Noname manuscript No.
(will be inserted by the editor)

Asynchronous Privacy-preserving Iterative Computation on Peer-to-peer Networks

J.A.M. Naranjo · L. G. Casado · Márk Jelasity

Received: date / Accepted: date

Abstract Privacy preserving algorithms allow several participants to compute a global function collaboratively without revealing local information to each other. Examples of applications include trust management, collaborative filtering, and ranking algorithms such as PageRank. Most solutions that can be proven to be privacy preserving theoretically are not appropriate for highly unreliable, large scale, distributed environments such as peer-to-peer networks because they either require centralized components, or a high degree of synchronism among the participants. At the same time, in peer-to-peer networks privacy preservation is becoming a key requirement. Here, we propose an asynchronous privacy preserving communication layer for an important class of iterative computations in peer-to-peer networks, where each peer periodically computes a linear combination of data stored at its neighbors. Our algorithm tolerates realistic rates of message drop and delay, and node churn, and has a low communication overhead. We perform simulation experiments to compare our algorithm to related work. The problem we use as an example is power iteration (a method used to calculate the dominant eigenvector of a matrix), since eigenvector computation is at the core of several practical applications. We demonstrate that our novel algorithm also converges in the presence of realistic node churn, message drop rates and message delay, even when previous synchronized solutions are able to make almost no progress.

Keywords asynchrony · churn · power iteration · privacy preservation · P2P

Final version published in *Computing* 94(8-10):763–782, 2012, doi:10.1007/s00607-012-0200-5

J.A.M. Naranjo and L. G. Casado
Dept. of Computer Architecture and Electronics, University of Almería,
Agrifood Campus of International Excellence (ceiA3), Spain
Telephone: +34 950 214393, E-mail: jmn843@ual.es

Márk Jelasity
University of Szeged, and Hungarian Acad. Sci., Research Group on AI, Hungary

1 Introduction

Large scale fully distributed networked systems are becoming more and more widespread. Examples include peer-to-peer systems, sensor networks, smart power grids, pervasive systems, and more recently, networks of people with smart phones. High performance computing and even multi-core computing architectures increasingly resemble these systems as well, as the number of computing elements is scaling up.

In these systems many useful functions can be implemented through global computations based on local attributes of the nodes. Some examples of the extensive literature on the topic include algorithms for data aggregation [16, 18, 26], spectral analysis [19], trust management [17], and distributed ranking and data mining [6, 10, 25]. These algorithms can simply monitor a system, or they can be used for control and optimization as well.

These networks face two important challenges. The first is that in any application, synchronization is a potential source of major performance and reliability issues. The reason is that in a large scale system there will always be nodes that fail, which makes synchronization points very complicated to implement. In addition, with synchronization load balancing becomes a major issue. The advantages of asynchronous algorithms have long been known (see [11] for a survey) and in the systems we target they are the only viable alternative. The second challenge is the unreliability of communication. Failing nodes can cause synchronization, load balancing, and reliability problems, but unreliable communication channels (for example, if a message can get lost undetected) can cause numerical inaccuracies, deadlocks, and other inconsistent behavior as well.

At the same time, the aspect of privacy preservation has been receiving increasing attention in recent years, due to legal and ethical concerns with the practice of the IT industry of collecting and analyzing vast amounts of personal data centrally. Peer-to-peer (P2P) networks are viewed as natural candidates for providing an alternative to centrally managed databases of user profiles and user behavior.

P2P networks are characterized by their large scale, and their extreme dynamism and unreliability. Computing global functions is already a challenge in these systems, but achieving an acceptable degree of privacy as well is even more complicated. In particular, if we wish to achieve a provable, non-stochastic, level of security, then the inherent unreliability of a P2P network that might involve nodes joining and leaving, and messages that are lost and delayed is a key factor to consider.

Privacy preservation has long been an active area of research in data mining [2, 20] and security [29]. A common approach to privacy preserving data mining involves adding noise to data records in such a way that noise is canceled out in the final result. In this paper, we seek a stronger sense of privacy based on cryptographic primitives that provide provable guarantees in well-specified circumstances. Cryptographic approaches often involve features that make them unsuitable for P2P networks, such as relying on a centralized

component, extensive communication involving all nodes, a large overhead to achieve privacy, or a requirement for synchronization. The nearest approach to our work was proposed by Bickson et al [5]. However, their algorithm also suffers from a number of drawbacks related to flexibility and synchronization.

The class of computations we tackle consists of iterative algorithms that rely on calculating a linear combination of values at neighboring nodes in each iteration. Our contribution is twofold: (1) we present a privacy preserving algorithm for distributed iteration that is extremely fault tolerant and has a low privacy-related overhead and (2) we evaluate the algorithm experimentally using the power method as an example, where we demonstrate that the algorithm tolerates realistic message drop rates and message delay, unlike synchronized solutions, and show that it converges in the presence of churn as well.

The outline of the paper is as follows. After discussing related work in Section 2, Sections 3 and 4 introduce the background and the motivation for our work. Section 5 describes our proposed algorithm and discusses its properties. Next, the strength of the privacy we provide is discussed in Section 6. Section 7 presents experimental results for different failure scenarios. Section 8 concludes the paper.

2 Related work

Techniques for privacy preservation have been summarized in several good surveys [1,2,20]. Privacy preserving algorithms are often distributed, since the point is to avoid collecting data to a central location. Clifton et al presented an early conceptual framework that identified primitives for creating privacy preserving distributed data processing algorithms [8] based on secret sharing schemes [30]. The primitives described include sum, set intersection and set union computations. These primitives can be combined to implement a large number of different algorithms.

Out of these primitives, computing the sum securely has received a lot of attention. For example, He et al [14] study global sum computation in wireless sensor networks. However, they do not achieve full asynchronism and node churn is not addressed either. Besides, their algorithm is not iterative in nature. Das et al [9] focus on sum computations in P2P networks. The solution they propose is similar to ours in that nodes can adjust their desired level of privacy. However, it is not fully asynchronous as message loss and node churn have not been taken into account.

Despite their potential applications mentioned in the Introduction, fully distributed privacy preserving iterative computing has not yet been studied very widely. Iterative computations can be implemented in a surprisingly fault tolerant way [11, 21]. A large class of such computations can also be implemented using sum as a primitive operation; this time in a local sense: nodes calculate the sum of the values of their neighbors, or in general a separable function of the neighbor values [24]. As mentioned above, the only study in

this area we are aware of was published by Bickson et al [5, 7]. Their solution can cope with message loss to some degree, but it cannot tolerate high failure rates and node churn.

3 Background

We assume that we are given a set of N nodes. The nodes communicate by exchanging messages. In order to send a message, the sender node needs to know only the network address of the target node. The messages can be lost or delayed. Each node can send a message to any other node provided that the address of the target is known. We assume that each node is known by at least k other nodes. The "knows about" relation defines a directed *overlay network*. From now on, when we refer to the *network*, we mean the overlay network of N nodes defined above. The overlay network is assumed to be dynamic: nodes can leave but they are assumed to rejoin eventually remembering their previous state and neighbors.

