INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A Secure, Anonymous and Scalable Digital Cash System

Feng Xue

School of Computer Science
McGill University, Montreal
August 1999

A thesis submitted to the
Faculty of Graduate Studies and Research
In partial fulfillment of the requirements for the degree of
Master of Science

Table of Contents

List of Figures and Tables

Résumé

Les signatures *à l'aveugle* réalisent l'anonymat dans les systèmes de paiements électroniques. Cependant, une fois déployées dans les systèmes d'argent numérique tels que « *eCash* », les signatures *à l'aveugle* engendrent des inconvénients tels que la mauvaise résistance aux changements d'échelle et l'anonymat non garanti. Dans ce mémoire, nous proposons de combiner les signatures numériques à un autre mécanisme appelé « *Money-exchange* » existant dans « *NetCash* » pour élaborer un nouveau système de paiement électronique. Dans le nouveau système, deux formes d'argent numérique sont introduites : les billets numériques et la monnaie numérique. Nous avons étendu le mécanisme « *Money-exchange* » pour permettre l'échange d'argent d'une forme à une autre. Du côté de l'émetteur d'argent numérique, au moins deux bases de données sont maintenues pour détecter le double paiement. Ce nouveau système d'argent numérique est très sécuritaire du fait qu'il utilise deux algorithmes de cryptage symétrique et asymétrique. La combinaison des signatures *à l'aveugle* avec notre extension du mécanisme « *Money-exchange* » offre un anonymat complet et inconditionnel. Cette combinaison permet aussi une meilleure résistance aux changements d'échelle du système par rapport au nombre de clients à servir.

Abstract

Blind signatures make anonymity a reality in digital cash systems. However, when deployed in digital cash systems such as *eCash*, blind signatures raise such drawbacks as bad scalability and unfair anonymity. In this thesis, efforts have been done to combine digital signatures and a mechanism called "money-exchange" found in *NetCash* to build a new digital cash system. In the new system, two forms of digital cash are introduced: digital notes and digital coins. "Money-exchange" is extended to permit cash exchange from one form to the other. At the side of the digital cash issuer, at least two databases are maintained to detect double-spending. The new digital cash system is secure due to its deployment of both symmetric and asymmetric encryption algorithms. The combination of blind signatures and the extended money-exchange mechanism offers unconditional and fair anonymity, and it makes the system more scalable with the regards to the number of clients served.

Acknowledgements

First of all, I wish to give my thanks to my thesis supervisors Professor Petre Dini and Professor Claude Crépeau for their guidance, advice, and encouragement throughout the research. This thesis benefits from their careful reading and constructive criticism.

I truly thank CRIM for supplying me a wonderful researching environment and the generous financial supports.

I also wish to thank the School of Computer Science for the graduate courses and the research environment. Thanks to our graduate secretary Franca Cianci for her wonderful works.

Finally, I am especially grateful for the supports and encouragements from my wife Tiao during the graduate studies and research.

Introduction

A secure, anonymous and scalable digital payment system is critical to persuade people into e-commerce activities. Since 1980s, many digital payment systems, mechanisms and protocols have been proposed, and among them, some are being evaluated or even have been commercially deployed. There are many forms of electronic payment systems just as there are many forms of traditional payment instruments. Generally, they fall into three categories: secure credit card system, credit-debit system, and digital cash system. Each one has its advantages as well as disadvantages. For example, the secure credit card system is the easiest one to implement, and it is the most similar to the current conventional bank payment systems. However, clients' privacy such as purchase habits is exposed to the financial institutions. Therefore, it does not offer anonymity. Currently, it is widely accepted that digital cash systems stand for the future of digital payment systems, because they offer anonymity to individuals.

The fundament of digital cash systems for offering anonymity is blind signature. Blind signature extends RSA digital signature algorithm in such a way that a message is concealed from the signer when it is being signed. When blind signatures are deployed in digital cash systems, they give a client a way to hide the identities of digital cash when they are withdrawn from a bank. Thus, the spending pattern of the client with the bank-blindly-signed digital cash is undetectable to others. *eCash* is such a digital cash system which makes full use of blind signature mechanism. However, it has been pointed out that blind signatures also produce some shortcomings. First, the scalability of the system is quite unsatisfactory due to the deployed mechanism of detecting double spent digital cash, and the bad performance of the mechanism is due to the deployment of blind signatures. Second, anonymity offered is not fair. Only payers are unconditionally anonymous, while payees are not.

There is another presented digital cash system, *NetCash*, which offer anonymity through a mechanism called money exchange. Although the provided anonymity is not unconditional, it is fair to both payers and payees and the performance of its double-spending detection mechanism is better than that of *eCash*. It is imaginable that combining blind signature and cash exchange could be a practical solution for building a better digital cash system.

The main purpose of this thesis is to present the attempts of building this new system. The contents of the paper are divided into five chapters. Chapter 1 introduces the current states of e-commerce and discusses cryptographic techniques. Chapter 2 addresses variant digital payment systems. Chapter 3 delves into two digital cash systems *eCash* and *NetCash*, with emphasis on blind signatures and money exchange mechanisms, respectively. In Chapter 4, the attempts of combining *eCash* and *NetCash* are presented in details. In Chapter 5, a partial implementation of the presented system is included. Finally, conclusions of the thesis and future works are given.

Chapter 1 E-commerce and Cryptographic Techniques

1.1 E-commerce and digital payment systems

Information technologies are significantly changing the ways we store, distribute and access information. Financial information is also greatly impacted by these technologies. In recent years, electronic commerce (*e-commerce*) has created quite a buzz in the world of finance, commerce and trade as well as IT industry. So what is e-commerce?

E-commerce doesn't always mean buying and selling goods and services on the Internet. In fact, it has been found [ITAA98] that the e-commerce activities are split between business-to-customer transactions (such as sales and customer service), and business-to-business transactions (such as those supporting sales, order processing, and resource management). Also it is discovered that the latter even has made more significant gains recently. Therefore, any activities involving financial, commercial or business information exchanges via telecommunication means such as telephone and computer networks may be regarded as e-commerce activities. In this thesis, the discussion is focused on one aspect of the business-to-customer transactions, the digital payment systems.

It is widely accepted that the Internet will be the main media and means for e-commerce because of its ever-increasing popularity. At the time of writing, it has been estimated that world total population online is 158 million [Nua]. However, among them, it is found that 82.5% do one, less than one or none purchase each month [GVU10]. And it is also discovered [GVU10] that only 4.3% are not at all concerned about security when making purchases or banking over the Internet. It is estimated [ITAA98] that more and more transactions are either solely Internet-based or based on a mix of proprietary networks and the Internet. Current e-commerce doesn't fulfill people's expectation for a secure and privacy-respecting e-commerce infrastructure. This state is in a large degree due to the vulnerable characteristics of Internet itself. We must accept the fact that e-commerce is

still under definition and construction, and a lot of work is needed to do to make the Internet a mature digital market.

Digital payment systems have been the prime target for research in e-commerce for over two decades. This should not be a surprise. Commerce always involves a payer, (who is called Alice in this thesis), a payee, (who is called Bob) and at least one financial institution, (who is named Cyber Banque). Analogous to the current finance and trade markets based on traditional payment instruments such as cash, checks, credit cards, debit cards, etc, a digital market must be based on a robust, secure and efficient digital payment scheme. Only after such a digital payment system is built up, a digital market with all the traditional commerce elements such as shopping, bidding and bargaining, brokerage, and delivery may be constructed.

1.2 Cryptography

Security is the most concerned in a digital payment system. Cryptographic techniques are the most often applied in various payment systems for security purposes. This section starts the introduction on some basic techniques in cryptography by addressing variant security attacks against the communication across an open network

1.2.1 Security Attacks

In the context of the communication across a network, the following attacks can be identified:

- Disclosure: Release of message contents to any unauthorized person.
- Traffic analysis: Discovery of the pattern of traffic between parties.
- Masquerade: Insertion of messages into the network from a fraudulent source.
- Content modification: Changes to the contents of a message, including insertion, deletion, transposition, and modification.

- Sequence modification: Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.
- Timing modification: Delay or replay of messages.
- Repudiation: Denial of receipt of message by destination or denial of transmission of message by source.

1.2.2 Security Services

Four general security services encompass the various functions required for an information security facility.

- *Confidentiality* ensures that the information is protected against unauthorized disclosure. In a digital payment system, disclosure of critical transaction information such as credit card numbers to illegitimate parties may cause a disaster.

- *Authentication* ensures that the origin of the information or the identity of a participant is correctly verified.

- *Integrity* ensures that the information is protected against unauthorized modification, and indicates whether or not such modification has occurred. In a digital payment system, attackers could attempt to change the transaction information by deleting, replaying or altering a message. For example, a malicious attacker could intercept the payment information from Alice to Bob, change some critical data such as the payment amount, and then retransmit it to Bob.

- *Non-repudiation* ensures that any party involved in the information transmission cannot deny either the participation in or the content of that transmission. In a digital payment system, for example, Alice could fraudulently claim that she did not, (in fact she did), authorize a payment to Bob, which could cause profit loss to Bob.

1.2.3 Cryptographic techniques

The first concern of cryptography is how to keep communications private. *Encryption* ensures privacy by keeping information hidden from anyone for whom it is not intended. Decryption is the reverse of encryption; it transforms encrypted data back into an intelligible form.

However, nowadays cryptography is more than encryption and decryption. Authentication is as important as privacy when information is communicated electronically. Digital signatures and digital certificates are among the cryptography mechanisms to offer authentication. In this section, these cryptographic techniques are discussed that are quite critical for developing digital payment systems.

1.2.3.1 Encryption and Decryption

How to render a message incomprehensible to an unauthorized reader is the traditional aim of cryptography. The words, characters, or letters of the original intelligible message contribute the *plain text*. The words, characters, or letters of the secret form of the message are called *cipher text*. The process of converting plain text into cipher text is *encryption*. The reverse process of converting cipher text into plain text is *decryption*. An encryption/decryption algorithm and the associated encryption key and decryption key are required for the processes. (See Figure 1.) The algorithm is a set of mathematical rules to determine the transformation process of encryption and decryption. The keys act as a parameter of encryption/decryption algorithm and control the transformation processes. If P is the plain text message, and E and D are the encryption and decryption algorithms respectively, it is required that

$$D(E(P)) = P$$

That is, decrypting the encrypted message results in the original plain text message.

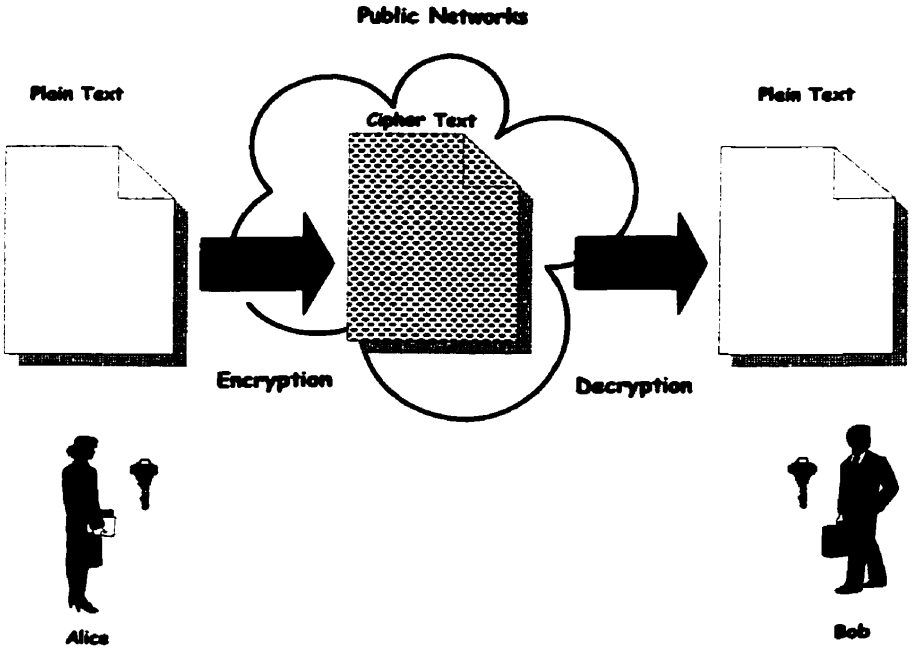

Figure 1 - Encryption and decryption

Confidentiality of transaction information in a digital payment system can be achieved by applying encryption techniques. Alice and Bob can agree on a certain method of encryption and decryption prior to the transmission of transaction data to ensure that the data is not understandable to others.

Note that, while modern cryptography is growing increasingly diverse, it is fundamentally based on problems that are difficult to solve. The problems are hard because they are intrinsically difficult to complete. Prime factorization of a large integer is an example of such problems. Given a large integer n, it is not easy to find the prime factors p and q such that $n = pq$. In fact, factoring is the underlying, presumably hard problem upon which several cryptography algorithms are based, such as the *RSA* algorithm, (refer to section 1.2.3.4). The security of the *RSA* algorithm depends on the unproved assumption of factoring problem being difficult and the presence of no other

types of attack[1]. The theory of computational complexity is relevant here, since it classifies algorithms according to their difficulty. Difficulty in this case refers to the computational requirements in finding a solution. If an encrypted message cannot be decrypted by an attacker within practical time and with acceptable amount of resources, it loses its value for the attacker. In other words, whether a cryptography system is secure or not is relative. For example, in 1997, it was found that, a *RSA* 512-bit key might be factored for less than $1,000,000 in cost and eight months of effort [Rob95]. Therefore, an *RSA* 512-bit key at that time was not secure enough for corporate use, while it might be secure enough for casual personal use. Today, it is recommended the key sizes of 768 bits for personal use, 1024 bits for corporate use, and 2048 bits for extremely valuable keys [RSA98].

Encryption/decryption algorithms fall into two categories: symmetric algorithms and asymmetric algorithms.

- ***Symmetric encryption***

Symmetric encryption, also referred to as secret-key encryption, makes use of an algorithm that applies a unique key for both encryption and decryption. The key is often referred to as secret key.

With the plain text P and the encryption key K as input, at the side of the sender Alice, the encryption algorithm computes the cipher text C denoted as

$$C = E(P, K)$$

The intended receiver Bob, in possession of K, is able to invert the transformation:

$$P = D(C, K)$$

[1] There has been some recent evidence that breaking RSA is not equivalent to factoring [DB98].

The fact is the encryption algorithm E and the decryption algorithm D are publicly known. The attacker Jack, observing C and having knowledge of E and D, but not possessing K or P, will find that it is impractical to recover P or K or both P and K.

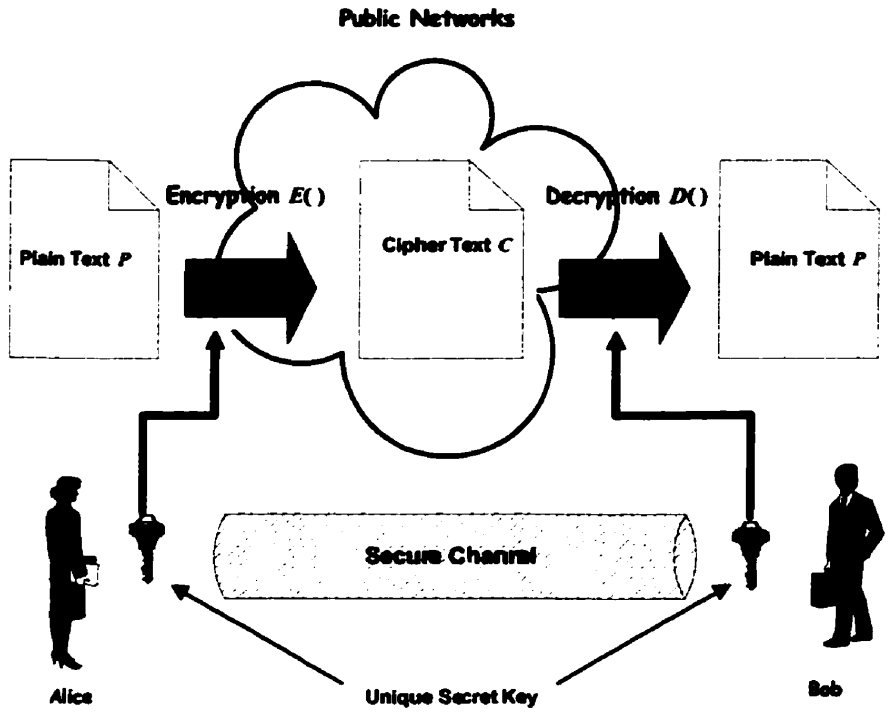

Figure 2 - Encryption and decryption with symmetric algorithms

The security of a symmetric encryption is based on the assumption that the sender and the receiver exclusively share the key. Therefore, when deploying symmetric encryption, the principal security problem is maintaining the concealment of the key. If Alice generates the key, she must also transmit it to Bob by means of some secure channel, such as a confidential mail. (See Figure 2.) Alternatively, a third party, trusted by both Alice and Bob, could generate the key and securely deliver it to both Alice and Bob.

The most widely used symmetric encryption algorithm is Data Encryption Standard (*DES*).

- ***Asymmetric encryption***

Asymmetric encryption algorithms are also referred to as public-key algorithms. Instead of using the same key, an asymmetric algorithm makes use of two separate but mathematically related keys for encryption and decryption respectively. The creator of the key pair maintains one key secret (the *private key*) and makes the other key (the *public key*) known to anybody he or she may want to correspond with. A message encrypted with one key of the pair can only be decrypted by the other. It is not computationally possible to deduce one key from the other given knowledge of the algorithm. Thus, a message encrypted by Alice with Bob's public key can only be decrypted by Bob using his private key. No other people can decrypt the message because only Bob knows his private key. (See Figure 3.)

With this approach, all participants have access to public keys of the correspondents, and private keys are generated and kept locally by each participant and therefore never need to be distributed. As long as the private keys are protected from disclosure, the communication is secure.

In this thesis, a public key is represented as the letter K with a subscript naming the owner; for instance, K_{Bob} stands for Bob's public key. The associated private key of Bob is represented as K^{-1}_{Bob}.

