# Gossip and Epidemic Protocols

Alberto Montresor

**Abstract**

A *gossip protocol* is a distributed communication paradigm inspired by the gossip phenomenon that can be observed in social networks. Initially born to efficiently disseminate information, as its human counterpart, it has been later used to solve several other problems, such as failure detection, data aggregation, distributed topology construction, resource allocation – just to name a few. Gossip protocols tend to be used in contexts where both the scale and the dynamism of the underlying communication network make the adoption of traditional communication protocols highly unpractical. In this article, we first introduce a collection of gossip protocols for information diffusion, and we provide an analytical model to study their performance with respect to speed and quality of the diffusion. We then introduce three examples of gossip-based protocols that solve the most diverse problems, namely membership management, aggregation and overlay topology construction.

**Keywords** Gossip protocols, epidemic protocols, information diffusion, distributed systems.

## 1 Introduction

A *gossip protocol* is a distributed communication paradigm inspired by both the spreading of epidemics and the gossip phenomenon that can be observed in social networks. Initially born to efficiently disseminate information, as its human counterpart, it has been later used to solve several other problems, such as failure detection, data aggregation, distributed topology construction and maintenance, resource allocation – just to name a few. Gossip protocols tend to be used in contexts where both the scale and the dynamism of the underlying communication network make the adoption of traditional communication protocols infeasible.

The term *epidemic protocol* is used as a synonym for gossip protocol, because the way gossip spreads information can be modeled as the spread of a virus in a biological community.

### 1.1 Historical overview

The first work to adopt the gossip and epidemic terminology was the seminal paper of Alan Demers et al. [1]; in this paper, the problem of information dissemination in a distributed collection of database servers (part of the Xerox internal network) has been solved using two main approaches, called *anti-entropy* and *rumor mongering*. In the former, nodes participating in the protocol periodically exchange information with random partners and try to reconcile potentially inconsistent copies of the replicated database; in the latter, databases updates are quickly disseminated pushing them towards randomly selected nodes, in the hope that they have not received the updates yet. Combined together, these approaches provide *eventual consistency*, a weak form of consistency that guarantees that in the absence of new updates, all the copies of the database will eventually become equal.

Following the original paper of Demers published at the end of the 80's [1], the 90's have been devoted mostly to translate the original ideas in the field of distributed reliable broadcast, with several optimizations regarding the speed of dissemination, the amount of overhead traffic, and reliability [2, 3, 4]. Important aspects that were introduced in the original paper but solved only several years later include the reduction of the amount of networking traffic traveling long distances in the underlying network – a problem called *spatial gossip* [5], and the development of efficient flow-control mechanisms [6].

At the beginning of years 2000s, the applicability of gossip protocols has been enormously widened: from simple information routers, nodes participating in a gossip protocol have been transformed in full-fledged computing units, capable of modifying their own state to reflect the information they received through gossip exchanges and to send modified information around. Using this approach, protocols have been proposed that are capable to maintain up-to-date membership information about the state of large-scale and dynamic P2P network [7], to perform failure detection [8], to implement garbage collection [9], to compute aggregate information [10], to self-organize complex overlay topologies [11], and to allocate resources [12].

The original information dissemination problem is extensively discussed in Section 2; after that, a general scheme for extending the applicability of gossip is presented in Section 3, followed by three examples of its application, in the fields of membership management (Section 4), aggregation (Section 5) and topology construction (Section 6).

## 1.2 Technological adoption

It is now possible to find implementations of gossip protocols in several distributed systems and architectures, sometimes disguised under different terminology.

The first example may be considered the NNTP protocol [13], whose specification predates, by one year, the definition of gossip protocols themselves. NNTP was the protocol adopted in Usenet, in which posts were exchanged among nodes through communication channels established temporarily using phone calls.

Modern peer-to-peer systems are often based on periodic, random exchanges of information among peers. The foremost example is the file-sharing application BitTorrent [14], in which peers mutually exchange chunks of files over a an overlay network that is randomly mutated. Several examples of peer-to-peer video streaming services have adopted gossip-based mechanisms to disseminate live video streams, the most important of which is the Chinese PPTV, based on the PPLive protocol [15].

Modern NoSQL architectures [16], where scalability has been one of the main drivers, have abandoned strong consistency models like linearizability and serializability in favor of the eventual consistency model offered by gossip protocols. Examples of such systems include key-value stores like Riak [17] and wide-column stores like Cassandra [18].

Apart from open-source NoSQL products, several sources (although often covered by corporate secrecy) seem to indicate the gossip paradigm is widely adopted even inside modern data centers maintained by cloud providers. For example, Amazon has been reported to use an anti-entropy gossip protocol to maintain a fully-connected overlay network inside their datacenters [19].

## 1.3 The gossip philosophy

With the extension of the areas and the problems where the gossip paradigm has been applied, the important question of what is exactly a gossip protocol has been raised [20, 21, 22]. Key aspects that have been identified are the repeated probabilistic exchange of information between two members. Probabilistic means that nodes select their communication partners in a non-deterministic way, choosing from a collection of potential candidates taken from the full set of participating nodes. Repetition is also crucial: in principle, gossip is an endless process, although very often the process may be activated when needed and stopped when the system has converged to the desired state. The mutual exchange between two nodes is finally another important aspect: nodes are supposed to provide as well as to consume information to and from their partners.

In general, gossip protocols are the right choice when reliability in spite of high level of failures and scalability are major concerns. Real deployments that include literally millions of nodes exchanging information using the gossip approach are not uncommon; at the same time, systems like that are subject to enormous dynamism in their composition, with large percentage of the entire system joining or leaving at any time.

# 2 Information dissemination

The most natural application of gossip (or epidemics) in distributed systems is the diffusion of information to a collection of nodes. The information to be spread may assume several forms: messages could be broadcast to all nodes, updates should be applied to all the copies of a replicated database, chunks of a live streaming video must be delivered to all connected clients. Depending on the nature of information, several different techniques and optimizations may be applied, but they all boil down to the same basic concept: some nodes are aware of a piece of information and some are not, and the nodes that have it should cooperate in order to diffuse these pieces to those that do not.

We thus describe the problem first from the point of view of a single variable that is replicated among all nodes, providing algorithmic abstractions to update it on all nodes. In such simple settings, the performance of the diffusion problem can be evaluated analytically. Later, we will extend the description by considering multiple, concurrent updates. The following presentation has been influenced by the original paper of Demers et al. [1], by the subsequent theoretical papers on epidemic spreading by Pittel et al. [23] and Karp et al. [24], and finally by the work of Jelasity [25].

## 2.1 Problem definition

We consider a system composed of fixed collection $P$ of nodes, which is known to all nodes. We will later relax this assumption. Nodes may communicate with each other by exchanging messages. Nodes may fail and messages can be lost, although in the analytical model we will not consider this possibility.

Each node maintains a single variable $value$ whose content is replicated among all nodes. Nodes receive updates to this variable, either by exchanging messages between each other, or by means of an external mechanism like a primitive invocation or a client request. Updates completely overwrite the content of the variable. We will later extend this model by considering more complex data structures.

In order to deal with the possibility of multiple updates, timestamps are associated to them. Field $value.time$ returns the timestamp of the latest update known to the node, taken from a totally ordered set of timestamps $\mathcal{T}$. The implementation of $\mathcal{T}$ depends on the nature of the replicated variable and on how it is updated; for example, if only one entity may update the variable, timestamps can be as simple as a sequence number; if concurrent updates are performed, either logical or synchronized clocks may be required. We assume that whenever $value$ is updated, a new timestamp is generated and associated to $value.time$.

The goal is to spread the updates to all the nodes; more precisely, if no new updates are injected after some time $t$, eventually all correct nodes will have the same copy of the variable. This requirement is called *eventual consistency* and is a weak consistency policy.

We adopt a terminology inspired from epidemiology. With respect to the most recent update, a node can be in one of three states:

- *Susceptible* (S): the node does not know about the update;
- *Infected* (I): the node knows the update and is actively spreading it;
- *Removed* (R): the node knows the update, but it does not participate in the spreading any more.

It is easy to see the parallelism with the spreading of a biological virus: a susceptible patient has not contracted the virus (yet); an infected one is carrying the virus and can infect other patients; finally, a patient is removed when is not contagious any more. The goal is to obtain a pandemic.

The three populations (or *compartments*) described above forms the basis of the so-called *compartmental models* of epidemiology, i.e. mathematical frameworks aimed at understanding the complex dynamics of these systems. Two models will be discussed here, namely the SI model (that only includes susceptible and infected compartments) and the SIR model (that includes all of them). Epidemiologists have studied several other models including additional states, but these go beyond the applicability in computer science.

## 2.2 The SI model, or simple epidemics

The first gossip protocol considered by Demers et al. is the SI model, also called *simple epidemics* or *anti-entropy*. Anti-entropy is an unfortunate name that only describes that the execution of the protocol is aimed at reducing the total disorder, or entropy, among the nodes.

Three styles of anti-entropy protocols have been proposed, shown in Algorithm 1:

- in the *push* style, nodes periodically send (push) the current content of variable $value$ to a node selected randomly from $P$;
- in the *pull* style, nodes periodically ask (pull) new updates of the replicated variable from other, randomly selected nodes;
- in the *push-pull* style, push and pull are combined together.

The execution is divided in a set of consecutive *rounds* of length $\Delta$ time units, represented by the **on timeout** code blocks at the beginning of each version, ended by a **set timeout** operation at the end. We use the term round also to indicate the collective execution of a timeout block by all the nodes in a time interval $\Delta$ time units long.

Apart from the timeout block, actions are executed whenever a message is received. Actions listed in each code block are executed atomically.

