Model Checking a Server-Side Micro Payment Protocol

Kaylash Chaudhary(✉) and Ansgar Fehnker

School of Computing, Information and Mathematical Sciences,
University of the South Pacific, Suva, Fiji
kaylash.chaudhary@usp.ac.fj

Abstract. Many virtual payment systems are available on the world wide web for micropayment, and as they deal with money, correctness is important. One such payment system is Netpay. This paper examines the server-side version of the Netpay protocol and provides its formalization as a CSP model. The PAT model checker is used to prove three properties essential for correctness: impossibility of double spending, validity of an ecoin during the execution and the absence of deadlock. We prove that the protocol is executing according to its description based on the assumption that the customers and vendors are cooperative. This is a very strong assumption for system built to prevent abuse, but further analysis suggests that without it the protocol does no longer guarantee all correctness properties.

Keywords: Model checking · Verification · CSP · Micropayment protocols · Virtual payment systems · PAT

1 Introduction

The Internet has grown into a virtual market where the exchange of a wide range of goods is everyday practice. For many payments users have to use their credit cards, even though the transaction costs are significant for small transactions. To facilitate payment of smaller amounts micro-payment technologies emerged [7]. With the increase of paid services and content on the Internet, these online payment system promise the ease of using cash. There are many micro-payment systems available for users to buy goods online such as Netpay [7], Millicent [8], Micro-mint [12], Payword [12], MiniPay [10], Micro-iKP [9] and POPCORN [11]. There are also many micro-payment systems proposed for content sharing in peer to peer networks [1] [2], [14], [15] and [16].

Every payment protocol should guarantee certain essential properties of correctness. In this paper we will model one such system, Netpay, formally, and show that it satisfies essential properties. This paper considers a variant called server-side Netpay in which the e-coins are kept by trusted brokers and vendors, while the customer will only have access to the e-coin ID. This is in contrast to the so-called client-side Netpay protocol in which e-coins are kept by the customers. Previous work considered the correctness of client-side Netpay [3].

This paper models the server-side Netpay using CSP [13]. The model in this paper covers the handling of e-coins, and omits the parts of the protocol that are concerned with redeeming e-coins, and digital signatures of e-coins. It is assumed that the correctness of these are guaranteed independently, especially the correctness of the used cryptographic hash functions. Instead, we prove that the protocol can guarantee that a trusted broker will correctly track an e-coin, even though it will be passed from vendor to vendor, that at any time only one vendor will have a copy of the e-coin, that the customer can not spend more than the e-coin is worth. Finally, we considered whether the system was deadlock free and discovered that some arguably strong assumptions are necessary to ensure correctness. If customers do not cooperate they can get the system into a state where deadlock is possible. We also analysed a design alternative that does not have this problem.

Section 2 introduces the server-side Netpay protocol. Section 3 shows the description of the Netpay protocol using the CSP language. Correctness of this protocol is discussed in Section 4. This paper concludes with discussion for future research in section 5.

2 The Netpay Protocol

The Netpay protocol with server-side e-wallet was proposed by Dai et.al. [5]. It has three types of e-wallets: client side, server-side and cookie-based e-wallet [6]. There are three parties involved in this protocol; customer, vendor and broker. It is assumed that the broker and vendors are honest and are trusted by the customers who may not be honest. To use the protocol, the customers and vendors need to register by opening an account and depositing funds with the broker. The broker is responsible for registration, e-coin generation, debiting and crediting accounts for customers and vendors respectively. The payment is between customers and vendors. Previous work has modeled and verified some properties of client-side Netpay [3]

Netpay uses a number of cryptography and micro-payment terminologies such as:

- **One-way Hash Function:** Netpay uses this function to generate and verify e-coins. In [6] MD5 was used, but it could be replaced by more secure SHA-1.
- **E-coin:** The one-way hash function is applied repeatedly to a seed to generate a series of paywords called e-coin. The paywords are represented in reverse order with the seed at the end. The length of the e-coin determines its value.
- **E-wallet:** An e-wallet is a database to store e-coins.
- **Seed:** It is a randomly selected value used for e-coin generation.
- **Touchstone:** This is the first payword of the e-coin. It is used to verify e-coins.