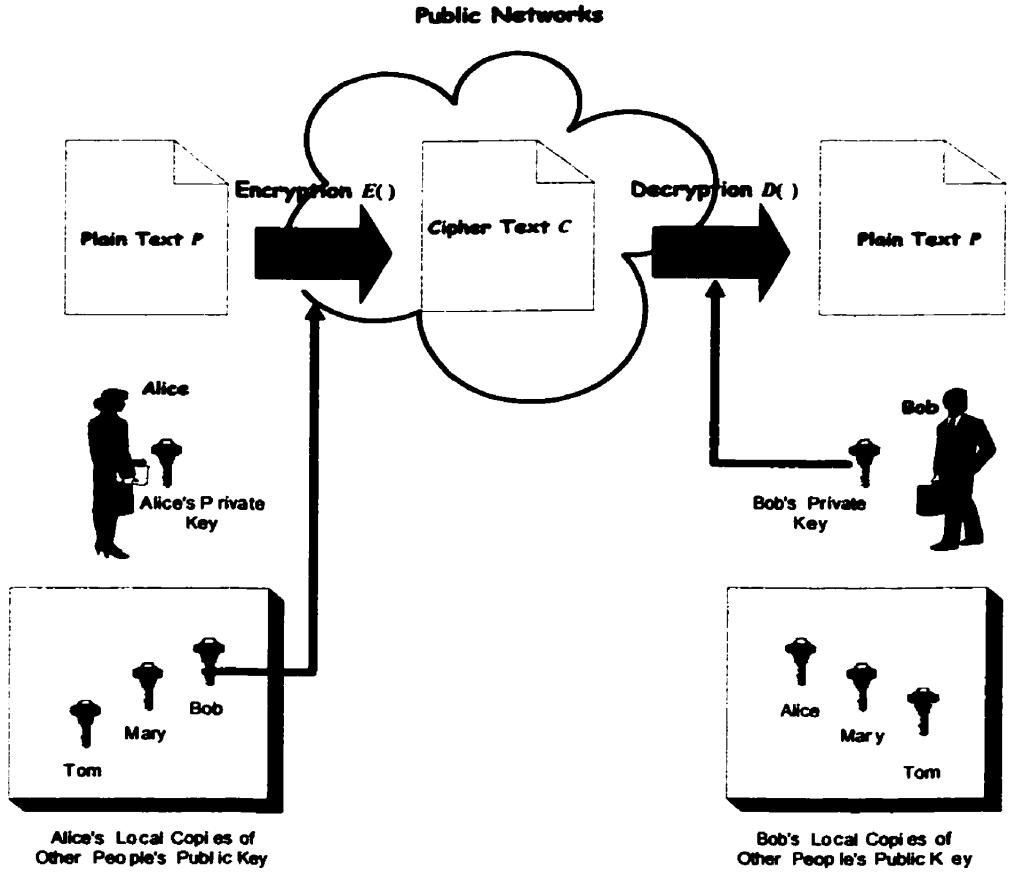

Figure 3 - Encryption and decryption with asymmetric algorithms

With the plain text message P intended for Bob and Bob's public key K_{Bob} as input, Alice forms the ciphered message C with the encryption algorithm E:

$$C = E(P, K_{Bob})$$

Bob, in possession his matching private key K^{-1}_{Bob}, is able to invert the transformation to recover the plain text with the decryption algorithm D:

$$P = D(C, K^{-1}_{Bob})$$

The attacker Jack, observing C, having access of K_{Bob} and having knowledge of E and D, but not possessing K^{-1}_{Bob} or P, will find that it is impractical to recover P or K^{-1}_{Bob} or both P and K^{-1}_{Bob}.

The *RSA* algorithm is the most widely accepted and implemented general-purpose approach to asymmetric encryption. The *RSA* algorithm will be discussed at the end of the chapter.

1.2.3.2 Digital signatures

Digital signatures are fundamental in authentication, authorization, and non-repudiation. Although asymmetric cryptography is the mostly employed means for digital signatures, symmetric cryptography can also be used to authenticate a message.

- ***Message authentication codes (MACs)***

A message authentication code (*MAC*) is an authentication tag (also called a checksum) sent along with the original message. A *MAC* is computed as a function of the message M and a secret key K shared by the sender and the receiver. That is, $MAC = f(M, K)$.

The receiver, knowing f and K, can also compute the *MAC* and compare it with the received one. If they match, the message's integrity and the sender's identity are correctly verified.

It should be noted that a sender could repudiate a previously authenticated message by claiming the secret was somehow compromised by the recipient who shares the secret. There is no means within *MACs* to resolve this dispute.

- ***Hash functions and message digests***

A hash function H is a transformation of the form:

$$h = H(x)$$

where x is a variable-length message and h is the fixed-length hash value. The hash value is also called the message digest. A hash function H employed in cryptography is usually chosen to have also the following properties:

- $H(x)$ is easy to compute for any given x ,
- $H(x)$ is one-way,
- $H(x)$ is collision-free.

For any given code h, if it is computationally infeasible to find x such that $H(x) = h$, the function H is referred to as one-way. For any given message x, if it is computationally infeasible to find $y \neq x$ such that $H(y) = H(x)$, H is referred to as weak collision-free. If it is computationally infeasible to find any pair (x, y) such that $x \neq y$ and $H(x) = H(y)$, H is referred to as strong collision-free.

Two well-known hash functions are Message Digest 5 (*MD5*) and Secure Hash Algorithm (*SHA-1*).

One of the main roles of a cryptographic hash function is in the provision of digital signatures.

- ***Digital signatures***

It is a good solution to authenticate messages using a secret key if the parties involved are confident in the key's secrecy and if no disputes can arise. However, such a solution can never prove that a message came from the sender. Since both the sender and the receiver know the key, either one could have sent the message. Verifying the sender's identity and resolving disputes require digital signatures.

A digital signature of a message is a block of data dependent on some secret known only to the signer, and, additionally, on the content of the message being signed. A digital signature is verifiable; if a dispute arises as to whether Alice signed a message, an unbiased third party should be able to resolve the matter equitably, without requiring access to Alice's secret information.

Thus, a digital signature scheme consists of a signature generation algorithm and an associated verification algorithm. A digital signature generation algorithm is a method for a signer to produce a digital signature on a particular message. A digital signature verification algorithm is a method for verifying that a digital signature is authentic, (i.e., was indeed created by the specified signer).

A digital signature generation algorithm (the signing process) must use some secret information that is only accessible to the signer. A digital signature verification algorithm (the verifying process), in contrast, uses some public information about the signer to verify the signature. Some public-key cryptography algorithms, among other mechanisms, can be deployed as digital signature schemes: the signer signs a message with his or her private key, while anyone else may verify the signature using the public key of the signer. It should be noted that with some public-key cryptography algorithms such as *RSA*, signing a message is actually employing the encryption algorithm on the message with the signer's private key, and verifying a signature is actually employing the decryption algorithm on the signature with the signer's public key. But this is not always the case for all public-key systems. The following discussions on digital signature mechanisms are based on public-key cryptography algorithms that are the likes of *RSA* algorithm.

Suppose Alice wants to send a signed message M to Bob (see Figure 4). Generally the first step is apply a one-way hash function H to the message, creating a message digest h.

$$h = H(M)$$

To create a digital signature S, the signing process usually signs the message digest h instead of the message itself. In this way, it saves a considerable amount of time, because a message digest is shorter compared to the message itself.

Next, by encrypting the message digest h with her private key K^{-1}_{Alice}, Alice actually creates a digital signature S on h with a secret of herself:

$$S = Sign(h, K^{-1}_{Alice})$$

Alice sends Bob the signature S together with the message. In order for Bob to verify the signature, he must first apply the same hash function H as Alice did to the message M she sent him:

$$h_1 = H(M)$$

Then he verifies Alice's signature S using her public key:

$$Verify(S, K_{Alice}, h_1)$$

The verification process above is actually accomplished in two steps. First, it decrypts S with Alice's public key K_{Alice} to get a value h_2. Second, it compares h_1 and h_2: if they are the same, it means that the signature is successfully verified; otherwise, the verification fails, that may suggest that either someone is trying to impersonate Alice, or the message itself has been altered since Alice signed it, or an error occurred during the transmission.

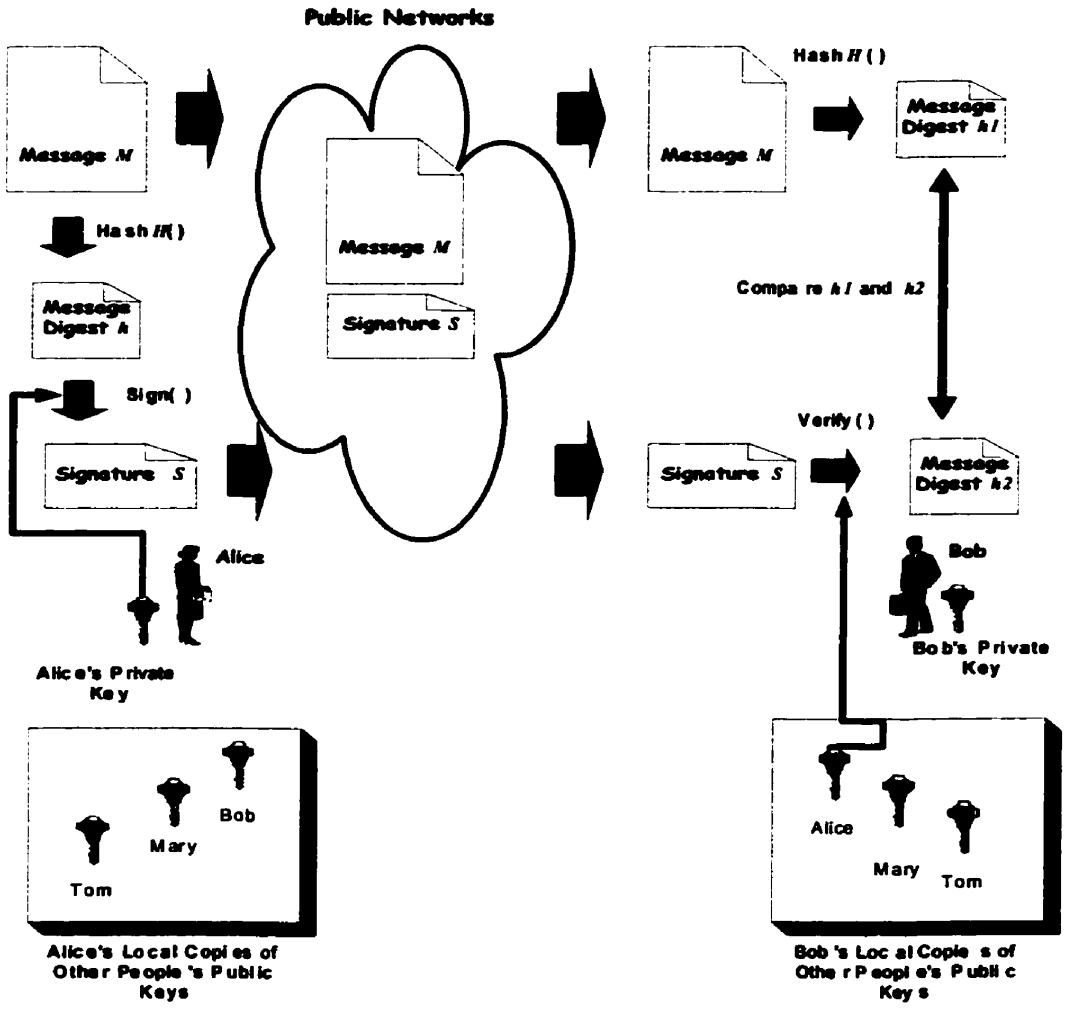

Figure 4 - Digital signatures with public-key algorithms

1.2.3.3 Certificates and certification authorities

Public keys are sometimes considered simpler to manage because they can be distributed across insecure channels. However, for example, when Alice wants to communicate with Bob she must be assured that she is using Bob's correct and authentic public key, and if

she receives a signed message from Bob she must have a way to check that the signature is authentic and still valid.

The usual solution is for users to register their public keys with a trusted registry or certification authority (*CA*). The CA distributes certificates, which are public keys carrying the digital signature of the CA. To verify a certificate, a user needs only the CA's public key that can be published in many ways and places.

A certificate contains more than just the public key. The user's public key is combined with other significant information before signed by the CA. Such information may include the name of CA, validity period (start and finish dates) and the name of the user.

1.2.3.4 RSA

RSA is an asymmetric cryptography system that offers both encryption and digital signatures [RSA78]. In this thesis, *RSA* will be employed to offer some most important features of the presented digital cash system, so in this section, a brief description of the algorithm is given.

RSA works as follows:

1. Randomly generate two large primes, p and q.
2. Compute their product $n = pq$; n is called the public modulus.
3. Select a number, e, less than n and relatively prime to $(p-1)(q-1)$, which means e and $(p-1)(q-1)$ have no common factors except 1.
4. Compute another number d, the multiplicative inverse of e modulo $(p-1)(q-1)$, which means $(ed - 1)$ is divisible by $(p-1)(q-1)$. (d is also denoted as e^{-1} or $1/e$.)

The public key is the pair (n, e); the private key is d. The value e is called the public exponent; the value d is called private exponent. The factors p and q may be secretly kept with the private key, or destroyed.

- ***RSA Encryption***

Suppose Alice wants to send Bob a message p, and she is in possession of Bob's public key (n, e). She creates the cipher text c as:

$$c = p^e \bmod n$$

She sends c to Bob. By decrypting it with his private key d, Bob get the plain message:

$$p = c^d \bmod n$$

The relationship between e and d ensures that Bob correctly recovers p. Moreover, since only Bob is in possession of d, no one else other than Bob can decrypt this message.

- ***RSA Digital Signature***

Suppose Alice wants to send Bob a digitally signed message p, and her public key (n, e) is known to Bob. Alice creates a digital signature s on p with her private key d:

$$s = p^d \bmod n$$

She sends p and s to Bob. To verify the signature, Bob checks if the following is satisfied:

$$p = s^e \bmod n.$$

Chapter 2 Digital Payment Systems

In Chapter 1, several cryptographic techniques, encryption/decryption, digital signatures and digital certificates, are discussed respectively. In this chapter, applications of these techniques on various digital payment systems are addressed. Digital payment schemes generally have three models: *secure credit card systems*, *credit-debit systems*, and cash-like system, (also called *digital cash systems*).

2.1 Secure credit card systems

This model enhances the traditional credit card payment system by securely protecting the credit card number from eavesdropping. This model is anticipated to be the most acceptable to all entities because it is based on a mechanism that both consumers and merchants are familiar with, and it maintains the benefits of the government and the traditional financial institutes. One proposal of this payment model, *iKP*, is shown in the following subsection, and other well-known proposals are addressed at the end of the section.

2.1.1 iKP

The *iKP* protocols are a family of protocols – *iKP* (*i* = *1*, *2*, *3*) – for secure digital payments over the Internet. The protocols implement credit card-based transactions between customers and merchants using the existing financial networks for clearing and authorization. [iKP95]

The reason *iKP* was focused on the credit card payment model is that its developers believed that this model "is anticipated to be the most popular in the future." However at the same time, it is claimed that they "can be extended to apply to other payment models, e.g., debit cards and electronic checks". [iKP95]

*iKP*s make use of public-key cryptography. As the increase of the number of parties that possess their own public/private key pairs, indicated in the name of each protocol by *i*, the *iKP* protocols offer increasing level of security and complexity.

- *1KP*, the simplest in the family, requires that only the "acquirer gateway" (the *iKP* term for a bank) possess public/private key pair. Customers and merchants need only to possess the authentic public key of the gateway, or the authentic public key of an "authority" that validates the gateway's public key via a signed certificate. [iKP95]

  When Alice makes payment to Bob, she encrypts her credit card number and possibly associated PIN with the public key of Cyber Banque, and binds them with the relevant purchase information. She then sends the message to Bob, who immediately forwards this message to Cyber Banque for verification. Cyber Banque decrypts the message with its private key, checks the card number and possibly the PIN. If they are valid, the bank will authorize the transaction by digitally signing it with its private key, then return it to Bob. It is easy to see that customers of *1KP* are actually authenticated based on their credit card numbers, and possibly associated secret PINs. In addition, since Alice's card number and PIN are encrypted with Cyber Banque's public key, therefore no one except Cyber Banque will be able to uncover the secrets.

  1KP does not offer non-repudiation for messages sent by customers and merchants. There is no mechanism for the acquirer to get a proof of transaction authorization by merchants. The proof of transaction authorization by customers are the credit card numbers and PINs, which are deniable proof since the disclosure of these secrets is not impossible.

- *2KP* requires that merchants should, in addition to acquirer gateways, possess public/private key pairs and key certificates. Non-repudiation is consequently offered for messages originated by merchants. This is achieved by requiring the merchants digitally signing the transaction data with their private keys. Moreover, *2KP* enables customers to verify that they are dealing with genuine merchants. This is achieved by

checking the merchants' certificates issued by a central certification authority (CA). In *2KP*, still no mechanism for authenticating payment orders originated by the customers, therefore non-repudiation is not offered for messages proposed from the customer.

- *3KP* requests that each involved party has a public/private key pair, therefore it provides full multi-party security. It achieves non-repudiation for all messages of all parties involved. Payment orders from the customer are authenticated both by the credit card number (and PIN), and a digital signature of the customer generated with her private key. Merchants consequently can authenticate the customer with her public key. A CA is required to provide certificate of the customer's public key as before.

The developers expect that, as public key technology becomes more pervasive, more and more parties will hold public/private key pairs. A gradual deployment of the *iKP* protocols thus makes sense: begin with *1KP*, then move to *2KP* and finally to *3KP*. [iKP95]

*iKP*s provide a customer some privacy or anonymity against the merchants in the sense that "the customer uses a pseudo-identity ... which is different in each transaction". However, since the system follows the credit card-based payment model, it requests the customers to reveal their identities for the credit card number verification, so it doesn't offer customers anonymity against the payment system provider.

2.1.2 Other systems in this category

There are many other digital payment proposals which fall into the category of secure credit card systems, such as *SET* (Secure Electronic Transaction) [SET] and *CyberCash* [CyberCash]. One of the common features among them is that they all deploy cryptography techniques for security reasons, and as more parties involved possess a public/private key pair, more security features are achieved, just like *iKP*.

2.2 Credit-Debit Systems

The common feature of payment systems in this category is there is a centralized accounting server who transfers balances between two accounts. There are no separate, identifiable digital tokens that can be stored on a hard drive or in the memory of a smart card, and subsequently transferred to others in payment without going through the centralized server. In other words, one can send a digitally signed payment message to the accounting server which results in a balance transfer, but one doesn't receive from the server a digital token or piece of digital cash carrying the server's signature, and promising to work as payment instruments on demand. A typical proposal in this category, *NetBill*, is described in this section.

NetBill

NetBill is designed by Carnegie Mellon University and Mellon Bank Corp. for micro-payments, especially for information delivered over the Internet, such as payment of a few cents for access to a Web page or an electronically archived research journal. The system attempts to guarantee that customers receive the information they request, and that merchants receive payment for goods delivered. [NetBill]

NetBill uses an account server, which maintains accounts for both customers and merchants, linked to conventional financial institutions. The *NetBill* server acts as an aggregator to combine many small transactions into larger conventional-sized transactions.

The payment structure is designed for the easy construction of pseudonyms to allow buyers to protect their identities.

The basic steps in the payment protocol are the following:

1. Alice requests a price quote from Bob.

2. Bob responds with a quote to Alice.
3. Alice accepts or rejects the price quote.
4. If she accepts, Bob delivers the information in encrypted form.
5. Alice sends an electronic payment order to Bob.
6. Bob sends the electronic payment order and key to the *NetBill* server.
7. The *NetBill* server sends a receipt to Bob.
8. Bob sends a receipt to Alice, allowing Alice to decrypt the information.

In step 1, Alice and Bob authenticate each other using public-key certificates. They establish a symmetric session key to encrypt subsequent messages. Bob's price quote (step 2) will be based on Alice's identity (to allow for price discrimination). If Alice accepts the quote (step 3), Bob sends the purchased information to her in an encrypted form while withholding the encryption key K (step 4). Alice makes up an electronic payment order that describes the transaction and includes a message digest of the information from Bob in step 4. The payment order is signed with the private key of Alice and sent to Bob (step 5). Bob verifies the message contents using Alice's public key, appends the key K, endorses the order with his digital signature, and sends it to the *NetBill* server. The *NetBill* server checks Alice's account balance, debits this balance and credits Bob's balance, and sends a digitally signed receipt—which includes the key K—back to Bob (step 7). Bob forwards the receipt to Alice (step 8), who now has the key K to decrypt the purchased information.

There is little personal anonymity in this process: the *NetBill* server keeps transaction records of purchases made from different merchants, and this potentially allows all information requests to be linked.

2.3 Digital cash systems

In a digital cash system, there exists at least one bank, (a cash server usually), minting digital cash which a customer can withdraw from and deposit to the bank, keep in his or her digital purse, (which is usually a smart card or a PC hard disk), and pay to others.

Therefore, there are at least three entities, a bank, a payer and a payee involved in at least three protocols, withdrawal, payment and deposit. (See Figure 5.)

