E-CASH

Anonymous Electronic Payments

Diploma dissertation

Michel Deriaz
University of Geneva

13th of June 2003

Table of contents

1. Introduction

The aim of this work is to choose and implement a payment system that can be easily integrated in a mobile device. Among the main expected characteristics, this electronic purse must allow direct client-to-client payments, e-shopping over Internet, and the transactions must be achieved anonymously.

Our research led us to the e-cash protocol. A comparison with other payment schemes will point out the pros and the cons of this system, and a complete Java implementation will help the reader to fully understand the mechanism of electronic money.

From gold coins to e-cash, including notes and credit cards, we will discover why there is not a unique payment system, and why it is so difficult to impose a new standard. Lots of protocols are designed, but the most of them will stay marginal. The next chapter, "Inventory of existing systems", will present among others the Swiss CASH system and introduce the main ideas of electronic cash.

The e-cash system uses different cryptography tools, namely asymmetric cryptography to establish a secure channel between participants, encryption to protect data from unauthorized readers, hash functions to increase efficiency, and digital signatures to authenticate the notes. The "Cryptography basis" chapter will present theses functionalities, which are essential to understand the e-cash protocol.

The chapter "How e-cash works" will first describe the protocol in a textual from, and then the complete algorithm will be presented and explained step-by-step. The reader will understand how cryptography prevents any attempt of cheating.

Once the reader is more familiar with e-cash, he can immerse into the Java code. The chapter "The e-cash package" presents the classes from the java.kernel package, that implements the e-cash algorithm, and the classes from the java.internet package, which adds the necessary functionalities to run the system over the Internet.

The next chapter, "JCash and JBank" is a demonstration of a complete e-purses application, running over Internet. The clients can buy notes at the bank, send them to other customers, and of course deposit them in order to credit their bank account.

At the end of this paper, the reader will be able to use the functionalities offered by the ecash package in order to set up his own system. Because the kernel package has been written with the idea of future possible improvements in mind, it is sufficient generic to make the adding of an extension easy. Today it works over computers connected to the internet, but tomorrow we will perhaps see this system running on PDA (Personnel Digital Assistant) or even on mobile phones.

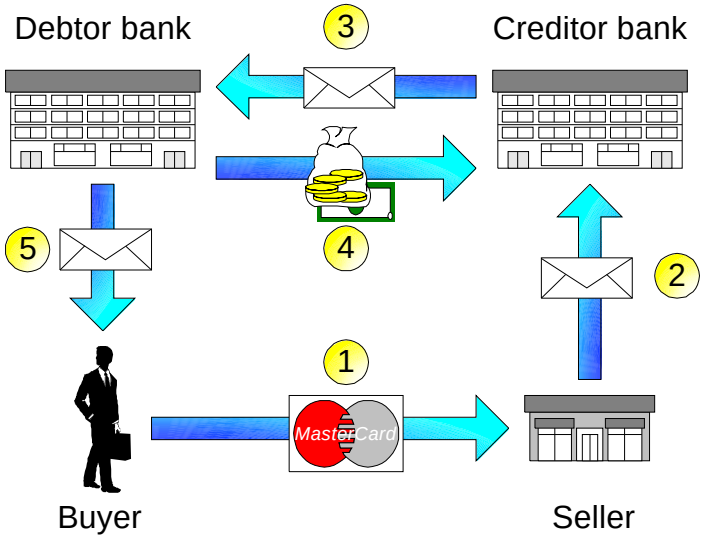
2. Inventory of existing systems

Business exists since Cro-Magnon Man. They probably didn't pay with credit cards but they made trade by swapping goods. Every kind of object was a currency. They just had to negotiate the exchange rates (how many flints for one spear?).

Much later, the need of a reference currency made that gold was chosen for that. It was the first kind of money but going shopping with your gold coins and gold plaques is not very adapted to today business, and particularly not for small amounts.

Thinks became easier with banknotes and silver coins. The amounts written on them corresponded to a certain quantity of gold, so it is actually fundamentally not so different. But like every token system, it has a price. Printing notes and pressing coins is quite expensive, so that small coins cost more to be manufactured than the value that they represent. And because physical mass is proportional to the amount of cash held, large amounts of cash are difficult and expensive to store and move. It has been estimated that the handling costs of transporting cash in the US amount to more than $60 billion a year. For the customers the disadvantages are more evident. The first question is which amount and which denominations to carry with? A too small amount limits your buying liberty, but a too big amount is heavier and increases the risk to be robbed. In the other hand, a too small choice in your denominations will be detrimental in cases where no change is given back, and a big choice of denomination will ruin the shape of your pockets.

The credit card system pushes back the limit of the available amount. A single card allows you to access to all your money, and this in a quite secure way. And you can withdraw the exact amount, so there is no change return problem anymore. The functioning is similar to the checks system. The buyer don't give directly money to the seller, but only an authorisation allowing the creditor bank to withdraw a certain sum from the debtor one.



1. The buyer gives an order of payment to the seller.
2. The seller sends an authorisation to his bank.
3. The creditor bank sends the authorisation to the debtor bank.
4. The money is transferred from the buyer's account to the seller's account.
5. The buyer gets a notification from the bank.

Let's compare the pros and cons of conventional money and the credit card system. Criteria are:

- Peer-to-peer: Means that a client can give money to another client without any specific device like a card reader.

- Paying over Internet. Means that the system allows doing transactions through Internet.

- Anonymous: Means that the identity of the client won't be revealed during the payment. For example the credit card system doesn't preserve your anonymity; because you reveal you card number, the bank knows where you spend your money.

- Adapted for micro-payments: We call micro-payments all kind of small-amount payments, like we do with coffee distributors or bus-tickets machines. If a system is adapted for micro-payments, it means that it is fast (no need to enter a code, no waiting for a connection to the bank), and that every amount can be treated.

- Adapted for large amounts: A system is qualified as adapted for large amounts if it allows a client to access to much more money than he usually bears with him (in cash), and this in a secure way. In a secure way means that he doesn't loose money in case of robbery.

- Free of charges: Means that the transactions are not extra-charged. For example a credit card company takes a percentage of every transaction; therefore this system isn't free of charges.

- Need of a third party: These kinds of transactions need to make a connection to a third party (the bank) in order to be achieved.

- Always exact change: Such a system allows the client to pay always the exact amount. For example, traditional banknotes and coins don't meet this requirement.

- Light: A light system is one that its weight doesn't increase with the available amount and that doesn't need a specific material (for example a card reader) to make transactions, at the buyer side.

- International: Means that the system allows the client to pay in every country and that the change of currency will be automatically done.

Conventional money

Pros	Cons
• Peer-to-peer • Anonymous • Adapted for micro-payments • Free of charges • No need of a third party	• No paying over Internet • Not adapted for large amounts • Not always exact change • Not light • Not international

Credit cards

Pros	Cons
• Paying over Internet • Adapted for large amounts • Always exact change • Light • International	• Not peer-to-peer • Not anonymous • Not adapted for micro-payments • Not free of charges • Need a third party

We find also card systems that are more similar than conventional notes and coins. For example the prepaid cards that we use on phone booth preserve your anonymity, work for micro-payments and in case of lost the result is the same; you loose the amount that was on the card, not more not less. The problem is that such a card has a limited range of acceptance; the phone prepaid card will be useful only in phone booths, and you won't be able to pay a coffee or to buy a bus ticket with it. And carrying a wad of cards won't change a lot from carrying a conventional purse containing silver coins with different denominations.

A better system is the one which a single card is accepted by different seller (newspaper, coffee, phone booths, ...), and which can be reloaded. In Switzerland this system is called CASH and is widely accepted. There are currently (June 2003) 3.7 millions people that own such a card, which allow them to pay in more than 30'000 points of acceptance, from parking spaces to some restaurants, including public transports and food distributors. This card can be loaded with an amount of maximum CHF 300.-- at the bank or at the post, and the bank account (or post account) is automatically debited. The system consists in a smart card that can be included in the ec/Maestro credit card or in the Postcard, in order to save a card (you can also get a neutral card that contains only the CASH functionalities). Such a system is very close to conventional money because it preserves your anonymity, it is adapted for micro-payments and transactions are not charged.

CASH

Pros	Cons
• Anonymous • Adapted for micro-payments • Free of charges • No need of a third party • Always exact change • Light	• Not peer-to-peer • No paying over Internet • Not adapted for large amounts • Not international

Two other payment systems deserve to be mentioned here. The first is the SMS paying, used for example to download a new logo or a new ring for a mobile phone. The client sends a SMS describing what he wants and the amount will be charged directly on his phone bill. The main advantage is that everybody who can send SMS can also use this system.

The second is the one introduced in Finland by Sonera Mobile, where the client has to compose a phone number to get a service. For example, the user composes with is mobile phone the number displayed on the coffee distributor, and the price of the coffee will be billed by its mobile phone company.

SMS / Phone payments

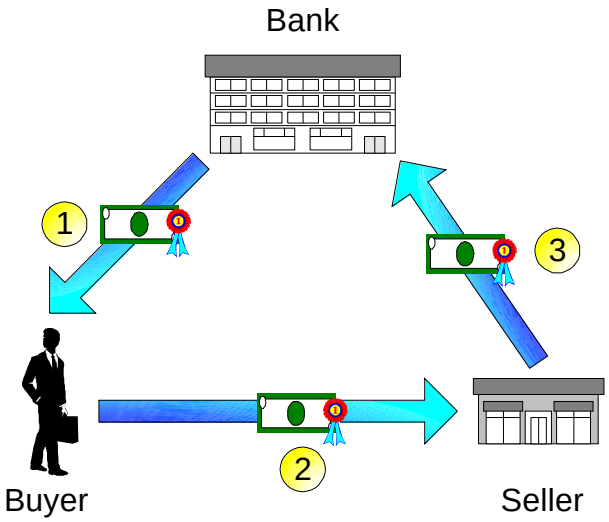
Pros	Cons
• Adapted for micro-payments • Free of charges • No need of a third party • Always exact change • Light	• Not peer-to-peer • No paying over Internet • Not anonymous • Not adapted for large amounts • Not international

Each system has its pros and cons and there is not a system that is better than another in all points of view. And there is no risk that modern payment ways are going to suppress our coins and notes in a short time. Even if big amounts are often paid electronically (for example salaries), about 90% of the world's payments are currently still made in cash.

We have just presented a sample of what exists today. They are a lot of systems that are developed but it is seldom that one of them is successful in the market. The reason is quite simple to understand. Imagine you find out a system that lists much more pros than cons. You prove that your system satisfies the client's needs. But the problem is who wants to be your first client? A system, even if it is wonderful, is completely useless if you are alone to use it. The success depends not only on the technology, but also on how to convince people (buyers and merchants) to invest in new products. On the consumer side, survey data shows the single most important factor is wide acceptance of the system.

The idea presented in this paper is to use electronic notes. Unlike our traditional silver coins and paper banknotes, electronic notes don't need any physical support. A note is only information that can be stored on a computer hard-disk, sent over the Internet, and why not printed out and sent by traditional post. It is very similar to conventional money in the way that the value of the money is in the money itself. But it is one-time-use money; a buyer buys a note at the bank, sends it to a seller, and the latter deposits it in his bank account. Such a system brings all our initial requirements together. It allows direct client-to-client payments, e-shopping over Internet, and the transactions are achieved anonymously. Our implementation will therefore be based on the e-cash protocol, invented by DigiCash in 1990, which describes how to do transactions with electronic notes.

To simplify we keep only one bank:



1. The buyer gets a digitally signed note from the bank.
2. The buyer spends his note to a seller.
3. The seller deposits his note in his bank account.

e-cash

Pros	Cons
• Peer-to-peer • Paying over Internet • Anonymous • Adapted for micro-payments • Free of charges • No need of a third party • Always exact change • Light	• Not adapted for large amounts • Not International

Because you don't need any special material (card reader), every customer can act either as a buyer (sending notes) or as a seller (receiving notes). And this point is really fundamental; a system that keeps the advantages of conventional money will of course be more easily accepted.

Although it seems to be very promising, do you know a lot of merchants using this system? Certainly not. And transforming your paper banknotes and silver coins into charged electrons on you hard-disk or in you PDA (Pocket Digital Assistant) is probably a greater abstract leap than the transformation of gold coins to conventional money.

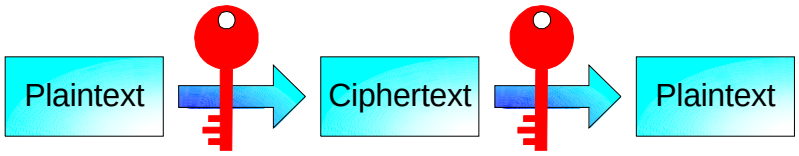
3. Cryptography basis

Cryptography is the science of using mathematics to encrypt and decrypt data. A text that can be read without any special measures is called plaintext. Encrypting this text consist in applying a mathematical function that disguise it in such a way as to hide its substance. The result is called ciphertext. The opposite way, transforming a ciphertext into a plaintext, is called decryption.

