The following paper was originally published in the
Proceedings of the Sixth USENIX UNIX Security Symposium
San Jose, California, July 1996.

A Framework for Building an Electronic Currency System

Lei Tang
Carnegie Mellon University

A Framework for Building an Electronic Currency System

Lei Tang *

GSIA
Carnegie Mellon University
Pittsburgh, PA15213-3891
ltang@cmu.edu

Abstract

We describe the framework for building an electronic currency system. We detail the design of the components of the electronic currency system and the relationship among them. Contrary to the previous electronic currency literatures, which focus exclusively on electronic currency protocol designs, we address how to achieve both transaction atomicity and transaction anonymity at the presence of the hostile failures, which are common in an electronic currency system if the customers or the merchants are dishonest or malicious. We also propose a recovery method called redo_transaction to recover from hostile failures so that the aborted electronic currency transactions caused by the hostile failures can be forced to commit eventually. The structure of the electronic currency system is protocol-independent in the sense that those Chaum-like off-line electronic currency protocol could be incorporated into our framework.

1 Introduction

An electronic currency system is the strongest electronic payment system in the sense that it provides anonymity (hence privacy) to its customers, as its counterpart paper currency does. One difference between the electronic currency and the paper currency is that the payer and the payee who complete a transaction through the electronic currency do not have to be face-to-face, as the case in a paper-currency transaction. The big challenge for building an electronic currency system is to build a system secure against malicious attacks from the adversaries (including both attacks on the system and cryptoanalysis attacks on the protocols) while preserving the transaction atomicity, which guarantees the consistent and non-repudiational termination of the transaction. Electronic currency protocol design and transaction processing management are two key components of an electronic currency system. Most of the work so far in the electronic currency area are concentrated on the protocol design, while little has been reported on the overall structure of an electronic currency system, and the foundation for building such a system–transaction management.

There are two implemented electronic currency systems: the DigiCash system by Chaum and the NetCash system [13] by Neuman and Medvinsky. The former is based on Chaum's blind signature technique [4]. The latter is based on a trusted third-party. It is an on-line system based on private key algorithms. Due to the protection of commercial secrecy, little has been reported on the structures of the systems and transaction management mechanisms. In this paper, we present a framework to build an electronic currency system. Our framework is protocol-independent so that the existing Chaum-like off-line electronic currency protocols could be incorporated into this framework.

There are many problems faced by electronic currency system designers and builders. We list a few that we think are crucial for building an electronic currency system.

- Electronic currency protocol design and protocol security analysis.
- Public key management, especially real-time key revocation since the electronic currency system is off-line.
- Secure and verifiable message transmission through an open computer network.
- Transaction management, especially how to

*The views and conclusions contained in this document are solely those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University.

achieve both atomicity and anonymity; how to recover from hostile failures caused deliberately by malicious participants involved in the transaction. Anonymity requirement of an electronic currency system makes building a robust recovery mechanism difficult.

It is impossible for us to address all these issues in a single paper. Instead, we focus on the description of the overall structure of the off-line electronic currency system, the transaction management, and the recovery mechanisms (especially recovery from hostile failures).

A concept called verifiable transaction atomicity [15] is used to design our system so that both anonymity and transaction atomicity can be achieved. We also use a technique called redo_transaction to recover from those hostile failures so that the aborted transactions caused by the hostile failures will be forced to commit eventually.

Our presentation is organized as follows. We give an overview of the structure of our system in section two. In section three, we give brief introduction to the electronic currency concept, and the verifiable transaction atomicity concept used to achieve both transaction atomicity and transaction anonymity. We detail the structure and the components of the bank service, and the relationships among all these components. Section five and section six describe the customer services and merchant services respectively. We conclude in section seven.

2 Overall structure of the system

Our system consists of parties belonging to three different categories. They are a bank, a group of customers, and a group of merchants. The bank issues electronic currency tokens to the customers and receives electronic currency tokens from the merchants for deposit. Once the electronic currency token is issued to the customer by the bank, the identity of the customer associated with this token is untraceable if the customer does not use the token twice. If the customer reuse the electronic currency token, his identity will be revealed by the bank when the merchant deposits this token to the bank. The customers and the merchants complete their financial transactions by passing the electronic currency tokens from one party to another. The customer's identity should not be revealed to the merchant during the payment phase. The merchants deposit those electronic currency tokens obtained from the customers to the bank for refund in the deposit phase.

