed.

4.5 GroupDistribution Service

The goal of the GroupDistribution[ℓ] service is to distribute rumor fragments to their final destination. To this point, for a partition ℓ, the rumor fragment $\rho_{0,\ell}$ has been distributed to processes in $P_{0,\ell}$, and the rumor fragment $\rho_{1,\ell}$ has been distributed to processes in $P_{1,\ell}$, provided that the rumor is admissible. Now, group $P_{0,\ell}$ collaborates to send the fragment $\rho_{0,\ell}$ to $\rho.D$, while $P_{1,\ell}$ does the same for fragment $\rho_{1,\ell}$. Of course there may be many different fragments active in each group, each with a different destination set.

The basic operation of the GroupDistibution is similar to that of the Proxy Service. Each process chooses a set of recipients at random, and sends each of them a message carefully composed to only include appropriate rumor fragments. The processes then gossip within their group (via GroupGossip[ℓ]), sharing information on which processes have already been notified, and which remain to be notified. At the same time, processes calculate the number of processes active in a group (have $status =$ active), which allows them to determine the appropriate number of messages to send. Here, for a process to have $status$ active, is enough to have been alive at the beginning of the second round of a block for a sufficient number of rounds, regardless if it has rumor fragments or not (as opposed to Proxy[ℓ]). Figure 4 outlines the GroupDistribution service for partition ℓ (a detailed pseudocode is given in Figure 10 in the Appendix).

For correctness, there are two key guarantees that the GroupDistribution service provides:

[GD:CONFIDENTIAL]: The GroupDistribution service ensures confidentiality, i.e., a process p_i only sends a fragment $\rho_{*,\ell}$ to processes in the destination set for rumor ρ. This is immediately true by construction, and is used in Lemma 3.

Outline of GroupDistribution[ℓ] **at** p_i**:**

- Time is divided into blocks of length $dline/4$.
- At the beginning of the second round of a block, if has been alive for at least $2dline/3$ rounds, then collect all the fragments that have been injected since the first round of the block, and set $status$ to active. (The first round of the block is spent waiting for rumor fragments from the previous block.)
- Each block is divided into *iterations* of $\sqrt{dline} + 2$ rounds. In each iteration, we maintain a set $collaborators$ of the active processes (i.e., processes with $status$ active) in the same group as p_i, i.e., $P_{p_i[\ell],\ell}$. We also keep track of a set $hitSet$ of processes that have been sent a message in this block; each process in this set was sent all the rumor fragments for this block. For each iteration, repeat (as long as $status =$ active):
  - Round 1: wait for rumor fragments to be injected.
  - Round 2: send every "appropriate" rumor fragment to $\Theta(n^{1+48/\sqrt{dline}} \log n/|collaborators|)$ processes chosen uniformly at random from group $P_{1-p_i[\ell],\ell}$, excluding processes in $hitSet$. By appropriate we mean that if p_j is a process chosen randomly by p_i, then p_i sends to p_j only the rumor fragments in which p_j is in the destination set. (Recall that each partial rumor contains the target destination set as part of the metadata.) Every process that receives rumor fragments can now reconstruct the rumor and return it to its user (via the ConfidentialGossip service).
  - Rounds $3, \ldots, \sqrt{dline + 2}$ rounds: initiate an instance of GroupGossip[ℓ] (with deadline $\sqrt{dline}$) in which processes in group $P_{p_i[\ell],\ell}$ share their $hitSet$s, as well as count how many members of the group are still active (have $status =$ active).
- In the last round of the block, initiate an instance of AllGossip (with deadline $dline/4 - 1$). Each process p_i gossips the information in its $hitSet$, but without including the rumor fragments themselves. That is, if the $hitSet$ of process p_i indicates that some rumor fragment $\rho_{0,\ell}$ was sent to some process p_j, and if $\rho_{0,\ell}$ has identifier r, then p_i gossips that the fragment 0 for partition ℓ of the rumor associated with identifier r was sent to p_j. This provides sufficient information for the source to determine whether the rumor was delivered, without revealing the contents of the rumor. (See the description of the ConfidentialGossip service, above, for how this information is used.)
- Upon recovering from a failure, obtain the round number from the global clock, set $status =$ idle and wait until a new block begins.

Figure 4: Outline of GroupDistribution[ℓ] at p_i.

[GD:CONFIRM]: The GroupDistribution only initiates an AllGossip containing metadata on a rumor fragment $\rho_{*,*}$ if that rumor fragment has been sent to all the processes in the destination set for ρ. This property is proved in Lemma 4.

For message complexity there are two key guarantees that the GroupDistribution service provides:

[GD:MESSAGES]: The GroupDistribution service does not send too many messages, i.e., it sends at most $O(n^{1+48/\sqrt{dline}} \log n)$ messages per round, by *all* the nodes collectively (excluding messages sent by GroupGossip and AllGossip). We show this in Lemma 7, and it follows immediately by construction because each collaborator sends messages to at most $O(n^{1+48/\sqrt{dline}} \log n/|collaborators|))$.

[GD:DELIVERY]: The GroupDistribution service guarantees the following: if R is the set of admissible rumor fragments received by every process in a group that is alive during the relevant period, then every rumor $\rho \in R$ is received by everyone in ρ's destination set within $dline/4$ time, with high probability. This is proven in Lemma 9. The lemma proves that members of the second group collaborate to distribute the rumor to the destination processes in the first group. The argument proceeds much like the argument for the proxy, showing that in each round of collaboration, either a large fraction of collaborators fail or a large fraction of destinations are informed.

Together, [PROXY:DELIVERY] and [GD:DELIVERY] (i.e., Lemma 8 and Lemma 9) show that every admissible rumor fragment is first distributed to both groups within $dline/4$ time by the Proxy service, and then is sent to the proper destination sets by the GroupDestination service.

5 Algorithm Analysis

The proof is divided into two parts. First, we focus on correctness, which follows essentially by construction: the algorithm explicitly prevents any process not in the destination set from receiving the rumor (i.e., both fragments in a pair), and it ensures that the deadline is met by sending the message directly if delivery has not been confirmed on time. The second and more critical part of the proof has to do with the message complexity. To that end, we carefully count the messages (and ensure that the rumors are delivered fast enough) to keep the message complexity low.

Throughout the analysis we assume that $dline > 48$. If it is not, then the desired bound can be trivially met simply by sending rumors directly to their destination sets by the source.

5.1 Correctness

We begin the analysis of algorithm CONGOS by stating its correctness, i.e., confidentiality is not violated and all admissible rumors are delivered on time.

Theorem 2 (Correctness) *Algorithm* CONGOS *correctly solves the Confidential Continuous Gossip problem under adversary $CRRI$.*

The proof of the theorem follows directly from the following two lemmas. The first lemma states that confidentiality is not violated.

Lemma 3 (Confidentiality) *In any execution of algorithm* CONGOS*, and for any rumor ρ, if $q \notin \rho.D$, then at no point during the execution q learns $\rho.z$. This occurs with probability 1.*

Proof: Consider a rumor ρ injected at a processes p at round t of an execution of algorithm CONGOS. Also consider a process $q \notin \rho.D$. We will show by investigating the flow of the algorithm that at no point will process q learn $\rho.z$, or be able to re-construct it.

Once ρ is injected at process p, based on the description of ConfidentialGossip at process p, and focusing on a partition ℓ, the rumor is split into $\rho_{0,\ell}$ and $\rho_{1,\ell}$. We consider the case where both processes are in the same group in partition ℓ. The other case is symmetric. If p fails before distributing the fragments, then clearly confidentiality is maintained.

Without loss of generality, say that they belong in group $P_{0,\ell}$. Per the description of ConfidentialGossip, $\rho_{0,\ell}$ is disseminated, using GroupGossip[ℓ], *only* to processes in group $P_{0,\ell}$ (this is guaranteed by $Filter[\ell]$).

Proxy Confidentiality. We will focus on ensuring that q does not receive fragment $\rho_{1,\ell}$, which is disseminated, using Proxy[ℓ], *only* to processes in group $P_{1,\ell}$. This follows due to the [PROXY:CONFIDENTIAL] property, which follows from the description of Proxy[ℓ]: For any iteration and any block, in the first round the processes in group $P_{0,\ell}$ send rumor fragments $x_{1,\ell}$ (including $\rho_{1,\ell}$) to processes in the other group (i.e., $P_{1,\ell}$) and then GroupGossip[ℓ] is used to disseminate these rumor fractions in group $P_{1,\ell}$ only. Hence, up to this point, process q knows (at most) only $\rho_{0,\ell}$.

GD Confidentiality. Next, the [GD:CONFIDENTIAL] property ensures that the confidentiality is maintained by the GroupDistribution service. Specifically, using the GroupDistribution[ℓ] service, processes in $P_{0,\ell}$ collaborate in sending (among other rumor fragments) $\rho_{0,\ell}$ to each process $w \in P_{1,\ell}$ where $w \in \rho.D$ (this knowledge is retrieved using the mentioned metadata). Similarly, the processes in $P_{1,\ell}$ disseminate $\rho_{1,\ell}$ to processes in $P_{0,\ell}$ that belong in $\rho.D$. By assumption $q \notin \rho.D$ and hence no process in $P_{1,\ell}$ ever sends $\rho_{1,\ell}$ to process q. The exchange of $hitSet$s does not reveal any information on the rumor fragments, nor does the instance of AllGossip used in the last round of the last iteration of each block in GroupDistribution[ℓ]. So process q still knows only $\rho_{0,\ell}$.

Finally, if the deadline of rumor ρ is about to expire, and there is no confirmation that ρ has been delivered (see the last two bullets in the outline of ConfidentialGossip), then the process p sends ρ directly to every process in $\rho.D$. Again, since $q \notin \rho.D$, p does not send ρ to q. Note that after this phase or if p receives a confirmation that ρ has been delivered, no further message is exchanged with respect to ρ and hence q never gets to learn ρ. As q cannot construct ρ only from $\rho_{0,\ell}$ or any combination of different partitions of ρ in the various group partitions (that run in parallel), the thesis of the lemma follows. □

We now show that algorithm CONGOS delivers admissible rumors before they expire.