Cryptography allows two people to exchange data through an insecure channel, like Internet. The sender encrypts the plaintext and sends the resulting ciphertext. The receiver decrypts this ciphertext and gets the initial plaintext. Even if a third person is listening and intercepting every message, he only gets ciphertexts, which is completely useless if he doesn't know the secret key used. But cryptography is not only limited to hiding data. It allows also among others to create a fingerprint of a document (hash functions) and to digitally sign a text (like a hand-written signature). The combination of theses techniques, as it is done in the e-cash protocol, makes of cryptography a really powerful tool. We distinguish two main techniques used to encrypt data: symmetric cryptography and asymmetric cryptography.

Symmetric cryptography

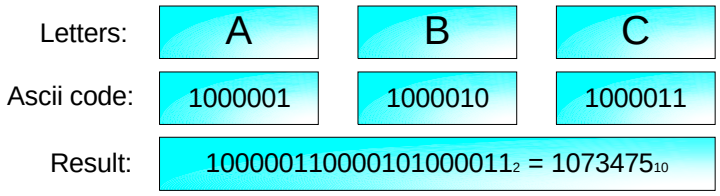
This method has been invented 2000 years ago, by Julius Caesar. It consists of using the same secret key to encrypt and decrypt data.



Only people that know the secret key can crypt and decrypt messages. The secret key of Caesar was to shift every letter of the plaintext by 3. So he replaced every A in his messages with a D, every B with an E and so on through the alphabet. Only someone who knew the "shift by 3" rule could decipher his messages. This system allows only 26 different keys. But 2000 years ago this system was secure enough to transmit orders to his generals without taking the risk to be read by the messengers. Today the keys are typically 128-bits integers. This means that even if you possess a billion computers doing a billion operations a second, you won't be able to try each possible secret key before the end of the universe.

In cryptography we work with numbers. So keys are numbers. And texts are also numbers, in fact: Remember that a letter is coded as a succession of bits. And if we align the different letters, respectively their bit-representation, we still obtain a succession of bits. And any

succession of bits is a number coded in base 2. For example the text "ABC" can be represented by the number 1073475_{10}:



Therefore every encrypting or decrypting operation will result in applying to data a mathematical function using the secret key. They are a lot of algorithms; the following list presents quickly the most important:

DES

DES stands for Data Encryption Standard. It was first developed by IBM in 1975 under the name "Lucifer". The NSA (National Security Agency) also had a hand in the algorithm. At any rate, DES has withstood more than 20 years of intense cryptanalytic scrutiny. But today its 56-bit key size is considered as not safe enough. To prove that a specialised machine called Deep Crack solved in January 1999 a key in only 22 hours and 15 minutes.

DESede

DESede, also called triple DES is a variant of the DES cipher algorithm. The blocks of plaintext are transformed into ciphertext using three DES keys and three applications of a normal DES cipher. So the plaintext is first encrypted with the first key, then decrypted using the second key and finally encrypted using the third key. It is this process of Encryption - Decryption - Encryption that gives DESede its name. The decryption process is naturally as follow: the ciphertext is decrypted with the third key, then encrypted using the second key and finally decrypted using the first key. DESede ciphertext is much harder to cryptanalyze than DES ciphertext. Effectively, the key increased to a length of 3 * 56 = 168 bits. Note that if you use three times the same key, the security will be the same than with DES. But in accordance with the RSA Laboratories, such a system provides not an equivalent security to an initial 168-bit cryptosystem.

IDEA

IDEA stands for International Data Encryption Algorithm. It was invented in Switzerland at the ETHZ and was first published in 1990. It uses a 128-bit key size, and seems to be really sure. It is used by the well known PGP (Pretty Good Privacy) program.

RC4

RC4 stands for Ron's Code 4 or Rivest's Cipher 4, in accordance with the first name and surname of its inventor. This cipher uses a key-length between 40 and 2048 bits. This is interesting because some countries restricts the length if keys that can be used. This algorithm is really fast and is used in the SSL (Secure Socket Layer) protocol, which allows making safe transaction like e-banking (https:// instead of http://).

Blowfish

Blowfish is one of the most common symmetric block ciphers implemented in Java. Blowfish and Twofish (Twofish is the version after Blowfish) were invented by one of the most famous cryptography authors, Bruce Schneier. The success of this algorithm in Java implementations can partly be explained by the fact that it is a non-patent and free algorithm to use. It takes a variable-length key from 32 bits to 448 bits. Since the key can be varied from a low to a high range, it is ideal for exporting; you just need to adapt the key-size to the exportations regulations and the local rules.

The following code uses Blowfish to crypt and to decrypt a message typed on the command line:

```
import javax.crypto.*;
import java.security.*;


public class TestBlowfish {
  final static int KEY_SIZE = 128;  // [32..448]


  public static void main(String[] args) {
    try {
      String plaintext = args[0];
      System.out.println("plaintext = " + plaintext);
      KeyGenerator keyGen = KeyGenerator.getInstance("Blowfish");
      keyGen.init(KEY_SIZE);
      Key secretKey = keyGen.generateKey();
      Cipher cipher = Cipher.getInstance("Blowfish");
      cipher.init(Cipher.ENCRYPT_MODE, secretKey);
      byte[] input = plaintext.getBytes("ISO-8859-1");
      byte[] ciphertext = cipher.doFinal(input);
      System.out.print("ciphertext = ");
      for (int i = 0; i < ciphertext.length; i++) {
        System.out.print(ciphertext[i] + " ");
      }
      System.out.println("");
      cipher.init(Cipher.DECRYPT_MODE, secretKey);
      byte[] plain = cipher.doFinal(ciphertext);
      String plaintext2 = new String(plain, "ISO-8859-1");
      System.out.println("plaintext2 = " + plaintext2);
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

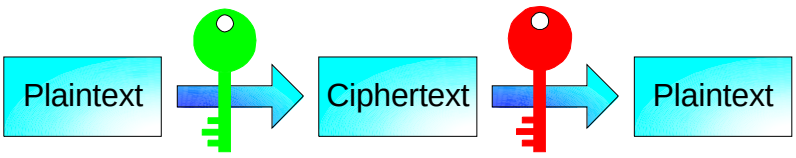
TestBlowfish.java

Asymmetric cryptography

Symmetric cryptography is fast and secure. But how is it possible to exchange the secret key in a secure manner? We can for example exchange it through another channel, like the phone. Or store the private key in a floppy and give it physically to the receiver. Of course they are a

lot of solutions, but they are not very handy. And what if you need to communicate securely with someone you've never met, or how to establish a secure channel between you and your bank to make e-banking?

The solution is called asymmetric cryptography, and is certainly one of the most innovations in the field. The concept was introduced by Whitfield Diffie and Martin Hellman in 1975. The idea is to use two keys, one for encryption and the other for decryption.



A user generates mathematically two keys. They are created so that the public key (green) encrypts a plaintext and that the private key (red) decrypts the resulting ciphertext. The security resides in the fact that it is computationally infeasible to deduce the private key from the public key if you don't know the secret values used to build them. Once a key pair is created, the user published the public key and keeps the private key secret. Therefore everybody is able to encrypt a message using the public key, but only the owner of the corresponding private key is able to decipher the message. The need for sender and receiver to share secret keys via some secure channel is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. Some examples of public-key systems (another designation for asymmetric cryptography) are Elgamal (named for its inventor, Taher Elgamal), RSA (named for its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman), Diffie-Hellman (the authors who introduced the concept of asymmetric cryptography), and DSA (Digital Signature Algorithm, invented by David Kravitz). The most popular is RSA, which is used in the SSL protocol, and the following description shows how its works:

RSA

First you generate the pair of keys:

1. Choose two different big prime numbers, p and q.
2. Compute the modulus $n = p \cdot q$ and $\phi(n) = (p-1) \cdot (q-1)$.
3. Choose a number $e \in]1..\phi(n)[$ so that $gcd(e,\phi(n)) = 1$.
4. Compute $d = e^{-1} \bmod \phi(n)$, i.e. $e \cdot d \bmod \phi(n) = 1$
5. Publish the public key = (e, n) and keep the private key (d, n) secret.

The security resides in the fact that it is computationally infeasible to factorize n if big enough. Of course, the intermediate values (p, q and $\phi(n)$) must be kept secret as well.

Encryption of a plaintext m into a ciphertext c:

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

Decryption of a ciphertext c into a plaintext m:

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

The following example uses very small numbers, but still shows how RSA works: Let's say that Bob want to send a secret message, 9726, to Alice.

Alice:

$p = 101$
$q = 113$
$n = p \cdot q = 11413$
$\phi(n) = (p-1) \cdot (q-1) = 11200$
$e = 3533$
$d = e^{-1} \bmod \phi(n) = 6597$

Alice publishes its public key (e, n) = (3533, 11413)
and keeps secret its private key (d, n) = (6597, 11413)

Bob:

$m = 9726$
$c = m^e \bmod n = 9726^{3533} \bmod 11413 = 5761$

Bob sends $c = 5761$

Alice:

$c = 5761$
$m = c^d \bmod n = 5761^{6597} \bmod 11413 = 9726$

The following program tests the RSA algorithm. The user can either leave the program working alone (AUTO = true) or enter himself the different values (AUTO = false). Note: All the entries are numbers, even the plaintext. And of course the latter must be smaller than the modulus *n*.

```
import java.io.*;
import java.math.*;
import java.security.*;


public class TestRSA {
  final static boolean AUTO = true;
  final static int KEY_SIZE = 512;  // [512..2048] if GEN_KEY_AUTO == true


  public static void main(String[] args) {
    BigInteger plaintext, p, q, n, fiN, e, d;
    if (AUTO) {
      p = BigInteger.probablePrime(KEY_SIZE, new SecureRandom());
      do q = BigInteger.probablePrime(KEY_SIZE, new SecureRandom());
```

```
      while (q.equals(p));
      fiN = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
      do e = new BigInteger(KEY_SIZE, new SecureRandom());
      while (!e.gcd(fiN).equals(BigInteger.ONE));
      plaintext = new BigInteger(KEY_SIZE, new SecureRandom());
    }
    else {
      p = readBI("p: ");
      q = readBI("q: ");
      e = readBI("e: ");
      plaintext = readBI("plaintext: ");
      fiN = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
    }
    n = p.multiply(q);
    d = e.modInverse(fiN);
    System.out.println("\np = " + p);
    System.out.println("q = " + q);
    System.out.println("n = " + n);
    System.out.println("fiN = " + fiN);
    System.out.println("e = " + e);
    System.out.println("d = " + d);
    System.out.println("plaintext = " + plaintext);
    BigInteger ciphertext = plaintext.modPow(e, n);
    System.out.println("ciphertext = " + ciphertext);
    BigInteger plaintext2 = ciphertext.modPow(d, n);
    System.out.println("plaintext2 = " + plaintext2);
    if (plaintext2.equals(plaintext)) System.out.println("\nOK\n");
    else System.out.println("\nError: plaintext2 != plaintext\n");
  }


  private static BigInteger readBI(String label) {
    BufferedReader keyboard = new BufferedReader(new
InputStreamReader(System.in));
    try {
      System.out.print(label);
      return new BigInteger(keyboard.readLine());
    }
    catch(Exception e) {}
    return null;
  }
}
```

TestRSA.java

Hybrid systems

Both previous techniques have advantages and disadvantages. Symmetric cryptography is much faster (about 1000 times), needs smaller keys (128 bits in symmetric system provides more or less the same security than 1024 bits in the asymmetric system) and is easier to manage (only one key). But asymmetric cryptography solves the weakest point of symmetric systems: How to transmit the key.

Hybrid systems combine the advantages of booth: The messages are transmitted thanks to symmetric cryptography and the secret key used to crypt and decrypt them is transmitted thanks to asymmetric cryptography. PGP (Pretty Good Privacy), like a lot of other systems, uses hybrid systems to achieve its goals.

Hash functions

A hash function is a one-way function that transforms an arbitrary long message into a fixed-size fingerprint. Such a function ensures that if the information is changed in anyway, even by just one bit, an entirely different output value is produced. It is computationally infeasible to find two different texts that produce the same message digest (fingerprint), and it is of course impossible to compute the original text from its fingerprint.

For example, if you are not entirely sure that a public key belongs to the claimer, you can (if you know him) phone and ask him to read the fingerprint of the key. Incidentally, more and more people are writing public key fingerprints on business cards. The two most common hash functions are enumerated here:

MD5

MD5 stands for Message Digest 5, and was invented by Ron Rivest (the same that put the R in RSA). It is an improvement of MD4 which is itself an improvement of MD2 (the higher the version, the more robust against collision). It produces a 128-bit fingerprint and it is a very fast algorithm. The main weakness remains the low resistance against collisions (a collision occurs when two different plaintexts produce the same fingerprint). Many believe that it would take a machine that costs $10 million about 24 days to find a collision.

SHA

SHA stands for Secure Hash Algorithm and was developed by the NIST (National Institute of Standards) and the NSA (National Security Agency). It is closely modelled after the MD4 algorithm and was designed for use with the DSA (Digital Signature Algorithm) in mind. Because the complexity and collision resistance is higher in SHA, it is about 30% slower than MD5.

Some examples of digests:

