Collaborative Mobile Gossip Learning

Árpád Berta

Supervisor

Dr. Márk Jelasity

Department of Computer Algorithms and Artificial Intelligence

Doctoral School of Computer Science

University of Szeged

A thesis submitted for the degree of

Doctor of Philosophy

Szeged

2020

Acknowledgements

First of all I would like to express my gratitude to my supervisor, Dr. Márk Jelasity, for supporting my research and being a great source of inspiration to me over the past eight years. He showed me how to think scientifically and acquire new knowledge. His constructive and thoughtful comments have always been of great value to me.

Next, I would like to thank to my colleagues who helped me to discover interesting areas of science and helped give birth to new ideas during our discussions. In alphabetic order: Dr. Vilmos Bilicki, Gábor Danner, Dr. István Hegedűs, Dr. Róbert Ormándi and Zoltán Szabó. I would like to thank to Júlia Bustya and David P. Curley for correcting this thesis from a linguistic point of view.

Of course, I would like to thank my wife, Renáta for all her support. She has been by my side all the time and she has always been a great motivating force. I am also grateful to my daughter Boróka for all the love and joy I have received from her, and all the patience she showed towards me during the preparation of this thesis. Last but not least, I would like to thank my parents for supporting me throughout my years of education.

This research work 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”), supported by grant 20391-3/2018/FEKUSTRAT of the Hungarian Ministry of Human Capacities, supported by grant TUDFO/47138-1/2019-ITM of the Hungarian Ministry for Innovation and Technology and supported by the European Union

and the European Social Fund under the grant number TAMOP-4.2.2.C-11/1/KONV-2012-0013 (“FuturICT.hu”). I am very grateful for this support, which definitely acted as a spur for the submission of this thesis.

Contents

List of Algorithms

List of Tables

List of Figures

CHAPTER 1

Introduction

Over the past few decades, we have witnessed an explosive growth of mobile and smart devices and their widespread use. These devices are present in almost every aspect of our daily lives. This trend has led to numerous intelligent applications based on data mining [109]. It is usually performed over collected data at a central location. This conventional process has become evermore problematic due to the increasing public awareness of the privacy issue. In the last few years stricter privacy protection laws have come into force [1]. For this reason, there is an increasing interest in methods that allow us to keep our private data in our devices and process them using collaborative algorithms.

There are, of course, many ways to address this challenge. We opted for gossip learning [82] due to the fact that it is fully decentralized, hence no central server is needed. Nodes exchange and aggregate models directly. This makes scalability significantly cheaper than the alternatives. It is a good opportunity for startups or communities with low budgets to provide robust intelligent smartphone services. It can serve the common good (e.g. public healthcare and public education). Although we focus on collaborative mobile platforms [85], gossip learning applications can be found in smart metering [89] and over Internet of Things platforms [106] as well.

Google introduced centralized federated learning to meet this challenge [56, 76]. This method performs data mining similar to the well-known parameter server archi-

tecture [25], but with the difference that here the data remains on device. The server maintains the current model, aggregates the received models and regularly distributes it to the nodes. There, an update step is performed on local data and propagated back to the parameter server. Therefore the method is optimized to minimize these communication costs. They handled this problem with some novel compression techniques. However, gossip learning is not only comparable to federated learning in terms of performance, but it can even outperform it in certain cases [44, 45].

In gossip learning, privacy is not guaranteed by default, but it is much easier to achieve. Moreover, theoretical notions of privacy such as differential privacy can be included as well. From this point of view, we propose a random walk service that can maintain a single walk and drive down the privacy budget. In addition to this, we present a secure sum protocol to prevent the collusion attack. Our long-term goal is to provide a fully open collaborative environment where those who provide data can enjoy the benefits of mining the collective data of the community. To realize this goal, we propose a multiple random walk service for maintaining independent decentralized tasks that might belong to different users. We also introduce a smartphone trace based on collected data in order to create more realistic simulations. It contains network properties and patterns of user behavior. We can also take into account the improving performance of the gossip learning. Hence, it is crucial to minimize the communication cost by reducing the dimensionality of the data mining tasks. For this reason, we propose a number of robust and efficient approaches to handle this.

As we can see, the main aim of this thesis is to offer several fully distributed protocols that make gossip learning more suitable for performing collaborative data mining. We provide methods with very diverse aspects and they can address the most interesting open questions of the gossip learning.

The thesis is organized as follows. First, we give a brief overview of the background in Chapter 2. It includes an introduction to stochastic gradient descent (SGD) machine learning optimization method, an outline of the applied system model and the data distribution; and we introduce the basic idea of the Gossip Learning Framework. Next, in chapters 3 – 6, we present our main contributions. We introduce our proposal for data collection over smartphone networks and a real trace of smartphone user behavior that can provide realistic network simulations. Afterwards, we present several fully distributed methods for the dimension reduction task. Then we propose solo and multiple random walk man-

agement algorithms. Lastly, we describe a tree-based mini-batch gradient descent method for privacy preservation.

We present a smartphone trace in Chapter 3 for simulating real user behavior. It helps us to make more realistic simulations for evaluating our proposed distributed protocols which are described this thesis. We present our locally developed Android app Stunner that collects information about the users such as NAT (network address translation) type, the availability of WiFi and cellular networks, the battery level, and many other attributes. Based on this data we identify and model the sessions during which a user can participate in, with a fully distributed protocol. We also demonstrate through the simulation of gossip protocols that it is feasible to develop smartphone-friendly applications. For many years we have been collecting data via smartphones, and we enhanced our app by taking actual P2P measurements. We outline our updated data collection method and the technical details, including some challenges we faced with data cleansing. We present a set of statistics based on the collected data.

In Chapter 4, we outline a number of robust and efficient decentralized approaches to dimension reduction. The first algorithm that we describe builds on searching for good random projections. We conclude that this method is preferable and provides good quality results when the output is required on a very short timescale, within tens of minutes. We propose a fully distributed variant of singular value decomposition (SVD) for dimension reduction. We also present a hybrid method that combines the advantages of random projections and SVD. We present a detailed experimental comparison of the proposed algorithms and compare them with each other. We demonstrate that the hybrid method provides a good performance over all time scales.

We present an approach in Chapter 5 for implementing a single random walk service. Our solution is based on a cheap gossip layer to broadcast a small global state and a replication mechanism based on this state. It meets our three major requirements. First, the random walk should be agile: it should progress as quickly as possible. Second, the implementation should be efficient: for example, it should induce only a minimal extra cost to achieve robustness. Third, the random walk should be long-lived, that is, it should perform as many steps as possible without resetting its state. Such a single random walk protocol can provide differentially private implementation of fully distributed SGD. In Chapter 5, we also propose a protocol to manage $O(n)$ random walks in a network of n nodes. Although our motivation is gossip learning, this protocol may be viewed as a

Table 1.1. The relationship between the chapters and the corresponding publications (where • and ◦ indicate the core and the related publications, respectively).

	Chapter 3	Chapter 4	Chapter 5	Chapter 6
P2P 2014 [9]	•	◦	◦	◦
TIST 2016 [43]	•	•	◦	◦
ICCGI 2017 [101]	•			
DAIS 2019 [103]	•			
PDP 2016 [5]	◦	•		
JOWUA 2016 [42]	◦		•	
PDP 2017 [6]	◦		•	
SCN 2018 [22]	◦			•
ESANN 2014 [10]		◦		
IJASO 2018 [102]	◦			

general middleware service for the management of multiple walks over networks. A key element of this protocol is a multi-level restarting mechanism designed to prevent the failure of random walks due to node churn, while respecting a set of bandwidth constraints. We demonstrate that the random walks are kept alive and are run at close to optimal speed under the given bandwidth constraints.

In Chapter 6, we propose a light-weight protocol to quickly and securely compute the sum query over a subset of participants, assuming a semi-honest adversary. During the computation the participants learn no individual values. We apply this protocol to efficiently calculate the sum of gradients as part of a fully distributed mini-batch stochastic gradient descent algorithm. The protocol achieves scalability and robustness by exploiting the fact that in this application domain a "quick and dirty" sum computation is acceptable. We utilize the Paillier homomorphic cryptosystem as part of our solution combined with extreme lossy gradient compression to make the cost of the cryptographic algorithms affordable. We demonstrate both theoretically and experimentally that the protocol is indeed practically viable.

Then, in Chapter 7 we summarize our research contributions. Finally, in Chapter 8 an overview of this thesis is given in Hungarian as well. And in Table 1.1 above, we list the relationship among the relevant publications and the thesis chapters.

CHAPTER 2

Background

We give an overview of this chapter, which summarizes the necessary background that plays a crucial role in understanding our results. First, we briefly introduce the Stochastic Gradient Descent optimization method for solving classification problem. Then later, we describe our fully distributed system model and data distribution. After, we turn to discuss the basics of the Gossip Learning Framework.

2.1 Stochastic Gradient Descent

Classification is an important problem in machine learning. Given a data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of n observations, where an object or an example is represented by a pair of a feature vector $x \in R^d$ and the corresponding class label $y \in C$, where d is the dimension of the problem and C is the domain of class labels. In the case of binary classification the number of possible class labels is two (e.g. $C = \{0, 1\}$). The problem of classification is often expressed as finding the parameters w of a function $f_w : R^d \to C$ that can correctly classify as many examples in D as possible, as well as outside D (this latter property is called generalization). In other words, we are looking for a parameter vector to optimize

the objective function of the problem

$$w = \arg\min_w J(w) = \arg\min_w \frac{1}{n}\sum_{i=1}^{n} \ell(f_w(x_i), y_i) + \frac{\lambda}{2}\|w\|^2, \tag{2.1}$$

where the $\ell()$ is a loss function and $(\lambda/2)\|w\|^2$ is the regularization term with parameter λ. Function f_w is called the model of the data set. The regularization term helps the model to avoid overfitting the data set, thus aiding generalization. The labeled data set is often split into two non-overlapping subsets; namely a training set for optimizing the parameters w of the model and a test set for measuring the generalization performance of the optimized model.

Gradient descent (GD) is an iterative method that can find the optimum of a convex function. It is often used for optimizing the above objective function. The parameter vector w is iteratively updated using the derivative of the objective function that is computed on the whole training set

$$\begin{aligned} w_{t+1} &= w_t - \eta_t \nabla J(w) \\ &= w_t - \eta_t(\lambda w + \frac{1}{n}\sum_{i=1}^{n} \nabla_w \ell(f_w(x_i), y_i)), \end{aligned} \tag{2.2}$$

where η_t is the learning rate at time t that scales the size of the gradient step.

Stochastic gradient descent (SGD) is similar, only it visits each example one at a time instead of working with the entire database. It computes the gradient based on only one training sample in an iteration instead of the whole training set. For index i, the update rule becomes

$$w_{t+1} = w_t - \eta_t(\lambda w + \nabla_w \ell(f_w(x_i), y_i)). \tag{2.3}$$

SGD is more preferable on very large training sets, or in distributed applications. It has two restrictions regarding the learning rate, namely we have to have $\sum_t \eta_t^2 < \infty$ and $\sum_t \eta_t = \infty$. These turn out to be necessary conditions for convergence [17].

A popular way to accelerate the convergence is the use of *mini-batches*, that is, to update the model with the gradient of the sum of the loss functions of a few training examples (instead of only one) in each iteration. This allows for fast distributed implementations as well [36].

In this thesis, we use *Logistic Regression* [78] optimization algorithm. It has an associated loss function that we can use along with SGD to train the corresponding model. In this case, the optimization problem is expressed as a maximization problem, since it is more natural to think of it as maximizing the logarithm of the likelihood

$$w = \arg\max_{w} \frac{1}{n} \sum_{i=1}^{n} \ln P(y_i|x_i, w) - \frac{\lambda}{2}||w||^2, \tag{2.4}$$

where $y_i \in \{0, 1\}$, $P(0|x_i, w) = (1 + exp(w^T x))^{-1}$ and $P(1|x_i, w) = 1 - P(0|x_i, w)$.

Although we have only discussed binary classification, here we will experiment with the more general multi-class algorithms, where we have instances taken from K different classes ($C = \{0, 1, \ldots, K-1\}$). A popular approach is to learn K distinct binary classifiers [14], one for each class. When using logistic regression the objective function can be readily generalized to multiple classes [14].

2.2 System Model and Data Distribution

As our system model we consider a network of a potentially large number of computational units (e.g. personal computers, smart phones, tablets, wearable units, or smart meters), called nodes. The nodes in the network can communicate via messaging with their neighbors. At every point in time each node has a set of neighbors forming a connected network.

The set of neighbors is either hard-wired, or given by other physical constraints (for example, proximity), or set by an overlay service [49, 91]. Such overlay services are described in the literature, but fall outside of the scope of our present discussion. It is not strictly required that the set of neighbors be random; however, we will assume this for the sake of simplicity. If the set is not random, then implementing a random walk with a uniform stationary distribution requires additional well-proven techniques such as Metropolis-Hastings sampling and structured routing [100].

Nodes can leave the network or fail at any time. In our simulations we will assume that when a node leaves the network it retains a subset of its state until it joins the network again; but this is not a critical assumption. We assume a reliable transfer protocol. This implies that messages are not dropped, so communication fails only if the source or

Algorithm 2.1 Gossip Learning Framework

```
1: (x, y) ← local training example
2: currentModel ← initModel()
3: loop
4:     wait(Δ)
5:     p ← selectPeer()
6:     send currentModel to p
7: end loop

8: procedure onReceiveModel(m)
9:     m.updateModel(x, y)
10:    currentModel ← m
11: end procedure
```

target node fails before transferring the full message. Messages can be delayed up to a finite delay. We do not assume synchronized time. In our experiments we will base our simulated model on a real smartphone trace.

We assume a horizontal distribution, which means that each node has full data records. We are mostly interested in the extreme case where each node has only a single record. The database that we wish to perform data mining over is given by the union of the records stored by the nodes. Hence, we shall suppose that the network size is equal to the number of all record and we will denote both by n.

2.3 Gossip Learning

The Gossip Learning Framework [82] is a possible way to learn models in fully distributed environment. The basic idea is that in the network many models perform random walks and are updated at each node using the local example. More precisely, every node executes Algorithm 2.1. A node in the network first initializes a local model, then iteratively sends its local model to a randomly selected node in the network. The address of the randomly selected node is provided by a peer sampling service (e.g. the NewsCast [105] protocol). When a node receives a model, it updates it via 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. It is possible to replace the local update step with a more sophisticated aggregation operation. Upon receiving a model, the node can merge it with the local model. Merging is typically achieved by averaging the model parameters. A node may have information about multiple data records instead of just a single record. In this case, a minibatch approach can be implemented.

CHAPTER 3

Smartphone Trace

Today in smart systems, distributed computing over the edge is becoming a popular research topic [33]. Research into algorithms that are suitable for such environments often involves actual deployments, because realistic conditions are non-trivial to model, but they are crucial for finding an optimally efficient and robust solution. However, this difficulty severely limits the possibilities of exploratory research. Smart portable devices also represent a seemingly ideal platform for peer-to-peer (P2P) protocols for a wide range of applications, but the adoption of P2P technology has been very slow. One important domain is smartphone applications, which can form a part of a variety of smart systems like smart city and e-health solutions [109]. In this domain, it is vital to fully understand the capabilities and limitations of the devices and their network access as well. This includes battery charging patterns, network availability (churn) and network attributes (for example, NAT type).

One important open problem is to understand the patterns of availability; that is, to build models of churn which reflect the intervals when a given device can potentially participate in a P2P protocol. Understanding these patterns could answer the question of what applications are feasible without cloud support, and what applications are of little use. It would also allow one to design specific algorithms that maximize the utility of the

available time of the devices.

For desktop systems, in-depth churn studies are available [99] but, as we will show, these are not applicable for smart devices. There are numerous data collection efforts related to energy usage [32, 51] or data traffic [52], but these are not sufficient to enable research over the edge. There are generic data collecting platforms, the closest to our work is Device Analyzer [108]. However, it does not detect the NAT type, which is a crucial part of P2P communication models; it only logs locally available data. What is more, the incentive model builds on the desire of users to contribute to scientific projects. In contrast, we wish to offer good functionality as the main incentive for people to download and use our client application [32].

Over the years, there have been many data collection campaigns that target smartphones. This includes the famous Mobile Data Challenge (MDC) [61], which sought to collect large amounts of data from smartphones for various research studies, including sensory data, cell towers and calls. It ran between 2009 and 2011 and produced the largest and most widely known mobile big data set so far. Later, the project that had similar results was the Device Analyzer Experiment. It commenced in 2011 at the University of Cambridge, and the team attempted to not only record similar attributes to the MDC, but also to record system-level information such as phone type, OS version, energy and charging [18, 107]. This trace was used, for example, to determine the most energy greedy Android APIs [65] and to reconstruct the states of battery levels on the monitored smartphones [34]. Our dataset is unique in that, apart from being over 5 years in duration, it contains all the necessary attributes to simulate *decentralized* applications.

Another set of projects was concerned with measuring the network (e.g., detecting NAT boxes) as opposed to collecting a full trace from the devices, which is our main goal. For instance, in 2014 a study was initiated to analyze the deployment rate of carrier-grade NATs that can hide entire areas behind a single public IP address [90]. The measurement was based on Netalyzr, as well as on crawls of BitTorrent DHT tables to detect possible leaked internal addresses due to hairpin NAT traversal. In another study across Europe, an application called NAT Revelio was developed [73]. Yet another data collection campaign attempted to collect traceroute sessions from smartphones using the custom TraceboxAndroid application [104]. The application detects the exact number of middleboxes and NAT translations encountered between the device and a specified test target. In a similar two-week campaign, the Netpicular application was deployed [110]. Also, a

Table 3.1. Comparison between various NAT measurement campaigns

Source	Collected Attributes	Length	Public	Tools
[90]	local, external and public IP addresses	2014-2016	No	Netalyzr
[73]	external IP, mapped port, traceroute results, UPnP query results	2016 May and August	No	NAT Revelio
[113]	traceroute results	2016 February - 2017 February	No	Mobile Tracebox
[110]	traceroute results, number of detected middleboxes	2011 January, 2 weeks	No	Netpiculet
[104]	traceroute results, number of detected middleboxes	2014 May - September	No	TraceboxAndroid

mobile application called Mobil Tracebox was deployed to carry out traceroute measurements [113]. This campaign ran for an entire year. A summary of these NAT studies can be found in Table 3.1.

Next, we will give an outline of how we carried out our data collection and we give a brief summary of the main milestones achieved. Then, we will introduce a time-inhomogeneous Markovian model and the smart phone trace for simulating churn. We shall use this gathered churn trace throughout the thesis for evaluating our fully distributed protocols, because we can realize more realistic network churn simulations based on it. After, we will describe some lessons that we learned from data collection over the years. We will include results obtained from real P2P connection experiments.

3.1 Collecting the Data

We developed and deployed an Android app that collects data covering most of the aspect that are relevant to the design of P2P protocols over networks of smartphones, including time series of network and battery status complete with information about NAT types, network types, and network providers. We have made our trace publicly available[1]. This allows the research community to design and validate, if they wish, realistic simulation models.

In 2014, we developed an Android app called Stunner that informs the user about the

[1] `http://www.inf.u-szeged.hu/stunner`

current network environment of the phone: private and public IP, NAT type, and other details.[2]

The app was launched in April 2014, when it was simply made public without much advertising. Most of the users in our survey installed our app voluntarily because it was useful for them. About 30 users were local students recruited for the survey. They installed the app, but they received no further instructions.

In the initial release of STUNNER, the data is collected by a background service that can be disabled by the user at any time. This background service listened to various events broadcast by Android that were related to the network interface and the status of the battery. In particular, it listened to TELEPHONYMANAGER, WIFIMANAGER, and BATTERYMANAGER. When such an event arrived, or when the user explicitly ran the app, it collected the status of the network and the battery and logged this information. There were periodic measurements as well every 10 minutes, if no other events occurred. The network properties we collected include network type (WiFi/cellular), carrier, signal strength, bandwidth, public and private IP and NAT type. Regarding the battery, we stored the temperature, voltage, load percentage, health and charging status (from AC/USB/WiFi). The data was timestamped using the UTC real-time clock of the phone, along with time zone information so that the local time can be calculated. The data was periodically uploaded to our server in an anonymized form. The devices were identified by a 128-bit random number that was generated during the installation of the application. After a month of data collection there were 622 installations to different mobile phones and we collected data from 91 countries and from 1425 different networks.[3] The geographic distribution of the first users is shown in Figure 3.1.

Over years, we found many pitfalls related to data collection. The data collector server was down in 2015, when the project was temporarily neglected. However, we continued the development in 2016 after a year shutdown. We had an intensive review on the source code of STUNNER and we proposed multiple data cleansing methods [101, 102]. Also, some of the STUN servers that were initially wired in to the clients disappeared over the years. Therefore, it was crucial to correct this error and also correct the failed NAT

[2]The type of the NAT is detected with the help of the STUN protocol using public STUN servers. We used the implementation `http://jstun.javawi.de`.

[3]Based on the AS number and the city determined by the public IP address using the service provided by Telize: `https://www.telize.com`

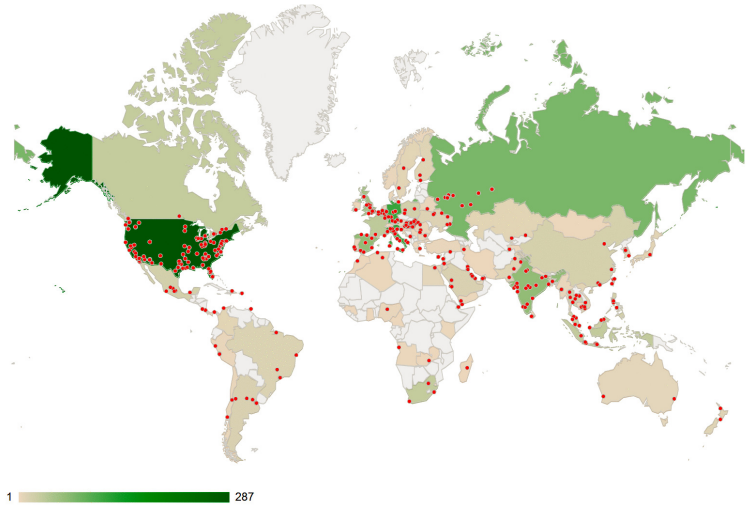


Figure 3.1. Locations of the contributions to our data set. The color coding represents the number of different network providers we collected NAT data from.

measurements. Later, we will describe a method that is used to tackle this problem in Section 3.4.1.

The latest version was completely redesigned and it was released in 2019. This was necessary because Android had become very hostile to background processes when the phone was not on a charger, in an effort to save energy. For this reason, we now collect data only when the phone is on a charger. This, however, is not a real issue, because for decentralized applications these are the most useful intervals, when it is much cheaper to communicate and to perform computing tasks in the background. Android event handlers have also became more restricted, so we can use them only under limited circumstances or on early Androids. The events raised by connecting to a charger or a network can still be captured by the Android job scheduler, but the timing of these events is not very reliable.

For this reason, instead of relying on event handlers, we decided to check the state of the phone every minute; and if there is a change in any important locally available networking parameter or in charging availability, we perform a full measurement. A measurement is still triggered if the user explicitly requests one, and it is also triggered by an incoming P2P measurement request. Also, if there is no measurement for at least 10 minutes, a full measurement is performed.

P2P connection measurements are also a new feature in the latest version that are per-

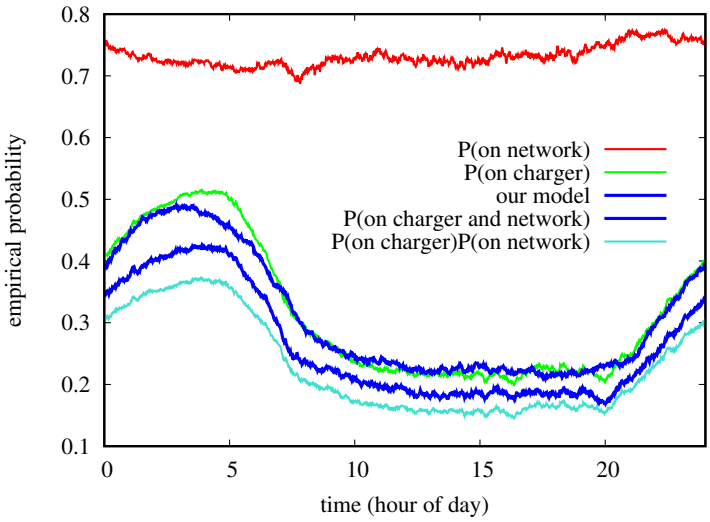


Figure 3.2. Diurnal pattern of availability. The plot shows the proportion (empirical probability) of different types of phones, as well as the prediction of our churn model, as a function of time.

formed every time a measurement is made. They are based on the WebRTC protocol [2], with Firebase as a signaling server [80], and a STUN server [70]. We build and measure only direct connections and the TURN protocol for relaying is not used. Every node that is online (has network access and is on a charger) attempts to connect to a peer. To do this, the node sends a request to the Firebase server after collecting its own network data. The server attempts to find a random online peer and manages the information exchange using the Session Description Protocol (SDP) to help create a two-way P2P connection over UDP. If the two-way channel is successfully opened then a tiny data message is exchanged. The channel is always closed at the end of the measurement. One connection is allowed at a time and every additional offer is rejected. The signaling server maintains an online membership list. The first version of our P2P connection measurement implementation also induced some downtime in 2018.

Over the years, at any point in time we had a user base of a few hundred to a few thousand users, and over 40 million measurements have been collected from all over the world. Now, we should add that these changes are in our latest article on STUNNER. So we have not yet exploited its advantages. We will describe it in some detail in Section 3.4. However, note that throughout this thesis we will just use the trace that was collected in 2014.

3.2 A Markovian Model for Simulating Smartphone Churn

Here, we describe and analyze a time-inhomogeneous Markovian model of churn that predicts the length of the next session based on the length of the previous session and the time of day. We validate our model by comparing it with the trace we collected. Simulating churn with such a synthetic trace is a good option when we wish to simulate a larger network than one available based on the real trace. This study was conducted on a data set that was collected in the first month of the data collection. We will restrict our data set to those continuous measurements that cover at least one day without interruptions. This is done so as to reduce the bias introduced by short measurement intervals arising from the diurnal pattern in the data.

When is a Phone Available?

We model each phone as a series of alternating *available* and *unavailable* sessions. Intuitively, the available sessions are those during which the phone can participate in a P2P protocol.

Unlike desktop systems, with smartphones the battery state is more important for determining availability than network connectivity. P2P applications crucially rely on bi-directional communication that costs energy, which is a precious resource on mobile devices. As shown in Figure 3.2, phones have a network connectivity around 75% of the time. Recall that this is a statistic over traces that are at least a day long, which indicates excellent connectivity in general. This is made more complicated because NAT types need to be taken into account (see Section 3.4.2). In spite of this, the major problem is energy, not communication.

Based on these observations, we can say that the peer is available if it has network connectivity and if it is on a charger. In this section, we do not differentiate between network types (WiFi or 3G) and charging types (USB or AC), although a finer grain analysis is also possible based on our data. Figure 3.2 tells us that being on a charger is more decisive. It is interesting to note that being on a charger and on a network are not independent properties. As seen in the figure, the observed probability of the co-occurrence of these two properties is higher than that predicted by an independence assumption. In other words, being on a charger increases the probability of the phone being connected to the Internet.

Modeling Availability

We model a user j as a series of alternating available and unavailable sessions will be denoted as

$$\ldots, a_{i-1,j}, u_{i,j}, a_{i+1,j}, u_{i+2,j}, \ldots \tag{3.1}$$

where $a_{i,j}$ and $u_{i,j}$ denote the length of the ith available or unavailable session of user j, respectively. Let $t_{i,j}$ denote the starting time of the ith session of user j. Note that $t_{i+1,j} = t_{i,j} + x_{i,j}$ (where $x = a$ or $x = u$, depending on i).

We would like to stochastically model this series based on our measurement data so that we can generate traces for the purposes of simulation. More precisely, for a given i and j, we wish to learn the distribution of $a_{i,j}$ (or $u_{i,j}$, respectively). In the general case, this probability distribution can be formulated as the conditional distribution

$$P(a_{i,j}|t_{i,j}, u_{i-1,j}, t_{i-1,j}, a_{i-2,j}, t_{i-2,j}, \ldots). \tag{3.2}$$

A similar formula can be given for $u_{i,j}$.

Evidently, this expression has too many parameters to approximate so we will now introduce a time-inhomogeneous Markovian model that considers only the length of the previous session and the starting time of the session:

$$P(a_{i,j}|t_{i,j}, u_{i-1,j}), \text{ and } P(u_{i,j}|t_{i,j}, a_{i-1,j}). \tag{3.3}$$

In principle all users j will have their own specific versions of these two distributions. However, since we had data that was collected for only a month, we did not have sufficient data for most of the users. For this reason, here we will assume that all the users have the same distributions. In other words, we will create a model of an average user. Note that the data will allow us in the future to identify user types and thus to create more specific models involving mixtures of users.

To make clear why we need to keep the previous session length and the starting time as conditions, Figure 3.3 shows the distributions $P(\ln a_{i,j}|t_{i,j})$, $P(\ln u_{i,j}|t_{i,j})$, $P(\ln a_{i,j}|u_{i-1,j})$, and $P(\ln u_{i,j}|a_{i-1,j})$, respectively. To be more exact, instead of absolute time we plot the dependence on the hour within a day as the condition, since the dependence on time is mainly due to the diurnal pattern of phone usage behavior. Note also that we work with the logarithm of the session lengths. This is because we noticed that the interesting patterns

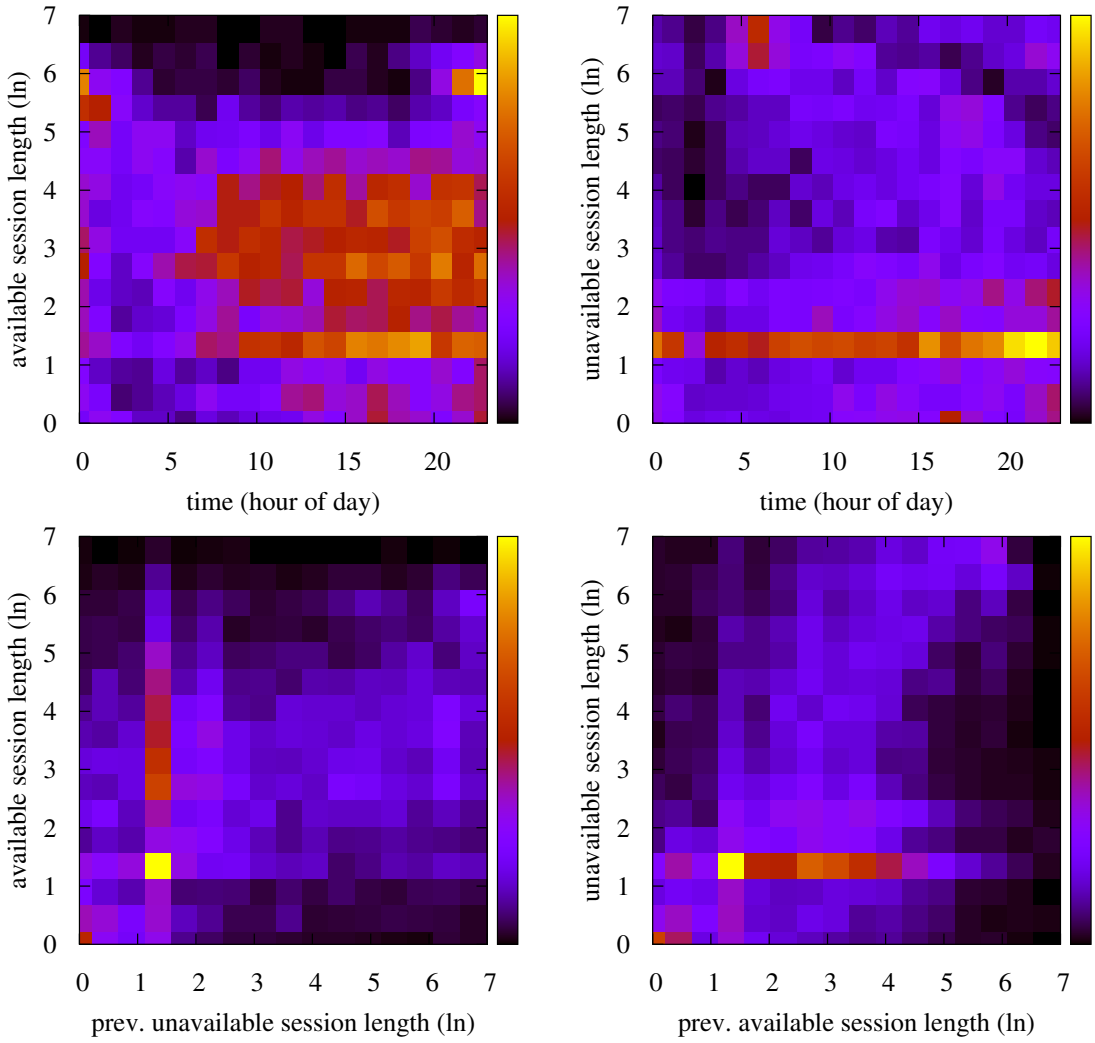


Figure 3.3. Conditional distributions of the logarithm of available and unavailable session lengths with the hour of day and previous session length as conditions. In the heatmaps warmer (lighter) colors indicate higher values. The original session lengths were measured in minutes.

of the distribution were more obvious on the log scale. On the linear scale the distribution appears to be a simple heavy tailed distribution without any apparent structure. For this reason we model the distribution of the logarithm of the lengths and then take the exponential to generate actual session lengths during simulation.

