Harnessing the power of implicit and explicit social networks through decentralization

Arnaud Jégou

▶ **To cite this version:**

Arnaud Jégou. Harnessing the power of implicit and explicit social networks through decentralization. Web. Université de Rennes, 2014. English. NNT : 2014REN1S069 . tel-01135867

HAL Id: tel-01135867
https://theses.hal.science/tel-01135867

Submitted on 26 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2015

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Arnaud JÉGOU

préparée à l'unité de recherche INRIA-Bretagne Atlantique
Institut National de Recherche en Informatique et Automatique
Université de Rennes 1

Harnessing the power of implicit and explicit social networks through decentralization.

Thèse soutenueà Rennes le 23 Septembre 2014

devant le jury composé de :

David GROSS-AMBLARD
Professeur, Université de Rennes 1 / *Président*
Arnaud LEGOUT
Chargé de recherche, Inria / *Rapporteur*
Laurent VIENNOT
Directeur de recherche, Inria / *Rapporteur*
Etienne RIVIERE
Maître assistant, Université de Neuchatel / *Examinateur*
Pascal MOLLI
Professeur, Université de Nantes / *Examinateur*
Anne-Marie KERMARREC
Directrice de recherche, Inria / *Directrice de thèse*
Davide FREY
Chargé de recherche, Inria / *Co-directeur de thèse*

Acknowledgments

First I would like to thank my advisors Anne-Marie Kermarrec and Davide Frey for their help and continuous support during these four years. Anne-Marie, thanks for always being supportive and encouraging when I was in doubt. Davide, thanks for always finding time when I needed help, I will never forget the numerous nights spent on papers submissions. I am truly grateful that you gave me the opportunity of working with you during all these years, and it has been a real pleasure.

I would also like to thank Arnaud Legout and Laurent Viennot for having accepted to review this thesis, David Gross-Amblard for doing me the honor of presiding my defense, and Pascal Molli and Etienne Rivière for accepting to participate in my jury.

Thanks also to all the members of the ASAP team for all this time together, the long discussions during coffee breaks, the drinks, and everything else. Thanks in particular to Nabil with whom I shared my office for almost four years, and Nupur who accepted to read and comment on my manuscript.

Finally, thanks to my familly and friends who supported me and cheered my up when needed during all these years. Thanks in particular to Lena, who made of my PhD a wonderful experience.

Thanks to all.

Contents

Table of contents 0

1 Introduction 3
- 1 Contributions of this thesis 5
- 2 Publications 7

2 State of the Art & Background 9
- 1 Recommender systems 9
- 2 Trust and accountability 12
- 3 Sybil attacks 16
- 4 Privacy 19
- 5 System Model 22

3 Trust-aware peer sampling: Leveraging explicit and implicit social networks. 23
- 1 Introduction 23
- 2 Objective 25
- 3 Trust Model 26
- 4 Trust-aware peer sampling 28
- 5 Trust-aware clustering 34
- 6 Concluding remarks 35

4 Privacy preserving trust-aware peer sampling 37
- 1 Introduction 37
- 2 Privacy-preserving Trust-Aware Peer Sampling 38
- 3 Evaluation 48
- 4 Related Work 53
- 5 Concluding remarks 53

5 Anonymity for gossip-based applications 55
- 1 Introduction 55
- 2 System models 56
- 3 FreeRec 57
- 4 Experimental setup 63
- 5 Results 64

6 Related Work . . . 70
7 Perspectives . . . 70
8 Conclusions . . . 71

6 Distributed privacy preserving clustering 73
1 Introduction . . . 73
2 Contribution . . . 73
3 DPPC Protocol . . . 74
4 Active adversary . . . 82
5 Experimental Setup . . . 83
6 Experimental results . . . 86
7 Related work . . . 93
8 Perspectives . . . 94
9 Concluding remarks . . . 94

7 Conclusion 95

A DynaSoRe 99
1 Introduction . . . 99
2 Problem Statement . . . 101
3 System Design . . . 103
4 Evaluation . . . 108
5 Related Work . . . 114
6 Conclusion . . . 115

B Résumé en Français 117

Bibliographie 127

Chapter 1

Introduction

The amount of information available on the Internet is growing every day. Online shops give access to millions of items to buy. Online streaming media propose thousands of movies and TV series. Online newspapers generate thousands of new news articles every day. This overload of information available to the user makes the navigation and the selection of interesting content a very difficult and time-consuming task. A typical way to filter the content is to rely on its popularity, e.g., based on the number of visits a page receives, the number of likes an article gets, the number of times an item is bought. However, using popularity only to filter the content is not sufficient. Indeed, different users have different tastes and different centers of interests, and popularity alone is not capable of capturing it.

Recommender systems aim at solving this problem by providing users with recommendations, personalized based on their interests. Typically, a recommender system will, given the list of ratings given by users to items, predict the rating a particular user would give to all the items she has not rated yet. Then, the system uses these predictions to select which items should be recommended to this user. This prediction uses the ratings given by the other users, and also takes into account the target user's personal ratings in order to provide them with personalized recommendations.

This however comes at a cost. Computing recommendations requires both a large storage capacity and an important computing power. Given this cost, companies providing personalization services need to somehow benefit from it. This can be achieved in various ways, for example by making the users pay for the service. Content producers may also pay in order to have their content included in the recommendations, or to be better ranked. This biases the recommendations towards content producers that are willing to pay the largest amount of money, which is not always in the interest of the users. A company can also add advertisements to the service, which usually implies collecting personal information of the user and give access to it to an ad company.

Whether they generate personalized advertisements or not, companies usually collect as much data on users as they can because some companies are willing to pay for this type of data. There exist several examples of companies, in a difficult situation, that

sold their user data to other companies.[1] This represents a serious threat to user privacy, since the user data collected may contain personal and potentially sensitive information such as name, address, e-mail address and telephone number, but also habits of connections to the system, type of content of interest, list of pages visited, etc. This data can not only be sold to other companies, but can also be released to public by someone hacking the company's website. The most famous example of such release of sensitive client personal information is the AOL search data leak. In August 2006, the AOL company released the search history of 650,000 users over 3 months for research purposes. Despite the fact that the data was anonymized before the release by assigning a random ID to each user, the keywords contained in the search queries contained enough sensitive information for some of the users to be identified.

Additionally, companies can also refuse to deal with content that is considered as negative by people or groups that are influential enough. For example, it is not uncommon for newspapers to leave news items unreported or censor the articles of their journalists if the information contained therein is negative for a shareholder of the company. Companies can also refuse to index content declared illegal by a state, be it rightfully (not indexing child pornography) or not (censoring political information). Services that do not comply with these restrictions have a hard time remaining available. For example the pirate bay, a website that provides links for downloading content through the BitTorrent protocol, had to change country several times in order to avoid being shut down. Another example is Lavabit, an email service that uses encryption to protect the users' privacy. In July 2013, Edward Snowden used Lavabit in order to send invitations to a press conference after his revelations about the mass surveillance from the USA. Shortly after that, the Lavabit service was suspended. Although the creator of Lavabit was not allowed to give any details, it became clear that he received pressures to give away the personal information, along with the content of the email of his users, and the only alternative was to shut down the system[2]. This clearly shows that such centralized services are sensitive to governments or influential groups, and for this reason, users using a recommender system in order to navigate the web might have access only to a limited, censored, fraction of the information available out there.

Peer-to-Peer (P2P) is a way to bypass the limitations of such centralized services. Indeed, in P2P systems where the load of storage and computation is handled by the users, there is no need for a central service provider supporting both the cost and the responsibility of the service. A P2P infrastructure means that: 1) there is no need to finance the service provider, thus incentive to favor specific content no longer exist nor the temptation to exploit the users personal information; 2) there is no central entity that can be influenced to get some content censored. P2P infrastructure therefore represents a much more robust alternative, since in order to prevent some content from being shared and spread over a P2P network, authorities have to track all the users sharing the content one by one. This is costly, time-consuming, and has never been efficient enough to prevent users from using P2P systems.

[1] A recent example is the Virgin Megastore chain that closed all its shops in France, and sold its clients file to a former competitor, La Fnac.

[2] http://lavabit.com/

In recent years, the use of P2P file sharing started to slightly decrease in France, mainly because the authorities started to target P2P networks specifically when fighting against illegal download. While their results are mitigated (less than a dozen users were condemned in 4 years), their attempt of tracking and pursuing users of P2P systems was possible for a reason: P2P systems, while avoiding the concentration of all the users' data on a central service provider, are not naturally good at hiding the users' data from other users. For example, in [LBLL+10] the authors were able collect 150 millions of IP addresses of users downloading content, as well as the IP address of users adding 70% of the content to BitTorrent. All this was performed over a short period of 103 days with a single computer.

However, P2P systems can also suffer from another issue which is the presence of misbehaving users. In a P2P system, the load of the system (storage, computation, bandwidth) is spread over the users, some of them however might not do their share of the job correctly, either because they do not want to give their resources away (freeriders) or because they aim at manipulating or corrupting the system (malicious users). While the freeriders simply degrade the performances of the system by increasing the load on the honest users, malicious users can significantly degrade the quality of the service provided. For example in a P2P recommender system, malicious users may attempt to bias the recommendations received by the other users towards their own content. This problem is made worse by the fact that in P2P systems, creating a fake identity is relatively easy [R+01]. Indeed, since it is very difficult to assert the users' identity in P2P systems, thus a user can easily create multiple fake nodes in order to increase her influence on the system (this is known as a Sybil attacks [Dou02]).

Yet, we believe that despite these potential issues, P2P systems remain an extremely attractive alternative to centralized services. In P2P systems, users are willing to pay for the service because they benefit from it and the impact of running part of the service remains low. But as previously explained, P2P systems suffer from several weaknesses that need to be addressed for P2P systems to be a realistic alternative to centralized ones. First, the privacy issue must be addressed so that users can participate in and benefit from the system without having to expose their personal information to the other users. Second, the trust and accountability between users must be improved so that applications can handle the negative impact of malicious users and freeriders without degrading the quality of service for other users. In this thesis, we address precisely these issues and tackle the associated challenges.

1 Contributions of this thesis

In this document, we present 4 contributions aiming at solving some of the privacy and trust issues of P2P systems described earlier in the context of online recommendations. This work took place in the more general context of the Gossple project that we detail further in the next chapter.

TAPS First, we propose Trust-aware peer sampling (TAPS), a peer-sampling protocol that leverages the presence of an underlying social network to provide users with an estimation of the trustworthiness of other users, as well as accountability. Trust between users is estimated by extracting information from the social network, and finding paths between pairs of users. In turn those paths are used to estimate a trust value between involved users. These paths are also used in order to achieve accountability in the sense that users have a way to contact and identify each other through the social network. TAPS is presented in Chapter 3.

PTAPS Second, we propose PTAPS, a privacy preserving version of the TAPS protocol, that provides the same functionalities while preserving users privacy. In the TAPS protocol, users still have to expose their list of acquaintances as well as the trust they have in them. In PTAPS instead, users can participate in the protocol and fully benefit from it without having to expose this information to unknown users. This is achieved by using encrypted paths, so that users can build paths to each-other without having to know the identity of the intermediary nodes in the path. In PTAPS, users can carefully select which information to share with a given user depending on whether she is a friend or not. PTAPS is presented in Chapter 4.

FreeRec Third, we propose FreeRec, an anonymous decentralized peer-to-peer architecture. FreeRec provides user-based recommendations, and uses onion routing techniques to anonymize communications involving the users personal data (i.e. their profile). In FreeRec, every user has a *proxy* in charge of receiving message and relaying them to the user. We also provide a detailed analysis of the probability that a user's proxy chain will become compromised by colluding adversaries, depending on whether the size of the chain is fixed or if it is a random value in a range $[m : M]$. We show that in our context, chains of fixed length provide significantly better privacy. FreeRec is presented in Chapter 5.

DPPC Last, we propose Distributed privacy-preserving clustering, or DPPC, a protocol aiming at hiding sensitive information of users. This is achieved by mixing users personal information with the ones of users interested in similar items in order to create fake (obfuscated) profiles for the users. These profiles can then be used to perform similarity computations to identify similar users in the network. This is the first step of many P2P personalization protocols such as [BFG+10]. DPPC is presented in Chapter 6.

During this PhD thesis, we also designed DynaSoRe, an in-memory storage system designed to efficiently store social network data into a data center. DynaSoReanalyzes request traffic to optimize data placement and substantially reduces network utilization. DynaSoReleverages free memory capacity to replicate frequently accessed data close to the servers reading them according to network distance. For the consistency of the document, we left this last contribution out of this thesis and appended the paper resulting from this work in the Appendix A.

2 Publications

As main author:

[1] Davide Frey, Arnaud Jégou, and Anne-Marie Kermarrec. Social market: combining explicit and implicit social networks. In Stabilization, Safety, and Security of Distributed Systems, pages 193–207. Springer, 2011

[2] Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, Michel Raynal, and Julien Stainer. Trust-aware peer sampling: Performance and privacy tradeoffs. Theoretical Computer Science, 512:67–83, 2013

[3] Xiao Bai, Arnaud Jégou, Flavio Junqueira, and Vincent Leroy. Dynasore: Efficient in-memory store for social applications. In Middleware 2013, pages 425–444. Springer, 2013

As one of the main authors:

[4] Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. Whatsup: A decentralized instant news recommender. In Parallel and Distributed Processing, pages 741–752, 2013

[5] Antoine Boutet, Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, and Heverson B Ribeiro. Freerec: an anonymous and distributed personalization architecture. In Networked Systems, pages 58–73. Springer, 2013

[6] Antoine Boutet, Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, and HeversonB. Ribeiro. Freerec: an anonymous and distributed personalization architecture. Computing, pages 1–20, 2013

Other publication:

[7] Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. Privacy-preserving distributed collaborative filtering. In Networked Systems. Springer, 2014

Chapter 2

State of the Art & Background

In this chapter, we provide some background in the area of recommendation systems, trust and accountability in distributed systems as well as privacy mechanisms.

1 Recommender systems

Recommender systems exploit users' data to provide them with recommendations, such as books, music, web content, etc. In order to provide accurate recommendations (i.e. recommendations that will actually be interesting for the user), recommender systems need to compute recommendations that are personalized for the users. Indeed, different users typically have very different centers of interests, thus providing every user with the same recommendation would be a good solution for only a fraction of the users, at best. Several techniques exist to compute personalized recommendations. One common feature is that they all receive some input from the user, such as the list of web pages they visit, the items they buy or the movies they watch. Based on that, there are two main techniques for recommender systems: content-based and collaborative filtering.

1.1 Content-based

Content-based techniques exploit the content of the items in order to identify similar items. Consider for example that items are video games. Then system may rely on information related to the type of game, the date of their release, the targeted platforms for the game or the producer to compute similarities between items. This information can be used along with user profiles for the system to recommend to users the game they are likely to be interested in. Typically, such recommender systems[LDGS11, DGLSB08, Jan08, MR00, PB07] perform well when a large amount of information is available for each item. However, collecting information about every item remains costly time-wise and requires a large storage capacity. This makes it difficult to apply in a distributed environment where each node only has a partial knowledge of the set of items.

1.2 Collaborative filtering

As opposed to content-based recommenders, the collaborative filtering technique is oblivious to the type and the content of items it recommends. The intuition is that items recommended to a user are those that similar users liked. Two main categories of collaborative filtering techniques exist: user-based [HKBR99] and item-based.

The idea behind user-based approaches is that if two users expressed similar interests in the past, they are likely to continue to be interested in the same items in the future. The goal of a user-based collaborative filtering system is to identify similar users and leverage their profiles to predict the ratings on items that a user who has not yet seen. Many similarity metrics have been presented in the literature, such as the Cosine similarity [SM86], the Jaccard similarity, the Pearson correlation. For the sake of simplicity, in this thesis we use the Cosine similarity, which is defined as follows [1]:

$$cos(u,v) = \frac{\sum_{i \in I} r_{u,i} \cdot r_{v,i}}{\sqrt{\sum_{i \in I} r_{u,i}^2} \cdot \sqrt{\sum_{i \in I} r_{v,i}^2}},$$

where I is the set of all items, and r_{ui} is the rating of user u for item i.

Once the system has selected the neighbourhood of a user, i.e. her similar users, it predicts the ratings of this user for items by averaging the ratings of the user's neighbors:

$$p_{ui} = \frac{\sum_{v \in N(u,i)} s_{uv} \cdot r_{vi}}{\sum_{v \in N(u,i)} s_{uv}},$$

where p_{ui} is the predicted rating of user u for item i, $N(u,i)$ the list of users similar to u who rated item i and s_{uv} the similarity between users u and v. Weighting the contribution of each neighbor by its similarity ensures that a highly similar neighbor will have a larger impact on the rating prediction. Similarly, a dissimilar user, even if selected, for instance if the dataset is sparse, will not impact the prediction significantly.

Item-based approaches are centered on the items the way user-based approaches are centered on users following the intuition that if a user liked a given item, she is likely to like similar items. Therefore, instead of computing similarities between users, the system computes similarity between items to identify close items. Here, each item is associated with a profile, containing the list of users who rated this item along with the ratings. Similarity computations between items are computed the same way as between users in a user-based approach and subsequently used to predict the users ratings.

Item-based techniques are usually preferred in centralized systems for scalability reasons as the number of items is typically much smaller than the number of users. However, P2P systems are scalable by design. In addition, as in content-based techniques, an item-based recommender require the knowledge of all the items. This makes such approach less natural to distribute. User-based techniques, instead, only require each user to know a set of similar users from which she can compute her recommendations. This clearly is a much better match to a distributed system and particularly a

[1] Note that the approaches presented in this thesis could use any other metric instead

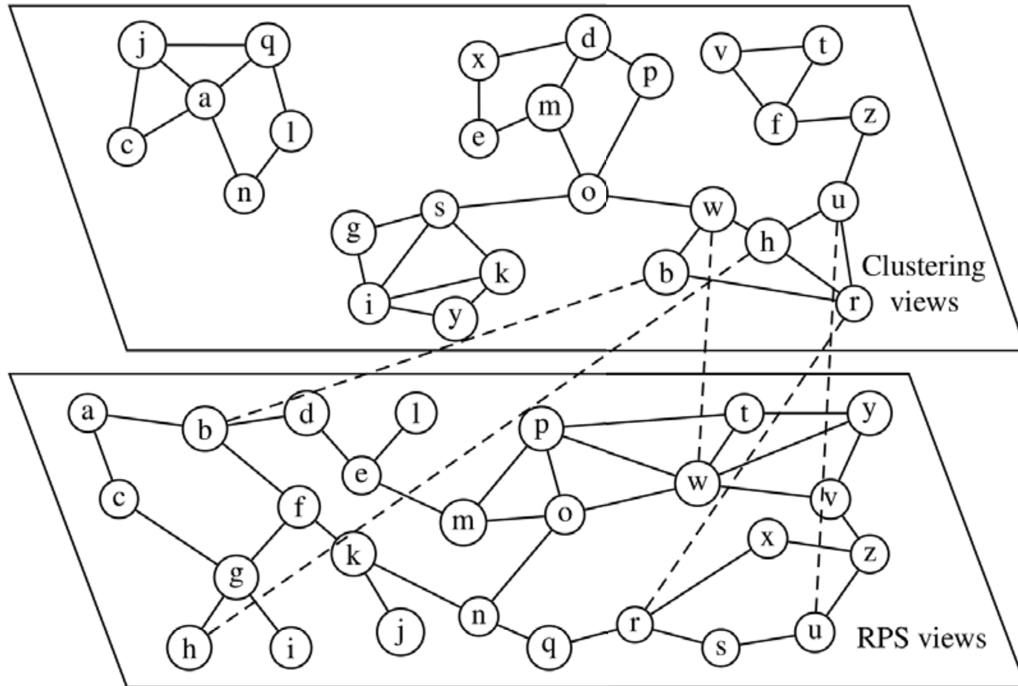


Figure 2.1: RPS and clustering view in the Gossple social network.

fully decentralized one. Typically, in a P2P environment each user is associated with a node in the system. Therefore user-based techniques are good candidates to build a distributed algorithm on a P2P infrastructure.

For these reasons, we favor user-based collaborative technique in a P2P setting and focus on such approaches in the rest of this thesis.

1.3 The context of this thesis: Gossple, a decentralized personalization system

Our work takes place in the context of the Gossple project. We now provide some background on the core network of Gossple on which we designed our contributions.

Gossple [BFG+10, Ker09] is a fully decentralized system providing personalized content to its users. Gossple does so by building for each user a community of acquaintances having similar interests to those of the user. This community is used by the system to provide the user with personalized services, such as query expansion [BGLK09], news dissemination [BFG+13] or content recommendation.

Gossple is built on top of two sub-protocols, a random peer sampling (RPS) protocol and a clustering protocol. For each of these protocol, each node maintains a data structure called a view, which contains a set of references to other nodes, respectively called the RANDOM view and the CLUSTER view, as shown in Figure 2.1. Each entry in a

view contains the IP address of a node, its profile, and a timestamp indicating when this entry was generated. In the RPS protocol, each node periodically selects a node from its RANDOM view, and both exchange a random sample of their respective RANDOM views. This provides each node with a continuously changing subset of the network. Several implementations of random peer sampling exist [JVG+07, VGVS05, BGK+09] and in this thesis we assume the use of the gossip-based peer sampling [JVG+07]. In the clustering protocol [VVS05], each node periodically selects the node from its CLUSTER view with the oldest timestamp, and both exchange a sample of their CLUSTER views. Here the samples are not selected randomly, but instead each node selects the nodes from its CLUSTER view that are the most similar to the other node. Once this exchange is completed, the node selects its k (where k is the maximal size of the CLUSTER view) most similar neighbors from the union of its RANDOM and CLUSTER views and insert them into its CLUSTER view in place of the previous ones. Over time, the CLUSTER view of each node converges towards the optimal view containing the k most similar nodes to the user, which we refer as its k-nearest neighbors, or KNN.

2 Trust and accountability

2.1 Trust

Collaborative applications require some level of trust management or accountability. Since the cooperation of users is required for the system to work, users expecting to benefit from the system without collaborating to the system, might harm the system either by degrading its performances, or by increasing the load on other users. For example, in a P2P file sharing protocol such as BitTorrent, some users can decide to download a file without sharing it with others (although the BitTorrent protocol embeds a mechanism to reduce the download capacity of such users, authors in [LMSW06] showed how this mechanism can be bypassed). In this case, honest users alone will have to upload the files for both the honest users and the dishonest ones. In a distributed recommendation system, users may attempt to get recommendations from the system without contributing their personal data, or by contributing fake data. In this case, the quality of service of the system will be degraded, as honest users will receive recommendations that are partially random. Such users who benefit from a collaborative system without contributing are called free-riders.

Even worse is the case of malicious users who, instead of simply not contributing to the system, aim at harming the system itself or its users. For instance, in a distributed file sharing system, a malicious user may share corrupted data to infect with a malware, users downloading the content. In a distributed recommendation system, malicious users may bias the recommendations by artificially increasing the popularity of some content, or burying some content they want to censor.

A way to limit the impact of free-riders or malicious users is to provide users with a mechanism to evaluate the amount of trust each user should put in others. By dealing only with trusted peers, users can reduce the impact of misbehaving users both on them and on the system.

2.2 Accountability

However, relying on trust only might not be sufficient in some settings. For instance, in a distributed marketplace, where users get in connection with other to buy or sell goods, trust might not be enough as it is impossible to tell for sure that a user, based on her history only, will continue to behave correctly in the future. In the context of a free-rider in a file-sharing system, this is typically not such a big concern as if a user appearing honest suddenly stop sharing her files, honest users will only keep sending her data for some time until they realize that she is not collaborating any more. In this case, only some pieces of data that have been shared are lost since nothing was shared in return.

However, in other settings such as an online marketplace, this turns out to be more problematic for money and goods are involved. A user might buy an object and never pay for it, or sell an object and never receive the money. The system might be significantly impacted before the user can be detected and excluded from the system. Accountability can solve this issue by providing users involved in a transaction with means to identify each other and make him/her responsible of his/her actions.

2.3 Reputation-based Trust

An alternative to trust and accountability is to rely on some reputation metric that can provide a probability with which users will behave in the future. Reputation systems are a class of systems in which the behavior of users is monitored over time in order to build a reputation for each user. They allow users to get information about how trustworthy other users are, providing them with a way to avoid dishonest or malicious users. These systems are widely used in e-commerce websites like eBay or Amazon.com where, after each transaction, the buyer and the seller are asked to give feedbacks about the transaction. These feedbacks can take various forms, but they typically consist of a rating and a textual explanation. On eBay for example, there are three possible ratings: positive, neutral or negative. On Amazon.com [LSY03] the rating is a number of stars, from 1 to 5, allowing the user to describe how well, or how bad, the transaction went. These ratings are used to build the reputation of a user (be it a buyer or a seller) allowing other users to assess the relevance of engaging in transactions with this user.

Reputation systems are also used in P2P networks. Since the real identity of users is difficult to assert, there is little incentive for users not to cheat and benefit from the system without participating. Thus, it is important for the system to work to give a motivation for the users to participate. For example, the BitTorrent protocol uses a tit-for-tat mechanism so that if user u_1 sends data to user u_2, she will have the priority when u_2 will select which node to send data to.

In the same context of P2P file sharing, the EigenTrust [KSGM03a] algorithm is a well known P2P reputation system. EigenTrust aggregates the ratings received by each node from other users in order to create a global reputation score for each user. After each interaction between two peers i and j, both decide whether the transaction was satisfactory or not. The local trust score node i has for node j is the number of

satisfactory transactions with j minus the number of unsatisfactory ones. The global reputation of node i is the sum of the local trust scores given by each node who interacted with i in the past, weighted by their own global reputation.

While EigenTrust uses mechanisms in order to protect against malicious users attempting to raise their reputation, it is not resilient to Sybil attacks due to the way the reputation of nodes is computed. A Sybil attack is an attack in which a user creates multiple nodes she controls, and uses these Sybil nodes in order to either increase her impact or even bias the system. More details about Sybil attacks are given in Section 3. In [CF05] the authors classify reputation functions in two categories: symmetric and asymmetric. Considering a graph $G = (V, E)$ where each vertex is the node in the P2P system, and the edges represent the trust rating between the nodes, and a reputation function f such that $f(G)_i$ is the reputation of node i in the system: A reputation function f is symmetric if given a graph isomorphism σ, for all graph $G = (V, E)$ with image G' under σ, and all nodes $i \in V$, $f(G)_i = f(G')_{\sigma(i)}$. The intuition is that if a malicious node creates one Sybil node for each honest node of the system, and connects the Sybil nodes to each other in order to mirror of the network of honest users, then each Sybil node will have the same reputation as the honest node it is mirroring. Thus, the malicious user can create a Sybil node that has the same reputation as the honest node with the highest reputation in the system.

In [GJA03] the authors present another reputation system in the context of file sharing. When a node requests data from another one, it sends a signed message containing its identity and the description of the requested data. This signed request is used by the sender peer to form a receipt by adding its own identity to the message, and signing it with its own key. Receipts are then sent to a trusted central entity that collects all the receipts from the nodes and computes a reputation score for each node. The trusted entity then signs reputation scores that remain valid for some period of time and send them back to the peers. While it is able to ensure that honest users will have credit for their participation in the network, even when they send data to malicious users, it cannot prevent colluding users from declaring fake transactions between each other in order to increase their credits.

Cheap pseudonyms and whitewashing Cheap pseudonyms is described in [R$^+$01] as the fact that on the Internet it is in general very easy, and inexpensive, to create multiple identities. This has two main consequences: 1) it allows users to create multiple fake identities in order to mount Sybil attacks, as explained in Section 3; and 2) it allows users to use whitewashing in reputation systems. Whitewashing is the action for a node to leave the system and enter it again with a new identity in order to get rid of its negative history. This is clearly an issue in reputation systems as negative reputation can be easily cleared.

In [R$^+$01] the authors study the impact of cheap pseudonyms on reputation systems and propose solutions to either discourage whitewashing or simply prevent it. First, they propose to impose fees for new users of the system to reduce the benefits that misbehaving users might get from creating multiple identities. The idea behind this is that if the entrance fees are high enough, they will exceed the potential gain malicious

users can get from whitewashing. However, they conclude that it is impossible to efficiently discourage misbehaving users from whitewashing without at the same time repelling a large number of them who cannot or do not wish to pay the entry fees. Second, they propose an alternative solution, the once-in-a-lifetime pseudonyms, which solves the problem of whitewashing by relying on a central authority that signs one and only one pseudonym for each user. This prevents whitewashing as a user will not be able to get a second signed pseudonym from the central authority. Users are still able to use unsigned pseudonyms instead, but this will be detected by the other users and considered suspicious. However, their system suffers from two major drawbacks. First, they rely on a trusted central authority that is able to authenticate once-in-a-lifetime pseudonyms and prevent a user from creating multiple ones. This central entity is a single point of failure that can be the target of attacks such as denial of service attacks. Moreover, it is difficult to agree on a trusted entity in a distributed environment. Second, they assume the existence of an infrastructure providing public/private key pairs for each user, and ensuring that a given user is assigned only one pair of keys.

In [Zac00], the author solves the problem of whitewashing by setting the default value for the reputation of a newcomer as the smallest reputation possible. By doing this, malicious users cannot get any benefit from whitewashing, as it can only decrease their reputation. While this does not prevent them from behaving badly, for example, by scamming other users in an online marketplace, this forces them to behave honestly for some time after each malicious action in order to build back their reputation. However, the drawback of this mechanism is that honest users joining the system will be treated as malicious users. Thus, newcomers have a hard time building their reputation as most users refuse to deal with them, which can easily discourage them.

2.4 Social-network-based Trust

An alternative way to build trust between users is to rely on online social networks and more specifically on the fact that connections in a social network reflect some social links and therefore are likely to be trusted.

As such, social networks can be leveraged to assess the trustworthiness of users the way reputation systems do. More specifically, in online social networks, users declare a list of acquaintances, and typically provide additional information about the type of acquaintance (friend, family, professional, etc.). Additionally, users connected in an online social network often know each other in real life, and in any case they are able to judge how they know each other and how they trust each other. This data can be exploited to build trust relationship between users. For example, assume a user wanting to buy a specific item online put on sale by an unknown user. Clearly, if that seller turns out to be the brother of one of the buyer's friends, it is likely that the buyer will feel safe to buy the item from that seller, especially upon recommendation of his/her friend. This is due to the 1) the fact that the user has some information about the seller her social links and 2) the fact that indirectly the seller and the buyer are connected through the social network. Therefore the buyer expects her friend to contact her brother if a problem occurs and assume that the seller will not dare misbehave.

Social networks provide several benefits compared to reputation systems. First, newcomers do not encounter the problem of having to build up there reputation when joining the system. Indeed, as the trust is based on pre-existing relationships between users, when a user enters the system it simply needs to connect to her acquaintances to build trust relationships with other users. The drawback here is that a user must know some users already in the system. However, given the wide spread of online social networks, it is likely that any newcomer can rely on at least one friend to enter the system. Second, social networks naturally provide accountability as, if a user A knows a user B through a sequence of acquaintances in the social network, there exists a path of acquaintances in real life, likely to be trusted from A to B. Several systems using this approach exist, such as NABT [LHL+10] and SUNNY [KG07].

3 Sybil attacks

Sybil attacks, first introduced in [Dou02], are a class of attacks in which a user creates a large number of fake identities in order to increase its influence in a system. For example in a reputation system, an attacker can create thousands of Sybil nodes and use them to increase her reputation by giving good ratings to herself. Sybil attacks are possible because in a distributed system it is very easy for a user to create multiple fake identities, as explained in Section 2.3.

3.1 Sybil prevention

Some of the techniques used against whitewashing, such as the entrance fee, can also be used to prevent an attacker from creating thousands of nodes. But in the case of Sybil attacks, specific methods can be used. Indeed, in order to prepare a Sybil attack a single user needs to create thousands of fake identities, and several methods rely on that to defend against Sybil attacks. For example in [AS04], the user is required to solve a problem that is easy for a human but complex for a computer such as a CAPTCHA [VABHL03]. Another possibility is to require the user's machine to solve a small cryptographic puzzle such as in [JB99]. In both cases, an honest user (or her computer) only needs to spend a few seconds to solve the problem, but becomes prohibitively time consuming for a user creating thousands of nodes.

However, these techniques are not efficient if the expected gain is large enough. Solving thousands of cryptographic puzzles takes a lot of time if the attacker does it on her own machine, but can easily be distributed on several machines. The attacker can spend a fraction of the expected gain to buy computing power, for example in a cloud computing platform. CAPTCHA solving, that requires human time, can also be distributed over a large number of humans by using a service such as Amazon Mechanical Turk [2]. For example in an online marketplace, an attacker who manages to subvert the reputation system in order to increase her reputation, and uses her high reputation to

[2] Amazon Mechanical Turk is a service where clients can submit small tasks that will be solved by humans against remuneration

trick honest users can expect to gain a lot of money. Thus in this context these methods are not efficient.

3.2 Sybil detection

Given the difficulty to prevent the creation of a large number of Sybil nodes, especially in a distributed environment, an alternative is to detect the Sybil nodes once they are created and mitigate their impact on the system.

In [FM02], the authors make the assumption that while the attacker can control a large number of nodes, these nodes will have the same IP prefix. When their protocol needs to select a set of nodes, nodes are selected with different IP prefixes in order to reduce the probability to select two successive malicious users. While this assumption holds when the Sybil nodes are in fact virtual nodes hosted on a single physical node, it does not hold in general, in particular when the attacker is using a zombie network to run its nodes.

A popular solution to defend users against Sybil attacks is to rely on an online social networks to detect them. In an online social network, such as Facebook or RenRen, the friendship is subject to the acceptance of requested users. Typically, users reject friend requests coming from unknown users, which means that although an attacker can create a very large number of Sybil nodes, and send friends requests to a lot of honest users, only a few will get accepted. If we see a social network as a graph, where users are nodes and edges represent friend relationship, then there are two regions in the graph: the honest region which contains honest nodes and the Sybil region which contains the Sybil nodes. The two regions are only connected through few edges Sybil nodes managed to establish with honest users, that are called attack edges.

Different protocols use this characteristic of the social network in order to defend against Sybil attacks, such as SybilGuard [YKGF08], SybilLimit [YGKX08], SybilInfer [DM09] and GateKeeper [TLSC11].

In SybilGuard and SybilLimit, nodes execute deterministic walks of fixed size on the social network. A node considers another node as honest only if their walks intersect. Because the number of attack edges between the honest region and the Sybil region is small, the probability that the walk of an honest node enters the Sybil region is very small. In the honest region, Sybil nodes cannot manipulate the random walks, and thus cannot increase the probability to cross the honest user's route. The walks are executed in a way, such that, it ensures that given the length of the walks w, the honest node will accept at most w Sybil nodes per attack edge.

3.3 Sybilproof algorithms

Alternatively to prevent the creation of Sybil nodes, or detecting them, algorithms can be designed to be Sybilproof. An algorithm is Sybilproof if it is designed in such a way that an attacker cannot increase its influence on the system by creating Sybil nodes. There exists a wide range of Sybilproof algorithms, from vote aggregation systems (SumUp [TMLS09]), distributed hash tables (Whanau [LLK10]) to reputation

systems.

Reputation systems In [CF05], the authors propose a formalization of Sybilproofness in the context of reputation systems. Considering the network as a graph where users are nodes, and edges represent the relationship between users, they describe a reputation mechanism as a set of two functions: g, that takes as input a path in the graph, and outputs the reputation over this path based on the values on the edges; and $\oplus$, that aggregates the reputation over several disjoint paths. While this does not cover every possible reputation management protocol, many of the existing work in this field fits in this model [KSGM03b, ZH07, KG07].

The authors classify Sybilproof algorithms into two categories, value Sybilproof and ranking Sybilproof, and give conditions on g and $\oplus$ for the algorithm to be in either category. An algorithm is value Sybilproof if it is impossible for a node to increase its reputation by creating Sybil nodes. An algorithm is ranking Sybilproof if, in addition to being value Sybilproof, the node cannot improve its ranking by decreasing the reputation of nodes with higher reputation.

The conditions for value Sybilproofness are:

- Given two paths P and P' such as $P \subseteq P'$, then $g(P) >= g(P')$.
  - A malicious node cannot increase its reputation by increasing the size of a path with Sybil nodes.
- g is non-decreasing with respect to the edges values
  - Increasing the values on the edges of a path cannot decrease the score of the path
- $\oplus$ is non-decreasing
  - Discovering an additional path toward a given node cannot decrease the reputation of this node
- If a path P is split into two disjoint parallel path $P1$ and $P2$, then $g(P) >= g(P1) \oplus g(P2)$
  - A node cannot increase its reputation by splitting paths leading to him by creating Sybil nodes

In order to achieve ranking Sybilproofness, the additional condition is:

- $\oplus = max$
  - Ensures that a node cannot decrease the reputation score of a better ranked node by cutting some of its links

Note that these conditions are sufficient, but not necessary.

4 Privacy

One of the motivations to move from centralized services to P2P ones is privacy. Indeed, the provider of a centralized service can collect, a large amount of personal information about the users of the service. This ranges from the information provided when registering for the service, such as name, address, phone number, age, and all sort of preferences, to the information that can be gathered when the user is using the service, such as when the service is accessed, where it is accessed from, what is the use of the service, etc.

For example, the Facebook social network has access to all the information provided by the users: name, age, profession, education, list of friends, photos, videos, posts, etc. But Facebook also implements a rating mechanism for web pages from other website, allowing users to say which pages they found interesting by using a 'like' button. Many websites add this 'like' button to their pages as it is an efficient way to gain visibility. However, this means that every time a page containing this button is loaded, the Facebook servers receive a query, and thus Facebook knows which pages are visited by which user whether or not the user does click on the like button.

When using P2P services instead of centralized ones, users become responsible of their own data, and are in position to control who has access to it. However, for the service to work, it is usually necessary for users to share information with other users, and thus leak some personal data to other users. For example, for a P2P file sharing protocol to work, users downloading the same file exchange pieces of that file. This means that when a user is downloading a file, she can know the list of users that are also downloading it. This is one of the methods used to track users downloading content illegally in P2P.

