Received 8 November 2023, accepted 25 December 2023, date of publication 1 January 2024,
date of current version 11 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3349021

RESEARCH ARTICLE

Optimizing Gossiping for Asynchronous Fault-Prone IoT Networks With Memory and Battery Constraints

CARLOS BARROSO-FERNÁNDEZ[1], ERNESTO JIMÉNEZ[2], JOSÉ LUIS LÓPEZ-PRESA[2], MARTA MORENO-CUESTA[2], AND RAMON XULVI-BRUNET[3]

[1]Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, 28903 Getafe, Spain
[2]Departamento de Sistemas Informáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain
[3]Departamento de Física, Escuela Politécnica Nacional, Quito 170109, Ecuador

Corresponding author: Ernesto Jiménez (ernesto.jimenez.merino@upm.es)

This work was supported in part by the Ministry of Economy and Competitiveness (MINECO) under the QOSDATA Project PID2020-119461GB-I00; in part by the Regional Government of Madrid (CM) under the EDGEDATA Project P2018/TCS-4499; in part by the Spanish National Research Agency under the TRUE5G Project PID2019-108713RB[1]C52/AEI/10.13039/501100011033; in part by Ministerio de Asuntos Económicos y Transformación Digital and European Union (EU)-NextGenerationEU through the Project Data&Code Continuum: Infraestructura y Sevicios para Datos y Código en el Continuo Cloud/Edge; and in part by Fundación Carolina for the Fellowship Estancias Cortas Postdoctorales, in 2020.

ABSTRACT A gossip protocol is a procedure by which a device disseminates its rumor to all devices on a network. The traditional definition of the gossip problem states that, in a system with n interconnected devices where each of them only knows initially its own rumor, after a known finite time of message exchange, all devices learn the n rumors. This definition considers synchronous systems where devices cannot fail. Gossip-based protocols can be very useful in Internet of Things (IoT) networks as a means of disseminating information. Thus, we have reformulated the problem to expand the spectrum of networks in which the gossip technique can be applied, including asynchrony and device failures, very common features in current IoT newtwoks. Gossiping consumes valuable memory and battery resources in devices. This can be particularly problematic for small IoT devices that operate with reduced memory in remote or inaccessible locations, where battery replacement or recharging is difficult. To address these issues, two efficient gossip protocols are presented, so that the efficiency and longevity of IoT devices is not compromised. The first protocol (battery-efficient protocol) allows to reduce battery consumption while still having a good time performance, but needs quadratic memory, like traditional versions. The second protocol (memory-optimal protocol) only needs linear memory to store and manage the rumors, and is capable of significantly reducing the number of messages sent, while obtaining similar (or, in many cases, better) performance than the first protocol. Both protocols are formally proved to be correct, and upper and lower limits of their number of messages sent are determined. These theoretical limits, however, differ so much among them that they do not permit to guess the number of messages that a real case may require. Hence, the practical performance of both protocols is assessed with experiments.

INDEX TERMS Gossip protocols, crash failures, IoT networks, partial-knowledge of membership, unknown maximum number of failures, reliable links, asynchronous systems, sensor networks.

I. INTRODUCTION

The dissemination of information is a fundamental aspect of the Internet of Things (IoT), where devices that are interconnected (not necessarily each pair of them directly linked) exchange data messages among them [1], [2], [3], [4]. Traditionally, gossip protocols have been widely used as an efficient way to ensure that information is distributed to all devices [5]. Gossip protocols are peer-to-peer communication procedures that rely on each device

The associate editor coordinating the review of this manuscript and approving it for publication was Qingchun Chen.

(called process) broadcasting its values (called rumors) to the rest of devices of the network (called system) by transmitting them only to its neighbors (directly linked devices). The way these procedures work is much like the way epidemics spread [6], where a person with a virus diffuses a disease by infecting a neighbor, who in turn infects another person, and so on. The similarities between gossiping and epidemiology have attracted a lot of attention in the literature [2], [7], [8], [9], [10], and [11].

The traditional definition of the deterministic gossip problem [12], [13] states that, in a system with n interconnected processes where each of them initially knows only its own rumor, after a known finite time period of message exchange, all processes eventually learn the n rumors [5]. Another family of gossip protocols, denoted as random [14], [15], is that in which the solution is achieved asymptotically with probability 1. We shall not consider random gossip protocols here.

The previous definition of the deterministic gossip problem considers synchronous systems where no processes can fail [2], [7], [8], [9], [10], [11], and makes no mention to the memory limitation that might be present in practice in many devices.

A. SYSTEM PROPERTIES

Gossip protocols work in rounds of communication, each round following the same basic framework (called *push scheme* [11], [16]). At the initial round, every process p_i, that only knows its own value r_i (called the rumor of p_i), randomly chooses another process connected to it, e.g. p_j, and sends value r_i to p_j. In each subsequent round, this sending action is repeated by p_i, but now adding the new rumors it has received in previous rounds. Even with this simple push scheme in mind, the design of efficient gossip protocols in distributed systems is not an easy task. The following fundamental issues need to be considered: *synchrony*, *reliability* and *knowledge of membership*.

- **Synchrony** [17]. Three main types of distributed systems have been studied in the literature. In *synchronous systems*, both the speed at which processes run and the time required for a message to reach the destination are known and bounded. Thus, the key parameters of the system are the processing time needed at each round (called δ) and the transmission time (called d). Conversely, in *asynchronous systems*, δ and d are unbounded and, hence, unknown. Finally, *partially-synchronous systems* are analogous to synchronous systems, except that the values of δ and d, which are also bounded, are not known by the processes. In this paper, we shall focus on asynchronous systems, the weakest type of systems, because, from a practical point of view, IoT networks are often asynchronous.
- **Reliability** [17]. Two types of elements need to be considered: processes and links. Regarding the reliability of the processes, we have a *non-crashing system* when all the processes are correct, that is, when processes never fail (by crashing permanently). Conversely, *fault-prone systems* are those in which there may be faulty processes, in addition to the correct ones. Concerning the reliability of links, two basic types of links are considered: *lossy links* [18], where a message transmitted between two correct processes may not be received, and, *reliable links* [19], where the messages transmitted between correct processes can never be lost or altered. UDP and TCP are two examples of protocols that provide lossy and reliable links, respectively. The protocols we propose in the following sections assume that links are reliable, since this choice simplifies the understanding of the protocols and makes them easier to implement. However, given that devices often stop working in IoT networks due to physical damage or battery discharge, our system considers that processes can fail.
- **Knowledge of membership** [20]. It relates to the knowledge each process has about the other processes in the system, and it is usually associated to the degree of interconnection among the processes. Thus, in a fully connected network –where every process is connected to any other process–, *global-knowledge of membership* can be useful or even necessary. However, when processes can directly reach only a few other processes (called their *neighbors*), then global knowledge might not be necessary in advance, and *partial-knowledge of membership* –that is, knowledge of the neighborhood– might suffice. In the case of IoT networks, devices are more likely to connect with only a few neighbors, since fully connected networks place significant constraints on communication power and network infrastructure. Thus, we shall asume in this paper that the knowledge of membership and connectivity of the system is partial.

B. PROTOCOL FEATURES

Synchrony, reliability and knowledge of membership affect three essential features of gossip protocols: the frequency of rounds (shortly, *frequency*), the selection of rumors to send in each round (shortly, *notification*), and the conditions to finish the protocol (shortly, *termination*).

Let's consider the simplest case of a distributed system: synchronous, non-crashing, global-knowledge of membership with all processes interconnected with reliable links, and using the *push scheme*, where, in each round r, all processes simultaneously perform the r-th sending round during a discrete time interval. The round frequency can easily be set depending upon the known values δ and d. With respect to notification, each process p_i transmits the rumors that it knows to a randomly chosen process. Finally, the termination condition is reached when all rumors have been received by all processes. A protocol of this kind, such as that proposed in [11], can diffuse a rumor in $O(\log n)$ rounds and using $O(n \log n)$ messages.

To reduce the number of rounds before termination, the *push–pull* scheme [2], [16] was developed for gossip protocols. With this scheme, in each round r, every process p_i selects a random neighbor p_j and sends to it a message with the rumors that p_i knows (push phase). Then, also in this same round r, p_i selects another random neighbor p_k and sends a request to p_k to ask for an update (pull phase). When p_k receives this request, it responds with the rumors it knows. Note that, in an asynchronous system with faulty processes, when a process p_i is waiting to receive a message from another process p_k and it does not arrive within a given time, p_i cannot know if this delay is because p_k has failed or because p_k is slow.

In [15], a gossip protocol with a *push–pull* scheme is presented in a synchronous system without any kind of failures and it is proved that a single rumor spreads, with a high probability, in $O(\log n)$ rounds and that the number of messages reduces to only $O(n \log \log n)$.

In distributed systems presenting "uncertainties" (with respect to synchrony, reliability, or knowledge of membership), deciding the frequency, notification, and termination in gossip protocols is an active area of research [9], [16], [20]. For example, if there is not a global knowledge of the membership, the gossip solution in [11] does not *terminate* because a process never knows when it has collected all rumors. If one includes failure of processes and asynchrony in the system, the gossip protocol in [15] may not finish, given that a correct process p_i, which may be waiting for the answer of process p_j, never knows if p_j is delaying its message because it is slower than p_i or because it has crashed (recall that δ and d are unbounded in an asynchronous system). In an asynchronous system, since a process can never have the certainty of receiving the rumors of all correct processes, it is impossible to devise a gossip protocol which satisfies proper termination when there may be an unknown number of process failures [21] (despite the fact that it is possible to guarantee that all correct processes eventually stop sending messages [21]). This *quiescence* feature is absolutely necessary to obtain efficient gossip protocols in asynchronous fault-prone systems.

An additional feature to be considered in gossip protocols is the amount of memory needed by processes to keep and manage the information of the rumors they know, since gossiping can consume a lot of memory and battery resources, leading to reduced device performance and shortening battery life. From a practical point of view, memory is therefore a very important feature in some IoT networks (e.g., WSN), where battery and memory can be so constrained that traditional gossip protocols are unable to properly run when either the number of network devices or the needed number of messages to be sent is large [22], [23], [24], [25], [26], [27].

To sum up, proper determination of frequency, notification and termination (or quiescence) is vital when configuring a gossip protocol and is closely related to the properties of the distributed system at hand. In general, a gossip solution can be feasible or not depending upon all these features [20].

C. OUR CONTRIBUTIONS

In this paper, we are interested in a more realistic environment for IoT networks: asynchronous systems where processes can fail and have low memory capacity. The system we shall consider is a small IoT network where the devices are sensors with very constrained battery and memory resources [22], [23], [24], [25], [26], [27].