Lemma 4 (Correctness wrt QoD) *In any execution of algorithm* CONGOS*, and for any rumor ρ injected at a process p at round t of the execution, if p and $q \in \rho.D$ are continuously alive for the lifetime of the rumor, then q learns ρ by round $t + \rho.D$. This occurs with probability 1.*

Proof: Fix $\rho = \langle z, d, D\rangle$ injected at process p at round t, and consider process $q \in \rho.D$. Both p and q are continuously alive for the rumors lifetime. Hence ρ is admissible for q. We show that q will learn, with probability 1, rumor ρ by time $t + \rho.d$.

From the last two bullets in the outline of ConfidentialGossip we have that if the deadline of rumor ρ is about to expire, and there is no confirmation that ρ has been delivered, then process p sends ρ directly to every process in $\rho.D$, including q. Hence, what remains to be proved is that if p receives confirmation that ρ was delivered before the rumor has expired, then q has indeed learned ρ (i.e., it has learned both fragments of ρ is some partition ℓ).

Process p will confirm that ρ has been delivered if a message from AllGossip confirms that, for some partition ℓ, both fragments of a rumor ρ have been sent to every destination in $\rho.D$ (including q). Thus, it remains only to prove that the GroupDistribution service satisfies the [GD:CONFIRM] property, which follows from these observations:

- No process will include q in its $hitSet$ if it has not sent its partial rumor to q in the GroupDistribution service of some partition ℓ.
- Say w is a process that has included q in its $hitSet$. The dissemination of $hitSet$ takes place after rumor fragments to the processes being "hit" are sent. If w would fail prior to hitting q, then its $hitSet$ containing q would not have been disseminated through the AllGossip service. Hence, when p receives a $hitSet$ (from AllGossip) containing q, it is the case that q was indeed hit.
- Messages sent from non-faulty processes to non-faulty processes are not lost (unless they go through the filter, which is not the case here).

We therefore conclude that the initiator of a rumor only gets a confirmation of the rumor when it was delivered; this completes the proof. □

5.2 Message Complexity

The remainder of the proof focuses on message complexity. In the next few lemmas, we state important properties needed for analyzing the message complexity of the algorithm.

First, we need to show that at least one of the partitions is "good," i.e., has at least one process in each group that is alive. (Otherwise, the subservices will surely fail.)

Lemma 5 *Given rumor ρ, injected at time t: if there are at least 2 processes that remain alive throughout the interval $[t, t + \rho.d]$, then for some partition ℓ, there is at least one process in $P_{0,\ell}$ and one process in $P_{1,\ell}$ that remain alive throughout the interval $[t, t + \rho.d]$.*

Proof: Let p_i and p_j be the two processes hypothesized to remain alive throughout the specified interval. Since identifiers are unique, let ℓ be some bit where the identifier of p_i and p_j differ. Thus in partition ℓ, processes p_i and p_j are divided between the two groups and remain alive throughout the specified interval. □

Next, we can bound the length of time that the Proxy and GroupDistribution services will take for each block. This will be important for proving [PROXY:DELIVERY] and [GD:DELIVERY], once we show that each service successfully delivers the fragments in a block.

Lemma 6 *In each block, the* Proxy[ℓ] *service and the* GroupDistribution[ℓ] *service execute at least $\sqrt{dline}/8$ iterations, if $dline > 4$.*

Proof: Each block is of length $dline/4$, which is divided into iterations of length $\sqrt{dline} + 2 \le 2\sqrt{dline}$. Thus each block contains at least $(dline/4)/(2\sqrt{dline}) \ge \sqrt{dline}/8$ iterations. □

We can now prove that the Proxy service satisfies the [PROXY:MESSAGES] property and the GroupDistribution service satisfies the [GD:MESSAGES] properties. Since the two services behave in a very similar manner, we prove the two properties together.

Lemma 7 *In each round, the* Proxy$[\ell]$ *service and the* GroupDistribution$[\ell]$ *service send at most* $O(n^{1+48/\sqrt{dline}} \log n)$ *messages per round. (This occurs with probability 1.)*

Proof: We first observe that, throughout a block, every process p_i that is sending messages remains in the *collaborators* set for every other p_j in the same group of partition ℓ: Initially, *collaborators* contains every process; in each iteration. If p_i and p_j both remain alive to proceed in a later iteration, it means they both sent a rumor in the previous iteration indicating that they were alive. By the (deterministic) guarantees of GroupGossip, this rumor must have been delivered, and hence $p_i \in collaborators_j$ (and *vice versa*). Notice that if a process is restarted, it does not begin sending messages again until the next block.

In Proxy$[\ell]$ and GroupDistribution$[\ell]$, each process sends $n^{1+48/\sqrt{dline}} \log n/|collaborators|$ messages in, respectively, the first and second round of an iteration. Since the *collaborators* set is at least as large as the set of processes performing this step, the desired result follows.

In Proxy$[\ell]$, each process that received a proxy request sends one additional message, i.e., a response, at the end of an iteration. Each response is the result of an earlier request in the first round of the iteration, and we have already bounded the message complexity of the first round of an iteration, leading here too to a bound of $O(n^{1+48/\sqrt{dline}} \log n)$. □

Next, we will prove that each admissibile rumor fragment is distributed to all the (non-failed) processed in the proper group. For rumor fragments that are created in the same group to which they belong, this follows in a straightforward fashion from the continuous gossip service.

For rumor fragments that need to be distributed to the opposite group, we depend on the Proxy service, and specifically, the [PROXY:DELIVERY] property. The challenge of showing the next lemma lies on the fact that the adversary is adaptive. If the adversary were oblivious, then we could simply analyze the random choices as independent. But because the adversary is adaptive and can schedule according to the random choices, there is subtle correlation among the random choices, and hence we have to show that the requisite properties hold despite all possible adversarial choices.

Lemma 8 *Given admissible rumor ρ, injected at time t at process p_i: if for some group k in partition ℓ, at least one process in $P_{k,\ell}$ remains alive throughout the interval $[t, t+\rho.d]$, then every process in $P_{k,\ell}$ that remains alive throughout the interval $[t, t+\rho.d]$ receives $\rho_{k,\ell}$ by time $t + 2dline/4 - (t \mod dline)$, with high probability. That is, for any constant c, it will succeed with probability at least $1 - 1/n^c$.*

Proof: Fix ℓ and k so that at least one process in $P_{k,\ell}$ remains alive throughout the interval. Assume that the initiator $p_i \in P_{k',\ell}$ for some group k'.

If $k = k'$, then the claim follows immediately from the property of the GroupGossip service: Since p_i injects rumor fragment $\rho_{k,\ell}$ into the GroupGossip$[\ell]$ service with deadline $\sqrt{dline}$, it is guaranteed to reach every process in $P_{k,\ell}$ that remains alive throughout the interval.

Assume that $k \neq k'$. It remains to analyze the behavior of the Proxy$[\ell]$ service, focusing on the first complete block after rumor ρ is injected beginning at time $t + dline/4 - (t \mod dline)$.

We need to show that each process in $P_{k',\ell}$ succeeds in finding a proxy in $P_{k,\ell}$. Let $Z = n^{48/\sqrt{dline}}$. We will argue that in every pair of iterations, one of the three following events occurs:

1. At least a $(1 - 1/Z)$ fraction of processes in $P_{k,\ell}$ that were alive at the beginning of the first iteration fail by the end of the second iteration.

2. At least a $(1 - 1/Z)$ fraction of processes in $P_{k',\ell}$ that were alive at the beginning of the first iteration fail by the end of the second iteration.

3. In the second iteration, at least a $(1 - 1/Z)$ fraction of processes in $P_{k',\ell}$ succeed in finding a proxy, with high probability.

Notice that from this claim, we can conclude that by the end of $3\log_Z(n)$ pairs of iterations, either every process in one of the two groups has failed, or every process in $P_{k',\ell}$ has succeeded in finding a proxy. Since by assumption there is at least one process alive in both groups k and k' of partition ℓ, and since $\log_Z(n) = \sqrt{dline}/48$, we conclude that by the end of $6\log_Z n \leq \sqrt{dline}/8$ iterations, every process in $P_{k',\ell}$ has succeeded in finding a proxy. And once a process has succeeded in finding a proxy, it follows from the (deterministic) guarantees of GroupGossip that its rumor fragment is distributed to every non-failed process in the other group.

We now argue that in each iteration, one of the three events described above occurs. Fix a pair of iterations, let A be the set of processes in $P_{k',\ell}$ still active at the beginning of the first iteration, and let B be the set of processes in $P_{k,\ell}$ still active at the beginning of the first iteration. Assume the neither event (1) nor event (2) occur, i.e., that at the end of the iteration there are at least $|A|/Z$ processes still active in $P_{k',\ell}$ and $|B|/Z$ processes still active in $P_{k,\ell}$.

We now calculate the probability that a process in A successfully finds a proxy in B that does not fail by the end of the second iteration. In the second iteration, each process in A has an estimate (for its set of collaborators) of the size of A that is at most $|A|$. Recall that each process sends some $\Theta((n/|A|)\cdot Z\log n)$ proxy messages. Fix the constant in the asymptotic notation to be at least $(c+3)$. Thus, every process in A sends at least $(c+3)(n/|A|)\cdot Z\log n$ proxy requests, for some constant c.

Moreover, in the first iteration, each process adds every process not in B to its set of *failed-proxies*, so we can conclude that every proxy request is sent to some process in B.