We assume that each node i has a variable x_i, and that each overlay link (i, j) is assigned a weight w_{ij}. First, at each node i we would like to compute the sum

$$\sum_{j \in IN^{[i]}} w_{ji} x_j, \tag{1}$$

in a privacy preserving way, where $IN^{[i]}$ is the set of in-neighbors of i. We also define $OUT^{[i]}$ as the set of out-neighbors of i. By computing the sum in a privacy preserving way we mean that we require that the private value x_j for each node j remains a secret, that is, no other node in the network will learn this value during the computation, yet the correct sum is computed and becomes known to node i only.

Once we are able to compute this sum, we can compute global functions, with additional assumptions. For example, if these sums are computed iteratively in a synchronized fashion, then we can implement many iterative methods including one for solving linear systems of equations or finding the dominant eigenvector of the matrix $W = [w_{ij}]_{i,j=1}^{N}$ [13]. For example, the well-known power method [3] for calculating the dominant eigenvector of W is based on the iteration defined by

$$x_i^{(t+1)} = \sum_{j \in IN^{[i]}} w_{ji} x_j^{(t)}, \tag{2}$$

where the vector $[x_i^{(t)}]_{i=1}^{N}$ converges to the dominant eigenvector of W under very mild conditions provided that the spectral radius of W is 1. Several key algorithms—including PageRank—rely on finding the dominant eigenvector of an appropriate matrix [4].

If we relax the assumption on synchronization, then many of the algorithms mentioned above can still be executed in a rather simple way, under very similar conditions [11, 15, 21]. This is a crucial point, because in P2P networks

it is essential to apply asynchronous methods. However, not all approaches for implementing privacy preservation tolerate asynchronism equally well. Our contribution lies in supporting asynchronous methods.

Let us now define a few basic concepts taken from cryptography that will be used to propose a privacy preserving layer for the problem above. The notion of k-out-of-n secret sharing ($k \leq n$) refers to a method that is used to share a secret with n nodes in such a way that any subset of k nodes can recover the secret by combining the information they have, but no subset of at most $k-1$ nodes is able to do so. One example is Shamir's secret sharing [27]: let node i have a secret $s \in \mathcal{F}$, where $\mathcal{F}$ is any algebraic field. Node i first generates $k-1$ random coefficients $a_1, \ldots, a_{k-1}$ to define the random polynomial $P(x) = s + a_1 x + \cdots + a_{k-1} x^{k-1}$ over $\mathcal{F}$. Note that $P(0) = s$, so the polynomial can be used to recover the secret. Now, node i can send the value of the polynomial at different random points to n different nodes. For example, if nodes have a unique ID from $\mathcal{F}$, then node j can receive $P(j)$. It is clear from algebra that any k nodes can determine the original coefficients of the polynomial via interpolating it using their point-value pairs, hence they can recover the secret. It is also easy to see that any $k-1$ nodes will have no information about the secret at all.

Finally, we let the adversary model be the so-called *semi-honest* adversary model, a common assumption in cryptography literature [12]. In this model it is assumed that the nodes follow the protocol, but the adversary has access to the internal state of corrupted nodes as well as all the messages they receive. It is important to note that here adversaries are not assumed to have the power to eavesdrop on arbitrary overlay links without corrupting nodes. This is not a crucial restriction, since secure messaging using asymmetric encryption eliminates eavesdropping.

4 How to Share the Secret?

Our motivation is to propose an efficient and scalable algorithm that tolerates realistic failure scenarios. The algorithm that most closely fulfills our goal was proposed by Bickson et al [5, 7]. In this section, we compare this algorithm with our novel proposal.

Bickson et al compute the sum in (1) based on Shamir's secret sharing scheme. From now on, we call their algorithm the SSS method. We note that although the SSS method has been extended to cope with malicious adversaries [7], it has not been extended to deal with the realistic failure models we target in this paper, so we discuss the version presented in [5].

As mentioned above, the SSS algorithm is based on Shamir's secret sharing algorithm. Let us assume that we need to compute (1), and node i has $|IN^{[i]}| = n$ neighbors. Let us pick a constant $k \leq n$, which will define the strength of the scheme. The SSS algorithm requires each $j \in IN^{[i]}$ to first share its value $w_{ji}x_j$ with all the other neighbors of i using the k-out-of-n Shamir's secret sharing scheme. As a result, a node $j \in IN^{[i]}$ will receive the values $P_1(j), \ldots, P_n(j)$

from the other neighbors of i, where P_l is the polynomial generated by node l. Now, let us define polynomial P of degree $k-1$ by

$$P(x) = \sum_{l \in IN^{[i]}} P_l(x) \tag{3}$$

The key observation is that $P(0)$ equals the sum in (1). Furthermore, a node $j \in IN^{[i]}$ can compute $P(j)$ locally. In the final step, all the neighbors of i send $P(j)$ to node i. This allows node i to interpolate polynomial P (if at least k messages are received), and then to find the sum in (1) by calculating $P(0)$.

The algorithm tolerates the loss of up to $|IN^{[i]}| - k$ messages to node i containing values $P(j)$, but each in-neighbor j of i must receive every share sent to it by all the other in-neighbors of i in order to be able to send a correct $P(j)$ value to i.

The message complexity of the above scheme is $O(n^2 + n)$ for each node because all the possible pairs of nodes in the set $IN^{[i]}$ have to communicate with each other, and then they all send a message to node i. Furthermore, the neighbors of node i have to agree on a common value of k. Nodes are not allowed to make unilateral decisions to, for example, increase or decrease k to adapt to observed failure scenarios. In general, any change in parameter values has to be agreed on in the neighbor-set of each node.

Apart from having to agree on a common parameter k, the members of a neighbor set $IN^{[i]}$ have to agree on group membership as well. A node cannot make a unilateral decision on joining or leaving the group. Any change in the membership has to be communicated and agreed on by the group, since the correctness of the protocol crucially depends on all nodes expecting input from the same set of nodes; otherwise it cannot be guaranteed that all nodes send an evaluation of the same polynomial P to node i.

The basic idea behind our proposal is to apply a k-out-of-k secret sharing scheme based on splitting the secret value into a random sum [22]. That is, for a secret $s \in \mathcal{F}$, let the first $k-1$ shares be random elements of $\mathcal{F}$: $s_1, \ldots, s_{k-1} \in \mathcal{F}$, and let

$$s_k = s - \sum_{i=1}^{k-1} s_i. \tag{4}$$

We can apply this scheme as a privacy layer to calculate (1) similarly to Shamir's secret sharing: all the neighbors of i send shares of their secret value to other neighbors of i, these are then summed up and sent to i. In the final step, node i simply needs to sum all messages it receives. This way we can relax many assumptions of SSS. For example, now every node $j \in IN^{[i]}$ can set its own value of k, since nodes in $IN^{[i]}$ no longer need to collect input from the same set of nodes. Also, each node $j \in IN^{[i]}$ may choose a different set of share recipients.

