Chapter 1
Gossip

Márk Jelasity

Anyone can start a rumor, but none can stop one.
(American proverb)

Abstract Gossip plays a very significant role in human society. Information spreads throughout the human grapevine at an amazing speed, often reaching almost everyone in a community, without any central coordinator. Moreover, rumor tends to be extremely stubborn: once spread, it is nearly impossible to erase it. In many distributed computer systems—most notably in *cloud computing* and *peer-to-peer computing*—this speed and robustness, combined with algorithmic simplicity and the lack of central management, are very attractive features. Accordingly, over the past few decades several gossip-based algorithms have been developed to solve various problems. In this chapter, we focus on two main manifestations of gossip: information spreading (also known as multicast) where a piece of news is being spread, and information aggregation (or distributed data mining), where distributed information is being summarized. For both topics we discuss theoretical issues, mostly relying on results from epidemiology, and we also consider design issues and optimizations in distributed applications.

Objectives

- Explain the basic properties of gossip-based information dissemination
- Show how the gossip approach can be used for another domain: information aggregation
- Discuss example systems which are based on gossip or which apply components based on gossip

Márk Jelasity
University of Szeged and Hungarian Academy of Sciences, H-6701 Szeged, PO Box 652. e-mail: jelasity@inf.u-szeged.hu

1.1 Introduction

1.1.1 Gossip

Like it or not, gossip plays a key role in human society. In his controversial book, Dunbar (an anthropologist) goes as far as to claim that the primary reason for the emergence of language was to permit gossip, which had to replace grooming—a common social reinforcement activity in primates—due to the increased group size of early human populations in which grooming was no longer feasible [5].

Whatever the case, it is beyond any doubt that gossip—apart from still being primarily a social activity—is highly effective in spreading information. In particular, information spreads very quickly, and the process is most resistant to attempts to stop it. In fact, sometimes it is so much so that it can cause serious damage; especially to big corporations. Rumors associating certain corporations to Satanism, or claiming that certain restaurant-chains sell burgers containing rat meat or milk shakes containing cow eyeball fluid as thickener, etc., are not uncommon. Accordingly, controlling gossip has long been an important area of research. The book by Kimmel gives many examples and details on human gossip [15].

While gossip is normally considered to be a means for spreading information, in reality information is not just transmitted mechanically but also processed. A person collects information, processes it, and passes the processed information on. In the simplest case, information is filtered at least for its degree of interest. This results in the most interesting pieces of news reaching the entire group, whereas the less interesting ones will stop spreading before getting to everyone. More complicated scenarios are not uncommon either, where information is gradually altered. This increases the complexity of the process and might result in emergent behavior where the community acts as a "collectively intelligent" (or sometimes perhaps not so intelligent) information processing medium.

1.1.2 Epidemics

Gossip is analogous to an epidemic, where a virus plays the role of a piece of information, and infection plays the role of learning about the information. In the past years we even had to learn concepts such as "viral marketing", made possible through Web 2.0 platforms such as video sharing sites, where advertisers consciously exploit the increasingly efficient and extended social networks to spread ads via gossip. The key idea is that shocking or very funny ads are especially de-

signed so as to maximize the chances that viewers inform their friends about it, and so on.

Not surprisingly, epidemic spreading has similar properties to gossip, and is equally (if not more) important to understand and control. Due to this analogy and following common practice we will mix epidemiological and gossip terminology, and apply epidemic spreading theory to gossip systems.

1.1.3 Lessons for Distributed Systems

Gossip and epidemics are of interest for large scale distributed systems for at least two reasons. The first reason is inspiration to design new protocols: gossip has several attractive properties like simplicity, speed, robustness, and a lack of central control and bottlenecks. These properties are very important for information dissemination and collective information processing (aggregation) that are both key components of large scale distributed systems.

The second reason is security research. With the steady growth of the Internet, viruses and worms have become increasingly sophisticated in their spreading strategies. Infected computers typically organize into networks (called botnets) and, being able to cooperate and perform coordinated attacks, they represent a very significant threat to IT infrastructure. One approach to fighting these networks is to try and prevent them from spreading, which requires a good understanding of epidemics over the Internet.

In this chapter we focus on the former aspect of gossip and epidemics: we treat them as inspiration for the design of robust self-organizing systems and services.

1.1.4 Outline

We discuss the gossip communication model in the context of two application domains: information dissemination (Section 1.2) and information aggregation (Section 1.3). For both domains we start by introducing key approaches and algorithms, along with their theoretical properties, where possible. Subsequently we discuss applications of these ideas within each section, where we pay more attention to practical details and design issues. In Section 1.4 we briefly mention application domains that we did not discuss in detail earlier, but that are also promising applications of gossip. In Sections 1.5 and 1.6 we list some key conclusions and suggest several articles that the reader may find helpful and worth perusing.

1.2 Information dissemination

The most natural application of gossip (or epidemics) in computer systems is spreading information. The basic idea of processes periodically communicating with peers and exchanging information is not uncommon in large scale distributed systems, and has been applied from the early days of the Internet. For example, the Usenet newsgroup servers spread posts using a similar method, and the IRC chat protocol applies a similar principle as well among IRC servers. In many routing protocols we can also observe routers communicating with neighboring routers and exchanging traffic information, thereby improving routing tables.

However, the first real application of gossip, that was based on theory and careful analysis, and that boosted scientific research into the family of gossip protocols, was part of a distributed database system of the Xerox Corporation, and was used to make sure each replica of the database on the Xerox internal network was up-to-date [4]. In this section we will employ this application as a motivating example and illustration, and at the same time introduce several variants of gossip-based information dissemination algorithms.

1.2.1 The Problem

Let us assume we have a set of database servers (in the case of Xerox, 300 of them, but this number could be much larger as well). All of these servers accept updates; that is, new records or modifications of existing records. We want to inform all the servers about each update so that all the replicas of the database are identical and up-to-date.

Obviously, we need an algorithm to inform all the servers about a given update. We shall call this task *update spreading*. In addition, we should take into account the fact that whatever algorithm we use for spreading the update, it will not work perfectly, so we need a mechanism for *error correction*.

At Xerox, update spreading was originally solved by sending the update via email to all the servers, and error correction was done by hand. Sending emails is clearly not scalable: the sending node is a bottleneck. Moreover, multiple sources of error are possible: the sender can have an incomplete list of servers in the network, some of the servers can temporarily be unavailable, email queues can overflow, and so on.

Both tasks can be solved in a more scalable and reliable way using an appropriate (separate) gossip algorithm. In the following we first introduce several gossip models and algorithms, and then we explain how the various algorithms can be applied to solve the above mentioned problems.

1.2.2 Algorithms and Theoretical Notions

We assume that we are given a set of nodes that are able to pass messages to each other. In this section we will focus on the spreading of a single update among these nodes. That is, we assume that at a certain point in time, one of the nodes gets a new update from an external source, and from that point we are interested in the dynamics of the spreading of that update.