As can be seen, the distributions have complex patterns. For example, it is clear that after 8pm many long available sessions start that last roughly till the morning Similarly, in the morning at around 6-7am many long unavailable sessions start. This has to do with

the fact that many phones are left on a charger during the night. Figure 3.2 also supports this particular interpretation. Also, most of the sessions are rather short, lasting only a few minutes. After inspecting the data carefully, we hypothesize that this behavior is mostly the result of a weak unreliable WiFi or mobile network signal that induces quickly alternating short sessions, as implied by the rightmost two plots in Figure 3.3.

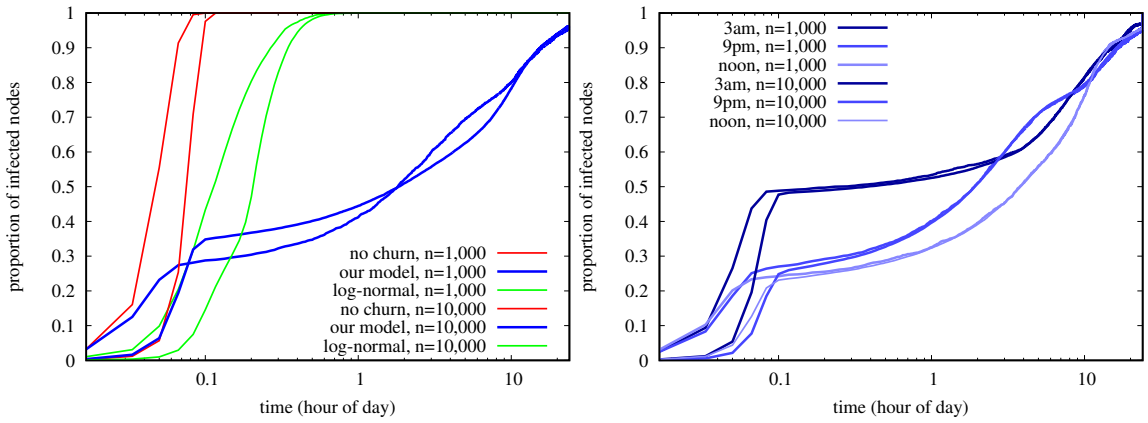
In our model, instead of introducing a parametric approximation of (3.3), we kept the original data (session lengths classified by starting time and previous session length) and resampled these classes when generating the next session length. In order to have enough data in each class, we reduced the resolution of the time and session length parameters that condition the distributions. As for time, we differentiate between 8 different intervals and divide the 24-hour day into 3 hour intervals starting at midnight. As regards previous session length, we define three different intervals over the logarithm of the lengths heuristically based on the observed distributions in Figure 3.3. These three intervals are $[0 : 2)$, $[2, 5)$ and $[5, 9]$. These two low resolution variables define $8 \cdot 3 = 24$ classes for both available and unavailable sessions.

Figure 3.2 shows the observed proportion of available phones when using the model. We modeled a network of 1000 nodes for 1000 days and calculated the statistics based on this. Note that the model does not control the proportion of available nodes directly; it is an emergent property that is suitable for validating the model. We can see that our predictions are slightly higher than the actual observed proportion. This is due to the fact that currently we are not always able to account for intervals when the phone is switched off and therefore the lengths of unavailable sessions are slightly underestimated. By taking into consideration only long continuous measurement intervals we minimize this effect.

An Example Application

Here we illustrate the application of our churn model using the simulation of a simple push-pull gossip broadcast protocol. Our motivation is to shed light on the importance of having a realistic churn model to help understand the behavior of fully distributed protocols. As we will see, in our model gossip behaves in a radically different way from that in simpler churn models.

The protocol we simulate is a classical push-pull gossip broadcast protocol over a



(a) under different churn models.

(b) started at different hours, using our churn model.

Figure 3.4. Results of experiments on push-pull gossip broadcast.

static overlay network. That is, each node has a fixed set of neighbors throughout the simulation. Each node i contacts one random available neighbor j (if there is one available) for each round and if i has the update, it sends it to j, and if j has the update, it sends it to i. The round length is one minute. Initially, one random node has the update. We ensure this initial node is available (online) at the start of the broadcast. Unless otherwise stated, the plots show the average of 1000 runs and are based on a random 20-out overlay topology (each node has 20 random out-neighbors).

In the first set of experiments, we compare three churn models; namely no churn, log-normal churn (a classical model generally used in P2P simulations [99]), and our model. The parameters of log-normal churn were set so that the average and the variance of the available session lengths were the same as in our model, and we also made sure that the average proportion of available nodes (that is kept constant) matches the daily average of our model as well. The log-normal model works by drawing the available session lengths from the log-normal distribution, and if the proportion of available nodes drops below the fixed threshold, unavailable nodes are made available at random.

Figure 3.4a shows the results for two different network sizes. It is striking how different the dynamics are from the more homogeneous log-normal model. The reason will become clearer after considering our next set of experiments, where we commenced the broadcast at specific hours. Figure 3.4b shows the results we got.

It is apparent that the broadcast reaches the nodes that are available at the start of the broadcast in a few rounds (see also Figure 3.2). Then progress slows down since

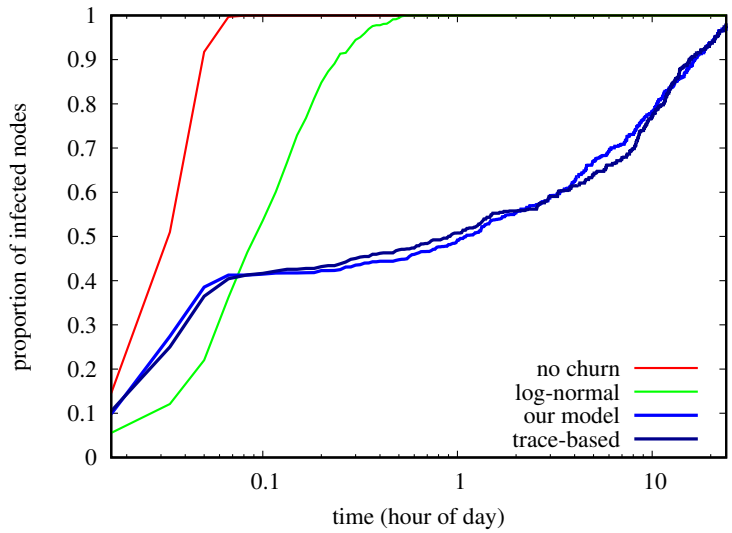


Figure 3.5. Validation experiments – Push-pull gossip broadcast under different churn models including a trace-based simulation (n=140 for all churn models).

new nodes join slowly. However, clearly, the speed at which it spread also depends on the distribution of session lengths in the different time intervals during spreading (see Figure 3.3).

We will also include a validation experiment, during which we simulate churn based on the exact trace we collected. That is, we select those users for which we have at least a 3-day long continuous measurement interval. We found 140 such users. In our simulation we simulated the exact availability of these users as recorded in our data. In Figure 3.5 we compare this trace-based simulation with the churn models we examined in Figure 3.4a. Here we used a 10-out random topology, and the plot shows the average of 3 days where spreading is started at midnight. The trace-based simulation closely follows our time-inhomogeneous Markovian model, which confirms the validity of the approach.

3.3 Simulations on a Smartphone Trace

Now, we introduce multiple network churn simulation techniques that will make the simulations more realistic. We base all of our simulations on a real trace of smartphone user behavior. Alongside this, we present a detailed description of trace properties. Throughout this thesis, we evaluate all of our proposed fully distributed protocols by using simu-

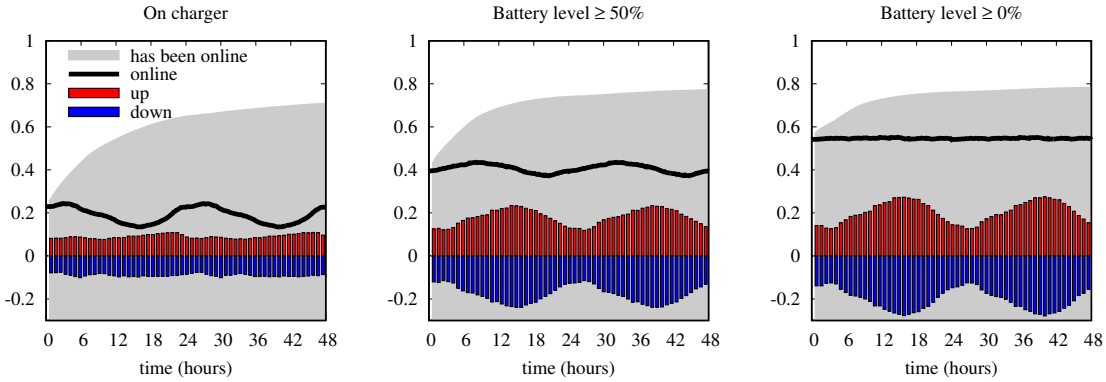


Figure 3.6. Proportion of users online, and proportion of users that have been online, as a function of time. The indicated time is GMT. The bars denote the proportion of the simulated users that log in and log out (shown as a negative proportion), respectively, in a given period.

lations with presented setups.

The examined trace was collected in 2014, which was the first year of the data collection. We have traces of varying lengths harvested from 1,191 different users. We divided these traces into one-day segments, resulting in 41,849 segments altogether. With the help of these segments, we can simulate a virtual period of up to one day by assigning a different, randomly selected segment to each simulated node. The sampling of one-day segments is performed without replacement. When the pool of segments runs out (which happens when we need more nodes than there are segments) we re-initialize the pool with the original 41,849 segments and continue the sampling without replacement. This way, we can simulate networks larger than 41,849 nodes. For example, later we will simulate a network of size 100,000 for a one-day period. Also, we created 2-day long segments (with a one-day overlap) for longer simulations. It resulted in 40,658 segments altogether.

The requirement of being on a charger or the expected minimum level of battery power can be a free parameter of any evaluation. We can implement different scenario by defining requirements of availability in network. If the predefined requirements are fulfilled on a node then it is available for communication. To ensure an algorithm is phone and user friendly, we can define a user to be online (available) when the user has a network connection and the phone is connected to a charger; then we never use battery power at all. In another approach, we can simulate the case where a participating phone is required to have at least a certain battery level. From the point of view of churn, though, the worst

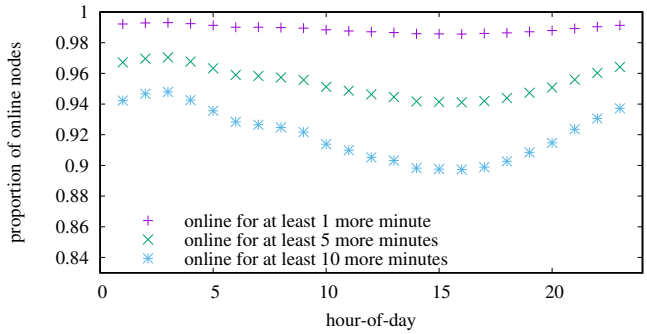


Figure 3.7. Expected availability of smartphones that have been online for at least 10 seconds. The hour-of-day is in UTC. All battery levels are allowed.

case is when phones with any battery level are allowed to join, because this results in a more dynamic scenario.

The observed churn pattern is shown in Figure 3.6 based on 2-day periods we identified. Here, we should add that these patterns of availability are based on significantly larger amounts of data than we have displayed in Figure 3.2. Although our sample contains users from all over the world, they were mostly from Europe, and some were from the USA. The indicated time is GMT, thus we did not convert times to local time. It is interesting to note that if we consider only network availability (any battery level is allowed), then the diurnal pattern becomes apparent in the login and logout frequencies due to short session lengths during the day, and long sessions during the night. The network availability itself is static. If we require the phone to be on a charger, then the diurnal pattern of availability becomes clear. During the night, more phones are available (as they tend to be on a charger), but the churn rate remains lower. Note that, due to this sampling method, users are represented with a probability proportional to the number of days they were online. This is motivated by the observation that our proposed protocols at any given point in time can operate only with users that are actually online, hence those types of users that spend more time online are indeed encountered proportionally to their online-time.

Figure 3.7 presents statistics about smartphone availability. For each hour, we calculated the probability that a node that has been online for at least a 10 seconds remains online for 1, 5 or 10 more minutes. Note that for us these probabilities are important because our protocols have to remain connected at least for the short amount of time that

it takes to propagate a message. As can be seen in the figure, these probabilities are rather high even for a 10 minute extra time. However, the first 10 seconds of each online session (or the entire session if it is shorter) are considered offline because extremely short online sessions would introduce unreliability. This technique was also explicitly implemented as part of our protocols: a node should simply wait 10 seconds before joining the network.

Users with a bandwidth of less than 1 Mbps were treated as offline. This choice was motivated by two factors. First, the Internet bandwidth available to users exceeds 1 Mbps in many developed countries, even for uploads [97]. Indeed, in our trace the probability of encountering a connection with a bandwidth lower than 1 Mbps was only 3.86%. Thus, excluding such devices will cause only a minimal loss of data but in return slow devices will not slow the entire network down. Second, utilizing a device with such a low bandwidth would place too much of a burden on the device, and user-friendly applications might wish to avoid this. Applications based on our algorithm will mostly run in the background while collecting data and communicating with other devices. To ensure that an application of this kind is user-friendly, all the background processing needs to be transparent to the user.

With this consideration, in our experiments we will use 1 Mbps not only as a lower bound, but also as an upper bound. That is, the algorithm is allowed to utilize at most 1 Mbps of the available bandwidth, irrespective of the total available bandwidth, in order to avoid overloading the device. Of course, utilizing all the available bandwidth would result in a more favorable convergence speed.

3.4 Lessons Learned on Smartphone Trace Over the Years

Our team began developing the smartphone app called STUNNER in 2014. Since then, we have collected a huge trace involving millions of individual measurements. This amount of data naturally contains noise and some incorrect records. We took great care to clean this data. Here, we propose a method to correct failed NAT measurements.

Our application called STUNNER has been collecting data for a much longer time than any of the other related applications, and this allows us to identify trends over time. We collected a wide range of properties simultaneously, including NAT type, battery level, network availability, and so on, to be able to fully model decentralized protocols.

In this section, we present some base statistics on NAT type distribution and real P2P connection measurements.

3.4.1 Data Cleansing

The main feature of our application is the discovery of the NAT type. Users can ask the application about their NAT information and public IP address. This method is based on User Datagram Protocol (UDP) message-based communication between the device and a randomly picked STUN server. A STUN server can discover the public IP address and the type of NAT that the clients are behind.

We were confronted by a problem that was caused by the prefixed STUN server list. It contains a list of 12 reliable servers that are suitable for NAT detection, this list being embedded inside the application code. It allows the device to randomly pick a STUN server. As a result, every measured NAT type in the timeline is based on a different STUN server's NAT test. Hence it makes the measured data more trustworthy. This random pick approach was well designed and worked well initially. However, after a time four of the STUN servers went offline without any prior notice. Since then these four failed STUN servers provide the same NAT discovery result code as firewall blocked connections, even though a part of those records had an online NAT type. Therefore we proposed a solution on how to correct it and make the collected data useable afterwards. Quite surprisingly, another solution was required to avoid connections to a failed STUN server. This problem appeared in year 2016. We had already corrected this problem in one of the previous versions of STUNNER. This kind of uncertainty in data appears in data only before the changes in implementation. Hence, the below-mentioned properties (e.g. trigger events) became obsolete after the latest release.

The server fails appeared with a 4/12 probability, and the event of consecutive repeated fails has an exponential pattern. Consequently we define sessions with consecutive repeated *Firewall blocked* discovery result codes and looked at their distribution. If the distribution is roughly an exponential distribution, then we can interpret them as online and we can define their network properties. Otherwise, the others that do not have an exponential fit will remain *Firewall blocked*. This means that in this way we cannot prove the opposite (a firewall blocks the connection). In general, we look for a session that begins and ends with the same network property and there are only uncertain online states between them. These sessions may be interpretable based on the begin-end enclosures. More specifically, the sessions must

- begin and end with the same NAT discovery result code

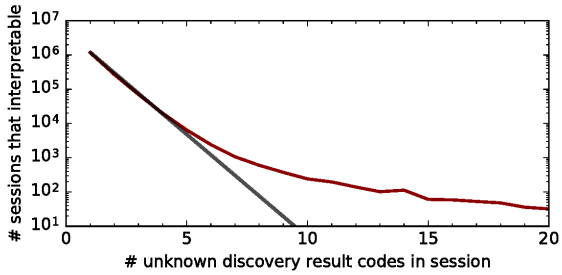


Figure 3.8. Discovery result code enclosed by sessions.

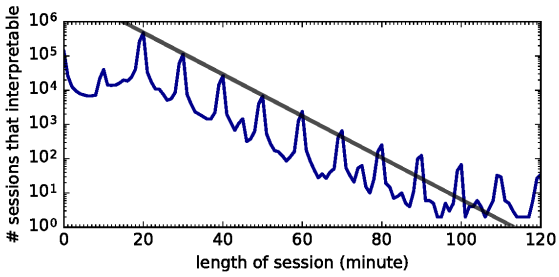


Figure 3.9. Length of candidate sessions.

- begin and end with the same IP address
- contain only uncertain online states
- contain a time gap between two records only in a range of 0 to 15 minutes based on the fact that the maximum time gap between two regular online records is almost 10 minutes. However, it is not very accurate because of the Android support scheduler with its inexact trigger time requirements.
- not be interrupted by trigger events that correspond to any potential change in network properties.

We show these candidate sessions in Figure 3.8 and Figure 3.9. Let us first take a look at how many uncertain discovery result codes are enclosed by these sessions in Figure 3.8. It is clear that the first four points seem to fit an exponential curve. Consequently, it is still open to interpretation and the rest of the points remain undefined. Next, Figure 3.9 shows the length of the above-defined sessions. There are some peaks roughly every 10 minutes. These peaks correspond to the trigger event that was scheduled every 10 minutes and this was the most common trigger event. For instance, if there is exactly one uncertain *Firewall blocked* value in the appropriate session and every taking of a measurement is triggered by this schedule event, then its length of time is around 20 minutes. Based on this example, an above-defined session that contains three unknown records lasts for 50 minutes. Accordingly, we examined the points from the first phase up to 50 minutes. Our analysis revealed that it also had an exponential pattern. In contrast to the distribution in Figure 3.8, this distribution appears more complex, but it is still acceptable. Next, we associate the two findings. More specifically, the intersection of the two sets is an above-defined session that contains fewer than five uncertain elements and it lasts no

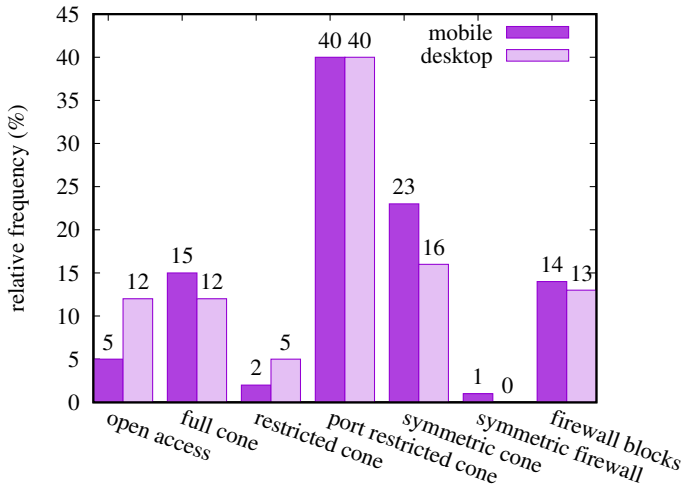


Figure 3.10. Relative frequency of NAT types in use aggregated over time.

longer than 50 minutes. Based on this rule we can correct the network properties of 6.7% measurement records.

In a paper, we give further details about data cleansing, such as cleaning data record duplication and correcting the overlap of the client-side timestamps [101].

3.4.2 NAT Type Distribution

As one of the unique aspects of the data we collected is that it includes NAT types, here we will briefly present some interesting statistics about NATs. Although in many cases NAT devices can be dealt with via low level "hole punching" solutions[92], they can also represent design constraints at higher levels due to the potentially high cost of (and perhaps the complete lack of) such solutions [91].

The results of our measurements show similar trends to that of earlier measurements [21] in the predominantly desktop P2P ecosystem, as shown in Figure 3.10. The chart is based only on successfully identified "classical" NAT types (in order to allow a comparison with earlier desktop data), and 10% of the identification attempts were unsuccessful. We used only those continuously measured time series that covered at least one day for a phone without interruptions to restrict bias due to diurnal patterns in the data. The chart is based on data that was captured without any server-related error in the first month of data collection. The most significant difference is the very low percentage of open access peers

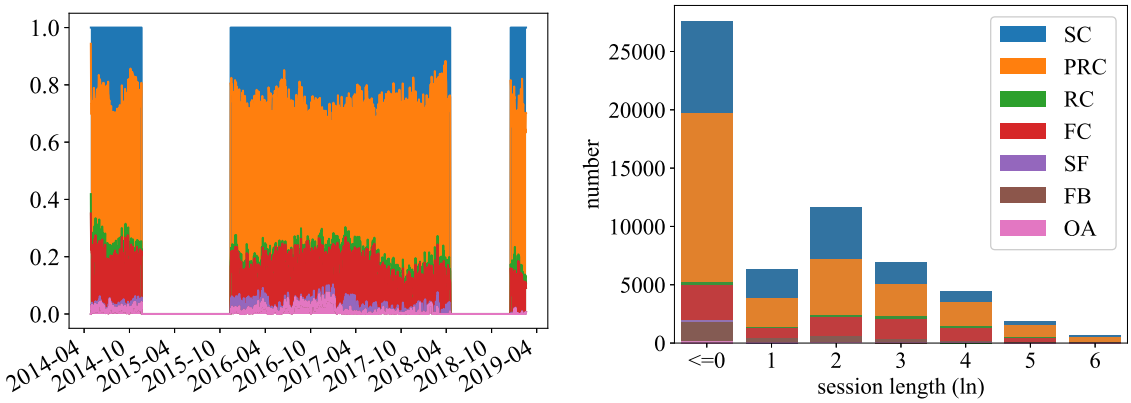


Figure 3.11. (1) NAT distribution per day over 5 years. (2) Session length distribution. Examined NAT types: SC - Symmetric Cone, PRC - Port Restricted Cone, RC - Restricted Cone, FC - Full Cone, SF - Symmetric UDP Firewall, FB - Firewall blocked, OA - Open Access

and a higher percentage of the symmetric cone NAT type. That is, smartphone users are slightly more restricted than desktop users.

Now, we illustrate the dynamics of the NAT distribution over the years (see Figure 3.11, left). The distribution is based on continuous sessions of online users. These continuous sessions of homogeneous network conditions were determined based on the measurement records. A session has a start time, a duration, and a NAT type. The distribution is calculated based on the number of aggregated milliseconds of session durations falling on the given day. The distribution of online time per day is always about 8%. Recall that here the online state is meant to imply that the phone is on a charger.

The plot has gaps because of the previously mentioned downtime in data collection. We also said that some of the STUN servers have stop working over the years. As a result, the *Firewall blocked* NAT type is not reliable, so we will exclude this category from the figure. Note that the distribution is surprisingly stable over the years.

We display the session length distribution as well in Figure 3.11 (right). The session length is in minutes and the bins for the histogram are defined on a logarithmic scale. Sessions shorter than one minute are not always measured accurately due to our one-minute period of observation, so we will group such sessions in one bin (<= 0).

Figure 3.12 contains stacked bar charts showing the distribution of different NAT types in the 6 continents and in the networks of the top 10 most represented providers in 4 different years. The most common NAT type is the Port Restricted Cone, except in Africa

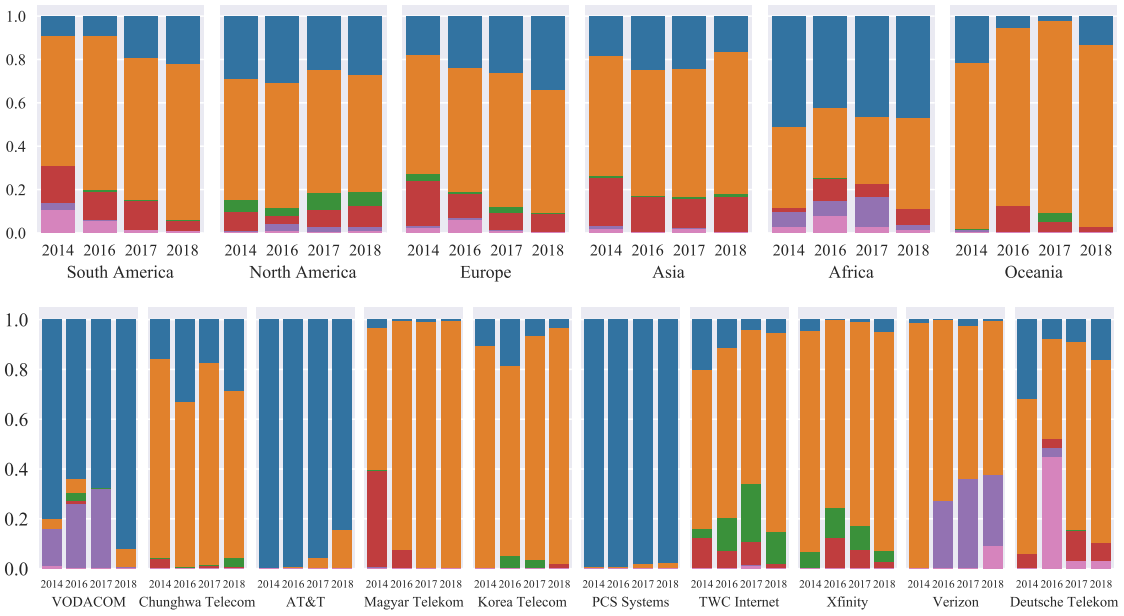


Figure 3.12. NAT type distribution by continent in 4 different years (top) and NAT type distribution by the top 10 providers in 4 different years (bottom). The colors represent types as defined in Figure 3.11.

where the Symmetric Cone has a relatively larger share. According to the chart the rarest NAT type is Open Access everywhere. Interestingly, the NAT type distribution is very different among the different providers, unlike the distributions based on geographic location.

3.4.3 Real P2P Connection Measurement Results

We extended the application called STUNNER to collect data concerning direct peer-to-peer capabilities based on a basic WebRTC implementation. Although our NAT measurements are simply based on STUN server feedback [70] (thus they underestimate the complexity of the network), our P2P measurements tell us that our NAT type data is offer a good basis for predicting connection success. As an illustration, we shall present some of the interesting patterns in our trace related to P2P connection measurements. Figure 3.13 shows the proportions of the outcome of 63184 P2P connection attempts. The successfully completed connections amount to 34%. Let us briefly outline the possible reasons for failure. First, *signaling related error* means that the SDP data exchange via the signaling server

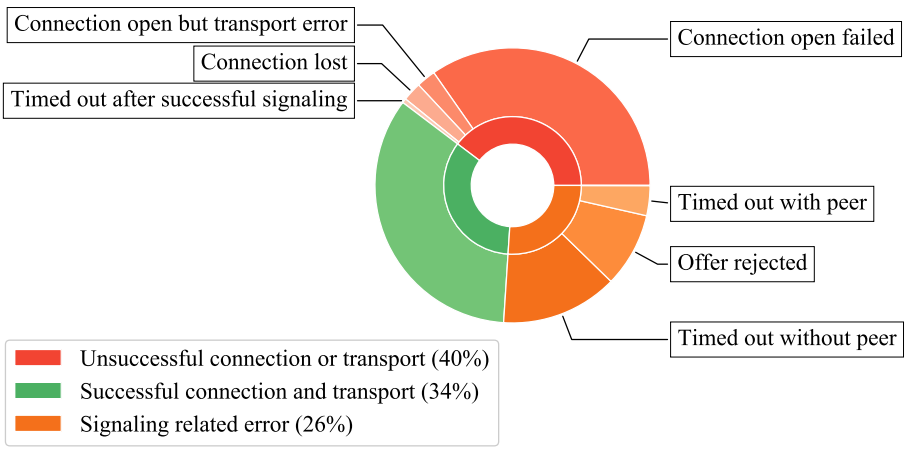


Figure 3.13. Proportions of the possible outcomes of P2P connection attempts.

failed. This can happen if the server contacts a possible peer but the peer replies with a reject message (offer rejected), or it does not reply in time (timed out with peer), or we cannot see any proof in the trace that a peer was actually connected (timed out without peer). Note that a peer rejects a connection if it has an ongoing connection attempt of its own.

If the signaling phase succeeds, we have a pair of nodes ready to connect. The most frequent error here is failing to open the channel, most likely due to incompatible NAT types. After the channel is open, sending the test message is still not guaranteed to succeed (transport error). Participant nodes may disconnect with an open connection (connection lost). In some rare cases a timeout also occurred after successful signaling; that is, the WebRTC call did not return in time.

Figure 3.14 shows some statistics over successful connections as a function of NAT type. Here, we do not include signaling related errors. Note that NAT type discovery is an independent process executed in parallel with the P2P connection test. Therefore, there are some cases where the NAT type information is missing but the signaling process is completed nevertheless.

3.5 Conclusions

In this chapter we argued that it is important to understand availability patterns of smartphones in order to assess the feasibility of P2P techniques. Our motivation was to enable

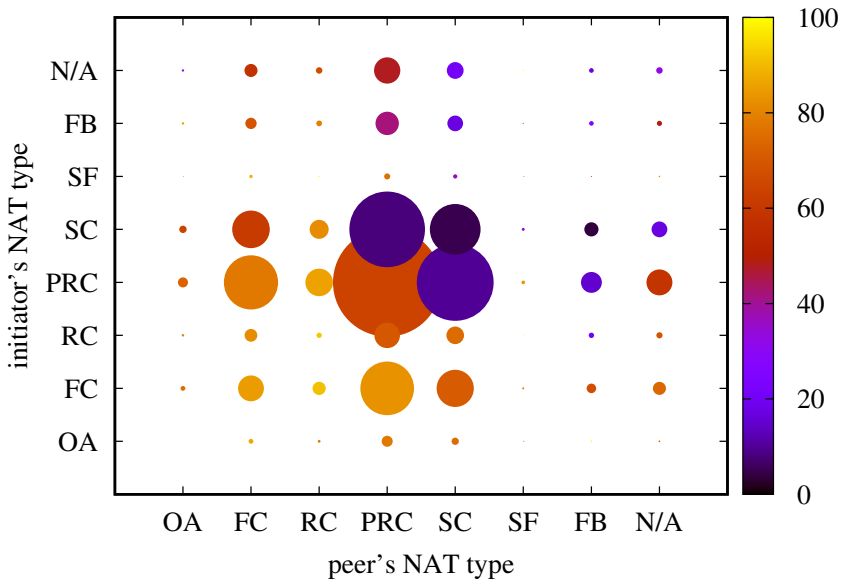


Figure 3.14. Statistics over successful connection as a function of NAT type. The area of a disk is proportional to its observed frequency, the color signifying the success rate. The examined NAT types are: OA - Open Access, FC - Full Cone, RC - Restricted Cone, PRC - Port Restricted Cone, SC - Symmetric Cone, SF - Symmetric UDP Firewall, FB - Firewall blocked, N/A-missing type

exploratory research into decentralized algorithms for edge systems. In order to be able to model the availability patterns of devices, we implemented an Android app to collect data. Our trace contains locally observable attributes such as battery status and network availability, STUN measurements, as well as direct P2P connection data. With this unique combination, we can combine these sources of data to be able to predict things like P2P connection success, or to simulate distributed protocols over the overlay networks of smartphones. Our trace spans over five years and contains over 40 million measurements. We summarized lessons that we learned over the years of data collection and analysis. We also made the anonymized version of our trace publicly available. Since our data is available, it becomes possible for the community to explore applications that can tolerate or even exploit these availability patterns.

Here, we proposed a time-inhomogeneous Markovian model based on the collected data where the conditional probability distributions of session lengths are captured by a set of the actual observations in the data that we resample when creating synthetic traces of users to model churn. We validated this model in multiple ways (see figures 3.2 and 3.5), and we found that the model captures observed availability as well as the behavior of

push-pull gossip broadcast.