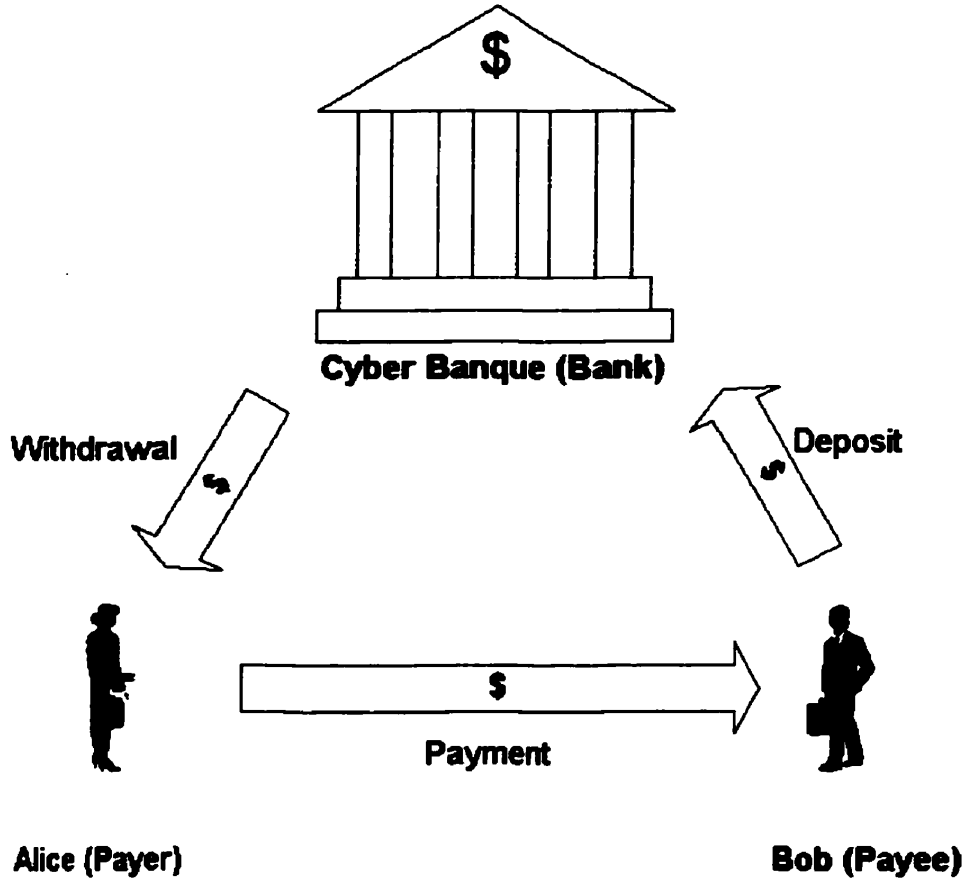

Figure 5 - Entities and protocols in typical digital cash payment systems.

2.3.1 *Properties of digital cash systems*

A digital cash system is an electronic payment system, so it must offer security first. Beyond that, distinctive from a secure credit card system or a credit-debit system, a digital cash system bears other properties such as anonymity (untraceability and unlinkability), transferability and divisibility etc.

- **Security**

Security means more for digital cash systems. First, it must be guaranteed that any entity will not suffer from any malicious attacks during the flow of value before, during and after one payment.

Additionally, for the payer Alice, she must be assured that:

1. She is withdrawing genuine digital cash of the exact amount as she required from the verified bank where she has an account set up.
2. After she deposits some amount of digital cash to her account, she is provided such proof from the bank that the bank won't be able to deny the receipt of the deposit.
3. The money will not be stolen by others while transferred via the network.
4. After she pays Bob (the merchant) for some merchandise or service, some proof is provided by him so that he won't have the ability to repudiate receiving that money.

Similarly, for the payee Bob, he must be assured that:

1. He will receive a proof from Alice, so that she wouldn't be able to repudiate ordering and paying for the service he provides.
2. He will also receive a proof from Alice after he has delivered the service she ordered, so that she won't be able to repudiate having received the service.
3. He must have a way to assure himself that money Alice paid him is not counterfeited, neither spent already. This issue is known as double-spending, which will be discussed in details later.

For the benefits of the bank, at least, it must be assured that:

1. It has such proof that its customers wouldn't be able to repudiate their withdrawal of particular amount, nor would they be able to claim a non-exiting deposit.
2. It has a way to verify the genuineness of the money deposited by its customers, and it must have a way to detect or eliminate double-spending.

Since security issues of multiple entities should be covered in a digital cash system, the term multi-party security is used to refer to these issues entirely. Cryptographic techniques are most often used to offer multi-party security.

- **Anonymity**

Anonymity generally means the inability to determine an individual's sources of income or spending patterns. In other words, it involves a couple of aspects, untraceability and unlinkability.

  - **Untraceability**

    Untraceability refers to the inability for a bank to match an individual's withdrawals of digital cash with his subsequent payments. To gain untraceability, the information a person reveals about himself by making payments must be statistically independent of the information a person reveals about himself while making withdrawals. [DC82]

  - **Unlinkability**

    Unlinkability refers to the inability of a bank (even colluding with merchants) to determine that two payments were made by the same person.

- **Scalability**

Scalability of a digital cash system refers to its ability to be expanded or contracted as needed to service more or less customers over time.

- **On-line vs. Off-line**

In an on-line payment system, a third party (usually the bank) must be contacted in order for a payment between a payer and a payee to be successfully proceeded. For example, in *DigiCash*'s *e-Cash*, after receiving digital cash payment from Alice, Bob must contact the *e-Cash* bank asking for aids of double-spending detection [eCash]. On-line systems obviously require more communication, while in general, they are considered more secure than off-line systems.

Off-line payments involve no contact with a third party during payment, i.e. the payment transaction involves only the payer and the payee. The obvious problem with off-line payments is that it is difficult to prevent double-spending. Present offline digital cash systems with no exception make use of tamper-resistant devices such as smart cards at the payer end. [CAFE, Mondex, SB95]

- **Transferability**

Transferability is a feature that allows an individual to spend the digital cash just received without having to contact the bank in between.

In a system offering transferable digital cash, the bank will not have any idea of the transactions with the digital cash after it is withdrawn from the bank. For example, if Cyber Banque mints transferable digital coins, Bob can consume a coin paid to him by Alice without contacting the bank. Furthermore, the person paid by Bob can spend the coin without contacting the bank too, and so on. Cyber Banque will be able to trace these transactions only if each consecutive spender of that coin cooperates with it.

In addition, it makes it harder for detecting forged or double-spent digital cash. A double-spent coin can only be discovered when two copies of the same coin have been deposited, and at that time, that coin may have been paid to many people.

- **Divisibility**

A digital cash system offers divisibility if the coins minted are divisible: a coin can be "divided" into smaller coins whose total values are equal to the value of the original coin, and the division can be done off-line. This property allows exact off-line payments to be made without the need to store a supply of coins of different denominations.

2.3.2 Double-spending

Double-spending refers to fraudulently spending the same money more than once. Since digital cash is just a bunch of bits, a piece of digital cash is very easy to duplicate. On-line systems and off-line systems deploy different mechanisms to detect or prevent double-spending.

In an on-line system, the bank typically maintains a database of records of digital cash that has been spent, and hence does not authorize transactions involving previously spent cash.

In some off-line systems, a similar database is maintained in a tamper-resistant chip in the user's smart card. The chip keeps records of all the pieces of digital cash spent by that smart card, and hence will detect an attempt of double spending and refuse to authorize the transaction. Since the chip is tamper-resistant, the owner cannot modify the database without permanently damaging the smart card.

In other off-line systems, some cryptographic mechanism is designed to reveal the identity of the double spender at the time the piece of double-spent digital cash makes it back to the bank. One way of doing this is that, before accepting a payment, the payee

will issue an unpredictable challenge to which the payer's equipment must respond with some information. By itself, this information reveals nothing about the payer. However, if the payer spends the same money a second time, the information yielded by the next challenge gives away his identity (or his secret key) when the cash is ultimately deposited.

The presented new digital cash system in this thesis is an on-line system and its effort on dealing with double-spending is focused on how to detect it. There is no mechanism developed to reveal a double-spender's identity in this thesis.

2.3.3 Digital Cash Proposals

NetCash by University of Southern California, *Mondex*, *eCash* and *CAFE* are among the most well known digital cash systems. Recently, Stefan Brands also proposed his digital cash system [SB95], which is being incorporated into *eCash* system. In this section, *CAFE* and *Mondex* are described briefly; both are off-line systems with additional hardware. In the next chapter two on-line systems, *eCash* and *NetCash*, are discussed in details because of their critical contributions to the presented system in this thesis.

- ***CAFE***

CAFE (Conditional Access For Europe) is aimed to be used as a pan-European cash-based mechanism for digital payments [CAFE]. The system makes use of public-key cryptography techniques and tamper-resistant hardware. It can be regarded as the off-line version of *eCash*.

The tamper-resistant device is an electronic wallet containing a "guardian", a smartcard with a dedicated cryptographic processor. The wallet protects the interests of the user, and the guardian protects the interests of the money issuer. No transactions are possible without the co-operation of the guardian, who maintains a record of all the coins spent to prevent double-spending. In addition, the guardian endorses each payment by giving it a

digital signature. The identity of the payer is encoded into the serial number of each coin he spent in such a way that the identity can and only can be recovered if the coin is spent more than once.

A user saves coins to his electronic wallet by withdrawing them from a coin issuer. When the user spends money, the device transfers some coins to the payee's device and mark them spent locally within the payer's device. The payee cannot spend these coins until they are deposited to a coin issuer. Thus, *CAFE* doesn't offer transferability.

- ***Mondex***

Mondex card is an electronic wallet on which *Mondex* cash can be transferred from a bank account by using specially equipped telephones. [Mondex]

When Alice wants to make a purchase at Bob's store, Bob uses a *Mondex* card reader to withdraw the purchase amount from the card and transfer it to a terminal in the store. Bob can then send his receipts directly to his bank account by telephone. Bob will not know Alice's identity. However, during the payment Bob's card reader will pull off a 16-digit identifying number and transmit it to the bank. Hence the use of the *Mondex* cash can be traced to Alice by the bank, and all her transactions can be linked.

As a feature, *Mondex* is designed to allow off-line transferability: two *Mondex* card holders can transfer cash between their cards.

Chapter 3 eCash and NetCash

In Chapter 2, three categories of digital payment systems, secure credit cards system, credit-debit system, and digital cash system are briefly discussed. In this chapter, two digital cash systems, *eCash* and *NetCash*, are discussed in more details. Both systems provide mechanisms to offer anonymity, which is the most researched topic within digital cash systems. The solution of *eCash* is blind signatures, while the mechanism of *NetCash* is "money exchange". Although anonymity offered by *NetCash* is not unconditional, *NetCash* has some advantages over *eCash* such as scalability and fair anonymity.

Most of the current works on digital cash systems are around security, anonymity and double-spending issues. Therefore, the discussion of the two systems is focused on theses issues as well as others such as scalability.

Note a fact that digital cash must be identified in some way, and usually it is a serial number that is applied. Bearing identification, digital cash will be able to be traced. Generally, to cut the link between the digital cash and people who owns them or spends them, identity of the digital cash must be changed either when they are withdrawn from the banks or before they are spent. [HB89] This chapter shows how *eCash* and *NetCash* cut the link respectively in this chapter.

3.1 eCash

eCash is designed by *DigiCash* who claims that the payment solution offers a secure, low cost and private payment option to consumers for payments of any amount, and the systems were specifically designed to be privacy protecting for consumers... [eCash]. Anonymity in *eCash* is achieved based on David Chaum's Blind Digital Signature algorithm [DC82].

3.1.1 Blind Signature

It is first realized by Chaum [DC82], that the most satisfactory way to ensure anonymity of payments is to destroy the relation between the information that the bank sees when it certifies a message, and the information that is transferred to the service provider in the corresponding payment protocol.

According to Chaum's blind signature scheme, to get the digital cash, Alice creates the serial number locally and submits it to Cyber Banque for signing. The number should be randomly chosen to provide uniqueness and should be encrypted in such a way that the bank won't be able to see it while Alice can decrypt it after it being signed. The mechanism of blinding the serial number will not affect the bank correctly signing the coin and validating its signature in the future when it is spent. In addition, distinctiveness of different coin values is achieved by using different signing keys by the bank.

Because the bank doesn't see the serial number of a particular coin, the link between the coin and its owner is in fact cut from the immediate start of the coin's circulating.

In the following sections, we will first give the cryptography theory for blind signature, and then discuss its mechanisms following the circulation of an *eCash* coin.

3.1.2 Cryptography Basis for Blind Signature

Blind Signature makes use of public-key cryptographic systems such as *RSA*. As before, if we denote encryption and decryption of a message M as:

$$E(M) \text{ and } D(M)$$

RSA has a property that both encryption and decryption are commutative. That is:

$$D(E(M)) = E(D(M)) = M$$

RSA also has the property of multiplicative homomorphism, that is, for any messages M_1, and M_2:

$$E(M_1 \cdot M_2) = E(M_1) \cdot E(M_2)$$
$$D(M_1 \cdot M_2) = D(M_1) \cdot D(M_2)$$

If Alice wishes a signature of Bob on message M, she first generates a random number r and encrypts it with Bob's public key, then gets M multiplied by the encrypted r. The resulting product, $M \cdot E(r)$, is submitted to Bob for a signature, and he has no clues about what the original message M looks like. Bob signs the product using the decryption algorithm with his private key:

$$D(M \cdot E(r)) = D(M) \cdot D(E(r)) = D(M) \cdot r$$

The result $D(M) \cdot r$ is sent back to Alice who can remove r from the product by multiplying it with $1/r$ which yields the signed message $D(M)$[1].

3.1.3 Withdrawal of eCash coins

When Alice requests, for example, a ten-dollar *eCash* coin from Cyber Banque, Alice's *eCash* client software will:

1. Generate a random serial number w for the coin. This w should be long enough, (Chaum suggests 100 digits), so that if used with a good random number generator, it will be guaranteed with high probability that two coins will not have the same serial number.
2. Choose a random blinding factor r.

[1] Refer to section 1.2.3.4 for a definition of 1/r.

3. Calculates the value:

$$w' = w \cdot r^{e} \bmod n$$

where e is the bank's "worth $10" *RSA* public exponent, and n is the *RSA* public modulus of the bank.

4. Sign w' with the Alice's private key, then
5. Encrypt it with the Cyber Banque's public key, and then
6. Send to Cyber Banque for signing.

Upon receiving the withdrawal request, Cyber Banque will:

1. Decrypt the message with its private key.
2. Check the signature of Alice.
3. Take out of her account the amount in the form of a digital coin (a ten-dollar coin in our scenario).
4. Sign the coin with its own private key by calculating like:

$$s' = (w')^{1/e} \bmod n = (r \cdot w^{1/e}) \bmod n$$

where $1/e$ acts as the private key for coins of ten-dollar denomination, (refer to Chapter 1 for details on *RSA*.)

5. Encrypt s' with Alice's public key.
6. Send it back to Alice.

With the bank-signed coin, Alice's software will:

1. Decrypt it with her private key.

2. Un-blind it by doing the following computation[2]:

$$s = s' \cdot 1/r \bmod n = w^{1/e} \bmod n$$

Note that r is only known to Alice. Now Alice got an anonymous $10 *eCash* coin signed by Cyber Banque, which bears two parts denoted as:

$$(w, s)$$

It is a genuine $10 coin because it is signed with the bank's 10-dollar private key $1/e$. It is anonymous because when Cyber Banque signed it, it could only see the product with r, and r is a secret of Alice. So the bank won't know which coin was issued, neither does it have the ability to associate Alice's withdrawal with the future circulation, including payment and deposit, of the same coin.

3.1.4 Payments with eCash Coins

It is quite easy for Alice to spend her *eCash* coins. Suppose she is paying Bob ten dollars with *eCash* coins. What she needs to do is as simple as:

1. Combine purchase information, such as type of service requested, amounts requested, together with *eCash* coins of the required value into a message package.
2. Optionally sign it with her private key if Bob doesn't accept anonymous purchase.
3. Encrypt the package with Bob's public key.
4. Send the message to Bob.
5. Receive the service or a receipt from Bob.

---

[2] $1/r$ is the multiplicative inverse of r modulo n, which means $r \cdot 1/r - 1$ is divisible by n.

For Bob, it is a little more complicated. When Bob receives a ten-dollar *eCash* coin from Alice, he can verify the value of the coin with Cyber Banque's public key for ten-dollar coins. However, Bob cannot convince himself the coin is not spent before without contacting the bank for verification. Here the issue of double-spending should be addressed.

The *eCash* way for preventing double-spending relies on a database. At the side of the bank, a database of all coins ever spent is maintained. The records of the database should include at least the serial numbers of the spent coins. Requested by Bob for double-spending detection, Cyber Banque can check against the database for the serial number of the coin Alice paid. Different results will trigger different actions:

- If there is a match, the bank knows that the coin was spent before and should notify Bob with an alarm; Bob, consequently can refuse accepting Alice's payment and refuse delivering the service. It is worth to note that the bank doesn't know exactly about whether Alice or Bob is cheating; however this unawareness does no harm to the bank.

- If there isn't a match, it means the coin is never spent before, the bank can notify Bob of the validity of the coin, meanwhile add the serial number to the database. Bob, consequently, is assured that the coin is valid and will make the delivery to Alice.

3.1.5 Deposit of eCash Coins

In the scheme of *eCash*, whenever a payee receives *eCash* coins, in addition to contact the bank for detecting double-spent coins, he must also deposit the coins if they pass the validation. Therefore, an *eCash* coin is not transferable. Moreover, because the coin is deposited to a non-anonymous account, the payee must reveal his identity to the bank.

3.1.6 Discussion

There is a prominent disadvantage of efficiency and scalability due to *eCash*'s scheme for double-spending detection. The database of spent coins will accumulate over time; therefore the average time per double-spending detection will increase in positive proportion to the service time of the bank. Therefore, the database will finally become the bottleneck of the whole system, and it would limit the scale of the system.

Why doesn't *eCash* maintain a database of all coins in circulation for preventing double-spending purpose, (the mechanism deployed by *NetCash*, which will be discussed later)? This is due to the blind signature mechanism. Since it is not the bank, but each customer, who gives coins identities (serial numbers), and since these numbers are hidden from the bank, the bank is not be able to know the numbers of coins in circulation.

Another reason for the bad performance in scalability is that the database is enforced with a one-record-per-coin policy. Since the value of a coin can be as small as one cent and as big as one hundred dollar, it is imaginable how quick the database would grow if a great part of the transactions take place with values like $999.99.

Another shortcoming of *eCash* is its unfairness with regarding to anonymity. In our scenario, Alice is granted unconditional anonymity, while Bob isn't. This can be overcome with a feature of *NetCash*. We will address it in next section.

3.2 NetCash

NetCash is a framework for electronic transactions that combines the benefits of anonymous transactions with the scalability of non-anonymous online payment protocols. It is proposed from the Information Science Institute at the University of Southern California [NetCash]. Unlike *eCash*, it is an academic research project only, and hasn't yet been applied for practical purposes.

3.2.1 Withdrawal of NetCash Coins

With *NetCash*, there is no blind signature in the picture. Unlike *eCash*, it's the bank that determines which serial number is assigned to each digital coin withdrawn by its clients. This number is recorded at the bank as it is issued, and will not be deleted until the coin is deposited or exchanged (refer to 3.2.2). Each digital coin is signed with the bank's private key.

3.2.2 Money-Exchange

Money-exchange (or cash-exchange) is another mechanism *NetCash* provides. That is, any customer of *NetCash* can exchange an old coin for a new coin. The customer generates a new symmetric secret key for the session and sends this key and other transaction information along with the coin to the bank. The whole message is encrypted with the bank's public key to keep it secret.