When discussing algorithms and theoretical models, we will use the terminology of epidemiology. According to this terminology, each node can be in one of three states, namely

- *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 has seen the update, but is not participating in the spreading process (in epidemiology, this corresponds to death or immunity)

These states are relative to one fixed update. If there are several concurrent updates, one node can be infected with one update, while still being susceptible to another update, and so on. For the purpose of theoretical discussion, we will formulate our algorithms assuming there is only one update in the system but assuming that the nodes do *not* know that there is only one expected update. This will allow us to derive the key theoretical properties of update propagation while keeping the algorithms simple.

In realistic applications there are typically many updates being propagated concurrently, and new updates are inserted continuously. In such scenarios additional techniques can be applied to optimize the amortized cost of propagating a single update. In Section 1.2.3 we discuss some of these techniques. In addition, nodes might know the global list or even the insertion time of the updates, as well as the list of updates available at some other nodes. This information can also be applied to reduce propagation cost even further.

The allowed state transitions depend on the model that we study. Next, we shall consider the SI model and the SIR model. In the SI model, nodes are initially in state S and can change to state I. Once in state I, a node can no longer change its state (I is an absorbing state). In the SIR model, we allow nodes in state I to switch to state R, where R is the absorbing state.

1.2.2.1 The SI Model

The algorithm that implements gossip in the SI model is shown in Algorithm 1. It is formulated in an asynchronous message passing style, where each node executes one process (that we call the active thread) and, furthermore, it has message handlers that process incoming messages.

The active thread is executed once in each Δ time units. We will call this waiting period a *gossip cycle* (other terminology is also used such as gossip round or period).

Algorithm 1 SI gossip

```
 1: loop
 2:     wait(Δ)
 3:     p ← random peer
 4:     if push and in state I then
 5:         send update to p
 6:     end if
 7:     if pull then
 8:         send update-request to p
 9:     end if
10: end loop

11: procedure ONUPDATE(m)
12:     store m.update      ▷ means switching to state I
13: end procedure
14:
15: procedure ONUPDATEREQUEST(m)
16:     if in state I then
17:         send update to m.sender
18:     end if
19: end procedure
```

In line 3 we assume that a node can select a random peer node from the set of all nodes. This assumption is not trivial. We will discuss random peer sampling briefly in Section 1.4.

The algorithm makes use of two important Boolean parameters called *push* and *pull*. At least one of them has to be true, otherwise no messages are sent. Depending on these parameters, we can talk about push, pull, and push-pull gossip, each having significantly different dynamics and cost. In push gossip, susceptible nodes are passive and infective nodes actively infect the population. In pull and push-pull gossip each node is active.

Notice that in line 7 we do not test whether the node is infected. The reason is that we assumed that nodes do not know how many updates are expected, and do not exchange information about which updates they have received. Obviously, a node cannot stop pulling for updates unless it knows what updates can be expected; and it cannot avoid getting known updates either unless it advertises which updates it has already. Therefore, in the simplest case, we pull for *any* update, all the time. Optimizations to mitigate this problem are possible, but solving it in a scalable way is not trivial.

For theoretical purposes we will assume that messages are transmitted without delay, and for now we will assume that no failures occur in the system. We will also assume that messages are sent at the same time at each node, that is, messages from different cycles do not mix and cycles are synchronized. None of these assumptions are critical for practical usability, but they are needed for theoretical derivations that nevertheless give a fair indication of the qualitative and also quantitative behavior of gossip protocols.

Let us start with the discussion of the push model. We will consider the propagation speed of the update as a function of the number of nodes N. Let s_0 denote the proportion of susceptible nodes at the time of introducing the update at one node. Clearly, $s_0 = (N-1)/N$. Let s_t denote the proportion of susceptible nodes at the end of the t-th cycle; that is, at time $t\Delta$. We can calculate the expectation of s_{t+1} as a function of s_t, provided that the peer selected in line 3 is chosen independently at each node and independently of past decisions as well. In this case, we have

$$E(s_{t+1}) = s_t \left(1 - \frac{1}{N}\right)^{N(1-s_t)} \approx s_t e^{-(1-s_t)}, \tag{1.1}$$

where $N(1-s_t)$ is the number of nodes that are infected at cycle t, and $(1-1/N)$ is the probability that a fixed infected node will not infect some fixed susceptible node. Clearly, a node is susceptible in cycle $t+1$ if it was susceptible in cycle t and all the infected nodes picked some other node. Actually, as it turns out, this approximative model is rather accurate (the deviation from it is small), as shown by Pittel in [17]: we can take the expected value $E(s_{t+1})$ as a good approximation of s_{t+1}.

It is easy to see that if we wait long enough, then eventually all the nodes will receive the update. In other words, the probability that a particular node never receives the update is zero. But what about the number of cycles that are necessary to let every node know about the update (become infected)? Pittel proves that in probability,

$$S_N = \log_2 N + \log N + O(1) \quad \text{as} \quad N \to \infty, \tag{1.2}$$

where $S_N = min\{t : s_t = 0\}$ is the number of cycles needed to spread the update.

The proof is rather long and technical, but the intuitive explanation is rather simple. In the initial cycles, most nodes are susceptible. In this phase, the number of infected nodes will double in each cycle to a good approximation. However, in the last cycles, where s_t is small, we can see from (1.1) that $E(s_{t+1}) \approx s_t e^{-1}$. This suggests that there is a first phase, lasting for approximately $\log_2 N$ cycles, and there is a last phase lasting for $\log N$ cycles. The "middle" phase, between these two phases, can be shown to be very fast, lasting a constant number of cycles.

Equation (1.2) is often cited as the key reason why gossip is considered efficient: it takes only $O(\log N)$ cycles to inform each node about an update, which suggests very good scalability. For example, with the original approach at Xerox, based on sending emails to every node, the time required is $O(N)$, assuming that the emails are sent sequentially.

However, let us consider the total number of messages that are being sent in the network until every node gets infected. For push gossip it can be shown that it is $O(N \log N)$. Intuitively, the last phase that lasts $O(\log N)$ cycles with s_t being very small already involves sending too many messages by the infected nodes. Most of these messages are in vain, since they target nodes that are already infected. The optimal number of messages is clearly $O(N)$, which is attained by the email approach.

Fortunately, the speed and message complexity of the push approach can be improved significantly using the pull technique. Let us consider s_t in the case of pull gossip. Here, we get the simple formula of

$$E(s_{t+1}) = s_t \cdot s_t = s_t^2, \tag{1.3}$$

which intuitively indicates a quadratic convergence if we assume the variance of s_t is small. When s_t is large, it decreases slowly. In this phase the push approach clearly performs better. However, when s_t is small, the pull approach results in a

significantly faster convergence than push. In fact, the quadratic convergence phase, roughly after $s_t < 0.5$, lasts only for $O(\log \log N)$ cycles, as can be easily verified.

