Privacy-Preserving Interest Matching for Mobile Social Networking

by

Qi Xie

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Qi Xie

Abstract

The success of online social networking has resulted in increased attention to mobile social networking research and applications. In mobile social networking, instead of looking for friends over the Internet, people look for friends who are physically located close and also based on other self-defined criteria. For example, a person could find other people who are nearby and who also share the same interests with her by using mobile social networking. As a result, they have common topics to talk about and may eventually become friends. There are two main approaches in the existing works. One approach focuses on efficiently establishing friendship and ignores the protection of private information of the participants. For example, some applications simply broadcast users' personal information to everybody and rely on the other users to report the matches. From a privacy point of view, this approach is bad, since it makes the users vulnerable to context-aware attacks. The other approach requires a central server to participate in each matchmaking process. For example, an application deploys a central server, which stores the profile information of all users. When two nearby client devices query the central server at the same time, the central server fetches the profile information of both devices from the server's database, performs matching based on the information, and reports the result back to the clients. However, a central server is not always available, so this approach does not scale. In addition, the central server not only learns all users' personal information, it also learns which users become friends.

This thesis proposes a privacy-preserving architecture for users to find potential friends with the same interests. The architecture has two matchmaking protocols to prevent privacy leaks. Our protocols let a user learn only the interests she has in common with the other party. One protocol is simpler, but works only if some assumptions hold. The other protocol is more secure, but requires longer execution time. Our architecture does not require any central server that is involved in the matchmaking process. We describe how the protocols work, analyze how secure the protocols are under different assumptions, and implement the protocols in a BlackBerry application. We test the efficiency of the protocols by conducting a number of experiments. We also consider the cheating-detection and friend-recognition problems.

Acknowledgements

I would like to thank my supervisor, Dr. Urs Hengartner, for his great help and support on my study and research. He guided me through the entire process of writing this thesis with patience and excellent advices on both technical part and English part.

I would like to thank Dr. Ian Goldberg and Dr. Doug Stinson for carefully examining my thesis and giving me valuable comments.

I would like to thank Earl Oliver for his helpful suggestions on Bluetooth programming.

I would like to thank my good friend and roommate Richard Jang for helping proofread my thesis and improve my basketball skills. He and other friends make my life in Waterloo interesting and joyful.

Finally, I would like to thank my parents. Without their support, I could not come to Canada or accomplish this thesis.

Dedication

This is dedicated to my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In the last decade, the number of users of two new services has boosted sharply. One service is online social networking sites, such as Facebook, MySpace, and Flickr, which have gained great success in the last few years. For example, statistics [15] show that Facebook has more than 200 million active users, and more than 100 million users log on to Facebook at least once per day. On average, each user has 120 friends on the site. These sites provide interfaces for people to communicate with known friends or look for potential friends from the same school or company. Mobile phone service is the other service that has grown so fast. A report [26] shows that there were 4.1 billion mobile cellular subscribers in total in March 2009. These numbers are still growing rapidly.

Mobile social networking integrates these two fast-growing services together. People walk around with their mobile devices and meet different people, known and unknown ones, every day. Mobile social networking applications take advantage of the mobility of the mobile devices and design systems for users to meet potential friends with similar interests or some other criteria. When two mobile devices are physically located close, they could start to exchange information without human interaction. In social networking sites, other than communicating with existing friends, people can find and make friends with other people with similar interests or from the same school or company, etc. In mobile social networking, users find new friends as they do at the social networking sites, but mobile social networking adds an extra matching criterion: physical distance between two users. More specifically, matched friends can immediately meet each other in person, since they are next to each other; while in social networking sites, this is not guaranteed. For example, a mobile social networking user may find out that the person setting next to her on the bus was her schoolmate, or her rigid boss likes the same comedy as she does.

Mobile social networking is currently a hot topic in the social networking area. There are numerous applications developed for mobile social networking by different companies or research groups, e.g., MobiClique [28], Loopt Mix [22], Gatsby [17], StreetSpark [6].

In general, there are two straightforward ways to implement mobile social networking applications.

1. One way is to broadcast a user's profile information to the public. MobiClique [28] is an example of the applications that take this approach. MobiClique users download their profile information from Facebook and send this information to any Bluetooth device nearby. After receiving a piece of profile information, a device performs a matching between the received profile information and its own profile information and decides whether the other party is of the owner's interest according to the matching result. PodNet [29] is another example that uses this approach, even though PodNet is not a typical mobile social networking application. PodNet is a project for mobile devices to share contents. It allows mobile devices to provide new ad hoc broadcasting domains and broadcast content, such as pictures, videos or voice information, to other devices within the radio range. This approach is quite risky because it leaks users' private information, and as a result, it increases the success rate of context-aware, targeted phishing attacks. Ideally, in our problem setting, we want two users to learn only the personal information that they have in common, because that is sufficient for them to establish friendship. For example, Alice is interested in playing basketball and fishing, and Bob is interested in reading and playing basketball. Both of them want to look for new friends who share any common interest. In this example, they care only about what they have in common. It does not matter if the other person likes fishing or boating besides playing basketball.

2. The other way is to introduce a trusted central server in the matchmaking process, and rely on the central server to perform the matching. Loopt Mix [22], Gatsby [17], and StreetSpark [6] are examples that use this approach. These applications work similarly. A central server stores all users' personal information. Users log into a central server, and the central server tracks users' location information. The server informs both users if any two of them are nearby and could become friends based on both users' requirements. This approach has the limitation that it requires the central server to be always available in order to find a friend. From a privacy point of view, the server learns all users' personal information, the pairs of users who meet each other, and maybe also the location information of the users.

We discuss more related work in section 1.2. In this thesis, we propose an architecture for mobile social networking. The architecture uses several matchmaking protocols that are different from the previous two approaches. We design our architecture based on two principles.

1. Ensure privacy by making users reveal only the minimum amount of information to other users or any trusted party in order to find new friends.

2. Not relying on any central server to perform matching. As a result, users do not need Internet access to find friends, and the users do not need to be worried about being tracked by the central server when using this architecture.

Here are the contributions of this thesis.

1. We present an architecture for mobile social networking. Namely, a mobile user can deploy this architecture to look for friends who share the same interests with her. Our architecture can also be used to find potential friends who graduated from the same school or worked for the same company before. However, in this thesis, we use interest matching as an example.

2. We propose two matchmaking protocols in this architecture for users to exchange information. One protocol has much less computational overhead, but it leaks more information than necessary under some circumstances. The other approach is more secure, but also more expensive in terms of computation. Furthermore, we propose a faster version of the first protocol. We analyze the security of the protocols and provide the flexibility for users to select any protocol to trade off privacy and performance.

3. We implement these two protocols and evaluate their performance.

4. We discover a new application to which we could apply a variant of our first protocol. We present the variant of our protocol for this application, implement it, and also evaluate the performance.

5. During our implementation, we discover that a Bluetooth feature in some mobile devices may become a barrier when applying our architecture. We provide a number of solutions, implement one of the solutions, and again evaluate the performance.

The rest of the thesis is organized as follows: We define the system architecture in section 2.1. In section 2.2, we describe how our system would be used under different scenarios. Section 2.3 presents the threats or attacks our system deals with or does not. We introduce our simple matchmaking protocol in section 3.2. We also analyze the security of this approach. To mitigate some of the weaknesses of this protocol, we propose some methods in section 3.3, including the DDH approach, which is a more complicated approach. A variant of our matchmaking protocol can be used for another application. Namely, we propose a protocol for a shopping notification application in section 3.4. We provide implementation details and an evaluation of our matchmaking protocols in section 4.1. During our implementation, we find some limitations

of Bluetooth communication. We provide several solutions for the problems in section 4.4. We list a number of research directions and conclude our thesis in section 5.

1.2 Related Work

1.2.1 Mobile Social Networking Applications

Research has been done in the mobile social networking field. Social serendipity [12] deploys a central server, which contains users' profiles and user-defined matchmaking preferences. The central server computes the similarity of users' profile information based on the profile information itself and matchmaking preferences. Users will be notified when they are nearby and the similarity of their profiles exceeds a threshold. Obviously, the central server has to be involved in every matchmaking process, which is a huge limitation of this system. In addition, the central server is able to learn which two users are nearby and become friends. Furthermore, the central server may be able to track a user's exact location. MobiClique [28] improves social serendipity by taking the central server away from the matchmaking process, although this system still requires a central server, Facebook, to assign identifiers to the users. MobiClique allows users to store their profile information on their mobile devices and exchange their profiles with their neighbours. New friendships are established based on the received user profiles. This system does not take malicious users into account. For example, users' profile information is exchanged in plaintext, which could be intercepted by anybody within range, and this information could be used for malicious purposes, such as context-aware attacks. Also, users could be impersonated, because the system does not have a method to validate identifiers. More importantly, their system reveals more information than necessary. Our approaches take malicious users into account, and we use signed identifiers from a legitimate agency to identify users. We try to let other users learn as little unnecessary information as possible.

SmokeScreen [7] looks at two different social networking problems. One is to hide a user's identifier among a group of friends. The other is "missed connections", the problem that users met before and they want to talk to each other again at a later time. In the "missed connections" problem, users convince each other that they were at the same location at the same time. SmokeScreen prevents users from revealing their real identifiers to undesired people. When users meet, they exchange opaque identifiers rather than their real identifiers. They record the time and position when they receive the opaque identifiers and then contact each other at a later time through a trusted third party, a broker, which is able to read the opaque identifiers. Again, this approach requests a trusted central server for every matchmaking process. Also, because no information is shared by the participants before their identifiers are exchanged, there is no incentive for the participants to exchange their real identifiers (they know nothing else about the other

party). In SmokeScreen, the broker knows who is interested in solving whose opaque identifier. SMILE [23] is an advanced version of SmokeScreen, and they further guarantee that the central server is not able to know which pairs of users encounter each other. Again, they require the central server to participate in each matchmaking process. They still do not take personal information, such as personal interests into account, but only location and time. Our scheme allows users to establish friendship as soon as they meet each other without requiring any third party to exist, and we have more matching criteria.

Social Net [34] is a system that detects the same activity pattern (e.g., the time slot a user goes to gym) between two users and then checks if these two users have a common friend. If they do, the system informs their friend to introduce these two users to each other. Their system matches two people depending on their activity patterns, and it also requires the two users to have a common friend, while our system matches two people depending on various personal information. In other words, we look at different types of matchmaking criteria. In addition, their approach requires a third user (the friend that both users have in common) to be physically present in order for those two users to be matched.

1.2.2 Matchmaking Protocols

The privacy-enhanced feature requests our system to run one or more kinds of matchmaking or private intersection protocols. Shin and Gligor [32] introduce a privacy-enhanced matchmaking protocol. This protocol is designed for wish matches. It is based on a PAKE (password-based authenticated key exchange) protocol, but it replaces users' real identifiers with pseudonyms, and after a session key based on their shared secret (they regard users' wishes as low-entropy secrets) is exchanged, each party encrypts all messages transferred during the execution (her real identity, public key, and the certificate for her public key) using the session key. Note that they ignore the wish credentials to simplify their paper. They have not explicitly defined the term "wish credentials"; however according to the context we believe wish credentials are certified wishes issued by a trusted server. Each party computes a digital signature of the messages. Then, they exchange the messages and signatures. They accept the other party as a matching partner only if the information that they receive can be verified. They also introduce a credential revocation method to forbid adversaries from executing the protocol after they are detected. In order to have this revocation method work, they suggest to have the matchmaker assign pseudonyms to participants before each protocol execution. Similar to their approach, we also have a third party assign user ids to the users, but we require users to use consistent user ids. Namely, a user uses only one id in all matchmaking processes. In addition, we require users to use certified interests to prevent malicious users from detecting other users' interests by including all possible elements. A third party server creates signatures for users including their user ids and personal information. We use time stamps in the signatures for user id revocation. We revoke user ids

by not issuing them new signatures with updated time stamps. One reason not to adopt their approach is that using pseudonyms makes it impossible for users to recognize each other after their first meeting. For example, the same two users might run the protocol repeatedly.

Camenisch and Zaverucha [3] propose a protocol for the private intersection problem. Suppose that there are two users, A and B. Each of them has a set, called S_A and S_B, respectively. Both user A and user B learn only the intersection of S_A and S_B, and nothing else. Camenisch and Zaverucha improve the previous works by taking certified inputs, so that an attacker cannot explore a user's set by including all possible elements. They use CL-signatures [2] to sign set elements. The computational overhead of their protocol is high. They use an advanced cryptographic approach to allow users to perform private intersection and also provide untraceability. More specifically, a person exchanges different processed information with the other party, and she is able to prove to the other party that she processed her information correctly. As a result, she appears as a different person each time running this protocol. It requires a large number of modular exponentiation operations. They require $O(k^2)$ (k is the size of an input set) modular exponentiations. Our experiments show that the mobile devices we use are slow when performing modular exponentiations. More data are presented in section 4.1. Our system does not require untraceability. In contrast, we want users to recognize each other when they meet again. Since they are physically close, if they cannot recognize each other, they are going to run the protocol over and over again.

De Cristofaro and Tsudik [8] propose several protocols that are simpler than Camenisch and Zaverucha's approach. They require only at most a linear number of exponentiations. Assume that the size of the client set is k_c and the size of the server set is k_s. The APSI protocol requires $O(k_c)$ exponentiations for the client and $O(k_c + k_s)$ exponentiations for the server. The more efficient PSI protocol does not require any exponentiation from the client and $O(k_c)$ exponentiations for the server. However, their protocols cannot verify if a user actually owns a certain certificate. Namely, a user could "borrow" or "steal" other users' certificates and pretend that the certificates are signed for her. After studying their protocols, we did not find a way to add this feature to their protocols.

Agrawal et al. [1] propose a protocol using a commutative encryption function for private intersection problems. A commutative encryption function has the property: $E_{k1}(E_{k2}(P)) = E_{k2}(E_{k1}(P))$. Either user only learns that $P = P'$, when $E_{k1}(E_{k2}(P)) = E_{k2}(E_{k1}(P'))$, but neither of them could learn the other party's information outside of the intersection because of lacking necessary key information. This paper suggests the power function, $f_e(x) = x^e \bmod p$, as an example of a commutative function. The security of their protocol is based on the Decisional Diffie-Hellman hypothesis (DDH). We improve and implement this approach as one of our approaches. Its computational overhead is not as bad as Camenisch and Zaverucha's approach. If we assume that the size of each user's set is k, this protocol requires k modular exponentiations for each user, and it is possible to add the features that we require to this protocol. More details

Alice's input: a
Bob's input: $B = \{b_1, b_2, ..., b_t\}$

1. Alice randomly chooses $R = \{r_1, r_2, ..., r_t\}$
2. Bob randomly chooses $S = \{s_1, s_2, ..., s_t\}$
3. Alice $\xrightarrow{a \oplus R = \{a \oplus r_1, a \oplus r_2, ..., a \oplus r_t\}}$ Bob
4. Bob computes $P = a \oplus R \oplus S = \{a \oplus r_{p(1)} \oplus s_{p(1)}, a \oplus r_{p(2)} \oplus s_{p(2)}, ..., a \oplus r_{p(t)} \oplus s_{p(t)}\}$
5. Alice $\xleftarrow{P, B \oplus S}$ Bob
6. Alice computes $B \oplus S \oplus R = \{b_1 \oplus s_1 \oplus r_1, b_2 \oplus s_2 \oplus r_2, ..., b_t \oplus s_t \oplus r_t\}$.
7. Alice computes $B \oplus S \oplus R \cap P$

Figure 1.1: Li et al.'s symmetric key based protocol

of this work are provided in section 3.3.

Some papers state that they could solve the private intersection problem by using simple symmetric key approaches; however, we have found some flaws in these papers. Li et al. [33] propose to use XOR as the symmetric commutative function, which sharply reduces the computational overhead. In their scheme, they want to find out if an element a from Alice is in the set B from Bob. The protocol is shown in Figure 1.1. In step 1 and 2, both users randomly choose a set of random nonces (assuming that Alice knows the size of Bob's set). In step 4, Bob does a permutation on $a \oplus R \oplus S$. We denote the set after the permutation P. If there is a non-empty intersection between $B \oplus S \oplus R$ and P, $a \in B$. However, Alice could easily explore Bob's whole set in this scheme. Suppose that Alice knows the superset of B, U, then for each element $e \in U$, it is trivial to find r'_i such that $e \oplus r'_i = a \oplus r_i$. For example, in our problem setting, attackers could collect the names of people's interests. When Alice receives the two sets from Bob in step 5, instead of computing $\{b_1 \oplus s_1 \oplus r_1, b_2 \oplus s_2 \oplus r_2, ..., b_t \oplus s_t \oplus r_t\}$, she could compute $B \oplus S \oplus R' = \{b_1 \oplus s_1 \oplus r'_1, b_2 \oplus s_2 \oplus r'_2, ..., b_t \oplus s_t \oplus r'_t\}$. If there is a non-empty intersection between $B \oplus S \oplus R'$ and P, $e \in B$. In fact, any commutative encryption function for which attackers can easily find k' for a given k such that $E_{k'}(P') = E_k(P)$ is improper for our problem setting. (P and P' are from a set that is generated by a trusted third party, so the values are out of the control of the attackers.)

Mascetti et al. [24] mention that the above protocol could be applied to their system. This paper actually acts as a counter-example for deploying this protocol. It would be insecure to do that. They propose the Hide & Crypt protocol to check if two users are located close to each other in a secure way. Let us call the users Alice and Bob. Alice is the initiator of the protocol. Alice's inputs are indices associated with the neighbourhood of her precise location, and Bob's input is the index associated with his precise location. In the paper, the authors mention that they could use Li et al.'s approach to find out if Bob's index is in the indices of Alice's set. If it is, then Alice and Bob are nearby. Because the indices are publicly known, no matter what Alice's input is, she is able to find out Bob's precise location. It is also helpful to realize that even if they used a really secure set-inclusion protocol in their scheme, their Hide & Crypt protocol would still be insecure. The reason is that in their scheme, in order for Alice to hide the cardinality of her set, she is allowed to add dummy indices to her set. This indicates that there is no way for Bob to validate Alice's inputs (there is no way to check if Alice is actually around the place as she claims). Alice, however, could include all possible indices associated with her guesses about Bob's location. It takes more computational effort for Alice to figure out Bob's location than using an insecure set-inclusion protocol, but the scheme itself is vulnerable.

Freedman and Nicolosi [16] state that they adopt a symmetric key encryption-based private intersection scheme for a friend-of-friend problem. For example, if a sender S sends an email to a recipient R, and R and S have a friend in common, then they have the friend-of-friend relationship. They claim that their scheme prevents R from learning information about S's friends outside of the intersection of their friends. Actually, their scheme fails to achieve this goal. Essentially, their scheme works in this way: R trusts a group of users, X. S is trusted by a group of users, Y. R chooses a secure seed S_R, and she computes $F_{S_R}(\text{“}arc\text{”}, R, X)$ (F is a cryptographic pseudo-random function) and $\sigma_{X \to R} = sign_{R's\ private\ key}(H(X), H(R), time\ stamp)$. She sends these to each X she trusts. In return, each X sends her secure seed to R. Also each Y who trusts S computes $F_{S_Y}(\text{“}arc\text{”}, Y, S)$, and $sign_{Y's\ private\ key}(H(S), H(Y), time\ stamp)$. Y sends these to S. In return, S sends her secure seed to Y. When S sends email to R, she generates an encryption key based on $F_{S_Y}(\text{“}arc\text{”}, Y, S)$, and encrypts $\sigma_{X \to R}$ using the encryption key. When R receives information, she checks if she has the right decryption key to decrypt the encrypted F. If she does, then she shares a friend with S. A simple attack could break this system. Let us assume that Eve registers a new email address, and she trusts everybody in the system. According to the algorithm, Eve receives the secure seeds from all other users. In other words, Eve is able to collect all secure seeds from other users in this system, and by applying the secure seeds to her regular email, she is able to detect all friends of any sender. It can be argued that a user should send her secure seed back only to users who she trusts. There are two problems with this argument. Firstly, the paper explicitly mentions that R receives the secure seeds from X after she sends $F_{S_R}(\text{“}arc\text{”}, R, X)$ and $\sigma_{X \to R}$ to them, and secondly, this affects the scalability of this system.

In short, there are problems with the symmetric key based private intersection approaches

even though they are cheap in terms of computation. We propose a simple protocol that tries to mitigate the security problems brought by the symmetric key based private intersection protocols. Our simple protocol is a little more expensive than Li et al.'s protocol, while it still requires much less computation power than asymmetric key based private intersection approaches.

We have seen that the existing mobile social networking approaches either require a central third party in each matchmaking process or do not consider protecting users' personal information. Our system needs only third parties in its setup phase and does not need them in each matchmaking process, because a central third party is not always reachable. Also, our system deploys two private intersection protocols, which help users to protect their privacy. Our system also provides users the flexibility to trade off privacy and cost. According to our analysis in section 3.2.4, our simple protocol still leaks unnecessary information under some circumstances. Our asymmetric key based protocol provides users more protection, but at more cost. The asymmetric key based protocol is for users who care more about their privacy than the execution time or battery life. Users have the flexibility to run either protocol, and they can make decisions based on their devices' statuses, such as the battery life.

Chapter 2

Architecture and Threat Model

In this chapter, we introduce the components of our architecture. We also describe what will happen when two users meet. Finally, we present a threat model for our architecture.

2.1 System Architecture

The architecture for our system for Mobile Social Networking consists of six modules: Profile Module, Config Module, Matchmaking Module, Schedule Module, Group Module, and File Sharing Module. Figure 2.1 shows an overview of our system and how information flows. Each user runs these modules on her mobile device.

The Profile Module stores the user's personal information, including the user id (more information in section 3.1.1) and matchmaking information, such as interests. Each field except the default contact information in this module is optional; users are not forced to fill any of these fields. To view or edit any information in this module, users have to provide pre-defined passwords. However, a user has to provide a piece of default contact information, such as her email address, phone number or profile picture.

The Config Module sets the information users want to use for looking for new friends. The module reads interests from the Profile Module. Only activated interests are going to be sent out to nearby users for matching. By default, none of the interests is activated. This means that a user is not looking for a friend. Users can activate or deactivate each interest manually or load them from the Schedule Module (more details are in Schedule Module). The Config Module also has system status, ways to communicate, and the types of matchmaking protocols. Users can set the system status to be online/offline to start the online/offline protocol. To run the online protocol, the user has to provide a profile picture of herself or her cell phone number depending

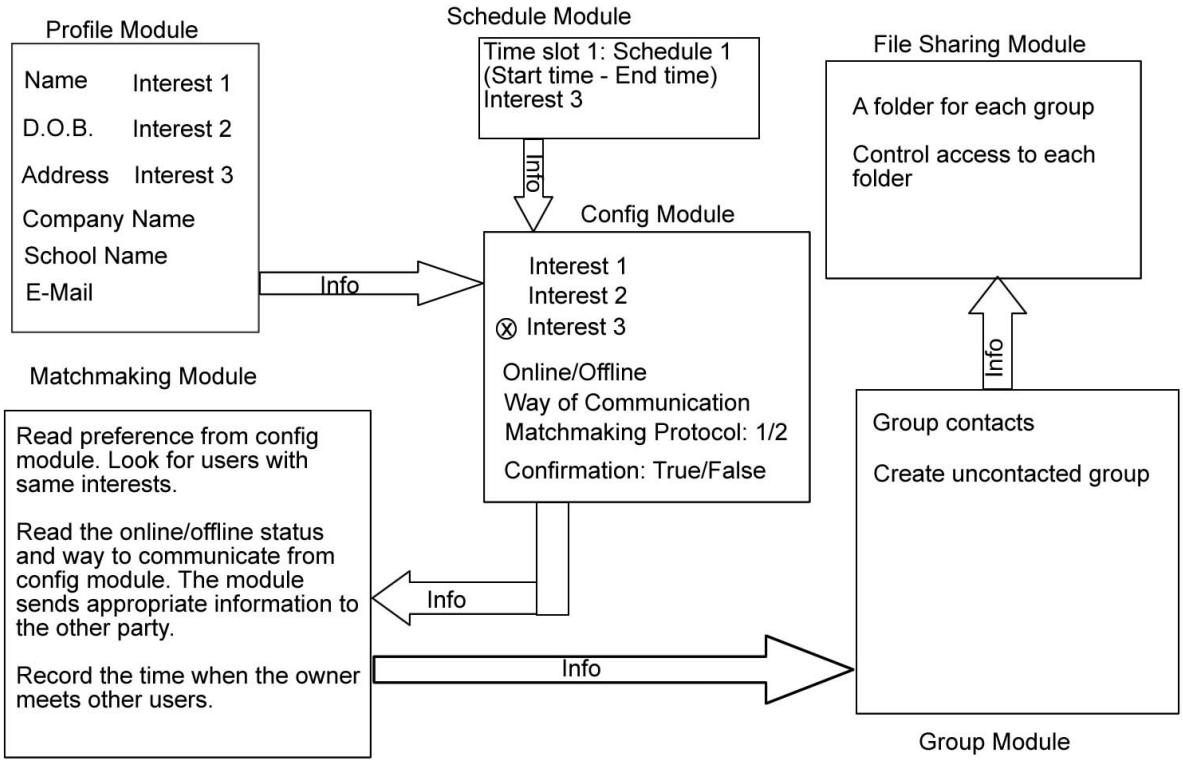

Figure 2.1: System Architecture

on the way how they set to communicate with other users. To run the offline protocol, the user has to provide her email address (as the default contact information). A confirmation field allows users to set if they want to automatically add a potential friend to their friend lists. Two users become potential friends as soon as they find that they have at least one common interest, but they will be friends only if they add each other to their friend lists. If the confirmation field is set to "YES", the user will be notified every time when a potential friend is found. Then, she can decide whether to add the potential friend to her friend list. Otherwise, a potential friend is going to be automatically added to the user's friend list. After the Config Module is set, the information is passed to the Matchmaking Module, and the Matchmaking Module is able to start looking for new friends. Users have the option to choose one of our matchmaking protocols (we will present the protocols in the next chapter) that they want to run in this module.