Fix some subset $A' \subseteq A$ of size at least $|A|/Z$, and fix some subset $B' \subseteq B$ of size at least $|B|/Z$. We now calculate the probability that a given process in A' fails to send a message to some process in B' as:

$$(1-|B'|/|B|)^{(c+3)nZ\log n/|A|} \leq (1-1/Z)^{(c+3)nZ\log n/|A|} \leq (1/e)^{(c+3)n\log n/|A|}$$

Similarly, the probability that *every* one of the at least $|A|/Z$ processes in A' fails to send a message to some process in B' is at most $e^{-(c+3)n\log n/Z}$.

We now take a union bound over all possible sets A' and B'. Specifically, there are at most $|A|^{|A|/Z} \leq e^{(|A|/Z)\log|A|}$ possible sets A'; there are at most $|B|^{|B|/Z} \leq e^{(|B|/Z)\log|B|}$ possible sets B'. Thus, the probability that there exists any set A' and any set B' where every process in A' misses every process in B' is at most:

$$\frac{e^{(|A|/Z)\log|A|+(|B|/Z)\log|B|}}{e^{(c+3)(n/Z)\log n}} \leq \frac{e^{(n/Z)\log n+(n/Z)\log n}}{e^{(c+3)(n/Z)\log n}} \leq e^{-(c+1)(n/Z)\log n} \leq 1/n^{c+1} .$$

Taking a union bound over the (at most) $\sqrt{dline}/8$ iterations, we conclude that with probability at least $1-1/n^c$, $|A|/Z$ processes do not successfully find proxies in each such pair of iterations, as required. □

We next prove the key claim that admissible rumor fragments are delivered to every process in the proper destination set. Having already proved [PROXY:DELIVERY], the key missing piece (and the bulk of the proof) is the analogous property, [GD:DELIVERY], for the GroupDistribution service. This shows that if the fragments are delivered to a group, then the GroupDistribution service will delivery them to the proper destination set.

Lemma 9 *Given admissible rumor ρ, injected at time t at process p_i: if for some group k in partition ℓ, at least one process in $P_{k,\ell}$ remains alive throughout the interval $[t, t+\rho.d]$, then every process $p_j \in \rho.D$ receives fragment $\rho_{k,\ell}$ by time $t+3dline/4-(t \mod dline)$, with high probability. That is, for any constant c, it will succeed with probability at least $1-2/n^c$.*

Proof: By Lemma 8, we know that if there is at least one process alive in group k of partition ℓ, then every active process in $P_{k,\ell}$ has received the rumor fragment with probability at least $1-1/n^c$. It remains to show that during the following block of rounds, for every process $p_j \in \rho.D$, at least one process from $P_{k,\ell}$ sends the rumor fragment to p_j. Fix some process $p_j \in \rho.D$ for the remainder of the proof. Without loss of generality, we focus on processes in $P_{k,\ell}$; the case for the other group is symmetric. We now examine the behavior of the GroupDistribution service.

Define $hitProcs_r = \{p_q \in [n] : \langle p_q, \cdot\rangle \in hitSet_r\}$ to be the set of processes that have already been successfully sent the fragment. We will consider the set $[n] \setminus hitProcs_r$, i.e., the set of processes that have *not* yet been hit and show that this decreases.

Let $Z = n^{48/\sqrt{dline}}$. We argue that in each pair of iterations of the GroupDistribution[ℓ] service, one of the following two events occurs:

1. At least a $(1-1/Z)$ fraction of processes in $P_{k,\ell}$ that were alive at the beginning of the first iteration fail by the end of the second iteration.

2. For every process p_r active throughout both iterations, the set of processes $[n] \setminus \mathit{hitProcs}_r$ decreases by a factor of Z by the end of the second iteration, with high probability.

From this claim, we conclude that within $2\log_Z n$ pairs of iterations, either every process in $P_{k,\ell}$ fails, or every process has been added to $\mathit{hitProcs}$. Since a process is only added to $\mathit{hitProcs}$ after it has been send all the available rumor fragments in $P_{k,\ell}$, and since we have assumed that at least one process in $P_{k,\ell}$ remains alive throughout the block, we conclude that by the end of $4\log_Z n \leq \sqrt{\mathit{dline}}/8$ iterations, every process has been sent all of its rumor fragments.

We now proceed to prove that one of the two above events occurs. Fix a pair of iterations, let A be the set of processes in $P_{k,\ell}$ still active at the beginning of the first iteration, and let H be the set of processes in $[n] \setminus \mathit{hitProcs}_r$ at the beginning of the first iteration, for the process p_r with the largest set $\mathit{hitProcs}_r$ at the beginning of the first iteration, where p_r is active through both iterations. Assume that event (1) does not occur, i.e., there are at least $|A|/Z$ processes still active in $P_{k,\ell}$ at the end of the second iteration.

In the second iteration, each process in A has an estimate of the size of A that is at most $|A|$. Recall that every process in A sends at least $\Theta((n/|A|) \cdot Z\log n)$ messages. Fix the constant in the asymptotic notation to be at least $(c+3)$. Thus, every process in A sends at least $(c+3)(n/|A|) \cdot Z\log n$ messages. Moreover, during the first iteration, processes share their hitSets, and hence in the second iteration, messages are only sent to processes that were not in hitSet_r.

For a given subset $A' \subseteq A$ of $|A|/Z$ processes, for a given subset $H' \subseteq H$ of $|H|/Z$ processes, we calculate the probability that no process in A' sends a message to some process in H':

$$(1-1/Z)^{(c+3)(n/|A|)(|A|/Z)\cdot Z\log n} \leq (1/e)^{(c+3)(n/Z)\log n}$$

There are at most $\binom{|A|}{|A|/Z} \leq e^{(|A|/Z)\log|A|}$ possible subsets A', and at most $\binom{|H|}{|H|/Z} \leq e^{(|H|/Z)\log|H|}$ subsets H'. Thus, by a union bound over all possible subsets, the probability that *any* subset of $|A|/Z$ processes does not hit some subset of $|H|/Z$ processes is at most:

$$\frac{e^{(|A|/Z)\log|A|+(|B|/Z)\log|B|}}{e^{(c+3)(n/|A|)\log n}} \leq \frac{e^{2(n/Z)\log n}}{e^{(c+3)(n/Z)\log n}} \leq e^{-(c+1)(n/Z)\log n}$$

By a union bound over all the $\sqrt{\mathit{dline}}/8$ iterations, we conclude that with probability at least $1-1/n^c$, the set H decreases by a factor of Z in each such pair of iterations, concluding our proof (via a union bound of the $1/n^c$ probability of error by the proxy service and the $1/n^c$ probability of error here). □

We can now show the final piece: that each admissible rumor is successfully delivered to its destination set, and that confirmation is delivered by the AllGossip service to everyone.

Lemma 10 *Given admissible rumor ρ, injected at time t at process p_i: if p_i does not fail by time $t+\rho.d$, then by round $t+\rho.d-1$, process p_i receives confirmation that rumor ρ was delivered, with high probability, i.e., with probability at least $1-2/n^c$ for any constant c.*

Proof: By Lemma 5, we know that if rumor ρ has even one admissible destination $\neq p_i$, then there is some partition ℓ where there is at least one process in $P_{0,\ell}$ and one process in $P_{1,\ell}$ that does not fail in $[t, t+\rho.d]$. By Lemma 9, we know that by time $t + 3\mathit{dline}/4 - (t \mod \mathit{dline})$, with high probability (i.e., $1-2/n^c$), rumor fragments $\rho_{0,\ell}$ and $\rho_{1,\ell}$ have been delivered to every destination in $\rho.D$ by the GroupDistribution$[\ell]$ service, allowing the destination processes to reconstruct ρ.

Once this (probabilistic) event occurs, confirmation always takes place: Since at least one process p_0 in $P_{0,\ell}$ and one process p_1 in $P_{1,\ell}$ does not fail during $[t, t+\rho.d]$, we conclude that p_0 and p_1 complete the block in which the rumor fragments for ρ are delivered to their destinations. At the end of the last round of the block, processes p_0 and p_1 inject sanitized versions of the $\mathit{hitSets}$ as rumors into the AllGossip service with deadline $\mathit{dline}/4-1$, thus ensuring that process p_i receives this information no later than round $t+\rho.d-1$. Process p_i then marks rumor ρ confirmed. □

Using Lemmas 5–10, we prove the final complexity result.

Theorem 11 (Per-round message complexity) *The per-round message complexity of* CONGOS *is:*

$$O\left((n^{1+48/\sqrt{\mathit{dmin}}} + n^{1+6/\sqrt[6]{\mathit{dmin}}})\,\mathit{polylog}\ n\right)$$

where $dmin$ is the minimum deadline of any rumor in the system in that round. This holds with high probability, i.e., with probability at least $1 - 2/n^{c-2}$ for any constant c.

Proof: From Lemma 10 we have that for a given rumor ρ injected at process p_i, with probability at least $1 - 2/n^c$, the rumor is confirmed prior to the deadline expiring. Since each process is injected at most one rumor per round (hence there can be $O(n\,\text{polylog}\ n)$ active rumors in the system at any given time, over all instances of continuous gossip and all deadlines), by a union bound, no source process sends any messages directly to the destinations with probability at least $1 - 2/n^{c-2}$.

For a deadline of $dline$, Lemma 7 shows that the per-round message complexity for each instance of $\mathsf{Proxy}[\ell]$ and $\mathsf{GroupDistribution}[\ell]$ is $O(n^{1+48/\sqrt{dline}} \log n)$, leading to a per-round message complexity of $O(n^{1+48/\sqrt{dline}} \log^2 n)$. Each instance of continuous gossip, invoked with rumors at least $\sqrt{dline}$, has message complexity $O(n^{1+6/\sqrt[6]{dline}}\text{polylog}\ n)$. There are $\log n + 1$ such instances of continuous gossip.