Using the one way hash function h, an e-coin $W_1, ..., W_n$ is constructed by applying the hash function $n + 1$ times to a seed i.e. $W_0 = h(W_1), W_1 = h(W_2), ..., W_n = h(W_{n+1})$ where W_{n+1} is the seed, and W_0 the touchstone.

The remainder of this section will describe the four basic types of transactions in this protocol. It is assumed that each customer and vendor have a unique ID.

Customer-Broker Transaction. The customer sends an e-coin request with parameter n to the broker, who generates e-coins of length n. Each chain has a unique e-coin ID. The broker stores this information in its database, and sends the e-coin ID to the customer.

Customer-Vendor Transaction. If a customer wishes to buy something from the vendor, the customer sends an e-coin ID. The vendor checks if it has the e-coin and verifies it. If the verification is successful, the customer is notified. If the vendor does not have the e-coin, it requests the location from the broker. The broker will reply with the location of the e-coin, the vendor requests this e-coin from that vendor or broker. Initially, the broker will have the e-coin and after that it will be transferred from one vendor to another.

Vendor-Vendor Transaction. This transaction occurs when one vendor requests an e-coin from another vendor.

Vendor-Broker Transaction. Vendors need to redeem the e-coins spent by customers. The vendor sends the e-coin IDs, touchstones, customer IDs, vendor ID, e-coins and amount to the broker. The broker will verify e-coins and credit the corresponding amount to the vendors account if the spent e-coins are valid. This paper focuses on the spending of e-coins, and omits redemption of e-coins from the model.

Properties of the Netpay Protocol. This paper considers three important properties. The first is on the validity of e-coins. Since there will be transfer of an e-coin from one vendor to another, an e-coin should remain valid in this chain of transfer. The second is on preventing double spending. This protocol prohibits a customer to double spend an e-coin at a different or same vendor. The last property is to show absence of deadlocks.

3 Description of Netpay Protocol Using CSP

This section provides models and description for the Server-Side Netpay protocol. The three parties of the protocol: customer, vendor and broker have been modeled as one process each. For simplicity we assume that there is only one broker, while there can be many customers and vendors.

3.1 Customer Process

Table 1 shows the *Customer*(CID) process for the customer CID. This process has three possible statuses: IDLE, BUYCOIN or SPENDING. Variable STATUS_C keeps track of the status of a customer.

The customer database is stored in variable DB_C. It contains e-coin ID's and amounts. Recall, that an e-coin is constructed by applying hash function to a seed. Each payword in Netpay is accompanied by an index to record number of

Table 1. Customer Process

```
Customer(CID) =
[STATUS_C[CID] == IDLE && (||y:{0..(MAXCOINS-1)}@(DB_C[CID][y][0] == -1))]
    BuyCoin!CID
        {STATUS_C[CID] = BUYCOIN;} ->Customer(CID)
[][STATUS_C[CID] == BUYCOIN ]
    SellCoin[CID]?id
        {
        var index = 0;
        while(index < MAXCOINS)
        {
            if (DB_C[CID][index][0] == -1)
            {
                DB_C[CID][index][0] = id ;
                DB_C[CID][index][1] = ISPOSITIVE;
                index = MAXCOINS;//break the loop
            }
            index = index + 1;
        }

        STATUS_C[CID] = IDLE} -> Customer(CID)
[]([]x:{0..(VENDORS-1)};y:{0..(MAXCOINS-1)}@([STATUS_C[CID] == IDLE
&& DB_C[CID][y][0] != -1 ]
    Spend[x]!CID.DB_C[CID][y][0].DB_C[CID][y][1]
        {STATUS_C[CID] = SPENDING;} ->Customer(CID)))
[][STATUS_C[CID] == SPENDING]
    Approval[CID]?eid1.amt
        {
        if (amt == ISZERO)
        {
            var index = 0;
            while(index < MAXCOINS)
            {
                if(DB_C[CID][index][0]==eid1)
                {
                    DB_C[CID][index][0] = -1;
                    DB_C[CID][index][1] = -1;
                }
            index = index + 1;
            }
        }
        STATUS_C[CID] = IDLE;} ->Customer(CID)
[][STATUS_C[CID] == SPENDING]
    Disapproval[CID]?vid
        {STATUS_C[CID] = IDLE;} ->Customer(CID)
```