In asynchronous fault-prone systems, where a process never knows when to *terminate*, the traditional deterministic definition is not valid anymore. A new definition, called *quiescent gossip problem*, states that termination takes place when, after an unknown finite time period of message exchange, each non-faulty process p_i learns eventually a set of rumors X_i that contains, at least, the rumors of all non-faulty processes [21]. We shall use a stronger version here related to the so-called *uniform quiescent gossip problem*, where all non-faulty processes eventually learn the same set of values, meaning that there is an unknown finite time after which $X_i = X_j$ for all non-faulty processes p_i and p_j.

To ensure that all processes give the same outcome, they must deterministically choose among values from the same set. This is necessary in applications where agreement on a common value or on the next operation to execute is required [28], [29], [30], [31]. A uniform quiescent gossip protocol can provide such common set, what makes it very suitable for this kind of problems.

We shall introduce two deterministic gossip-based protocols for asynchronous IoT sensor networks that allow any number of processes to fail (as long as the network always remains minimally connected). The first protocol (BE-gossip, for battery-efficient) is based on the push scheme and is capable of reducing battery consumption, but needs quadratic memory, like traditional protocols. Our second protocol (MO-gossip, for memory-optimal) only needs linear memory to store and manage the rumors known. This second protocol, that uses a push-pull scheme in order to achieve this important property of linear memory usage, is also able to significantly reduce the number of messages sent while still maintaining a quiescence time similar to that of the first protocol.

D. OUTLINE

In Section II, we introduce our asynchronous fault-tolerant gossip system and the formal definitions of the gossip problem. In Section III, we present protocols BE-gossip and MO-gossip, prove their correctness, and estimate upper and lower bounds of their number of messages exchanged. (It is worth noting that the gap between these bounds is so wide that an experimental evaluation is needed to assess the practical performance of the protocols. Note also that a theoretical analysis of quiescence time is not possible in asynchronous systems). In Section IV, experiments are provided to analyze the practical performance of both protocols, focusing in particular on the trade-off between the speed of rumor diffusion, the number of messages that need to be sent, and the quiescence time. This analysis is done both when the number of faulty processes is low and when the number of faulty processes is large. Additionally, the analysis is repeated for different values of data transmission rates between devices. Finally, in Section V, we present some concluding remarks.

II. THE ASYNCHRONOUS FAULT-PRONE SYSTEM S

The system S is composed by a finite set Π of n processes, i.e., $|\Pi| = n$. Each process p_i has its own unique identity i.

In some gossip systems, the identities of the processes in the system are known to all members of the system and there is a direct all-to-all connection between them. However, in System S, processes do not need to be connected to all other members of the system, nor do they need to know their identities in advance. Instead, a process only needs to know the identities of the processes with which it has a direct connection, i.e., its *neighbors*. The set of neighbors of process p_i is denoted by $\mathcal{N}_i$. Note also that the system where all processes are neighbors is the particular case of System S where, for all process $p_i \in \Pi$, $\mathcal{N}_i = \Pi$.

A process can fail by crashing, i.e., it eventually stops its execution permanently without having reached the end of its program. We say that a process $p_i \in \Pi$ is *correct* if it does not fail, and *faulty* otherwise. We denote by $\mathcal{C}$ the set of correct processes and by $\mathcal{F}$ the set of faulty processes. In traditional gossip systems, processes may know $f = |\mathcal{F}|$. However, since processes do not know n in S, it does not make sense to know f. Thus, in order to minimize the knowledge of the processes in the system, we consider that the processes have no knowledge on faults.

Let $\mathcal{L}$ be the set of links connecting processes. $G = (\Pi, \mathcal{L})$ is the (undirected) graph representing the network, where each process is a node of the graph and each connection (i, j) between two processes $p_i, p_j \in \Pi$ is an edge. In S, processes do not need to have the same number of neighbors, i.e., G does not need to be regular. Let $G_{\mathcal{C}} = (\mathcal{C}, \mathcal{L}_{\mathcal{C}})$, where $\mathcal{L}_{\mathcal{C}} = \{(i, j) \in \mathcal{L} : p_i, p_j \in \mathcal{C}\}$, be the subgraph induced by the set of correct processes. Analogously, let $G_{\mathcal{F}} = (\mathcal{F}, \mathcal{L}_{\mathcal{F}})$, where $\mathcal{L}_{\mathcal{F}} = \{(i, j) \in \mathcal{L} : p_i, p_j \in \mathcal{F}\}$, be the subgraph induced by the set of faulty processes.

Property 1: For all $p_i, p_j \in \mathcal{C}$, there is a path in $G_{\mathcal{C}}$ connecting them.

Note that $G_{\mathcal{C}}$ must be connected to guarantee that every rumor from a correct process can reach all other correct processes. Note that the faulty processes might fail from the beginning. In fact, Property 1 is the weakest requirement that the network topology must satisfy for gossiping to be solved. In system S, there may be any number of faulty processes, as long as Property 1 holds.

Communication among processes is performed by exchanging messages among them, i.e., S is a message-passing system. Thus, a process p_i executes $send(m, p_j)$ to send message m to a process $p_j \in \mathcal{N}_i$. Every message contains the identity of the sender, so the receiver knows which process sent the message. Connections between processes are bidirectional and *reliable*. The loss, change, duplication or creation of a spurious message is impossible in a reliable link [19]. Thus, if a process $p_i \in \mathcal{C}$ executes $send(m, p_j)$ and $p_j \in (\mathcal{N}_i \cap \mathcal{C})$, then m is received by p_j once, and unaltered, within a finite time. However, this finite time is unknown to all processes in Π. Note that if p_i or p_j were faulty processes and crashed while the transmission was being performed, m might be delivered to p_j or not, but always without change.

The system S is *asynchronous* in the sense that the maximum time it takes for a message to be delivered after it is sent is unbounded, though finite, and the speed at which processes run is not necessarily the same. Even more, each process has its own clock and these clocks need not be synchronized.

The traditional *gossip problem* in asynchronous fault-prone systems states that, initially, each process only knows its own rumor and all correct processes eventually know at least the rumors of every correct process. Furthermore, every correct process eventually finishes its execution. A stronger version of the problem, known as *uniform gossip*, is widely studied. It adds the requirement that all the correct processes eventually know the same set of rumors forever. Let us formally define this problem.

Definition 1: (**T-gossip**) Initially each process p_i knows only its own value (called the initial rumor of p_i) and the following properties eventually hold:

- *Validity*: All rumors known by each process must be initial rumors.
- *Termination*: Every process stops running permanently.
- *Uniform Agreement*: All correct processes know the same set of rumors, and this set contains the initial rumors of all correct processes.

In an asynchronous system where n is not known, like S, a process can never be sure it has the rumors from every process in the system, so it cannot terminate. Therefore, traditional gossiping cannot be solved [21]. Then, we define a relaxed version called *uniform quiescent gossip* where, unlike traditional gossiping, processes are not required to eventually stop running, but to stop sending messages.

Definition 2: (**Q-gossip**) Initially, each process knows only its own initial rumor and the following properties eventually hold:

- *Validity*: All rumors known by each process must be initial rumors.
- *Quiescence*: Every process stops sending messages permanently.
- *Uniform Agreement*: All correct processes know the same set of rumors, and this set contains the initial rumors of all correct processes.

III. GOSSIP PROTOCOLS

When a process wants to communicate something to another process, it issues a *SPREAD* message. This is the typical *push* action. In some cases, it can be useful to reply to these messages to acknowledge receipt and update each other. In this case, when one process communicates something to another, the second responds with its own knowledge, issuing an *OK* message. This follows the *push-pull* fashion. Both communication schemes will be studied.

Eager protocols pass on what they know as soon as they learn something new. This can result in a huge number of packets being sent, probably with very little information in each one. In the protocols proposed, a constant τ will determine the minimum time between two *SPREAD* actions. Note that the eager version corresponds to the case of τ=0. Increasing the value of τ may reduce the number of packets sent, but delaying *SPREAD* actions may have an impact on the time needed to spread the rumors. Therefore, the practical behavior of the protocols will depend on τ, which will be analyzed through simulations.

All the protocols proposed in this work are *event-driven*, i.e., they are asynchronous and only execute code when some events occur; otherwise, they are in a waiting state, without executing instructions, what reduces energy consumption. The events that trigger actions in the protocols are the receipt of a message and the need to initiate a *SPREAD* action.

To deal with time, the system provides a function named currentTime() to obtain the current time. It is assumed that an action can be scheduled for an instant of time in which a condition is fulfilled both on the state of the process and on the current time. This is specified with **when** actions. These **when** actions are executed atomically, i.e., for each process, only one of these clauses may be running at any instant of time, thus avoiding concurrency issues among them.

A. BATTERY-EFFICIENT GOSSIP PROTOCOL (BE-GOSSIP)

To start with, we propose a simple protocol to solve Q-gossip in system S called BE-gossip. As it will be seen, this protocol becomes quiescent when the process assumes that all its neighbors have the same knowledge on rumors as itself. Being event-driven, it does not need to be permanently executing instructions nor does it need to be activated periodically. Therefore, it can be considered a battery-efficient protocol.

Protocol 1 (BE-gossip) uses the array $known\_by_i$ (for each process p_i), indexed by the identity of the processes in the system, to store the knowledge on rumors assumed for each of its neighbors and itself. This knowledge is a set of pairs (r, j) where r is the initial rumor of process p_j. Therefore, each member of the array is of size $O(|\Pi|)$, and this array has $|\mathcal{N}_i|+1$ members. Thus, the total memory requirement of the protocol, for process p_i, is $O(|\Pi| \times (|\mathcal{N}_i|+1))$. Two additional variables are used. Variable N_i is used to store the set of neighbors supposed to know less rumors than p_i, which has size $O(|\mathcal{N}_i|)$. Variable T is used to store the guard time until which no *SPREAD* action should be executed. This variable needs $O(1)$ space.

The protocol works as follows. When it starts, process p_i only knows its own rumor r_i (received as a parameter), so $known\_by_i[i]$ is set to the set which only contains the pair (r_i, i) (see Line 1). With respect to its neighbors, nothing is known, as set in Line 2. Therefore, all neighbors are assumed to know less rumors than p_i. Note that p_i knows its own rumor, but its neighbors do not yet. Therefore, a *SPREAD* action should be executed immediately (see Lines 3–4), i.e., the set of neighbors that know less rumors than p_i (N_i) is $\mathcal{N}_i$, and T is the current time. Then, the event-driven task T1 is started, which processes two types of events. Thus, the clause of Lines 11–18 will be ready to be executed.