Thus the total per round message complexity is:

$$\sum_{dline=dmin}^{\Theta(\log^6(n))} O\left((n^{1+48/\sqrt{dline}} + n^{1+6/\sqrt[6]{dline}})\text{polylog}\ n\right).$$

This summation is dominated by the first term, yielding the desired result. $\square$

6 Gossiping in the Presence of Collusion

In this section we extend our investigation of the confidential gossip problem by additionally assuming that processes outside of a rumor's destination set may collude in an attempt to learn the rumor.

More formally, given a rumor ρ injected in the system at a process p_i, we denote by C_ρ the collusion set of ρ. In particular, C_ρ may contain any process $q \notin \rho.D \cup \{p_i\}$. We assign adversary $CRRI$ with the additional task of "selecting" the colluding processes in an adaptive way during the execution. We will be referring as $CRRI(\tau)$ the subset of the adversarial patterns of $CRRI$ for which $|C_\rho| \leq \tau$ for any rumor ρ injected in the system. Finally, we will be calling *τ-collusion-tolerant* an algorithm that it is designed to solve confidential gossip under adversary $CRRI(\tau)$.

6.1 Lower Bound

We prove that a class of algorithms generalizing our algorithm CONGOS suffers from collusion, in terms of message complexity. We say that a gossip algorithm is *partition-based* if it allows only two operations tampering with the content of the rumors: splitting, which allows to split a given initial rumor into disjoint smaller fragments, and merging, which allows to merge given fragments of the *same* rumor into a larger fragment of this rumor[1]. Otherwise, the protocol must treat the rumor (and its fragments) as nonmalleable tokens.

The effect of collusion, as demonstrated by the following theorem, might be significant (especially for large number of colluders), even against an *oblivious adversary* that can only arrange the rumors destination sets and identify the colluding processes prior to the start of the computation.

Theorem 12 *For any constant $\varepsilon > 0$, every randomized, τ-collusion-tolerant, partition-based algorithm solving confidential gossip has a maximum per-round message-complexity of at least $\Omega(\min\{n\tau, n^{(3/2)-\varepsilon}\}/dmax)$, with probability 1, against an oblivious adversary, where $dmax$ is the longest deadline of the injected rumors.*

Proof: We assume the same initial setting of parameters and rumors, including their destination sets and deadlines, as considered in the proof of Theorem 1, which are as follows. We may assume that n is sufficiently large (in fact, $n \geq 8$ is sufficient). Let c be a constant and x be a parameter, to be specified in the same way as in the proof of Theorem 1, depending on ε. Suppose that only rumors with uniform deadlines $dmax$ are injected, all at the same time. Moreover, assume that each process is injected one rumor with the same destination set as in the proof of Theorem 1.

Now consider a single execution of a given algorithm in this setting. Let a *rumor interval* be a set of rumor fragments such that any set of fragments sufficient to reconstruct the rumor includes some fragment from the rumor

[1]Notice that this does not allow other algebraic manipulation of the rumor, as in "network coding" techniques.

interval. (Informally a rumor interval corresponds to a sub-sequence of rumor bit-string representation and to all rumor fragments that contain this sub-sequence.) Two cases are possible:
Case 1: More than half of the rumors satisfy the following property each: there is a rumor interval such that none of its contained fragments is ever transmitted to a process outside the destination set.

It follows that for such a rumor interval, each destination process receives some rumor fragment in this rumor interval directly from the rumor's source (the process that the rumor was injected at) or relayed entirely through the processes in the destination set. Therefore, the messages carrying rumor fragments in this rumor interval altogether suffer from the same constraints as it would an original rumor propagated within its destination sets only. By Theorem 1, the number of such messages is proportional to the size of the destination set, for the considered setting of destination sets, and since there are more than $n/2$ such rumors, we get the lower bound $\Omega(n^{(3/2)-\varepsilon})$ on the total number of such messages in the considered period of length $dmax$. Hence, the per round message complexity in this case is $\Omega(n^{(3/2)-\varepsilon}/dmax)$.
Case 2: At least half of the rumors satisfy the following property each: fragments of the rumor transmitted outside the destination set cover the whole original rumor.

In this case for each such rumor there are at least $\tau + 1$ processes outside its destination set that receive a fragment of the rumor directly from some processes in the destination set or the rumor's source (the process that the rumor was injected at); otherwise at most τ such outside processes could collude and get fragments covering the whole rumor, thus violating the definition of confidentiality (which must hold for every execution). We call such at least $\tau + 1$ fragments *border fragments*. Therefore there are at least $\tau + 1$ point-to-point messages sent from some processes in the destination set or the rumor's source to the considered at least $\tau + 1$ outside processes. Call these messages *border messages*. It follows that there are at least $(\tau + 1)n/2$ copies of border fragments sent via border messages. Recall the property of the considered configuration of destination sets as proved in Theorem 1: each process is in at most c destination sets, where c is a constant. It follows that a process sends at most c border fragments per border message, which gives at least $\frac{(\tau+1)n/2}{c} = \Omega(n\tau)$ border messages. Hence the per-round message complexity in this case is $\Omega(n\tau/dmax)$.

Observe that these two cases are complementary. Indeed, notice that if fragments of a rumor transmitted outside the destination set *do not* cover the whole original rumor (there are at most half of such rumors in case 2), then some of its rumor intervals consist of fragments not transmitted outside the destination set, e.g., containing the part of the rumor which is not covered by the fragments transmitted outside the destination set. These rumors satisfy the requirement in case 1. Hence, if there are at most half of such rumors, then case 2 is satisfied, otherwise case 1 is fulfilled.

In each case, the message complexity is $\Omega(\min\{n\tau, n^{(3/2)-\varepsilon}\}/dmax)$. This bound holds for any execution of the algorithm. The adversary is oblivious, as it uses the same setting as in the proof of Theorem 1 and does not need to specify colluding processes. To justify the latter, observe that both cases hold regardless of the choice of colluding processes, and the only place the adversary threads the algorithm by possibility of collusion is in Case 2 when it enforces at least $\tau + 1$ border messages; but for this it does not need to specify online the set of colluding processes, and the argument says only that if the algorithm broke it, the adversary could choose a set of colluders violating confidentiality. This completes the proof of the theorem. $\square$

6.2 Collusion-tolerant CONGOS

We modify algorithm CONGOS in the following way. Instead of $\log n$ partitions used in algorithm CONGOS, we use $c\tau \log n$ partitions given as a part of the input of the algorithm, for an appropriate choice of constant c. Each partition contains $\tau + 1$ groups, instead of the originally used 2 groups. For this purpose, rumors are now divided into $\tau + 1$ fragments. If we view $\tau = 1$ as a collusion of a process with itself, then the original algorithm CONGOS can be viewed as 1-collusion-tolerant confidential gossip algorithm.

The set of $c\tau \log n$ partitions needs to satisfy the following properties, for appropriate choice of constants c and c':

- **Partition-Property 1:** In each partition, each group contains at least one process.
- **Partition-Property 2:** For every set S of at least $2c'\tau \log n$ processes, there exists a partition such that every group in the partition contains at least one process in S.

The first property ensures well-formedness, i.e., that the partition is a proper division of the processes into non-empty groups. The second property ensures good performance: as long as there are $\Omega(\tau \log n)$ processes alive, then one partition has live processes in every group and hence can be used to distribute the rumor fragments. We now argue that there exists a good set of partitions that meets these requirements:

Lemma 13 *If $\tau < n/\log^2 n$, then there is a set of $c\tau \log n$ partitions satisfying the above conditions, for some constants $c, c' > 0$.*

Proof: We proceed via the probabilistic method: first, we randomly select the groups for each partition, and then we show that with some positive probability, the two properties are satisfied. We begin by assuming that for each partition, each process is independently assigned, uniformly at random, to one of the $\tau + 1$ groups in that partition.

We begin by examining the first required property. For each group g, the probability that a process chooses group g is $1/(\tau+1)$. Thus the probability that a given group is not chosen by any process is at most $(1-1/(\tau+1))^n \leq 2^{-\log^2 n}$. In total, there are $(\tau + 1) \cdot (c\tau \log n) \leq 2cn^2$ groups, and thus by a union bound, the probability that any group is not chosen by some process is at most $2^{-(\log^2 n - \log(2cn^2))} < 1/2$, for sufficiently large n.

We now examine the second required property. We fix some set S of size $2c'\tau \log n$. For a given partition, for a given group in that partition, the probability that no process in set S is assigned to that partition is at most $(1 - 1/(\tau + 1))^{2c'\tau \log n} \leq 1/n^{c'}$. Thus, by a union bound, the probability that any group in the partition does not contain a process in S is at most $(\tau + 1)/n^{c'} \leq 1/n^{c'-1}$.

Since each partition is selected independently, the probability that for every one of the $c\tau \log n$ partitions, at least one group is empty is at most $(1/n^{c'-1})^{c\tau \log n} \leq n^{-c \cdot c'\tau \log n/2}$.

Now, consider all $\binom{n}{2c'\tau \log n}$ choices of the set S. There are at most $n^{2c'\tau \log n}$ such sets S. Taking a union bound over all the sets S, the probability that there exists a set S for which every partition has at least one empty group is at most $n^{-(c \cdot c'\tau \log n/2 - 2c'\tau \log n)} < 1/2$, for appropriately large n and choice of c and c'.

Thus, the probability that the selected partition does not satisfy the two requisite properties is smaller than 1, and hence, by the probabilistic method, a partition satisfying the two desired properties exists. □

We leave the polynomial time construction of partitions satisfying the required conditions as future work.