Thus, P2P only is not enough to preserve the privacy of users, and additional mechanisms must be designed in order for users to be able to benefit from the system without leaking their personal data.

Anonymity A natural way to preserve the privacy of a user is to provide anonymity such that interactions of a user with others cannot be linked back to the user. Several systems can provide anonymity to the users, such as Tor [DMS04], AP3 [MOP+04] or Tarzan [FM02]. These systems not only allow the users to anonymously send messages to other users, but they also allow a user to be contacted by other users without them knowing the user's true identity. This is called pseudonymity.

A popular way to achieve anonymity online is to use what is called onion routing. In this context, when a user A wants to anonymously send a message to a user B, it starts by selecting a list of k other nodes, $i_1...i_k$, and use these nodes as intermediary to relay the message from A to B. A will send the message to i_1, who will forward it to i_2 and so on until i_k who will forward the message to B. The message transmitted is encrypted several times in a way that ensures that none of the relay nodes knows more than the previous node and the next node in the path. If $k \geq 2$, no relay node knows the identity of both A and B. This ensures that B doesn't know where the message come from, and no relay node can know which nodes are communicating, thus ensuring

$A's$ anonymity.

However, providing such feature is not always enough to provide anonymity to the users. In [BMA+11] the authors describe how the anonymity of Tor can be broken if a user uses an insecure application. The authors take the example of users using BitTorrent over Tor, and show that 70% of these users only use Tor to connect to the tracker, and then establish direct connections with the other peers. If an attacker controls a Tor exit node, it can manipulate the information returned by the BitTorrent tracker so that the user establishes a direct connection with a node controlled by the attacker. If the user does establish such a direct connection, the attacker knows real IP address of the user. Because Tor channels several TCP streams inside the same Tor circuit (the list of nodes relaying the messages for the user), the attacker can associate the IP address found via BitTorrent to all the TCP streams sharing the same circuit, even though they might originate from perfectly secure applications, such as a secure web browser.

Furthermore, even if the system is able to provide each user with a pseudonym, and prevent any attacker from mapping the pseudonym to the user identity, the information provided by the user to the system can be enough to disclose its identity. For example, in the case of a recommender system, personal data from the user is used in order to generate recommendations. These data contain information about what a user is interested in, e.g., what movies she watched, which items she bought. Some recommender systems also include geographic information about the places visited by the user. These data might be enough to clearly identify a user, hidden behind a pseudonym. The AOL search data leak [Fou09] mentioned in introduction where users were identified by keywords despite the trace anonymization clearly illustrates this.

The second Netflix prize [BL07] is yet another example. The Netflix prize is a competition organized by Netflix, an online movie renting platform, awarding 1,000,000$ for the first algorithm able to improve the precision of their recommendation algorithm by at least 10%. For the competitors to test their algorithms, Netflix released a dataset containing a sample of their full dataset. As with the AOL search data, this dataset was anonymized by removing any user ID from the dataset and replacing them with random IDs. The competition was canceled after researchers from the University of Texas managed to disclose some of the users true identity by crossing the Netflix dataset with the one of IMDb (Internet Movie Database) [NS08].

This shows that providing anonymity or pseudonymity to users is not enough in the context of a collaborative application as users are required to share data with each other. Such data may contain enough information to identify the users. Thus, it should be used in conjunction with additional mechanisms that ensure that data shared with other users does not leak information that can lead to the identification of the user.

Protecting the users data As described in the previous paragraph, hiding the identity of the users is not sufficient not to break anonymity. Thus, it is necessary to alter users data exposed to others in order to either prevent the disclosure of users identities or to remove the sensitive information from the exposed data.

P3 P3 [AN11] is a privacy-preserving middleware providing personalization services to users without exposing their private data to other users. This is done in four main steps:

- Users compute locally the groups of interests to which they belong.
  - A user can belong to multiple groups.
  - The list of groups is predefined.
  - This computation is based only on the users private data and some data publicly available.
- The node responsible for each group collects the profiles of all the users belonging to this group.
  - Nodes profiles are split into several slices.
  - Different slices are sent to different slices collectors.
  - Slices collectors send the slices they collected to a group-wise aggregator.
- Group-wise aggregators generate recommendations based on the slices they received.
  - It can also use the aggregated information in order to get recommendations from sources outside of the system.
- Users retrieve the recommendations from the group-wise aggregator of their different groups of interests.
  - The retrieval is made through an anonymizing mechanism.

While this approach is interesting, it relies on the existence of predefined groups of interests, and the recommendations received by the users are at the granularity of groups. This limits the flexibility of the system as when new topic of interest emerges among the users, the system needs to be manually tuned in order to adapt the list of groups. The second limitation is that the system relies on a limited number of middleware nodes to collect the profile slices from the users, aggregate them and build the recommendations. Moreover, a single node is responsible for generating the recommendations of a given group. Unless coupled with a strong mechanism ensuring that the group-wise aggregators behave honestly, the system is very vulnerable to malicious or corrupted middleware nodes.

Differential privacy [Dwo06, Dwo08, AGK12] attracted recently a lot of interest in many areas, yet there are no clear definition of differential privacy in the context of recommender systems and therefore we leave it out of this section.

5 System Model

In the rest of this thesis, we consider a system consisting of a set of users equipped with interconnected computing devices that enable them to exchange information in the form of messages. Each user is associated with a user profile that characterizes her interests, her past behavior, her geographical location, and whatever other information the user wishes to add. A profile is a vector of strings that can represent, for example, URLs, words, or phrases. We refer to each such strings as keywords. Each keyword in a profile is also associated with a counter, its weight, which captures how many times the keyword has been added to the profile. The weight basically measures how relevant a given keyword is in the user profile. Keywords can be added by the user, or automatically extracted from her browsing history, as well as from her interaction with the system. To simplify the notation, we refer to a user and her profile with the same symbol $u \in U$, where U is the universe of all user profiles. We also use the terms node and user interchangeably to refer to a user and the machine associated with her.

Chapter 3

Trust-aware peer sampling

Leveraging explicit and implicit social networks.

1 Introduction

The advent of Online Social Networks has shifted the core of Internet applications from devices to users. Explicit social networks like Facebook, or LinkedIn enable people to exploit real-world connections in an online setting. Collaborative tagging applications such as delicious, CiteULike, or flickr form dynamic implicit networks of users on the basis of their online activities, interest profiles, or search queries. Users can not only access and introduce new available content but they themselves become accessible through the online infrastructure.

Explicit vs implicit networks Existing online social networks can be grouped into two main categories: explicit and implicit. In explicit networks, users explicitly determine which other users they should be connected to. In Facebook or MySpace, they issue and accept friendship requests. In Twitter, they decide that they wish to follow the tweets of specific users. The topology of the resulting network reflects the choices of users and often consists of links that already exist between real people. Explicit networks are thus very useful to exploit existing connections but provide little support for discovering new content [BCK+07, AHK+07].

Implicit networks fill this gap by taking an approach which allows users to discover new content, and acquaintances [BFG+10]. They form dynamic communities by collecting information about collateral activities of users, such as browsing websites or tagging documents, URLs, or pictures. A given user may or may not be aware of the other members of her own communities. Other users should be clearly visible if the purpose of the application is to discover new people, while they may be hidden for the sake of privacy when they are simply being used as proxies to access new interesting content. In either case, the ability to establish new social connections is key to identifying new and useful data.

The problem addressed in this chapter: Trust Recent years have seen the emergence of a significant number of research efforts and applications exploring the power of the explicit and implicit paradigms. Nonetheless, a lot more can be achieved if both the approaches are combined into a single framework. Consider the following example. John, who lives in London, bought two electronic tickets for a classical-music concert in Paris, Berenice by Handel, but an unexpected event makes him and his friend unable to travel to Paris. The concert is tomorrow and John would like to sell the tickets to someone who can actually attend the event. Unfortunately, while John has many friends interested in classical music, they are all based in the United Kingdom. He does know a few people in Paris, but they are mostly people he met while traveling and who do not share his musical tastes. He tries calling a few of them but his best bet is Joseph, who claims to have a friend whose parents often go to classical-music concerts. Unfortunately, this friend of Joseph is out of town and Joseph does not know how to reach his parents. As a last resort, John posts a message on a French classical music forum, linking to an EBay ad. However, none of the classical music fans on the forum are responsive enough and some of them even become suspicious that the electronic ticket being sold by this new forum user is actually fake.

John's problem would be very easy to solve if he was able to contact someone that, albeit not knowing him directly, was at the same time interested in the concert and would trust him enough to buy an electronic ticket from him. This is exactly what can be achieved by combining the discovery potential of implicit networks, with the real-world guarantees provided by trusted social links in explicit ones. While implicit networks do not convey any kind of trust, explicit links almost always carry some kind of trust properties resulting from being friends, co-workers and so on. In our example, the implicit network allows John to identify people that could be interested in the concert. Among these, he discovers François, a music teacher from Paris who is trying to buy two tickets for one of his students and himself. A cross check on the explicit network then allows John to assess François's trustworthiness. He finds out that François is actually the cousin of a French colleague of his wife. This allows the two to gain confidence in each other and thus complete a safe transaction without external help.

Our contribution: TAPS (Trust-Aware Peer Sampling), a novel protocol that operates by directly incorporating trust relationships extracted from an explicit social network into a gossip-based overlay. This provides each user with a set of neighbors that are not only similar but that are also be highly trusted. Through extensive simulations, we show that TAPS achieve performances that are comparable to those obtained by protocols equipped with global system knowledge. This makes our solution directly applicable to situations like the social transaction example described above. Moreover, our results open new directions for making existing gossip-based applications more robust in the presence of unreliable users. The evaluation of TAPS will be presented jointly with the one of PTAPS in the next chapter.

2 Objective

2.1 Requirements of the system

Here we detail how the system should behave, what service it should provide to the users, and what it should ensure against malicious users.

Bring trust into gossip-based protocols As described earlier, trust is a necessary component in online services requiring interactions between users. This is especially true for P2P systems, as in this context it becomes very difficult to assert the users' identities. We are interested in a specific class of P2P protocols, the gossip-based protocols. Gossip-based protocols are unstructured. On the contrary of structured P2P protocols, where peers are organized in a structure and each peer is required to interact only with a limited subset of the peers in the system, in an unstructured P2P protocol peers potentially interact with all the users of the system. This increases further the uncertainty of interactions between users in gossip-based protocols, and it is necessary to enhance them with a trust management system in order to build applications requiring a high level of trust between users, such as an online marketplace.

For reasons described previously, we chose to rely on an underlying explicit social network to manage the trust between users instead of a reputation-based system.

Trust estimation The objective of the protocol is to add trust into gossip protocols, thus it should provide each user with information regarding the trustworthiness of any peer it is lead to interact with, but not necessarily to provide trust information for all peers in the system. This trust information is extracted from an underlying social network, in which acquaintances (that are connected in the social network) assign trust values to each-other, based on what they know of each-other. We make two assumptions on the way users evaluate each-other's trustworthiness: 1) if a user knows that one of his acquaintances is not honest, she will not assign her a high trust value, unless the user is herself dishonest and they are colluding; 2) if a user does not know another user, or know very little about her, but still adds her as an acquaintance in the social network, she will not assign her a high trust value.

Accountability In addition to a trust estimation, the protocol should ensure the accountability of the users. If two users *Alice* and *Bob* execute a transaction, and *Bob* does not perform his part of the contract, *Alice* should be able to identify him. To this end, in addition to the trust estimation, the protocol should provide the path between *Alice* and *Bob* corresponding to the trust estimation. Both *Alice* and *Bob* should be able to use this path in order to navigate the social network and finally reach the other user. Since transactions will be conducted between nodes connected with a highly trusted path, and since we assume nodes only give high trust values to the users they know, all links between nodes in a trusted path should be between nodes that actually know each other, and know each-other's identities.

Reliability The trust estimation provided by the protocol should be reliable. More precisely, a malicious user should not be able to lie in order to make other users believe that he or any other node is more trustworthy than he actually is. If *Alice* has a path towards *Bob*, she should be able to compute the exact trust value of the path even if *Bob* is trying to lie about it. This verification should be done at least before any transaction.

Sybilproofness We do not make strong assumptions on the underlying social network, and we do not expect it to ensure a strong verification of the users' identities. Thus, we assume that a node can create as many identities as he wants, connect them between each other the way he wants, and attempt to establish connections with any other nodes of the social network. The system should ensure that even though a user might create a group of Sybil nodes, it cannot increase his own trust value or decrease the one of other users. To achieve this, the trust model of the system should be designed according to the definition of Sybilproof reputation mechanisms presented in Section 3.3 of Chapter 2.

3 Trust Model

In this section we describe the trust model used in TAPS.

3.1 Underlying social network

TAPS operates on top of a social network. We assume that each node running TAPS has access, through this social network, to the list of acquaintances of its user along with their trust value. In addition, we assume that the social network also provides TAPS with the IP address and port of the user's acquaintances that are currently online. Thus, from the TAPS point of view, the social network can be seen as a list of triplet $< UserId, Trust, Address >$.

We do not put any additional restriction on the social network, it can be a centralized social network such as Facebook, Google+ or RenRen, as well as a decentralized one such as Diaspora*.

3.2 Requirement of the trust model

The trust of a path cannot exceed its least trusted edge If on a path P between users i and j, there exists a user k that is very poorly trusted by her predecessor, her opinion about the next node cannot be trusted. Indeed, there are two reasons why a user can give a low trust value to another user: 1) she does not know the other user well enough, and thus cannot assert that this user is trustworthy; 2) she knows the other user well enough to know that this user should not be trusted. Thus, the trust over a path should never be greater than its least trusted link:

$$\tilde{t}_{1,n} \leq \min_{i=2}^{i=n} t_{i-1,i}.$$

Uncertainty adds up The untrustworthiness of a path should increase with the number of poorly-trusted links in the path. Consider the case of a path P_1 of length k containing one link with a medium trust value, and all other links with the maximal trust value. Now consider the case of another path P_2 of the same length, but containing only links with a medium trust value. Obviously, P_2 carries more uncertainty than P_1 and P_1 should be better trusted than P_2.

Decreasing with path length Consider two paths, $P1$ of length 3 and $P2$ of length 10, both only containing links with the maximal trust value. Obviously, the path $P1$ with only 2 intermediary nodes should be better trusted than $P2$ containing 9 intermediaries. Indeed, whatever the trust value on each link of the path, each additional intermediary brings up some additional uncertainty. Thus, independently of the trust value on the links, adding one extra hop to a path should always decrease its trust value.

$$\tilde{t}_{1,n} < \tilde{t}_{1,n-1}.$$

3.3 TAPS trust model

Trust between users is sometimes represented using two values, the trust itself and the confidence the user has in his judgment. This is to differentiate the case where a user is known to be untrustworthy (low trust and high confidence) from the case of a user who seems trustworthy, but for which we have very little information (high trust, low confidence).

While this level of details is useful in the context of a reputation system where the objective is to gather information from multiple sources in order to estimate how trustworthy a node is, it is not in our case. Indeed, our objective is to provide users with references to a set of trustworthy users, and not to evaluate the trustworthiness of all the nodes in the network. From the point of view of the trustworthiness, if a node has either a low trust score or a low confidence score, the result is the same: the information available does not allow us to trust it. Instead, we represent trust as a single value comprised in $[0 : 1]$, where 0 is the minimum, and 1 is the maximum trust value.

To ensure that 1) the inferred trust value of a path cannot exceed its least trusted edge and 2) that uncertainty adds up, we compute the inferred trust of a path as the product of the trust on each edge. Given a path $u_1, u_2..., u_n$ with trust values $t_{1,2}, t_{2,3}..., t_{n-1,n}$, the inferred trust between u_1 and u_n is :

$$\tilde{t}_{1,n} = \prod_{i=2}^{i=n} t_{i-1,i}.$$

Additionally, to ensure that 3) the trust decreases with the path length because of the uncertainty brought by each additional node in the path, we introduce a trust transitivity coefficient τ. τ is a coefficient in $[0 : 1]$ that tunes how quickly the trust

decays with the length of a path by applying it for each intermediary node in the path (i.e., each node in the path except the two extremities)

This yields to a new equation for the inferred trust between u_1 and u_n :

$$\tilde{t}_{1,n} = \tau^{n-2} \prod_{i=2}^{i=n} t_{i-1,i}. \tag{3.1}$$

3.4 Sybilproofness

In Section 3.3 of Chapter 2 we introduced the notion of Sybilproofness from [CF05]. If a reputation system is Sybilproof, then a user cannot increase her reputation by creating Sybil nodes, and cannot either decrease the one of users better trusted.

They describe a reputation system as a set of two functions:

- g: computes the reputation over a path,
- $\oplus$: aggregates the reputation of several paths.

A reputation system is Sybilproof if these two functions satisfy these properties:

- Given two paths P and P' such as $P \subseteq P'$, then $g(P) >= g(P')$.
- If a path P is split into two disjoint parallel path $P1$ and $P2$, then $g(P) >= g(P1) \oplus g(P2)$
- g does not decrease if the trust of an edge increases
- $\oplus = max$

In TAPS, when a node is faced with several paths leading to the same node, it always chooses the one with the highest trust value, thus we have: $\oplus = max$. The function that computes the trust over a path is $\tilde{t}_{1,n} = \tau^{n-2} \prod_{i=2}^{i=n} t_{i-1,i}$, which satisfies the above properties. Thus, the trust model used by TAPS makes it Sybilproof.

4 Trust-aware peer sampling

Trust-aware peer sampling, or TAPS, is a service providing a node with a continuously changing set of references to other nodes associated with trusted paths, the same way a random peer sampling service provides a node with references to random nodes. Similarly to the random peer sampling, references to other nodes can be associated with any kind of meta-data, such as the user's profile.

The trust-aware peer sampling can be seen as an additional layer working on top of a random peer sampling service, adding trust information extracted from the social network to the nodes found through the RPS. However in practice it would be impractical to implement it as a layer over the RPS. Indeed, each time a new node reference is discovered through the RPS the system would have to search for a trusted path on

the network. Searching the most trusted path, or even any trusted path, between two nodes in a distributed network is expensive, especially in the case of very large networks. Moreover, if the trust-aware peer sampling was built on top of a random peer sampling, it would provide trusted paths towards random nodes of the network, which would in majority be far away and thus poorly trusted. Given that the point of the trust-aware peer sampling is to find nodes that are trustworthy, most of the discovered entries would be useless.

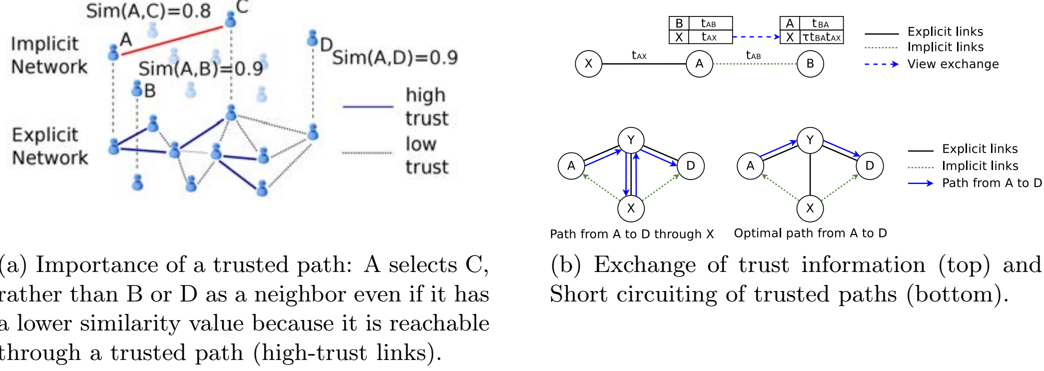
In order to avoid the prohibitive cost of searching the graph, and allow the trust-aware peer sampling to identify highly trusted nodes instead of random ones, we instead create a new block that can transparently replace the random peer sampling block. By integrating the trust and trusted paths management at the same layer as the node discovery instead of adding it on top of it, we are able to provide a trust-aware peer sampling service that do not require expensive crawling of the graph in order to estimate the trust between two nodes. This algorithm, that we call TAPS, is based on the gossip-based peer sampling protocol. The objective of TAPS is to explore the graph in search of trusted nodes and provide them to an upper layer that will select a subset of these node according to some metric (for example the nodes providing the best trade-off between trust and similarity). In the following, we will describe the core of the TAPS protocol as well as a list of functionalities that greatly improve its performances.

4.1 Overview

In gossip-based peer sampling [JVG+07], when a node joins the system, a bootstrap mechanism provides it with a set of random node references to initialize its view. Then, it periodically selects one of the node from its view, and they both exchange a sample of their respective views. By doing this operation periodically, the view of a node converges, after a few initial cycles, towards a random sample of the network that is continuously changing from cycle to cycle.

TAPS follows the general structure of gossip-based peer sampling, but extends it to integrate trust information about the users and bias the nodes selected towards trusted nodes. To this end, TAPS attach additional information to each node reference. In addition to the ID and IP address of the node, its profile and a timestamp, TAPS adds the inferred trust value for this node as well as a path in the social network starting from the node holding the reference and ending to the referenced node. When a node joins the system, its view is bootstrapped with the node's friends in the social network. Then, the node periodically selects a node from its view and they exchange samples of their respective views. After every view exchange, any reference R received by a node N from the sender S is modified by N in order to update the information about the trust and the path from N to R. This is done by concatenating the paths $N - S$ and $S - N$ and by aggregating the trust values of the two paths. Using this mechanism, the view of a node contains a set of changing references to other nodes for which a path in the social network and a trust estimation are always known.

For clarity, in the following we will use the term host to refer to the node we are mainly concerned with, and the term node to refer to any other node of the system.



(a) Importance of a trusted path: A selects C, rather than B or D as a neighbor even if it has a lower similarity value because it is reachable through a trusted path (high-trust links).

(b) Exchange of trust information (top) and Short circuiting of trusted paths (bottom).

Figure 3.1: Trusted paths.

4.2 Views

TAPS associates each node with two views, the EXPLICIT VIEW and the TAPS VIEW.

The EXPLICIT VIEW contains the friends of the host in the social network. Each entry of the EXPLICIT VIEW contains the target node ID, its IP address (we assume that the social network knows when the user is online and can provide us its current IP address), and the trust value extracted from the social network. When the social network is modified, for example when a new link is created between the host and another node, the new entry is simply added into the EXPLICIT VIEW of both the host and the other node, and TAPS will automatically take it into account in the future view exchanges.

The TAPS VIEW contains entries for nodes who are not in the EXPLICIT VIEW of the host, meaning that they are not friends in the social network. Each entry of the taps view contains the ID of the target node, its current IP address an inferred trust value, a trusted path and a timestamp. The trusted path is a list of node IDs, where any two successive nodes are friends in the social network, the first node in the path being a friend of the host. The inferred trust value is the value of trust inferred from the trusted path using Equation 3.1. Finally, the timestamp indicates when the target's profile was updated for the last time.

4.3 Initialization

When a node joins the network, its EXPLICIT VIEW is empty and TAPS initializes it with neighbors from the node's social network. Once the EXPLICIT VIEW is initialized, the node initializes its TAPS VIEW. The TAPS VIEW is initialized by using entries from the EXPLICIT VIEW. If the EXPLICIT VIEW contains more entries that the maximal size of the TAPS VIEW, a random sample of the EXPLICIT VIEW is selected to fill the TAPS VIEW. Otherwise, all the elements of the EXPLICIT VIEW are added to the TAPS VIEW. For each of these entries, a trusted path is created containing only the target node, and the inferred trust value is set to the actual trust value between the two nodes.

Once the EXPLICIT VIEW and TAPS VIEW have been initialized, the node is in a valid state and can start executing the protocol.

4.4 View exchanges

The objective of TAPS is to provide the node with a set of references to other nodes that is continuously changing over time. For this reason, the TAPS VIEW needs to be updated periodically with new entries. To this end, each node periodically selects one node from either of its views, and they both exchange a sample of their respective views.

For simplicity, we call the node initiating the view exchange the source, and the node selected for the view exchange the target.

Selecting a target Periodically, each node selects a target node to perform a view exchange with it. This node is picked with an equal probability from either the EXPLICIT VIEW or the TAPS VIEW. It is important that the target can be selected from both views, and not from one of them only. If the target was selected only from the EXPLICIT VIEW, the discovery of distant nodes in the social network would take a lot of time. Indeed, in this case after each view exchange the size of the paths in the TAPS VIEW of a node would increase by at most one hop. On the other hand, if the target was selected only from the TAPS VIEW, then the length of the paths in the TAPS VIEW would increase continuously, by at least 1 hop after each view exchange. This means that after only a few dozens of cycles, all the paths of a node's TAPS VIEW would be dozens of hops long, which would make them useless as paths needs to be short in order to be trustworthy.

Selecting targets from one of the two views at random provides the benefit from both: the exploration of the network is fast thanks to the exchanges with targets from the TAPS VIEW while the exchanges with targets from the EXPLICIT VIEW allow to discover short paths.

Selecting a view sample Once the source selected its target, it selects a sample from its views that it will share with the target. Here again, the nodes to share with the target are selected from both the EXPLICIT VIEW and the TAPS VIEW. The reason for that are the same as for the target selection. If the source was only sending nodes from its TAPS VIEW, the path lengths would keep increasing over time. And if only nodes from the EXPLICIT VIEW were selected, paths length would only increase by at most one hop after each exchange, which would make the discovery of distant nodes slower.

Once the source has selected the nodes from its views to share, it prepares the message to send to the target. It includes, for each node selected, the node ID, its current IP address, its profile, the trusted path from the source to this node and the associated inferred trust value. The source also includes the path and inferred trust value it knows between itself and the target. This last information is important, as the target might not know a path towards the source, thus this information must be attached so that the target can build a complete path from itself to the nodes it receives.

Once this message is ready, the source sends it to the target. Upon receiving such a message, the target first selects its own view sample to share with the source by following the same procedure and sends it to the source. Then, it includes the received entries into its own TAPS VIEW.

By selecting both the targets and the entries to share from both the TAPS VIEW and EXPLICIT VIEW, we perform all possible kind of exchanges:

- TAPS to TAPS: Quickly discovers distant part of the network
- EXPLICIT to EXPLICIT: Create new short paths to prevent the TAPS VIEW from being filled with very long paths.
- EXPLICIT to TAPS: Share with a distant node the node's direct friends
- TAPS to EXPLICIT: Share with a direct friend the distant nodes that we discovered

4.5 Path concatenation and trust inference

When a node B receives a set of entries from a node A, it first needs to update them before adding them to its own TAPS VIEW. Indeed, the entry corresponding to a node X sent by A contains a path going from A to X, and the inferred trust value from A to X. Before adding the entry of X to its view, B needs to build the path $p_{B,X}$ and infer the value $t_{B,X}$.

As an example, consider a node A exchanging information with another node B as shown in the top diagram of Figure 3.1b. A sends to B a subset of its view as well as the path $p_{A,B}$ it has to reach B and the value it has for $t_{A,B}$, the inferred trust between A and B. B uses this information to update the entries it received from A before adding them to its own view.

Specifically, let $p_{A,X}$ be the path from A to X , and $t_{A,X}$ be the associated inferred trust value, then B computes its own path and trust for X, as follows. First B verifies if it already has a path $p_{B,A}$ associated with a trust value $t_{B,A}$. If it does and $t_{B,A} > t_{A,B}$, then it keeps this value and the associated path for A. Otherwise, it inverts the path $t_{A,B}$ and adds it to its view with the trust $t_{A,B}$, replacing the previous entry for A if any. Given that a path is composed of a list of node ids, inverting a path is done inverting its list of ids.

It then uses the selected $p_{B,A}$ and $t_{B,A}$, and computes the resulting path $p_{B,X}$ and inferred trust value $t_{B,X}$ as

$$p_{B,X} = p_{B,A} : p_{A,X} \quad \text{and} \quad t_{B,X} = \tau \cdot t_{B,A} \cdot t_{A,X}$$

where : denotes path concatenation.

This provides B with a path going from it to X through A, as well as the inferred trust value $t_{B,X}$. This is applied to all the entries received from A, and all the updated entries are added to a temporary view V_A. The next step is to merge this temporary view with B's TAPS VIEW (that we will call V_B for clarity)

4.6 Merging views

The size of the TAPS VIEW is limited, thus when B receives new entries from a node, it has to select a subset of these entries to keep in its view.

During a view exchange a node can receive many new entries for nodes that are far away, or poorly trusted. This is especially true when the exchange is done with a node from the TAPS VIEW. We want to limit the presence of these nodes in the TAPS VIEW are they are less useful, but we do not want to strictly prevent them from entering the TAPS VIEW as some nodes with a relatively low trust can still be useful. In order to filter out most of the poorly trusted nodes without banning them completely, we randomly select which entries to keep with an advantage for the nodes with a high trust value.

Consider node B that needs to update its view V_B with information received from a node A, V_A. First, B computes the union of the two views $U = V_B \cup V_A$. Then it selects n_{TAPS} nodes from $V_B \cup V_A$ one after the other by associating each candidate node with a probability obtained by dividing its trust values by the sum of all the trust values of the nodes in $V_B \cup V_A$ that have not been selected. Therefore, the probability that a node be in the selected sample increases with its trust value. This causes the view to be biased towards nodes with higher values and thus reduces the likelihood that a node with a low trust value may enter the TAPS VIEW.

4.7 Multiple reference to the same node

When a node merges two views, it happens that a reference for the same node appears in the two views. In the presence of multiple paths and trust values for the same node X a node B always selects the entry with the highest trust value. If the profiles associated with the two entries, the node keeps the most recent version according to the timestamp. This might lead B to merge the path from one entry with the meta-data from another one (both being about same node X).

4.8 Paths reversal

When a node B receives an entry X from node A, it concatenates the two paths in order to form a path from B to X. But if B already knows a path towards X, it can reverse the path $p_{A,X}$ and concatenate it with its path $p_{B,X}$ in order to create a path towards A going through X. If $\tau \cdot t_{B,X} \cdot t_{X,A} > t_{B,A}$, then B can replace its current path to A with the one it just created, and update all the entries received from A with this new path.

4.9 Shortcutting paths

Consider the situation depicted in the bottom diagram of Figure 3.1b. Node X holds a reference to D with a trust value $t_{X,D}$ and a path going through Y and sends it to A. Node A should combine this with the trust value it has for node X, $t_{A,X}$, also obtained through Y. However, this would lead to a path that uselessly goes twice through node Y and once through X.

Node A can prevent this by short-circuiting the path thereby also improving its trust value. Specifically, A knows that the aggregated impact on trust of the path segment YX cannot be greater than τ^n, n being the number of useless links in the path, each link being counted once for each time it is traversed (n = 2 in this case). It can therefore conclude that the trust value of node D as seen from A, $t_{A,D}$ is at least the following:

$$t_{A,D} \geq \frac{t_{A,X} \cdot t_{X,D}}{\tau^n}.$$

4.10 Trust verification

A final aspect of TAPS is its trust-verification mechanism. While not an architectural component, this mechanism plays a major role in guaranteeing safe transactions based on the information provided by TAPS. Specifically, trust verification allows nodes to obtain a confirmation for the trust values of the entries that are in their TAPS VIEW. This provides protection from nodes that attempt to cheat on their trust values when communicating with nodes that are not direct friends. Consider Figure 3.1b (bottom): node D could try to enter A's view by making A believe that it has a high-trust link to node Y. To prevent this, A asks D to forward a verification message back to A along the trusted path. The message starts with a trust value of 1. Nodes along the path multiply the message's value by τ and by their trust for the node they received the message from. Y thus multiplies 1 by $\tau t_{Y,D}$ in the example. This process causes the verification message to reach A with the correct value for $t_{A,D}$ thereby invalidating D's cheating attempts.

5 Trust-aware clustering

The TAPS VIEW constitutes the main source of information for the trust-aware clustering protocol (TAC). TAC is a variation of well-known protocols [VVS05, BFG+10] and maintains a TAC VIEW that collects the nodes that offer the best compromise between trust and similarity. Upon receipt of another node's view, a node X combines the received view and its own TAC VIEW, and selects the entries associated with the best trade-off between similarity and trust.

Score computation Using the similarity and the trust it has toward a node N, a node X can compute a score $\sigma_{X,N}$ as a weighted product between its trust and its similarity:

$$\sigma_{X,N} = {s_{X,N}}^{2-\epsilon} {t_{X,N}}^{\epsilon},\ \epsilon \in [0, 2],$$

where ϵ, the trust weight, determines the importance of trust in the trade-off, $t_{X,N}$ denotes the trust between X and N and $s_{X,N}$ is the similarity between X and N.

TAC view exchanges Each node periodically selects the node with the oldest timestamp in its TAC VIEW and exchanges the content of its TAC VIEW with it. These ex-

changes speed up convergence with respect to only relying on information from the TAPS layer by providing nodes that tend to have higher similarities.

5.1 Minimum-Trust Threshold

An additional mechanism employed by TAPS to limit the proliferation of long paths is a dynamic threshold on trust values. For any node N, this is defined as the minimum trust value that a hypothetical node A with similarity 1 would need to enter B's cluster. Let σ_i denote the score of a node i from the CLUSTER view of n, and let t_i and s_i be respectively their reciprocal trust and similarity values. Because of the definition of the score ($\forall i : \sigma_i = t_i^{\epsilon} s_i^{2-\epsilon}$), such a perfectly similar node would have a score of $\sigma_{n'} = t_{n'}^{\epsilon}$. For it to enter the CLUSTER view of n, this score would need to be higher than the lowest score of a node currently in the view, i.e. $t_{n'}^{\epsilon} > \min\{\sigma_i, i \in \text{CLUSTER}\}$, which is equivalent to $t_{n'} > \min\{\sigma_i, i \in \text{CLUSTER}\}^{1/\epsilon}$. Clearly, this minimum requirement also applies to nodes that have lower similarity values as their scores are necessarily lower than those of perfectly similar nodes with the same trust values. Thus, because the goal of the TAPS VIEW view is to feed the corresponding EXPLICIT VIEW, TAPS ignores all the nodes that have trust values lower than $\min\{\sigma_i, i \in \text{CLUSTER}\}^{1/\epsilon}$.

This threshold is dynamic: a node recomputes it whenever it modifies its cluster view. It then removes from its TAPS VIEW all the nodes whose trust values are lower than the threshold, and refuses the insertion in the view of similarly untrustworthy nodes. This process gets more and more refined as nodes select the candidates with the best scores as their neighbors. This causes a corresponding increase in the value of the threshold, making nodes more and more demanding as their TAC VIEWS improve.

Finally, it is worth observing that the trust threshold induces a maximal length for acceptable paths that can be calculated by: $d_{\max} = \log_{\tau}(t_{\min})$. Ultimately, this causes the threshold to have a positive impact on the quality of the TAPS VIEW and the EXPLICIT VIEW because it allows nodes to focus on the parts of the explicit network in which they can find the best neighbors.

6 Concluding remarks

We presented trust-aware peer sampling, a novel distributed application enabling trusted collaborative actions between similar users in a social-network environment. TAPS proposes a solution to the challenges of trust and accountability in P2P systems, by creating an RPS-like overlay taking into account the mutual trust expressed by users when joining an explicit social network.

In the next chapter, we will presents PTAPS, an enhanced version of TAPS that trades parts of TAPS's ability to find the most trustworthy and similar nodes with strong privacy guarantees with respect to the users' social networks. The evaluation of the performances of TAPS is deferred to the next chapter, where it will be evaluated in parallel with PTAPS.

Chapter 4

Privacy preserving trust-aware peer sampling

1 Introduction

In the previous chapter we presented trust-aware peer sampling, a protocol enhancing traditional peer sampling protocols by providing trustable peers to the users, as well as accountability. However, while TAPS can improve the safety of distributed applications by allowing users to avoid malicious users, and provides a mechanism to identify misbehaving users if required, it does nothing to protect the users privacy.

In the case of the TAPS, the information that is asked to the users is the list of their friends along with an estimation of how trustworthy these friends are. These information, especially the second trust information, can legitimately be considered as very sensitive. This is probably the kind of information users would not be happy to share with everyone.

In this chapter we propose a modified version of TAPS, called PTAPS, achieving the same goal as TAPS while ensuring that the information provided by a user concerning her friends in the social network, as well as the associated trust information, network cannot be accessed by users who are not one of the user's friends.

PTAPS differs on a few key points from TAPS: 1) The trusted paths are encrypted in a way that allows the path concatenation and routing as in TAPS, but prevents an observer from discovering any friend relationship by observing the encrypted path; 2) The view exchange are constrained, so that a user will never disclose her direct friends to a user who is not one of her friends; 3) Paths going several times through the same node are never shortcutted.

We provide a formal proof that with these modifications a node will never be able to know for sure that any pair of nodes is connected by a direct link if he is more than two hops away from these two nodes.

Finally, we evaluate the performances of PTAPS and show that these modifications only slightly affect the performances when compared to TAPS.

2 Privacy-preserving Trust-Aware Peer Sampling

TAPS assumes an honest-but-curious adversary that is also completely passive.Specifically, the adversary should not actively stock information about trusted paths in order to calculate direct trust values between arbitrary nodes. TAPS, in fact, hides the trust values of the individual segments composing a trusted path by disclosing only an aggregate trust value. As a consequence, an adversary equipped with sufficient memory and time could use TAPS to collect and store information about a large portion of the paths in the network, and combine them to compute the trust associated with the links they consist of. Consider a node A receiving information on a trusted path ABX with value t_X and another $ABXY$ with value t_Y. A can easily compute the trust value of the link XY, as $t_{X,Y} = \frac{t_Y}{\tau t_X}$.

To understand the impact of this kind of attack, we ran simulations in which nodes store the information they receive and use it to infer the trust values associated with explicit links. An average node is able to discover the trust values of 40% of the 14000 explicit links in the social network in as little as 1000 cycles.

We address this vulnerability of TAPS by proposing PTAPS, a protocol that trades off part of TAPS's ability to discover the best paths for provable privacy guarantees even with active attackers. We present the details of the protocol in Section 2.1, and then prove its privacy properties in Section 2.2.