We apparently pay a price for this increased flexibility: we lose some of the redundancy in the final step, when the neighbors of i send their summaries to i, since all the neighbor messages have to reach node i, while in SSS it was

sufficient to receive k out of n neighbor messages. However, in reality, the sum-splitting scheme is more robust. To see this, one has to take into account every message that must be delivered successfully, not only those that are sent in the final step, since the overall message complexity determines the probability of a successful round. The message complexity of the splitting-based secret sharing algorithm in one round is $O(nk + n)$, while for SSS it is $O(n^2 + n)$. If $k \approx n$ then the robustness of the two schemes are similar. If $k \ll n$ then $O(nk + n)$ is much more favorable than $O(n^2 + n)$ so, despite the reduced redundancy in the final step, the significantly fewer messages that have to pass through mean more robustness to benign failure overall. For the sake of completeness, we should mention that in our complete final algorithm $O(n)$ additional control messages are sent in one round as well, but this does not affect the above conclusion.

Most importantly, our secret sharing approach allows us to support fully chaotic asynchronous iterations. Asynchronism is desirable, because achieving strictly synchronous rounds of updates in highly unreliable environments is extremely difficult and expensive, especially in the presence of churn. An asynchronous iteration method does not rely on rounds, instead, each message can be processed independently. Such an adaptation is possible only if the iterative numerical computation that is being implemented also supports asynchronism [11, 15, 21]. But this is not enough: the secret sharing scheme also has to be flexible enough to fully support the asynchronous version of the computation.

The sum-splitting secret sharing scheme meets this requirement owing to the following property: any share can be computed from the secret value and the subset of the other $k - 1$ shares. This allows node j to recompute s_k whenever its secret value changes without refreshing the rest of the shares. In addition, any node can change any of its secret shares without changing the remaining shares.

In contrast to this, polynomial interpolation as used in SSS is less compatible with asynchronism because the value of the polynomial computed by each node directly depends on all the secret values. As a change in any secret value or any random coefficient in the neighborhood of node i affects all the neighbors, it is clear that a complete round has to be executed after any change.

5 Implementing Power Iteration

From now on, we concentrate on *power iteration* (see (2)), and we present our asynchronous algorithm in this context. For other numerical methods that support asynchronism, like those that solve systems of linear equations, the same method is applicable with trivial modifications. The implementation is based on the asynchronous method of Lubachevsky and Mitra [21].

We first introduce some notations to express the ideas presented in the previous section in this context, and then we describe the protocol. Let us

pick a node $j \in IN^{[i]}$. As mentioned before, node j will have to use the k-out-of-k secret sharing method based on random sums. For this, node j needs a set of collaborators from $IN^{[i]}$. The size and composition of this set can be freely chosen by node j. Let us denote this collaborator set by $OC^{[ji]} \subseteq IN^{[i]}$ (out-collaborators).

According to the random sum secret sharing method, node j generates a random share $s^{[j\alpha i]}$ for each $\alpha \in OC^{[ji]}$. The value $s^{[j\alpha i]}$ will be shared with node α, while node j keeps

$$m^{[ji]} = w_{ji}x_j - \sum_{\alpha \in OC^{[ji]}} s^{[j\alpha i]}. \tag{5}$$

This equation is equivalent to (4) using our new notation.

During the iteration, $s^{[j\alpha i]}$ has to be occasionally changed to maintain security. It is up to node j when it changes these values (that is, when it generates new ones); all nodes can do this independently. Since we assume there is no synchronization, we assign a *version number* t to the shares, and we use the notation $s_t^{[j\alpha i]}$. From now on, when the version number t is omitted from the notation, we assume t is the latest existing version.

Node j can also be part of the collaborator group of other nodes from $IN^{[i]}$. We denote the set of nodes that have node j in their collaborator group by $IC^{[ji]}$ (in-collaborators). That is, $IC^{[ji]} = \{\beta \mid j \in OC^{[\beta i]}\}$. Assuming that both $IC^{[ji]}$ and $OC^{[ji]}$ are available at node j, as well as the random shares from all the nodes in $IC^{[ji]}$, node j can compute

$$M^{[ji]} = m^{[ji]} + \sum_{\beta \in IC^{[ji]}} s^{[\beta ji]} = w_{ji}x_j - \sum_{\alpha \in OC^{[ji]}} s^{[j\alpha i]} + \sum_{\beta \in IC^{[ji]}} s^{[\beta ji]}, \tag{6}$$

which is the value that must be sent to node i. If we could guarantee that there is no delay or message loss, and that all nodes have completely up-to-date information about the secret shares, then node i could simply compute

$$x_i = \sum_{j \in IN^{[i]}} M^{[ji]} = \sum_{j \in IN^{[i]}} w_{ji}x_j \tag{7}$$

to update its own state according to the power iteration.

Figure 1 illustrates one round of the synchronous version of the communication scheme we propose, using the notation defined above. Let us assume nodes $\{2, 3, 4, 5\} = IN^{[1]}$ want to share a linear combination of their private values with node 1. The left part of the figure depicts the nodes sending random shares to other randomly chosen in-neighbors of 1. Note that the number of shares, i.e., the value of k, can be different for every node. The right part of the figure shows the nodes sending the final messages to 1. Node 1 simply adds these final messages to obtain the linear combination.

However, we do assume that there is delay, message loss, and churn, so it is not guaranteed that the information that reaches node i is consistent: secret shares that belong to a given node can have different version numbers, or the collaborator set views could be inconsistent at different members of the set.

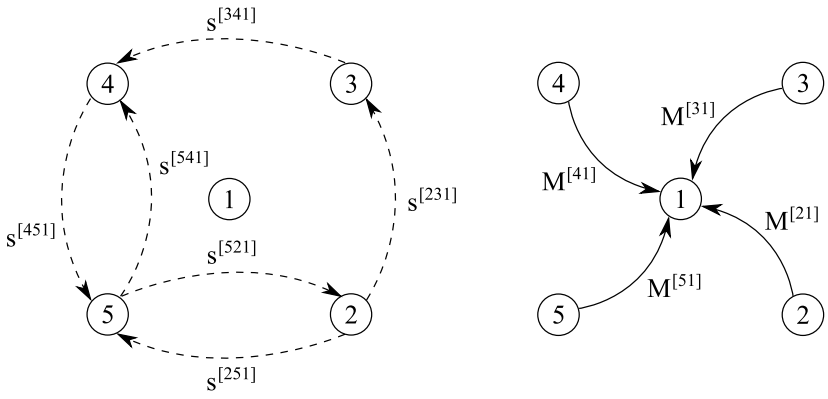

Fig. 1: Communication during the sum-splitting scheme.

To handle this problem, node j has to send additional information along with $M^{[ji]}$ to make it possible to check the consistency at node i, so node j also sends two sets to node i:

$$LOC^{[ji]} = \{(j, \alpha, t) \mid \alpha \in OC^{[ji]}\} \tag{8}$$

$$LIC^{[ji]} = \{(\beta, j, t) \mid \beta \in IC^{[ji]}\}, \tag{9}$$

where j, α, β, t are taken from $s_t^{[j\alpha i]}$ and $s_t^{[\beta ji]}$. Based on these sets, node i will update its value according to (7) only if the following condition is satisfied:

$$\bigcup_{j \in IN^{[i]}} LOC^{[ji]} = \bigcup_{j \in IN^{[i]}} LIC^{[ji]}, \tag{10}$$

which expresses the fact that all collaborator groups are consistent, and use the same version of the secret shares.