We also proposed a real smartphone trace for simulating fully distributed protocols. We highlighted the trace main properties, then we examined the free parameter for battery related requirements. From these, we can have two significantly different scenarios. In the worst case churn scenario, we can state the expectation for a participant smartphone having at least a certain battery level. In a more user and phone friendly scenario, we expect the device to be on a charger.

Lastly, we briefly discussed some measurements details on NAT types and real P2P connections.

Contribution

The contributions of the author are processing and analyzing the collected data for trace based simulations, the development of trace-based simulation techniques, the development of a time-inhomogeneous Markovian model and the development of a data cleansing method to correct failed NAT measurements. The author regards most of the presented smartphone trace statistics as his own contribution with only some exceptions. The locations distribution map of users in Figure 3.1 was made by Vilmos Bilicki. And the NAT type distribution by continent and providers in Figure 3.12 was made by Krisztián Téglás.

The above-presented smartphone trace was collected by the STUNNER Android app that was a joint development. The contribution of the author includes the implementation of the latest release of STUNNER with its updated data collection methodology. The initial development of real P2P connection attempts was carried out by Krisztián Téglás. The release management of STUNNER and the description of NAT measurement module were both done by Zoltán Richárd Jánki. The interpretation of the anomalies in the measurements; implementation of a preceding release; and an exploration of the related work was done by Zoltán Szabó. His work includes a summary of NAT studies presented in Table 3.1.

CHAPTER 4

Dimension Reduction Methods

Our research targets networked systems where each networked device stores only a small amount of data (typically collected locally), while there are possibly millions of participating devices in the network. This model covers a wide range of applications including smart metering [89], collaborative mobile platforms [85] and Internet of Things platforms [106]. To implement machine learning algorithms in such a system model, we opted for gossip learning algorithms [82]. There are, of course, many ways to perform data mining over data originating from a fully distributed environment [94], so one has to make a number of design choices. We opted for gossip learning [82] due to its natural emphasis on privacy and full decentralization, while being efficient enough for most data mining tasks. Privacy is achieved in part by assuming that devices do not share raw data either with each other or with any external party. In addition, theoretical notions of privacy such as differential privacy can be incorporated into this framework as well [28, 82, 96]. In-place data processing could also provide better scalability for certain tasks in the limit of extremely large systems compared to cloud-based solutions by exploiting local resources and networks, as proposed e.g. by Cisco in its ongoing fog computing initiative [16].

However, one key concern in the above scenario is communication complexity, which is often proportional to the size of the raw data. Raw data might be very high dimensional, such as images from a surveillance camera, or text documents containing private

communications. It is essential to compress these data locally using a shared method. Algorithms that compress raw data while preserving its information content are called dimension reduction methods [40]. Our goal in this study is to propose practical, distributed dimension reduction algorithms for gossip learning that are efficient yet perform well in given learning tasks.

We propose dimension reduction methods that are compatible with the gossip learning framework. Our first algorithm is based on evolving random projections through a distributed selection process. The second method is a fully distributed implementation of the singular value decomposition (SVD) algorithm. The third method is a hybrid algorithm based on random projection selection and SVD. Our contributions are the following:

- We propose a method for dimension reduction based on selecting good random projections using an algorithm compatible with the decentralized system model.
- We propose a gossip-based fully distributed robust SVD algorithm to approximate the projection matrix of Principal Component Analysis (PCA).
- We propose a hybrid algorithm based on SVD and random projections that combines the advantages of both pure algorithms.
- We perform an extensive empirical analysis of these algorithms using real smartphone traces over several learning tasks.

4.1 Related Works

Unlike our methods, most known methods for distributed dimension reduction are unsuitable for gossip learning. Often it is assumed that there is an aggregator node that processes and aggregates output from all the networked nodes [71]. It is also usually assumed that all the nodes have sufficient data to produce meaningful partial results to be aggregated. These assumptions violate both our system model (where there is very little data at each node) and our objective of decentralization. Methods in this class include several feature selection methods that have been implemented in the MapReduce framework [59]. Landmark-based methods also have distributed variants [72]. Here, extremal points are selected from the raw data that are used to encode a lower dimensional distance preserving representation. Many methods seek to find an optimal linear mapping based on spectral properties of the data such as Principal Component Analysis (PCA) [53].

Calculating SVD is a well-studied problem. One can define a raw data matrix A of dimensions $n \times d$ where n in number of data and d is the number of the features. The essence of the matrix A is computed by finding a low-rank decomposition $A \approx XY^T$, where matrices X and Y^T are of dimension $n \times k$ and $k \times d$, respectively, and where k is a free parameter. The X matrix is the compressed representation of A in a k-dimensional feature space. Y can be interpreted as a projection matrix. SVD is a special low-rank decomposition that exists for any fully defined matrix A with the attractive property that the decomposition consists of orthogonal matrices. One approach is based on treating it as an optimization problem (see Section 4.2) and using *gradient search* to solve it [37]. Guan et al. also follow this approach adapted for the non-negative case and propose an efficient gradient algorithm [38].

In general, parallel versions of gradient search often assume the MapReduce framework [19] or a similar, less constrained, but still centralized model [62]. In these approaches, partial gradients are calculated over batches of data in parallel, and these are either applied to blocks of X and Y in the map phase or summed up in the reduce phase. Zinkevich et al. propose a different approach in which SGD is applied on batches of data and the resulting models are then combined [112]. Petroni et al. propose performance optimizations based on graph partitioning in a similar framework [86]. Gemulla et al. [35] propose an efficient parallel technique in which blocks of X and Y are iteratively updated while only blocks of Y are exchanged. In contrast to these approaches, we work with fully distributed data: we do not wish to make X public and we do not rely on central components or synchronization—a set of requirements ruling out the direct application of earlier approaches.

Another possibility is using fully distributed *iterative methods*. GraphLab [68] supports iterative methods for various problems including SVD. In these approaches, the communication graph in the network is defined by the non-zero elements of matrix A; in other words, A is stored as edge-weights in the network. This is feasible only if A is (very) sparse and well balanced; a constraint rarely met in practice. In addition, iterative methods need access to A^T as well, which violates our constraint that the rows of A are not shared. Using the same edge-weight representation of A, one can implement another optimization approach for matrix decomposition: an iterative optimization of subproblems over overlapping local subnetworks [66]. The drawback of this approach is, again, that the structure of A defines the communication network and access to A^T is required.

The approach of Ling et al. [67] also needs global information: in each iteration step, a global average needs to be calculated.

The first fully distributed algorithm for spectral analysis was given in [55] where data is kept at the compute nodes and partial results are sent through the network. This algorithm, however, does not compute the whole projection matrix and hence is insufficient to provide dimension reduction. Similarly, the algorithm described in [57] computes the low-rank approximation but not the decomposition. This drawback is also circumvented in [48]. Our approach is based on a stochastic gradient algorithm similar to the online algorithm presented in [39], which calculates Y under an additional non-negativity constraint with the help of a constant-sized buffer of samples. We, however, work in a different, non-streaming setting: we are given a fixed decentralized database where we wish to calculate X as well. Also, in order to preserve data privacy, only one row of A is processed in each step and X is strictly decentralized, so no buffering can be implemented.

4.2 Background

Here, we summarize the necessary background on the dimension reduction. We will mention the Random Projection, Principal Component Analysis and Singular Value Decomposition (SVD) methods. As a motivation to understanding SVD, we will also take a brief look at the base concept of Low–Rank Decomposition.

4.2.1 Dimension Reduction

Dimension reduction is an important tool of data mining to overcome the problem of the *curse of dimensionality*.

In machine learning we are given a set of *training examples* from an unknown distribution. Here we assume that an example is of the form (x, y), where x is a d dimensional real *feature* vector ($x \in \mathbb{R}^d$) and y is the *class* of the example. In the classification task, we look for a function $f(x; w) : \mathbb{R}^d \to C$ that categorizes any feature vector x into a finite number of classes, where C is the set of classes and w is a parameter vector that we wish to learn. The function $f(x; w)$ is often called the *model* of the data. Parameter w is typically found through some local gradient method that optimizes a loss (or error) function, which characterizes the accuracy of the model over a set of classified training examples.

If the number of features d is high, we might experience the curse of dimensionality, which means that classification algorithms will perform poorly [14]. Formally, the dimension reduction methods are functions that transform the data into a lower dimensional space $\mathbb{R}^k$, where $k \ll d$.

Dimension reduction methods can be classified as *feature selection* and *feature extraction* techniques. A good overview can be found in [40]. Feature selection algorithms try to find a subset of the original features such that the size of this subset is small yet it preserves most of the information. The basic idea is that these algorithms define an evaluation function over subsets of features based on the accuracy of a machine learning method over the given subset to characterize the performance of the selected features. This function then guides a search algorithm. The idea is simple, but a naive implementation would be rather expensive, as the number of possible subsets is exponentially large. Thus, the applied search algorithm is mostly a greedy heuristic. One such common heuristic is *forward feature selection (FFS)*. As the name of this method suggests, the method starts with an empty set and iteratively adds the feature that results in the highest increase in the accuracy of a given machine learning algorithm until the desired dimension k is reached. A wide range of variants and optimizations of this method is known [40].

In feature extraction, the methods project the training examples into a lower dimensional space. This generalizes selecting only a subset of features. Here we focus on linear projection methods, as this is the focus of our study. In the case of linear projection, we wish to find a matrix $P \in \mathbb{R}^{k \times d}$ with k rows and d columns. Thus the projection of a training example x is a simple vector-matrix multiplication Px. Here we consider two feature extraction methods, namely the *Random Projection* [12] and the *Principal Component Analysis (PCA)* [53].

The simpler method is random projection, where the data is transformed by a random matrix to a lower dimension. The Johnson-Lindenstrauss lemma gives a lower and upper bound on the error of the projection, and the random projection technique exploits this result [3, 12].

Principal Component Analysis (PCA) is a method to find those k directions in the d dimensional space that span a subspace where the covariance of the data is maximized. As a side effect, PCA optimally preserves the distances between data points. Now, let us define matrix $A = [a_1, \ldots, a_n]^T \in \mathbb{R}^{n \times d}$ that consists of the training examples (one example in each column). The projection matrix here is given by the matrix of the eigenvectors

that correspond to the first k eigenvalues of the covariance matrix of the training examples A^TA.

In this chapter we will present a distributed algorithm for calculating the singular value decomposition (SVD) of A to get the projection matrix of PCA. We first assume that the database is such that all the dimensions (features) have zero mean. When using PCA, one first computes the covariance matrix of the data $A^TA \in \mathbb{R}^{d\times d}$, which is a symmetric, positive semidefinite matrix. Diagonalizing the covariance matrix

$$A^TA = PDP^T \tag{4.1}$$

gives the matrix P. The projection matrix of PCA is given by keeping the first k rows of P^T. Matrix P can be calculated using the SVD of A, where

$$A = U\Sigma V^T. \tag{4.2}$$

By substituting this into the covariance matrix, we get

$$A^TA = (U\Sigma V^T)^T(U\Sigma V^T). \tag{4.3}$$

Since U is orthonormalized, we have

$$A^TA = V\Sigma^2V^T. \tag{4.4}$$

Letting $D = \Sigma^2$ we have $P = V$.

As a final note, consider the fact that both random projection and PCA are independent of the machine learning algorithm we wish to apply on the data, while feature selection methods heavily depend on it and can thus be optimized for a given algorithm. At the same time, feature selection represents only a very limited subset of projections, which limits its ability to find an optimal dimension reduction.

4.2.2 Low–Rank and Singular Value Decomposition

The rank k matrix approximation problem is defined as follows. We are given a matrix $A \in \mathbb{R}^{n\times d}$. We are looking for matrices $X \in \mathbb{R}^{n\times k}$ and $Y \in \mathbb{R}^{d\times k}$ such that the error function

$$J(X, Y) = \frac{1}{2}\|A-XY^T\|_F^2 = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{d}(a_{ij} - \sum_{l=1}^{k} x_{il}y_{jl})^2 \tag{4.5}$$

is minimized. We say that the matrix XY^T that minimizes this function is an optimal rank k approximation of A. Clearly, matrices X and Y^T—and hence XY^T—have a rank of at most k. Normally we select k such that $k \ll \min(n, d)$ in order to achieve a significant compression of the data. As a result, matrices X and Y^T can be thought of as high level features (such as topics of documents or semantic categories for words) that can be used to represent the original raw data in a compact way.

Singular value decomposition (SVD) is closely related to the above matrix decomposition problem. The SVD of a matrix $A \in \mathbb{R}^{n\times d}$ involves two orthogonal matrices $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{d\times d}$ such that

$$A = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T, \tag{4.6}$$

where the columns of the matrices $U = [u_1 u_2 \cdots u_n]$ and $V = [v_1 v_2 \cdots v_d]$ are the left and right singular vectors, and $\Sigma \in \mathbb{R}^{n\times d}$ is a diagonal matrix containing the singular values $\sigma_1, \sigma_2, \ldots, \sigma_r \geq 0$ of A ($r = \min(n, d)$). The relationship between SVD and-low rank decomposition is that $U_k\Sigma_k V_k^T$ is an optimal rank-k approximation of A, where the matrices $U_k \in \mathbb{R}^{n\times k}$ and $V_k \in \mathbb{R}^{d\times k}$ are derived from U and V by keeping the first k columns, and $\Sigma_k \in \mathbb{R}^{k\times k}$ is derived from Σ by keeping the top left $k \times k$ rectangular area, assuming without loss of generality that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ [98]. In order to unify the notation, we use an alternate definition of SVD as a matrix factorization $A \approx X^* Y^{*T}$ with

$$X^* = U_k\Sigma_U, \quad Y^* = V_k\Sigma_V, \tag{4.7}$$

such that Σ_U and Σ_V are diagonal and $\Sigma_U\Sigma_V = \Sigma_k$. In other words, although X^* and Y^* are not uniquely defined, we require that they contain orthogonal columns that are scaled versions of left and right singular vectors of A, respectively.

4.3 Algorithms

Here, we present three algorithms for fully distributed dimension reduction. The first one is based on the idea of generating random projections, evaluating them by quickly training a model using only a few updates, and selecting the projection that results in the best preliminary model.

The second one is a fully distributed variant of SVD that keeps X and A strictly local. As a special feature, our algorithm updates several versions of Y by sending them around the network.

The third is a hybrid algorithm that combines SVD and random projections. This way we have a very quick convergence, inherited from the random projection method, and high quality, inherited from SVD.

4.3.1 Random Projection Selection

Random projections are very cheap to generate, and a random matrix can be communicated by sending just the corresponding random seed. Due to this, we have the possibility of evaluating a lot of different random matrices and searching for the best one for the problem at hand. Given a fixed machine learning problem (data and a learning algorithm), our idea is to evaluate a random projection based on the accuracy of the outcome of the learning algorithm, provided that the given random projection is used as the dimension reduction algorithm.

To evaluate a random projection, a naive approach would be to fully train a model based on the given random projection and then fully evaluate this model on a test set. Instead of this, we train the model using gossip learning and before each update step we evaluate it on the local training example (using it as a test example). We then use the running average of these evaluations as a rough approximation of the true performance of the model. Based on this evaluation method, we would like to find the best random matrices that can be generated for the machine learning problem at hand.

The skeleton of our algorithm to be run on each node is shown in Algorithm 4.2. The algorithm resembles the periodic gossip learning algorithm (Algorithm 2.1), but there is an important difference. Instead of being periodic, the algorithm implements the random walks in a "hot potato" style. We shall now describe the algorithm in more detail.

Local state and initialization

Each node is initialized when joining the network. This consists of setting up the local variables and potentially starting a random walk of a new model to initiate gossip learning.

The local data includes the training example (x, y) and models under training. Since we would like to find the best possible projection (that is, the one that results in the best model) and we also wish to keep exploring new projections, every node needs to keep track of two different models:

- The BESTMODEL: the best model known by the node
- The CURRENTMODEL: the model currently under evaluation (as in gossip learning)

Algorithm 4.2 Random projection selection at node i

```
1:  procedure INITNODE
2:      (x, y) ← local training example
3:      initModel(currentModel)
4:      initModel(bestModel)
5:      if selectedAtRandom(π) then
6:          p ← selectPeer()
7:          send currentModel and bestModel to p
8:      end if
9:  end procedure
10: procedure ONRECEIVEMODELS(m_c, m_b)
11:     currentModel ← m_c
12:     update(currentModel)
13:     bestModel ← getBetter(bestModel,m_b)
14:     update(bestModel)
15:     if currentModel.age ≥ minAge then
16:         bestModel ← getBetter(bestModel,currentModel)
17:         initModel(currentModel)
18:     end if
19:     p ← selectPeer()
20:     send currentModel and bestModel to p
21: end procedure
22: procedure UPDATE(m)
23:     R ← createSparseRandomMatrix(k, d, m.seed)
24:     x_red ← Rx
25:     ŷ ← predict(m.model, x_red)
26:     m.error ← (1 − λ)m.error + λ|y − ŷ|
27:     m.model ← updateSGD(m.model, x_red, y)
28:     m.age ← m.age + 1
29: end procedure
30: procedure INITMODEL(m)
31:     m.age ← 0
32:     m.error ← 0
33:     m.seed ← getNextRandomSeed()
34:     m.model ← initSGD()
35: end procedure
```

Each model stores the random seed that is used to generate the random projection that in turn is used to compress the local data x. In addition, the model stores the machine

learning model itself, which is being trained via gossip learning. We also keep track of the number of updates (age) and the accumulating error that is used to assess the performance of the model, and hence the performance of the random projection.

Every node initially picks a random seed for a new projection matrix at random. Based on the seed, the projection matrix $R \in \mathbb{R}^{k \times d}$ is always generated in line 23 as follows: for any index (i, j) let

$$r_{ij} = \sqrt{3} \cdot \begin{cases} +1 & \text{with probability } 1/6 \\ 0 & \text{with probability } 2/3 \\ -1 & \text{with probability } 1/6. \end{cases} \tag{4.8}$$

With this choice, the matrices are sparse but still satisfy the Johnson-Lindenstrauss lemma [3, 12]. Obviously, the same pseudo random generator should be used at each node so that the same projection matrix is generated for the same seed.

Finally, with probability π the node initiates a random walk. The probability defines the number of overall random walks in the network, which is πn in expectation. This defines, among other properties, the overall bandwidth utilization.

Updating the models

In the method called UPDATE, first dimension reduction is performed on the local example that creates vector x_{red} of length k. Next, before performing the model update step, we update the running average of the prediction error using the local example. This is used to approximate the prediction performance of the model (and thus the random projection). The running average has a parameter λ that determines the balance of old and new information. If λ is too large then only the most recent examples will count and the measure will contain too much noise. If it is too low then the old examples will count too much, and the improvement in time is not reflected sufficiently.

We then perform the model update according to gossip learning; that is, we apply a stochastic gradient descent (SGD) update step using the local data. The implementation of the SGD update depends on the learning algorithm of choice. In our experimental evaluation, we will use logistic regression as our learning algorithm.

Processing received models

Models arrive in pairs. Model m_b is the sending node's approximation of the best model. This model participates in a gossip-based minimum search; that is, we compare it with

the local approximation of the best model and keep the better one of the two. The method called GETBETTER selects the model with the smaller error, or with the higher age in case at least one of the models is younger than MINAGE steps. Model m_c participates only in the gossip learning algorithm. When it reaches the age (number of updates) of MINAGE (a parameter), it becomes eligible for competing for the title of best model in the network. Since CURRENTMODEL is now participating in the global minimum search, we start a new model to test. This is our exploration strategy: new random projections are tested continuously while the mature ones contribute to the global minimum search.

Before reaching the age of MINAGE the error approximation of a model is considered to be too unreliable. That is, the value of parameter MINAGE will determine the number of updates needed to get the first reliable error approximation. Note that our goal is to have an error approximation that is suitable for ordering different candidates as opposed to approximating the exact error. This means that after MINAGE updates the error might be quite different from the exact error, so MINAGE could in principle be rather small. We will take a closer look at this issue in Section 4.4.

4.3.2 Singular Value Decomposition

Our SVD algorithm also has its roots in the GoLF framework [82]. Algorithm 4.3 contains a version of the GoLF algorithm adapted to our problem. Each node i has its own approximation of the full matrix Y^* and an approximation x_i of row i of X^*. Thus, the nodes collectively store one version of the matrix X (the approximation of X^*) distributed just like matrix A with each node storing one row. Note that at every point in time, every node has its local approximation of the full matrix Y^* that may differ across the nodes. As we will see, these approximations interact through the single approximation of X^* and should converge to the same matrix Y^*. The output of the algorithm at any point in time is readily available as the local approximation at each node, so there is no need to explicitly combine the different approximations centrally.

All local approximate versions of Y^* perform a random walk in the network and get updated by the local data (a row of A and X) when visiting a node. First, each node i in the network initializes Y and x_i uniformly at random from the interval $[0, 1]$. The nodes then periodically send all the approximations of Y they received in the previous round to a randomly selected peer from the network. To select a random peer, we rely on a peer sampling service, as mentioned in Section 4.2. When receiving an approximation $\tilde{Y}$ (see

Algorithm 4.3 P2P low–rank factorization at node i

```
1: a_i                                                    ▷ row i of A
2: initialize Y
3: receivedY.add(Y)                                       ▷ initialize receivedY
4: initialize x_i                                         ▷ row i of X
5: quiet ← 0                                              ▷ rounds without receiving models
6: loop
7:     wait(Δ)
8:     if receivedY.isEmpty() then
9:         quiet ← quiet+1
10:        if quiet ≥ Δ_q then
11:            p ← selectPeer()
12:            send Y to p
13:        end if
14:    else
15:        quiet ← 0
16:        repeat                                         ▷ send all the received models
17:            Ỹ ← receivedY.remove()
18:            p ← selectPeer()
19:            send Ỹ to p
20:        until receivedY.isEmpty()
21:    end if
22: end loop
23: procedure onReceiveY(Ỹ)
24:    (Y, x_i) ← update(Ỹ, x_i, a_i)
25:    receivedY.add(Y)
26: end procedure
```

procedure onReceiveY) the node updates both Y and x_i using a stochastic gradient rule and it subsequently stores this approximation in a list to be forwarded in the next round.

The algorithm involves periodic message sending from every node with a period of Δ. Later on, when we refer to one *iteration* or *round* of the algorithm, we simply mean a time interval of length Δ. Note that we do not require any synchronization of rounds over the network. Messages are sent independently. To avoid the termination of random walks due to failures, the algorithm includes a restarting mechanism based on the length of time during which no models are received. The timeout for this restart is $\Delta_q \cdot \Delta$. Unless otherwise stated, we use $\Delta_q = 10$. Note that if peer sampling is uniform, the distribution of the number of received models in a round is Poisson(1), which means that not receiving models for 10 rounds has a probability of $e^{-10} \approx 4.5 \cdot 10^{-5}$.

Algorithm 4.4 rank-k update at node i

1: η ▷ learning rate
2: **procedure** UPDATE(Y, x_i, a_i)
3: $\quad$ err $\leftarrow a_i - x_i Y^T$
4: $\quad$ $x_i' \leftarrow x_i + \eta \cdot \text{err} \cdot Y$
5: $\quad$ $Y' \leftarrow Y + \eta \cdot \text{err}^T \cdot x_i$
6: $\quad$ **return** (Y', x_i')
7: **end procedure**

Algorithm 4.3 requires an implementation of the procedure UPDATE that computes the new approximations of Y^* and x_i. We will now elaborate on two different versions that implement different stochastic gradient update rules.

Update Rules for General Rank–k Factorization

Let us first consider the error function given in Equation (4.5) and derive an update rule to optimize this error function. The gradients of $J(X, Y)$ by X and Y based on Equation (4.5) are

$$\nabla_X J(X, Y) = (XY^T - A)Y, \quad \nabla_Y J(X, Y) = (YX^T - A^T)X. \tag{4.9}$$

Since only x_i is available at node i, the gradient is calculated only w.r.t. x_i instead of X. Accordingly, the stochastic gradient update rule with a learning rate η can be derived by substituting x_i in the way shown in Algorithm 4.4. Although function J is not convex, it has been shown that all the local optima of J are also global [98]. Thus, for a small enough η, any stable fix point of the dynamical system implemented by Algorithm 4.3 with the update rule in Algorithm 4.4 is guaranteed to be a global optimum.

Update Rule for Rank–k SVD

Apart from minimizing the error function given in Equation (4.5), let us now also set the additional goal that the algorithm converges to the SVD in the form of X^* and Y^*, as defined by Equation (4.7). This is indeed a harder problem: while (X^*, Y^*) minimizes (4.5), any other pair of matrices (X^*R^{-1}, Y^*R^T) will also minimize for any invertible matrix $R \in \mathbb{R}^{k\times k}$, and Algorithm 4.4 is free to converge to any of them.

From now on, we will assume that the non-zero singular values of A are all unique, and that the rank of A is at least k; that is, $\sigma_1 > \cdots > \sigma_k > 0$. This makes the discussion simpler, but these assumptions can be relaxed, and the algorithm is applicable even if

Algorithm 4.5 rank-k SVD update at node i

η ▷ learning rate
procedure UPDATE(Y, x_i, a_i)
  $a_i' \leftarrow a_i$
  for $\ell = 1$ to k **do** ▷ y_ℓ : column ℓ of Y
    $\text{err} \leftarrow a_i' - x_{i\ell} \cdot y_\ell^T$
    $x_{i\ell}' \leftarrow x_{i\ell} + \eta \cdot \text{err} \cdot y_\ell$
    $y_\ell' \leftarrow y_\ell + \eta \cdot \text{err}^T \cdot x_{i\ell}$
    $a_i' = a_i' - x_{i\ell} \cdot y_\ell^T$
  end for
  return (Y', x_i')
end procedure

these assumptions do not hold.

Our key observation is that any optimal rank-1 approximation $X_1 Y_1^T$ of A is such that $X_1 \in \mathbb{R}^{n\times 1}$ contains the (unnormalized) left singular vector of A that belongs to σ_1, the largest singular value of A. Similarly, Y_1 contains the corresponding right singular vector. This is because for any optimal rank-k approximation XY^T, there is an invertible matrix $R \in \mathbb{R}^{k\times k}$ such that $X = X^*R$ and $Y^T = R^{-1}Y^{*T}$[98]. For $k = 1$ this proves our observation because, as defined in Section 4.2, $X^* \sim u_1$ and $Y^* \sim v_1$. Furthermore, for $k = 1$,

$$X_1 Y_1^T = X^* Y^{*T} = \sigma_1 u_1 v_1^T, \tag{4.10}$$

which means that (using Equation (4.6)) we have

$$A - X_1 Y_1^T = \sum_{i=2}^{r} \sigma_i u_i v_i^T. \tag{4.11}$$

Thus, a rank-1 approximation of the matrix $A - X_1 Y_1^T$ will reveal the direction of the singular vectors corresponding to the second largest singular value σ_2. A simple approach based on these observations is to first compute $X_1 Y_1^T$, a rank-1 approximation of A. Subsequently, we compute a rank-1 approximation $X_2 Y_2^T$ of $A - X_1 Y_1^T$. The rank-2 approximation of A containing the first two left and right singular vectors is obtained as $[X_1, X_2][Y_1, Y_2]^T$ according to the above observations. We then repeat this procedure k times to get the desired decomposition X^* and Y^* by filling in one column at a time sequentially in both matrices.

A more efficient and robust approach is to let all rank-1 approximations in this se-

quential naive approach evolve at the same time. Intuitively, when there is a reasonable estimate of the singular vector corresponding to the largest singular value, the next vector can already start progressing in the right direction, and so on. This idea is implemented in Algorithm 4.5.

4.3.3 Communication complexity

As for the size of a single message in the Random Projection protocol, the nodes send two models in a message that contains only the seed of the random projection matrix, and the model parameters, which—as we will see—have a size of $O(k)$ in the case of linear learning algorithms. To be more precise, assuming an 8 byte representation of both the floating point and integer parameters, we have a message size of $2 \cdot (8 \cdot k + 3 \cdot 8)$. This leads to a very small, by today's standards practically negligible, message size since k is typically small. In contrast to this, in the SVD protocol the whole projection matrix of size $O(k \cdot d)$ needs to be sent in each message [47]. This can be very large since d is potentially in the order of millions.

In both protocols the overall communication complexity can be controlled by the number of random walks πn in the network that determines the overall bandwidth consumption. Thus, given any bandwidth quota, these protocols can be executed within that quota by setting the number of random walks. Note that, due to the extremely different message sizes, this means that with the same bandwidth quota, the two algorithms will have a very different iteration speed.

4.3.4 A Hybrid Algorithm

The motivation is that the SVD algorithm requires a messages size of $O(k \cdot d)$ that can be very large for a large d. Thus, with a fixed bandwidth, the SVD algorithm converges much slower than random projection selection. At the same time, the SVD algorithm provides a very high quality projection that is expected to outperform random projections. Our goal is to combine the advantages of the two algorithms; that is, with a fixed bandwidth we would like to get a good projection at any point in time that eventually arrives at the SVD output.

Now, let R_t be the best random projection at time t and P_t be the projection calculated by the SVD algorithm at the same time. Another detail of the SVD algorithm is that the

rows of the projection matrix converge in a sequential order, and in the converged state the rows are pairwise orthogonal. Our basic idea is that we run the two algorithms in parallel and define a projection $\hat{P}_t(R_t, P_t)$ that will use the converged rows from P_t and it fills in the rest of the rows from R_t.

Hence, all we need is a method to determine the row index up to which P_t is considered to have converged. We first introduce a measure of orthogonality o_i for a given row p_i^T:

$$o_i = \frac{1}{k-1} \sum_{j \neq i} \frac{p_i^T \cdot p_j^T}{\|p_i^T\| \|p_j^T\|}, \tag{4.12}$$

which is the average cosine distance of p_i^T from the rest of the rows. We include in $\hat{P}_t$ the first i rows of P_t where i is such that $o_i \leq$ minOrt but $o_{i+1} >$ minOrt (or $i = k$). Here, minOrt is a threshold parameter that defines how aggressively we include SVD vectors. A value close to zero is conservative, while a smaller value is more aggressive.

Although we propose to run the two algorithms in parallel, we do not assign the same bandwidth quota to each. Instead, we allow the SVD algorithm to consume almost all the bandwidth quota, and assign only 1% of it to random projection selection.

4.4 Experimental Results

4.4.1 Experimental Setup

Here, our goal is to demonstrate that the proposed methods can be applied over real-world datasets under realistic network conditions. The key parameter in dimension reduction is k; that is, the reduced dimensionality. Our motivation is to demonstrate empirically how our distributed algorithms perform with different values of k, so here k is our main free parameter.

Trace Properties

We simulate node churn based on a real trace of smartphone user behavior. We also ensure that we use the phone only when it is connected to a charger, so as to save the battery. The trace we used was collected by a locally developed openly available smartphone app called STUNner, as described previously in Section 3.3. We divided these traces into

Table 4.1. The key properties of the data sets

	MNIST	Farm ads	HAR
Training set size	60 000	3 314	7 352
Test set size	10 000	829	2947
Number of features	784	54 877	561
Original number of classes	10	2	6
Positive examples	10%	53%	17%

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. When we needed more users than segments (e.g. 60,000 user), we re-sampled the segments to artificially inflate the number of users.

Data sets

In our experiments we used data sets taken from various machine-learning domains with different properties. Our first data set, called MNIST [64], contains gray level images of size 28×28 of handwritten digits (from 0 to 9).

The remaining two data sets are part of the UCI machine-learning repository [32]. The second data set we chose was the Farm ads data set. This is a text classification data set that has a large number of features. These features include those extracted by the well-known bag-of-words technique from websites as well as higher level features. The task is to decide whether the owner of a Web content approves an ad or not.

Finally, we used the Human Activity Recognition (HAR) data set as well. Here the features were preprocessed by the owner of the data. The goal is to discriminate different activities based on the data of smart phone sensors (e.g., accelerometer and gyroscope).

In our experiments, we performed binary classification. For this reason, we had to transform the MNIST and HAR data sets to binary classification problems. To achieve this, we selected the classes “number 7” and “walking” as positive classes from the MNIST and HAR data sets, respectively, and the examples in the remaining classes were treated as negative examples. The key properties of these three data sets are summarized in Table 4.1.

Table 4.2. Parameter settings

Alg.		MNIST	Farm ads	HAR
RPSVD	minOrt	0.5	0.5	0.5
RP	λ	0.05	0.05	0.05
	minAge	200	200	200
	message size (Mbit, $k = 1$)	0.0003	0.0003	0.0003
	message size (Mbit, $k = 64$)	0.0084	0.0084	0.0084
SVD	α	10^{-4}	10^{-2}	10^{-2}
	message size (Mbit, $k = 1$)	0.05	3.51	0.03
	message size (Mbit, $k = 64$)	3.21	224.77	2.29
Log. Reg.	α	10^{-2}	10^{-2}	10^{-2}

Algorithm parameters

The algorithm parameters are summarized in Table 4.2.