2.1 Protocol Description

PTAPS augments TAPS with three novel features. First, it reduces the amount of information exchanged, to provide provable privacy guarantees. Second, it hides the intermediary nodes of a path from users, thereby preventing the attack described above. Finally, it strengthens the trust threshold to prevent all communication with untrusted users.

2.1.1 Non-disclosure of explicit friends.

The goal of PTAPS is to prevent an attacker from inferring the existence of an explicit link between arbitrary nodes outside its social circle, that is, between nodes that are at least two hops away from the attacker. The first step in this direction is to make it impossible to tell which of the paths and trust values advertised in a view update correspond to a node's explicit friends. An attacker that can identify a node's explicit friends can in fact use their trust values to infer those associated with the node's friends' friends, and so on.

To avoid this problem, PTAPS modifies the exchange mechanism of TAPS by allowing a node to have entries corresponding to indirect paths to its explicit friends in its TAPS VIEW. This allows nodes to learn alternative longer paths to their explicit friends, thereby hiding their status of friends from other nodes. Consider a node n with direct neighbors $N(n)$ that is performing a PTAPS exchange with another node p. As in TAPS, n chooses p either from its direct neighbors or from its TAPS VIEW with equal probability ($P(p \in N(n)) = P(p \in tapsview) = \frac{1}{2}$). If $p \notin N(n)$ (p is not an explicit friend), then

n selects the entries to send to p only from its TAPS VIEW and not from its EXPLICIT VIEW. This achieves the goal of hiding the explicit-friend status of the entries sent to non-friends during gossip exchanges in two ways. First, it replaces one-hop paths with longer paths making paths to explicit friends indistinguishable from the others. Second, it causes the probability of sending a TAPS entry about an explicit neighbor to be comparable to that associated with other nodes, making it difficult to guess the identity of friends based on statistical attacks.

However, nodes use this modified exchange mechanism only when exchanging information with non-friends. It is easy to see that a node must know the identities of its friends' friends in order to bootstrap the process. As a result, if $p \in N(n)$ (p is an explicit friend) n continues to operate as in TAPS and chooses a view to send to p by selecting each entry with equal probability from the TAPS VIEW or the EXPLICIT VIEW.

2.1.2 Trusted-path protection

Being unable to identify another node's friends is essential to protect trust information, but it is impossible if attackers have access to complete path information. As a result PTAPS also employs a novel path-encryption and routing mechanism. Specifically, it guarantees that if a node is represented in clear text in a path, then its predecessor and its successor appear in encrypted form. Moreover, a given node in a path can only decrypt information about its own predecessor and successor. This provides each node with the ability to manipulate and combine paths to other nodes, while hiding the information regarding the internal links of the path. For simplicity, we start by describing a basic version of path encryption that only hides the links of a path, and then modify it so that it also hides path length. Finally, we present the details of the path encryption protocol.

Basic path encryption Essentially, an encrypted path consists of a sequence of routing blocks, each of them containing information about the next or the previous hop in the path. These blocks can either be secret, i.e. the identity of the node they refer to is encrypted, or public, i.e. the node they refer to is unencrypted and thus publicly known. A secret routing block toward a node F, f_N, consists of two sub-blocks: an identity block containing F's identity, and a randomly generated block. The concatenation of the two sub-blocks is encrypted by a key private to N. Because of this, N is the only node that can decrypt the block and thus know that it refers to F. Also, the presence of a random-noise sub-block means that two routing blocks encrypted by the same node N and referring to the same node F will look different. We use the notations $\mathrm{f_n}$ and $\mathrm{f'_n}$ when we need to emphasize the fact that two blocks $\mathrm{f_n}$ and $\mathrm{f'_n}$, while containing the same data, use a different random block. A public routing block f on the other hand simply contains F's identity, which is not hidden or encrypted by any key.

Since we need to be able to follow an encrypted path both forwards and backwards, routing blocks always appear in pairs representing links. Specifically, a link $N - F$ contains a routing block f or $\mathrm{f_n}$ for node F, followed by a routing block n or $\mathrm{n_f}$ for node N, yielding for example $\mathrm{f_n n_f}$. The reason for reversing the order of N and F in the link

Algorithm 1 Maintaining Routing Information.

```
(01) Periodically
(02)     for all F ∈ ExplicitView
(03)         F_N = encrypt(F)
(04)         F'_N = encrypt(F)
(05)         send Update(F_N, F'_N) to F

(06) When Update(F_N, F'_N) received by F from N
(07)     N_F = encrypt(N)
(08)     N'_F = encrypt(N)
(09)     ExplView[N].Secret ← N_F · F_N
(10)     ExplView[N].Pub_out ← N · F'_N
(11)     ExplView[N].Pub_in ← N'_F · F
(12)     send Reply(N_F, N'_F) to N

(13) When Reply(N_F, N'_F) received by N from F
(14)     ExplView[F].Secret ← F_N · N_F
(15)     ExplView[F].Pub_out ← F · N'_F
(16)     ExplView[F].Pub_in ← F'_N · N
```

representation (e.g. $\mathrm{f_n n_f}$) is to facilitate the routing process. Specifically, when reading the link from left to right, the first of the two blocks will be used by node N to identify F as a next hop. The second block will instead used by F to identify N as a previous hop. Reversing the link ($\mathrm{n_f f_n}$) enables navigation in the opposite direction, F using N as its next hop, and N using F as its previous one.

This model allows us to represent any path as a sequence of links. For example, the path consisting of the sequence of nodes A,B,C, and D can be represented by concatenating the links $A-B$, $B-C$, and $C-D$, yielding $\mathrm{b_a a \cdot c_b b_c \cdot d c_d}$. Note that the blocks referring to the first and last node are public as they identify the endpoints of the path.

Those corresponding to the first and the last node are not the only blocks in a path that may be public. Blocks corresponding to intermediary nodes may also be public as a result of the way the path was constructed. Nonetheless, as stated above, the goal of the protocol is to prevent an attacker, X, from discovering the explicit friends of any other node, N, that is at least 2 hops away. This leads to two requirements. First (i), a secret routing block referring to a node N must never be used together with a public routing block referring to the same node N. Second (ii), a path must never concatenate the public routing blocks corresponding to two neighboring nodes. Rather if a node in a path is represented by a public routing block, then both the previous and the following nodes in the path must appear as secret blocks. In this respect, it is worth observing that the relevant order is that of the nodes in the path and not that of the blocks in the path representation.

As an example, consider the path consisting of the sequence of nodes F, A, B, C, and D in Figure 4.1b. The representation $\mathrm{a_f f \cdot b a_b \cdot c_b b \cdot d c_d}$ is allowed (note that the

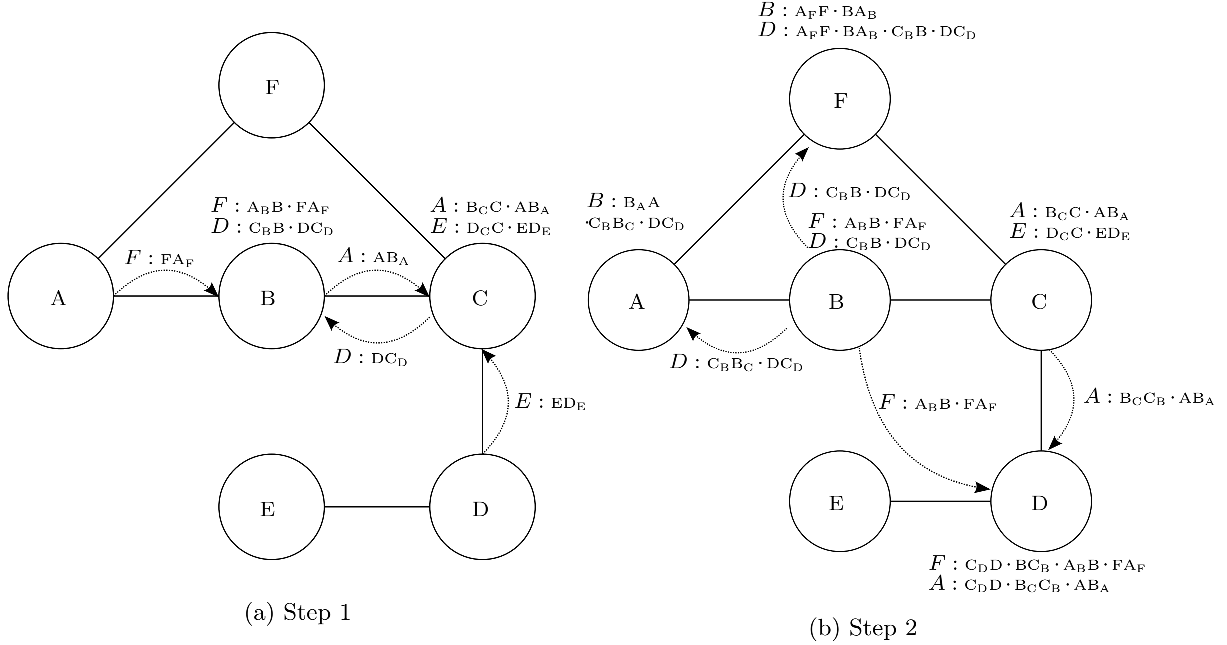
Algorithm 2 Path management during view exchanges.

```
(01) Before sending a path P_AZ to an explicit friend C
(02)     if Z is an explicit friend of A
(03)         send ExplView[Z].Pub_out to C
(04)     else
(05)         Given B the node after A in P_AZ
(06)         P_BZ ← RemoveFirstLink(P_AZ)
(07)         send ExplView[B].Secret · P_BZ to C

(08) Before sending a path P_AZ to an Taps node C
(09)     Nothing special to do
(10)         send P_AZ to C

(11) When a path P_BA is received by a node A from a node B with P_BA.Dest = A
(12)     if P_AB ∉ Tapsview
(13)         P_AB ← ReversePath(P_BA)
(14)     else
(15)         if P_AB.trust < P_BA.trust
(16)             P_AB ← ReversePath(P_BA)

(17) When a path P_BC is received by a node A from an explicit friend B
(18)     // The path P_BC starts by a link where B is hidden
(19)     P_AC ← ExplView[B].Pub_in · P_BC

(20) When a path P_BC is received by a node A from an non-friend B
(21)     // The path P_BC starts by a link where B is public
(22)     P_AC ← P_AB · P_BC
```

node preceding B is A and not F; likewise the one following B is C and not D), while $\mathrm{a_f f{\cdot}ba_b{\cdot}c_b b_c{\cdot}dc_d}$ is not. The first one only reveals that F and B share a friend, and that so do B and D, but it does not give any information about the identities of these common friends. However, the second representation, because of the concatenation of the blocks $\mathrm{ba_b}$ and $\mathrm{c_b b_c}$, reveals that B is the first node of the link $\mathrm{c_b b_c}$. If a node had both this information and for example the path $\mathrm{b_g g \cdot c_b b_c \cdot dc_d}$, it could see that the $\mathrm{c_b b_c}$ block is common between the two and find out that B and G are friends because successive links always have a common node, B in this case. If instead the two requirements stated above are satisfied, no attack is possible as we will show in Section 2.2.

Hiding path length As a further protection measure, PTAPS also has the ability to hide the length of an encrypted path. Specifically, a node N may use additional encrypted blocks, $\mathrm{n_n}$, containing its own identifier (and the random noise). Whenever N writes the identity of a node F either as a successor or as a predecessor in a path, it writes the $\mathrm{f_n}$ block followed by a random number of copies of the $\mathrm{n_n}$ block (each with a different noise sub-block), thus making it impossible for another node to guess the length of a path.

Due to path reversal, a sequence of blocks to be decrypted may have either of two



(a) Step 1

(b) Step 2

Figure 4.1: Building routing information.

forms: $\mathrm{f_n, n_n, ..., n_n}$, or $\mathrm{n_n, ..., n_n, f_n}$. Moreover it can be followed either by a block $\mathrm{f_n}$ encrypted by n or by a block encrypted by a different node. To decrypt it, n continues to decrypt blocks until it has found a block of type $\mathrm{f_n}$ with $F \neq N$ and has reached either a block encrypted by a different node, or a block of type $\mathrm{s_n}$ with $S \neq N, F$. In the latter case, it leaves $\mathrm{s_n}$ in the path for later decryption. To simplify the presentation of the protocol, in the rest of the paper, we ignore the use of $\mathrm{n_n}$ blocks to hide path length. Their integration in the descriptions and pseudo-code that follow is conceptually simple and is thus left to the reader.

Path-encryption details Algorithms 1 to 3 provide the details of the path-encryption protocol. Each node, N, maintains three additional fields for each entry, F, in its explicit view. The first is an outgoing public link to F, $Pub_{out} = \mathrm{fn_f}$, which will be used to construct encrypted paths with F as an endpoint. The second is an incoming public link to N, $Pub_{in} = \mathrm{f_n n}$, which will be used in paths that have N as an endpoint. Finally, the last field is a secret link, $Secret = \mathrm{f_n n_f}$, which will be used for paths in which neither N nor F is an endpoint.

To initialize these fields, N sends an UPDATE message to F carrying two blocks containing the identity of F. The first secret block, $\mathrm{f_n}$, will be used to create the secret link between N and F. The second one, $\mathrm{f'_n}$, will be used to build the public link referring to N. Upon receiving these blocks, F also generates a pair $\mathrm{n_f}$ and $\mathrm{n'_f}$ and sends it back to N. Then the two nodes have the material to create the three encrypted links needed by the protocol, as described in Algorithm 1.

After initialization, nodes operate as in the TAPS protocol when exchanging infor-

Algorithm 3 Sending and Relaying Application Messages.

```
(01) To send an application message M to X through the path P_X
(02)     NextBlock ← GetFirstBlock(P_X)
(03)     NextHop ← ExtractIdentity(NextBlock)
(04)     P'_X ← RemoveFirstBlock(P_X)
(05)     send AppMsg(M, P'_X) to NextHop.

(06) When AppMsg(M, P_X) received from Sender
(07)     PreviousBlock ← GetFirstBlock(P_X)
(08)     PreviousHop ← ExtractIdentity(PreviousBlock)
(09)     if Sender ≠ PreviousHop
(10)         return
(11)     P'_X ← RemoveFirstBlock(P_X)
(12)     if P'_X = ∅
(13)         deliver M
(14)     else
(15)         NextBlock ← GetFirstBlock(P'_X)
(16)         NextHop ← ExtractIdentity(NextBlock)
(17)         P''_X ← RemoveFirstBlock(P'_X)
(18)         send AppMsg(M, P''_X) to NextHop.
```

mation from their respective TAPS VIEWS. However, some special actions are needed depending on the type of exchanges outlined in Section 2.1.1. The first type of exchanges, explicit-explicit, involve a node that sends information about an explicit friend to another explicit friend. Consider node C in Figure 4.1a sending information about its friend F to its other friend D. First, C sends D the outgoing public link in its routing-table entry for F, $\mathrm{fc_f}$ (lines 2 and 3, in Algorithm 2). Upon receiving this link, D concatenates it with the incoming public link in its routing table entry for C , $\mathrm{c_d d}$. This yields the path $\mathrm{a_b b \cdot fa_f}$ (lines 17-19 in Algorithm 2).

The second type of exchanges, taps-explicit, involve a node that sends a path leading to a taps-view node to an explicit friend. As an example, consider again node C sending a path leading to A to its friend D. Also, let B be the intermediate node in this path: $\mathrm{b_c c \cdot ab_a}$. Node C cannot send this path to D as it is. If it were to do so, D would combine it with an incoming public link, $\mathrm{c_d d}$, and would break privacy, since the path $\mathrm{c_d d \cdot b_c c \cdot ab_a}$ contains node C both in a secret and in a public block, disclosing the fact that $\mathrm{c_d}$ refers to C. Thus, C sends a modified path, where it replaces the public link $\mathrm{b_c c}$ by the secret one, $\mathrm{b_c c_b}$, yielding $\mathrm{b_c c_b \cdot ab_a}$. Node D can then concatenate this path to $\mathrm{c_d d}$, yielding $\mathrm{c_d d \cdot b_c c_b \cdot ab_a}$, which does not pose any privacy problem.

The final type of exchanges are those in which a node N sends information to a node from its TAPS VIEW. As explained in Section 2.1.1, PTAPS forbids explicit-taps exchanges. Therefore, the only content of an interaction with a node from the taps view is always information extracted itself from the taps view. These are easier to handle and require no special processing since paths contained in the TAPS VIEW (and also in the TAC VIEW) all start and end with a public link. This means that they can be concatenated without any special actions. As an example, consider again Figure 4.1a. This time, node C wishes to inform node E about a path to A. As in the standard

TAPS protocol, C provides E with a path $C \rightarrow A$ ($\mathrm{b_c c \cdot ab_a}$) as well as with a $C \rightarrow E$ ($\mathrm{d_c c \cdot ed_e}$) path. Node E can then combine $C \rightarrow A$ with the best path to C it has, either one it already had or the reverse of the $C \rightarrow E$ it just received. In the latter case, E simply takes the blocks of the received path in reverse order yielding $\mathrm{d_e e \cdot cd_c}$. It then combines this $E \rightarrow C$ path with the $C \rightarrow A$ path it received, yielding $\mathrm{d_e e \cdot cd_c b_c c \cdot ab_a}$.

Encrypted paths clearly allow routing of application messages. These include the verification message described in Section 4.10 of the previous chapter, or messages employed to establish transactions between nodes. The routing algorithm is given in Algorithm 3. Consider node F in Figure 4.1b wishing to send a message along its path to node D. Node F picks the first block of the path and extract the identity of the node it contains (line 03). Since the identity of a node is either encrypted or in clear form, extracting it from a block means either decrypting it or simply returning the clear-text value in the block. The identity extracted by F indicates that A is the next node on the path, thus F forwards the message to A along with the path stripped of the first block. Upon reception, node A reads the first block of the received path (line 08), which confirms that F is a previous hop. Then, because the path contains more blocks, A proceeds by extracting its next hop (line 16). The process goes on until the message reaches D, which reads the last block of the path on line 08, confirming node C as a previous hop, and then delivers the message at line 13.

Finally it is worth observing that maintaining routing tables at each node with a predecessor and a successor along each path would quickly lead to a waste of storage resources. TAPS exchanges lead to the creation of a huge number of paths, many of which are simply discarded when better paths are found. While stale paths could, in principle, be garbage collected, this would require expensive coordination among a large number of distant nodes, and would become almost impossible in the presence of churn.

2.1.3 Strong Threshold

In addition to encrypting paths and restricting the dissemination of information about the trust values on explicit links, PTAPS also reinforces the trust threshold we introduced in Section 5.1 of the previous chapter. Specifically, TAPS removes from a node's view all the nodes whose trust values are below the threshold, but it allows the node to exchange information with them, possibly providing them with trust information about itself and other nodes. PTAPS, on the other hand, also forbids such exchanges by ignoring all the communication arriving from nodes whose trust value is below the threshold and by suppressing all outgoing communication towards the same nodes. While this additional measure is not necessary to prevent nodes from discovering trust values on explicit links, it contributes to limit the ability of attackers to build even an approximate map of the network's trust values by eventually confining nodes' communication to the part of the network where there are sufficiently trusted. Finally, it is important to observe that in PTAPS, the trust threshold becomes the primary mechanism for limiting the length of trusted paths, due to the impossibility to perform on-line shortcuts in the presence of encrypted path information.

2.2 Privacy preservation in PTAPS

We now analyze PTAPS's ability to provide strong privacy guarantees by hiding the direct trust values between explicit friends. Intuitively, this ability is provided by the non-disclosure of explicit friends discussed in Section 2.1.1. In the following, we formalize it by clearly specifying the hypothesis under which PTAPS's strong guarantees hold.

Extracting information from a single path We begin by showing that an attacker cannot extract information about direct links between nodes by examining a single path.

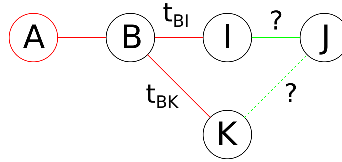


Figure 4.2: If $t_{\text{BK}} \geq t_{\text{BI}} \cdot t_{\text{IJ}}$, A can't guess if the path goes through I or K.

Lemma 1 In the routing information provided during path exchanges, if a node's information appears in clear form, then the previous and next nodes (if any) have their routing information hidden.

Proof. The proof relies on the following observations:

- EXPLICIT-EXPLICIT exchanges, which constitute the bootstrap process, ensure that the routing information associated with paths of length two hides the intermediary node;
- TAPS-EXPLICIT exchanges ensure that the identities of the second and second-to-last nodes in a path remain hidden after the exchange;
- TAPS-TAPS exchanges, which comprise path-reversal and concatenation operations, also preserve the property of the lemma because the endpoints of the paths being concatenated are always represented by public blocks.

As a result, for every path, the identity of at least one node every two remains hidden during all exchanges, thereby proving the lemma.

$\square_{Lemma\ 1}$

Corollary 1 An attacker cannot guess the existence of a direct link in the explicit social network from the routing information contained in a path.

Remark: The identities encrypted by the attacker itself in paths are those of its friends. Consequently, even if the attacker can read this information, it can only discover links that join one of its friends to another node, but these links are already disclosed during EXPLICIT-EXPLICIT exchanges.

Extracting information from multiple paths Next, we present a set of results that show that an attacker cannot extract information about direct links by trying to identify common path sub-sequences in two paths. These results cover cases in which an attacker gathers a finite number of paths and combines them in pairs by trying to identify that one path is a subpath of the other. However, they do not cover situations in which an attacker can gather some form of exhaustive knowledge about the paths in the network: for example by being sure to have collected all the two-hop paths starting from a node it wishes to attack.

It is easy to see that the only combination of two paths that might lead to the identification of a direct link consists of a path, p_1, and a path p_2 corresponding to p_1 with the addition of a final node J. Lemmas 2 and 3 prove that even this combination cannot leak friendship information. Their proofs, as well as those of the theorem and corollary that follow, rely on a few standard assumptions on the cryptographic algorithms used to encrypt routing blocks.

- It is impossible to tell which key was used to encrypt a block without having the appropriate key.
- It is impossible to tell if two blocks were encrypted with two different keys without being able to decrypt at least one of them.
- It is impossible to tell whether two blocks encrypted with different keys contain the same information.

Lemma 2 Let A be an attacker that has the knowledge of two paths $p_1 = ABI$ and $p_2 = ABIJ$ along with their associated routing information. If neither I nor J is a friend of A, and if B has at least another friend K such that: (i) K is different from A, I, and J, (ii) K is not a friend of A, and (iii) $t_{BK} \geq t_{BI} \cdot t_{IJ}$, then A cannot guess the existence of link $I - J$.

Proof. Because of the behavior of EXPLICIT-EXPLICIT exchanges, the routing information of path p_1 is b'$_{\mathrm{a}}$a · ib'$_{\mathrm{i}}$, while that for path p_2 is b'$_{\mathrm{a}}$a · i$_{\mathrm{b}}$b$_{\mathrm{i}}$ · jb'$_{\mathrm{j}}$. For A , it is therefore impossible to detect that ib'$_{\mathrm{i}}$ and i$_{\mathrm{b}}$b$_{\mathrm{i}}$ refer to the same link. First A does not have B 's key and is thus unable to match block i$_{\mathrm{b}}$ with block i. Second, she does not have I 's key either, and thus the routing blocks b$_{\mathrm{i}}$ and b$'_{\mathrm{i}}$ also look different to her. This means that because of the hypothesis that B has another friend $\mathrm{K} \neq \mathrm{A, I, J}$, A cannot distinguish the routing information for p_2 from that associated with a path $p'_2 = \mathrm{ABKJ}$. Moreover, because A is not a friend of K, she cannot exclude the existence of link $\mathrm{K} - \mathrm{J}$. As a result, the only way for A to establish that p_2 is not p'_2 would be to rely on trust values, and realize that the trust value associated with the link $\mathrm{B} - \mathrm{K}$ is too low for p'_2 to have the same trust value as p_2. However, this would require the trust value of $\mathrm{B} - \mathrm{K}$ to be $t_{BK} < t_{BI} \cdot t_{IJ}$, which contradicts the hypothesis. This proves that A cannot identify if p_2 goes through I or through K, and thus cannot guess the existence of the link between I and J .

$\square_{Lemma\ 2}$

Lemma 3 Let A be an attacker that knows two paths $p_1 = B_1 \ldots B_n I, n \geq 2$ and $p_2 = B_1 \ldots B_n IJ$. If $A \notin \{B_1, \ldots, B_n, I, J\}$ and if neither B_n, I nor J is a friend of A, then A cannot guess the existence of link $I - J$.

Proof. The routing information of p_1 ends with the blocks ib'$_{ni}$, while that of p_2 ends either with the blocks $\mathrm{i}_{\mathrm{b}_n}\mathrm{b}_{n\mathrm{i}} \cdot$ ji'$_{\mathrm{j}}$, or with the blocks i'$_{\mathrm{b}_n}\mathrm{b}_n \cdot$ ji'$_{\mathrm{j}}$. We now prove that the attacker is unable to match the two paths because the corresponding blocks in their final links differ. First, i differs from its encrypted counterpart $\mathrm{i}_{\mathrm{b}_n}$. Second, b'$_{ni}$ differs both from its its unencrypted counterpart b_n and from b_{ni}. The encrypted block for B_n appearing in public links, b'$_{ni}$, is, in fact, different from the one appearing in secret links, b_{ni}, as described in Section 2.1.2.

In addition A is neither a friend of B_n nor a friend of J, thus it has no way to exclude that path p_2 contains ends, for example, by the sequence $\mathrm{B}_n - \mathrm{K} - \mathrm{J}$. This, proves that she cannot determine whether p_2 contains a link between I and J .

$\square_{Lemma\ 3}$

Remark: The lemma trivially holds with the same arguments if $p_1 = \mathrm{IB}_n \ldots \mathrm{B}_1$ and $p_2 = \mathrm{JIB}_n \ldots \mathrm{B}_1$.
Lemmas 2 and 3 state the hypotheses required to prevent attacks from a specific attacker. In the following we provide sufficient conditions that are valid regardless of the attacker.

Theorem 1 Let I and J be two explicit friends such that: (i) neither I nor J is friends with a malicious node, (ii) each friend of I has at least two other friends $\neq J$ which are not friends with each other and that it trusts at least as much as I, and (iii) each friend of J has at least two other friends $\neq I$ which are not friends with each other and that it trusts at least as much as J. Then, no attacker, regardless of its position in the network, can discover with certainty that a given path contains an explicit link between I and J through any combination of two paths.

Proof. From the hypotheses, it immediately follows that, if the attacker and I (resp. J) have a common friend, then this has at least one friend $\mathrm{K} \neq \mathrm{A}, \mathrm{I}, \mathrm{J}$ which it trusts at least as much as I (resp. J) and which is not friends with A. Thus the hypotheses of Lemmas 2 and 3 are verified for any attacker A.

$\square_{Theorem\ 1}$

Corollary 2 If every node in the network gives its maximum trust value to at least three of its friends which are not friends with each other, then given any two explicit friends I and J that are not friends with malicious nodes, no attacker can discover with certainty that a given path contains an explicit link between I and J.

Proof. It is easy to observe that the fact that each node has at least three friends which are not friends with each other and which have the same highest trust values makes it possible to find, for each friend of I's (resp. J's), one friend which is (i) at least as trusted as I (resp. J), (ii) not equal to J (resp. I) or A , and (iii) not friends with A. This verifies the hypotheses of Theorem 1. $\square_{Corollary\ 2}$

The above results precisely describe the circumstances that prevent an attacker from discovering the existence of friendship relationships. Specifically, Lemma 1 and Corollary 1 guarantee that an attacker cannot extract link or trust information from a single path. Lemma 2, Lemma 3, Theorem 1, and Corollary 2 extend this guarantee to an attacker trying to extract information from the combination of two paths.

3 Evaluation

3.1 Setting

We evaluated both TAPS and PTAPS on real data traces consisting of 3000 users extracted from the Facebook and Digg social websites. The Facebook trace[1] contains friendship links and a list of social interactions. To obtain a treatable subset for our experiments, we first cleaned up the trace by removing all the users that had only one friend, as they would be too isolated to benefit from our social platform. We then selected the user with the largest number of interactions and proceeded in a spiral fashion by selecting her friends, then the friends of her friends, and so on, until we reached 3000 users. We associated each of these users with a random user profile from the Digg social network. We obtained these profiles by crawling Digg in late 2010.

Friendship links in the Facebook trace and profiles in the Digg trace provided the base of explicit links and profiles for our experiments. On top of them, we built several traces by varying the trust patterns between the nodes. We distinguish our traces into two groups: binary traces in which users either fully trust or fully distrust each other, and multi-valued traces in which the trust is a real value in [0:1]. In both case, we make the assumption that the more two users interact, the more they trust each other (this assumption is done for evaluation purpose, but is not part of the protocol itself).

Our choice of a random mapping between users from Facebook and profiles from Digg is due to the absence, to the best of our knowledge, of a real trace combining profiles and trust information. The random mapping may in fact underestimate the performances of TAPS. Indeed, TAPS is particularly good at discovering trusted paths toward close nodes in the explicit social network and, as shown in [MSLC01], friends in a social network tend to be more similar to each other than to random nodes.

Binary traces In binary traces we assigned a binary trust value to each link in the data set. Specifically, we sorted the friends of each user according to the number of interactions she had with them. Then, for a user with $|N|$ friends, we assigned a trust value of 1 to the $\beta|N|$ friends with the largest number of interactions. As this process creates asymmetric trust values, we then set the symmetric trust value of each link as the logical OR of the two asymmetric values. This leads to the proportions of trusted links depicted in Table 4.3a.

Multivalued traces While binary traces provide a simple experimental setting, reality tends to be more complex. Thus, we also considered traces with trust values of 1, 0.8, 0.5, and 0. Similar to the binary case, we sorted each user's friends by the number of interactions and assigned a trust value of 1 to the top $\gamma_1|N|$, 0.8 to the following $\gamma_{0.8}|N|$ and so on. As this process creates asymmetric trust values, we then set the symmetric trust value of each link as the maximum of the two asymmetric values. This leads to the distribution of trust values depicted in Table 4.3b.

[1] Network A at `http://current.cs.ucsb.edu/facebook/index.html`.

Name	β	% 1	% 0
Binary-0.4	0.4	53.5	46.5
Binary-0.6	0.6	71.3	28.7
Binary-0.8	0.8	89.3	10.7

(a) Binary Traces

Name	% 1	% 0.8	% 0.5	% 0
MultiValued-1	41.3	23.8	23.4	11.5
MultiValued-2	57.4	27.9	13.3	1.4
MultiValued-3	76.2	12.3	10.1	1.4

(b) Multi-valued Traces

Figure 4.3: Trust values distribution for different traces.

3.2 Terms of Comparison

We compared the performance of our protocols with three alternatives. Best is an ideal system that, powered with global knowledge, always provides each user with the set of neighbors that offer the best combination of similarity and trust. This allows us to assess the ability of our protocol to reach similar results in a decentralized way. Similar consists of a standard similarity-based implicit network [BFG+10], augmented with an oracle providing the best trusted paths to neighbors. This maximizes profile similarity at the cost of possibly lower trust. Finally, Trusted selects the most trustworthy nodes, providing very high trust, but possibly poor similarity. In all experiments, we compute the average scores of the TAC VIEWS over all nodes using TAC, as well as Similar and Trusted. Then, we normalize each of these averages by dividing it by the average score over all nodes obtained by Best, which provides optimal views. Unless otherwise specified, we used default values for τ and ϵ: $\tau = 0.75$ balances trust decay and path length (0.56 at 3 hops, 0.23 at 6), and $\epsilon = 1$ gives a fair trade-off between trust and similarity.

3.3 Results

Impact of trust density We start our performance comparison by examining the results obtained in the various traces to highlight the effect of the number of trusted links. Figure 4.4 shows the average score values in the TAC VIEWS with TAPS and PTAPS as well as with their competitors as percentages of the scores of Best. TAPS performances are always better than those obtained by Similar and Trusted with the use of global knowledge. As expected, they particularly outperform Similar whenever the social network has a limited proportion of high-trust links (Binary0.4 and Multi-valued1). This can be explained by observing that Similar always selects the nodes with the best similarity and is therefore penalized in networks with lower trust density. On the other hand, Trusted performs best with a low trust density, but its performance remains bad in any configuration. We also observe that TAPS and PTAPS achieve a stable behavior across all traces with respect to Best, contrary to Similar and Trusted.

Impact of trust transitivity Figure 4.5 confirms the above observations by examining the impact of trust transitivity, τ, in the Binary-0.8 and MultiValued-1 traces. The performance of both our protocols is particularly good when trust decays faster as this gives more importance to nodes that are closer in the social network even if they may be

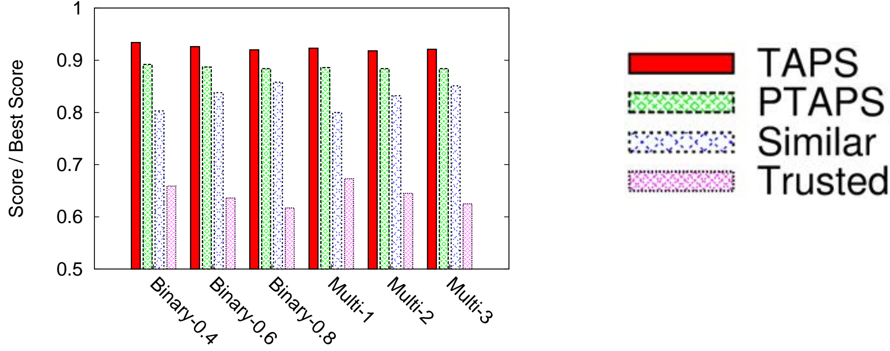
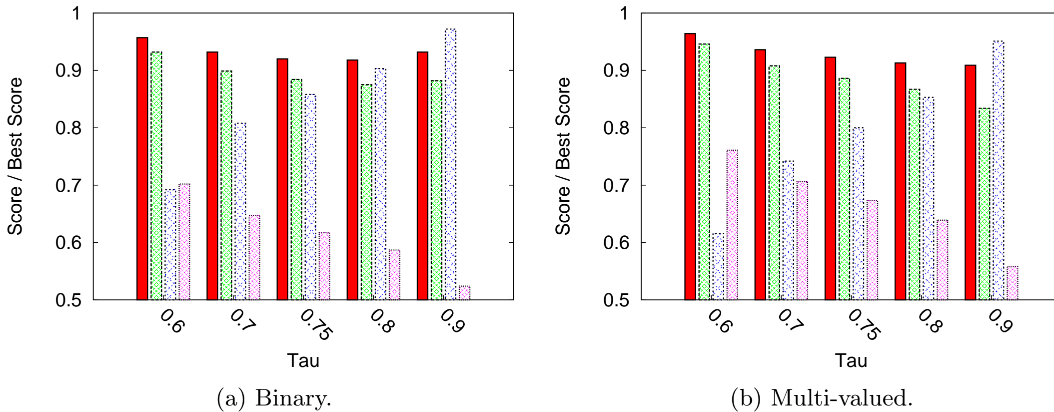


Figure 4.4: Impact of trust density.

less similar. This suggests that our TAPS protocol is particularly suited for important transactions and situations in which people can tolerate only limited amounts of risk.

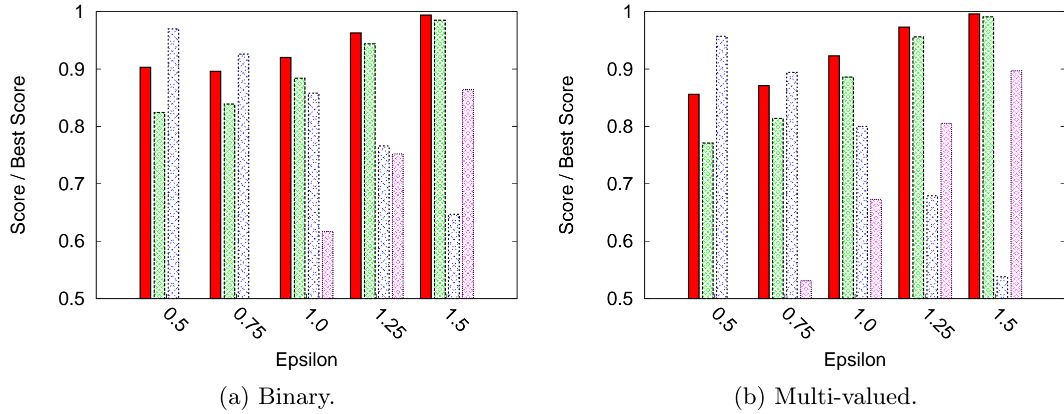
With very high values of τ, trust decays more slowly. In this case, a protocol like Similar that selects the most similar nodes before searching for a trusted path may be viable. However, Similar achieves this through global knowledge. A distributed protocol to compute trusted paths would probably be either very costly or ultimately equivalent to TAPS in networks characterized by high trust densities. Moreover, such a protocol would remain weak in situations where the density of trusted links is low, as shown in Figure 4.4.

Figure 4.5 also shows that the difference of performances between TAPS and PTAPS is greater when τ is high. High τ values make it possible for the protocols to select nodes that are further away in the social network, which penalizes PTAPS as its privacy preserving mechanisms (the restricted view exchange and the impossibility to perform shortcuts on paths) make it less efficient at discovering long paths.



(a) Binary. (b) Multi-valued.

Figure 4.5: Impact of τ in the Binary0.8 (left) and Multi-valued-1 (right) traces.



(a) Binary.

(b) Multi-valued.

Figure 4.6: Impact of ϵ in the Binary0.8 (left) and Multi-valued1 (right) traces.

Impact of trust weight Next, we evaluate the impact of the trust-weight factor. Figure 4.6 shows the results in the Binary-0.8 (left) and Multi-valued-1 (right) traces respectively. Both plots show that the benefits of a protocol like TAPS become more important as we place more weight on trust. With values of 1 or above, TAPS perform better than Similar and reach almost optimal results with $\epsilon = 1.5$, while Trusted is only able to achieve acceptable results with a very high value of ϵ. Similar to Figure 4.5, in Figure 4.6 we observe that the difference between TAPS and PTAPS increases when the optimal nodes become further away.