When a *SPREAD* action must be executed, one process $p_j \in N_i$ is randomly chosen (Line 12), and the set of rumors known by process p_i but not assumed to be known by p_j is sent to this process p_j in Line 14. Sending this difference instead of the whole knowledge of process p_i consumes processing power, but reduces the size of the *SPREAD* messages exchanged. Since communication links are reliable, p_i assumes that p_j will eventually receive this message and will store these rumors, so p_j will eventually know all the rumors p_i currently knows (see Line 15). Therefore, p_j is removed from N_i (Line 16). Note that p_i assumes p_j is a correct process. If p_j is a faulty process, p_i might send messages to p_j which would consume energy and bandwidth, but would never reach process p_j. Finally, the guard time until which no other *SPREAD* action should be executed is set to the current time plus τ in Line 17.

When a *SPREAD* message is received from some process p_k, it carries a set of pairs (r, j) where r is the initial rumor of process p_j. This set includes the rumors known by p_k, but which p_i may not know yet. Then, p_i learns about these rumors (Line 7), and it also records they are known by p_k (Line 8). Finally, the set of neighbors supposed to know less rumors than p_i is recomputed in Line 9. This may trigger the event that starts the **when** action of Line 11.

Protocol 1 BE-gossip(r_i)

```
Input: rumor of process $p_i$
 1: $known\_by_i[i] \leftarrow \{(r_i, i)\}$
 2: $known\_by_i[k] \leftarrow \emptyset, \forall p_k \in \mathcal{N}_i$
 3: $N_i \leftarrow \mathcal{N}_i$
 4: $T \leftarrow$ currentTime()
 5: start Task Push()

Task: Push()
 6: when ($SPREAD$, $rumors$) received from $p_k$ do
 7:     $known\_by_i[i] \leftarrow known\_by_i[i] \cup rumors$
 8:     $known\_by_i[k] \leftarrow known\_by_i[k] \cup rumors$
 9:     $N_i \leftarrow \{p_j \in \mathcal{N}_i : known\_by_i[j] \subset known\_by_i[i]\}$
10: end when
11: when currentTime() $\geq T \wedge N_i \neq \emptyset$ do
12:     let $p_j$ be a random process in $N_i$
13:     $unknown \leftarrow known\_by_i[i] \setminus known\_by_i[j]$
14:     $send((SPREAD, unknown), p_j)$
15:     $known\_by_i[j] \leftarrow known\_by_i[i]$
16:     $N_i \leftarrow N_i \setminus \{p_j\}$
17:     $T \leftarrow$ currentTime() $+ \tau$
18: end when
```

Let us formally prove the correctness of this protocol. To be correct, it must fulfill the three properties specified in Definition 2: validity, quiescence and uniform agreement.

Lemma 1: For each process $p_i \in \Pi$ running protocol BE-gossip, $known\_by_i$ contains only initial rumors of processes in Π.

Proof: Using recursive induction, note that, initially, for each $p_i \in \Pi$, $known\_by_i[i]$ is set to $\{(r_i, i)\}$ and $known\_by_i[j]$ is set to $\emptyset$ for all $p_j \in \mathcal{N}_i$ (see Lines 1 and 2), so $known\_by_i$ contains only initial rumors. Subsequently, it is updated every time a *SPREAD* message is sent or received. In the first case (Lines 11–18), $known\_by_i[j]$ is set to $known\_by_i[i]$. Assuming that $known\_by_i[i]$ contains only initial rumors, then $known\_by_i[j]$ will also contain only initial rumors. In the second case (Lines 6–10), the rumors that come in the *SPREAD* message are added to $known\_by_i[i]$ and $known\_by_i[k]$. Since the links are reliable, these rumors must have been sent by process p_k in Line 14 and they are a subset of the messages in $known\_by_k[k]$. Assuming that $known\_by_k[k]$ only contained initial rumors, then the new values of $known\_by_i[i]$ and $known\_by_i[k]$ will also contain only initial rumors.

Corollary 1: All rumors known by a process running protocol BE-gossip must be initial rumors.

Proof: The rumors known by a process $p_i \in \Pi$ are those stored in $known\_by_i[i]$. Hence, from Lemma 1, all rumors known by a process running protocol BE-gossip must be initial rumors.

Corollary 2: For each process $p_i \in \Pi$ running protocol BE-gossip, there is a time after which no more rumors are added to $known\_by_i$.

Proof: Follows directly from Lemma 1 and the fact that the set of processes Π (and hence the set of initial rumors) is finite.

Lemma 2: For each process $p_i \in \mathcal{C}$ running protocol BE-gossip, there is a time t after which N_i becomes permanently empty.

Proof: By the way of contradiction, note that, if there is no time t after which N_i is permanently empty, then there must be a process p_j which never leaves N_i or which gets into N_i an infinite number of times. Each time the event of Line 11 occurs, one process leaves N_i in Line 16. If N_i is not empty, every τ units of time, a process leaves N_i. Therefore, if some process is never chosen in Line 12, then there must be at least another process in N_i every time Line 12 is executed. Hence, both cases reduce to the case where a process p_j gets into N_i an infinite number of times. However, if p_j gets into N_i, then $known\_by_i[j] \subset known\_by_i[i]$ (see Line 9). Since $known\_by_i[j]$ is set to $known\_by_i[i]$ in Line 15 each time it is chosen in Line 12, an infinite number or rumors must be added to $known\_by_i[i]$. However, that is not possible from Corollary 2.

Lemma 3: Eventually, every process running protocol BE-gossip stops sending messages permanently.

Proof: By the way of contradiction, assume that there is a process p_i which sends an infinite number of messages. Since these messages are sent in Line 14, the condition of Line 11 must be fulfilled an infinite number of times. Therefore, $N_i \neq \emptyset$ an infinite number of times. However, that is not possible from Lemma 2.

Let us show now two invariants satisfied by the **when** clauses in protocol BE-gossip. These invariants are statements that hold both before the execution of a clause starts, and when it ends. Since the **when** clauses are executed atomically, these invariants are protocol invariants.

Remark 1: Let $p_i \in \Pi$. For all $p_j \in \mathcal{N}_i$, $known\_by_i[j] \subseteq known\_by_i[i]$.

Proof: It is easy to see that $known\_by_i[j] \subseteq known\_by_i[i]$ in the beginning, since $known\_by_i[j]$ is set to $\emptyset$ in Line 2. Assume that the claim holds at the beginning of the execution of a **when** clause. Then, each time new rumors are received from a process $p_j \in \mathcal{N}_i$ (**when** clause of Lines 6–10), these rumors are included both in $known\_by_i[i]$ (Line 7) and $known\_by_i[j]$ (Line 8), so the claim still holds at the end of the clause. Considering the **when** clause of Lines 11–18, $known\_by_i[j]$ is modified in Line 15, where it is set to $known\_by_i[i]$, so again the claim still holds at the end this **when** clause.

Remark 2: Let $p_i \in \mathcal{C}$. For all $p_j \in \mathcal{N}_i$, $known\_by_i[j] \subset known\_by_i[i] \iff p_j \in N_i$.

Proof: By induction, note that, in the beginning, $known\_by_i[i] = \{(r_i, i)\}$ (Line 1), $known\_by_i[k] = \emptyset$, for all $p_k \in \mathcal{N}_i$ (Line 2), and $N_i = \mathcal{N}_i$, so the claim holds. Assume that the claim holds before a **when** clause is executed. Then, when the clause of Lines 6–10 is executed, $N_i = \{p_j \in \mathcal{N}_i : known\_by_i[j] \subset known\_by_i[i]\}$ (Line 9), so the claim holds at the end of that clause. In the case of the clause of Lines 11–18, some process p_j is removed from N_i (Line 16) after $known\_by_i[j]$ is set to $known\_by_i[i]$, so again the claim holds at the end of the clause.

To prove that protocol BE-gossip satisfies *uniform agreement*, some preliminary results are necessary, which will be shown by the following remarks.

Remark 3: For each correct process p_i running protocol BE-gossip, there is a time after which for all process $p_j \in \mathcal{N}_i$, $known\_by_i[i] = known\_by_i[j]$ permanently.

Proof: From Remark 1, for all $p_j \in \mathcal{N}_i$, $known\_by_i[j] \subseteq known\_by_i[i]$. Furthermore, from Remark nekb[j]->j-in-N]nekb[j]->j-in-N]2, for all $p_j \in \mathcal{N}_i$, $known\_by_i[j] \subset known\_by_i[i] \iff p_j \in N_i$. However, from Lemma 2, there is a time t after which N_i becomes permanently empty. Hence, after time t, no process may be in N_i, so for all process $p_j \in \mathcal{N}_i$, $known\_by_i[i] = known\_by_i[j]$ permanently.

Remark 4: Let p_i be a correct process running protocol BE-gossip. For all $p_j \in (\mathcal{N}_i \cap \mathcal{C})$, $(r_k, k) \in known\_by_i[j]$ implies that either (r_k, k) is already in $known\_by_j[j]$ or it will eventually be in $known\_by_j[j]$.

Proof: Consider the first time (r_k, k) is inserted into $known\_by_i[j]$. That can happen either in Line 8 or in Line 15. In the first case, that rumor came in a *SPREAD* message received from p_j (see Line 6), and that message was sent by process p_j in Line 14, what implies that (r_k, k) was already in $known\_by_j[j]$ when that message was sent. In the second case, p_i previously sent a *SPREAD* message to p_j in Line 14 which included (r_k, k) and, since both p_i and p_j are correct and the links are reliable, p_j will eventually receive that message. Then, it will be processed by the **when** clause of Lines 6–10. Therefore, in Line 7, (r_k, k) will be inserted in $known\_by_j[j]$ (if it was not yet). Hence, (r_k, k) was already in $known\_by_j[j]$ or it would eventually be in $known\_by_j[j]$.

Remark 5: Let p_i be a correct process running protocol BE-gossip. For all $p_j \in (\mathcal{N}_i \cap \mathcal{C})$, eventually, $known\_by_i[i] = known\_by_j[j]$ permanently.

Proof: Let p_i be any correct process. From Remark 3, eventually, for all process $p_j \in \mathcal{N}_i$, $known\_by_i[i] = known\_by_i[j]$ permanently. Furthermore, from Remark 4, if $p_j \in (\mathcal{N}_i \cap \mathcal{C})$, then $(r_k, k) \in known\_by_i[j]$ implies that either (r_k, k) is already in $known\_by_j[j]$ or it will eventually be in $known\_by_j[j]$. Therefore, for all $p_j \in (\mathcal{N}_i \cap \mathcal{C})$, eventually, $known\_by_i[i] = known\_by_i[j] \subseteq known\_by_j[j]$. Note that the same argument can be used to p_i with respect to p_j, so $known\_by_j[j] = known\_by_j[i] \subseteq known\_by_i[i]$. Hence, eventually, $known\_by_i[i] = known\_by_j[j]$ permanently.

Lemma 4: Eventually, all correct processes running protocol BE-gossip know the same set of rumors, and this set contains the initial rumors of all correct processes.

