2018 IEEE 38th International Conference on Distributed Computing Systems

Token Account Algorithms: The Best of the Proactive and Reactive Worlds

Gábor Danner
University of Szeged, Hungary
Email: danner@inf.u-szeged.hu

Márk Jelasity
University of Szeged, Hungary and
MTA SZTE Research Group on AI
Email: jelasity@inf.u-szeged.hu

***Abstract*—Many decentralized algorithms allow both proactive and reactive implementations. Examples include gossip protocols for broadcasting and decentralized computing, as well as chaotic matrix iteration algorithms. In proactive systems, nodes communicate at a fixed rate in regular intervals, while in reactive systems they communicate in response to certain events such as the arrival of fresh data. Although reactive algorithms tend to stabilize/converge/self-heal much faster, they have serious drawbacks: they may cause bursts in bandwidth consumption, and they may also cause starvation when the number of messages circulating in the system becomes too low. Proactive algorithms do not have these problems, but nodes waste a lot of time sitting on fresh information. Here, we propose a novel family of adaptive protocols that apply rate limiting inspired by the token bucket algorithm to prevent bursts, but they also include proactive communication to prevent starvation. With the help of our traffic shaping service, some applications approach the speed of the reactive implementation, while maintaining strong guarantees regarding the total communication cost and burstiness. Due to the proactive component we can help maintain a certain level of activity despite losing messages due to faults or the application semantics. We perform simulation experiments in different scenarios including a real smartphone availability trace. Our results suggest up to a fourfold speedup in a broadcast application, and an order of magnitude speedup in the case of gossip learning, when compared to the purely proactive implementation.**

1. Introduction

Token bucket and leaky bucket algorithms and their variants have long been used for traffic shaping in packet switched networks. These algorithms control the rate at which packets are sent from or forwarded by a networked device. The primary motivation for applying such methods is to prevent large bursts of traffic to protect the network and also to enforce quality-of-service contracts by controlling the rate of traffic.

In the application layer, decentralized applications are also confronted by the issue of traffic shaping. However, since applications have many other key characteristics to worry about, such as performance and fault tolerance, traffic shaping methods have not received much emphasis. Take gossip-based broadcast, for example. The conventional approach is to simply adopt a *proactive* design pattern where nodes gossip periodically in regular intervals [1]. This solves the traffic shaping problem (we have a constant rate) so we can focus on other design decisions that are related to performance and fault tolerance.

In this study, we challenge this design philosophy. Our main message is that fine details of traffic shaping actually have a profound effect on many key global application characteristics that seem unrelated to traffic shaping at first. For example, as we will show in detail, when gossip-based broadcast is implemented using our token account algorithm instead of the periodic, round-based communication pattern, convergence becomes dramatically faster, approaching the speed of flooding, without sacrificing the rate limiting feature (as flooding does).

The techniques discussed here are applicable in many decentralized asynchronous message passing applications where the main goal is to reach a target global state quickly and cheaply. These applications include gossip-based algorithms, asynchronous (chaotic) numeric algorithms, and distributed data mining as well. The common characteristics of these applications include nodes receiving messages, updating their state based on these messages, and sending messages as a function of their state.

In such applications, there is typically a large degree of freedom regarding the number and the scheduling of the outgoing messages. Unlike in the networking layer, where messages are simply forwarded, here the messages that are received and sent might be decoupled. The current practice does not exploit this design space fully; traffic shaping and its side-effects have not been given enough attention. There are two kinds of popular approaches, namely proactive and reactive. In a proactive approach, each node sends messages periodically, based on the information accumulated in the previous round. The rounds of the nodes in the system may or may not be synchronized. In a reactive approach, nodes immediately send messages whenever their state changes (typically after receiving a message).

In the proactive approach, time is often wasted, since nodes frequently sit on new information, doing nothing until the next round comes. However, traffic shaping is optimal due to the constant rate. In the reactive approach, information is spread much faster initially; however, the amount of traffic and its burstiness is out of control, which might harm the network as well as the application itself. Our goal here is to propose techniques that inherit the best properties of both approaches while avoiding their drawbacks.

We achieve this by generalizing the token bucket algorithm, introducing a family of token account algorithms. In a nutshell, at each node, these algorithms grant one token to the node in regular periods, and spend a token when the node sends a message. The details of when to send messages, how many, and how exactly to limit the number of accumulated tokens are captured with two functions: the proactive and reactive functions. This design space includes the purely proactive and reactive protocols and a spectrum of algorithms in between.

It should be mentioned that, unlike token bucket algorithms, our token account protocols are targeted to serve the application layer where they fulfill many functions at once, all of which are equally important: *rate limiting*, *speeding up convergence* and *fault tolerance*. The speedup effect is due to the reactive behavior that reduces the idle time, during which nodes sit on new information. Fault tolerance is due to the proactive behavior that maintains a certain level of messaging activity even when messages are lost due to faults or due to the semantics of the application.

Our contribution here is twofold. First, we introduce the token account service along with three different implementations. The parameters of these implementation allow us to span the design space between proactive and reactive algorithms. Second, we evaluate the proposed token account protocols using three applications (gossip learning, push gossip and chaotic power iteration) in simulation. Our scenarios include a real smartphone trace that represents a promising application domain for the class of asynchronous message passing algorithms discussed here.

2. Background

Here, we describe our system model and the three applications we selected to test our token account service: gossip learning, gossip-based broadcasting and chaotic power iteration. These applications are all based on local message passing and their goal is to converge to a desirable state through an iterative process as quickly and as cheaply as possible. Yet, they have a rather diverse set of requirements that allow us to demonstrate the broad applicability of our algorithms and to better cover their advantages and limitations. We first present the most common, proactive implementation of the demonstrator applications, and later on we shall reformulate them over the token account API (Algorithm 4).

2.1. System model

We model our system as a large set of nodes that communicate via message passing. In this study, we evaluate our protocols assuming a reliable transfer protocol, which implies that we do not consider message drop failure. However, the protocols themselves do not require this assumption. Nodes are allowed to leave the network at any time and node failures are also permitted. We do not require time to be synchronized over the nodes. At every point in time each node is assumed to have a set of neighbors, typically a small subset of the nodes. We assume that the failure of a neighbor is detected by the node. The neighbor set can change over time, but nodes can send messages only to their current neighbors. The set of neighbors might be a uniform random sample from the network or it might be defined by any fixed overlay network, depending on the application. When sending a message, nodes access their neighbors through the peer sampling service using the method SELECTPEER(). In this study we treat this method as a black box, noting that many implementations are available [2], [3] that might depend on the given networking environment and the application requirements as well.