The Schedule Module allows users to set during which time slot they want to send which interest to other users. Users need to turn on the Schedule Module to have it work. In this module, users could save different combinations of their interests, and give names to the combinations. They set the starting time and ending time for a self-defined time slot and input the name of a combination, so that in this time slot, only the interests in the combination would be sent. If a time period does not exist and the Schedule Module is on, the device automatically turns the system off. This module provides the flexibility for the users to improve the system performance. For example, if Alice is a graduate student in the security and privacy group of the computer science department, it does not make any sense for her to look for people interested in security and privacy while she is meeting her group-mates. Also, for example, if Alice likes playing basketball, and she brings her mobile phone to the basketball court in her university, then it is unnecessary for her to look for friends who like basketball at that time. Users can turn the schedule module off. Users can also schedule the time to switch this module off or on.

The Matchmaking Module is responsible for exchanging information with other users. In this thesis, we focus on the design, implementation, and evaluation of this module. Ideally, we want to guarantee that two devices learn only the common interests between them. If there is a non-empty intersection, they consider each other to be a potential friend. Other than the interests they have in common, nothing else should be revealed. If both participants are running an online protocol, then both of them will be notified immediately after a new potential friend is found. A user could add the new friend to a group using the Group Module, which is introduced in the next paragraph. For example, if their common interest is playing chess, then she can add this friend to the chess group using the Group Module (more about this module in the next paragraph). If at least one of the participants uses the offline protocol, the Matchmaking Module records the default contact information of the new potential friend and the time of the interaction, so that they could get in touch at a later time. In this case, both users run the offline protocol, and nobody will be notified. The Matchmaking Module also keeps a log of any participants on when and who it talked to.

The Group Module provides an interface for users to (re-)group their contacts. A database is created to store the relationship between groups and contacts in the device. The File Sharing Module creates a folder for a group. Users can put data into the folders to share with different friends. When two users meet, they authenticate each other first. We will see how devices authenticate each other in section 3.2.2. Then, the system knows which groups the other party belongs to and then grants the other party accesses to the proper information. There is also an un-contacted group for the new friends found in the offline protocol. A user is allowed to create a public folder to share information to everyone.

2.2 Scenarios

The architecture could be applied in settings where people do not move far from each other in a short period of time. For example, a user could use this architecture for mobile social networking purposes in a bar, bus, train, restaurant, or gym. The reason why we need such settings is that usually it takes time for a Bluetooth device to discover other Bluetooth devices and then exchange information. Usually, it takes about 10 seconds to finish scanning for nearby Bluetooth devices. We provide the execution time of our matchmaking protocols in section 4.1.7.

2.2.1 First Encounter

When two users encounter each other and their devices get connected to each other, their devices first authenticate each other and then check if they have met before. This is done by checking if the other user's user id exists in the database. We say that two users meet for the first time if their devices cannot find each other's user id in their databases. They then run the matchmaking protocol. If they cannot find at least one common interest, they store the time they meet and each other's user id in their local databases as an unmatched stranger. If they successfully find one or more common interests, we say that each user finds a new potential friend. Now, depending on their config settings, they choose to contact online or offline. For the offline protocol, users are not notified. The devices automatically exchange their default contact information and their public key for the next meeting, and the other party's name is kept as a potential friend in the local database of each device, until the owner of each device manually sets the name as a friend later. We can assume that the name of each user/device is globally unique. We are going to discuss how a name of a user/device is created in section 3.1.1. For the online protocol, users could require the other party's profile picture or cell phone number (ways of communication) in order to establish friendship. If a user requires a picture or cell phone number from another user, she automatically agrees to send other users her picture or cell phone number. If a user does not set this information, she would be notified once other users send her the requests. If one

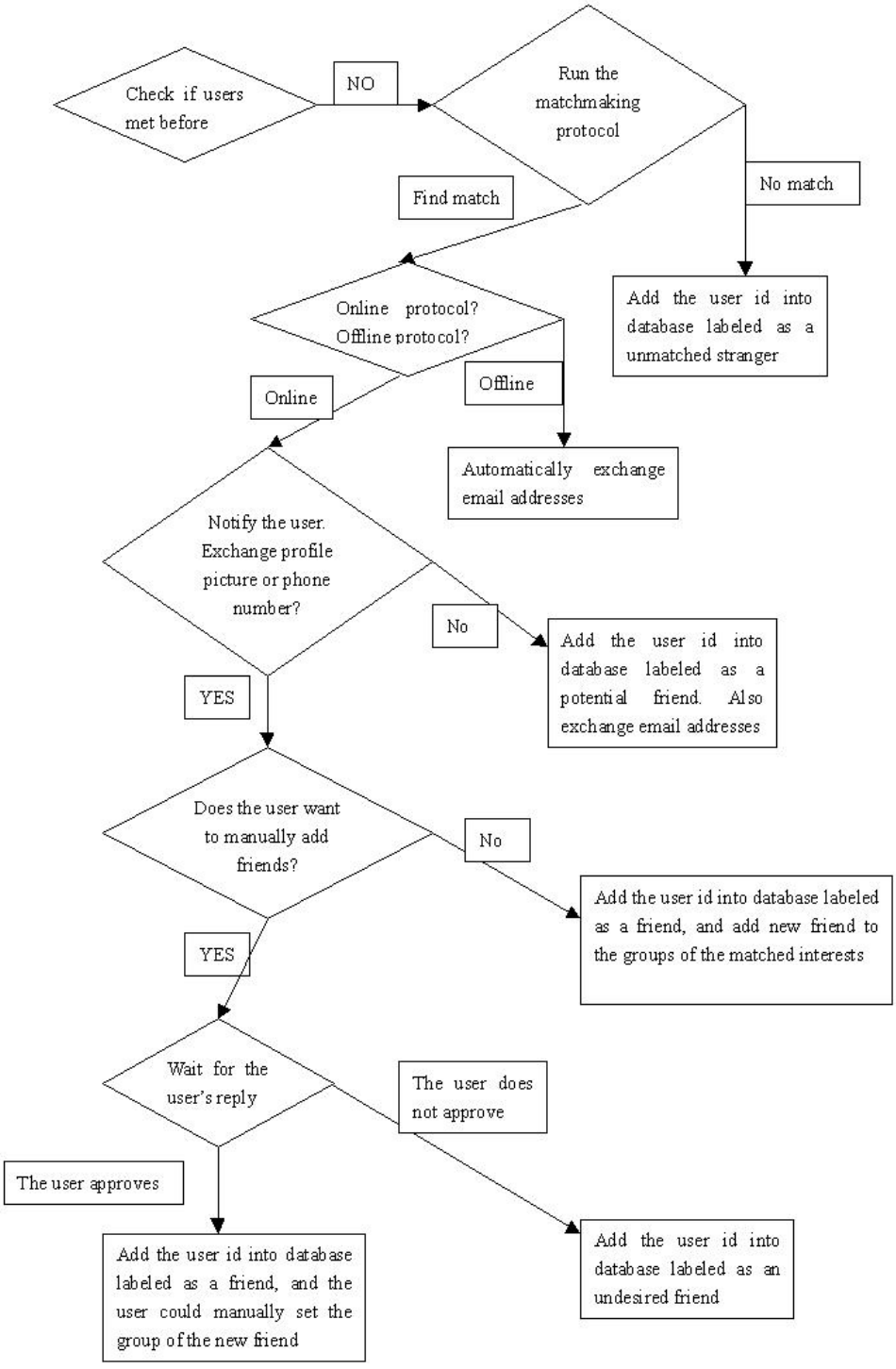

Figure 2.2: Flow Chart for First Encounter

party does not want to provide such information, they will not establish friendship, but they will exchange a public key for the next meeting. In this case, the application records each other's user id in local databases, marks them as a potential friend, and stores their public key. If neither user sets to exchange the profile picture or cell phone number, then they are treated as running the offline protocol. Again, users could report to a trusted third party if they find out that somebody sent them fake information (e.g., some people may use other people's pictures as their profile pictures). As a result, failure to provide valid information could cause suspension of service. Ideally, a trusted third party should sign the contact information and/or profile pictures for the users. After the contact information or pictures is exchanged under an online protocol, the system checks if a user wants to manually add a friend (notification field). If a user sets to not manually add new friends in the Config Module, the application automatically adds the other party's user id into the database of her device as a new friend. Otherwise, the application adds the other party as a new potential friend and notifies the user. The user will decide if she wants to change it to a friend. It also records the public key for future contact. Users can also group their new friends using the Group Module. The flow chart for what happens when two users meet for the first time is shown in Figure 2.2.

2.2.2 Meet Again

If the other party's user id exists in a user's local database, it means that they have met before. If the user id exists as a friend, the user will be notified that there is an existing friend nearby, and the devices are not going to run a matchmaking protocol again. If the user id exists as a potential friend, the devices are going to check if the owner wants to add the other party to be a friend this time. If the user id exists as an unmatched stranger, it means that they do not share any common interest. In this case, the devices check if there is an update of their interests since their last meeting. This could be done by requiring users to record the updates (both the changes of their interests and the time when the changes are made) of their own interests (Note that the Matchmaking Module also records the time when two users meet each other). Then, they inform each other whether there is any update. If there is no update, they simply do not exchange any other information. Otherwise, they run the matchmaking protocol again using the new information. Since there could be many unmatched strangers, our device will keep only the most frequently encounted user ids among the most recent encounters. For those who we meet once in a while, it does not hurt to run the matchmaking protocol again, if their user ids are erased from the database (because of the limited size of the device's memory). Users run the matchmaking protocol only when their user ids do not exist in anyone's database. The flow chart for what happens when two users meet again is shown in Figure 2.3.

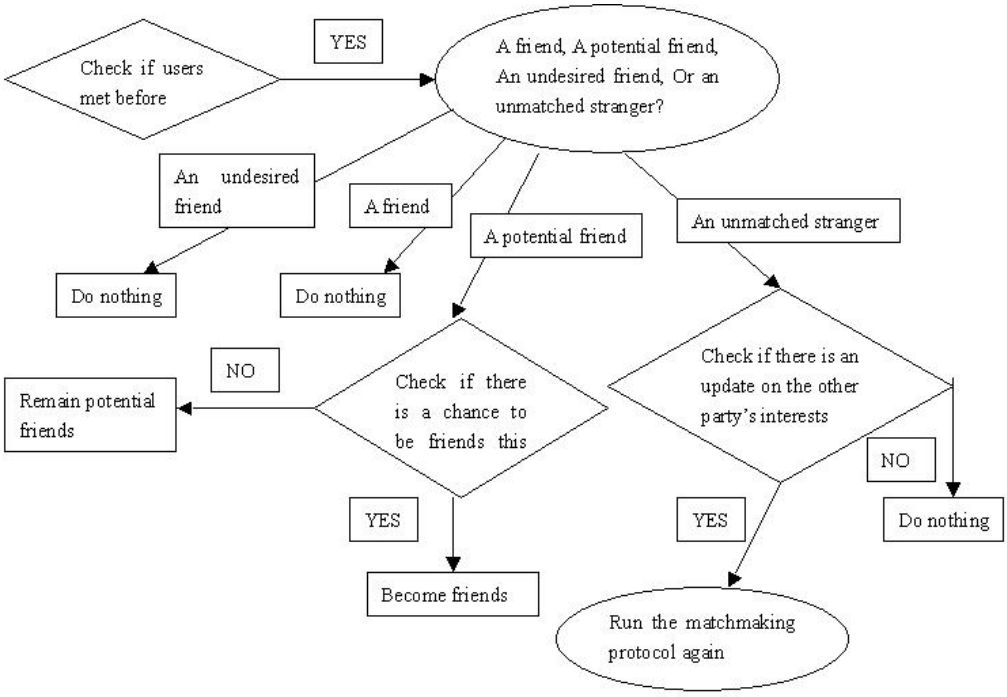

Figure 2.3: Flow Chart for Meet Again

2.3 Threat Model

In our architecture, we provide various protocols and handle various threats. In general, all of our protocols prevent attackers from

1. Getting users' interests without getting caught for cheating unless they actually own the same interests. We provide a detailed analysis later in the thesis to prove this statement.

2. Exploring users' interests by including all possible or a large number of popular elements in their interest set (brute force attack). We allow users to create only a limited number of interests. This is reasonable because a normal user should not have too many interests. Otherwise, people do not have time to deal with all of them. We eliminate the brute force attack by adding signatures to the matchmaking protocol. A third party server assigns signatures to only a limited number of interests for each user.

3. Impersonating other users. We ask each user to create a pair of asymmetric keys and use the hash value of the public key as her user id. The third party server will include a user's user id in a signed certificate. After two users meet each other, they first exchange their public keys, and they then negotiate a secret using each other's public key. They can derive a session key using this secret. This authenticates the partner of the protocol.

4. Eavesdropping the communication between any two users. Sensitive information is encrypted by the session key that two users established.

These are some threats we do not consider in our system and some assumptions that we make.

1. Users keep their private keys safe, so that malicious users could not steal their private keys to impersonate them.

2. The third party server is not compromised by attackers.

3. In our first protocol, we assume that most users are rational and they are honest but curious. This means that most users are not going to reveal information if it brings them negative effects.

4. Users trust that their matched friends will not disclose their matched information.

5. Users are going to finish running the protocols once they start.

6. We do not consider tracking attacks. In our architecture, we require users to use consistent user ids. This allows friends to recognize each other, and this makes tracking attacks possible. However, our communication is based on Bluetooth, which is a short-range communication technique. The attackers can track the users only when they are nearby. In other words, this requires the attackers to physically follow the users. A user is vulnerable to this kind of attack, no matter whether she is using our architecture or not. Even if we allow users to use different user ids, they will be still traceable. More details are provided in section 3.2.5.

7. For tampering attacks, we cannot distinguish between the user sending invalid data or a man in the middle tampering with the traffic. For example, in the case that a signature from Alice cannot be verified by Bob, Bob should report Alice to our *PIS* server (we will introduce this server in section 3.1.2). However, this does not necessarily mean that Alice sent invalid data; our *PIS* server assumes that Alice is malicious only if there are enough users who report Alice for the same problem.

Chapter 3

Matchmaking Protocols

In this chapter we introduce two matchmaking protocols for our architecture. For each protocol, we analyze the security properties.

The Matchmaking Module deploys these two matchmaking protocols. Its goal is to find potential friends for users, while preserving users' interests from unnecessary leaks. The users with non-empty intersection of their interests are considered to be potential friends.

3.1 Setup

3.1.1 Id Signer

An id signer is used to assign an identifier and a certificate of the identifier to each user. It is a trusted third party; however, this party learns only about identity information and nothing else. We need this trusted third party because there is no other trivial way to identify and authenticate a user. A user's id should be globally unique, and each user should be assigned only one id. The Bluetooth address could be used as an identifier for each Bluetooth device because it is free and globally unique. However, it is proven that the Bluetooth address can be modified [18]. IMEI is an identifier for a mobile phone. Theoretically, IMEI can be used as a user's id, but so far, we have not found any literature using IMEI in such a way. One possible reason is that it is not easy for a mobile device to validate a received IMEI. This could be addressed if the telecom companies issued IMEI certificates to their customers. If that is the case, the telecom companies are our id signers.

In short, we need an id signer to verify users' personal information and issue one user id and one certificate of the user id for one user. For example, a trusted certificate authority or CA (e.g.,

VeriSign) can act as an id signer. A CA issues a digital id for a user, including a public key, name and email address, name of the CA, serial number of the digital id, digital signature of the CA, and so on. Usually, it costs money to register a digital id, but people could use the digital id for other applications, such as secure email. In general, the CAs do not guarantee that they assign only one user id for each person/device, but this can be done if they co-operate with us. The CAs usually have detailed information about their customers (sufficient to identify a person). For example, if a customer requests a user id and the certificate for our system (a CA can just include any special label in a normal certificate to distinguish the certificates for our system from the normal ones), the CAs can check the history of this customer and refuse to issue a new user id for her if she already has one.

3.1.2 Personal Info Signer

When two users communicate with each other, they should show that they are actually using their "real" information. Since there is no way to guarantee that users use their "real" information, we limit the number of interests every user could use. By doing this, we prevent a user from detecting other users' information by including all possible interests. The *personal info signer* (*PIS*) is responsible for signing users' interests, so that a user could prove to other users that she is not using any random information, but certified information. A user should trust another user's information only if it is signed by the *PIS*.

The *PIS* provides a web page for users to create and/or look up the names of their interests. Before a user asks the *PIS* to sign her matchmaking information, she first looks up the web page to see if her interests already exist. For the interests that already exist, she just re-uses the existing ones. For the non-existing interests, she creates her own interest names on the page, and the *PIS* assigns an id for each interest. The ways how the *PIS* creates interest ids are different based on the matchmaking protocol that the users want to run. The reason why we request users to look up their interest information online is to standardize users' inputs, since it is harder to find a match when inputs are made in different people's preferred fashions (different wording, shorthand). For example, users may use the term "ping pong" or "table tennis" as the name of the same kind of sport. A user submits her valid identifier, which she gets from the id signer, and the names of her interests, which she gets from the web page, to the *PIS*. The *PIS* creates corresponding interest ids and signatures and then sends them back to the user.

The content to be signed is different depending on the matchmaking protocol users adopt. Generally speaking, the content to be signed includes a user's id, an interest id, and a time stamp (time stamps are used for revoking the users' signatures that are signed by the *PIS*). Failure to provide valid signatures or the signed content to other users results in no responses. The *PIS* keeps a database of users' identifiers and their interests and tracks the changes in users' interests. If a user frequently changes her interests, it is most likely that she does not use her real

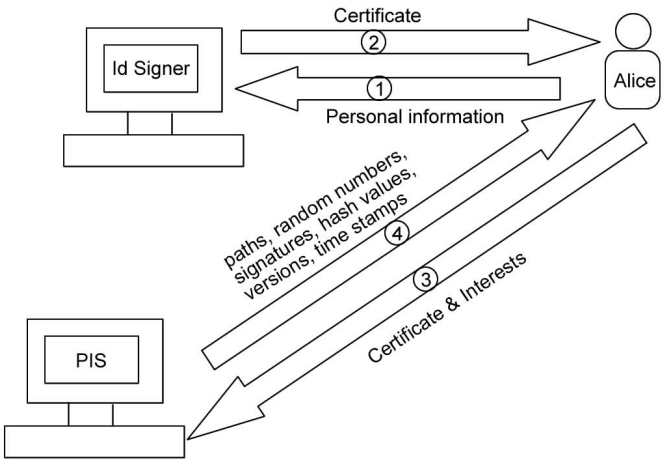

Figure 3.1: Fair Exchange Setup

information, and she just wants to explore other people's interests. To discourage this kind of behaviour, the *PIS* is going to give a warning to the user at the first time. If it happens again, the *PIS* is going to not renew signatures for this user, and as a result, the user is not able to run the matchmaking protocol with other users anymore.

3.2 Fair Exchange Protocol

The basic idea of the fair exchange protocol is for two users to exchange information step by step, and in each step they reveal only a limited amount of information to each other. The fair exchange protocol has two phases: the initial phase and the matchmaking phase.

3.2.1 Initial Phase

The initial phase takes two steps. The flow chart of the initial phase is shown in Figure 3.1.

The first step takes place between the id signer and a user (e.g., Alice). Alice sends her personal information to the id signer. Alice also generates a pair of RSA keys and sends the

public key to the id signer. The id signer identifies, authenticates, and issues a certificate (e.g., a digital id) to Alice. The certificate includes Alice's RSA public key and Alice's user id (a user uses the hash value of her RSA public key as her user id in this protocol). The id signer should issue exactly one certificate and one user id to one user.

The second step takes place between the *PIS* and a user (e.g., Alice). The *PIS* builds a binary tree. The number of leaf nodes of the tree is an integer exponent of base 2. The number of nodes is larger than the number of existing interests. The *PIS* assigns one interest to one leaf node. This tree is secret, so if Alice does not collude with any other user, she only knows the positions of the leaf nodes of the interests she has. Alice looks up the names of her interests or creates names for her interests as described in section 3.1.2. Alice submits the names of her interests and her user id, along with the certificate issued by the id signer to the *PIS*. The *PIS* verifies the certificate and makes sure that Alice did not misbehave before (based on other users' complaints). Here let's assume that Alice is an honest user, and no other users complained about her. Then, the *PIS* locates the interests in the tree and gets the paths from the root to the corresponding leaf nodes. Each interest can be identified by the path from the root to its leaf node. A path consists of several subpaths. Each subpath is either left or right to indicate the left or right branch an interest takes from one level to the next level. Let us denote the path of a user Alice to her interest Y P_{A_Y}, and the subpath of Alice's interest Y from level i-1 to level i $P_{A\_Y\_i}$. For a path in an L level tree, there are $L-1$ subpaths. The idea of the fair exchange protocol is to exchange the subpaths of interests level by level from the top to the bottom of the tree. A user assembles the subpaths she receives from the other party and checks if the subpaths down to the current level of each of her interests match any of the ones she received. The user stops sending the remaining subpaths of an interest to the other user when she fails to find a match for that interest.

Here is a scheme to guarantee that the users are using their actual subpath information. For each user, the *PIS* generates a random number with fixed length for each subpath of each interest (we denote the random number for $P_{A\_Y\_i}$ $R_{A\_Y\_i}$). The purpose of using random numbers is for commitment. Then, the *PIS* creates a signature for Alice's interest Y in the following fashion (we use $h()$ to denote a cryptographic hash function, and we use $s(A_Y)$ or $sign_{PIS\_pri\_key}(A\_ID\|c)$ to stand for computing a signature using the *PIS*'s private key on the content c for Alice): $s(A_Y) = sign_{PIS\_pri\_key}\left(A\_ID\|h\left(P_{A\_Y\_1}\|R_{A\_Y\_1}\right)\|...\|h\left(P_{A\_Y\_(L-1)}\|R_{A\_Y\_(L-1)}\right)\right)$. [1] The *PIS* also computes the $h(P_{A\_Y\_i}\|R_{A\_Y\_i})\ \forall\ i \in (0,L)$. The *PIS* sends the subpaths, the signature, the random numbers, and the hash values ($h(P_{A\_Y\_1}\|R_{A\_Y\_1})$,...,$h(P_{A\_Y\_(L-1)}\|R_{A\_Y\_(L-1)})$) back to Alice.