Proof: Consider any process $p_i \in \mathcal{C}$. Initially, $known\_by_i[i]$ contains the initial rumor of p_i (see Line 1). Since $known\_by_i[i]$ only grows, $known\_by_i[i]$ contains the initial rumor of p_i permanently. Besides, from Remark 5, eventually, for all process $p_j \in (\mathcal{N}_i \cap \mathcal{C})$, $known\_by_i[i] = known\_by_j[j]$. Therefore, $known\_by_j[j]$ eventually contains the initial rumor of p_i permanently. Furthermore, since there is a path from p_i to every correct process in Π, eventually, for all $p_k \in \mathcal{C}$, $known\_by_i[i] = known\_by_k[k]$ and $known\_by_k[k]$ will contain the initial rumor of p_i. Since this argument can be extended from p_i to every other correct process, it can be concluded that, eventually, all correct processes know the same set of rumors, and this set contains the initial rumors of all correct processes.

Theorem 1: Protocol 1 (BE-gossip) solves Q-gossip.

Proof: Direct from Corollary 1 (validity) and Lemmas 3 (quiescence) and 4 (uniform agreement).

Once correctness has been proved, let us focus on the performance of protocol BE-gossip. Since the system is asynchronous, time limits cannot be computed. Therefore, we will only consider the number of messages and the number of rumors the protocol exchanges to complete its task.

Theorem 2: The number of rumors exchanged until quiescence by protocol BE-gossip is $O(|\Pi| \times |\mathcal{L}|)$ and $\Omega(|\mathcal{C}| \times |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|)$.

Proof: Recall that each correct process p_i stores the rumors known by itself and its neighbors in variable $known\_by_i$ and only the rumors in $known\_by_i[i] \setminus known\_by_i[j]$ are sent from p_i to p_j (see Line 14). Since each time a *SPREAD* message is sent from p_i to p_j, $known\_by_i[j]$ is set to $known\_by_i[i]$ (see Line 15), then each rumor is sent from p_i to p_j at most once, for each pair of processes $p_i, p_j \in \Pi$. Furthermore, from Theorem 1, protocol BE-gossip solves Q-gossip, so $known\_by_i[i]$ eventually contains the rumors of every correct process and, maybe, some of the rumors of the faulty processes. If no process crashes, then each rumor traverses each link in $\mathcal{L}$ at most once in each direction. Note that it is possible that a process p_i sends a rumor r to p_j and p_j sends rumor r to p_i before it receives r from p_i. Otherwise, if p_j has received r from p_i, then it will not send r back to p_i, since it will already be in $known\_by_j[i]$. Thus, at most $2 \times |\Pi| \times |\mathcal{L}|$ rumors are sent, and at least $|\Pi| \times |\mathcal{L}|$ rumors are sent. In fact, the first time a process p_i sends its own rumor r_i, e.g. to some other process p_j, it is impossible that p_j can be sending r_i to p_i, because it does not know it yet. Then, the maximum number of rumors sent is $2 \times |\Pi| \times |\mathcal{L}| - |\Pi|$. Yet, it is $O(|\Pi| \times |\mathcal{L}|)$. In case there are faulty processes, then the number of rumors exchanged is decreased by the number of rumors the faulty processes do not send. If every faulty process crashes before sending any message, then the number of rumors exchanged would be at least $|\mathcal{C}| \times |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|$ since each rumor would be sent only once through links shared by at least one correct process and only correct processes would send messages. Thus, it can be concluded that the number of rumors sent is $O(|\Pi| \times |\mathcal{L}|)$ and $\Omega(|\mathcal{C}| \times |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|)$.

The number of messages needed by the protocol to become quiescent depends on the topology of the network and the instants of time in which the different processes send the rumors they know, as well as the target processes they choose. However, a coarse worst case bound on the number of messages can be considered to be the number of rumors exchanged, since each message carries at least one rumor. Therefore, the following corollary can be stated.

Corollary 3: The maximum number of messages exchanged until quiescence by protocol BE-gossip is $O(|\Pi| \times |\mathcal{L}|)$.

Since the system is asynchronous, it is possible to build an scenario where messages are sent in an order such that each message is sent at the right time to collect as much information as possible. Thus, the number of messages needed in the best case is linear in the number of nodes and edges in the system, as proved in the following theorem.

Theorem 3: The minimum number of messages exchanged until quiescence by protocol BE-gossip is $\Omega(|\mathcal{C}| + |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|)$.

Proof: In the best case, all the faulty processes crash before sending any messages, since in this way the information to be disseminated is minimal. Since $G_{\mathcal{C}}$ is connected (from Property 1), there is a spanning tree that connects the correct processes. Hence, in the best case, with $|\mathcal{C}| - 1$ messages, it is possible to collect all the rumors and carry them to the root of that spanning tree. Then, with $|\mathcal{C}| - 1$ messages containing all the rumors, it would be possible to spread this full knowledge to every correct process. However, the protocol must send, at least, one message with all the rumors through every link in $\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}}$. Hence, $|(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|$ will be required. Thus, the total number of messages sent in the best case is $\Omega(|\mathcal{C}| + |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|)$.

B. MEMORY-OPTIMAL GOSSIP PROTOCOL (MO-GOSSIP)

Protocol 2 (MO-gossip) solves Q-gossip in system S. The main feature of this protocol is that, per process, it only needs linear memory on the size of the system, i.e. $O(|\Pi|)$, which is the minimum required to be able to store the rumors from every process in the system. However, this has some drawbacks that need to be mitigated as it will be explained. Since MO-gossip is event-driven, it keeps some advantages of the previous protocol. Let us start by explaining its main features.

Each process p_i uses an array named $known_i$ of size $|\Pi|$ to store the rumors it knows. Additionally, it uses an array named by_i of size $|\mathcal{N}_i| + 1$ to annotate how many of these rumors each of its neighbors and itself know, such that, for all process $p_j \in \mathcal{N}_i$, $by_i[j] \leq by_i[i]$. For each process $p_j \in \mathcal{N}_i$, process p_i knows that p_j already knows the rumors $known_i[0]$, $\ldots$, $known_i[by_i[j]-1]$, i.e., p_j knows the first $by_i[j]$ rumors in $known_i$. For the sake of simplicity, in an abuse of notation, $known_i[x..y]$ is used to denote the set which contains the elements $known_i[x]$, $known_i[x+1]$, $\ldots$, $known_i[y]$ (see, for example, Line 14). Likewise, $known_i$ is sometimes used as the set which contains the elements $known_i[0]$, $known_i[1]$, $\ldots$, $known_i[by_i[i]]$ (see, for example, Line 9).

To keep the knowledge about the rumors known by each of its neighbors up to date while using linear memory to store these rumors, a push-pull model is used for communications. Thus, when p_i receives a *SPREAD* message from some process $p_k \in \mathcal{N}_i$, p_i responds by sending an *OK* message to p_k. This *OK* message serves two purposes: it keeps p_k up to date with p_i, and it tells p_k that p_i is still active (i.e., it is not crashed). Thus, even if p_i and p_k are already up to date with each other, an *OK* message is sent to p_k with an empty set of rumors. Each process p_i uses variable P_i to store the set of processes from which an *OK* message is pending, and variable S_i to store the set of processes to which a *SPREAD* message must eventually be sent. Thus, process p_i will choose the candidate p_j to which to send a *SPREAD* message among the processes in S_i (but never one in P_i), since it would be of no use if p_j is crashed. Once the *OK* message from a process in P_i is received, it can be considered again as a candidate to which to send a *SPREAD* message (moving it to S_i).

As in the previous protocol, the **when** clauses are executed atomically and there is a minimum lapse of time τ between two *SPREAD* actions (see Lines 6, 31 and 38), which allows to reduce the number of messages exchanged. Additionally, when process p_i sends a *SPREAD* message to p_k, p_k is included in P_i (Line 36), so p_i will not consider p_k as a candidate to be sent a new *SPREAD* message (see Lines 17 and 29) until p_i receives an *OK* message from p_k (Lines 19 and 20).

When a *SPREAD* message is received from p_k, the rumors communicated by p_k which are not already known to p_i are added to $known_i$ (Lines 8–13). Then, in order to keep the knowledge on rumors up to date between p_i and p_k, the rumors known by p_i but not already considered to be known by p_k are sent to p_k in an *OK* message (Lines 14–15) and, then, they are considered known by p_k (Line 16).

When an *OK* message is received from p_k (Line 19), the rumors sent by p_k which are not already known by p_i are added to $known_i$ (Lines 21–25). Note that some of the rumors in $known_i[by_i[k]..by_i[i]-1]$ might be also known by p_k (at least those in the message being processed). Unfortunately, due to the memory management scheme used by the application, no gaps may exist for any process in the array that contains the rumors, so it is only possible to increment $by_i[k]$ while the rumor in $known_i[by_i[k]]$ is one of those in the *OK* message (Lines 26–28).

To be correct, protocol MO-gossip must fulfill the three properties specified in Definition 2: validity, quiescence and uniform agreement. Its correctness will be proved below.

Lemma 5: For each process $p_i \in \Pi$ running protocol MO-gossip, $known_i$ contains only initial rumors of processes in Π.

Proof: For all process $p_i \in \Pi$, initially, $known_i$ contains the initial rumor of p_i. Hence, it only contains initial rumors. By induction, assume that, for all process $p_i \in \Pi$, $known_i$ only contains initial rumors at some instant of time. Then, when it is updated in Lines 9–13 or Lines 21–25, the new values added are those sent at Lines 15 or 34 (recall that links are reliable in System S). These values are rumors previously stored in $known_j$ for some $p_j \in \Pi$ (see Lines 14–15 and 33–34) which, from the induction hipothesis, only contain initial rumors. Therefore, $known_i$ still contains only initial rumors after each update.

Corollary 4: All rumors known by a process running protocol MO-gossip must be initial rumors.

Proof: The rumors known by a process $p_i \in \Pi$ are those stored in $known_i$. Hence, from Lemma 5, all rumors known by a process running protocol MO-gossip must be initial rumors.

Corollary 5: For each process $p_i \in \Pi$ running protocol MO-gossip, there is a time after which no more rumors are added to $known_i$.

Proof: Follows directly from Lemma 5 and the fact that Π is finite (as well as the set of initial rumors).

Lemma 6: For each process $p_i \in \mathcal{C}$ running protocol MO-gossip, there is a time t after which S_i becomes permanently empty.