Overview of collusion-tolerant CONGOS**:** In a nutshell, the modified version of algorithm CONGOS operates as follows for a newly injected rumor ρ at a process p_i. If $\tau \geq n/\log^2 n$ then all rumors are sent directly to their destinations. Otherwise, procedure ConfidentialGossip is called in which the rumor, for each different partition ℓ, is divided into the fragments $\rho_{0,\ell}, \ldots, \rho_{\tau,\ell}$ such that all fragments (from the same partition) are needed in order for ρ to be re-assembled. A way to do this is as follows: Let $\rho_{0,\ell}, \ldots, \rho_{\tau-1,\ell}$ be different random binary strings and set $\rho_{\tau,\ell} = (\rho \textbf{ xor } \rho_{0,\ell} \textbf{ xor } \ldots \textbf{ xor } \rho_{\tau-1,\ell})$. Then ρ can be computed when all $\tau + 1$ fragments are received. Note that this scheme makes the algorithm partioned-based.

Say that in partition ℓ, process p_i belongs in group x. Then it injects fragment $\rho_{x,\ell}$ in GroupGossip[ℓ] and all other fragments into Proxy[ℓ]. Via procedure GroupGossip[ℓ], the fragment $\rho_{x,\ell}$ is gossiped in the members of group x and via Proxy[ℓ] each other fragment is gossiped into every other corresponding group (such that every other group learns a different fragment of the rumor). Then procedure GroupDistribution[ℓ] is called so that the processes in each group collaborate in sending their corresponding fragment of the rumor only to the processes of the rumor's destination set. These processes receive all fragments and hence can reassemble the rumor. Lemma 13 assures the existence of at least one partition ℓ in which all admissible rumors are received by the live processes of the rumor's destination set. Detailed outlines of the modified procedures are given below; the differences from algorithm CONGOS are included in a box and annotated with **boldface text**. In particular, Figure 5 outlines ConfidentialGossip, Figure 6 outlines Proxy and Figure 7 outlines GroupDistribution.

6.3 Analysis

We now prove that the collusion-tolerant algorithm still satisfies the desired correctness and message complexity requirements. The structure of the proof closely follows the analysis in Section 5, modified to deal with the larger number of partitions.

The first lemma states that confidentiality is not violated. This is almost identical to Lemma 3, looking at $\tau + 1$ groups instead of 2 groups.

Lemma 14 (Confidentiality) *In any execution the algorithm, and for any rumor ρ, if $q \notin \rho.D$, then at no point during the execution q learns $\rho.z$. This occurs with probability 1.*

Proof: Consider a rumor ρ injected at a processes p at round t of an execution of algorithm CONGOS. Also consider a process $q \notin \rho.D$. We will show by investigating the flow of the algorithm that at no point will process q learn $\rho.z$ or be able to re-construct it from fragments.

Once ρ is injected at process p, based on the description of the algorithm at process p, and focusing on a partition ℓ, the rumor is split into $\rho_{0,\ell}, \rho_{1,\ell}, \ldots, \rho_{\tau,\ell}$. Each rumor fragment is distributed by the Proxy service and the GroupGossip service only to its proper group, with the filter enforcing this restriction. Similarly, the GroupDistribution service sends fragments only to processes in the proper destination set. Finally, the last step in the algorithm sends the rumor directly to the destination set (as a fallback mechanism), i.e., not to any other process.

Thus no process that is not in the destination set learns more than one fragment, i.e., the fragment for the group it is assigned to. Since there are at most τ colluders and $\tau + 1$ fragments, we conclude that the colluders learn at most τ fragments and hence cannot reconstruct the rumor. □

We now show that the algorithm delivers admissible rumors before they expire. The correctness in this case follows by similar arguments as for algorithm CONGOS, since the partitions used for the modified algorithm satisfy the same conditions explored in the analysis as the partitions used in the original algorithm CONGOS.

Lemma 15 (Correctness wrt QoD) *In any execution of the algorithm, and for any rumor ρ injected at a process p at round t of the execution, if p and $q \in \rho.D$ are continuously alive for the lifetime of the rumor, then q learns ρ by round $t + \rho.D$. This occurs with probability 1.*

Proof: Fix $\rho = \langle z, d, D\rangle$ injected at process p at round t, and consider process $q \in \rho.D$. Both p and q are continuously alive for the rumors lifetime. Hence ρ is admissible for q. We show that q will learn, with probability 1, rumor ρ by time $t + \rho.d$.

From the last two bullets in the outline of the algorithm (Figure 6.2), we have that if the deadline of rumor ρ is about to expire, and there is no confirmation that ρ has been delivered, then process p sends ρ directly to every process in $\rho.D$, including q. Hence, what remains to be proved is that if p receives confirmation that ρ was delivered before the rumor has expired, then q has indeed learned ρ (i.e., it has learned all the fragments of ρ is some partition ℓ).

Process p will confirm that ρ has been delivered if a message from AllGossip confirms that, for some partition ℓ, all the fragments of a rumor ρ have been sent to every destination in $\rho.D$ (including q). Thus, it remains only to prove that the GroupDistribution service satisfies the [GD:CONFIRM] property, which follows from these observations:

- No process will include q in its $hitSet$ if it has not sent its partial rumor to q in the GroupDistribution service of some partition ℓ.
- Say w is a process that has included q in its $hitSet$. The dissemination of $hitSet$ takes place after rumor fragments to the processes being "hit" are sent. If w would fail prior to hitting q, then its $hitSet$ containing q would not have been disseminated through the AllGossip service. Hence, when p receives a $hitSet$ (from AllGossip) containing q, it is the case that q was indeed hit.
- Messages sent from non-faulty processes to non-faulty processes are not lost (unless they go through the filter, which is not the case here).

We therefore conclude that the initiator of a rumor only gets a confirmation of the rumor when it was delivered; this completes the proof. □

We now give the main result of this section.

Theorem 16 *Collusion-tolerant* CONGOS *solves the confidential gossip problem under adversary $CRRI(\tau)$ with per-round message complexity of*

$$O\left(\left(n^{1+48/\sqrt{dmin}} + n^{1+6/\sqrt[6]{dmin}}\right)\tau^2 \textit{polylog}\ n\right) ,$$

where $dmin$ is the minimum deadline of any rumor in the system in that round; this holds with high probability.

Proof: If $\tau \geq n/\log^2 n$ then the correctness argument is straightforward (as all rumors are straightaway sent to their destinations) and the per-round message complexity is $O(n^2)$, as this is the upper bound on the number of links to be used in a round. We have:

$$O(n^2) \leq O(n \cdot (n/\log^2 n)^2) \leq O\left(\left(n^{1+48/\sqrt{dmin}} + n^{1+6/\sqrt[6]{dmin}}\right)\tau^2 \text{polylog}\ n\right) .$$

In the remainder, we consider the case $\tau < n/\log^2 n$.

Lemma 15 shows that every admissible rumor is delivered by the deadline (as required), and Lemma 14 shows that confidentiality is maintained.

For the message complexity, ignoring the fallback transmission in the last step of the algorithm: for every group and for every partition, the Proxy and the GroupDistribution services send $O(n^{1+48/\sqrt{dmin}})$ messages and the GroupGossip service sends $O(n^{1+6/\sqrt[6]{dmin}})$ messages. Since there are $\tau+1$ groups and $O(\tau \log n)$ partitions, the total per round message complexity is as claimed.

It remains to show that the fallback procedure does not induce too many messages. There are now two cases to consider. Assume that there is *not* some set of at least $2c'\tau \log n$ processes that are alive throughout the relevant period. Then in the final round of the protocol, there are at most $2c'\tau \log n$ processes that are still alive (and have not crashed at some point), and each sends at most n messages (for a fallback transmission). Hence the per-round message complexity is at most $O(n\tau \log n)$.

Now consider the case where there are at least $2c'\tau \log n$ processes that do not fail throughout the relevant time interval. Then, by Partition-Property 2, we know that there exists a partition such that every group in the partition contains at least one process that remains alive throughout. We fix this partition ℓ.

It remains to show that each rumor is delivered by partition ℓ with high probability before the fallback transmission. Since there are at most $O(n \text{polylog } n)$ active rumors at any time, this is sufficient to ensure (by a union bound) that with high probability all the active rumors are delivered before the fallback transmission, and hence in any given round, with high probability there are no fallback transmissions.

Recall that Lemmas 8 and 9 were stated and proven in terms of the behavior of the Proxy and GroupDistribution services for specific groups, with no dependence on the partition. This behavior is unchanged, and hence the lemmas hold unchanged.

We can now show that given an admissible rumor ρ, injected at time t at process p_i, then by round $t + \rho.d - 1$, process p_i receives confirmation that rumor ρ was delivered, with high probability, i.e., with probability at least $1 - 2/n^c$ for any constant c. Specifically, for each of the groups in partition ℓ, we know that one process remains alive (by the choice of ℓ). And so by Lemma 9, we know that by time $t + 3dline/4 - (t \mod dline)$, with high probability (i.e., $1 - 2/n^c$), rumor fragments $\rho_{0,\ell}, \rho_{1,\ell}, \ldots, \rho_{\tau,\ell}$ have been delivered to every destination in $\rho.D$ by the GroupDistribution$[\ell]$ service, allowing the destination processes to reconstruct ρ.

Once this (probabilistic) event occurs, confirmation always takes place: Since for each k, at least one process p_k in $P_{k,\ell}$ does not fail during $[t, t+\rho.d]$, we conclude that each p_k completes the block in which the rumor fragment $\rho_{k,\ell}$ is sent to its destinations. At the end of the last round of the block, each process p_k injects sanitized versions of the $hitSets$ as rumors into the AllGossip service with deadline $dline/4 - 1$, thus ensuring that process p_i receives this information no later than round $t + \rho.d - 1$. Process p_i then marks rumor ρ as confirmed. Thus in this case, no fallback message is sent, and the per round message complexity is as required. $\square$

Observe from Theorem 16 that when $dmin = \Theta(\log^6 n)$, the per-round message complexity is $O(n\tau^2 \text{polylog } n)$. When contrasted with Theorem 12 it follows that for $\tau < n^{1/4}$, the per-round message complexity is within a factor of $\tau \text{polylog } n$ of the lower bound. (For $\tau = O(\text{polylog } n)$ the algorithm is optimal within log factors.)