[1] An alternative is to use $HMAC_R(P)$ instead of $h(P\|R)$

3.2.2 Matching Phase

Let us use the single matching criterion example to demonstrate how matchmaking works. First, two users run a signature-based authentication protocol [25, p. 404] to identify and authenticate each other. The protocol is shown in Figure 3.2. In step 1, Alice sends a fixed-length random number, R_A, and her user id to Bob. Then, Bob also generates a fixed-length random number, R_B and creates a signature, $sign_{B\_pri\_key}(R_A \| R_B \| A\_ID)$. In step 2, Bob sends his public key, R_B, Alice's user id and the signature to Alice. After step 2, Alice verifies $sign_{B\_pri\_key}(R_A \| R_B \| A\_ID)$ using Bob's public key, and computes $B\_ID = h(B\_pub\_key)$ to guarantee that Bob's user id is the hash value of Bob's public key. Then, Alice creates a signature, $sign_{A\_pri\_key}(R_B \| R_A \| B\_ID)$. In step 3, Alice sends her public key, Bob's user id and the signature to Bob. After step 3, Bob is going to verify $sign_{A\_pri\_key}(R_B \| R_A \| B\_ID)$ using Alice's public key, and checks if $A\_ID = h(A\_pub\_key)$. They accept each other's user ids only if they can verify the signatures and the user ids are the hash values of their public keys. Then, the users run the fair exchange protocol shown in Figure 3.3. In the first two steps, each user sends the signature of his/her interest and the hash values issued by the *PIS* to the other party. Each user verifies that the signature he/she received is computed from the hash values and user id of the other party. If the signatures are invalid, they abort and the party who aborts the protocol reports the malicious behaviour by the other party to the *PIS*. Otherwise, they continue. In the rest of the protocol, after each user received information from the other party in each step, he/she first checks whether the interest has the same subpath information at the current level or not. Then, the users verify if $h(P_{X\_Y\_i} \| R_{X\_Y\_i})$ for subpath i of interest Y of user X (Alice or Bob) equals the corresponding hash value they received in step 1 or 2. Generally speaking, if they have different subpath information or the hash value cannot be verified, they abort. In the latter case, the party who aborts should also report the other party's malicious behaviour to the *PIS*. Otherwise, they continue and exchange the subpath of the next level. In fact, Alice and Bob behave a little differently. Alice aborts as soon as she finds out that Bob's subpath does not match hers. However, Bob should still send the subpath of the current level to Alice even though his subpath of the current level is different from that of Alice's. This is designed for the fairness of this protocol and ensures that Alice and Bob learn the same amount of information.

Figure 3.4 is a multiple matching criteria example. Let us assume that Alice has m interests, and Bob has n interests. For each of a user's interests, the user has one path. As a result, Alice and Bob are going to learn the number of each other's interests. After step 2, at each step, Alice sends the subpaths of a level of her interests that from her point of view Bob could also have. In step 3, Alice sends all subpaths of the first level of her interests to Bob. After step 3, Alice does a matching between the subpaths she has and the ones she received from Bob, and only sends certain subpaths of the current level of her interests to Bob. These subpaths belong to the interests that are the same as Bob's subpaths down to the previous level. Bob responds to Alice with the subpaths of all his interests that they have in common down to the previous level. After

R_i: A fixed length random number generated by user i
Alice's inputs: R_A, A_ID, A_pub_key, and A_pri_key
Bob's inputs: R_B, B_ID, B_pub_key, and B_pri_key

1. Alice $\xrightarrow{R_A \| A\_ID}$ Bob
2. Alice $\xleftarrow{B\_pub\_key \| R_B \| A\_ID \| sign_{B\_pri\_key}(R_A \| R_B \| A\_ID)}$ Bob
3. Alice $\xrightarrow{A\_pub\_key \| B\_ID \| sign_{A\_pri\_key}(R_B \| R_A \| B\_ID)}$ Bob

Figure 3.2: Signature-Based Authentication [25, p. 404]

that Bob does a matching between Alice's subpaths and the ones he has to find out what they have in common down to the current level. Later, we will illustrate how to decide which subpaths should be sent to the other party. After Bob performs the i^{th} comparison, Bob has n_i interests that share the same subpaths down to the current level with those of Alice. After Alice performs the i^{th} comparison, Alice has m_i interests that share the same subpaths down to the current level with those of Bob. Again, the users should perform the signature-based authentication before they run this matchmaking protocol. Similar to the previous example, in the first two steps, users exchange the signatures of their interests and the hash values. They verify if the signatures are computed from the hash values and the user id of the other party. They continue executing the protocol only if the signatures are valid. In the rest of the protocol, each user sends the subpath information of a certain level of her interests to the other party. Each user also keeps a log of the current received subpaths for each interest. This log is used to reconstruct the other party's paths. After a user receives new information from the other party, she assembles the currently received subpaths into the log. Note that for each interest she has, she checks whether any of her interests has the same subpath information down to the current level as the received ones. Then, she verifies if $h(P_{X\_Y\_i} \| R_{X\_Y\_i})$ for subpath i of interest Y that user X (Alice or Bob) has equals the corresponding hash value she received in step 1 or 2. A user stops sending the other party the interest whose subpaths do not match any of the received ones. An example is provided to help explain this in the next paragraph. They continue and exchange the subpath information of the remaining interests that do match. If any hash value cannot be verified, they abort.

Figure 3.5 shows an example of how users log received subpaths and how users decide which subpaths they are going to send. We remove the random number, signature, and hash values from the example for the sake of simplicity. In step 1, Alice sends the subpaths of the first level of her interests to Bob. Bob logs Alice's subpaths. Bob responds to Alice with all his subpaths. Then, he checks his interests and finds that Alice does not have his interest 1 because neither subpath from Alice's interests starts with "r". In the remaining steps, Bob will not send Alice

Alice's inputs: Interest I, $R_{A\_I\_i} \forall i \in (0,L)$, $h(P_{A\_I\_i} \| R_{A\_I\_i}) \forall i \in (0,L)$, $s(A_I)$= $sign_{PIS\_pri\_key}(A\_ID \| h(P_{A\_I\_1} \| R_{A\_I\_1}) \| ... \| h(P_{A\_I\_(L-1)} \| R_{A\_I\_(L-1)}))$, $P_{A\_I\_i} \forall i \in (0,L)$, A_ID, B_ID

Bob's inputs: Interest I', $R_{B\_I'\_i} \forall i \in (0,L)$, $h(P_{B\_I'\_i} \| R_{B\_I'\_i}) \forall i \in (0,L)$, $s(B_{I'})$= $sign_{PIS\_pri\_key}(B\_ID \| h(P_{B\_I'\_1} \| R_{B\_I'\_1}) \| ... \| h(P_{B\_I'\_(L-1)} \| R_{B\_I'\_(L-1)}))$, $P_{B\_I'\_i} \forall i \in (0,L)$, B_ID, A_ID

1. Alice $\xrightarrow{h(P_{A\_I\_i} \| R_{A\_I\_i}) \forall i \in (0,L) \| s(A_I)}$ Bob
2. Alice $\xleftarrow{h(P_{B\_I'\_i} \| R_{B\_I'\_i}) \forall i \in (0,L) \| s(B_{I'})}$ Bob
3. Alice $\xrightarrow{P_{A\_I\_1} \| R_{A\_I\_1}}$ Bob
4. Alice $\xleftarrow{P_{B\_I'\_1} \| R_{B\_I'\_1}}$ Bob

... ...

2L-1. Alice $\xrightarrow{P_{A\_I\_(L-1)} \| R_{A\_I\_(L-1)}}$ Bob

2L. Alice $\xleftarrow{P_{B\_I'\_(L-1)} \| R_{B\_I'\_(L-1)}}$ Bob

Note. Alice and Bob verify the signatures as soon as they receive the hash values and signatures from each other. They will execute the remaining steps only if the signatures are valid. For steps 3 - 2L, Alice aborts the protocol if she fails to verify the commitment or the subpath she receives does not match the one she has. Bob aborts the protocol if he fails to verify the commitment; however, he should still send the current subpath to Alice when the subpath he receives does not match the one that he has (for the sake of fairness).

Figure 3.3: Fair Exchange protocol (Single Matching Criterion)

Alice's inputs: Interest I_j $\forall j \in (0,m]$, $R_{A\_I_j\_i}$ $\forall i \in (0,L)$ $\forall j \in (0,m]$, $h(P_{A\_I_j\_i} \| R_{A\_I_j\_i})$ $\forall i \in (0,L)$ $\forall j \in (0,m]$, $P_{A\_I_j\_i}$ $\forall i \in (0,L)$ $\forall j \in (0,m]$, A_ID, $s(A_{I_j})$= $sign_{PIS\_pri\_key}(A\_ID \| h(P_{A\_I_j\_i} \| R_{A\_I_j\_i}) \forall i \in (0,L))$ $\forall j \in (0,m]$, B_ID

Bob's inputs: Interest I'_j $\forall j \in (0,n]$, $R_{B\_I'_j\_i}$ $\forall i \in (0,L)$ $\forall j \in (0,n]$, $h(P_{B\_I'_j\_i} \| R_{B\_I'_j\_i})$ $\forall i \in (0,L)$ $\forall j \in (0,n]$, $P_{B\_I'_j\_i}$ $\forall i \in (0,L)$ $\forall j \in (0,n]$, B_ID, $s(B_{I'_j})$= $sign_{PIS\_pri\_key}(B\_ID \| h(P_{B\_I'_j\_i} \| R_{B\_I'_j\_i}) \forall i \in (0,L))$ $\forall j \in (0,n]$, A_ID

1. Alice $\xrightarrow{(h(P_{A\_I_j\_i} \| R_{A\_I_j\_i}) \forall i \in (0,L) \| s(A_{I_j})) \forall j \in (0,m]}$ Bob
2. Alice $\xleftarrow{(h(P_{A\_I_j\_i} \| R_{A\_I_j\_i}) \forall i \in (0,L) \| s(A_{I_j})) \forall j \in (0,n]}$ Bob
3. Alice $\xrightarrow{P_{A\_I_j\text{-}1} \| R_{A\_I_j\text{-}1} \forall j \in (0,m]}$ Bob
4. Alice $\xleftarrow{P_{B\_I'_j\text{-}1} \| R_{B\_I'_j\text{-}1} \forall j \in (0,n]}$ Bob

... ...

2L-1. Alice $\xrightarrow{P_{A\_I_j\text{-}(L-1)} \| R_{A\_I_j\text{-}(L-1)} \forall j \in (0,m_{L-2}]}$ Bob

2L. Alice $\xleftarrow{P_{B\_I'_j\text{-}(L-1)} \| R_{B\_I'_j\text{-}(L-1)} \forall j \in (0,n_{L-2}]}$ Bob

Note. Alice and Bob verify the signatures as soon as they receive the hash values and signatures from each other. They will execute the remaining steps only if the signatures are valid. For steps 3 - 2L, Alice aborts the protocol if she fails to verify the commitment or none of the subpaths she receives matches the ones she has. Bob aborts the protocol if he fails to verify the commitment; however, he should still send the subpaths of the current level of all the interests that he and Alice have in common down to the previous level to Alice (for the sake of fairness).

Figure 3.4: Fair Exchange protocol (Multiple Matching Criteria)

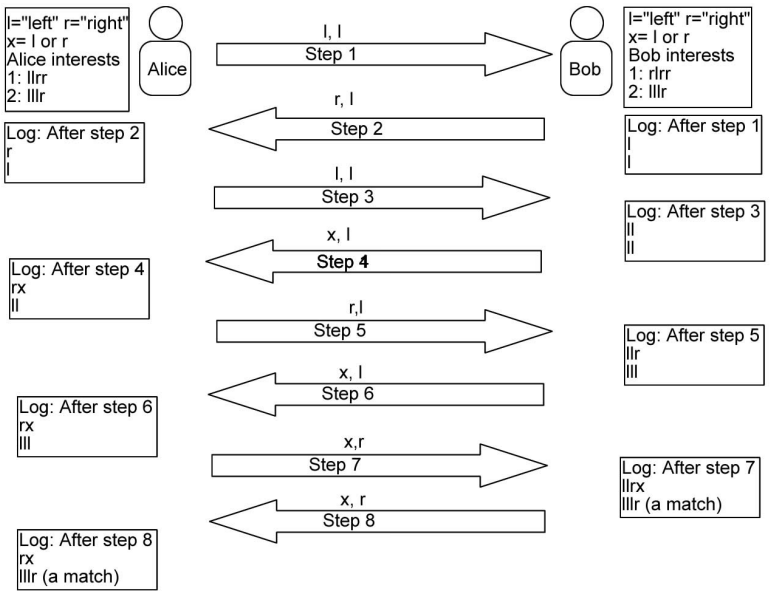

Figure 3.5: An Example of Multiple Matching Criteria

the subpaths of his interest 1, because there is no way that Alice could have the same interest. A similar thing happens after step 6. Alice knows that none of Bob's interests starts with subpaths "llr", so she does not send the next level subpath of her interest 1 to Bob. The users keep tracking the subpaths they receive for each interest of the other party. If they have not received an "x" (the "x" acts as a placeholder so that the users so that users do not need to worry about the ordering of the interests and can simply send one character for each of their interests[2]) for an interest, it means that they could both have this interest, and they keep appending the subpaths of this interest when they receive it. If a user receives an "x" for an interest, it means that the other party knows that they cannot both have this interest.

Users should negotiate a session key when they reach the middle of the tree. Krawczyk [20] provides several authenticated Diffie-Hellman protocols for users to negotiate a session key. Users should encrypt all subpath and random number information located in the lower half of the tree to prevent other users from eavesdropping the information. Users could negotiate a session key when they start the protocol and encrypt all messages between them; however, the upper half of the subpath and random number information is less sensitive. Namely, even if other users learn this information, the real interest of a user is still protected by a large number of other interests. In addition, it is more expensive to encrypt and decrypt all messages. We could use an authenticated Diffie-Hellman protocol as our signature-based authentication protocol. However,

[2]As an alternative, it is possible to leave the "x" away and to include additional meta-information.

in our implementation and evaluation, we separate these two steps. Namely, two users run the signature-based authentication first, and then they exchange the subpath information until the middle of the tree is reached. After that, they run the authenticated Diffie-Hellman protocol and compute a common session key. Finally, they exchange their remaining subpath information located under the middle of the tree, and before they send out the information, they encrypt it using the session key. One reason for doing this is that we are not sure in real life how often two users are going to exchange their subpaths down to the middle of the tree. It is more expensive to generate a Diffie-Hellman key pair than simply generating a random number. If it is not a common case that users will exchange their subpaths down to the middle of the tree, then we should separate these two steps, since in most cases, users do not even need to negotiate a session key. We will do more study in the future and find out which way is the better approach.

Note that Alice and Bob are re-using the same sets of random values when they encounter different users until they renew their signatures. Arguably, this is dangerous, since after Alice and Bob run this protocol, they learn at least part of each other's random values. What if they disclose this information to other users? Here, we have to assume that Alice and Bob would not do such a thing. In fact, even if we require users to use different sets of random values when meeting different users, a user could disclose another user's subpath information to other users instead of the random values. In the most extreme example, Alice and Bob have an interest in common. Nobody could stop Alice from telling Carol that Bob has such an interest, no matter what matchmaking protocol they use.

3.2.3 Signature Renewal, Signature Revocation and Tree Update

As mentioned, time stamps should be included in the signed content. If a user is caught cheating, the *PIS* is not going to renew the signatures for this user anymore. If possible, users should renew their signatures every day, and only accept the most up-to-date signatures. However, in our architecture, we relax this requirement and let users tolerate out-of-date signatures. To tolerate out-of-date signatures means that a user may have more chances to interact with detected malicious users. In section 3.2.4, we show that if the tree structure for the interests is known by attackers, our protocol may reveal users' interests. Therefore, the *PIS* reconstructs the tree structure every day to mitigate the effect of leaking the tree structure. Users who want to tolerate out-of-date signatures should keep their out-of-date paths, random numbers, and signatures. Otherwise, they are not able to find appropriate potential friends because of changes to the tree structure.

How can users decide about the issue date of each other's signatures and thus which version of paths, random numbers, and signatures to exchange with each other? Here we propose a protocol for users to negotiate and decide which information they want to send out. The *PIS* maintains a version number for each tree structure. Note that the tree structure remains the same if only

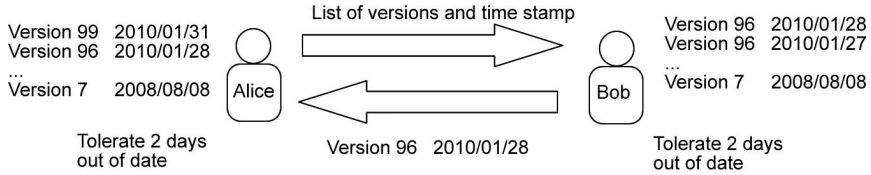

Figure 3.6: Users with Multiple Version of Tree Structure

more matchmaking information is added to the tree leaves. In other words, the version number of a tree structure changes only when the *PIS* re-shuffles the tree nodes or extends the level of the tree. The *PIS* also informs the users about the time stamps included in their signatures. When two users meet, they should first exchange the version numbers of the trees that they possess. Then, they should load paths, random numbers, and signature information from the most recent version tree (most recent version and time stamp) that they have in common. A user can refuse to exchange information with the other user if she thinks that none of the other user's information is tolerable. If a user can tolerate two days of out-of-date signatures, it means that she accepts signatures that are signed within two days from her most recent signatures. Figure 3.6 shows an example of how two users with different versions of tree information perform the fair exchange protocol. In this example, Bob would accept Alice's version 96, which is signed on 2010/01/28; however, Alice would refuse to accept any of Bob's versions, because Bob's newest signature is issued on 2010/01/28, which is more than two days from 2010/01/31. Note that in Figure 3.6 we do not assume that the *PIS* reconstructs the tree structure every day.

3.2.4 Security Analysis

In this section, we at first analyze the security of our protocol under the following assumption: all users know only the path information of their own interests. Arguably, a user could publish the path information of her interests to a publicly accessible place, and as a result, all other users learn the corresponding paths. Later, we use game theory to prove that if the users are rational, they are not going to reveal the paths of their interests to other users. Then, we also study the case when this assumption does not hold. Furthermore, we assume that the users do not leak each other's subpath information. The reason for this assumption is mentioned in section 3.2.2. In section 3.3, we provide a more secure protocol, which does not require these assumptions. However, that protocol has more computational overhead, and according to our evaluation, that protocol takes longer to complete under the same setting.

Here are two properties of the fair exchange protocol.

First, a user learns all interests that she has in common with the other party; otherwise the other party cheats, and the cheating behaviour would be detected. The users exchange the com-

mitment before they exchange the paths for their interests. Based on the fact that it is hard to find $h(\text{“}l\text{”}\|random\_number\_1) = h(\text{“}r\text{”}\|random\_number\_2)$ (“l” and “r” represents the left and right subpath, and the random numbers have a fixed length), the users could not lie about their subpaths without being caught. Note that there is an exception when the initiator stops sending the actual subpath information and sends “x” instead for her interests even though the responder still could have these interests. However, in this case, nobody would gain more information than the other party. Since we also assume that the main purpose that users use our system is to look for new friends, but not to explore other users' interests, we did not consider this exception in our design. Nevertheless, it is easy to prevent this exception from happening. We could require the initiator to still send the actual subpath of an interest to the responder at the level she just found that the responder does not have the interest.

Second, a user cannot learn the interests that she does not have in common with the other party, unless she cheats, and the cheating behaviour would be detected. Obviously, an attacker cannot listen to the communication between two users or replay other users' messages to get useful information. The reason is that all sensitive information is encrypted by a session key known only by the two users in the conversation. A user cannot use any uncertified interest/path information as her input, because the signatures of the interests/paths are verified in the first two steps and also because of the property of the cryptographic hash functions. The only possible attack we could find is the guessing attack where an attacker tries to guess a user's interests based on the subpath information that the attacker obtains from the user.

We define security as the anonymity of Alice's unmatched information after she runs the protocol with a user, Eve. In the analysis, we are going to use the anonymity metric mentioned in [19], $Anonymity = \frac{H(x)}{H_{max}} = \frac{\sum -P(x)\log(P(x))}{\log(N)}$. N is the number of interests in this case, and $P(x)$ is the probability that an interest belongs to Alice in Eve's point of view. Assume that Alice has 1 interest and Eve has m interests. The number of all interests in the tree is z ($z \gg m$), the depth of the tree is d, and Alice and Eve do not have any interests in common.

The protocol is perfectly secure if the assumptions hold that a user does not leak her and other users' (sub)paths. $Anonymity = \frac{(z-m)\times(-\frac{1}{z-m}\times\log(\frac{1}{z-m}))}{\log(z)} = \frac{\log(z-m)}{\log z}$. This is the best result any private intersection protocol could get. It is reasonable to assume that it is hard for other users to learn a large portion of the tree structure, if we assume that most users value preserving their own privacy higher than exploring other people's privacy. However, nobody could stop users from posting the paths representing their interests to a publicly accessible place. If enough users do so, then the tree structure could be reconstructed. Reconstructing the tree structure requires a large number of users to publish the paths for their own interests, and we are going to prove that there is no motivation for them to do so, because it brings negative benefits to the publishers. We are going to prove that if a user publishes one of her paths, then her value

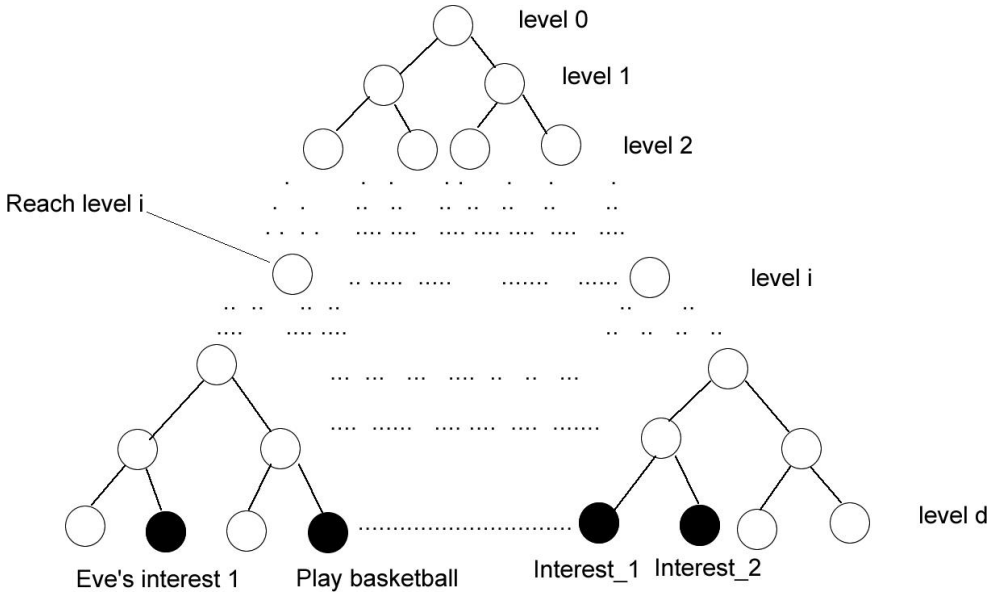

Figure 3.7: Tree

of the *Anonymity* for the given interest decreases.