In the push style, the content of the variable is sent to a randomly selected node through a PUSH message, irrespective of whether this node has already received the update or not. When the update is received, its timestamp is compared with the local one; if the update is newer, it is copied in $value$.

In the pull style, the latest timestamp known to $p$ is sent to a randomly selected peer through a PULL message. When a PULL message is received, the node compares it with the local timestamp, and if the local timestamp is newer, a REPLY message is sent with the content of $value$. When the REPLY message is received, the timestamp of the received value is compared with the local one again, because other updates may be received in the period between the PULL and the REPLY.

In the push-pull style, the content of $value$ is sent by $p$ to a randomly selected peer $q$ through a PUSHPULL message. When such message is received, the timestamps are compared: if the received timestamp is newer, the update is copied on the local variable; if the received timestamp is older, a REPLY message is sent to the sender of PUSHPULL; otherwise, if they are equal, the content is ignored.

For the purpose of the analysis, we will assume that (i) nodes do not fail; (ii) communication is reliable; (iii) consecutive rounds are synchronized, i.e. they start at the same time on all nodes; (iv) all the messages sent during the execution of a round (PUSH, PULL, PUSHPULL and their potential REPLY) arrive before the next round starts, i.e. rounds are not intermixed. The theoretical derivations obtained under these assumptions give a fair indicator of the quantitative behavior of the gossip protocol in their absence.

**Algorithm 1:** Anti-entropy protocol executed by process $p$

```
% Push version
on timeout
    $q \leftarrow \mathsf{random}(P)$
    send $\langle \text{PUSH}, value \rangle$ to $q$
    set timeout $\Delta$

on receive $\langle \text{PUSH}, v \rangle$
    if $value.time < v.time$ then
        $value \leftarrow v$
```

```
% Pull version
on timeout
    $q \leftarrow \mathsf{random}(P)$
    send $\langle \text{PULL}, p, value.time \rangle$ to $q$
    set timeout $\Delta$

on receive $\langle \text{PULL}, q, t \rangle$
    if $value.time > t$ then
        send $\langle \text{REPLY}, value \rangle$ to $q$

on receive $\langle \text{REPLY}, v \rangle$
    if $value.time < v.time$ then
        $value \leftarrow v$
```

```
% Push-pull version
on timeout
    $q \leftarrow \mathsf{random}(P)$
    send $\langle \text{PUSHPULL}, p, value \rangle$ to $q$
    set timeout $\Delta$

on receive $\langle \text{PUSHPULL}, q, v \rangle$
    if $value.time < v.time$ then
        $value \leftarrow v$
    else if $value.time > v.time$ then
        send $\langle \text{REPLY}, value \rangle$ to $q$

on receive $\langle \text{REPLY}, v \rangle$
    if $value.time < v.time$ then
        $value \leftarrow v$
```

**Push style** We will start by analyzing the push model with regards to the spreading of a single (the latest) update. Let $n = |P|$ the total number of nodes and let the consecutive rounds be numbered starting from 1. Let $s_t$ and $i_t$ be the proportion of nodes belonging to the susceptible and the infected compartments after the execution of $t$ rounds, respectively. Clearly, $s_t = 1 - i_t$. At the beginning, before executing any rounds, $i_0 = 1/n$ and $s_0 = 1 - 1/n$, as only one node has obtained the update (through external means). We can compute the expectation of $s_{t+1}$ as a function of $s_t$, provided that the randomly selected node is chosen uniformly at each node, independently from other nodes and independently of past decisions. In such conditions, we have:

$$E(s_{t+1}) = s_t \left(1 - \frac{1}{n}\right)^{n(1-s_t)} \approx s_t e^{-(1-s_t)} \tag{1}$$

where the approximation holds for large $n$. In other words, a node is susceptible at time $t+1$ if it was susceptible at time $t$ (the term $s_t$) and if it is not selected among all possible nodes (the probability $1 - 1/n$) by any of the infected nodes (which are $n(1 - s_t)$).

Under this model, eventually all the nodes will become infected, as the expected proportion of susceptible nodes tends to zero. Pittel et al. [23] estimated that the expected number rounds $T(n)$ for a system composed of $n$ nodes to become completely infected is:

$$T(n) = \log_2 n + \ln n + O(1) = O(\log n) \tag{2}$$

as $n$ tends to infinity. While the proof is complex, the formula can be intuitively explained as follows. The execution of the protocol can be divided in three phases. During the initial one, most of the nodes are susceptible, so infected nodes will only be able to contact susceptible ones and $i_t$ will double after each round, explaining the $\log_2 n$ term. In the final phase, few susceptible nodes are left, and $E(s_{t+1})$ will be approximated by $s_t e^{-1}$, leading to the term $\ln n$. The "middle" phase between the initial and the final phases can be shown to last only a constant number of rounds.

**Pull style** Starting again from $i_0 = 1/n$ and $s_0 = 1 - 1/n$, the expected number of susceptible nodes at round $t + 1$ can be evaluated as follows:

$$E(s_{t+1}) = s_t \cdot s_t = s_t^2 \tag{3}$$

In other words, a node is susceptible at time $t + 1$ if it was susceptible at time $t$ and it contacts another susceptible node during its pull request, leading to the second term $s_t$. At the beginning, the first infected node may have to wait some rounds before infecting anybody else. But eventually – in $O(\log n)$ rounds w.h.p. – half of the nodes will be infected. After $s_t$ becomes smaller than $1/2$, the expected number of rounds to complete the process is in the order of $O(\log \log n)$ [24].

Figure 1 shows the probability of remaining in the susceptible state after a given number of rounds, shown on the x-axis; in other words, the probability of not being reached by the update. It is immediate to see the superiority of pull over push.

**Push-pull style** Clearly, a combination of the two approaches would retain the benefit of both, at the cost of an increased communication overhead. This style is called push-pull. Although faster in practice, the expected number of rounds needed to reach all nodes is still $O(\log n)$ [24].

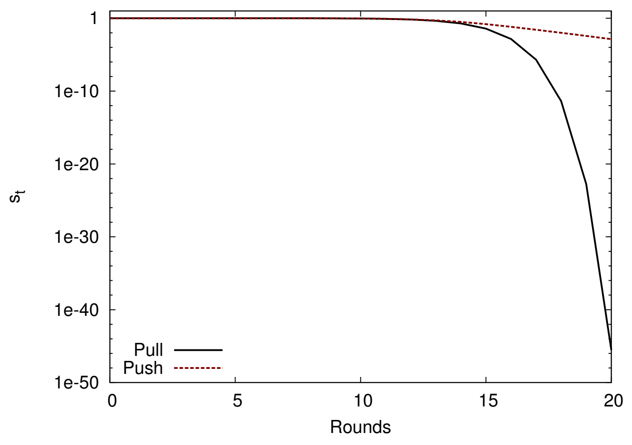


Figure 1: Probability for a node to remain in the susceptible state after $t$ rounds, for both push and pull protocols.

## 2.3 The SIR model, or complex epidemics

The SI model is designed to run forever, and thus is appropriate in systems where a continuous flux of updates is generated. In many scenarios, however, updates are rare; to deal with these cases, the SIR model, also called *complex epidemics* or *rumor mongering*, has been introduced. This model includes a third state, called *removed*, that describes a node that has received an update but it is not actively distributing it anymore.

The SIR model is based on the push style, although pull could be used as well. As soon as a new update is available at a single infected node, the update is pushed towards other random nodes. Susceptible nodes receiving the update become infected, and start pushing the update as well. Eventually, the protocol terminates, when all susceptible and infected nodes have switched to the removed state.

Reaching an agreement on termination among an extremely large number of nodes through a consensus protocol would be prohibitively costly; on the other hand, deciding to terminate based on fixed thresholds relative to the age of the messages, irrespective of the size of the network, is difficult. Instead, termination is decided locally by each node, based on the history of local exchanges it has performed so far.

In the original paper [1], the decision to transition to the removed state is influenced by the following factors:

- *When*: the evaluation on whether to transition to the removed state can be performed at each round (*blind*), or only after having received a message that confirms that the exchange partner was already aware of the update (*feedback*).
- *How*: the transition can be performed after a fixed number $k$ of rounds in which it has been evaluated (*counter*), or with a given probability $1/k$ at each round (*coin*).

The combination of these factors leads to four different variants: blind/counter, blind/coin, feedback/counter or feedback/coin. Algorithm 2 shows two of these combinations (blind/coin and feedback/counter), from which the other variants may be derived.

The evaluation of the performance of these variants is based on the following figures of merit:

- **Residue**. The number of nodes that are still susceptible at the end of the protocol. This value, denoted as $s^*$, may be different from zero because the protocols may turn all the infected nodes into removed ones before being able to spread the updates to all nodes.
- **Total traffic**. While some nodes will send more updates than others, it is convenient to measure the total traffic $m$ generated by the protocol as the total amount of updates sent by all nodes. To understand the average load on single nodes, sometimes is useful to measure $m/n$, i.e. the average number of updates sent by nodes.
- **Delay**. The average delay $t_{avg}$ is the difference between the time of initial injection of the update and its arrival at a given node, averaged over all nodes. The maximum delay $t_{max}$ is the difference between the time of initial injection and the time when the last node has received the update. In both cases, time can be measured in number of rounds.