Algorithm 1 Gossip Learning Framework

```
1: $(x, y) \leftarrow$ local training example
2: currentModel $\leftarrow$ initModel()
3: loop
4:     wait($\Delta$)
5:     $p \leftarrow$ selectPeer()
6:     send currentModel to $p$
7: procedure ONMODEL($m$)
8:     $m \leftarrow$updateModel($m, x, y$)
9:     currentModel $\leftarrow m$
```

2.2. Gossip learning

Our first demonstrator application that can take advantage of our token account service is gossip learning [4]. We present the most common, proactive implementation. The goal of this application is to learn from distributed data using stochastic gradient descent (SGD) [5]. We assume that every node in the network has only one training example (x, y), but we can benefit from having more local data. The set of these local examples form our machine learning database. We would like to learn a model over these instances in a fully distributed manner.

The basic idea is that in the network many models perform random walks and are updated at every node using the local example. The number of walking models is a parameter of the protocol; it may be as many as the number of nodes in the network. More precisely, every node executes Algorithm 1. A node in the network first initializes a local model, then iteratively sends its local model to a peer. When a node receives a model, it updates it by its locally stored training example using the SGD update rule, and then stores the updated model as its local model. Using this protocol the models stored by the nodes will converge to the same global optimum.

We intentionally present the simplest form of gossip learning, but more sophisticated techniques can also be applied (for example, local model averaging) to speed up convergence. These techniques also benefit from the token account service as the communication pattern remains identical.

2.3. Push gossip

Our second example application is the classical push gossip protocol [1], as shown in Algorithm 2.

In this simple setup, we assume that every node stores a single update, and whenever a new, fresher update arrives, it

Algorithm 2 Push gossip

```
1: update ← null
2: loop
3:     wait($\Delta$)
4:     $p$ ← selectPeer()
5:     send update to $p$
6: procedure ONUPDATE($m$)
7:     if $m$ is fresher than update then
8:         update ← $m$
```

Algorithm 3 Asynchronous iteration executed at node i

```
1: $b_{ki}$ ← any positive value for all k
2: loop
3:     wait($\Delta$)
4:     $x_i \leftarrow \sum_{k \in \text{in-neighbors}_i} A_{ik} b_{ki}$
5:     $p$ ← selectPeer()
6:     send weight $x_i$ to $p$
7: procedure ONWEIGHT($m$)
8:     $k$ ← m.sender
9:     $b_{ki} \leftarrow m.x$
```

replaces the old one. Furthermore, all the nodes periodically push the update they know about to a neighboring node. Here, we do not consider any stopping criteria as we assume that updates arrive frequently and continuously.

Although the push-pull variant is superior to push according to a number of performance metrics, and it could also be used alongside our token account service, we chose push for the sake of simplicity. This is because pull variants have benefits mainly in the final phase of convergence, which (as confirmed by our preliminary experiments) is not actually observed in our setup here due to the continuous stream of new updates.

2.4. Chaotic asynchronous power iteration

Our third example is power iteration. Given a square matrix A, vector x is an *eigenvector* of A with *eigenvalue* λ, if $Ax = \lambda x$. Vector x is a *dominant* eigenvector if there are no other eigenvectors with an eigenvalue larger than $|\lambda|$ in absolute value. In this case λ is a *dominant eigenvalue* and $|\lambda|$ is the *spectral radius* of A.

We concentrate of the abstract problem of calculating the dominant eigenvector of a weighted neighborhood matrix of some large network, in a decentralized way, when the elements of the vector are held by individual network nodes, one vector element per node. The matrix A is defined by physical or overlay *links* between the network nodes. More precisely, A contains the *weights* assigned to these links: let matrix element A_{ij} be the weight of the link from node j to node i. If there is no link from j to i then $A_{ij} = 0$.

In [6], Lubachevsky and Mitra present a chaotic asynchronous family of message passing algorithms to calculate the dominant eigenvector of a non-negative irreducible matrix, that has a spectral radius of one. Algorithm 3 shows an instantiation of this framework, that we will apply here.

In the algorithm, the values x_i represent the elements of the vector that converge to the dominant eigenvector. The values b_{ki} are buffered incoming weighted values from incoming neighbors in the graph. These values are not necessarily up-to-date, however, as shown in [6], the only assumption about message failure is that there is a finite upper bound on the age of these values. The age of value b_{ki} is defined by the time that elapsed since k sent the last update successfully received by i. This bound can be very large, so delays and message drop are tolerated to a very large extent.

3. Token Account Algorithms

The example algorithms presented so far were fully proactive sending messages in regular time intervals. This provides excellent load balancing, but slows down convergence. We could consider the naive reactive variants of these algorithms, where, instead of a regular timer, every message received would trigger message sending immediately. This would result in a faster convergence but the uncontrolled communication load would lead to large bursts of traffic. In our framework we introduce an abstraction that allows for a fine control over the tradeoff between these two approaches.

One idea to achieve this tradeoff is to apply the token bucket algorithm. In this algorithm, a token is assigned to the node in regular intervals of length Δ. The application works in purely reactive mode, spending one token per message. If no tokens are available, no sending is allowed (so sending is either skipped or blocked, depending on the application semantics). Our approach is similar in spirit, but it offers a fine control over the proactive and reactive characteristics of the application and it also allows for application specific adaptation in a natural manner. This allows us to achieve almost optimal speedup while preventing bursts and providing fault tolerance as well.

3.1. Token Account Framework

In our framework, each node has an *account*, which can hold a non-negative integer number of tokens. We introduce two functions that will control the proactive and reactive behavior of the node as a function of the number of tokens.

The *proactive function* PROACTIVE(a) returns the probability of sending a proactive message as a function of the account balance a. We require that the proactive function should be monotone non-decreasing in a, that is, a higher balance should not result in a lower probability of sending a proactive message.