Proof: By the way of contradiction, assume that there is no time t after which S_i becomes permanently empty. Then, there must be a process p_j which never leaves S_i or which gets into S_i an infinite number of times. Each time the event of Line 31 occurs, one process p_k leaves S_i (Line 35), i.e. every τ units of time, a process leaves S_i. Therefore, if some process is never chosen in Line 32, then there must be at least another process in S_i every time Line 32 is executed. Therefore, both cases reduce to the case where a process p_j gets into S_i an infinite number of times. However, if p_j gets into S_i, then $by_i[j] < by_i[i]$ (see Lines 17 and 29). Since $by_i[j]$ is set to $by_i[i]$ in Line 37 each time it is chosen in Line 32 and it is never decremented, then $by_i[i]$ must grow infinitely. However, $by_i[i]$ is only incremented when new rumors are

Protocol 2 MO-gossip(r_i)

```
Input: rumor of process $p_i$
 1: $known_i[0] \leftarrow (r_i, i)$
 2: $by_i[i] \leftarrow 1$
 3: $by_i[k] \leftarrow 0, \forall p_k \in \mathcal{N}_i$
 4: $S_i \leftarrow \mathcal{N}_i$
 5: $P_i \leftarrow \emptyset$
 6: $T \leftarrow$ currentTime()
 7: start Task Push-Pull()

Task: Push-Pull()
 8: when $(SPREAD, rumors)$ received from $p_k$ do
 9:     $updates \leftarrow (rumors \setminus known_i)$
10:     for all $(r_q, q) \in updates$ do
11:         $known_i[by_i[i]] \leftarrow (r_q, q)$
12:         $by_i[i] \leftarrow by_i[i] + 1$
13:     end for
14:     $unknown \leftarrow known_i[by_i[k]..by_i[i] - 1] \setminus rumors$
15:     $send((OK, unknown), p_k)$
16:     $by_i[k] \leftarrow by_i[i]$
17:     $S_i \leftarrow \{p_j \in \mathcal{N}_i : by_i[j] < by_i[i] \wedge p_j \notin P_i)\}$
18: end when
19: when $(OK, rumors)$ received from $p_k$ do
20:     $P_i \leftarrow P_i \setminus \{p_k\}$
21:     $updates \leftarrow (rumors \setminus known_i)$
22:     for all $(r_q, q) \in updates$ do
23:         $known_i[by_i[i]] \leftarrow (r_q, q)$
24:         $by_i[i] \leftarrow by_i[i] + 1$
25:     end for
26:     while $(by_i[k] < by_i[i] \wedge known_i[by_i[k]] \in rumors)$ do

27:         $by_i[k] \leftarrow by_i[k] + 1$
28:     end while
29:     $S_i \leftarrow \{p_j \in \mathcal{N}_i : by_i[j] < by_i[i] \wedge p_j \notin P_i\}$
30: end when
31: when currentTime() $\geq T \wedge S_i \neq \emptyset$ do
32:     let $p_k$ be a random process in $S_i$
33:     $unknown \leftarrow known_i[by_i[k]..by_i[i] - 1]$
34:     $send((SPREAD, unknown), p_k)$
35:     $S_i \leftarrow S_i \setminus \{p_k\}$
36:     $P_i \leftarrow P_i \cup \{p_k\}$
37:     $by_i[k] \leftarrow by_i[i]$
38:     $T \leftarrow$ currentTime() $+ \tau$
39: end when
```

added to $known_i$ (see Lines 9–13 and Lines 21–25) and, from Corollary 5, there is a time after which no more rumors are added to $known_i$.

Lemma 7: Eventually, every process running protocol MO-gossip stops sending messages permanently.

Proof: By the way of contradiction, assume that there is a process p_i which sends an infinite number of messages. Since *OK* messages are sent in Line 15 as a consequence of the reception of a *SPREAD* message (see Line 8), an infinite number of *SPREAD* messages must be sent. Since these messages are sent in Line 34, the condition of Line 31 must be fulfilled an infinite number of times. Therefore, $S_i \neq \emptyset$ an infinite number of times. However, that is not possible from Lemma 6.

The following remark states an invariant satisfied by the **when** clauses in protocol MO-gossip. This invariant holds both before the execution of a **when** clause starts, and when it ends. Since these clauses are executed atomically, these invariants are protocol invariants.

Remark 6: For each correct process p_i running protocol MO-gossip, for all $p_j \in \mathcal{N}_i$, $by_i[j] < by_i[i]$ implies that either $p_j \in S_i$ or $p_j \in P_i$.

Proof: Note that $by_i[i]$ is set to 1 in Line 2, while $by_i[j]$ is set to 0 in Line 3 for all $p_j \in \mathcal{N}_i$. Then, in Line 4 S_i is set to $\mathcal{N}_i$, and P_i is set to $\emptyset$ in Line 5. Hence, the claim holds in the beginning. It is easy to see that Lines 17 and 29 guarantee that the claim holds when the execution of the clauses of Lines 8–18 and Lines 19–30 end their execution. In the case of the clause of Lines 31–39, since the values stored in by_i are never decremented and $by_i[i]$ is not increased in this clause, if a process $p_j \in \mathcal{N}_i$ was not in S_i when the clause started executing, then $by_i[j] >= by_i[i]$ after the clause ends its execution. Therefore, if the claim is satisfied before the clause is executed, then it must hold after it is executed.

For the sake of clarity, to prove *uniform agreement*, some preliminary results will be presented below in an incremental way.

Remark 7: For each correct process p_i running protocol MO-gossip, there is a time after which, for all $p_j \in \mathcal{N}_i$, $by_i[j] = by_i[i]$ forever or $p_j \in P_i$ permanently.

Proof: Note that the values stored in by_i are never decremented, so they can only grow. Furthermore, every time $by_i[j]$ is increased (for any $p_i \in \mathcal{N}_i$), it never exceeds the value of $by_i[i]$ (See Line 16, Lines 26–28 and Line 37). Therefore, $by_i[j] \leq by_i[i]$. From Lemma 6, there is a time t after which S_i is permanently empty. Recall that, from Remark 6, for all $p_j \in \mathcal{N}_i$, $by_i[j] < by_i[i]$ implies that either $p_j \in S_i$ or $p_j \in P_i$. Therefore, if S_i is permanently empty after time t, then, after time t, for all $p_j \in \mathcal{N}_i$, $p_j \in P_i$ permanently or $by_i[j] = by_i[i]$.

Remark 8: Let p_i be a correct process running protocol MO-gossip. If $p_j \in P_i$ permanently, then $p_j \in \mathcal{F}$), i.e. $p_j \notin \mathcal{C}$).

Proof: By the way of contradiction, assume that $p_j \in \mathcal{C}$. Note that p_j is added to P_i in Line 36 after a *SPREAD* message is sent (by p_i) to p_j. Since the links are reliable and both $p_i, p_j \in \mathcal{C}$ from the initial assumption, then p_j will eventually receive that message and, hence, it will execute Line 15, where it will send an *OK* message back to p_i. Again, since links are reliable both processes are correct, p_i will eventually receive that *OK* message. Then, p_i will execute Line 20, and p_j will be removed from P_i. Therefore, p_j cannot be in P_i permanently unless p_j is not correct.

Remark 9: For all $p_i \in \mathcal{C}$ and $p_j \in (\mathcal{N}_i \cap \mathcal{C})$ running protocol MO-gossip, there is a time after which $by_i[i] = by_i[j]$ permanently.

Proof: From Remark 7, there is a time after which, for all $p_j \in \mathcal{N}_i$, $by_i[j] = by_i[i]$ forever or $p_j \in P_i$ permanently. Furthermore, from Remark 8, if $p_j \in P_i$ permanently, then $p_j \notin \mathcal{C}$. Therefore, if p_i and p_j are both correct, then there is a time after which $by_i[i] = by_i[j]$ permanently.

Remark 10: Let p_i be a correct process running protocol MO-gossip. For all $p_k \in (\mathcal{N}_i \cap \mathcal{C})$, $(r_j, j) \in known_i[0..by_i[k]-1]$ implies that either (r_j, j) is already in $known_k$ or it will eventually be in $known_k$.

Proof: A rumor is included in $known_i[0..by_i[k]-1]$ by increasing the value of $by_i[k]$. Consider the moment when (r_j, j) is added to $known_i[0..by_i[k]-1]$. There are three cases in which this can happen.

First, in Line 16, when an *OK* message is sent to p_k (Line 15) with the rumors in $known_i[by_i[k]..by_i[i]-1] \setminus rumors$ (see Line 14) after a *SPREAD* message has been received from p_k (Lines 8–18). If (r_j, j) was in *rumors*, then it must have been in $known_k$, since these rumors must have been sent in Line 34 where a subset of $known_k$ was sent by p_k to p_i (recall that links are reliable, so messages cannot be lost nor altered in transmission). If it was not in *rumors*, then it must have been in $known_i[by_i[k]..by_i[i]-1]$. Then, since it is sent to p_k in the *OK* message (Line 15) and the links are reliable, this message will be eventually received by p_k and processed in Lines 8–18, where (r_j, j) will be added to $known_k$ in Lines 10–13, in case it was not already in $known_k$.

Second, in Line 27, when an *OK* message from p_k is being processed. Notice that, in this case, (r_j, j) must come in *rumors* (see Lines 26–28). Hence, since links are reliable, this rumor was already in $known_k$ when the *OK* message was sent by process p_k in Line 15.

Finally, the third case corresponds to Line 37. In this case, all the rumors in $known_i[by_i[k]..by_i[i]-1]$, including (r_j, j), are sent to p_j in a *SPREAD* message which will be received by p_k unaltered (since the links are reliable) and processed in Lines 10–13, where they will be stored in $known_k$. Hence, if (r_j, j) was not already in $known_k$, it will eventually be.

Remark 11: For each correct process running protocol MO-gossip, eventually, for all $p_k \in (\mathcal{N}_i \cap \mathcal{C})$, $known_i = known_k$ permanently.

Proof: From Remark 10, $(r_j, j) \in known_i$ implies that either (r_j, j) is already in $known_k$ or it will eventually be in $known_k$. Therefore, for all $p_k \in (\mathcal{N}_i \cap \mathcal{C})$, eventually, $known_i \subseteq known_k$. Since the same argument can be applied to p_i with respect to p_k, eventually, $known_k \subseteq known_i$. Furthermore, from Remark 9, eventually, for all process $p_k \in (\mathcal{N}_i \cap \mathcal{C})$, $by_i[i] = by_i[k]$ permanently, i.e., $known_i[0..by_i[k]-1] = known_i[0..by_i[i]-1] = known_i$. Hence, eventually, $known_i = known_k$ permanently.

Lemma 8: Eventually, all correct processes running protocol MO-gossip know the same set of rumors, and this set contains the initial rumors of all correct processes.

Proof: Consider any pair of neighbors $p_i, p_j \in \mathcal{C}$. From Remark 11, eventually, $known_i = known_j$ permanently. Note that $known_i$ initially contains (r_i, i) and $known_j$ initially contains (r_j, j) (see Lines 1–2). Furthermore, they only grow (see Lines 11–12 and Lines 23–24). Therefore, eventually, (r_i, i) and (r_j, j) belong to both $known_i$ and $known_j$. From Property 1, there is a path between every two correct processes in $G_{\mathcal{C}}$, so this argument can be extended to cover every process in $\mathcal{C}$. Hence, it can be concluded that, eventually, all correct processes know the same set of rumors and this set contains the initial rumors of all correct processes.