It is possible to evaluate $s^*$ analytically using differential equations, a standard technique in epidemiology [26]; we show here the analysis for the case feedback/coin. Let $i$, $s$ and $r$ be the fraction of nodes infected, susceptible and removed respectively, so that $i + s + r = 1$. The analysis is based on the following differential equations:

$$\frac{ds}{dt} = -si \tag{4}$$

$$\frac{di}{dt} = +si - \frac{1}{k}(1-s)i \tag{5}$$

**Algorithm 2:** Rumor-mongering protocol executed by process $p$

```
% Blind/coin variant

on update($v$)
    $state \leftarrow$ INFECTED
    $value \leftarrow v$
    set timeout $\Delta$

on timeout
    if $state =$ INFECTED then
        $q \leftarrow$ random($P$)
        send ⟨PUSH, $value$⟩ to $q$
        if tossCoin($1/k$) then
            $state \leftarrow$ REMOVED
        set timeout $\Delta$

on receive ⟨PUSH, $v$⟩
    if $state =$ SUSCEPTIBLE then
        $value \leftarrow v$
        $state =$ INFECTED
```

```
% Feedback/counter variant

on update($v$)
    $value \leftarrow v$
    $state \leftarrow$ INFECTED
    $counter \leftarrow k$
    set timeout $\Delta$

on timeout
    if $state =$ INFECTED then
        $q \leftarrow$ random($P$)
        send ⟨PUSH, $p$, $value$⟩ to $q$
        set timeout $\Delta$

on receive ⟨PUSH, $q$, $v$⟩
    send ⟨REPLY, $state$⟩ to $q$
    if $state =$ SUSCEPTIBLE then
        $value \leftarrow v$
        $state =$ INFECTED
        $counter \leftarrow k$

on receive ⟨REPLY, $s$⟩
    if $s \neq$ SUSCEPTIBLE then
        $counter \leftarrow counter - 1$
        if $counter = 0$ then
            $state \leftarrow$ REMOVED
```

The first one suggests that susceptible nodes will be infected at a rate that depends on $si$, i.e. on the fraction of susceptible nodes to which infected nodes are actively pushing. The second one has an additional term for loss due to infected nodes that are becoming removed with probability $1/k$ after having contacted non-susceptible nodes.

The system of equations is solved by taking their ratio to eliminate $t$. Let us solve for $i$ as a function of $s$:

$$\frac{ds}{di} = -\frac{k+1}{k} + \frac{1}{ks}, \tag{6}$$

which yields

$$i(s) = -\frac{k+1}{k}s + \frac{1}{k}\ln s + c, \tag{7}$$

where $c$ is the constant of integration, that can be determined using the initial condition: $i(1 - 1/n) = 1/n$. For large $n$, $1/n$ goes to zero, giving

$$c = \frac{k+1}{k} \tag{8}$$

and a solution

$$i(s) = \frac{k+1}{k}(1-s) + \frac{1}{k}\ln s. \tag{9}$$

We are now interested in the value $s^*$ in which $i(s^*) = 0$, i.e. the time in which there are no infected nodes and thus the protocol has terminated. The residue $s^*$ can be computed as follows:

$$s^* = e^{-(k+1)(1-s^*)} \tag{10}$$

While the equation is implicit in $s^*$, it shows that the residue decreases exponentially with $k$, meaning that increasing $k$ is an effective way to make sure that everybody get the update. For example, already with $k = 1$, the residue is as low as $20\%$, while with $k = 5$ only $0.24\%$ of the nodes will miss the update.

When discussing total traffic, a surprising observations applies to all four variants. Since each infected node selects its partner independently at random from the entire set $V$, each push message sent has the same probability $1/n$ to hit that particular node, irrespective of how nodes switch to the removed state. This means that the probability of a given node to stay susceptible after $m$ update message are sent can be computed by:

$$s(m) = \left(1 - \frac{1}{n}\right)^m \tag{11}$$

which can be approximated as $s(m) \approx e^{-m/n}$ in the limit of large $n$. Substituting the desired value of $s^*$, we can easily compute the total number of messages that need to be sent: $m \approx -n \ln s^*$. For example, if we set the residue $s^*$ to $1/n$, i.e. we allow only for a single node node to miss the update, then we need $m \approx n \ln n$ messages; in the blind/counter variant, this means that $k$ has to be set equal to $\ln n$.

Based on this observation, the only parameter that distinguishes the four variants is delay; among them, counter and feedback provides the shortest delay, with counter playing a more significant role than feedback on both $t_{avg}$ and $t_{max}$, while blind-coin provides the worst delay (see Table 1).

| Blind/coin | | | | |
|---|---|---|---|---|
| Coin | Residue | Avg. Traffic | Delay | |
| $k$ | $s$ | $m/n$ | $t_{avg}$ | $t_{max}$ |
| 1 | 0.960 | 0.04 | 19.0 | 38.0 |
| 2 | 0.205 | 1.59 | 17.0 | 33.0 |
| 3 | 0.060 | 2.82 | 15.0 | 32.0 |
| 4 | 0.021 | 3.91 | 14.1 | 32.0 |
| 5 | 0.008 | 4.95 | 13.8 | 32.0 |

| Feedback/counter | | | | |
|---|---|---|---|---|
| Counter | Residue | Avg. Traffic | Delay | |
| $k$ | $s$ | $m/n$ | $t_{avg}$ | $t_{max}$ |
| 1 | 0.176 | 1.74 | 11.0 | 16.8 |
| 2 | 0.037 | 3.30 | 12.1 | 16.9 |
| 3 | 0.011 | 4.53 | 12.5 | 17.4 |
| 4 | 0.004 | 5.64 | 12.7 | 17.5 |
| 5 | 0.001 | 6.68 | 12.8 | 17.7 |

Table 1: Performance of the blind/coin and feedback/counter variants of rumor mongering illustrated in Algorithm 2 [1].

## 2.4 Implementation details for information dissemination

When translating the single-update abstract protocols discussed in the previous sections into practical ones, several implementation details need to be considered. We discuss here some of the most important ones.

**Mixing anti-entropy and rumor mongering** The first aspect to be considered is that the rumor mongering and the anti-entropy variants of epidemic protocols can be profitably run together. Rumor mongering protocols can spread updates rapidly with very low network traffic. Unfortunately, some of the nodes may fail to receive the updates, meaning that the replicated database would end up into an inconsistent state and stay there forever. While the probability of such event can be made arbitrarily low, it will never be zero. To avoid this problem, the anti-entropy protocol can be run on top of rumor mongering, less frequently, to reconciliate the state of database replicas. In this way, we get a rapid spreading of updates with low overhead, and an eventual consistency guarantee, again with low overhead due to the reduced gossip frequency.

**Beyond single values or updates** For the purpose of analysis and protocol description, the original gossip protocols have been described from the point of view of a single variable (anti-entropy) or a single update to be disseminated (rumor mongering) [1]. Clearly, this is not the case in real systems.

In the case of anti-entropy, the problem is strongly dependent on the kind of database that is replicated. Several modern systems based on gossip are key-value stores [18, 19], and this was also the case for the Xerox database [1]. Clearly, the option of exchanging the entire content of the database is not a viable option, also because most of the replicas are already identical. The first solution that comes to mind is to transform the push-, pull-, and push-pull exchanges into multi-stage reconciliation protocols, where checksums are exchanged first and only if they differ, the entire database is exchanged. This is not a satisfactory idea either, because even if they need reconciliation, they are probably already mostly identical. A more sophisticated approach defines a time window $\tau$ large enough that updates are expected to reach all nodes within time $\tau$. Each node maintains a *recent update lists* containing all the updates whose age is smaller than $\tau$, and update lists are exchanged until checksums show that the databases have been reconciled.

While this approach could be a viable solution, it fails to deliver good performances in systems overloaded by a high rate of updates. Recently, a novel reconciliation protocol called Scuttlebutt has been introduced [6], and it has been applied in Cassandra [18]. In a nutshell, Scuttlebutt aggressively selects updates that have not been made obsolete by later updates, but without starving updates that are not yet obsolete.

In the case of rumor-mongering, the problem has been reduced to the problem of set reconciliations (with sets containing the IDs of the updates that have been already received), and they have been treated extensively by Byers, Considine and Miztenmacher [27] and Minsky, Trachtenberg and Zippel [28].

**Flow control** In all the models above, it has been implicitly assumed that the communication channels have sufficient bandwidth to transmit all gossip messages, even in the presence of concurrent updates. Clearly, this assumption has to be revisited under heavy load. The problem was originally recognized in the Demers' paper [1] in terms of *connection limits*; in other words, nodes may limit the maximum number of exchanges that are accepted/performed during one round. It has been noted that under extremely low connection limits (e.g., 1-2 connections per round), the pull version gets significantly worse, and, paradoxically, push get significantly better. Several papers have appeared after that, with the goal of optimizing the throughput of probabilistic broadcast protocols based on gossip [29, 2]. More recently, the Scuttlebutt protocol [6] contains specific features for flow control. The goal is to adaptively determine the maximum rate at which a participant can submit updates without creating a backlog of updates. Missing a global oversight, the flow control mechanism has to be completely decentralized. Using a form of decentralized aggregation (see Section 5), nodes may compute the average update rate and adapt their sending behavior to such value.

## 2.5 Dealing with failures

Failures have not been discussed so far. The reason for this is that they have very little effect on the behavior of gossip, apart from slowing down the diffusion process. Message losses may result in void rounds (rounds with no exchanges) for some of the nodes. Crashed nodes leave the system and do not participate in the protocol any more; other nodes keep exchanging messages, occasionally randomly selecting nodes that have crashed. If the percentage of crashed nodes is large, many gossip exchanges are wasted and the protocol could slow down considerably. This has raised the issue of keeping the membership list up-to-date, a problem which is discussed in the next sections.

## 3 Beyond dissemination

Surprisingly, the gossip paradigm can be applied to much more than just information dissemination. When spreading messages, gossip nodes act just as forwarders: they receive messages and they take local decisions about whether to keep spreading them and to which nodes. The key observation here is that nodes may act as full-fledged information processors: they may hold some potentially complex state, they may update the local state based on the information which is received, and they may modify the information which is sent around.