For our purpose, each customer or merchant has a computer running an operating system that is trusted for local security and is not shared with other users. The bank provides banking services to the customers and the merchants. The banking services are well protected. All of the nodes in the distributed system are connected by an insecure network. Different nodes communicate by sending and receiving messages over the links of the network. We assume the system is synchronous in the sense that bounds exist (and are known) for both relative speeds of processes and message delays, so the communication failures can be detected by a timeout mechanism (the detailed definition of synchronous is given in [2, 10]).

At any time a process is either faulty or correct. A process is faulty in an execution if its behavior deviates from that prescribed by the algorithm it is running. Otherwise, it is correct. A model of failure specifies in what way a faulty process can deviate from its algorithm. There are several types of process failures. They fall into two broad categories: the benign failures and the hostile failures. The definition of a *benign failure* is given in [10]. A process that suffers a benign failure does not arbitrarily change state, or send an arbitrary message that is not prescribed by its algorithm according to its present state. We call failures with more severe faulty behaviors than those faulty behaviors exhibited by benign failures the *hostile failures*. The benign failures (such as system crashes) are the ones that arises most commonly where no monetary transactions are involved. Most of the fault-tolerant protocols in distributed systems are focused almost exclusively on those benign failures. In the electronic currency system, a dishonest or malicious process may exhibit any behaviors which will help itself to benefit financially from its behaviors by sabotaging the transactions. The failures (caused by a faulty process) exhibiting an arbitrary behavior are hostile failures (also called Byzantine failures, or arbitrary failures). With arbitrary failures, a faulty process may claim to have received a particular message from a correct process, even though it never did; a faulty process may claim to have never received a particular message from a correct process, even though it did; or a faulty process may fabricate failures appeared to be benign failures any time during the transaction.

The setting for our system is a distributed system where each node runs the Unix operating system. Some distributed transaction facility such as the Encina family of products developed by Transarc Corporation is layered above the operating system. It provides mechanisms for constructing reliable distributed programs that are robust against benign

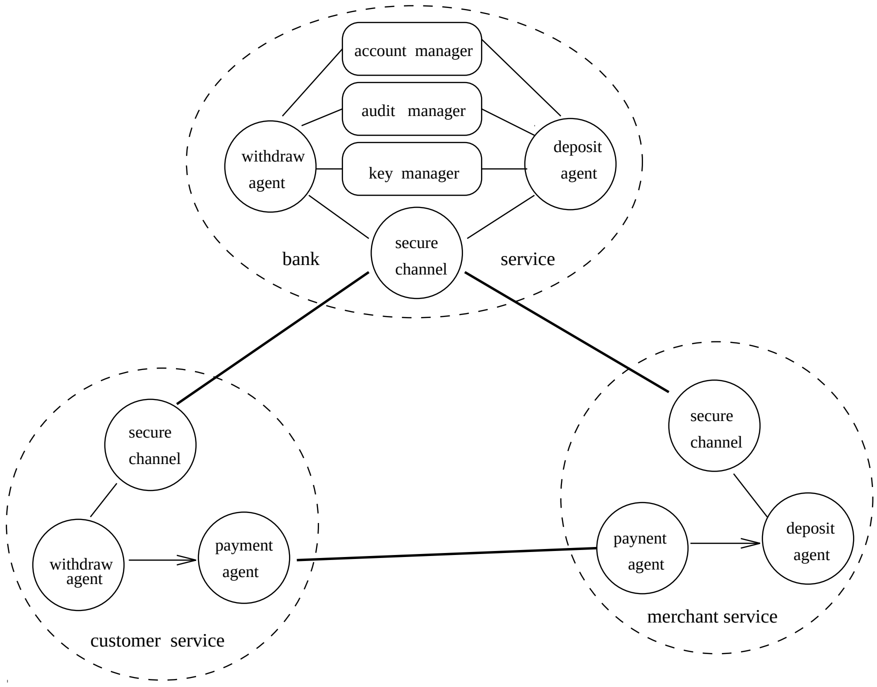

Figure 1: The structure of the electronic currency system

failures and guarantee transaction consistency.