One can, of course, combine push and pull. This can be expected to work faster than either push or pull separately, since in the initial phase push messages will guarantee fast spreading, while in the end phase pull messages will guarantee the infecting of the remaining nodes in a short time. Although faster in practice, the speed of push-pull is still $O(\log N)$, due to the initial exponential phase.

What about message complexity? Since in each cycle each node will send at least one request, and $O(\log N)$ cycles are necessary for the update to reach all the nodes, the message complexity is $O(N \log N)$. However, if we count only the updates, and ignore request messages, we get a different picture. Just counting the updates is not meaningless, because an update message is normally orders of magnitude larger than a request message. It has been shown that in fact the push-pull gossip protocol sends only $O(N \log \log N)$ updates in total [10].

The basic idea behind the proof is again based on dividing the spreading process into phases and calculating the message complexity and duration of each phase. In essence, the initial exponential phase—that we have seen with push as well—requires only $O(N)$ update transmissions, since the number of infected nodes (that send the messages) grows exponentially. But the last phase, the quadratic shrinking phase as seen with pull, lasts only $O(\log \log N)$ cycles. Needless to say, as with the other theoretical results, the mathematical proof is quite involved.

1.2.2.2 The SIR Model

In the previous section we gave some important theoretical results regarding convergence speed and message complexity. However, we ignored one problem that can turn out to be important in practical scenarios: termination.

Push protocols never terminate in the SI model, constantly sending useless updates even after each node has received every update. Pull protocols could stop sending messages *if* the complete list of updates was known in advance: after receiving all the updates, no more requests need to be sent. However, in practice not even pull protocols can terminate in the SI model, because the list of updates is rarely known.

Here we will discuss solutions to the termination problem in the SIR model. These solutions are invariably based on some form of detecting and acting upon the "age" of the update.

We can design our algorithm with two different goals in mind. First, we might wish to ensure that the termination is optimal; that is, we want to inform all the nodes about the update, and we might want to minimize redundant update transmissions at the same time. Second, we might wish to opt for a less intelligent, simple protocol and analyze the size of the proportion of the nodes that will not get the update as a function of certain parameters.

One simple way of achieving the first design goal of optimality is by keeping track of the age of the update explicitly, and stop transmission (i.e., switching to the removed state, hence implementing the SIR model) when a pre-specified age is

Algorithm 2 an SIR gossip variant

```
 1: loop
 2:     wait(Δ)
 3:     p ← random peer
 4:     if push and in state I then
 5:         send update to p
 6:     end if
 7:     if pull then
 8:         send update-request to p
 9:     end if
10: end loop
11:
12: procedure ONFEEDBACK(m)
13:     switch to state R with prob. 1/k
14: end procedure
15: procedure ONUPDATE(m)
16:     if in state I or R then
17:         send feedback to m.sender
18:     else
19:         store m.update              ▷ now in state I
20:     end if
21: end procedure
22:
23: procedure ONUPDATEREQUEST(m)
24:     if in state I then
25:         send update to m.sender
26:     end if
27: end procedure
```

reached. This age threshold must be calculated to be optimal for a given network size N using the theoretical results sketched above. This, of course, assumes that each node knows N. In addition, a practically error- and delay-free transmission is also assumed, or at least a good model of the actual transmission errors is needed.

Apart from this problem, keeping track of the age of the update explicitly represents another, non-trivial practical problem. We assumed in our theoretical discussions that messages have no delay and that cycles are synchronized. When these assumptions are violated, it becomes rather difficult to determine the age of an update with an acceptable precision.

From this point on, we shall discard this approach, and focus on simple asynchronous methods that are much more robust and general, but are not optimal. To achieve the second design goal of simplicity combined with reasonable performance, we can try to guess when to stop based on local information and perhaps information collected from a handful of peers. These algorithms have the advantage of simplicity and locality. Besides, in many applications of the SIR model, strong guarantees on complete dissemination are not necessary, as we will see later on.

Perhaps the simplest possible implementation is when a node moves to the removed state with a fixed probability whenever it encounters a peer that has already received the update. Let this probability be $1/k$, where the natural interpretation of parameter k is the average number of times a node sends the update to a peer that turns out to already have the update before stopping its transmission. Obviously, this implicitly assumes a feedback mechanism because nodes need to check whether the peer they sent the update to already knew the update or not.

As shown in Algorithm 2, this feedback mechanism is the only difference between SIR and SI gossip. The active thread and procedure ONUPDATEREQUEST are identical to Algorithm 1. However, procedure ONUPDATE sends a feedback message when the received update is known already. This message is processed by procedure ONFEEDBACK, eventually switching the node to the removed state. When in the removed state, procedure ONUPDATEREQUEST will no longer deliver the update.

Mathematical models of SIR algorithms are more complicated than those of the SI model. A typical approach is to work with differential equations, as opposed to the discrete stochastic approach we applied previously. Let us illustrate this approach via an analysis of Algorithm 2, assuming a push variant. Following [2, 4], we can write

$$\frac{ds}{dt} = -si \tag{1.4}$$

$$\frac{di}{dt} = si - \frac{1}{k}(1-s)i \tag{1.5}$$

where $s(t)$ and $i(t)$ are the proportions of susceptible and infected nodes, respectively. The nodes in the removed state are given by $r(t) = 1 - s(t) - i(t)$. We can take the ratio, eliminating t:

$$\frac{di}{ds} = -\frac{k+1}{k} + \frac{1}{ks}, \tag{1.6}$$

which yields

$$i(s) = -\frac{k+1}{k}s + \frac{1}{k}\log s + c, \tag{1.7}$$

where c is the constant of integration, which can be determined using the initial condition that $i(1 - 1/N) = 1/N$ (where N is the number of nodes). For a large N, we have $c \approx (k+1)/k$.

Now we are interested in the value s^* where $i(s^*) = 0$: at that time sending the update is terminated, because all nodes are susceptible or removed. In other words, s^* is the proportion of nodes that do not know the update when gossip stops. Ideally, s^* should be zero. Using the results, we can write an implicit equation for s^* as follows:

$$s^* = \exp[-(k+1)(1-s^*)]. \tag{1.8}$$

This tells us that the spreading is very effective. For $k = 1$, 20% of the nodes are predicted to miss the update, but with $k = 5$, 0.24% will miss it, while with $k = 10$ it will be as few as 0.00017%.

Let us now proceed to discussing message complexity. Since full dissemination is not achieved in general, our goal is now to approximate the number of messages needed to decrease the proportion of susceptible nodes to a specified level.