unspent paywords; the amount. The amount will be abstracted in this paper as either IsPositive or IsZero. This is because the properties shown in this paper are independent of hash function and the exact amount.

Initially, the customer will have no e-coin ID stored in DB_C. The only enabled event is to buy e-coins on the *BuyCoin* channel with parameter CID, which is the customer ID. Events that use the *BuyCoin* channel have precondition STATUS_C[CID] == IDLE i.e. the customer should be IDLE in order to send a message on this channel. After sending the message to the broker, the customer will change to the BUYCOIN state. The broker can reply on the *SellCoin* channel, since the condition STATUS_C[CID] == BUYCOIN for the customer is met. The customer adds the e-coin ID and the amount to DB_C, and changes to IDLE.

If customer has e-coins and is in the IDLE state, it can buy goods from a vendor on channel *Spend*, which will synchronize with one of the vendors. The channel has two parameters: e-coin ID and amount. After this event, the customer will change the state to SPENDING. The vendor will reply on channel *Approval* or *Disapproval*, depending on whether the payment was accepted or not. In either case, the customer's state will change to IDLE.

There is not much processing done by the customer side compared to the Client-Side Netpay protocol [3]. Most processing is done by the vendor. Note, that the customer can try spending an e-coin as often as it wants. It is up to the vendors and brokers to prevent double spending.

3.2 Broker Process

The process *Broker*(BID) in Table 2 and Table 3 models the broker with unique broker ID BID. The broker has a database DB_B to store all generated e-coins. In addition it has a database LOOKUP that maps an e-coin ID to a vendor or broker. The broker can have status IDLE, BUYCOIN, REQLOC or REQCOIN. Variable STATUS_B is used for tracking the status. Variables *vid_B*, *eloc_B*, *cid_B*, *eid_B* and *amount_B* are used to store intermediate results while generating e-coins or replying to request from customers or vendors.

This process models three tasks for the broker. The first task is the generation of new e-coins, lines 6 - 15 of Table 2. A customer requests a new e-coin on channel *BuyCoin* with parameter *cid*, the ID of the requesting customer. This event is guarded by the expression STATUS_B == IDLE. The event changes the status to BUYCOIN. The broker replies to the customer on channel *SellCoin* with two parameters, an e-coin id and the amount of coins. The broker then updates the two databases (Table 2, lines 26 - 29). The status of the process changes to IDLE.

The next task is to pass new e-coins to vendors upon request. The request by a vendor is modeled by the channel *ReqCoin*[BID]. The model uses an array of channel *ReqCoin*, one channel for each vendor. The vendor *vid* is requesting the broker BID to send the e-coin *eid1* on channel *ReqCoin*[BID] with parameters *vid* and *eid1*. It is enabled if the status is IDLE. This event will change the status to REQCOIN and the variables *eid_B*, *amount_B* and *vid_B* will store the e-coin ID, the amount, and the ID of the requesting vendor respectively. These will be used by the broker to send the e-coin to the requesting vendor BVID on channel *SendCoin*[BVID]. This event is enabled, if the status is REQCOIN. The status will then change to IDLE.