7 Discussion

In this paper we have considered the problem of *confidential* gossip, where each rumor is learned only by processes in the rumor's specified destination set. Assuming an adaptive and omniscient adversary that dynamically and continuously injects rumors into the system and causes process crashes and restarts, we have designed an efficient (w.r.t. per-round message complexity) algorithm which we call algorithm CONGOS. As an alternative to cryptographic schemes, which can be expensive in such a dynamic environment, the algorithm deploys a simple rumor splitting technique that enables an efficient "all-process" collaboration while guaranteeing confidentiality. For this purpose, the algorithm combines, in a non-trivial way, a black-box efficient non-confidential continuous gossip service with other auxiliary services (namely, Filter, Proxy, GroupDistribution). While we have focused on continuous gossip, we believe that the same techniques apply to other gossip variants (e.g., single-instance gossip, etc.).

Open questions: collusion. We have also discussed the problem of collusion, and shown how to tolerate a moderate amount of collusion at a limited cost. An interesting open question is whether we can tolerate higher levels of collusion if the adversary is oblivious, or if we allow some small probabilistic violation of confidentiality.

In addition, as currently presented, the algorithm guarantees the confidentiality of rumors, but various other meta-data is released. For example, processes learn of the existence of rumors, roughly how many rumors are active, the source of each rumor, a sequence number of each rumor, and the set of destinations for each rumor. Some of this information can be readily hidden. For example, the sequence number can be replaced with a pseudorandom identifier. Other information appears more difficult to hide, for example, the proxies learn precisely who is requesting that they act as a proxy, and this seems, to some extent, unavoidable.

The destination set associated with each rumor can be hidden, without increasing the overall message complexity, but at the cost of increasing the message size (significantly). When a rumor ρ is injected at process p_i, the source creates n new rumors, each with a single process in its destination set. For every process in $\rho.D$, the new rumor contains a copy of the injected rumor's content. For the remaining new rumors, the contents of the new rumor are chosen at random. The source then proceeds to distribute this entire collection of rumors. Only the processes in the destination set can determine whether a rumor contains real content or simply a random string, and hence processes cannot determine the real destination set.

Similarly, the very existence of rumors can be hidden by the continual injection of fake content-free rumors, at the cost of wasted messages. In this way, a process cannot determine how many real rumors are currently active.

Open questions: message and communication complexities. One natural question is whether the message complexity presented here is optimal. It seems likely that at least some improvement is possible. Specifically, the message complexity depends (roughly) on $n^{1+1/d^{\ell}}$, for some constant ℓ. Ideally, we would only need $O(n^{1+1/d})$ messages. In part, the additional cost comes from the Continuous Gossip service, which already requires $\ell = 3$. To do better, we need a better basic contiuous gossip protocol. In part, the aditional cost comes from using the continuous gossip protocol as a black box: currently, the Proxy and GroupDistribution service each repeatedly initiate gossip requests, each with a deadline of $\sqrt{dline/4}$. It seems plausible that by overlapping some of the work in the Proxy and GroupDistribution services with the continuous gossip protocol, we could improve the message complexity.

A second natural question is whether we could improve the communication complexity, i.e., the total number of bits transmitted in each round (rather than just the number of messages). For gossip protocols, however, the communication complexity depends significantly on the precise application. If no aggregation of rumors is possible, i.e., if sending two rumors costs twice as much as sending one rumor, then the communication complexity could necessarily be at least $\Omega(n^2/d)$ rumors per round: every process may be required to send/receive n different rumors over d rounds. In such a case, gossip would have little benefit. Another issue is the cost in terms of control bits. Our solution adds a fairly large number of control bits to ensure that rumors are delivered. Perhaps allowing some probability on the timely delivery of rumors (i.e., some rumors could miss the deadline with low probability), the communication complexity could be reduced. This is an interesting direction for future investigation.

Open questions: malicious users. Finally, an interesting open question is whether we can tolerate truly malicious processes, i.e., those that do not follow the protocol. In fact, we believe that the approach for tolerating collusion may be extended to deal with malicious processes, if the adversary is oblivious. In that case, we can tolerate some groups misbehaving and failing to deliver their message fragments.

References

[1] S. Baehni, P.T. Eugster, and R. Guerraoui. Data-Aware Multicast. *In DSN 2004,* pages 233–242.

[2] A.J. Ballardie. *A New Approach to Multicast Communication in a Datagram Network*, Ph.D. Thesis, University College London, 1995.

[3] A. Beimel, K. Nissim, and E. Omri. Distributed Private Data Analysis. In *CRYPTO 2008*, pages 451–468.

[4] J. Brickell and V. Shmatikov. Privacy-preserving Graph Algorithms in the Semi-honest Model. In *ASIACRYPT 2005*, pages 236–252.

[5] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast Security: A Taxonomy and Some Efficient Constructions. *In INFOCOM 1999,* pages 708–716.

[6] B.S. Chlebus and D.R. Kowalski. Time and Communication Efficient Consensus for Crash Failures. *In DISC 2006,* pages 314–328.

[7] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. SpiderCast: A Scalable Interest-Aware Overlay for Topic-Based Pub/Sub Communication. *In DEBS 2007,* pages 14–25.

[8] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing Scalable Overlays for Pub/Sub with Many Topics. *In PODC 2007,* pages 109–118.

[9] G. Delposte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert. Secretive Birds: Privacy in Population Protocols. *In OPODIS 2007*, pages 329–342.

[10] B. Doerr, T. Friedrich, and T. Sauerwald. Quasirandom Rumor Spreading: Expanders, Push vs Pull, and Robustness. *In ICALP 2009,* pages 366–377.

[11] Y. Fernandess and D. Malkhi. On Spreading Recommendations via Social Gossip. *In SPAA 2008,* pages 91–97.

[12] A. Fiat and M. Naor. Broadcast Encryption. *In CRYPTO 1993,* pages 480–491.

[13] Ch. Georgiou, S. Gilbert, and D.R. Kowalski. Meeting the Deadline: On the Complexity of Fault-Tolerant Continuous Gossip. *Distributed Computing,* 24(5):223–244, 2011. (A preliminary version appears *in PODC 2010,* pages 247–256.)

[14] Ch. Georgiou, S. Gilbert, R. Guerraoui, and D.R. Kowalski. Asynchronous Gossip. *Journal of the ACM,* 60(2), article 11, 2013.

[15] O. Goldreich. *Foundations of Cryptography: Volume II (Basic Applications)*. Cambridge University Press, 2004.

[16] I. Gupta, A.M. Kermarrec, and A.J. Ganesh. Efficient Epidemic-style Protocols for Reliable and Scalable Multicast. *In SRDS 2002,* pages 180–189.

[17] Havard D. Johansen, Andre Allavena, and Robbert van Renesse. Fireflies: Scalable Support for Intrusion-tolerant Network Overlays. *In EuroSys 2006,* pages 3–13.

[18] J. Hromkovic, R. Klasing, A. Pelc, P. Ruzika, and W. Unger. *Dissemination of Information in Communication Networks: Broadcasting, Gossiping, Leader Election, and Fault-Tolerance,* Springer-Verlag, 2005.

[19] R. Karp, C. Schindelhauer, S. Shenker, B. Vocking. Randomized Rumor Spreading. *In FOCS 2000,* pages 565–574.

[20] D. Kempe, J. Kleinberg, and A. Demers. Spatial Gossip and Resource Location Protocols. *Journal of the ACM,* 51:943–967, 2004.

[21] A. Kermarrec, L. Massoulie, A. Ganesh. Probabilistic Reliable Dissemination in Large-scale Systems. *IEEE Transactions on Parallel and Distributed Systems,* 14(3):248–258, 2003.

[22] L. Kissner and D. Song. Privacy-preserving Set Operations. In *CRYPTO 2005*, pages 241–257.

[23] D. R. Kowalski and M. Strojnowski. On the Communication Surplus Incurred by Faulty Processors. *In DISC 2007,* pages 328–342.

[24] Y. Lindell and B. Pinkas. Privacy Preserving Data Mining. *J. Cryptology*, 15(3):177–206, 2002.

[25] D. Malkhi, Y. Mansour, and M.K. Reiter. Diffusion Without False Rumors: On Propagating Updates in a Byzantine Environment. *Theoretical Computer Science,* 299:289–306, 2003.

[26] D. Micciancio and S. Panjwani. Corrupting One Vs. Corrupting Many: The Case of Broadcast and Multicast Encryption. *In ICALP 2006,* pages 70–82.

[27] Y.M. Minsky and F.B. Schneider. Tolerating Malicious Gossip. *Distributed Computing,* 16:49–68, 2003.

[28] S. Mittra. Iolus: A Framework for Scalable Secure Multicasting. *SIGCOMM Comput. Commun. Rev.,* 27(4):277–288, 1997.

[29] Multicast Security. `http://datatracker.ietf.org/wg/msec/`

[30] M. Onus and A.W. Richa. Minimum Maximum Degree Pub/Sub Overlay Network Design. *In INFOCOM 2009,* pages 882–890.

[31] J. Pang and C. Zhang. How to Work with Honest but Curious Judges? *In Proc. 7th International Workshop on Security Issues in Concurrency*, pages 31–45, 2009.

[32] S. Panjwani. Tackling Adaptive Corruptions in Multicast Encryption Protocols. *In TCC 2007*, pages 21–40.

[33] A. Pelc. Fault-tolerant Broadcasting and Gossiping in Communication Networks. *Networks,* 28: 143–156, 1996.

[34] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[35] A.T. Sherman and D.A. McGrew. Key Establishment in Large Dynamic Groups Using One-Way Function Trees. *IEEE Transactions on Software Engineering,* 29(5):444–458, 2003.

[36] D.R. Stinson. *Cryptography: Theory and Practice,* CRC Press, 3rd edition, 2005.