Impact of the biased view and the trust threshold We now concentrate on the components of our protocols and analyze the impact of the biased view selection and of the trust threshold on TAPS and PTAPS on the MultiValued-2 trace. Figure 4.7 highlights that biasing view selection significantly improves the performance of both TAPS and PTAPS by directing the exploration of the network toward highly trusted links. The trust threshold complements this by obtaining an additional increment in performance. The presence of the trust threshold becomes even more critical if we disable the biased-view-selection feature. The threshold alone, even if not as effective as when the view selection is biased, prevents non-trustworthy nodes from entering a node's view.

Graph properties of TAPS We conclude our evaluation by examining the graph properties of our TAPS overlays. We do not shot PTAPS because its results are almost indistinguishable from those of PTAPS. Figure 4.8a shows the cumulative distributions of the local clustering coefficients of the nodes in the TAPS VIEW and TAC VIEW for TAPS and compares them with those of a standard peer sampling protocol coupled with a clustering protocol (RPS and RPS Cluster, which is equivalent to Similar). The figure shows that TAPS is clearly more clustered than RPS. This is a clear effect of TAPS biased view selection which tends to favor nodes that are closer in the network. The figure

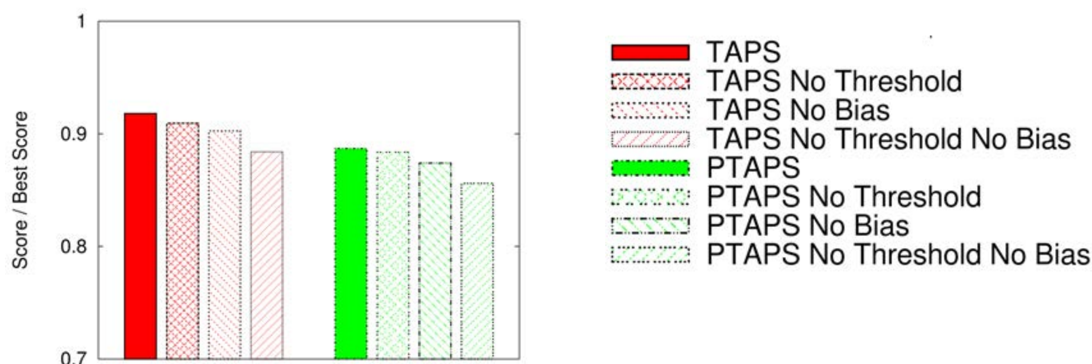
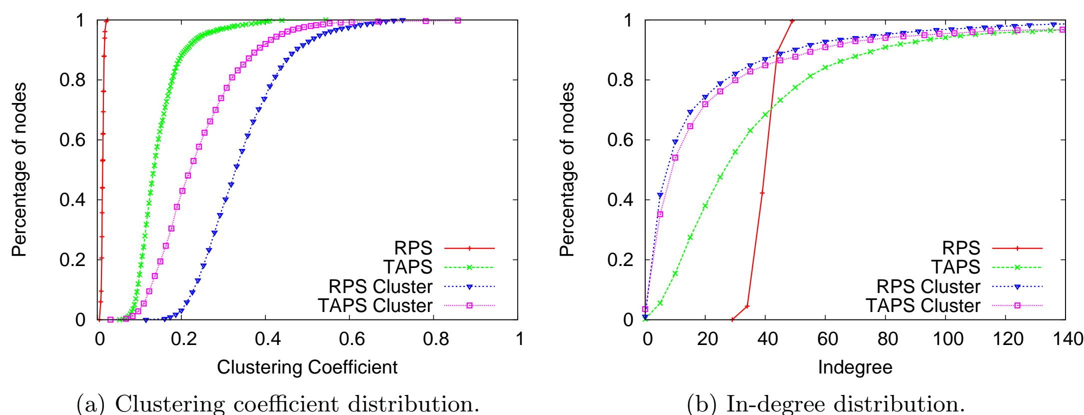


Figure 4.7: Impact of view bias and trust threshold.



(a) Clustering coefficient distribution.

(b) In-degree distribution.

Figure 4.8: Cumulative distributions of the local clustering coefficients (left) and of the in-degree (right) distributions for TAPS-based and standard protocols.

also shows that the TAC VIEWS generated by both TAPS are less dense than those of Similar as very similar nodes that form tight clusters in Similar may not be connected by highly trusted links.

The in-degree distribution shown in Figure 4.8b also shows some differences with respect to traditional protocols. The in-degree distribution of the TAPS VIEW is in fact slightly skewed because nodes that are not trusted by many others tend to have fewer neighbors. This is indeed desirable as untrusted nodes could potentially harm the system through illicit behaviors. Finally, we observe that the in-degree distribution for the TAC VIEWS does not show any significant variation between TAPS and Similar.

4 Related Work

The concept of trust in explicit social networks has been exploited in domains ranging from peer-to-peer security to recommendation systems. SybilGuard [YKGF08] and SybilLimit [YGKX08] propose protocols that exploit trust relationships between friends to protect peer-to-peer systems from sybil attacks. Reliable Email [GKF+06] uses a similar approach to build an email-whitelisting system based on friend-to-friend relationships, while Ostra [MPDG08] exploits social trust to limit the incidence of unwanted communication in messaging and content-sharing systems.

NABT [LHL+10] proposes the use of trust between friends to prevent freeriding behaviors using a more efficient form of tit-for-tat based on indirect trust relationships. NABT's credit-based approach can be viewed as a basic form of trust inference between friends of friends. A more advanced approach is adopted by SUNNY [KG07], a centralized protocol that takes into account both trust and confidence to build a Bayesian network. Even if SUNNY is centralized, its confidence-based idea could lead to interesting improvements for TAPS.

A number of research efforts have instead investigated the use of trust links to improve the performance of recommendation systems. The work in [WV05] uses an approach similar to that of [KG07], while TrustWalker [JE09] combines trust and item-based collaborative filtering. TaRS [MA07] builds a recommendation system capable of operating both with global and with local trust metrics. Global trust metrics [BP98] predict a global reputation value for each node. Local trust metrics [MA05], on the other hand, take an approach similar to ours and compute trust values that are dependent on the target user.

Finally, our path-encryption mechanism shares similar aspects with onion-routing approaches [RSG98]. In our protocol however, the endpoints of a path need to be able to reverse paths, and, most importantly, they must not know the identities of the nodes in a path, making the use of [RSG98] impossible.

5 Concluding remarks

We presented PTAPS, an privacy preserving alternative to the TAPS protocol. While TAPS focuses on finding nodes with the best trust and similarity values, PTAPS also provides strong privacy guarantees by hiding the existence of explicit links from nodes that are at more than two hops from them.

Our results encourage us to extend the TAPS family in a number of ways. First, we are currently evaluating whether the protection offered by PTAPS can be extended to combinations of more than two paths. Second, we are working on extensions capable of addressing situations in which attackers may inject false information with the aim of polluting other nodes' views and make the system unusable. A third direction we are considering is to extend our protocol family to networks characterized by asymmetric trust values as opposed to symmetric ones. This would make it possible to render trust information even more private as users would not need to disclose to their friends the trust they have for them. Both with symmetric and asymmetric trust values, it would

also be interesting to explore the use of multiple redundant trusted paths instead of a single one. From a systems perspective, we instead plan to evaluate and possibly improve the performance of our protocols in the presence of churn, for example by having nodes rely on trusted peers to disseminate their information while disconnected. Finally, we aim to integrate TAPS in our existing prototype applications within the Gossple project as a way to reinforce the trust of users in the recommended content.

Chapter 5

Anonymity for gossip-based applications

1 Introduction

Anonymity services provide an attractive way to overcome the privacy issues associated with personalized services. They allow users to communicate with other users or services without disclosing their identity. In its simplest form, an anonymity service is a node acting as a proxy between a user (the source) and whoever she wants to communicate with. While the proxy effectively hides the source's identity from the user she communicates with, it knows the identity of both users, which means that the users need to trust the proxy. More advanced anonymity services, such as AP3 [MOP+04], Tarzan [FM02] or Cashmere [ZZZR05] do not rely on a single node to act as a proxy, but instead use a sequence of nodes to relay the message from the source to the destination. Because each node in the sequence only knows its predecessor and its successor, no node knows both the sender and the recipient of the message.

Some of these services also provide anonymous pseudonyms, for example AP3 and Tarzan. A pseudonym is a virtual identity that a user can take in order to hide her true identity. This is typically implemented by having the node who wants a pseudonym to select another node as a proxy. The proxy forwards, through an anonymous channel, the message it receives to the node. That way, the node can advertise its presence in the system and be contacted by other nodes without having to reveal its true identity.

This is a very important feature in decentralized systems such as Gossple, as users a required to communicate with each-other (to the contrary of a centralized system, where the service provider does not need to hide its identity and simple anonymity for the user is enough).

In this chapter we present FreeRec, a system that anonymizes communications in gossip-based systems. We present FreeRec in the context of a distributed user-based collaborative filtering although it can be used for other purposes. We assume a system such as Gossple as presented in Chapter 2. Using FreeRec, we are able to build an anonymous interest-based topology and use it for recommendation purposes. By

abuse of language, we present Freerec not only as an anonymous protocol but as the decentralized recommendation platform.

Unlike existing decentralized personalization platforms, FreeRec protects the users' privacy by hiding the link between the users' identity and their interest profiles. This makes FreeRec a generic personalization architecture that can be leveraged to build a number of distributed applications that may benefit from personalization services.

FreeRec builds anonymous chains of nodes by relying on three layers of gossip protocols providing mutual anonymity. A Byzantine-resilient peer-sampling protocol provides nodes with the members of their anonymous chains. A second private peer-sampling protocol uses these chains to provide each node with an anonymous sample of the network. A top clustering layer implements a decentralized collaborative-filtering overlay by creating decentralized clusters of anonymous profiles. This layered architecture makes FreeRec self organizing and capable to adapt to the arrival and departure of nodes and to changes in the interests of users. We evaluate FreeRec using simulations, a real deployment on PlanetLab, and a probabilistic analysis. Our results on a news-personalization use case show that users are able to effectively receive and publish content even in presence of path failures with reasonable overhead.

2 System models

2.1 Recommendation system

We consider a user-based CF model as Gossple, which build an interest-based overlay networks by clustering nodes according to the similarity among their interest profiles. This task relies on two protocols: a random peer sampling protocol (rps) and a clustering protocol (clustering). The rps [VGVS05] protocol ensures connectivity by providing each node with a continuously changing random sample of the graph. While the rps allows nodes to continuously discover new nodes, the clustering protocol identifies, at each node, the k-nearest neighbors in term of interests, and ensures connectivity between the node and this neighborhood.

Periodically, each protocol selects the node in its view with the oldest timestamp and sends it a message containing its profile with half of its view for the rps and its entire view in case of the clustering protocol (standard parameters [JVG+07, VVS05]). In the rps, the receiving node renews its view by keeping a random sample of the union of its own view and the received one. In the clustering protocol, instead, it computes the union of its own and the received view, and selects the nodes whose profiles are closest to its own according to a similarity metric. As in the previous chapters, we use the well known cosine similarity metric.

2.2 Adversary

We consider adversaries following the Honest-But-Curious model [Gol03] where malicious nodes can collude to extract information from the system but do not deviate from the protocol (i.e. they do no forge, modify, replay or drop messages). As a conse-

quence, adversaries can monitor all exchanges where they are involved. The goal of the adversaries is to break the anonymity of a user by finding its profile.

3 FreeRec

Our anonymous personalization architecture extends the model described in Section 2 to achieve anonymity by executing gossip exchanges through onion-like encryption chains: the proxy chains. The proxy chain of a node n is a sequence starting with n and containing l other nodes as depicted in Figure 5.1. We refer to node n, the first node in the chain, as its initiator. The last, p, is the chain's proxy, (or n's proxy), while the remaining ones are intermediary nodes. Messages can travel along the chain in two directions: forward, from the initiator to the proxy; or backward, from the proxy to the initiator. The proxy acts as a placeholder for n, hiding n's identity in all the gossip exchanges that include n's interest profile.

Proxy chains effectively hide the very fact that two nodes are communicating. Two nodes n and m can communicate and exchange each-other's profiles without knowing their respective identities. Moreover, their profiles are hidden from all other nodes in the chains. A node n that wishes to send a message to another node encrypts it once for each node in its proxy chain along with routing information telling the node who is the next node on the path. Each of the nodes along its proxy chain removes one of these layers and sends the inner encrypted layer to the next hop indicated in the message. The process continues until the destination node's proxy. At this point, the message goes through the destination chain in the backward direction using routing information and encryption keys maintained by each node in the chain. Each of the nodes in the target's chain, starting from the proxy, adds one encryption layer and routes the message until it arrives at the destination node, which removes all the layers.

The size of the proxy chain can either be fixed, such as in Tor [DMS04], or be selected randomly in a range $[m : M]$ when the chain is created, like in [ZZZR05] and [ZH04]. We will see in the evaluation that, in our context, chains of fixed size l are preferable as they significantly reduce the probability for an attacker to successfully disclose one's identity when compared to a variable-length chain with the same average length: $l = \frac{\mathsf{m+M}}{2}$.

3.1 Chain-Based Routing

We now present the data structures that allow nodes to build, maintain, and use proxy chains.

3.1.1 Chain and Message Keys

The onion-like encryption process outlined above relies on three types of keys: two sets of public/private key pairs, and one set of secret keys. First, each node, n, maintains a key pair, (K_n, k_n)[1], called message key pair. Nodes use it to send and receive encrypted

[1] We use uppercase characters for public keys and lowercase for private or secret keys.

messages through proxy chains to and from any other node while preventing the proxies and the other chain nodes from accessing the content of this communication.

Each node also maintains a second key pair: the chain key pair, (C_n, c_n). While the message key pair hides the content of a message from the nodes in the chain, the chain key pair makes it possible to construct the onion-like encryption layers when traversing the chain in the forward direction.

FreeRec requires the two distinct public/private key pairs described above to ensure anonymity. Nodes advertise their public chain key as their own, thus chain keys alone would not provide anonymity. The use of two pairs instead guarantees that an adversary cannot associate a public message key with the corresponding public chain key, and thus with the identity of the node.

Finally, each node, n, generates and dispatches a secret key, s_i^n, to each node, i, in its own proxy chain. Nodes use this key to add onion layers to messages that travel along the chain in the backward direction, i.e. towards n. The use of onion-like encryption in the forward and backward directions causes messages to change at each hop, thus preventing external observers from recognizing the messages in a proxy chain. We summarize the roles of the three types of keys in Table 5.1, and provide details about their distribution in Section 3.3.

3.1.2 Data Structures and Routing IDs.

To route messages along proxy chains we use a combination of source and hop-by-hop routing. Each node maintains information about the members of its own proxy chain in a chainTable. This data structure contains, for each node in the chain, its identifier, its public chain key and its secret key. The information in the table allows the initiator of a chain to encrypt messages in onion layers and decrypt incoming messages.

The destination proxy, however, cannot use source routing to reach the destination node: a node may in fact act as a proxy or an intermediate node in multiple proxy chains. To route backwards along the chain, we therefore use a set of routingIds as depicted in Figure 5.2. For routing purposes, all the nodes in a chain could use the same routingId to identify their next hops. However, this would easily allow colluding nodes to verify if they are part of the same chain. We therefore associate a unique (with high probability) routingId with each link in a chain. The proxy routingId (e.g. p_a in Figure 5.1 and p_a and p_b in Figures 5.1 and 5.2) serves as a pseudonym for the destination node, while the remaining ones (r_{ij} in the figures) enable backward routing on the destination chain.

Nodes store the routingIds of the chains they belong to in a routingTable. With reference to Figure 5.2, let node p be a proxy in the chain of node b. p's routingTable contains an entry indexed by b's proxy routingId (p_b in the figure). This entry contains (1) p's secret key for the chain (s_p^b), (2) the identifier of the previous node in the chain (v), (3) the public chain key of v (C_v), and (4) the routingId of the link between p and v (r_{pv}). Intermediate chain nodes also have an analogous entry in their routing tables, but indexed by the routingId of the link to the next node in the chain. Node v therefore has an entry indexed by r_{pv} and containing (1) v's secret key for the chain (s_v^b), (2) the

identifier of z, (3) z's public chain key (C_z), and (4) the routingId of the link to z (r_{zv}).

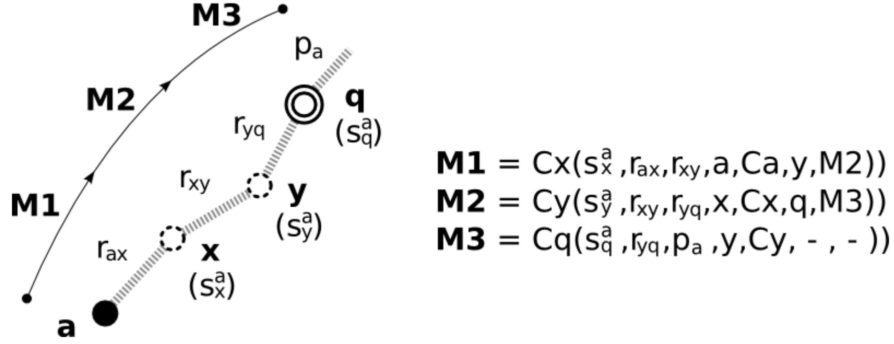


Figure 5.1: *Proxy chain creation.*

3.2 FreeRec Three-Layer Architecture

Our goal in building proxy chains is to enable the architecture described in Section 2 to operate anonymously. To make this possible, we replace the two protocols of Section 2 with a three-layer architecture. We introduce a rps protocol layer, which provides each node with a sample of the network from which to choose the members of its proxy chain. The rps operates like a normal peer sampling protocol with one addition: it associates each node n with the information required for creating the chains. This comprises only the node's IP address, its public chain key C_n, and a timestamp. Interest profiles do not appear in the rps views: they are protected by the anonymous prps layer.

The prps (Proxied Random Peer Sampling) uses the information provided by the rps to build a proxy chain for each node. It then exploits these chains in gossip exchanges thereby providing each node with a random sample of anonymous nodes. In doing this, it also allows nodes to learn about the necessary information to route messages anonymously to other nodes. Consider a prps view containing an entry for node b. The entry does not include b's IP address and port. Rather, it is identified by b's proxy routingId (p_b). In addition, it contains the IP address and port of b's proxy (p), p's public chain key (C_p), b's public message key (K_b), and b's interest profile.

prps views allow nodes to learn about the anonymous information referring to another node without being able to associate it with the node's precise identity. Nodes exchange views like in a standard rps. However, they channel all view exchanges through their proxy chains. The prps thus replaces the rps protocol of the architecture of Section 2, thus enabling anonymous profile exchanges.

The prps serves as a basis for the top layer of our architecture: a clustering protocol. Like the prps, the clustering protocol performs all its view exchanges using the proxy chains built by the prps layer. This allows our architecture to build decentralized personalized services in a completely anonymous manner.

(C_n, c_n)	Chain key pair of node n	c_n private key, C_n public key
(K_n, k_n)	Message key pair of node n	k_n private key, K_n public key
s_n^x	Secret key generated by node x and shared with node n	
RPS	Random peers sampling	@ip, timestamp, C_n
PRPS	Random proxies sampling	@ip, timestamp, ROUTINGID, C_n, profile, K_n
CLUSTERING	Interest-based neighborhoods	@ip, timestamp, ROUTINGID, C_n, profile, K_n
RT	ROUTINGTABLE $[r_{pv}]$	s_v^b, z, C_z, r_{zv}

Table 5.1: *Data structures maintained on a node v, followed by p and preceded by z in b's chain.*

3.3 Protocol Details

In the remainder of this section, we provide additional details about how the prps protocol manages chains and encrypted routing.

3.3.1 Building Proxy Chains

A node a can start building its proxy chain once its rps view is filled with a random set of nodes. Specifically, a first determines how many other nodes should be in its chain by extracting a random number k within $[\mathsf{m}, \mathsf{M}]$. Then it extracts k nodes from its rps view and it sets the first extracted node as a proxy p and the remaining ones (if any) as intermediate nodes i in the order they were extracted. a builds a create-chain message as described on Figure 5.1.

The message consists of concentric onion layers. Each layer is a createChain message encrypted with the chain key of one of the nodes that will constitute the chain. The innermost message, $M3$ in the figure, is encrypted with the proxy's chain key and contains (1) the proxy's secret key (s_q^a), (2) the proxy routingId for the chain (p_a in the figure), and (3) the routingId for the link between the proxy and the last intermediate node (r_{yq} in the figure), (4) the previous node's IP address and port (y), and (5) its public chain key (C_y).

After creating $M3$, the initiator creates a message for the last intermediate node in the chain (y in the figure). This message contains (1) the previously encrypted message for the proxy ($M3$), (2) the next node's IP address (the IP address of the proxy q in this case) and port, (3) node y's secret key (s_y^a), (4) the routingId of the link between y and the next node in the chain (r_{yq}), (5) the routingId of the link between y and the previous node in the chain (r_{xy}), (6) the previous node's IP address and port, and (7) its public chain key (C_x). The initiator encrypts $M2$ with y's chain key and then it repeats the process by adding a layer for each of the nodes it selected for its chain, the last of these being the one closest to the initiator itself, x in the figure. The initiator then sends the outermost message to this node initiating the chain-creation process.

Each node receiving a createChain message decrypts it and uses its content to update the information in its routing table. It then forwards the encrypted inner-layer message to the next node in the chain, which operates analogously. The proxy performs the

same operations except that it does not forward the message further. If a chain node is already part of another chain with the same routingId, it replies with an error message to the initiator, which will recreate the chain using a different routingId.

3.3.2 Sending Messages through Chains

FreeRec achieves mutual anonymity: when two nodes exchange messages, both the sender and the receiver are anonymous. Nodes use their proxy chains to send and receive encrypted messages as part of the prps and clustering protocols. Consider the example in Figure 5.2. Node a is sending a message m to a node with proxy p (not knowing b's id), public message key K_b, and proxy routingId p_b. Node a will have discovered this node, which happens to be b, through prps or clustering exchanges. As a result, the association between b's identity and p, K_b or p_b is unknown to a as well as to every other node in the system. The process unfolds as follows.

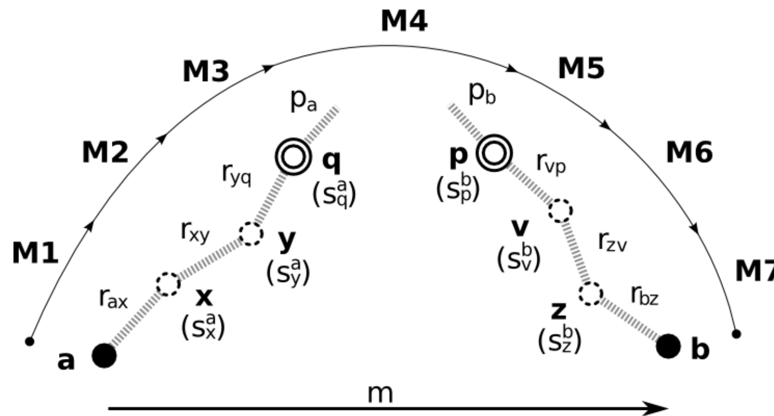


Figure 5.2: *Message exchange between nodes a and b: a knows b's profile, the identity of p, but not the identity of b. Node b knows a's profile, the identity of q, but not the identity of a. Nodes in the chain cannot access a's or b's profile.*

First, a encrypts m using the destination node's public message key yielding $K_b(m)$. Then it prepares the first layer of its onion message. Specifically, a includes $K_b(m)$, and the destination's proxy incoming routingId, p_b in the figure. Then it encrypts the resulting message with the destination proxy's public chain key, yielding $M4$ in the figure. Node a continues the creation of the onion message by adding one layer from each of the nodes in its own chain, starting from the proxy. The first of these layers, $M3$ is encrypted with the proxy's public chain key, and contains both $M4$ and the address of $M4$'s target: the destination node's proxy. The subsequent one, $M2$ contains $M3$ and the address of $M3$'s target, q in the figure. In general, consider a node n that is followed by a node m in a proxy chain. The corresponding onion layer will be encrypted with n's public chain key and will contain the IP address and port of m together with the immediate inner onion layer encrypted with m's public chain key. In the case of the figure, the outermost onion layer ($M1$) will be encrypted with node x's public chain key and will contain $M2$ and the IP address and port of node y.

After creating $M1$, node a sends it to x, which starts peeling off the first layer. It first decrypts the message using its private chain key and then forwards the contained encrypted message ($M2$) to the node indicated in $M1$ (y). Upon receiving the corresponding onion layer, each node in the chain proceeds analogously until the source node's proxy (q) forwards the innermost layer ($M4$) to the destination's proxy (p). This completes the first part of the routing process.

The destination's proxy (p) initiates the second part. It decrypts $M4$ and retrieves its content: a routingId, p_b, and an encrypted message for the destination node ($K_b(m)$). p first looks up p_b in its routing table and it retrieves (1) the associated secret key (s_p^b), (2) the address and port of the previous node in the destination chain (v), and (3) the routingId of the link leading to this node (r_{vp}). It then encrypts $K_b(m)$ using the retrieved secret key, yielding ($s_p^b(K_b(m))$). Finally it builds a message containing $s_p^b(K_b(m))$, and the routingId of the link to the previous node in the destination chain (r_{vp}). It encrypts this message using v's public chain key, yielding $M5$, and sends it to v.

When v receives $M5$, it decrypts it using its private chain key and retrieves $s_p^b(K_b(m))$ and the routingId of its link to p (r_{vp}). It looks up this routingId in its routing table and retrieves its own secret key s_v^b, the IP address and port of the previous node in the chain (z), z's public chain key, and the routingId of the link leading to it (r_{zv}). Node v encrypts $s_p^b(K_b(m))$ using s_v^b and places it in a message together with the retrieved routingId. It then further encrypts this message with z's public chain key and sends it to z. This process repeats at each of the intermediate nodes in the chain. Each adds an onion layer by encrypting the content of the message with its secret key and then wraps the result into a message with the routing information for the previous node in the chain.

When the destination node (b) receives the final message, it first decrypts it using its private chain key. Then it starts peeling off each of the onion layer added by the nodes in its proxy chain. To do so, it uses the secret keys it stored in its chainTable, starting with the one associated with the first intermediate node. After decrypting the layer added by its proxy, it obtains $K_b(m)$, which it further decrypts using its own private message key, ultimately retrieving the original message m.

3.3.3 Initialization.

For this process to work, the source of a message must not only have built its proxy chain, but it must also have the necessary information about the destination node. This consists of the destination node's public message key (K_b), and of its proxy's IP address, public chain key (C_p), and routingId (p_b). During normal operation, nodes obtain this information through prps exchanges. However, this poses a problem during initialization when the prps view of a node is still empty.

Consider a node n with an empty prps view. The first time n establishes a proxy chain, it sends a prps view containing only its information to all the nodes in its rps view. The corresponding messages go through the proxy chain of n until its proxy and then go directly to their targets. Consider a target node t receiving one such message.

If t is a proxy for another node m, then it forwards the message to m along m's proxy chain. Otherwise t caches the message until it becomes a proxy for some other node. When a node m receives the initialization message forwarded by its proxy, it adds its content to its prps view. After a short time, m will initiate a view exchange with n who will thus receive entries for its prps view.

In principle, the target node t could also add the information received from n to its own prps view. However, this would weaken the protocol's anonymity guarantees. An attacker n could send its entry to only one target node t. If it subsequently received a message from a proxy p, it could conclude that p is likely to be the proxy of t, and thus match t and its profile.

3.3.4 Changing proxy.

Nodes change their proxy chains periodically. This provides several benefits. It sustains anonymity over time by limiting the impact of a malicious proxy that could, for example, drop all messages that are not coming from another malicious user. Moreover, it allows a node to react to path failures in its chain as a result of churn.

To change proxy chain, a node repeats the chain creation process every $t1$ time units. Once it has established a new anonymous path, it informs all the nodes in its prps and clustering view of its new proxy. To keep track of these changes, all proxies and intermediate nodes associate a timer $t2$ with each of the entries in their routing tables. When $t2$ expires, they delete the corresponding entry. Nodes choose the timer value so that $t2 > t1 + \delta$ where δ is an upper bound on the time required to create a chain.

After a node has set up a new chain, it initiates a prps exchange with all nodes in its prps view and a clustering exchange with all those in its clustering view. A node that receives a fresher prps entry with the same message key as an existing entry (i.e. entry pointing to the same destination node) updates this entry with the new proxy identifier, proxy chain key, and profile.

4 Experimental setup

We evaluated FreeRec in the context of a news-personalization use case. We combine FreeRec with a gossip-based dissemination protocol to recommend news items to a population of users. A user interest profile contains the news items she received and liked. When a user generates an item or expresses a positive opinion on a received item, she forwards it to her neighborhood in FreeRec's anonymous interest-based topology. Gossip frequency in all protocols is set to one per simulation cycle and of one every 2s in PlanetLab.

4.1 Dataset

We use a real dataset: we conducted a survey on around 250 news items (selected randomly from a set of RSS feeds on various topics). We exposed the item list to

around 100 colleagues and relatives and gathered their reactions (like/dislike) to each news item. This provided us with a small but complete dataset of users exposed to exactly the same news items. To scale our system, we generated 5 instances of each user and news item in the experiments. The resulting dataset gathers 1235 news items for 530 users. We inject each item into the system at a random time instant by selecting a random source node.

4.2 Metrics

We evaluate FreeRec along two metrics of performance and quality. Firstly we measure the overhead of the system in terms of the network traffic it generates. For simulations, we compute the total number of sent messages, the number of messages which have not reached their destinations due to message loss and the number of hops for messages. For our PlanetLab deployment, we instead measure the average consumed bandwidth and the latency to receive a message. Secondly, to assess the impact of FreeRec on the quality of the recommendation, we compute *recall* and *precision*. Both measures are in $[0, 1]$. For an item, a recall of 1 means that all interested users have received the item. Yet, this measure does not account for spam since a trivial way to ensure a maximum recall is to send all news items to all users. This is precisely what precision accounts for. A precision of 1 means that the news item has reached only the users that are interested in it. An important challenge in information retrieval is to provide a good trade-off between these two metrics. This is expressed by the F1-Score, defined as the harmonic mean of precision and recall [VR79].

$$Precision = \frac{|\{interested\ \ users\} \cap \{reached\ \ users\}|}{|\{reached\ \ users\}|}$$

$$Recall = \frac{|\{interested\ \ users\} \cap \{reached\ \ users\}|}{|\{interested\ \ users\}|}$$

$$F1 - Score = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

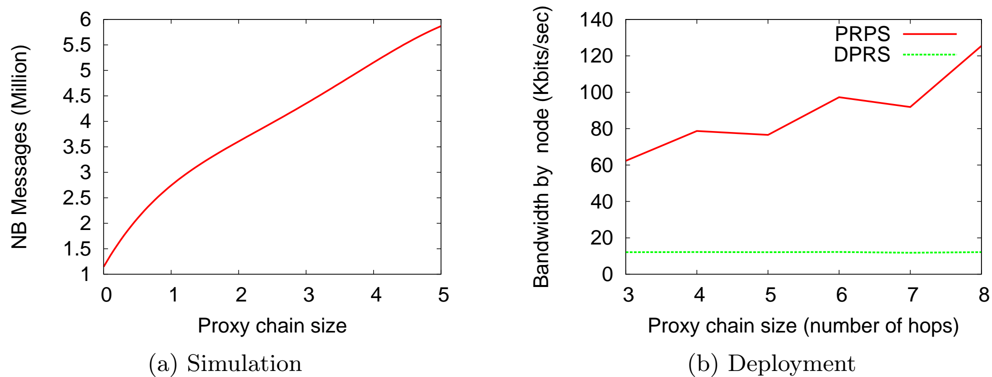
5 Results

We carried out an extensive evaluation of FreeRec by simulation and by deploying its implementation on PlanetLab. We present the results by analyzing the overhead, the impact of proxy changes, the latency, and the probability for a proxy chain to be compromised by a set of colluding nodes.

5.1 Overhead analysis

We start by considering the overhead of the proxy chain in terms of number of messages. Clearly the longer the chain, the more anonymous the system. This cost is a function

of the length of the proxy chain: the more the intermediate nodes in the chain, the higher the cost. Figure 5.3 depicts the number of messages according to the size of the proxy chain with a neighborhood fixed at 25. Results (Fig. 5.3(a)) show that a chain with only one proxy without intermediate nodes (i.e. size=1) brings a three-fold increase in the number of messages. This is because a message needs to go through two proxies (i.e. 3 hops) to reach its destination. Further adding intermediate nodes in the proxy chain proportionally increases the number of hops and the number of messages. Fig. 5.3(b) shows the overhead in PlanetLab of the two protocols RPS and PRPS in terms of bandwidth consumption. We observe that the RPS overhead remains stable regardless of the size of the proxy chains for RPS exchanges carry only information about chain keys while PRPS carries the encrypted messages. For this reason, the cost of PRPS increases linearly with the length of the chain.[2]



(a) Simulation (b) Deployment

Figure 5.3: *Effect of the proxy chain size on the number of messages and bandwidth consumption.*

5.2 The impact of proxy changes

To remain anonymous over time, nodes periodically renew their proxy chains. After setting up a new chain, a node advertises the information about its new proxy through prps and clustering exchanges. However, propagating this information takes time and some nodes only learn about the new proxy after several cycles. During this interval, a node that is unaware of the proxy change will send its messages to the old proxy. Consequently, messages will correctly go through the source node's possibly-new proxy chain, but they will reach the destination node's old proxy chain. If any of the nodes in this chain has already removed the corresponding entries from its routing table, it will silently discard the message.

As explained in Section 3.3.4, nodes remove entries from their routing tables δ time units after the creation of the new chain. During this time, the nodes in the old chain can still forward information backwards towards the chain owner. This leaves some time for the propagation of the new chain's information, but it does not eliminate the

[2] Clustering is not shown for it has a similar behavior as PRPS

possibility of losing messages. Figure 5.4 evaluates the impact of this aspect in the context of our news-dissemination testbed as a function of the size of the clustering view, with $\delta = 10$ cycles.

Figure 5.4a shows that the impact of message loss on the F1-Score is limited. When nodes change proxy every 60 cycles (i.e. t1=60), performance is almost indistinguishable from the stable case where nodes keep the same proxy over the whole experiment. When the chain changes more frequently (smaller values of $t1$) the percentage loss in F1-Score is slightly higher, but it remains lower than 11%.

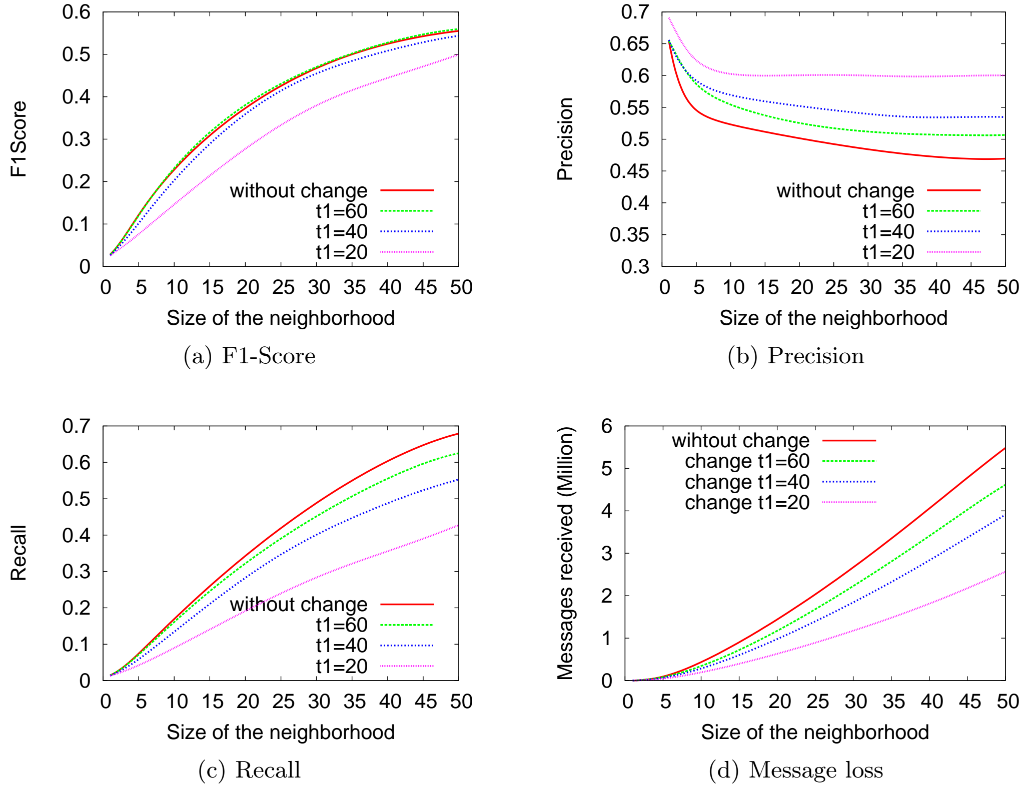
To analyze more precisely this F1-Score reduction, precision and recall are depicted in Figure 5.4b and Figure 5.4c, respectively. Proxy change decreases the recall which means that users can miss interesting news items during this proxy change process. When nodes change proxies every 40 cycles (i.e. three times in the experiment), the recall decreases up to 18% for a size of neighborhood of 50. The augmentation of the precision according to the frequency of proxy change is a side effect of this message loss. System model described in Section 2.1 implies redundancy as shown in [BFG$^+$13]. As users are clustered according to their interests (i.e. high clustering coefficient), an item liked by users in a neighborhood has more chance to be received several times by their neighbors than an item that arouses less interest. As a consequence, message loss is more likely to reduce the impact of disliked items than liked ones on the global precision (while decreasing the F1-Score as shown in Figure 5.4a).

Figure 5.4d completes these results by comparing the number of received messages without proxy change with different values of t1. Clearly message loss increases with the frequency of proxy changes. When nodes change proxies every 40 cycles the number of lost messages is one fourth of the total number of messages.