The final task keeps track of the e-coin location and responds to the location request by vendors. The channel *ReqLoc* is used for requests from vendors *vid*

Table 2. Broker Process

```
Broker(BID) =
[STATUS_B == IDLE && (||x:{0..(BROKER_SIZE-1)}@(DB_B[x][0]==-1))]
    BuyCoin?cid
        {
        cid_B = cid;
        var index = 0;
        while(index < BROKER_SIZE)
        {
            if (DB_B[index][0] == eid)
            {
                eid = (eid+1)%MAXEID;
                index = BROKER_SIZE;
            }
            index = index + 1;
        }
        STATUS_B = BUYCOIN;} -> Broker(BID)
[][STATUS_B == BUYCOIN ]
    SellCoin[cid_B]!eid
        {
        var index = 0;
        while(index < BROKER_SIZE)
        {
            if (DB_B[index][0] == -1 && DB_B[index][1] == -1)
            {
                DB_B[index][0] = eid;
                DB_B[index][1] = ISPOSITIVE;
                LOOKUP[index][0] = eid;
                LOOKUP[index][1] = BID;
                index = BROKER_SIZE;
            }
            index = index + 1;
        }
        cid_B = 0;
        eid = (eid+1)%MAXEID;
        STATUS_B = IDLE;} -> Broker(BID)
[][STATUS_B == IDLE]
    ReqLoc?vid.eid1
        {
        vid_B = vid;
        var index = 0;
        while(index < BROKER_SIZE)
        {
            if (LOOKUP[index][0] == eid1)
            {
                eloc_B = LOOKUP[index][1];
                LOOKUP[index][1] = vid;
                index = BROKER_SIZE;//break the loop
            }
            index = index + 1;
        }
        STATUS_B = REQLOC;} -> Broker(BID)
```

Table 3. Broker Process (continued)

```
[][STATUS_B == REQLOC]
    SendLoc[vid_B]!eloc_B
        {
        vid_B = 0;
        eloc_B = -1;
        STATUS_B = IDLE;} -> Broker(BID)
[][STATUS_B == REQCOIN]
    SendCoin[vid_B]!eid_B.amount_B
        { STATUS_B = IDLE;} -> Broker(BID)
[][STATUS_B == IDLE]
    ReqCoin[BID]?eid1.vid
        {
        STATUS_B = REQCOIN;
        var index = 0;
        vid_B = vid;
        while (index < BROKER_SIZE)
        {
            if (DB_B[index][0] == eid1 && DB_B[index][0] != -1)
            {
                eid_B = DB_B[index][0];
                amount_B = DB_B[index][1];
                index = BROKER_SIZE; //end loop
            }
            index = index + 1;
        }
        } -> Broker(BID);
```

for the location of the e-coin *eid1*. This event is enabled if the status is IDLE. The broker will look up the entry for the e-coin ID *eid1* in the LOOKUP database, and save it in variable *Beloc*, and also update the location of the e-coin in the LOOKUP database to that of the requesting vendor *vid*. The state will change to REQLOC, which enables the channel *SendLoc*. This channel is used for the reply to the vendor request. The status of the process changes to IDLE.

3.3 Vendor Process

The process *Vendor*(VID) shown in Tables 4 and 5 models a vendor with ID VID. Each vendor maintains an e-wallet EWALLET which contains e-coins of the various customers. A vendor can have status IDLE, SPENDING, HAVECOIN, HAVELOC, NOCOIN, REQLOC, RECVREQ and REQCOIN. The status is stored in variable STATUS_V. Other variables used to store intermediate results are *vid_V*, *eloc_V*, *cid_V*, *amount_V* and *eid_V*.