The *reactive function* REACTIVE(a, u) returns the number of messages that the node will send as a reaction to an incoming message, as a function of the account balance a and the usefulness of the received message u. Clearly, the higher the balance the more messages we might want to send so the function should be monotone non-decreasing in a. The usefulness u expresses the notion that some messages are more important than others in most applications. For example, in the broadcast application, the received message is useful if and only if it contains new information for the

Algorithm 4 Token account

```
1: $a \leftarrow$ initial number of tokens
2: **loop**
3:     wait($\Delta$)
4:     **do with probability** proactive($a$)
5:         $p \leftarrow$ selectPeer()
6:         $m \leftarrow$ createMessage()
7:         send $m$ to $p$
8:     **else**
9:         $a \leftarrow a + 1$
10:    **end do**
11: **procedure** ONMESSAGE($m$)
12:     $u \leftarrow$ updateState($m$)
13:     $x \leftarrow$ randRound(reactive($a, u$))
14:     $a \leftarrow a - x$
15:     **for** $i \leftarrow 1$ to $x$ **do**
16:         $p \leftarrow$ selectPeer()
17:         $m \leftarrow$ createMessage()
18:         send $m$ to $p$
```

node. Currently we assume that u is a Boolean value (the message is either useful or not). Finer grading is possible in the future. The function should be monotone non-decreasing in u as well, that is, more useful messages should not result in fewer reactive messages being sent. Also, the value returned is at most a (we do not allow overspending).

The purely proactive strategy is a special case given by PROACTIVE(a) $\equiv 1$ and REACTIVE(a,u) $\equiv 0$. With relaxing the non-negativity constraint of the balance, the purely reactive strategy can be expressed as well as PROACTIVE(a) $\equiv 0$ and REACTIVE(a,u) $\equiv k$ (or REACTIVE(a,u) $\equiv uk$) for a constant $k \geq 1$.

The pseudo-code for the token account algorithm is shown in Algorithm 4. In each round, the node either sends a message to a peer, or saves the token for later use; the former occurs with probability PROACTIVE(a). When receiving a message, the application-specific code updates the state of the node using method UPDATESTATE() that also returns the usefulness of the received message. Next, the reactive function returns the number of messages to be sent and the same number of tokens are removed from the account. The return value r of the reactive function is probabilistically rounded by sampling $\lfloor r \rfloor + \xi$ where ξ is a random variable with the distribution $\xi \sim$ Bernoulli($r - \lfloor r \rfloor$).

The framework can be instantiated by implementing the proactive and reactive functions. We will discuss our proposed implementations in Section 3.3. First, however, we turn to the implementation of our three application examples within the framework.

3.2. Applications within the framework

To implement our applications in the framework we have to provide the application specific implementations of two methods: CREATEMESSAGE() that is responsible for constructing a message to be sent based on the current state, and UPDATESTATE(m) that is responsible for updating the current state based on the new message that has been received. This includes defining the usefulness of the received message m because UPDATESTATE(m) has to return this information.

The implementation of CREATEMESSAGE() is simple in all three cases: we just copy the current state. In the gossip learning application, the state consists of a machine learning model with a counter (age) that keeps track of how many times the given model was updated. In the push gossip application the state consist of an update with a timestamp. In chaotic iteration the state is the value x_i at node i.

In our three applications, the implementations of UPDATESTATE(m) are given by ONMODEL, ONUPDATE and ONWEIGHT, respectively. In addition, we have to return usefulness, as we explain below. In gossip learning, usefulness is 0 if the current model of the node is older (in terms of the number of visited nodes) than the received model, and 1 otherwise. In the former case, the state is unchanged, while in the latter case, the received model is trained on the local data and stored as the new state. Note that in our simulations, we did not implement any actual machine learning tasks, but just simulated the age of the models as this forms the basis of our performance metric.

In the broadcast application, usefulness is 1 if and only if the received message contains a newer update than the locally stored update at the node. In our simulations, we considered the following scenario: new updates are regularly injected into random online nodes of the network. A newer update makes older updates obsolete, that is, only the freshest update known by the given node is stored and propagated. Our performance metric is the difference between the average timestamp of the freshest update known by each node and the timestamp of the freshest update in the whole network.

In the chaotic iteration application, usefulness is 1 if and only if the received message causes a change in the local state. Our convergence metric is the angle (or cosine distance) between the approximation of the eigenvector and the actual eigenvector that should converge to zero.

3.3. Implementations of the framework

Let us turn to the instantiations of the framework. In order to implement the framework, one has to provide the two functions PROACTIVE(a) and REACTIVE(a,u) taking into account the constraints we described previously. We have already described the implementation of the purely proactive solution within the framework as an example. Here we propose three additional implementations.

3.3.1. Simple token account. The first implementation is called *simple* token account. This implementation serves as a baseline, and it is similar to the token bucket algorithm although there are important differences as well. We introduce a parameter $C \geq 0$ that controls the capacity of the token account. That is, the maximal number of tokens will be C. Using this parameter, we define

$$\textsc{proactive}(a) = \begin{cases} 1 & \text{if } a \geq C \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

$$\textsc{reactive}(a, u) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Note that when $C = 0$ we have the purely proactive protocol as a special case. The reactive part is identical to that of the token bucket algorithm, however, this implementation also shows proactive behavior but only when the account is full. The account typically fills up with tokens when too few messages are arriving relative to the allowed communication rate. This in turn happens most often when, due to failures, fewer and fewer messages are circulating in the network. The default proactive behavior helps maintain a certain level of communication rate naturally even under high message drop rates, which is impossible in a purely reactive implementation. Of course, the effect of this error correction strongly depends on the application semantics.

3.3.2. Generalized token account. Our second implementation is called *generalized* token account. Here, our goal is to design a reactive function that is able to increase the number of messages sent when the number of tokens is high. In addition, we want to send twice as many messages in response to useful messages. To achieve this goal, the proactive function should be the same as the one in (1) and we propose the following reactive function:

$$\text{REACTIVE}(a,u) = \begin{cases} \lfloor (A-1+a)/A \rfloor & \text{if } u \\ \lfloor (A-1+a)/(2A) \rfloor & \text{otherwise,} \end{cases} \quad (3)$$

where parameter A is a positive integer and it controls what proportion of the available tokens we wish to use. Let us first consider the case when the incoming message was useful (u = true). Here, the reactive function is designed so that when $A = 1$ we use all the available tokens. Increasing A decreases the returned value. When $A \geq a$, the function returns 1. This also implies that the maximal meaningful value for A is $A = C$ in which case the reactive function will be equivalent to equation (2). Now, let us consider the case when u = false. Here, we simply divide the returned value by two. This also means that, due to rounding the output down to an integer, the function will return 0 when $A \geq a$. In other words, when the tokens are scarce, we do not waste them for reacting to messages that are not useful.