Requested for coin exchange, the bank first verifies its signature on the old coin to detect forged coin. Then the bank checks the coin against the maintained database of coins in circulation. If there is a match found, the coin is not spent before. The bank now can create a new digital coin bearing a new serial number, and update the database by replacing the record of the old coin with a new record for the new coin. Finally, the bank digitally signs the new coin and returns it to the customer, encrypted with the session key.

It is easy to see that the maintained database is relatively static and should be much smaller than that an *eCash* system maintains. Thus, the efficiency of coin verification will be improved and scalability of the system won't be a great obstacle.

3.2.3 Payments with NetCash Coins

The payer sends the payee some digital coins, together with the identifier of the purchased service, a freshly created secret key and other information, all encrypted with

the payee's public key. The secret key will be employed later during the session by the payer and the payee to encrypt and decrypt their transaction information.

Receiving coins from the payer, the payee first contacts the bank and asks for double-spending detection. This is done in the same way as shown in coin-exchange. If the coins pass the detection, the payee may deposit them or request for a coin-exchange right away in order to claim the ownership of the coins. Otherwise, the payer would be able to spend the coins again before the payee can spend it; even worse, if another payee deposits or exchanges the coins double spent by the fraudulent payer, there is no way for the first payee to claim his ownership of those coins. If the payee has been properly paid by the payer, he returns the payer a receipt signed with his private key and encrypted with the session secret key.

The buyer can then use the transaction identifier and the session key to obtain the service purchased.

3.2.4 Deposit of NetCash Coins

Receiving a request for depositing a *NetCash* coin, the bank first detects whether it was spent before, and, if not, deletes the coin from the database it maintains. Finally, the bank credits the user's account by the amount of the coin. The communication between the customer and the bank is encrypted to protect the privacy and the transaction is digitally signed for authentication and non-repudiation.

3.2.5 Discussion

It should be noted that scalability of *NetCash* is gained by loss of anonymity. The bank now has a way to record to whom it has issued a particular digital coin, and in the future, when some merchant deposits that particular coin, the bank will be able to link it with the previous withdrawal.

Fortunately, *NetCash* provides some conditional anonymity for its users. This is also achieved by the coin-exchange mechanism. As long as the bank can verify the old coin is valid, (neither forged nor double-spent), it is not necessary to verify the origin of the exchange request. This means an exchange request won't require any customer identification. Hence, anonymity is achieved.

However this kind of anonymity is quite weak and the preserved anonymity depends on such elements as:

- the bank does not keep records pairing the coins accepted for exchange with those newly issued;

- the payee will choose first exchanging the coins received from the payer rather than depositing them to the bank directly.

Consider this scenario: Alice withdraws one ten-dollar *NetCash* coin from Cyber Banque. Before issuing the coin to Alice, the bank may connect the coin's serial number, say *N1*, with the identification of Alice, noted as:

N1 ←→ Alice

Some time later, Alice spends the coin *N1* for a service from Bob. To claim the ownership of the coin, Bob will deposit or exchange it. Suppose Bob chooses to deposit it rather than exchange it, the bank now gets a connection like

Alice ←→ N1 ←→ Bob

Thus, the bank detects that Alice consumed ten dollars for some service from the merchant Bob.

Alice has a choice to maintain her purchase anonymity by herself. Before spending the coin *N1*, Alice exchanges the coin for a new one that bears a new serial number *N2*. Since both Alice herself and anyone else who is paid with the coin *N1* can make the exchange, the bank doesn't know exactly who has made such an exchange, thus it won't be able to connect *N2* with Alice now. However, it can be proved that anonymity achieved this way is conditional. Consider the worst situation if all merchants who can receive *NetCash* coins collude with the bank in such a way that none of them will exchange coins before deposition, the bank will absolutely know it is Alice who made the exchange if there ever occurred an exchange request for coin *N1*. Consequently, the bank can update the connection between the coin and Alice:

N1 ←→ N2 ←→ Alice

The proceeding story of Alice is the same as before: her purchase will be linked with her previous withdrawal. Noted that the same thing would happen to the payee Bob if Alice colludes with the bank. That means, if Alice likes, she can inform the bank which coin she has paid to Bob. The bank then will have the ability to trace that particular coin and, with the same assumption as in the case of Alice discussed previously, link Bob's purchase in the future even he exchanges the coin before spending it.

Of course, if every payer and payee involved is honest, anonymity of purchase and incomes can be achieved against the bank, and this anonymity is fair to both payers and payees, (this is different from *eCash*). However, in practice the only entity one can trust is he himself, so unconditional anonymity can only be achieved by each people himself. By now, the only solution is still blind signature.

In *NetCash*'s digital cash mechanism, although it doesn't offer transferability for digital cash, by deploying the coin exchange mechanism, it gives a sense of transferability: the payee, although still has to contact the bank on-line for exchange, doesn't have to deposit the coin before he can spend it. This is another advantage *NetCash* over *eCash*.

3.3 Conclusion

First of all, a shortcoming must be indicated in the implemented anonymity of both systems: the network address of a client is exposed to the bank. This gives the bank a clue to relate the transactions from a particular network address with the particular client. One solution is that the client may try not to stick to a network address for many transactions.

Table 1 compares the performance of *eCash* and *NetCash* and it suggests that blind signatures and money-exchange combined together may give a system offering both anonymity and scalability. This is the topic in Chapter 4.

		ECash	*NetCash*
Anonymity	Untraceability	▪ Unconditional ▪ Payers only	▪ Conditional ▪ Fair to Payers and Payees
	Unlinkability	Not Offered[3]	Not Offered[4]
Scalability		Bad	Good
Others	Multi-party Security	Offered	Offered
	Off-line or On-line	On-line	On-line
	Transferability	Not Offered	Not Offered[5]
	Divisibility	Not Offered	Not Offered

Table 1 - Comparing *eCash* and *NetCash*

[3,4] Unlinkability can be achieved by deploying anonymous accounts as shown in [HB89].
[5] *NetCash* makes its coins seem transferable, but users have to be on-line in order to make the value of a coin "transferable".

Chapter 4 Combination of *eCash* and *NetCash*

In this chapter, a new digital cash proposal is presented. The presentation has been divided into two sections. In the first section, it will show that the presented proposal is actually the combination of mechanisms from *eCash* and *NetCash*, and how multi-party security, unconditional anonymity and enhanced scalability are all achieved by this combination. The second section introduces possible mechanisms to extend the system.

4.1 Combine eCash and NetCash

It will be shown that, in the proposed system, not all digital cash services require an entity to have opened an account in the bank in order to be qualified for the services. However, let us start our discussion assuming that each customer has a non-anonymous account in a common bank that offers digital cash services.

4.1.1 Digital Coin and Digital Note

Unlike other systems, there are two forms of digital cash involved, **digital coin** and **digital note**. Each item of cash, no matter it is a coin or a note, must bear a serial number as its identification. Practically, serial numbers of digital notes and coins don't have to share a common name space although it is not a problem if they do. For instance, all digital notes may have serial numbers starting with a "N", while all digital coins may have serial numbers starting with a "C". Therefore, N123456789 may be the serial number of a note, while C123456789 may be the serial number of a coin.

The most distinctive difference between a digital coin and a digital note is analogous to that between a metal coin and a paper note: usually a metal coin bears lower value while a paper note bears relatively higher value. Similarly, a digital coin usually bears relatively

smaller value than a digital note. (For example, Cyber Banque may mint digital coins of such denominations as 1-cent, 5-cent, 10-cent, 25-cents, 1-dollar and 2-dollar; meanwhile, it may issue digital notes of such denominations as 5-dollar, 10-dollar, 20-dollar, 100-dollar and 1000-dollar). However, in the system there is no mechanism that prevents the bank from issuing a digital coin with a denomination the same as or higher than that of a digital note. In other words, the bank can issue, for example, 500-dollar digital coins even if the highest denomination of digital notes it ever issues is, for example, 100 dollars. No matter what choices the bank makes on the denominations of notes and coins, a note of any denomination must be able to be changed to a set of coins whose total value is the same as the note's denomination. Such a prerequisite is necessary to make possible the features of the presented system. For this reason, we assume the Cyber Banque doesn't mint digital coins with a higher denomination than that of digital notes it issues.

It is another very important feature in the system that a user can only withdraw digital notes. Withdrawals of digital coins are not supported. This regulation is reasonable in real life since most people are inclined to withdraw money of relatively large denomination and then spend it little by little. If the customer wants to get digital coins, he must ask the bank to exchange the notes for coins, (the exchange mechanism will be discussed later). In addition, it will be shown that this withdrawal restriction contributes to the enhanced scalability of the system.

There is no distinction between coins and notes with respect to payment abilities however. That means a customer can pay a service provider with notes, coins or both for a single payment; on the other hand, the service provider can accept notes, coins or both.

4.1.2 Entities and Protocols

This proposal involves three entities: a consumer Alice, a service provider Bob, (both Alice and Bob are customers of the bank), and a bank Cyber Banque, and four protocols: withdraw, exchange, payment and deposit. (Refer to Figure 6.)

Unlike *NetCash*, the exchange protocol can be deployed either from a digital note to digital coins or from digital coins to digital coins.

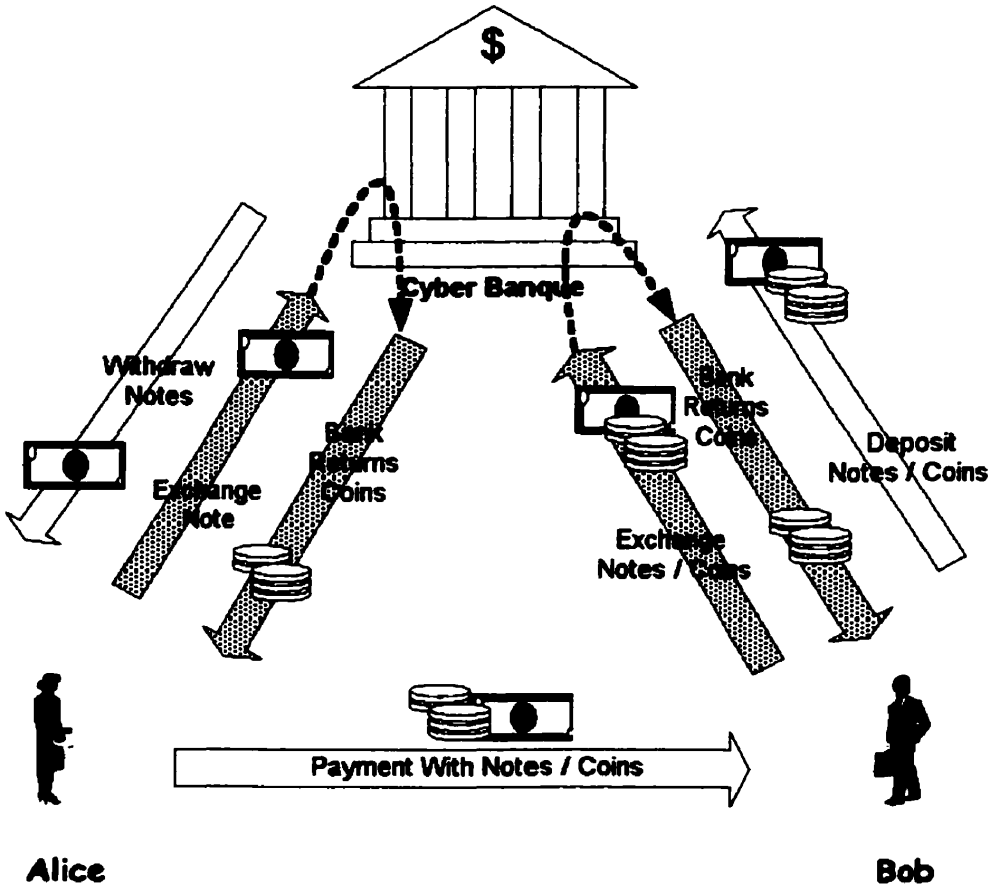

Figure 6 - Entities and Protocols in the new digital cash system.

4.1.3 Server Side Mechanisms for Double-Spending Detection

The bank has similar mechanism as that of *eCash* and *NetCash* for detecting double-spending. Like *eCash* and *NetCash*, the presented system is also an on-line system: a client must contact the bank for security reasons such as double-spending detection. However, unlike *eCash* and *NetCash*, there are two rather than one database maintained by the bank. One database maintains all digital coins in current circulation (like *NetCash*), which we denominate as DB_{coins}, while a second database maintains digital

notes exchanged or deposited already, which we denominate as DB_{notes}. Practically, we may have serial numbers of those coins in circulation maintained in DB_{coins}, and serial numbers of those notes exchanged or deposited maintained in DB_{notes}. (See Figure 7.) We will talk about the mechanisms of both databases in more details later.

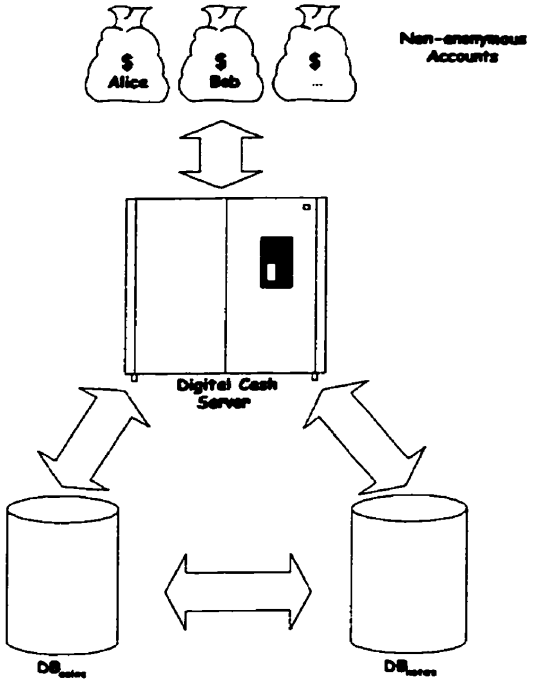

Figure 7 - Server-side databases for detecting double-spending

4.1.4 Digital Cash Withdrawal

Recall that in the presented proposal, only digital notes can be withdrawn, while digital coins cannot. It must be stated that to maintain anonymity for its customers, Cyber Banque must limit the types of digital notes denominations it ever mints to a small number, and these denominations shouldn't change regularly. (The reason will be shown later, now we just assume it). This doesn't mean the total cash amount per withdrawal is limited: as long as Alice has enough balance in her account, she is not prohibited from withdrawing an amount of money her balance permits.

Suppose Cyber Banque mints 100-dollar digital notes. If Alice is withdrawing 100 dollars the withdrawal protocol is initialized by Alice who:

1) Randomly generates a serial number *N* for the note. As *eCash*, the bits of the number should be long enough so that it is not possible for any people (including Alice herself) to generate the same number again.
2) Blinds *N*. The same mechanism as that of *eCash* is deployed for blinding a message. For each different denomination, a distinct public key is known to Cyber Banque's customers for blinding withdrawal request messages.
3) Digitally signs the blinded message with her private key. In this proposal, we make use of an asymmetric digital signature algorithm such as *RSA* combined with a one-way hash function such as *MD5*, to sign a message.
4) Encrypts the message with the public key of Cyber Banque, and then sends the encrypted message to the bank.

Receiving the withdrawal request, Cyber Banque:

1) Decrypts it with its private key.
2) Checks Alice's signature against her public key.
3) If verification fails, it notifies Alice (encrypted and signed.) If verification succeeds, the bank then checks Alice's account to see if there is enough balance for the withdrawal. If not, it notifies Alice (encrypted and signed); otherwise, the bank debits 100 dollars from Alice's account, signs Alice's original message with its 100-dollar private key, encrypts it with Alice's public key, then sends it back to Alice.

Receiving the message, Alice:

1) Decrypts it with her private key.
2) Un-blinds it.
3) Checks the validity of the note with the 100-dollar public key.

4) Keeps the note in her "digital purse".

Note that nobody other than Alice has the knowledge of what serial number the note bears, thus a note withdrawn this way is protected from being traced in its future circulation.

4.1.5 Digital Cash Exchange

There are two kinds of exchanges supported: from digital notes to digital coins and from digital coins to digital coins. At any time, any party, no matter whether or not he has an account in Cyber Banque, as long as he has digital cash issued by Cyber Banque, he is able to perform cash exchanges. This point is quite different from withdrawal and deposit.

Like *NetCash*, cash exchange is client anonymous. In other words, if Alice wants to exchange her money, she doesn't have to disclose her identification to the bank.

4.1.5.1 Exchange Notes for Coins

It is necessary for Alice to make this kind of exchange if she is intended to pay Bob, for example, 2 dollars, while in her "digital purse" she only has a 100-dollar digital note. (Although we will extend the scheme to make the digital money divisible, we will not cover this issue in this section). She can request Cyber Banque for exchanging her 100-dollar note to get a 2-dollar coin.

To request for such an exchange, Alice will:

1) For this exchange transaction, generate a new secret key of a symmetric encryption algorithm (such as *DES*).

2) Take the note out of her "digital purse", combine it with the transaction secret key to make up a message, encrypt the message with Cyber Banque's public key, and send the message to the bank.

Upon receiving this exchange requirement, the bank will:

1) Decrypt it with its private key to get the note and the transaction key.
2) Verify its signature on the note. If the verification fails, the note is not genuine and the bank will refuse to proceed with the exchange.
3) Otherwise, check the validity of the note against the database DB_{notes}.
4) If there is a match, it means the note is exchanged or deposited already, in other words, there is an attempt to double exchange or deposit the same note. Cyber Banque may refuse the request.
5) If there isn't a match, Cyber Banque can be assured that this is the first time the note is being exchanged. The bank will add the serial number of the note to DB_{notes}, generate some digital coins whose total values are equal to that of the digital note, add the serial numbers of these coin to DB_{coins}. The information about what coin denominations Alice would like to get can be included in the withdrawal request message as an optional feature.
6) Cyber Banque signs these coins digitally, gets them encrypted with the secret transaction key provided in Alice's original message, then sends back to Alice.

Receiving these coins, Alice can decrypt (with the secret key) and verify them just as she did before.

Note that since the bank doesn't know who possesses a particular digital note, and the exchange processing is anonymous, it won't be able to know who will possess the returned digital coins.

4.1.5.2 Exchange Coins for Coins

Requested by Alice (or Bob) for exchanging a digital coin for a new coin, (as the previous case of exchanging a note for coins, the customer will provide a new symmetric secret key for this exchange transaction), Cyber Banque will:

1) Verify its signature on the coin. If the verification fails, the coin is not genuine, and the bank may refuse the request.
2) Check its validity against the database DB_{coins}. If there isn't a match, the coin has already been exchanged or deposited. If it was the payer Alice who made the exchange request, then Alice is attempting to exchange a coin multiple times. If it was the payee Bob who made the exchange request, then Bob is cheated by Alice with a coin spent before, (we will talk about this later). However, the bank cannot distinguish between these two cases because it doesn't know who made the request. Anyway, the bank may notify the requester that the coin is not in circulation and refuse the request. If there is a match, the bank can now generate a new coin bearing a new serial number which has never been used by any coin before, replace the old serial number with the new one in the database DB_{coins}, and send the new coin back to the requester after encrypting it with the secret transaction key. It is easy to see that the mechanism here is the same as that of *NetCash*. It will be discussed later that this kind of exchange will offer a payee the ability to "launder" the money he was paid, so that he can spend it anonymously.

4.1.6 Digital Cash Payment

To make the discussion aimed at the essence of the system, now we assume that Alice has the exact amount of digital cash (notes, coins or combination of both) that Bob charges her. In other words, Alice won't have to get her money changed to be able to pay Bob the exact amount of charge. To pay Bob with digital coins, Alice will:

1) For this payment transaction, generate a new secret key of a symmetric encryption algorithm (such as *DES*).
2) Take some digital cash out of her "digital purse", use Bob's public key to encrypt the package including the cash, the transaction secret key and other purchase information, and send the message to Bob.