```
Message: Transfer $2000 to account S314542.0
MD5: 96 23 122 -10 -28 124 -44 -106 -43 26 19 52 -31 -3 -68 3
SHA: 74 90 20 47 -32 -110 -98 123 125 32 -117 125 -118 102 31 66 -47 60 -123 -73
```

```
Message: Transfer $20000 to account S314542.0
MD5: 87 -2 13 -30 -85 -10 -106 -44 105 111 126 -74 30 -72 98 39
SHA: 121 67 72 28 -32 71 32 -118 12 -1 -108 -8 91 6 -99 -56 90 -56 -69 96
```

```
Message: Transfer $2000 to account S314542.1
MD5: 46 -56 64 14 -115 76 59 -15 -106 -41 -40 50 -39 59 -15 -107
SHA: 106 44 43 -30 36 -4 -126 102 64 -6 -34 93 115 -38 17 -13 -62 -100 -14 2
```

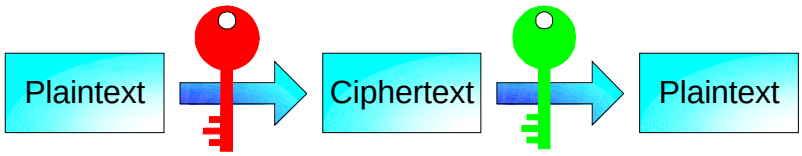
The following program has been used to produce these outputs:

```
import java.security.*;
import java.io.*;


public class TestHash {
  final static String ALGO = "SHA";  // SHA or MD5


  public static void main(String[] args) {
    try {
      MessageDigest md = MessageDigest.getInstance(ALGO);
      md.update(args[0].getBytes("ISO-8859-1"));
      byte[] digest = md.digest();
      System.out.println("Algo: " + ALGO + " (" + digest.length + " bytes)");
      for (int i = 0; i < digest.length; i++) {
        System.out.print(digest[i] + " ");
      }
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

TestHash.java

Digital signatures

Asymmetric cryptography provides the possibility for everybody to encrypt a message (using the public key) and to be sure that only the owner of the corresponding private key will be able to decrypt it. But it doesn't provide authentication; all the messages are anonymous. What about Oscar how passes of as the bank and pleases Alice to identify her by sending her bank account number and her password? Digital signatures allow a person to digitally sign a document. An easy way consists simply in encrypting (signing) a document with the private key, and so everybody can decrypt (checking the signature) with the corresponding public key:

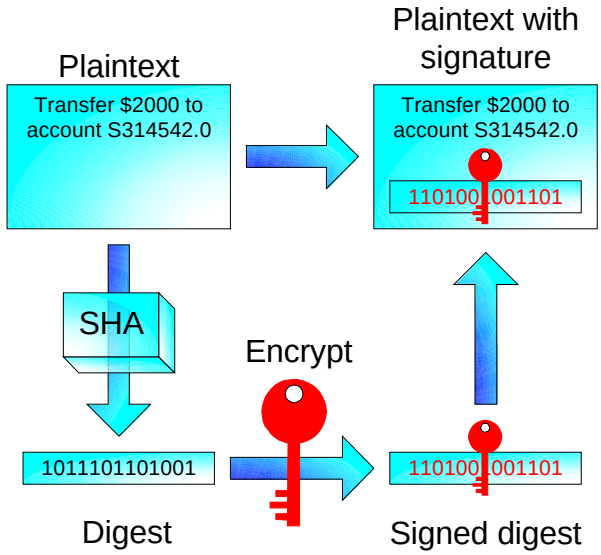


Digital signatures have the same purposes than handwritten ones, but provide a much higher rate of security. A classical signature is quite easy to counterfeit, and it is even easier to modify the initial text afterwards. With digital signatures, even an insignificant modification will produce a completely different signed document (ciphertext) and therefore also an unreadable plaintext during the signature checking process.

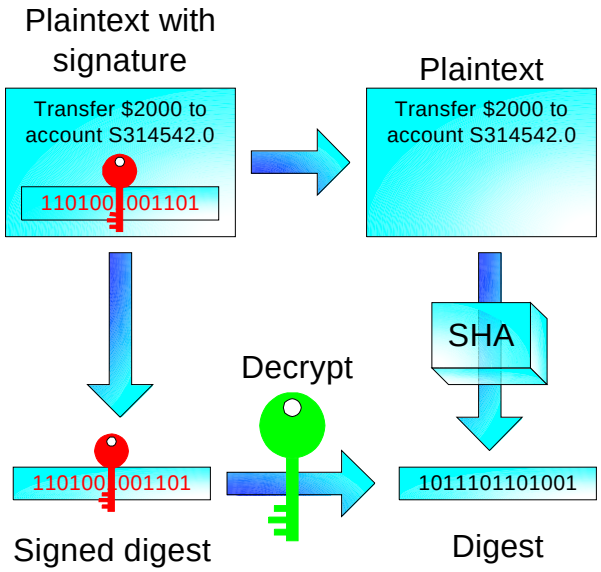
Of course the ciphertext can be read be anyone. But sometimes it doesn't matter. For example, you may not care if anyone knows that you just deposited $100 in you bank account, but you

want to be darn sure it was the bank teller you were dealing with. And if does matter, nothing will prevent you from encrypting the signed text (using another pair of keys).

In practice however, it is not very efficient to sign a whole text using this method. Remember that asymmetric cryptography is very slow. That's why we actually sign only the fingerprint of the message:



To check the signature, we recreate the digest from the original plaintext, we decrypt the signed digest using the public key, and finally we make sure that the two results are equal:



If either the text or the signature changes, the test will fail. The following program signs a text and then verifies the signature. The global constants CHEAT_TEXT and CHEAT_SIGNATURE allow a user to modify the text or the signature after signing, in order to check the verification process:

```
import java.io.*;
import java.security.*;


public class TestDSA {
  final static boolean CHEAT_TEXT = false;
  final static boolean CHEAT_SIGNATURE = false;


  public static void main(String args[]) {
    try {
      byte[] text = "Transfer $2000 to account S314542.0".getBytes("ISO-8859-1");
      System.out.println("\nGenerating a pair of 512-bit DSA keys...");
      KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
      keyPairGen.initialize(512, new SecureRandom());
      KeyPair kp = keyPairGen.generateKeyPair();
      System.out.println("Signing the text...");
      Signature signature = Signature.getInstance("DSA");
      signature.initSign(kp.getPrivate());
      signature.update(text);
      byte[] sig = signature.sign();

      if (CHEAT_TEXT) text[0] = 0;
      if (CHEAT_SIGNATURE) sig[4] = 0; // don't change sig[0] (exception)

      System.out.println("\nVerifying the signature...");
      signature.initVerify(kp.getPublic());
      signature.update(text);
      boolean ok = signature.verify(sig);
      System.out.println("Signature is " + (ok ? "OK" : "NOT OK") + " !\n");
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

TestDSA.java

4. How e-cash works

Introduction

If we compare real-life money (coins and notes) with the popular electronic payments (credit cards), we quickly notice that they have different advantages. With coins and notes a user stays anonymous and don't need a third party (the bank) to make a transaction. Therefore a payment is free of charge. The credit card system needs the availability of the bank, is charged for every transaction, is not anonymous, but avoids a user to carry large amount of money, avoid forgery and it seems to be a quite good secure system.

E-cash is a system that tries to take party of both sets of advantages, allowing free-of-charge payments and micro-payments, working for spending on the Internet, and all this by preserving the anonymity of the buyer. Instead of printed notes, the idea is to use electronic notes. Therefore a note has no physical support anymore; a note is actually only information. In Java we represent such a note with a BigInteger, which is an unlimited-size integer. The algorithm described later in this chapter shows how cryptography makes such a system feasible and un-forgeable.

Description

The functioning of this kind of money is very simple, in a first point of view. A buyer buys some electronic notes at the bank, and the requested amount is deduced from his bank account. He spends then his notes to another client, called the seller. And finally the latter sends this money to the bank in order to credit his bank account. Of course, in a practical situation, clients won't make a connection to the bank for every transaction. Remember that one aim of electronic cash is to avoid the availability of a third party. A final-user program could be configured so that a connection is automatically made to the bank each time a client needs a note that isn't available in his purse. And during this single connection, the missing note is bought, the already used notes (given by a buyer) are credited, and the purse is updated again with different denominations.

Cheating seems to be very easy. You just need to make a copy of all your notes to become twice as rich; and copying digital information is trivial. Cryptography is used to avoid double-spending. The first point is that the bank signs digitally each note that comes out. This avoids people to create their own bank notes. And the second point is that in addition to the amount, each digital note contains a coded form of the buyer's identity. During a payment operation (buyer-seller), a part of this identity is revealed, but just not enough to loose the anonymity. But if a buyer tries to use twice his note, he will have to supply too many information about his identity, and then he won't stay anonymous anymore.

Even the seller can't cheat by depositing twice the same note. The bank records all the notes that are deposited and checks this list each time a new one is credited. And a seller, like a buyer, is unable to create a valid note without the signature of the bank.

To stay anonymous even for the bank, which can record all the released notes, it is impossible to let him build the notes alone. That's why it is actually the client who builds them. Then he puts each note in an envelope and asks the bank to sign this envelope. Mathematically, it consists roughly in multiplying the note by a secret number, the blinding factor, then to make the bank sign this result and finally to divide the signed envelope by the blinding factor in order to get a signed note. Because we obtain information that the bank signed without seeing it, this process is called blind-signature.

The problem of the blinding process is the buyer who makes the bank sign an envelope containing a large amount but who says that it is only a small note. For example, if a buyer announces a $20 note and makes the bank signing an envelope containing a $1000 note, the bank will only deduce $20 from the bank account of the buyer, but will have to refund $1000 to the seller when the latter sends this note. That's why a buyer actually don't send only one note to the bank, but several (let's say 100), all with the same amount. The bank chooses then one of them, and asks the buyer to reveal all the information needed (identities and blinding factors) to build the other envelopes. If everything is ok and if all the amounts are the same, the banks supposes that the last one is also all right and returns it signed.

The text just above gives a roughly description on the way electronic cash works. The main point to remember is that an e-note is just a big number containing an amount and information about the initial owner, so that there is not enough information to work out his identity in normal use, but so that it is very easy to compute his identity in case of cheating. To understand fully the process, which is in fact not much more complex, a good understanding of cryptography basis is necessary.

The e-cash algorithm

The first point consists in generating valid bank notes, recognizable by the bank. To stay anonymous, the buyer creates a note, hides it in an envelope, and asks the bank to sign it. This process is called blind signature; the bank signs something without seeing it. Asymmetric cryptography is used to achieve this goal.

The bank generates a pair of key. The public key is (e, n) and the private key is (d, n). These letters stand for:

- e: public encryption exponent;
- d: private decryption exponent;
- n: the modulus.

Let's take a sequence of bits that identifies the client, for example it's bank account. Then we split this value with a XOR function, so that

$$x \oplus x' = User_{id}$$

Let's repeat that several time, so that:

$$x_i \oplus x_i' = User_{id} \forall i$$

This sequence of x is called the RIS (Random Identity String).

A hash function applied to each item of the RIS will produce a sequence of y:

$$y_i = H(x_i) \ , \ y_i' = H(x_i')$$

A bank note will looks like:

$$M = (amount, y_1, y_1', y_2, y_2', ..., y_p, y_p')$$

Because a hash is a one-way function, it is impossible to deduce the identity hided in the bank note.

In this project, *M* is a BigInteger that contains all the values in an Indian file. Because it is quite a huge value, the cryptographic operations are made on a hash of *M*:

$$m = H(M)$$

The buyer generates a random number *k*, the blinding factor, and sends the envelope to the bank:

$$Buyer \rightarrow Bank : (m \cdot k^e) \bmod n$$

The bank returns the signed envelope to the buyer:

$$Bank \rightarrow Buyer : (m \cdot k^e)^d \bmod n$$

The client computes:

$$\frac{(m \cdot k^e)^d}{k} \bmod n = \frac{m^d \cdot k^{ed}}{k} \bmod n = \frac{m^d \cdot k}{k} \bmod n = m^d \bmod n = signedNote$$

At this point the client owns a note signed by the bank, despite the latter never seen it. The question is now how to be sure that the note the bank signs corresponds really to the amount announced be the client? They are two ways to solve this problem. The first is simply to use a different signature for each amount. It's very simple, but it is also limited; a pair of keys is needed for each different amount, what makes impossible the creating of notes of any amount. The second solution is the one discussed previously: Make several notes (all with the same amount), send them together, let the bank choose one of them and finally reveal all the information (RIS and the blinding factors *k*) used to build the other notes; if everything is

correct, the bank signs blindly the last envelope and returns it. This second approach has been chosen for this project.

The buyer can now pay a seller by sending him the note *M* and the signed hash of *M*:

$$Buyer \rightarrow Seller : M, signedNote$$

The seller verifies that the signature is correct, i.e. the bank has actually signed the note:

$$H(M) = signedNote^e \bmod n$$

The seller sends a random challenge to the client to make sure that the identity written on the note is correct:

$$Seller \rightarrow Buyer : r \text{ (p bits)}$$

The buyer has to reveal a part of his RIS (enough to check its validity but just not enough to find out his identity):

$$Buyer \rightarrow Seller : RISpart = [x_i \text{ if } r_i = 0,\ x_i' \text{ if } r_i = 1\ \forall i \in (1..p)]$$

The seller verifies that:

$$\text{if } r_i = 0 : H(x_i) = y_i$$
$$\text{if } r_i = 1 : H(x_i') = y_i'$$

If everything is correct, the seller accepts the note. It is important to notice that this kind of money is one-time use; once a seller got a note, he needs to send it to the bank in order to credit his bank account.

The last step consists in depositing the note at the bank. In addition to the note, the seller sends also the *RISpart* and the challenge *r*:

$$Seller \rightarrow Bank : M, signedNote, RISpart, r$$

The bank first verifies that the signature and the values given in the *RISPart* are correct. Then if this note has not already been deposited, the latter is accepted and the account of the seller is credited. But if this note is already in the used-notes list, the bank will have to discover the cheater:

- If both *RISpart* are the same, then it is the seller who cheats; he tries to deposit twice the same note.

- If the *RISpart* is different for each note, then it is the buyer who cheats. Only a buyer is able to generate a valid RIS. Therefore, if the bank gets two different valid *RISpart*, it means that they have been generated in response to the two different challenges, [r_{11}, r_{12}, r_{13}, ... , r_{1p}] and [r_{21}, r_{22}, r_{23}, ... , r_{2p}], imposed by the sellers. This allow the bank to identify

the cheater; because the challenges are random, they are different from each other (with probability $1-2^{-p}$), and therefore it exists at least one position *i* where

$$r_{1i} \neq r_{2i}$$

which allows the bank to identify the cheater:

$$Cheater = x_{1i} \oplus x'_{2i} \text{ or } x_{2i} \oplus x'_{1i}$$

Two examples, ECashFile and ECashNet, are completely described in appendix 1 and appendix 2.

5. The e-cash package

This chapter gives some more precise information about the ecash package. It is meant to be studied simultaneously with the javadoc. The classes are actually shared out in two sub-packages: ecash.kernel and ecash.internet.

Package ecash.kernel

This package contains all the low-level classes used to implement the e-cash protocol.

Class	Short description
BankChoice	*Represents a number chosen by the bank, with n = [0..nbEnveloppe[.*
BuyerPurse	Contains all the notes and information used to build them.
BuyerPurseUnit	This class represents a single item of the BuyerPurse class.
Const	This class loads some parameters from a properties file, and make them available through corresponding static fields.
EnvelopesList	*The list of envelopes sended to the bank.*
FullNote	This class represents a single note with all it's related information: envelope, blinding factor, RIS, ...
FullNotesList	This class constructs a list of full notes.
KeyRing	This class represents a key ring, i.e. a public key and a private key.
Note	*Contains a bank note and its corresponding signed hash.*
PubKey	*Public key of the bank, used to sign the bank notes.*
RevealedNote	This class is used by buyers to reveal to the bank the secret information used to build a note.
RevealedNotesList	*This class, which represents a list of RevealedNote objects, is used by buyers to reveal at the bank the secret information used to build all the notes.*
RISPart	*The revealed part of the RIS that the buyer send to the seller in order to prove the validity of the note.*
SellerPurse	Contains all the notes and information obtained from buyers during transactions.
SellerPurseUnit	This class represents a single item of the SellerPurse class.
Tools	This class provides some tools.
UsedNote	This class represents a single item of the UsedNotesList class.
UsedNotesList	A list of all the notes already returned to the bank.

The classes written in *italic* represent objects that are sent during transactions, and they respect for that a particular protocol, made of two points:

String representation

To each object correspond what we call a String representation. A String representation is simply a single string containing all the object's values in an Indian file. It provides roughly the same possibilities than the standard java serialization process with two main advantages, security and efficiency: Sending objects can jeopardize the system robustness. For example, a cheater could send a sub-class of the asked object which modifies the security checks. We can of course avoid that by declaring all this classes as "final", but this could hamper further developments. The second advantage was efficiency; transferring only the values minimizes the number of exchanged bytes and therefore improves global performances. And a String can easily be transformed into an array of bytes, which is often the entry format for cipher algorithms.

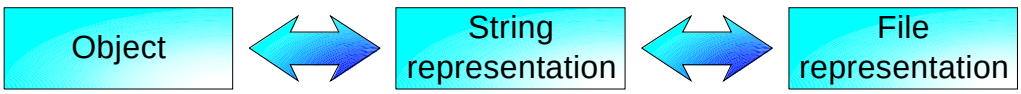
To achieve this goal each class redefines the toString() method, which writes all the values in an Indian file, and posses a constructor which reconstructs the initial object from its String representation.

File representation

Similarly to the String representation, each object owns a corresponding file representation. Such a file contains only one String: the String representation of the object. Because this file is written according to a standard, it makes it possible for another program, even written in a different language, to access to the exchanged information of the e-cash protocol. The ECashFile program allows a user to do electronic transactions over his e-mail (which can be encrypted using PGP, for example).

To achieve this goal each class defines a save() method, which transforms the object into its file representation and writes it in the specified file, and defines also a load() static method, which reconstructs the initial object from its file representation. This two methods actually only made a call to the Tools.writeFile() method and to the Tools.readFile() method respectively.

In an information point of view, there is clearly a fully equivalence between the object, its String representation and its file representation. We can consider the Sting representation as an intermediate step.



Another class that deserves to be mentioned here is the Const class, which loads some parameters from a properties file, and makes them available through corresponding static fields. The first point to notice is that a call to its load() static method is obligatory before any request of a particular constant. The best place for such an instruction is probably the constructor of the main user's program class. The default values are stored in a file called ECash.properties, so the complete syntax of the call is:

```
Const.load("ECash");
```

This file, which can be freely modified, contains initially the following values:

```
DIR=Data/
KEY_SIZE=1024
NB_ENVELOPE=10
NB_RIS=10
REGEXP=[0-9]+\\x2E?[0-9]*(\\x28[0-9]+\\x29)?
```

ECash.properties

DIR

DIR is the short name for directory. None of the package classes uses this value; it is meant to be used by final-user programs. It allows for example to specify where to store information, like it was done for the ECashFile program.

KEY_SIZE

The public and private keys are generated by the RSA program, which asked a key size between 512 and 2048 bits. The smaller the key the more efficient the algorithm, but remember that a big key provides more security.

NB_ENVELOPE

The number of envelopes sent to the bank. The bank chooses one of them and asks the client to reveal all information used to build the other ones. Without this precaution the client has two easy way of cheating:

The first consists in announcing a smaller amount (which will be deduced from the bank account) than it really is. When the seller will deposit this note at the bank, the latter will have to credit his account with the amount that is on the note, and not the amount announced by the cheater! And the second, worse, consists in giving a wrong RIS; this allows the client to re-use his note as many times he wants without being identified.

It is difficult to set the right value for this constant. If we choose 10, we get efficiency but perhaps we take some risks; there is only 90% chance that we detect an attempt of cheating. But on the other hand, who will dare cheating if there is only 10% chance of success? And unlike possible involuntary cheating (like given a wrong password), if someone cheats, then he is a hacker; there should be no chance to create fake notes with an end-user program! If we choose a big number, let's say 100, it is clear that the chances of cheating drop down, but at what cost of efficiency?

NB_RIS

The number of RIS pairs in each note. A small number implicates a short challenge (there are as many bits in the challenge as number of RIS pairs) from the seller, and a short challenge implicates more chance that two different sellers generate the same one. The problem occurs if the buyer tries to double-spend its note; if he has to answer twice the same challenge, the bank won't be able to identify him.

The chance of working a cheater's identity out increases exponentially with the number of RIS pairs. Remember that the algorithm allows finding out the cheater if only at least one bit is different in booth challenges. The probability is expressed by:

$$P = 1 - \frac{1}{2^n}$$

Therefore if $n = 10$ we have:

$$P_{10} = 1 - \frac{1}{2^{10}} \cong 0.999 = 99.9\%$$

REGEXP

REGEXP is the regular expression that describes precisely the syntax of a notes expression.

Definition of a notes expression

A term which is often used in this paper and that deserves to be defined is "notes expression". A notes expression is a way of representing a list of amounts using a particular and strict syntax. The grammar of this expression is given below:

```
notes_expression = item {"-" item}
item = number {"(" factor ")"}
number = digit {digit} {"."} {digit}
factor = digit {digit}
digit = "0" | "1" | "2 | "3 | "4" | "5" | "6" | "7" | "8" | "9"
```

Some examples:

- "20": A single note of 20.
- "20(5)": Five notes of 20.
- "0.5 - 30 - 40(3)": A note of 0.5, one of 30 and three of 40.

In java we use regular expressions to validate such grammars. In this case the expression is:

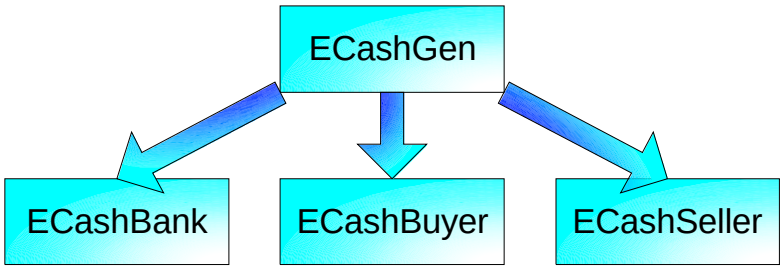
```
REGEXP = [0-9]+\\x2E?[0-9]*(\\x28[0-9]+\\x29)?
```

Package ecash.internet

This package contains specialized classes to run the e-cash protocol over Internet. It uses some kernel classes. The aim is to hide the different steps of the e-cash protocol, where as providing high level primitives. The different classes and *interfaces (in italic)* are:

Class / *interface*	Short description
BuyReport	Extension of the Report class which adds the account number.
CreditDebitReport	Such a report is returned to a client when he buys or deposits some money at the bank.
DepositReport	Extension of the Report class which adds the account number and also the account number of the cheater, in the case of a buyer re-using a note twice.
ECashBank	Extension of the ECashGen class which adds specific methods for a bank.
ECashBankListener	*This interface must be implemented by every class that creates an ECashBank object.*
ECashBankState	Represents the state of an ECashBank object.
ECashBuyer	Extension of the ECashGen class which adds specific methods for a buyer.
ECashBuyerListener	*This interface must be implemented by every class that creates an ECashBuyer object.*
ECashBuyerState	Represents the state of an ECashBuyer object.
ECashGen	Generic class for ECashBank, ECashBuyer and ECashSeller.
ECashSeller	Extension of the ECashGen class which adds specific methods for a seller.
ECashSellerListener	*This interface must be implemented by every class that creates an ECashSeller object.*
ECashSellerState	Represents the state of an ECashSeller object.
PayReport	Extension of the Report class which adds the nickname of the buyer, in order to allow the seller to identify the origin of the payment.
Report	A Report indicates how an operation succeeded, from the server point of view.

The e-cash system defines three actors: the bank, the buyer and the seller. In a real-life case the buyer and the seller could be implemented by the same client, so that a person can use his e-purse as well for buying thinks and cash money from other users (like we do it with conventional notes and coins). But it is easier here to study each case separately. We have one specific main class for each actor: ECashBank for the bank, ECashBuyer for the buyer and ECashSeller for the seller. Because they have several properties in common they are all extension of a more generic class, ECashGen.



The bank

During transactions with buyers or sellers, the bank behaves always as a server. We call server the partner who waits for connections and client the program who sets up these connections. The term client can be ambiguous because it designs as well a connection partner, an end-user program and a physical person who owns a bank account. In client-server talks it is always the first definition that counts.

A bank program creates an instance of the ECashBank class. Its single constructor requests an ECashBankListener as a parameter. The class which implements this interface (ECashBankListener) must define the following methods:

- `public boolean checkPassword(String accountNumber, String password);`
  which returns true if the password correspond with the account number, false otherwise.

- `public String getAmount(String accountNumber);`
  which returns the available amount in the specified bank account.

- `public void report(DepositReport depositReport);`
  which is called after a client deposited some money on his bank account.

- `public void report(BuyReport buyReport);`
  which is called after a client withdrew some money from his bank account.

The bank makes then its services available by calling the startServer() method. Note that the corresponding stopServer() method is defined in the super-class ECashGen. The bank is able to answer only one request at the time. The waiting port number has been arbitrary chosen and equals 10000.

When a client makes a connection, the transaction is divided into three steps:

Handshake

The handshake creates a secured secret channel between the client and the server. The transmissions are secured by symmetric cryptography, and its secret key is transmitted thanks to asymmetric cryptography.

The client starts with creating a pair of keys using the RSA algorithm and sends the public key to the server (bank). The size of this key is a fixed parameter in the ECashGen class and must not be mistaken with the size of the public key used by the bank to sign notes.

The server (bank) creates the shared secret key that will be used to encrypt and decrypt any further messages during the whole transaction. The symmetric cryptography algorithm is BlowFish, one of the most common symmetric block ciphers implemented in Java. Like for the client, the size of the key is a fixed parameter in the ECashGen class.

The server then sends the secret key encoded with the public key of the client. They now both possess the secret key, so any further transmissions are encrypted. For more security, these keys are all freshly regenerated for each transaction. It means that even if a hacker

can corrupt a secret key and obtain all the information of the transaction, at least it will be useless for further connections.

Choice of the operation

The client sends to the server a special message describing the wanted service:

- ECashGen.OP_GET_KEY: The client (buyer or seller) wants a copy of its public key.
- ECashGen.OP_BUY: The client (buyer) wants to buy some notes.
- ECashGen.OP_DEPOSIT: The client (seller) wants to deposit some notes.

Operation

Client and server exchange information. Timely (typically after each note) the server sends the ECashGen.TRANS_OK code that acknowledges the last transmitted messages. At the end of a buy or a deposit operation, a report is created and returned to the end-user program. This is done through a call to the report() methods, specified in the ECashBankListener interface.

Thanks to the reports, the program is informed each time that a transaction modifies any bank account. Therefore, for a maximum security, a backup of the bank state can be done at this point. The getState() method returns a ECashBankState object which contains the key-ring and the list of all used notes. This object can be added to other ones, like the list of all the bank accounts, and stored on the hard-disk. Next time the program is loaded, the saved state can be restored using the setState() method.

The buyer

A buyer behaves always as a client (in the client-server concept). He can make connections either to the bank, in order to get its public key or to buy notes, or to a seller, in order to send him some notes. Connections to the bank will be done on port 10000 where as connections to the seller will be done on port 10001. These two values have been arbitrary chosen and are defined in the ECashGen class.

A buyer uses the ECashBuyer class. Its constructor requests as a parameter a class implementing the ECashBuyerListener interface, which defines the following method:

- `public void buyerPurseChanged();`
  which is called after a buy or a pay operation.

Like described for the bank, the client agrees with the server on a secret key to exchange data (handshake), sends a message that identify the wanted operation and achieve the transaction. Three operations are possible:

- ECashGen.OP_GET_KEY: The buyer wants a copy of the public key of the bank.
- ECashGen.OP_BUY: The buyer wants to buy some notes.
- ECashGen.OP_PAY: The buyer wants to send some notes to a seller.

After a buy or a pay transaction the buyerPurseChanged() method is called in the listener class, which is a good opportunity to backup the current state of the buyer. The getState() method returns an ECashBuyerState object which contains the public key of the bank and the buyer's purse (containing all the notes and related information). Next time the program is loaded, the saved state can be restored using the setState() method.

The seller

A seller can be, in the client-server concept, either a client or a server. With the bank the seller behaves as a client, and with a buyer he behaves as a server.

A seller uses the ECashSeller class. Its constructor requests as a parameter a class implementing the ECashSellerListener interface, which defines the following methods:

- `public void sellerPurseChanged();`
  which is called after a pay or deposit operation.
- `public void report(PayReport payReport);`
  which is called after a client sent some notes.

How a transaction works has already been described for the bank and the buyer and won't be discussed here. Three operations are possible:

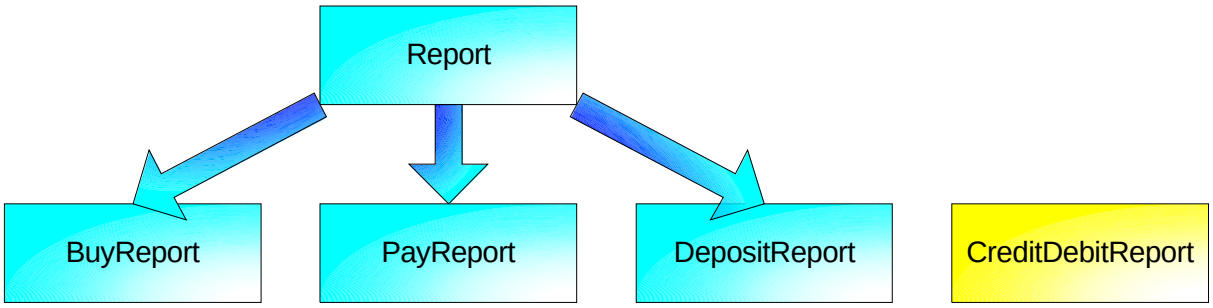
- ECashGen.OP_GET_KEY: The seller wants a copy of the public key of the bank.
- ECashGen.OP_PAY: The seller gets some note from a buyer.
- ECashGen.OP_DEPOSIT: The seller wants to deposit some notes at the bank.

After a pay or a deposit transaction, the sellerPurseChanged() method is called in the listener class, which is a good opportunity to backup the current state of the seller. The getState() method returns an ECashSellerState object which contains the public key of the bank and the seller's purse (containing all the notes and information given by the buyer during a pay operation). Next time the program is loaded, the saved state can be restored using the setState() method.

Reports

After every transaction, each partner gets a report which informs of the successfulness of the last one. For example, after a buy operation, the bank gets a report which indicates if the buyer tried to cheat, and if yes what kind of misuse was done (wrong password, reusing notes, ...). If no cheating was detected, the report indicates how much must be debited from the client's bank account.

There are two kinds of reports, client reports and server reports. All the client reports are instances of the same class, CreditDebitReport, where as server reports are subclasses of the Report class.



The only transaction that doesn't involve reports is the ECashGen.OP_GET_KEY, because there is no way of cheating. There is no point for a bank to give a fake public key! The other transactions are described here:

Buy operation

A buy operation, identified by the ECashGen.OP_BUY code, is when a buyer asks the bank to send him some notes. The buyer, who acts as a client, calls the ECashBuyer.buy() method and gives as parameters the bank address, his bank account and his password, and a notes expression to specify which amounts he needs. This method returns a CreditDebitReport object, which informs of the current amount still available on the bank account, and a code indicating if the transaction was successful. The code can be:

- ECashGen.OK: Indicates that the transaction finished successfully, that the server detected no cheating attempt.
- ECashGen.NOTES_EXP_SYNTAX_ERROR: The notes expression contains one or more syntax errors.
- ECashGen.SERVER_NOT_READY: The server is not able to answer client's request. For a bank the most probable reason is that another client is already using the connection.
- ECashGen.CHEAT_WRONG_SIGNATURE: After un-blinding the fresh note, the client checks that the signed note corresponds to the unsigned one. It is more meant to allow a client verifying that his copy of public key is still valid than to be doubtful of the honesty of the bank.

- ECashGen.CONNECTION_CLOSED_BY_SERVER: Indicates that the connection was closed by the server during the transaction. This can happen either because of a technical problem, or because the client cheats. The term "cheat" covers involuntary cheating, like providing a wrong password or a wrong account number, or voluntary cheating, like having different amounts in the revealed notes list. The most likely reason for such a code to appear is when a client enters a wrong password.

In a buy operation the bank acts as a server. The main program is informed of the transaction via a BuyReport, which indicates the account number of the buyer, the amount of all the notes, and a code indicating how the operation happened. The code can be:

- ECashGen.OK: Indicates that the transaction finished successfully and that no attempt of cheating has been detected.

- ECashGen.CHEAT_WRONG_PASSWORD: The client gave a wrong password. In fact this code just verifies that the password corresponds to the bank account, so this same code is used if the client gives a wrong account number.

- ECashGen.CHEAT_AMOUNTS_NOT_ALL_THE_SAME: This is clearly a voluntary attempt of cheating, indicating that the different amounts in the revealed notes list are not all the same. Because it is not possible to produce such a case with the provided classes, it is clearly a hacker who tries to make the bank sign a note with a bigger amount than expected.

- ECashGen.CHEAT_WRONG_REVEALED_NOTES: Similarly to the previous code, this attempt of cheating indicates that the revealed information is not the one used to build the notes sent in different envelopes.

In addition to these values, the Report class defines also two extra methods:

- `public String computeSum()`
  which returns the sum of all the bought notes.

- `public String computeNotesExp()`
  which creates and return a notes expression of the different amounts.