5.3 Latency analysis

Figure 5.5 analyzes latency in our PlanetLab deployment (PL). The plot shows the time required by the prps protocol to establish a proxy chain, and by a message exchange that uses the chains both on the source and on the destination side. In the case of chain creation (CC), latency results from key generation, encryption/decryption operations, and message transmission. In the case of message exchanges (ME), there are only encryption/decryption operations and message transmission; yet messages have to travel for twice as many hops as in the case of chain creation.

The time required to create the proxy chain increases significantly with its size, while time required for exchanging messages increases only slightly. Moreover, creating the chain takes approximately two to three times as long as forwarding a message (40s vs 15s with 8-hop chains), even though forwarded messages have to travel for twice as many hops. This clearly shows that latency results mainly from computational cost. To understand the reasons for this seemingly poor performance, we ran the same test by instantiating all the nodes on a local and dedicated server (LS), a quad-core Intel Xeon processor at 3.07GHz with 16Gb of RAM memory. On this server, both operations complete in less than 3s. This confirms that the high latency exhibited in a Planet-Lab setting results mainly from long processing times when performing cryptographic



(a) F1-Score (b) Precision

(c) Recall (d) Message loss

Figure 5.4: *F1-Score, precision, recall and message received according to various proxy chain change timeout (simulations).*

operations on overloaded virtualized nodes.[3]

5.4 Attacks

As described in Section 2.2, we consider an adversary model with Honest-But-Curious colluding nodes trying to identify the owner of a specific profile. In FreeRec, this mapping between users and profiles is hidden behind anonymous proxy chains, providing mutual anonymity between users exchanging their profiles. Neither a proxy nor intermediate relays in a proxy chain are able to know the identity of the source of the chain but are only aware of the previous and/or the next relay in the chain. However, the anonymity might be broken if the adversary controls all the nodes acting as relays in a proxy chain (i.e. all the segments of the chain). In this case, malicious nodes can retrieve the identity of the owner of a profile served by a proxy by following the proxy chain until its initiator. For a chain size l, the adversaries needs to control at least $l-1$ nodes otherwise it is impossible for them to monitor all the nodes in the proxy chain.

[3]Each PlanetLab node hosts a large number of virtual machines (slivers) that together result in high resource consumption.

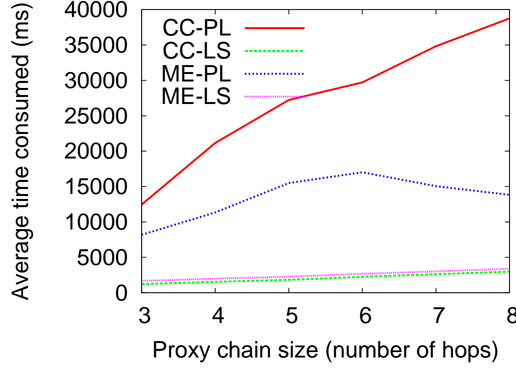


Figure 5.5: *Latency of chain creation and message forwarding (deployment).*

In the rest of this section, we refer to the number of intermediate relay nodes that must be controlled in a successful attack as $c = l - 1$.

5.4.1 Expected success rate.

We start our analysis by comparing the resilience of our deterministic-length chain, l, with that of a variable-length chain. We do this by proving the following simple property.

Property 1 Attackers wishing to associate users with their profiles will have a higher expectation of succeeding in a system based on variable-length chains with an average of c intermediate nodes than in one based on fixed-sized chains with exactly c intermediate nodes.

Proof. Let us consider a system of N nodes with $F \cdot N$ attackers. Also, let us assume that $N >> F \cdot N$ so that the probability of having an attacker in a chain is independent of the number of attackers it already contains. Under these assumptions, a coalition of attackers will compromise a fixed-chain protocol if it manages to control all the c intermediate nodes in a chain. This happens with a probability of success of $p_{s,fix} = F^c$ because the probability of having an attacker in a chain is equal to the fraction F of attackers in the system.

In the case of a variable-chain protocol, however, the length of the chain is no longer a constant c, but a random variable C with expectation $E(C) = c$. We can thus compute, the probability of a successful attack as follows.

$$p_{s,var} = \sum_{i=\mathrm{m}}^{i=\mathrm{M}} Pr(C = i) F^i = E(F^C)$$

Because F^x is a convex function, Jensen's inequality implies that $E(F^C) \geq F^{E(C)}$, and thus $E(F^C) \geq F^c$. The expected number of successful attacks with a variable chain is therefore at least as high as that obtained in the presence of a fixed chain. $\square_{Property\ 1}$

In the following section, we integrate this qualitative result, with a numerical analysis of the associated probabilities.

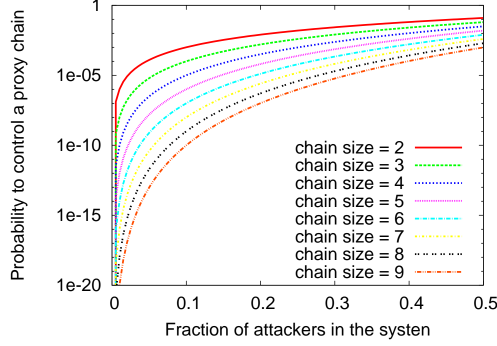


Figure 5.6: *Probability of a proxy chain being compromised by a set of attackers.*

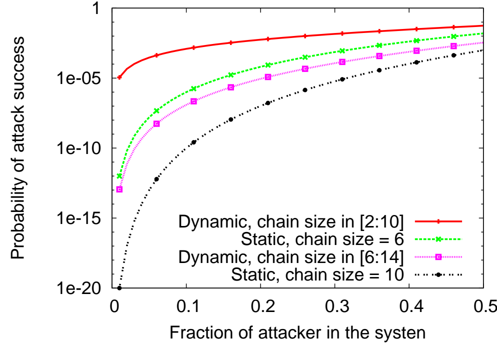


Figure 5.7: *Comparison of the attack effectiveness on fixed and variable chain size.*

5.4.2 Fixed chain size.

Figure 5.6 plots the probability of a successful attack in the case of a fixed size proxy chain in the presence of colluding attackers. The x-axis shows the fraction of colluding nodes in the system, while the y-axis shows the probability for a proxy chain to be controlled by the colluders. As described above, an attack is successful with probability $p_{s,fix} = F^c$. For example, with 10% of colluders and a $c = 6$, the probability that the attackers fully control one's chain is 10^{-6}, and the node can on average regenerate a new proxy chain $1,000,000$ times before its chain becomes compromised.

5.4.3 Variable chain size.

The case with variable chain sizes is a bit more complex. Every time a node has to build a new chain, she selects the chain size uniformly at random in $[m : M]$, not including the proxy. In this situation, the attackers can only match a user with her profile with certainty when they are *max* attackers in a row in a chain. Otherwise, there is a probability that a sequence of attackers only controls a subset of a chain, and that the attackers are surrounded by a random node and a random profile.

As we observed in the proof of Property 1, we can compute the probability of success of an attack, as the expectation $E(F^C)$ where C is a random variable. Because

the number of intermediate nodes in the chain C is uniformly distributed between $\mathsf{M}-1$ and $\mathsf{m}-1$, we have the following probability of success.

$$p_{s,var} = \sum_{i=\mathsf{m}}^{i=\mathsf{M}} \frac{F^i}{\mathsf{M}-\mathsf{m}+1}$$

Figure 5.7 plots this probability and compares it with the probability of success in the fixed case. As implied by Property 1, variable chain sizes invariably favor attackers that wish to associate a profile with its owner.

6 Related Work

Several distributed anonymity services exist that have features similar to FreeRec, such as Tarzan [FM02], Cashmere [ZZZR05] or AP3 [MOP+04].

Tarzan has an interesting mechanism to defend against malicious users running multiple Tarzan nodes. The authors make the assumption that while a user can easily create multiple Tarzan nodes, all these nodes will likely have their IPs in a small contiguous range. Thus, when a user selects a list of relays to create an anonymous channel, it selects nodes with IP addresses from different domains to avoid these malicious nodes. This feature could be included in FreeRec to increase the resilience against such attacks. One weak point of Tarzan is that every nodes needs to know the list of all the nodes currently in the system. This poses a problem of scalability as maintaining such data when millions of nodes are participating in the system is prohibitively costly.

One problem of anonymity services relying on a sequence of nodes to relay message is the high churn rate that is sometimes encountered in distributed systems, and forces nodes to periodically rebuild their proxy chains. Cashmere solves this problem by relying on groups of nodes, instead of single nodes, to relay the messages. As long as at least one node in a group is online, all the channels using this group as a relay will remain valid. However, Cashmere only provides anonymous communications, and not pseudonyms and is thus not suitable for our needs.

AP3 is a lightweight peer-to-peer anonymizer that also provides pseudonyms to the users. However, it only provides probable innocence, which is not enough in our context because the user profiles can be tracked, which allows for an iterative attack where the attacker can observe multiple communications.

Gossple [BFG+10] does provide anonymity to its users with what they call the gossip-on-behalf. With the gossip-on-behalf each node selects a proxy as well as a chain of nodes to communicate with it, similarly to what we do in FreeRec. However, the proxy does not simply serve to relay messages for the node, but instead runs the Gossple protocol in place of the user. Thus, the proxy is in charge of managing the profile and the neighbors of the user. While this solution reduces the amount of traffic between a node and its proxy, it leaves full control to the proxy over the node's data structures, which is not desirable.

7 Perspectives

We presented in Section 5.4 an analysis of the resilience of the fixed and variable chain size against a group of colluding adversaries. This analysis shows that the probability for a chain to be fully controlled by the attackers is is greater with the variable chain size. However, when the chain size is variable, the adversaries may not be able to tell whether they fully control

the chain, or if they control a subset of the chain. If they systematically guess that they fully control the chain as soon as they are at least m consecutive attackers in the chain, they will most of the time wrongly associate a user with a profile that is not hers. We are currently working on an analysis of how this uncertainty affects the adversaries' capacity to correctly associate users with their true profiles.

8 Conclusions

In this chapter we presented FreeRec, a decentralized architecture for building anonymous personalized services. FreeRec equips nodes with bidirectional onion-routing-like proxy chains that allow nodes to exchange their interest profiles without ever revealing their identities. FreeRec's core consists of three layers of gossip protocols. The bottom one is a Byzantine-resilient peer sampling protocol that provides nodes with the material to build their proxy chains. The middle layer, the prps, is an augmented rps protocol: it builds and maintains proxy chains and uses them to provide each node with a continuously changing anonymous sample of the network. The top layer completes the picture by providing each node with a cluster of anonymous interest profiles that most closely resemble its own. We extensively evaluated FreeRec in several ways: through simulations, with a PlanetLab deployment, and through a probabilistic analysis. Freerec can be used as a generic infrastructure and the work presented in Chapter 6 can use freerec to anonymize communication.

Chapter 6

Distributed privacy preserving clustering

1 Introduction

In Chapter 4 we introduced PTAPS, a modified version of TAPS including different mechanisms in order to protect the social network of the user (its social connections and their associated trust value). In the previous chapter, we introduced FreeRec, a anonymization protocol designed for gossip-based collaborative applications. These mechanisms manage to efficiently prevent an observer from discovering the identity of the users, their friends and how much they trust each other. However in both cases only the social network is protected and the user must disclose their profile to benefit from these applications. The content of these profiles can be highly sensitive, as it typically contains opinions about movies, items, pieces of news, or even very personal information such as browsing history.

In this chapter we go beyond and tackle the privacy issues related to the user profiles themselves. We propose DPPC (for Distributed privacy-preserving clustering), a distributed algorithm that provides each user with a *public* profile that can be shared with other users without harming the user's privacy. These *public* profiles are used to compute similarities between users in order for each of them to select her $k-nearest-neighbors$, which is the first step of user-based collaborative filtering.

2 Contribution

We introduce DPPC, a protocol that builds a profile for each user that does not reveal her personal data in clear and that she can therefore share with other users without exposing her privacy. Such a public profile can be used to perform similarity computation. DPPC is a distributed protocol, every user runs her own instance of DPPC locally on her own machine. Users private data remains on her machine and only a subset of the profile is exposed to other users in a anonymous way. The challenge though is to build a profile that remains useful with respect to similarity computations. The intuition behind DPPC is to rely on the items to "carry" users similarities.

DPPC uses the users participating in the protocol to create a repository for every item. The repository of an item is used to aggregate the profiles of all the users who have this particular item in their profile, and to publish this aggregated profile that we call the item profile. These

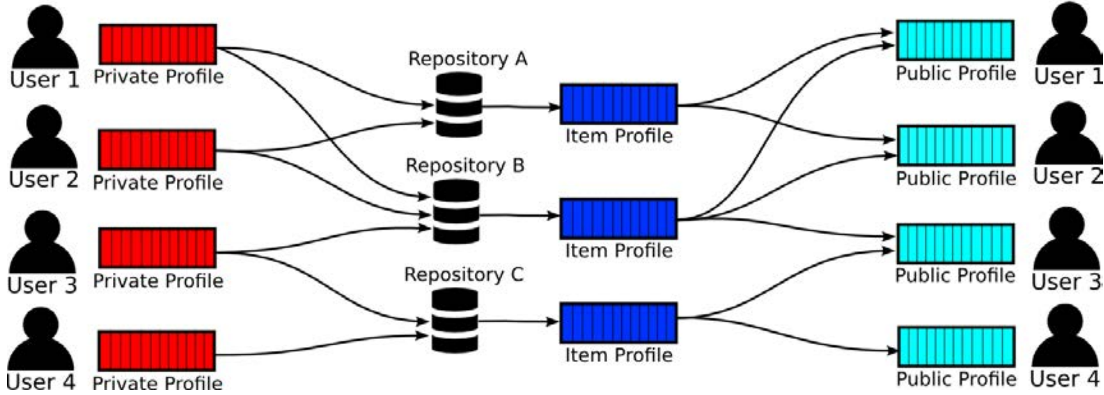


Figure 6.1: Workflow of the *public* profile construction.

aggregated profiles are built as follows: for each item a user likes, she sends a part of her profile to the repository of this item through an anonymous channel. An item repository is hosted by users themselves. An item repository is in charge of collecting the profiles from all the users whose profile contain the item, and making the item profile available to users. The aggregation of user profiles is done in a way that ensures that the repositories do not know which user liked which item. Given that item ratings are numerical values in a certain range, such as [0-10], we consider that a user likes an item if it rates it equally or above the average (for example, if the ratings are values in the range [0-10], a ratings greater or equal to 5 is a like).

For each item of her profile, a user fetches the item profile from the item repository through an anonymous channel. The *public* profile of a user is built by aggregating locally all such item profiles and can be shared with other users without revealing the user private data, called the *private* profile. The *public* profile of a particular user is thus the result of the aggregation of the profiles of the users who have the same items in their profiles preserving both privacy and utility as we will show in the following.

Public profile can typically be used to perform similarity computation and achieve k nearest neighbors computation using any KNN algorithm. We evaluate the performances of DPPC through experimentations based on the MovieLens [1] and the Jester [GRGP01] datasets. We first evaluate the quality of the clusters formed by our protocol by computing the KNN for each user with our protocol and then compare it to the optimal KNN obtained when the users use their *private* profile. We then evaluate the quality of the recommendations drawn from the clusters formed with our algorithm. Finally, we use different privacy metrics to compute how much information is leaked by a user's *public* profile with respect to her *private* profile.

3 DPPC Protocol

DPPC is a fully decentralized protocol that provides users with a profile that is "safe" to publish and share for clustering purposes without revealing the user's *private* profile.

3.1 DPPC in a nutshell

In order for users to compute their closest neighbors in the network, they need to be able to compute their similarity with each other. Our goal is to achieve this without disclosing the

[1] http://grouplens.org/datasets/movielens/

users profiles. Instead, our protocol provides each user with a profile that does not contain her private information, but nevertheless allows her to accurately compute her similarity with other users. We call such a profile a *public* profile, as opposed to the user's *private* profile that contains her private information. For the similarity between the *public* profiles of two users to be representative of the similarity between their *private* profiles, similar users should be provided with similar *public* profiles. The main intuition behind DPPC is to make similar users collaborate to build their *public* profiles. Since similar users have similar items in their profile, DPPC relies on items as the center of the cooperation between users.

To this end, DPPC creates a repository for each item, in charge of maintaining a profile for the item. The profile of an item is a digest of the *private* profiles of the users liking this item. Item profiles are computed using samples of the profile of users who liked the items that they send to the repository of this item (Section 3.5). The repository then aggregates all the samples of *private* profiles it received, to build the item profile and make it available to the users (Section 3.6). A user builds her *public* profile by querying the repositories of the items she likes, retrieving the item profiles and aggregating them locally (Section 3.7). This *public* profile can then be used it to perform similarity computation and compute her *k-nearest-neighbors*.

In the following, we describe the different steps of DPPC.

3.2 Adversary model

We assume an honest but curious adversary that executes the protocol correctly, but can analyze all the data it has access to in order to discover information about a user's *private* profile. DPPC is designed to protect the users' privacy against a single or even a colluding group of honest but curious adversaries.

However, in its standard form DPPC does not hold against an active adversary, *i.e.* an adversary that can deviate from the protocol, for example by sending bogus information to the other users. We give details about the attacks that can be performed by an active adversary in Section 4. We also describe and evaluate the effect on the performances of DPPC of a protection mechanism against active adversaries.

3.3 Item repositories

A repository is created for each item to collect samples of the *private* profiles of the users liking this item, aggregate all these profile samples into an item profile and provide it to the users willing to build their *public* profile.

Hosting item repositories In our system, we assume that every user is represented by a physical node, which holds the user's personal data and interacts with the nodes of other users. Items however do not have such a natural hosting mechanism, yet our objective is to design a fully decentralized repositories management and not to rely on an external service to host the items repositories. We achieve this by relying in the users to host item repositories. Each repository is hosted by one of the users node, who is in charge of storing the repository's data and exchanging data with the users querying the repository.

Locating an item repository When a user needs to access a repository, either to push her profile or pull the item's profile, she first needs to locate it. We consider two different types of items to recommend in our system, user-generated content and static content.

A user-generated content is, as expressed by its name, a content created by a user, such as a blog post or a picture. An example of such a system is Digg [2] in which users can post new content and express their opinion about the content created by other users. In this case, an item has a single known source which is the user who created it. We assume that the source node is in charge of choosing the node that hosts the repository. The source can host the repository itself or request a random node to host it. The source then attaches the address of the repository, as well as a public encryption key, to the item. When a node encounters an item through the recommender system, it knows how to access the repository.

Static content represent content that exists independently of the system, e.g., movies, news articles or books. In this case, users rate content that exist outside the system, thus the same content can be rated, commented and shared by multiple users concurrently. For example, the Netflix [3] or MovieLens [4] websites allow users to express their opinion about movies that are part of their database. For such content without a clear identified source, we assume the use of a DHT [SMK+01, RD01, ZHS+04] in which all nodes participate in order to store the items repositories. The users use the ID of an item as a key in the DHT in order to locate and communicate with the item repository.

Note that a DHT could also be used for the user-generated content. However, maintaining a DHT has a cost that we can avoid by having the source selecting the nodes hosting the repositories. Since this is not the focus of DPPC, we do not provide further details on repository management but there exist a large number of systems that could be used for that purpose.

Churn P2P systems usually encounter a high churn rate, which means that users connect and disconnect from the system frequently. This can strongly impact DPPC as the node responsible of a given item repository might not always be available. As a consequence, a user may have to build its *public* profile based on only a fraction of the item repositories which is likely to reduce the utility of this profile as it will contain less information.

The typical solution to alleviate the impact of churn is to replicate the content over several nodes, so that the probability that at least one will be online at any time remains high, which can be done in DPPC by replicating the repositories. In the case of user-generated content, the source of the item can simply select several nodes to host the item repository, and attach a list of addresses to the item instead of a single one. In the case of the static content, DHTs typically handle at least a moderate level of churn, and some are specifically designed to handle very high churn rates.

Churn management strategies are out of the scope of this document, thus these solutions are not integrated and evaluated in DPPC. As for item repositories, there exist several mechanism that can be used to replicate them. We do however evaluate the effect of churn on DPPC without including any anti-churn mechanism in Section 6.4. We show that DPPC behaves well in the presence of churn if the number of items in the system is large. Indeed, when the number of items is large, and similar users like many items in common, even if a large fraction of the items repositories are missing similar users still have access to several repositories they share, which is enough to create similar *public* profiles.

3.4 Anonymous communication

In DPPC, nodes need to share parts of their *private* profiles with the repositories in such a way that a repository cannot link a *private* profile with its owner. Thus, communications

[2] http://digg.com
[3] http://www.netflix.com
[4] http://movielens.umn.edu

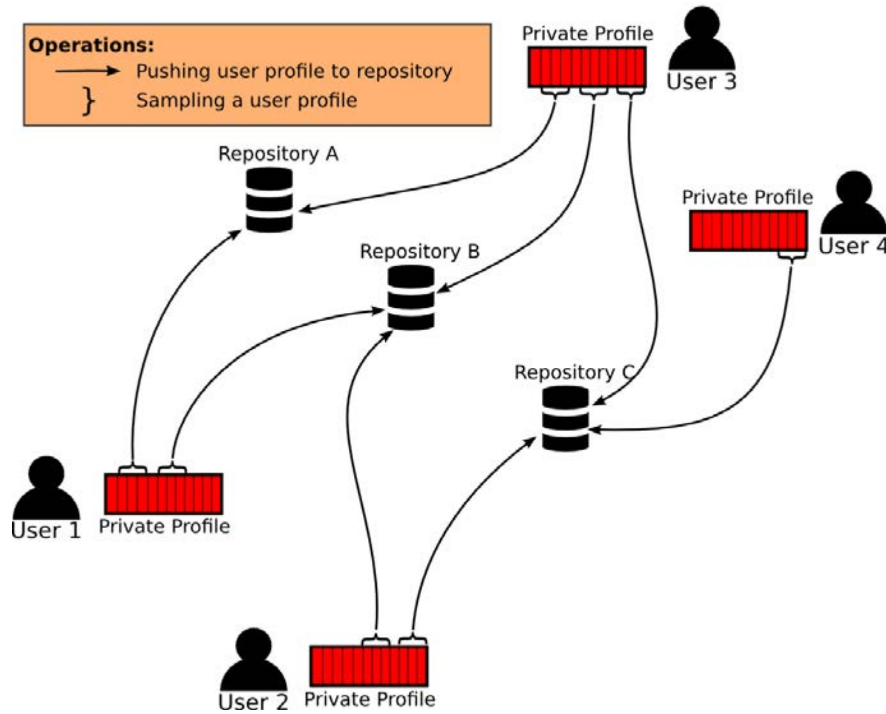


Figure 6.2: Nodes pushing samples of their *private* profile to the repositories of items they liked.

between a node and a repository must be done through an anonymous communication channel, that allows the node to send and receive messages to and from the repository without the repository knowing the actual node's identity. In addition, since nodes only communicate with the repositories of the items they like the fact that a user u communicates with a repository r should be hidden as well.

This is done by encrypting the data to be exchanged between the node and the repository, and use several intermediary nodes to relay the message so that none knows both the identity of the node and the repository. Multiple systems provide this feature, such those presented in Chapter 5

In some of these systems, the data is not encrypted by the sender (for example AP3). In this case, the intermediary nodes can see the content of the profiles samples being transmitted, which weakens the privacy provided by DPPC. If such a system is used, a public encryption key can be associated with each repository, and used by the nodes to encrypt the private information they send to the repository. This key can be shared with all the nodes either by attaching it to the item in the case of user-generated content, or accessible through the dht for static content.

3.5 Pushing to a repository

When a node adds an new item it likes to its profile, it locates the node hosting its repository. If the repository is online the node sends it its *private* profile, as shown in Figure 6.2. As explained in Section 3.4, all communications between the node and the repository are done through an anonymous communication channel, thus the repository does not know the identity of the owner of the private information it receives. However, the *private* profile of a user can

Algorithm 4 Sample User Profile: RANDOM sampler

Input: $profile$ ▷ User profile to sample
Input: $RS_{fraction}$ ▷ Fraction of the profile to select
Input: $rand$ ▷ Random number generator
1: $sample \leftarrow \emptyset$
2: $toSelect \leftarrow |profile| \cdot RS_{fraction}$
3: $remaining \leftarrow |profile|$
4: **for all** $< item, rating > \in profile$ **do**
5: $probaSelection \leftarrow \frac{toSelect}{remaining}$
6: **if** $rand.get() < probaSelection$ **then**
7: $sample \leftarrow sample \cup < item, rating >$
8: $nbToSelect \leftarrow toSelect - 1$
9: **end if**
10: $remaining \leftarrow remaining - 1$
11: **end for**
12: **return** sample

contain personal information that, if linked together, can allow an observer (here the repository) to discover the true identity of the user. for example, an information related to geolocation or time schedule is not enough to identify a user, but combining multiple data can lead to discover where someone lives, works, the places she likes to visit, etc.

To reduce the effectiveness of this type of analysis nodes alter their profiles before sending them to the repositories. The objective of the alteration is to limit the information that can be inferred about a user from her profile. The solution we propose is to send only a sample of the user's profile in order to limit the information it contains as detailed below.

Sampling a user profile

We propose two sampling strategies, the random sampler and the collusion-resilient sampler.

Random sampler Each time a sample needs to be sent to a repository, the random sampler selects a random sample of the profile. The size of the sample is determined based on a parameter, $RS_{fraction}$, that is the fraction of the items from the *private* profile that should be included into the sample. This ensures that a user will never expose more than $|profile| \cdot RS_{fraction}$ items of her profile to a repository. The pseudo-code of the random sampler is given in Algorithm 4

As shown in the evaluation (Section 6), the random sampler provides good performance from the utility point of view and effectively protects the privacy of users from a single adversary. However, if several repositories are colluding they may be able to rebuild the profile of a user from its different samples. Indeed, two random samples of the same profile are likely to overlap (some items from the profile will appear in both samples), and this can be used to estimate likelihood that two samples originate from the same profile. For example, with two samples of size 100 of a profile of size 1000 on average the two samples will share 10 items.

A set of colluding repositories may detect that profile samples with a large overlap, are likely to originate from the same profile, and combine them to rebuild the user profile. The probability to perform this attack with success (i.e., without including samples from another user) strongly depends on the total number of items and the number of users. If the number of items is small, and the number of users is large, it is likely that many users will have very

similar similar profiles and generate overlapping samples, thus this attack may be inaccurate. However, in the general case it is possible for this attack to succeed. In this situation we propose an alternative profile sampler, the collusion-resilient sampler, which prevents this attack.

Collusion-resilient sampler The objective of the COLLUSION-RESILIENT sampler (or CR sampler) is to ensure that colluding repositories cannot rebuild a user's *private* profile from the profile samples she created. We consider two attacks, the *matching* attack and the *elimination* attack.

The *matching* attack, consist in finding pairs of samples that strongly overlap, and that are thus very likely to originate from the same *private* profile. By comparing all the samples they know, the attackers may be able to fully rebuild a user's *private* profile.

The *elimination* attack, is the opposite and consists in discovering that pairs of samples originate from different profiles. A repository can tell that two samples originate from different profiles if the same item is rated differently in both samples, we say that the two samples conflict. By analyzing the content of a sample of user u, a repository, r_1, can see a subset of the items rated by u. If an item is highly rated, then the repository, r_2, of this item contains another sample of u's *private* profile (because u liked this item). r_1 can collude with r_2 and search for u's sample in r_2's list of samples by eliminating all the samples that conflict, until only one (or a few) remain. Among these remaining samples is u's sample.

The CR sampler prevents these attacks as follows:

I) The set of all items, I is split into CR_{groups} groups, $I_1, I_2 ... I_{CR_{groups}}$.
   a) $I_1 \cup I_2 \cup ... \cup I_{CR_{groups}} = I$.
   b) $I_i \cap I_j = \emptyset$ unless $i = j$.
   c) The distribution of items into groups is the same for all users.

II) A sample contains all the items of a *private* profile belonging to a given group.
   a) $sample_1 \cup sample_2 \cup ... \cup sample_{CR_{groups}} = private\ profile$.
   b) $item \in sample_i \implies item \in private\ profile$
   c) $item \in sample_i \implies item \in I_i$

III) The samples received by the repository of item i only contain items from the group of i.

The *matching* attack is prevented by I) and II), as it ensures that two samples from the same profile are either identical or do not overlap at all. Because of I)c), two samples from different profiles do not overlap unless they belong to the same group. Thus to perform the *elimination* attack, the adversary needs to eliminate pairs of samples from different groups. As previously explained, the first possibility is to find a common item with a different rating, which is impossible as samples from different groups do not share any items. The second possibility is to eliminate pairs of samples received by the same repository. However because of III) a repository only receives samples from the same group, thus it cannot eliminate pairs of samples from different groups.

The association of items to groups is required to be the same for all users, thus it should be computed based on knowledge available to all users. We take the modulo of the item ID by the number of groups:

$$group_i = ID(i)\ mod\ CR_{groups},$$

where $ID(i)$ is, for example, the hash of the item content, and CR_{groups} the number of groups. The pseudo-code of the colluding resilient sampler is given is Algorithm 5).

Algorithm 5 Sample User Profile: COLLUDING RESILIENT sampler

```
Input: profile                                   ▷ User profile to sample
Input: CR_groups                                 ▷ Number of groups
Input: repository_id                             ▷ Id of the item repository
 1: repository_group ← group(repository_id, CR_groups)
 2: sample ← ∅
 3: for all < item, rating >∈ profile do
 4:     item_group ← group(item_id, CR_groups)
 5:     if repository_group == item_group then
 6:         sample ← sample ∪ < item, rating >
 7:     end if
 8: end for
 9: return sample
```

Algorithm 6 Create Item Profile: MEAN aggregator

```
Input: userProfiles                              ▷ List of user profiles
 1: itemProfile ← ∅
 2: for all item ∈ Items do
 3:     rating ← 0
 4:     occurences ← 0
 5:     for all profile ∈ userProfiles do
 6:         if profile.contains(item) then
 7:             rating ← rating + profile.get(item)
 8:             occurences ← occurences + 1
 9:         end if
10:     end for
11:     if occurences > 0 then
12:         rating ← rating ÷ occurences
13:         itemProfile ← itemProfile ∪ < item, rating >
14:     end if
15: end for
16: return profile
```

3.6 Aggregating profiles in a repository

Each repository maintains a list of the profiles it received in order to build the item profiles. The repository creates an item profile upon a user request by aggregating the user profiles from this list. The aggregation is performed by the mean aggregator, whose pseudo-code is given in Algorithm 6. The mean aggregator starts by creating an empty item profile and iterates over all the items. For each item i, it computes the average rating of i over all the user profiles it received in which i is rated. If no user profile contains i, i is not added to the item profile. Otherwise, i is added to the item profile, associated with the average rating computed previously. The resulting item profile is an average of the profiles of the users interested in i. Profiles generated with the mean aggregator are composed of a list of items associated with ratings, which makes them similar to user profiles in structure.

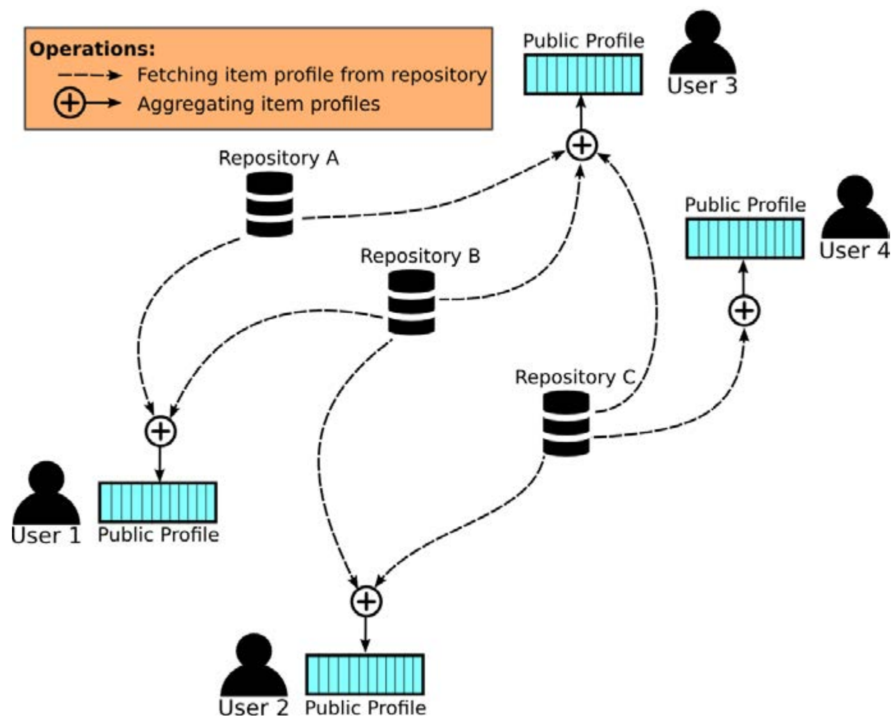


Figure 6.3: Nodes pulling the profiles of the items they like from their repositories and aggregate them in order to build their *public* profile.

3.7 Creating a public profile

Nodes periodically rebuild their *public* profile (the profile they share with other users to compute their similarity) to keep it up to date with the new items they store in their profile. This process is shown in Figure 6.3 The node starts by collecting the profiles of all the items it liked. For each of these items, the node locates its repository, and sends a request to fetch the item profile through an anonymous channel, as described in Section 3.4. Once the node has received all the item profiles, it needs to aggregate them locally to create its *public* profile. User profiles and item profiles have the same structure, and consist of a mapping from an item, or an item ID, to a rating. Users use the mean aggregator to aggregate the different item profiles they collect, the same way repositories aggregate user profiles.

3.8 Computing similarities

Once the users have built their *public* profiles, they share it with the others users for similarity computation purposes. After computing its similarity with other nodes, the node selects the most similar of them to form its KNN. The similarity is computed between a node's *public* profile and the other nodes' *public* profiles, the *private* profiles are never involved.

Technically, nodes could use their *private* profiles to compute their similarities with the *public* profiles of other users. However, this would lead to a privacy breach. Indeed, the output of the KNN computation would depend on the content of the *private* profile, and this can be exploited by an adversary do disclose the node's *private* profile. For example, an adversary could forge a set of *public* profiles, and discover which are the most similar to the user's *private* profile. By generating a large number of *public* profiles, the adversary could ultimately find out the content of the user's *private* profile.

By using only the *public* profiles to compute the similarity, an adversary cannot gain any additional information by observing the output of the KNN selection. Indeed, the adversary can know the *public* profile of the user, as well as the ones of the other users, and can thus compute by itself the closest nodes to the user.

The only point in the whole protocol where information from the *private* profiles are disclosed is when they are sampled to be sent to the repositories, which is done anonymously.

4 Active adversary

DPPC protects the users privacy against a set of colluding passive adversary trying to rebuild a user's profile by combining different samples, but respects the protocol. However, an attacker could deviate from the protocol in order to perform a more efficient attack.

One attack that becomes possible is the watermarking attack. In the watermarking attack, an attacker adds some elements into an item profile so that if a user includes this item profile into her public profile, the attacker will be able to detect it. For example, assuming that the profiles contain mappings from item ids to ratings, an active attacker could add to an item profile a set of item ids that do not correspond to any item[5]. Once these fake ids are added to the item profile, any user liking this item and fetching its profile to build its public profile will add the fake ids to it. The attacker then look at all the public profiles it sees, and searches for the fake ids into it. If a user has the fake ids in her public profile, she liked the item.

There are two ways an attacker can modify an item profile: If the attacker is responsible for hosting the repository of this item, she can easily do any modification she wants to the item profile; If the attacker does not control the item's repository, then it can send a profile sample

[5]We assume that the attacker is able to know the list of all the items that exist in the system

Algorithm 7 Create User Profile: THRESHOLD aggregator

```
Input: itemProfiles                                          ▷ List of item profiles
 1: publicProfile ← ∅
 2: for all item ∈ Items do
 3:     rating ← 0
 4:     occurences ← 0
 5:     for all profile ∈ itemProfiles do
 6:         if profile.contains(item) then
 7:             rating ← rating + profile.get(item)
 8:             occurences ← occurences + 1
 9:         end if
10:     end for
11:     if occurences > aggr_threshold then
12:         rating ← rating ÷ occurences
13:         publicProfile ← publicProfile ∪ < item, rating >
14:     end if
15: end for
16: return profile
```

containing the fake ids to the repository. Unless the repository knows the list of valid item ids, it is impossible for it to reject the fake ids.

Our solution to reduce the efficiency of this attack is to modify the way users aggregate the item profile when building their public profile. Instead of including all the items that appear in at least one of the item profiles, users include them only if they appear in at least $aggr_{threshold}$ item profiles. We call this alternative aggregator the threshold aggregator. Algorithm 7 shows the corresponding pseudo-code.

When the users are using this aggregator, the attacker needs to watermark at least $aggr_{threshold}$ item profiles, and the users must like at least $aggr_{threshold}$ of the watermarked items for the attack to succeed.

5 Experimental Setup

5.1 Competitors

We evaluate the performances of DPPC by comparing it against two reference KNN protocols: Clear profiles and Random neighbors.

Clear profiles CLEAR PROFILES is a KNN protocol that computes the closest neighbors of each user based on the similarity between *private* profiles. This protocol computes the best possible KNN, but does not protect the privacy of users. The objective of DPPC is to provide privacy to users while maintaining a high level of utility, i.e. the clusters formed when using DPPC should differ as little as possible from the ones formed without DPPC.

Random neighbors RANDOM NEIGHBORS selects random neighbors for each user. This protocol provides a high level of privacy as the users do not need to compute their similarities, and thus are not required to share any private information. However as we will see, selecting random neighbors causes this protocol to achieve poor utility. We use RANDOM NEIGHBORS as a lower bound for the clustering quality.

DPPC We evaluate DPPC with various sets of parameters. First, we vary the size of the profile samples sent by the users to the repositories. Second, we evaluate the effect of churn on DPPC. Third we evaluate the impact of the THRESHOLD aggregator which protects users against an active adversary on the performances. Finally, we evaluate how well DPPC protects users' privacy.