The vendor performs two major task: verifying e-coins received from a customer and transferring an e-coin to a requesting vendor. The verification of the e-coins has two cases; either the current vendor has the e-coin in its e-wallet,

Table 4. Vendor Process

```
Vendor(VID) =
[STATUS_V[VID] == IDLE &&
(||x:{0..(EWALLET_SIZE-1)}@(Ewallet[VID][x][0]==-1))]
    Spend[VID]?cid.Eid.amount
        {
        cid_V[VID]=cid;
        var index = 0;
        var flag = false;
        while(index < EWALLET_SIZE)
        {
            if (Ewallet[VID][index][0] == Eid )
            {
                flag = true;
                index_V[VID] = index;
                index = EWALLET_SIZE;//break the loop
            }
            index = index + 1;
        }
        if (flag == true)
            STATUS_V[VID] = HAVECOIN;
        else{
            eid_V[VID] = Eid;
            STATUS_V[VID] = NOCOIN;
            }
        } ->Vendor(VID)

[][STATUS_V[VID] == HAVECOIN && Ewallet[VID][index_V[VID]][1] == ISPOSITIVE]
    Approval[cid_V[VID]]!Ewallet[VID][index_V[VID]][0].ISPOSITIVE
        {
        Ewallet[VID][index_V[VID]][1] = ISPOSITIVE;
        index_V[VID] = 0;
        cid_V[VID] =-1;
        STATUS_V[VID] = IDLE;} ->Vendor(VID)
[][STATUS_V[VID] == HAVECOIN && Ewallet[VID][index_V[VID]][1] == ISPOSITIVE]
    Approval[cid_V[VID]]!Ewallet[VID][index_V[VID]][0].ISZERO
        {
        Ewallet[VID][index_V[VID]][1] = ISZERO;
        index_V[VID] = 0;
        cid_V[VID] = -1;
        STATUS_V[VID] = IDLE;} ->Vendor(VID)
[][STATUS_V[VID] == HAVECOIN ]
    Disapproval[cid_V[VID]]!VID
        {STATUS_V[VID] = IDLE;} ->Vendor(VID)
[][STATUS_V[VID] == NOCOIN]
    ReqLoc!VID.eid_V[VID]
        {STATUS_V[VID] = REQLOC;} ->Vendor(VID)
```

or it is with another vendor or broker. In this case it has to first lookup the location, and then request the e-coin at that location.

The first task is initiated by the customer process on channel *Spend*. The customer, *cid*, sends an e-coin ID, *Eid*, and the amount, *amount*, to a vendor.

Table 5. Vendor Process (continued)

```
[][STATUS_V[VID] == REQLOC]
    SendLoc[VID]?loc
        {
        eloc_V[VID] = loc;
        STATUS_V[VID] = HAVELOC;
        } ->Vendor(VID)

[][STATUS_V[VID] == REQCOIN]
    SendCoin[VID]?eid1.amt
        {
        var index = 0;
        while(index < EWALLET_SIZE)
        {
            if (Ewallet[VID][index][0] == -1 && Ewallet[VID][index][1] == -1)
            {
                Ewallet[VID][index][0] = eid1;
                Ewallet[VID][index][1] = amt;
                index_V[VID] = index;
                index = EWALLET_SIZE;//break the loop
            }
            index = index + 1;
        }
        STATUS_V[VID] = HAVECOIN
        } ->Vendor(VID)
[][STATUS_V[VID] == RECVREQ]
    SendCoin[vid_V[VID]]!eid_V[VID].amount_V[VID]
        {
        vid_V[VID] = -1;
        amount_V[VID]=0;
        eid_V[VID] = -1;
        STATUS_V[VID] = IDLE;} ->Vendor(VID)
[][STATUS_V[VID] == IDLE]
    ReqCoin[VID]?eid1.vid
        {
        var index = 0; vid_V[VID] = vid;
        while (index < EWALLET_SIZE)
        {
            if (Ewallet[VID][index][0] == eid1)
            {
                eid_V[VID] = Ewallet[VID][index][0];
                amount_V[VID] = Ewallet[VID][index][1];
                Ewallet[VID][index][0] = -1;
                Ewallet[VID][index][1] = -1;
                index = EWALLET_SIZE; //end loop
            }
            index = index + 1;
        }
        STATUS_V[VID] = RECVREQ;} ->Vendor(VID)
[][STATUS_V[VID] == HAVELOC]
    ReqCoin[eloc_V[VID]]!eid_V[VID].VID
        {
        eloc_V[VID]= -1;
        eid_V[VID]=-1;
        STATUS_V[VID] = REQCOIN;} ->Vendor(VID);
```

This event is enabled, if the vendor status is Idle. If the vendor has an e-coin with a matching ID (line 12 of Table 4) it will enter status HaveCoin. If not it will store the e-coin ID and change the status to NoCoin.