Let us first consider the push variant. In this case, we make the rather striking observation that the value of s depends only on the number of messages m that have been sent by the nodes. Indeed, each infected node picks peers independently at random to send the update to. That is, every single update message is sent to a node selected independently at random from the set of all the nodes. This means that the probability that a fixed node is in state S after a total of m update messages has been sent can be approximated by

$$s(m) = (1 - \frac{1}{N})^m \approx \exp[-\frac{m}{N}] \tag{1.9}$$

Substituting the desired value of s, we can easily calculate the total number of messages that need to be sent in the system: it is

$$m \approx -N \log s \tag{1.10}$$

If we demand that $s = 1/N$, that is, we allow only for a single node not to see the update, then we need $m \approx N \log N$. This reminds us of the SI model, that had an $O(N \log N)$ message complexity to achieve full dissemination. If, on the other hand, we allow for a constant proportion of the nodes not to see the update ($s = 1/c$) then we have $m \approx N \log c$; that is, a linear number of messages suffice. Note that s or m cannot be set directly, but only through other parameters such as k.

Another notable point is that (1.9) holds irrespective of whether we apply a feedback mechanism or not, and irrespective of the exact algorithm applied to switch to state R. In fact, it applies even for the pure SI model, since all we assumed was that it is a push-only gossip with random peer selection. Hence it is a strikingly simple, alternative way to illustrate the $O(N \log N)$ message complexity result shown for the SI model: roughly speaking, we need approximately $N \log N$ messages to make s go below $1/N$.

Since m determines s irrespective of the details of the applied push gossip algorithm, the speed at which an algorithm can have the infected nodes send m messages determines the speed of convergence of s. With this observation in mind, let us compare a number of variants of SIR gossip.

Apart from Algorithm 2, one can implement termination (switching to state S) in several different ways. For example, instead of a probabilistic decision in procedure ONFEEDBACK, it is also possible to use a counter, and switch to state S after receiving the k-th feedback message. Feedback could be eliminated altogether, and moving to state R could depend only on the number of times a node has sent the update.

It is not hard to see that the the counter variants improve load balancing. This in turn improves speed because we can always send more messages in a fixed amount of time if the message sending load is well balanced. In fact, among the variants described above, applying a counter without feedback results in the fastest convergence. However, parameter k has to be set appropriately to achieve a desired level of s. To set k and s appropriately, one needs to know the network size. Variants using a feedback mechanism achieve a somewhat less efficient load balancing but they are more robust to the value of k and to network size: they can "self-tune" the number of messages based on the feedback. For example, if the network is large, more update messages will be successful before the first feedback is received.

Lastly, as in the SI model, it is apparent that in the end phase the pull variant is much faster and uses fewer update messages. It does this at the cost of constantly sending update requests.

We think in general that, especially when updates are constantly being injected, the push-pull algorithm with counter and feedback is probably the most desirable alternative.

1.2.3 Applications

We first explain how the various protocols we discussed were applied at Xerox for maintaining a consistent set of replicas of a database. Although we cannot provide a complete picture here (see [4]), we elucidate the most important ideas.

In Section 1.2.1 we identified two sub-problems, namely update spreading and error correction. The former is implemented by an SIR gossip protocol, and the latter by an SI protocol. The SIR gossip is called *rumor mongering* and is run when a new update enters the system. Note that in practice, many fresh updates can piggyback a single gossip message, but the above-mentioned convergence properties hold for any single fixed update.

The SI algorithm for error correction works for every update ever entered, irrespective of age, simultaneously for all updates. In a naive implementation, the entire database would be transmitted in each cycle by each node. Evidently, this is not a good idea, since databases can be very large, and are mostly rather similar. Instead, the nodes first try to discover what the difference is between their local replicas by exchanging compressed descriptions such as checksums (or lists of checksums taken at different times) and transmit only the missing updates. However, one cycle of error correction is typically much more expensive than rumor mongering.

The SI algorithm for error correction is called *anti-entropy*. This is not a very fortunate name: we should remark here that it has no deeper meaning than to express the fact that "anti-entropy" will increase the similarity among the replicas thereby increasing "order" (decreasing randomness). So, since entropy is usually considered to be a measure of "disorder", the name "anti-entropy" simply means "anti-disorder" in this context.

In the complete system, the new updates are spread through rumor mongering, and anti-entropy is run occasionally to take care of any undelivered updates. When such an undelivered update is found, the given update is redistributed by re-inserting it as a new update into the database where it was not present. This is a very simple and efficient method, because update spreading via rumor mongering has a cost that depends on the number of other nodes that already have the update: if most of the nodes already have it, then the redistribution will die out very quickly.

Let us quickly compare this solution to the earlier, email based approach. Emailing updates and rumor mongering are similar in that both focus on spreading a single update and have a certain small probability of error. Unlike email, gossip has no bottleneck nodes and hence is less sensitive to local failure and assumes less about local resources such as bandwidth. This makes gossip a significantly more scalable solution. Gossip uses slightly more messages in total for the distribution of a single update. But with frequent updates in a large set of replicas, the amortized cost of gossip (number of messages per update) is more favorable (remember that one message may contain many updates).

In practical implementations, additional significant optimizations have been performed. Perhaps the most interesting one is *spatial gossip* where, instead of picking a peer at random, nodes select peers based on a distance metric. This is important because if the underlying physical network topology is such that there are bottleneck

links connecting dense clusters, then random communication places a heavy load on such links that grows linearly with system size. In spatial gossip, nodes favor peers that are closer in the topology, thereby relieving the load from long distance links, but at the same time sacrificing some of the spreading speed. This topic is discussed at great length in [12].

We should also mention the removal of database entries. This is solved through "death certificates" that are updates stating that a given entry should be removed. Needless to say, death certificates cannot be stored indefinitely because eventually the databases would be overloaded by them. This problem requires additional tricks such as removing most but not all of them, so that the death certificate can be reactivated if the removed update pops up again.

Apart from the application discussed above, the gossip paradigm has recently received yet another boost. After getting used to Grid and P2P applications, and witnessing the emergence of the huge, and often geographically distributed data centers that increase in size and capacity at an incredible rate, in the past years we had to learn another term: *cloud computing* [7].

Cloud computing involves a huge amount of distributed resources (a cloud), typically owned by a single organization, and organized in such a way that for the user it appears to be a coherent and reliable service. Examples include storage service or Grid-style computing services. Recently big players in the IT sector have introduced their own cloud computing solutions, for example, Google and IBM [16], and Amazon [1].

These applications represent cutting edge technology at the time of writing, and it is not always clear how they work due to corporate secrecy, but from several sources it seems rather evident that gossip protocols are involved. For example, after a recent crash of Amazon's S3 storage service, the message explaining the failure included some details:

> (...) Amazon S3 uses a gossip protocol to quickly spread server state information throughout the system. This allows Amazon S3 to quickly route around failed or unreachable servers, among other things.[1] (...)