5.2 Similarity metric

DPPC provides each user with a public profile that is a mapping from item IDs to a real value, which is similar to a classical user profile. Thus, it does not rely on any particular similarity metric, and any metric that can be used to compute similarity between two users profiles can be used with DPPC. In this evaluation, we use the well known cosine similarity metric [SM86].

5.3 Datasets

We evaluated the performances of DPPC over two different datasets, MovieLens 100k and Jester.

MovieLens The MovieLens 100k [6] dataset is a dataset containing ratings of movies. It contains 1,000 users, 1,700 items and 100,000 ratings, which makes 100 rating per user and 59 ratings per movie on average. Each rating is an integer value in $[1, 5]$.

Jester The Jester dataset [GRGP01] contains ratings given by users to jokes. It contains 25,000 users, 100 items and 1,800,000 ratings, which makes 72 ratings per user and 18,000 ratings per joke on average. Each rating is a real value in $[-10, 10]$

We split both datasets into two independent sets: a training set containing 80% of the ratings, and a testing set containing the remaining 20%. The computation of the k-nearest-neighbors of each user is based only on the ratings from the training set. The rating prediction are computed based on the training set and checked with the ratings contained in the testing set.

In DPPC, nodes interact with the repositories of the items they like. However in these two datasets, the users opinion is not expressed by a boolean value, but by a numerical rating. We make the assumption that a user likes an item if her rating is greater or equal than the average of the extreme ratings. This average is $\frac{1+5}{2} = 3$ for the MovieLens dataset and $\frac{-10+10}{2} = 0$ for the Jester dataset.

5.4 Evaluation metrics

We evaluate DPPC along several metrics to assess (i) the differences between the clustering views formed by DPPC are those obtained using clear profiles, and (ii) how good they are from a recommendation point of view.

5.4.1 Clustering quality

We start by evaluating the quality of the clustering obtained by the different algorithms. We run each experiment with each algorithm and compare the neighborhoods obtained for each user after letting the algorithms converge (i.e., we let the clustering algorithms run until the neighbors of each user no longer change).

[6] http://grouplens.org/datasets/movielens/

A straightforward way to evaluate the quality of the clustering would be to compute the overlap between the optimal cluster of a user (i.e., the one formed by clear profiles) and the one created by DPPC. However, for a clustering algorithm to be good, it does not necessarily need to select the best nodes. Selecting good nodes usually turns out to be enough [BFG+13]. Moreover, the overlap is too strict as a metric, as selecting the $k+1$ to $2k$ best nodes instead of the 1 to k best nodes would lead to the same score as selecting any set of nodes that are not in the top k.

Instead, we use two different metrics to assess the clustering quality: the nodes ranking and the similarity ratio.

Node ranking In the node ranking metric, for each node A, we sort all the other nodes according to their real similarity with A (the one based on their clear profiles), and use the resulting order as a reference ranking. For each algorithm, we compute the k closest neighbors of each node, and compute the average ranking of the selected nodes with respect to the reference ranking. We then take the difference between this ranking and the reference ranking, which can be computed as follow:

$$\text{average reference ranking} = \frac{1}{k} \cdot \sum_{n=1}^{k} n = 1 + \frac{k}{2}$$

This metric expresses how far the selected cluster is from the reference one from the ranking point of view. The lower the average ranking, the better the quality of the clustering.

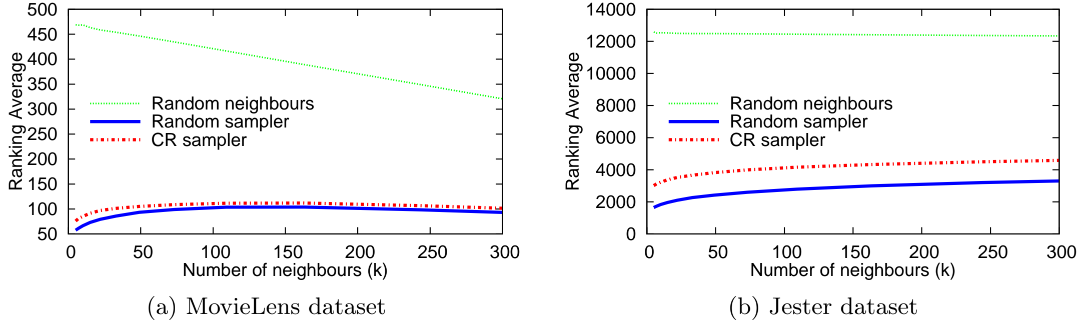
Similarity ratio The node ranking metric is a useful metric to evaluate a set of neighbors with respect to the reference neighbors. However, it is not necessarily an accurate metric for the utility of the clustering. For a user-based recommender system, which is the main target of DPPC, what is important is to find highly similar neighbors; finding the most similar ones is not always necessary. Indeed, consider a case where the average similarity in the reference cluster is 0.8, and that there exist dozens of nodes with a similarity of 0.79. An algorithm that selects some of these users with a 0.79 similarity may have a poor node ranking while nodes with 0.79 similarity are probably almost as good as nodes with 0.8 similarity when used to compute recommendations.

We use a different metric to evaluate the utility of the clusters from the similarity point of view, which we believe is the most important for recommender systems, the similarity ratio. The similarity ratio metric computes the ratio between the average similarity inside the selected cluster and the average similarity inside the reference cluster. It is a measure of how close the similarity of the cluster is to the reference. A value close to 1 means that the selected cluster is almost as good as the reference cluster, regardless of the node ranking.

5.4.2 Rating prediction

We evaluate the quality of the predictions computed using the generated clusters. As described in Section 5.3, the datasets are split between a training set and a testing set. The KNN of each user is computed based only on the training set, and the rating prediction phase attempts to predict the 20% of ratings of the testing set based on the KNN.

The objective of this evaluation is to complement the evaluation on the clustering quality by observing the impact of the different algorithms in the specific context of a user-based recommendation system. This is not part of DPPC itself which only provides privacy to the users for the similarity computation and KNN selection. Thus during the phase of rating predictions,



(a) MovieLens dataset (b) Jester dataset

Figure 6.4: Average ranking of the KNN, $RS_{fraction} = 0.2$ and $CR_{groups} = 5$.

we assume that users share their *private* profile with others similar users so that they can predict their ratings.

We evaluate the quality of the recommendation by computing the mean absolute error of the predictions. For each user, we attempt to predict every rating of this user that is in the testing set (the 20% of ratings that are not used during the clustering phase). For each prediction, we compute the absolute value of the difference between the predicted rating and the true rating, and average it over all the ratings of all users

$$MAE = \frac{\sum_{i=1}^{n} |rating_i - prediction_i|}{n},$$

with n being the number of ratings in the testing set.

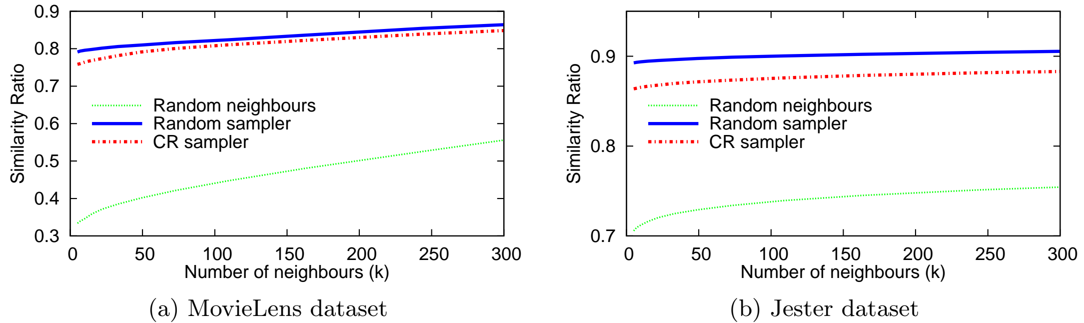
The lower the MAE the better, a value of 0 meaning that all the ratings were predicted perfectly.

6 Experimental results

6.1 Clustering quality

Figure 6.4 shows the average ranking of the neighbors selected with DPPC and at RANDOM. Results are displayed for various values of k, and for both dataset. We observe that for both datasets, the gap between CLEAR PROFILES and DPPC is fairly stable when k changes. For MovieLens (Figure 6.4a), the nodes selected by DPPC are on average ranked 100 positions below clear profiles, and 4000 positions below for Jester (Figure 6.4b). With random neighbors and MovieLens however, this gap decreases when k increases. This is because the MovieLens dataset contains only 950 users, and when random nodes are selected, their average ranking is around 475 and is not affected by k. Given that the average ranking of clear profiles is $1 + \frac{k}{2}$ as explained in Section 5.4.1, the gap between RANDOM NEIGHBORS and CLEAR PROFILES is $475 - 1 + \frac{k}{2}$, which decreases with k. The same reasoning applies to Jester, but because the Jester dataset contains 25,000 users, the effect is very small for the values of k we used. While DPPC selects clearly better neighbors than RANDOM NEIGHBORS, there is a significant gap between CLEAR PROFILES and DPPC.

In Figure 6.5 we observe that, despite the discrepancy of the ranking of nodes selected by DPPC and the ones selected by CLEAR PROFILES, the similarity between a node and its KNN is almost as high with DPPC as with CLEAR PROFILES. Indeed, for the MovieLens dataset, the



(a) MovieLens dataset (b) Jester dataset

Figure 6.5: Similarity ratio of the KNN, $RS_{fraction} = 0.2$ and $CR_{groups} = 5$.

average similarity of the nodes in the DPPC clusters is around 85% of the one in the CLEAR PROFILES clusters, and around 90% in the Jester dataset. This means that in practice, from the utility point of view, there is very little difference between CLEAR PROFILES and DPPC.

It is interesting to see that while the gap of raking in the Jester dataset is greater than in the MovieLens dataset, the average similarity of the selected nodes is closer to the reference. This can be explained by the fact that the Jester dataset is denser than the MovieLens one, thus each user has on average more users that are highly similar to her in the dataset.

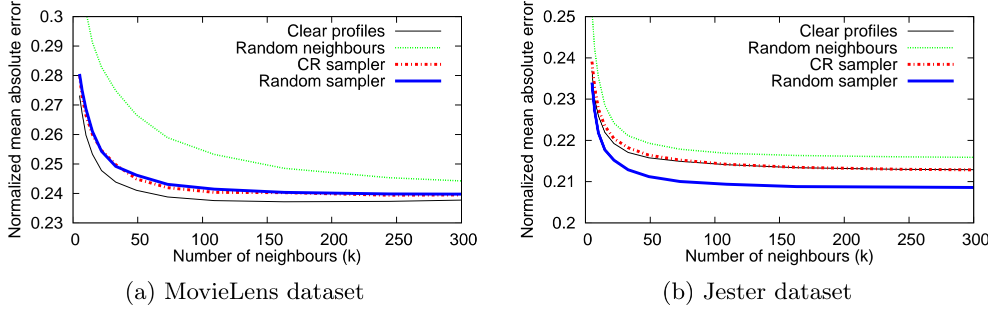
Another observation from Figure 6.4 and Figure 6.5 is that with the MovieLens dataset, the RANDOM and the COLLUDING RESILIENT samplers give very similar results, while we witness a clear advantage of the random sampler in the Jester dataset.

The results also show that even if DPPC clusters are not as good as those obtained with clear profiles, they remain much better than random clusters.

6.2 Recommendation quality

Figure 6.6 shows the normalized mean absolute error (NMAE) achieved by the different protocols on both the MovieLens and Jester datasets. For the MovieLens dataset (Figure 6.6a) we observe that DPPC achieves performances very close to CLEAR PROFILES. This is consistent with our previous observations regarding the clustering quality. We also see that the profile samplers have very little impact on performance with this dataset.

The results are fairly different with the Jester dataset, and we can notice 2 interesting things from Figure 6.6b. First, DPPC with the random sampler achieves significantly better results than clear profiles. This is possible for DPPC to perform better, because even if clear profiles computes its rating predictions based on a cluster that is optimal according to a similarity metric, this does not translate into optimality with respect to rating prediction. Our explanation for this phenomenon is that it results from the way ratings are predicted. As explained in Section 5.4.2, we do a prediction for each rating of the testing set based on the ratings of this item in the user's neighborhood. But in the case where none of the user's neighbors rated this item, we revert to a default prediction which is the average rating given by this user. Users who are the most similar usually have very close profiles, which means that most of the items that a user saw have also been seen by her closest neighbors. For example, if all the neighbors of a user are perfectly similar to that user, i.e., they rated exactly the same items, this user will not be able to extract any recommendation from these neighbors because they do not have any item to propose. In this case, users that are slightly less similar, but have more items to propose



(a) MovieLens dataset (b) Jester dataset

Figure 6.6: Accuracy of the rating prediction, $RS_{fraction} = 0.2$ and $CR_{groups} = 5$.

become more useful. As we saw in Section 6, the nodes selected by DPPC are slightly less similar than the ones of CLEAR PROFILES, which explains why DPPC performs well in this context.

The second interesting point about Figure 6.6b is that the performances of DPPC are strongly impacted by the sampler. We believe that this is due to the fact that the Jester dataset contains only 100 items, and many users liked only few of them.

With the CR sampler, if a user likes no items from a given group, no item from this group will appear in her public profile. Thus, a difference of only one item between two *private* profiles can result in a large difference between the two associated public profiles. This situation is more likely to happen when the number of items is low, as in the Jester dataset. On the other hand, the random sampler does not suffer from this drawback as a user can have a public profile containing any of the items as soon as she liked at least one item.

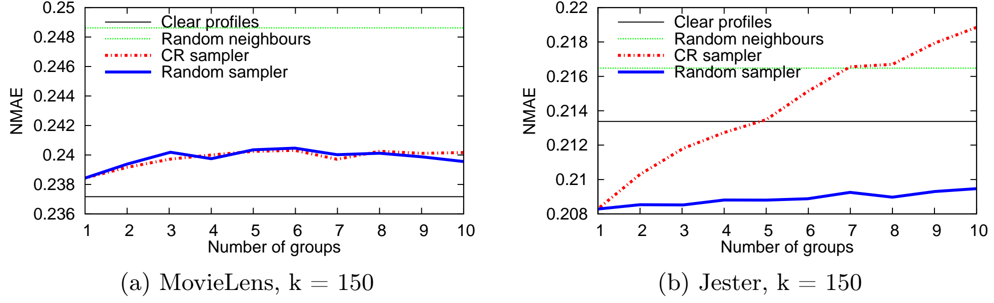
With the CR sampler, all the ratings received by an item repository belong to the same group (the group of the repository), which means that if a user liked items from only n different groups, its public profile will only contain ratings from items of n different groups. Consider two very similar users u_1 and u_2 whose profile are identical except for a single item of a group g that u_1 liked and u_2 did not rate. The public profile of u_1 is likely to contain several items from group g, while the public profile of u_2 will not contain any (assuming that u_2 did not like any item of group g). This difference of a single item between u_1 and u_2 will lead to a large difference between u_1's and u_2's public profiles. On the other hand, the random sampler sends random items to the repositories, which means that each repository can contain any of the items. In such a situation, two users whose *private* profile differ by only one element are very unlikely to have a large difference between their public profiles.

6.3 Effect of the sample size

The level of privacy provided by DPPC depends on the configuration of the samplers. The larger the samples sent to the repositories, the higher the probability that a repository will be able to make deductions about the user based on the sample. For the CR sampler, the items are split into CR_{groups} groups, thus the expectation of the number of items in a sample of a profile P is:

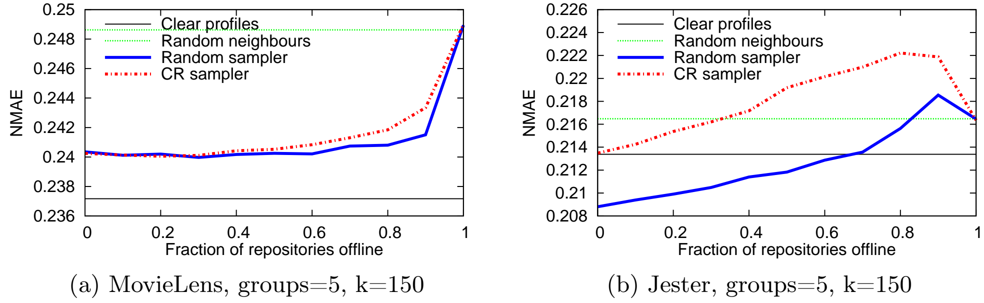
$$E(|CR_{sample_P}|) = \frac{|P|}{CR_{groups}}.$$

The Random sampler is controlled by the $RS_{fraction}$ parameter, which defines the fraction of the profile that should be included in the sample. The number of items in a random sample



(a) MovieLens, k = 150 (b) Jester, k = 150

Figure 6.7: Impact of CR_{groups} and $RS_{fraction}$ on the prediction accuracy



(a) MovieLens, groups=5, k=150 (b) Jester, groups=5, k=150

Figure 6.8: Impact of the churn ratio of the repositories on the NMAE.

is:

$$|RS_{sample_P}| = |P| \cdot RS_{fraction}.$$

Figure 6.7 shows the evolution of the nmae when the number of item groups increases. For simplicity, we only display the CR_{groups} parameter on the x-axis and assume $RS_{fraction} = \frac{1}{CR_{groups}}$. We observe than with the MovieLens dataset (Figure 6.7a) the number of item groups has little impact on the recommendations for both samplers. However with the Jester dataset (Figure 6.7b) there is again a strong difference in the behavior of the random sampler which is only slightly affected by the number of groups and the cr sampler which performances degrades very strongly. As previously, we explain this by the fact that the Jester dataset contains only 100 items, against 1600 for MovieLens.

6.4 Impact of churn

As any distributed algorithm, DPPC is subject to churn. One potential weakness of DPPC is that the nodes need to access the repositories in order to build their public profiles. Unless a mechanism is set up to maintain the repositories online (such as replicating the content of the repositories over several nodes), the repositories will not always be accessible, some might even be lost forever if the nodes responsible for them leave and never come back in the system. While this is not the focus of our work, we still evaluate the impact of churn on DPPC. To this end, we simulated churn by removing from the system a fraction of the repositories and having the nodes build their public profiles only with the repositories that remain online.

Results are displayed in Figure 6.8 in terms of NMAE against the fraction of offline repositories. We observe that on the MovieLens dataset, which has a relatively large number of items (1600), churn has very little impact on the rating prediction until 60% of the repositories are offline. This is because there is a form of implicit redundancy in DPPC due to the fact that similar users share many items, and thus share many items repositories. For example, if two users share 100 items, even if 60 of them are unavailable, there are still 40 repositories that they will both use to create their public profile, which will make their public profiles close enough to each other.

With the Jester dataset however, the prediction accuracy continuously decreases as repositories are taken out of the system. We believe that this is a consequence of the small number of items in the Jester dataset which reduces the redundancy. The NMAE decreases when the churn rate reaches 1 because then all users have an empty public profile and thus select random neighbors.

6.5 Effect of the watermarking protection

As discussed in Section 4, an active adversary in charge of a repository could modify the content of the item profile it hosts in order to be able to identify any user including it into her public profile. One mechanism we proposed to protect users against this kind of user attack is to use an alternative aggregator when building the public profiles. This aggregator, the threshold aggregator only includes entries that appear in at least $aggr_{threshold}$ profiles of items the user liked. This methods makes the watermarking attack less efficient, however it also makes the public profiles less accurate, as some of the items are removed from them.

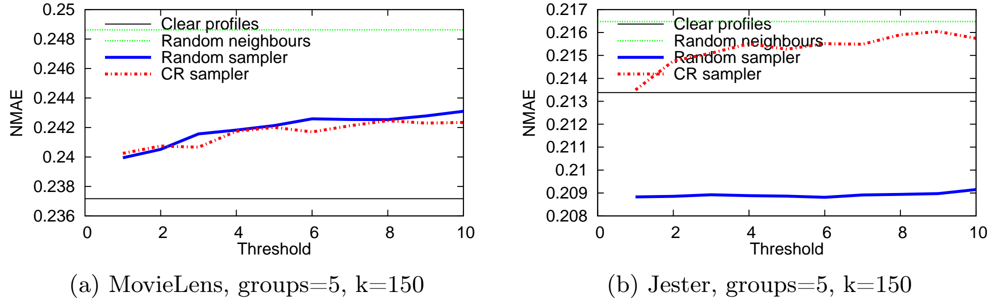
Figure 6.9 shows the impact of the threshold aggregator on the nmae when varying $aggr_{threshold}$. We observe that the variation of the $aggr_{threshold}$ is similar to the effect of the variation of the CR_{groups}. With the MovieLens dataset, it causes a slight increase of the nmae for all the samplers. With the Jester dataset, it has very little impact on the random sampler, but a significant one on the CR sampler. When the number of items is small, it becomes less likely to find an entry corresponding to an item i in a large number of item profiles. This is aggravated by the cr sampler, which further divides this probability by the number of groups. In the case of the Jester dataset, with the cr sampler and $CR_{groups} = 5$, there are only 20 items per group. If $aggr_{threshold} = 10$, then for an item to appear into a user's profile, it must be in at least 10 of the item profiles of its group that the user likes, which makes the probability very low and explains why with $aggr_{threshold} = 10$ the performances with the cr sampler are very close to those of the RANDOM NEIGHBORS.

We conclude that the THRESHOLD aggregator, that protects the users against the watermarking of the item profiles, can be used without hampering too much the quality under the condition that the number of items is large.

6.6 Privacy

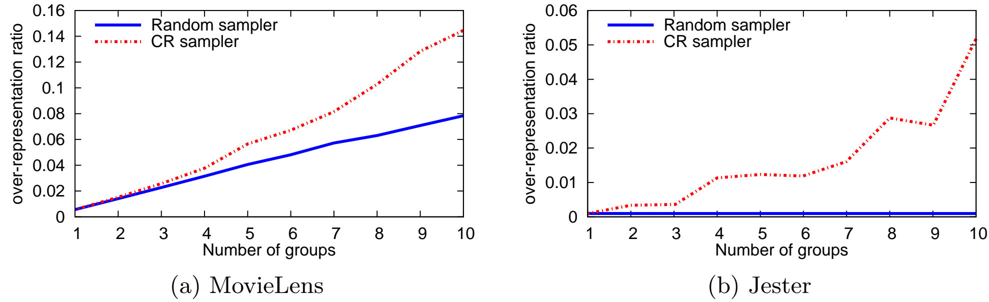
In DPPC, an attacker can observe a user's public profile and attempt to infer information about her *private* profile. Public profiles contain two types of information: a list of items, and their associated ratings. Both are influenced by the user's *private* profile, and can thus leak information about the user's *private* profile. We start by evaluating the amount of information that can be extracted from the list of items composing the public profiles, and in a second step we perform a similar evaluation about the ratings.

List of items in the public profiles Because the *public* profile of a user is an aggregation of the *private* profiles of similar users, items from the topics of interests of the user (to which



(a) MovieLens, groups=5, k=150 (b) Jester, groups=5, k=150

Figure 6.9: Impact of the THRESHOLD aggregator on the NMAE.



(a) MovieLens (b) Jester

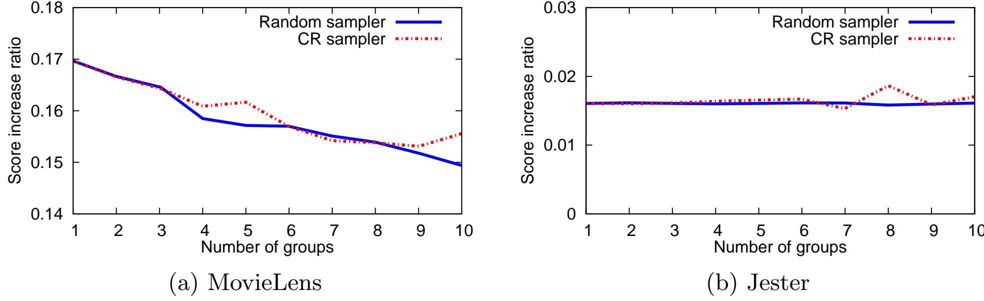
Figure 6.10: Over-representation of private items in the public profile.

items from the *private* profile belong) are more likely to be in the *public* profile of the user than the other items. We evaluate how more likely items from the *private* profile are to be in the *public* profile than the other items with the *over-representation* ratio. The *over-representation* ratio is the fraction of *private* items in the *public* profile over the fraction of other items in the *public* profile, minus one:

$$over - representation\ ratio = \frac{|private \cap public| \ / \ |private|}{|\overline{private} \cap public| \ / \ |\overline{private}|} - 1,$$

where $\overline{private}$ is the set of items that are not in the user's *private* profile. An *over-representation* ratio of 0 means that items from the *private* profile are not more likely than the others to appear in the *public* profile. In that case, the list of items present in the *public* profile reveals no information. If the *over-representation* ratio is 1, this means that items from the *private* profile are twice as likely to be in the *public* profile as the other items.

Figure 6.10 displays for both dataset the average *over-representation* ratio over all users. We make two observations in the MovieLens dataset: (i) the *over-representation* ratio increases with the number of groups; and (ii) it is higher with the CR sampler than with the RANDOM sampler. (i) happens because the number of different items in each repository decreases when the number of groups increases. Thus, the users' public profiles built from the repositories contain fewer items. However, the number of items from the *private* profiles included in the public profiles decreases slower than the item that are not from the *private* profile, which explains why the *over-representation* ratio increases. The reason for (ii) is that, as for the clustering quality, item repositories only contain items from their own group when using the CR sample. Thus, they



(a) MovieLens (b) Jester

Figure 6.11: Score increase ratio.

contain a small variety of items and some users can have no items they like in a given group, and thus have no item of this group in their public profiles. Here again, this affects more items that are not in the user's *private* profile.

With the MovieLens dataset, in the worst case scenario (CR sampler and $CR_{groups} = 10$), an item of a user's *private* profile is only 14% more likely to appear in the user's public profile than an item outside its *private* profile. With the RANDOM sampler, the *over-representation* ratio decreases to 8% (still in the worst case). With the Jester dataset, a similar trend is observed with the CR sampler, however with the RANDOM sampler the items from the *private* profile are not over-represented. This happens because of the high density of the jester dataset. On average, users rated 72 items over 100, and the *private* profiles of the users contain on average 57 items (we only consider the 80% ratings of the training set). With the random sampler, each repository contains most of the items, and the public profiles always contain all 100 items of the system.

The fact that the probability for an item from a user's *private* profile to be included into its public profile is only slightly larger than the one of random items shows that very little can be deduced from the list of items in the public profile of a user.

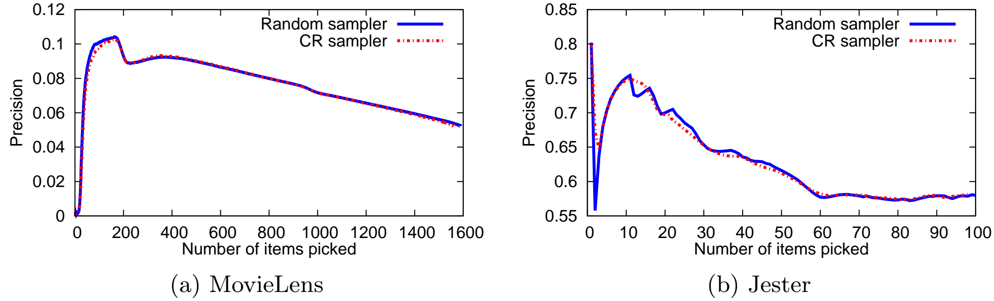
Ratings of items in the public profiles We now look into the difference of ratings in the *public* profiles between items from the *private* profiles and the other ones. Like for the *over-representation* ratio, we compute the *score increase* ratio for the item ratings as the average rating of items from the *private* profile over the average rating of the other items, minus 1. The *score increase* ratio for a user u is computed as follow:

$$score\ increase\ ratio_u = \frac{\sum_{i \in private_u \cap public_u} r_{ui} / |private_u \cap public_u|}{\sum_{i \in \overline{private_u} \cap public_u} r_{ui} / |\overline{private_u} \cap public_u|} - 1,$$

where r_{ui} is the rating of user u for item i.

Figure 6.11 shows the average *score increase* ratio over all users as a function of CR_{group}. Only the results for the RANDOM sampler are displayed as the results with the CR sampler were almost identical. We observe that with the MovieLens dataset, ratings of items from the *private* profile are on average between 15% and 17% higher than those of the other items. On the Jester dataset however, the difference of ratings is very small, less than 2%.

This difference in the ratings suggests that an attacker could guess which items are in the nodes *private* profile based on their rating. Since the items from the *private* profile have a higher rating, the natural method is to select the items with the highest rating. Based on this we designed an attack, in which the attacker selects the n items with the highest rating, and



(a) MovieLens (b) Jester

Figure 6.12: Precision of the rating attack.

guesses that they belong to the node's *private* profile. We then compute for each value of n the fraction of the selected items that actually are in the node's *private* profile, which we call the *precision* of the attack. Figure 6.12 shows the results for $CR_{groups} = 0.2$ (results were very similar for other values of CR_{group}). We observe that in the MovieLens dataset, the precision of the attack never exceeds 10%, which is only twice as good as randomly picking items from the set of all items. For the Jester dataset, we observe a similar behavior, except for a precision peak when the attacker only select the one or two items with the highest rating. It is important to keep in mind that in the Jester dataset, the users *private* profiles contain on average 57 items, thus guessing at random that a user rated a given item yields a precision of 0.57. In the best case (when selecting between 10 and 15 items), the precision of the attack is 0.75, which is only 30% more effective than random guessing.

7 Related work

To the best of our knowledge, the closest work to DPPC is P3 [AN11]. In P3, each user computes locally, based on her profile and some globally available information, which group or groups she belongs to. To each user group is associated a group-wise aggregator, which is a node in charge of aggregating the profiles of the users of its group. Once the users have computed the group they belong to, they send their profiles to their group-wise aggregator. As in DPPC, users do not want anyone to be able to access to their full profile, thus instead of directly sending their profile to the aggregator, they slice it into several samples that they send to different collectors, which are nodes in charge of relaying the pieces of profiles they receive to the aggregator. That way, the aggregator receives the full profile of each user, but does not know which pieces goes with which. Upon receiving the users profiles, the aggregator compute recommendations for the users based on the aggregate of all profiles. In addition, the aggregator can access a service external to the system to get recommendations for the group. These recommendations are then retrieved and processed by the users.

In DPPC we do not use collectors for the users to be able to send their full profiles to the item repositories, because it requires nodes to all send their profiles roughly at the same time. Otherwise, if users were to send their profiles samples at any time, an aggregator could easily guess that a set of samples received in a short period of time all belong to a single or few users. In addition, users periodically need to re-send their profiles to the aggregator. Thus, the aggregator can remember which samples it saw in the past and when a new user becomes part of the group the aggregator can detect the set of new samples of this user. The mechanism used by P3 is thus not very good in a situation in which we assume that people discover new

content over time, and thus potentially start belonging to new groups.

Another difference is that P3 and DPPC have different objectives. While P3 aims at recommending content to the users, the goal of DPPC is to provide a public profile through which users can perform similarity computation and select their k nearest neighbors. This can be used to provide recommendations, but it can also be used in a different context, such as collaborative content distribution [BFG+13] or collaborative caching [FGK14].

8 Perspectives

8.1 Correlation attack

The protection of the users' privacy provided by the CR sampler makes the assumption that the only way to link two profiles samples together is by finding items present in both samples. However, the ratings a user gives to two items can sometimes be highly correlated. For example, if the movie star wars episode IV is highly rated in a sample, it is likely that one of the other samples of the same profile contains the movie star wars episode V, also with a high rating. On the other hand, a sample containing the movie star wars episode I with a high rating is likely not to originate from the same profile. This could be used by an adversary to estimate how likely two profile samples are to originate from the same profile. Even though this attack requires a large amount of background knowledge to compute the correlations between items, it may be possible to perform it successfully at least on a subset of the users. This attack can be prevented by assigning items to groups based on their correlation with the other items of the different groups. This way, highly correlated items would be part of the same profile samples, and could not be used to link two samples together. We plan to evaluate the efficiency of this attack as well as the feasibility of the correlation-based CR sampler in the future.

8.2 Item-based recommendations

In DPPC, the *item* profiles are used to create the users' *public* profiles, which are used to compute the KNN of each user. Recommendations are then computed based on these KNNs. However, recommendations can be computed directly from the *item* profiles. For example, a user could use the profiles of the items she liked in place of the profiles from her KNN. This would remove the need for users to share their *public* profile and to compute their $k-nearest-neighbors$. We evaluated this approach with a naive algorithm, but the results were not satisfactory. However we believe that with a carefully designed algorithm, this method could provide similar or better recommendations than what can be achieved with DPPC.

9 Concluding remarks

We presented DPPC, a distributed protocol allowing users to compute their $k - nearest - neighbors$ in a privacy-preserving manner. This is achieved by creating for each user a *public* profile that is an aggregate of the profile of similar users. This *public* profiles, which do not contain the users' personal information, are then used to compute the similarity between users and select their $k - nearest - neighbors$. We evaluated the impact of DPPC on the quality of the KNN by comparing it to a classical, non privacy-preserving, clustering algorithms. We believe these results are encouraging, and this pushes us to improve DPPC on two main directions. First, we want to improve DPPC by protecting the users against the correlation attack described in Section 8.1. Second, we plan to evaluate the possibility of computing item-based recommendations in DPPC, as described in Section 8.2.

Chapter 7

Conclusion

Context

Over the past few years, personalization of content has become a crucial feature on the Internet. This comes from the fact that personalization help users navigating through the overflow of available information by filtering out the content that the users are not interested in. Personalization systems require to learn a big deal of data about users to provide an efficient service. Collecting such personal data can be explicit, by allowing the user to express her opinion (for example the like feature in Facebook), or implicit, by simply recording Web navigation, e.g. visited pages. The latter approach dominates for it does not require the collaboration of the users. However, this implies that users lose control over their data and that potentially sensitive information might be collected . This clearly raises a major privacy issue. For example, if a user visits pages on a medical website about a particular disease, it is likely that she or someone she knows suffers from this disease. From this observation to denying insurance to that user is a few feet away. In addition, personalization remains both costly, as it requires to store a large amount of data per user, and is computationally intensive. This cost is typically supported by the user, either in the form of advertisements displayed upon navigation, or by selling their personal data to other companies. (For example, a health insurance company could be interested to know if a user applying for an insurance frequently visits medical websites).

Peer-to-peer personalization systems offer an attractive alternative to centralized service solving many of the problems caused by the presence of a central authority. Typically, the cost of personalization is shared between users, whose machines are used to both store the data and compute the recommendations. They also allow users to fully control which data can be accessed by the system, and thus prevent sensitive data to be publicly exposed. They, however, only partially solve the privacy issues that personalization raises. Typically, other users participating to the system may have access to the user data. They can also suffer from manipulation, for example from malicious users trying to corrupt the system to promote their own content.

Summary of the contributions

In this thesis, we present four contributions improving P2P systems precisely on the two aforementioned points. We first propose, TAPS, a peer sampling protocol leveraging the existence of an underlying social network to provide users with trusted peers. Our second contribution improves on the first one by protecting the users' privacy with respect to their social network.

Our third contribution is a collaborative filtering system preserving the users' anonymity. Our last contribution is a distributed privacy-preserving clustering protocol that can provide personalized content to user without requiring them to disclose their personal information.

Trust-aware peer sampling P2P applications are typically unreliable in several contexts because of the difficulty to assert one's identity. For example in the context of an online social market, users might be reluctant to buy goods if they cannot have precise information regarding the sellers, or a mechanism to get in contact with them if the transaction goes wrong. TAPS provides users with information regarding the trustworthiness of the other users. In addition, it provides a path from friends to friends extracted from the underlying social network, allowing each user involved in a transaction to identify the other user if needed. We evaluated TAPS, and showed that, despite being distributed and having to cope with partial information, it provides almost optimal results in most of the situations.

Privacy-preserving TAPS Privacy is becoming an increasing concern for users of online applications. Moving from centralized to distributed applications gives back to the user the control over her data. However, personalization services precisely require such data to ensure the service. The TAPS protocol requires access to the user's social acquaintances, which can then be accessed by the other users participating in the system. We propose PTAPS, a privacy-preserving alternative to TAPS. PTAPS ensures that a user cannot discover who are the friends of another user except if this other user is one of her friends. Our evaluation shows that both TAPS and PTAPS achieve close to optimal performances in most of the situations.

Anonymous collaborative filtering While PTAPS improves TAPS by protecting the identity of the users' friends, FreeRec takes this one step further and hides the identity of the users themselves. In FreeRec all the interactions involving sensitive data, such as user profiles and k-nearest-neighbors, are done through an anonymization protocol ensuring that the users personal data cannot be linked back to them. We also provide an theoretical analysis of the comparison between fix length and variable length onion routing. This analysis showed that, in the context of FreeRec, the fixed length paths provide significantly better privacy to the users.

Distributed privacy-preserving clustering The PTAPS protocol preserves the users' privacy by hiding their social links from the other users, and FreeRec anonymize the user interactions. However they do not solve the problem of similarity computation between users. All user-based recommendation systems require to compute the similarity between pairs of users, and this typically requires the users to display publicly the content of their profiles for this computation to take place. We propose DPPC, a distributed protocol allowing users to compute their similarities in a privacy-preserving manner. This is achieved by creating for each user a public profile that does not contain her personal information but instead exploit the items to convey similarities between users. We show on real datasets that relying on the proposed public profiles enables to accurately compute their similarities.

Perspectives

In this thesis we presented four different contributions, and described possible extensions at the end of each chapter. We now present a more global perspective on possible future work.

Social Market In Chapter 3 we presented TAPS, a protocol introducing trust into gossip-based applications. The main application we had foreseen when we designed TAPS was the Social Market, a distributed social marketplace. The objective of the Social Market, as described with John's example in Chapter 3, is to help users to find other users who can sell them (or buy from them) some goods they are searching for. Along with locating such users, the Social Market aims at making transactions between users more secure by 1) estimating how trustworthy a buy or a seller is from a particular user's point of view, and 2) providing a way, once a transaction was executed, for one of the two parties of the transaction to contact and reach the other party through a sequence of users who trust each-other.