Several problems have been solved in this way. Gossip has been used to maintain up-to-date membership information about the state of large-scale and dynamic P2P network [7], to perform failure detection [8], to implement garbage collection [9], to compute aggregate information [10], to self-organize complex overlay topologies [11], to allocate resources [12] and for distributed machine learning [30]. The large body of research work in some of these areas – namely membership management and aggregation – have been summarized in survey papers [7, 31].

In the rest of the paper, we expose three of these protocols following a generic solution scheme that has emerged over the years, based on the push-pull style. The scheme is shown in Algorithm 3.

**Algorithm 3:** General scheme executed by $p$:

```
on initialization
    $state$is initialized
    set timeout $\Delta$

on timeout
    $q \leftarrow$ selectNeighbor()
    $msg \leftarrow$ prepareRequest($state, q$)
    send $\langle$REQUEST, $msg, p\rangle$ to $q$
    set timeout $\Delta$

on receive $\langle$REQUEST, $req, q\rangle$
    $rep \leftarrow$ prepareReply($state, req, q$)
    send $\langle$REPLY, $rep, p\rangle$ to $q$
    $state \leftarrow$ mergeRequest($state, req, q$)

on receive $\langle$REPLY, $rep, q\rangle$
    $state \leftarrow$ mergeReply($state, rep, q$)
```

The protocol can be described as follows:

- The local state is initialized in a protocol-dependent way, and a timeout is set up to start the periodic execution of push-pull rounds.
- Every $\Delta$ time units, each node sends a digest of its current state in a REQUEST message to a node $q$ selected through the selectNeighbor() primitive. The digest is extracted from the current state through function prepareRequest() and may be specifically addressed to the selected node $q$, which is passed as input to the primitive.
- Upon receipt of a REQUEST message from a process $q$, node $p$ generates a reply through function prepareReply() and sends a REPLY message back to $q$. prepareReply() may be affected by the content of the request and its sender, and thus it takes both these parameters as input. The local state is updated through function mergeRequest(), that takes the local state plus the received digest as input.
- Finally, when receiving a REPLY message, the local state is updated through the mergeReply() function, which again takes the received digest as input.

By customizing the primitives selectNeighbor(), prepareRequest(), prepareReply(), mergeRequest() and mergeReply(), different problems can be solved, as demonstrated in the following sections.

## 4 Membership management

Soon after the development of the first gossip protocols [32], it became immediately clear that the problem of maintaining the list of nodes participating in the protocol is a dissemination problem by itself. In the original Xerox scenario, the complete membership list was sent to all nodes whenever an update was needed. The list was relatively small (several hundred of machines) and although not specified in the original paper, it was probably fairly static.

In modern systems, such as peer-to-peer networks and very large datacenters, the diffusion of the membership list presents two important issues: scale and dynamism. The list may comprise several thousand of machines; furthermore, because of failures, it may need to be updated continuously, particularly if the system is subject to churn (nodes joining and leaving at their will). The original solution of keeping all the nodes up-to-date cannot be practiced any more.

## 4.1 Problem statement

An important observation is that nodes do not need the complete membership list to randomly select a partner to exchange information with; in fact, nodes actually need just a sample of such list, selected uniformly at random. This sample is called *partial view*, it could be (actually, it should be) different at each node, and it must be taken from a reasonably up-to-date version of the membership list. The problem of providing nodes with such up-to-date partial views is called *membership management* or *peer sampling*.

Modern gossip protocols are thus typically based upon a peer sampling service, that provides them a $\mathsf{getPeer}()$ primitive that returns a single randomly selected peer. Not surprisingly, it is possible to implement such important component of gossip-based systems through gossip.

## 4.2 Algorithm

The first solution to deal with the problem of dynamic membership through gossip has been proposed by van Renesse et al. [8], who proposed a gossip failure detection service in 1998. The first system to properly define the membership problem, however, was LBCast [3] in 2003, followed by Newscast [33] and Cyclon [34]. A generic framework encompassing most of the previous solutions is given in [7].

We introduce here Newscast [33], modified to fit into the generic scheme proposed in Section 3. The local state maintained at each node is a partial view containing $c$ *node descriptors*, where a descriptor is formed by a pair (node address, timestamp). The address is used to communicate with the node, while the timestamp represents the age of the descriptor itself, and could be implemented with a counter that is increased after each round.

The primitives listed in Algorithm 3 are customized as follows:

- Primitive $\mathsf{selectNeighbor}()$ returns a random node taken from the partial view:

  $$\mathsf{selectNeighbor}() \rightarrow \mathsf{random}(\mathit{state})$$

  where $\mathsf{random}()$ returns a random element from a set.

- Primitives $\mathsf{prepareRequest}()$ and $\mathsf{prepareReply}()$ returns the entire partial view, plus a fresh identifier representing the local node $p$:

  $$\begin{aligned}\mathsf{prepareRequest}(\mathit{state}) &\rightarrow \mathit{state} \cup \{(p, \mathsf{now}())\}\\ \mathsf{prepareReply}(\mathit{state}, \mathit{req}, q) &\rightarrow \mathit{state} \cup \{(p, \mathsf{now}())\}\end{aligned}$$

  where $\mathsf{now}()$ returns a new timestamp. In other words, at most $c + 1$ descriptors are sent in each message.

- Primitives $\mathsf{mergeRequest}()$ and $\mathsf{mergeReply}()$ merge the $c$ descriptors contained in the local view and the $c + 1$ received with the message; from this union, the $c$ freshest descriptors (in timestamp order) are extracted through function $\mathsf{extractFreshest}()$:

  $$\begin{aligned}\mathsf{mergeRequest}(\mathit{state}, \mathit{req}, q) &\rightarrow \mathsf{extractFreshest}(\mathit{state} \cup \mathit{req})\\ \mathsf{mergeReply}(\mathit{state}, \mathit{rep}, q) &\rightarrow \mathsf{extractFreshest}(\mathit{state} \cup \mathit{rep})\end{aligned}$$

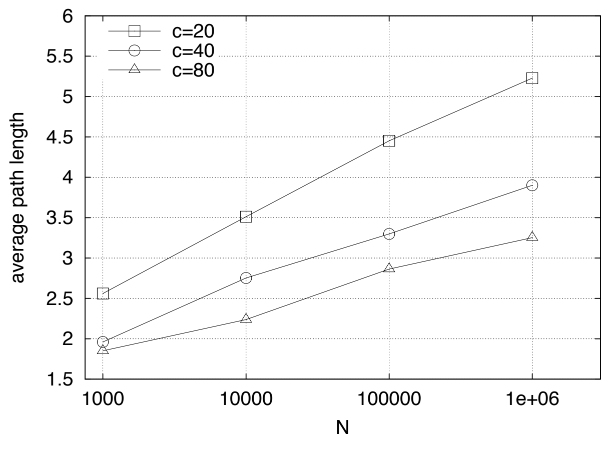
  Ties are solved by randomly selections.

In order to join the system, a node $p$ must discover the address of at least one node $q$ already present in the network. At that point, $p$ may start an exchange with $q$, obtaining its first partial view. The fresh descriptor of $p$ is added to the view of $q$. From then onwards, descriptors are continuously shuffled between nodes through the random selection of partners, and old descriptors are removed from the system thanks to the selection of fresh descriptors. A node leaving the system (or crashing, for that matter) is not required to perform any special action: it stops injecting fresh identifiers of itself in the system, and it will be quickly forgotten by all the other nodes.

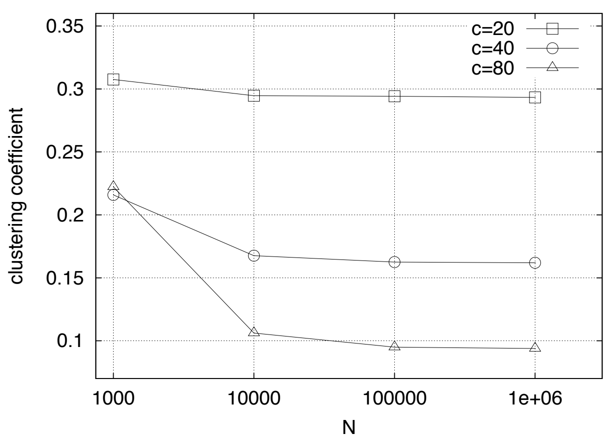
## 4.3 Experimental results

The overlay topology that is obtained by running this protocol should be as random as possible. Figures 2a and 2b show two key figures of merit that characterize the obtained topology: *average path length* and *clustering coefficient*. The average path length is extremely low, grows logarithmically with the network size and it is in line with the expected value in a random network. The clustering coefficient is larger than the expected value in a random network, and this is explained by the fact that after each exchange, two nodes have the same partial view.

From the point of view of the ability to self-clean the network, the expected number of rounds needed to forget a descriptor belonging to a node that has crashed or leaved is logarithmic in the size of the partial view, $c$; this is because after each exchange, the node doubles its knowledge related to descriptors having the same age, and the obsolete descriptor quickly become superseded by more than $c$ other descriptors and thus is expelled from the system.



(a) Average path length.



(b) Clustering coefficient.

Figure 2: Average path length and clustering coefficient as a function of networks size and partial view size $c$ [7].

Related to communication costs, each exchange requires to send and receive $c+1$ descriptors; in large networks, each node actively starts one exchange and may participate in $Poisson(1)$ exchanges per round, in the limit of large networks. Given a round length in the order of seconds and few bytes to store a descriptor, the bandwidth cost associated to this protocol are in the order of few hundred bytes per second.