Receiving the message, Bob will:

1) Decrypt the message with his private key, and draw out the digital cash from the message.
2) Verify Cyber Banque's signature on each of the cash with the bank's public key associated with the denomination of each of the cash.
3) If verification fails, return the cash to Alice and notify her (encrypted with the transaction key sent in Alice's message and digitally signed with his own private key).
4) If verification succeeds, Bob has the following options to deal with the money:
   - Deposit the cash to the bank.
   - Request the bank to exchange the coins for new coins, and exchange notes (if there are any) for coins.

   Bob must undertake one of the actions above, because if he doesn't, he actually doesn't claim his ownership of the money, i.e., Alice will have the ability to spend them again or deposit them without being caught.

   No matter which option Bob takes, (we will talk about the bank's role in a deposit protocol later), the bank will first undertake a process to detect double exchanged or double deposited digital cash. If there is such notes or coins detected, the notification from the bank may cause Bob to refuse Alice's payment and return her digital cash with a notification of the reason (again encrypted with the transaction secret key and digitally signed.)

5) If the coins pass the tests of step 4, Bob can deliver the service or a receipt of the payment to Alice (encrypted with the transaction secret key and digitally signed with Bob's private key.)

Note that in this payment protocol, Alice's purchase is maintained anonymous. Because Alice generates a new secret key for each payment, Bob has no more information about her other than her network address. The bank cannot figure out Alice's identity either, since it cannot connect the digital cash Alice paid Bob with her identity.

4.1.7 Digital Cash Deposit

No matter which kind of digital cash Alice has, notes or coins, the protocol for depositing them to her account in the bank is almost the same. Note that in either case, she must disclose her identification to the bank.

To deposit digital cash, Alice signs the deposit request with her private key, encrypts the message with Cyber Banque's public key, and then sends it to the bank.

Receiving the message, the bank will:

1) Decrypt the message with its public key.
2) Verify Alice's digital signature on the message.
3) Check its own signature on the cash.
4) If the cash is a digital note, detect if it was exchanged or deposited before by checking it against the database DB_{notes}.
5) If the cash is a digital coin, detect if it was spent or deposited before by checking it against the database DB_{coins}. (Refer to the section about the exchange protocol).
6) If the cash passes the checks, credit Alice's account with the amount of the cash, then update either database. If the deposited cash is coins, delete their records from DB_{coins}, otherwise add records of the notes to DB_{notes}.

7) Notify Alice with a receipt of deposit (signed digitally and encrypted with Alice's public key.)

4.1.8 Discussion

The proposal until now is constructed by combining features of *eCash* and *NetCash*. The advantage of *eCash* is its unconditional anonymity for payers, but due to its deployment of blind signatures, its scalability and efficiency is limited. Moreover, *eCash* doesn't offer payees any anonymity. *NetCash*, on the contrary, is scalable and efficient, and it offers fair anonymity to both the payer and the payee. The disadvantage of *NetCash* is the anonymity it offers is not unconditional.

The presented proposal offers multi-party security and unconditional anonymity to both the payer and the payee, and its scalability and efficiency is improved compared to that of *eCash*.

4.1.8.1 Multi-party security

It is common to the three entities (payers, payees and a bank) that:

- Communication between any two entities is encrypted so that the information won't disclose to anyone else other than entities involved.
- Each digital note and coin is digitally signed with the bank's private key, which makes fraudulent counterfeiting computationally infeasible.

Requests for withdrawal and deposit are all digitally signed by the customers. This gives the bank an un-deniable proof to protect it against repudiation from the customers.

Because of those two databases (DB_{coins} and DB_{notes}) deployed at the bank, the bank and payees are protected against payers double spending a coin, double exchanging a note and double depositing any digital cash.

Requiring payees to return payers signed receipts after the payees receive the payment, the system protects payers from repudiation of received payment by payees.

4.1.8.2 Unconditional Anonymity

For the payer Alice, the presented system offers her unconditional anonymity. In the withdrawal protocol, the serial number on the digital note is concealed from the bank; in the exchange protocol, the identity of the customer is not disclosed to the bank; in the payment protocol, the identity of the payer is not disclosed to neither the payee nor the bank. Thus, the purchase by the payer with the exchanged digital coins is not traceable by the bank even if the bank colludes with the payee.

It is not a good idea not to limit the types of the denomination of cash (both notes and coins) the bank can issue. For the bank, if it offers the customer an option to determine arbitrary denomination of the cash, such as 14.99-dollar, it makes the bank impossible to assign each denomination with a unique asymmetric key pair. For customers of the bank, if they could withdraw a note with the denomination, for example, 14.99 dollars, this particular denomination would act as a clue for the bank to link this note with her future exchange and purchase even if she has blinded the serial number of the note.

For the payee Bob, he can also get some degree of anonymity. By exchanging received cash for new coins instead of depositing them directly, Bob can conceal this profit from the others except Alice, and he can spend the new coins without revealing his identity in the future. Note that Bob's identity is revealed to Alice during the procedure of purchase and payment, (this conforms to the pattern of conventional commerce). Alice has the exact idea what serial number each coin bears, so if Alice colludes with the bank, the bank can trace Bob's exchanging of coins and further payment with the new coins.

In the next sections, we will extend the system to offer even Bob unconditional anonymity so that he can spend the payment from Alice un-traceably by anyone else. But

we won't extend it so that the truth that he has got paid by Alice is unconditionally concealed from others, because current policy doesn't encourage it.

It must be pointed out that sometimes the anonymity can be impaired. Suppose at some time, (for example, when the Cyber Banque just began its service), the bank notices that all 100-dollar digital notes it ever issued have been deposited or exchanged to coins, and at that time Alice withdraws a 100-dollar note. And some time later before the bank issues any other 100-dollar notes, it receives a 100-dollar note for exchange. Since the withdrawal exposes the identity of Alice, the bank knows it is Alice who is exchanging that 100-dollar note. Consequently, the bank will be able to trace the circulation of the exchanged coins. It can be imagined that the more notes of 100-dollar are still in the hands of customers, the less possibility is there for the bank to have a chance to impair Alice's anonymity.

4.1.8.3 Enhanced Scalability

Scalability is enhanced comparing with *eCash*. The bank maintains two databases now. The first, DB_{coins}, records all digital coins in circulation, the second, DB_{notes}, records all digital notes exchanged or deposited.

It is easy to see that the size of DB_{coins} will be statistically fluctuating around a relatively static value. Note-to-coins exchange is the only contribution to the increment of DB_{coins}, while coin-deposits decrease the size. Transferring a coin between two customers of the bank doesn't increase the size of DB_{coins}. In other words, the size of DB_{coins} won't increase dramatically as time elapses as soon as the amount of the digital cash issued reaches the minting capacity of the bank.

As to DB_{notes}, it is true that its size will increase as time elapses, hence bring in scalability problems (similar to *eCash*). However, it is worth to be noted that it is a database of digital notes and each note bears the value of many coins in combination. For instance, one 100-dollar digital note can be exchanged for digital coins of forty 2-dollar's, ten one-

dollar's, twenty 25-cent's, forty 10-cent's, ten 5-cent's and fifty 1-cent's. Suppose Alice withdraws 100 dollars respectively from Cyber Banque who provides services presented in this thesis, and an *eCash* bank, she contributes one record to DB_{notes}, and comparatively she might contribute 170 (that is 40+10+20+40+10+50) or even more (if she prefers smaller denominations for the coins) records to the database in the *eCash* server. Thus, the database DB_{notes} will not increase too quickly as time elapses.

The main cause for scalability problem of *eCash* is because the size of its database for detecting double spent cash increases quickly as time elapses. By introducing two separate databases, one maintained a static size statistically and the other decreased its size expansion speed, we overcome the shortcoming of *eCash*, i.e., enhance the scalability of the digital cash system.

4.2 Extending the System

In this section, efforts of extending the proposed system will be presented. Before we continue, let's first find out what problems or disadvantages still exist in the system:

A. Customers are bound to a common financial institute;
B. The database DB_{notes} grows anyway despite its low growing speed;
C. The payees are not unconditionally anonymous when they spend the digital cash received;
D. Digital cash is not divisible.

For A, it is easy to solve by deploying the multiple banks solution in *NetCash*, which will not be covered in this thesis. For B, we will address in subsection 4.2.1, and for C and D, we will cover in subsection 4.2.2.

4.2.1 Enhance scalability

The enhancement is achieved as following. In addition to the serial number, Alice now also provides the bank two other parameters for a digital note: the initial date of validation (*IV*) and the duration of its validation (*DV*). In order not to impair her anonymity, she must blind them too. One practical way to do that is to make *IV* and *DV* contributes some predefined digits as part of the serial number. For example, it might be a rule between the server at the bank and Alice's digital cash software that a serial number of a digital note is made up of three fields:

YYYYMMDD (*IV*)	XX (*DV*)	SN

The first 8-digit long field *IV* holds the initial year (YYYY), month (MM) and day (DD) of the note. The second 2-digit long field *DV* holds the number of days (XX) of the note's validity. SN is the randomly generated serial number which Chaum suggests length of 100 digits.

As an example, suppose Alice is withdrawing a digital note from Cyber Banque, and she needs the note valid from March 6, 1999 and its validity lasts for 10 days, the serial number will look like:

19990306 10 XX XX

14243

100 Randomly Generated Digits

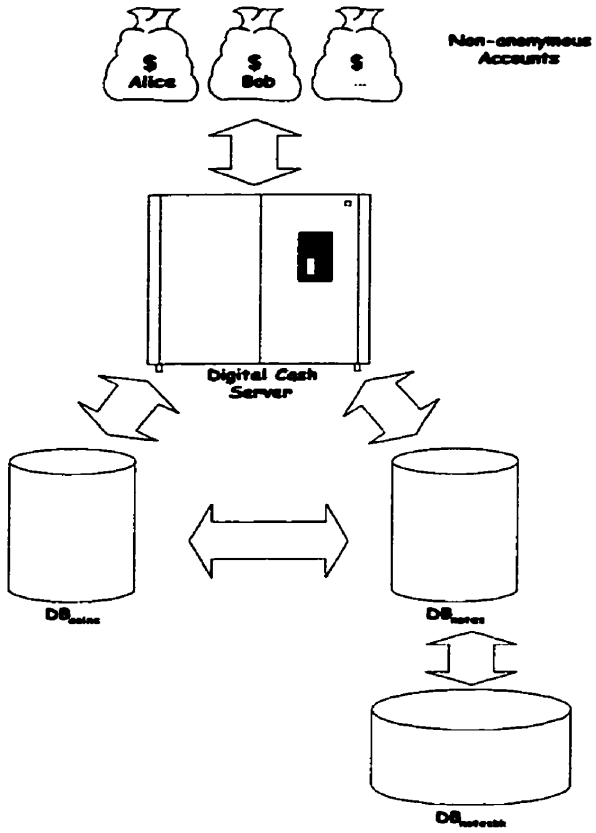

Figure 8 - One new database is added for enhancing the system scalability.

At the side of the bank, in addition to the databases DB_{coins} and DB_{notes}, one more database $DB_{notesbk}$ is deployed. (Refer to figure 8.) This new database keeps records of all notes which are exchanged or deposited and have expired, and the old DB_{notes} now only maintains records of notes that are exchanged or deposited and are still in their validity period. The server software at the bank thus has to do some routine works by moving the records of expired notes from DB_{notes} to $DB_{notesbk}$. The interval of the routine is determined by the minimal time scale deployed in the system to define a note's validity period. It is acceptable to both the bank and its customers to deploy "day" as the minimal scale, since it is the case in most conventional financial transactions.

The process of detecting double exchanged or deposited notes is adapted also. Whenever a request for withdrawal of digital note is received, the bank signs it knowing none of its

three fields above. Whenever a request for note-to-coins exchange or note deposit is received, the bank will:

1. Verify its signature on the note.
2. Get the fields *IV* and *DV* out of the serial number of the note, and find if this note is in its validity period.

   For example, if the date is now March 12, 1999, then Alice's note is in its validity period, the bank will proceed to detect whether it is exchanged or deposited before by checking it against DB_{notes} as before. If the date is before March 6,1999, then the note isn't valid yet. In this case, if Alice is asking for a deposit, the bank will deposit it to her account; if Alice is asking for an exchange, the bank will refuse it and return the note with a notification of the reason. If the date is after March 16, 1999, then the note is expired, the bank will return it with a notification even if the note is not exchanged or deposited before.

   For expired notes, customers can rescue the money by arguing to the bank. This can be done by checking the serial number of the note against $DB_{notesbk}$. If there is a match, then the note is exchanged or deposited before, i.e., the customer is cheating; otherwise, the note is never exchanged or deposited, the bank can rescue the money for the customer. There are a few ways to do that. The bank may return the customer a new generated note carrying a customer generated serial number, or exchange the note for coins and return the customer the coins, or deposit the value to the customer's account. The bank may choose an option following customer's preference. The process can be either anonymous or not depending on the policy of the bank, and the bank can charge a fee for the processing to encourage its customers to exchange a note for coins within its validity period.

Noted that by separating the exchanged or deposited notes into two databases according to whether they are expired or not, the scalability is enhanced even more. Since the bank will punish a customer if he exchanges or deposits an expired note, most of the detection will be checked against DB_{notes}, whose size is statistically static. Thus, the scalability issue in the presented proposal due to the ever-growing DB_{notes} is solved.

As to $DB_{notesbk}$, it can be stored in some secondary storage devices such as tapes since it is not frequently accessed. The bank may maintain a separate database of this kind for, for example, each month to decrease the average accessing time per query. In this way, scalability can be improved even further.

It also should be pointed out that the customer, if she expects anonymity, should not always define IV as the date when she withdraws, since this may give the bank some clues to link the note with her identification. It would also be deployed by the bank that, although not necessary, DV is limited to some bank predefined values, such as, following the previous example, 01 (for one day), 07 (for one week), 30 (for one month) and so on.

4.2.2 Extend Cash Exchange Mechanism

In this subsection, we try to extend the exchange mechanism in the following ways:

A. Make it possible to exchange some coins for a note or an old note for a new note. This will offer the payee unconditional anonymity when he spends the received money.
B. Make it possible to exchange a coin of a larger denomination for coins of smaller denominations. This will give a sense of dividable digital cash.

4.2.2.1 Offer Payees Unconditional Anonymity

We have shown in the previous section that the payee Bob is not offered unconditional anonymity when he spends the digital cash paid by the payer Alice. In order to fix it, we extend the exchange mechanism to make a note exchangeable for a new note (refer to Figure 9), and coins of some particular value exchangeable for a note (refer to Figure 10). To generate the new note, the customer requesting for this service must provide the randomly generated serial number and again must blind it from the bank.

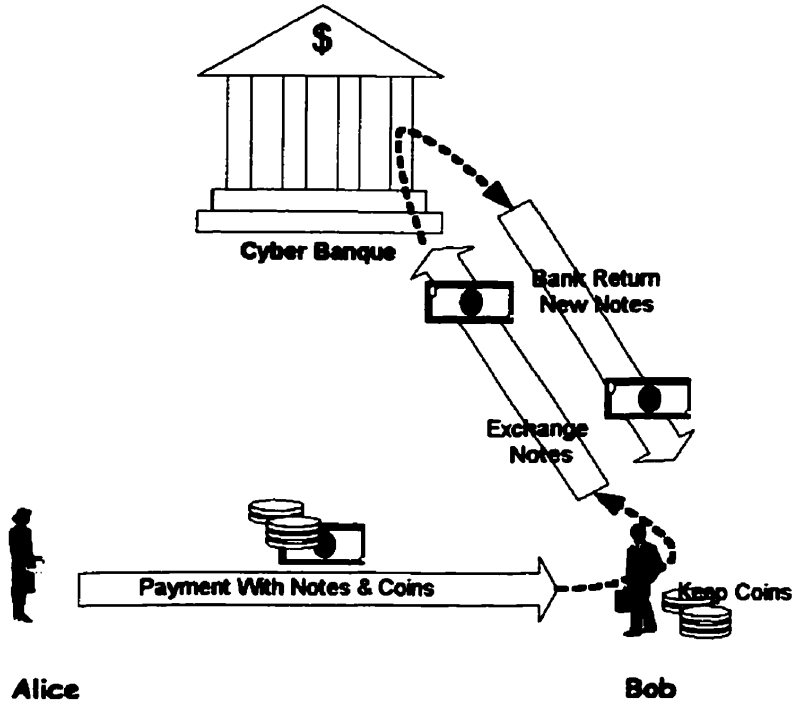

Figure 9 - Exchange a note for a new note.

For example, when Alice pays Bob 100 dollars (a valid 100-dollar digital note) and 10 cents (a valid 10-cent digital coin), Bob can exchange the note for a new note right away. Of course if Bob likes, he can do a second exchange: from the new note to coins. Anyway, Bob's future purchase with the new digital cash is anonymous since the serial number of the new note (and the serial numbers of the exchanged coins) cannot be linked to his identification due to the deployment of blind signature on the new note. As to the 10-cent coin received, Bob can first make a coin-to-coin exchange to claim his ownership

on it then store it in his "digital purse". In the future, when he has already accumulated, for example, 10 dollars of coins, he can request the bank to exchange them together for a 10-dollar digital note. Once again, he must generate the serial number for the note randomly and blind it against the disclosure to the bank.

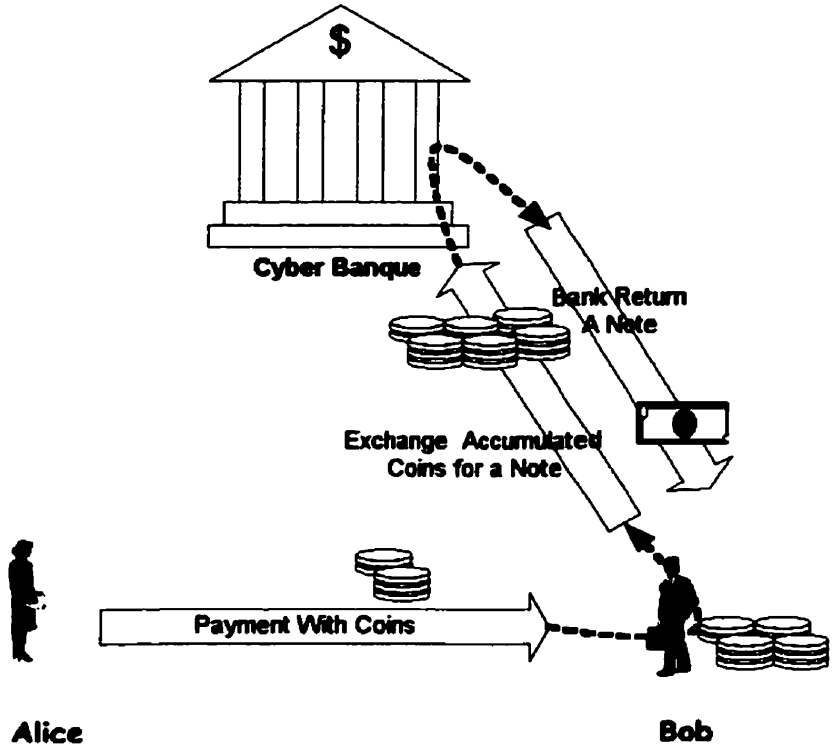

Figure 10 - Exchange coins for a note.

It is worth to note that in the exchange process, Bob must provide the bank with a secret key (such as one in *DES*) for the exchange transaction, in order to give the bank a way to encrypt the transaction while not being able to know his identity.