Pay operation

A pay operation, identified by the ECashGen.OP_PAY code, is when a buyer sends some notes to a seller. The buyer, who acts as a client, calls the ECashBuyer.pay() method and gives as parameters the seller address, a nickname to help the seller to identify the origin of the payment, and a notes expression indicating the notes to send . This method returns just an integer (which acts as a report) indicating if the operation was successful. The code can be:

- ECashGen.OK: Indicates that the transaction finished successfully, that the server detected no cheating attempt.

- ECashGen.NOTES_EXP_SYNTAX_ERROR: The notes expression contains one or more syntax errors.

- ECashGen.SERVER_NOT_READY: The server is not able to answer client's request. This can happen either because the seller hasn't started his server, or because another client is already using the connection.

- ECashGen.NOTES_NOT_AVAILABLE: Indicates that the amounts specified in the notes expression are not available in the buyer's purse.

- ECashGen.CONNECTION_CLOSED_BY_SERVER: Indicates that the connection was closed by the server during the transaction. This can happen either because of a technical problem, or because the client cheats.

In a pay operation the seller acts as a server. The main program is informed of the transaction via a PayReport, which indicates the nickname of the buyer, the amount of all the notes, and a code indicating how the operation happened. The code can be:

- ECashGen.OK: Indicates that the transaction finished successfully and that no attempt of cheating has been detected.

- ECashGen.CHEAT_WRONG_SIGNATURE: The signature of the note is not valid. This can happen either because the seller has not an up-to-date copy of his public key, or because the buyer cheats; he made and signed the notes himself.

- ECashGen.CHEAT_WRONG_RIS: This is clearly a voluntary attempt of cheating, indicating that the buyer doesn't reveal his real RIS.

Remember that in addition to these values, the Report class defines also two extra methods, computeSum() and computeNotesExp(), as described previously.

Deposit operation

A deposit operation, identified by the ECashGen.OP_DEPOSIT code, is when a seller deposits some notes on his bank account. The seller, who acts as a client, calls the ECashBuyer.deposit() method and gives as parameters the bank address, his bank account and his password, and a notes expression to specify which notes he wants to deposit. This method returns a CreditDebitReport object, which informs of the current amount still available on the bank account, and a code indicating if the operation was successful. The code can be:

- ECashGen.OK: Indicates that the transaction finished successfully, that the server detected no cheating attempt.

- ECashGen.NOTES_EXP_SYNTAX_ERROR: The notes expression contains one or more syntax errors.

- ECashGen.SERVER_NOT_READY: The server is not able to answer client's request. For a bank the most probable reason is that another client is already using the connection.

- ECashGen.NOTES_NOT_AVAILABLE: Indicates that the amounts specified in the notes expression are not available in the seller's purse.

- ECashGen.CONNECTION_CLOSED_BY_SERVER: Indicates that the connection was closed by the server during the transaction. This can happen either because of a technical problem, or because the client cheats. The term "cheat" covers involuntary cheating, like providing a wrong password or a wrong account number, or voluntary cheating, like depositing twice the same note. The most likely reason for such a code to appear in a deposit operation is when a client enters a wrong password.

In a deposit operation the bank acts as a server. The main program is informed of the transaction via a DepositReport, which indicates the account number of the seller, the account number of the buyer if the latter double-spend notes, the amount of all the notes, and a code indicating how the operation happened. The code can be:

- ECashGen.OK: Indicates that the transaction finished successfully and that no attempt of cheating has been detected.

- ECashGen.CHEAT_WRONG_PASSWORD: The client gave a wrong password. In fact this code just verifies that the password corresponds to the bank account, so this same code is used if the client gives a wrong account number.

- ECashGen.CHEAT_WRONG_SIGNATURE: The signature of the note is not valid. This can happen either because the seller has not an up-to-date copy of his public key, or because he cheats; he made and signed the note himself.

- ECashGen.CHEAT_WRONG_RIS: The RIS is not correct. One reason could of course be that the seller hasn't verified the RIS of the buyer during the pay operation (and the buyer cheats...) but the seller is still responsible; the protocol imposes that verification. Therefore such a case cannot happen with the provided classes if no hacking is done.

- ECashGen.CHEAT_NOTE_REUSED_BY_SELLER: The seller cheats by trying to deposit the same note for the second time.

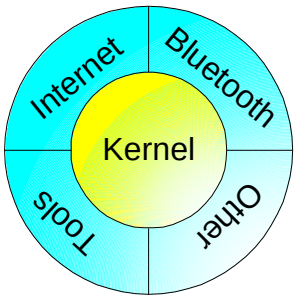
- ECashGen.CHEAT_NOTE_REUSED_BY_BUYER: A note has been double-spent by a buyer. The bank got twice the same note but with different valid RIS; the cheater's bank account number has been computed and is available in the report.

Remember that in addition to these values, the Report class defines also two extra methods, computeSum() and computeNotesExp(), as described previously.

Additional packages

We described the content of two packages, ecash.kernel and ecash.internet. The first regroups all the low-level classes that implement the e-cash protocol. The second hides the different steps of the e-cash protocol, where as providing high level primitives to run the e-cash over the Internet. It can therefore be seen as an extension of the kernel classes.

The aim of such a structure is to provide the possibility to create easily new packages. For example one called ecash.bluetooth that allows portable devices to work as e-purses. Or another called ecash.tools which provides some utilities like a class encrypting e-purses stored on hard-disks.



6. JCash and JBank

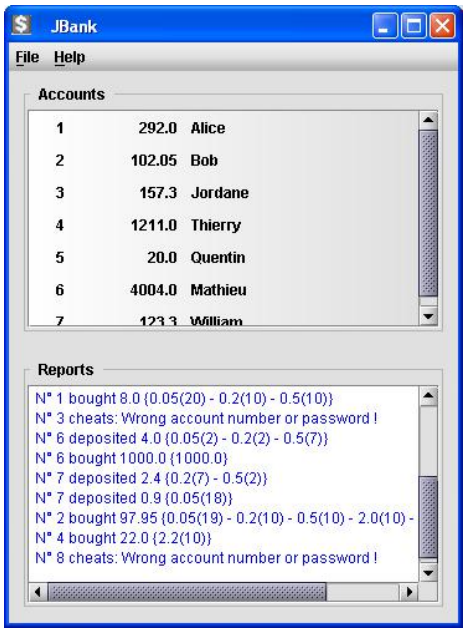
JCash is a program which works as an e-purse; it behaves simultaneously as a buyer and as a seller. It runs over the Internet and uses therefore the ecash.internet package (in addition to the ecash.kernel one). It can be seen as an improvement of the ECashNet example and gives an idea what can be done with the e-cash protocol.

JBank is a separate program which acts as a bank for the JCash e-purses. Like JCash, it uses exactly the same protocol than the ECashNet classes. It means that you can mix these two projects. This program is essential to test fully JCash; it will therefore be presented first.

These two programs are supposed to be a demonstration and not an academic example for understanding e-money. That's why this chapter in build like a manual and why the programs are given as jar files.

JBank

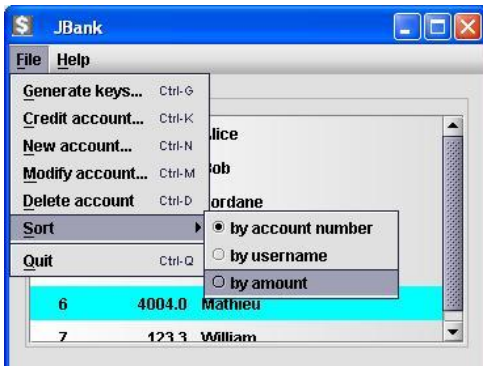
The program consists in an executable jar file and in a directory, called System. The latter contains initially only two icons. The .png is a small image (16 x 16) that is displayed at the top-left corner in windows. The .bmp is a bigger image (32 x 32) that can be placed as an icon on the desktop, if executed under the Microsoft Windows system. The third file, called BankState.obj, will appear later. It is a backup of the bank accounts, the key-ring, and the list of all the used notes. This file is up-to-dated after each transaction. The user interface looks like this:



The top window shows users' accounts. Each account is made of an account number, the current amount and the name of the owner. The last value, the password, is not displayed. Note that the name is facultative. It is used to find quickly an account (if you don't remember the number) or as a witness; for example if someone shows up physically at the bank and wants to credit his bank account, the name is a confirmation that it is the right account which is credited.

The bottom window shows the most important information collected during transactions. It indicates how accounts are credited or debited, and prints what misuse has been detected in case of cheating.

The functionalities of this program can be accessed via the File menu:



Some of them ("Credit account", "Modify account" and "Delete account") are accessible only if a bank account is currently selected.

Generate keys

Generates a new pair of keys. In a real-life case, this command should be used only once, and certainly not appear as the first item in a menu. Changing the public key invalidates all the notes that are currently stored in e-purses. And all the clients will have to connect to the bank in order to get a fresh copy of the new key.

Credit account

Credits or debits an account. A dialog box appears and asks the user to enter the amount of money that must be added to or subtract from (amount starting with the sign "-") the current available amount. This functionality can also be accessed by double-clicking over an account.

New account

Creates a new account. The user enters the name, the password and secondarily the current amount, if not 0. The account number is decided by the bank and is each time incremented by one, even if a smaller number is available (after deleting an account for example). The reason is that the account number is the primary key of every record. Therefore if a new

client obtains an account that belonged to an unmasked cheater, he can be mistaken for him. Example: Oscar, who owns the account number 13, double-spend a note and shortly after asks the bank to remove him from the clients list. Douglas, a new client, obtains the first free number, 13. But when the second seller deposits the double-spent note, the program will accuse the owner of account 13 of cheating...

Modify account

Modifies an account. This action shows all the fields of the selected account and allows the user to modify the content of them. This function can also be used to remind a client of his password.

Delete account

Deletes an account. This action deletes the current selected account. A confirmation is asked to the user to avoid an accidental erasing.

Sort

Sorts the accounts. By default the accounts are sorted by account numbers. The user can modify this option by clicking another item in the sub-menu. Sorting clients by their name helps the banker to find out a particular account and sorting them by amounts regroups all the people that are overdrawn.

Quit

Quits the program. Because the current state of the bank is saved after each operation, all the information will be restored at the next execution. Warning: The current state consists only on the bank accounts, the key-ring, and the list of all the used notes; therefore the messages printed in the reports box are not preserved.

In the accounts box, a popup menu gives a quick access to the main functionalities.

JCash

The program consists in an executable jar file and in two directories, System and ID. Like for the JBank program, the System directory contains two icons (one for the program and one for the desktop, if run under Microsoft Windows) and a file called ECashState.obj, containing the purses (new notes and notes cashed from other users) and the public key of the bank. In addition to that we find also a file called JCash.properties, recording user's inputs, a file called Contacts.txt, containing the user's address book, and a file called Help.txt, containing the text that the program shows when the user selects the Help tab.

The ID directory has to be filled by the user. When paying a seller, the program sends a string that identifies the origin of the payment. Depending how the seller is, the buyer will give

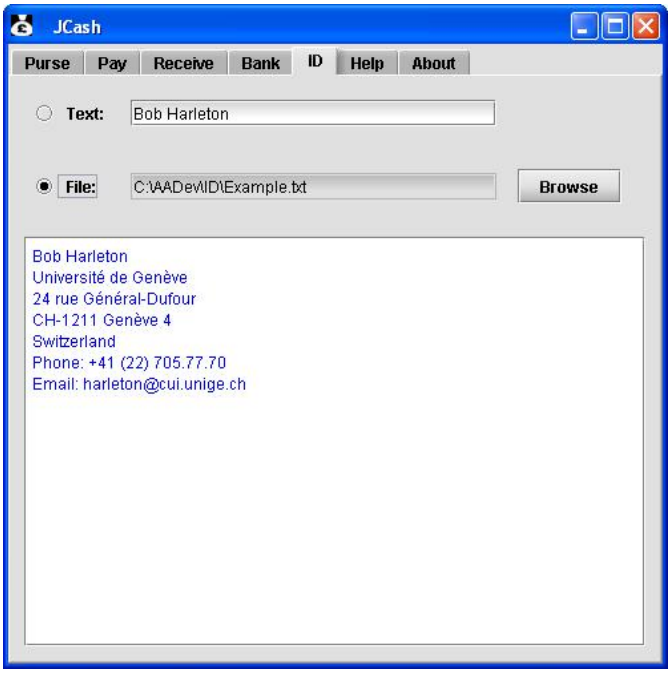
more or less information, from given the full name and address to just a nickname. This directory contains identity files that the user will choose before a payment.

When the user launches the program for the first time (respectively when the ECashState.obj file is absent), a dialog box appears and requests the address of the bank in order to get a copy of the public key:

Once done, the main program starts and shows its different tabs. Two colours appear frequently in the user interface: green and red. The green colour represents fresh notes, which a buyer can send to a seller. And the red colour represents used notes, which must be deposited at the bank. The different tabs of the program are presented here in the order that a new user would discover the functionalities of the program.

ID tab

This tab allows you to choose the identity you want to transmit to a buyer in order to identify the origin of a payment.

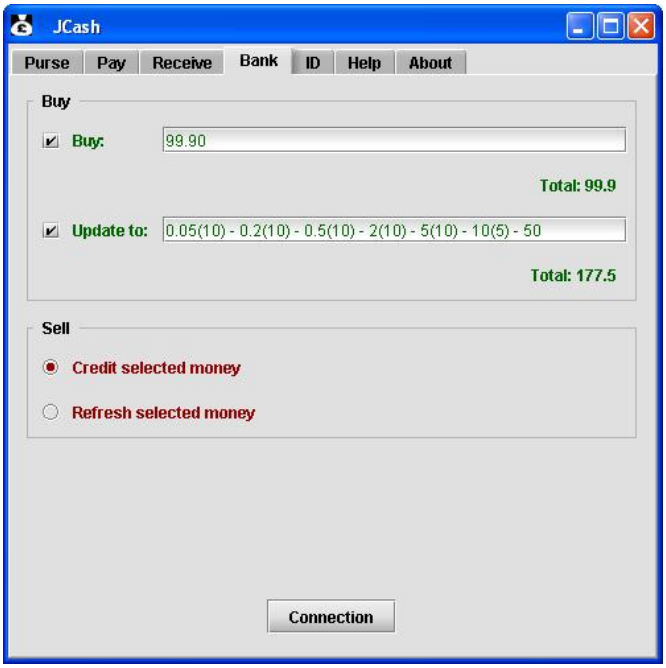


The string that will be sent to the seller is the one that appears in the bottom box. You can either enter the text or choose a file in the ID directory. Note that the box in not editable; to

create a new identity file, a text editor must be used. For security reasons, it is the text checkbox that is selected by default each time the program is restarted.

Bank tab

The bank tab is used to make a connection to the bank, in order to buy notes, to deposit notes, or to check the current available amount in the bank account.



The "Buy" field contains a notes expression that specifies which amounts the user wants to acquire.

Reminder:

A notes expression is a way of representing a list of amounts using a particular and strict syntax. The grammar of this expression is given below:

```
notes_expression = item {"-" item}
item = number {"(" factor ")"}
number = digit {digit} {"."} {digit}
factor = digit {digit}
digit = "0" | "1" | "2 | "3 | "4" | "5" | "6" | "7" | "8" | "9"
```

Some examples:

- "20": A single note of 20.
- "20(5)": Five notes of 20.
- "0.5 - 30 - 40(3)": A note of 0.5, one of 30 and three of 40.

If the checkbox is selected and the syntax of the expression is correct, the sum is computed and displayed.

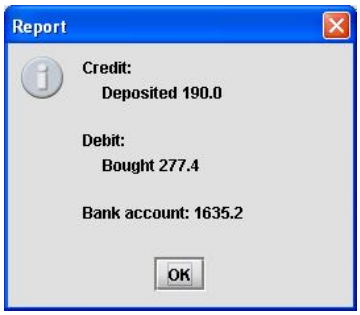
The "Update to" field contains a notes expression that represents which denominations the buyer wants to have at least in his purse. Only the missing notes will be bought. Like for the previous filed, the sum is displayed as soon as the checkbox is selected and the syntax of the expression is correct.

For the used notes, the client can choose between crediting the selected notes and refreshing them. If the second choice is selected, the notes will be first credited and then added to the ones specified in the "Buy" field.

The "Connection" button will take into account theses values and print the following dialog:

The client can complete the fields and check that the summary at the bottom is correct.

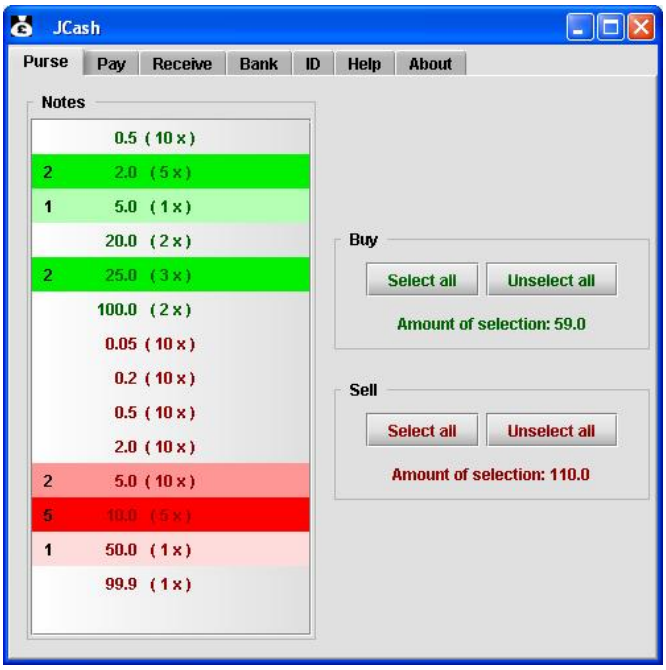
If the transaction works properly, the client gets a report which informs how much money has been debited, credited and how much money is still on the bank account.



A user who just wants to know his current available money on his bank account can unselect all the used notes (Purse tab), unselect the fields "Buy" and "Update to", and make the connection.

Purse tab

This tab shows all the notes that are in the e-purse. A left-click on a note will select it, and a right-click will unselect it (shift + left-click = right-click). Each line contains a number written in black (visible only when different from 0), indicating how many notes of this amount are selected, the amount of the note, and finally the number of notes with that denomination.

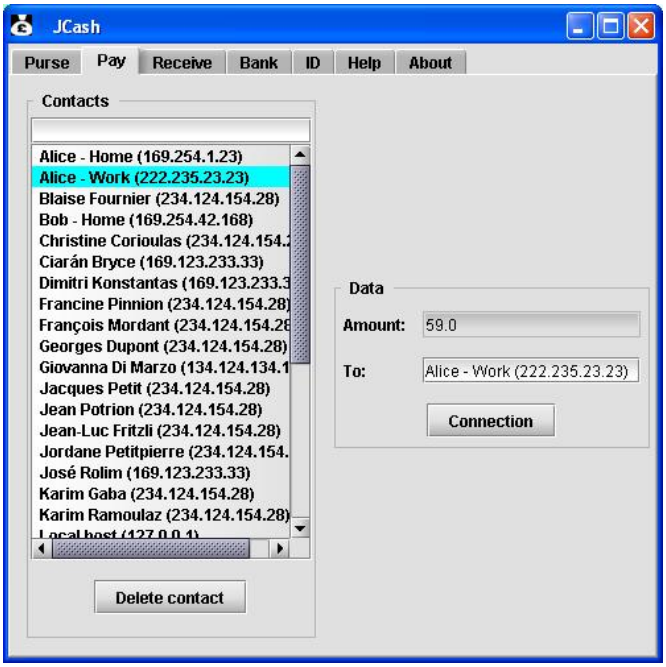


The highlighting colours are visual hint:

- Light green or light red: selected once.
- Green or red: selected twice.
- Dark green or dark red: selected three times or more.

Pay tab

Before paying a seller, the user will have to select in the Purse tab the notes he wants to send. Of course, only green notes can be taken into account. When clicking on the "Pay" tab, the selected amount is displayed in the "Amount" field.

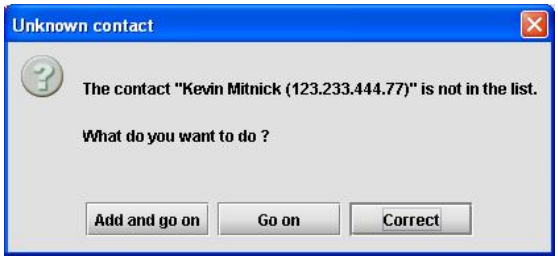


The client can now directly click on a person in the contact list or type his name in the text field. If a name is unknown, the list will become red. The enter key will copy the selected line in the Data box.

To enter a new contact, use the following procedure:

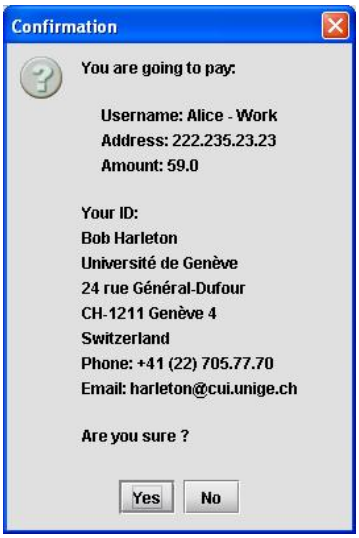
1. Type the name or a nickname followed by the IP address within parenthesis.

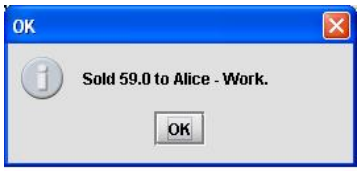
2. Press return. The following dialog box appears:



- "Add and go on" records this new contact in the list and writes it in the Data box.
- "Go on" just writes this contact in the Data box, without recording it in the list.
- "Correct" corrects the entry.

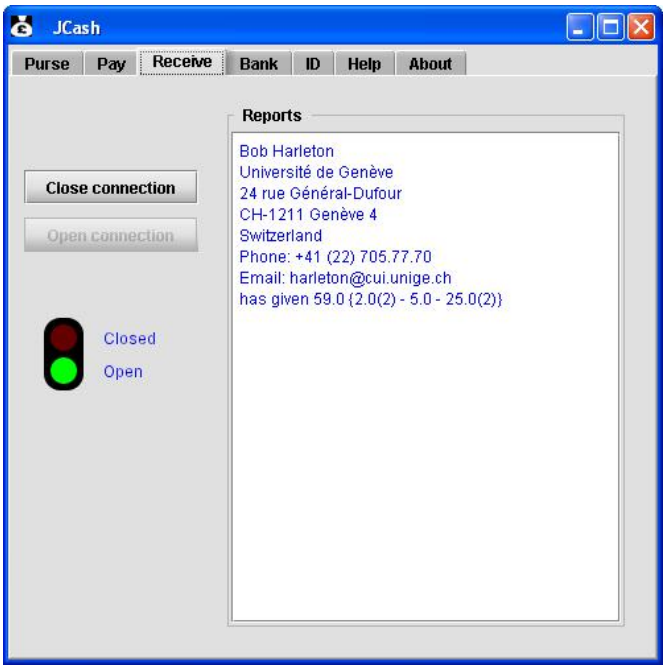
The "Connection" button checks that the amount is different from 0 and that the syntax of the addressee is correct. If everything is ok, a confirmation dialog box prints the information related to the payment and remind the user of which ID he chose to transmit. If the payment finishes successfully, a dialog box displays the amount and the creditor.





Receive tab

This tab must be active in order to cash a payment. Click on the "Open connection" button or directly on the lights to start the server. Note that for security reasons, changing the current tab will automatically stop the server. All the incoming information will be written in the Report box.



7. Conclusion

This paper presented electronic money and compared it to other payment schemes, including conventional paper banknotes and silver coins, credit cards, and the Swiss CASH system. The main advantages that have been discussed are anonymity, peer-to-peer and the online shopping:

Anonymity

People are used to feel free when they spend their money. They don't want that each of their purchases is recorded and linked to them. Conventional money offers this advantage, but don't allow buying over Internet. Credit cards allow the online buying, but the seller obtains some information about you (including your credit card number) during the transaction. With electronic money you stay anonymous even if you do your shopping over Internet.

Peer-to-peer

A lot of merchants accept your credit card. But they need a card reader and also to be recorded in all the credit cards companies they want to deal with. Therefore, client-to-client or respectively peer-to-peer payments are not possible. You are either a buyer or a seller, but not both at the same time. The Swiss CASH system suffers from the same disadvantage: it is impossible to transfer some money to another card-holder directly. E-cash solves this problem and is like conventional money; each client can act as a buyer and as a seller as well.

Online shopping

Physical coins and notes are accepted only in physical shops... We saw a lot of similarities between conventional money and e-cash. We noticed of course some practical aspects provided by electronic money, like the fact that even much money doesn't ruin the shape of your pockets, but the main difference is the fact that this new kind of money can be used over Internet. And once it works over Internet, the system just need some adaptations to work over Bluetooth or either SMS/MMS, for example.

But we saw also that it is difficult to suggest a new system. Even if you can convince potential users that the technology will suit all theirs needs, more important is to convince them that they won't be alone to use it.

This paper presented a complete e-purse implementation. Even if a particular attention has been brought to its user interface, it must be considered as a demonstration and not as a product ready to be launched on the market. The first reason is that only the securities issues related to the e-cash protocol has been studied, implemented and tested. For example, we could expect that the public key of the bank comes accompanied with a digital certificate, in

order to be sure that it is really the bank we are dealing with. Or we could expect that the file containing all the notes is encrypted on the hard-disk. The second reason is that this entire work consists in a basis for future developments. Among them two projects have already been discussed, Bluetooth and Trust:

Bluetooth

Starting you laptop each time you want to buy a coffee or a bus ticket in not very convenient. An e-purse must be either extremely small and flat (like a credit card) or to be included in a device that the client already owns and carries with him. Even if the system presented in this paper works well for online shopping (Internet), we wouldn't be able to convince a lot of potential clients as long as the system is heavier than a wallet containing five pieces of each denomination...

More and more people are carrying a mobile phone or a PDA with them. It is therefore a good target to add an e-purse as a new functionality. Bluetooth is a protocol that allows devices (mobile or not) to build networks and communicate over radio frequencies, and which is more and more present in mobile devices.

Actually we have two packages. The first is called ecash.kernel and contains all the classes used to run the e-cash protocol. The second is called ecash.internet and adds the necessary functionalities to run the all over the Internet. A new package, called ecash.bluetooth, will allows you to pay your coffee anonymously with your mobile phone or PDA.

Trust

Authentication in electronic payment systems is almost always based on a username and a password. As soon as you know someone's password you obtain exactly the same privileges than him. This approach is simple but don't reflect how it works in the real-life. For example imagine that a very good friend of yours comes at your home and asks you to lend him a $10 note. You are probably going to accept. But now it is a guy you never met that rings your bell and asks you the same $10. You will probably be much more suspicious, even if he shows you his ID card. In real-life we have an extra parameter during an authentication process: the trust. You are going to lend money to someone you trust much easily than to someone you don't know, even if you have never seen any ID card of your friend (and therefore you are not absolutely sure that his real name is the one he gave to you!).

Trust is build, like in the real-life, from experience and computation. You trust your friend because he has always refunded you till now (experience). If now the same friend explained you that the unknown guys who wanted $10 was actually a member of his family, then you trust for this new person will be built from the description you will ear from you friend (computation). And this trust will of course change with the time, according to the experiences you will have with this new guy.

With electronic purses trust relations can be built quickly. Suppose that the e-purse of Bob records after each transaction how it happened. For example he recorded that Charlie tried twice to give him a note with a wrong signature. He recorded also that the coffee machine accepts the payment before checking if coffee is still available. When he meets Alice, his best friend, all the observations made by Bob will be transferred to Alice's purse. And

because Alice set up a high trust level in Bob, she will avoid dealing with Charlie and perhaps choices only cold drinks!

In the same manner that gold coins were replaced by paper banknotes and silver coins, we can expect that one day electronic money will be the standard way of paying. So the question is perhaps not "if" but "when".