In addition, a recent academic publication on the technology underlying Amazon's computing architecture provides further details on gossip protocols [3], revealing that an anti-entropy gossip protocol is responsible for maintaining a full membership table at each server (that is, a fully connected overlay network with server state information).

1.3 Aggregation

The gossip communication paradigm can be generalized to applications other than information dissemination. In these applications some implicit notion of spreading

[1] `http://status.aws.amazon.com/s3-20080720.html`

information will still be present, but the emphasis is not only on spreading but also on *processing* information on the fly.

This processing can be for creating summaries of distributed data; that is, computing a global function over the set of nodes based only on gossip-style communication. For example, we might be interested in the average, or maximum of some attribute of the nodes. The problem of calculating such global functions is called data aggregation or simply *aggregation*. We might want to compute more complex functions as well, such as fitting models on fully distributed data, in which case we talk about the problem of *distributed data mining*.

In the past few years, a lot of effort has been directed at a specific problem: calculating averages. Averaging can be considered the archetypical example of aggregation. It is a very simple problem, and yet very useful: based on the average of a suitably defined local attribute, we can calculate a wide range of values. To elaborate on this notion, let us introduce some formalism. Let y_i be an attribute value at node i for all $0 < i \leq N$. We are interested in the average $\bar{y} = \sum_{i=1}^{N} y_i/N$. If we can calculate the average then we can calculate any mean of the form

$$g(z_1, \ldots, z_N) = f^{-1}\left(\frac{\sum_{i=1}^{N} f(z_i)}{N}\right), \tag{1.11}$$

where $f()$ is a suitable function. For example, $f(x) = \log x$ generates the geometric mean, while $f(x) = 1/x$ generates the harmonic mean. Clearly, if $y_i = f(z_i)$ then we can easily calculate $g(z_1, \ldots, z_N) = f^{-1}(\bar{y})$. In addition, if we calculate the mean of several powers of y_i, then we can calculate the empirical moments of the distribution of the values. For example, the (biased) estimate of the variance can be expressed as a function over averages of y_i^2 and y_i:

$$\sigma_N^2 = \bar{y^2} - \bar{y}^2 \tag{1.12}$$

Finally, other interesting quantities can be calculated using averaging as a primitive. For example, if every attribute value is zero, except at one node, where the value is 1, then $\bar{y} = 1/N$, so the network size is given by $N = 1/\bar{y}$.

In the remaining parts of this section we focus on several gossip protocols for calculating the average of node attributes.

1.3.1 Algorithms and Theoretical Notions

The first, perhaps simplest, algorithm we discuss is push-pull averaging, presented in Algorithm 3.

Each node periodically selects a random peer to communicate with, and then sends the local estimate of the average x. The recipient node then replies with its own current estimate. Both participating nodes (the sender and the one that sends the reply) will store the average of the two previous estimates as a new estimate.

Algorithm 3 push-pull averaging

```
1: loop
2:     wait(Δ)
3:     p ← random peer
4:     send push(x) to p
5: end loop

6: procedure onPushUpdate(m)
7:     send pull(x) to m.sender
8:     x ← (m.x + x)/2
9: end procedure
10:
11: procedure onPullUpdate(m)
12:     x ← (m.x + x)/2
13: end procedure
```

Similarly to our treatment of information spreading, Algorithm 3 is formulated for an asynchronous message passing model, but we will assume several synchronicity properties when discussing the theoretical behavior of the algorithm. We will return to the issue of asynchrony in Section 1.3.1.1.

For now, we also treat the algorithm as a one-shot algorithm; that is, we assume that first the local estimate x_i of node i is initialized as $x_i = y_i$ for all the nodes $i = 1 \dots N$, and subsequently the gossip algorithm is executed. This assumption will also be relaxed in Section 1.3.1.2, where we briefly discuss the case, where the attributes y_i can change over time and the task is to continuously update the approximation of the average.

Let us first look at the convergence of the algorithm. It is clear that the state when $x_i = \bar{y}$ for all i is a fixed point, assuming there are no node failures and message failures, and that the messages are delivered without delay. It is not very difficult to convince oneself that the algorithm converges to this fixed point in probability. We omit the technical details of the proof; the trick is to first observe that the *sum* of the approximations remains constant throughout. More precisely,

$$\sum_{i=1}^{N} x_i(t) = N\bar{y} \tag{1.13}$$

for any t. This very important property is called *mass conservation*. We then look at the difference between the minimal and maximal approximations and show that this difference can only decrease and, furthermore, it converges to zero in probability, using the fact that peers are selected at random. But if all the approximations are the same, they can only be equal to $\bar{y}$ due to mass conservation.

The really interesting question, however, is the *speed* of convergence. The fact of convergence is easy to prove in a probabilistic sense, but such a proof is useless from a practical point of view without characterizing speed. The speed of the protocol is illustrated in Figure 1.1. The process shows a diffusion-like behavior. Recall, that diffusion is executed on top of a random network over the pixels and not on the two dimensional grid. The arrangement of the pixels is for illustration purposes only.

It is possible to characterize convergence speed by studying the variance-like statistic

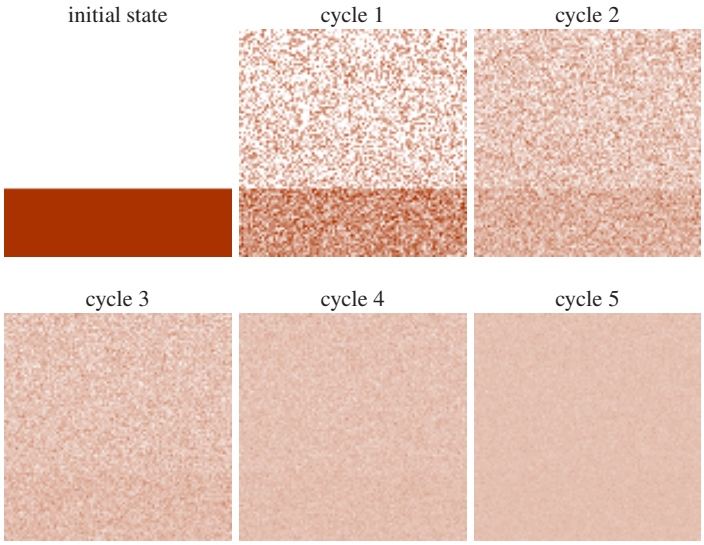

Fig. 1.1 Illustration of the averaging protocol. Pixels correspond to nodes (100x100 pixels=10,000 nodes) and pixel color to the local approximation of the average.

$$\sigma_N^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{y})^2, \tag{1.14}$$

which describes how accurate the set of current estimates is. It can be shown (see [11, 9]) that

$$E(\sigma_N^2(t+1)) \leq \frac{1}{2}\sigma_N^2(t), \tag{1.15}$$

where the time index t indicates a cycle. That is, variance decreases by a constant factor in each cycle. In practice, 10-20 cycles of the protocol already provide an extremely accurate estimation: the protocol not only converges, but it converges very quickly as well.