## 4.4 Dealing with failures

One important property of an overlay network is robustness to random node removal. Figure 3 shows the size of the largest connected component in the graph that is left when a given percentage of nodes is removed, starting from an initial size of $100.000$ nodes. It can be seen that with a partial view size of $80$, practically all the remaining nodes are included in it. Experiments shows that the first small component starts to disconnect from this large component only after removing a large number of nodes; on average, $68\%$, $83\%$ and $94\%$ of the nodes in the case of partial view sizes of $20$, $40$ and $80$, respectively.

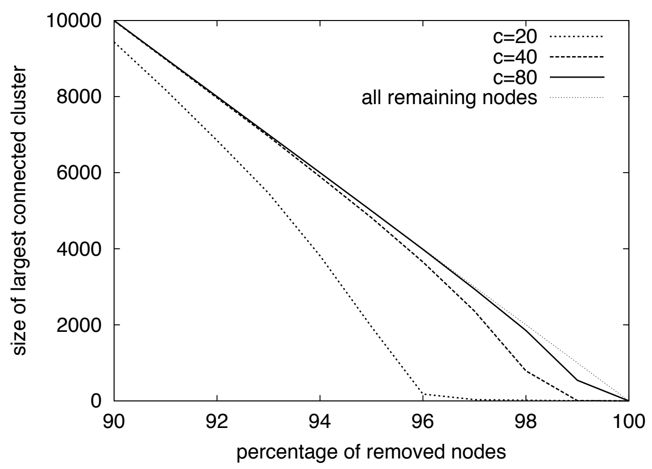


Figure 3: Size of the largest connected component after the random removal of a variable number of nodes. Initial network size is $100.000$ nodes [7].

# 5 Aggregation

Distributed aggregation is a common name for a set of functions that provide a summary of some global system property. In other words, they allow local access to global information in order to simplify the task of controlling, monitoring and optimizing distributed applications. Examples of aggregation applications include the computation of network size, total free storage, maximum load, average uptime, location and intensity of hotspots, etc.

While it is obviously possible (and often desirable) to compute such functions in a centralized way, the gossip paradigm offers an alternative solution which is completely decentralized and does not present any single point of failure or bottleneck, demonstrating once again its enormous potentiality.

The first proposal to adopt gossip to perform aggregation was Astrolabe [35], where aggregation was used to monitor distributed applications. Astrolabe, however, was a complex system mixing hierarchical techniques with epidemic protocols. The first protocols completely based on gossip have been proposed by Kempe et al. [36], based on the push style. Since then, several protocols have been proposed; we propose here a collection of simple protocols based on the push-pull style [10]. More recently, flow-updating protocols have been proposed [37, 38], overcoming the limitations from the fault tolerance point of view.

Section 5.1 illustrates how to perform average aggregation, i.e. how to compute an average over a set of values distributed among a collection of nodes. Section 5.3 shows how the basic average protocol can be extended to compute minimum/maximum, counting, sum, variance, top-$k$ ranking, distribution estimation, etc. Section 5.4 discusses implementation details of aggregation protocols, such as the problem of termination and how failures may affect the global computation.

## 5.1 Problem definition

Consider a collection of nodes. Nodes have access to a peer sampling protocol that provides them with random samples of the entire population of nodes, accessed through a getPeer() primitive that returns a random node among the ones participating in the protocol. Nodes may communicate with each other by exchanging messages. Nodes may fail and messages can be lost, although in the analytical model we will not consider this possibility.

Each nodes stores a single numerical value; the goal is to compute the average of all values and distribute it to all nodes. To reach this goal, each node maintains an estimate of the average, which is updated through repeated gossip exchanges with the other participants until the estimates stored by all nodes have converged enough.

## 5.2 Algorithm

The solution proposed here is based on the general scheme discussed in Section 3, customized as follows. First of all, the local state is represented by a single numerical variable representing the current estimate of the average aggregate. It is initialized with the numerical value known to $p$.

selectNeighbor() returns a random node as obtained by calling the getPeer() method of the underlying peer sampling service. In the original paper [10], the rest of the primitives were customized as follows:

$$\mathsf{prepareRequest}(\mathit{state}, q) \rightarrow \mathit{state} \tag{12}$$
$$\mathsf{prepareReply}(\mathit{state}, \mathit{req}, q) \rightarrow \mathit{state} \tag{13}$$
$$\mathsf{mergeRequest}(\mathit{state}, \mathit{req}, q) \rightarrow (\mathit{state} + \mathit{req})/2 \tag{14}$$
$$\mathsf{mergeReply}(\mathit{state}, \mathit{rep}, q) \rightarrow (\mathit{state} + \mathit{rep})/2 \tag{15}$$

In other words, the primitives prepareRequest() and prepareReply() just return the current estimate, while primitives mergeRequest() and mergeReply() compute a simple local average between the local state and the message received from the exchange partner.

After each exchange, the sum of the two local estimates remains unchanged, since the operations simply redistribute the initial sum equally among the two nodes, under the assumption that exchanges are performed atomically. So, the operation does not change the global average but it decreases the variance over the set of all estimates in the system. It is easy to see that the variance tends to zero, that is, the value at each node will converge to the true global average, as long as the network of nodes is not partitioned into disjoint clusters.

Theoretical results on the speed of convergence have been formally proved [10]. Let $\sigma^2(t)$ be the variance computed over all the estimates after the execution of $t$ gossip rounds by all nodes. It is possible to prove that:

$$E[\sigma^2(t+1)] = \rho \cdot \sigma^2(t) \tag{16}$$

where $\rho = \frac{1}{2\sqrt{e}} \approx 0.3032$. In other words, the expected variance after $t+1$ rounds corresponds to the variance after $t$ rounds, reduced by a constant *convergence factor* $\rho$, meaning that variance is expected to be reduced by a factor of $\rho^t$ after $t$ rounds. Particularly interesting is the fact that this result is independent of the size of the network, although the initial variance and the desired variance may be influenced by the size.

The proof is based on strong synchrony assumptions, but experimental analysis shows that the result holds even in asynchronous distributed systems [10], as illustrated in Figure 4.

Nevertheless, in the original algorithm, concurrent (non-atomic) gossip exchanges may interfere with each other: a node $p$ may initiate an exchange with a partner $q$, and before receiving a reply, may receive a request from another node $r$. If this request is accepted and replied, the sum of the estimates stored by the three nodes after the two exchanges will differ from the sum before. In fact, let $e_p$, $e_q$ and $e_r$ be the estimates at $p$, $q$ and $r$ before the exchanges, respectively; after the exchanges, $q$ will hold $\frac{e_p+e_q}{2}$; $r$ will hold $\frac{e_p+e_r}{2}$; and $p$ will hold $\left(\frac{e_p+e_r}{2} + e_q\right)/2$; the sum of these values is equal to $\frac{5e_p+4e_q+3e_r}{4}$, which is different from $e_p + e_q + e_r$.

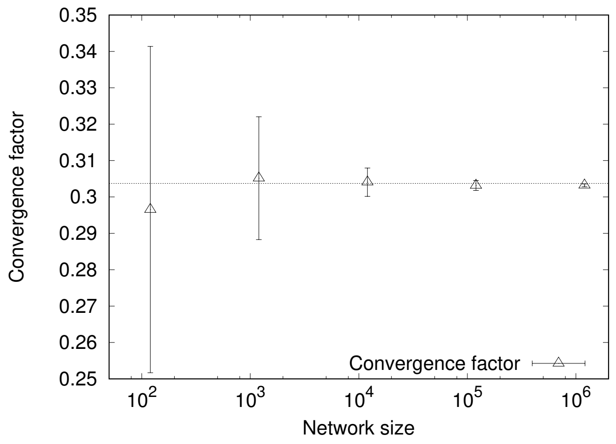


Figure 4: Convergence factor as a function of network size. Values are averages and standard deviations for 50 independent runs. Dotted lines correspond to the theoretically predicted convergence factor $\frac{1}{2\sqrt{e}} \approx 0.3032$ [10].

While refusing the request from $r$ is an acceptable solution to this problem, that has been adopted in the original aggregation paper [10], it has been later shown that it is possible to avoid this problem by adopting the following customizations of the prepare/merge functions [39]:

$$\begin{aligned}
\textsf{prepareRequest}(state, q) &\rightarrow state && (17)\\
\textsf{prepareReply}(state, req, q) &\rightarrow state - (state + req)/2 = d && (18)\\
\textsf{mergeRequest}(state, req, q) &\rightarrow state - d && (19)\\
\textsf{mergeReply}(state, d, q) &\rightarrow state + d && (20)
\end{aligned}$$

In this way, each concurrent exchange is performed by adding and subtracting the same quantity to both the initiator of the exchange and its partner, allowing concurrent exchanges to overlap each other at no risk.

It is possible to show that if primitive getPeer() provides nodes selected sufficiently at random, the number of exchanges for each node during a round can be described by the random variable $1 + Poisson(1)$. Thus, on average, two exchanges per round are executed by each node (one initiated by the node and the other one coming from another node), with a very low variance.

## 5.3 Other aggregation functions

It is easy to extend the above approach to other aggregate functions:

- Computing the minimum and the maximum of all values is straightforward: methods prepareRequest() and prepareReply() just return the local state, while mergeRequest() and mergeReply() compute the minimum or maximum between the two values.
- It is possible to count the number of nodes in the system by using the following trick: let assume that one nodes starts with value 1, while all other nodes start with value 0. Computing the average gives $1/n$, from which is easy to deduce $n$. Obviously, selecting that one node is a problem per se, but it can be easily avoided by allowing multiple "labeled" aggregation protocols to be executed concurrently, with each node starting a new protocol with a probability that depends on previous size estimates.
- Once both the average of the values $\mu$ and the size of the network $n$ are known, it is easy to compute the total sum $\mu \cdot n$.
- Variance can be computed as mean of square minus square of mean: $\overline{a^2} - \overline{a}^2$.