3.3.3. Randomized token account. So far all the strategies had the simple proactive function in equation (1). In the *randomized* token account implementation we propose a more fine-grained handling of proactive messages, while we will treat reactive messages in a similar way to the generalized token account implementation. In addition, instead of rounding it down to an integer, the reactive function will use the value of a similar formula as the expected value of a discrete distribution, from which a sample is returned.

Let us first discuss the proactive function given by

$$\text{PROACTIVE}(a) = \begin{cases} 0 & \text{if } a < A-1 \\ \dfrac{a-A+1}{C-A+1} & \text{if } a \in [A-1, C] \\ 1 & \text{otherwise.} \end{cases} \quad (4)$$

Parameters A and C have the same semantics as in previous implementations: C controls the capacity of the account, A controls the rate of spending the tokens. The actual formula might seem slightly ad-hoc, but it is derived from a few simple requirements. First, we wish the function to return 1 when $a \geq C$ as in all previous implementations. Second, we wish to add some proactive behavior even when $a < C$, so the returned value was chosen to be linear starting from $a = A - 1$ until $a = C$. The starting point of this linear segment was chosen to be $A - 1$ because if $a < A$ then the reactive function (to be discussed below) will be able to send less than one messages on average (in other words, we are not guaranteed to be able to respond to important messages) so in that range we wish to maintain the purely reactive behavior.

The reactive function is given by

$$\text{REACTIVE}(a,u) = \begin{cases} a/A & \text{if } u \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

Note that this time we apply no rounding, so the returned value might be less than 1. As shown in Algorithm 4, a randomized rounding is performed on this value to get an integer.

3.4. A note on rate limitation properties

The algorithm variants above have rather different reactive functions, some of them allowing for spending the full account at once. This means that the largest possible burst of traffic is defined by the largest possible account balance. Let us take a closer look at the maximum possible size of the account balance. For an arbitrary implementation of the token balance framework, let C be the smallest number for which PROACTIVE(C)= 1 holds. If there is no such C, it means the balance of the account might in principle grow indefinitely, which is not a desirable property, since we wish to limit the size of bursts. In our implementations we have such a C, in fact, it is an explicit parameter. Due to the definition of C, any additional tokens are guaranteed to be spent immediately when the account has at least C tokens. We call C the *token capacity* of the token strategy, that is, the maximal number of tokens that can be accumulated. This also gives an upper bound on the number of messages that a node may send within a period of time t: a node cannot send more than $\lceil t/\Delta \rceil + C$ messages, where Δ is the length of a proactive round.

4. Experimental Analysis

The overall goal of our experiments is to examine the speedup of our token account solutions relative to the baseline proactive implementations, while keeping the same overall communication rate. In order to evaluate our protocols, we ran simulation experiments using the PeerSim simulation environment [7]. Since one of our main targeted application domain is smartphone networks, our experimental scenarios include a real smartphone trace as well with realistic availability and churn patterns.

4.1. Experimental setup

The protocols we test consist of the combination of our three applications (gossip learning, push gossip, and chaotic iteration) and our three proposed instantiations of the token account framework: simple token account, generalized token account and randomized token account. These applications and implementations are described in sections 2 and 3 in detail. The token account protocols have two parameters: A and C. In our experiments we explore this parameter space.

The baseline proactive protocol is given as a special case of simple token account with $C = 0$; this variant is also included in our experiments. Note that the other extreme of the spectrum, namely the pure reactive strategy, is not included as a baseline, since it is obviously not a viable strategy. Without any rate control, our applications would generate a continuous burst and use up all the available bandwidth.

The number of initial tokens assigned to the nodes before the start of the experiment is zero. The communication topology (that is, the overlay network) was a fixed 20-out network (each node had 20 out neighbors that did not change through the experiment) and the network size was $N = 5,000$ or $N = 500,000$. The fixed 20 neighbors were drawn independently and uniformly at random. This is perhaps the simplest practical approximation of uniform peer sampling suitable for the applications we study here. It can be implemented by maintaining 20 TCP connections for the lifetime of the application. The value 20 allows for a robust connected network while the cost of managing the connections to all the 20 neighbors is still practically affordable. The chaotic iteration experiment uses a different topology as we describe later; this is because the 20-out network mixes too well and power iteration converges too fast over this topology.

As for timing, we simulate a virtual two-day period, with $\Delta = 172.80\,\mathrm{s}$, allowing for 1000 periods during the two-day interval. This is a long period so we allow only a very low utilization of the available bandwidth in all the applications, which is consistent with the requirements in the domains we target. In all the applications, we assume the transfer time for one message to be $1.728\,\mathrm{s}$, a hundredth of the proactive period. Again, the point here is that we wish to simulate a scenario where low bandwidth utilization is required, because in such a scenario it is much harder to achieve a convergence speed competitive with that of the purely reactive solution that utilizes all the bandwidth.

Regarding the failure patterns, we simulate the protocols in two scenarios. In the first scenario the communication and the nodes are reliable. In the second scenario, we simulate the protocols over a smartphone trace that captures realistic failure and accessibility patterns. In both cases, the same random 20-out network is used as the communication overlay, as described above. The trace was collected by STUNner [8]. In a nutshell, STUNner is an app that monitors and collects information about charging status, battery level, bandwidth, and NAT type.

Traces of varying lengths are available for 1191 different users. We divided these traces into 2-day segments (with a one-day overlap), resulting in 40,658 segments altogether. With the help of these segments, we were able to simulate a virtual 2-day period by assigning a different segment to each simulated node.

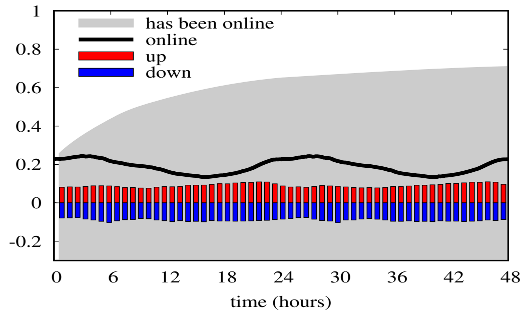

Figure 1. Proportion of users online, and proportion of users that have been online, as a function of time. The bars indicate the proportion of the simulated users that log in and log out (shown as a negative proportion), respectively, in the given period.