Due to our unreliable message passing model, node j needs proof that a given out-collaborator l actually uses the latest share that it was sent. Out-collaborators can be offline, and messages can get lost or delayed. If node j incorrectly assumes that node l uses the latest version of the share, then the iteration will not progress due to inconsistency. For this reason, node j requires proof that a new share was delivered *and* at least one message with this share has reached node i, before actually switching to the latest version.

Note that this proof is needed on a node-by-node basis, hence we need no consensus among the collaborators. In fact, it is typical that a different number of share-updates are performed by node j with different collaborators. This also means that if node l has not actually received the new share, node j will not switch, and the iteration will progress, even if node l is offline.

To provide proof, in each round, node i sends back all active share-parameters that involve j. That is, node i sends back the set of triplets (j, α, t) and (β, j, t) that refer to j. Hence, j can now check which version of the shares are effectively in use. If the version of some of the shares is not the latest, node j resends the corresponding shares. We call the above set of triplets a checklist, and will denote it by $CH^{[ji]}$.

5.1 Algorithm Description

After having introduced the notations and the basic ideas, we can now elaborate on the algorithm, which is based on asynchronous message passing. We specify three types of messages:

Type 1 (random share): node $j \in IN^{[i]}$ sends $s_t^{[j\alpha i]}$ to node $\alpha \in OC^{[ji]}$.
Type 2 (checklist): node i sends to $j \in IN^{[i]}$ two sets,

$$CH_t^{[ji]} = \{(j, \alpha, t) \in \bigcup_{\alpha \in IN^{[i]}} LIC^{[\alpha i]},\ (\beta, j, t) \in \bigcup_{\beta \in IN^{[i]}} LOC^{[\beta i]}\},$$

and

$$online\_nodes_t^{[i]} = \{\gamma \in IN^{[i]},\ \text{that } i \text{ believes to be online}\},$$

Type 3: node $j \in IN^{[i]}$ sends $V_t^{[ji]} = (M^{[ji]}, LOC^{[ji]}, LIC^{[ji]})$ to node i.

Note that the subscripts t do not necessarily refer to the same value, they are relative to the entity they index. In the case of Type 2 and Type 3 messages a sequence number t is added so that nodes can drop delayed out-of-order messages. In $s_t^{[j\alpha i]}$ the index t denotes the active version number. The set $online\_nodes^{[i]}$ is maintained by each node i based on recording the senders of recently received messages. In a Type 2 message, node i sends this set to node j.

All the nodes execute the same algorithm. At every node, the algorithm is divided in two threads that run in parallel. The active thread is mainly in charge of sending information to other nodes, while a passive thread waits for incoming messages and reacts accordingly. One of the tasks of the passive thread is to update the secret state of the node when a Type 3 message arrives and the conditions for updating are satisfied. Note that the updating frequency of the secret state is only determined by incoming messages and that it has no influence on the sending rate.

The algorithm at a node j takes as input the local structure of the communication graph (the overlay network), that is, $OUT^{[j]}$, $IN^{[i]}$ and w_{ji} for all nodes $i \in OUT^{[j]}$, where w_{ji} is the link weight. In addition, node j has an initial secret value x_j, and a period Δ that determines the frequency of the execution of the loop in the active thread. Note that all the nodes could in principle select a different Δ or they could execute their active thread even

Algorithm 1 async. privacy preserving power method, active thread

1. **for-each** node $\lambda \in IN^{[j]}$ *#initialize incoming type 3 msgs*
2. $V^{[\lambda j]} \leftarrow (0, \{\}, \{\}, \text{timestamp})$
3. choose random $s\_timeout^{[ji]}$ and a randomly sized set of nodes $OC^{[ji]} \in IN^{[i]}$
4. **for-each** node $\alpha \in OC^{[ji]}$ *#send shares*
5. send type 1 msg $(s_{t=0}^{[j\alpha i]})$ to α
6. **while**(true)
7. $online\_nodes^{[j]} \leftarrow \{\}$ *#it is updated in the passive thread*
8. wait(Δ) *#determines message sending frequency*
9. **if** $i \in online\_nodes^{[j]}$ **and** $s\_timeout^{[ji]} \leq 0$ *#renew one share*
10. **if** $OC^{[ji]} \cap online\_nodes^{[j]} = \{\}$ **and** $IN^{[i]} \cap online\_nodes^{[j]} \neq \{\}$
11. add a node α to $OC^{[ji]}$ from $IN^{[i]} \cap online\_nodes^{[j]}$
12. **if** $OC^{[ji]} \cap online\_nodes^{[j]} \neq \{\}$
13. choose one node $\alpha \in OC^{[ji]} \cap online\_nodes^{[j]}$
14. generate new share $s_{t \leftarrow t+1}^{[j\alpha i]}$
15. send new type 1 msg $(s_t^{[j\alpha i]})$ to α *# t was increased by 1*
16. choose new random $s\_timeout^{[ji]}$
17. **for-each** share $s_t^{[j\alpha i]}$ *#check shares and versions in checklist*
18. **if** $(j, \alpha, t) \notin CH^{[ji]}$
19. resend $s_t^{[j\alpha i]}$ to α *#resend the share if needed*
20. **for-each** node $l \in IN^{[j]}$ *#send type 2 messages*
21. compose $CH^{[lj]}$ according to its definition
22. send type 2 message: $(CH^{[lj]}, online\_nodes^{[j]}, \text{timestamp})$ to node l
23. $LOC^{[ji]} \leftarrow \{\}$ *#send type 3 messages*
24. $M^{[ji]} \leftarrow w_{ji} x_j$
25. **for-each** $(j, \alpha, t) \in CH^{[ji]}$ *#share version t from checklist*$^{[ji]}$
26. $M^{[ji]} \leftarrow M^{[ji]} - s_t^{[j\alpha i]}$
27. store (j, α, t) in $LOC^{[ji]}$
28. **for-each** $(\beta, j, t) \in received\_shares^{[ji]}$ *#shares received from in-collaborators*
29. **if** $\beta \in online\_nodes^{[j]}$ *#decide which version of the share to use*
30. $M^{[ji]} \leftarrow M^{[ji]} + s_t^{[\beta ji]}$ with t the newest version received
31. store (β, j, t) in $LIC^{[ji]}$
32. **else**
33. $M^{[ji]} \leftarrow M^{[ji]} + s_{t'}^{[\beta ji]}$ with t' obtained from $CH^{[ji]}$
34. store (β, j, t') in $LIC^{[ji]}$
35. **if** $LOC^{[ji]} = \{\}$ **and** $LIC^{[ji]} = \{\}$ *#happens only during bootstrap*
36. send type 3 message $V^{[ji]}$: $(0, LOC^{[ji]}, LIC^{[ji]}, \text{timestamp})$ to node i
37. **else**
38. send type 3 message $V^{[ji]}$: $(M^{[ji]}, LOC^{[ji]}, LIC^{[ji]}, \text{timestamp})$ to node i

irregularly, since the asynchronous iteration tolerates this. For simplicity we assume that all the nodes use the same period Δ.

Algorithm 1 shows the active thread of the protocol at node $j \in IN^{[i]}$ that is responsible for sending information to node i. An instance of this thread is run at node j for all $i \in OUT^{[j]}$, that is, for all the nodes i, for which $j \in IN^{[i]}$.