1.3.1.1 Asynchrony

In the case of information dissemination, allowing for unpredictable and unbounded message delays (a key component of the asynchronous model) has no effect on the correctness of the protocol, it only has an (in practice, marginal) effect on spreading speed. For Algorithm 3 however, correctness is no longer guaranteed in the presence of message delays.

To see why, imagine that node j receives a PUSHUPDATE message from node i and as a result it modifies its own estimate and sends its own previous estimate back to i. But after that point, the mass conservation property of the network will

Algorithm 4 push averaging

```
1: loop
2:     wait(Δ)
3:     p ← random peer
4:     send (x/2, w/2) to p
5:     x ← x/2
6:     w ← w/2
7: end loop

8: procedure ONUPDATE(m)
9:     x ← m.x + x
10:    w ← m.w + w
11: end procedure
```

be violated: the sum of all approximations will no longer be correct. This is not a problem if neither node j nor node i receives or sends another message during the time node i is waiting for the reply. However, if they do, then the state of the network may become corrupted. In other words, if the pair of push and pull messages are not *atomic*, asynchrony is not tolerated well.

Algorithm 4 is a clever modification of Algorithm 3 and is much more robust to message delay. The algorithm is very similar, but here we introduce another attribute called w. For each node i, we initially set $w_i = 1$ (so the sum of these values is N). We also modify the interpretation of the current estimate: on node i it will be x_i/w_i instead of x_i, as in the push-pull variant.

To understand why this algorithm is more robust to message delay, consider that we now have mass conservation in a different sense: the sum of the attribute values at the nodes *plus* the sum of the attribute values in the undelivered messages remains constant, for both attributes x and w. This is easy to see if one considers the active thread which keeps half of the values locally and sends the other half in a message. In addition, it can still be proven that the variance of the approximations x_i/w_i can only decrease.

As a consequence, messages can now be delayed, but if message delay is bounded, then the variance of the set of approximations at the nodes and in the messages waiting for delivery will tend to zero. Due to mass conservation, these approximations will converge to the true average, irrespective of how much of the total "mass" is in undelivered messages. (Note that the variance of x_i or w_i alone is not guaranteed to converge zero.)

1.3.1.2 Robustness to failure and dynamism

We will now consider message and node failures. Both kinds of failures are unfortunately more problematic than asynchrony. In the case of information dissemination, failure had no effect on correctness: message failure only slows down the spreading process, and node failure is problematic only if every node fails that stores the new update.

In the case of push averaging, losing a message typically corrupts mass conservation. In the case of push-pull averaging, losing a push message will have no effect, but losing the reply (pull message) may corrupt mass conservation. The solutions to this problem are either based on failure detection (that is, they assume a node

is able to detect whether a message was delivered or not) and correcting actions based on the detected failure, or they are based on a form of rejuvenation (restarting), where the protocol periodically re-initializes the estimates, thereby restoring the total mass. The restarting solution is feasible due to the quick convergence of the protocol. Both solutions are somewhat inelegant; but gossip is attractive mostly because of the lack of reliance on failure detection, which makes restarting more compatible with the overall gossip design philosophy. Unfortunately restarting still allows for a bounded inaccuracy due to message failures, while failure detection offers accurate mass conservation.

Node failures are a source of problems as well. By node failure we mean the situation when a node leaves the network without informing the other nodes about it. Since the current approximation x_i (or x_i/w_i) of a failed node i is typically different from y_i, the set of remaining nodes will end up with an incorrect approximation of the average of the remaining attribute values. Handling node failures is problematic even if we assume perfect failure detectors. Solutions typically involve nodes storing the contributions of each node separately. For example, in the push-pull averaging protocol, node i would store δ_{ji}: the sum of the incremental contributions of node j to x_i. More precisely, when receiving an update from j (push or pull), node i calculates $\delta_{ji} = \delta_{ji} + (x_j - x_i)/2$. When node i detects that node j failed, it performs the correction $x_i = x_i - \delta_{ji}$.

We should mention that this is feasible only if the selected peers are from a small fixed set of neighboring nodes (and not randomly picked from the network), otherwise all the nodes would need to monitor an excessive number of other nodes for failure. Besides, message failure can interfere with this process too. The situation is further complicated by nodes failing temporarily, perhaps not even being aware of the fact that they have been unreachable for a long time by some nodes. Also note that the restart approach solves the node failure issue as well, without any extra effort or failure detectors, although, as previously, allowing for some inaccuracy.

Finally, let us consider a dynamic scenario where mass conservation is violated due to changing y values (so the approximations evolved at the nodes will no longer reflect the correct average). In such cases one can simply set $x_i = x_i + y_i^{new} - y_i^{old}$, which corrects the sum of the approximations, although the protocol will need some time to converge again. As in the previous cases, restarting solves this problem too without any extra measures.

1.3.2 Applications

The diffusion-based averaging protocols we focused on will most often be applied as a primitive to help other protocols and applications such as load balancing, task allocation, or the calculation of relatively complex models of distributed data such as spectral properties of the underlying graph [8, 13]. Sensor networks are especially interesting targets for applications, due to the fact that their very purpose is data aggregation, and they are inherently local: nodes can typically communicate with

their neighbors only [19]. However, sensor networks do not support point-to-point communication between arbitrary pairs of nodes as we assumed previously, which makes the speed of averaging much slower.

As a specific application of aggregation, we discuss a middleware system called Astrolabe [18] in more detail, which is rumored to be in use at Amazon in some form. Astrolabe is all about aggregation, but the basic idea is *not* diffusion. Rather, aggregates are propagated along a virtual hierarchical structure. However, Astrolabe *is* based on gossip in many ways, as we will see, and it is the most "real world" application of the aggregation idea we are aware of, so it is useful to devote a few paragraphs on it.

Let us start with the hierarchical structure Astrolabe is based on. First of all, each node is assigned a name (when joining) similar to a domain name, which consists of *zones*. For example a node could be called "/Hungary/Szeged/pc3", which assumes three zones: the root zone "/" and the three successively more specific zones.

Each zone has a descriptor: a list of attributes. The leaf zones (for example, "pc3") have the raw system attributes in their descriptor such as storage space, stored files, and so on. These attributes will be the input for aggregation. Descriptors of higher level zones contain summaries of lower level zones; accordingly, the root zone descriptor contains system wide aggregates.