To ensure that our algorithm is phone and user friendly, we defined a user to be online (available) when the user has a network connection and the phone is connected to a charger, hence we never use battery power at all. In addition, we also treated those users as offline who had a bandwidth of less than 1 Mbit/s. Finally, we consider a user online only after it spent at least one minute on a charger with a network connection.

The observed churn pattern is illustrated in Figure 1 based on all the 2-day periods we identified. Although our sample contains users from all over the world, they are mostly from Europe, and some are from the USA. The indicated time is GMT, thus we did not convert times to local time. The diurnal pattern of availability is quite clear. During the night, more phones are available (as they tend to be on a charger), but the churn rate remains lower. Note that during the simulated 2-day period, about 30% of the users remain permanently offline based on our definition.

Let us now describe the specific settings for each application.

4.1.1. Gossip learning setup.

Our goal is to study the speed of convergence. In the case of gossip learning, the learning speed depends on how many nodes a given machine learning model can visit in a given amount of time. The maximal number of visited nodes at time t (let us denote this by $n^*(t)$) is achieved by the pure reactive strategy, where no model is ever delayed at any of the nodes. Since the transfer time for one model was assumed to be $1.728\,\mathrm{s}$, at any point in time t we have $n^*(t) = t/1.728$. Our performance metric is defined as the relative number of visited nodes compared to this ideal number. More precisely, let $n_i(t) (\leq n^*(t))$ denote the number of nodes that the model at node i has visited up to time t. Our performance metric at time t is

$$\frac{1}{N}\sum_{i=1}^{N}\frac{n_i(t)}{n^*(t)} = \frac{1}{Nn^*(t)}\sum_{i=1}^{N} n_i(t), \qquad (6)$$

where N is the size of the network. This describes the relative speed of our protocols compared to the maximal speed. Note that no actual machine learning task is necessary for this metric.

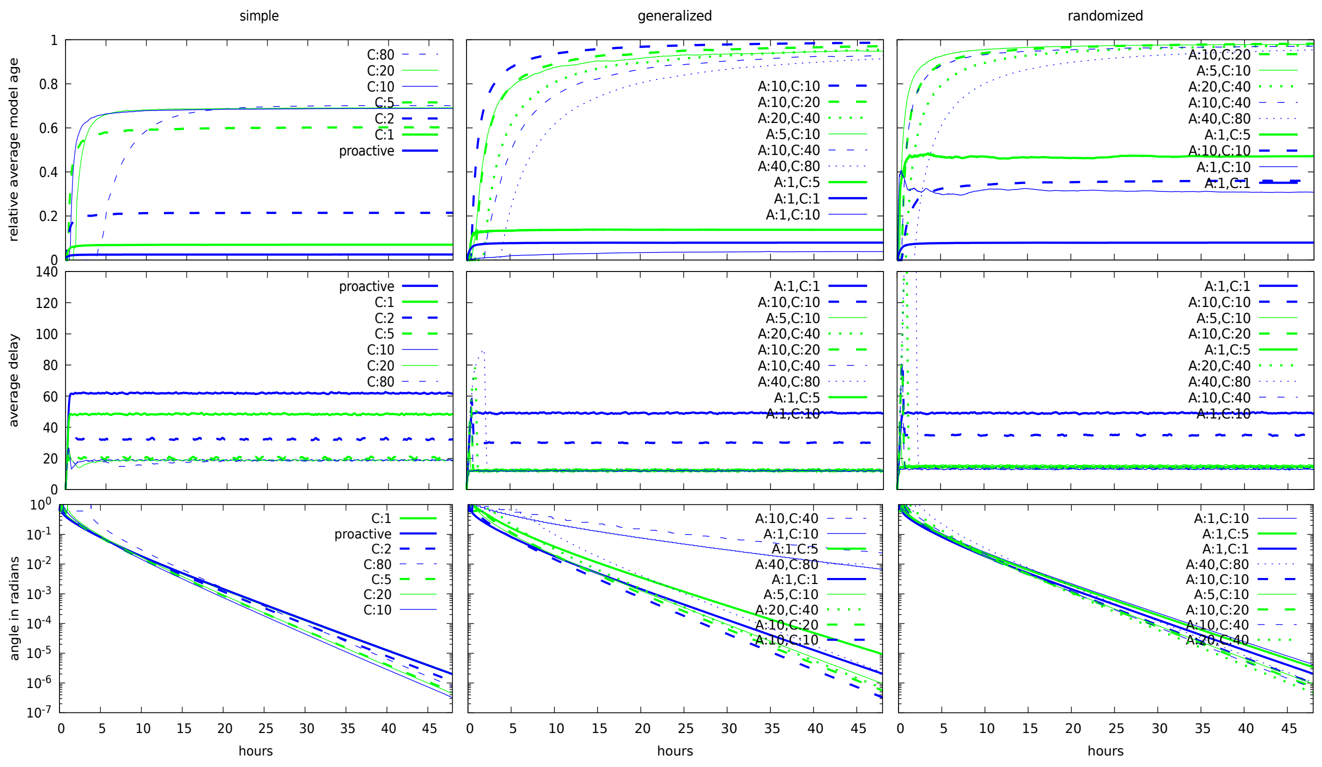

Figure 2. Token account strategies in the failure-free scenario for gossip learning (top row), push gossip (middle row) and chaotic iteration (bottom row).

4.1.2. Push gossip setup. In the case of push gossip, the spreading of a single update is relatively difficult to evaluate. For this reason, we inject new updates in regular time intervals at randomly selected nodes in the network. The period of inserting new updates is $17.28\,$s, that is, we insert 10 updates in every proactive period. Updates have a timestamp so every node can replace a locally stored update by a newer one. Our performance metric at time t is based on the average time lag experienced by the nodes relative to the freshest update available anywhere in the network at time t. For the sake of simplifying our notation, let us assume that at time t the freshest update is the t-th update, and let node i store the t_i-th update ($t_i \leq t$). Our performance metric for push gossip is

$$\frac{1}{N}\sum_{i=1}^{N}(t - t_i) = t - \frac{1}{N}\sum_{i=1}^{N} t_i. \qquad (7)$$

In the churn scenario, nodes that come back online first send a single initial pull request to a random online neighbor. If this neighbor has tokens, a message is sent back with the latest update (burning a token). Otherwise, no answer is given so the pull request is unsuccessful.