In each node, a new operating system component called the *secure communication agent* manages identity authentication of the party with which the node communicates, and establishes a secure channel between them. All subsequent communication associated with this session has to go through this secure channel. There are two different currency withdrawal agents located in the bank's node and the customer's node respectively for the customer to withdraw electronic currency tokens from the bank. There are also two different currency deposit agents located in the bank's node and the merchant's node respectively for the merchant to deposit electronic currency tokens to the bank. The customers and the merchants do transaction through the payment agents in them. The bank also has other management servers for key management, account management, and audit management. The structure of our system is shown in Fig 1.

3 Background

In this section we give a brief introduction to the electronic currency concepts, the protocol designs, and the special properties. Then we explain the difference between the atomic commitment problem in an electronic currency system and that in the traditional database system. We describe a concept called *verifiable transaction atomicity* [15] and briefly describe its specifications. We also define some notations used throughout this paper.

3.1 Electronic currency concept and protocol

An electronic currency token denotes money by a sequence of bytes. An ideal electronic currency system should have properties that the paper currency has: unforgeable, uncopyable, untraceable, off-line between the merchant and the bank (so that the merchant does not have to consult the bank to check if an electronic currency token is valid or not whenever the merchant receives a token from a customer), and transferrable from one customer to another. Uncopyability, unforgeability, and untraceability properties make the design of an electronic currency system challenging. Paper currency has physical properties that add security (it is difficult to copy or counterfeit dollar bills). Electronic currency is only represented as a sequences of bytes. It is harder to prevent copying, counterfeiting, and tracking the flow of electronic currency.

Chaum introduced the *blind signature technique* [4] to solve the problem of untraceability. The basic idea of Chaum's blind signature is as follows: Let (r, n) and (d, n) be the RSA public key/secret key pair for a bank. When a customer wants an electronic token worth one unit, he chooses a random number x and a number y of a special form. Then he sends $C = f(y) \cdot x^r \bmod n$ to the bank, here f is a publicly known one-way hash function. After receiving C, the bank sends $D = C^d \bmod n$ to the customer. The customer computes the value of $f(y)^d \bmod n$ from $D = f(y)^d \cdot x \bmod n$ since the customer knows the value of x. When some merchant presents $(y, f(y)^d \bmod n)$ (sometimes denoted by $(y, f(y)^{1/r} \bmod n)$) to the bank, the bank can not determine which customer spent this token. Here the untraceability is achieved through the blind factor x generated randomly by the customer. We use the terminology *blind manipulation* to describe the technique of hiding values by multiplying them by random factors.

Chaum's original system did not protect against a customer making multiple copies of an electronic token and using it with multiple merchants. It is possible to solve this problem by requiring that the bank be on-line and participate in every transaction, but this raises difficult scaling and reliability issues. Using the cut-and-choose technique first introduced by Rabin [14], Chaum, Fiat, and Naor improved Chaum's original protocol by using an interactive protocol to determine if an electronic token was reused [5]. In the Chaum-Fiat-Naor protocol, many special values are computed by a customer and sent to the bank. The bank issues a cryptographic challenge, and the customer replies after performing additional computations. A set of lightweight electronic currency protocols are designed after the Chaum-Fiat-Naor protocol [6, 7, 13]. We call all those off-line electronic currency protocols which are based on RSA algorithm and Chaum blind signature the Chaum-like protocols. In this paper, we design an electronic currency framework which could accommodate all these Chaum-like protocols. In the sequel, we use $f(y)^{1/r}$ to denote the electronic currency token in the Chaum-like electronic currency protocols for simple exposition.