Let us take a look at the change of the value of *Anonymity* for a user's interest after she publishes the path of her interest. For example, a user, Alice has an interest, play basketball. She runs the protocol with Eve and reaches the i^{th} level of the tree. Before she publishes the path of her information, $Anonymity = \frac{\log(z-m)}{\log z}$ as shown above, because the probability of each interest belonging to Alice (except the m interests Eve has) is $\frac{1}{z-m}$. After Alice or somebody else publishes the path of the interest, play basketball, to an publicly accessible place, from Eve's point of view, the possibility of the published interest belonging to the Alice is $\frac{1}{2^{d-i}-1}$ (assume that Eve has only 1 interest that shares the same subpath down to the i^{th} level of the tree with Alice, so Eve knows that Alice's interest is definitely not the one Eve has). The possibility of any other interest belonging to Alice is $(1-\frac{1}{2^{d-i}-2}) \times \frac{1}{z-m-1}$. Fig 3.7 shows this example.

$$
\begin{aligned}
&Anonymity \\
&= \frac{(z-m-1) \times \left(-\left(\left(1-\frac{1}{2^{d-i}-2}\right) \times \frac{1}{z-m-1}\right) \times \log\left(\left(1-\frac{1}{2^{d-i}-2}\right) \times \frac{1}{z-m-1}\right)\right) - \frac{1}{2^{d-i}-1} \log \frac{1}{2^{d-i}-1}}{\log z} \\
&= \frac{\left(-\left(\left(1-\frac{1}{2^{d-i}-1}\right)\right) \times \log\left(\left(1-\frac{1}{2^{d-i}-1}\right) \times \frac{1}{z-m-1}\right)\right) - \frac{1}{2^{d-i}-1} \log \frac{1}{2^{d-i}-1}}{\log z}
\end{aligned}
$$

$$= \frac{\left(\frac{2^{d-i}-2}{2^{d-i}-1}\right) \times \log\left(\left(\frac{2^{d-i}-1}{2^{d-i}-2}\right) \times (z-m-1)\right) + \frac{1}{2^{d-i}-1}\log(2^{d-i}-1)}{\log z}$$

$$= \frac{\log\left(\left(\left(\frac{2^{d-i}-1}{2^{d-i}-2}\right) \times (z-m-1)\right)^{\frac{2^{d-i}-2}{2^{d-i}-1}} \times \left(2^{d-i}-1\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z}$$

$$= \frac{\log\left(\left(\frac{(z-m-1)\times\left(2^{d-i}-1\right)}{2^{d-i}-2}\right)^{\frac{2^{d-i}-2}{2^{d-i}-1}} \times \left(2^{d-i}-1\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z}$$

$$< \frac{\log\left(\left(\frac{(z-m)\times\left(2^{d-i}-1\right)}{2^{d-i}-2}\right)^{\frac{2^{d-i}-2}{2^{d-i}-1}} \times \left(2^{d-i}-1\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z}$$

$$= \frac{\log\left(\left(\left(\frac{(z-m)\times\left(2^{d-i}-1\right)}{2^{d-i}-2}\right)^{2^{d-i}-2} \times \left(2^{d-i}-1\right)\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z}$$

$$= \frac{\log\left(\left((z-m)^{2^{d-i}-2} \times \frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}}\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z}$$

Anonymity decreases after a user publishes her paths if $\frac{\log\left(\left((z-m)^{2^{d-i}-2} \times \frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}}\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z} < \frac{\log(z-m)}{\log z}$ holds.

$(z-m>1)\&(z>1)$

$$\frac{\log\left(\left((z-m)^{2^{d-i}-2} \times \frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}}\right)^{\frac{1}{2^{d-i}-1}}\right)}{\log z} < \frac{\log(z-m)}{\log z}$$

$$\Leftrightarrow \left((z-m)^{2^{d-i}-2} \times \frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}}\right)^{\frac{1}{2^{d-i}-1}} < (z-m)$$

$$\Leftrightarrow \left((z-m)^{2^{d-i}-2} \times \frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}} \right) < (z-m)^{2^{d-i}-1}$$
$$\Leftrightarrow \frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}} < (z-m)$$

We are most interested in the case where Alice and Eve come close to the bottom of the tree, because the more information they exchange, the more significant effect publishing a path would bring. Let us take a look at the last 8 steps. When $i = d-8$,
$\frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}} = \frac{255^{255}}{254^{254}} \approx 692 < z-m$ (z should be more than two orders of magnitude greater than m, in our application m is usually less than 20 and z is more than 2000.) We have proved that at the 8^{th} last level, $\frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}} < z-m$, and since $\frac{(2^{d-i}-1)^{2^{d-i}-1}}{(2^{d-i}-2)^{2^{d-i}-2}}$ decreases as the value of i increases, if a user exchanges her path information with an attacker down to the last 8 levels of the tree, the user's value of *Anonymity* for the given interest decreases if her path information is known by the attacker.

Let us turn this problem into a normal form game. A normal form game consists of a finite set of players. Each player could take a finite number of actions. The outcome for each player is computed depending on the actions each player takes. In our problem setting, when a user's value of *Anonymity* increases by taking an action, she has positive outcome. She has negative outcome when her value of *Anonymity* decreases. In addition, a user gets positive outcome when any other user's value of *Anonymity* decreases. Assume that users care more about their own privacy than exploring other users' privacy (so users stop exchanging their information with each other when the information does not match). In our case, we define the players as the users of our application. They could choose to either publish or not publish their interests to a publicly accessible place. In this case, Nash equilibria are defined for each player as the actions that she takes that always generate at least as good an outcome as taking any other actions, no matter what actions the other players take.

Let us take a look at two arbitrarily chosen players, Alice and Bob. What will their outcomes be when they choose to publish or not to publish their paths? Assume that at the moment, no other players publish their paths. As shown in both Table 3.1 and 3.2, Alice and Bob have better outcomes when they choose not to publish their information (Nash equilibrium). For example, in Table 3.1, when Alice publishes her path information but Bob does not, Alice's value of *Anonymity* decreases, and as a result, Bob has a better chance to guess Alice's interest right. So in this case, Alice has a negative outcome while Bob has a positive outcome. In Table 3.2, we have a slightly different situation. After Alice publishes the path of her interest, Bob's value of *Anonymity* also decreases. Bob certainly has a better chance to guess Alice's interest right, while Alice or any other user also has a better chance to guess Bob's interest right. Since we assume that a user values pretecting her own information higher than exploring other users' information,

		Bob	
		Publish	Not Publish
Alice	Publish	-1, -1	-1, 1
	Not Publish	1, -1	0, 0

Table 3.1: Publishing Alice's path does not decrease Bob's *Anonymity*

		Bob	
		Publish	Not Publish
Alice	Publish	-1, -1	-1, -1
	Not Publish	-1, -1	0, 0

Table 3.2: Publishing Alice's path does decrease Bob's *Anonymity*

both users have negative outcomes in this case.

Of course, there could be some people who post their information regardless of the consequences, but the number of such people should be relatively small if users are rational and their utility function follows our assumption. However, a user's utility function does not necessarily follow our assumption. For example, a group of reporters may collude and just share the path information among themselves to guess a famous person's interests. In this case, the reporters obviously value exploring other people's privacy higher than other purposes (but average users do not need to worry about this), and our previous analysis is not true. This is a motivation for introducing our second matchmaking protocol, which is against the guessing attacks. However, we also emphasize that it requires a large number of users to collude for the attack to be effective, since the *PIS* only assigns a limited number of interests to a user. We still consider the case when the entire tree structure leaks even though we have seen that it is unlikely to happen. Then the security of users' privacy also depends on: 1. the depth of the tree; 2. the portion of every path a user sent to the other user; 3. the context information; 4. the interests that are in each subtree. Here let us assume that a user has the same chance to choose any interest. We will discuss this assumption later in this section. A user will not reveal her exact interest to the other party until she sends the entire path of her interest to the other party. There is a slight chance that users with different interests could learn each others' information. It occurs only if the nodes representing their personal information are siblings in the bottom of the tree. Assume that Alice chooses $interest\_1$ and Bob chooses $interest\_2$ (the positions of $interest\_1$ and $interest\_2$ are shown in Fig 3.7). According to the protocol, they are going to reveal the entire paths to each other. The chance for this to occur is slim. Assume that both Alice and Bob are looking for one interest in common, and z is the number of total interests. The chance for the above case to occur is $\frac{1}{z}$. z is usually a large number, and even if z is small, the server could provide sample interests to enlarge the z value. From a user's point of view, she cannot tell whether an interest is a sample or

submitted by other users. Let us again use the previous example where Alice has 1 interest and Eve has m interests. The value of *Anonymity* of Alice's interest after she sends out the subpath information of her interest for the first i levels is $\frac{\sum -P(x)\log(P(x))}{\log(z)} = \frac{\log(2^{d-i}-m')}{\log(z)}$. It reduces to $\frac{\log(2^{d-i-1}-m'')}{\log(z)}$ after Alice sends out the subpath from the i^{th} level to the $(i+1)^{th}$ (m' is the number of Eve' interests located in the same sub-tree as Alice's interest at the i^{th} level and m'' is the number of Eve' interests located in the same sub-tree as Alice's interests at the $(i+1)^{th}$ level). It is important to compute the probabilities for different number of paths, which Alice shares with Eve, down to each level. For m interests, and 2^L paths, at each level, Alice possibly takes [0, $\min(m, 2^L)$] (we can assume that $2^L \gg m$) same subpaths to the next level with Eve. This makes the problem very hard to solve. We wrote a program to simulate this problem. From the result, we conclude that this approach can still provide privacy protection even though the whole tree structure is known by attackers. More details are provided in section 4.1.

The above value of *Anonymity* of Alice's interest at each level is computed assuming that each interest is chosen by Alice with the same probability; however in real life this is not always the case. For example, some interests are much more common than others. Also, usually people can be divided into groups based on their gender, age, and even race. Each group of people has a stereotype, and this possibly makes the success rate of attacks higher when a user's context information is known. For example, assume that an attacker knows that he is exchanging interest information with a young lady, and the young lady likes one of studying math, gardening, mountain climbing, and shopping. Then, the attacker may guess that the lady likes shopping, because that is the stereotype of young ladies. The popularity of different interests and stereotypes may help attackers to achieve higher success rates, but they could also help mislead the attackers. As a result, in our simulation, we assume that from the attackers' point of view, the chance of each interest chosen by a user is the same. Another reason for having this assumption is that right now we do not have any statistics about the popularities of different interests. We also do not know whether users would want to look for friends based on the popular interests (they would have tons of friends who have those interests). As a result, maybe there would be more users who look for friends based on the non-popular interests. In the future, we would like to do more user studies on this issue. For now, we can analyze only the case where we assume that all interests have the same popularity in the mobile social networking environment. users if they are worried about their privacy might be comprised because of the leaking of the tree structure.

3.2.5 A Failed Approach

We have tried to use multiple pseudonyms for the same person to mitigate the problem of the fair exchange protocol brought by leaking the tree structure. Suppose that Alice has several interests,

and she runs the protocol with Eve. In the case that Alice has at least one common interest with Eve, since Eve learns the tree structure, she is able to guess the rest of Alice's information, and thus learn extra information about Alice. If Alice could use different pseudonyms for different interests, from Eve's point of view, the interests would look like from different people. Even if Eve establishes friendship with Alice, Eve cannot tell which other information that she received belongs to Alice, and she cannot even tell whether or not Alice sent another piece of information. This protocol requires the presence of other people; otherwise, Eve could be certain that all pseudonyms belong to Alice. Users may use the Schedule Module to run this protocol when there are many people around. For example, if a user knows that there is a party at 3:00 pm, she may only turn on the protocol and look for new friends after the party starts. The presence of other people makes guessing attacks more unlikely to succeed. However, Eve could still detect whether two pseudonyms belong to the same person if she receives a pair of pseudonyms at the same time for several times. For example, Eve meets Alice several times, and she always sees those two pseudonyms whenever they meet. So it is necessary for the central server to assign a large number of pseudonyms (we call the pseudonyms for a certain interest *pseudonym pool*) for each interest. Alice sets different time slots to exchange different interests. Before sending out a piece of information, the user's device randomly chooses a pseudonym from the information's *pseudonym pool*, so that different interests appear from different users. The device should automatically change the Bluetooth address when using different pseudonyms. This approach requires more communication overhead and computational overhead. Also, in order to defeat replay attacks, a user has to prove that she actually owns this pseudonym. Using the hash value of a public key as a pseudonym is a solution. One problem is that the central server needs to assign dozens of public keys to each user for each of their interests and create digital signatures combining an interest with different pseudonyms. It increases the *PIS*'s burden; however, the *PIS* does not have to wait until it receives users' requests to update the users' signatures. If the *PIS* knows that a group of users are going to renew their signatures every day, it can create the signatures for them before they request the signatures and when the *PIS* is idle. Then, the *PIS* stores the signatures on its local disk, and sends them to the users when the users request them. (Note: users do not need to renew their public keys, but they have to renew the digital signatures.)

This approach does not work well because of two reasons. First, it becomes hard for users to recognize each other when they meet again. This adds lots of overhead, since there are lots of people we meet every day or quite frequently. By using this approach, we are going to run the matchmaking protocol with every pseudonym a user owns. Second, the relationship between different pseudonyms can still be trackable. Even though we use different pseudonyms for the same interest, the path of this interest does not change. For example, if a user detects that when he interacts with a number of pseudonyms, they always exchange their subpaths down to the 8^{th} level. This may indicate that these pseudonyms belong to the same user.

Alice's inputs: S_A, set key K_A
Bob's inputs: S_B, set key K_B

1. Alice computes $h(s_i)^{K_A} \forall s_i \in S_A$.
   Bob computes $h(s_i)^{K_B} \forall s_i \in S_B$.

2. Alice $\xrightarrow{h(s_i)^{K_A} \forall s_i \in S_A}$ *Bob*

3. Alice $\xleftarrow{h(s_i)^{K_B} \forall s_i \in S_B}$ *Bob*
   Alice $\xleftarrow{(h(s_i)^{K_A}, (h(s_i)^{K_A})^{K_B})\ (\forall h(s_i)^{K_A} \text{ received from Alice})}$ *Bob*

4. Alice computes $Z = (h(s_i)^{K_B})^{K_A}\ (\forall h(s_i)^{K_B}$ received from Bob)
   Alice creates $Y = (s_i, (h(s_i)^{K_A})^{K_B})\ \forall s_i \in S_A)$

5. Alice gets $s_i,\ \forall s_i \in (S_A \bigcap S_B)$ by intersecting the second element of Y and Z.

Figure 3.8: Original DDH Protocol [1]

3.3 DDH Protocol

We have seen that our fair exchange protocol requires some assumptions to hold to be secure. We have shown that the assumptions are reasonable. Also, our simulation shows that users' privacy could be still be protected in most case even if the tree structure is leaked. However, here we propose another protocol that provides stronger security guarantee for users who have higher security requirements.

This approach is similar to the one proposed by Agrawal et al. [1]. Let us do a quick review of their protocol. Assume the users are Alice and Bob. Alice has a set S_A, and Bob has a set S_B. Alice randomly chooses a secure key, K_A, and Bob randomly chooses a secure key, K_B. Their protocol is shown in Figure 3.8. In Figure 3.8 and the rest of the thesis, we use a^e to represent $a^e \mod p$, where p is a safe prime. p is a safe prime if and only if p is a prime number and $(p-1)/2$ is also a prime number. a is a quadratic residue modulo p. To be consistent with the notation in section 3.2, we use $h()$ to represent a cryptographic hash function. In step 2, Alice sorts the elements lexicographically and then sends them to Bob. In step 3, Bob also reorders the elements lexicographically and then sends them to Alice. In step 4, Alice creates $(s_i, (h(s_i)^{K_A})^{K_B}) \forall s_i \in S_A)$ by replacing $h(s_i)^{K_A}$ in the pairs that she received from Bob in step 3 with corresponding $s_i,\ \forall s_i \in S_A$. In step 5, Alice computes the intersection between the second element of the pair $(s_i, (h(s_i)^{K_A})^{K_B})\ \forall s_i \in S_A)$ and Z. The first elements of pairs whose second elements are in the intersection are the information Alice and Bob have in common.

This protocol has several problems. First of all, it does not deal with certified information. Second, Alice learns more information than Bob does, and she could lie about the result. Third, there is no guarantee that Bob will pair $(h(S_A)^{K_A}, (h(S_A)^{K_A})^{K_B})$ correctly and send the pairs back to Alice. We improve this protocol and make it fit our requirements. Our DDH protocol has two phases: initial phase and matchmaking phase.

3.3.1 Initial Phase

The initial phase takes two steps.

The first step takes place between the id signer and a user (e.g., Alice). Alice sends her personal information to the id signer. Alice also generates a pair of RSA keys and sends the public key to the id signer. The id signer identifies and authenticates Alice and issues a certificate to her. The certificate includes Alice's RSA public key. The id signer should only issue one certificate to one user. A user would use the hash value of her public key as her user id in this protocol. In summary, the id signer works the same as the one of the fair exchange protocol.

The second step takes place between the *PIS* and a user (e.g., Alice). The *PIS* is different from the *PIS* of the fair exchange protocol. The *PIS* generates a safe prime, p, the first time when it starts. When a user creates a name for a new interest (no other user has submitted the name yet), the *PIS* chooses a quadratic residue modulo p as the id of this interest. Assume that Alice submits the names of her interests, along with her certificate and her user id to the *PIS*. The *PIS* first makes sure that Alice did not misbehave before (based on other users' complaints). Here let us assume that Alice is an honest user, and no other users complained about her. Let us denote the ids of Alice's interests by $X_i \; \forall i \in (0,m]$ (assume that Alice has m interests). The *PIS* creates another certificate including Alice's id and a time stamp. This certificate is for Alice to provide to other users in the signature-based authentication to prove that she is a honest user up to the date when the certificate is created. The *PIS* sends $X_i \; \forall i \in (0,m]$, p, $sign_{PIS\_pri\_key}(A\_ID \| X_i) \; \forall i \in (0,m]$, and the certificate back to Alice. Before each matchmaking process, Alice randomly chooses a value a (a is in the range of $[1, q-1]$, and $q = (p-1)/2$) as her exponent, and computes the exponentiations, $(X_i)^a \mod p \; \forall i \in (0,m]$. Users could pre-compute the exponentiations for different exponent values when their devices are idle and store the results and corresponding exponent values for future uses.

3.3.2 Matchmaking Phase

Suppose that Alice has interests: X_1, X_2,...,X_m (m is the size of Alice's set), and Bob has interests: Y_1, Y_2,...,Y_n (n is the size of Bob's set). We are going to use X_i^a for $(X_i)^a \mod p$, $l(s)$ for the length of the string s, and $h()$ for a cryptographic hash function. Let us assume that in this

protocol, the party who initiates the communication reports the result first. Assume that Alice and Bob agree on a value g. Note that $user\_ID = h(pub\_key)$. Before running this protocol, users should run the signature-based authentication to identify and authenticate each other. Also, when they run the signature-based authentication, both users should exchange their certificates issued by the *PIS* to prevent communicating with malicious users. Figure 3.9 shows our DDH protocol. Steps 1 to 4 are straightforward. Step 5 is required for cheating detection purposes. R is a random number with fixed length used for commitment purposes. More details are provided in the next section. In step 6, Bob reports his computed results to Alice for matching. Alice sends the message to Bob in step 7 for two purposes. First, Alice needs to report her computed result to Bob, so Bob can perform the matching. Second, Alice needs to reveal the values that she committed to in step 5. There is no guarantee that Alice will pair $\left(Y_i^b, \left(Y_i^b\right)^a\right) \forall\ i \in (0,n]$ correctly in step 7. For example, Alice can pair Y_i^b with $\left(Y_j^b\right)^a$ for $i \neq j$. Later, we will show that Alice could take advantage from mispairing the data. Bob could also do the same trick to Alice. As a result, we require Alice and Bob to provide the signatures of the matched information later. More details are provided in the next section. Alice and Bob both compute the interests they have in common in step 8. They should not send plaintext of the ids and signatures of the matched interests to each other, because the information could be eavesdropped, and that is why we need step 9 to step 12. Alice and Bob negotiate a secret using authenticated Diffie-Hellman [20] to encrypt sensitive messages, such as their interest ids and corresponding signatures. We could have combined step 9 to step 12 with our signature-based authentication. However, it would be more expensive to compute g^a than to generate a random number, and we expect that the chance that two arbitrary users share at least one common interest is not high. Alice and Bob should exchange the signatures issued by the *PIS* of the interests that they have in common to prove that they really have the interests. After step 14, both Alice and Bob verify the signatures and record the messages if the other party fails to provide correct interest ids and signatures. Even though everything looks fine, both users should record the other party's messages at a given probability, so that at a later time they could submit the recorded messages to the *PIS* to check if the other party cheated or not. We explain why this is necessary in the next section.

3.3.3 Security Analysis

Assume that Alice sends messages before Bob does. Agrawal et al. prove that given DDH, $< x, f_e(x), y, f_e(y) > (f_e(x) = x^e)$ is indistinguishable from $< x, f_e(x), y, z >$, when e is not given. Further, they conclude that for polynomial t and k, $\binom{x_1 \cdots x_t \cdots x_k}{f_e(x_1) \cdots f_e(x_t) \cdots f_e(x_k)}$ is indistinguishable from $\binom{x_1 \cdots x_t \cdots x_k}{f_e(x_1) \cdots f_e(x_t) \cdots z_k}$. These properties give us the following results:

1. Bob cannot map x^a back to x if he does not know the value of a, which is only known by

1. Alice $\xrightarrow{l(X_1^a)\|X_1^a\|...\|l(X_m^a)\|X_m^a\|sign_{A\_pri\_key}\left(B\_ID\|l(X_1^a)\|X_1^a\|...\|l(X_m^a)\|X_m^a\right)}$ Bob

2. Alice $\xleftarrow{l(Y_1^b)\|Y_1^b\|...\|l(Y_n^b)\|Y_n^b\|sign_{B\_pri\_key}\left(A\_ID\|l(Y_1^b)\|Y_1^b\|...\|l(Y_n^b)\|Y_n^b\right)}$ Bob

3. Alice and Bob verify signatures.

4. Alice computes $(Y_i^b)^a,\ \forall\ i \in (0,n]$
   Bob computes $(X_i^a)^b,\ \forall\ i \in (0,m]$

5. Let $cm_A = l((Y_1^b)^a)\|(Y_1^b)^a\|...\|l((Y_n^b)^a)\|(Y_n^b)^a\|R$
   Alice $\xrightarrow{h(cm_A)\|sign_{A\_pri\_key}(B\_ID\|h(cm_A))}$ Bob

6. Let $cm_B = l((X_1^a)^b)\|(X_1^a)^b\|...\|l((X_m^a)^b)\|(X_m^a)^b$

   Alice $\xleftarrow{((X_i)^a,(X_i^a)^b),\forall i\in(0,m]\|sign_{B\_pri\_key}(A\_ID\|cm_B\|l(X_i^a)\|X_i^a\ \forall i\in(0,m])}$ Bob

7. Alice $\xrightarrow{(Y_i^b,(Y_i^b)^a),\forall\ i\in(0,n]\|R\|sign_{A\_pri\_key}(B\_ID\|cm_A\|l(Y_i^b)\|Y_i^b\ \forall i\in(0,n])}$ Bob

8. Bob gets $Y_i, \forall Y_i \in (Y_i^b)^a \bigcap (X_i^a)^b$
   Alice gets $X_i, \forall X_i \in (Y_i^b)^a \bigcap (X_i^a)^b$

9. Alice $\xrightarrow{g^a}$ Bob

10. Alice $\xleftarrow{g^b\|sign_{B\_pri\_key}(g^a\|g^b\|A\_ID)}$ Bob

11. Alice $\xrightarrow{sign_{A\_pri\_key}(g^b\|g^a\|B\_ID)}$ Bob

12. Alice computes $k = (g^b)^a$ & Bob computes $k = (g^a)^b$

13. Assume that there are m' interests in the intersection
    Alice $\xrightarrow{E_k(X_i\|sign_{PIS\_pri\_key}(A\_ID\|X_i)),\forall\ i\in(0,m']}$ Bob

14. Alice $\xleftarrow{E_k(Y_i\|sign_{PIS\_pri\_key}(B\_ID\|Y_i)),\forall\ i\in(0,m']}$ Bob

Figure 3.9: Our DDH Protocol

Alice. Proof: if Bob could map x^a back to x, Bob could compute $f_a^{-1}(z)$, and by checking whether $f_a^{-1}(z) = y$ or not, Bob can distinguish $< x, f_a(x), y, f_a(y) >$ and $< x, f_a(x), y, z >$. This makes sure that before Alice sends Bob the computed result, Bob is not able to get any of Alice's interests.