From now on, RPSVD will denote the hybrid algorithm and RP will denote random projection selection. Logistic regression is used inside RP as the gossip learning component. In logistic regression, the model is given by

$$f(x; w, b) = \frac{1}{1 + e^{w^T x + b}}. \tag{4.13}$$

The corresponding stochastic gradient update rule is given by

$$\begin{aligned} w &\leftarrow (1 - \gamma\alpha)w - \gamma(y - f(x; w, b))x, \\ b &\leftarrow b - \alpha(y - f(x; w, b)), \end{aligned} \tag{4.14}$$

where $y \in \{0, 1\}$, and $\gamma \leftarrow 1/(1 + \alpha t)$. Based on preliminary experiments we set $\alpha = 0.01$.

The communication cost of each protocol was limited so that a node consumes 1 Mbit/s on average. This was achieved by limiting bandwidth in the simulation. In the case of RPSVD we set $\pi = 0.01$ in the RP component and everywhere else we set $\pi = 1$.

Recall that we use only those nodes that are on a charger, and that we include only those nodes that have a larger bandwidth than 1 Mbit/s. As a result, a node typically experiences much less load when averaged over the simulation period unless it is on a

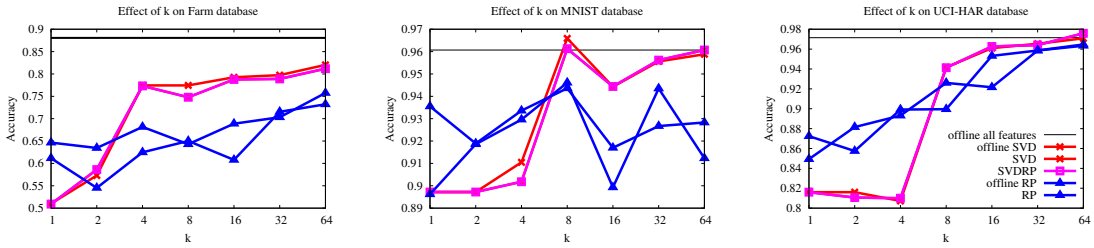


Figure 4.1. Accuracy after two days of simulated time as a function of k.

charger as well as online all the time. That is, based on the real trace we use, this is a practical setting.

Apart from bandwidth, message latency also limits the speed of the random walks. When the message size is very small (as in the case of RP), this can be the main limiting factor, so the RP algorithms will use much less bandwidth than allowed. We simulated a 100 ms latency. This means that in RP algorithms it takes as little as 20 seconds to evaluate a random projection candidate (assuming MINAGE=200), as the computation time is practically negligible.

The peer sampling service is assumed to be based on a static network, in which every node has 50 random neighbors from the whole population. This is a realistic setup on the real Internet, since in the case of stable connections one needs to perform NAT traversal only once, potentially with the assistance of a server in the connection phase [91]. Note that most neighbors are offline at any point in time (see Figure 3.6).

4.4.2 Discussion

Figure 4.1 shows the prediction accuracy of our three distributed algorithms at the end of the simulated two days. Accuracy is the fraction of correctly classified instances:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n}\delta(y_i = f(x_i; w)), \tag{4.15}$$

where n is the number of test examples. To evaluate our distributed protocols, we chose a uniform random online node from the network and calculated its current dimension reduction offline. This was done by taking the dimension reduction matrix, transforming the training set and then training using R (the OPTIM gradient solver), and calculating the

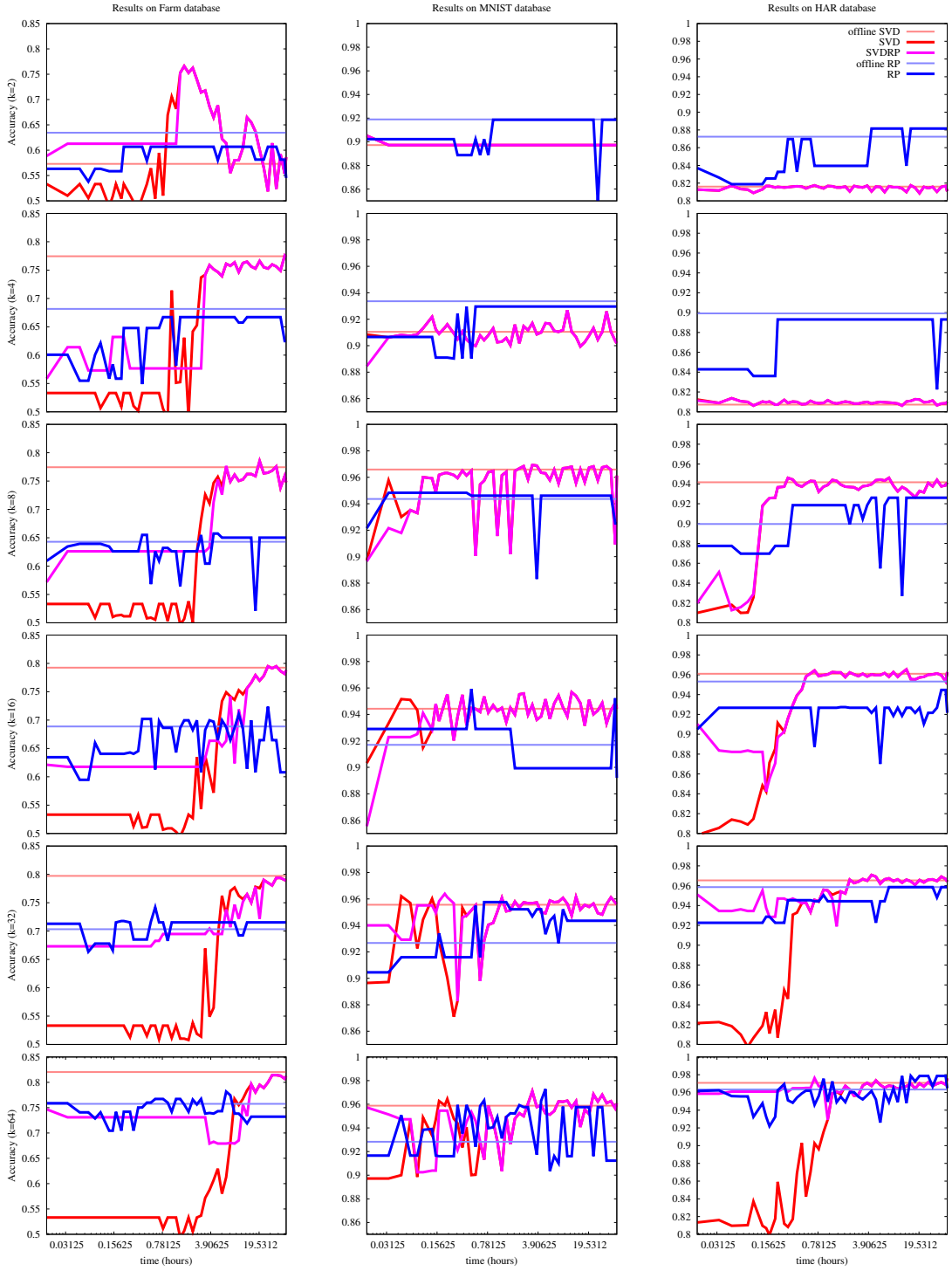


Figure 4.2. Experimental results showing the prediction accuracy as it evolves in time (time is on a logarithmic scale).

accuracy on the test set. We opted for this methodology because it was not computationally feasible for us to report statistics at every measurement point. However, variance is illustrated by the smoothness of the curves. The algorithms are compared to offline (centralized) variants. The offline SVD was calculated using R (the SVD function) and the offline RP is the best of 10,000 random projections, where the evaluation during the selection process is identical to that of the distributed version. We also include the accuracy based on training a logistic regression model that uses all the features.

We may conclude that the distributed algorithms approximate the offline variants very well at the end of the second day. As we increase k, we can approximate the performance of the full feature set, except in the highest dimensional data set. It is also clear that the SVD-based dimension reduction method outperforms random projections for larger k-s. Clearly, the hybrid method, by design, is identical to the SVD algorithm in its converged state.

Figure 4.2 shows our results as a function of time. Now, we plot the accuracy of the distributed methods for each minute, following the same evaluation methodology as before.

It is clear that, for SVD, the most important parameter is d, the original number of features. In the Farm Ads data set we have the largest number of features and thus SVD converges rather late (note the logarithmic scale of the plots). This is because the message size (hence the number of iterations) depends on d. However, RP converges almost instantly, independently of d. This is not surprising as the communication complexity of RP is independent of d and the message size is very small, allowing for a large number of iterations in a very short time. However, the best performance of RP remains below that of SVD, especially for larger values of k.

The hybrid approach SVDRP combines the advantages of the two methods, and provides a good dimension reduction transformation at all timescales, at the same cost as any of the individual algorithms. The significance of this finding is that SVDRP is a method that is more robust than either SVD or RP alone and it can be applied with minimal knowledge to the problem at hand without parameter tuning for a certain wall-clock-time budget. The only exceptions are the smallest values of k, where RP is better on its own. However, in practice, one rarely uses such an extreme dimension reduction.

4.5 Conclusions

In this study we presented a fully distributed algorithm for selecting a good random projection and a gossip-based fully distributed robust SVD algorithm that can be used to reduce the dimensionality of a machine learning problem. We also proposed a hybrid approach which combines random projection selection with a fully distributed SVD solver. We evaluated these algorithms over a real smartphone trace over three machine-learning data sets. The simulations assumed that we use only phones that are on a charger and that have a bandwidth of at least 1 Mbit/s, hence we took into account the energy problem in mobile computing.

We conclude that the proposed random projection selection algorithm is very fast and efficient, but the quality of the dimension reduction is somewhat lower than that of SVD. However, the SVD algorithm converges in a time proportional to the original dimensionality of the problem, which can be quite slow. Our hybrid approach combines the best aspects of the two approaches and (assuming the same communication cost) it can provide a good quality dimension reduction, independently of the time available for convergence.

Contribution

In this chapter, the contributions of the author were the development of an algorithm that builds on searching for good random projections; the development of a hybrid method that combines the advantages of random projections and SVD; and the evaluation of the distributed SVD on dimension reduction in a comparative study. The core idea of the distributed low rank matrix decomposition and the distributed SVD was developed by István Hegedűs [43].

CHAPTER 5

Management of Random Walks

In large decentralized systems, random walks have found many applications. One example, gossip learning [82], is a decentralized approach to machine learning that is based on stochastic gradient descent (SGD) search. Here, the model that is being fit on the data performs a uniform random walk over the network and it is updated before each step using the local data. Recently, the same idea has been applied to matrix factorization and this is useful, for instance, in implementing a decentralized recommender system [43]. It should be mentioned here that—although our motivation is gossip learning for distributed environments—the random walk management middleware services are potentially useful in a more general context as well as random walks have been applied to many different functions other than machine learning. In early peer-to-peer systems they were proposed to implement a search [69], and since then random walk techniques have been applied to various functions that include a membership service [8, 74, 91], sampling [60], and computing various queries including community detection in large networks [87] and network size [74].

Despite the large number of applications, we are not aware of related work where random walks are treated as a general abstraction that is implemented in a robust manner. Such an abstraction should provide the illusion of a reliable single random walk while

in the background it should manage the walk by possibly restarting or replicating it to cope with node and communication failures. Self-stabilizing leader election algorithms are perhaps the closest in that they provide the abstraction of a single entity despite faults and dynamism in arbitrary networks (see, for instance, [27]). However, our problem, our system model and our priorities will be quite different.

A reliable random walk abstraction is very useful, especially when only a small number of important random walks need to be run carrying valuable states. If we also take into account techniques for privacy preservation, such as differential privacy [31] that we applied to our random walk based machine learning framework [46], this requirement becomes crucial, since with differential privacy every data item can be visited only a limited number of times so losing a walk can cause irrecoverable damage.

Our other long-term research goal is to allow gossip learning to be deployed in a multi-user decentralized environment where users or software agents can launch learning tasks over the collection of the local data of the participants of the network. We are interested in multi-user environments as our goal is to create a fully open collaborative environment where those who provide data can also enjoy the benefits of mining the collective data of the community. The notion of decentralization is also important, as has been recognized by other researchers as well. One reason is that distributed computing allows better scalability compared to cloud-based solutions by exploiting local resources and networks, as proposed e.g. by Cisco in its ongoing fog computing initiative [16]. Another reason is the increasing need for privacy as the personal data collected and stored by ubiquitous personal computing devices such as smart meters, sensors and mobile devices is becoming richer and richer [24].

The requirement of differential privacy that every data item can be visited only a limited number of times can be satisfied by maintaining a very limited number of random walks. Hence, our contribution for this goal is the Single Random Walk Service, which was presented in Section 5.1. In contrast, an entirely different aspect is when we have to deal with $O(n)$ random walks to realize a multi-user decentralized environment. Hence, we introduce the Multiple Random Walk Service in Section 5.2 to tackle this other problem.

5.1 The Single Random Walk Service

Our contribution is an approach to implement a reliable random walk abstraction. We identify three requirements for this abstraction. First, the implemented random walk has to be *agile*; that is, it should progress as quickly as possible. Second, the implementation should be *efficient*, so it should induce only a minimal extra cost to achieve robustness. For example, maintaining several replicated walks is not acceptable; ideally, the cost should be very close to that of running a single walk in a reliable system. Third, the random walk should be *long-lived*; that is, it should perform as many steps as possible without resetting its state.

Our solution relies on maintaining a very small shared state through gossip that describes the progress of the random walk. Based on this shared state, each node decides independently whether to restart the random walk. Nodes that store a more recent state restart the walk with a higher probability. The approach is fully decentralized and only incurs a relatively small overhead if the random walk has a large state. Motivated by the fact that our machine-learning applications are intended for networks of smart devices, we demonstrate the main properties of our solution using a real smartphone trace that we collected.

Our present study is based on gossip learning in the sense that we focus on SGD algorithms that are implemented through a random walk of the evolving model over the network. We will assume that this random walk itself is secure. Ideas for achieving secure random walks were recently outlined in [13] and elsewhere. Here, we focus on privacy. In order to achieve privacy, we will apply a differentially private variant of the local update step, as explained below.

The network nodes hold one training example (x, y) and they calculate the local gradient for a given model w and time t locally, and they also add the appropriate noise term N_t to achieve differential privacy based on Equation (5.3). They are free to publish the resulting w_{t+1} and to send it to the next node. Here, the parameter ϵ of differential privacy is a globally known constant.

5.1.1 Background on Differentially Private SGD

Differential privacy [28] is concerned with the leakage of personal information due to publication of the results of a given query over a database. Even if performed securely, the result of a query can leak information about individual records. For instance, the maximum of a set of values is an individual record in itself. Differential privacy is achieved if noise is added to the query result in such a way that the following definition is satisfied.

Definition 1 (Differential Privacy)**.** *A randomized query $F : \mathcal{D} \mapsto \mathbb{R}^d$ is* ϵ-differentially private *iff*

$$\forall x : e^{-\epsilon} \leq \frac{P(F(D) = x)}{P(F(D') = x)} \leq e^{\epsilon} \tag{5.1}$$

for all pairs of databases D and D' that differ in at most one record, where $\mathcal{D}$ is the set of possible databases.

That is, if we change one element in the database, the same output should be expected with a probability close to that over the original database. This way, one record never "matters too much", thereby limiting the information leakage as a result of the query.

A randomized query typically means adding noise to an otherwise deterministic query. This added noise is designed specifically for a given query and parameter ϵ such that the definition of $\epsilon-$differential privacy is satisfied. In more detail, to generate the additive noise we need to pick a noise distribution and the right distribution parameters. A common approach to take is to first determine the so-called *sensitivity* of the query [28, 30]:

Definition 2 (Global Sensitivity)**.** *The* global L^1-sensitivity Z_F *of F is given by*

$$Z_F = \max_{D,D' \text{ differ in one record}} \|F(D) - F(D')\|_1, \tag{5.2}$$

where $\|\cdot\|_1$ is the L^1 norm.

The definition can be generalized by replacing the L^1 norm with a different norm. However, the usual norm to apply is the L^1 norm. In this case, the following noise distribution can be used: we need to add to all the dimensions of the output independent noise drawn from Laplace$(0, Z/\epsilon)$ (where Z is the global sensitivity of the query), which will result in $\epsilon-$differential privacy. Based on the theoretical results described in [30], noise can be generated for any other norms.

Now, for one SGD update (as defined in Equation (2.3)) the private query we need to compute is the gradient $\nabla_w \ell(f_w(x_i), y_i)$. If we can guarantee that this gradient is bounded, the bound defines sensitivity directly. Having determined the sensitivity, we can then add the appropriate noise N_t to the gradient and perform the differentially private local update

$$w_{t+1} = w_t - \eta_t(\lambda w_t + \nabla_w \ell(f_w(x_i), y_i) + N_t). \tag{5.3}$$

We are then free to publish w_{t+1} and send it to the next node.

To run SGD, we require multiple queries because we need the gradients based on many learning examples multiple times. Having seen how one can protect a single update, let us mention two useful concepts from differential privacy; namely the sequential and parallel composition of queries [77].

In a sequential composition we are given a series of queries $F_i,\ i = 1, \ldots, k$. It can be proven that if all of these queries are ϵ-differentially private, then the entire sequence of these k queries will be $k \cdot \epsilon$-differentially private. Note that the queries can depend on the results of the previous queries.

However, in the special case where the k queries are executed over pairwise disjoint subsets $D_i,\ i = 1, \ldots, k$—a case we call parallel composition—the entire sequence of queries will remain ϵ-differentially private. Most importantly, in the case of SGD we have parallel composition, since updates are typically performed on a disjoint subset—in our case on a single record. Naturally the same record can be visited many times, and these updates will compose sequentially.

In general, we can think of each example as having a privacy budget of ϵ, which is spent when the given example is visited but which is not affected otherwise. This way, when each example has spent its privacy budget of ϵ, the entire SGD algorithm over the entire database will spend only ϵ due to parallel composition.

5.1.2 Privacy Budget

The ϵ parameter is often called the privacy budget because, owing to the different compositional properties of series of queries, one can, say, decide to run one query with parameter ϵ or two sequentially composing queries with parameter $\epsilon/2$, or several parallel queries with parameter ϵ. All of these options result in an overall ϵ-differential privacy.

Now, let us elaborate on the management of the privacy budget for SGD.

As mentioned in Section 5.1.1, every training example (i.e., every node) in effect has its own ϵ budget for the updates. This budget can be used in a number of different ways. One can, for instance, set a finite number of k allowed updates and use ϵ/k for each one. This means multiplying the magnitude of the noise term by k for each update. We can also follow a different approach and divide ϵ into an infinite number of parts by using $\epsilon/2^t$ for update t. This way, the noise increases exponentially, but we can execute as many updates as we wish using the same example. Note, however, that SGD will not converge in this case due to the exponentially increasing noise, so this approach is practical only for a small finite number of rounds.

The above implies a deeper result: it is not possible to run SGD until we get convergence with differential privacy because we either compute just a finite number of updates (and SGD needs an unlimited number of updates for theoretical convergence) or the signal-to-noise ratio will tend to zero in the update rule, which also prevents convergence. So the best we can achieve in theory is an approximation based on a relatively small number of updates per sample. For a large number of samples, however, this may be sufficient.

Let us point out a major difference between our differentially private SGD implementation and gossip learning. In our SGD implementation there is only one random walk in the entire network, while in gossip learning there are many parallel walks. However, if there are many walks in parallel, they all "burn" the privacy budget so each walk will be assigned a smaller number of updates that is inversely proportional to the number of walks. It is therefore essential to run only one walk. However, the state of the walk is public, so it is possible to continuously broadcast the latest model w_t in the network if required. The broadcast can be implemented in a distributed way (e.g. via gossip), or by publishing the latest update on a server. With public key cryptography the broadcast can be implemented securely as well. As mentioned before, it is non-trivial to run only a single random walk robustly in an unreliable system. Here, we present a service to realize this problem.

As a last point connected to using the budget, let us consider the exact method of peer sampling used by our random walk. If we use uniform sampling with replacement, then the walk will take needless steps when it visits a training example that has no more budget left. To be precise, the probability that a node is not visited at all during the first

n updates in a network of size n is $\exp(-1)$ according to the Poisson distribution, which is quite a large probability. Depending on the budget management option, this results in wasted bandwidth and time. This shows that the ideal random walk should use sampling without replacement; that is, it should follow a permutation of the network, and when all nodes have been visited, it should start a new permutation until the privacy budget has been spent. This, however, is hard to realize in a decentralized manner.

Notes on Privacy and Security

It should be stressed here that any uncorrupted node is protected by this scheme regardless of fabricated input or the security of the random walk in general. In other words, even if the random walk is compromised and a given uncorrupted node gets arbitrary input and gets queried an arbitrary number of times, the node will be protected by ϵ-differential privacy.

Now, we focus on privacy only. Based on the comment above, this is indeed an independent problem as we can guarantee the privacy of uncompromised nodes regardless of the security of any other components of the implementation. Nevertheless, security is still vital in a complete system as without it the global output can be corrupted and vandalized. In particular, the random walk needs to be secure to maintain an unbiased sampling of the learning examples. Also, an adequate protection is required against vandalism, when adversaries or faulty nodes inject arbitrary information into the system. Again, however, the privacy of local data is completely in the hands of the local node, and it is independent of the outside world.

5.1.3 Algorithm

Let us now turn our attention to the implementation of the single random walk service. A random walk can be viewed as a mobile agent that has a state and that jumps from node to node based on local decisions at each node. The state of the walk is application dependent; for example, it can represent a machine learning model that is updated at each node based on local information. As we mentioned before, our goal here is to propose a protocol that robustly maintains a single random walk in the system, since any extra random walks will waste the privacy budgets of the nodes without contributing to the final model. In this chapter the statistical properties of the random walk are irrelevant. The selection of the

next peer at each node is performed by the method SELECTPEER, which we treat as a black box here. By default, one can for example assume a uniform random walk.

Owing to our *agility* requirement, the walk is performed in a "hot potato" style, that is, the walk moves on as soon as the local state update is performed. We should mention here that the state of the walk can be very large, possibly in the order of megabytes or more.

Let us now describe the protocol that maintains a single random walk. At any point in time, ideally there is only a single walk in the network, but—as we will see—there can be more than one walk in practice due to restarted walks based on false alarms. We will manage these walks by broadcasting a small global state about the progress of the best walk.

Broadcasting Global Updates

Every time a walk completes a new step an update is broadcast about this event. This update contains the step count of the walk in question, as well as a unique id. Thus, the update is extremely small as it contains only two integer values.

The update is broadcast via a standard push-pull gossip algorithm (see Algorithm 5.6) that runs continuously with a period of Δ.

Every node stores only a single update locally in a variable called RWPROPS. This variable represents the local approximation of the step count of the leader random walk in the system. When a new update is received through gossip (procedure ONRECEIVERWPROPS) it has to be decided whether the new update should replace the locally stored one. Intuitively, we should replace the local update if the new update represents fresher, more up-to-date information about the best live random walk than the local RWPROPS.

In order to decide whether the update represents a live random walk, we use a timeout mechanism that is based on the age of the update. Clearly, live random walks generate events every time they make a new step. We can measure the age of these events, without global synchronization, if we accumulate the time intervals that an update spent on the nodes it visited and the total transfer time the update spent traveling over network links. This approach introduces some error into the age approximation, but our protocol is not sensitive to this error. We will revisit this issue later on.

Algorithm 5.6 does not contain details about the above-mentioned age accounting

Algorithm 5.6 Single Random Walk Protocol

```
 1: rwprop:                                          ▷ local variable storing information about the random walk
 2: rw:                                              ▷ local variable storing the state of latest visiting walk
 3: Δ:                                               ▷ gossip period
 4: δ:                                               ▷ update timeout
 5:
 6: loop                                             ▷ push-pull gossip protocol to broadcast walk updates
 7:     wait(Δ)
 8:     p ← selectPeer()
 9:     send rwprop to p
10:     send pull request to p
11: end loop
12:
13: procedure onReceiveRWProps(rwprop')
14:     if (rwprop.steps < rwprop'.steps and
                                  (rwprop.age() ≤ rwprop'.age() < δ or rwprop.age() > rwprop'.age()))
        or (rwprop.steps ≥ rwprop'.steps and
                              (rwprop.age() ≥ δ > rwprop'.age() or rwprop.age() > rwprop'.age() +δ)) then
15:         rwprop ← rwprop'
16:     end if
17: end procedure
18:
19: procedure onUpdateTimeout(i)                     ▷ called when rwprop.age() reaches i · δ
20:     if rwprop.rwsteps − rw.steps ≤ i then
21:         forwardRandomWalk()                      ▷ a walk is restarted
22:     end if
23: end procedure
24:
25: procedure onReceiveRandomWalk(rw')
26:     rw'.steps ← rw'.steps+1
27:     if rw.steps < rw'.steps then
28:         rw ← rw'
29:     end if
30:     if rwprop.steps < rw.steps or rwprop.age() ≥ δ then
31:         rwprop ← new RWProp(rw.steps)
32:         forwardRandomWalk()
33:     end if
34: end procedure
```

mechanism of the updates, as it is rather technical. The approximated age is presented by the method called RWPROPS.AGE that returns the current age of the update in terms of the wall-clock time elapsed since the update was created.

Now, we introduce a timeout threshold δ, which represents our heuristic that if a given update is older than δ then it probably belongs to a dead random walk. The idea behind this is that if a walk does not generate updates for more than a δ time, then the last update

it created will time out at all nodes at about the same time, clearing the way for any new walks to compete for the leadership position again.

The exact conditions for replacing the local update with the incoming one are stated in line 14. This complex formula takes into account all possible combinations of local and incoming step counts and ages, and maximizes the probability that the local update will belong to a live random walk with a maximal step count. For example, even if the local update records a larger step count, it is replaced by the incoming update if its age is smaller by at least δ, since we assume that—although the incoming update can also be outdated—the random walk recorded by the local update was probably already dead when the incoming update was created so the incoming update almost certainly represents more up-to-date information. The rest of the cases are more straightforward.

Restarting and Dropping Random Walks

In our system model—where we assumed reliable connections—a live random walk can crash only if the node that currently hosts the walk is not able to transmit the walk to any neighbor before crashing or leaving the network. With a small random walk state the probability of this is very small, but with the large state we have in mind it is more common for a node to crash before completing the transmission to the next node. This means that, with a positive probability, every walk can crash in each step, so the number of walks will decrease if there is no restarting mechanism in place.

First of all, to allow restarting, each node maintains a local copy of the state of the best random walk it has been visited by (variable RW) managed by the method called ONRECEIVERANDOMWALK. There, we store the received random walk if it has a larger step count than the previous local copy. In addition, if the random walk has a larger step count than the current best live random walk the node knows about, then the random walk is forwarded and a new update is generated. Otherwise the random walk is dropped.

As explained above, we can detect the crashing of the leader random walk due to our timeout mechanism. The restarting method is based on this timeout, also taking into account our requirement that random walks should be *long-lived*.

In Algorithm 5.6 event handler ONUPDATETIMEOUT takes care of restarting random walks. This handler is called when the age of the current update (RWPROPS) reaches $i \cdot \delta$. When $i = 1$, only the node right before the last step of the walk will try to restart the walk.

If this is successful, we lose only the last step. During the next period of δ, the new walk will propagate its new updates, or the nodes will reach a timeout of 2δ. In the latter case, now both the last two nodes try to restart the walk ($i = 2$), and so on. This continues until eventually a node can successfully restart a walk. This walk will generate and broadcast new update events that will replace the timed-out updates at all the nodes.

The method called forwardRandomWalk is responsible for sending the local random walk state rw to a neighbor, thus implementing one step of the walk. Here, we will not go into details about the method used in Algorithm 5.6. The implementation picks a neighbor and attempts to transfer the walk. This is repeated until the transfer is successful or until the forwarding is no longer necessary. The latter condition occurs if in the meantime the node gets a new update about a live walk that has a larger step count than the walk that the node is trying to forward.

Analysis

First let us give a sketch of the proof that if there are online nodes that have received at least one update before and that form a connected network then there will always be a live walk after at most a finite amount of waiting time. This is easy to see, because if there is no live walk in the network then no new events are generated, so all the nodes will eventually reach a timeout of δ. This will trigger the restart mechanism, which will eventually be successful if there is at least one online node, since eventually i will become large enough to involve all the online nodes (see the method called onUpdateTimeout(i)). From this point, all the online nodes will attempt a restart in every period δ until the first successful update overwrites the timed-out update.

Let us note, however, that our method does not actually guarantee that eventually there will be only a single walk. Indeed, for example, if there are two walks with the same step count that progress exactly in synchrony, making steps at exactly the same time then it is in principle possible that both of them survive indefinitely. However, we did not feel it necessary to improve our protocol to deal with this case (it would be possible with some complications) since this scenario has a very low probability. Also, if symmetry is broken then there is a positive probability that the walk with the smaller step count will hit a node that has a fresh-enough update about the walk with the larger step count, which will eventually end the walk. Instead, we opted for keeping the protocol simple and we prove

experimentally that the number of concurrent walks is close to one in practice.

Let us now consider the cost of the protocol. The push-pull broadcast involves very small messages of a few dozen bytes that generate a negligible traffic on a link even if Δ is small (will will use $\Delta = 100\,\text{ms}$ in our tests). At the same time, push-pull broadcast results in an expected convergence time of $O(\Delta \log n)$ (where n is the network size) if peer selection is random [54]. This, considering that Δ is small, results in a reasonably fast broadcast process. To give an illustration, if the random walk has a state of 1 MB, and we set a bandwidth limit of 100 kbit/s for our application then it takes over a minute to make one step. This is about an order of magnitude more time than the broadcast convergence time in a typical network.

The random walk itself induces little traffic overall, given that we maintain just a single walk that visits any given node very rarely. Of course, with extremely bad parameter settings one could generate many random walks in parallel. Here, we will evaluate the parameters experimentally later on.

Additional Details

We close the discussion of the algorithm by mentioning a few improvements and details that were omitted from Algorithm 5.6 for the sake of clarity. As mentioned above, each update has a unique id. We use this id in two ways. First, when we restart a walk it carries the id of the (timed-out) update that triggered its restart. This way, when the walk visits a node where the same update has not yet timed out—recall that we cannot achieve perfect agreement about the age of an update—the update will be forced to time out so the walk is forwarded and not dropped.

Second, each update is accepted only once; that is, in the method called onReceiveRWProps if the id of the update is the same as that of the current local update rwprops then we drop the received update. This is needed to overcome another problem related to the lack of agreement about update age: it is possible that rwprops has already timed out while rwprops' has not. With our solution it is guaranteed that whenever an update times out, it will not be revived again.

Let us now consider the case where a node rejoins the network after an offline period. In this case, the node waits until it receives a fresh gossip message either via push or pull before taking part in the protocol. This is to prevent premature restarted walks based

on outdated updates. Typically, a fresh message will be received almost immediately after joining the network. Note that this fresh message could have the same id as the old update of the joining node, in which case the node will of course participate in the ongoing restarting effort.

In the special case when the joining node was trying to forward a walk when going offline, after receiving the first fresh update it determines whether it is still supposed to forward the same walk; that is, whether the walk is still considered the leader. This situation occurs if the offline period was relatively short, which is a typical situation in, for example, mobile networks.

Finally, we also need to discuss how to start the very first walk. This is a special case because at that point the local variables at the nodes (RW and RWPROPS) are undefined, so our restarting mechanism is not functional. This can be solved by initializing RW at every node using an initial random walk state and a uniform random negative step count from the interval $[-n, -1]$ where n is the network size, and initializing RWPROP to a special update that never times out. Knowing the exact network size is not critical, it can be approximated by piggybacking the push-pull gossip broadcast protocol using known methods [63]. The node that starts the first walk will broadcast an update with step count zero (note that normally only the receiving node creates a new update). We assume that the node that starts the walk is online and connected to the network long enough to broadcast this first update. Of course, when a walk with a negative step count is picked for restarting, it should set the step count to zero.

5.1.4 Experiments

In order to experimentally analyze the Single Random Walk Service, we simulate node churn based on a real trace of smartphone user behavior. To perform the simulations, we used PeerSim [79].

Parameters and Evaluation Metrics

The trace we used and its properties were presented previously in Section 3.3. 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 are able to simulate a virtual period of up to 2 days by assigning a different segment to each simulated node. To achieve more

dynamic churn scenario, we allow user to participate with any battery levels.

We set $\Delta = 100\,ms$. Our two free parameters are δ and the random walk transmission time δ_{rw}. Transmission time is a function of the size of the random walk state and the bandwidth allocated to the application; it is more convenient to vary transmission time directly. The network size was $n = 10000$.

In our overlay network every node had 50 fixed neighbors. Most of these neighbors were offline at any given time, but the online nodes still formed a connected network. The random walk as well as the gossip messages select a uniform random online neighbor.