3.2 Transaction Atomicity and Anonymity

An electronic currency system should have anonymity (or untraceability) properties. Basically, the electronic currency token should be anonymous, which means that the token can not be associated with the identity of the customer once it is issued by the bank. Also the identity of the customer should be anonymous when the electronic currency token is paid to the merchant. The anonymity requirement makes transaction atomicity difficult to accomplish. For example, in the currency withdrawal protocol, a customer sends a currency withdrawal request to the bank. The bank authenticates the customer's identity. If the authentication is successful, the bank issues an electronic currency token to the customer and deducts the customer's account. In this protocol, a dishonest customer may cheat as follows. The dishonest customer does not sign an acknowledgement to the bank server to commit the transaction after he receives the currency token. Instead, the customer fabricates a failure (e.g., he unplugs the network connection to pretend a communication failure) deliberately so that there is no way for the bank server to detect the cause of the failure. Hence the bank server can not determine if the electronic currency token has been received by the customer or lost in the transmission. If the transaction is managed by a traditional transaction manager (according to the two-phase commitment protocol [2, 8, 11]), the bank will abort the transaction and undo the updates on this customer's account. The dishonest customer obtains a valid currency token, but his bank account is not deducted. Since the electronic currency token is untraceable, the identity of the dishonest customer can not be revealed if he spends the electronic currency token once. The bank will lose money because of the dishonest behavior of the customer during the withdrawal procedure. This is unacceptable to the bank. This is also an example that the traditional transaction management techniques used to achieve transaction atomicity are not robust enough to handle hostile failures caused by malicious or dishonest participants involved. To remedy this problem, a concept called *verifiable transaction atomicity* is proposed in [15]. Beside the properties which the atomic commitment protocol should have [9, 1], the verifiable atomic commitment protocol should have two additional properties.

- The interactions (communications) and the contents of the interactions among all participants are verifiable (by a third party) if necessary.
- If a participant receives a message with which some monetary value is associated, the transaction in which the participant is involved will be committed eventually (even though the participant may abort the transaction unilaterally).

Verifiable transaction atomicity guarantees both transaction atomicity and transaction anonymity.

3.3 Notations

We use the symbols S and R to denote the sender and the receiver respectively. T_a represents the times-

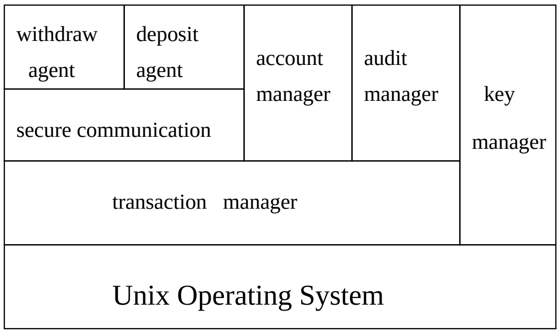

Figure 2: The structure of the bank service

tamp read from A's clock. We also use the symbols below to represent the following.

K_a A's public key.
K_a^{-1} A's private key.
K_{ab} The key shared between A and B.
$\{x\}_K$ x encrypted with key K.
x, y x concatenated with y.
Id_a A's true identity.
P_a A's pseudonym.

4 The bank service

Our electronic currency system consists of three parts: the bank, the customer, and the merchant. The bank provides services for authentication and secure communication with the customers and the merchants. It also provides services for currency withdrawals with the customers and currency deposits with the merchants. The customers (the merchants) withdraw (deposit) currency with the bank service via the client process running in the customers' (the merchants') local node. The bank services consists of many agents implementing all token exchanges and validations. All of them can be implemented as independent servers. They are the *secure communication agent*, the *withdrawal agent*, the *deposit agent*, the *key manager*, the *account manager*, the *audit manager*, and the *transaction manager*. The structure of the bank service is shown in Figure 2.

The secure communication agent is responsible for authenticating the customer's (or the merchant's) identity and establishing a secure channel between the bank and the customer (or the merchant). All subsequent communication has to go through the secure channel. It is also responsible for achieving verifiable message transmission, which will be elaborated later in this section. The withdrawal agent generates electronic currency tokens securely for the customers. The deposit agent receives and verifies tokens deposited by the merchants. The account manager is responsible for updating bank accounts for the customers and the merchants. The transaction manager provides fault-tolerance and consistency to our system. The audit manager audits all system activities and logs these operations to the hard disk efficiently. The key manager is responsible for registering and revoking public keys for the customers and the merchants. We will describe the structures of the secure communication agent, the withdrawal agent, the deposit agent, and the transaction manager in the following sections.

4.1 The secure communication agent

The secure communication agent handles the following functions.

- Authenticating the identity of the customer (or the merchant) running the client process.
- Generating a secure session key known only to the bank service and the client process for currency withdrawal transactions and currency deposit transactions. This session key is cached during the transaction and is flushed after the transaction. We use **secure-send**(m) and **secure-receive**(m) to denote the secure communication primitives.