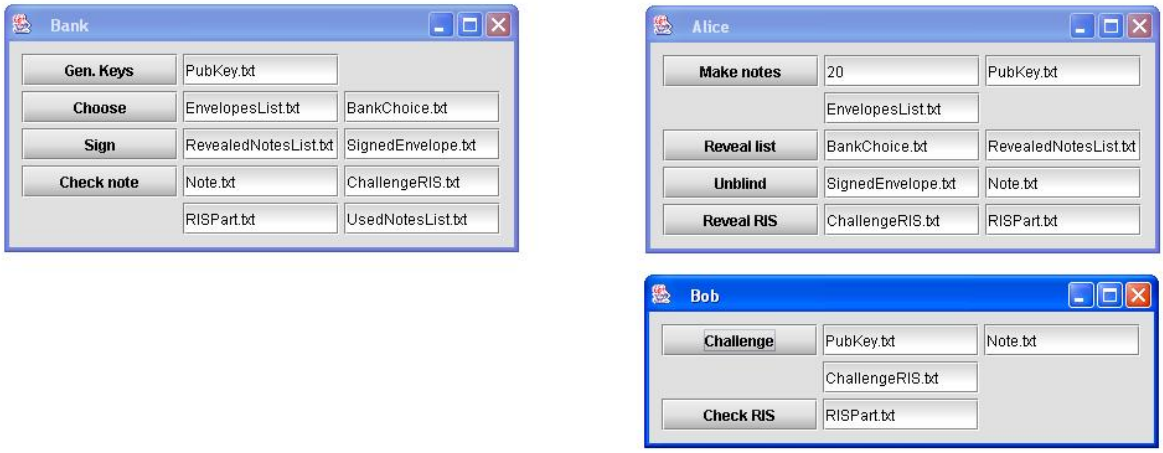
Appendix 1: ECashFile

ECashFile is the name of the first program used to test the ecash.kernel package. The scenario consists in three sets of transactions: Alice buys a note at the bank, sends it to Bob, and the latter deposits it at the bank again. It involves only classes that are part of the ecash.kernel package. Every exchanged object (public key, note, challenge, ...) between participants (bank, buyer or seller) is written in a file on the hard-disk. So it makes it possible to do financial transactions over e-mail. But the main purpose of this program is to help the reader to understand fully the e-cash process by analysing step by step what happen. Every exchanged object is written only as a sequence of strings, and each string represents a BigInteger (unlimited size integer). This allows the user to modify easily the values afterward and to check that any attempt of cheating is correctly detected.

1. Copy the ecash directory and the files of this example in your working directory and compile them.

2. Execute either the ECashFile class (java -classpath .;kunststoff.jar ECashFile), which launches the tree actors in a single console, or separately the Bank (java -classpath .;kunststoff.jar Bank), Alice (java -classpath .;kunststoff.jar Alice), and Bob (java -classpath .;kunststoff.jar Alice).

   Your screen should look like this:



   The buttons correspond to the different command while the text fields correspond to the parameters, mostly filenames. A tool tip help appears when the mouse stops over one of them. All the data files will be written in a directory called Data.

3. Gen. Keys The bank generates a new pair of keys and writes a file called "PubKey.txt". The value *e* is the public exponent and *n* is the modulus.

e
n

PubKey.txt

4. Make notes Alice generates several notes and, thanks to the public key of the bank, puts them in different envelopes.

envelope_1
envelope_2
...

EnvelopesList.txt

5. Choose The bank records the envelopes and choose randomly a number to designate which envelope she is going to sign.

choice

BankChoice.txt

6. Reveal list Alice reveals all information used to build the notes (blinding factors and RIS) except for the one chosen by the bank.

amount k_1 x_{11} x'_{11} x_{12} x'_{12} ... x_{1p} x'_{1p}
amount k_2 x_{21} x'_{21} x_{22} x'_{22} ... x_{2p} x'_{2p}
...

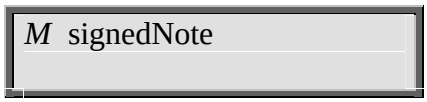
RevealedNotesList.txt

7. Sign The Bank checks that all information given by Alice is correct and signs the chosen envelope.

signed_envelope

SignedEnvelope.txt

8. Unblind Alice uses its blinding factor in order to unblind the envelope and obtain a signed note. The note is made of two parts: *M* and its corresponding signed hash, signedNote. The note is sent to Bob.

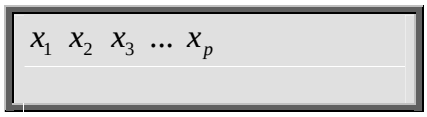


Note.txt

9. Challenge Bob records the note given by Alice and generates the challenge. The value of each bit of this number designate which side of every RIS-pair must be revealed.

ChallengeRIS.txt

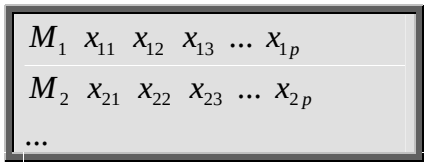
10. Reveal RIS Alice reveals the asked part of her RIS



RISPart.txt

11. Check RIS Bob checks the information given by Alice and accepts the note if everything is all right. This note can now be deposited at the bank.

12. Check note The bank verifies the signature of the note and if the latter has already been deposited. If so, the bank identifies the cheater; same RIS: Bob cheats; different RIS: Alice cheats. An accepted note is credited to the seller account and added in the UsedNotesList.txt file.



UsedNotesList.txt

The current state of each participant is stored after each step, which means that the previous process can be stopped and resumed at any time. To clean everything and restart the all freshly, delete the content of the Data directory and the following files: BankState.obj, AliceState.obj and BobState.obj.

The default filenames can obviously be changed in order to manage several notes. And the name of the default directory (Data) can be changed in the e-cash properties file: ecash/kernel/ECash.properties.

Appendix 2: ECashNet

This example runs the e-cash protocol over Internet. Each set of transmissions is made during a single transaction, so the former 10 steps (ECashFile) are reduced to 3: buying notes, selling notes and depositing notes. This program uses classes belonging to the ecash.internet package, which runs some kernel classes. The aim is to hide the different steps of the e-cash protocol, where as providing high level primitives.

1. Copy the ecash directory and the files of this example in your working directory and compile them.

2. Execute either the ECashNet class (java -classpath .;kunststoff.jar ECashNet), which launches the tree actors in a single console, or separately the Bank (java -classpath .;kunststoff.jar Bank), the buyer (java -classpath .;kunststoff.jar Buyer), and the seller (java -classpath .;kunststoff.jar Seller).

   Your screen should look like this:

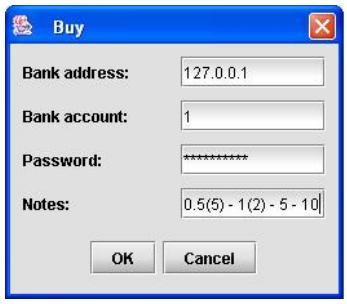
   In the example described here, the three frames were launch via the ECashNet class. To test fully the possibilities of this program, the different actors can be executed in different consoles on different computers, and of course nothing prevents you from launching several banks, several buyers and several sellers.

3. Gen. keys The bank generates a pair of keys.

4. Create account The bank creates two accounts: (1, password_1) and (2, password_2).

5. Get key The buyer and the seller connect to the bank in order to get a copy of its public key.

6. Buy The buyer chooses some notes and buys them at the bank.

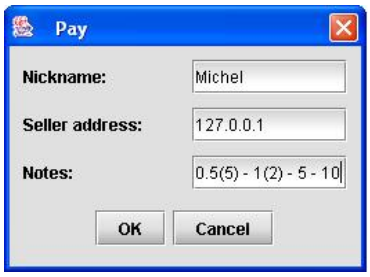


A notes expression is used to select the different amounts. In our current case, the buyer asks to buy 5 notes of 0.5, 2 notes of 1, one note of 5 and one note of 10.

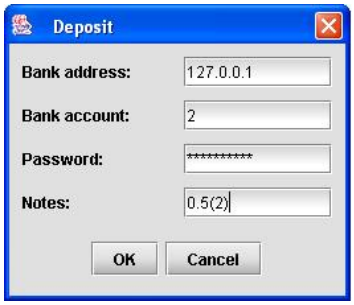
7. Notes list The buyer (or the seller) can at every time check the content of his purse.

8. Start server The seller needs to start his server in order to accept buyers connections.

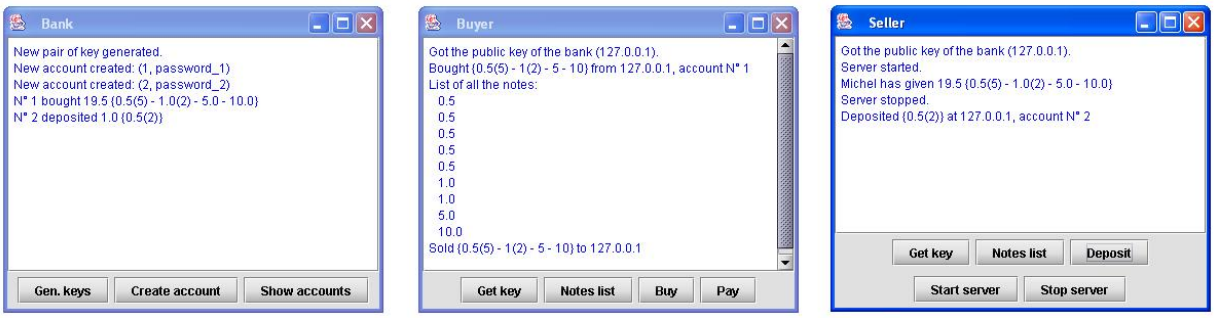
9. Pay The buyer chooses a nickname (in order to identify the payment) and sends the specified notes to the seller.



10. Deposit The seller sends the specified notes to the bank.



After these steps, the frames should look like the following figure:



Like for the ECashFile program, the current state of each participant is stored after each step, which means that the previous process can be stopped and resumed at any time. To clean everything and restart the all freshly, delete the following files: BankState.obj, BuyerState.obj and SellerState.obj. Note: Entries of text fields are stored in properties files (Buyer.properties and Seller.properties).

Bibliography

Books

- "Java Cryptography", O'Reilly, 1998, Jonathan Knusden
- "Java Security Solutions", Wiley, 2002, Rich Helton and Johennie Helton
- "Au coeur de Java 2, notions fondamentales", CampusPress, 2001, Cay S. Horstmann & Gary Cornell
- "Au coeur de Java 2, fonctions avancées", CampusPress, 2000, Cay S. Horstmann & Gary Cornell
- "Halte aux Hackers", OEM, 2002, Stuart McClure, Joel Scambray, George Kurtz
- "Compilateurs avec C++", Addison-Wesley, 1994, Jacques Menu

Bachelor's degrees

- "Micro-paiements par téléphones mobiles", University of Geneva, 2002, Dorothée Stadler.
- "Réalisation d'une CA pour l'UNIGE", University of Geneva, 2000, Alain Hugentobler.

WEB

Name	Address
Algo crypto	http://www.mit.edu/afs/sipb.mit.edu/user/jmorzins/Public/mindbright/security/
Cryptix	http://www.cryptix.org/
DigiCash, information about DigiCash	http://www.rambit.qc.ca/plamondon/digicash.htm
E-Cash, basic security of the ecash payment system (PDF)	http://www.win.tue.nl/~berry/papers/cosic.pdf
E-Cash, algorithm	http://www.google.ch/search?q=cache:NjXL21FW2qoC:euro.ecom.cmu.edu/program/courses/tcr763/2002pgh/ecash11.ppt+ecash+protocol&hl=de&ie=UTF-8
E-Cash, papers, masters, PhD	http://www.tcs.hut.fi/~helger/crypto/link/protocols/ecash.html
E-commerce, Semper, several links	http://www.semper.org/sirene/outsideworld/ecommerce.html

Electronic Check Processing	http://www.cryptologic.com/ecash/ecp.html
Electronic money, generalities	http://www.tcm.hut.fi/Opinnot/Tik-110.501/1995/ecash.html
Encryption 1	http://garbo.uwasa.fi/pc/crypt.html
Encryption 2	http://jeremy.hksys.com/hotlist.html
Google, products and tools	http://directory.google.com/Top/Computers/Security/Products_and_Tools/
GPG (GNU Privacy Guard)	http://www.gnupg.org/
Handbook of cryptography	http://www.cacr.math.uwaterloo.ca/hac/
Java Security Solution, website of the book	http://www.richware.com/JavaSecuritySolutions/
Listener, shareware, TigerTools	http://www.tigertools.net/
Listener, free, ethereal	http://www.ethereal.com/
Listener, free, polito	http://analyzer.polito.it/
Micro-payments, Cash card, Swiss system	http://www.cashcard.ch/
Micro-payments, Bantry technologies, development and services company, payments over mobile phones	http://www.bantry-technologies.com/
Micro-payments, Mastercardint International, presentation of OneSmart	http://www.mastercardintl.com/newtechnology/mcommerce/
Micro-payments, Mondex, system part of the OneSmart program	http://www.mondex.com
Micro-payments, Newgenpay, presents the Valuto system, a platform for multiple applications (Micro-payments, wireless, ...)	http://www.newgenpay.com/
Micro-payments, Proton, belgian cash system	http://www.proton.be
Micro-payments, Transaction Net, list of payments systems (especially micro-payments)	http://www.transaction.net/payment/
Micro-payments, Visa	http://www.visa.com
Micro-payments, W3C,	http://www.w3.org/ECommerce/Micropayments/

links to different systems	
PGP	http://www.pgp.com/
PGPI	http://www.pgpi.org/
PGP International	http://www.pgpinternational.com/
Strong Privacy Protection vs. Data Trail	http://waste.informatik.hu-berlin.de/Grassmuck/Texts/ecash.e.html