The key idea is that fresh replicas of the zone descriptors are maintained at nodes that belong to the given zone *and* at nodes that are siblings of the given zone. For example, staying with the previous example, the descriptor of zone "Szeged" will be replicated at all the nodes in the same zone, and in addition, at all the nodes in the zones under "Hungary". This way, any node under "Hungary" can compute the descriptor of "Hungary" (they will all have a replica of the descriptors of all the zones under "Hungary"). The descriptor of "Hungary" will in turn be replicated in each zone under the root zone, etc. Eventually every single node is able to compute the descriptor of the root zone as well.

It should be clear by now that the implementation boils down to replicating databases, as we saw in the previous sections on information dissemination, since each node has a set of zone descriptors that need to be replicated. Indeed, Astrolabe uses a version of anti-entropy for this purpose; only the situation is a bit more delicate, because not every node is interested in all the descriptors. As we explained, each node is interested only in the descriptors of its own zones (for example, "/", "Hungary/", "Szeged/", and "pc3/") and the descriptors that are siblings of some of its own zones. This way, the system can avoid replicating the potentially enormous complete set of descriptors, while still being able to calculate the descriptors for its own zones locally.

This twist has two important implications: first, not all nodes will need a list of all other nodes, and second, nodes can learn more from nodes that are closer to them in the hierarchy so they will want to communicate to such nodes more often. This requires a non-trivial peer selection approach. Astrolabe itself uses the descriptors themselves to store contacts for all zones of interest. These contacts can then be used to gossip with (running the anti-entropy over the common interest of the two nodes). This has the advantage that as long as a node has at least one contact from any of

Algorithm 5 The push-pull gossip algorithm skeleton.

```
1: loop
2:     wait(Δ)
3:     p ← selectPeer()
4:     send push(state) to p
5: end loop
6: procedure onPushUpdate(m)
7:     send pull(state) to m.sender
8:     state ← update(state,m.state)
9: end procedure
10:
11: procedure onPullUpdate(m)
12:     state ← update(state,m.state)
13: end procedure
```

the zones it is interested in (which can be the root zone as well) it will eventually fill in the contact list for each relevant zone automatically.

As usual, there are a number of practical issues, such as detecting failed nodes (and zones), the joining procedure, and other details involving asynchrony, the timestamping of updates, and so on. These can be solved without sacrificing the main properties of the system, although at the cost of losing some of its original simplicity.

Finally, we should point out that Astrolabe is an extremely generic system that supports much more than average calculation. In fact it supports an SQL-like language for queries on the virtual database that is defined by the set of descriptors of all the zones in the system. Astrolabe behaves as if it was a complete database, while in fact the nodes replicate only a small (roughly logarithmic) proportion of the complete set of data and emulate a complete database via progressive summaries.

1.4 What is Gossip after all?

So far we have discussed two applications of the gossip idea: information dissemination and aggregation. By now it should be rather evident that these applications, although different in detail, have a common algorithmic structure. In both cases an active thread selects a peer node to communicate with, followed by a message exchange and the update of the internal states of both nodes (for push-pull) or one node (for push or pull).

It is rather hard to capture what this "gossipness" concept means exactly. Attempts have been made to do so, with mixed success [14]. For example, periodic and local communication to random peers appears to be a core feature. However, in the SIR model, nodes can stop communicating. Besides, we have seen that in some gossip protocols neighboring nodes need not be random in every cycle but instead they can be fixed and static. Attempting to capture any other aspect appears to lead to similar philosophical problems: any characterization that covers all protocols we intuitively consider gossip seems to cover almost all distributed protocols. On the other hand, restrictions invariably exclude protocols we definitely consider gossip.

Accordingly, we do not intend to characterize what this common structure is, but instead we propose the template (or design pattern) shown in Algorithm 5. This

template covers our two examples presented earlier. In the case of information dissemination the state of a node is defined by the stored updates, while in the case of averaging the state is the current approximation of the average at the node. In addition, the template covers a large number of other protocols as well.

One notable application we have not covered in this chapter is the construction and management of *overlay networks*. In this case the state of a node is a set of node addresses that define an overlay network. (A node is able to send messages to an address relying on lower layers of the networking stack; hence the name "overlay".)

In a nutshell, the state (the set of overlay links) is then communicated via gossip, and nodes select their new set of links from the set of all links they have seen. Through this mechanism one can create and manage a number of different overlay networks such as random networks, structured networks (like a ring) or proximity networks based on some distance metric, for example semantic or latency-based distance.

These networks can be applied by higher level applications, or by other gossip protocols. For example, random networks are excellent for implementing random peer sampling, a service all the algorithms rely on in this chapter when selecting a random peer to communicate with.

Apart from overlay management, other applications include load balancing, resource discovery and assignment, and so on.

1.5 Conclusion

In this chapter we discussed two applications of the gossip communication model: information dissemination and aggregation. We showed that both applications use a very similar communication model, and both applications provide probabilistic guarantees for an efficient and effective execution.

We should emphasize that gossip protocols represent a departure from "classical" approaches in distributed computing where correctness and reliability were the top priorities and performance (especially speed and responsiveness) was secondary. To put it simply: gossip protocols—if done well—are simple, fast, cheap, and extremely scalable, but they do not always provide a perfect or correct result under *all* circumstances and in *all* system models. But in many scenarios a "good enough" result is acceptable, and in realistic systems gossip components can always be backed up by more heavy-weight, but more reliable methods that provide eventual consistency or correctness.

A related problem is malicious behavior. Unfortunately, gossip protocols in open systems with multiple administrative domains are rather vulnerable to malicious behavior. Current applications of gossip are centered on single administrative domain systems, where all the nodes are controlled by the same entity and therefore the only sources of problems are hardware, software or network failures. In an open system nodes can be selfish, or even worse, they can be malicious. Current secure gossip al-

gorithms are orders of magnitude more complicated than basic versions, thus losing many of the original advantages of gossiping.

All in all, gossip algorithms are a great tool for solving certain kinds of very important problems under certain assumptions. In particular, they can help in the building of enormous cloud computing systems that are considered the computing platform of the future by many, and provide tools for programming sensor networks as well.

Key Points

- Gossip protocols are based on periodic information exchange, during which all nodes communicate with randomly selected peer nodes. This gossip communication model supports the implementation of a number of functions such as information dissemination, aggregation or overlay topology construction.
- Gossip protocols represent a probabilistic mindset where we trade off deterministic guarantees of success (replacing it with strong probabilistic guarantees) for a significant increase in flexibility and scalability.
- Push-pull gossip-based broadcasting of an update can reach a time complexity of $O(\log N)$ and a message complexity of $O(N \log \log N)$, which approaches the performance of tree-based broadcasting, while being extremely robust and scalable to benign failure.
- Gossip protocols are a key component in the emerging cloud computing systems, as well as in sensor networks and other fully distributed dynamic systems, which makes them highly relevant in practical applications.

Acknowledgements While writing this chapter, M. Jelasity was supported by the Bolyai Scholarship of the Hungarian Academy of Sciences.