Theorem 4: Protocol 2 (MO-gossip) solves Q-gossip.

Proof: Direct from Corollary 4, Lemma 7 and Lemma 8.

Since there is a trade-off between network traffic and memory space, protocol MO-gossip requires more information to be exchanged than protocol BE-gossip. Recall that, since the system is asynchronous, time analysis does not apply. Thus, only the number of rumors and messages exchanged is considered. There are two types of messages: *SPREAD* and *OK*. *SPREAD* messages always carry at least one rumor. Note that S_i only contains a process p_j if $by_i[j] < by_i[i]$ (see Lines 17, 29 and 32–37). However, *OK* messages might not carry any rumor since, as soon as a *SPREAD* message is received, the corresponding *OK* message is sent back (see Lines 14–15) and there might not be any novelties to notify.

Theorem 5: The number of rumors exchanged until quiescence by protocol MO-gossip is $O(|\Pi| \times |\mathcal{L}|)$ and $\Omega(|\mathcal{C}| \times |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|)$.

Proof: Note that each rumor is sent through a link at most once in each direction. This is so because each time a *SPREAD* message is sent, its rumors are recorded as already known by the recipient (see Lines 32–37) and, in the case of an *OK* message, its rumors are also recorded (see Lines 14–16). Hence, in the worst case, each rumor can traverse each link in two concurrent *SPREAD* messages. However, it is not possible to have, in the same link, a *SPREAD* message in one direction and a concurrent *OK* message in the other direction, since *SPREAD* messages are not sent to processes from which an *OK* message is pending (see Lines 32–37). When the *OK* message from a process is received, it is removed from P_i (Line 20). Finally, each time S_i is recomputed, the processes in P_i are excluded (see Lines 17 and 29). Hence, the maximum number of rumors sent is $2 \times |\Pi| \times |\mathcal{L}| - |\Pi|$ (recall that the first time a process p_i sends its own rumor r_i, e.g. to some other process p_j, it is impossible that p_j can be sending r_i to p_i, because it does not know it yet). Therefore, the number of rumors sent is $O(|\Pi| \times |\mathcal{L}|)$. The most favorable is that in which all the faulty processes crash before sending any message, and each rumor is sent only once per link shared by at least one correct process. Therefore, the number of rumors sent is at least $|\mathcal{C}| \times |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|$, i.e., $\Omega(|\mathcal{C}| \times |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|)$.

The number of rumors exchanged until quiescence by protocols MO-gossip and BE-gossip are analogous. However, the number of messages is likely to increase. A coarse worst case bound on the number of messages can be considered to be the number of rumors exchanged, since each *SPREAD* message carries at least one rumor, plus the number of *OK* messages sent, since these *OK* messages might not carry any rumor.

Theorem 6: The maximum number of messages exchanged until quiescence by protocol MO-gossip is $O(|\Pi| \times |\mathcal{L}|)$.

Proof: Recall that the maximum number of rumors sent by the protocol is $2 \times |\Pi| \times |\mathcal{L}| - |\Pi|$. In the worst case, every *SPREAD* message carries only one rumor, while every *OK* message carries no rumor. Since each *SPREAD* message is followed by an *OK* message, then the maximum number of messages would be $2(2 \times |\Pi| \times |\mathcal{L}| - |\Pi|)$, i.e., $O(|\Pi| \times |\mathcal{L}|)$.

Note that the number of potential messages doubles with respect to protocol BE-gossip, but it remains in the same order. To find a lower bound on the number of messages exchanged in the best case, let us take advantage of the fact that the system is asynchronous, so the most advantageous sequence of messages can be selected.

Theorem 7: The minimum number of messages exchanged until quiescence by protocol MO-gossip is $\Omega(2(|\mathcal{C}|-1)+|(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})|-1+|\mathcal{L}_{\mathcal{C}}|)$.

Proof: Consider that all the faulty processes crash before sending any messages, since in this way the information to be disseminated is minimal. Since $G_{\mathcal{C}}$ is connected (from Property 1), there is a spanning tree that connects the correct processes. Therefore, it is possible to collect all the rumors and carry them to the root of that spanning tree with $|\mathcal{C}| - 1$ *SPREAD* messages. Upon reception of each *SPREAD* message, an *OK* message is immediately sent back. Thus, $|\mathcal{C}| - 1$ *OK* messages are sent back with partial knowledge of rumors, what makes a total of $2 \times (|\mathcal{C}| - 1)$ messages. At most, one of the *OK* messages sent will give complete information on rumors. Therefore, $|(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})| - 1$ *SPREAD* messages will be sent. Consequently, $|\mathcal{L}_{\mathcal{C}}|$ *OK* messages will be sent back (without information on rumors). Thus the minimum number of messages is $\Omega(2(|\mathcal{C}| - 1) + |(\mathcal{L} \setminus \mathcal{L}_{\mathcal{F}})| - 1 + |\mathcal{L}_{\mathcal{C}}|)$.

Comparing the minimum number of messages needed by both protocols, it is easy to see that protocol MO-gossip requires almost double the messages needed by protocol BE-gossip. This shows the trade-off between memory space used and messages exchanged. However, the memory requirements are reduced from quadratic to linear, while the number of rumors stays the same and the number of messages only doubles, so it's worth it.

IV. EXPERIMENTAL EVALUATION

Despite the theoretical interest in the upper and lower limits of the number of packets sent, practical performance is of great importance to evaluate the feasibility of protocol deployments in real scenarios. In fact, in asynchronous systems, where time limits cannot be determined theoretically, performance can only be evaluated in an appropriate way through experiments. This section compares the practical performance of both proposed protocols in a particular simulated scenario.

The scenario we consider is a network with a particular topology. It is well known that network topology (which, incidentally, can change when node failures occur) affects the actual spread of rumors, even if exactly the same protocol is used. In order to more realistically capture the structure of a typical sensor network, we run our protocols on a small network whose degree distribution is that of a random graph [32], [33], [34] that is "spatialized" to ensure that nodes connect to nodes that are spatially (or geographically) neighbors. We build the network by generating first a classical random graph (with the desired number of nodes and undirected edges) [35], then rewiring the nodes in such a way that the length of the new edges is as short as possible, and finally spliting each undirected edge joining two nodes, say node A and node B, into two directed edges, the edge going from A to B and the edge going from B to A. The second step, the rewiring of nodes, is carried out following a procedure similar to the one presented in [36], with the only difference that the selected pair of edges is always that whose total length, geographically speaking, is the smallest. Our final, *random spatialized network* has 100 nodes and 600 directed edges (or 300 bidirectional edges).

On this network, we run our two protocols with different parameter values. Note that, depending on parameter values (such as the data rate at which nodes can transmit information or the time lapse between two *SPREAD* messages), the performance of the protocols may change significantly, even if they run on exactly the same network.

For simplicity, all network links are set to have the same data rate. In the simulations, four different data rates have been used: 10Mbps, 1Mbps, 100Kbps, and 10Kbps. We have used these low-medium rates because they are very common in many IoT networks with battery and memory constraints [37], [38], [39], [40], [41], [42]. The transmission delays of the edges are considered negligible, since we asume our sensor network to be spread in a small, geographical area. In addition, to avoid a systolic behavior, the times at which each process (node) sends its first *SPREAD* packet follow a Gaussian distribution with mean $\mu = 100\mu s$ and standard deviation $\sigma = 20\mu s$. As it is usual in current gossip protocols, we also set a waiting time between sending two consecutive *SPREAD* messages, but, since we focus on asynchronous systems, each node has a different waiting time, drawn from another Gaussian distribution with mean $\mu = \tau$ (where τ will change in our simulations from $\tau = 1\mu s$ to $\tau = 10^8\mu s$) and standard deviation $\sigma = 0.2\tau$. For simplicity, we consider that the capacity of each link buffer is sufficient to avoid packet loss. All rumors, again for simplicity, have the same size (8 bytes). The size of the packets' header, given that protocol MO-gossip needs to accomodate the *OK*, is different for each protocol: BE-gossip header has 8 bytes, while MO-gossip header has 12 bytes.

Regarding node failures, only two cases are considered: a first case where only a few nodes fail and a second case where a large number of nodes fail. In both cases, nodes are allowed to fail, following a uniform distribution, along a period of time ranging from $0\mu s$ to $10^4\mu s$. When a node fails, we make sure that the subnetwork induced by the correct nodes remains connected, so that Property 1 always holds.

The simulations have been carried out in a custom simulator [43], [44] written in Fortran. To validate the

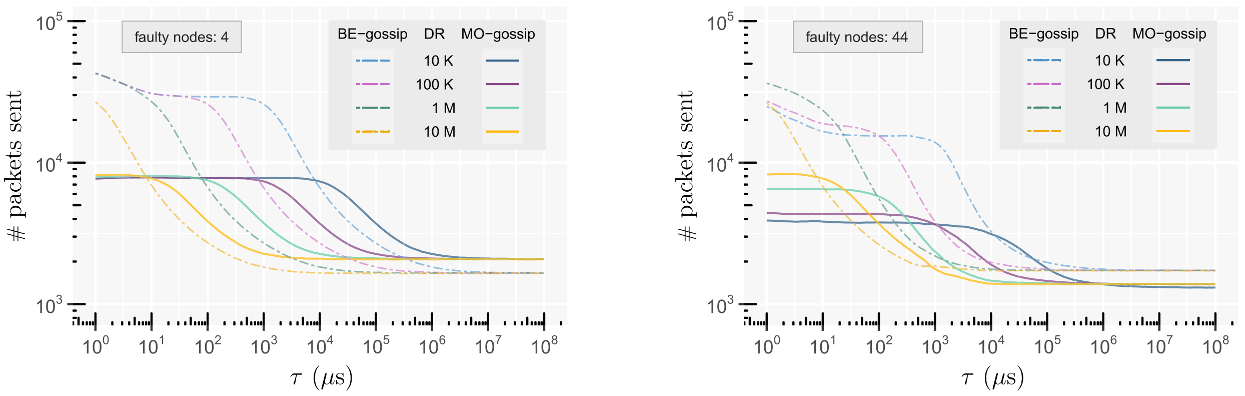

FIGURE 1. **Number of packets sent as a function of τ, for each protocol and data rate. Left panel: Low number of faulty processes (4 out of the initial 100). Right panel: High number of faulty processes (44 out of the initial 100).**

accuracy of this simulator, some of the experiments have also been carried out in a customized version of the discrete event simulator Peersim [45], [46], which is much slower, but guarantees that the results obtained are correct.