Two verifiable message transmission primitives **delivery**(m, *receiver*) and **receipt**(m, *sender*) are designed to achieve two properties. First, if the receiver receives a message m from the sender, it can not deny the receipt of m, or the content of m. Second, if the sender transmits a message m to the receiver and the receiver receives it, the sender can not deny the sending of m or the content of m. The two primitives corresponding to that the sender transmits a message m to the receiver and the receiver receives a message m from the sender respectively. The secure communication agent is responsible for verifiable message transmission. We use the pseudo-Pascal syntax with the usual sequential control flow structure to describe these two primitives.

The **wait-for** statement is used to block the message receiver until the receipt of a particular message. If the message may arrive at unspecified times and should be received without blocking the message receiver, then the **upon** statement is used. δ is the upper bound on the message delay, this consists of the time it takes for sending, transporting, and receiving a message over a link. In case of a blocking wait, an optimal timeout may be set to trigger at a particular (local) time using the **set-timeout-to** statement. The timeout value in effect is that set by the most recent **set-timeout-to** before the execution of a **wait-for** statement. If the event being waiting for does not occur by the specified time, then the **on-timeout** clause of the **wait-for** statement is executed rather than its body. The body and the time-out clause of wait-for are mutually exclusive.

We assume that there are some daemon processes (e.g., a httpd server, a ftpd server, or a gopher server) in the sender's side to export a portion of the local file system. The **location** sent to receiver by the sender could be a Universe Resource Locator (URL). In case that the sender wants to prevent the participants other than the receiver from accessing the information located at **location**, the file name of the URL can be randomized so that only the intended receiver can obtain the URL. The content of the location exported by the sender can be verified by a third party if the receiver disputes the content of the file in **location**.

In the algorithm, a certificate to the public key is sent together with any message signed by the corresponding private key. We do not write down explicitly the certificate for simple exposition. In case that the anonymous sender nonrepudiation property is desired, the pseudonym [3] of the sender P_s is included in the message and certificate instead of the true identity of the sender. The protocol of verifiable message transmission is shown in Figure 3 [1].

The protocol consists of two tasks, one is executed by the sender (the procedure **delivery**) and the other executed by the receiver (the procedure **receipt**). The sender starts by sending out the message with an encrypted form and waiting for the signed acknowledgement of the receipt of the encrypted message from the receiver. When the receiver receives such a message, the receiver sends out the signed acknowledgement. After receiving the signed acknowledgement from the receiver, the sender sends the decryption key to the receiver and puts the decryption key on the URL **location** no matter whether the sender receives an acknowledgement about the receipt of the key from the receiver or not. If the sender does not receive the signed acknowledgement in time, the sender will unilaterally decide to abort the delivery. After receiving a message, the receiver checks if the message has the specific form signed by the sender, then the receiver responds with a signed acknowledgement and waits for the arrival of the decryption key. If the receivers receives the key from the sender directly, he can obtain the message m. Otherwise, he fetches the key located in **location**, which is sent to him by the sender in the first round of the communication. The sender may send a false key K' to the receiver, or put a false key K' in **location**. But he gains nothing since the sender has to give k to the receiver under a trusted arbitrator.

[1]Figures 3-6 are all borrowed from [15]

SENDER:
procedure delivery(m, receiver)
  invent K;
  $Em := \{\{m\}_K, Id_s, Id_r, T_s, \textbf{location}\}_{K_s^{-1}}$
  secure-send(Em) **to** receiver
  set-timeout-to $T_s + 2\delta$
  wait-for (**secure-receive**(Dm) **from** receiver)
    secure-send ($\{K, T_s, Id_r\}_{K_s^{-1}}$) **to** receiver
    $Ek = \{K, Id_r, Id_s, T_s\}_{K_s^{-1}}$
    put Ek into **location**
    log(signed receipt of m from receiver)
  on-timeout
    abort delivery
    decide according to terminate-protocol()

RECEIVER:
procedure receipt(m, sender)
  upon (**secure-receive**(Em) **from** sender)
    verify if Em is in the proper format
    $Dm := \{\{m\}_K, Id_r, Id_s, T_r\}_{K_r^{-1}}$
    secure-send(Dm) **to** sender
    set-timeout $T_r + 2\delta$
    wait-for (**secure-receive**(Ek) **from** sender)
      $Dk := \{K, T_r, Id_r\}_{K_r^{-1}}$
      secure-send(Dk) **to** sender
      decrypt $\{m\}_K$
    on-timeout
      fetch K from **location**
      decrypt $\{m\}_K$

Figure 3: Verifiable Message Transmission Protocol

Besides the procedures **delivery** and **receipt**, we also provide a procedure **anon_delivery**. It is the same as **delivery** except that the sender's pseudonym P_s instead of his true identity Id_s is included in the messages and the public key certificate passed between the sender and the receiver. **anon_delivery** achieves the anonymous sender nonrepudiation property.