2. Given the value of $x_1, f_a(x_1), ..., x_t, f_a(x_t), ...x_{k-1}, f_a(x_{k-1})$, Bob cannot compute the value of a. Proof: assume that Bob could compute the value of a, so he is able to compute $f_a(x_{k-1})$, and by checking if $f_a(x_{k-1}) = z$, Bob could distinguish $\binom{x_1}{f_a(x_1)} \cdots \binom{x_t}{f_a(x_t)} \cdots \binom{x_k}{f_a(x_k)}$ from $\binom{x_1}{f_a(x_1)} \cdots \binom{x_t}{f_a(x_t)} \cdots \binom{x_k}{z_k}$. This guarantees that Bob cannot obtain Alice's exponent value, a. As a result, even if Bob collects the id of an interest that he does not have, he cannot compute id^a. He cannot detect if Alice has this interest by checking if id^a equals $X_i^a \; \forall i \in (0, m]$.

This protocol prevents Alice and Bob from learning any interest other than the interests that they have in common. If any user does learn extra interests, they are definitely going to be caught by the other party for cheating.

1. By sending out $\{Y_1^b, ...Y_n^b\}$, and getting back $\{X_1^a, ...X_m^a\}$ and $\{(Y_1^b, (Y_1^b)^a), ..., (Y_i^b, (Y_i^b)^a)\}$, Bob can learn only whether or not Alice has a number of elements in set $\{Y_1, ...Y_n\}$. According to the above two properties, if and only if $(Y_i^b)^a = (X_i^a)^b$, Bob could learn if X_i from Alice matches Y_i by performing step 8. Otherwise, it is impossible to get the value of X_i. Note that Bob cannot be certain which interests Alice really shares with him until they exchange the signatures signed by the *PIS* including the names of the interests. The reason is because it is impossible to map x^a back to x. In the above protocol, Alice could pair the Y_j^b with $\left(Y_i^b\right)^a$ for $j \neq i$, and Bob would think Alice has Y_j instead of Y_i if $\left(Y_i^b\right)^a$ is in the intersection. Bob would send Y_j to Alice. As a result, Alice is able to gain extra information without being detected. Exchanging the signatures of the original information prevents this attack from taking place. As a result, it is necessary for both users to exchange the signatures of the ids of the interests in the intersection. If a user cannot provide a valid signature signed by the *PIS* for all her interests in the intersection, the other user should record the protocol transcript and send it to the *PIS*. Because Alice receives messages from Bob that are symmetric to the messages that she sends to Bob, she is also not able to learn extra interests.

2. Users could not use information not signed by the *PIS* for them if they do not want to be banned. For example, Alice does not register X', but she sends $(X')^a$ to Bob, and Bob actually has X'. According to the protocol, at the end of the matching, both parties have to exchange the signatures of the interests that they have in common. These signatures are signed by the *PIS* including the interests and their user ids. In this case, Alice would not be

able to do that. Also, because Alice has to provide a signature including all exponents that she computed to Bob in step 1, Bob could provide this signature to the *PIS* as an evidence of Alice's misbehaviour. Alice is not able to compute $(X_i)^{a'} = (X')^a$ (assume that Alice has X_i), and the *PIS* is going to ban Alice. Note that in order to answer the *PIS*'s challenge, a user should keep a list of the exponents that she uses for different users.

3. Users cannot get useful information by replaying other users' responses. Users have to run signature-based authentication before they run the DDH protocol. This prevents attackers from using the signatures signed by other users.

4. Users cannot get useful information by listening to other users' communication, because all sensitive information is encrypted.

Both parties will learn all information that they have in common, unless any party cheats. The DDH protocol provides cheating detection methods. We are going to take a look at how the protocol detects cheating in two different cases. Let us assume Alice is malicious.

1. Alice wants only to explore Bob's information but does not want to find a new friend. She is going to send $sign_{A\_pri\_key}(B\_ID \| h(garbage \| R))$ instead of $sign_{A\_pri\_key}(B\_ID \| h(cm_A))$ in step 5. Later, Alice sends some "garbage" information to Bob instead of $(Y_i^a)^b \ \forall i \in (0,n]$. Bob is unable to find out since he is not able to map $(Y_i^a)^b$ back to Y_i^a. However, he is going to find out immediately if Alice refuses to run step 7. Bob is going to report Alice's behaviour to the *PIS*. If enough users report Alice's malicious behaviour, the *PIS* is not going to renew Alice's signatures any more. If Alice does run step 7 but with incorrect information, Bob is not going to find out Alice's malicious behaviour immediately; however, Alice takes the risk that Bob may record his message and report to the *PIS* later (As we mentioned before, both users should record the other party's messages at a given probability, so that at a later time they could submit the recorded messages to the *PIS* to check if the other party cheated or not). In this case, Bob has the signed information from Alice, so the *PIS* is certain that Alice cheated in this case (Alice cannot prove to the *PIS* how the values are computed from), and stops renewing the signatures of her interests and her certificate.

2. Alice wants to find new friends, but she wants to reveal only one interest if that is enough to find a new friend. If we remove step 5 from our protocol, Alice could easily achieve this goal. For example, Alice and Bob have two interests in common. After Alice receives $(X_i^b)^a \ \forall i \in (0,m]$ from Bob, she knows that. Alice can lie and reply only one matched result back to Bob. In this case, Alice finds a new friend and learns more information than Bob does. However, this will not happen in our protocol. Alice has to execute step 5 honestly, since she does not know what they have in common at this step. As a result, she

has to report $(Y_i^a)^b$ and R correctly to Bob, since it is hard to find $hash(x') = hash(x)$ for $x' \neq x$. Otherwise, Bob will detect her malicious behaviour immediately.

3.3.4 Cheaper DDH Protocol

The above protocol is secure, but it requires users' devices to perform twice the number of exponentiations of their set size for each matchmaking process (assuming that both users have the same set size). Is it possible to let the users re-use an exponent value for different matchmaking processes? Or is it possible to let the *PIS* perform some computation and thus reduce the client devices' loads? Figure 3.10 is a cheaper version of the DDH protocol. In the initial phase of this protocol, we require the *PIS* to select an exponent for a user and compute the exponentiation for each of the user's interests. For example, the *PIS* randomly generates an exponent, a (a is in range of $[1, q-1]$, and $q = (p-1)/2$) for Alice, and computes $(X_i)^a \; \forall \; i \in (0,m]$ and $sign_{PIS\_pri\_key}(A\_ID \| (X_i)^a) \; \forall \; i \in (0,m]$. The *PIS* sends $X_i \; \forall \; i \in (0,m]$, a, p, $(X_i)^a \; \forall \; i \in (0,m]$, $sign_{PIS\_pri\_key}(A\_ID \| X_i) \; \forall \; i \in (0,m]$, and $sign_{PIS\_pri\_key}(A\_ID \| (X_i)^a) \; \forall \; i \in (0,m]$ back to Alice. In the matchmaking phase, only steps 1 and 2 are different from the previous DDH approach. Here, Alice and Bob exchange the exponentiation results and signatures they got from the *PIS*. Note that in this protocol, since users need to exchange the signatures created by the *PIS* at first, there is no need for them to exchange their certificate in the signature-based authentication anymore. They should also compute a signature on the whole message that they want to exchange and also exchange the signatures.

Note that our security analysis is still valid except the following point: In the previous analysis, users could not use information not signed by the *PIS* for them if they do not want to be banned. Here, users could not use information not signed by the *PIS* for them at all. For example, Alice has to provide $sign_{PIS\_pri\_key}(A\_ID \| X_i^a) \; \forall \; i \in (0,m]$ for all elements in her set to prove that she really is assigned to this information before the matching. Unlike the symmetric encryption based commutative functions, nobody could find $X'^{a'} = X^a$ because of the discrete logarithm and the way how the X and X' values are created. This guarantees that Alice can only use $sign_{PIS\_pri\_key}(A\_ID \| X^a)$ to prove her ownership of interest X but nothing else. Since this cheaper DDH protocol is less costly, we implement it instead of the DDH protocol we propose in Figure 3.9.

3.4 A Variant of the Fair Exchange Protocol

Our fair exchange protocol can be applied to other applications that require a light-weight private-intersection protocol. For example, a user inputs a list of items that she wants to purchase to her

1. Alice $\xrightarrow{\forall\, i\in(0,m],\, X_i^a \| sign_{PIS\_pri\_key}(A\_ID \| X_i^a) \| \text{Signature\_on\_the\_whole\_message}}$ Bob
2. Alice $\xleftarrow{\forall\, i\in(0,n],\, Y_i^b \| sign_{PIS\_pri\_key}(B\_ID \| Y_i^b) \| \text{Signature\_on\_the\_whole\_message}}$ Bob
3. Alice and Bob verify signatures.
4. Alice computes $\left(Y_i^b\right)^a,\ \forall\, i \in (0,n]$
   Bob computes $\left(X_i^a\right)^b,\ \forall\, i \in (0,m]$
5. Let $cm_A = l((Y_1^b)^a) \| (Y_1^b)^a \| ... \| l((Y_n^b)^a) \| (Y_n^b)^a \| R$
   Alice $\xrightarrow{h(cm_A) \| sign_{A\_pri\_key}(B\_ID \| h(cm_A))}$ Bob
6. Let $cm_B = l((X_1^a)^b) \| (X_1^a)^b \| ... \| l((X_m^a)^b) \| (X_m^a)^b$
   Alice $\xleftarrow{((X_i)^a, (X_i^a)^b), \forall i\in(0,m] \| sign_{B\_pri\_key}(A\_ID \| cm_B \| l(X_i^a) \| X_i^a\ \forall i\in(0,m])}$ Bob
7. Alice $\xrightarrow{(Y_i^b, (Y_i^b)^a), \forall\, i\in(0,n] \| R \| sign_{A\_pri\_key}(B\_ID \| cm_A \| l(Y_i^b) \| Y_i^b\ \forall i\in(0,n])}$ Bob
8. Bob gets $Y_i, \forall Y_i \in \left(Y_i^b\right)^a \bigcap \left(X_i^a\right)^b$
   Alice gets $X_i, \forall X_i \in \left(Y_i^b\right)^a \bigcap \left(X_i^a\right)^b$
9. Alice $\xrightarrow{g^a}$ Bob
10. Alice $\xleftarrow{g^b \| sign_{B\_pri\_key}(g^a \| g^b \| A\_ID)}$ Bob
11. Alice $\xrightarrow{sign_{A\_pri\_key}(g^b \| g^a \| B\_ID)}$ Bob
12. Alice computes $k = (g^b)^a$ & Bob computes $k = (g^a)^b$
13. Assume that there are m' interests in the intersection
    Alice $\xrightarrow{E_k(X_i \| sign_{PIS\_pri\_key}(A\_ID \| X_i)), \forall\, i\in(0,m']}$ Bob
14. Alice $\xleftarrow{E_k(Y_i \| sign_{PIS\_pri\_key}(B\_ID \| Y_i)), \forall\, i\in(0,m']}$ Bob

Figure 3.10: Cheaper DDH Protocol

mobile device. In this thesis, we are going to use the term "item" to represent a kind of products that a store sells. When she walks by a store, her mobile device exchanges information with the store following our fair exchange protocol to find out whether the store has any of the items that the user wants. If the store does, the device notifies the user. However, if we take a closer look at this particular example, we find that the proposed fair exchange protocol will not work. Usually, a store owns hundreds or even thousands items. According to our fair exchange protocol, the user's device needs to verify all items' signatures and also download a large number of random numbers from the store in each step. The computation and communication overheads for mobile devices are huge.

The shopping notification application is obviously different from the application that we introduced in previous sections for looking for friends who have common interests. First, we do not need to protect the items available in stores from being learnt by anyone. Second, as mentioned in the last paragraph, a store usually owns a large number of items, and this is also a reason why we do not want to deploy the DDH protocol for this application. We modify our fair exchange protocol and make it useful for the shopping notification application. The variant of our fair exchange protocol consists of an initial phase and a matchmaking phase.

3.4.1 Initial Phase

The initial phase of this protocol is different for a user who wants to do shopping and for a store, which sells items. The initial phase for a store is similar to the fair exchange protocol introduced in section 3.2. The *PIS* builds up a tree. Each leaf node represents an item that a store might sell. Stores send their items to the *PIS*. The *PIS* first checks whether an item already exists in the tree. If the item does not exist in the tree (this store is the first one that submits this item), the *PIS* assigns an id to the item and puts the item into an available leaf node. If the item exists, the *PIS* locates the item and fetches the path of the item in the tree and its id. The *PIS* computes a signature for each item and sends the ids, paths, and the signatures back to the store. The way how the signatures are created is different from the one introduced in section 3.2. The signatures for store X are of the following form: $s(X) = sign_{PIS\_pri\_key}(X\_ID \| item\_id)$. These signatures do not include a random number as a commitment. It is reasonable to remove the commitment, since we can show that even if the store claims that it owns items that it does not actually have, the user's privacy would still be protected by a large cloak (a large number of other items she does not want to buy). Also, the cheating behaviour of the store could be easily discovered by the users (they just have to go into the store and they will find out), and this results in a poor reputation of this store.

The *PIS* also posts a list of existing items on a publicly accessible website with both the path and id information. Users who want to go shopping can look up existing items online to input the items they want to buy to their devices or download a list of existing items including the path and

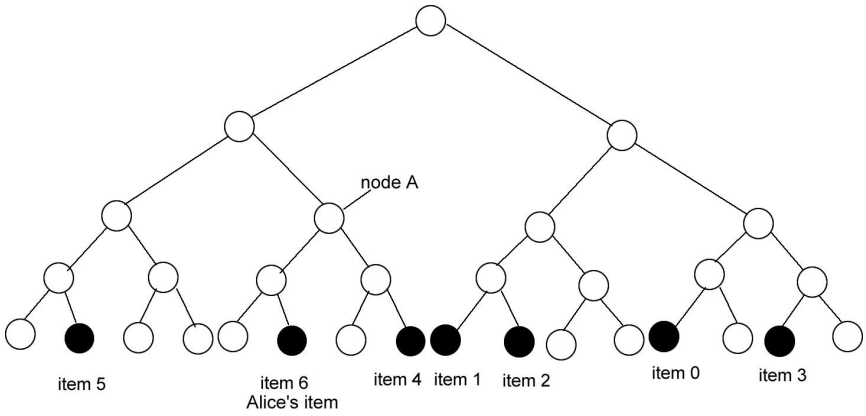

Figure 3.11: A Small Example of the Distribution of a Store's Items

id information of all items from the *PIS* directly. In the latter case, the users perform a matching between their own shopping lists and the list that they downloaded from the *PIS* to extract the information that they want. We also modified the matchmaking phase of our previous protocol, so that even though the tree structure is publicly known, users' privacy is still well protected.

3.4.2 Matchmaking Phase

Every user sets a k value, so that from other people's (including the store's) point of view, there are at least k possible items that the user may want to purchase. (The *PIS* provides sample items to fill all the leaf nodes, but the leaf nodes that contain the sample items are still available for taking new real items from stores.) Assume that we have an L level tree. When a user is nearby the store, the user device sends the top $L-1-\lceil log_2(k) \rceil$ levels of subpaths of all its items to the store. After the store receives the user's information, it sends all ids and corresponding signatures of its items that have the same top $L-\lceil log_2(k) \rceil$ levels of subpaths to the user. The user's device does a matching on the ids that she received and the ids of the items that she plans to purchase and verifies the signatures of the items in the intersection. The device notifies the user if the intersection is not empty and all signatures are valid. If the store cannot provide valid signature for its items, the user device should record it, and the user should report the store's malicious behaviour to the *PIS*. It is important that the signatures are obtained before the matchmaking. If users request the signatures after the matchmaking, they would require only the signatures of the items that the store has (the ones in the intersection of the user's list and the store's list). If a user's device downloads a list of items from the store, but never asks for any signature, the store would know that the user wants to purchase something else. Since the tree structure is publicly known (users are able to download the whole tree to their local devices), the cloak size to protect the user's real shopping list might become less than k.

We use Figure 3.11 as a small example to illustrate how the matchmaking phase works. Figure 3.11 shows the distribution of a store's items. The black leaf nodes are items sold by the store. Assume that user Alice sets $k = 4$. When Alice is around the store, her device sends the top $L - 1 - \lceil log_2(k) \rceil = 4 - \lceil log_2(4) \rceil = 2$ levels of subpaths of her items to the store. So Alice's device sends "lr" (left and right) to the store. The store knows only that the user wants to purchase at least 1 item of the 4 items rooted at node A. The store sends the ids and signatures of item 6 and item 4 back to the user device. The user device finds out that the store has item 6 and then verifies the signature of item 6. The user device notifies the user if the signature is valid. Note that the user does not have to go into the store to purchase item 6, even if she receives the notification. For example, she might want to go shopping at a different time or in a different store. As a result, if a user requires the items rooted at node A, and she does not go into the store, the store cannot conclude that the user wants something other than item 6 and item 4. From the store's point of view, the user has the same probability to buy any of the 4 items.

The above approach requires the user's device to download a lot of digital signatures. If we take a deeper look at this problem, we find that the signatures can be removed in this protocol. The whole point of having the signatures in our original fair exchange protocol is for a user to provide a commitment to other users to show that this user has registered a piece of information with the *PIS*. In that case, we want to preserve both users' privacy and do not want to reveal too much information at each step. Here, since we do not care about the store's privacy at all, and we allow the user's device to query any number of items from the store, we do not need this commitment anymore. One may argue that by providing the signatures, the store can prove to the users that they actually registered the items with the *PIS*. However, as we briefly mentioned before, the store does not have any motivation to cheat in this case, because it could be found out by the users easily, and the consequence is huge. Possibly, no one wants to go shopping in those stores that misbehaved. In this simpler approach, we requires the store to create a digital signature by signing its digital id (issued by some trusted identifier issuer) and the item ids that a user requests. The digital signature will be sent to the user along with the item ids. In this way, the user is able to verify that the message is from a registered store and able to report to *PIS* if cheating behaviour is detected.

The choice of the value of k is a tradeoff between performance and privacy. Obviously, the larger the k value is, the more privacy the user has. However, it also means that (most likely) the more id and signature information needs to be sent to the user by the store, and the longer it takes for the data transfer. As a result, if a user sets a small k value, the cloak size to protect the item she wants to purchase is small. If a user sets a large k value, there could be not enough time to complete the matchmaking phase while the user is passing by the store.

We can improve the previous scheme to address this problem. We require a user to set a k value, and a t value, which is the maximum time that it can take for the matchmaking to complete. A user no longer sends the top $L - \lceil log_2(k) \rceil$ levels of subpaths of her items to the

store. Instead, she queries the number of the store's items. According to the current data rate, the amount of data that needs to be transferred, and the t value, the user's device can decide whether it should request the store to send ids and signatures of all items or whether it should send the top level of the subpaths of the user's item to the store, and request the store to return the number of items that share the same subpath as the user's item. At each level $d < L - 1 - \lceil log_2(k) \rceil$, the user's device is going to make a decision whether it should send the next level subpath of its item to the store and query the number of the store's items that share the same subpaths (including the one it just sent), or whether it should require the store to send all items that share the same subpaths down to the current level. The protocol should be aborted if the current level $d = L - 1 - \lceil log_2(k) \rceil$, and there is still not enough time for the store to send all information to the user. All data communication between a user's device and the store should be protected by encryption. Of course, this improvement could be done only when we find a good indicator of the current data transferring rate. We did not find a good one in our implementation, so we implemented a slightly different mechanism. In addition, we did not find a way to establish HTTP persistent connections with BlackBerry devices, so that a device has to establish a new HTTP connection for each HTTP request. This adds extra cost when a device queries a web server multiple times. According to [11], we cannot reuse the same HTTPConnection Object in JavaMe.

Since we did not find a good indicator for data transferring rates and there is this HTTP connection problem (one HTTP connection for one HTTP request), we propose an alternative way to make our application more flexible. We provide the users three options, high/medium/low privacy, for this application. We should make it clear to the users that high privacy requires the most data downloading and it takes the longest to execute, and so on. The high privacy option means that the device will download every item's id from the store. The medium privacy option starts from one quarter down the tree, and downloads all items that share the same subpaths with the items the user wants down to the first quarter of the tree. The low privacy option starts from the middle of the tree, and downloads all items that share the same subpaths with the items the user wants down to the middle of the tree.

Suppose that we have a 20 level tree, the user wants 10 items, which are in a store, and the store has the same number of items in each subtree before the middle of the tree. By running the medium privacy option, the device needs to download less than 1/3 of the store's item ids, and by running the low privacy option, the device needs to download less than 1/100 of the store's item ids. In addition, the less id information the user's device downloads, the less time it takes to finish the matchmaking (the time for the matchmaking is linear in the number of ids a device downloads). We have not finished implementing this approach yet, and this will be handled in the future. We implemented the "high" privacy mode and present it in the section 4.2.

Private Information Retrieval (PIR) [5] is an alternative approach for the shopping notification application. Olumofin et al. [27] introduce a PIR approach that allows a user to query multiple

databases owned by different companies. The user is able to hide the information that she is really interested from each company and is still able to get the relevant information back. This approach requires several companies to collaborate to host multiple databases and at the same time, they should not collude to subvert a user's privacy. This makes it hard to deploy. There is ongoing work to improve the efficiency of single-server PIR [4].

Chapter 4

Implementation and Evaluation

In this chapter, we first present how we implemented our architecture, including the implementation of the *PIS*es (one for each protocol) and a client program for different protocols. Then, we evaluate the performance of our protocols and provide a brief analysis of the results. During our implementation, we found that a Bluetooth feature of some mobile devices reduces the flexibility and scalability of our system. We propose several solutions for this problem and implement one of the solutions. At last, we evaluate the performance of the implemented solution.

4.1 Fair Exchange and DDH Protocols

4.1.1 Development Environment

We implemented our protocols on two BlackBerry 8820 devices. The two BlackBerry devices act as the mobile devices owned by two users. We deploy a web, an application and a database server to build the *PIS*es. We did not implement the id signer, and we assume that the users already hold valid identity certificates. We use the Java platform, Standard Edition 6 development kit to develop our PIS programs and the BlackBerry Java Development Environment 4.5.0 for users' devices. A *PIS* includes the PIS program, interface and backend databases. More information about the PIS programs is in section 4.1.2. A Facebook application serves as the web interface to our *PIS*es. The Facebook application is implemented using PHP and HTML. More details about the *PIS*es are also presented in section 4.1.2. The data communication is done through Bluetooth. The Bluetooth communication is done using the RFCOMM StreamConnection. RFCOMM provides emulation of RS 232 serial ports. The most straightforward fashion for two Bluetooth devices to communicate is the server-client model. We will introduce how a device chooses its role in this model in section 4.1.3. We adopt this model in our application.

Our first goal was to understand the Bluetooth data exchange time and transfer rate. We conducted a number of tests on the data exchange time. The data exchange time of the client is from the time just before it sends the message (before the write() function is called) to the time it completely receives the message from the server (after the read() function returns). The data exchange time of the client consists of the following periods:

1. time for the client to send its message;
2. time for the server to receive the data;
3. time for the server to send its message (the server sends its message to the client as soon as it received all data from the client);
4. time for the client to receive the data.