In its current state, TAPS already does a large part of this task by putting in contact similar and trustworthy users. However, it still lacks the possibility for a user to search for specific objects or to find buyers for an object she wants to sell. We believe that the work initiated on DPPC, presented in Chapter 6, can be extended to solve this problem. In DPPC, each item is associated with a repository in charge of aggregating the profiles of interested users. Repositories could be enhanced in two ways: 1) they could search and reference similar items, such that users can navigate from item to item, the same way they navigate from one book to related books in Amazon.com. 2) they could be used to reference users interested in buying or selling this item, so that buyers can easily find sellers. This can be done by using the privacy-preserving properties of PTAPS, such that no one can discover the true identity of a buyer or a seller until the time of the transaction.

Thus, a clear perspective to this work would be to integrate all of those contributions in a single framework.

Privacy evaluation The objective of PTAPS and DPPC is to preserve the users' privacy while they participate in a collaborative application. However, privacy is not perfectly preserved. The mere fact that some of their private data are used by the system means that they lose some privacy. An interesting direction for future work would be to evaluate, from a theoretical point of view, the amount of private information that is lost by the user by sharing her data with the system. Clearly, quantifying this loss of privacy is down the avenue.

Appendix A

DynaSoRe

In this appendix, we present DynaSoRe, an in-memory store for social network applications. This work, which took place during an internship at Yahoo! Research Barcelona from June to August 2012 in collaboration with Xiao Bai, Flavio Junqueira and Vincent Leroy, lead to a publication in the *Middleware 2013* conference. We decided to leave this contribution aside of the other contribution of this thesis as it takes place in a different context. However, we believe this contribution interesting and decided to append the associated paper in this appendix.

> Social network applications are inherently interactive, creating a requirement for processing user requests fast. To enable fast responses to user requests, social network applications typically rely on large banks of cache servers to hold and serve most of their content from the cache. In this work, we present DynaSoRe: a memory cache system for social network applications that optimizes data locality while placing user views across the system. DynaSoRe storage servers monitor access traffic and bring data frequently accessed together closer in the system to reduce the processing load across cache servers and network devices. Our simulation results considering realistic data center topologies show that DynaSoRe is able to adapt to traffic changes, increase data locality, and balance the load across the system. The traffic handled by the top tier of the network connecting servers drops by 94% compared to a static assignment of views to cache servers while requiring only 30% additional memory capacity compared to the whole volume of cached data.

1 Introduction

Social networking is prevalent in current Web applications. Facebook, Twitter, Flickr and Github are successful examples of social networking services that allow users to establish connections with other users and share content, such as status updates (Facebook), micro-blogs (Twitter), pictures (Flickr) and code (Github). Since the type of content produced across application might differ, we use in this work the term event to denote any content produced by a user of a social networking application. For this work, the format of events is not important and we consider each event as an application-specific array of bytes.

A common application of social networking consists of returning the latest events produced by the connections of a user in response to a read request. Given the online and interactive nature of such an application, it is critical to respond to user requests fast. Therefore, systems typically use an in-memory store to maintain events and serve requests to avoid accessing a persistent, often slower backend store. Events can be stored in the in-memory store in the

form of materialized views. A view can be producer-pivoted and store the events produced by a given user, or it can be consumer-pivoted and store the events to be consumed by a given user (e.g., the latest events produced by a user's connections) [STCR10]. We only consider producer-pivoted views in this work.

When designing systems to serve online social applications, scalability and elasticity are critical properties to cope with a growing user population and an increasing demand of existing users. For example, Facebook has over 1 billion registered and active users. Serving such a large population requires a careful planning for provisioning and analysis of resource utilization. In particular, load imbalance and hotspots in the system may lead to severe performance degradation and a sharp drop of user satisfaction. To avoid load imbalances and hotspots, one viable design choice is to equip the system with a mechanism that enables it to dynamically adapt to changes of the workload.

A common way to achieve load balancing is to randomly place the views of users across the servers of a system. This however incurs high inter-server traffic to serve read requests since the views in a user's social network have to be read from a large amount of servers. SPAR is a seminal work that replicates all the views in a user's social network on the same server to implement highly efficient read operations [PES+10]. Yet, this results in expensive write operations to update the large amount of replicated views. Moreover, SPAR assumes no bounds on the degree of replication for any given view, which is not practical since the memory capacity of each individual server is limited. Consequently, it is very important to make efficient utilization of resources.

In this work, we present DynaSoRe (Dynamic Social stoRe), an efficient in-memory store for online social networking applications that dynamically adapts to changes of the workload to keep the network traffic across the system low. We assume that a deployment of DynaSoRe comprises a large number of servers, tens to hundreds, spanning multiple racks in a data center. DynaSoRe servers create replicas of views to increase data locality and reduce communication across different parts of data centers. We assume a realistic tree structure for the networking substrate connecting the servers and aim to reduce traffic at network devices higher up in the network tree.

Our simulation results show that with 30% additional memory (to replicate the views), DynaSoRe reduces the traffic going through the top switch by 94% compared to a random assignment of views and by 90% compared to SPAR. With 100% additional memory, DynaSoRe only incurs 2% of the total traffic with the random assignment, significantly outperforming SPAR which incurs 35% of the traffic with the random assignment.

Contributions. In this paper, we make three main contributions:

- We propose DynaSoRe, an efficient in-memory store for online social applications, which dynamically adapts to changes of the workload to keep the network traffic across the system low.
- We provide simulation results showing that DynaSoRe outperforms our baselines, random assignment and SPAR, and that it is able to reduce the amount of network traffic across the system.
- We show that DynaSoRe is especially efficient compared to our baselines when assuming a memory budget, which is an important practical goal due to cost of rack space in modern data centers.

Roadmap. The remainder of this paper is structured as follows. We specify in Section 2 the model and requirements of an efficient in-memory store. We present the design of DynaSoRe

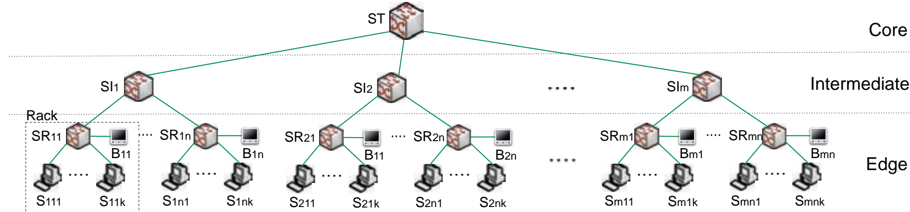


Figure A.1: System architecture of DynaSoRe.

in Section 3 and evaluate it in Section 4. We discuss related work in Section 5 and present our conclusions in Section 6.

2 Problem Statement

2.1 System model

DynaSoRe is a scalable and efficient distributed in-memory store for online social applications that enable users to produce and consume events. We assume that the social network is given and that it changes over time. The events users produce are organized into views, and the views are producer-pivoted (contain the events produced by a single user). A view is a list of events, possibly ordered by timestamps. DynaSoRe supports `read` and `write` operations. A `write` request from user u of an event e writes e to the view of u. A `read` request from user u reads the views of all connections in the social network of u. This closely follows the Twitter API. According to Twitter, status feeds represent by far the majority of the queries received[1]. Consequently, the benefits of DynaSoRe (that are only measured with the read/write operations) are significant even in a complete social application that also supports other kinds of operations.

DynaSoRe spans multiple servers, as the views of all users cannot fit into the memory of a single server. Distributing the workload across servers is critical for scalability. Our applications reside in data centers. Servers inside a data center are typically organized in a three-level tree of switches, which has a core tier at the root of the tree, an intermediate tier, and an edge tier at the leaves of the tree [AFLV08, GHJ$^+$09, HB09] as shown in Figure A.1. The core tier consists of the top-level switch (ST), which connects multiple intermediate switches. The intermediate tier consists of intermediate switches (SI) and each of them connects a subset of racks. The edge tier consists of racks and each rack is formed by of a set of servers connected by a rack switch (SR). The network devices, i.e., switches at different levels of the tree architecture, only forward network traffic. The views of users are maintained in the servers (S), connected directly to rack switches. The servers have a bounded memory capacity and we established its capacity by the number of views it can host. We use b to denote the number of bytes we use for a view. Brokers (B) are also servers connecting directly to rack switches and they are in charge of reading and writing views on the different servers of the data center.

Note that DynaSoRe could be deployed on several data centers by adding a virtual switch representing communications between data centers, which would then be minimized by DynaSoRe . In practice, Web companies such as Facebook do not have applications deployed across data centers. Instead, they replicate the content of each data center through a master/slave mechanism[NFG$^+$13]. Thus in this work we focus on the case of a single data center.

In the system, events are organized in views and stored as key-value pairs. Each key is a

[1] http://www.infoq.com/presentations/Twitter-Timeline-Scalability

user id and the value is the user view comprising events the user has produced. This memory store is back-ended by a persistent store that ensures data availability in the case of server crashes or graceful shutdowns for maintenance purposes. We focus in this work on the design of the memory store and a detailed discussion of the design of the persistent store is out of scope.

2.2 Requirements

To provide scalable and efficient `write` and `read` operations, DynaSoRe provides the following properties: locality, dynamic replication, and durability.

Locality. A user can efficiently read or write to a view if the network distance between the server executing the operation and the server storing the view is short. As one of our goals is to reduce the overall network traffic, we define the network distance between two servers to be the number of network devices (e.g., switches) in the network path connecting them. DynaSoRe ideally ensures that all the views related to a user (i.e., her own view and the views of her social connections) are placed close to each other according to network distance so that all requests can be executed efficiently. This requires flexibility with respect to selecting the server that executes the requests and the servers storing the views.

Dynamic replication. Online social networks are highly dynamic. The structure of the social network evolves as users add and remove social connections. User traffic can be irregular, with different daily usage patterns and flash events generating a spike of activity. Adapting to the behavior of users requires a mechanism to dynamically react to such changes and adjust the storage policy of the views impacted, for both number and placement of replicas. One goal for DynaSoRe is to trace user activity to enable an efficient utilization of its memory budget through accurate choices for the number and placement of replicas. Such a replication policy needs also to consider load balancing and to satisfy the capacity constraints of each server.

Durability and crash tolerance. We assume that servers can crash. Missing updates because of crashes, however, is highly undesirable, so we guarantee that updates to the system are durable. To do this, we rely upon a persistent store that works independently of DynaSoRe. Updates to the data are persisted before they are written to DynaSoRe to guarantee that they can be recovered in the presence of faulty DynaSoRe servers. Since we replicate some views in our system, copies of data might be readily available even if a server crashes. In the case of a single replica, we need to fetch data from the persistent store to build it. In both cases, single or multiple replicas, crashes additionally require live servers to dynamically adjust the number of replicas of views to the new configuration.

2.3 Problem formulation

Given the system model and requirements, our objective is to generate an assignment of views to servers such that (i) each view is stored on at least one server and (ii) network usage is minimized. The first objective guarantees that any user view can be served from the memory store. We eliminate the trivial case in which the cluster does not have the storage capacity to keep a copy of each view. DynaSoRe is free to place views on any server, as long as it satisfies their capacity constraints. Any available space can be used to replicate a view and optimize the second objective. We define the amount of extra memory capacity in the system as follows : Given V the set of views in the system, and b the amount of memory required to store a single view, the system has $x\%$ extra memory if its total memory capacity is $(1 + x/100) \times |V| \times b$.

To reduce network traffic, we need to assign views to servers such that it reduces the number of messages flowing across network devices. Note that a message between servers reaching the top switch also traverses two intermediate switches and two rack switches. Consequently, minimizing the number of messages going through the top switch is an important goal to reduce network traffic.

We show in Section 4 that DynaSoRe is able to dynamically adapt to workload variations and to use memory efficiently. In this work, we focus on the mechanisms to distribute user views across servers and on the creation and eviction of their replicas. Although important, fault tolerance is out of the scope of this work and we discuss briefly how one can tolerate crashes in Section 3.3.

3 System Design

In this section, we present the design of DynaSoRe. We first present its API, followed by the algorithm we use to make replication decisions. We end this section with a discussion on some software design issues.

3.1 API

DynaSoRe is an in-memory store used in conjunction with a persistent store. The API of DynaSoRe matches the one used by Facebook for memcache [NFG+13]. It consists of a `read` request that fetches data from the in-memory store, and a `write` request that updates the data in the memory store using the persistent store. Consequently, DynaSoRe can be used as a drop-in replacement of memcache to cache user views and generate social feeds.

`Read(u, L):` u is a user id and L is a list of user ids to read from. For each id u' in L, it returns view(u').

`Write(u):` u is a user id. It updates view(u) by fetching the new version from the persistent store.

3.2 Algorithm

3.2.1 Overview

DynaSoRe is an iterative algorithm that optimizes view access locality. DynaSoRe monitors view access patterns to compute the placement of views and selects an appropriate broker for executing each request. Specifically, DynaSoRe keeps track, for each view, the rates it is read and written, as well as the location of brokers accessing it. When DynaSoRe detects that a view is frequently accessed from a distant part of the cluster, consuming large amounts of network resources, it creates a replica of this view and places it on a server close to those distant brokers. This improves the locality of future accesses and reduces network utilization. Similarly, when a broker executes a request, DynaSoRe analyzes the placement of the views accessed, and selects the closest broker as a proxy for the next instance of this operation.

3.2.2 Routing

DynaSoRe optimizes view access locality by placing affine views on servers that are close according to network distance, and replicating some of the views on different cluster sub-trees to further improve locality. Using such tailored policies for view placement requires a routing layer to map the identifiers of requested views to the servers storing them.

Brokers. Each request submitted to DynaSoRe is executed by a broker. A request consists of a user identifier and an operation: `read` or `write`. DynaSoRe creates, for each user, a read proxy and a write proxy, each of them being an object deployed on a broker. The motivation of using two different proxies per user stems from the fact that they access different views. The write proxy updates the view of a user, while the read proxy reads the views of a user's social connections. These views may be stored in different parts of the cluster. Allowing DynaSoRe to select different brokers gives it more flexibility and impacts network traffic. The mapping of proxies to brokers is kept in a separate store and is fetched by the front-end as a user logs in. Once a front-end receives a user request, it sends it to the broker hosting the proxy for execution.

Routing policy. When multiple servers store the same view, the routing layer needs to select the most appropriate replica of the view for a given request. The routing policy of DynaSoRe favors locality of access. Following the tree structure of a cluster, a broker selects, among the servers storing a view, the closest one, i.e. the one with which it shares the lowest common ancestor. This choice reduces the number of switches traversed. When two replicas are at equal distance, the broker uses the server identifier to break ties.

Routing tables. The write proxy of a user is responsible for updating all the replicas of her view and for storing their locations. Whenever a new replica of the view is created or deleted, the write proxy serves as a synchronization point and updates its list of replicas accordingly.

The read proxy of a user is in charge of routing her read requests. To this end, each broker stores in a routing table, for every view in the system, the location of its closest replica according to the routing policy described earlier. The routing table is shared by all the read proxies executed on a given broker. The write proxy of a view is also responsible for updating the routing tables whenever a view is created or deleted. As the routing policy is deterministic, only brokers affected by the change are notified.

Servers also store some information about routing. Each view stores the location of its write proxy, so that a server may notify a proxy in case of an eviction or replication attempt of its replica. When several replicas of a view exist, each replica also stores the location of the next closest replica. Both information are used to estimate the utility of a view that will be described in Section 3.2.4.

Proxy placement. To reduce the network traffic, the proxies should be as close as possible to the views they access. Whenever a request is executed, the proxy uses the routing table to obtain the location of the views and execute the operation. As a post-processing step, the proxy analyzes the location of these views and computes a position that minimizes the network transfers. Starting at the root of the cluster tree, the proxy follows, at each step, the branch from which most views were transferred, until it reaches a broker. If the obtained broker is different from the current one, the proxy migrates to the new broker for the next execution of this request. In the case of a write proxy, this migration involves in sending notification messages to view replicas.

3.2.3 Access statistics

To dynamically improve view access locality, DynaSoRe gathers statistics about the frequency and the origin of each access to a view. This information is stored on the servers, along with the view itself. The origin of an access to a view is the switch from which the request accessing this view comes. Consequently, two brokers directly connected to the same switch correspond

to the same origin. The writes to a given view are always executed in the location of its write proxy. However, reads can originate from any broker in the cluster. This explains why their origins should be tracked.

To reduce the memory footprint of access recording, DynaSoRe makes the granularity coarser as the network distance increases. Considering a tree-shaped topology, a server records accesses originating from all the switches located between the server and the top switch, as well as their siblings. For example, in Figure A.1 the server S_{111} records the accesses originating from the switches SR_{11} (the accesses from the local broker) to SR_{1n} and from SI_2 to SI_m instead of an individual record for every switch. In this way, in a cluster of m intermediate switches and n rack switches per intermediate switch, every replica records maximum $m-1+n$ origins instead of $m \times n$ origins. While significantly reducing the memory footprint, this solution does not affect the efficiency of DynaSoRe. The algorithm still benefits from precise information in the last steps of the convergence, and relies on aggregated statistics over sub-trees for decisions about more distant parts of the cluster.

DynaSoRe is a dynamic algorithm that is able to react to variations in the access patterns over time. We use rotating counters to record the number of accesses to views. Each counter is associated to a time period, and servers start updating the following counter at the end of the period. For example, to record the accesses during one day with a rotating period of one hour, we can use 24 counters of 1 byte. The number of counters, their sizes and their rotating periods can be configured depending on the reactivity we expect from the system, the accuracy of the logs and the amount of memory we can spend on it. It is possible to compress these counters efficiently. For instance, one may decrease the probability of logging an access as the counter increases to account for more accesses on 1 byte. One may also store these counters on the disk of the server to enable asynchronous updates of the counters. These optimizations are out of the scope of this work, and in the remainder of this paper we assume that the size of the counters is negligible with respect to the size of the views.

3.2.4 Storage management

A DynaSoRe server is a in-memory key-value store implementing a memory management policy. A server has a fixed memory capacity, expressed as the number of views it can store. DynaSoRe manages the servers as a global pool of memory, ensuring that the view of each user is stored on at least one server. Each server stores several views, some of them being the only instance in the system, while others are replicated across multiple servers and therefore optional. The objective of DynaSoRe is to select, for each server, the views that will minimize network utilization, while respecting capacity constraint. We assume that the events generated by users have a fixed size, such as those of Twitter (140 characters). Heavy content (e.g., pictures, videos, etc.) are usually not stored in cache but in dedicated servers.

View utility. Each server maintains read and write access statistics for the views it stores, as described in Section 3.2.3. Using these statistics, DynaSoRe can evaluate the utility of a view on a given server, i.e., the impact of storing the view on this server in terms of network traffic. DynaSoRe uses the statistics about the origins of read requests to determine which of them are impacted by the view. It then computes the cost of routing them to the next closest replica instead of this server, which represents the read gains of storing the view on the server. The traffic generated by write requests represents the cost of maintaining this view, and is subtracted from the read gains to obtain the utility. The details of the utility computation are presented in Algorithm 8. The utility of a view is positive if its benefits in terms of read requests locality outweighs the cost of updating it when write requests occur. The goal of DynaSoRe is to optimize network utilization. Hence, views with negative utility are automatically removed.

Algorithm 8 Estimate Profit

```
1: function Estimate Profit(logs, server, nearest)
2:     serverReadCost ← 0
3:     nearestReadCost ← 0
4:     for all < source, reads >∈ logs.reads() do
5:         serverReadCost + = reads · cost(source, server)
6:         nearestReadCost + = reads · cost(source, nearest)
7:     end for
8:     serverWriteCost ← writes · cost(broker, server)
9:     serverProfit ← nearestReadCost − serverReadCost − serverWriteCost
10:    return serverProfit
11: end function
```

Replication of views. Servers regularly update the utility of the views they store and use this information to maintain an admission threshold so that a sufficient amount (e.g., 90%) of their memory is occupied by views whose utility is above the admission threshold. If less memory is used, the admission threshold is 0. These admission thresholds are disseminated throughout the system using a piggybacking mechanism. Each broker maintains the admission threshold of the servers located in its rack, and transmits the lowest threshold to other racks upon accessing them. Thus, each server receives regular updates containing the lowest access threshold in other racks. A replica of a view on a given server serves either the brokers of the whole cluster, when this is the unique replica, or the brokers of a sub-tree of the cluster, when multiple replicas exist. Upon receiving a request for a view, a server updates its access statistics and evaluates the possibility of replicating it on another server of this sub-tree. This procedure is detailed in Algorithm 9. The utility of the replica is computed by simulating its addition on one of the servers, following the approach described previously. If the utility exceeds the admission threshold of the server, a message is sent to the write proxy of the view to request the creation of a replica.

When no replicas can be created, the server attempts to migrate the view to a more appropriate location. The computation of the utility of the view at the new location is slightly different from the replication case, since it assumes the deletion of the view on the current server and therefore generates higher scores. Algorithm 10 details this procedure. The migration of the view is subject to the admission threshold. Using the admission threshold avoids the migration of views rarely accessed to servers with high replication demand.

Eviction of views. To easily deploy new views on servers, DynaSoRe ensures that each server regularly frees memory. When the memory utilization of a server exceeds a given threshold (e.g., 95%), a background process starts evicting the views that have the least utility. Views that have no other replica in the system have infinite utility and cannot be evicted. Since multiples servers could try to evict the different replicas of the same view simultaneously, DynaSoRe relies on the write proxy of the view as a synchronization point to ensure at least one replica remains in the system. Servers typically manage to evict a sufficient amount of views to reach 95% capacity. One exception happens when the full DynaSoRe cluster reaches it maximum capacity, in which case there is no memory left for view replication. This proactive eviction policy decouples the eviction of replicas from the reception of requests, thus ensuring that memory can be freed at any time even when some replicas do not receive any requests.

Algorithm 9 Evaluate Creation of Replica

```
1: procedure Evaluate Creation of Replica(logs)
2:     newReplica ← ∅
3:     bestProfit ← 0
4:     for all < source, reads >∈ logs.reads() do
5:         profit ← Estimate Profit(logs, source, this)
6:         threshold ← Admission threshold(source)
7:         if profit > threshold & profit > bestProfit then
8:             newReplica ← Least loaded server(source)
9:             bestProfit ← profit
10:        end if
11:    end for
12:    if newReplica ≠ ∅ then
13:        Send(newReplica) to broker
14:    end if
15: end procedure
```

Algorithm 10 Compute Optimal Position of Replica

```
1: procedure Compute Optimal Position of Replica(logs)
2:     nearest ← nearest replica
3:     bestPosition ← this
4:     bestProfit ← Estimate Profit(logs, this, nearest)
5:     for all < source, reads >∈ logs.reads() do
6:         profit ← Estimate Profit(logs, source, nearest)
7:         threshold ← Admission threshold(source)
8:         if profit > bestProfit & profit > threshold then
9:             bestPosition ← Least loaded server(source)
10:            bestProfit ← profit
11:        end if
12:    end for
13:    if bestProfit < 0 then
14:        Send(removeThis) to broker
15:    else
16:        if bestPosition ≠ this then
17:            Send(bestPosition, removeThis) to broker
18:        end if
19:    end if
20: end procedure
```

3.3 Software Design

3.3.1 Durability

DynaSoRe complies with the architecture of Facebook, and relies on the same cache coherence protocol [NFG+13]. When a user writes an event, this command is first processed by the persistent store to generate the new version of the view of a user. The persistent store then notifies DynaSoRe by sending a `write` request to the write proxy of the user, which fetches the new version of the view from the persistent store and updates the replicas. The persistent store logs `write` requests before sending them, so they can be re-emitted in case of a crash. If a server crashes, the views can be safely recovered from the persistent store. Also, frequently accessed views are likely to be already replicated in the memory of other servers, allowing faster recovery and avoiding cache misses during the recovery process. We have chosen this design because memory is limited, and replicating frequently accessed views leads to higher performance compared to replicating rarely accessed views for faster recovery. However, if a large amount of memory is available, DynaSoRe can also be configured to keep multiple replicas

of each view on different servers. In that case, the threshold for infinite utility is set to the minimum number of replicas and recovery is fully performed from memory. The state of brokers and the location of the proxies of users are persisted in a high performance disk-based write-ahead log such as BookKeeper [JKR13], so that the setup of DynaSoRe is also recoverable.

3.3.2 Cluster modification

The configuration of the cluster on top of which DynaSoRe is running may change over time. For example, the number of servers allocated to DynaSoRe can grow as the number of users increase. There are three different ways a server can be added to the system:

1. The additional server is added into an existing rack. In this case, the new server will become the least loaded server in the rack, and all the new replicas deployed into this rack are stored in this new server until it becomes as loaded as the other servers in the rack.
2. A new rack is added below an existing intermediate switch. The same reasoning for the previous case applies here. The new rack is automatically used to reduce the traffic of the top router.
3. A new branch is added to the cluster by adding a new intermediate switch. In this case, DynaSoRe has no incentive to add data to the new servers since no requests will originate from there. When adding a new branch to the data center, we consequently need to move some views and proxies onto the new servers to bootstrap it. This procedure is, however, not detailed in this paper.

Removing servers on the other hand requires the views hosted by the servers to be relocated. Before removing a server, the views that have no other replica should be moved to a near server. The views that exist on multiple servers can simply be deleted as DynaSoRe will recreate them if needed.

3.3.3 Managing the social network

As described earlier, DynaSoRe does not maintain the social network, and instead receives the list of users from which data need to be retrieved when executing a `read` request. Consequently, the only direct impact of a modification to the social network is the modification of the list of views accessed by reads. The addition of a link between users u_1 and u_2 increases the probability to have either u_2's view replicated near u_1's read proxy, or u_1's read proxy migrated closer to a replica of u_2's view. DynaSoRe adapts to the modifications to the social network transparently, without requiring any specific action. When a new user enters the system, DynaSoRe needs to allocate a read proxy, a write proxy, and a view on a server for this user. The server chosen is the least loaded one at the time of the entrance of the user, and the two proxies are selected to be as close as possible to this server.

4 Evaluation

4.1 Baseline

Random In-memory storage systems, such as Memcached[2] and Redis[3], rely on hash functions to randomly assign data to servers. This configuration is static in the sense that it is not affected by the request traffic. For this scheme, the proxies of a user are deployed on the broker located in the same rack in which the user view is located. This is the simplest baseline we

[2] http://memcached.org/
[3] http://redis.io/

compare against, as it ignores the topology of the data center, the structure of the social graph, and does not leverage free memory through replication.

METIS Graph data can be statically partitioned across servers using graph partitioning. This leverages the clustering properties of social graphs and increases the probability that social friends are assigned to the same sever. We rely on the METIS[4] library to generate partitions, and randomly assign each of them to a server. The read and write proxies of users are deployed on the broker located in the rack hosting their view. This solution does not take into account the hierarchy of the cluster, and does not perform replication. It also does not handle modifications to the social graph, and needs to re-partition the whole social graph to integrate them.

Hierarchical METIS We improve the standard graph partitioning to account for the cluster structure. We first generate one partition for each intermediate switch, and then recursively re-partition them to assign views to rack switches and then servers. Compared to directly partitioning across servers, this solution significantly reduces the network distance of views of social friends assigned to different servers.

SPAR SPAR [PES+10] is a middleware that ensures the views of the social friends of a user are stored on the same server as her own view. SPAR assumes that it is always possible to replicate a view on a server, without taking into account memory limitations. We adapt SPAR to limit its memory utilization. The views of the friends of a user are copied to her server as long as storage is available. When the server is full, these views are not replicated. Similarly to the graph partitioning case, the proxies of a user are located in the rack hosting her view.

4.2 Datasets

4.2.1 Social graphs

We evaluate the performance of DynaSoRe by comparing it against our baselines on three different social networks (summarized in Table A.1):

- a sample from the Twitter social graph from August 2009 [CHBG10]
- a sample from the Facebook social graph from 2008 [WBS+09]
- a sample from the LiveJournal social graph [BHKL06]

	# users	# links
Twitter	1.7M	5M
Facebook	3M	47M
LiveJournal	4.8M	69M

Table A.1: Number of users and links in each dataset

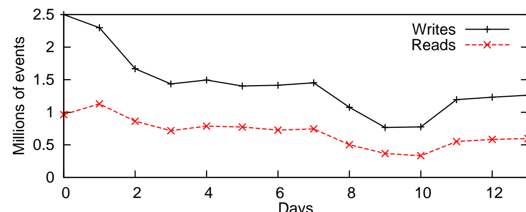


Figure A.2: Number of reads and writes in the Yahoo! News Activity dataset

[4]http://glaros.dtc.umn.edu/gkhome/views/metis

4.2.2 Request log

In this section, we rely on two different kinds of request logs for our experiments, a synthetic one and a real one. The real one is obtained from Yahoo! News. We discuss the logs we used in more details below.

Synthetic logs. Huberman et al. [HRW09] argue that the read and write activity of users in social networks is proportional to the logarithm of their in and out degrees in the social graphs. Silberstein et al. [STCR10] observe that there are approximately 4 times more reads than writes in a social system. Using this information, we create a random traffic generator matching these distributions and obtain, for each social graph, a request log. We additionally assume that each user issues on average one write request per day and that requests are evenly distributed over time. Compared to real workloads, these synthetic workloads show low variation, which enables DynaSoRe to accurately estimate read and write rates.

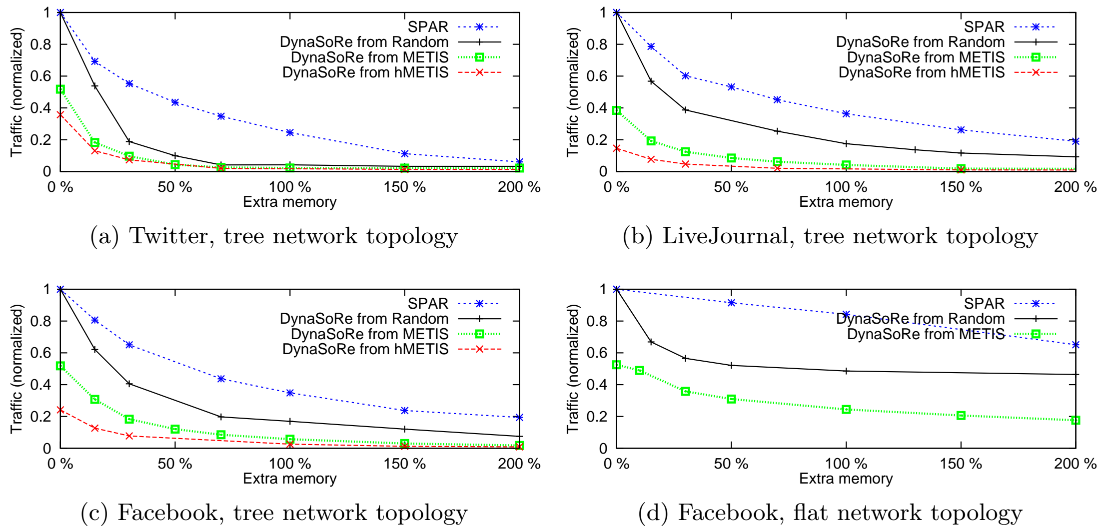
Real user traffic. Yahoo! News Activity is a proprietary social platform that allows users to share (`write`) news articles, and view the articles that their Facebook friends read (`read`). We use a two-week sample of the Yahoo! News Activity logs as a source of real user traffic in the experiments. We focus in this experiment on users who performed at least one read and one write during the two weeks. This selection results in a dataset of 2.5M users with 17M writes and 9.8M reads. Figure A.2 depicts the distribution of read and write activities per day. Users can consult the activity of their friends both on the Yahoo! website, or on Facebook. In the latter case, the reads are not processed by the Yahoo! website and do not appear in the log, which explains the prevalence of writes in our dataset. Because we do not have access to the Facebook social graph, we map the users of Yahoo! News Activity to the users in the Facebook social graph presented in Section 4.2.1. We rank both lists of users according to their number of writes and their number of friends, respectively, and connect users with the same rank. Because the Facebook social graph has more users, we only consider the first 2.5 million users according to the number of friends.

4.3 Simulator and cluster configuration

We implement a cluster simulator in Java to evaluate the different view management protocols on large clusters. The simulator represents all the servers and network devices in order to simulate their message exchanges and measure them. The virtual data center used in our experiment is composed of a top switch, 5 intermediate switches, each connected to 5 rack switches, for a total of 25 racks containing 10 machines each. In every rack, 1 machine is broker while the 9 others are servers to store views. Servers keep view access logs using a sliding counter of 24 slots shifted every hour. After each shift, the replica utility is recomputed and the server's admission threshold is updated. Each server has the same memory capacity, and the total memory capacity is a parameter of each simulator run. Finally, we assume that each application message, i.e., `read`, `write` request and their answer, is 10 times longer than a protocol message. In fact, most protocol messages do not carry any user data and are therefore much smaller.

4.4 Initial data placement and performance

The random placement and graph partitioning approaches produce static assignments of views to servers, which persists during the whole experiment. SPAR places views as the structure of the social network evolves. We first create one replica for each user, and we simulate the



(a) Twitter, tree network topology

(b) LiveJournal, tree network topology

(c) Facebook, tree network topology

(d) Facebook, flat network topology

Figure A.3: Top switch traffic with varying memory capacity

addition of all the edges of the social graph to obtain the its view placement. Once the memory of all servers has been used, the view layout remains constant. For DynaSoRe, the system is deployed on an existing social platform and uses this configuration as an initial setup. It then modifies this initial view placement by reacting to the request traffic.

When initializing DynaSoRe we consider three different view placement strategies: Random, METIS and hierarchical METIS (hMETIS). Using the synthetic request log, we evaluate the performance of each system after convergence, i.e., once the content of the servers stabilizes. Figures A.3a, A.3b and A.3c depict the traffic of the top switch for the 3 different social graphs. The traffic is normalized with respect to the traffic of Random. On the x-axis, we vary the extra memory capacity of the cluster. $x = 0\%$ means the capacity matches exactly the space required to store all the views without replication. With $x = 100\%$ memory capacity doubles, so the algorithms can replicate views up to 2 times on average.

Considering the initial data assignment ($x = 0\%$), we can clearly see that graph partitioning approaches (METIS and hMETIS) outperform Random data placement. Furthermore, hierarchical partitioning leads to a two-fold improvement over standard clustering. These results are expected: partitioning increases the probability that views of social friends will be stored on the same rack, which reduces the traffic of the top switch upon accessing them. hMETIS further improves this result by taking into account the hierarchy of the cluster. Thus, when the views of 2 friends are not located on the same rack, they are likely to be communicated through an intermediate switch rather than the top switch.

As we increase the memory capacity, both DynaSoRe and SPAR are able to replicate and move views. Yet, the results indicate that DynaSoRe is much more efficient than SPAR for using the available memory space. For example, in the case of Twitter, with 30% memory in addition to the amount of data stored, SPAR reduces the traffic by 42% compared to a Random, while DynaSoRe reduces it by 80%. These figures also demonstrate the importance of the initial data placement in the case of DynaSoRe. As DynaSoRe relies on heuristics to place views in the cluster, a good initial placement allows it to converge to better overall configurations, while a random placement converges to slightly worse performance. As the amount of available

memory further increases, the performance of DynaSoRe converges and part of the memory remains unused. Indeed, DynaSoRe detects that replicating some views does not provide an overall benefit, since the cost of writing to the extra replicas outweighs the benefits of reading them locality, which induces higher network traffic.

Table A.2 and Table A.3 present the average switch traffic at the top, intermediate, and rack levels for two memory configurations. We normalize the traffic value by the equivalent switch traffic using Random. DynaSoRe is initialized using hMETIS. Note that network traffic drops more significantly for the top switch which is the most loaded with Random. As fewer requests access different racks, rack switches also benefit from DynaSoRe, but to a lesser extent. Comparing absolute values, the traffic of the top switch almost drops to the level of a rack switch. Ultimately, DynaSoRe is able to relax the performance requirements for top and intermediate switches.

Figure A.4 shows the traffic on the top switch for the Facebook graph using the real user traffic extracted from Yahoo! News Activity. For space reasons, we only display the performances achieved by SPAR and DynaSoRe starting from the placement generated by Random and METIS with 50% extra memory. The figure shows the evolution of the traffic over time, and we can see that the traffic on the top switch follows the request pattern observed in Figure A.2. This figure shows that DynaSoRe is able to converge to an efficient view placement configuration, even in the case with high variance traffic. DynaSoRe still clearly outperforms the baseline, confirming the results obtained with the synthetic logs. Our results (not shown here for space reasons) show that the performance of DynaSoRe is consistently better (3 times when starting from Random, 9 times when starting from METIS) than Random independently of the traffic variation, confirming the robustness of DynaSoRe under high traffic.

4.5 Behavior in flat network topologies

The results presented above assume a tree topology for the network of the data center. This setup is common in data centers, hence DynaSoRe was specifically tailored for it. For the sake of fairness (as the baselines are designed without considering any network topology of data centers), we also evaluate DynaSoRe on a flat network topology. In this case, all of the 250 servers act as both caches and brokers, and are directly connected to a single switch. This configuration is similar to the one used to evaluate SPAR in [PES+10]. Figure A.3d shows that the performances of DynaSoRe and SPAR on the Facebook social graph using the synthetic request logs. Given that DynaSoRe was specifically tailored to tree topologies, the performance gap between DynaSoRe and SPAR is not as large as that presented in Figure A.3c. DynaSoRe still clearly outperforms SPAR, in particular in the configurations of low memory, thanks to its better replication policy. In the remainder of the evaluation, we focus on the tree network topology.

	Fbook	Twitter	LiveJ
Top switch DynaSoRe	.07	.06	.04
Top switch SPAR	.65	.55	.60
Inter switch DynaSoRe	.14	.11	.08
Inter switch SPAR	.77	.61	.70
Rack switch DynaSoRe	.60	.59	.57
Rack switch SPAR	.94	.84	.90

Table A.2: Switch traffic, 30% extra memory

	Fbook	Twitter	LiveJ
Top switch DynaSoRe	.01	.01	.01
Top switch SPAR	.24	.11	.26
Inter switch DynaSoRe	.03	.02	.02
Inter switch SPAR	.39	.13	.37
Rack switch DynaSoRe	.54	.53	.53
Rack switch SPAR	.77	.60	.75

Table A.3: Switch traffic, 150% extra mem.