[37] C.K.Wong, M. Gouda, and S. Lam. Secure Group Communications Using Key Graphs. *IEEE/ACM Transactions on Networking,* 8(1):16–30, 2000.

[38] G. Xu, G. Amariucai, and Y. Guan. Delegation of Computation with Verification Outsourcing: Curious Verifiers. *In PODC 2013*, pages 393–402.

[39] A.C. Yao. Protocols for Secure Computations. *In FOCS 1982*, pages 160–164.

Appendix: Detailed Pseudocode of Algorithm CONGOS

In this section we present the detail pseudocode for algorithm CONGOS. In particular we give a separate pseudocode for each service of the algorithm at a process i. Figure 8 describes the operation of the ConfidentialGossip service which is the main control of the algorithm. It basically coordinates the other services that run in parallel: Proxy (Figure 9), GroupDistribution (Figure 10) and Filter (Figure 11); the code for these services is given for a certain partition ℓ.

As mentioned before, a non-confidential Continuous Gossip service (protocol) is assumed (e.g., the one presented in [13]), which guarantees rumor dissemination within a specified deadline. GroupGossip[ℓ] refers to an instance of the service for partition ℓ that is filtered (from the Filter[ℓ] service) in restricting rumor disseminations in certain process groups. AllGroup refers to an instance of the gossip service for partition ℓ that it is not filtered (hence rumors are sent to all processes in $[n]$).

We now explain the operation of the function **random-split** used in line 14 of the ConfidentialGossip service (Figure 8). Recall that a rumor r is a tuple (z, d, D) where $r.z$ is the data to be disseminated, $r.d$ the deadline and $r.D$ the rumor destination set (only processes in the set must learn r). Once the function is executed, rumor r_0 has as $r_0.z$ the tuple $\langle z_0, r.D, counter \rangle$, $r_0.d = \sqrt{dline/6}$, and $r_0.D = [n]$. Rumor r_1 is similar ($r_1.z$ contains z_1 and not z_0). As explained before, z_0 is a random binary string and $z_1 = z\mathbf{XOR}z_0$. Note that r is split differently for each partition ℓ. The function **merge** in line 32 of the ConfidentialGossip service works reversely to reconstruct a rumor from its two fragments. We will be using the notation $rumor.z.D$ to denote the original destination set of a rumor (which $rumor$ is a fragment of it) and $rumor.z.cnt$ the value of the $counter$ that the rumor was assigned upon injection into the source process. See for example lines 27 and 28 of the GroupDistribution service (Figure 10).

Throughout the codes, we denote by R the data type which is a set representing all rumors, that is, all rumors of the form $\langle z, d, D \rangle$. We also consider $round \in \mathbb{Z}$ to be a global counter representing time (round numbers), taken from the global clock.

Outline of ConfidentialGossip **service at** p_i**:**

- Do in parallel for each $\boxed{\ell = 1, \ldots, \boldsymbol{c\tau} \log n}$:
  1. Split rumor ρ into a $\boxed{\textbf{sequence } \langle \rho_{0,\ell}, \rho_{1,\ell}, \boldsymbol{\ldots, \rho_{\tau,\ell}} \rangle}$.
  2. If p_i is in group b of partition ℓ, inject $\rho_{b,\ell}$ into GroupGossip[ℓ], and inject $\boxed{\textbf{all } \boldsymbol{\rho_{a,\ell}, a \neq b},}$ into Proxy[ℓ]. Together, these two services ensure that each rumor fragment is delivered to every non-failed process in the appropriate group of the partition.
  3. For each rumor fragment received from GroupGossip[ℓ] or Proxy[ℓ], inject the fragment into GroupDistribution[ℓ].
  4. Save every fragment received from GroupDistribution[ℓ], and reassemble and deliver rumors as fragments become available.
- Whenever a message from AllGossip confirms that, for some partition ℓ, $\boxed{\textbf{all } \boldsymbol{\tau + 1}}$ fragments of a rumor ρ, initiated at p_i, have been sent to every destination in $\rho.D$, confirm that ρ has been delivered.
- Whenever a deadline is about to expire for some rumor ρ initiated at p_i, and there is no confirmation that ρ has been delivered, send ρ directly to every process in $\rho.D$. (A simple optimization would be to only send rumors to destinations for which no confirmation was received. Since this is a low probability event, it has little impact on performance.)

Figure 5: Outline of ConfidentialGossip service at p_i – with collusion

Outline of Proxy[ℓ] **at** p_i**:**

- Time is divided into blocks of length $dline/4$.
- At the beginning of a block, i.e., in the first round of a new block, if has been alive for at least $dline/4$ rounds, then collect all the fragments that have been injected since the last block began, and if there is at least one such fragment, then set $status$ to active.
- Each block is divided into *iterations* of $\sqrt{dline}+2$ rounds. In each iteration, we maintain a set $collaborators$ of the active processes (i.e., processes with $status$ active) in the same group as p_i. Say that p_i is in group $P_{b,\ell}$. We also keep track of $failed\text{-}proxies$, i.e., those that we have already learned (in previous iterations) have failed in this block. For each iteration, repeat (as long as $status =$ active):
  - Round 1: $\boxed{\textbf{for each other group } P_{a,\ell}, a \neq b\textbf{,}}$ send every rumor fragment associated with that group (i..e, $\rho_{a,\ell}$) to $\Theta(n^{1+48/\sqrt{dline}} \log n / |collaborators|)$ processes chosen uniformly at random from group $P_{a,\ell}$, excluding processes in $failed\text{-}proxies$. (Notice that as long as the set $collaborators$ is a good estimate of the set of collaborators, this ensures a good bound on the message complexity of this step.) Every process that receives a request to be a proxy for $\boxed{\textbf{some} \text{ other group}\textbf{(s)}\ P_{a,\ell}, a \neq b}$ caches the received rumor fragments.
  - Rounds $2, \ldots, \sqrt{dline}+1$: initiate a GroupGossip[ℓ] in which processes in $P_{b,\ell}$ share the set of $failed\text{-}proxies$, as well as establish the set of $collaborators$, i.e., members of the group that still have $status$ active. Processes also share all the rumor fragments received from the other $\boxed{\text{group}\textbf{s}\ P_{a,\ell}, a \neq b}$. (The deadline for rumors in GroupGossip[ℓ] here is $\sqrt{dline}$.)
  - Round $\sqrt{dline}+2$: Any process that was asked to be a proxy for the other $\boxed{\text{group}\textbf{s}\ P_{a,\ell}, a \neq b}$ sends an acknowledgment that proxying was successful. Any process that sent a request, and does not receive an acknowledgment, adds the non-acknowledging processes to the set of $failed\text{-}proxies$.
- Upon recovering from a failure, obtain the round number from the global clock, set $status =$ idle and wait until a new block begins.

Figure 6: Outline of Proxy[ℓ] at p_i – with collusion

Outline of GroupDistribution[ℓ] **at** p_i**:**

- Time is divided into blocks of length $dline/4$.
- At the beginning of the second round of a block, if has been alive for at least $2dline/3$ rounds, then collect all the fragments that have been injected since the first round of the block, and set $status$ to active. (The first round of the block is spent waiting for rumor fragments from the previous block.)
- Each block is divided into *iterations* of $\sqrt{dline} + 2$ rounds. In each iteration, we maintain a set $collaborators$ of the active processes (i.e., processes with $status$ active) in the same group as p_i. Say that p_i is in group $P_{b,\ell}$. We also keep track of a set $hitSet$ of processes that have been sent a message in this block; each process in this set was sent all the rumor fragments for this block. For each iteration, repeat (as long as $status =$ active):
  - Round 1: wait for rumor fragments to be injected.
  - Round 2: $\boxed{\textbf{for each other group } P_{a,\ell}, a \neq b\textbf{,}}$ send every "appropriate" rumor fragment to $\Theta(n^{1+48/\sqrt{dline}} \log n/|collaborators|)$ processes chosen uniformly at random from group $P_{a,\ell}$, excluding processes in $hitSet$. By appropriate we mean that if p_j is a process chosen randomly by p_i, then p_i sends to p_j only the rumor fragments in which p_j is in the destination set. (Recall that each partial rumor contains the target destination set as part of the metadata.) Every process that receives rumor fragments can now reconstruct the rumor and return it to its user (via the ConfidentialGossip service).
  - Rounds $3, \ldots, \sqrt{dline + 2}$ rounds: initiate an instance of GroupGossip[ℓ] (with deadline $\sqrt{dline}$) in which processes in group $P_{b,\ell}$ share their $hitSet$s, as well as count how many members of the group are still active (have $status =$ active).
- In the last round of the block, initiate an instance of AllGossip (with deadline $dline/4 - 1$). Each process p_i gossips the information in its $hitSet$, but without including the rumor fragments themselves. That is, if the $hitSet$ of process p_i indicates that some rumor fragment $\rho_{0,\ell}$ was sent to some process p_j, and if $\rho_{0,\ell}$ has identifier r, then p_i gossips that the fragment 0 for partition ℓ of the rumor associated with identifier r was sent to p_j. This provides sufficient information for the source to determine whether the rumor was delivered, without revealing the contents of the rumor. (See the description of the ConfidentialGossip service, above, for how this information is used.)
- Upon recovering from a failure, obtain the round number from the global clock, set $status =$ idle and wait until a new block begins.