The free parameters δ and δ_{rw} took values from $\delta_{rw} \in \{\Delta, 100\Delta\}$ and $\delta \in \{\delta_{rw} + 10\Delta, \delta_{rw} + 20\Delta, \delta_{rw} + 100\Delta\}$ taking all possible 6 combinations. Note that we have to have $\delta \geq \delta_{rw}$ because an update will not be overwritten for a time period of δ_{rw} on average. In addition, δ has to account for the logarithmic dissemination time of the gossip broadcast. Thus, the three values of δ can be considered small, realistic, and large, respectively.

We wish to measure *agility, efficiency* and *longevity*. As a function of time, we recorded the age of the oldest random walk, which characterizes both agility (the steepness of this function) and longevity (the absolute value of this function). We recorded the number of random walks that were propagating concurrently as a function of time, which characterizes efficiency.

Results

Figure 5.1 shows our results. The number of random walks is shown as dots (integers, translated slightly vertically by random noise to illustrate density) and the step count of the oldest random walk is represented as colored points, different colors indicating different random walks. We also give the average number of concurrent walks during the experiment in each plot.

In these simulations agility and longevity are optimal, in the sense that the maximal age grows at the theoretical maximum speed (whose speed is in fact indicated by a straight line, which is completely covered by the dots). As expected, the smallest value of δ results in the largest number of restarts, as the gossip broadcast is not always able to converge. Clearly, as we increase δ, the number of random walks quickly decreases to one, sometimes dropping to zero or jumping to two for a very short time. Interestingly, the very large δ does not result in a visible slowdown of the walks. This is because the

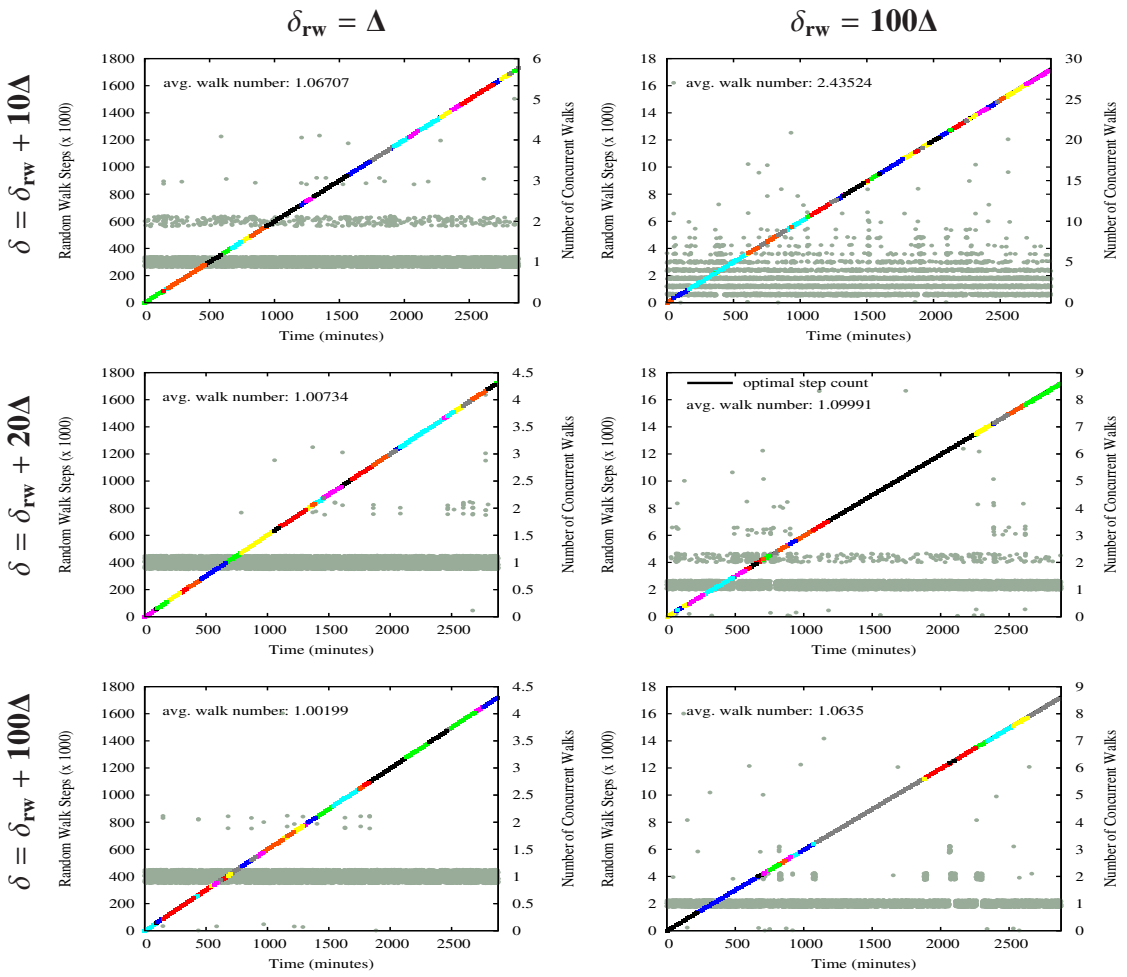


Figure 5.1. Experiments with all the combinations of δ_{rw} and δ. The number of random walks is shown as green dots (integers, translated slightly vertically by random noise to illustrate density) and the step count of the oldest random walk is represented as colored points, different colors indicating different random walks.

extinction events are actually quite rare in our simulated trace: they are indicated by the number of walks dropping to zero.

We also wanted to stress-test the algorithm by artificially increasing the number of random walks that die out. For this reason, we artificially killed each random walk with a probability of 5% for each step of the walk, thereby allowing 20 steps on average. This is an extreme and highly unrealistic scenario. We repeated our experiments, as shown in Figure 5.2.

In this extreme scenario the effect of the different parameter settings is more clearly

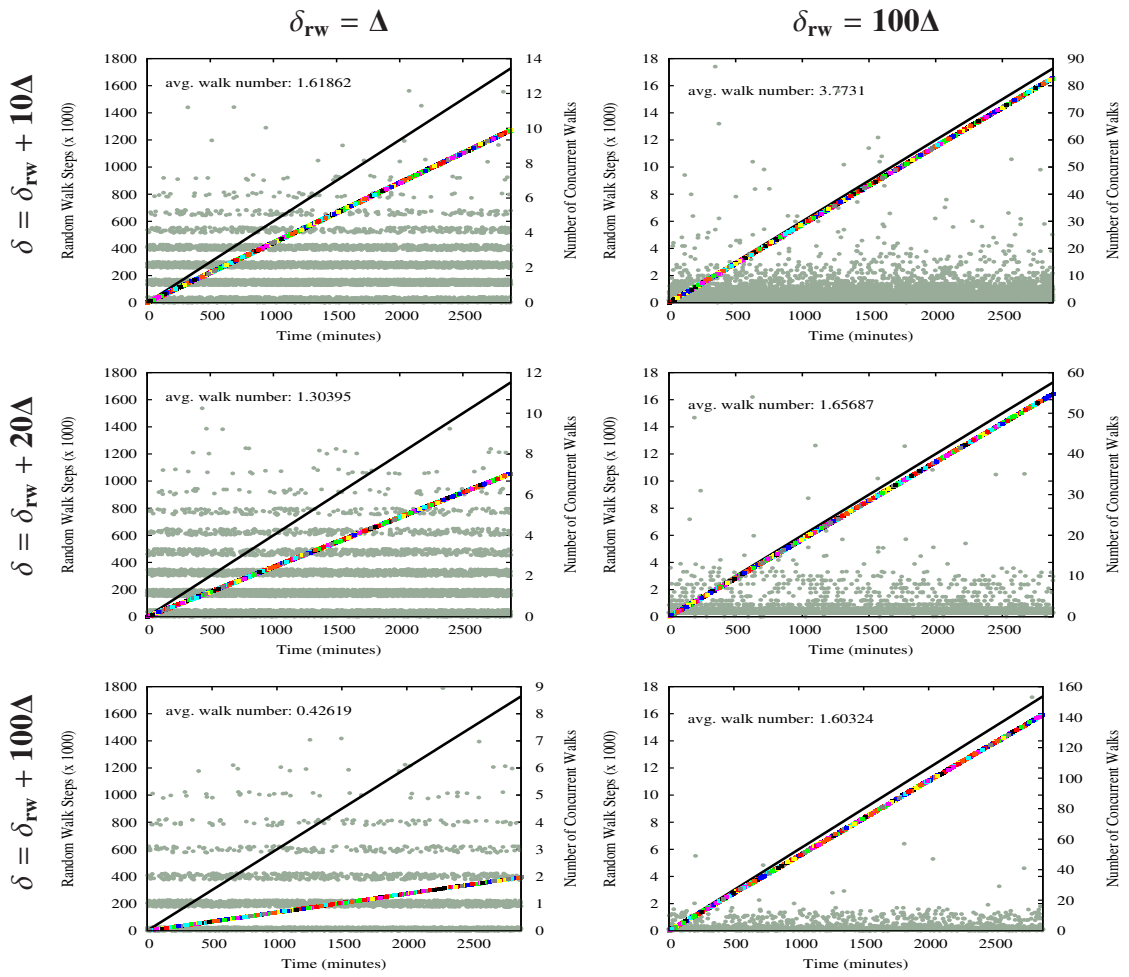


Figure 5.2. Experiments with a 5% drop probability. The number of random walks is shown as green dots (integers, translated slightly vertically by random noise to illustrate density) and the step count of the oldest random walk is represented as colored points, different colors indicating different random walks.

visible. Increasing δ decreases the redundant walks to close to optimal levels, although very rarely for very short periods of time there may be many walks (the plots span the full range covering the outliers as well). On average, however, the efficiency is acceptable. At the same time, the speed of the walks is noticeably reduced.

If we increase δ further, we can no longer increase the efficiency; however, the speed of the walks continues to decrease. We stress again that this scenario has been included only to illustrate an extreme corner of our parameter space. Nevertheless, if the random walks go extinct very frequently then the slowing effect of δ becomes more pronounced,

and there will be a tradeoff between agility and efficiency. Also, the system is more stable assuming large transmission times (that is, large random walk states), as the average number of walks is close to one.

5.2 The Multiple Random Walk Service

Multi-user gossip learning itself raises many research issues. Among these, here we also focus on the management of multiple random walks. As we describe in more detail later on, we will assume that in an overlay network many random walks are run, each representing a learning task. Each task might be owned by a separate user. Our problem here is to ensure that all the walks keep progressing in spite of benign faults caused by nodes joining and leaving (that is, node churn). Also, we would like all the walks to progress quickly; that is, without delay, taking into account the bandwidth that is assigned to the nodes.

The challenge lies in the decentralized nature of the system. Our solution is based on a multi-level protocol in which we have three "lines of defense" that are similar to competence levels in a hierarchical organization. Problem solving is first attempted at the lowest level and in the case of failure the problem is escalated to the next level. The first two levels are completely decentralized. Ideally, these two decentralized mechanisms should handle the vast majority of faults and the third level—which is implemented by the central control of the owner—should be reached only very rarely. The motivation behind this design philosophy is that we wish to offer a conceptually simple and cheap solution that avoids accessing central resources almost all the time, as opposed to a complex and/or expensive protocol that provably works without any central control all the time.

We should add that this is a quite different problem from the one we tackled via the Single Random Walk Service described in Section 5.1. Here, we cannot apply the above-proposed method because it does not scale well. In this case, the shared state size tends to $O(n)$ and it can be very large since n is potentially in the order of millions. Broadcasting with this amount of transmission cost is not feasible along with the payload of multiple random walks.

Here, our contribution is twofold. First, we propose a multi-level decentralized protocol to run $O(n)$ random walks in a network of n nodes that can tolerate benign failures. Second, we demonstrate through simulation that the protocol indeed protects the random

walks and that the walks progress at a near-optimal speed. We base our simulations on a real trace of smart phones which is detailed in Section 3.3.

5.2.1 Algorithm

Let us first define random walks at the level of abstraction that is required for the description of our algorithms. A random walk may be viewed as a mobile agent with a state (consisting of payload and metadata) that jumps from node to node at random. The nature of random neighbor selection is not critical here, but it does affect load balancing, so here we assume a random node is picked from the network with the help of a suitable peer sampling service. The payload of the walk is application dependent. In gossip learning, it represents a machine-learning model that is updated at each node based on local information.

The metadata of the random walk includes a unique walk ID, a restart ID unique within the scope of the same walk ID, and a step count that counts the hops completed by the walk. When the node responsible for the current hop fails before successfully completing the hop, the walk will be restarted using an earlier state that is (hopefully) still available in some previously visited nodes. The restarted walk will have the same walk ID, but it will get a new restart ID.

In the systems we envision there will be n nodes and $O(n)$ random walks each working on different tasks. What we wish to provide is a fault tolerant implementation that is able to restart the failed walks without creating redundant copies.

Bird's Eye View

In a nutshell, our solution is made up of three conceptual levels. At the first (lowest) level a local mechanism is implemented. Here, after completing a random walk hop, every node monitors the success of the next hop. In the case of a failure, the monitoring node will restart the walk. This mechanism is local because the monitoring node retains a copy of the payload that it has just transmitted to the monitored node.

The idea is that in the vast majority of failures this local mechanism will fix the problem, but when it does not, the problem gets escalated to level two. This happens when the monitoring node fails before it can detect the failure of the walk. The node performing the current hop therefore monitors the monitoring node (called the supervisor) and invites

a new supervisor if the current supervisor fails. This new supervisor, however, might not store the payload, or it might store only an outdated copy, so at level two a more expensive mechanism has to be used. Namely, when detecting a failure, the supervisor broadcasts the restarting request that will eventually reach those nodes that have fresh versions of the payload. These nodes then attempt to restart the walk in a sequential order determined by how old their copy of the payload is. After a successful restart another broadcast is sent about the success, which prevents further attempts at restarting. A simple mechanism is also in place to stop most of the redundant restarted walks.

The third (and final) level is implemented by the central control carried out by the owner of the walk. The random walk can report its state to the owner regularly (if the owner is reachable), which allows for appropriate interventions. In our simulations this happens extremely rarely, as we will demonstrate later on.

As can be seen from this short summary, we opt for a best effort multi-level mechanism without formal guarantees that nevertheless attempts to escalate as little work as possible to the increasingly expensive upper levels. We think that in the complex systems we focus on this is a preferable approach that allows us to carry out most of the control tasks in a decentralized way while keeping the system conceptually simple and manageable.

Detailed Description of the Protocol

The pseudo code of the protocol run by all the nodes can be seen in Algorithm 5.7.

As for the local state of the node, *sendQueue* is a FIFO queue that keeps sending its next entry to a random node until the queue is not empty. If the recipient node fails before completing the transaction, the queue selects another random node and tries sending the current entry again until the transmission succeeds. This queue also informs the node about each successful transmission by invoking the method onTransmissionComplete(). In this method we simply cancel the monitoring (supervision) of this completed hop and start to monitor the next hop. We also store the walk in *storageQueue*, which is also a FIFO queue with a fixed storage capacity. When it is full, the next entry is removed.

When a random walk arrives successfully, onRandomWalkArrival() is invoked where the node records which node its level one supervisor is, then the payload is updated and the next hop is scheduled.

Failure detection is implemented via the event handler onConnectionTimeout() that

Algorithm 5.7 Multiple Random Walk Protocol

```
 1: δ:                                          ▷ estimated time for full broadcast
 2: Δ:                                          ▷ gossip round length
 3: sendQueue:                                  ▷ queue where walks to be forwarded wait
 4: storageQueue:                               ▷ queue where we store recent random walks
 5: rwEvents:                                   ▷ fresh events broadcast to manage random walks

 6: loop                                        ▷ push-pull gossip protocol to broadcast walk events
 7:     wait(Δ)
 8:     p ← selectPeer()
 9:     rwEvents.cleanup()
10:     send rwEvents to p
11:     send pull request to p
12: end loop

13: procedure onReceiveRWEvents(rwEvents')
14:     for event in rwEvents' \ rwEvents do    ▷ examine the new events
15:         if rwEvents.isObsolete(event) then
16:             continue                        ▷ jump to next event
17:         end if
18:         if event type is RestartRequest then
19:             if storageQueue.contains(event.rw) then
20:                 restartThreadsFactory.start(event)
21:             end if
22:         else if event type is Restarted then
23:             restartThreadsFactory.stop(event)
24:             if rwEevents.containsConflict(event) then
25:                 event ← new MultipleRestarts(rwEvents,event)
26:             end if
27:         end if
28:         if event type is MultipleRestarts then
29:             restartThreadsFactory.stop(event)
30:             for rw in sendQueue do
31:                 if conflict(rw,event.rw) then   ▷ kill redundant walk
32:                     sendQueue.remove(rw)
33:                     p ← getSupervisor(rw)       ▷ either level 1 or 2
34:                     send cancelSupervision(rw) to p
35:                 end if
36:             end for
37:         end if
38:         rwEvents.add(event)
39:         rwEvents.cleanup()
40:     end for
41: end procedure
```

```
42: procedure RESTARTTHREADSFACTORY.START(event)
43:                                ▷ this should be run in a new thread, presentation is simplified
44:     rw ← storageQueue.get(event.rw)
45:     window ← event.rw.steps − rw.steps
46:     restartWindowStart ← window·δ+ event.creationTime()
47:     restartWindowEnd ← (window+1)·δ+ event.creationTime()
48:     wait while currentTime() < restartWindowStart
49:     if currentTime() < restartWindowEnd then
50:         sendQueue.add(rw)                                          ▷ level 2 restart
51:         supervisorAtLevel2.add(newSupervisor(rw))
52:         rwEvents.add(new Restarted(rw))
53:     end if
54: end procedure

55: procedure ONCONNECTIONTIMEOUT(p)
56:     for rw in getSupervisedRWsAtLevel1(p) do
57:         sendQueue.add(rw)                                          ▷ level 1 restart
58:         supervisorAtLevel2.add(newSupervisor(rw))
59:     end for
60:     for rw in getSupervisedRWsAtLevel2(p) do
61:         rwEvents.add(new RestartRequest(rw))
62:     end for
63:     if isSupervisor(p) then                                        ▷ either level 1 or 2
64:         supervisorAtLevel2.add(replaceSupervisor(p))
65:     end if
66: end procedure

67: procedure ONRANDOMWALKARRIVAL(rw, p)
68:     supervisorAtLevel1.add(rw, p)
69:     update(rw)
70:     sendQueue.add(rw)
71: end procedure

72: procedure ONTRANSMISSIONCOMPLETE(rw, p)
73:     q ← getSupervisor(rw)                                          ▷ either level 1 or 2
74:     send cancelSupervision(rw) to q
75:     supervisedAtLevel1.add(rw, p)
76:     storageQueue.add(rw)
77: end procedure
```

is invoked when a neighbor that the node is currently in contact with fails. Here, if a monitored node fails then in the case of level one monitoring (the node was the previous sender) the node simply schedules the restarting of the walk while also assigning a supervisor to itself. In the case of level two monitoring (the node does not have the (fresh) payload) the node schedules for broadcast a new request for restarting the walk. Finally, if the failing node was the node's supervisor then a new one is selected. Note that we do not detail the algorithm for finding (or replacing) supervisors here; it involves contacting live nodes from the network and negotiating with them. Note also that the failing node might have been the supervisor for more than one walk at both level one and two, so all instances need to be replaced.

Let us now move on to the discussion of the second level where restarting is achieved through various broadcast messages. To implement the broadcast primitive, each node runs a basic push-pull gossip broadcast protocol in an active loop with round length Δ. The local set *rwEvents* contains those messages that are currently actively broadcast. Each message is gossiped up to a given maximal number of hops that is set such that all the nodes receive the broadcast with very high probability. The method called rwEvents.cleanup() removes those messages that have reached this limit.

There are three kinds of gossip messages, namely RestartRequest, Restarted and MultipleRestarts. All of these messages refer to the failure of a given random walk instance, identified by the walk ID and the restart ID. For this reason, from now on we will assume that the messages mentioned in the discussion belong to the same failure event, unless otherwise stated.

A RestartRequest is generated by a level two supervisor when it detects that a walk has failed. This request has a reference to the walk ID and the restart ID that failed. A Restarted message is generated by a node when it decides to restart a walk based on a RestartRequest. Apart from the walk ID and the old restart ID, this event also refers to the restart ID of the new walk. A MultipleRestarts message is generated by a node that receives multiple Restarted messages that belong to independent restarts of the same walk following the same failure event. Once again, this event refers to the walk ID and the old restart ID, and in addition it contains the new restart ID that is picked to be kept alive.

The method called onReceiveRWEvents() processes the incoming broadcast messages. There, only the new messages are processed that are not already included in the local set. First it is checked whether a given message is obsolete or not. This is defined

based on a natural dominance relation over the messages. Restarted messages dominate RestartRequests and MultipleRestarts messages dominate the other two types.

In addition, within a given type, an older RestartRequest (with the larger step count) dominates a younger one. Note that normally there should be only one RestartRequest being broadcast but due to the unreliability of the applied failure detector we could in theory have more than one active supervisor for the same walk so more requests may get generated. Messages of type Restarted do not dominate each other, instead, multiple Restarted messages indicate a failure (redundant restarts). In this case a MultipleRestarts message is generated that contains information about which new walk to keep alive: this will be the one with the minimal restart ID. MultipleRestarts messages also have a dominance relation: the message with the smaller restart ID to keep alive wins. Note that due to the timing variance and unreliability of the broadcast primitive, different conflicts might be picked up by different nodes so we may indeed have various different MultipleRestarts messages.

The non-dominated new messages are then processed. In the case of a RestartRequest event if the node has a copy of the payload then a restart timer is started in a separate timer thread. This thread calculates a restart window in which this node is allowed to restart the walk. The window depends on the age of the local copy of the payload. This way, the different copies of the payload in the network attempt a restart in a sequential order with a high probability, avoiding redundant copies. The window is relative to the first creation of the RestartRequest. Knowing the creation time of the request does not necessarily require synchronized clocks, as an approximation is sufficient that can be computed via summing the approximate hop-times during broadcast.

When a Restarted event is received, we stop any restart timers for this walk and check for conflicting (that is, redundant) restarts. Should there be any such redundant restarts, a new MultipleRestarts event is placed in the broadcast message set.

Finally, when a MultipleRestarts event arrives, the node stops any related restart timers and it also removes all the copies of the redundant walk while also canceling any supervisors for these walks. The method called *sendQueue* also checks *rwEvents* before sending the next message for possible conflicts (not shown in the pseudo code).

The new event is then added to *rwEvents* that is also cleaned up, which means that the dominated events and the old events are removed.

Additional Details and Remarks

Let us now discuss a number of issues that were left out of the discussion above. For example, the behavior of re-joining nodes needs to be considered. In our approach we assume that nodes that leave the network (detected as failed) will keep their *storageQueue* when joining again, but they empty their *rwEvents* cache. In addition, they move the content of their old *sendQueue* to *storageQueue*. This prevents outdated messages from arising as a result of re-joining.

It is also worth mentioning that the broadcast messages are temporary, that is, we delete them immediately after their maximal hop-count is reached. This means that without failure events no broadcasting is going on as all the caches are empty.

Let us now clarify the handling of the different restart IDs in the message processing. When determining the dominance relation and the conflicts described previously, we compare only those messages where the walk ID and the old restart ID are the same. In other words, only those messages are compared that belong to the same failure event. This means that, for instance, it is possible that we have two Restarted messages with the same walk ID but with different old restart IDs, and in this case there will be no conflict. The reason is that in principle this could be a situation when the walk failed, was restarted, then failed again and was restarted again within a short time.

However, when removing conflicting walks in response to a MultipleRestarts message, we remove all the walks with the same walk ID and different restart ID irrespective of the previous (old) restart ID of the removed walk. For the above reasons, there is a tiny probability that some walks that should not be removed are in fact removed.

It is possible to handle these temporal ordering issues, but we opted for simplicity and we follow our multi-level principle, namely those problems resulting from such obscure corner cases are escalated to the next level of the system. We justify this choice in our simulation experiments.

Best Effort Design

We stress again that the first two levels of our algorithm may fail in various ways, many of which we have not discussed here. For example, because of the inevitable delay between a failure event and its detection, it is possible that no supervisor is present for a short interval, during which the node might fail, resulting in a failed random walk. Another

example is that it is theoretically possible that no node receives the restart request that has a copy of the payload of the given walk. This could be due to the unreliability of the broadcast or to the fact that most copies were deleted from storage. It is also theoretically possible that the walk will stay alive when (mistakenly) detected as failed due to the unreliability of the failure detector, which will result in undetected redundant walks. It is also possible that a restart attempt at level two eventually fails after emitting a Restarted message (which is sent before and not after the transfer completes) and although this will be detected by the supervisor of the restarter node, this can still result in incorrectly detected multiple restarts with a very small probability. Temporary network partitioning is also a problem if the supervisor and the supervised nodes are in different partitions. The list goes on.

Obviously, one could set the goal of designing a solution with provable properties under carefully selected assumptions. In a complex problem like ours, this approach would almost certainly lead to a protocol that is very hard to implement, understand and manage.

Instead, we propose a multi-level protocol with each level doing its best and escalating any unsolved problems to the next level. The function of the levels is rather clear and all of the levels allow for failures as these will be handled by the next level. Thus, the design process of the algorithms of each level is not an "all or nothing" task but rather a multi-objective optimization process where we minimize the number of failures while maximizing the simplicity and manageability of the protocol. As for research methodology, our main tool is simulation where we demonstrate the cost and reliability of the system as a whole under realistic settings.

The design goal is to ensure that our final fallback mechanism (level three) has a minimal load. At this level, the owner of the walk (and the associated task) has to provide only minimal resources like a mobile phone for 10 minutes each day, or a very limited public cloud service. During, for example, one short daily visit by the user, the walks of the user (if any) report back to the user who can remove or restart walks as needed. Thus, the system as a whole does not require an expensive infrastructure and can remain open and free for all the potential users.

Note that this approach is in line with several related studies in the area of P2P-assisted systems that use unreliable distributed protocols only as a first level and a central service provides the guarantees for the reliability of the application (for example, [58, 84]). Here

Table 5.1. Fixed Parameters

Δ	100 ms
δ	2000 ms
storageQueue size	10 entries of maximal size
max gossip hop count	20 steps

our goal is slightly different in that we wish to reduce the contribution of the central final level to an absolute minimum, and also in that we organize the distributed protocol itself into hierarchical levels in a similar manner.

5.2.2 Experiments

In order to experimentally analyze our protocol, we simulate node churn based on a real trace of smartphone user behavior. To perform the simulations, we used PeerSim [79].

Experimental Setup

All our experiments were run on top of the churn trace described in Section 3.3. We simulate the network based on the worst case churn scenario when phones with any battery level are allowed to join. This results a more dynamic scenario to evaluate our proposed protocol. In each experiment the algorithm parameters listed in Table 5.1 were fixed. Note that δ (the length of the restarting window) is calculated as 20Δ, which is the longest time a given broadcast message is expected to spend in the network. This increases the possibility that restarting windows are indeed sequential and non-overlapping. The maximum allowable amount of data in the storage queue is set so that the queue could store ten entries of the maximal size. Thus, in scenarios where the payload size is variable, the queue might store more than ten entries.

In our experiments we varied three main parameters of the application environment. These were the network size, the number of random walks to maintain, and the distribution of the size of the payload of the random walks. As for network size, we experimented with $n = 1000$ and $n = 100,000$.

Regarding the number of random walks, we designed three scenarios with an expo-

nentially increasing number of random walks. To determine how many walks to start, we first examined the churn trace and found that, on average, 54% of the nodes are online. Based on this, as a baseline setup we started $0.54n$ random walks in expectation. As for the implementation, each node was assigned a walk initially with a probability of 0.54. We also ran experiments with ten times more and ten times fewer walks than this baseline. The exact number of random walks can be found in Table 5.2.

The payload size was defined in terms of transmission time assuming a fixed bandwidth limit at the nodes. We defined a small and a large payload with a transmission time of 1000 ms, and 10,000 ms, respectively. We ran simulations with only small and only large payloads, as well as with a mixture of payloads where each random walk was assigned a transmission time at random with a uniform distribution over the interval [1000 ms,10,000 ms].

The overlay network was implemented by independently assigning 50 randomly selected neighbors to each node. This setting was used for all the network sizes. We assume that each node maintains an active TCP connection with its neighbors as suggested in [91]. If a node fails, its neighbors will detect this only with a one second delay. The neighbor set is constant in our simulations; that is, when a neighbor fails it remains on the list and it is reconnected when it comes back online. The size of our neighbor set was large enough for the overlay network to remain connected.

We should also mention that we applied a short warm-up period before the simulation that is not included in our reported statistics. We did this to model the realistic usage scenario where new random walks are added to the system by their owners, making sure that the walk completes at least a few hops so that there are copies in the network to restart from. For this reason, those nodes that were assigned a walk to start were kept online for five minutes and we started the trace-based simulation only after this period. In addition, to avoid an immediate synchronized spike in failures (a simulation artifact), we made sure the nodes that started a walk were assigned a trace with an initial online period.

Results

Let us first take a look at the statistics of the various experimental scenarios at the end of the simulated day in Table 5.2. We can see that in all the cases there is a large number of restarts, more than ten times as many as the number of walks. The most important result is that the vast majority of these restarts happen at level one.

Table 5.2. Overview of the results concerning the Multiple Random Walk Protocol at the end of the simulated day

Scenarios			Multiple Random Walk Protocol				without level 2 restarts		without restarts
network size	transmission time (s)	# random walks	# restarts at level 1	# restarts at level 2	lost random walks	max. # events broadcast	# restarts at level 1	lost random walks	lost random walks
10^3	1	40	672	0	0.00% (0)	0	672	0.00% (0)	100%
		553	9888	4	0.72% (4)	1	9922	1.44% (8)	100%
		5440	93620	253	0.62% (34)	3	90818	5.88% (320)	100%
	rand(1,10)	40	661	3	2.50% (1)	1	643	7.50% (3)	100%
		553	9512	20	1.98% (11)	1	9465	6.69% (37)	100%
		5440	75650	1083	0.58% (32)	7	67965	16.04% (873)	100%
	10	40	654	1	0.00% (0)	1	705	7.50% (3)	100%
		553	9011	46	1.44% (8)	1	8862	7.23% (40)	100%
		5440	79488	2055	0.75% (41)	7	65236	27.37% (1489)	100%
10^5	1	54383	923938	379	0.82% (449)	3	922568	1.43% (780)	100%
	rand(1,10)	54383	907904	2269	0.71% (391)	4	889612	4.85% (2642)	100%
	10	54383	887857	4183	0.74% (407)	5	858649	7.83% (4262)	100%

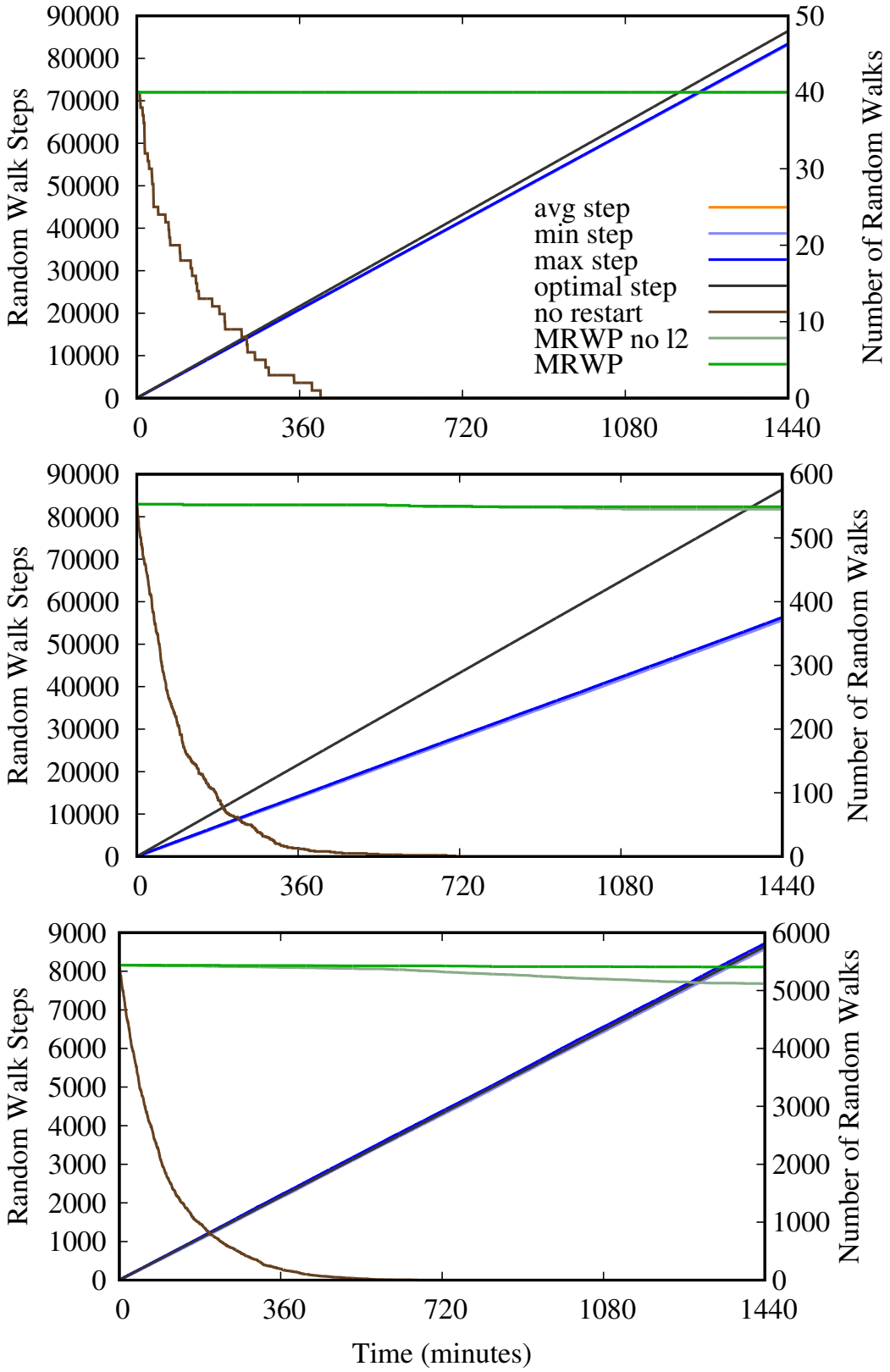


Figure 5.3. Experimental results with $n = 1000$, and small payload (1000 ms transmission time) with a varying number of random walks.