The secure communication agents in the customer service and the merchant service also implement the above verifiable message transmission primitives.

4.2 The withdrawal agent

The currency withdrawal agent processes the currency withdrawal operations between the customer and the bank. After the customer's identity is authenticated, the withdrawal agent issues electronic currency tokens to the customer and informs the account manager to deduct the customer's account.

All withdrawal operations inside the withdrawal agent are bracketed by **begin_transaction** and **end_transaction**. The withdrawal agent has the functions described as follows.

- Executing a currency withdrawal protocol with the customer and issuing the currency tokens to the customer.
- Notifying the account manager to deduct the customer's account after the currency tokens are issued or to credit the merchant's account after the currency tokens are deposited.
- Inquiring the key manager about the validity of the customer's public key.

The currency withdrawal agent implements a secure currency withdrawal protocol between the customer's client (called C) and the bank service (called B). After the withdrawal protocol is completed, the customer obtains a fresh electronic currency token.

The **auth** and **verify** modules are procedures of authentication between the sender and the receiver. **redo_transaction** is the procedure to force an aborted transaction to commit. We describe **redo_transaction** in section 4.4. **atomic_commitment** is a generic atomic commitment protocol [1, 2, 8] with two additional operations when the transaction is aborted: the identities of the participants are logged into the redo_list and the redo records are copied to the redo_log when the transaction is aborted. A YES vote indicates that the local execution was successful and the participant has received the nonrepudiational receipt, and that the participant is willing and able to make the updates to the data permanent. In other words, the updates and the nonrepudiational receipt have been written to the stable storage so that they can be installed as the new data values even if there are future failures. A NO vote indicates that for some reason (e.g., storage failure, deadlock, invalid monetary token, etc.) the participant is unable to install the results of the transaction as the new permanent data value. In the withdrawal protocol, the redo_transaction forces a dishonest customer to commit a transaction if he did receive a valid electronic currency token, which is untraceable once issued; or to complete a transaction if the number $xf(y)^{1/r}$ in which the electronic currency token is included gets lost during the transmission due to some network malfunctions. The withdrawal agents in the bank service and the customer service are shown in Figure 4.

```
withdrawal agent in the bank service:
procedure withdraw(Id_c, Id_b)
    upon receipt of withdrawal request from Id_c
    If (verify(Id_c)= success) Then
      If ( Id_c ∈ redo_list ) Then
        redo_transaction(Id_c)
      Else
        receipt( x^r f(y) , Id_c)
        compute x f(y)^{1/r}
        delivery( x f(y)^{1/r} , Id_c )
      set-timeout-to  T_b + 2δ
      wait-for(the receipt of Dk)
        update Id_c's account
        vote YES
        atomic_commitment(trans_id, Id_c, Id_b)
      on-timeout
        log Id_c to redo_list
        copy transaction log to redo_log
        vote NO
        abort(trans_id, Id_c, Id_b)

withdrawal agent in the customer service:
Procedure withdraw(Id_c, Id_b)
    sends withdrawal request to bank
    auth(Id_c)
    If (bank requests redo_transaction) Then
      redo_transaction(Id_b)
    Else
      receipt(x f(y)^{1/r} , Id_b)
      If f(y)^{1/r} is valid Then
        update account record and vote YES
        atomic_commitment(trans_id, Id_c, Id_b)
      Else
        vote NO
        copy transaction log to redo_log
        abort(trans_id, Id_c, Id_b)
```

Figure 4: Electronic Currency Withdrawal Agents

4.3 The deposit agent

The currency deposit agent processes the currency deposit between the merchant and the bank. The major part of the currency deposit agent is the replay checker. All deposited currency tokens are stored in the old token database. As time goes on, the database will grow bigger and bigger, and it takes longer to check if a token is a replayed one or not. One way to alleviate this problem is for the bank to change its public key at regular intervals. If a token is not used, the customer can ask the bank to exchange the old currency token for a new one. We delay the discussion of the deposit agent in section 6.