4.1.3. Chaotic iteration setup. In this application the overlay network not only defines the communication channels but it defines the computational task as well, since we are calculating the eigenvector of the normalized adjacency matrix itself. The 20-out matrix used in the other applications is not suitable because it converges very fast due to the good mixing properties of the network, which hides the effects of the different protocols. Here, we use an overlay network based on the Watts-Strogatz model in order to be able to control (slow down) the speed of convergence [9]. The network is based on a ring in which every node is connected to its closest 4 neighbors. In addition, we rewire every link to a random target with a probability of 0.01. The network size remains $N = 5000$.

The performance metric used in this application is simply the convergence rate of power iteration to the correct eigenvector expressed as the angle of the current approximation and the correct eigenvector. An angle of zero means a perfect solution. In the case of power iteration, there is no natural optimally fast protocol since the convergence speed also depends on the choice of the neighbors. Here, we simply present the convergence as a function of time for the different parameter settings, which still allows for a clear comparison among the different options.

4.2. Experimental results

We first explored the parameter space of the protocols. The parameter space included all the combinations defined by $A = 1, 2, 5, 10, 15, 20, 40$ and $C - A = 0, 1, 2, 5, 10, 15, 20, 40, 80$ (note that we have to have $A \leq C$). Based on these runs a representative selection is shown in Figure 2 for our three applications in the failure-free scenario. We performed 10 independent runs for every parameter combi-

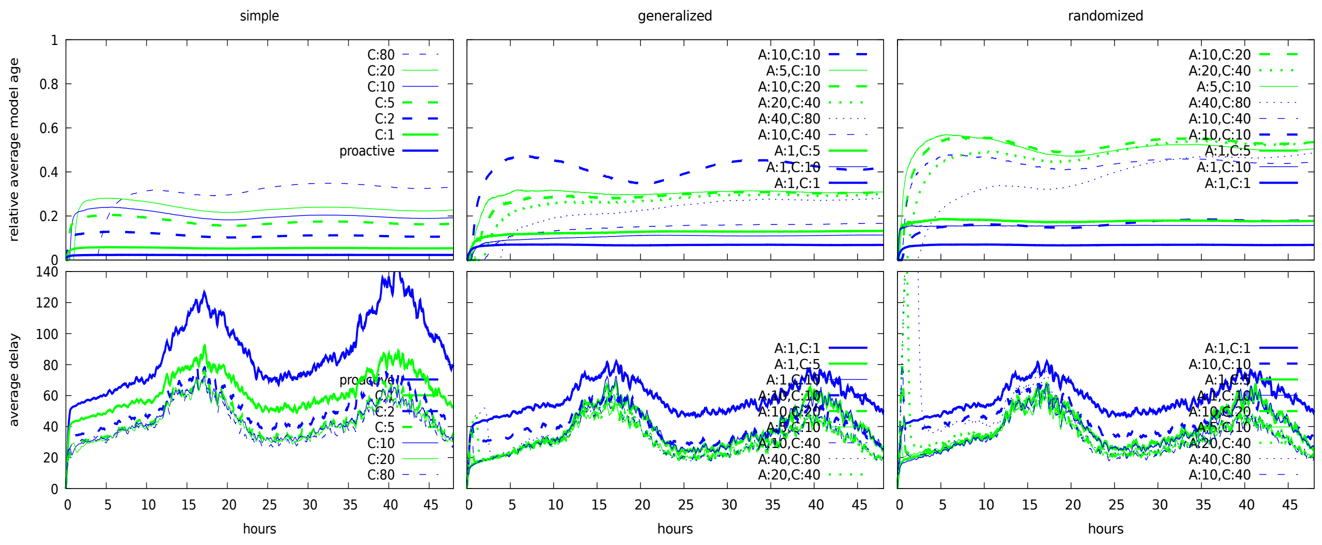

Figure 3. Token account strategies in the smartphone trace scenario for gossip learning (top row) and push gossip (bottom row).

nation, and the average of these runs is shown in the plots. On the plots showing push gossip we applied smoothing based on averaging measurements over 15 minute periods.

Note that in general it makes little sense to set C much larger than A, since a small A means an aggressive reactive message strategy (we spend most of our tokens), while a large C represents a very low probability of sending proactive messages. This combination results in a very poor error correction ability: if the number of messages in circulation decreases due to faults or due to the application semantics, we cannot replace them efficiently with proactive messages. This is because the aggressive reactive strategy quickly bursts all the tokens, but it takes a very long time until the account is full again (and so proactive messages can be sent).

The main conclusion from this exploration is that, relative to our purely proactive baseline, all the parameter combinations result in a very significant performance improvement in the case of gossip learning and push gossip, and we can also improve chaotic iteration significantly with most parameter combinations.

In the case of push gossip most of the parameter settings result in an almost identical performance, except two settings that are inferior. Clearly, in the broadcast example, it makes sense to be more aggressive in the reactive function and spread the fresh information to multiple nodes when possible; with $A = C$, only at most one reactive message is sent. Gossip learning is more sensitive to the parameter setting. Here, the key appears to be setting a large enough C, which allows us to accommodate the maximal variance in the number of random walks forwarded within a round. Fortunately, settings as low as $C = 20$ already provide close to optimal performance while still offering good rate limiting as well. Note also that larger values of C have a handicap in our experiments since we initialize the accounts to have zero tokens. In the long run, this disadvantage disappears.

It is interesting to note that some settings, such as $A = 10, C = 10$, behave quite differently in different applications. This setting is among the worst in push gossip for reasons mentioned above but it is the best in gossip learning and chaotic iteration. At the same time, $A = 10, C = 20$ is among the best in all three applications.

For the gossip learning application the results have an interesting implication. In this case, machine learning models walk nearly without any delay but the overall communication in the system is not more than in the proactive case. This is possible only if the number of models that walk in the network is less than that in the proactive case. In other words, the token account service has a side-effect of reducing the number of models at the cost of speeding them up at the same time. In fact, we can observe an emergent evolutionary process in which random walks fight for bandwidth and only those survive that happen to reach a given node the soonest after the node received a token.