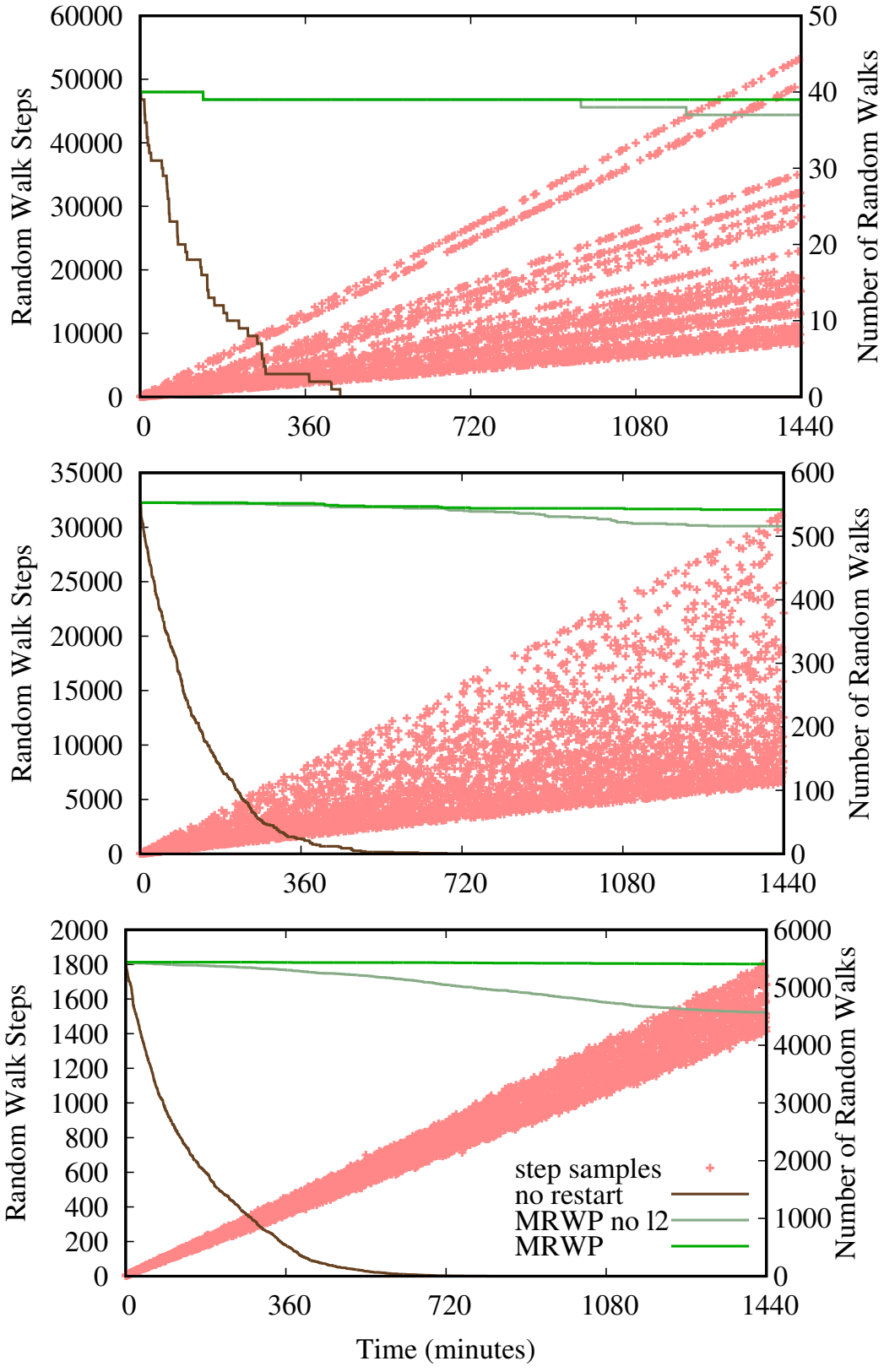


Figure 5.4. Experimental results with $n = 1000$, and mixed payload (between 1000 ms and 10000 ms transmission time) with a varying number of random walks.

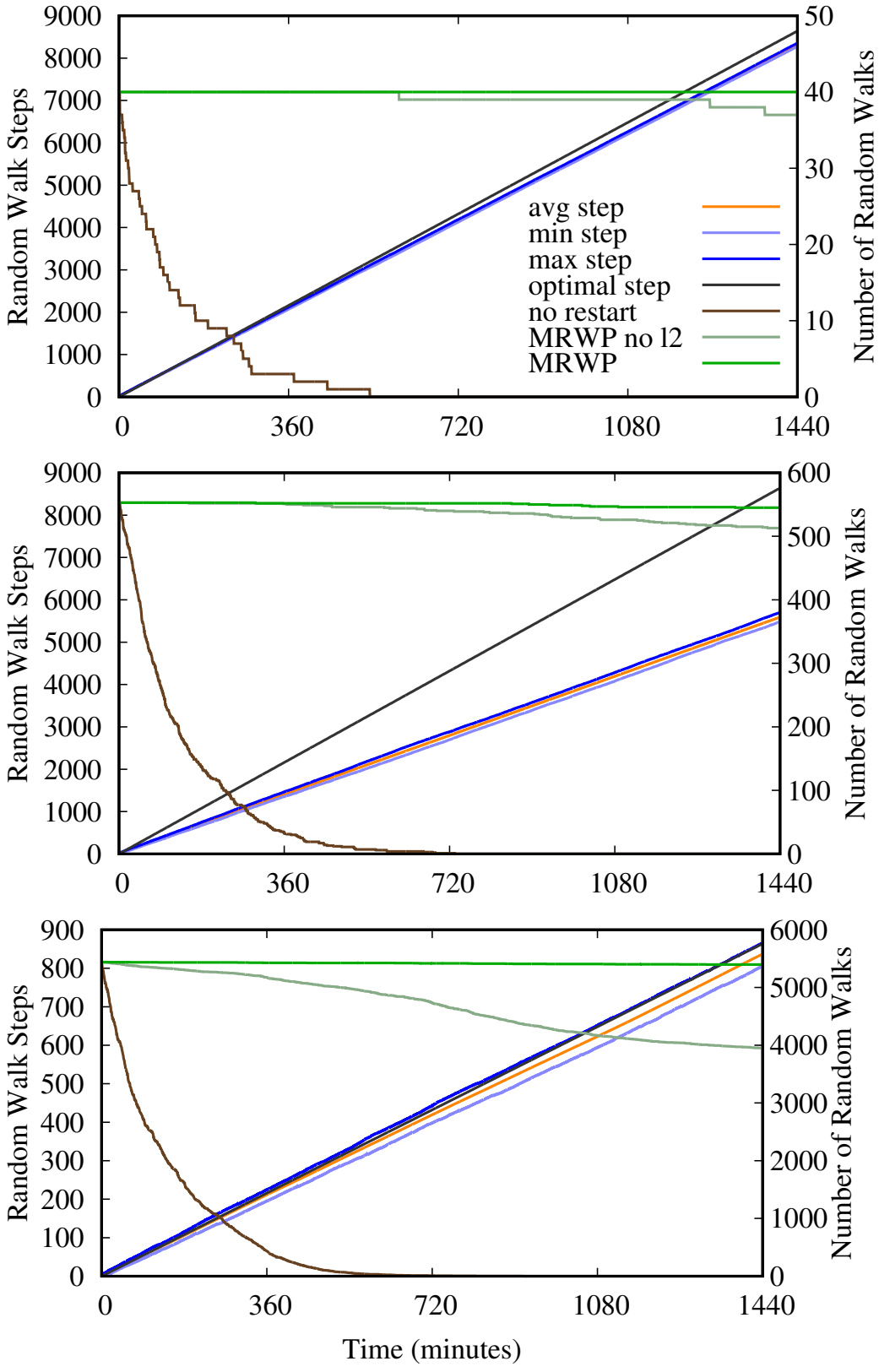


Figure 5.5. Experimental results with $n = 1000$, and large payload (10000 ms transmission time) with a varying number of random walks.

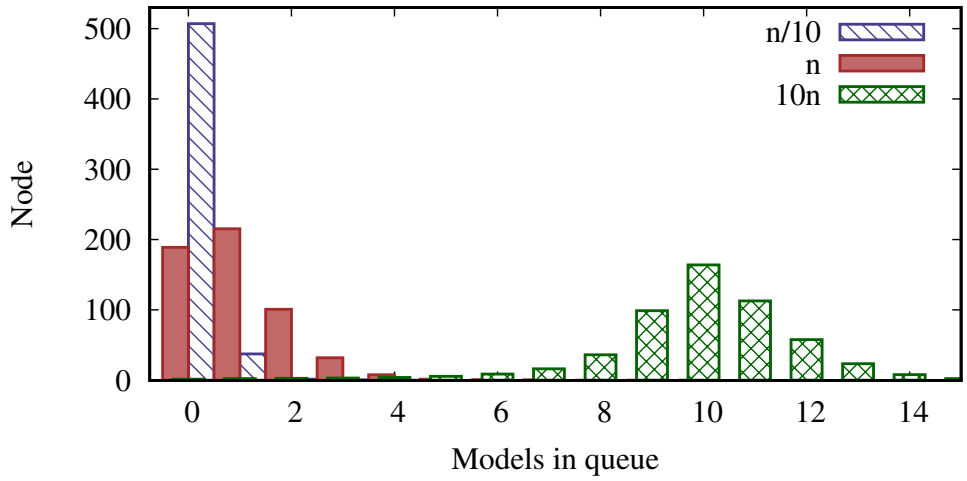


Figure 5.6. The histograms of the sendQueue sizes in the three scenarios with 1000 ms payload transmission time. The notations $n/10, n$ and $10n$ represent our three settings for the number of random walks.

It is clear that the number of level two restarts depends mostly on the size of the payload of the walks. With a large payload (and long transmission time) there is a larger probability that the level one supervisor fails before the transmission gets completed, thus triggering a level two process.

Also, we get a disproportionate increase in the number of level two restarts when we increase the number of walks beyond the number of online nodes. In that scenario, apart from the fact that there are more walks and thus more restarts, the walks spend a lot of time in the sending queues of the nodes, so the expected time for the supervision becomes an order of magnitude longer. This in turn increases the probability of the failure of the level one supervisor.

As for level three events, we did not observe any instances of redundant walks, and only around one percent of the walks got lost.

A very interesting issue is the communication cost of the protocol. From the table, we can see that the maximal size of the broadcast table is extremely small; it is in fact negligible when considering that the entries in the table are also very small. Note that the maximal value is indicated, but in fact the broadcast tables are empty most of the time. This means that the overhead of the protocol is small, so the communication costs are dominated by the random walks.

Let us now examine the effect of the various levels. When omitting level two, not

surprisingly, slightly fewer restarts happen at level one, since those walks that get lost will not need further restarts. When there is no restarting mechanism in place, we lose all the walks.

Finally, let us note that our experiment with the large network size of $n = 100,000$ also produces very few level three failures and the broadcast cost is as low as in the smaller networks. This confirms the scalability of the approach.

Table 5.2 does not show the dynamic properties of the statistics, and the speed of the random walks cannot be seen either, which is a key property we wish to maximize. Thus, we include plots as well that contain the number of hops the random walks complete along with the the number of walks that are alive as a function of time.

Figures 5.3, 5.4 and 5.5 were obtained using the three different payload size distributions we experimented with, and each figure contains three plots that correspond to the three different numbers of random walks.

The plots tell us how quickly the random walks die out without any restart mechanism. As we saw in Table 5.2, here we also observe that relying only on level one restarts (and not using level two) more walks get lost, although, as we saw previously, the difference is not large.

The plots also contain statistics about the number of hops (steps) the random walks completed by the given point in time in the form of average, minimum and maximum in the homogeneous payload scenarios and in the form of samples from individual walks in the mixed scenario. The optimal speed is also included. When calculating the optimal speed, we took the number of walks into account. That is, when there are ten walks on average for each online node, the optimal speed gets divided by ten to account for the fixed bandwidth limit we assume when transmitting random walks. Otherwise, the optimal speed is given by always transmitting the walk without any delay at the maximum bandwidth.

Clearly, in the case of the scenarios with a small number of models we achieve a near-optimal random walk speed in the homogeneous payload size scenarios. In the case of mixed payload sizes, the speed of the walks with a large payload is close to optimal, but the walks with a small payload suffer delays due to queuing behind large payloads. Note that, although for all the walks the average queuing time for one hop is the same irrespective of the payload size, for walks with a small payload there are a larger number of hops on average so these walks spend more time in the queues in total.

In the scenario where the number of walks is the same as that of the average online nodes, walks slow down somewhat even in the homogeneous scenarios. This is because in this case the sending queue will often contain one or more walks to queue behind, which causes delays. This is illustrated well by the histograms shown in Figure 5.6 where the variance of the queue size can also be seen.

Interestingly, in the case of the largest number of walks the speed is again close to optimal. This is because here the queuing time is the most important factor that determines the speed and in the queues the waiting time can average out due to the larger number of walks. The histograms in Figure 5.6 show the queue size distribution for this scenario as well.

5.3 Conclusions

First, we introduced a protocol to implement a robust random walk that is fast, efficient and long-lived in a dynamic network. Our motivation was that in many applications, most importantly, in distributed data mining, the random walk is an important primitive that has to be implemented reliably in realistic network environments. Furthermore, if we want to reduce the privacy budget then every data item can be visited only a limited number of times. We demonstrated that the proposed Single Random Walk Service meets our requirements—agility, longevity and efficiency—in a network of mobile smartphones. We simulated this service over a smartphone trace and we found that the protocol is robust to its main parameter, the timeout threshold δ, which determines when a random walk is considered dead. We obtained an acceptable performance even in an unrealistic extreme scenario where we artificially removed random walks with a 5% probability in each step. In this case, this protocol is more sensitive to δ, but with a sensible setting a good compromise can be achieved between efficiency and agility.

In this chapter we also introduced a protocol to maintain $O(n)$ random walks over an overlay network, where the random walks represent independent decentralized tasks that might belong to different users. We motivated this service with a decentralized data mining application, gossip learning, but any applications based on random walks could be supported. The protocol follows a three-level design where problems not solved at a lower level get escalated to the next level. During our experimental evaluation we used a smartphone trace to model churn. We demonstrated that in all the scenarios we tested

the vast majority of failures are dealt with at the lowest level, which is purely local and therefore scalable. Only a small fraction of the problems get escalated to level two, which is based on a broadcast primitive. In this case the cost of broadcast messages was shown to be almost negligible due to the small number of failure cases that reach this level. Thus, the overhead introduced by level two is small relative to the communication cost of the random walks. Level three, the central control by the task owner, was reached only a few times during all our simulations. We also demonstrated in experiments that the speed of the random walks is close to optimal.

Contribution

In this chapter, most of the above-presented results are regarded as main contributions of the author of this dissertation. More specifically, he developed and evaluated both of the presented random walk services. The contribution by István Hegedűs to this topic deserves a mention as well. He developed and evaluated a robust decentralized stochastic gradient descent method that satisfies the criteria of differential privacy [42]. His results rely heavily on advantages of our single random walk service. The statements in this chapter that related to differential privacy were proposed based on his contribution. However, the actual details of his results fall outside the scope of this dissertation.

CHAPTER 6

Mini-Batch Gradient Descent

Data mining over personal data harvested from mobile devices is a very sensitive problem due to the strong requirements of privacy preservation and security. Recently, the *federated learning* approach was proposed to solve this problem by not collecting the data in the first place but instead processing the data in place and creating the final models in the cloud based on the models created locally [56, 76].

We go one step further and propose a solution that does not utilize centralized resources at all. The main motivation for a fully distributed solution in our cloud-based era is to preserve privacy by avoiding the central collection of any personal data, even in pre-processed form. Another advantage of distributed processing is that this way we can make full use of all the local personal data, which is impossible in cloud-based or private centralized data silos that store only specific subsets of the data. The key issue here of course is to offer decentralized algorithms that are competitive with approaches like federated learning in terms of time and communication complexity, and that provide increased levels of privacy and security.

Previously, we proposed numerous distributed machine-learning algorithms in a framework called gossip learning. In this framework models perform random walks over the network and are trained using stochastic gradient descent [82] (see Chapter 2). This involves an update step in which nodes use their local data to improve each model they receive, and then forward the updated model along the next step of the random walk.

Assuming the random walk is secure—which is a research problem on its own, see e.g. [50]—it is hard for an adversary to obtain the two versions of the model right before and right after the local update step at any given node. This provides reasonable protection against uncovering private data.

However, this method is susceptible to collusion. If the nodes before and after an update in the random walk collude they can recover private data. In this chapter we address this problem, and improve gossip learning so that it can tolerate a much higher proportion of honest but curious (or semi-honest) adversaries. The key idea behind the approach is that in each step of the random walk we form groups of peers that securely compute the sum of their gradients, and the model update step is performed using this aggregated gradient. In machine learning this is called mini-batch learning, which—apart from increasing the resistance to collusion—is known to often speed up the learning algorithm as well (see, for example, [26]).

It might seem attractive to run a secure multiparty computation (MPC) algorithm within the mini-batch to compute the sum of the gradients. The goal of MPC is to compute a function of the private inputs of the parties in such a way that at the end of the computation, no party knows anything except what can be determined from the result and its own input [111]. Secure sum computation is an important application of secure MPC [20].

However, we not only require our algorithm to be secure but also fast, light-weight, and robust, since the participating nodes may go offline at any time (see Chapter 3) and they might have limited resources. One key observation is that for the mini-batch algorithm we do not need a precise sum; in fact, the sum over any group that is large enough to protect privacy will do. At the same time, it is unlikely that all the nodes will stay online until the end of the computation. We propose a protocol that—using a binomial tree topology and Paillier homomorphic encryption—can produce a "quick and dirty" partial sum even in the event of failures, has adjustable capability of resisting collusion, and can be completed in logarithmic time.

We also laid great emphasis on demonstrating that the proposed protocol is practically viable. This is a non-trivial question because homomorphic cryptosystems can quickly become very expensive when applied along with large-enough key-sizes (such as 2048 bit keys), especially considering that in machine learning the gradients can be rather large. To achieve practical viability, we propose an extreme lossy compression, where we discretize

floating point gradient values to as few as two bits. We demonstrate experimentally that this does not affect learning accuracy yet allows for an affordable cryptography cost. Our simulations are based on a real smartphone trace which are detailed in Section 3.3.

6.1 Related Work

There are many approaches that have goals similar to ours, namely to perform computations over a large and highly distributed database or network in a secure and privacy preserving way. Our work touches upon several fields of research including machine learning, distributed systems and algorithms, secure multiparty computation and privacy. Our contribution lies in the intersection of these areas. Here we focus only on related work that is directly relevant to our present contributions.

Algorithms exist for completely generic secure computations, Saia and Zamani give a comprehensive overview with a focus on scalability [93]. However, due to their focus on generic computations, these approaches are relatively complex and in the context of our application they still do not scale well enough, and do not tolerate dynamic membership either.

Approaches targeted at specific problems are more promising. Clifton et al. propose, among other things, an algorithm to compute a sum [20]. This algorithm requires linear time in the network size and it does not tolerate node failure either. Bickson et al. focus on a class of computations over graphs, where the computation is performed in an iterative manner through a series of local updates [11]. They introduce a secure algorithm to compute local sums over neighboring nodes based on secret sharing. Unfortunately, this model of computation does not cover our problem as we wish to compute mini-batches of a size independent of the size of the direct neighborhood, and the proposed approach does not scale well in that sense. Besides this, the robustness of the method is not satisfactory either [81]. Han et al. address stochastic gradient search explicitly [41]. However, they assume that the parties involved have large portions of the database, so their solution is not applicable in our scenario.

Bonawitz et al. [15] address a similar problem setting where the goal is to compute a secure sum in an efficient and robust manner. They also assume a semi-honest adversarial model (with a limited set of potentially malicious behaviors by a server). However, their solution requires a server and an all-to-all broadcast primitive even in the most efficient

version of their protocol. Our solution requires a linear number of messages only.

The algorithm of Ahmad and Khokhar is similar to ours [4], as they also use a tree to aggregate values using homomorphic encryption. However, in their solution all the nodes have the same public key and the private key is distributed over a subset of elite nodes using secret sharing. The problem with this approach in our mini-batch gradient descent application is that for each mini-batch a new key set has to be generated for the group, which requires frequent access to a trusted server, otherwise the method is highly vulnerable in the key generation phase. In our solution, all the nodes have their own public/private key pair and no keys have to be shared at any point in time. What is more, these key pairs may remain the same in every mini-batch the given node participates in without compromising our security guarantees.

We need to mention the area of differential privacy [28], which is concerned with the the problem that the (perhaps securely computed) output itself might contain information about individual records. The approach is that a carefully designed noise term is added to the output. Gradient search has been addressed in this framework (for example, [88]). In our distributed setup, this noise term can be computed in a distributed and secure way [29].

We also strongly build on our previous work [23]. There, we proposed an algorithm very similar to the one presented here. In this study we offer several optimizations of the algorithm and we propose the binomial topology for building the mini-batch overlay tree. We also explore the issue of gradient compression necessary for keeping the cost of cryptography under control and we perform a detailed experimental study of the algorithm based on a smartphone churn trace.

6.2 Adversarial model

We assume that the adversaries are honest but curious (or semi-honest). That is, nodes corrupted by an adversary will follow the protocol but the adversary can see the internal state of the node. The goal of the adversary is to learn about the private data of other nodes (note that the adversary can obviously see the private data on the node it observes directly). Wiretapping is allowed, since all the sensitive messages in our protocol are encrypted.

We assume a static adversarial model, which means that the corrupted nodes are picked a priori, independently of the state of the protocol or the network. As of the

number of corrupted nodes, we will consider the threshold model, in which at most a given number of nodes are corrupted, as well as a probabilistic model, in which any node can be corrupted with a given constant probability [75].

We also assume that adversaries are not able to manipulate the set of neighbors. In each application domain this assumption translates to different requirements. For example, if an overlay service is used to maintain the neighbors then this service has to be itself secure.

6.3 Our Solution

As explained previously, at each step, when a node receives a model to update, it coordinates the distributed computation of a mini-batch gradient and then uses this gradient to update the model. Based on the assumptions stated in Section 2.2, Section 6.2 and building on the GOLF framework outlined in Section 2.3 we now present our algorithm for computing a mini-batch gradient.

6.3.1 Mini-Batch Tree Topology

The very first step for computing a mini-batch gradient is to create a temporary group of random nodes that form the mini-batch. In our decentralized environment we do this by building a rooted overlay tree. The basic version of our algorithm will require the overlay tree not only to be rooted at the node computing the gradient but also to be *trunked*.

Definition 3 (trunked tree)**.** *Any rooted tree is* 1-trunked*. For $k > 1$, a rooted tree is* k-trunked *if the root has exactly one child node, and the corresponding subtree is a* $(k-1)$*-trunked tree.*

Let N denote the intended size of the mini-batch group. We assume that N is significantly less than the network size. Let S be a parameter that determines the desired security level ($N \geq S \geq 2$). We can now state that we require an *S-trunked tree* rooted at the node that is being visited by gossip learning. As we will see later, this is to prevent a malicious root from collecting too much information.

Apart from the trunk, the tree can be arbitrary, but here we propose a *binomial tree* as a preferable choice. If every node already in the tree spawns a new child node in periodic rounds (starting from a single root node) then the result is a binomial tree. It is not possible

to construct a tree of a given size faster, since in the case of a binomial tree each node keeps working continuously so the efficiency is maximal. Of course we assumed here that child nodes can be added only sequentially at a given node. However, if we also assume that all the nodes have the same up- and download bandwidth cap then adding nodes in parallel will be proportionally slower thus parallelism provides no advantage as long as we utilize the maximal available bandwidth. The same up- and download bandwidth requirement is naturally satisfied in our application domain because we assume that the protocol is allowed to use only a fixed, relatively small amount of bandwidth (such as 1 Mbps) and low bandwidth connections are excluded from the set of possible overlay connections.

Another advantage of binomial trees is that we can use the links in reverse order of construction for uploading and aggregating data along the tree. This way, we get a data aggregation schedule that is similarly efficient and also collision-free in the sense that each node communicates with at most one node at a given time.

The tree overlay network we have described so far can be constructed over a random overlay network by first building the trunk (which takes a random walk of $S-1$ steps) and then recursively constructing a binomial tree of depth D, resulting in an S-trunked tree of size $2^D + S - 1$ and total depth $d = D + S - 1$. Every child node is chosen randomly from those neighbors of the node that are both online and not in the tree already. No attention needs to be paid to reliability. We generate the tree quickly and use it only once quickly. Normally, some subtrees will be lost in the process because of churn but our algorithm is designed to tolerate this. The effect of certain parameters, such as the binomial tree parameter and node failures, will be discussed later in the evaluation.

6.3.2 Calculating the Gradient

The sum we want to calculate is over vectors of real numbers. Without loss of generality, we discuss the one-dimensional case from now on for simplicity. Homomorphic encryption works over integers, to be precise, over the set of residue classes $\mathbb{Z}_n$ for some large n. For this reason we need to discretize the real interval that includes all possible sums we might calculate, and we need to map the resulting discrete intervals to residue classes in $\mathbb{Z}_M$, where M defines the granularity of the resolution of the discretization. This mapping is natural and we do not go into details here. Since the gradient of the loss function for

Algorithm 6.8 Fully distributed algorithm for computing a mini-batch gradient

```
procedure INIT
    shares ← new array[1..S]
    for i ← 1 to S do
        shares[i] ← Encrypt(0, Ancestor(i))
    end for
    knownShare ← 0
end procedure
procedure ONMESSAGERECEIVED(msg)
    for i ← 1 to S − 1 do
        shares[i] ← shares[i] ⊕ msg[i + 1]
    end for
    knownShare ← knownShare + Decrypt(msg[1])
end procedure
procedure ONNOMOREMESSAGESEXPECTED
    if IAmTheRoot() then
        for i ← 1 to S − 1 do
            knownShare ← knownShare + Decrypt(shares[i])
        end for
        Publish((knownShare + localValue) mod M)
    else
        randSum ← 0
        for i ← 1 to S − 1 do
            rand ← Random(M)
            randSum ← randSum + rand
            shares[i] ← shares[i] ⊕ Encrypt(rand, Ancestor(i))
        end for
        knownShare ← knownShare + localValue − randSum
        shares[S] ← Encrypt(knownShare mod M, Ancestor(S))
        SendToParent(shares)
    end if
end procedure
```

most learning algorithms is bounded, this is not a practical limitation. Also, in Section 6.5 we evaluate the effect of discretization on learning performance and we show that even an extreme compression (discretizing the gradient down to two bits) is tolerable due to the high robustness of the mini-batch gradient method itself.

In a nutshell, the basic idea of the algorithm is to divide the local value at each node

into S shares, encrypt these with asymmetric additively homomorphic encryption (such as the Paillier cryptosystem), and send them to the root via the chain of ancestors. Although the shares travel together, they are encrypted with the public keys of different ancestors. Along the route, the arrays of shares are aggregated, and periodically re-encrypted. Finally, the root calculates the sum.

The algorithm consists of three procedures, shown in Algorithm 6.8. These are run locally on the individual nodes. Procedure Init is called once after the node becomes part of the tree. Here, the function call Ancestor(i) returns the descriptor of the ith ancestor on the path towards the root. The descriptor contains the necessary public keys as well. During tree building this information can be given to each node so the nodes can look up the keys of their ancestors locally. For the purposes of the Ancestor function, the parent of the root is defined to be itself. Function Encrypt(x, y) encrypts the integer x with the public key of node y using an asymmetric additively homomorphic cryptosystem.

Procedure OnMessageReceived is called whenever a message is received by the node. A message contains an array of dimension S that contains shares encoded for the S closest ancestors to the sender child. The first element (msg[1]) is thus encrypted for the current node, so it can decrypt it. The rest of the shares are shifted down by one position and added (with homomorphic encryption) to the local array of shares to be sent (operation $a \oplus b$ performs the homomorphic addition of the two encrypted integers a and b to get the encrypted form of the sum of these integers). Note that the ith element ($1 \leq i \leq S - 1$) of the array shares is encrypted with the public key of the ith ancestor of the current node and is used to aggregate a share of the sum of the subtree except the local value of the current node. The Sth share is aggregated in variable knownShare unencrypted. The value of share[S] is not modified in this method, it will be initialized using knownShare after all the child nodes that are alive have responded.

After all the shares have been processed, procedure OnNoMoreMessagesExpected is called. This happens when the node has received a message from all of its children, or when the remaining children are considered to be dead by a failure detector. The timeout used here has to take into account the depth of the given subtree and the maximal delay of a message. In the case of leaf nodes, this procedure is called right after Init. When calling OnNoMoreMessagesExpected, we know that the ith element ($1 \leq i \leq S - 1$) of the array shares already contains the ith share of the sum of the subtree rooted at the current node (except the local value of the current) encrypted with the public key of the ith ancestor of

the current node. We also know that knownShare contains the Sth share of the same sum unencrypted.

Now, if the current node is the root then the elements of the received array are decrypted and summed. The root can decrypt all the elements because it is the parent of itself, so all the elements are encrypted for the root when the message reaches it. Here, Decrypt(x) decrypts x using the private key of the current node. Function Publish(x) announces x, the output of the algorithm; that is, the final unencrypted sum.

If the current node is not the root then the local value has to be added, and the Sth element of the array has to be filled. First, the local value is split into S shares according to the S-out-of-S secret-sharing scheme discussed in [75]: $S-1$ out of the S shares are uniformly distributed random integers between 0 and $M-1$. The last share is the difference between the local value and the sum of the random numbers (mod M). This way, the sum of shares equals the local value (mod M). Also, the sum of any non-empty proper subset of these shares is uniformly distributed, therefore nothing can be learned about the local value without knowing all the shares. Function Random(x) returns a uniformly distributed random integer in the range $[0, x-1]$.

The shares calculated this way are then encrypted and added to the corresponding shares, and finally the remaining Sth share is encrypted with the public key of the Sth ancestor and put into the end of the array. This array—which now contains the S shares of the sum of the full sub-tree including the current node—is sent to the parent.

6.3.3 Working With Vectors

We now describe how to efficiently extend our method to vectors of discrete numbers, by packaging multiple elements into a single block of encrypted data. Let us first calculate the number of bits that are required to represent one vector element. Assume that the elements of the input vectors are in the range $[0, m]$. This means that the elements of the output vector fall in range $[0, Nm]$, where N is the mini-batch (tree) size. That is, $M = Nm + 1$. After applying the secret-sharing scheme on an input vector, the elements of the resulting shares also fall in the range $[0, Nm]$ due to the S-out-of-S secret-sharing scheme we apply.

However, when working with homomorphic cryptography, we keep adding encrypted shares together without performing the modulo operation that is required for the correct

decoding in our S-out-of-S secret-sharing scheme and for keeping the values in the range $[0, Nm]$. Hence, we need a larger range to accommodate the sum of at most N shares giving us the range of $[0, N^2m]$. This means that $\lceil \log_2(1 + N^2m) \rceil$ bits are required per element.