Figure 7: Outline of GroupDistribution[ℓ] at p_i – with collusion

```
service ConfidentialGossip($(dline)_i$)
state
    $delivered\text{-}rumors, rumors\text{-}parts \subseteq R$ //$R$ denotes the set rumors
    $r, r0, r1 \in R$
    $rumor\text{-}cache \subset R \times \mathbb{Z} \times \mathbb{Z}$
    $hitSetM[1 \ldots \log n][0 \ldots 1]$ is a two dimensional matrix where each element is a tuple in $[n] \times \mathbb{Z}$.
    $counter \in \mathbb{Z}$
//Time is divided into blocks of $dline/4$ rounds.
//When a rumor is injected, in the first block it is split, in the second block the fragments are distributed via GroupGossip and the Proxy,
//in the third block the fragments are reassembled via GroupDistribution, and in the fourth block the sender receives confirmation.
//Upon a recovery from failure, the process retrieves the $round$ number from the global clock and proceeds analogously.
input rumor-inject($r$) //This marks the beginning of round 1 of a new block
    $counter \leftarrow counter + 1$
    for every $\ell \in \{1, \ldots, \log n\}$ do in parallel
        $\langle r0, r1 \rangle \leftarrow$ random-split($r$, $counter$, $\sqrt{dline}$, $[n]$)
        $rumor\text{-}cache \leftarrow rumor\text{-}cache \cup \langle r, counter, round \rangle$
        if $i[\ell] = 0$ then    // $i[\ell]$ represents the $\ell$-th bit of the binary representation of $i$.
            GroupGossip[$\ell$].gossip($r0$)    //Gossip the rumor fragment in group
            Proxy.distribute[$\ell$]($r1$)    //Find a proxy to distribute the other rumor fragment in the other group
        else
            GroupGossip[$\ell$].$gossip$($r1$)
            Proxy[$\ell$].distribute($r0$)

input Proxy[$\ell$].return($R$)
    GroupDistribution[$\ell$].distribute($R$)

input GroupGossip[$\ell$].deliver($R$)
    GroupDistribution[$\ell$].distribute($R$)

input GroupDistribution[$\ell$].return($R$)
    $rumor\text{-}parts \leftarrow rumor\text{-}parts \cup R$
    for every $r_1, r_2 \in rumor\text{-}parts$ do
        if merge($r_1, r_2$) $= \langle$success, $r\rangle$ then
            if $r \notin delivered\text{-}rumors$ then
                $delivered\text{-}rumors \leftarrow delivered\text{-}rumors \cup r$
                return($r$)
            $rumor\text{-}parts \leftarrow rumor\text{-}parts \setminus \{r_1, r_2\}$

input AllGossip.deliver($R$)
    for every $(\langle$distribution$, j, partition, h\rangle, *, *) \in R$ do
        $hitSetM[partition, j[partition]] \leftarrow hitSetM[partition, j[partition]] \cup \{h\}$
    for every $\langle r, c, t\rangle \in rumor\text{-}cache$ do
        if $\exists \ell \in [1, \ldots, \log n]$ where:
            $\{\langle p_k, c\rangle : p_k \in r.D\} \subseteq hitSetM[\ell, 0]$
                and
            $\{\langle p_k, c\rangle : p_k \in r.D\} \subseteq hitSetM[\ell, 1]$
        then $rumor\text{-}cache \leftarrow rumor\text{-}cache \setminus \{\langle r, c, t\rangle\}$
In every round:
    if $\exists \langle r, c, t\rangle \in rumor\text{-}cache$ where $round = t + r.d$ then
        for every $j \in r.D$ do
            Network.send($\langle$shoot$, r\rangle, i, j$)

input Network.receive($m, src, dest$)
        If $m = \langle$shoot$, r\rangle$ then return(r)
```

Figure 8: Main protocol at process i.

```
service Proxy$(dline, \ell)_i$
state
    $failed\text{-}proxies, current\text{-}proxies, proxy\text{-}ack, collaborators \subseteq [n]$
    $status \in \{$idle, active$\}$
    $my\text{-}rumors, waiting\text{-}rumors, proxy\text{-}buffer \subseteq R$
    $r \in R$
    $wakeup \in \mathbb{Z}$
 //Time is divided into blocks of $dline/4$ rounds.
//Each block is divided into iterations of $\sqrt{dline} + 2$ rounds.
//Each iteration consists of 1 sending round, 1 gossip instance of $\sqrt{dline}$ rounds, and 1 acknowledging round.
 On recovery from failure:
    $wake\text{-}up \leftarrow round$  // the value of $round$ is retrieved from the global clock
    $status \leftarrow$ idle

At the beginning of round 1 of a new block:
    if $|round - wakeup| \geq dline/4$ then
            $my\text{-}rumors \leftarrow waiting\text{-}rumors$
            $waiting\text{-}rumors \leftarrow \emptyset$
            if $my\text{-}rumors \neq \emptyset$ then
               $status \leftarrow$ active
               $failed\text{-}proxies, partial\text{-}rumors, proxy\text{-}buffer, proxy\text{-}ack \leftarrow \emptyset$
               $collaborators \leftarrow \{j \in [n] : j[\ell] = i[\ell]\}$

At the beginning of round 1 of an iteration:
    if $status =$ active then
            $current\text{-}proxies \leftarrow \Theta(n^{1+48/\sqrt{dline}} \log n)/|collaborators|$ processes chosen uniformly at random
                                                              from $\{j \in [n] : j[\ell] \neq i[\ell]\} \setminus failed\text{-}proxies$
            for every $j \in current\text{-}proxies$ do  Network.send($\langle$proxy, $my\text{-}rumors\rangle, i, j$)

At the beginning of round 2 of an iteration:
    $collaborators \leftarrow \emptyset$
    if $status =$ active then GroupGossip[$\ell$].gossip($\langle proxy\text{-}buffer, failed\text{-}proxies, i\rangle, \sqrt{dline}, [n]$)

At the beginning of the last round of an iteration:
    if $status =$ active then for every $j \in proxy\text{-}ack$ do  Network.send(proxy-ack, $i, j$)

At the end of the last round of a block:
    ConfidentialGossip[$\ell$].return($partial\text{-}rumors$)

input ConfidentialGossip[$\ell$].distribute($r$)
    $waiting\text{-}rumors \leftarrow waiting\text{-}rumors \cup \{r\}$

input GroupGossip[$\ell$].deliver($R$)
    for every $(\langle m, F, j\rangle, *, *) \in R$ do
            $failed\text{-}proxies \leftarrow failed\text{-}proxies \cup F$
            if $status \neq$ idle then $collaborators \leftarrow collaborators \cup \{j\}$
            $partial\text{-}rumors \leftarrow partial\text{-}rumors \cup m$

input Network.receive($\langle$proxy, $m\rangle$, $src$, $dest$)
    $proxy\text{-}buffer \leftarrow proxy\text{-}buffer \cup \{m\}$
    $proxy\text{-}ack \leftarrow proxy\text{-}ack \cup \{src\}$

input Network.receive(proxy-ack, $src$, $dest$)
    $status \leftarrow$ idle
```

Figure 9: Proxy search at process i for partition ℓ.

```
service GroupDistribution($dline, \ell$)$_i$
state
    $partials, waiting\text{-}partials \subseteq [n] \times R$
    $target\text{-}procs, collaborators, hitProcs \subseteq [n]$
    $hitSet \subseteq [n] \times \mathbb{Z}$
    $target\text{-}msg \subseteq R \times R \times \cdots R \times [n]$
    $wakeup \in \mathbb{Z}$
    $status \in \{$idle, active$\}$
 //Time is divided into blocks of $dline/4$ rounds.
//Each block is divided into iterations of $\sqrt{dline} + 2$ rounds.
//Each iteration consists of one initialization round, one distribution round and one gossip instance of $\sqrt{dline}$ rounds.
 On recovery from failure:
    $wakeup \leftarrow round$
    $status \leftarrow$ idle

At the beginning of round 2 of a block:
    if $|round - wakeup| \geq 2dline/3$ then
        $status \leftarrow$ active
        $partials \leftarrow waiting\text{-}partials$
        $hitSet, waiting\text{-}partials \leftarrow \emptyset$
        $collaborators \leftarrow \{j \in [n] : j[\ell] = i[\ell]\}$

At the beginning of round 2 of an iteration:
    if $status =$ active then
        $hitProcs = \{p \in [n] : \langle p, \cdot\rangle \in hitSet\}$
        $target\text{-}procs \leftarrow \Theta(n^{1+48/\sqrt{dline}} \log n/|collaborators|)$ processes chosen uniformly at random
                                                from $\{j \in [n] : j[\ell] \neq i[\ell]\} \setminus hitProcs$
        for every $j \in target\text{-}procs$ do
            $target\text{-}msg \leftarrow \{r_k \in partials : j \in r_k.z.D\}$
            $hitSet \leftarrow hitSet \cup \{\langle j, r_k.z.cnt\rangle : r_k \in target\text{-}msg\}$
            Network.send($\langle$partials, $target\text{-}msg\rangle, i, j$)

At the beginning of round 3 of an iteration:
    $collaborators \leftarrow \emptyset$
    if $status =$ active then GroupGossip[$\ell$].gossip($\langle$share, $hitSet, i\rangle, \sqrt{dline}, [n]$)

At the end of the last round of a block:
    AllGossip.gossip($\langle$distribution, $i, \ell, hitSet\rangle, dline/4 - 1, [n]$)

input ConfidentialGossip[$\ell$].distribute($r$)
    $waiting\text{-}partials \leftarrow waiting\text{-}partials \cup \{r\}$

input GroupGossip[$\ell$].deliver($\langle$share, $h, j\rangle$)
    if $status =$ active then
        $collaborators \leftarrow collaborators \cup \{j\}$
        $hitSet \leftarrow hitSet \cup h$

input Network.receive($m$, $src$, $dest$)
    if $m = \langle$partials, $r_1, r_2, \ldots\rangle$ then for every $r_k \in m$ do ConfidentialGossip[$\ell$].return($r_k$)
```

Figure 10: Rumor distribution between groups at process i for partition ℓ.

```
service Filter($\ell$)$_i$
input GroupGossip[$\ell$].send($m$, $src$, $dest$)
    if $i[\ell] = src[\ell]$ then Network.send($m$, $src$, $dest$)

input Network.receive($m$, $src$, $dest$)
    GroupGossip[$\ell$].receive($m$, $src$, $dest$)
```

Figure 11: Filter ℓ at process i.