We performed the same exploration over the smartphone trace as well. Figure 3 illustrates the same parameter combinations as shown in the failure-free case. Note that nodes only receive tokens when online (and thus have a chance of actually spending it) and only the online nodes were considered when computing our performance metrics. The chaotic iteration application is not shown here, because in such an extremely dynamic setting with aggressive churn it is not possible to define convergence for this application and so our performance metric is not applicable. Apart from the apparent diurnal pattern due to the variation of node availability, the results are rather consistent with those in the failure-free scenario. Relative to the proactive strategy we achieve very significant improvements, of course, with the same overall communication cost as in the proactive strategy.

To illustrate the scalability of the protocols, we ran them over a network of size $N = 500,000$ in the failure free

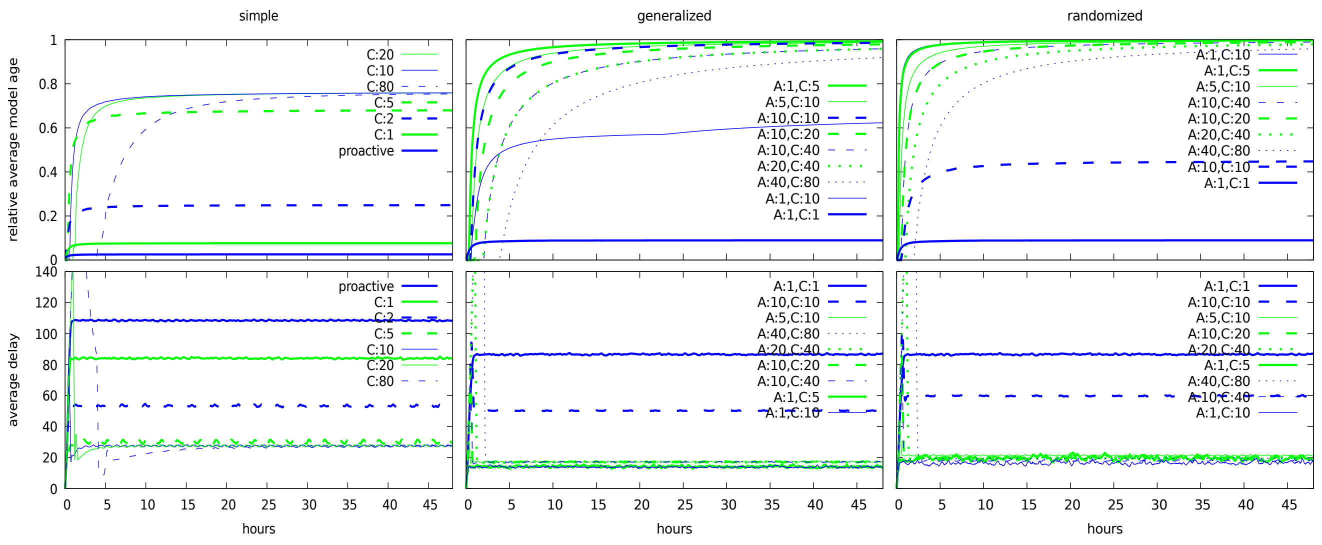

Figure 4. Token account strategies in the failure free scenario and $N = 500,000$ for gossip learning (top row) and push gossip (bottom row).

scenario. The results are shown in Figure 4. Comparing with the plots in Figure 2, it is clear that in the case of push gossip the protocols are still very robust to the parameter settings, since all the settings that allow for an exponential spreading of new updates (that is, where $C > A$) still have an almost identical performance. Of course, the average delay increases somewhat, but this is due to the larger diameter of the network: a logarithmic increase is expected even with flooding (the reactive variant) with increasing network size (note that our overlay network has a logarithmic diameter).

In the case of gossip learning, we can see that some of the best variants perform very similarly over different network sizes, with two notable exceptions: $A = 1, C = 5$, and $A = 1, C = 10$. These variants were among the worst in the small network but they are among the best in the large network. Note that these variants are the most aggressive reactive variants, they replicate the good random walks burning all the available tokens locally. The reason for the dramatic difference is that—due to finite size effects—in the small network all the random walks get stalled periodically, effectively rendering the dynamics similar to that of the proactive protocol. In the large network there are proportionally more random walks and at every point in time a few of these walks can still make progress and later also replicate to replace those walks that were less lucky.

Nevertheless, even for gossip learning, there are robust parameter choices, for example, $A = 5, C = 10$. This parameter setting is also suitable for push gossip in all the settings we examined.

As a final note, let us compare the performance of our different algorithm variants. Even SIMPLE represents a significant improvement over the proactive approach, but GENERALIZED and RANDOMIZED outperform it robustly. Considering the best parameter settings, GENERALIZED has a slight advantage over RANDOMIZED in the push gossip application, and the reverse is true in gossip learning.

4.3. A Note on the Number of Tokens

Although this study has a strong experimental focus, for completeness we present a short analytical derivation of the average number of tokens in the system. This property is interesting as the dynamics of the system depends on the available tokens. We assume a failure-free scenario. We use our previous notations, but here let $a(t)$ denote the average number of tokens over the nodes at time t and let $w(t)$ be the average number of messages sent (or, equivalently, received) by a node until time t. Now, we can write the mean-field model

$$\frac{da}{dt} = \frac{1}{\Delta} - \frac{dw}{dt} \tag{8}$$

$$\frac{d^2w}{dt^2} = \frac{dw}{dt}(\text{reactive}(a,u) - 1) + \frac{1}{\Delta}\text{proactive}(a) \tag{9}$$

The first equation states that a is increased by the constant rate of generated tokens (one per each cycle of length Δ) and decreased by the number of tokens used up. The second equation states that the change of the message sending rate is given by the number of reactive messages triggered by the incoming messages (also taking into account the fact that the one incoming message is "replaced" by the reactively generated messages triggered by it) and the number of proactive messages that are sent once in every cycle.

Now, assuming the equilibrium state when $da/dt = 0$ and $d^2w/dt^2 = 0$, solving the resulting equations gives us

$$1 = \text{reactive}(a,u) + \text{proactive}(a). \tag{10}$$

This can be solved for a for a fixed u. For the most promising version: randomized token account, solving the equation gives us $a = A \cdot C/(C+1)$ for $u = 1$ (this means $a \approx A$). The assumption $u = 1$ is acceptable for gossip learning where most incoming messages are better than

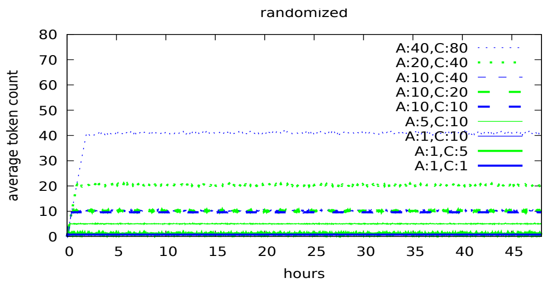

Figure 5. Average number of tokens (gossip learning, failure free scenario).

the locally stored random walk. Indeed, our validation runs (Figure 5) show a very good agreement with the predicted value.