The list of application to the aggregation problem does not end here. For example, Haridasan and van Renesse [40] proposed several gossip-based protocols based on distributed synopsis to estimate the distribution of a set of values; Sacha and Montresor [39] used the aggregation average protocol discussed above to identify the top-$k$ most frequent items in a distributed multiset of values.

## 5.4 Implementation details for aggregation

Building on the simple ideas presented in the previous sections, we now complete the details so as to obtain a full-fledged solution for gossip-based aggregation in practical settings.

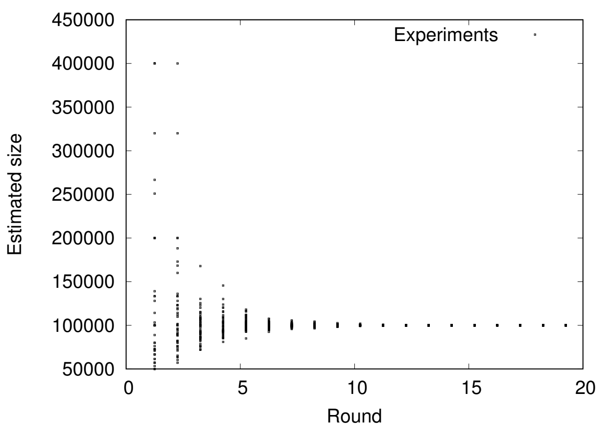
**Adaptivity** The generic protocol described so far is not adaptive, as it does not take into account the dynamism of the network or the variability in values that are being aggregated. To provide up-to-date estimates, the protocol must be periodically *restarted*: at each node, the protocol is terminated and the current estimate is returned as the aggregation output; then, the current local values are used to re-initialize the estimates and aggregation starts again with these fresh initial values.

To implement termination, each node executes the protocol for a predefined number of rounds $\gamma$, depending on the required accuracy of the output and the convergence factor that can be achieved in the particular overlay topology adopted. An execution of the gossip protocol is called *epoch*.

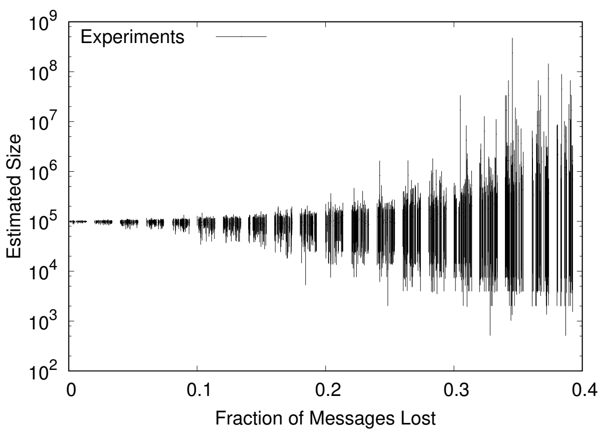
**Synchronization** The protocol described so far is based on the assumption that rounds and epochs proceed in lock step at all nodes. In a large-scale distributed system, this assumption cannot be satisfied due to the unpredictability of message delays and the different drift rates of local clocks. To provide a simple form of barrier synchronization among all nodes, nodes terminate their current instance of aggregation protocol and start a new one whenever they discover a new epoch identifier. Experimental evaluation has shown that this simple approach is sufficient to obtain the desired results.

## 5.5 Dealing with failures

Failures of nodes and communication channels may have adverse effect on the computation of the true average (or other aggregate functions). For example, the crash of a node may remove significant quantities from the values to be aggregated. This is particularly relevant at the beginning of a protocol execution, when values may be widely different from each other; it becomes less of a problem after a few gossip iterations, as differences are quickly smoothed out. See Figure 5a for an example of such effect on a count estimation, which is particularly affected by such problem.



(a) Network size estimation, where $50\%$ of the nodes crash suddenly. The x-axis represents the round since the beginning of the protocol at which the "sudden death" occurs.



(b) Network size estimation as a function of lost messages. Bar lengths illustrate the distance between the minimal and maximal estimated size over the set of nodes within a single experiment.

Figure 5: Network size estimation in a network of size $10^5$ nodes. [10]

In the case of communication failures, different issues may occur if either a link is broken or single messages may be lost. Missing links have only the effect of slowing down the computation; the REQUEST messages are not delivered, nodes miss their chance of starting a gossip exchange, but no other adverse effect occur. Losing messages, on the other hand, may have sever effects on the estimate, as quantities stored in REPLY messages may be lost forever. The effect becomes significant, however, only with severe loss rates, as illustrated in Figure 5b.

To solve this problem, the same trick used to evaluate the size of the network may be applied. Multiple concurrent instances of the aggregation protocol may be run, and outliers are discarded. As it turns out, the resulting implementation is extremely accurate, with errors smaller than $5\%$ under realistic conditions. In harsher failure conditions, approaches based on flow updating [37, 38] may provide much better quality.

# 6 Overlay topology construction

Apart from building random topologies through peer sampling protocols, gossip protocols have been employed to build other kinds of overlay topologies.

Overlay networks have emerged as the single-most important abstraction when implementing a wide range of functions in large, fully decentralized systems. The overlay network needs to be designed appropriately to support the application at hand efficiently. For example, application-level multicast might need carefully controlled random networks or trees, depending on the multicast approach [35, 3]; search protocols may require superpeer networks [41], networks that are organized based on proximity and/or capacity of the nodes [42], or distributed hash tables [43].

Most of the proposed approaches typically assume that a given network already exists and only a relatively small proportion of nodes join and leave concurrently [11]. Gossip-based topology construction protocols enable the construction of complex overlay topologies from scratch, or the concurrent addition of large number of nodes or even the merger of two existing networks [44].

## 6.1 Problem definition

Consider a set of nodes that may communicate with each other by exchanging messages through a routed network. Each node has a *profile* containing additional information about the node that is relevant for the definition of an overlay network. Node ID, geographical location, available resources, etc. are all examples of profile information. The address and the profile together form the *node descriptor*.

As in peer sampling, each node maintains a partial view that contains a set of node descriptors. Views can be interpreted as sets of edges between nodes, naturally defining a directed graph over the nodes that determines the topology of an *overlay network*.

Initially, partial views are initialized by nodes taken from a peer sampling service. The goal is to build a desirable overlay network by filling the views of all nodes with descriptors of the appropriate neighbors. For example, if nodes have to be organized in a ring topology in increasing order of node ID, the view must contain the successor and the predecessor in such order.

A way to describe the desired overlay is through the *ranking method*, that sorts a set of nodes (potential neighbors) according to the "taste" of a given base node. More formally, the input of the problem is a set of $n$ nodes, the *target view size* $k$ (bounded by $n$) and a *ranking method* rank(). The ranking method takes as parameters the base node $p$ and a set of nodes and returns an ordered list of them. All nodes in the network apply the same ranking method, known a priori. The first $k$ nodes in this order are then selected to be the neighbors of the base node.

The *target graph* to be built is defined by the ranking method. One way of actually defining useful ranking methods is through a distance function that defines a metric space over the set of nodes. The ranking method can simply return an ordering of the given set according to non-decreasing distance from the base node. For example, to build the ring described before, let $k = 2$ and let the profile of a node be a real number in the interval $[0, 1[$. The ranking method can be defined based on the one-dimensional distance function between nodes $a$ and $b$ as $d(a, b) = \min(M - |a - b|, |a - b|)$, to obtain a circular structure. Multiple ranking policies can be applied concurrently, for example to separately identify the closest predecessor and the closest successor in the ring.

It is interesting to note the strong link to the construction of $k$-nearest neighbor (KNN) graphs, an important data-mining problem where a set of objects in a metric space need to be associated with the $k$ nearest objects based on some distance measure. Recently, algorithms have been proposed to compute approximate KNN graphs on stand-alone multicore machines, in a way similar to what is proposed here [45].

## 6.2 Algorithm

A simple but inefficient approach to build the target graph could be the following: each node disseminates its descriptors to every other node, using the dissemination protocols described in Section 2, and collects all the descriptors that it receives. At this point, each node sorts this set of descriptors according to the ranking method and picks the first $k$ elements to be its neighbors.

The cost of this approach is $O(n)$ both in terms of space and communication overhead, clearly not acceptable. Once again, the concept of partial view can solve the problem: nodes collect all the descriptors that they receive, but instead of sending around all of them, or a random selection, they send a subset of them, selected based on the ranking preferences of the receiving node. In this way, nodes tend to accumulate descriptors that are closer to their target, and by starting new exchanges with them, they tend to communicate with nodes that are closer and closer. For this reason, the protocol converge to the correct set of neighbors.