The algorithm first initializes several variables: a copy of the last received Type 3 message from each in-neighbor (line 1), the set $OC^{[ji]}$ and a timeout for updating the shares of the members of the set (line 3). The first set of shares is sent in line 5.

Algorithm 2 async. privacy preserving power method, passive thread

```
1.  while(true)
2.    msg ← receive_message()
3.    if type 1: s_t^[βji], β ∈ IN^[i]                          # s_t^[βji] received
4.      store s_t^[βji] in received_shares^[ji] (replace s_{t-1}^[βji] if exists)
5.      online_nodes^[j] ← online_nodes^[j] ∪ β
6.    else if type 2: CH^[ji], online_nodes^[i]                 # checklist received from i
7.      store CH^[ji] (replace older version if exists)
8.      online_nodes^[j] ← online_nodes^[j] ∪ online_nodes^[i] ∪ i
9.    else if type 3: V^[λj], λ ∈ IN^[j]                        #V msg received
10.     store V^[λj] (replace older version if exists)
11.     online_nodes^[j] ← online_nodes^[j] ∪ λ
12.     if ∪_{λ∈IN^[j]} LOC^[λj] = ∪_{λ∈IN^[j]} LIC^[λj] then   #check Eq. (10)
13.       x_j ← Σ_{λ∈IN^[j]} M^[λj]                             #update internal state
14.       s_timeout^[ji] ← s_timeout^[ji] − 1
```

The main loop runs with a period of Δ time units that defines the frequency of sending messages. First, we clear the set of nodes that j estimates to be online. This set is filled in the passive thread during the waiting period (see Algorithm 2). Then, if the share renewal timeout expires, we first identify those out-collaborators that are probably online, we try to add new collaborators if the old ones seem to be offline (line 11), and then we send the new shares (line 15). Note that sending new shares to collaborators that are offline is not a problem, and via adding new collaborators we speed up the convergence. From line 17 to 19, the freshest available checklist is used to discover which shares $s^{[j\alpha i]}$ have not been installed successfully, and these are re-sent. Node j then creates new Type 2 messages for its own in-neighbors (line 20).

Finally, a Type 3 message is created and sent to node i in line 38. For those shares sent by j (lines 25 and 27), node j uses the version in the checklist received from i. This way j makes sure it is using the last share that reached i. The case of shares received by j is more interesting. The sender β wants proof that node j has started using the new version before switching to it itself. However, if β is offline, it can get the proof only when it comes back online, which may in fact never happen. So node j is cautious and switches to the new version only if there is a good chance that β will detect it, and will switch too. This is only a heuristic, as with a small probability it might happen that β is incorrectly considered to be online. However, this is not a problem, because node j will switch back to the working version included in the checklist in the next round, should β stay offline.

The passive thread in Algorithm 2 handles message arrivals at node j, updates its private state when possible and fills the set $online\_nodes^{[j]}$. If the incoming message is of Type 1 (line 3) then the received share is stored (line 4). For a Type 2 message (line 6) the checklist is stored and the set $online\_nodes^{[i]}$ is added to $online\_nodes^{[j]}$ (line 8). Incoming Type 3 messages are stored and if the condition in (10) holds (line 12) then the internal state x_j is updated

and the corresponding share timeout is decreased. Note that it is not necessary to update shares if the private state is not changing, so in that case we do not decrement the timer. The sender of any message is added to $online\_nodes^{[j]}$ (lines 5, 8 and 11).

5.2 Message Complexity

Let us now discuss the message length overhead added by our solution in relation to SSS in order to tolerate churn, message loss and delay. SSS uses two types of messages. The first one conveys a share for an out-collaborator, but must also indicate the IDs of the sender and central nodes, as well as some time information in order to discard old messages (we assume the latter since it is necessary for the correct operation of the algorithm, though it is not mentioned in the article). The second message carries a sum of shares, as well as the sender and central node IDs, and time information. Assuming the use of integer data types to represent node IDs and timestamps (the round number, for example) and double data types to store shares and share sums, each message is at least $3 \cdot 32 + 64 = 160$ bits long. In one round, for a given node, there are n^2 messages of the first type, and n messages of the second type.

In our scheme Type 1 messages are similar to the share messages of SSS: they include two node IDs, time information and the share itself therefore they are 160 bits long. Type 2 messages contain two different lists: $CH_t^{[ji]}$ of at most $2 \cdot k \cdot 3 \cdot 32 = 192k$ bits (assuming an average k for every node), and $online\_nodes_t^{[i]}$ with at most $32n$ bits. Hence Type 2 messages are at most $192k + 32n$ bits long. Finally, Type 3 messages include a share (64 bits) and the $LIC^{[ji]}$ and $LOC^{[ji]}$ lists with $3 \cdot 32 \cdot k$ bits each. This gives a total of $192k + 64$ bits. In one round, for a given node, there are nk Type 1 messages, n Type 2 messages, and n Type 3 messages.

It is easy to see that our messages are larger than those of SSS; this is the price we pay for an increased robustness. At the same time, $k \leq n$ is a parameter that can be freely selected, therefore k^2 can be much smaller than n^2.

5.3 Notes on Asynchrony

In our scheme, each node i can update its private state upon the receipt of a single Type 3 message from any neighbor, provided that the condition in (10) holds. This permits a chaotic asynchronous communication model in contrast to the model used in SSS. Hence our scheme can cope with high rates of message loss, message delays and node churn.

In the synchronous SSS model a single iteration may progress if and only if at least k messages arrive at node i within the iteration (which need to be

preceeded by many more successful message transmissions among the neighbors of i during the same iteration, as explained earlier). For this reason, the method is practically unable to progress in the presence of significant message loss and delay, let alone node churn.

5.4 Node Churn

In the churn model that we consider every node can leave and then rejoin using its previous state and neighbors. In other words, we assume that the output of the computation (the eigenvector in our case) is assumed not to change. This makes performance evaluation more informative, since at any point in time we can compare the actual solution to a known correct output.

Let us explain some of the advantages of our method over SSS in the presence of churn. On one hand, SSS requires each neighborhood to agree on the parameter k which determines both the degree of the polynomial used in the secret sharing scheme and the minimum number of out-collaborators per node. If, due to churn, the number of online nodes in a neighborhood falls below k, then the remaining nodes must renegotiate a new k. This issue is not discussed in [5] and it is not straightforward. However, if we picked a static value of k for the whole network, neighborhoods with less than k nodes would not be able to progress at all.

On the other hand, our scheme can adapt to churn. First, a node chooses its collaborator set (its members and its size) freely. Second, there is no lower bound on the number of collaborators that are needed to be online for a node to progress. For example, node j can still send valid Type 3 messages to i even if all its out-collaborators are offline. There is only one case, however, in which node i will not be able to update its state, namely when share version inconsistencies occur at node i and all the nodes that could resolve it are offline. This may happen only if the two nodes involved in a version update go offline *at the same time* after having used a different share version in their last Type 3 messages.

In addition, our scheme allows each individual node to increase its individual degree of privacy by choosing new out-collaborators when it is time to update a share but no out-collaborator is found online.