4.2.2.2 Offer "Divisibility" for Digital Cash

It is easy to extend the exchange mechanism to offer the customers the ability to change the digital coins between different denominations. For example, a 50-cent coin can be exchanged for two 25-cent coins, and vise versa. So if Alice is charged 25 cents by Bob,

she can requests the bank to exchange her 50-cent coin for two 25-cent coins and then pay Bob with one of the 25-cent coins. This gives the sense of dividable digital cash. (Note that, this "divisibility" is on-line based, so it is not real divisibility according to our definition in Chapter 2). Consequently, the bank should update the database DB_{coins} by deleting the record of the 50-cent coin, and adding two records of the two 25-cent coins.

Chapter 5 Implementation

5.1 Introduction

In the previous chapters, a digital payment system is presented, which combines *eCash* and *NetCash*. In the presented system, there are three entities, a payer, a payee and a bank, involved in four protocols, withdrawal, exchange, payment and deposit. Cryptographic techniques such as digital signature and encryption/decryption are deployed to assure multi-party security. Moreover, it has been shown that the presented system is based on two mechanisms: blind signature and cash exchange. Blind signatures make unlinkability and untraceability available for digital cash, while exchange mechanism solves the drawbacks, such as bad scalability and unfair anonymity, which blind signatures introduce. In the presented system, there are two types of digital cash, digital notes and digital coins. Only notes can be withdrawn from the bank, while they can be exchanged for coins if necessary. At the bank, some databases are maintained to check the digital cash's genuineness and detect double-spending.

The system presented has been partly implemented. In this chapter, ideas of the implementation will be talked. The reason for implementation is just to show that the system is applicable. In other words, the implemented system is not a commercial product.

5.2 System requirements and architecture

The system is a typical three-tiered client/server system (refer to Figure 11). At the client tier, a GUI presentation should be implemented for digital cash clients to easily request and receive services from the bank. It should be noted that in the presented system, a client is active in participating in many aspects of the digital cash service. As an example, a client will generate and blind a note serial number, and on receiving signed note from the bank, he will un-blind the signature. Therefore, the client tier is not only a GUI, but also a computation concentric module.

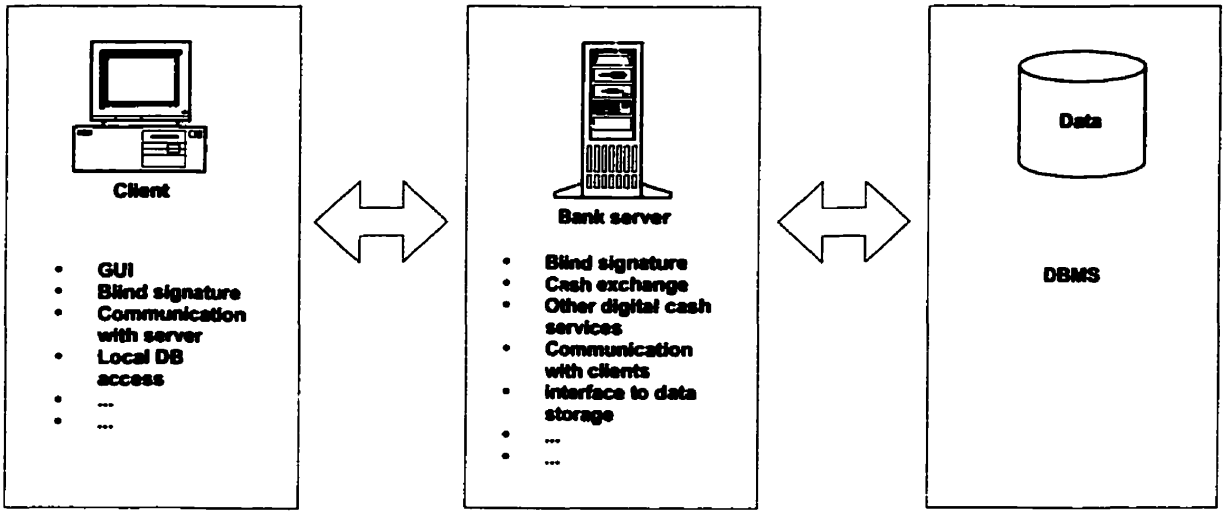

Figure 11 - System architecture

The middle-tier implements the digital cash services offered by the bank. This tier exhibits most of the logics of digital cash services. First, these two basic mechanisms, blind signature and cash exchange, have to be implemented. Blind signature depends on *RSA* cryptographic techniques, while other security issues such as digital signatures and encryption/decryption involved in the protocols rely on cryptography too. That means, we have to either implement some cryptographic algorithms or deploy some existing cryptography implementations. Cash exchange is primarily based on database processing, on which other features such as prevention of double-spending is built. Therefore, an efficient accessing interface to the third tier, the database systems, is another critical aspect in this tier. Another important issue worth to be noted is to implement the communication methods between the first and the second tier. TCP/IP networks are the most popular currently, and its future is very encouraging. Therefore, implementing the communication on the TCP/IP protocols should be a smart decision.

The third tier is the data storage. In this tier, all the information about clients, transactions, etc is maintained. It can be estimated that most digital cash services providers will be the existing financial institutes, who may already have some kinds of database management systems. The third tier can be build on these DBMS's, as long as

they provide some interfaces such as API's or drivers for accessing data they maintain. Therefore, for the data storage tier, the main task is to find some way to integrate it with the first and the second tiers.

5.3 Implementation tool

Java 2 SDK is chosen as the implementation tool for this system. Java's platform-independence property makes the first reason for this selection. Second, Java's object-oriented characteristic makes the implementation easy to be managed; it also permits the Java codes to be easily extended to fully implement the presented system. Third, Java 2 SDK offers some very intriguing packages that are extremely critical for the system. Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) provide the cryptographic engines such as digital signature and encryption/decryption. Java Database Connectivity (JDBC) offers the ability of accessing different relational databases with the same APIs. Java Foundation Class (JFC) makes Java competent in the development of graphic user interface (GUI).

5.4 What have been done

At this writing, several classes have been developed (refer to Figure 12). For the two basic mechanisms (blind signature and cash exchange) only the former is implemented. As to the four protocols (withdrawal, exchange, payment and deposit), only the first is implemented.

The class *DcashServer* has the *main* method of the server side program. It has an instance of class *Bank*, which defines the properties and actions of a bank that offers digital cash services. The class *Bank* makes use of class *KeyMachine* to generate *RSA* key pairs for each denomination of digital cash it is to mint. The class *DcashClient* has the *main* method of the client side program. It implements a simple GUI, and it has an instance of class *Client*, which defines the properties and actions of a digital cash client.

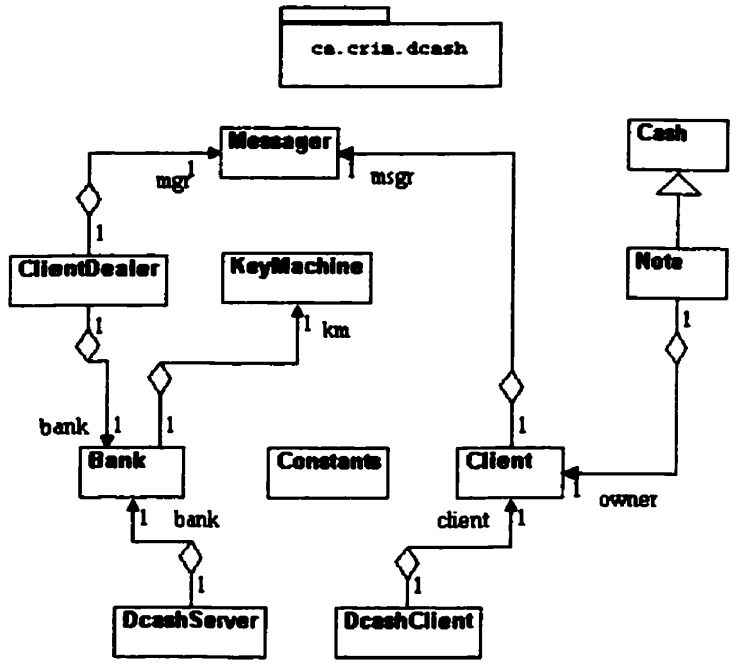

Figure 12 - Class diagram in package ca.crim.dcash (UML)

A *Bank* generates an instance of class *ClientDealer* whenever a *Client* requests for services. This *ClientDealer* serves the *Client* with a reference to the *Bank*. The communication between the *ClientDealer* and the *Client* is carried out by class *Messager*, of which both the *ClientDealer* and *Client* each has an instance.

The class *Cash* is the super class of the class *Note*. The former implements digital cash, while the latter implements digital note, which is one form of digital cash. In a *Note* object, an instance of *Client* named *owner* is defined to stand for the owner of a note.

The class *Constants* defines some constants shared by classes in the package. The next sections will address how these classes are implemented respectively.

5.5 Implement blind signature

One of the main efforts of the implementation is to implement blind signature. Blind signature is based on *RSA* digital signature, as shown in chapter 1. The cryptographic engines in Java that provide for digital signatures and the like are provided as a set of abstract classes in the Java security package. However, Sun does not provide an implementation of the *RSA* digital signature engine. Fortunately, the Java cryptography infrastructure allows third-party implementations of the engines. And, in the terms of programming, the infrastructure provides a consistent API that can be used by all programs, regardless of who is providing the actual implementation. In this sense, several third-party implementations of *RSA* digital signature is tested for the purpose of blind signature. Namely, they are JCE's from Cryptix, Forge Research, and Australian Business Access (ABA). The results are disappointing. As we have shown, the signature returned from the signer is signed on the blinded message. This signature is actually the multiplicative inverse of the signature that was signed on the un-blinded original message. Therefore, some modular arithmetic, as shown in chapter 3 and chapter 4, must be done on the returned signature.

However, Java cryptography infrastructure defines an RSA digital signature as *byte[ ]*, which is a DER-encoded PKCS#1 block as defined in RSA Laboratory's Public Key Cryptography Standards Note #1. This results to that decoding the signature must be done before we can do any arithmetic processing on it. For this reason, blind signature was directly implemented in this thesis following David Chaum's theory without making use of any existing RSA digital signature implementations.

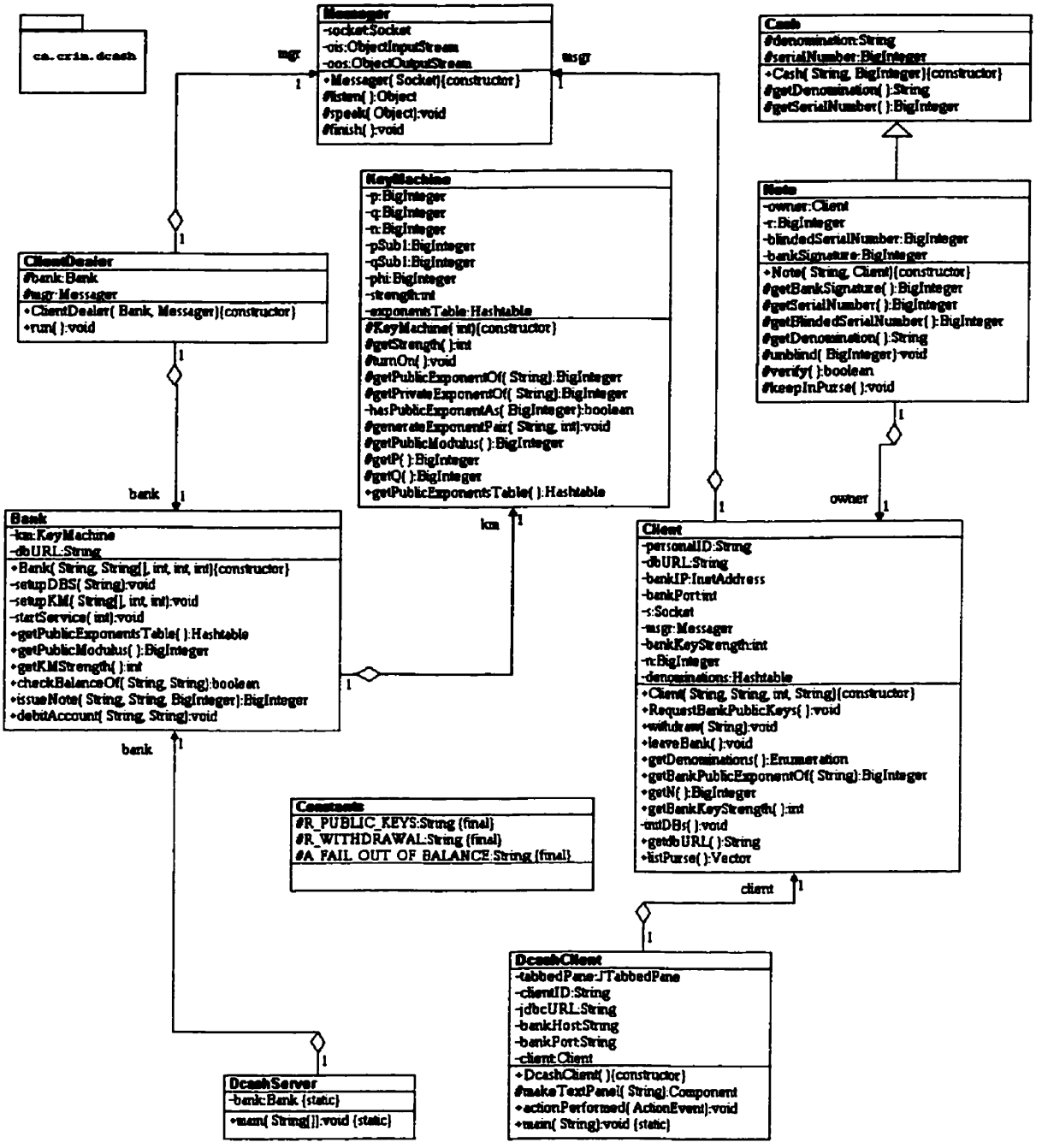

Figure 13 - Classes in package ca.crim.dcash with details (UML)

5.5.1 Generate public/private key pairs

The following codes generate p, q, n and ϕ of an RSA signature scheme:

```
p = new BigInteger(keyStrength, certainty, random);      // randomly generate large primes p

do {
        q = new BigInteger(keyStrength, certainty, random);
                                                         // and q, each roughly the same size of
                                                         // keyStrength
} while (p.equals(q));                                   // make sure p and q are distinct

if (p.compareTo(q) < 0) {                                // make sure p > q
        BigInteger tmp = p;
        p = q;
        q = tmp;
}

n = p.multiply(q);                                       // compute n = pq
pSub1 = p.subtract(one);
qSub1 = q.subtract(one);
phi = pSub1.multiply(qSub1);                             // and φ = (p-1)(q-1)
```

The code snipping above is pasted from the *turnOn* method of the class *KeysMachine*. This class generates the public/private key pairs for signing and verifying the digital cash issued by the bank. As mentioned in chapter 3 and chapter 4, for each denomination of digital cash, the bank should deploy a distinct key pair. It is also suggested by *DigiCash* that it is applicable to deploy the same public modulus n for each digital cash denomination, and deploy distinct public exponent e for each denomination. The following code snipping is pasted from the method *generateExponentPair*, which generates the public/private exponent pairs (*e/d* pairs).

```
BigInteger one = BigInteger.valueOf(1);
BigInteger three = BigInteger.valueOf(3);
BigInteger e,d;

do {
        do {
                e = new BigInteger(eNumBits, random);
                                                    // Generate a random e,

        } while ( e.compareTo(three) < 0  || hasPublicExponentAs(e) || e.compareTo(phi) >= 0 );

                                                    // which is bigger than 3, smaller than ϕ, and
                                                    // distinct from those for other denominations,
        while (!e.gcd(phi).equals(one)) {
                e = e.add(one);
        }                                           //  such that gcd(e, ϕ) = 1.

} while (e.compareTo(phi) >= 0 || hasPublicExponentAs(e));

d = e.modInverse(phi);                              // Compute d, which is e's multiplicative inverse
                                                    // of  modulus ϕ.
BigInteger[] exponentPair = {d, e};
exponentsTable.put(denomination, exponentPair);
                                                    // Save the exponentS pair.
```

Thus, we get a group of public/private key pairs: a public key is denoted as *(n, e)*, while its associated private key is *d*. One more thing need to be noted that, verification of signatures is significantly faster than signing if the public exponent is chosen to be a small number, and no weakness have been reported resulting from such a policy [AJM 97]. Since the implementation is based on client/server pattern, it is applicable to deploy this public exponent selection policy to make the computation on the client side more efficient. Thus, in the implementation, the *eNumBits* parameter, which determines the maximum length *e*, is chosen to be 7 bits in length.

5.5.2 Blind the serial number

The following code snipping is pasted from the constructor of the *Note* class, which defines the digital note as presented in chapter 4. The client program will call this constructor to generate a *Note* object. As shown in chapter 3 and chapter 4, this is done, at the client side, by randomly generating a serial number for the note, and blinding it with an also randomly generated blinding factor *r*, then sending the blinded serial number to the server side for signing.

```
SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
serialNumber = (new BigInteger(bankKeyStrength, certainty, sr)).mod(n);
                                                    // Randomly generate the serial number which is
                                                    // smaller than n
BigInteger g;
BigInteger one = BigInteger.valueOf(1);

do {
        r = new BigInteger(bankKeyStrength, certainty, sr);
                                                    // Randomly generate a prime r,
        r = r.mod(n);                               // which is smaller than n,
        g = r.gcd(n);
} while (g.compareTo(one) != 0);                    // and gcd(r,n) = 1.

BigInteger e = owner.getBankPublicExponentOf(denomination);
                                                    // Get the bank's public exponent e for the
                                                    // particular denomination.

blindedSerialNumber = (serialNumber.multiply(r.pow(e.intValue()))).mod(n);
                                                    // Blind the serial number, compute the blinded
                                                    // serial number.
```

5.5.3 Sign a blinded serial number

The following code snipping is pasted from the method *issueNote* of the class *Bank*. The server program at the bank generates such a *Bank* object, who deals with the issue of signing a blinded serial number. Note that the variable *serialNumber* holds the serial number the bank sees, in other words, the blinded serial number.

```
BigInteger bankSignature = serialNumber.modPow(d, getPublicModulus());
                                                    // Sign the blinded serial number
return bankSignature;                               // Return the signature
```

5.5.4 Unblind and verify a bank's signature

The following code snipping is pasted from the method *unblind* from the class *Note*. It unblinds a signature from the bank.

```
bankSignature = (blindedSignature.multiply(r.modInverse(n))).mod(n);
                                                    // Unblind the signature of the bank
```

The following code snipping is pasted from the method *verify* of the class *Note*. It verifies a bank's signature after it is unblinded.

```
BigInteger e = owner.getBankPublicExponentOf(denomination);
BigInteger n = owner.getN();
BigInteger x = bankSignature.modPow(e, n);      // Verify

if (serialNumber.compareTo(x) == 0) {
        return true;                            // Signature verified
}
else {
        return false;                           // Signature is not valid.
}
```

5.6 Implement the server-client communication

The communication between the bank server and a client program is message based. Usually, a client sends to the server a request, then waits for the response from the server. On the server's side, it listens to any request from the client, and responds to it by returning some messages. The class *Messager* is developed to provide this communication.