Since the simulations have a stochastic component, 1000 simulations have been carried out for each configuration (different data rate and τ value), and the results shown in the graphs correspond to the averages obtained. Deviations from the average were small, so they are not shown in the graphs for clarity. The results we focus on are the number of packets sent and the time required to reach quiescence.

A. PACKETS VS. τ

Figure 1 shows the total number of packets transmitted by both protocols as a function of τ, for the four data rates considered. On the left, the number of faulty nodes is 4, while, on the right, the number of faulty nodes is 44.

For high values of τ, the number of packets sent is almost the same for each protocol, regardless of the data rate. This is due to the fact that, when τ is high enough, every node receives the messages of its neighbors before it sends its next packet. Hence, for large values of τ, the system becomes systolic and the information collected by each node before sending the next *SPREAD* message does not depend noticeably on the value of τ, nor on the data rate.

Comparing the results for 4 and 44 faulty nodes for high values of τ, it is easy to see that, while in the case of 4 faulty nodes, Protocol BE-gossip sends fewer packets than Protocol MO-gossip, in the case of 44 faulty nodes, the situation is reversed and MO-gossip sends fewer packets than BE-gossip. In fact, BE-gossip sends slightly more messages when there are 44 faulty nodes than when there are only 4, even though faulty nodes send less. The reason is that the correct nodes continue to send messages to the faulty nodes, which do not cooperate in spreading rumors, and the correct nodes must insist on sending rumors to other nodes. Since MO-gossip follows the push-pull scheme, one node can choose a crashed neighbor only once as the recipient of a *SPREAD* message which results in reducing the number of sent messages to nodes that crashed. Therefore, it is more effective than BE-gossip and sends around one third less messages when 44 nodes crash than when only 4 nodes crash.

In the case of MO-gossip, for each data rate, there is a value of τ below which the number of packets sent does not change appreciably. Recall that, if a process sends a *SPREAD* message to one of its neighbors, it will not send another until it receives an *OK* message from that neighbor. Thus, if τ is smaller than the round trip time, the speed at which *SPREAD* messages are sent is determined by the data rate, and not by the value of τ. Since BE-gossip does not have any contention mechanism that prevents it from generating more and more packets, the number of packets it sends always increases as τ decreases, although not always in the same proportion.

Comparing the results for 4 and 44 faulty nodes for low values of τ, in the case of MO-gossip, there is a reduction on the number of packets sent in the case of 44 faulty nodes, which is larger for smaller data rates, and almost unnoticeable for 10Mbps. Recall that faulty nodes crash following a uniform distribution during the first $10^4\mu$s. Note also that quiescence time for the case of 10Mbps, as shown in Figure 2, when $\tau < 10^2\mu$s is much smaller than 10^4. Therefore, only a few nodes crash before quiescence, so the behavior of MO-gossip is almost the same as in the case of only 4 faulty nodes. As the data rates decrease, the time until quiescence increases (as shown in Figure 2) and the number of nodes that crash before quiescence increases, what leads to a reduction on the number of packets sent, as shown in Figure 1, right panel.

For intermediate values of τ (between push-pull contention and systolic behavior), MO-gossip sends more packets when the value of τ decreases (Figure 1) while time until quiescence decreases (Figure 2). This fact clearly illustrates the trade-off between packets sent and quiescence time, which is best shown in Figure 3.

B. TIME VS. τ

Figure 2 shows how the time to quiescence varies as a function of τ, for the four data rates considered. Again, on the

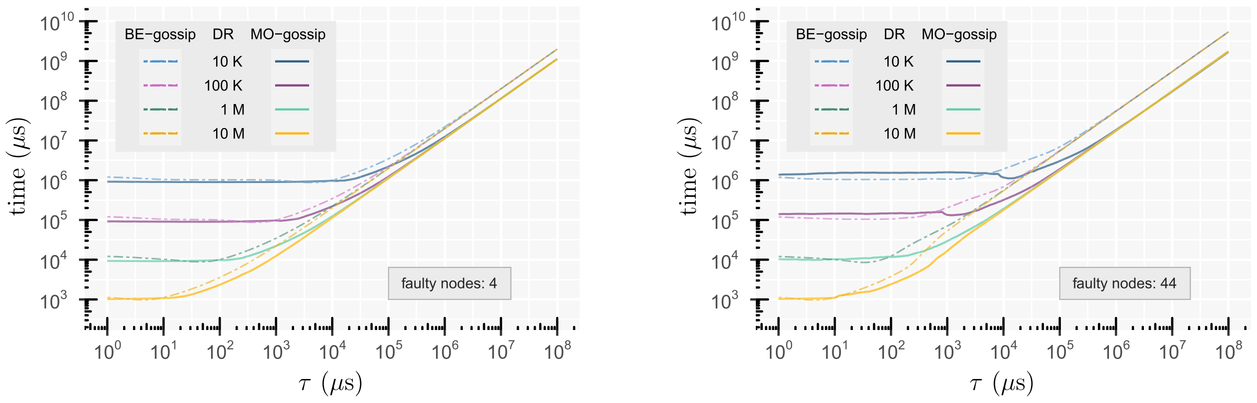

FIGURE 2. Quiescence time as a function of τ, for each protocol and data rate. Left panel: Low number of faulty processes (4 out of the initial 100). Right panel: High number of faulty processes (44 out of the initial 100).

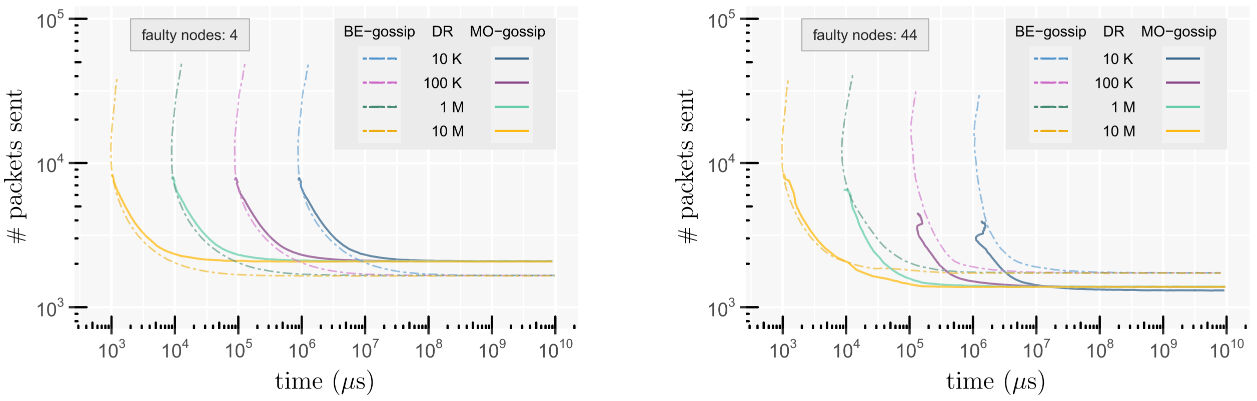

FIGURE 3. Number of packets sent as a function of Quiescence time, for each protocol and data rate. Left panel: Low number of faulty processes (4 out of the initial 100). Right panel: High number of faulty processes (44 out of the initial 100).

left panel, the number of faulty nodes is 4, while, on the right panel, the number of faulty nodes is 44.

The most notable difference between Figure 2 and Figure 1 is that, while for small values of τ there is no clear dependence between the number of packets sent and the data rate, the time to quiescence clearly depends inversely on the data rate.

For cases in which only a few nodes fail (left panel), MO-gossip is better than BE-gossip for most values of τ. For all data rates, the minimum time is always reached by BE-gossip. However, that minimum is just slightly better than the time reached by MO-gossip and the value of τ at which the minimum is located cannot be easily computed analytically, since it depends not only on the data rate selected but also on the very topology of the network.

For cases in which a lot of nodes fail (right panel), both protocols behave similarly to the case where only a few nodes fail, but now protocol MO-gossip improves the performance of protocol BE-gossip only for large values of τ, and its performance is slightly worse both for small and moderate values of τ. We must remark, however, that the optimum time of protocol BE-gossip is just slightly better than the time of protocol MO-gossip corresponding to the smallest value of τ, ($\tau = 1\mu$s).

Both panels show, however, that the differences in time between both protocols are not very significant, with the exception of the difference when τ is large (and also quiescence times), where MO-gossip is clearly better than BE-gossip.

C. PACKETS VS. TIME

Finally, the trade-off between quiescence time and packets sent is shown in Figure 3 for the case in which the number of faulty nodes is low (left panel) and for the case of a large number of faulty nodes (right panel). The dots in the curves are ordered according to the value of τ to which they correspond.

In the case of BE-gossip, the curves have a very similar shape both in the case of few faulty nodes and in the case of many faulty nodes, although in the second case the minimum time until quiescence is obtained with a slightly larger number of packets. It is noteworthy that, as the value of τ decreases, the time also decreases (while the number of packets increases), until a point where the trend changes and the time increases (while the number of packets continues to grow). As discussed above, it is not easy to analytically determine the value of τ for which the time is optimal.

The most striking thing about the curves for Protocol MO-gossip is that they are much shorter than those corresponding to BE-gossip. This is because, below a certain value of τ, both the time and the number of packets remain almost constant. Therefore, there are many coincident points on those curves. Small variations are only observed in the case of many faulty nodes for the lowest data rates.

The most notable result that can be obtained from these experiments is that Protocol MO-gossip achieves, for arbitrarily small values of τ, quiescence times comparable to the best of BE-gossip, but with a number of packets that is similar when the number of faulty devices is low and noticeably lower when the number of faulty devices is high. Thus, MO-gossip shows that memory optimization can be achieved without losing performance and demonstrating remarkable resilience: when there are many faulty devices in the network, not only is its performance not degraded but it even improves in some cases.

V. CONCLUSION

In a network of small IoT devices that have limited memory and battery capacity, traditional gossip protocols are unable to efficiently disseminate information, specially when the number of devices or the number of required messages is large.

Gossip protocols for IoT networks are found in the literature where devices are allowed to fail and operate asynchronously. However, this paper is the first to also consider IoT devices with limited battery and memory. Two gossip-based protocols have been proposed and their correctness has been formally proved. They solve the uniform quiescent gossip problem when the correct devices form a connected network.

Protocol BE-gossip allows to reduce battery consumption following a push scheme. Like traditional versions, it requires quadratic memory in the devices. Protocol MO-gossip follows the push-pull scheme. Thus, it only needs linear memory to store and manage rumors, making it a memory-optimal protocol. MO-gossip is also capable of significantly reducing the number of messages sent (compared to BE-gossip), while yielding similar quiescence times, independently of the number of faulty processes.