6 Privacy

As mentioned previously, we assume the semi-honest adversary model where the adversary knows the private state and the incoming messages of corrupted nodes, but eavesdropping on arbitrary links is not allowed.

As mentioned earlier, the SSS algorithm tolerates the collusion of up to $k-1$ compromised nodes, but any subset of $IN^{[i]}$ of size at least k compromised nodes can recover the secret values of *all members* of $IN^{[i]}$ [5]. From the point of view of privacy, our algorithm has two main differences compared to SSS:

it is based on the random sum secret sharing scheme, and it allows nodes to reuse their shared values so as to save bandwidth. We examine the effect of both of these differences.

First, we note that due to the random sum scheme we do not only gain flexibility, but we also markedly increase the level of security against collusion. To see this, let us assume that each node $j \in IN^{[i]}$ has a set $OC^{[ji]}$ of size k, with $k < |IN^{[i]}|$. Let us also assume that we implement the scheme in a synchronous fashion, and new shares are generated in each round. Now, we ask how many nodes we need to compromise to learn the private value of node j. Obviously, we assume we cannot compromise j directly. One observation is that we have to compromise at least one node $i \in OUT^{[j]}$, since the nodes that are not out-neighbors of j never receive anything that is related to the private value of j; they only receive random numbers from j. This extra constraint already improves security. Let us assume that we have a compromised node $i \in OUT^{[j]}$. In this case i does not receive a share from j, but the value $M^{[ji]} = w_{ji}x_j - \sum_{\alpha \in OC^{[j]}} s^{[j\alpha i]} + \sum_{\beta \in IC^{[j]}} s^{[\beta j i]}$. Clearly, we need to know $\sum_{\alpha \in OC^{[j]}} s^{[j\alpha i]} + \sum_{\beta \in IC^{[j]}} s_j^{[\beta j i]}$ in order to recover $w_{ji}x_j$. Given that $|OC^{[ji]}| = k$, it is clear that $|OC^{[ji]}| + |IC^{[ji]}| \geq k$. Thus, the collusion of at least k nodes, *and* node i, is needed to recover the value of j. But note that to recover the private values of nodes other than j ($\sum_{\beta \in IC^{[j]}} s^{[\beta j i]}$), *another set* of colluding nodes might be needed, as opposed to SSS. In sum, we have proven that the random sum scheme offers a strictly higher theoretical degree of privacy than SSS.

Now let us turn to the second difference; namely that shares are reused to save bandwidth. This weakens the scheme because with a small probability the expression $M^{[ji]} = w_{ji}x_j - \sum_{\alpha \in OC^{[j]}} s^{[j\alpha i]} + \sum_{\beta \in IC^{[j]}} s^{[\beta j i]}$ could be a constant apart from the secret value $w_{ji}x_j$. This means that node i can calculate the variation of the secret value. However, it is a highly non-trivial question to use the variation to guess what the actual value is. The reason is that in the initial phase of the computation $w_{ji}x_j$ is dominated by x_j that can be selected at random in the initialization phase, and after the solution has converged the variation becomes very small and provides little guidance. In addition, even if we can guess $w_{ji}x_j$, it is still difficult to guess x_j and w_{ji} separately.

Finally, it is interesting to remark that the fact that nodes can choose different k values allows them to adjust the privacy level on a node-by-node basis. Moreover, nodes can even change k in real time should they wish to modify the initial privacy settings.

7 Experimental results

In our simulation experiments, we used the event-based PeerSim simulator [23]. We implemented the SSS algorithm and our random sum based proposal. Parameter Δ is the cycle length for both schemes, see Algorithm 1.

Recall that in this paper we focus on the power method, in particular, its asynchronous version [21]. The power method is an iterative algorithm for finding the dominant eigenvector of a matrix. In our distributed setting, each

node calculates a single element of the dominant eigenvector, and network links correspond to non-zero elements of the matrix. For the sake of comparison, each node's private value was initialized to 1.0, although in a real environment a random initial value should be used where possible to mask variations in the private value.

The method described in [21] works only with non-negative, irreducible matrices with a spectral radius of one. We used artificially generated sparse matrices derived from the adjacency matrix of two directed graphs presented in [15], each with 5000 nodes. The matrices are called "random k-out normalized" (Rnd) and "small gap normalized" (SmlG).

In Rnd, 8 random out-links are added to each node. SmlG was generated starting with an undirected ring, and adding two random out-links from all the nodes. The result is a matrix with a small gap between its two largest eigenvalues: this characteristic makes the power method converge slowly. Both matrices were normalized so the sum of each column is 1, which ensures that the spectral radius is one.

To provide a fair comparison, the privacy parameter k was given a constant small value of 3. This corresponds to tolerating a relatively large number of messages lost for SSS, at the expense of a lower level of privacy. Analogously, the size of the set of out-collaborators in our scheme was randomly chosen from $[1, \frac{|IN^{[i]}|}{2}]$, which is a close setting to that of SSS. Recall from Section 5 that our scheme looks for new out-collaborators at the time of share renewal if no collaborators are thought to be online. The $s\_timeout$ parameter is randomly drawn from $[150\Delta, 300\Delta]$.

The aim of our experiments was to show that our method can cope with unreliable and heterogeneous links. The message loss rates used were $Drop$=0 (no loss) and $Drop$=0.1 (10% of the messages are lost). The message delays were 0 (ideal conditions, immediate arrival), or a uniformly distributed random value from [0, 0.1Δ], or from [0, Δ].

Three churn scenarios were considered, which are based on measurements presented in [28]. Let Weibull(a, b) denote the Weibull distribution with a denoting the shape parameter and b the scale parameter. The first scenario involves no churn so that we can compare the best performance of both schemes. Second, a scenario with online session lengths drawn from Weibull$(0.4, 20\Delta)$ and offline session lengths drawn from Weibull$(0.4, 40\Delta)$ was applied. We call this scenario *fast churn*. Third, a scenario characterized by a slower churn was used that we call *slow churn*: here online session lengths follow Weibull$(0.4, 40\Delta)$, while offline lengths are drawn from Weibull$(0.4, 80\Delta)$. Note that simulating a slower churn with the same Δ is equivalent to assuming the same fast churn but with a smaller cycle length Δ.

The metric used to test convergence was the following. The correct dominant eigenvector $\mathbf{w}$ was pre-calculated. During simulations, we tested the distance between the current solution $\mathbf{x}$ in the network and $\mathbf{w}$ by calculating the angle between the two vectors given by $\arccos \frac{||\mathbf{w}^T \cdot \mathbf{x}||}{||\mathbf{w}||\ ||\mathbf{x}||}$. With convergence,

		SSS		Ours					
		no churn		no churn		fast churn		slow churn	
Drop	Delay	mean	std	mean	std	mean	std	mean	std
0	0	145	0	52	0	3,438	209	4,678	731
0	$\in [0, 0.1\Delta]$	144	0	54	0	3,393	347	5,565	1,079
0	$\in [0, \Delta]$	745	0	117	0	6,776	258	10,031	369
0.1	0	225	32	80	0	4,772	397	7,604	1,278
0.1	$\in [0, 0.1\Delta]$	248	32	90	0	5,125	232	8,234	1,367
0.1	$\in [0, \Delta]$	9,774	186	169	0	7,068	424	12,621	1,199

Table 1: Average number of messages sent per node on Rnd. The mean and the standard deviation are shown for three independent runs. In the presence of churn we could not run simulations long enough to reach convergence with SSS.