Problems - Exercises

1.1. We have seen that in some protocols it is useful to keep track of the age of the update; that is, to know how long ago a given update was inserted. If clocks are synchronized, this is easy: one needs only a timestamp. Try to think of algorithms for the various algorithms we discussed for keeping track of or at least approximating the age of the update if the clocks are *not* synchronized! To make the problem harder, suppose that the cycles are not synchronized either (that is, all the nodes have the same period Δ, but they start the cycle at a random time). Can you do anything if messages have a random unpredictable and unbounded delay?

1.2. We assumed that the updates arrive unexpectedly and that nodes that do not have the update do not know that they miss it. What difference does it make if we know the list of updates that are expected? What optimizations are possible for push, pull and push-pull? What kind of running times and message complexities can we reach?

1.3. We discussed the speed of spreading under strong synchrony assumptions: we assumed cycles are synchronized and messages are delivered without delay or errors. Think about the speed of spreading if we drop the synchrony assumption. Is it slower or faster?

1.4. We said that if messages can be delayed then push-pull averaging can get corrupted. Work out an example that proves this!

1.6 Further Reading

Gossip-Based Networking. Captures very well the state of the art of the field at the date of publication. Contains mostly overview and position papers evolved during a Lorentz Center workshop in Leiden, a rich source of references and ideas. (A.-M. Kermarrec, and M. van Steen, editors, ACM SIGOPS Operating Systems Review 41., 2007.)

Epidemic algorithms for replicated database maintenance. This is a citation classic that is percieved as the work that initiated this area of research. It is still very well worth reading; our discussion was partly based on this paper. (Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D., In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), pp. 1–12. ACM Press, 1987.)

The mathematical theory of infectious diseases and its applications. An old book that is still a good starting point for the relevant theoretical aspects. (N. T. J. Bailey, Griffin, London, second edition, 1975.)

References

1. Amazon Web Services: `http://aws.amazon.com`
2. Bailey, N.T.J.: The mathematical theory of infectious diseases and its applications, second edn. Griffin, London (1975)
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP'07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 205–220. ACM, New York, NY, USA (2007). DOI 10.1145/1294261.1294281
4. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), pp. 1–12. ACM Press, Vancouver, British Columbia, Canada (1987). DOI 10.1145/41840.41841

5. Dunbar, R.: Grooming, Gossip, and the Evolution of Language. Harvard University Press (1998)
6. Golub, G.H., Van Loan, C.F.: Matrix Computations, third edn. The Johns Hopkins University Press (1996)
7. Hand, E.: Head in the clouds. Nature **449**, 963 (2007). DOI 10.1038/449963a
8. Jelasity, M., Canright, G., Engø-Monsen, K.: Asynchronous distributed power iteration with gossip-based normalization. In: A.M. Kermarrec, L. Bougé, T. Priol (eds.) Euro-Par 2007, *Lecture Notes in Computer Science*, vol. 4641, pp. 514–525. Springer-Verlag (2007). DOI 10.1007/978-3-540-74466-5_55
9. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems **23**(3), 219–252 (2005). DOI 10.1145/1082469.1082470
10. Karp, R., Schindelhauer, C., Shenker, S., Vöcking, B.: Randomized rumor spreading. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00), pp. 565–574. IEEE Computer Society, Washington, DC, USA (2000). DOI 10.1109/SFCS.2000.892324
11. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03), pp. 482–491. IEEE Computer Society (2003). DOI 10.1109/SFCS.2003.1238221
12. Kempe, D., Kleinberg, J., Demers, A.: Spatial gossip and resource location protocols. J. ACM **51**(6), 943–967 (2004). DOI 10.1145/1039488.1039491
13. Kempe, D., McSherry, F.: A decentralized algorithm for spectral analysis. In: STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, pp. 561–568. ACM, New York, NY, USA (2004). DOI 10.1145/1007352.1007438
14. Kermarrec, A.M., van Steen, M. (eds.): ACM SIGOPS Operating Systems Review 41 (2007). Special issue on Gossip-Based Networking
15. Kimmel, A.J.: Rumors and Rumor Control: A Manager's Guide to Understanding and Combatting Rumors. Lawrence Erlbaum Associates (2003)
16. Lohr, S.: Google and I.B.M. join in 'cloud computing' research. The New York Times (2008)
17. Pittel, B.: On spreading a rumor. SIAM Journal on Applied Mathematics **47**(1), 213–223 (1987). DOI 10.1137/0147013
18. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems **21**(2), 164–206 (2003). DOI 10.1145/762483.762485
19. Xiao, L., Boyd, S., Lall, S.: A scheme for robust distributed sensor fusion based on average consensus. In: IPSN'05: Proceedings of the 4th international symposium on Information processing in sensor networks, p. 9. IEEE Press, Piscataway, NJ, USA (2005). DOI 10.1109/IPSN.2005.1440896

Solutions

Problems of Chapter 1

Since the problems here are supposed to make the reader think (as opposed to drills), we can give only hints instead of complete solutions.

1.1 Let us assume that nodes cooperate. If in each hop the gossip message contains the local time at the sender then differences in local times at the sender and the target can be dealt with, as long as the transmission time is near zero (like in wireless networks) or at least small and bounded. Otherwise, if a statistical model of transmission delay is available, then one can still have a rather good estimate, since estimation errors average out over several hops, as long as the model is unbiased. But sometimes transmission time cannot be approximated or modeled a priory. In such cases one may learn such models on the fly, using several tricks, such as injecting special messages, counting the hops of the message, and observing their time of arrival back at the sender, and so on. One needs to be creative here.

1.2 It makes a lot of difference. The possibilities for optimizations are endless, with the situation being similar to some file-sharing systems, where indeed nodes do know the file chunks that float around and that they are supposed to have. Accordingly, one can apply strategies for trying to even out the spreading of each simultaneous message (like the rarest-first strategy of BitTorrent), one can try to get more aggressive when a given message should have arrived a long time ago (according to the predicted spreading time), and so on. Also, pull-based strategies are of course more efficient here, since we can stop pulling if we do know that no more updates are around.

1.3 Asynchronous spreading can be faster, because "lucky" messages can move on to the next hop immediately when they arrive at a node. With smaller and smaller probability, this phenomenon can repeat in quick succession, so some messages can in fact cover several hops in a very short time. This is impossible in the cycle-based model. The situation is akin to the difference between the Jacobi and Gauss-Seidel iterations in matrix computations [6].

1.4 Problems can arise, for example, if a node sends a push message, and then it receives a push message before receiving the answer to its own push message. If one attempts to make the value exchange between a pair of nodes (that is, sending a push message and receiving the answer) an atomic transaction, then highly non-trivial problems arise regarding deadlocks, let alone the complication of the protocol. Push only averaging is an elegant solution to the delay problem.