It is noteworthy that MO-gossip shows that memory optimization is not incompatible with achieving good performance even in the case of device failures. In fact, failure detection can reduce both the number of messages sent and quiescence time. Hence, MO-gossip presents very good resilience, which is a notable feature in current IoT networks. Finally, the performance of MO-gossip does not depend on the value of the parameter τ, provided that its value is arbitrarily small enough.

REFERENCES

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 41–88, May 1999.

[2] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson, "Epidemic algorithms for replicated database maintenance," in *Proc. 6th Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*. New York, NY, USA: ACM, 1987, pp. 1–12, doi: 10.1145/41840.41841.

[3] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," *ACM Trans. Comput. Syst.*, vol. 21, no. 4, pp. 341–374, Nov. 2003.

[4] D. Gavidia, S. Voulgaris, and M. van Steen, "A gossip-based distributed news service for wireless mesh networks," in *Proc. 3rd Annu. Conf. Wireless On-Demand Netw. Syst. Services*, 2006, pp. 59–67.

[5] R. Tijdeman, "On a telephone problem," *Nieuw Archief Voor Wiskunde*, vol. 3, no. 19, pp. 188–192, 1971.

[6] N. T. Bailey, *The Mathematical Theory of Infectious Diseases and its Applications*. High Wycombe, England: Charles Griffin & Company, 1975.

[7] G. Giakkoupis and T. Sauerwald, "Rumor spreading and vertex expansion," in *Proc. 23rd Annu. ACM-SIAM Symp. Discrete Algorithms*, Y. Rabani, Ed., Kyoto, Japan. Philadelphia, PA, USA: SIAM, Jan. 2012, pp. 1623–1641, doi: 10.1137/1.9781611973099.129.

[8] B. Haeupler, "Simple, fast and deterministic gossip and rumor spreading," *J. ACM*, vol. 62, no. 6, pp. 1–18, Dec. 2015.

[9] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman, "A survey of gossiping and broadcasting in communication networks," *Networks*, vol. 18, no. 4, pp. 319–349, Dec. 1988, doi: 10.1002/net.3230180406.

[10] J. Hromkovič, R. Klasing, B. Monien, and R. Peine, *Dissemination of Information in Interconnection Networks (Broadcasting & Gossiping)*. Boston, MA, USA: Springer, 1996, pp. 125–212.

[11] B. Pittel, "On spreading a rumor," *SIAM J. Appl. Math.*, vol. 47, no. 1, pp. 213–223, Feb. 1987, doi: 10.1137/0147013.

[12] J. Liu, S. Mou, A. S. Morse, B. D. O. Anderson, and C. Yu, "Deterministic gossiping," *Proc. IEEE*, vol. 99, no. 9, pp. 1505–1524, Sep. 2011.

[13] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE Trans. Autom. Control*, vol. AC-31, no. 9, pp. 803–812, Sep. 1986.

[14] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE Trans. Inf. Theory*, vol. 52, no. 6, pp. 2508–2530, Jun. 2006, doi: 10.1109/TIT.2006.874516.

[15] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking, "Randomized rumor spreading," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, Redondo Beach, CA, USA, 2000, pp. 565–574, doi: 10.1109/sfcs.2000.892324.

[16] M. Jelasity, "Gossip," in *Self-Organising Software—From Natural to Artificial Adaptation*. Germany: Springer, 2011, pp. 139–162, doi: 10.1007/978-3-642-17348-6.

[17] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, no. 1, pp. 77–97, Jan. 1987, doi: 10.1145/7531.7533.

[18] A. Basu, B. Charron-Bost, and S. Toueg, "Simulating reliable links with unreliable links in the presence of process crashes," in *Distributed Algorithms*, Ö. Babaoğlu and K. Marzullo, Eds. Berlin, Germany: Springer, 1996, pp. 105–122.

[19] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996, doi: 10.1145/226643.226647.

[20] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, May 2004, doi: 10.1109/MC.2004.1297243.

[21] C. Georgiou, S. Gilbert, R. Guerraoui, and D. R. Kowalski, "Asynchronous gossip," *J. ACM*, vol. 60, no. 2, pp. 1–42, Apr. 2013, doi: 10.1145/2450142.2450147.

[22] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Comput. Netw.*, vol. 38, no. 4, pp. 393–422, 2002, doi: 10.1016/S1389-1286(01)00302-4.

[23] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.

[24] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta, "Computation in networks of passively mobile finite-state sensors," *Distrib. Comput.*, vol. 18, no. 4, pp. 235–253, Mar. 2006, doi: 10.1007/s00446-005-0138-3.

[25] Y. Cho, M. Kim, and S. Woo, "Energy efficient IoT based on wireless sensor networks," in *Proc. 20th Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2018, pp. 294–299.

[26] S. Lee, M. Bae, and H. Kim, "Future of IoT networks: A survey," *Appl. Sci.*, vol. 7, no. 10, p. 1072, Oct. 2017.

[27] F. Samie, L. Bauer, and J. Henkel, "IoT technologies for embedded computing: A survey," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2016, pp. 1–10.

[28] S. Kar and J. M. F. Moura, "Sensor networks with random links: Topology design for distributed consensus," *IEEE Trans. Signal Process.*, vol. 56, no. 7, pp. 3315–3326, Jul. 2008.

[29] C. C. Moallemi and B. van Roy, "Consensus propagation," *IEEE Trans. Inf. Theory*, vol. 52, no. 11, pp. 4753–4766, Nov. 2006.

[30] R. Olfati-Saber and R. M. Murray, "Consensus problems in networks of agents with switching topology and time-delays," *IEEE Trans. Autom. Control*, vol. 49, no. 9, pp. 1520–1533, Sep. 2004.

[31] L. Xiao, S. Boyd, and S. Lall, "A scheme for robust distributed sensor fusion based on average consensus," in *Proc. 4th Int. Symp. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2005, pp. 63–70.

[32] P. Erdös and A. Rényi, "On random graphs. I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, Jun. 1959.

[33] B. Bollobás, *Random Graphs* (Cambridge Studies in Advance Mathematics), 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2001.

[34] E. N. Gilbert, "Random graphs," *Ann. Math. Statist.*, vol. 30, no. 4, pp. 1141–1144, 1959.

[35] M. E. J. Newman, *Networks: An Introduction*. New York, NY, USA: Oxford Univ. Press, 2010.

[36] S. Maslov, K. Sneppen, and A. Zaliznyak, "Detection of topological patterns in complex networks: Correlation profile of the Internet," *Phys. A, Stat. Mech. Appl.*, vol. 333, pp. 529–540, Feb. 2004, doi: 10.1016/j.physa.2003.06.002.

[37] C. C. Byers, "Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled IoT networks," *IEEE Commun. Mag.*, vol. 55, no. 8, pp. 14–20, Aug. 2017.

[38] J. de Carvalho Silva, J. J. Rodrigues, A. M. Alberti, P. Solic, and A. L. Aquino, "LoRaWAN—A low power wan protocol for Internet of Things: A review and opportunities," in *Proc. 2nd Int. Multidisciplinary Conf. Comput. Energy Sci. (SpliTech)*, Jul. 2017, pp. 1–6.

[39] A. El Hakim, "Internet of Things (IoT) system architecture and technologies," Researchgate, Berlin, Germany, White Paper, vol. 10, 2018, doi: 10.13140/RG.2.2.17046.19521.

[40] R.-A. Koutsiamanis, G. Z. Papadopoulos, X. Fafoutis, J. M. D. Fiore, P. Thubert, and N. Montavont, "From best effort to deterministic packet delivery for wireless industrial IoT networks," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4468–4480, Oct. 2018.

[41] Z. Qin, F. Y. Li, G. Y. Li, J. A. McCann, and Q. Ni, "Low-power wide-area networks for sustainable IoT," *IEEE Wireless Commun.*, vol. 26, no. 3, pp. 140–145, Jun. 2019.

[42] U. Raza, P. Kulkarni, and M. Sooriyabandara, "Low power wide area networks: An overview," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 2, pp. 855–873, 2nd Quart., 2017.

[43] C. Barroso-Fernáandez, E. Jiménez, J. L. López-Presa, M. Moreno-Cuesta, and R. Xulvi-Brunet. (2023). *Simulator Fortran*. [Online]. Available: https://github.com/resilient-gossiping/event-driven-simulator

[44] R. Xulvi-Brunet. (2023). *Simulator Fortran*. [Online]. Available: https://blog.epn.edu.ec/ramon_xulvi-brunet

[45] C. B. Fernández. (Jul. 2021). *Simulación, Análisis y Evaluación de Algoritmos Epidémicos*. No Publicado. [Online]. Available: https://oa.upm.es/70672/

[46] F. D. Muñoz. (Oct. 2020). *Optimización de Peersim: Un Simulador de Eventos Discretos Para Redes 5G*. No Publicado. [Online]. Available: https://oa.upm.es/68357/

CARLOS BARROSO-FERNÁNDEZ received the degree in telematics from Universidad Politécnica de Madrid, in 2021, and the M.Sc. degree in computational and applied mathematics from Universidad Carlos III de Madrid, in 2022, where he is currently pursuing the Ph.D. degree. His main research interests include applied mathematics and artificial intelligence.

ERNESTO JIMÉNEZ received the Graduate degree in computer science from Universidad Politécnica de Madrid, Madrid, Spain, in 1995, and the Ph.D. degree in computer science from Rey Juan Carlos University, Madrid, in 2004. He is currently an Associate Professor with Universidad Politécnica de Madrid. His research interests include fault tolerance in distributed systems, computer networks, and parallel and distributed processing.

JOSÉ LUIS LÓPEZ-PRESA received the B.S. degree in computer science from the University of the Basque Country (UPV/EHU), in 1987, and the Ph.D. degree (Hons.) from Rey Juan Carlos University (URJC), in 2009.

From 1987 to 2012, he was an Assistant Professor with the Technical University of Madrid (UPM). Since 2012, he has been an Associate Professor. His main research interests include computer networks, traffic engineering, distributed systems, fault-tolerant systems, parallel and distributed processing, and graph theory, with a special focus on graph isomorphism.

MARTA MORENO-CUESTA received the Graduate degree in telecommunication engineering from Universidad Politécnica de Madrid, Madrid, Spain, in 2005, where she is currently pursuing the Ph.D. degree in computer science. Her main research interests include computer science and technology applied to smart cities, with a special focus on fault tolerant distributed systems.

RAMON XULVI-BRUNET received the degree in physics from Universitat de València, Spain, and the Ph.D. degree in theoretical physics from Humboldt Universitaet zu Berlin, Germany. He was a Postdoctoral Researcher with The University of Sydney, Australia, Upenn, Harvard University, NECSI, and UCSB, USA. He is currently an Associate Professor with Escuela Politécnica Nacional, Ecuador. His research interests include complex systems, statistical physics, complex networks, systems biology, neuroscience, complexity economics, self-organized criticality, epidemiology, and artificial neural networks.