The data transfer time, the time to transfer a certain amount of data from one device to the other, is the total time of period 1 and 2, or period 3 and 4. If we assume that the total time of period 1 and 2 equals the total time of period 3 and 4, the data transfer time is half of the data exchange time of the client. Figure 4.1 shows the data transfer time for different amounts of data. Each point in our graph is half of the average value of 10 measurements of the data exchange time of the client. In the figure, we use an error bar to show the standard deviation of the 10 measurements. For the same amount of information, the data exchange time is different when it is sent at once or at multiple times (send one byte in each write() call). We can see that the data transfer time is slow and could be the bottleneck of the overall performance. The average data transfer rate is about 30 Kbps when we are using one write() call. However, this value is much smaller than the bandwidth of Bluetooth 2.0, which is 2.1 Mbps. There are three possible reasons why the data rate is low in our devices. First, we use the RFCOMM StreamConnection, which has extra costs to guarantee ordering, perform error checking, and so on. Second, it is costly to read data out of the receiving buffer. We tried the readFully() functions to fetch the data from the buffer, and they are much faster than the read() functions that we are currently using. However, the readFully() functions are problematic when two devices are not fully synchronized. Fully synchronized means that when one device is sending data, the other party has to fetch the data at the same time. Third, according to [10] and [9], the data rates of BlackBerry devices are slow and differ significantly depending on their OS versions. We would like to investigate more on the data transfer issues in the future. Since now the data transfer is slow, it means that we are able to improve the performance of our protocol by trading some data transfer operations (IO operations) for some simple computational operations. We process the data that users try to send to each other to shrink the size of the data, and as a result, we get higher performance. This motivates us to implement a faster version of our fair exchange protocol that sends less data. More details about the fast fair exchange protocol can be found later in this chapter. In addition,

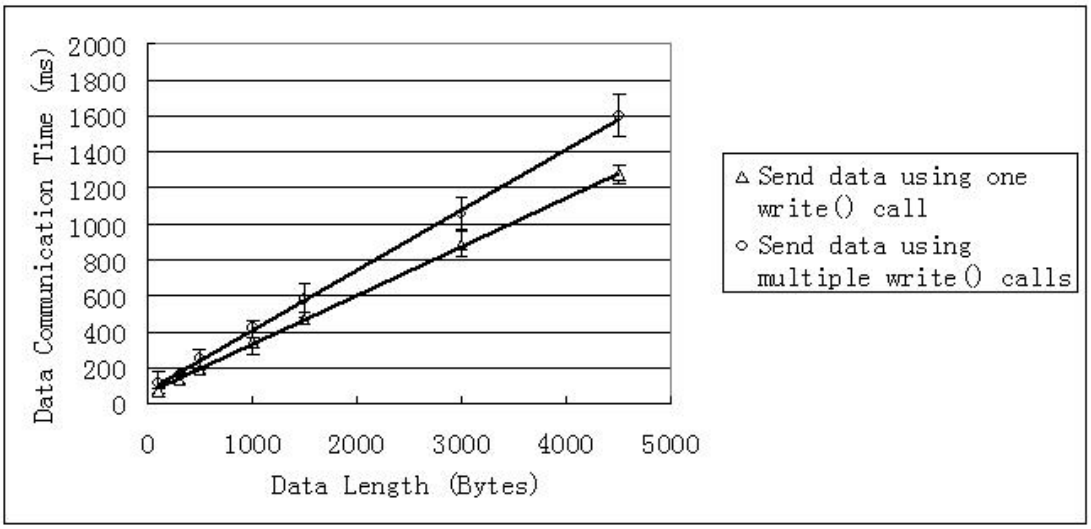

Figure 4.1: Time to Transfer Data

in our implementation, we also add a zlib on top of the data communication channel to compress the content that needs to be exchanged.

We set up two different *PIS*es. One is for the fair exchange protocol, and the other one is for the DDH protocol. We assume that our users want to re-use their interests from Facebook for our application. We implemented a Facebook application as the web interface for users to submit their interests to the *PIS*es to sign. We set up a MySQL database for storing users' id and interest information. The database consists of 6 tables. Three of them are for the fair exchange protocol and the other three are for the DDH protocol. For each protocol, there is a table containing the public key and private key information for the *PIS*es, and for the DDH protocol, there is another field in this table to store the big safe prime information. Let us call this table the PIS table. Again, for each protocol, there are two other tables. One table is used to store users' mobile device ids (a mobile device id is just the user id we mentioned in previous sections. Here we call it the mobile device id to distinguish it from the Facebook user id), Facebook user ids, and their interest information. There is another field in this table named processed. This field indicates whether a row in the table is processed by the *PIS*es. Let us call this table the userInfo table. The other table simply contains the mobile device id for the users waiting for processing. We call this table the waiting list table. We are going to illustrate the usage of these tables in the next paragraph.

Figure 4.2 shows the interface of our Facebook application. The Facebook user id and interests are exported from the Facebook database. We configure our application to use IFrame as the render mode for our application's canvas page ("A canvas page is the address where your

Figure 4.2: Interface of Facebook Application

application lives on Facebook." [14]). The mobile device id field contains a user's mobile device id. More specifically, this field should contain the hex value of the hash value of the public key of a user's mobile device. Everytime a user accesses this page, the page checks if the mobile device id associated with this user's Facebook user id exists in the userInfo tables. If it does, this field is automatically filled with the mobile device id. Otherwise, the user has to enter the id herself. A user can choose to request the protocol she wants to run with other users. If she wants to run both protocols, she has to submit this form twice, once for each protocol. The text area contains this user's interests. The information is from the interest field of the user's Facebook profile. Figure 4.3 shows the interaction among a user, Facebook, and our appplication server. More details about how a traditional IFrame canvas works can be found in [13]. Figure 4.4 shows the flowchart of how a user interacts with the *PIS*es, including the backend database and programs, through our Facebook application. We will describe how the PIS programs work in section 4.1.2. We treat the interaction between Facebook and our appplication interface as a blackbox, and call it *PIS* interface. Figure 4.5 illustrates what happens in each step.

4.1.2 Setup of the PIS Programs

The PIS programs are the core of our *PIS*es. Our PIS programs are two Java Jar files that are run on the same machine as our web and database server. One PIS program is for the fair exchange protocol. It takes an integer, n, as an argument and constructs an n-level tree (2^n interests in total) as described in section 3.2, when it starts. The number of levels of the tree can be increased if

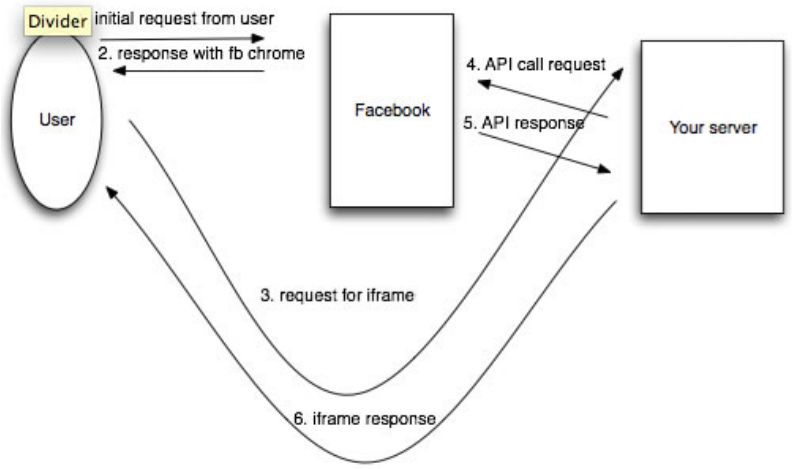

Figure 4.3: Interaction Among a User, Facebook, and Our Application Server (Picture from Facebook)

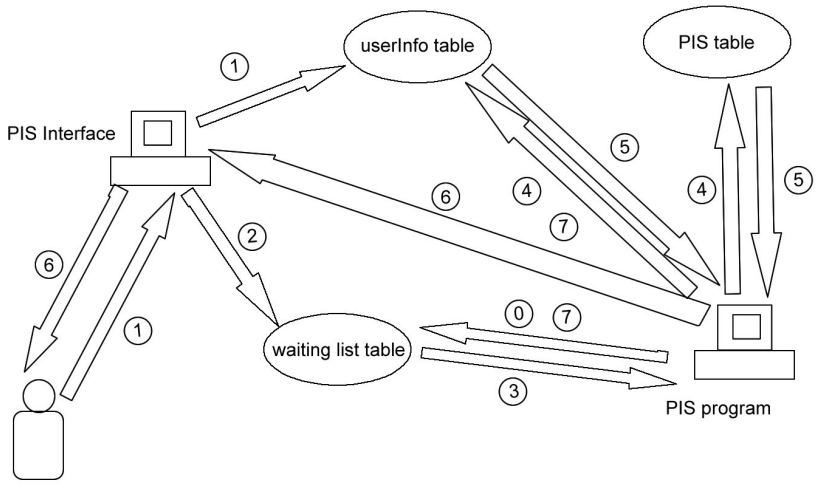

Figure 4.4: Users Interact with PIS Server Through Our Facebook Application

0. A PIS program keeps querying the waitinglist table for user ids. (This step takes place before any user can have her interests signed.)

1. A user submits the form through the *PIS* interface as shown in Figure 4.2. The user's interests along with her Facebook id and mobile device id are inserted into the userInfo table, and the processed fields for the new records are labelled to be 0.

2. After the new records are inserted into the userInfo table, the mobile device id is inserted into the waitinglist table.

3. Remember that before any user can submit her information to sign, we have a PIS program keep querying the waitinglist table for user ids. After step 2, the waitinglist table finally has something, the mobile device id(s) just inserted in, to send back to the PIS program. (Here, we require the PIS program to pull information from the database to sign, because the *PIS* interface is implemented in PHP and this is the best way that we found to pass information from a PHP program to a Java program.)

4. The PIS program queries the unprocessed records associated with the mobile device id(s) returned by the waitinglist table. At the same time, the PIS program queries the PIS table for the server private key information, and the big prime number if the user wants to sign her information for the DDH protocol.

5. The userInfo table and PIS table return the information that the user queries.

6. The PIS program creates an XML file using the information it gets from the userInfo table and makes the file available to the user for downloading.

7. The PIS program updates the processed field of all records that it just processed to be 1. The PIS program deletes the mobile device id(s) it just processed from the waitinglist table.

Figure 4.5: Steps in Figure 4.4

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Interest>
<item0>
<name>Programming</name>
<Path>leftrigtleftrigtrigtleftleftrigtleftrigtleftleftleftrigt</Path>
<Signature>
87329fa260f931d1dcc89c70c4706f8ba9a9de47ce6979f88b2ee332c1420e8b5189b9bebbdee38ec3e242c7e6de3248db61b31ae3f5234ca38b424c462e77a43f
42ac5aaf7fa6454d976938e48759ae302b2b9bc33ae6e9da242a8abbdc727fa8b23f215349b9cf05a218d02b3e799aada31b5438fb46a7788458b80721e49b
</Signature>
<RandomNumber>
db7b0f3e97abbd1685f30e9df0c514a1913a8c89c2fb8fd704d85c2e24c4933f4db85f0837db642e0c88273a8251f12da2b7d5c9fb9c80173f851d7c0aa481c618
1c121824924c11690825c309678e77e91ad5f0b2458c620dbe07efa9b965ad6892882450177f2909fcc1eade113ad2057bb8ab25333e2bd4ac7ef2e4a43c5e17f5
edfcfad877724eb0e3defa4230ac2ebde47e87c81375f7f6063a6bf810114cd3ede9bcda4f88aa66d7d648cc973d4bcbaccd6d1754bbc937e750790b399cdc10c
4151b205b438c56c15328c1656928273edbaac73057ff39dda4b25b2cf018fb5a28247fa39f260d696c1864ed0ee9c86cee0f2fa23a9407c86c76df2d66530bff9
489481f4ccd4a15dfa6a9da8587df85296b712c7</RandomNumber>
<Hashes>
4298f4b60c5a677e7bf0a46580429942c09554fd2240bc404b04313ffa8cc4bbe52180c63eeefd9756454949ee9c072b0672d350b96468e8b109c36915667d9009
287f66be2db956bfdb2d4b5bf67dd1e2447ce6edb20d310f09438c2ab7ba6801996f19e278f2fe364285137821d09741b02d0857c8fda872e5252802566350f9
23c076c69790c0a3affda5a3e771924804cb71f3d1b7064764c81dc44b4603ba154e3f6b1601fd926b8b1b8fcb07c84e12373711cabfa95f1f4b47c18585cdacbc
dfca7e2d589f1c81c1bb176afc6767c5a777a7ba6835709cc5f386bac8154540b21c7736218e1f60d37f3d9b2c69c11c58cdf6bad5c4553c310b3b20294b0174ae
bcf6ad09ecfa4792e9df888ec40d66671060eb67</Hashes>
</item0>
```

Figure 4.6: XML File

there are more interests in real life. More tests and user studies should be done before we can decide on a proper n value. The n value should also be related to the number of interests in the real world. The PIS program takes unprocessed information from the userInfo table as inputs and outputs XML files. An example of the XML file generated by the PIS program is shown in Figure 4.6. A path field in the XML file acts as the id of an interest. It is a string composed of "left" and/or "rigt" presenting left and right subpaths. We use "rigt" for a right subpath to keep the string the same length as "left", so that attackers cannot perform guessing attacks based on the length of the subpaths. We also could just use the character "l" and "r" to represent the subpaths. Note that an XML file cannot contain binary values, so we have to encode the binary values into hex or Base64 representation. The standard Java does not support Base64 even though it is much more efficient than the hex representation. We tried a number of open source Java Base64 applications and found none of them work compatibly with the BlackBerry JDE. As a result, we encode all our binary values into hex numbers.

In our original protocol, the PIS program randomly generates a 128-bit long number for each subpath, and composes the numbers into a string, and puts it in a random number field. We modified the original fair exchange protocol and made a fast fair exchange protocol. In the fast fair exchange protocol, we allow users to exchange the top $n/2$ levels of the subpaths of their interests at once. As we can see from Figure 4.1, the time for exchanging data is expensive, while computing hash values is cheap (we provide data in section 4.1.7). We generate the random numbers for the subpaths of different interests in a special way, so that when users run the fast fair

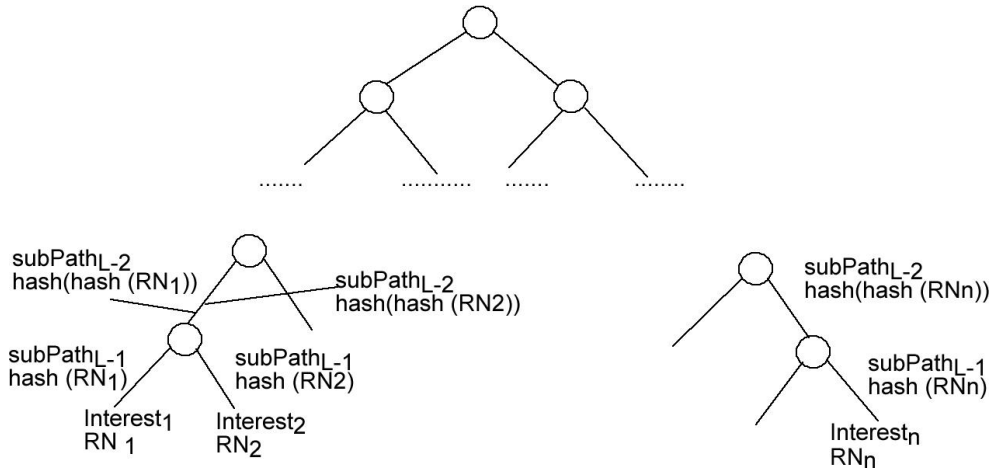

Figure 4.7: Random Numbers

exchange protocol, the amount of data they need to exchange can be sharply reduced, and thus the protocol execution time is shorter. More information about the fast fair exchange protocol can be found in section 4.1.5. Here is how we generate the random numbers. For each user, we randomly generate one 128-bit number for each interest. We compute the hash of this random number and use it as the random number for the bottom level subpath. Then, we compute the hash of the random number of the bottom level subpath for the second last level subpath, and so on. Figure 4.7 shows the way to generate the random numbers. Note that "interest 1" and "interest 2" share same subpaths, but there are different random numbers for the same subpath for different interests. We use the nextByte() function in Sun Java's SecureRandom class to generate the random number. The PIS program computes the hash value for the subpath and the corresponding random number of each level of an interest, and puts them into the hash value field separated by commas. We use Sun Java's Signature class and get the "SHA1withRSA" instance to compute signatures. We use SHA-1 as the cryptographic hash function, RSA private keys to sign signatures, and RSA public keys to verify signatures. The output XML files include the information described above and the *PIS*es' public key. All signatures, hash values and random numbers are converted into hex representations before they are stored in the XML file. We assume that the communication between the *PIS*es and a user is secure. Ultimately, the communication should be protected by SSL.

The other PIS program is for the DDH protocol. It takes inputs from the DDH tables and outputs XML files. An example XML file of the DDH protocol is shown in Figure 4.8. The PIS program generates a 1026-bit safe prime, p, and puts it in the "modulus" field in the XML file. We use the probablePrime() function in Sun Java's BigInteger class. This function does not guarantee to return a prime, but the chance that the returned number is not a prime is extremely

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Interest>
<item0>
<name>Programming</name>
<id>2e56b9407d0d58d6d3daff9c11d767628bd32b5</id>
<IdSignature>
44fe054af6ada4eca10386eb3d646efe9fb489b3fa9e535c45ebd23a4ef9e9d04c9d1f121b391e99a6f10f2eb0c900da88f5919d198ffc7462875265d0b9d7ba69
69ede69387d0a45ab2231f16126ee8c9670824770c1dacd0a7c30ab74c02959e23874434b81f970100d51f477ef616f88a543376cf3d9518e7045143095d74
</IdSignature>
<PowerId>
010641db1fce72496dcfa51db727ac02d0c4210428e0f63fc0e3fda3641b12af6d87145f069736dbc80124bfcbce2a8611a8593ddb7e1cd2cbb009143d6c036c73
06dd731eba3216c91fb2712eca17c8aee4783d9ac0f4ca9e13e306e6d099982e40d095d1054fe917ec5b85599c08af4309dc8e35f38d16404967f0b749cd46e7
</PowerId>
<PowerIdSignature>
5ed19436bf6fcbe69bbda68fad792dc878fcb746ceb05317c1fa42032c7b0e4a63a0df82b193c89ca3947d3ed5cf3f399071bf3178cf50a4d74356d286604ed1f3
4dc12007543cbaf30332a3add846f525d90f42eaba606c0782659fff502feb982bdfdf89258ff95cdb3f5706f5a8b02658b89b9815ee0cf545ae1e198277b3
</PowerIdSignature>
</item0>
```


Figure 4.8: DDH XML File

small. By default, there is only $1/2^{80}$ chance that the value is not a prime. The input from a user includes her mobile device id and her interests. The PIS program hashes the name of each interest and uses the square of the hash values as the interest id. The PIS program puts the interest id into the id field. The PIS program randomly generates a 1024-bit number as a user's exponent of the power function. It checks and makes sure that the exponent is smaller than $p/2$, and puts it in the exponent field. The PIS program computes $id^{exponent}$ and puts it into the "PowerId" field. The PIS program also computes signatures for the id and powerId and puts them in the "idSignature" and "PowerIdSignature" fields. Again, all values stored in the XML file are in their hex representations.

4.1.3 Setup of the Devices

The first time a user uses this BlackBerry application on her device, the application provides an interface for her to create a new mobile device id. She simply presses a button to create a new pair of RSA keys. The hex string of the hash value of the public key is computed and used as the mobile device id. The key pair and the mobile device id are stored in the persistent storage of the device. She needs to submit her interests through the Facebook application to sign. After she does so, she will receive an XML file for each protocol. She stores the XML files in the device memory, and then she is ready to run the matchmaking protocol. As mentioned, we adopt the server-client model for two Bluetooth devices to communicate. Users can choose "Run as a Server" or "Run as a Client" from the menu bar of our application to start a server or a client. We will see more about server and client devices in the next two sections. The application also provides an interface for users to select which protocol they are going to run, and for which

interests they want to look for potential friends.

Our application is now used only for testing purposes, so users need to manually select to run as a server or a client. In the future, we will improve our implementation, so that a device can automatically change its roles. A device should run as a server most of the time and wait for incoming connections, and it periodically becomes a client and searches for nearby servers.

Server Device

Selecting "Run as a Server" from the menu bar makes the device a server in a two party communication and starts a service on the device. Each service is identified by a unique 128-bit id, which is called UUID. We randomly pick a UUID value and use this value as the UUID value for our matchmaking services on all mobile devices. In other words, all servers use this value as the UUID of this service. After the service starts, it blocks the application and listens to incoming connections. When a client connects, the server is going to open an output stream to send messages and an input stream to receive messages.

Client Device

Selecting "Run as a Client" from the menu bar makes the device a client. A client first scans for nearby Bluetooth devices and returns them. If nothing returns, it means that either there is no Bluetooth device around or there is a Bluetooth device around, but the client device fails to detect it. For each returned device, the application is going to search services with the UUID of the matchmaking service. Only the service using this UUID will be returned. The returned service contains a connection URL for a client to connect to the server. The client then opens an output stream to this URL to send messages, and an input stream to receive messages.

4.1.4 The Implemented Fair Exchange Protocol

Our implementation allows the server and the client to send 256 sub-messages and 65,536 bytes in each sub-message at once. We need the sub-messages to convey the information for different interests. The first byte indicates the number of sub-messages included in this piece of message. The first two bytes of each sub-message indicate the actual length of the information in this sub-message. The first byte indicates the most significant bits and the second byte indicates the least significant bits.

Our application requests a client to send information first. Suppose that a client successfully connects to a server. They run the signature-based authentication protocol. In this protocol, they

generate 128-bit random numbers using RIM's PKCS1MGF1PseudoRandomSource class. They successfully exchange their RSA public keys and verify that the other party has the corresponding private key. The devices compute each other's user ids using the other party's public key. This guarantees that the other party is who they claim they are. Then, the devices exchange their signatures and hash values of the subpaths and their corresponding random numbers in the first two steps. After verifying all the signatures, the client and server are going to exchange their subpath information and random numbers. Each time, they exchange the information for a level. Upon receiving the information, each party computes the hash values and checks if the hash values match the corresponding hash values that they received in the first two steps. Each party uses a StringBuffer for tracking the subpaths received so far for each interest. If the hash values match, they append the received subpaths to the StringBuffers. Otherwise, they append the string "NULL", and report an error. Each party also keeps track of the paths it received so far and compares the received paths to the paths that it owns. If a party finds that the other party cannot have one of her interests (a part of one of her paths does not match any of the received subpaths so far), the party labels this path as "unmatched" and next time she sends the string "NULL" instead of the actual subpath and random number information. (Note that for fairness, the server performs the comparison after it sends out the information of the current level.) If a party finds that it does not have one of the other party's paths, it also label this path as "unmatched". If the devices receive a "NULL" string as the subpath or random number for a path, they should first check if this path is labelled as "unmatched". If it is not, the devices report an error. Otherwise, they do not compute the hash value for this subpath and random number, but add a "NULL" string to the corresponding StringBuffer directly. The devices stop when one of them finds out that it has to send a "NULL" string for all subpath information or when the bottom of the tree is reached. The communication of the second half of the path information is encrypted using AES with the key derived from the secret generated from the authenticated Diffie-Hellman protocol, in case that other nearby devices may eavesdrop the information. In the authenticated DH protocol, we use RIM's DHCryptoSystem class to generate a DH public/private key pair. In order to exchange g^a and g^b as described, we exchange the DH public keys. Then, the devices share a secret using the generateSharedSecret() function of RIM's DHKeyAgreement class. This function generates "the shared secret using a given public key (from another party) and a private key with the option of using cofactor exponentiation" [31]. For simplicity, in this thesis, let us say that this function computes a value $k = (g^a)^b = (g^b)^a$.

4.1.5 The Implemented Fast Fair Exchange Protocol

The fast fair exchange protocol is the same as the original fair exchange protocol except it allows both parties to exchange the first half of the subpaths and random numbers at once. Because of the special way that the *PIS* generates the random numbers for interests (or paths), we only

require the users to exchange the random numbers for the last level subpaths of the first half of the tree. The rest of the random numbers can be computed by hashing the received random numbers.

Here we sacrifice some privacy for performance. We allow users to exchange the first half of their subpaths, since even if two users do not have any information in common down to the middle of the tree, there is still a large enough cloak to protect both users' real information. (When the tree structure is not compromised, this protocol is as secure as the normal fair exchange protocol.) For example, there are two users, each of them has 1 interest, and they try to find out if they have an interest in common. Assume that the tree structure is known to the users. For a 21-level tree, two users exchange the top 10 subpaths. Even in the case that they do not have anything in common, from each user's point of view, the other party has equal chance to have any of the $2^{11} = 2048$ interests. Note that users do not need to run the fast protocol for a small tree structure, if they think the cloak size is not large enough to protect their real information, and it does not take too long to execute the fair exchange protocol for a small tree structure anyway. More information about the executation time of different protocols is in section 4.1.7.