4.4 The transaction manager

The bank services consists of a group of servers to process the currency withdrawal operations between the customer and the bank, and to process the currency deposit operations between the merchant and the bank. This group of servers have to coordinate with each other and to be robust against failures, which occur frequently in the distributed environment. These failures include system crashes, network delays and partitions, failures caused deliberately by a malicious participant, and other unexpected failures. We have given such an example in section 2.3.

Transaction technique[2, 8] are usually used to build systems robust against those benign failures. Transactions were developed to solve problems caused by failures and concurrency. Failure recovery is achieved by a so-called two-phase commitment protocol and consistency is guaranteed by a so-called two-phase lock protocol. In our design, transaction is use mainly for failure recovery, so we will not discuss the consistency problem in our paper. Transactions have what are sometimes called the ACID properties: they are *Atomic*, *Consistent*, *Isolated*, and *Durable*. However, as we mentioned in Section 3.2, the traditional transaction management is not robust against those hostile failures.

We proposed a new method called *redo_transaction* to recover the failures (especially the hostile failures) and to force the participant who has received the electronic currency token to commit the transaction.

We give a sketchy description of how the redo_transaction procedure works. The redo_transaction forces two logs to the stable storage: the *redo_list* and the *redo_log*. The redo_list records the receiver's identity if the sender receives the signed acknowledgement of the encrypted electronic currency tokens but the transaction is aborted. The redo_log records the uncommitted transaction in which the sender has received the signed acknowledgement of the encrypted electronic currency token but the transaction is aborted. The redo_log is indexed by the unique identities of the receivers instead of the LSNs (log sequence numbers) in a usual transaction system. The redo_log is used to recover those failed transactions. The redo recovery in an electronic currency system is different from those redo algorithms focusing exclusively on recovering from a system crash [12, 2, 8]. The recovery mechanisms in a redo-transaction is more complicated than that in an ordinary transaction in the sense that the recovery mechanisms should not only handle recovery from transaction abort and system crashes, but also be able to process uncommitted transaction to force an aborted transaction in which the sender has received the signed acknowledgement of the receipt of the encrypted monetary token from the receiver to commit eventually. Redo_transaction guarantees that both anonymity and atomicity can be achieved in an electronic payment transaction even in the presence of malicious failures.

A redo_transaction begins with mutual authentication between the sender and the receiver. If the sender's identity is found in the redo_list, the logs of corresponding uncommitted transaction are fetched from the redo_log and are sent to the sender. The sender has to commit this transaction before the two parties can begin another new transaction. If the transaction is aborted, the transaction has to be redone until the transaction is committed.

payment agent in the customer service:
procedure payment(P_c, Id_m)
    anon_delivery($f(y)^{1/r}, Id_m$)
    set-timeout $T_c + 2\delta$
    wait-for(receipt of Dk and goods)
      update record and vote YES
      atom_commitment($trans\_id, P_c, Id_m$)
    on-timeout
      disputation_resolve()

payment agent in the merchant service:
procedure payment(P_c, Id_m)
    upon receipt of $f(y)^{1/r}, P_c$
      If $f(y)^{1/r}$ is valid **Then**
        delivery(goods)
        vote YES
        atomic_commitment($trans\_id, P_c, Id_m$)
      Else
        vote NO
        abort($trans\_id, P_c, Id_m$)

Figure 5: Electronic Currency Payment Agents

5 The customer service

The customer service consists of a secure communication agent, a withdrawal agent, and a payment agent. The secure communication agent authenticates the customer's identity to the bank and implements two primitives we described in section 4.1. The withdrawal agent obtains electronic currency tokens from the bank through the procedure shown in Figure 3. The payment agent pays the currency tokens to the merchant and obtains goods or services in return. We describe the payment agent in details in this section.