		SSS		Ours					
		no churn		no churn		fast churn		slow churn	
Drop	Delay	mean	std	mean	std	mean	std	mean	std
0	0	1,273	0	139	0	56,272	3,215	165,930	45,496
0	$\in [0, 0.1\Delta]$	1,279	10	155	2	63,027	1,529	161,080	25,684
0	$\in [0, \Delta]$	14,079	1,305	303	6	68,345	4,776	158,360	23,254
0.1	0	2,548	35	175	2	63,369	11,877	116,800	6,329
0.1	$\in [0, 0.1\Delta]$	2,609	82	191	0	63,010	2,308	140,560	20,666
0.1	$\in [0, \Delta]$	67,538	3,883	346	7	70,144	8,101	133,420	19,621

Table 2: Average number of messages sent per node on SmlG. The mean and the standard deviation are shown for three independent runs. In the presence of churn we could not run simulations long enough to reach convergence with SSS.

this angle tends to zero. Our stop condition for convergence was that the angle be less than ϵ. For Rnd, $\epsilon = 0.05$ and for SmlG, $\epsilon = 0.1$.

Our main measure of performance is the speed of convergence measured in terms of the average number of messages sent per node until convergence. The reason we applied this metric was the necessity of finding a fair comparison between synchronous and asynchronous schemes, given that the concept of *iteration* makes little sense in the asynchronous approach. This measure also takes different message complexities into account in a natural way. The total number of messages sent throughout the network can be easily found by multiplying the average number of messages per node by the network size.

Tables 1 and 2 show the average number of messages sent per node under different scenarios over the two matrices. Statistics are shown for three independent runs, rounded to an integer. Figures 2 to 5 depict the convergence trend for the same cases. The figures show a single run for each scenario.

In the case of the scenarios involving no churn (Figures 2 and 3), it is clear from the results that our scheme outperforms SSS in the case of both matrices. The first reason for this is the lower message complexity that allows our method to converge faster even in the failure-free scenario. The second

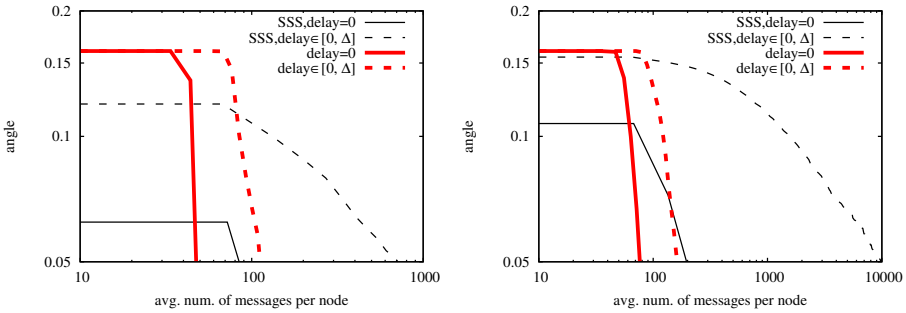

Fig. 2: Matrix Rnd with no churn, and message drop probability 0 (left) and 0.1 (right).

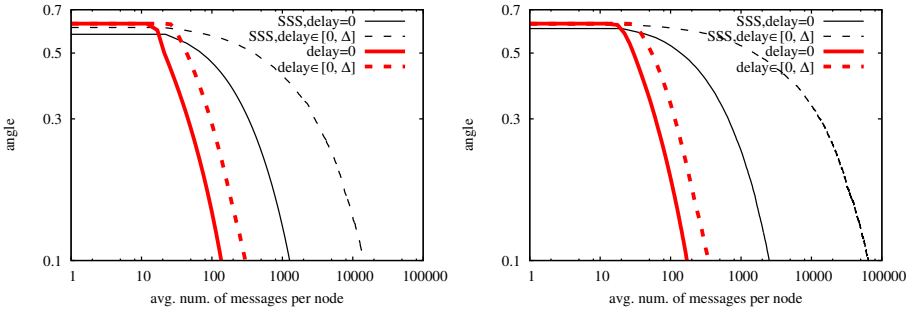

Fig. 3: Matrix SmlG with no churn, and message drop probability 0 (left) and 0.1 (right).

reason is the increased fault tolerance, that is most apparent in the long delay scenario. One interesting consequence of this result is that with our method the iteration period Δ can be reduced to very small values, still resulting in a proportional speedup, while the SSS scheme will suffer from the increasing relative message delay that eventually makes convergence impossible.

In the churn scenarios SSS is not able to make observable progress, so the corresponding results are omitted. Our algorithm in contrast tolerates all scenarios. We observe a proportional slowdown in convergence as nodes spend more and more time offline, relative to the period Δ. This effect is stronger in the case of the SmlG matrix. One interesting conclusion is that in the case of slower churn it does not make sense to maintain the original iteration period Δ: one can increase Δ without slowing down convergence, while saving on communication. This is because increasing Δ is equivalent to speeding up the churn. An adaptive mechanism to optimize Δ would be an interesting direction for future research.

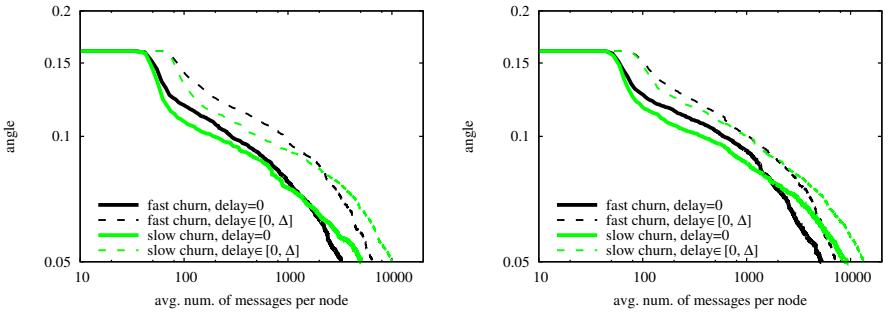

Fig. 4: Matrix Rnd with churn, and message drop probability 0 (left) and 0.1 (right).

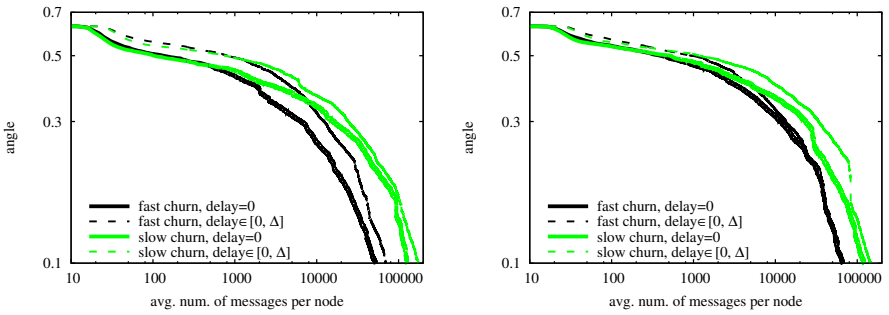

Fig. 5: Matrix SmlG with churn, and message drop probability 0 (left) and 0.1 (right).