5. Related Work

Raghavan et al. [10] used a gossip protocol to implement a distributed token bucket limiter, where the goal was to control the global aggregate traffic through the cooperation of individual rate limiters. This is orthogonal to our work, because we wish to control the local traffic at all the individual nodes, while at the same time we wish to optimize a global application-specific performance measure, such as speed of convergence.

Rodrigues et al. [11] used token buckets as part of their adaptive broadcast solution. There, the emission rates were adaptive and the token bucket was used to control the input rate, that is, the rate at which a node accepts new events to broadcast. The gossip protocol itself was purely proactive, thus the efficiency of the broadcast under a fixed cost (the focus of our work) was not addressed. Frey et al. [12] applied token bucket rate limiting over the upload links to evaluate the effect of limited bandwidth, but other options for rate limiting were not investigated.

Wolff et al. [13] present a distributed data mining approach based on a decentralized algorithm to test whether the Euclidean norm of the average of vectors is within a threshold. They apply a leaky bucket algorithm for rate limiting, which makes their system periodic (thus, proactive). Here, a significant improvement in convergence speed could be expected using simple token bucket algorithms instead, and further optimization using our various token account algorithms may be possible.

Another application area is decentralized replication schemes where the dominant approach used to be reactive (for example, replicate when the number of replicas is below a threshold). First, Sit et al. [14] proposed a fully proactive scheme to deal with bursts, and this was followed by several hybrid proactive/reactive systems. For example, Duminuco et al. [15] proposed an adaptive version of the proactive scheme as well as a hybrid scheme that switches to purely reactive operation when the availability of data is critically low. Controlling the available repair-budget with the help of a token account method is a promising approach in this area as well.

6. Discussion and Conclusions

In this paper, we introduced the token account service that serves as a communication layer for a large class of decentralized applications. This class includes asynchronous decentralized message passing applications such as gossip broadcast, gossip-based machine learning, and chaotic iteration methods. Any decentralized protocol might benefit from the service that is based on some form of periodic proactive local communication.

The main motivation was that we wanted to combine the advantages of proactive and reactive communication models. The reactive communication model has a crucial advantage: it often results in very fast convergence in several different applications. This is because nodes react immediately to new information, there is no idle time. However, since the number of messages is not controlled explicitly, they can generate too many or too few messages. Too many messages are generated when bursts occur due to cascading instantaneous reactions to propagating new information. But too few messages can also be generated since messages are sent only in response to other messages. If some of the messages are never delivered due to failures or due to application specific filters, the overall amount of communication can decrease and the system might even arrive at a complete standstill.

The proactive communication model controls the number of messages explicitly, but it often results in an inferior convergence speed due to sitting on new information until the next round starts.

Token account algorithms were demonstrated here to maintain a very tight control over the number of messages we send, yet they were also shown to achieve a very significant speedup relative to a purely proactive implementation. In the case of gossip learning, we saw that the token account algorithm approximates the speed of a “hot potato” random walk, when the walk wastes no time at any of the nodes. In the case of the push gossip application, the delay of receiving the freshest update is one third of that of the proactive implementation. We achieved a significant speedup even in the case of chaotic iteration.

In our future work we have several promising directions. One such direction is to examine chaotic iteration in more detail. Although we demonstrated that token account protocols are beneficial, we are convinced that we have only scratched the surface regarding the potential optimizations of chaotic iterations, a key tool in high performance computing. Another direction is the exploration of the implications of the method in decentralized data mining, our main area of research. Our significant speedup here opens up the possibility of the practical implementation of many complex machine learning models while keeping the communication cost low.

7. Acknowledgments

This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 (“Internet of Living Things”).

References

[1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*. Vancouver, British Columbia, Canada: ACM Press, August 1987, pp. 1–12.

[2] R. Roverso, J. Dowling, and M. Jelasity, "Through the wormhole: Low cost, fresh peer sampling for the internet," in *Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing (P2P 2013)*. IEEE, 2013.

[3] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems*, vol. 25, no. 3, p. 8, August 2007.

[4] R. Ormándi, I. Hegedűs, and M. Jelasity, "Gossip learning with linear models on fully distributed data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 556–571, 2013.

[5] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer Berlin Heidelberg, 2012, vol. 7700, pp. 421–436.

[6] B. Lubachevsky and D. Mitra, "A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit radius," *Journal of the ACM*, vol. 33, no. 1, pp. 130–150, January 1986.

[7] A. Montresor and M. Jelasity, "Peersim: A scalable P2P simulator," in *Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing (P2P 2009)*. Seattle, Washington, USA: IEEE, September 2009, pp. 99–100, extended abstract.

[8] Á. Berta, V. Bilicki, and M. Jelasity, "Defining and understanding smartphone churn over the internet: a measurement study," in *Proceedings of the 14th IEEE International Conference on Peer-to-Peer Computing (P2P 2014)*. IEEE, 2014.

[9] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, 1998.

[10] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 337–348.

[11] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec, "Adaptive gossip-based broadcast," in *International Conference on Dependable Systems and Networks (DSN-2003)*, June 2003, pp. 47–56.

[12] D. Frey, R. Guerraoui, A.-M. Kermarrec, and M. Monod, "Boosting gossip for live streaming," in *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. IEEE, August 2010, pp. 1–10.

[13] R. Wolff, K. Bhaduri, and H. Kargupta, "Local l2-thresholding based data mining in peer-to-peer systems," in *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA*. SIAM, 2006, pp. 430–441.

[14] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz, "Proactive replication for data durability," in *The 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006.

[15] A. Duminuco, E. Biersack, and T. En-Najjary, "Proactive replication in distributed storage systems using machine availability estimation," in *Proceedings of the 2007 ACM CoNEXT Conference*, ser. CoNEXT '07. New York, NY, USA: ACM, 2007, pp. 27:1–27:12.