4.1.6 The Implemented DDH Protocol

The first step of the DDH protocol, the signature-based authentication is the same as in the previous protocols. Then, the devices exchange information in the "PowerId"(PI) field in their XML file. They retrieve the information in the "exponent"(e) and "modulus"(m) fields, and cast them into two big integer numbers using RIM's CryptoInteger class. They also cast the PI information to big integer numbers. Then, they compute $PI^e \mod m$. Instead of exchanging the pair $(PI, PI^e \mod m)$ in the steps 6 and 7 in our proposed DDH protocols, we only exchange $h(PI^e \mod m)$. This sharply cuts the communication overhead but adds little computational overhead. Since as we mentioned in section 3.3 that it is impossible for a user who does not know the value of e to map $PI^e \mod m$ back to PI, it is the same whether users send the pair $(PI, PI^e \mod m)$ or $h(PI^e \mod m)$. The other steps are straightforward and the same as shown in Figure 3.10.

4.1.7 Evaluation

In general, the fast fair exchange protocol is the fastest protocol, and the fair exchange protocol is faster than the DDH protocol. The execution times of the three protocols are shown in Figure 4.9. In the tests, the PIS program builds an 11-level tree for the fast and original fair exchange protocols. The computation of modular exponents is expensive. The time to sign a message is about 120 ms; while the time to verify a message is about 20 ms, because the RSA public

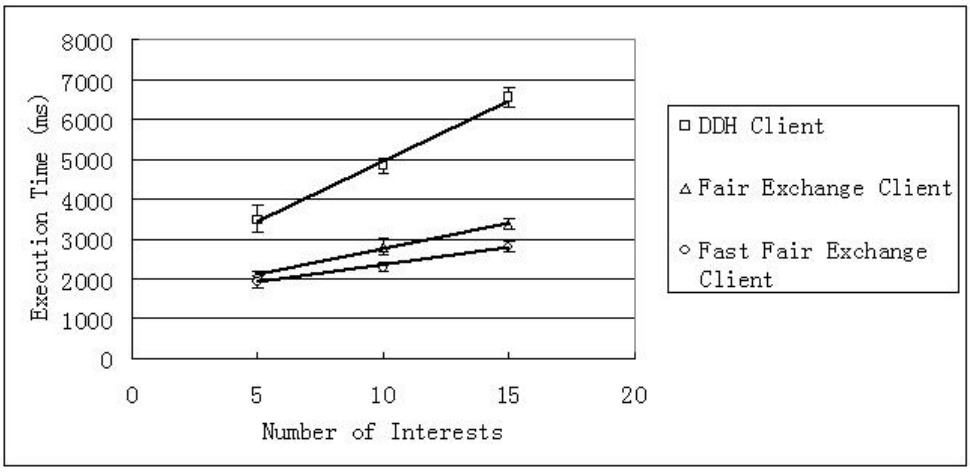

Figure 4.9: Execution Time

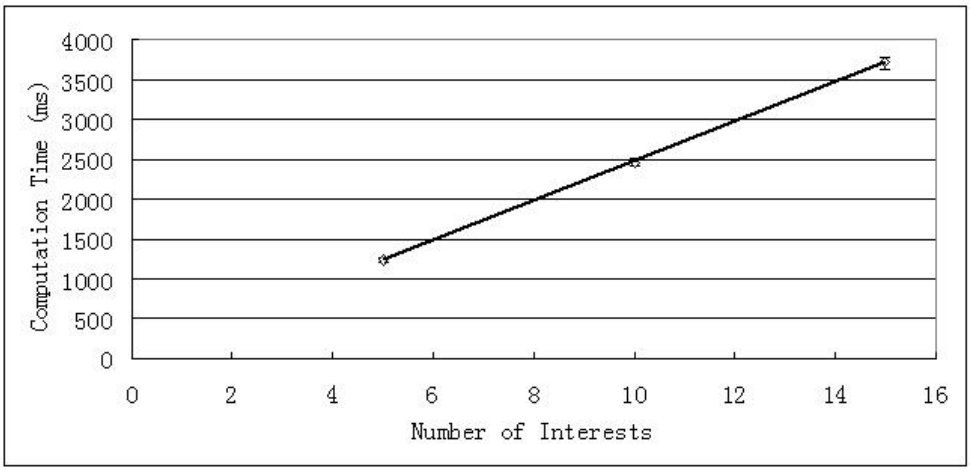

Figure 4.10: Modular Exponentiation Time

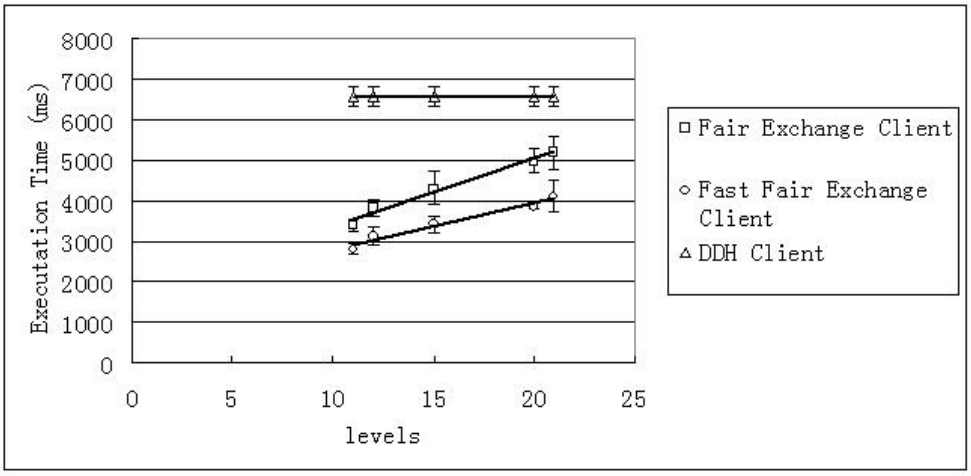

Figure 4.11: Execution Time vs. The number of levels of the tree

keys are much shorter than the private keys. Figure 4.10 shows the time to perform modular exponentiations on the *PI*s (the "PowerId" values). It takes about 250 ms to perform a modular exponentiation on the value of one *PI*. We used the functions from the BlackBerry JDE for signing and modular exponentiation. We do not have the source code of these functions and do not know why the modular exponentiation function in the CryptoInteger class takes longer than the signing function in the PKCS1SignatureSigner class. We guess that RIM optimized the signing function. For each interest, a device needs to perform exactly one modular exponentiation in our DDH protocol. We also find that the data communication overhead is more expensive than the computational overhead of computing the hash values, and that is why we generate the random values in the special way. For the 15 interests and 11-level case, each user would transfer 960 less bytes but compute 60 more hash functions. The time for computing 60 hashes is about 30 ms.

The execution time of the fair exchange protocol and fast fair exchange protocol increases as the number of levels of the tree increases. Figure 4.11 shows the trend of the increase (each user has 15 interests). We also provide the execution time of our DDH protocol, which is independent of the number of levels, in the figure for comparison purposes. Every data point in Figures 4.9, 4.10, and 4.11 is the average value of 10 experiments.

4.2 Implemented Shopping Notification Application

We also implemented the shopping notification application as a part of our BlackBerry application, and evaluated the performance. We use an access point from the Computer Science Building

at the University of Waterloo and an access point from the author's home. For testing purposes, we hardcoded the server URL to be a server located at the University of Waterloo. This server is used to simulate a server owned by a store. Note that an access point can be set up and configured to directly access a server according to the service type, and the URLs of the actual servers should not be hardcoded. The communication between the device and the access point is done by WiFi.

We also create the *PIS* on the same server. The *PIS* of this application is much simpler than the ones of the fair exchange and DDH protocols. For simplicity, we have not implemented the web interface for stores to sign their items or for users to look up their items. The *PIS* connects to a database, which has a table for storing the id and path information of the items the store has. The id and path information is created by a Java executable Jar file that works similar to the *PIS* for our (fast) fair exchange protocol. The only difference is that there is no signature created for the items. Again, for testing purposes, we assume that a user has 10 items in her shopping list and the store has all of these items. We randomly choose 10 items from the database as our sample shopping list and store them in the BlackBerry local disk. In order to make the matchmaking process faster, we require the shopping list and the ids that the store sends to the user to be sorted.

Figure 4.12 shows the test results for the execution time of our application when the store has different numbers of items. Every data point in the graph is the average value of 10 experiments. In the test, we download all item ids that the store owns to simulate the "high" privacy mode of this protocol. In the test, the *PIS* builds a 20 level tree for storing all items from different stores. An item's id is the SHA-1 hash value of its path, so each id is 20 bytes. If the store owns 500 items, the mobile device has to download 10,128 bytes of information from the store's server (128 is the length of the signature that the store creates). We can see that the execution time of this application using the school access point is faster than the one using the home access point. We believe that this is because that the server is located at the University of Waterloo and it is faster to access it through the internal network. We did not investigate this observation in more detail because it is out of the scope of this thesis.

As mentioned in section 3.4, we have not found a good indicator for the data downloading speed. We tried the getAPInfo() function from the WLANInfo class, and then used the getDataRate() function to fetch the current data rate in Mbps. However this always returns the theoretical upper bound of the access point. We also used the getSignalLevel() function to get the signal strength in dBm; however, this is not a good indicator, either. The value of the signal strength changes rapidly and does not seem to have a strong correlation with the data transfer speed.

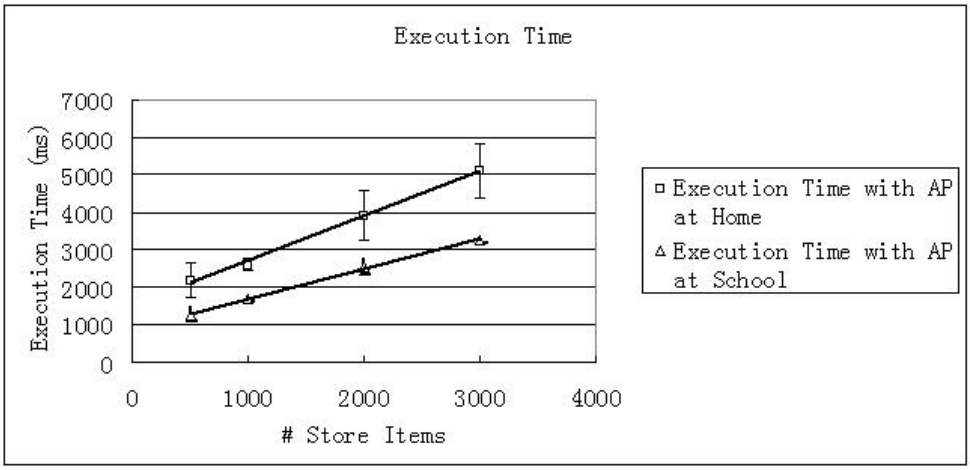

Figure 4.12: Execution Time vs. the Number of Items the Store Owns

4.3 Simulation of the Security of Our Fair Exchange Protocol

In this simulation, we want to test how effectively the fair exchange protocol can protect users' privacy after the tree structure is leaked. We vary the number of levels of the tree that the *PIS* constructs, and the number of interests that each user has. We randomly generate a number of interests for each user, and test whether they have at least one common interest. If they do not, then we test the number of levels of subpaths that they are going to exchange before they abort (the size of the cloak). We say that two users reach level n of the tree if the two users abort the protocol before they exchange the $(n+1)^{th}$ level of subpaths. Figures 4.13, 4.14, and 4.15 show our simulation results for 11-, 16-, and 21-level tree, respectively. For every graph, we ran the simulation 500 times when both users have 5, 10, or 15 interests (1500 simulations in total for every graph). For every simulation, we recorded the highest level of the tree that the two users can reach (with 0 being the root level). The x-axis represents the levels that the two users can reach at each simulation. The y-axis represents the number of times that the two users reach a level in the 500 simulations. Note that the y values of the last point of the x-axes ($x = 12$) represent the times that the two users reach the bottom of the tree and they have at least one common interest (e.g., 11-level tree cannot have 12 levels). From the graphs, we can see that when the number of the levels of the tree is fixed, the more interests each party has, the more chances they are going to reveal their unmatched information to each other. When the number of interests each party wants to exchange with the other is fixed, the more depth the tree has, the lower the chance they will reveal their unmatched information to each other. Namely, there will be a large cloak protecting her real interests. As a result, the more users use our application and the more interests are stored on our server, the more privacy users would have after the tree structure is leaked. On the other

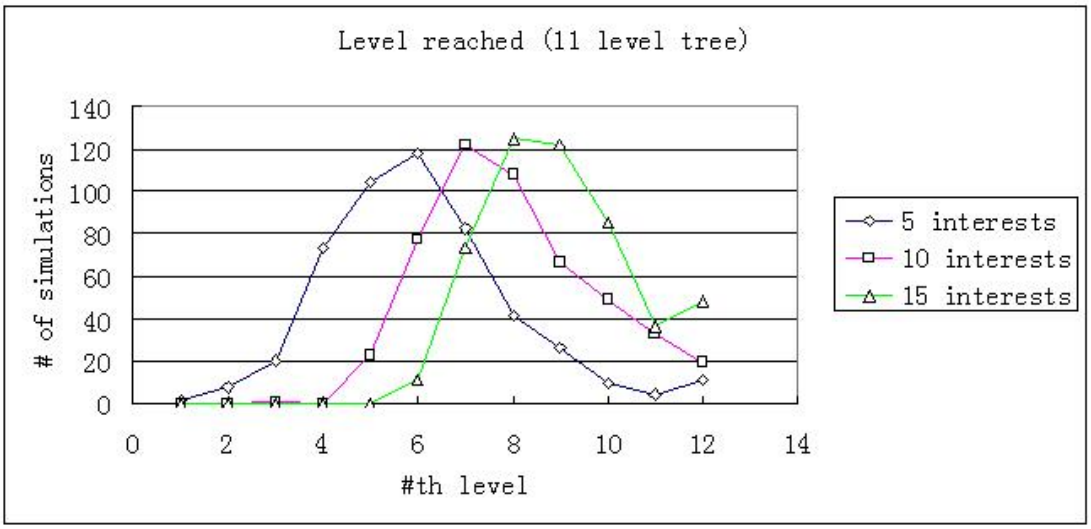

Figure 4.13: 11-Level Tree

hand, the more users we have, the more chances the tree structure is going to leak.

4.4 Avoiding Manual Pairing of Bluetooth Devices

In order for two BlackBerries to communicate through Bluetooth, we need to pair those devices by manually entering a pairing key. This is really a limitation when somebody wants to run our application without user interaction. This does not mean that our application is useless. Essentially, this is just a feature of BlackBerry Bluetooth. To the best of our knowledge, there are various kinds of mobile devices that do not have this requirement. Windows Mobile devices give their users the option to allow unpaired devices to connect [35]. A developer also mentioned in the BlackBerry forum that many other kinds of mobile devices do not have this limitation [36]. BlackBerries have this feature to prevent an arbitrary Bluetooth device from connecting, thus unpaired devices cannot perform attacks through Bluetooth. However, we believe that RIM can do a better job by having a finer access control granularity. For example, they can set different services to different access levels. If a service or application allows arbitrary users to connect, then the service or application should not access any critical data. All data are default to be critical unless specified by the owners. In this way, the unpaired attackers are unable to access any sensitive information through the service or application they connect to.

We now provide some solutions for devices that do not support automatic pairing.

The first and the simplest solution is to require two users to manually pair their devices and

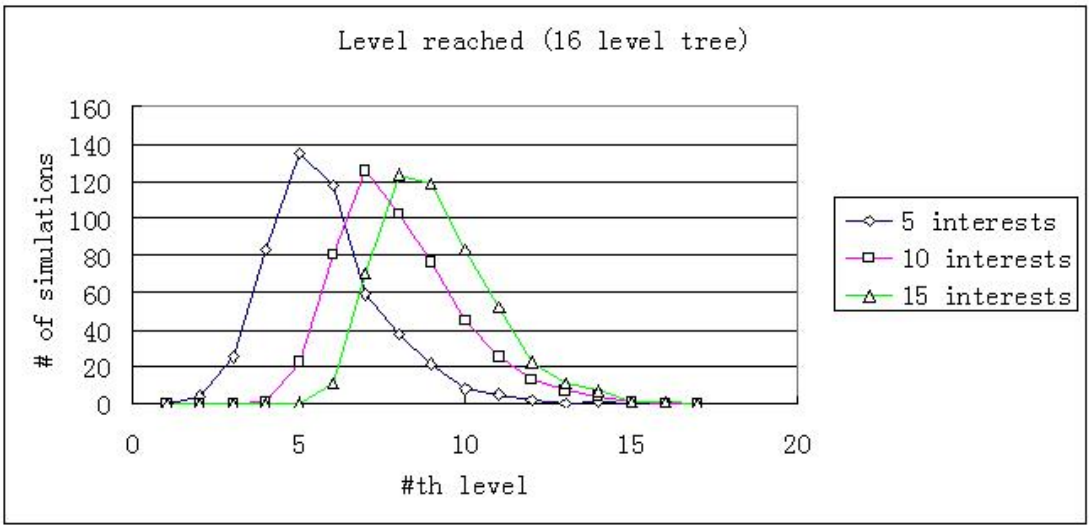

Figure 4.14: 16-Level Tree

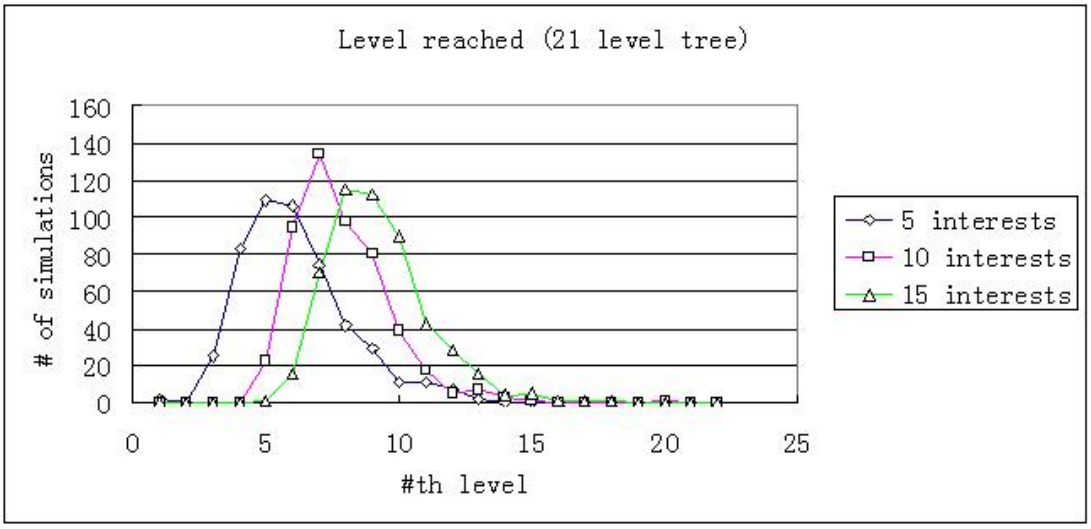

Figure 4.15: 21-Level Tree

then run our application to find out what they have in common. This is a little awkward, because a user has to find out whether another one is using our application. However, once two users pair their devices, they may find an interesting topic to talk about based on their matching result. In this solution, the users should make sure that they do not have any other critical Bluetooth service running during the matchmaking period. They should remove each other's device from their paired device list before they restart the other critical Bluetooth services if they do not trust each other.

The second solution is a little more complicated and also requires extra hardware, but it lets users find out who else is running our application automatically. In this solution, we need an Access Point (AP) and a back end server. Note that this back end server is not our *PIS*. The AP and server are provided by bars, long distance coach companies, or any other companies who want to attract more customers by supporting our application. We assume that the server is honest but curious. The companies may want to collect their customers' information, but they do not want to be caught for misbehaving. Otherwise, they may lose business. In this solution, the server can learn the users who run our application, but the server is not going to learn whether or not two users become friends or which interests that the users have in common. Figure 4.16 shows the solution. It requires that users set their devices' Bluetooth device identifiers to the the hex values of the hash values of their public keys as a part of their BlackBerry names. Assume when Alice queries the server, there are no other users looking for friends using our application.

However, the range of WiFi may be different from the range of Bluetooth (usually the range of WiFi is larger than the range of Bluetooth). In other words, Bob may not be able to find Alice because the distance between them is too far. Also, again, because the two devices need to be paired, the users should make sure that they do not have any other critical Bluetooth services running during the matchmaking period, and they should remove each other's device from their paired device list before they restart the other critical Bluetooth services if they do not trust each other.

The third solution takes the reverse way that the second one does. In the second solution, the AP and server simply find potential application participants for the users. The matchmaking processes are still done through Bluetooth, and thus there is a potential issue as mentioned. The third solution uses Bluetooth for finding nearby people, but not for running the matching protocol and therefore requires no pairing. Assume that Alice and Bob are two users using this solution, and they have got their information signed by the *PIS*. Alice has m interests and Bob has n. Figure 4.17 shows how this solution works. Note that in order to use this solution, we have to assume that the server is honest and the server does not want to learn users' information at all. Essentially, any passive listener could reveal Alice's s value to the server.

The fourth solution performs matchmaking through the server using WiFi. BlackBerry devices do not support "ad hoc" networks, so that devices cannot communicate each other directly [30]. In this solution, the server is going to know who and who have something in common,

1. Alice queries the server (through the AP) to check whether there is any user available before t_1, and finds out that currently there is no available users. Alice notifies the server that she will be back and check again at t_1. Alice also tells the server about her device id. The server puts Alice's information in a waiting queue.

2. Bob submits his device id and queries the server to check if any user is available before t_2. Assume that $t_2 \geq t_1$, the server checks its database, and randomly chooses one user who satisfies the criteria. Also, the server guarantees that the two users have not talked during the current day. We assume that this time Alice is chosen. Alice and Bob have not met during the current day. The server randomly generates a 4-digit pairing key and sends it to Bob along with the value of $t_1 + \varepsilon$ and Alice's device id. ε is a small integer whose unit is seconds. This value is to guarantee that Alice has enough time to fetch information from the server. Also, the server now removes Alice's information from the waiting queue, and adds Alice and Bob's information as a pair to its history.

3. Alice queries the server at t_1, and the server sends Alice the pairing key, Bob's device ID and the ε.

4. Alice and Bob are both notified by their devices at $t_1 + \varepsilon$. Bob is going to search for Alice using Bluetooth, and pair with Alice using the pairing key.

5. Alice will run our application as a server, and Bob will be the client.

Figure 4.16: The Second Solution

1. Assume that when Alice tries to search for nearby protocol participants, there is no one around her. Alice randomly generates a key, s. Alice computes hash values, $h_i = h(l(s)\|s\|X_i)\ \forall i \in (0,m]$(An alternative is to use HMAC), and signs each hash value using her own private key, $s_i = sign_{A\_pri\_key}(A\_ID\|h_i)\ \forall i \in (0,m]$. Alice sends the hash values, signatures, along with her public key to the server. Alice uses s as part of her BlackBerry' Bluetooth name. Alice also uses her device id as part of her Bluetooth name.

2. Bob is near Alice. Bob searches for nearby Bluetooth devices, and is able to obtain Alice's device name. As a result, Bob gets s and Alice's device id. Bob computes $h_i = h(l(s)\|s\|Y_i)\ \forall i \in (0,n]$, and signs each hash value using his own private key, $s_i = sign_{B\_pri\_key}(B\_ID\|h_i)\ \forall i \in (0,n]$. Bob uploads the information to the server and informs the server that he wants to match with Alice (by providing Alice's device id).

3. The server verifies both users signatures first. Then, it checks whether a hash value submitted by Alice matches a hash value uploaded by Bob. If so, the server sends the corresponding signatures from Bob to Alice, and also the corresponding signatures from Alice to Bob.

4. The two users should also exchange the signatures (the ones signed by the *PIS*) of their information that they have in common.

Figure 4.17: The Third Solution

1. The server groups a pair of users the same way as that of the second solution, except at this time, the server does not generate the pairing key.

2. Alice and Bob connect to the server at $t_1 + \varepsilon$.

3. Alice and Bob run the signature based authentication protocol through the server (the server simply forwards the messages from one user to another).

4. Alice and Bob run the fair exchange or DDH protocol through the server.

Figure 4.18: The Naïve Fourth Solution

1. The server groups a pair of users the same way as that of the second solution, except that this time, the server does not generate the pairing key.