The algorithm follows the scheme of Algorithm 3, executed by process $p$, customized as follows:

- The variable $state$ is a partial view containing the descriptors that node $p$ has collected so far; initially, this set is initialized with a random collection of descriptors, potentially taken from a peer sampling protocol.
- Method selectNeighbor() returns a node selected randomly from the first $\psi$ nodes returned by the ranking function rank() executed on the base node $p$ over the local view contained in $state$.

$$\text{selectNeighbor}() \rightarrow \text{random}(\text{extract}(\text{rank}(state, p), \psi))$$

- Methods prepareRequest() and prepareReply() return the first $m$ nodes returned by the ranking function rank() executed on the base node $p$ over the local view contained in $state$.

$$\text{prepareRequest}(state, q) \rightarrow \text{extract}(\text{rank}(state, q), m))$$
$$\text{prepareReply}(state, req, q) \rightarrow \text{extract}(\text{rank}(state, q), m))$$

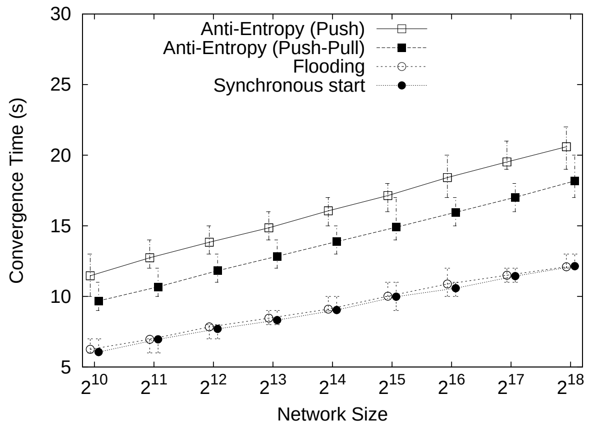
- Methods mergeRequest() and mergeReply() just returns the set union of the local variable and the received message:

$$\text{mergeRequest}(state, req) \to state \cup req$$
$$\text{mergeReply}(state, rep) \to state \cup rep$$

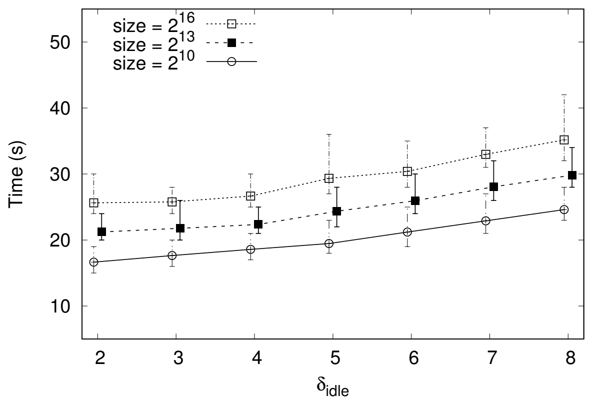
In the previous descriptions, extract$(L, k)$ returns the first $k$ elements of the ordered sequence $L$ (or the entire sequence, if it is smaller than $k$ elements), while random$(S)$ returns a random element from the set $S$. $\psi$ is a parameter describing the size of the pool of nodes from which a random partner should be selected, while $m$ is the maximum number of descriptors to be included in messages. If $\psi = m = n$, the protocol behaves like a dissemination protocol with unbounded message sizes, corresponding to the inefficient protocol described at the beginning of the section. Too small values of $\psi$ make the protocol select the same neighbors over and over again, while too large value of $\psi$ make the protocol select nodes that are potentially not close to the target destination and thus behave like a random dissemination protocol with bounded messages. To solve the former problem, it is possible to apply a small tabu list containing the last few nodes that have been contacted.

## 6.3 Implementation details

Figure 6a shows the convergence time needed by the protocol to converge from a random topology to a ring one, under different starting mechanisms. Convergence time is measured as the number of rounds needed to obtain the target topology. The synchronous start mechanism means that all the nodes start the protocol at the same time; flooding means that a reliable broadcast approach is used [46], while anti-entropy push and anti-entropy push-pull are defined in Section 2. Experimental analysis shows that the convergence time depends logarithmically on the number of nodes in the system [11].



(a) Convergence time as a function of size, using different starting protocols.



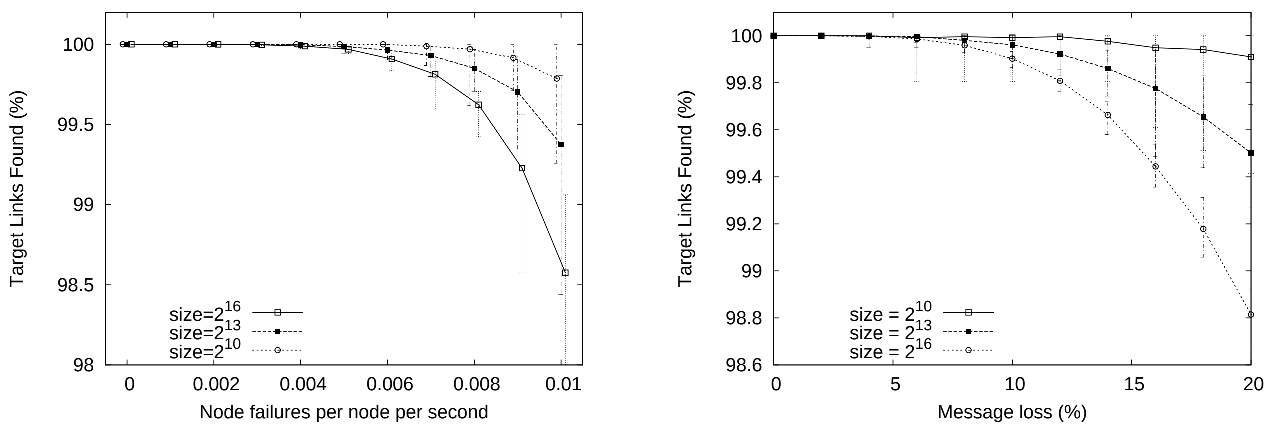
(b) Termination time as a function of $\delta_{idle}$, using anti-entropy push-pull.

Figure 6: Convergence and termination time [11].

To make the system practical, a mechanism to stop the protocol when convergence is reached is needed. A simple but effective, and most importantly, *local* approach is based on measuring how many rounds the local partial view has gone unchanged. If this number goes beyond a threshold $\delta_{idle}$, the node stops participating in the protocol. Eventually, all the nodes will stop, either because they have reached convergence, or without reaching it (a scenario that may occur with very low values of $\delta_{idle}$, like 2 or 3). Figure 6b shows the total termination time using the push-pull starting mechanism with variable values of $\delta_{idle}$.

## 6.4 Dealing with failures

In case of benign failures like node crashes or message losses, no particular modifications are needed. The resulting topology may be an approximation of the target one; but only particularly high levels of churn and messages losses may reduce the quality of the converged overlay, as shown in Figures 7a and 7b, respectively. Failure rate is measured as the ratio of node failing per round; for example, with a round length of 1 second, a failure rate of 0.01 means that the average lifetime of a node is less than 2 minutes. In such extreme scenario, 98.5% of the target edges have been correctly discovered, a number similar to a scenario when $20\%$ of the messages are lost. In more reasonable situations, where around $2\%$ of the messages are lost and the failure rate is around $0.001$, convergence to the target topology is reached.



(a) Target links found by the termination time as a function of failure rate.

(b) Target links found by the termination time as a function of message loss rate.

Figure 7: Resilience to failures [11].

## 7 Conclusions

Gossip protocols are a powerful paradigm to design decentralized distributed protocols that are both robust and efficient. While initially specialized in information dissemination, they have been later used to solve the most diverse problems, including but not limited to protocols capable to maintain up-to-date membership information about the state of large-scale and dynamic P2P network [7], to perform failure detection [8], to implement garbage collection [9], to compute aggregate information [10], to self-organize complex overlay topologies [11], and to allocate resources [12].

After several years of application, the major challenge left is how to apply the epidemic paradigm in environments where Byzantine failures are possible. While preliminary efforts have already started to appear [47, 48, 49], a general framework similar to the one presented in this document is still missing.

The adoption of gossip protocols by the industry appears to be rather limited. Looking just at the academic literature, there seems to be very limited exceptions - such as the reported adoption of gossip protocols in Amazon's internal products like S3 and Dynamo [19]. In reality, there is an increasing number of open-source and commercial products adopting the epidemic approach, such as Apache Cassandra [18], Riak [17] by Basho, Consul [50] and Nomad [51] by HashiCorp, to conclude with ScaleCube [52]. Sometimes, the adoption of gossip protocols is hidden under other names and technologies; for example, the BitTorrent [14] protocol finds its roots in epidemic dissemination, while its PEX extension is a primitive membership protocol [53].

The adoption of gossip protocols is mostly motivated by extreme large-scale and failure uncertainty; in the absence of such conditions, gossip protocols are overkill. Even when applicable, the epidemic approach is threatened by the omnipresence and the high availability of the cloud. Developing fully decentralized services is difficult and error-prone, while centralized approaches have longly proved their reliability. Yet, the growing diffusion of the Internet of Things will open up novel application spaces where the epidemic approach could be profitably applied.

## References

[1] Demers, A. *et al.* (1987) Epidemic Algorithms for Replicated Database Management, in *Proc. of 6th ACM Symposium on Principles of Distributed Computing*, Vancouver, PODC'87. URL `http://www.acm.org/dl`.

[2] Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. (1999) Bimodal multicast. *ACM Trans. Comput. Syst.*, **17** (2), 41–88, doi:10.1145/312203.312207.

[3] Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., and Kermarrec, A.M. (2003) Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, **21** (4), 341–374, doi:10.1145/945506.945507.

[4] Kermarrec, A.M., Massoulié, L., and Ganesh, A.J. (2003) Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst.*, **14** (3), 248–258, doi:10.1109/TPDS.2003.1189583.

[5] Kempe, D., Kleinberg, J., and Demers, A. (2004) Spatial gossip and resource location protocols. *J. ACM*, **51** (6), 943–967.

[6] van Renesse, R., Dumitriu, D., Gough, V., and Thomas, C. (2008) Efficient reconciliation and flow control for anti-entropy protocols, in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, ACM, LADIS '08, doi:10.1145/1529974.1529983.

[7] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., and van Steen, M. (2007) Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, **25** (3), doi:10.1145/1275517.1275520.