Using this many bits, we can simply concatenate the elements of a share together to form a single bit vector before encryption. Homomorphic addition will result in the corresponding elements being added together. After decryption, the vector can be restored by splitting the bit vector, and element-wise modulo can be performed. This method can be trivially extended to arrays of blocks of a desired size, by packaging the elements into multiple blocks.

6.3.4 Practical Considerations and Optimizations

We stress again that if during the algorithm a child node never responds then its subtree will be essentially missing (will have a sum of zero), but other than that the algorithm will terminate normally. This is acceptable in our application, because for a mini-batch we simply need the sum of any number of gradients, and this will not threaten the convergence of the gradient descent algorithm.

The pseudocode discussed above describes a simple and basic version of our algorithm that allows for optimizations to speed up execution. Execution time is important because a shorter execution time allows less time for nodes to fail; in addition, the machine learning algorithm will execute faster as well. A simple optimization is, for example, if, as part of their initialization, all the nodes instantly start encrypting the $S - 1$ shares of their local data with the public keys of its $S - 1$ closest ancestors.

Another optimization is the parallelization of encryption and sending. Note that encrypting data typically takes much longer than sending it; we will evaluate this in more detail later on. Here, when calculating the message to send to the parent, the node immediately sends the first encoded share to the parent (that is, the share that the parent can decrypt) so that the parent can start working on the decryption. The node then sends all the remaining shares except the Sth share, while calculating its own encryption of the Sth share. Finally, when the encryption is ready, the node sends the Sth share as well.

Also, consider that due to the binomial tree structure, all the leaves are created at about the same time, so they will start to send their message to the parent at about the

same time resulting in a more or less round-based aggregation protocol. This makes the time complexity of one such aggregation round in which the aggregation moves up one level (starting from the leaves) $E + T + L$, where E is the encryption/decryption time of a share, T is the transmission time of an encrypted share, and L is the network latency (assuming $E+T > ST$ and that the cost of homomorphic addition is negligible). Note that the actual algorithm does not rely on the existence of synchronized aggregation rounds. In fact, in realistic environments these rounds often overlap if, for example, a node finishes sooner due to losing its children. The rounds are merely an emergent property in reliable environments, a side-effect of using binomial trees as our tree topology.

Another possibility for optimization is based on the observation that shares that would be encrypted with the public keys of the ancestors of the root do not need to be encrypted at all, so the root in fact performs only a single decryption.

6.3.5 Variants

Apart from optimizations, one can consider slightly modified versions of the algorithm that can be useful for trading off security and robustness or that allow for a minimal involvement of a central server.

The first variation—that we will actually utilize during our evaluation in Section 6.6—is setting a lower bound on the size of the subtree that we accept. Indeed, we have to be careful when publishing a sum based on too few participants. Let us denote by R the minimal required number of actual participants ($S \leq R \leq N$). Let the nodes pad their messages with an (unencrypted) integer n indicating the number of nodes its data is based on. When the node exactly $S - 1$ steps away from the root (thus in the trunk) is about to send its message, it checks whether $n + S - 1 \geq R$ holds (since the remaining nodes towards the root have no children except the one on this path). If not, it sends a failure message instead. The nodes fewer than $S - 1$ steps away from the root transmit a failure message if they receive one, or if they fail to receive any messages. This way, no nodes can decode the sum of a set that is not large enough.

On a different issue: one can ask the question whether the trunk is needed, as the protocol can be executed on any tree unmodified. However, having no trunk makes it easier to steal information about subtrees close to the root. If the tree is well-balanced and the probability of failure is small, these subtrees can be large enough for the stolen partial

sums to not pose a practical privacy problem in certain applications. The advantages include a simpler topology, a faster running time, and increased robustness.

Another option is to replace the top $S-1$ nodes with a central server. To be more precise, we can have a server simulate the top $S-1$ nodes with the local values of these nodes set to zero. This server acts as the root of a 2-trunked tree. From a security point of view, if the server is corrupted by a semi-honest adversary, we have the same situation when the top $S-1$ nodes are corrupted by the same adversary. As we have shown in Section 6.4.1, one needs to corrupt at least S nodes in a chain to gain any extra advantage, so on its own the server is not able to obtain extra information other than the global sum. Also, the server does not need more computational capacity or bandwidth than the other nodes. This variation can be combined with the size propagation technique described above. Here, the child of the server can check whether $n \geq R$ holds.

6.4 Analysis

We first consider the level of security that our solution provides, and we also characterize the complexity of the algorithm.

6.4.1 Security

To steal information, i.e. to learn the sum over a subtree, the adversary needs to catch and decrypt all the S shares of the corresponding message that was sent by the root of the subtree in question. Recall that if the adversary decrypts less than S shares from any message, it still has only a uniform random value due to our construction. To be more precise, to completely decrypt a message sent to node c_1, the adversary needs to corrupt c_1 and all its $S-1$ closest ancestors, denoted by $c_2, .., c_S$, so he can obtain the necessary private keys.

The only situation when the shares of a message are not encrypted with the public keys of S *different* nodes—and hence when less than S nodes are sufficient to be corrupted—is when the distance of the sender from the root is less than S. In this case, the sender node is located in the trunk of the tree. However, decrypting such a message does not yield any more information than what can be calculated from the (public) result of the protocol and the local values (gradients) of the nodes needed to be corrupted for the decryption. This

is because in the trunk the sender of the message in question is surely the only child of the first corrupted node, and the message represents the sum of the local values of all the nodes, except for the ones needed to be corrupted. To put it in a different way, corrupting less than S nodes never gives more leverage than learning the private data of the corrupted nodes only.

Therefore, the only way to steal extra information (other than the local values of the corrupted nodes) is to form a continuous chain of corrupted nodes $c_1, .., c_S$ towards the root, where c_{i+1} is the parent of c_i. This makes it possible to steal the partial sums of the subtrees rooted at the children of c_1. For this reason we now focus only on the $N - S$ vulnerable subtrees not rooted in the trunk.

As a consequence, a threshold adversary cannot steal information if he corrupts at most $S - 1$ nodes. A probabilistic adversary that corrupts each node with probability p can steal the exact partial sum of a given subtree whose root is not corrupted with probability p^S.

Even if the sum of a given subtree is not stolen, some information can be learned about it by stealing the sums of other subtrees. However, this information is limited, as demonstrated by the following theorem.

Theorem 1. *The private value of a node that is not corrupted cannot be exactly determined by the adversary as long as at least one of the S closest ancestors of the node is not corrupted.*

Proof. Let us denote by t the target node, and by u the closest ancestor of t that is not corrupted. The message sent by t cannot be decrypted by the adversary, because one of its shares is encrypted to u (because u is one of the S closest ancestors of t). The same holds for all the nodes between t and u. Therefore the smallest subtree that contains t and whose sum can be stolen also contains u. Due to the nested nature of subtrees, bigger subtrees that contains t also contains u as well. Also, any subtree that contains u also contains t (since t is the descendant of u). Therefore u and t cannot be separated. Even if every other node is corrupted in the subtree whose sum is stolen, only the sum of the private values of u and t can be determined. □

Therefore p^S is also an upper bound on the probability of stealing the exact private value of a given node that is not corrupted.

6.4.2 Complexity

In a tree with a maximal branching factor of B each node sends only one message, and receives at most B. The length of a message (which is an array of S encrypted integers) is $\mathcal{O}(SC)$, where C is the length of the encrypted form of an integer. Let us now elaborate on C. First, as stated before, the sum is represented on $\mathcal{O}(\log M)$ bits, where M is a design choice defining the precision of the fixed point representation of the real values. Let us assume for now that we use the Paillier cryptosystem [83]. In this case, we need to set the parameters of our cryptosystem in such a way that the largest number it can represent is no less than $n = \min(B^S M, NM)$, which is the upper bound of any share being computed by the algorithm (assuming $B \geq 2$). In the Paillier cryptosystem the ciphertext for this parameter setting has an upper bound of $\mathcal{O}(n^2)$ for a single share. Since

$$S \log n^2 = S \log \min(B^S M, NM)^2 \leq 2(S^2 \log B + S \log M), \tag{6.1}$$

the number of bits required is $\mathcal{O}(S^2 \log B + S \log M)$.

The computational complexity is $\mathcal{O}(BSE)$ per node, where E is the cost of encryption, decryption, or homomorphic addition. All these three operations boil down to one or two exponentiations in modular arithmetic in the Paillier cryptosystem. Note that this is independent of N.

The time complexity of the protocol is proportional to the depth of the tree. If the tree is balanced, this results in $S + \mathcal{O}(logN)$ steps altogether.

6.5 Compressing the Gradient

As mentioned in Section 6.3.2, it is essential that we compress the gradient because in a realistic machine learning problem there are at least a few hundred parameters, often a lot more. Encoding and decoding this many floating point numbers with full precision can be prohibitively expensive for our protocol, especially on a mobile device. For this reason, we evaluated the effect of gradient compression on the performance of gradient descent learning. Similar techniques have been used before in a slightly different context [56].

Let us first introduce the exact algorithms and learning tasks we used for this evaluation. As for the learning tasks, we used two data sets. The first is the Spambase binary classification data set from the UCI repository[7], which consists of 4601 records with 57 features. Each of these records belongs to an email that was classified either as spam or

as a regular email. The features that represent a piece of email are based on, for example, word and character frequencies or the length of capital letter sequences within the email. 39.4% of the records are positive examples. 10% of the records were reserved for testing. Each node had one record resulting in a network size of 4140, the remaining part of the data set (461 records) was used for testing. The second dataset we used was based on Reuters articles.[1] It contains 1000 positive and 1000 negative examples, with 600 additional examples used for testing. The examples have 9947 features. The dataset contains Reuters articles and the task is to decide whether a given document is about "corporate acquisitions" or not. The documents are represented by word stem feature vectors, where each feature corresponds to the occurrence of a word. Hence, the representation is very high-dimensional and sparse (that is, each vector contains mostly zeros).

We tested two machine learning algorithms. The first is logistic regression [14]. We used the L2-regularized logistic regression online update rule

$$w \leftarrow \frac{t}{t+1}w + \frac{\eta}{t+1}(p-y)x \tag{6.2}$$

where w is the weight vector of the model, t is the number of samples seen by the model (not including the new one), x is the feature vector of the training example, y is the correct label (1 or 0), p is the prediction of the model (probability of the label being 1), and η is the learning parameter. We generalize this rule to mini-batches of size E as follows:

$$w \leftarrow \frac{t}{t+E}w + \frac{\eta}{t+E}\sum_{i=1}^{E}(p_i - y_i)x_i \tag{6.3}$$

where $(y_i - p_i)x_i$ is supposed to be calculated by the individual nodes, and summed using Algorithm 6.8. After the update, t is increased by E instead of 1. η was set to 10^5. The second algorithm was linear SVM [95]. The setup is very similar to that of logistic regression, only the batch update rule we used is

$$w \leftarrow \frac{t}{t+E}w + \frac{\eta}{t+E}\sum_{i=1}^{E}[y_i w^T x_i < 1]y_i x_i, \tag{6.4}$$

where $[\cdot]$ is the Iverson bracket notation (1 if its parameter is true, otherwise 0). Here y is

[1] `http://download.joachims.org/svm_light/examples/example1.tar.gz`

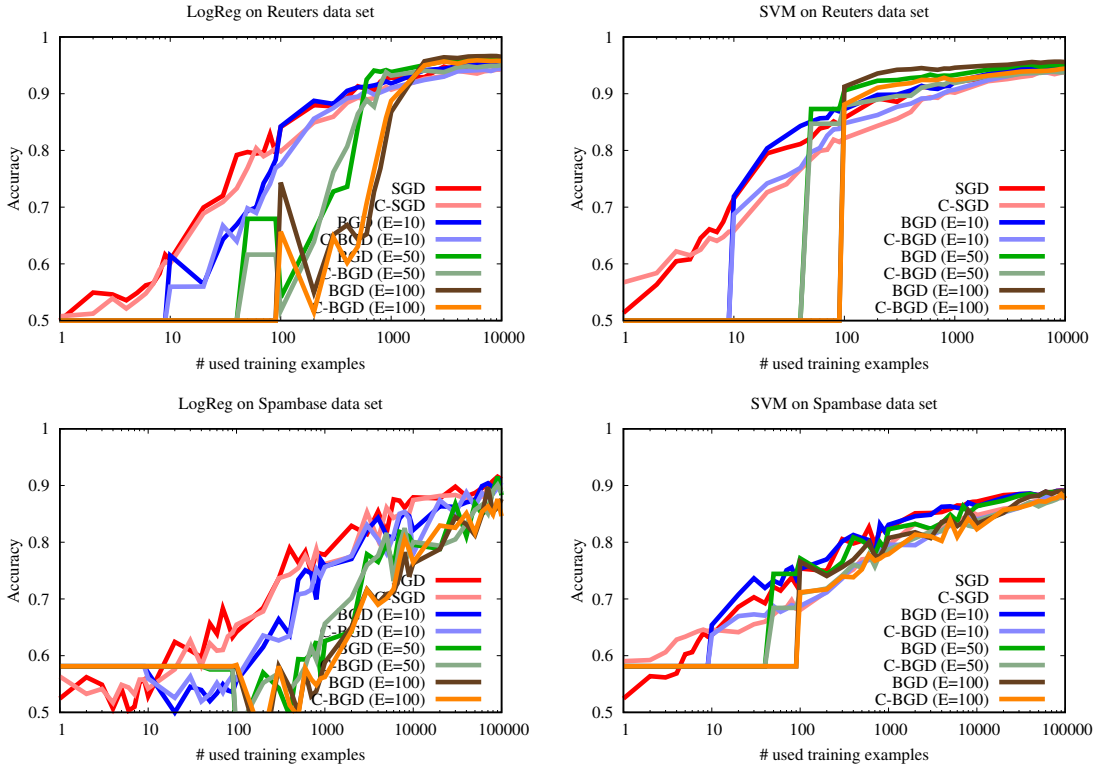


Figure 6.1. Classification accuracy of the compressed gradient update on the data sets with various batch sizes.

the correct label as before, however, now $y \in \{-1, +1\}$.

The compression method we used was the following. All the individual gradients within the mini-batch were computed using a 32-bit floating point representation. These gradients were then quantized by mapping each attribute to one of only three possible values: 1, 0 and -1. This mapping was achieved by stochastic quantization. The quantized value requires only 2 bits to encode, a dramatic compression compared to the original floating point representation of 32 bits. In fact, since we have only three levels, theoretically only a trit is needed for the encoding. We exploit this fact when summing the gradients: the upper bound of the sum of trits (represented on two bits) is lower than the sum of two-bit values. These compressed gradients were then used in equations (6.3) and (6.4) where no further compression is applied.

We ran experiments with all the four possible combinations of learning algorithms and datasets, using four different batch sizes: $E = 1$, 10, 50, and 100. The results are shown

in Figure 6.1. The figure shows how the classification accuracy evolves as a function of the number of training examples seen. Accuracy is the proportion of correctly classified instances; that is, the sum of the number of the true positive and the true negative test examples divided by the size of the test set. The databases are well balanced with respect to the class labels, making this metric adequate. The compressed versions are indicated by the "C-" prefix. It is clear that in these experiments there is virtually no difference between the compressed and original versions. This result is quite striking, and is probably explained by the fact that mini-batch gradients still contain a lot of noise compared to the full gradient even if they are computed exactly.

In the following, we assume that gradient attributes can be safely encoded in two bits only.

6.6 Experimental Evaluation

In this section, our goal is to demonstrate that the decentralized secure mini-batch gradient search we proposed is practically viable; that is, the running time in a real system with realistic parameters is acceptable and the learning algorithm offers a good performance under realistic failure conditions.

Recall that the solution we proposed consists of three components. The first is the overlay tree building algorithm, which defines the mini-batches. The second is the secure sum computation algorithm, which assumes that an overlay tree is given. The third is the applied machine learning algorithm. These three components are modular, different solutions for any of these components can be combined.

We exploit this modularity in our experimental evaluation. First, for each scenario we determine the time that is needed to encrypt and decrypt the messages defined by our secure sum protocol based on the Paillier cryptosystem. We then plug these values into a simulation of the tree building and aggregation protocols under realistic network and failure conditions. The end result of this simulation is a series of mini-batch sizes that are defined by the effective tree-sizes we observe, along with a time-stamp for each mini-batch that depends on the simulated duration of the secure mini-batch gradient computation. Finally, we use these series of mini-batch sizes as well as their timing to assess the performance of the machine learning algorithm in our system. This is possible, because the only important factor for machine learning is the effective size of the tree in

Table 6.1. Parameter setups for realistic simulations. Time consumption of the protocol. Rate of allowed trees based on distributions.

Parameter setups								Time consumption (seconds)					Results
S	#feature D (f)		max tree size (N)	bits per feature (b)	key size (n)	blocks $\lceil \frac{fb}{n} \rceil$	message size to parent $S\,2n\lceil \frac{fb}{n} \rceil$	encrypt / decrypt a block	send plaintext model	encrypt $S-1$ shares	one aggregation round	overall time of mini-batch	prob. of good tree
4	10^2	4	19	10	1024	1	8192	0.041	0.103	0.123	0.143	1.847	0.999
					2048	1	16384	0.300	0.103	0.900	0.404	4.451	0.997
		6	67	14	1024	2	16384	0.041	0.103	0.246	0.186	2.850	0.997
					2048	1	16384	0.300	0.103	0.900	0.404	5.466	0.996
	10^4	4	19	10	1024	99	811008	0.041	0.420	12.177	4.362	45.649	0.969
					2048	50	819200	0.300	0.420	45.000	15.305	155.074	0.904
		6	67	14	1024	137	1122304	0.041	0.420	16.851	5.998	74.609	0.951
					2048	69	1130496	0.300	0.420	62.100	21.083	255.624	0.850

each step. We assume that each tree defines a uniform random subset, which is a good approximation if the underlying overlay network is random.

To model the network required for simulating the tree building protocol, we used a real trace of smartphone user behavior. The rest of the parameters defining the computational cost and network utilization were set based on realistic examples. We used PeerSim [79] for our simulations.

6.6.1 Time Consumption

As mentioned above, we first describe the time consumption of the most important operations in our protocol. In order to do this, we carefully have to consider the size of each message that is transmitted and the time needed for encrypting and decrypting these messages. We performed these calculations in a number of scenarios with different parameters that represent interesting use cases. The different scenarios as well as the corresponding message sizes and the amount of time needed to complete a number of different tasks are shown in Table 6.1. In the following we explain these scenarios and the computed values

within these scenarios in detail.

For all the trees that we would like to build we fix $S = 4$, as indicated in the first column. This is our security parameter, introduced in Section 6.3.1. The value of $S = 4$ represents a good tradeoff between efficiency and the offered level of security. The binomial tree parameter D (the number of rounds used to build the tree) was set to 4 or 6, giving us the maximum tree sizes of 19 and 67, computed by the formula $N = 2^D + S - 1$, which was explained in detail in Section 6.3.1. The motivation for these settings is that our preliminary experiments with our machine-learning application told us that increasing the mini-batch size beyond 67 is not beneficial. The lower value of 19 is motivated by the fact that smaller trees do not offer a sufficient level of privacy, since the sum is computed based on too few nodes. Also, in a very small tree, the trunk represents a considerable proportion of the tree, which limits the possibilities for parallelization; hence the efficiency is not ideal.

The number of features in the learning problem was modeled to be 100 or 10,000. This setting accommodates the number of features in our datasets that are 57 for the Spambase dataset and 9947 for the Reuters dataset (see Section 6.5). Note that we rounded the number up to the closest power of 10 so that we have a 100 times scaling factor, which makes comparison more intuitive.

Based on the tree size N and the quantization parameter m, we can compute the number of bits (b) needed to represent a share of one element of the secret-shared gradient vector. As explained in Section 6.3.3 in detail, the formula is given by $b = \lceil \log_2(1 + N^2 m) \rceil$. We used $m = 2$ based on our results on compressing the gradient vector in Section 6.5. The next column shows the key size (or block size) n, a parameter for the Paillier cryptosystem that defines the level of security. We examine the common values 1024 and 2048. Note that 2048 is currently recommended for sufficient security[2].

Based on the parameters we have already defined, we can now compute the number of blocks to be encoded per gradient share: $\lceil \frac{fb}{n} \rceil$. Finally, let us compute the message size to be sent by a node in the tree to its parent. According to the protocol, this message is composed of the S encrypted shares of the compressed gradient. The size of the message is $S\,2n\lceil \frac{fb}{n} \rceil$ bits. This is due to the fact that the size of an encrypted block is $2n$, and we need $\lceil \frac{fb}{n} \rceil$ blocks per share.

[2] https://www.keylength.com/

We have now computed almost all the values necessary to determine the time consumption of some important operations of the protocol. The last bit of information required for that is the time consumption of encoding a single block. The Paillier encryption and decryption time of a block is experimentally measured using an unoptimized Java implementation based on BigIntegers on a real Android device (Samsung SM-T280). This can be considered a worst case scenario because the implementation we used has a lot of room for optimization and the device itself is not an up-to-date model. Both the encryption and decryption take 0.041 s with a 1024 bit key and 0.300 s with a 2048 bit key.

Sending the model in plaintext from the parent to the child is required when building the tree. We assume single precision floating-point arithmetic (32 bits) so the sizes of the linear models are 3,200 bit and 320,000 bit for 100 and 10,000 features, respectively. The actual sending time is given by the 1 Mbps bandwidth we allow between online nodes and assuming a 100 ms latency. After receiving the model in plaintext the node instantly starts encrypting $S - 1$ shares as discussed in Section 6.3.4. This takes $S - 1$ times the encryption time of all the required blocks. The computed values are shown in Table 6.1.

The next column shows the time of one aggregation round, that is, the time needed for a child node to propagate information up to the parent. In Section 6.3.4 we described a number of variants of the protocol that involve different optimizations compared to the basic variant. Here, we assume the variant, in which children in the tree start encrypting their share while they simultaneously upload the other $S - 1$ shares to their parents. In all our scenarios uploading $S - 1$ shares is faster than encrypting one share. This means that the time needed for one aggregation round is the time of encoding one share plus the time of uploading this share (which consists of transmission time and network latency). The column indicating the time needed for one aggregation round shows this value for each parameter setting.

The column that corresponds to the overall mini-batch time sums up all the required times for completing the mini-batch, assuming the network is error free. This involves sending the plain text model to the children down the tree during tree building as well as the aggregation rounds up to the root. These operations are performed for each level of the tree; note that the depth of the whole tree is $D + S - 1$. The time of encoding $S - 1$ shares also needs to be added because the leaves must first complete this encoding before starting the first aggregation round. If nodes can fail, in an actual run these times may be slightly longer because of the delay introduced by the failure detector, but they may

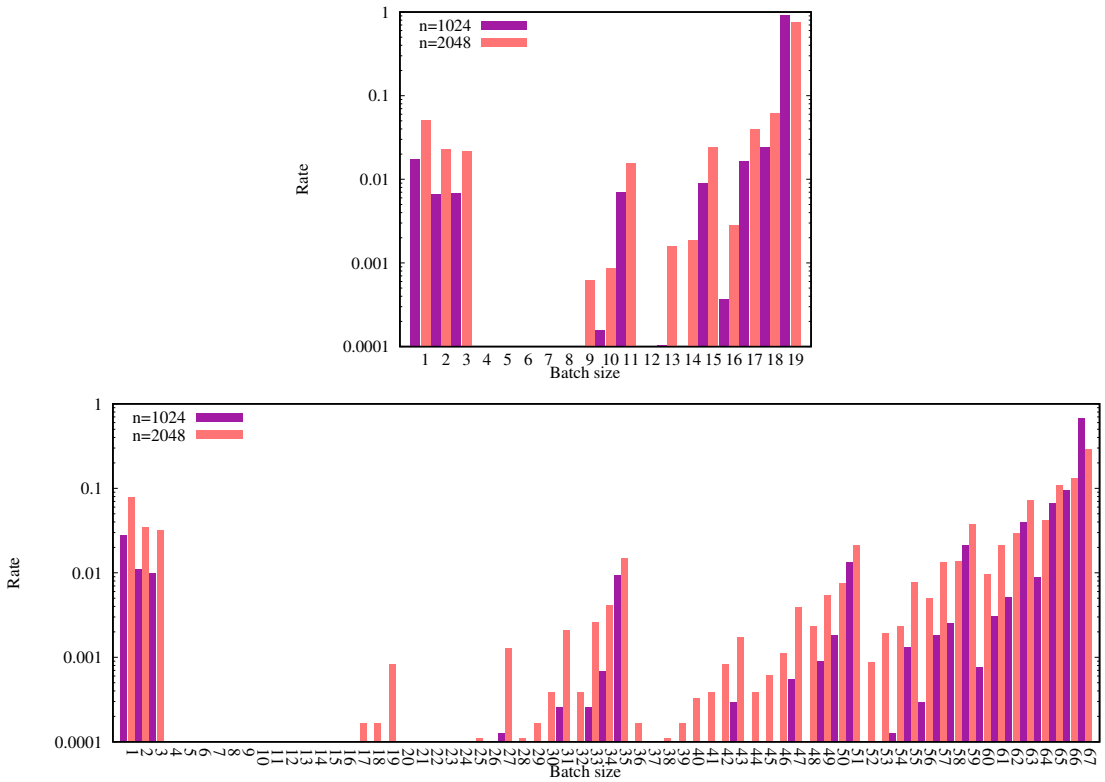


Figure 6.2. Distribution of effective mini-batch sizes for scenario of 10,000 features. The histograms have a logarithmic scale.

also be slightly shorter, due to a smaller tree. Our simulations account for these effects. Note that we ignored the time consumption of the single gradient update step that has to be performed as well at every node. This is because the encryption operation is orders of magnitude slower than the gradient update.

6.6.2 Simulating Tree Building

All of our experiments were run on top of the churn trace described in Section 3.3. Here, we should mention that we can simulate the case where a participating phone is required to have at least a certain battery level. From the point of view of churn, though, the worst case is when when phones with any battery level are allowed to join, because this results in a more dynamic scenario. The network size was 100,000. The membership overlay network was implemented by independently assigning 100 randomly selected outgoing

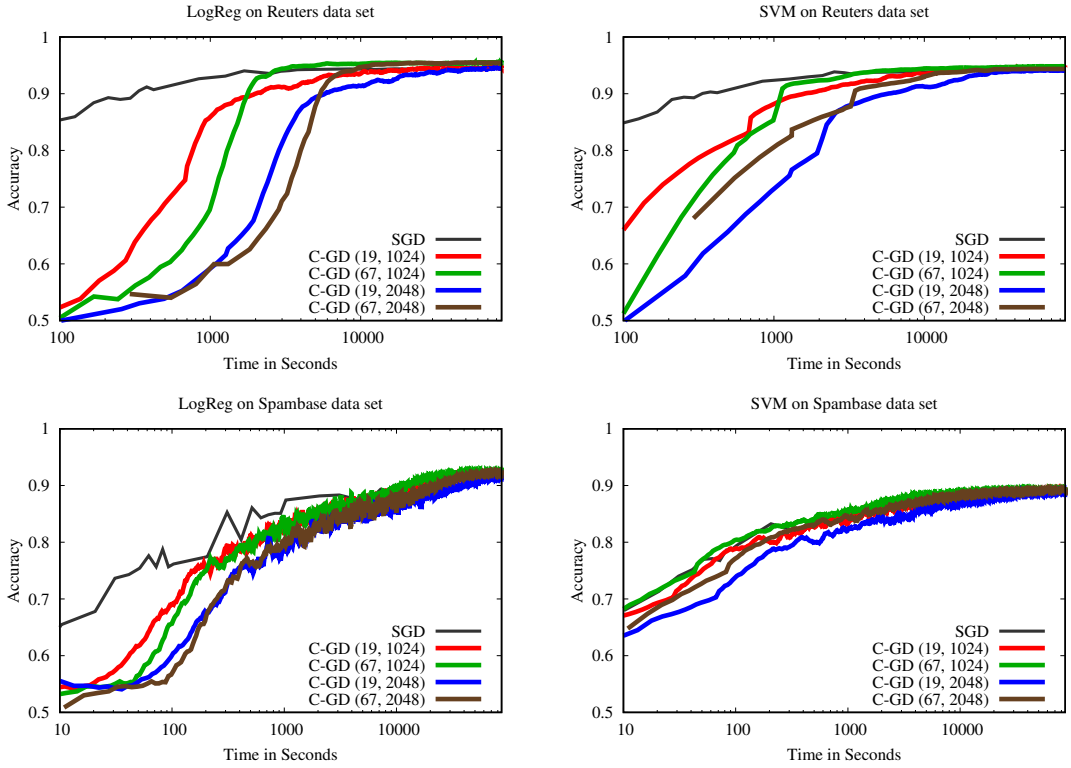


Figure 6.3. Classification accuracy of the compressed gradient update on the data sets based on trace-based simulation. We vary the key size (1024 or 2048) and maximum tree size (19 or 67).

neighbors to each node and then dropping the directionality of the links. This network forms the basis of tree building, the tree neighbors are selected from these nodes. We assume that each node maintains an active TCP connection with its neighbors as suggested in [91]. If a node fails, its neighbors will detect this only with a one second delay. The neighbor set is constant in our simulations; that is, when a neighbor fails it remains on the list and it is reconnected when it comes back online. The size of our neighbor set was large enough for the overlay network to remain connected.

Initially a random online node is picked from the network at time 0:00 and we simulate building the first tree using that node as root. This simulation involves building the tree and propagating the aggregated gradient up to the root, simulated based on the time consumption of these operations described previously. When this is completed, we pick a new random node that is online at the time of finishing the first mini-batch and simulate a new mini-batch round. We repeat this procedure until the end of the simulated day. With

this methodology, we record the effective mini-batch sizes (which determines the number of gradients the sum of which the root actually received) and we examine the distribution of these effective mini-batch sizes.

The empirical distributions of the effective mini-batch sizes for the case of 10,000 features are shown in Figure 6.2. In every scenario we simulated a sample of at least 15,000 tree building attempts. The figure shows the histograms based on these samples. The histograms use a logarithmic scale to better illustrate the structure of the distribution. However, note that most of the probability mass belongs to the largest effective sizes. For 100 features almost all the trees are complete due to the very quick building times (not shown). The relatively high probability mass for tree sizes 1, 2 and 3 are due to the vulnerability of the trunk.

In our experiments, we used the variant of the protocol that limits the effective tree size from below, as explained in Section 6.3.5. We accepted a mini-batch for gradient update only if its size was greater than or equal to $\lfloor \frac{N}{2} \rfloor$. The reason is that smaller trees represent reduced privacy. We call such trees a "good tree". The last column of Table 6.1 contains the probability of getting a good tree. Clearly, only a very small proportion of tree building attempts are unsuccessful.

6.6.3 Machine Learning Results

We now present our results with the actual learning tasks. The setup for the learning problems is identical to that presented in Section 6.5. The only difference is that now the batch sizes used in each update step are variable and depend on the effective batch size that is obtained in our tree building simulation based on the smartphone trace, and the time needed to complete a given mini-batch is also given by the output of the simulation. The results are shown in Figure 6.3. Note that the horizontal axis of the plots now shows the time, covering one full day. It is clear that the main factor for convergence speed is the encryption key size, with 2048 being significantly slower than 1024. This could be expected based on Table 6.1 as well. We see that our example learning tasks can converge within one day, which is adequate for many practically interesting learning problems.

6.7 Conclusion

We proposed a secure sum protocol to prevent the collusion attack in gossip learning. The main idea is that instead of SGD we implement a mini-batch method and the sum within the mini-batch is calculated using our novel secure algorithm. We can achieve high levels of robustness and good scalability in our tree building protocol through exploiting the fact that the mini-batch gradient algorithm does not require the sum to be precise. The algorithm runs in logarithmic time and it is designed to calculate a partial sum in case of node failures. It can tolerate collusion unless there are S consecutive colluding nodes on any path to the root of the aggregation tree, where S is a free parameter. The algorithm is completely local therefore it has the same time-complexity independently of the network size.

We evaluated the protocol in realistic simulations where we took into account the time needed for encryption and message transmission, and we used a real smartphone trace to simulate churn. We demonstrated on a number of learning tasks that the approach is indeed practically viable even with a key size of 2048. We also demonstrated that the gradients can be compressed by an order of magnitude without sacrificing prediction accuracy.

Contribution

In this chapter, most of the above-presented results rely heavily on the achievements of Gábor Danner. His work includes a robust secure sum protocol and a theoretical proof about its capability of preventing the collusion attack. The fully distributed mini-batch gradient descent that relies on the construction of a k-long-trunked binomial overlay tree is based on the secure sum protocol that was also developed by Gábor Danner. Here, it was crucial to demonstrate that the proposed overlay tree building is practically viable. Accordingly, the author of this dissertation performed a detailed empirical evaluation on top of the smartphone churn trace. This result has led to further progress in understanding the smartphone trace that was described in Chapter 3.