An important task of the payment agents in the

deposit agent in the merchant service:
procedure deposit(Id_m, Id_b)
  auth(Id_m)
  delivery($f(y)^{1/r}$, Id_m)
  set-timeout-to $T_m + 2\delta$
  wait-for(the receipt of Dk)
    update account record
    vote YES
    atomic_commitment($trans\_id, Id_m, Id_b$)
  on-timeout
    disputation_resolve()

deposit agent in the bank service:
Procedure deposit(Id_m, Id_b)
  If (**verify**(Id_n)= success) **Then**
    receipt($f(y)^{1/r}$, Id_m)
    If ($f(y)^{1/r}$ is valid and not reused) **Then**
      update Id_m's account
      vote YES
      atomic_commitment(trans_id,Id_m, Id_b)
    Else
      vote NO
      abort($trans\_id, Id_m, Id_b$)

Figure 6: Electronic Currency Deposit Agents

customer service and the merchant service is to find who has reused an electronic currency token, the customer or the merchant, if the bank finds that an electronic currency token was reused. Two methods could be used to prevent a dishonest merchant from framing its customers, or to prevent a dishonest customer from reusing a token and denying it. One is to ask the merchant to check replay before it accepts an electronic currency token. Another method is to sign the electronic currency token during the payment. In the second case, the customer's identity should not be revealed to merchant while the merchant can validate the signature. The primitive **anon_delivery** is used to satisfy the above requirements. The primitive serves two functions. One function is to prevent the customer from denying reusing the electronic currency token if he did. Another function is to prevent a dishonest merchant from reusing an electronic currency token and framing that a customer had reused it. If an electronic currency token is detected to be reused by the bank (the details are shown in the deposit protocol), the bank can decide who reused it, the merchant or the customer, by asking the merchant to turn in the corresponding anonymous signature on the electronic currency token. The operations of the payment agents in the customer service and the merchant service are shown in Figure 5.

6 The merchant service

The merchant service consists of a payment agent, a secure communication agent and a deposit agent. I have discussed the payment agent and the secure communication agent in the previous sections. Here we only describe the deposit agents in the merchant service and the bank service. The structures of the deposit agents are shown in Figure 6.

7 Conclusion

We have described a framework for building an electronic currency system. We presented the overall structure of the system. We also introduced the basic techniques to achieve both atomicity and anonymity in an electronic currency system. Our framework is also protocol-independent in the sense that those Chaum-like electronic currency protocols can be incorporated into the framework. We are still in the process of designing recovery mechanisms to give a detailed specification of the redo_transaction procedure.

References

[1] O. Babaoglu and S. Toueg. Non-Blocking Atomic Commitment. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, pages 147–168. Addison-Wesley, 1993.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] D. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Comm. of ACM*, 24(2):84–88, 1981.

[4] D. Chaum. Security without Identification: Transaction Systems to Make Big Brother Obsolete. *Communications of ACM*, 28(10):1030–1044, 1985.

[5] D. Chaum, A. Fiat, and M. Naor. Untraceable Eletronic Cash. In *Advances in Cryptology-Crypto88*, pages 319–327. Springer-Verlag, 1989.

[6] Niels Ferguson. Extensions of Single-term Coin. In *Advances in Cryptolog-Crypto'93*. Springer-Verlag, 1993.

[7] M. Franklin and M. Yung. Towards Provable Secure Efficient Electronic Cash. Technical Report TR CUCS-018-92, Columbia University, 1992.

[8] J. Gray and A. Reuter. *Transaction Processing.* Morgan-Kaufmann, 1994.

[9] V. Hadzilacos. On the Relationship Between the Atomic Commitment and Consensus Problems. In *Fault-Tolerant Distributed Computing*, pages 201–208. Springer-Verlag, 1990.

[10] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, pages 97–145. Addison-Wesley, 1993.

[11] B. Lampson. Atomic Transactions. In *Distributed Systems–Architecture and Implementation*, pages 246–265. Springer-Verlag, 1981.

[12] D. Lomet and M. Tuttle. Redo Recovery after System Crashes. In *Proceedings of the 21st International Conference on Very Large DataBases*, September 1995.

[13] G. Medvinsky and C. Neuman. NetCash: a Design for Practical Electronic Currency on the Internet. In *Proceedings of the first ACM Conference on Computer and Communications Security*, Nov 1993.

[14] M. Rabin. Digitalized signatures. In *Foundations of Secure Computation.* Academic Press, 1978.

[15] Lei Tang. Verifiable Transaction Atomicity for Electronic Payment Systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems.* IEEE Computer Society, May 1996.