Messages between the server and a client are quite varied. For instance, when a client asks for the public key of the bank, the bank will respond with a signature, which in Java's definition, is a *Byte[ ]*. In another situation when the client requests for a withdrawal of a digital note, the bank should return a *Note* object. Therefore, it will be very difficult and inefficient if each type of message is addressed in the class *Messager*, especially it is the case when the system needs to be extended to support more functions. The solution to this issue is defining the message as a Java *Object*. In this way, what the class *Messager* deals with is always the same type: *Object*. The following code snipping is pasted from the class *Messager*.

```
public class Messager {

        private Socket socket;
        private ObjectInputStream ois;
        private ObjectOutputStream oos;


        protected Object listen() {
                // ......
                // ......
                ois = new ObjectInputStream(socket.getInputStream());
                return ois.readObject();
                // ......
                // ......
        }

        protected void speak(Object obj) {
                // ......
                // ......
                oos = new ObjectOutputStream(socket.getOutputStream());
                oos.writeObject(obj);
                oos.flush();
                // ......
                // ......
        }

}
```

Following this design pattern, there are at least two ways to parse the uniform *Object* messages into particular types. We can extend the class *Messager* to a particular one, who deals with a particular type of messages, such as *NoteMessager* who deals with the *Note* objects. The second way is leave the parsing responsibility to the receiver or deliver of the *Object* message. The present implementation follows the second way. There is a Class *ClientDealer* developed, one of whose functions is parse a client's request and respond according to the result of the parsing. The following code snipping is pasted from the class *ClientDealer*, showing how this works.

```
Object clientRequest;

while (! (clientRequest = mgr.listen()).equals(null) ) {        // A Messager mgr listen to the
                                                                // request from the client
        if (clientRequest instanceof  String) {                 // The received Object is a
                                                                // String
                String sRequest = (String) clientRequest;
                if (sRequest.equals(Constants.R_PUBLIC_KEYS)) {
```

```
                                                    // Client requests public keys of
                                                    // the bank
                    // ......
                    // ......
                    // ......
                    mgr.speak(bank.getN());
                    mgr.speak(bank.getPublicExponents());
                                                    // Respond the client with the
                                                    // bank's public keys.

                }
                else if (sRequest.equals(Constants.R_WITHDRAWAL)) {
                                                    // Client request a withdrawal.
                    // ......
                    // ......
                    mgr.speak(bankSignature);       // Respond with bank's
                                                    // signature on the issued cash
                }
                // Else ?
        }
        // Else ?
}
```

5.7 Implement database access

At both server and client side, there is database access involved. The server program at the bank maintains clients' balance, and the databases for cash double-spending checking. At the client side, it should at least maintain a local database of digital cash withdrawn or exchanged from the bank, or paid by other clients. Java offers a package called JDBC, which can access any relational database system as long as it provides a JDBC driver. Most of the main RDBMS's (Relational Database Management System) support JDBC, and in this implementation Microsoft Access 97 was chosen as the RDBMS, and *JDBC-ODBC Bridge* was chosen as the JDBC driver. However, it must be noted that JDBC's benefit is its RDBMS-independence. That means, only a bit of codes need to be modified to confirm to any other RDBMS such as ORACLE and SYBASE.

At the time of writing, the double-spending ability has not been implemented yet. It is easy to implement, however, since all it involves is just a verification of the bank's signature on digital cash, (which we have shown above in *verify* method of the class *Note*,), and some database queries as shown in chapter 3 and 4.

At the time of writing, one database is developed at each side. At the server side, a database *Bank* is generated with two tables, *Accounts* and *Denominations*. The former maintains clients' balance; the latter maintains denominations of digital bills together with their public exponents and private exponents. The following table shows what the latter looks like.

denomination	D	e
10.00	90864849230124340821617085422140986365913 85357125099886025396322974684439546230858 31634988528937644451644631982738 3	7
100.00	19570890603411396484655987629384212448042 98384611559975451623823409932033133034338 71429074452386569574200382273205 17	13
200.00	19358163531635185653301031242108297095346 86445648390845283671390546780597990283965 46739627990947585122306899857192 07	23
25.00	98698025887893680547618558303360036914699 53060325539531372413247369053787782974897 82638004781432268973338134739870 9	29
5.00	51294672952489547238009644996369911658177 17540312556387272401150066354119098678710 33987493524400283158186485796707 1	31

Table 2 - Table *Denominations* in the database *Bank*

It is clear to see, for each denomination the bank mints, its public exponent e and private exponent d is maintained. Because the public modulus n is shared by all denominations, it is not kept in this table.

At the client side, a database *Client* is generated with one table *Notes*, which maintains the digital notes in the client's "purse". In this table, each digital note has three fields: *Denomination*, *SerialNumber*, and *BankSignature*. The following shows what a client has in his "purse" after he made some withdrawals.

There are some classes at the time of writing dealing with database access. The following code snipping is pasted from the method *debitAccount* of the class *Bank*, just to show how to manipulate a database with JDBC.

```
Connection dbConnection = DriverManager.getConnection(dbURL);
                                                    // Connect to the database
Statement st = dbConnection.createStatement();
                                                    // Initialize a SQL statement object
st.executeUpdate("UPDATE Accounts " +
                 "SET balance = [balance] - " + denomination +
                 " WHERE customer = '" + client + "'");
                                                    // Define the SQL statement, and
                                                    // execute it.
st.close();
dbConnection.close();                               // Clearing work.
```

This method is called to debit the client's account after the bank issues a digital note to a client. The next code snipping is pasted from the method *keepInPurse* of the class *Note*. The method is called after the client received a note from the bank and has verified the bank's signature on the note.

```
dbConnection = DriverManager.getConnection(owner.getdbURL());
Statement st = dbConnection.createStatement();

String sqlString = "INSERT INTO Notes ( Denomination, SerialNumber, BankSignature )" +
                   "VALUES ( '" + getDenomination() + "', " +  "'" +
                                  getSerialNumber().toString() + "', " + "'" +
                                  getBankSignature().toString()+ "' )";

st.executeUpdate(sqlString);

st.close();
dbConnection.close();
```

10.00	279756966546134202847659425456 346454264023228358286123965557 299782166164530480684116826872 27769719593438444486109524	16375211366820888756761567560789426114451 40568504835902824248115670969567215958061 64004183622766546535091267537305921 0
100.00	216086031311420097029042533865 302471034166336752830816120481 255016877591439951438756507522 317717303910481335160481 81	23991464498392740493350548977172028647 14 83192385304033535250381096676328580390 24 85332107279078783182291376656628626 9
200.00	282485339934277287527999813319 221217513204281301988565794138 023000062500473955253265488366 050961959421665125626356 69	95362916709058713468992435405367065535 16 90348035047565802336044967463861222036 94 81734656262710625769504847902676254
25.00	292648465404317989804400444263 670357227370225455511227710361 839063051740224435683695711768 48980114670396154472931417	28863554435590047792237830010703947476 02 08341721881055696379544614633946592688 07 66696913314907665704592129486717176 4
5.00	256413889896667534204659956344 895879779348642548479809310445 495842774542561281934276379016 85575188418768888762849781	19537681130304550307617289370365874942 41 10531275301562109148011418213992486489 98 48067552374762466627107765536307604 7

Table 3 - Table *Notes* in the database *Client*

5.8 Implement graphic user interface (GUI)

The implementation includes a simple GUI for the client side. It is implemented with Java Foundation Class (JFC). The following figures show what it looks like.

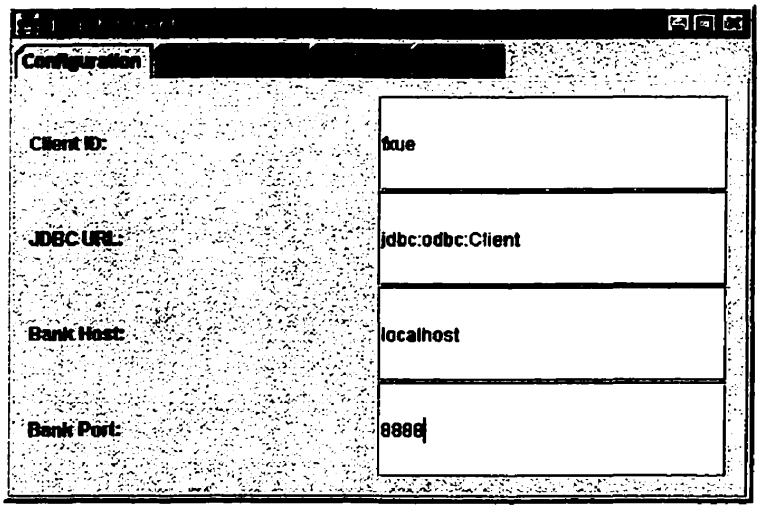

Figure 14 - GUI: Configuration

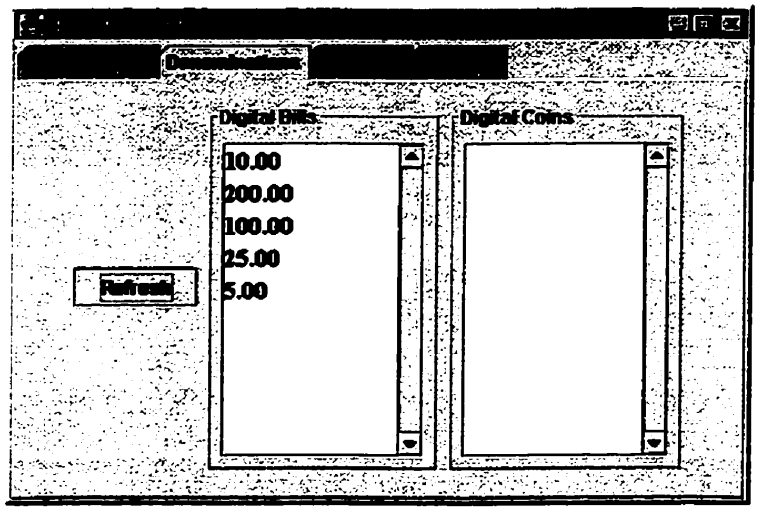

Figure 15 - GUI: Denominations

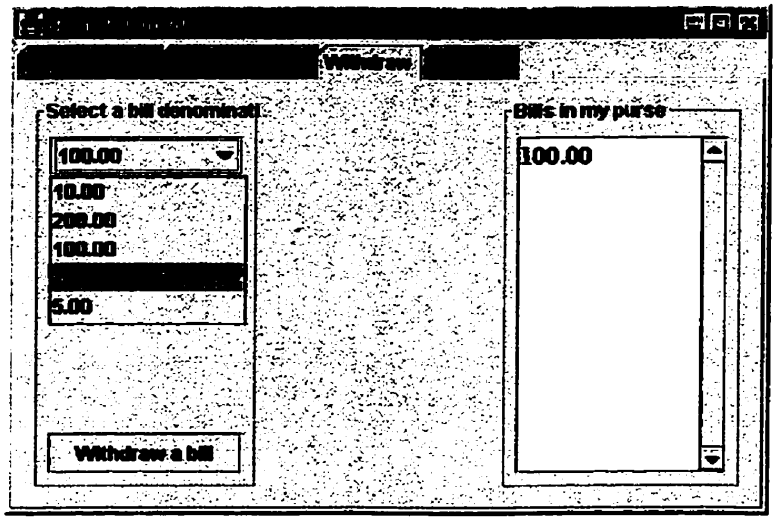

Figure 16 - GUI: Withdraw

Conclusion and future works

In this thesis, we have discussed three categories of digital payment systems: secure credit card system, credit-debit system, and digital cash system. Among the three, digital cash systems are most discussed. We have delved into two digital cash systems respectively: *eCash* and *NetCash*. The former deploys the blind signature mechanism to offer unconditional anonymity, while the latter deploys a mechanism called money-exchange to offer some degree of anonymity. Blind signatures, however, also produce such payoffs as unsatisfactory system scalability and unfair anonymity. On the contrary, the mechanism of cash-exchange, although offers anonymity which is not unconditional, it is fair to both payers and payees, and its scalability performance is better.

A new digital cash system is presented by combining both blind signatures and cash-exchange mechanisms. The most important feature of the system is that it defines two forms of digital cash, digital notes and digital coins. A note often bears a bigger denomination than a coin, and only notes can be withdrawn from the bank. A note can be exchanged to coins. Payments and deposits can be undertaken with both notes and coins. An extended system also allows exchanges between notes, or between coins, or from coins to a note. At the bank, two, instead one, databases are deployed to detect double-spending, double-exchange and double-deposit. An extended system even adds a third database to enhance its scalability further. The resulting system is a digital cash system which offer fair unconditional anonymity, and its scalability is greatly improved.

The system is partially implemented with Java SDK 2. It makes uses of JCA, JCE, JDBC and JFC to fulfill requirements of the system implementation. Blind signatures, server-client communication, database access, withdrawal protocol and a client side GUI have already be implemented. In the future, another base mechanism cash-exchange needs to be implemented. Then, other protocols (payment, exchange, and deposit) can be built upon blind signatures and cash-exchange.

Reference

[AJM97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone "Handbook of applied cryptography", ISBN 0-8493-8523-7

[CAFE] The ESPRIT project CAFE: High Security Digital Payment Systems <http://www.informatik.unihildesheim.de/FB4/Projekte/sirene/lit/abstr94.html#BBCM1_94>

[CyberCash] <http://www.cybercash.com/>

[DB98] Dan Boneh, R. Venkatesan: Breaking RSA may not be equivalent to factoring; In Proceedings Eurocrypt '98, Lecture Notes in Computer Science, Vol. 1233, Springer-Verlag, pp. 59--71, 1998.

[DC82] David Chaum, "Blind signatures for untraceble payments", Advances in Cryptology –Proceedings of Crypto '82, Lecture Notes in Computer Science, Springer-Verlag, pp. 199-203.

[eCash] <http://www.digicash.com>

[GVU10] GVU's Tenth WWW User Survey Graphs <http://www.gvu.gatech.edu/gvu/user_surveys/survey-1998-10/>

[HB89] H. Bürk, A. Pfitzmann. Digital Payment Systems Enabling Security and Unobservability. *Computers & Security,* 8/5 (1989), 399-416. <http://www.semper.org/sirene/publ/BufPf_89.ps.gz>

[iKP95] Mihir Bellare et al. IKP – A family of Secure Electronic Payment Protocols (Extended Abstract) <http://www.zurich.ibm.com/Technology/Security/extern/ecommerce/>

[ITAA98] E-Commerce Market Snapshot by Information Technology Association of America.
<http://www.itaa.org/ipecmmr1.htm>

[Mondex] <http://www.mondex.com/>

[NetBill] <http://www.ini.cmu.edu/NETBILL/>, <http://www.netbill.com>

[NetCash] Gennady Medvinsky & B. Clifford Neuman, "NetCash: A design for practical electronic currency on the Internet"

[NetCheque] <http://nii-server.isi.edu/info/netcash>

[Nua] Nua Internet How Many Online.
<http://www.nua.ie/surveys/how_many_online/index.html>

[Rob95] M.J.B. Robshaw. Security estimates for 512-bit RSA. Technical Note, RSA Laboratories, June 1995.

[RSA78] R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2): 120-126, February 1978.

[RSA98] RSA Data Security, Inc. Frequently asked questions about today's cryptography, version 4.0.
<ftp://ftp.rsa.com/pub/labsfaq/labsfaq4.pdf>

[SB95] Stefan Brands. Electronic Cash on the Internet
<http://www.cwi.nl/ftp/brands/e-cash.ps>

[SET] The SET Standard Specification
<http://www.setco.org/set_specifications.html>

[WS98] William Stallings, "Cryptography and network security. Principles and Practice", second edition, page 300. ISBN 0-13-869017-0.