CHAPTER 7

Summary

The main aim of the thesis was to tackle a number of diverse problems on mobile gossip learning in order to make gossip learning more suitable for performing distributed data mining. In this chapter, we give a brief summary of each chapter (chapters 3-6). At the end of each section the results that the author regards as his main contributions are presented in an itemized list.

7.1 Smartphone Trace

We proposed a real smartphone trace for simulating fully distributed protocols. The presented trace was collected by a locally developed Android app. We highlighted the trace main properties, and we examined its free parameter about battery-related requirements. Accordingly, we can obtain two significantly different simulation scenarios. In the worst case churn scenario, we can state the expectation for a participant smartphone having at least a certain battery level. In another approach, we expect the device to be on charger for a more user and phone friendly scenario. We also proposed a time-inhomogeneous Markovian model based on the collected data in which the conditional probability distributions of session lengths are captured by a set of the actual observations in the data that

we resample when creating synthetic traces of users to model churn. We found that the model captures observed availability as well as the behavior of push-pull gossip broadcast. Since the initial release of our data collection app, we have collected a very large trace involving millions of individual measurements. We took great care to clean this data. We proposed a method to correct failed NAT measurements. We extended the application to collect data related to direct P2P capabilities based on a basic WebRTC implementation. Then, we presented a brief introduction on its base statistics and properties.

The main contributions of the author are

- A trace for simulating realistic smartphone churn.
- An introduction on the base statistics and properties of a smartphone trace.
- A time-inhomogeneous Markovian model as an alternative to simulating churn.
- A data cleansing method to correct failed NAT measurements.

7.2 Dimension Reduction Methods

Here, we presented a fully distributed algorithm for selecting a good random projection and a gossip-based fully distributed robust SVD algorithm that can be used to reduce the dimensionality of a machine learning problem. We also proposed a hybrid approach which combines random projection selection with a fully distributed SVD solver. We evaluated these algorithms over a real smartphone trace and we took into account the energy problem in mobile computing. We conclude that the proposed random projection selection algorithm is very fast and efficient, but the quality of the dimension reduction is slightly lower than that of SVD. However, the SVD algorithm converges in a time proportional to the original dimensionality of the problem, which can be quite slow. Our hybrid approach combines the advantages of the two approaches and it can provide a good quality dimension reduction, independently of the time available for convergence.

The main contributions of the author are:

- An algorithm that builds on searching for good random projections.

- A hybrid method that combines the advantages of random projections and SVD.
- The proposed methods on dimension reduction were evaluated in a comparative study.

7.3 Management of Random Walks

Here, we introduced two random walk management services with quite different goals. Both our presented studies are based on gossip learning and evaluated over a real smartphone trace.

We proposed the Single Random Walk Service for a differentially private random walk management. It relies on maintaining a very small shared state through gossip. The approach is fully decentralized and only incurs a relatively small overhead if the random walk has a large state. We proposed to implement SGD on top of this. Hence, every data record is visited only a limited number of times and this keeps the privacy budget low. We demonstrated that the proposed method is agile, efficient and long-lived. We found that the protocol is robust to its main parameter and we obtained an acceptable performance even in an unrealistic, extreme scenario. We also presented the Multiple Random Walk Service to maintain $O(n)$ random walks over an overlay network, where the random walks represent independent decentralized tasks that might belong to different users. The protocol follows a three-level design where problems not solved at a lower level get escalated to the next level. We demonstrated that in all the scenarios we tested the vast majority of failures are dealt with at the lowest level, which is purely local and therefore scalable. Only a small fraction of the problems get escalated to level two, which is based on a broadcast primitive that has a small overhead. Level three, the central control by the task owner, was reached only a few times. We also demonstrated that the speed of the random walks is close to optimal.

The main contributions of the author are:

- A method for managing a single random walk.
- A method for managing multiple random walks.

7.4 Mini-Batch Gradient Descent

We proposed a secure sum protocol to prevent the collusion attack in gossip learning. The main idea is that we implement a mini-batch method and the sum within the mini-batch is calculated using our novel secure algorithm. We can achieve high levels of robustness and good scalability in our tree building protocol by exploiting the fact that the mini-batch gradient algorithm does not require the sum to be precise. The algorithm runs in logarithmic time and it has to calculate a partial sum in the case of node failures. It can tolerate collusion unless there are S consecutive colluding nodes on any path to the root of the aggregation tree, where S is a free parameter. The algorithm is completely local therefore it has the same time-complexity independently of the network size. We evaluated the protocol in realistic simulations where we took into account the time needed for encryption and message transmission, and we used a real smartphone trace to simulate churn. We demonstrated on a number of learning tasks that the approach is indeed practically viable. We also demonstrated that the gradients can be compressed by an order of magnitude without sacrificing prediction accuracy.

The main contribution of the author is:

- The empirical evaluation of a fully distributed mini-batch gradient descent that is based on a secure sum protocol and the construction of a k-long-trunked binomial overlay tree on top of the smartphone trace.

CHAPTER 8

Összefoglaló

A tézis fő célkitűzéseként számos, változatos okostelefonos pletykaalapú tanulási problémára mutattunk be robusztus módszereket annak érdekében, hogy a pletykaalapú tanulás még kedvezőbb alternatívája legyen a jelenleg alkalmazott, központi szerveren történő adatbányászatnak. Az alábbiakban röviden összefoglaljuk az eredményeket, majd minden fejezet végén kiemeljük azokat az eredményeket, amelyeket a szerző a saját hozzájárulásának tekint.

8.1 Okostelefonos trace

A teljesen elosztott protokollok kiértékeléséhez mutattunk be egy valós okostelefonos felhasználói viselkedési adatokon alapuló trace-t. Az ehhez szükséges adatokat egy saját fejlesztésű Android alkalmazás segítségével gyűjtöttük össze. A fejezetben részletesen kiemeltük a trace főbb jellemzőit. Megvizsgáltuk az akkumulátor töltöttséggel kapcsolatos szabad paraméterét. Ennek deklarálása alapján két jelentősen különböző forgatókönyv szimulálható. A fel- és lekapcsolódások szempontjából legdinamikusabb forgatókönyvet akkor érhetjük el, ha nem szabunk meg semmilyen feltételt a töltöttséggel kapcsolatban. Ezzel szemben ha készülék- és felhasználóbarát elosztott alkalmazás mögött álló pro-

tokollok kiértékelését szeretnénk elvégezni, akkor csak olyankor tekintünk egy eszközt elérhetőnek, ha a töltőhöz és a hálózathoz is kapcsolódva van. Ezen felül bemutattunk egy időben inhomogén Markov-modellt, amely alternatívája lehet a le- és felcsatlakozások szimulációjának, és amely szintén az okostelefonos alkalmazás által gyűjtött adatokon alapul. Ennek elérése érdekében megvizsgáltuk az elérhetőségi állapotok hosszának a feltételes valószínűségi eloszlásait az előző állapot hossza illetve az aktuális napszak függvényében. A megfigyelt eloszlás újramintavételezésével szintetikus trace-t hoztunk létre a felhasználók viselkedéséről. Az adatgyűjtő alkalmazás megjelenése óta több millió adatrekord gyűlt össze. Ilyen mértékű adatmennyiség esetén fontos volt, hogy figyelmet fordítsunk a megfelelő adattisztításra. Javasoltunk egy módszert, amely javítani tud a hibás NAT méréseken. Bemutattuk az ajánlásainkat az évek során lezajlott adatgyűjtés tanulságai alapján. Továbbá kiegészítettük az adatgyűjtést tényleges P2P kapcsolat-kiépítési próbálkozás mérésével. Ezekről jól érthető, tömör leírást és alapvető statisztikákat is közöltünk.

A szerző kapcsolódó főbb eredményei:

- Valós okostelefon-felhasználói szokásokon alapuló trace a le- és felkapcsolódások szimulációjára.
- Az okostelfonos trace jellemzése és alapvető statisztikák bemutatása.
- Időben inhomogén Markov-modell a le- és felkapcsolódások szimulációjára.
- Adattisztító eljárás hibás NAT mérések javítására.

8.2 Dimenziócsökkentő módszerek

Ebben a fejezetben jó véletlen projekciós mátrix kiválasztásán alapuló teljesen elosztott módszert mutattunk be. Ezen felül bemutattunk egy pletykaalapú teljesen elosztott SVD metódust gépi tanulási problémák dimenziójának csökkentésére. Továbbá bemutattunk egy hibrid módszert is, amely ötvözi az előbbi kettő előnyeit. Kiértékeltük ezeket a módszereket okostelefonos trace-en alapuló szimulációk révén, több közismert tanító halmazon. Ezek során arra jutottunk, hogy a véletlen projekciók közül választó módszer gyors és hatékony, de minőségi szempontból az SVD valamivel előrébb tart. Ugyanakkor az

SVD számára nagyságrendekkel több időre van szükség a konvergencia eléréséhez, mindaddig pedig elmarad a teljesítménye. A hibrid módszerünk azonnal elfogadható teljesítményt tud nyújtani, majd az SVD konvergenciájával közel azonos időben javítani tud a kezdeti teljesítményén. Tehát a legtöbb időpontban kiemelkedő teljesítményre képes.

A szerző kapcsolódó főbb eredményei:

- Egy olyan elosztott algoritmus, amely a legjobb véletlen projekciós mátrixot keresi.
- Egy hibrid algoritmus, amely egyesíti az előnyeit a véletlen projekciós módszernek és az SVD-nek.
- Az elosztott SVD algoritmus dimenziócsökkentéssel kapcsolatos kiértékelése, bemutatott algoritmusokkal való összevetése.

8.3 Véletlen séták menedzselése

Ebben a tézispontban két véletlenséta-menedzselő hálózati szolgáltatásra tettünk javaslatot két jelentősen eltérő feladatra. A módszereket okostelefonos trace-en értékeltük ki.

Az első feladat esetében az a célunk, hogy az egész hálózatban lehetőleg minél kevesebb, legjobb esetben egyetlen véletlen séta legyen. Ez azért fontos, mert így az SGD módszer esetén a privacy büdzsé alacsony marad. Ezzel pedig elérhető a differential privacy, amennyiben mindemellett az SGD modell a megfelelő mértékű zajjal lett terhelve. Ehhez egy teljesen elosztott véletlenséta-menedzselő eljárást mutattunk be, amelyre egy okostelefon-hálózatban teljesül, hogy agilis, hosszú-életű és hatékony. Ezen felül kijelenthető az is, hogy a javasolt módszer robusztus a fő paraméterére és még a valóságtól elrugaszkodott, extrém forgatókönyv mellett is elfogadható teljesítményre képes.

A második probléma esetében egy olyan környezetben oldottuk meg a véletlen séta menedzselését, ahol minden felhasználó számára elérhető az a szolgáltatás, hogy egy egyedi véletlen sétát indítson a hálózatban. Tehát $O(n)$) véletlen séta menedzselésének igényére lehet számítani. Erre a problémára mutattunk be egy olyan módszert, amely három konceptuális szinttel rendelkezik, annak eldöntésére, hogy szükséges-e egy séta újraindítása. A lokális szinttől lépünk tovább egy kis költségű kollaboratív szintre, majd legvégső esetben a központi vezérlés szintjére. Az általunk vizsgált összes forgatókönyv esetében a hibák jelentős hányadával az első, lokális szinten megbirkóztak a hálózat

résztvevői. A hibák kis része eszkalálódott a második szintre, és a teljes kiértékelés során csupán néhány esetben jutottunk el a harmadik, központi irányítású szintre. Ezenkívül demonstráltuk a véletlen séták sebességét, amely nagyon közel van az optimálishoz.

A szerző kapcsolódó főbb eredményei:

- Egy teljesen elosztott módszer egyetlen véletlen séta menedzselésére.
- Egy robusztus módszer $O(n)$ véletlen séta menedzselésére egy olyan hálózatban, ahol n résztvevő van.

8.4 Mini-batch gradiens módszer

Ebben a fejezetben egy olyan biztonságos összegző protokollt mutattunk be, amely képes védekezni az összeesküvéses támadás ellen. A protokoll egy teljesen elosztott mini-batch gradiens alapú módszert valósít meg, amely ideiglenesen önszerveződő csoportokat hoz létre a hálózatban. A csoportok kollaboratívan számolják ki a gradienst. A módszer alkalmazása során azt használjuk ki, hogy nem szükséges a csoport minden tagjának végig jelen lennie, mivel a mini-batch gradiens módszer számára nem szükséges, hogy az összeg pontos legyen. Az összeget közösen számoló csoportok úgy alakulnak ki, hogy egy gyökérből induló k hosszú törzzsel rendelkező binomiális fa épül az overlay hálózaton. A fa gyökerének elérhetőnek kell maradnia ahhoz, hogy elfogadható legyen az összeg. Ezért ebben a munkában kiemelkedően fontos volt bemutatni, hogy praktikusan megvalósítható-e egy ilyen overlay hálózat. Az okostelefonos trace-en végrehajtott empirikus vizsgálataink alapján teljesen világossá vált, hogy csak nagyon kis részben sikertelen az ilyen faépítési próbálkozás. Ezenkívül azt is bemutattuk, hogy a gradiens összeg titkosított megvalósítása során nem veszítünk a modell predikciós teljesítményéből.

A szerző kapcsolódó főbb eredménye:

- Egy k hosszú törzzsel rendelkező binomiális fa építésén, és egy biztonságos összegző protokollon alapuló decentralizált mini-batch módszer empirikus kiértékelése okostelefonos trace-en alapuló overlay hálózaton.

References

[1] European commission: General data protection regulation (GDPR). 2018. https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules.

[2] Webrtc 1.0: Real-time communication between browsers. 2018. https://www.w3.org/TR/webrtc/.

[3] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, June 2003.

[4] Waseem Ahmad and Ashfaq Khokhar. Secure aggregation in large scale overlay networks. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM '06)*, San Francisco, CA, USA, November 2006.

[5] Árpád Berta, István Hegedűs, and Márk Jelasity. Dimension reduction methods for collaborative mobile gossip learning. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 393–397, Feb 2016.

[6] Árpád Berta and Márk Jelasity. Decentralized management of random walks over a mobile phone network. In *2017 25th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 100–107, Mar 2017.

[7] Kevin Bache and Moshe Lichman. UCI machine learning repository, 2013.

[8] Ziv Bar-Yossef, Roy Friedman, and Gabriel Kliot. RaWMS – random walk based lightweight membership service for wireless ad hoc networks. *ACM Transactions on Computer Systems*, 26(2):5:1–5:66, June 2008.

[9] Árpád Berta, Vilmos Bilicki, and Márk Jelasity. Defining and understanding smartphone

churn over the internet: a measurement study. In *14-th IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 1–5, Sept 2014.

[10] Árpád Berta, István Hegedüs, and Róbert Ormándi. Lightning fast asynchronous distributed k-means clustering. In *22th European Symposium on Artificial Neural Networks*, ESANN 2014, pages 99–104, 2014.

[11] Danny Bickson, Tzachy Reinman, Danny Dolev, and Benny Pinkas. Peer-to-peer secure multi-party numerical computation facing malicious adversaries. *Peer-to-Peer Networking and Applications*, 3(2):129–144, 2010.

[12] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 245–250, New York, NY, USA, August 2001. ACM.

[13] Ken Birman, Márk Jelasity, Robert Kleinberg, and Edward Tremel. Building a secure and privacy-preserving smart grid. *ACM SIGOPS Operating Systems Review*, 49(1):131–136, January 2015.

[14] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[15] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *Proceedings of the Workshop on Private Multi-Party Machine Learning (NIPS 2016 Workshop)*, Barcelona, Spain, April 2016.

[16] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, August 2012. ACM.

[17] Léon Bottou. Stochastic gradient descent tricks. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2012.

[18] Xiang Cheng, Luoyang Fang, Xuemin Hong, and Liuqing Yang. Exploiting mobile big data: Sources, features, and applications. *IEEE Network*, 31(1):72–79, 2017.

[19] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288. MIT Press, 2007.

[20] Chris Clifton, Murat Kantarcioglu, Jaideep Vaidya, Xiaodong Lin, and Michael Y. Zhu. Tools for privacy preserving distributed data mining. *SIGKDD Explorations Newsletter*, 4(2):28–34, December 2002.

[21] L. D'Acunto, J. A. Pouwelse, and H. J. Sips. A measurement of NAT and firewall characteristics in peer-to-peer systems. In *Proc. 15-th ASCI Conference*, pages 1–5. Advanced School for Computing and Imaging (ASCI), June 2009.

[22] Gábor Danner, Árpád Berta, István Hegedűs, and Márk Jelasity. Robust fully distributed mini-batch gradient descent with privacy preservation. *Security and Communication Networks*, 2018:15, 2018.

[23] Gábor Danner and Márk Jelasity. Fully distributed privacy preserving mini-batch gradient descent learning. In Alysson Bessani and Sara Bouchenak, editors, *Proceedings of the 15th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2015)*, volume 9038 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2015.

[24] Yves-Alexandre de Montjoye, Erez Shmueli, and Alex Sandy Wang, Samuel Sand Pentland. Openpds: Protecting the privacy of metadata through safeanswers. *PloS One*, 9(7):e98790, 2014.

[25] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223—1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[26] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(1):165–202, January 2012.

[27] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *Parallel and Distributed Systems, IEEE Transactions on*, 8(4):424–440, April 1997.

[28] Cynthia Dwork. A firm foundation for private data analysis. *Communications of the ACM*, 54(1):86–95, January 2011.

[29] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proceedings of the 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques - Advances in Cryptology - EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 486–503, St. Petersburg, Russia, May 2006. Springer.

[30] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, volume 3876 of *LNCS*, pages 265–284. Springer Berlin Heidelberg, 2006.

[31] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, August 2014.

[32] Denzil Ferreira, Vassilis Kostakos, and Anind K. Dey. Lessons learned from large-scale user studies: Using android market as a source of data. *International Journal of Mobile Human Computer Interaction*, 4(3), 2012.

[33] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, September 2015.

[34] Franck Gechter, Alastair R. Beresford, and Andrew Rice. Reconstruction of battery level curves based on user data collected from a smartphone. In *Proceedings of the 17th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA 2016)*, pages 289–298, Cham, September 2016. Springer.

[35] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 69–77. ACM, 2011.

[36] Kevin Gimpel, Dipanjan Das, and Noah A. Smith. Distributed asynchronous online learning for natural language processing. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL'10)*, pages 213–222, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[37] Genevieve Gorrell. Generalized hebbian algorithm for incremental singular value decomposition in natural language processing. In Diana McCarthy and Shuly Wintner, editors, *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, Stroudsburg, PA, USA, April 2006. The Association for Computer Linguistics.

[38] Naiyang Guan, Dacheng Tao, Zhigang Luo, and Bo Yuan. Nenmf: An optimal gradient method for nonnegative matrix factorization. *IEEE Transactions on Signal Processing*, 60(6):2882–2898, 2012.

[39] Naiyang Guan, Dacheng Tao, Zhigang Luo, and Bo Yuan. Online nonnegative matrix factorization with robust stochastic approximation. *IEEE Transactions on Neural Networks and Learning Systems*, 23(7):1087–1099, 2012.

[40] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[41] Shuguo Han, Wee Keong Ng, Li Wan, and Vincent C. S. Lee. Privacy-preserving gradient-descent methods. *IEEE Transactions on Knowledge and Data Engineering*, 22(6):884–899, 2010.

[42] István Hegedűs, Árpád Berta, and Márk Jelasity. Robust decentralized differentially private

stochastic gradient descent. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 7(2):20–40, June 2016.

[43] István Hegedűs, Árpád Berta, Levente Kocsis, András A. Benczúr, and Márk Jelasity. Robust decentralized low-rank matrix decomposition. *ACM Transactions on Intelligent Systems and Technology*, 7(4):62:1–62:24, May 2016.

[44] István Hegedűs, Gábor Danner, and Márk Jelasity. Gossip learning as a decentralized alternative to federated learning. In José Pereira and Laura Ricci, editors, *19th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2019)*, pages 74–90. Springer International Publishing, 2019.

[45] István Hegedűs, Gábor Danner, and Márk Jelasity. Decentralized recommendation based on matrix factorization: A comparison of gossip and federated learning. In Peggy Cellier and Kurt Driessens, editors, *International Workshops of ECML PKDD 2019, Decentralized Machine Learning at the Edge*, number 1167 in Communications in Computer and Information Science, page 317–332. Springer Nature Switzerland AG, 2020.

[46] István Hegedűs and Márk Jelasity. Distributed differentially private stochastic gradient descent: An empirical study. In *Proceedings of the 24th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'16)*, Heraklion, Greece, February 2016. IEEE Computer Society.

[47] István Hegedűs, Márk Jelasity, Levente Kocsis, and András A. Benczúr. Fully distributed robust singular value decomposition. In *Proceedings of the 14th IEEE International Conference on Peer-to-Peer Computing (P2P 2014)*. IEEE, 2014.

[48] Sibren Isaacman, Stratis Ioannidis, Augustin Chaintreau, and Margaret Martonosi. Distributed rating prediction in user generated content streams. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, pages 69–76, Chicago, IL, USA, October 2011. ACM.

[49] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[50] Gian Paolo Jesi, Alberto Montresor, and Maarten van Steen. Secure peer sampling. *Computer Networks*, 54(12):2086–2098, 2010.

[51] Raul Jimenez, Gunnar Kreitz, Björn Knutsson, Marcus Isaksson, and Seif Haridi. Integrating smartphones in spotify's peer-assisted music streaming service. Technical Report diva-134609, KTH, Stockholm, Sweden, 2013.

[52] Yu Jin, Nick Duffield, Alexandre Gerber, Patrick Haffner, Wen-Ling Hsu, Guy Jacobson, Subhabrata Sen, Shobha Venkataraman, and Zhi-Li Zhang. Characterizing data usage patterns in a large cellular network. In *Proceedings of the 2012 ACM SIGCOMM Workshop on*

Cellular Networks: Operations, Challenges, and Future Design (CellNet'12), pages 7–12, New York, NY, USA, 2012. ACM.

[53] Ian T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York, 2nd edition, 2002.

[54] Richard Manning Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 565–574, Washington, DC, USA, 2000. IEEE Computer Society.

[55] David Kempe and Frank McSherry. A decentralized algorithm for spectral analysis. In *Proceedings of the 36th Symposium on Theory of Computing (STOC)*, pages 561–568, Chicago, IL, USA, June 2004. ACM.

[56] Jakub Konecný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *Proceedings of the Workshop on Private Multi-Party Machine Learning (NIPS 2016 Workshop)*, Barcelona, Spain, April 2016.

[57] Satish Babu Korada, Andrea Montanari, and Sewoong Oh. Gossip pca. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 209–220. ACM, June 2011.

[58] Gunnar Kreitz and Fredrik Niemela. Spotify – large scale, low latency, P2P Music-on-Demand streaming. In *Tenth IEEE Intl. Conf. Peer-to-Peer Computing (P2P'10)*, pages 1–10. IEEE, 2010.

[59] Jeremy Kubica, Sameer Singh, and Daria Sorokina. Parallel large-scale feature selection. In Ron Bekkerman, Mikhail Bilenko, and John Langford, editors, *Scaling up Machine Learning: Parallel and Distributed Approaches*, pages 352–370. Cambridge University Press, 2011.

[60] Maciej Kurant, Minas Gjoka, Carter T. Butts, and Athina Markopoulou. Walking on a graph with a magnifying glass: stratified sampling via weighted random walks. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS '11)*, pages 281–292. ACM, June 2011.

[61] Juha Laurila, Daniel Gatica-Perez, Imad Aad, Jan Blom, Olivier Bornet, T.-M.-T Do, Olivier Dousse, Julien Eberle, and Markus Miettinen. The mobile data challenge: Big data for mobile computing research. In *Proceedings of the 10th International Conference on Pervasive Computing*, Newcastle, UK, June 2012. Springer.

[62] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In John Langford and Joelle Pineau, editors, *Proceedings of the 29th International*

Conference on Machine Learning (ICML), pages 81–88, Madison, WI, USA, June 2012. Omnipress.

[63] E. Le Merrer, A.-M. Kermarrec, and L. Massoulie. Peer to peer size estimation in large and dynamic networks: A comparative study. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 7—17, 2006.

[64] Yann Lecun, Corinna Cortes, and Burges Christopher, J.C. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[65] Li Li, Bruce Beitman, Mai Zheng, Xiaorui Wang, and Feng Qin. edelta: Pinpointing energy deviations in smartphone apps via comparative trace analysis. In *Proceedings of the 8th International Green and Sustainable Computing Conference (IGSC 2017)*, pages 1–8, Orlando, Florida, October 2017. IEEE.

[66] Yongjun Liao, Pierre Geurts, and Guy Leduc. Network distance prediction based on decentralized matrix factorization. In Mark Crovella, LauraMarie Feeney, Dan Rubenstein, and S.V. Raghavan, editors, *Proceedings of the 9th International IFIP TC 6 Networking Conference*, volume 6091 of *LNCS*, pages 15–26, Chennai, India, May 2010. Springer.

[67] Qing Ling, Yangyang Xu, Wotao Yin, and Zaiwen Wen. Decentralized low-rank matrix completion. In *Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2925–2928, March 2012.

[68] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, Catalina Island, CA, USA, July 2010.

[69] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing (ICS'02)*, New York, NY, USA, June 2002.

[70] Derek C. MacDonald and Bruce Lowekamp. Nat behavior discovery using session traversal utilities for nat (stun). no. rfc 5780. (2010). `http://www.rfc-editor.org/info/rfc5780`.

[71] Panagis Magdalinos. *Linear and non linear dimensionality reduction for distributed knowledge discovery*. PhD thesis, Athens University of Economics and Business, Greece, 2010.

[72] Panagis Magdalinos, Christos Doulkeridis, and Michalis Vazirgiannis. Enhancing clustering quality through landmark-based dimensionality reduction. *ACM Trans. Knowl. Discov. Data*, 5(2):11:1–11:44, February 2011.

[73] Anna Maria Mandalari, Andra Lutu, Amogh Dhamdhere, Marcelo Bagnulo, and KC Claffy. Tracking the big nat across europe and the u.s. *ArXiv*, abs/1704.01296, April 2017.

[74] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing (PODC '06)*, pages 123–132, New York, NY, USA, July 2006. ACM.

[75] Ueli Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[76] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282, Fort Lauderdale, FL, USA, April 2017. PMLR.

[77] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 19–30, New York, NY, USA, 2009. ACM.

[78] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 2 edition, 1997.

[79] Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In *Proceedings of 9th IEEE Int. Conf. on Peer-to-Peer Comp.*, pages 99–100. IEEE, 2009. extended abstract.

[80] Laurence Moroney. *Firebase Cloud Messaging*, pages 163–188. Apress, Berkeley, CA, USA, 2017.

[81] Juan A. M. Naranjo, Leocadio G. Casado, and Márk Jelasity. Asynchronous privacy-preserving iterative computation on peer-to-peer networks. *Computing*, 94(8-10):763–782, 2012.

[82] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571, 2013.

[83] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques - Advances in Cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 1999. Springer.

[84] Amir H. Payberah, Hanna Kavalionak, Vimalkumar Kumaresan, Alberto Montresor, and Seif Haridi. Clive: Cloud-assisted p2p live streaming. In *IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*, pages 79–90, September 2012.

[85] Alex (Sandy) Pentland. Society's nervous system: Building effective government, energy, and public health systems. *Computer*, 45(1):31–38, January 2012.

[86] Fabio Petroni and Leonardo Querzoni. Gasgd: Stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning. In *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, pages 241–248, New York, NY, USA, 2014. ACM.

[87] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In Pinar Yolum, Tunga Güngör, Fikret Gürgen, and Can Özturan, editors, *Proceedings of the 20th International Symposium on Computer and Information Sciences (ISCIS 2005)*, volume 3733 of *Lecture Notes in Computer Science*, pages 284–293. Springer, October 2005.

[88] Arun Rajkumar and Shivani Agarwal. A differentially private stochastic gradient descent algorithm for multiparty classification. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, volume 22 of *Proceedings of Machine Learning Research*, pages 933–941, La Palma, Canary Islands, April 2012. PMLR.

[89] Alfredo Rial and George Danezis. Privacy-preserving smart metering. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society (WPES'11)*, pages 49–60, New York, NY, USA, October 2011. ACM.

[90] Philipp Richter, Florian Wohlfart, Narseo Vallina-Rodriguez, Mark Allman, Randy Bush, Anja Feldmann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. A multi-perspective analysis of carrier-grade nat deployment. In *Proceedings of the 2016 ACM on Internet Measurement Conference (IMC16)*, Santa Monica, CA, USA, November 2016. ACM.

[91] Roberto Roverso, Jim Dowling, and Márk 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.

[92] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. NATCracker: NAT combinations matter. In *Proceedings of 18th Internatonal Conference on Computer Communications and Networks (ICCCN)*, pages 1–7, August 2009.

[93] Jared Saia and Mahdi Zamani. Recent results in scalable multi-party computation. In *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'15)*, volume 8939 of *LNCS*, Pec pod Sněžkou, Czech Republic, January 2015. Springer.

[94] Ali Sayed. Adaptation, learning, and optimization over networks. *Foundations and Trends in Machine Learning*, 7(4-5):311–801, July 2014.

[95] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming B*, 2010.

[96] Shuang Song, Kamalika Chaudhuri, and Anand D. Sarwate. Stochastic gradient descent

with differentially private updates. In *Proceedings of the 1st IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 245–248, Austin, TX, USA, December 2013.

[97] Speedtest. Market reports, 2017.

[98] Nathan Srebro and Tommi Jaakkola. Weighted low-rank approximations. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pages 720–727. AAAI Press, 2003.

[99] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC'06)*, pages 189–202, New York, NY, USA, 2006. ACM.

[100] Daniel Stutzbach, Reza Rejaie, Nick Duffield, Subhabrata Sen, and Walter Willinger. On unbiased sampling for unstructured peer-to-peer networks. *IEEE/ACM Transactions on Networking*, 17(2):377–390, April 2009.

[101] Zoltán Szabó, Vilmos Bilicki, Árpád Berta, and Zoltán Richárd Jánki. Smartphone-based data collection with Stunner using crowdsourcing: Lessons learnt while cleaning the data. In *The Twelfth International Multi-Conference on Computing in the Global Information Technology (ICCGI 2017)*, pages 28–35, Jul 2017.

[102] Zoltán Szabó, Vilmos Bilicki, Árpád Berta, and Zoltán Richárd Jánki. Smartphone-based data collection with Stunner, the reality of peer-to-peer connectivity and web real-time communications using crowdsourcing: Lessons learnt while cleaning the data. *International Journal On Advances in Software*, 11(1-2):120–130, 2018.

[103] Zoltán Szabó, Krisztián Téglás, Árpád Berta, Márk Jelasity, and Vilmos Bilicki. Stunner: A smart phone trace for developing decentralized edge systems. In José Pereira and Laura Ricci, editors, *Proceedings of the 19th International Conference on Distributed Applications and Interoperable Systems (DAIS 2019)*, pages 108–115, Cham, 2019. Springer International Publishing.

[104] Valentin Thirion, Korian Edeline, and Benoit Donnet. Tracking middleboxes in the mobile world with traceboxandroid. In *Proceedings of the 7th International Workshop on Traffic Monitoring and Analysis (TMA 2015)*, Barcelona, Spain, April 2015.

[105] Norbert Tölgyesi and Márk Jelasity. Adaptive peer sampling with newscast. In *Euro-Par 2009*, volume 5704 of *LNCS*, pages 523–534. Springer, 2009.

[106] Chun-Wei Tsai, Chin-Feng Lai, Ming-Chao Chiang, and L.T. Yang. Data mining for internet of things: A survey. *Communications Surveys Tutorials, IEEE*, 16(1):77–97, 2014.

[107] Daniel Wagner, Andrew Rice, and Alastair Beresford. Device analyzer: Understanding smartphone usage. In *Proceedings of the 10th International Conference on Mobile and*

Ubiquitous Systems: Computing, Networking, and Services, pages 195–208, Tokyo, Japan, December 2013. Springer.

[108] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. Device analyzer: Large-scale mobile data collection. In *Workshop on Big Data Analytics*, 2013.

[109] Ji Wang, Bokai Cao, Philip S. Yu, Lichao Sun, Weidong Bao, and Xiaomin Zhu. Deep learning towards mobile applications. In *Proceedings of the IEEE 38th International Conference on Distributed Computing Systems (ICDCS 2018)*, pages 1385–1393, Vienna, Austria, July 2018. IEEE.

[110] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):374—-385, 2011.

[111] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, Chicago, IL , USA, November 1982.

[112] Martin A. Zinkevich, Alex Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23 (NIPS 2010)*, pages 2595–2603, Vancouver, Canada, December 2010. MIT Press.

[113] Raffaele Zullo, Antonio Pescapè, Korian Edeline, and Benoit Donnet. Hic sunt nats: Uncovering address translation with a smart traceroute. In *Proceedings of the 2017 Network Traffic Measurement and Analysis Conference (TMA 2017)*, Dublin, Ireland, June 2017. IEEE.