In addition, note that this slowdown is in fact a side-effect of our setup, namely that we measure convergence as the distance from a global correct solution. In a different setup we could assume that the set of nodes is changing, and that we follow a moving target, that is, where the correct solution would depend on the set of nodes that are currently online.

8 Conclusions

Here we introduced an asynchronous privacy preserving communications layer for distributed iterative algorithms, focusing on its application in peer-to-peer networks, and other unreliable large scale networks under realistic conditions. We developed our approach for the case of power iteration. To the best of our knowledge, it is the first analysis of an asynchronous privacy preserving approach for this problem. Our algorithm was compared experimentally with a recent synchronous solution called SSS, and the advantages of using an

asynchronous approach were demonstrated: experiments included the failure-free scenario as well as scenarios with message loss, message delay and node churn.

Apart from asynchrony, our random sum secret sharing scheme was also shown to be clearly preferable to SSS in terms of flexibility and the guaranteed level of privacy, even in a fully reliable environment.

We would like to add that, though we focused on power iteration, the layer can be applied along with any iterative method that is based on calculating local sums of neighbor values, and also has an asynchronous implementation.

Acknowledgements J.A.M. Naranjo and L.G. Casado were supported by the Spanish Ministry of Science and Innovation (TIN2008-01117) and Junta de Andalucía (P11-TIC-7176), and partially by the European Regional Development Fund (ERDF). M. Jelasity was supported by the Bolyai Scholarship of the Hungarian Academy of Sciences and by the FET programme FP7-COSI-ICT of the European Commission through project QLectives (grant no.: 231200).

References

1. Aggarwal, C.C., Yu, P.S.: A general survey of privacy-preserving data mining models and algorithms. In: C.C. Aggarwal, P.S. Yu, A.K. Elmagarmid (eds.) Privacy-Preserving Data Mining, *The Kluwer International Series on Advances in Database Systems*, vol. 34, pp. 11–52. Springer US (2008). DOI 10.1007/978-0-387-70992-5_2
2. Agrawal, R., Srikant, R.: Privacy-preserving data mining. SIGMOD Rec. **29**(2), 439–450 (2000). DOI 10.1145/335191.335438
3. Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H. (eds.): Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide. SIAM, Philadelphia (2000)
4. Bianchini, M., Gori, M., Scarselli, F.: Inside pagerank. ACM Transactions on Internet Technology **5**(1), 92–128 (2005). DOI 10.1145/1052934.1052938
5. Bickson, D., Dolev, D., Bezman, G., Pinkas, B.: Peer-to-Peer secure multi-party numerical computation. In: IEEE International Conference on Peer-to-Peer Computing, pp. 257–266. IEEE Computer Society (2008). DOI 10.1109/P2P.2008.22
6. Bickson, D., Malkhi, D.: A unifying framework of rating users and data items in peer-to-peer and social networks. Peer-to-Peer Networking and Applications **1**(2), 93–103 (2008). DOI 10.1007/s12083-008-0008-4
7. Bickson, D., Reinman, T., Dolev, D., Pinkas, B.: Peer-to-peer secure multi-party numerical computation facing malicious adversaries. Peer-to-Peer Networking and Applications **3**(2), 129–144 (2010). DOI 10.1007/s12083-009-0051-9
8. Clifton, C., Kantarcioglu, M., Vaidya, J., Lin, X., Zhu, M.Y.: Tools for privacy preserving distributed data mining. SIGKDD Explor. Newsl. **4**(2), 28–34 (2002). DOI 10.1145/772862.772867
9. Das, K., Bhaduri, K., Kargupta, H.: Multi-objective optimization based privacy preserving distributed data mining in peer-to-peer networks. Peer-to-Peer Networking and Applications **4**(2), 192–209 (2011). DOI 10.1007/s12083-010-0075-1
10. Datta, S., Bhaduri, K., Giannella, C., Wolff, R., Kargupta, H.: Distributed data mining in peer-to-peer networks. IEEE Internet Computing **10**(4), 18–26 (2006). DOI 10.1109/MIC.2006.74
11. Frommer, A., Szyld, D.B.: On asynchronous iterations. Journal of Computational and Applied Mathematics **123**(1-2), 201–216 (2000). DOI 10.1016/S0377-0427(00)00409-X
12. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing, STOC '87, pp. 218–229. ACM, New York, NY, USA (1987). DOI 10.1145/28395.28420

13. Golub, G.H., Van Loan, C.F.: Matrix Computations, third edn. The Johns Hopkins University Press (1996)
14. He, W., Liu, X., Nguyen, H.V., Nahrstedt, K., Abdelzaher, T.: PDA: Privacy-preserving data aggregation for information collection. ACM Trans. Sen. Netw. **8**(1), 6:1–6:22 (2011). DOI 10.1145/1993042.1993048
15. 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
16. 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
17. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The eigentrust algorithm for reputation management in p2p networks. In: Proceedings of the 12th international conference on World Wide Web (WWW'03), pp. 640–651. ACM Press, New York, NY, USA (2003). DOI 10.1145/775152.775242
18. 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
19. Kempe, D., McSherry, F.: A decentralized algorithm for spectral analysis. In: Proceedings of the 36th ACM Symposium on Theory of Computing (STOC'04), pp. 561–568. ACM, New York, NY, USA (2004). DOI 10.1145/1007352.1007438
20. Lindell, Y., Pinkas, B.: Privacy preserving data mining. Journal of Cryptology **15**(3), 177–206 (2002). DOI 10.1007/s00145-001-0019-2
21. Lubachevsky, B., Mitra, D.: A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit radius. Journal of the ACM **33**(1), 130–150 (1986). DOI 10.1145/4904.4801
22. Maurer, U.: Secure multi-party computation made simple. Discrete Applied Mathematics **154**(2), 370–381 (2006). DOI 10.1016/j.dam.2005.03.020
23. Montresor, A., Jelasity, M.: Peersim: A scalable P2P simulator. In: Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing (P2P 2009), pp. 99–100. IEEE, Seattle, Washington, USA (2009). DOI 10.1109/P2P.2009.5284506. Extended abstract
24. Mosk-Aoyama, D., Shah, D.: Fast distributed algorithms for computing separable functions. IEEE Transactions on Information Theory **54**(7), 2997–3007 (2008). DOI 10.1109/TIT.2008.924648
25. Parreira, J.X., Donato, D., Michel, S., Weikum, G.: Efficient and decentralized PageRank approximation in a peer-to-peer web search network. In: Proceedings of the 32nd international conference on Very large data bases (VLDB'2006), pp. 415–426. VLDB Endowment (2006)
26. 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
27. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979). DOI 10.1145/359168.359176
28. Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC'06), pp. 189–202. ACM, New York, NY, USA (2006). DOI 10.1145/1177080.1177105
29. Yao, A.C.: Protocols for secure computations. In: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 160–164 (1982). DOI 10.1109/SFCS.1982.38
30. Yao, A.C.C.: How to generate and exchange secrets. In: Proc. 27th Annual Symposium on Foundations of Computer Science (FOCS), pp. 162–167 (1986). DOI 10.1109/SFCS.1986.25