Appendix: Source codes of the implementation

```
/**********************
 * DcashServer.java
 **********************/

import ca.crim.dcash.Bank;

class DcashServer {

    private static Bank bank;

    public static void main(String[] Args) {

        try {
            String dbURL = "jdbc:odbc:Bank";
            String[] mintTable = {"200.00", "100.00", "25.00", "10.00", "5.00"};
            int kmStrength = 384;
            int eNumBits = 7;
            int servicePort = 8888;
            bank = new Bank(dbURL, mintTable, kmStrength, eNumBits, servicePort);
        }
        catch (Exception e) {
            System.out.println(e);
        }

    }

}
```

```
/**********************
 * Bank.java
 **********************/

package ca.crim.dcash;

import java.util.Hashtable;
import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;
import java.math.BigInteger;
import java.sql.*;

public class Bank {

    private KeyMachine km;
    private String dbURL;

    public Bank(String dbURL, String[] mintTable, int kmStrength, int eNumBits, int servicePort) {

        setupDBS(dbURL);                                    // set up databases
        setupKM(mintTable, kmStrength, eNumBits);           // set up key machine
        startService(servicePort);

    }

    private void setupDBS(String dbURL) {

        this.dbURL = dbURL;
```

```
        try {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
        }
        catch (Exception e) {
        }
    }

    private void setupKM(String[] mintTable, int kmStrength, int eNumBits) {

        km = new KeyMachine(kmStrength);
        km.turnOn();

        try {
            Connection dbConnection = DriverManager.getConnection(dbURL);
            PreparedStatement pst = dbConnection.prepareStatement(
                                        "INSERT INTO Denominations ( denomination, d, e )" +
                                        "VALUES (? , ?, ? )"
                                    );

            String denomination;

            for (int i = 0; i < mintTable.length; i++) {

                denomination = mintTable[i];
                km.generateExponentPair(denomination, eNumBits);

                pst.setString(1, denomination);
                pst.setString(2, km.getPrivateExponentOf(denomination).toString());
                pst.setString(3, km.getPublicExponentOf(denomination).toString());
                pst.execute();

            }

            pst.close();
            dbConnection.close();

        }
        catch (SQLException e) {
        }
    }

    private void startService(int servicePort) {
        try {
            ServerSocket server = new ServerSocket (servicePort);
            System.out.println("Ready for service ... ");
            while (true) {
                Socket s = server.accept();
                Messager mgr = new Messager(s);
                ClientDealer cd = new ClientDealer(this, mgr);
                cd.start ();
            }
        }
        catch (IOException e) {
        }
    }

    public Hashtable getPublicExponentsTable() {
        return km.getPublicExponentsTable();
    }

    public BigInteger getPublicModulus() {
        return km.getPublicModulus();
    }

    public int getKMStrength() {
        return km.getStrength();
```

```
    }

    public boolean checkBalanceOf(String client, String denomination) {
        try {
            Connection dbConnection = DriverManager.getConnection(dbURL);

            Statement st = dbConnection.createStatement();
            ResultSet rs = st.executeQuery("SELECT balance FROM Accounts WHERE customer = '" + client + "'");

            if (rs.next()) {
                if ( rs.getFloat("balance") > Float.parseFloat(denomination) ) {
                    st.close();
                    dbConnection.close();
                    return true;
                }
                else {
                    st.close();
                    dbConnection.close();
                    return false;
                }
            }
            else {
                return false;
            }
        }
        catch (Exception e) {
            return false;
        }
    }


    public BigInteger issueNote(String client, String denomination, BigInteger serialNumber) {
                                                                    // this number is given & blinded by client
        if (! checkBalanceOf(client, denomination)) return null;

        BigInteger d = km.getPrivateExponentOf(denomination);
        if (d == null) return null;

        BigInteger bankSignature = serialNumber.modPow(d, getPublicModulus());
        debitAccount(client, denomination);

        return bankSignature;

    }

    public void debitAccount(String client, String denomination) {

        try {
            Connection dbConnection = DriverManager.getConnection(dbURL);
            Statement st = dbConnection.createStatement();
            st.executeUpdate("UPDATE Accounts " +
                             "SET balance = [balance] - " + denomination +
                             " WHERE customer = '" + client + "'");
            st.close();
            dbConnection.close();
        }
        catch (Exception e) {
        }
    }

}
```

```
/**********************
 * KeyMachine.java
 **********************/

package ca.crim.dcash;

import java.util.Hashtable;
import java.util.Enumeration;
import java.math.BigInteger;
import java.security.SecureRandom;

public class KeyMachine {

    private BigInteger p, q, n, pSub1, qSub1, phi;
    private int strength;
    private Hashtable exponentsTable;

    protected KeyMachine(int strength) {
        this.strength = strength;
    }

    protected int getStrength() {
        return strength;
    }

   protected void turnOn() {
        try {

            int keyStrength = (strength + 1) / 2;
            SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
            int certainty = 100;
            BigInteger one = BigInteger.valueOf(1);

            p = new BigInteger(keyStrength, certainty, random);

            do {
                q = new BigInteger(keyStrength, certainty, random);
            } while (p.equals(q));

            if (p.compareTo(q) < 0){
                BigInteger tmp = p;
                p = q;
                q = tmp;
            }

            n = p.multiply(q);
            pSub1 = p.subtract(one);
            qSub1 = q.subtract(one);
            phi = pSub1.multiply(qSub1);

            exponentsTable = new Hashtable();
        }
        catch (Exception ex) {
        }
    }

    protected BigInteger getPublicExponentOf(String denomination) {
        BigInteger[] keyPair = (BigInteger[]) (exponentsTable.get(denomination));
        if (keyPair == null) {
            return null;
        }
        else {
            return keyPair[1];
        }
    }
```

```
protected BigInteger getPrivateExponentOf(String denomination) {
    BigInteger[] keyPair = (BigInteger[]) (exponentsTable.get(denomination));
    if (keyPair == null) {
        return null;
    }
    else {
        return keyPair[0];
    }
}

private boolean hasPublicExponentAs(BigInteger e) {

    Enumeration exponentPairs = exponentsTable.elements();
    BigInteger[] exponentPair;

    while (exponentPairs.hasMoreElements()) {
        exponentPair = (BigInteger[]) (exponentPairs.nextElement());
        if (exponentPair[1].equals(e)) return true;
    }

    return false;

}


protected void generateExponentPair(String denomination, int eNumBits) {

    try {
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
        BigInteger one = BigInteger.valueOf(1);
        BigInteger three = BigInteger.valueOf(3);

        BigInteger e,d;

        do {
            do {
                e = new BigInteger(eNumBits, random);
            } while (e.compareTo(three)<0 || hasPublicExponentAs(e) || e.compareTo(phi)>=0);

            while (!e.gcd(phi).equals(one)) {
                e = e.add(one);
            }

        } while (e.compareTo(phi) >= 0 || hasPublicExponentAs(e));

        d = e.modInverse(phi);
        BigInteger[] exponentPair = {d, e};
        exponentsTable.put(denomination, exponentPair);
    }
    catch (Exception ex) {
    }

}


protected BigInteger getPublicModulus() {
    return n;
}

protected BigInteger getP() {
    return p;
}

protected BigInteger getQ() {
    return q;
```

```
    }

    public Hashtable getPublicExponentsTable() {
        Hashtable publicExponentsTable = new Hashtable();
        String k;
        BigInteger publicExpo;

        for (Enumeration e = exponentsTable.keys(); e.hasMoreElements(); ) {
            k = (String) e.nextElement();
            publicExpo = ((BigInteger[])(exponentsTable.get(k)))[1];
            publicExponentsTable.put(k, publicExpo);
        }

        return publicExponentsTable;
    }

}
```

```
/**********************
 * ClientDealer.java
 **********************/

package ca.crim.dcash;

import java.net.*;
import java.io.*;
import java.util.*;
import java.math.BigInteger;

public class ClientDealer extends Thread {

    protected Bank bank;
    protected Messager mgr;

    public ClientDealer (Bank bank, Messager mgr) throws IOException {
        this.bank = bank;
        this.mgr = mgr;
    }

    public void run () {

        try {

            Object clientRequest;

            while (! (clientRequest = mgr.listen()).equals(null) ) {
                if (clientRequest instanceof  String) {
                    String sRequest = (String) clientRequest;
                    if (sRequest.equals(Constants.R_PUBLIC_KEYS)) {
                        mgr.speak(new Integer(bank.getKMStrength()));
                        mgr.speak(bank.getPublicModulus());
                        mgr.speak(bank.getPublicExponentsTable());

                    }
                    else
                        if (sRequest.equals(Constants.R_WITHDRAWAL)) {
                            System.out.println("Client withdrawal requested.");
                            Object o0 = mgr.listen();
                            if (! (o0 instanceof String)) {
                                System.out.println("Protocol unexpected.");
                                return;
```

```
                        }

                        String client = (String) o0;

                        Object o1 = mgr.listen();
                        if (! (o1 instanceof String)) {
                            System.out.println("Protocol unexpected.");
                            return;
                        }

                        String denomination = (String) o1;

                        Object o2 = mgr.listen();
                        if (! (o2 instanceof BigInteger)) {
                            System.out.println("Protocol unexpected.");
                            return;
                        }

                        BigInteger serialNumber = (BigInteger) o2;

                        BigInteger bankSignature = bank.issueNote(client, denomination, serialNumber);

                        if (bankSignature != null) {
                            mgr.speak(bankSignature);
                        }
                    }
                }
            }

            mgr.finish();
        }
        catch (Exception e) {
        }

    }

}
```

```
/**********************
 * Messager.java
 **********************/

package ca.crim.dcash;

import java.net.*;
import java.io.*;

public class Messager {

    private Socket socket;
    private ObjectInputStream ois;
    private ObjectOutputStream oos;

    public Messager(Socket socket) {
        try {
            this.socket = socket;
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
```

```
    protected Object listen() {

        try {
            ois = new ObjectInputStream(socket.getInputStream());
            return ois.readObject();
        }
        catch (Exception e) {
            return null;
        }

    }

    protected void speak(Object obj) {

        try {
            oos = new ObjectOutputStream(socket.getOutputStream());
            oos.writeObject(obj);
            oos.flush();
        }
        catch (Exception e) {
        }

    }

    protected void finish() {

        try {
            socket.close();
            ois.close();
            oos.close();
        }
        catch (IOException ioe) {
        }

    }

}
```

```
/**********************
 * DcashClient.java
 **********************/

import ca.crim.dcash.Client;
import java.util.Enumeration;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class DcashClient implements ActionListener {

    JTabbedPane tabbedPane;
    String clientID;
    String jdbcURL;
    String bankHost;
    String bankPort;
    Client client;
```

```
public DcashClient() {

    tabbedPane = new JTabbedPane(SwingConstants.TOP);
    Border paneEdge = BorderFactory.createEmptyBorder(10,10,10,10);

    JPanel panel1 = new JPanel(false);
    panel1.setBorder(paneEdge);
    panel1.setLayout(new GridLayout(4,2));
    JLabel label11 = new JLabel("Client ID:");
    JTextField textField11 = new JTextField(20);
    textField11.setActionCommand("txtfld11");
    textField11.addActionListener(this);
    JLabel label12 = new JLabel("JDBC URL:");
    JTextField textField12 = new JTextField(20);
    textField12.setActionCommand("txtfld12");
    textField12.addActionListener(this);
    JLabel label13 = new JLabel("Bank Host:");
    JTextField textField13 = new JTextField(20);
    textField13.setActionCommand("txtfld13");
    textField13.addActionListener(this);
    JLabel label14 = new JLabel("Bank Port:");
    JTextField textField14 = new JTextField(20);
    textField14.setActionCommand("txtfld14");
    textField14.addActionListener(this);
    panel1.add(label11);
    panel1.add(textField11);
    panel1.add(label12);
    panel1.add(textField12);
    panel1.add(label13);
    panel1.add(textField13);
    panel1.add(label14);
    panel1.add(textField14);
    tabbedPane.addTab("Configuration", null, panel1, "View / Modify system parameters");
    tabbedPane.setSelectedIndex(0);

    JPanel panel2 = new JPanel(false);
    panel2.setBorder(paneEdge);
    JButton button21 = new JButton("Refresh");
    button21.setActionCommand("bttn21");
    panel2.add(button21);
    final JTextArea textArea21 = new JTextArea();
    textArea21.setEditable(false);
    textArea21.setFont(new Font("Serif", Font.BOLD, 16));
    JScrollPane areaScrollPane21 = new JScrollPane(textArea21);
    areaScrollPane21.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    areaScrollPane21.setPreferredSize(new Dimension(150, 250));
    areaScrollPane21.setBorder(
    BorderFactory.createCompoundBorder(
    BorderFactory.createCompoundBorder(
    BorderFactory.createTitledBorder("Digital Bills"),
    BorderFactory.createEmptyBorder(5,5,5,5)),
    areaScrollPane21.getBorder()));
    panel2.add(areaScrollPane21);
    JTextArea textArea22 = new JTextArea();
    textArea22.setFont(new Font("Serif", Font.BOLD, 16));
    JScrollPane areaScrollPane22 = new JScrollPane(textArea22);
    areaScrollPane22.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    areaScrollPane22.setPreferredSize(new Dimension(150, 250));
    areaScrollPane22.setBorder(
    BorderFactory.createCompoundBorder(
    BorderFactory.createCompoundBorder(
    BorderFactory.createTitledBorder("Digital Coins"),
    BorderFactory.createEmptyBorder(5,5,5,5)),
    areaScrollPane22.getBorder()));
    panel2.add(areaScrollPane22);
```

```
        tabbedPane.addTab("Denominations", null, panel2, "Get dcash denominations the bank offers");

        JPanel panel3 = new JPanel(new BorderLayout(), false);
        panel3.setBorder(paneEdge);
        JPanel panel31 = new JPanel(new BorderLayout(), false);
        panel31.setBorder(
        BorderFactory.createCompoundBorder(
        BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder("Select a bill denomination"),
        BorderFactory.createEmptyBorder(5,5,5,5)),
        panel31.getBorder()));
        JButton button31 = new JButton("Withdraw a bill");
        final JComboBox comboBox31 = new JComboBox();

        panel31.add(comboBox31, BorderLayout.NORTH);
        panel31.add(button31, BorderLayout.SOUTH);
        final JTextArea textArea31 = new JTextArea();
        textArea31.setFont(new Font("Serif", Font.BOLD, 16));
        JScrollPane areaScrollPane31 = new JScrollPane(textArea31);
        areaScrollPane31.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        areaScrollPane31.setPreferredSize(new Dimension(150, 250));
        areaScrollPane31.setBorder(
        BorderFactory.createCompoundBorder(
        BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder("Bills in my purse"),
        BorderFactory.createEmptyBorder(5,5,5,5)),
        areaScrollPane31.getBorder()));
        panel3.add(panel31, BorderLayout.WEST);
        panel3.add(areaScrollPane31, BorderLayout.EAST);
        tabbedPane.addTab("Withdraw", null, panel3, "Withdraw dcash bills from the bank");

        Component panel4 = makeTextPanel("Blah blah blah blah");
        tabbedPane.addTab("Payment", null, panel4, "Does nothing at all");

        button21.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                textArea21.setText(null);
                client = new Client(clientID, bankHost, Integer.parseInt(bankPort), jdbcURL);
                System.out.println("Connected to bank.");
                client.RequestBankPublicKeys();
                System.out.println("Got Keys.");

                Enumeration denos = client.getDenominations();
                while (denos.hasMoreElements()) {
                    String de = (String) denos.nextElement();
                    textArea21.append(de + "\n");
                    comboBox31.addItem(de);
                    //c.withdraw(de);
                    System.out.println(de);
                }
            }
        });

        button31.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String deno = (String) (comboBox31.getSelectedItem());
                client.withdraw(deno);
                Enumeration notes = client.listPurse().elements();
                textArea31.setText(null);
                while (notes.hasMoreElements()) {
                    textArea31.append(((String) notes.nextElement()) + "\n");
                }
            }
        });

    }
```

```
    protected Component makeTextPanel(String text) {
        JPanel panel = new JPanel(false);
        JLabel filler = new JLabel(text);
        filler.setHorizontalAlignment(JLabel.CENTER);
        panel.setLayout(new GridLayout(1, 1));
        panel.add(filler);
        return panel;
    }

    public void actionPerformed(ActionEvent evt) {

        String src =  evt.getActionCommand();

        if (src.equals("txtfld11")) {
            clientID = ((JTextField) evt.getSource()).getText();
            System.out.println("Client ID: " + clientID);
        }
        else if (src.equals("txtfld12")) {
            jdbcURL = ((JTextField) evt.getSource()).getText();
            System.out.println("JDBC URL: " + jdbcURL);
        }
        else if (src.equals("txtfld13")) {
            bankHost = ((JTextField) evt.getSource()).getText();
            System.out.println("Bank Host: " + bankHost);
        }
        else if (src.equals("txtfld14")) {
            bankPort = ((JTextField) evt.getSource()).getText();
            System.out.println("Bank Port: " + bankPort);
        }

    }

    public static void main(String s[]) {
        JFrame frame = new JFrame("dcash Client");

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        ClientGUI gui = new ClientGUI();
        frame.getContentPane().setLayout(new GridLayout(1, 1));
        frame.getContentPane().add(gui.tabbedPane);

        frame.pack();
        frame.setVisible(true);
    }

}
```

```
/**********************
 * Client.java
 **********************/

package ca.crim.dcash;

import java.io.*;
import java.net.*;
import java.sql.*;

import java.math.BigInteger;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;

public class Client {

    private String personalID;
    private String dbURL;

    private InetAddress bankIP;
    private int bankPort;
    private Socket s;
    private Messager msgr;

    private int bankKeyStrength;
    private BigInteger n;
    private Hashtable denominations;

    public Client(String ID, String bankHost, int bankPort, String dbURL) {

        this.personalID = ID;
        this.dbURL = dbURL;
        initDBs();

        try {
            bankIP = InetAddress.getByName(bankHost);
        }
        catch (UnknownHostException e) {
            System.out.println("Unknown DNS name.");
        }

        this.bankPort = bankPort;

        try {
            s = new Socket(bankIP, bankPort);
            msgr = new Messager(s);
        }
        catch (IOException e) {
            System.out.println("Connection fails.");
        }

    }

    public void RequestBankPublicKeys() {

        msgr.speak(Constants.R_PUBLIC_KEYS);
        Object o0 = msgr.listen();

        if (! (o0 instanceof Integer)) {
            System.out.println("Protocol unexpected.");
            return;
        }
```

```
        bankKeyStrength = ((Integer) o0).intValue();

        Object o1 = msgr.listen();

        if (! (o1 instanceof BigInteger)) {
            System.out.println("Protocol unexpected.");
            return;
        }

        n = (BigInteger) o1;

        Object o2 = msgr.listen();
        if (! (o2 instanceof Hashtable)) {
            System.out.println("Protocol unexpected.");
            return;
        }

        denominations = (Hashtable) o2;
    }


    public void withdraw(String denomination) {
        try {
            Note note = new Note(denomination, this);
            msgr.speak(Constants.R_WITHDRAWAL);
            msgr.speak(personalID);
            msgr.speak(denomination);
            msgr.speak(note.getBlindedSerialNumber());

            Object o0 = msgr.listen();
            System.out.println("Got: " + o0);

            if (o0 instanceof BigInteger) {
                note.unblind((BigInteger) o0);
                if (note.verify()) {
                    note.keepInPurse();
                }
            }
            else {
                if ((o0 instanceof String) && ((String)o0).equals(Constants.A_FAIL_OUT_OF_BALANCE)) {
                        System.out.println("Bank responds: Out of balance.");
                }
                else System.out.println("Protocol unexpected.");
            }
        }
        catch (Exception e) {
        }
    }

    public void leaveBank() {
        msgr.finish();
    }

    public Enumeration getDenominations() {
        return denominations.keys();
    }

    public BigInteger getBankPublicExponentOf(String denomination) {
        return (BigInteger) (denominations.get(denomination));
    }

    public BigInteger getN() {
        return n;
    }
```

```
    public int getBankKeyStrength() {
        return bankKeyStrength;
    }

    private void initDBs() {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
        }
        catch (Exception e) {
        }
    }

    public String getdbURL() {
        return dbURL;
    }

    public Vector listPurse() {
        try {
            Vector notes = new Vector();

            Connection dbConnection = DriverManager.getConnection(dbURL);
            System.out.println("DB: connect ok.");

            Statement st = dbConnection.createStatement();
            System.out.println("Statement ok.");

            ResultSet rs = st.executeQuery("SELECT Denomination FROM Notes");
            System.out.println("Execute SQL ok.");

            while (rs.next()) {
                notes.addElement(rs.getString("Denomination"));
            }
            st.close();
            dbConnection.close();

            return notes;
        }
        catch (Exception e) {
            return null;
        }
    }


}
```

```
/**********************
 * Cash.java
 **********************/

package ca.crim.dcash;

import java.math.BigInteger;

public class Cash {

    protected String denomination;
    protected BigInteger serialNumber;

    public Cash (String denomination, BigInteger serialNumber) {
        this.denomination = denomination;
        this.serialNumber = serialNumber;
```

```
    }

    protected String getDenomination() {
        return denomination;
    }

    protected BigInteger getSerialNumber() {
        return serialNumber;
    }
}
```

```
/**********************
 * Note.java
 **********************/

package ca.crim.dcash;

import java.math.BigInteger;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
import java.sql.*;

public class Note extends Cash {

    private Client owner;

    private BigInteger r;
    private BigInteger blindedSerialNumber;

    private BigInteger bankSignature;

    public Note(String denomination, Client owner) throws NoSuchAlgorithmException{

        super(denomination, BigInteger.ZERO);

        this.denomination = denomination;
        this.owner = owner;

        SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
        int bankKeyStrength = owner.getBankKeyStrength();
        BigInteger n = owner.getN();
        int certainty = 100;

        serialNumber = (new BigInteger(bankKeyStrength, certainty, sr)).mod(n);

        BigInteger g;
        BigInteger one = BigInteger.valueOf(1);

        do {
            r = new BigInteger(bankKeyStrength, certainty, sr);
            r = r.mod(n);
            g = r.gcd(n);
        } while (g.compareTo(one) != 0);

        BigInteger e = owner.getBankPublicExponentOf(denomination);

        blindedSerialNumber = (serialNumber.multiply(r.pow(e.intValue()))).mod(n);

    }
```

```
    protected BigInteger getBankSignature() {   // called after the bank's signature is un-blinded
        return bankSignature;
    }

    protected BigInteger getSerialNumber() {
        return serialNumber;
    }


    protected BigInteger getBlindedSerialNumber() {
        return blindedSerialNumber;
    }

    protected String getDenomination() {
        return denomination;
    }

    protected void unblind(BigInteger blindedSignature) {

        BigInteger n = owner.getN();
        bankSignature = (blindedSignature.multiply(r.modInverse(n))).mod(n);

    }

    protected boolean verify() throws Exception {

        BigInteger e = owner.getBankPublicExponentOf(denomination);
        BigInteger n = owner.getN();
        BigInteger x = bankSignature.modPow(e, n);

        if (serialNumber.compareTo(x) == 0) {
        return true;
     }
     else {
        return false;
     }
    }

    protected void keepInPurse() {

        Connection dbConnection;

        try {
            dbConnection = DriverManager.getConnection(owner.getdbURL());
            Statement st = dbConnection.createStatement();
            String sqlString = "INSERT INTO Notes ( Denomination, SerialNumber, BankSignature )" +
                               "VALUES ( '" + getDenomination() + "', " +
                               "'" + getSerialNumber().toString() + "', " +
                               "'" + getBankSignature().toString()+ "' )";
            st.executeUpdate(sqlString);
            st.close();
            dbConnection.close();
        }
        catch (SQLException e) {
            System.out.println("In keepInPurse: " + e);
        }

    }

}
```

```
/**********************
 * Constants.java
 **********************/

package ca.crim.dcash;

public class Constants {
    protected static final String R_PUBLIC_KEYS = "BKK";
    protected static final String R_WITHDRAWAL = "WDRL";

    protected static final String A_FAIL_OUT_OF_BALANCE = "FBLC";
}
```