[8] van Renesse, R., Minsky, Y., and Hayden, M. (1998) A gossip-style failure detection service, in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer-Verlag, Middleware '98, pp. 55–70. URL `http://dl.acm.org/citation.cfm?id=1659232.1659238`.

[9] Guo, K., Hayden, M., van Renesse, R., Vogels, W., and Birman, K. (1997) GSGC: An efficient gossip-style garbage collection scheme for scalable reliable multicast, *Tech. Rep. TR97-1656*, Cornell CS, Ithaca, NY.

[10] Jelasity, M., Montresor, A., and Babaoglu, O. (2005) Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, **23** (1), 219–252.

[11] Jelasity, M., Montresor, A., and Babaoglu, O. (2009) T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, **53** (13), 2321 – 2339, doi:http://dx.doi.org/10.1016/j.comnet.2009.03.013.

[12] Jelasity, M., Montresor, A., and Babaoglu, O. (2004) A modular paradigm for building self-organizing peer-to-peer applications, in *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, Springer-Verlag, no. 2977 in Lecture Notes in Artificial Intelligence, pp. 265–282.

[13] Kantor, B. and Lapsley, P. (1986), Network News Transfer Protocol, RFC 977 (Proposed Standard). URL `http://www.ietf.org/rfc/rfc977.txt`, obsoleted by RFC 3977.

[14] Cohen, B. (2003) Incentives build robustness in BitTorrent, in *Proceedings of the 6th Workshop on Economics of Peer-to-Peer Systems*, pp. 68–72.

[15] Huang, G. (2007) PPLive: A practical P2P live system with huge amount of users, in *Proceedings of the ACM SIGCOMM Workshop on Peer-to-Peer Streaming and IPTV Workshop*, pp. 22–28.

[16] Han, J., E, H., Le, G., and Du, J. (2011) Survey on NoSQL databases, in *Proceedings of the 6th International Conference on Pervasive Computing and Applications*, ICPCA'11, pp. 363–366, doi:10.1109/ICPCA.2011.6106531.

[17] Basho Technologies, Inc. (2017), Riak, `http://basho.com/products/#riak`.

[18] Lakshman, A. and Malik, P. (2010) Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, **44** (2), 35–40, doi:10.1145/1773912.1773922.

[19] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007) Dynamo: Amazon's highly available key-value store, in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ACM, SOSP '07, pp. 205–220, doi:10.1145/1294261.1294281.

[20] Kermarrec, A. and van Steen, M. (2007) Gossiping in distributed systems. *Operating Systems Review*, **41** (5), 2–7, doi:10.1145/1317379.1317381. URL `http://doi.acm.org/10.1145/1317379.1317381`.

[21] Birman, K. (2007) The promise, and limitations, of gossip protocols. *Operating Systems Review*, **41** (5), 8–13, doi:10.1145/1317379.1317382. URL `http://doi.acm.org/10.1145/1317379.1317382`.

[22] Costa, P., Gramoli, V., Jelasity, M., Jesi, G.P., Merrer, E.L., Montresor, A., and Querzoni, L. (2007) Exploring the interdisciplinary connections of gossip-based systems. *Operating Systems Review*, **41** (5), 51–60, doi:10.1145/1317379.1317388. URL `http://doi.acm.org/10.1145/1317379.1317388`.

[23] Pittel, B. (1987) On spreading a rumor. *SIAM Journal on Applied Mathematics*, **47** (1), 213–223, doi:10.1137/0147013.

[24] Karp, R., Schindelhauer, C., Shenker, S., and Vocking, B. (2000) Randomized rumor spreading, in *Proc. of the 41st Annual Symposium on Foundations of Computer Science*, FOCS'00, pp. 565–574.

[25] Jelasity, M. (2011) *Gossip*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 139–162, doi:10.1007/978-3-642-17348-6_7. URL `http://dx.doi.org/10.1007/978-3-642-17348-6_7`.

[26] Keyfitz, B. and Keyfitz, N. (1997) The mckendrick partial differential equation and its uses in epidemiology and population study. *Mathematical and Computer Modelling*, **26** (6), 1 – 9, doi:http://dx.doi.org/10.1016/S0895-7177(97)00165-9. URL `http://www.sciencedirect.com/science/article/pii/S0895717797001659`.

[27] Byers, J., Considine, J., and Mitzenmacher, M. (2002) Fast approximate reconciliation of set differences, *Tech. Rep.*, BU Computer Science.

[28] Minsky, Y., Trachtenberg, A., and Zippel, R. (2003) Set reconciliation with nearly optimal communication complexity. *Transactions on Information Theory,*, **49** (9), 2213–2218.

[29] Sun, Q. and Sturman, D.C. (2000) A gossip-based reliable multicast for large-scale high-throughput applications, in *Proceedings of the International Conference on Dependable Systems and Networks*, DSN'00, pp. 347–358.

[30] Ormándi, R., Hegedüs, I., and Jelasity, M. (2013) Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, **25** (4), 556–571.

[31] Jesus, P., Baquero, C., and Almeida, P.S. (2015) A survey of distributed data aggregation algorithms. *IEEE Communications Surveys Tutorials*, **17** (1), 381–404.

[32] Golding, R.A. and Taylor, K. (1992) Group membership in the epidemic style, *Tech. Rep.*, University of California at Santa Cruz, Santa Cruz, CA, USA.

[33] Jelasity, M., Guerraoui, R., Kermarrec, A.M., and van Steen, M. (2004) The peer sampling service: Experimental evaluation of unstructured gossip-based implementations, in *Proceedings of Middleware 2004*, *Lecture Notes in Computer Science*, vol. 3231, Springer-Verlag, *Lecture Notes in Computer Science*, vol. 3231, pp. 79–98.

[34] Voulgaris, S., Gavidia, D., and Van Steen, M. (2005) CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, **13** (2), 197–217.

[35] Van Renesse, R., Birman, K.P., and Vogels, W. (2003) Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, **21** (2), 164–206, doi:10.1145/762483.762485.

[36] Kempe, D., Dobra, A., and Gehrke, J. (2003) Gossip-based computation of aggregate information, in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, FOCS '03. URL `http://dl.acm.org/citation.cfm?id=946243.946317`.

[37] Jesus, P., Baquero, C., and Almeida, P.S. (2015) Flow updating: Fault-tolerant aggregation for dynamic networks. *Journal of Parallel and Distributed Computing*, **78**, 53 – 64, doi:http://dx.doi.org/10.1016/j.jpdc.2015.02.003. URL `//www.sciencedirect.com/science/article/pii/S0743731515000416`.

[38] Almeida, P.S., Baquero, C., Farach-Colton, M., Jesus, P., and Mosteiro, M.A. (2016) Fault-tolerant aggregation: Flow-updating meets mass-distribution. *Distributed Computing*, pp. 1–11, doi:10.1007/s00446-016-0288-5. URL `http://dx.doi.org/10.1007/s00446-016-0288-5`.

[39] Sacha, J. and Montresor, A. (2013) Identifying frequent items in distributed data sets. *Computing*, **95** (4), 289–307.

[40] Haridasan, M. and van Renesse, R. (2008) Gossip-based distribution estimation in peer-to-peer networks, in *Proceedings of the 7th International Conference on Peer-to-peer Systems*, USENIX Association, IPTPS'08. URL `http://dl.acm.org/citation.cfm?id=1855641.1855654`.

[41] Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., and Löser, A. (2003) Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks, in *Proceedings of the 12th International Conference on World Wide Web*, ACM, WWW '03, pp. 536–543, doi:10.1145/775152.775229.

[42] Voulgaris, S., Kermarrec, A.M., and Massoulie, L. (2004) Exploiting semantic proximity in peer-to-peer content searching, in *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems*, IEEE, FTDCS '04, pp. 238–243, doi:10.1109/FTDCS.2004.1316622.

[43] Galuba, W. and Girdzijauskas, S. (2009) *Distributed Hash Table*, Springer, pp. 903–904, doi:10.1007/978-0-387-39940-9_1232.

[44] Shafaat, T.M., Ghodsi, A., and Haridi, S. (2007) Handling network partitions and mergers in structured overlay networks, in *Proceedings of the 7th IEEE International Conference on Peer-to-Peer Computing*, IEEE, P2P '08, pp. 132–139, doi:10.1109/P2P.2007.42.

[45] Dong, W., Moses, C., and Li, K. (2011) Efficient k-nearest neighbor graph construction for generic similarity measures, in *Proceedings of the 20th International Conference on World Wide Web*, ACM, pp. 577–586.

[46] Hadzilacos, V. and Toueg, S. (1993) A modular approach to fault-tolerant broadcasts and related problems, in *Distributed Systems (2nd edition)* (ed. S. Mullender), Addison-Wesley. Chapter 5.

[47] Li, H.C., Clement, A., Wong, E.L., Napper, J., Roy, I., Alvisi, L., and Dahlin, M. (2006) Bar gossip, in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, USENIX Association, OSDI '06, pp. 191–204.

[48] Bortnikov, E., Gurevich, M., Keidar, I., Kliot, G., and Shraer, A. (2009) Brahms: Byzantine resilient random membership sampling. *Computer Networks*, **53** (13), 2340 – 2359.

[49] Jesi, G.P., Montresor, A., and van Steen, M. (2010) Secure peer sampling. *Computer Networks*, **54** (13), 2086–2098.

[50] Hashi Corp (2017), Consul, `https://www.consul.io/`.

[51] Hashi Corp (2017), Nomad, `https://www.nomadproject.io/`.

[52] Scale Cube Team (2017), Scale cube, `http://scalecube.io/`.

[53] Vuze (2017), Peer EXchange, `http://wiki.vuze.com/w/Peer_Exchange`.