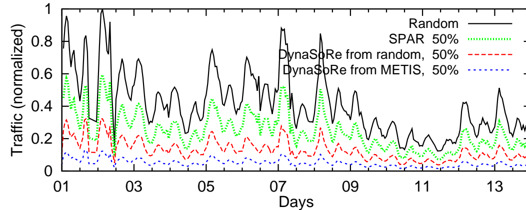


Figure A.4: Top switch traffic with Yahoo! News Activity requests, Facebook

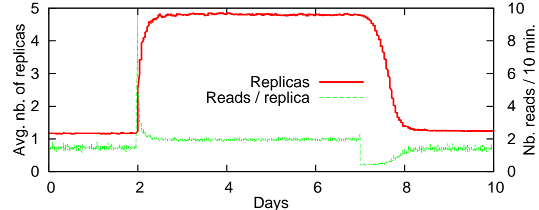


Figure A.5: Flash event : Addition of 100 followers, Facebook, 30% extra memory

4.6 Flash Events

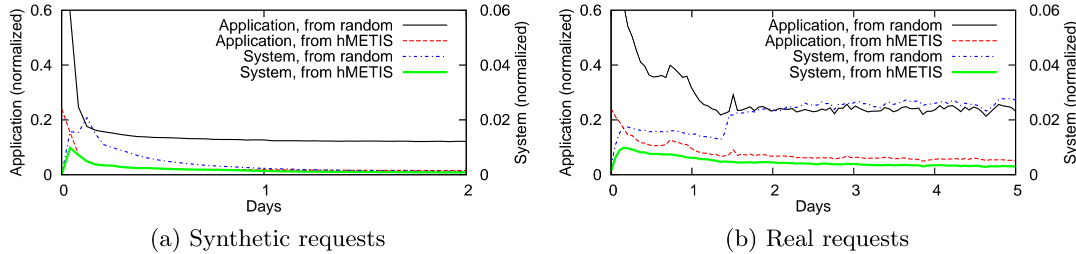
Online social networks are often subject to flash events, in which the activity of a subset of users suddenly spikes. To evaluate the reactivity of DynaSoRe, we simulate a sudden increase of popularity of some users, and measure the evolution of their number of replicas and the number of requests each of them processes. More precisely, at time $t = 2$ days in the simulation run, we randomly select a user and make this user popular by adding 100 random followers to read her view. Five days later ($t = 7$ days) all these additional followers are removed. We repeat this experiment 100 times on the Facebook dataset with a 30% extra memory capacity. We present the average results in Figure A.5.

At the beginning of the experiment, the user is not particularly active, and has 1.15 replicas on average. As new followers arrive, DynaSoRe detects that the number of reads of the view increases, and starts replicating it on other servers. DynaSoRe stabilizes in a configuration close to 5 replicas for this view. Given that the users reading this view are selected at random, they originate from all racks of the cluster and DynaSoRe generates a replica per intermediate switch. After replication, the average number of reads per replica is very close to the initial situation. The utility of replicas is high enough to be maintained by the system, but additional replicas do not pass the admission threshold. At the end of this period, the number of reads per replica drops sharply. DynaSoRe is able to detect and adjust the utility of the replicas, which leads to their eviction before the end of the following day. These results illustrate DynaSoRe's ability to react quickly to flash events, and evict replicas once they become useless.

4.7 Convergence time

SPAR and the three static approaches to assign views only require the social graph to determine the assignment of views. They do not react dynamically to traffic changes, and consequently, they do not require any time to converge as long as the social graph is stable. For DynaSoRe, however, it is important to evaluate the convergence time, using both stable synthetic traffic and real traffic. Before converging to a stable assignment, DynaSoRe replicates views regularly. This traffic of replicas generates messages that also consume network resources. Once the system stabilizes, the overhead of system messages becomes negligible.

Figure A.6a shows the traffic of the top switch when running DynaSoRe on the Facebook social graph using synthetic traffic with an extra memory of 150%. We separate the application traffic and the system traffic to study the convergence of DynaSoRe over time (x axis). After a few hours of traffic, DynaSoRe has almost reached its best performance, starting both from a random placement and from a placement based on graph partitioning. The amount of system messages sent by the protocol rapidly drops and reaches its minimum after one day. Note that less memory capacity makes the time to converge shorter, since DynaSoRe performs fewer replication operations. Figure A.6b displays the results of the same experiments executed using



(a) Synthetic requests

(b) Real requests

Figure A.6: Top switch traffic over time, Facebook, 150% extra memory

the real request trace from Yahoo! News Activity. As the workload presents more variation, DynaSoRe does not fully converge, and the system traffic remains at a noticeable level as views are created and evicted. The request rate of the real workload is lower than the synthetic one, which explains the slower convergence: DynaSoRe is driven by requests. Initializing DynaSoRe using graph partitioning, however, induces an initial state that is more stable, allowing the system traffic to remain low. Despite slower convergence, the application traffic still reaches its best performance after one day.

5 Related Work

DynaSoRe enables online data placement in in-memory store for social networking applications. We review in this section related work on in-memory storage systems, offline data placement algorithms, online data placement algorithms, and discuss their differences with DynaSoRe.

In-memory storage RAMCloud [ORS+11] is a large-scale in-memory storage system that aggregates the RAM of hundreds of servers to provide a low-latency key-value store. RAMCloud does not currently implement any data placement policy, and could benefit from the algorithms used in DynaSoRe. RAMCloud recovers from failures using a distributed log accessed in parallel on multiple disks. This is similar to write-ahead logging approach described in Section 3.3.1 and could be also used in DynaSoRe.

Offline data placement Curino et al. describe Schism [CJZM10], a partitioning and replication approach for distributed databases to minimize the amount of transactions executed across multiple servers. Schism uses an offline standard graph partitioning algorithms on the request log graph to assign database tuples to servers. DynaSoRe is an online strategy, creating and placing views dynamically. As a consequence, it is much easier to react to changes in access patterns that frequently occur in social applications. DynaSoRe benefits from graph clustering techniques similar to those used in Schism to generate more effective initial placement of views for faster convergence to ideal data placement.

Zhong et al. consider the case of object placement for multi-object operations [ZSS08]. Using linear programming, they place correlated objects on the same nodes to reduce the communication overhead. However, this solution focuses on correlation and does not take access frequencies into account. It does not account for the hierarchy of network either.

Duong et al. analyze the problem of statically sharding social networks to optimize read requests [DGHV13]. They demonstrate the benefits of social-network aware data placement strategies, and obtain moderate performance improvements through replication. Nonetheless,

these results are limited by the absence of write requests in the cost model. In addition, it only supports static social networks and does not account for network topology.

There are a few graph processing engines [GLG+12, MAB+10, YYZK12] that split graphs over several machines using offline partitioning algorithms. Messages exchanged between partitions are the results of partial computations, which can be further reduced through the use of combiners. While these approaches lead to important gains, they cannot be applied to all kind of requests, and they mostly benefit long computational tasks rather than low latency systems considered in this paper.

Online data placement SPAR [PES+10] is a middleware for online social networking systems that ensures that the server containing the view of a user also contains those of her friends. This favors reads, but sacrifices writes as all the replica of a user's view need to be updated. Similar to DynaSoRe, SPAR uses an online algorithm that reacts to the evolution of the social network. The main differences between SPAR and DynaSoRe stems from the assumption on the storage layer. SPAR assumes that storage is cheap enough to massively replicate views, up to 20 times for 512 servers, largely exceeding fault tolerance requirements. DynaSoRe is much more flexible, and operates at a sweet spot, trading a small storage overhead for high network gains. By default, DynaSoRe does not guarantee that each view is replicated multiple times, and relies on the stable storage to ensure durability. Yet, DynaSoRe can be configured to provide an in-memory replication equivalent to SPAR, as explained in Section 3.3.1.

Silberstein et al. propose to measure users' events production and consumption rates to devise a push-pull model for social feeds generation [STCR10]. The specialized data transfer policy significantly reduces the load of the servers and the network. DynaSoRe is inspired by this work and also relies on the rates of reads and writes of events to decide when to replace views. However, DynaSoRe addresses different problem and focuses on determining where to maintain the views, which will lead to performance gain in addition to this approach.

DynPart [LGAP+12] is a data partitioning algorithm triggered upon inserting tuples in a database. DynPart analyzes requests matching a tuple and places the tuple on the servers that are accessed when executing these requests. While DynPart handles insertion of data, it never reverts previous decisions and therefore cannot deal with new requests or changes in request frequency. Social networks are frequently modified, leading to different requests, and are subject to unpredictable flash events. For these reasons, DynaSoRe is a better fit for social applications.

6 Conclusion

Adapting to workload variations and incorporating detail of the underlying network architecture are both critical for serving social networking applications efficiently. Typical designs that randomly and statically place views across servers induce a significant amount of load to top tiers of tree-based network layouts. DynaSoRe is an in-memory view storage system that instead adapts to workload variations and uses the network distance between servers to reduce traffic at the top tiers. DynaSoRe analyzes request traffic to optimize view placement and substantially reduces network utilization. DynaSoRe leverages free memory capacity to replicate frequently accessed views close to the brokers reading them. It selects the brokers that serve each request and places them close to the views they fetch according to network distance. In our evaluation of DynaSoRe, we used different social networks and showed that with only 30% additional memory, the traffic of the top switch drops by 94% compared to a static random view placement, and 90% compared to the SPAR protocol.

Appendix B

Résumé en Français

La quantité d'information disponible sur Internet augmente chaque jour. Les magasins en ligne donnent accès a des millions de biens à acheter. Les sites d'information génèrent des centaines d'articles chaque jour. Cette surcharge d'information rend la navigation et la sélection de contenu intéressant très difficile et chronophage. Une méthode classique pour filtrer le contenu est de se baser sur sa popularité, par exemple le nombre de fois qu'une page web est visitée ou le nombre de fois qu'un objet est acheté. Néanmoins, utiliser uniquement la popularité pour filtrer le contenu n'est pas suffisant. En effet, les différents utilisateurs ont des gouts différents et des centres d'intérêts différents que la popularité n'est pas capable de capturer.

Les systèmes de recommandation ont pour objectif de résoudre ce problème en fournissant aux utilisateurs des recommandations personnalisées en fonction de leurs centres d'intérêts. Typiquement, un système de recommandation va, étant donné la liste des avis donné par les utilisateurs aux objets, prédire l'opinion qu'un utilisateur particulier donnerait à tous les objets sur lesquels il n'a pas encore exprimé d'opinion. Ensuite, le système utilise ces prédictions pour sélectionner quels objets devraient être recommandés à cet utilisateur.

Cela a un cout. Calculer des recommandations nécessite a la fois une importante capacité de stockage de données, et une grande capacité de calcul. A cause de ce cout, les entreprises fournissant des services de recommandations personnalisées ont besoin d'en tirer un bénéfice. Cela peut se faire de différentes façons, par exemple en faisant payer les utilisateurs pour le service. Les producteurs de contenu peuvent également payer le fournisseur de service afin de voir leur contenu mieux classé et plus souvent recommandé. Cela biaises les recommandations en faveur des producteurs de contenu qui sont près a payer le plus cher, ce qui n'est pas dans l'intérêt de l'utilisateur. Une entreprise peut également ajouter de la publicité a son service, ce qui nécessite de collecter des informations personnelles sur les utilisateurs, et y donner accès à des entreprises de publicité.

D'une manière générale, les entreprises collectent autant de données qu'elles le peuvent sur leurs utilisateurs, car ces données ont de la valeur. Il existe plusieurs exemples d'entreprises dans une situation difficile qui ont vendu leur fichiers clients à d'autres entreprises. Cela représente une menace sérieuse pour la vie privée des utilisateurs, car ces données peuvent contenir des données sensibles comme par exemple le nom, l'adresse et le numéro de téléphone de l'utilisateur, mais aussi ses habitudes de connections, le type de contenu qui l'intéresse, la liste des pages web visitées, et ainsi de suite. Ces données peuvent non seulement être vendues à d'autres entreprises, mais elles peuvent également être rendu publiques, volontairement ou pas. L'exemple le plus connu est celui de la publication de l'historique de recherche d'AOL. En Aout 2006, l'entreprise AOL a rendu public 3 mois d'historique de recherche de 650.000 utilisateurs à des fins de recherche. Malgré le fait que les données aient été anonymisées avant la publication en

assignant un identifiant aléatoire a chaque utilisateur, les mots clé contenu dans les requêtes contenues suffisamment d'informations sensibles pour identifier certains utilisateurs.

De plus, certaines entreprises peuvent également refuser de traiter du contenu considéré comme négatif par des personnes ou des groupes ayant suffisamment d'influence. Par exemple, il n'est pas rare que des journaux ne traitent pas d'informations négatives concernant les dirigeants du journal. Certaines entreprises peuvent également refuser d'indexer du contenu déclaré illégal par un état, que cela soit légitime (ne pas indexer de pédopornographie), ou pas (censurer des opinions politiques). Les services ne se conformant pas à ces restrictions ont des difficultés à rester disponibles. Par exemple, The Pirate Bay, un site internet fournissant des liens permettant le téléchargement de fichiers via le protocole BitTorent, à dû changer de pays d'hébergement plusieurs fois afin d'éviter d'être coupé. Un autre exemple est Lavabit, un service d'échange d'email utilisant de la cryptographie afin de protéger la vie privée des utilisateurs. En Juillet 2013, Edward Snowden a utilisé Lavabit afin d'envoyer des invitations à une conférence de presse après ses révélations sur le système de surveillance de masse mis en place par les états-unis. Peu après cela, le service de Lavabit a été suspendu. Bien que le créateur de Lavabit n'ai pas été autorisé à donner de détails, il est devenu clair qu'il avait reçu des pressions pour donner access aux informations de ses utilisateurs, et que la seule alternative a été de fermer le service. Cela montre clairement que les services centralisés sont sensibles aux gouvernements et groupes d'influence, et pour cette raison des utilisateurs utilisant un service de recommandation pour naviguer sur le web risquent de n'avoir accès qu'a un sous ensemble limité et censuré de l'information disponible.

Le pair-à-pair (ou P2P) est une manière de contourner les limitations de ces services centralisés. En effet, dans les systèmes pair-à-pair la charge de stockage et de calculs est prise en charge par les utilisateurs, il n'y a donc pas besoin d'un fournisseur de service centralisé portant à la fois le cout et la responsabilité du service. Une infrastructure en pair-à-pair signifie que : 1) il n'y a pas besoin de financer le fournisseur de service, donc la motivation de favoriser de type de contenu particulier n'existe plus, de même que la tentation d'exploiter les données personnelles des utilisateurs; 2) il n'y a pas d'entité centralisée qui puisse être influencée afin de censurer du contenu. Le pair-à-pair représente donc une alternative bien plus robuste étant donné que pour empêcher certain contenu d'être partagé et diffusé, les autorités ont besoin de traquer tous les utilisateurs partageant ce contenu, un par un. Ceci est couteux et nécessite beaucoup de temps, cela n'a donc jamais été suffisamment efficace pour empêcher les utilisateurs d'utiliser les systèmes pair-à-pair.

Dans les dernières années, l'utilisation de systèmes de partage de fichiers en pair-à-pair a commencé à légèrement diminuer, principalement parce que les autorités ont commencé à traquer spécifiquement les réseaux pair-à-pair pour combattre le téléchargement illégal. Tandis que leur résultats ont été mitigés (en France, seulement une dizaine de personnes ont été condamnées en quatre ans), leur tentative de traque des utilisateurs a été rendu possible pour une simple raison: les systèmes pair-à-pair ne sont pas efficaces pour masquer les données des utilisateurs des autres utilisateurs. Par exemple, dans [LBLL+10] les auteurs sont parvenus à collecter les adresses IP de 150 millions d'utilisateurs téléchargeant du contenu sur BitTorrent, ainsi que les adresses IP des utilisateurs ajoutant en ligne 70% du contenu. Tout ce dont ils ont eu besoin pour cela était un simple ordinateur qui a traqué les utilisateurs sur une période de 100 jours.

Les systèmes pair-à-pair souffrent aussi d'un autre problème qui est la présence d'utilisateurs se comportant mal. Dans un système pair-à-pair, la charge du système (stockage, calculs, bande passante) est répartie sur tous les utilisateurs, et certains d'entre eux ne font pas leur part du travail correctement, soit parce qu'ils ne veulent pas collaborer, soit parce qu'ils veulent manipuler le système. Par exemple dans un système de recommandation, certains utilisateurs

peuvent vouloir biaiser les recommandations reçues par les autres utilisateurs en faveur de leur propre contenu. Cela est difficile à éviter car il est très difficile de s'assurer de l'identité d'un utilisateur dans un système pair-à-pair. Il est donc facile pour un utilisateur d'augmenter son influence en créant plusieurs faux utilisateurs.

Malgré ces défauts, nous pensons que le pair-à-pair est le seul moyen de résoudre le problème de vie privée causé par les services centralisés. Il faut néanmoins améliorer les systèmes pair-à-pair afin de mieux protéger la vie privée de leurs utilisateurs et augmenter la confiance entre les utilisateurs du système en empêchant les utilisateurs malveillant de corrompre le système. Dans cette thèse nous présentons quatre contributions allant dans ce sens.

Notre première contributions, TAPS, est un algorithme d'échantillonnage de pairs exploitant un réseau social, comme par exemple Facebook, afin d'attester de la fiabilité des utilisateurs. Un algorithme d'échantillonnage est un algorithme pair-à-pair fournissant à chaque utilisateur un échantillon aléatoire des pairs du système. Ces algorithmes sont utilisé par de nombreuses applications en pair-à-pair et servent de base à la construction de systèmes plus complexe, comme par exemple des systèmes de recommandation ou des tables de hachage distribuées. Pour chaque pair proposé par TAPS à l'utilisateur, une estimation de la fiabilité de ce pair ainsi qu'un chemin sur le réseau social reliant l'utilisateur à ce pair sont fourni. Par exemple, TAPS informera un utilisateur, Bob, qu'un autre utilisateur, Carole, est la soeur d'un collègue de sa femme Alice. Cette information permet a Bob d'avoir des information sur Carole via sa femme, et peut ainsi savoir si elle est digne de confiance. De plus, Bob sait que s'il achète un objet à Carole et qu'elle ne lui envoie pas cet objet, il pourra la contacter via le collègue de sa femme afin d'obtenir réparation. TAPS fonctionne en explorant le réseau social, en partant des amis l'utilisateur, puis leurs amis et ainsi de suite, de façon totalement distribuée. Au dessus de TAPS fonctionne un autre algorithme, TAC, chargé de trouver pour chaque utilisateur les meilleurs voisins possible. Un bon voisin est un utilisateur à la fois fiable et utile du point de vue de la recommandation. TAC extrait les informations fournies par TAPS et sélectionne les voisins donnant le meilleur compromis entre leur fiabilité et leur utilité. Nous avons évalué le couple TAPS/ TAC en comparaison avec deux autres algorithmes, l'un sélectionnant comme voisins les utilisateurs les plus fiables, l'autre sélectionnant les plus utiles. Notre évaluation montre que TAPS et TAC fournissent des résultats proches de l'optimal et systématiquement meilleurs que les deux autres algorithmes.

Notre seconde contribution, PTAPS, est une version alternative de TAPS qui préserve la vie privée des utilisateurs. Dans TAPS, les utilisateurs doivent fournir au système la liste de leurs amis, ainsi que le degré de confiance entre eux. Cette information peut ensuite être accédée par tous les autres utilisateurs, ce qui pose un problème car ces informations peuvent légitimement être considérées comme sensible. PTAPS résoud ce problème en masquant ces informations aux autres utilisateurs du système grâce à un mécanisme de chiffrement des chemins entre les utilisateurs et en contrôlant précisément quelles informations sont partagées avec quels utilisateurs. Notre mécanisme de chiffrement permet de concaténer plusieurs chemins, par exemple un chemin allant de Alice à Bob et un chemin allant de Bob à Carole, sans avoir à déchiffrer ces chemins. De plus, il est possible de naviguer sur le réseau social en suivant un de ces chemins en le passant d'utilisateur en utilisateur, sans que qui que ce soit n'ai besoin de déchiffrer l'intégralité du chemin. En plus de ce mécanisme de chiffrement, dans PTAPS un utilisateur ne fournit jamais à un autre utilisateur la liste de ses amis, sauf aux utilisateurs qui sont eux même ses amis. Ces deux mécanismes assurent que la liste des amis d'un utilisateur, ainsi que la confiance qu'il leur porte, n'est accessible qu'à ses amis directs, et a aucun autre utilisateur. Ces mécanismes réduisent l'efficacité de PTAPS par rapport à TAPS, mais notre évaluation montre que les performances de PTAPS sont bonnes et restent très proches de celles de TAPS.

Notre troisième contribution, FreeRec, est un système de personnalisation assurant l'anonymat des utilisateurs. Les problèmes de vie privée touchant les réseaux pair-à-pair sont dû en grande partie au fait qu'il est possible d'associer les actions d'un utilisateur, comme par exemple la visite de certains site web ou le téléchargement de certain contenus, à l'identité réelle de l'utilisateur (tout du moins leur adresse IP, ce qui permettre d'identifier l'utilisateur). Une solution pour éviter cela est de cacher l'adresse réelle de l'utilisateur aux autres utilisateurs en utilisant ce que l'on appelle un proxy. Un proxy est un ordinateur, qui peut être celui d'un autre utilisateur, relayant les message d'un utilisateur afin que les autres utilisateurs ne connaissent pas son adresse réelle. Par exemple, si Alice souhaites envoyer un message à Bob, mais qu'elle ne veut pas que Bob connaisse son adresse, elle peut envoyer le message à Carole en lui demandant de l'envoyer à Bob. Ainsi, Bob reçoit le message par Carole et ne sait pas qui le lui a envoyé. De plus, Bob peut répondre à Alice en envoyant un message à Carole qui sera relayé à Alice. En pratique, ces systèmes sont un peu plus complexe et utilisent plusieurs noeuds intermédiaires, ainsi que des mécanismes de chiffrement, afin de s'assurer que personne, pas même Carole, ne sache que Alice est en train de Communiquer avec Bob. Nous proposons FreeRec, un algorithme d'échantillonnage associant un proxy à chaque utilisateur, de façon a ce que toutes les communications entre deux pairs se fassent de manière anonyme. Au dessus de cet algorithme est construit un système de recommandations anonyme. Pour des raisons de sécurité, le proxy de chaque utilisateur doit être changé à intervalle régulier, ce qui perturbe le mécanisme de recommandation. Nous avons évalué FreeRec sur deux aspects. Tout d'abord le cout en termes de bande passante engendré par l'utilisation de proxy, ainsi que l'effet que cela a sur la qualité de la recommandation. Nous montrons que les recommandations ne sont que très peut affectées par FreeRec.

Notre quatrième et dernière contribution, DPPC, est un algorithme masquant les données personnelles des utilisateurs dans un système de recommandation. Les données personnelles des utilisateurs, comme par exemple l'historique de navigation, peuvent contenir des information très précises sur l'utilisateur. Il a été démontré que ces données seules sont parfois suffisantes pour découvrir l'identité réelle de l'utilisateur. L'anonymat proposé dans FreeRec n'est donc pas toujours suffisant, et les données de l'utilisateur elles mêmes doivent être protégées. DPPC accomplit cela en construisant pour chaque utilisateur un profil publique ne contenant pas ses données personnelles et pouvant être librement partagé avec les autres utilisateurs (par opposition au profile privé de l'utilisateur contenant ses données personnelles). Le profil d'un utilisateur est composé d'items, un item pouvant être n'importe quel information décrivant l'intérêt de l'utilisateur, comme par exemple l'adresse d'une page web ou le titre d'un livre. Pour chaque item, DPPC crée un dépôt chargé d'agréger le profile de tous les utilisateurs intéressés par cet item. Chaque item, en agrégeant les profils des utilisateurs intéressés, créé un profil d'item, représentant les intérêts moyens des utilisateurs intéressés par cet item. Les utilisateurs créent ensuite leur profil publique en collectant les profiles de tous les items qu'ils ont trouvé intéressant, et les agrègent a leur tour en en faisant une moyenne. Le profil publique ainsi créé ne contient pas les données de l'utilisateur, mais est représentatif des utilisateurs similaires. Notre évaluation montre que DPPC ne dégrade que très peu la qualité des recommandations, et que les attaques possible contre les profiles publiques des utilisateurs ne permettent pas de deviner le profil privé de l'utilisateur.

Ensemble, ces contributions permettent réduire les inconvénients des systèmes de personnalisation pair-à-pair et en font ainsi de bonnes alternatives aux systèmes centralisés. Nous prévoyons dans l'avenir de fusionner l'ensemble de ces contributions en un système unique afin de fournir une base solide, aussi bien en termes de respect de la vie privée que de confiance entre les pairs, pour de nombreuses applications distribué, allant de la recommandation de contenu à une place de marché en ligne.

Bibliography

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In SIGCOMM, pages 63–74, 2008.

[AGK12] Mohammad Alaggan, Sébastien Gambs, and Anne-Marie Kermarrec. Blip: non-interactive differentially-private similarity computation on bloom filters. In Stabilization, Safety, and Security of Distributed Systems, pages 202–216. Springer, 2012.

[AHK+07] Yong-Yeol Ahn, Seungyeop Han, Haewoon Kwak, Sue Moon, and Hawoong Jeong. Analysis of topological characteristics of huge online social networking services. In Proceedings of the 16th international conference on World Wide Web, pages 835–844, 2007.

[AN11] Makram Bouzid Animesh Nandi, Armen Aghasaryan. P3: A privacy preserving personalization middleware for recommendation-based services. In Proceedings of 4th Hot Topics in Privacy Enhancing Technologies Symposium, HotPETS 2011, 2011.

[AS04] Baruch Awerbuch and Christian Scheideler. Group spreading: A protocol for provably secure distributed name service. In Automata, Languages and Programming, pages 183–195. Springer, 2004.

[BCK+07] Matthias Bender, Tom Crecelius, Mouna Kacimi, Sebastian Michel, Josiane Xavier Parreira, and Gerhard Weikum. Peer-to-peer information search: Semantic, social, or spiritual? IEEE Data Eng. Bull., 30(2):51–60, 2007.

[BFG+10] Marin Bertier, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Vincent Leroy. The gossple anonymous social network. In Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware, pages 191–211, 2010.

[BFG+13] Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. Whatsup: A decentralized instant news recommender. In Parallel and Distributed Processing, pages 741–752, 2013.

[BFG+14] Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. Privacy-preserving distributed collaborative filtering. In Networked Systems. Springer, 2014.

[BFJ+13a] Antoine Boutet, Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, and Heverson B Ribeiro. Freerec: an anonymous and distributed personalization architecture. In Networked Systems, pages 58–73. Springer, 2013.

[BFJ+13b] Antoine Boutet, Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, and HeversonB. Ribeiro. Freerec: an anonymous and distributed personalization architecture. Computing, pages 1–20, 2013.

[BGK+09] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. Computer Networks, 53(13):2340–2359, 2009.

[BGLK09] Marin Bertier, Rachid Guerraoui, Vincent Leroy, and Anne-Marie Kermarrec. Toward personalized query expansion. In Proceedings of the Second ACM EuroSys Workshop on Social Network Systems, pages 7–12, 2009.

[BHKL06] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In SIGKDD, pages 44–54, 2006.

[BJJL13] Xiao Bai, Arnaud Jégou, Flavio Junqueira, and Vincent Leroy. Dynasore: Efficient in-memory store for social applications. In Middleware 2013, pages 425–444. Springer, 2013.

[BL07] James Bennett and Stan Lanning. The netflix prize. In Proceedings of KDD cup and workshop, volume 2007, page 35, 2007.

[BMA+11] Stevens Le Blond, Pere Manils, Chaabane Abdelberi, Mohamed Ali Dali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. One bad apple spoils the bunch: exploiting p2p applications to trace and profile tor users. arXiv preprint arXiv:1103.1518, 2011.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems, 30(1):107–117, 1998.

[CF05] Alice Cheng and Eric Friedman. Sybilproof reputation mechanisms. In Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems, pages 128–132, 2005.

[CHBG10] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In In ICWSM, 2010.

[CJZM10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. the VLDB Endowment, 3(1-2):48–57, 2010.

[DGHV13] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. Sharding social networks. In WSDM, pages 223–232, 2013.

[DGLSB08] Marco De Gemmis, Pasquale Lops, Giovanni Semeraro, and Pierpaolo Basile. Integrating tags in a semantic content-based recommender. In Proceedings of the 2008 ACM conference on Recommender systems, pages 163–170, 2008.

[DM09] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In NDSS, 2009.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In Proceedings of the 13th Conference on USENIX Security Symposium, 2004.

[Dou02] John R Douceur. The sybil attack. In Peer-to-peer Systems, pages 251–260. Springer, 2002.

[Dwo06] Cynthia Dwork. Differential privacy. In Automata, languages and programming, pages 1–12. Springer, 2006.

[Dwo08] Cynthia Dwork. Differential privacy: A survey of results. In Theory and Applications of Models of Computation, pages 1–19. Springer, 2008.

[FGK14] Davide Frey, Mathieu Goessens, and Anne-Marie Kermarrec. Behave: Behavioral cache for web content. In Distributed Applications and Interoperable Systems, pages 89–103, 2014.

[FJK11] Davide Frey, Arnaud Jégou, and Anne-Marie Kermarrec. Social market: combining explicit and implicit social networks. In Stabilization, Safety, and Security of Distributed Systems, pages 193–207. Springer, 2011.

[FJK+13] Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, Michel Raynal, and Julien Stainer. Trust-aware peer sampling: Performance and privacy tradeoffs. Theoretical Computer Science, 512:67–83, 2013.

[FM02] Michael J Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In Proceedings of the 9th ACM conference on Computer and communications security, pages 193–206, 2002.

[Fou09] Electronic Frontier Foundation. Aol's massive data leak. `http://w2.eff.org/Privacy/AOL/`, June 2009.

[GHJ+09] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In SIGCOMM, pages 51–62, 2009.

[GJA03] Minaxi Gupta, Paul Judge, and Mostafa Ammar. A reputation system for peer-to-peer networks. In Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video, pages 144–152, 2003.

[GKF+06] Scott Garriss, Michael Kaminsky, Michael J Freedman, Brad Karp, David Mazières, and Haifeng Yu. Re: Reliable email. In NSDI, volume 6, pages 22–22, 2006.

[GLG+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In OSDI, 2012.

[Gol03] Oded Goldreich. Cryptography and cryptographic protocols. Distributed Computing, 16(2-3):177–199, 2003.

[GRGP01] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. Information Retrieval, 4(2):133–151, 2001.

[HB09] Urs Hoelzle and Luiz Andre Barroso. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan and Claypool Publishers, 2009.

[HKBR99] Jonathan L Herlocker, Joseph A Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, pages 230–237, 1999.

[HRW09] Bernardo A. Huberman, Daniel M. Romero, and Fang Wu. Social networks that matter: Twitter under the microscope. First Monday, 14(1), 2009.

[Jan08] Dietmar Jannach. Finding preferred query relaxations in content-based recommenders. In Intelligent Techniques and Tools for Novel System Architectures, pages 81–97. Springer, 2008.

[JB99] Ari Juels and John G Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In NDSS, volume 99, pages 151–165, 1999.

[JE09] Mohsen Jamali and Martin Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 397–406, 2009.

[JKR13] Flavio Paiva Junqueira, Ivan Kelly, and Benjamin Reed. Durability with bookkeeper. ACM SIGOPS Operating Systems Review, 47(1):9–15, 2013.

[JVG+07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. ACM Transactions on Computer Systems (TOCS), 25(3):8, 2007.

[Ker09] Anne-Marie Kermarrec. Challenges in personalizing and decentralizing the web: An overview of gossple. In Stabilization, Safety, and Security of Distributed Systems, pages 1–16. Springer, 2009.

[KG07] Ugur Kuter and Jennifer Golbeck. Sunny: A new algorithm for trust inference in social networks using probabilistic confidence models. In AAAI, volume 7, pages 1377–1382, 2007.

[KSGM03a] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In Proceedings of the 12th international conference on World Wide Web, pages 640–651, 2003.

[KSGM03b] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In Proceedings of the 12th international conference on World Wide Web, pages 640–651, 2003.

[LBLL+10] Stevens Le Blond, Arnaud Legout, Fabrice Lefessant, Walid Dabbous, and Mohamed Ali Kaafar. Spying the world from your laptop: Identifying and profiling content providers and big downloaders in bittorrent. In Proceedings of the 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More, LEET'10, pages 4–4, Berkeley, CA, USA, 2010. USENIX Association.

[LDGS11] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. Content-based recommender systems: State of the art and trends. In Recommender systems handbook, pages 73–105. Springer, 2011.

[LGAP+12] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, and Patrick Valduriez. Dynamic workload-based partitioning for large-scale databases. In DEXA, volume 7447, pages 183–190, 2012.

[LHL+10] Zhengye Liu, Hao Hu, Yong Liu, Keith W Ross, Yao Wang, and Markus Mobius. P2p trading in social networks: the value of staying connected. In INFOCOM, 2010 Proceedings IEEE, pages 1–9, 2010.

[LLK10] Chris Lesniewski-Lass and M Frans Kaashoek. Whanau: A sybil-proof distributed hash table. In Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, 2010.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free riding in bittorrent is cheap. In Proc. Workshop on Hot Topics in Networks (HotNets), pages 85–90, 2006.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. Internet Computing, IEEE, 7(1):76–80, 2003.

[MA05] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: An experimental study on epinions. com community. In Proceedings of the National Conference on artificial Intelligence, page 121, 2005.

[MA07] Paolo Massa and Paolo Avesani. Trust-aware recommender systems. In Proceedings of the 2007 ACM conference on Recommender systems, pages 17–24, 2007.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In SIGMOD, pages 135–146, 2010.

[MOP+04] Alan Mislove, Gaurav Oberoi, Ansley Post, Charles Reis, Peter Druschel, and Dan S Wallach. Ap3: Cooperative, decentralized anonymous communication. In Proceedings of the 11th workshop on ACM SIGOPS European workshop, page 30, 2004.

[MPDG08] Alan Mislove, Ansley Post, Peter Druschel, and P Krishna Gummadi. Ostra: Leveraging trust to thwart unwanted communication. In NSDI, volume 8, pages 15–30, 2008.

[MR00] Raymond J Mooney and Loriene Roy. Content-based book recommending using learning for text categorization. In Proceedings of the fifth ACM conference on Digital libraries, pages 195–204, 2000.

[MSLC01] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. Annual review of sociology, pages 415–444, 2001.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In NSDI, pages 385–398, 2013.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In Security and Privacy, 2008. SP 2008. IEEE Symposium on, pages 111–125, 2008.

[ORS+11] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In SOSP, pages 29–41, 2011.

[PB07] Michael J Pazzani and Daniel Billsus. Content-based recommendation systems. In The adaptive web, pages 325–341. Springer, 2007.

[PES+10] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. In SIGCOMM, pages 375–386, 2010.

[R+01] Paul Resnick et al. The social cost of cheap pseudonyms. Journal of Economics & Management Strategy, 10(2):173–199, 2001.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Middleware 2001, pages 329–350, 2001.

[RSG98] Michael G Reed, Paul F Syverson, and David M Goldschlag. Anonymous connections and onion routing. Selected Areas in Communications, IEEE Journal on, 16(4):482–494, 1998.

[SM86] Gerard Salton and Michael J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York, NY, USA, 1986.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review, 31(4):149–160, 2001.

[STCR10] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In SIGMOD, pages 831–842, 2010.

[TLSC11] Nguyen Tran, Jinyang Li, Lakshminarayanan Subramanian, and Sherman SM Chow. Optimal sybil-resilient node admission control. In INFOCOM, 2011 Proceedings IEEE, pages 3218–3226, 2011.

[TMLS09] Dinh Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient online content voting. In NSDI, pages 15–28, 2009.

[VABHL03] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. Captcha: Using hard ai problems for security. In Advances in Cryptology, pages 294–311. Springer, 2003.

[VGVS05] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. Journal of Network and Systems Management, 13(2):197–217, 2005.

[VR79] C. J. Van Rijsbergen. Information retrieval. Butterworth, 1979.

[VVS05] Spyros Voulgaris and Maarten Van Steen. Epidemic-style management of semantic overlays for content-based searching. In Euro-Par 2005 Parallel Processing, pages 1143–1152. Springer, 2005.

[WBS+09] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In Eurosys, pages 205–218, 2009.

[WV05] Yao Wang and Julita Vassileva. Bayesian network trust model in peer-to-peer networks. In Agents and Peer-to-Peer Computing, pages 23–34. Springer, 2005.

[YGKX08] Haifeng Yu, Phillip B Gibbons, Michael Kaminsky, and Feng Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. In Security and Privacy., pages 3–17, 2008.

[YKGF08] Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham D Flaxman. Sybilguard: Defending against sybil attacks via social networks. Networking, IEEE/ACM Transactions on, 16(3):576–589, 2008.

[YYZK12] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In SIGMOD, pages 517–528, 2012.

[Zac00] Giorgos Zacharia. Trust management through reputation mechanisms. Applied Artificial Intelligence, 14:881–907, 2000.

[ZH04] Yingwu Zhu and Yiming Hu. Tap: A novel tunneling approach for anonymity in structured p2p systems. In Parallel Processing, 2004. ICPP 2004. International Conference on, pages 21–28, 2004.

[ZH07] Runfang Zhou and Kai Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. Parallel and Distributed Systems, IEEE Transactions on, 18(4):460–473, 2007.

[ZHS+04] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. Selected Areas in Communications, IEEE Journal on, 22(1):41–53, 2004.

[ZSS08] Ming Zhong, Kai Shen, and Joel Seiferas. Correlation-aware object placement for multi-object operations. In ICDCS, pages 512–521, 2008.

[ZZZR05] Li Zhuang, Feng Zhou, Ben Y Zhao, and Antony Rowstron. Cashmere: Resilient anonymous routing. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 301–314, 2005.