2. Alice and Bob connect to the server at $t_1 + \varepsilon$.

3. Alice and Bob run the signature-based authentication protocol and the authenticated Diffie-Hellman protocol to negotiate a session key, k, through the server (the server simply forwards the messages from one user to another).

4. Alice computes hash values, $h_i = h(length(k)\|k\|X_i)\ \forall i \in (0,m]$, and signs each hash value using her own private key, $s_i = sign_{Alice\_pri\_key}(Alice\_ID\|Bob\_ID\|h_i)\ \forall i \in (0,m]$. Alice sends the hash values, signatures, along with her public key to the server. So does Bob. (An alternative approach is to compute a HMAC of the interest using k as the MAC key and then sign the MAC value.)

5. The server verifies all the signatures provided by Alice and Bob, and it will continue the next step if all signatures are valid.

6. The server finds out the intersection between the hash values. Then, it informs both Alice and Bob about the h_i, and s_i values that they have in common.

7. Alice and Bob then exchange their corresponding signatures signed by the *PIS* through the server. The signatures are encrypted by k.

Figure 4.19: The Fourth Solution

but the server is not going to find out what they have in common. Figure 4.18 shows the naïve fourth solution. In this solution, the server simply acts as a bridge between two users. Note that in our fair exchange and DDH protocols, users are going to run an authenticated DH protocol, and sensitive information is encrypted by the generated session key. As a result, the server will not learn any useful information. We could improve the fourth solution to hide users' identities from the server, and we are going to leave this for future work. However, this solution is rather tedious and expensive. A message from Alice to Bob now becomes two messages: a message from Alice to the server and another from the server to Bob. In fact, we can always ask the server to do more. For example, we can ask the server to perform the matchmaking. Figure 4.19 shows how the protocol works, but note that if we require the server to perform the matchmaking, we have to assume that the server does not collude with either Alice or Bob. Otherwise, for example, Bob could share the k value with the server or the server sends Alice's hash values to Bob. Then, Alice's privacy would be broken. We also assume that users connect to the server through a secure channel. If either party cheats by using information that is not signed by the *PIS*, they are going to be found out right away for not being able to provide valid signatures. Since they signed the information at step 5 using their own private keys, the signatures become evidence of their misbehaviour. As a result, it is important that the server validates all signatures, but we could not hide the device identities from the server in this approach. Also note that in this solution, we use a symmetric-key based approach, which is much simpler than an asymmetric-key based approach, such as DDH. If we require the users to run the DDH protocol but require the server to perform the matching, the assumption that the server does not collude with any user would still hold for the solution to be secure. Otherwise, for example, Alice computes $((Y_i)^b)^a \forall i \in (0,n]$ and Bob computes $((X_i)^b)^a \forall i \in (0,m]$. They submit the results to the server for matching. The server could simply inform Alice that there is no common information between Alice and Bob, while the server informs Bob about Alice's computation results. It is worthwhile to notice that this approach is a little more secure than the symmetric-key based approach, since in this case, Bob and the server learn only Alice's interests that match the ones that Bob has.

4.4.1 Implemented Solution and Experimental Results

We implemented the second version of our fourth solution, and evaluated the performance. We re-use the XML document issued by the *PIS* for the DDH protocol as the input of each client device. The matching server is located at the University of Waterloo.

The matching server hosts four "static" tables and some dynamically generated tables. By static, we mean that the table always exists in the database, no matter whether there are devices communicating with each other through the server or not. The first "static" table is the waitingQueue. This table is used to store the information of the devices that have not found a partner to run this protocol with. This table records the device id, the public key and the time when the

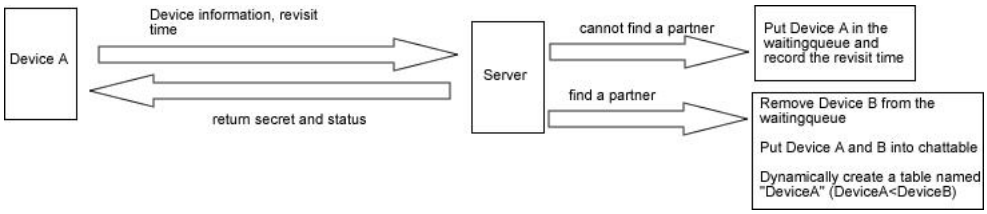

Figure 4.20: First Visit

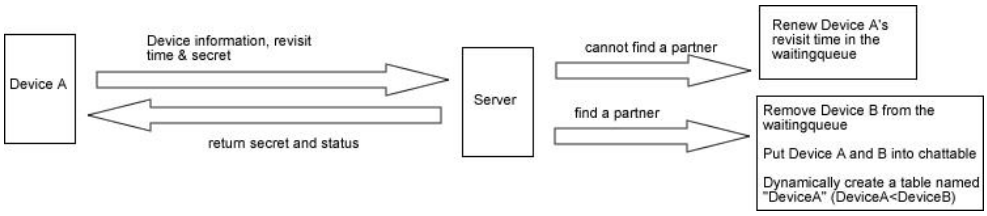

Figure 4.21: Further Visits (if necessary)

device is coming back to revisit the server for querying a partner. The second "static" table is the chattable. This table keeps all the pairs of devices that are currently exchanging information with each other through the server. For each pair of devices in the chattable, the server dynamically creates a table. This table contains the messages that each device sends to its partner and the server. This table's name should be unique. We planned to use the concatenation of the two devices' ids as the table's name; however, MySQL cannot take such a long string as a table name. We use the id of the device with the smaller id value as the table name, since we assume that a device is going to talk to at most one other device at one time. Alternatively, we could get rid of this assumption by using the hash value of the concatenation of the two devices' ids as the table's name. The third one is the history table. This table keeps the records of the pairs of devices that have executed this protocol in the past 24 hours. This table is used to make sure that no two same devices would be paired again by the server within a day. The fourth table is verificationQ. This table is to inform a Java program to verify the signatures signed by the client devices. Our server code is written in PHP, and the signature related functions are from OpenSSL. The OpenSSL signature verification function requires the public keys in PEM format. However, we cannot convert Java generated public keys to the PEM format. We wrote a Java program, which frequently (every 0.02s when it is idle) checks the verificationQ for the names of the tables that require signature verification. The Java program verifies the signatures and updates the table with the verification result. Our PHP server script is going to respond to the clients according to the verification results.

After having introduced the tables of the database, let us take a look at the flow of the execution. Figure 4.20 shows what may happen when a device visits the server for the first time. A

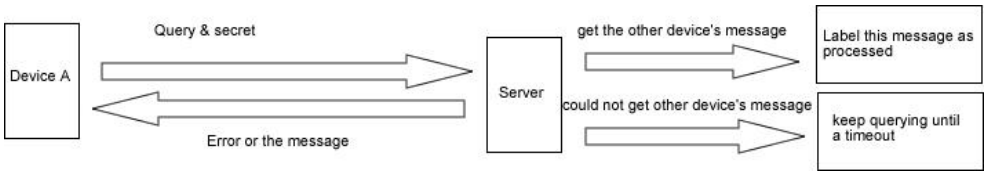

Figure 4.22: Download a Message

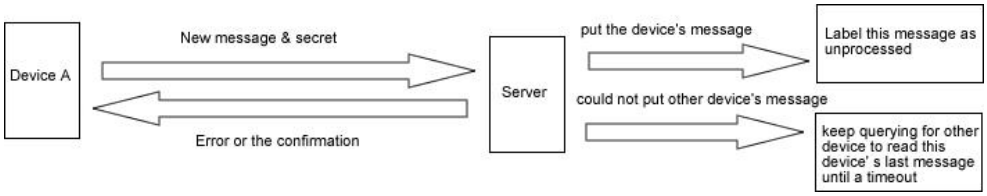

Figure 4.23: Upload a Message

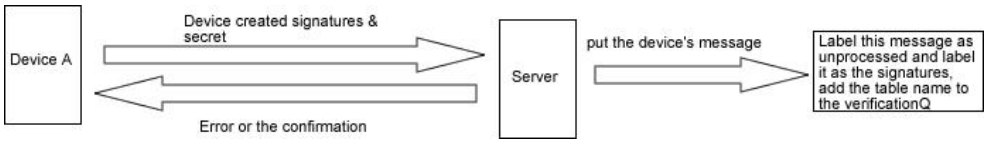

Figure 4.24: Upload Signatures

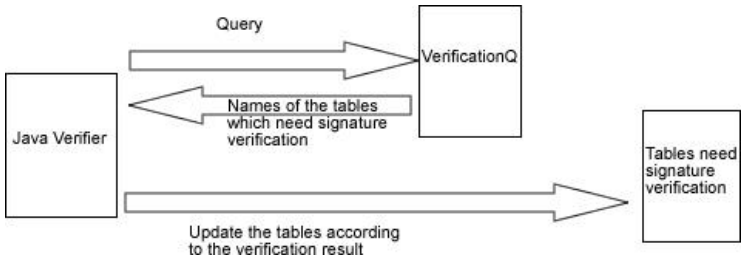

Figure 4.25: Server Verifies the Signatures

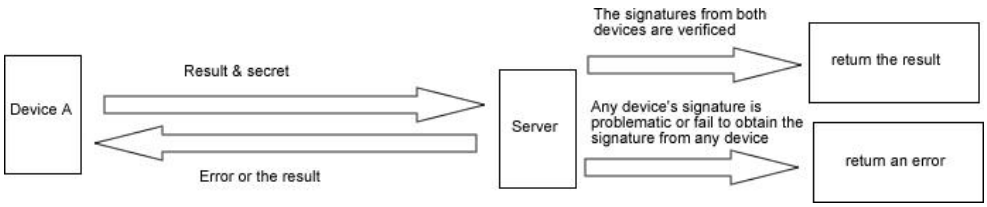

Figure 4.26: Query Matching Result

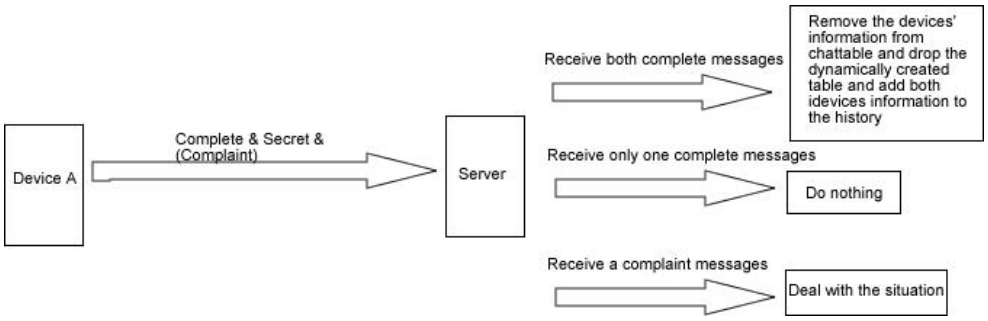

Figure 4.27: Complete the Protocol

device sends its information, such as its device id, public key, and the time to revisit the server, to the server. The server checks if there is a partner for this device. The server randomly generates and sends a secret (e.g. a cookie) to the device for future authentication. The server also informs the device about the partner's information if there is a partner for this device. The communication between the server and a client is encrypted by a session key that is shared between them. (They run an authenticated DH protocol to share a session key. As an alternative, they can communicate through HTTPS.) Figure 4.21 shows what a device should do if it does not find a partner in the first visit. Basically, the device has to resubmit its information along with the secret that it got from the server. Then, the server will either return it the partner's information or ask it to try again next time. Figure 4.22 shows how a device fetches the messages from its partner through the server. The device sends the secret that it got from the server and its query to the server. The server would return the result or an error if anything goes wrong to the device. Figure 4.23 shows how a device puts a message onto the server for the partner to retrieve. The device sends the secret and its message to the server. The server checks whether the other party has read the previous message from this device. (The reason for doing this is explained in the next paragraph.) If it has, the server inserts the message into the database. Otherwise, the server delays the insertion until the other party reads the previous message from this device. After that, a comfirmation will be sent to the device. Figures 4.24 and 4.25 show how a device submits its signed signatures to the server and how the server deals with the signatures. After a user submits her signatures to the server, a Java-based verifier program is going to verify the signatures and report the result to the server. Figure 4.26 shows how a device queries the server for their matching result. It is described in the step 6 of Figure 4.19. Figure 4.27 shows how a device informs the server about the completion of the protocol. If both devices do so, the server will clean up the chat logs between these two devices and record them into the history table, so that they do not get paired again in the next 24 hours.

We start measuring the execution time after a device downloads the other's public key from the dynamically created table. The public key information is stored into the database when a

device visits the server for the first time, and it will be moved to the dynamically created table by the server when a partner is found. We start measuring from this point because if we started measuring earlier, a device might wait for an arbitrary long time for a partner. Also, the public key information is the first message that they will get after both of them know that they found a partner. We finish measuring the execution time after the device verifies the partner's signatures issued by the *PIS*. In order to make our code clean, we require that a client downloads at most one message per visit (otherwise, there are more cases to handle). As a result, we also require a device not to upload a message to the server for its partner until its previous message has been read by the partner. We performed multiple experiments. Each of them consists of a number of individual tests. In each test, we run the implemented solution on both of the devices. For each of the experiments, most of the tests finish in about 7 or 8 seconds. However, this is not always the case. For example, around 4 or 5 o' clock in the afternoon, the execution time of the protocol becomes much longer (more than 30 seconds), and during our tests, we tried to avoid this time slot. Even when excluding this time slot, we still have some tests that take more than 10 seconds to complete. We did not further investigate the reason of this. As a representative example, we list the 6 tests from one of the experiments in Table 4.1 to show that WiFi is slower and more unstable than Bluetooth. Table 4.1 shows the distribution of the execution time of this protocol. In the table, we list the execution time and the distribution of the execution time of all 6 tests as well as the averages and standard deviations. The time to upload or download a message is recorded just before the device sends out the request and ended just after it receives the result or confirmation from the server. For simplicity, the execution time is tested on one device. The total execution time of both devices is about the same in most of the cases. In the tests, each user has 8 interests, and 5 of them are the same. We perform the tests in our lab at the University of Waterloo. From the table, it is obvious that the most time consuming part of this protocol is the data communication. Indeed, there is little computation involved except to sign and verify the signatures. As opposed to earlier experiments, we show individual measurement results for this experiment. For the Bluetooth experiments, the variation was low. For WiFi, the time for data communication is rather unstable, and it varies a lot from one test to another. Generally speaking, the execution time of this solution is longer than the protocols running through Bluetooth. One reason is that we did not find a way to establish HTTP persistent connections with BlackBerry devices, so that a device has to establish a new HTTP connection for each HTTP request. According to [11], we cannot reuse the same HTTPConnection Object in JavaMe. The other reason is that BlackBerry devices have issues with their WiFi [21]. Another drawback of this solution is that WiFi is more power consuming than Bluetooth. In addition, this solution requires extra hardware, such as a WiFi access point and a matchmaking server. As a result, this should only be considered a solution only if automatic Bluetooth pairing is not an option.

	1	2	3	4	5	6	Ave	St. Dev
Total execution time (ms)	6248	6136	6368	7484	6768	14956	7993.3	3446.1
Time to put dh public key (ms)	716	772	412	568	588	2312	894.7	705.6
Time to download dh public key & signature (ms)	828	1236	700	616	616	1748	957.3	451.4
Time to put dh signature (ms)	740	560	736	652	852	660	700	99.5
Time to sign signatures (ms)	976	952	1024	1012	980	976	986.7	26.5
Time to upload signatures (ms)	724	732	664	676	820	608	704	72.4
Time to query result (ms)	524	508	712	2056	728	4924	1575.3	1740.8
Time to exchange signatures (ms)	1280	972	1704	1500	1788	3336	1763.3	825.4

Table 4.1: Execution time

Chapter 5

Conclusion and Future Work

In this thesis, we presented an architecture for preserving users' interest information from unnecessary leaks in mobile social networking applications. In particular, we proposed and studied two matchmaking protocols. These two protocols are designed to allow only users with same interests to learn each other's interests that they have in common. As a byproduct, we also presented a shopping notification protocol. This protocol allows a user to query a store's item list to find out if there is any item she wants to purchase. This protocol hides the real item the user wants to purchase in a number of dummy items. We implemented a BlackBerry application that runs these protocols and evaluated the performance. During our implementation, we discovered that the BlackBerry Bluetooth has a feature that requires two devices to be manually paired before they can exchange information. We proposed several solutions for this problem and implemented one of the solutions.

One obvious and important problem we have to solve in the future is to fix the Bluetooth pairing problem. As we briefly mentioned before, we believe that it should be safe to allow arbitrary Bluetooth device to connect, if we carefully design the access control scheme for mobile devices. In the prototype we implemented, we focused on implementing the Matchmaking Module of our system. In the future, it would be necessary to complete the implementation of all other modules. For example, the Schedule, Group, and File Sharing Modules are not implemented yet. Right now, we simply display the matchmaking results to the screen on both devices. Later, a better handler should be added to allow devices to exchange users' contact information or other profile information according to the configurations. In the current stage, we display only the information to user's device if cheating is detected, and we should improve our implementation to report the malicious behaviours to our *PIS*. We should keep looking for better alternative matchmaking protocols that fit our setting. It would be interesting to perform more sophisticated matching as well. Right now, we allow only the exact matching (two terms should be exactly the same). We proposed a protocol for our shopping notification application. We would like to finish the

implementation of this protocol.

Mobile social networking has brilliant future, and there are also a lot of privacy-related problems. We believe that our architecture and protocols are able to solve part of the problems and motivate more useful solutions.

References

[1] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 86–97. ACM Press, 2003. ix, 6, 37

[2] Jan Camenisch and Anna Lysyanskaya. A Signature Scheme with Efficient Protocols. In *SCN*, pages 268–289, 2002. 6

[3] Jan Camenisch and Gregory M. Zaverucha. Private Intersection of Certified Sets. In *Financial Cryptography*, pages 108–127, 2009. 6

[4] Carlos Aguilar Melchor. High-Speed Single-Database PIR Implementation. `http://petsymposium.org/2008/hotpets/PIR.pdf`, 2008. accessed 07 2010. 49

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, page 41, Washington, DC, USA, 1995. IEEE Computer Society. 48

[6] Christina Warren. StreetSpark: Foursquare for Dating. `http://mashable.com/2010/05/11/streetspark/`. accessed 05 2010. 2

[7] Landon P. Cox, Angela Dalton, and Varun Marupadi. Smokescreen: flexible privacy controls for presence-sharing. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 233–245, New York, NY, USA, 2007. ACM. 4

[8] Emiliano De Cristofaro and Gene Tsudik. Practical Private Set Intersection Protocols with Linear Computational and Bandwidth Complexity. Cryptology ePrint Archive, Report 2009/491, 2009. `http://eprint.iacr.org/2009/491.pdf`. 6

[9] Daniel. Bluetooth Stack Test Methodology. `http://brainmurmurs.com/blog/2006/12/13/bluetooth-stack-test-methodology/`, 12 2006. accessed 07 2009. 51

[10] Daniel. What did RIM Do To Their Bluetooth Performance? http://brainmurmurs.com/blog/2006/12/12/what-did-rim-do-to-their-bluetooth-performance/, 12 2006. accessed 07 2009. 51

[11] Darryl L. Pierce. The JavaME Frequently Asked Questions List. http://bellsouthpwp.net/m/c/mcpierce/javamefaq.html#reuse_httpconnection_object. accessed 07 2010. 48, 76

[12] Nathan Eagle and Alex Pentland. Social Serendipity: Mobilizing Social Software. *IEEE Pervasive Computing*, 4(2):28–34, 2005. 4

[13] Facebook. Choosing_between_an_FBML_or_IFrame_Application. http://wiki.developers.facebook.com/index.php/Choosing_between_an_FBML_or_IFrame_Application. accessed 02 2010. 53

[14] Facebook. Facebook developer: Get started. http://developers.facebook.com/get_started.php. accessed 02 2010. 53

[15] Facebook. Statistics. http://www.facebook.com/press/info.php?statistics. accessed 07 2009. 1

[16] Michael J. Freedman and Antonio Nicolosi. Efficient Private Techniques for Verifying Social Proximity. In *Proc. 6th International Workshop on Peer-to-Peer Systems (IPTPS 07)*, Bellevue, WA, February 2007. 8

[17] Gatsby. About Gatsby. http://meetgatsby.com/about. accessed 05 2010. 2

[18] Hardware. Change or Modify Bluetooth Device hardware (MAC) address. http://www.siddharthabbineni.com/tech/hardware/change-bluetooth-device-mac-address.html. accessed 05 2010. 19

[19] Sachin Katti, Jeffery Cohen, and Dina Katabi. Information Slicing: Anonymity Using Unreliable Overlays. In *Proceedings of the 4th USENIX Symposium on Network Systems Design and Implementation (NSDI)*, April 2007. 30

[20] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols. In *CRYPTO*, pages 400–425, 2003. 27, 39

[21] Kyle McInneson. BlackBerry browser comparisons has WiFi been fixed? http://www.blackberrycool.com/2009/09/14/blackberry-browser-comparisons-has-wifi-been-fixed/, 09 2009. accessed 07 2010. 76

[22] Looptmix. Looptmix. http://www.looptmix.com/. accessed 05 2010. 2

[23] Justin Manweiler, Ryan Scudellari, and Landon P. Cox. SMILE: Encounter-Based Trust for Mobile Social Services. In *Proceedings of ACM CCS 2009*, November 2009. 5

[24] Sergio Mascetti, Claudio Bettini, Dario Freni, X. Sean Wang, and Sushil Jajodia. Privacy-Aware Proximity Based Services. In *MDM '09: Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 31–40, Washington, DC, USA, 2009. IEEE Computer Society. 8

[25] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. ix, 23, 24

[26] MOCOM2020. 4.1 Billion Mobile Phone Subscribers Worldwide. http://www.mocom2020.com/2009/03/41-billion-mobile-phone-subscribers-worldwide/, 03 2009. accessed 07 2009. 1

[27] Femi G. Olumofin, Piotr K. Tysowski, Ian Goldberg, and Urs Hengartner. Achieving Efficient Query Privacy for Location Based Services. In *Privacy Enhancing Technologies*, pages 93–110, 2010. 48

[28] A-K Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot. Mobiclique: Middleware for Mobile Social Networking. In *WOSN'09: Proceedings of ACM SIGCOMM Workshop on Online Social Networks*, August 2009. 2, 4

[29] Podnetadmin. PodNet - Mobile Distribution of User-generated Content. http://www.podnet.ee.ethz.ch/. accessed 05 2010. 2

[30] Research In Motion Limited. Wi-Fi enabled BlackBerry smartphones. http://na.blackberry.com/eng/ataglance/networks/WiFiCellularWhitepaper.pdf. accessed 07 2010. 68

[31] RIM. BlackBerry JDE 4.5.0 API Reference: Class DHKeyAgreement. http://www.blackberry.com/developers/docs/4.5.0api/net/rim/device/api/crypto/DHKeyAgreement.html. accessed 07 2009. 60

[32] Ji Sun Shin and Virgil D. Gligor. A New Privacy-Enhanced Matchmaking Protocol. In *16th Annual Network & Distributed System Security Symposium*, 2008. 5

[33] Li Shundong, Wang Daoshun, Dai Yiqi, and Luo Ping. Symmetric cryptographic solution to Yao's millionaires' problem and an evaluation of secure multiparty computations. *Inf. Sci.*, 178(1):244–255, 2008. 7

[34] Michael Terry, Elizabeth D. Mynatt, Kathy Ryall, and Darren Leigh. Social net: using patterns of physical proximity over time to infer shared interests. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 816–817, New York, NY, USA, 2002. ACM. 5

[35] Unknown. Pair a Windows Mobile device and a Windows XP SP2 Computer. `http://www.sembee.info/windowsmobile/sync-bluetooth2.asp`. accessed 05 2010. 66

[36] User udi. Obex Push messages to from Nokia to BB using J2ME. `http://supportforums.blackberry.com/t5/Java-Development/Obex-Push-messages-to-from-Nokia-to-BB-using-J2ME/m-p/228088;jsessionid=A8D7AFF45C647A12ADC5DDD35F323EB2`. accessed 05 2010. 66