If the vendor does not have the e-coin and the status is NoCoin, it will first request the e-coin location from the broker and then, wait for the broker to reply with the e-coin location, and then request the e-coin from that vendor or broker. This three step process is modeled as follows:

– The model uses channel *RecLoc* for the request of the location. It is enabled when the process is in the status NoCoin (Table 4, line 45)
– After this request the vendor changes its status to ReqLoc. The vendor then waits for the broker to reply on channel *SendLoc* with the e-coin location *loc* (Table 5, line 5). The vendor then stores the location, $Veloc$[vid], and changes the status to HaveLoc.
– The request of the vendor vid from the vendor/broker $Veloc$[vid] to send e-coin eid[vid] uses channel *ReqCoin*. The reply uses the *SendCoin* channel, upon which the vendor vid changes the status to HaveCoin.

In status HaveCoin, the vendor approves on channel *Approval* if the amount is positive (lines 28 and 35 of Table 4). Otherwise, it will disapprove the transaction on channel *Disapproval*. If the payment is approved, the vendor saves the remaining e-coins (line numbers 31 and 37 of Table 4).

The second task for the vendor is to send an e-coin to requesting vendor. A request for an e-coin *eid1*, from another vendor, *vid*, is modeled using channel *ReqCoin*. This is a message from vendor *vid* to vendor *VID*. The event is enabled in status Idle. This event stores the details of the e-coins and changes the status to RecvReq. This enables the reply to the requesting vendor on channel *SendCoin*. The status will change to Idle and the e-coin will be removed from the e-wallet.

The Netpay process is composed of customer, vendor and broker process. There are three vendors, two customers and one broker in this model as shown in [4]. The next section will look at the correctness of this protocol to prove three different properties namely chain of trust, preventing double spending and non-blocking behavior.

4 Correctness of the Netpay Protocol

This section assumes that vendors are cooperative and trusted, which seems consistent with the fact that in the server-side protocol the customer e-coins are stored by vendors. Since the touchstone and payword are always stored together at a trusted party there is no need to prove that the payword remains valid. In contrast, that was an important property to prove for the client-side Netpay protocol [3], in which e-coins were stored by the customers, while the broker and vendors kept the touchstones to verify the e-coins.

For the server-side model we will show three properties. The first is that the e-coin will not be lost by the vendors, which means that the location of the e-coin as recorded by the broker will be correct at the end of a transaction. While a payment is ongoing, it might be temporarily incorrect.

Furthermore, we show that at most one vendor can have a copy of an e-coin. The length of an e-coin, and thus its amount, is abstracted, and we assume that subtracting from the amount is dealt correctly by the trusted vendor. The only remaining way to double spend would be to have two coins. We show that an e-coin cannot be spent twice at different vendors. Finally, we show that there is a deadlock in the protocol and we will present a solution for this.

4.1 Chain of Trust

E-coins are transferred from broker to vendor and from one vendor to another vendor, and the customer should be sure that the location of the e-coin will be tracked in this process. For the server-side Netpay protocol, the following two properties can be shown to hold:

1. If the customer is in the IDLE or BUYCOIN state, then the broker will have the location of the e-coin pointing to the vendor with the e-coin.
2. If the customer is in the SPENDING state, then the broker will have the location of the e-coin, or it will point to the vendor which will receive the e-coin after the next exchange of e-coins.

The following lists the goals defined in the PAT model checker:

- Property 1 shows for each e-coin ID held by a customer, that there exists a corresponding e-coin in the broker database.
- Property 2 shows for each e-coin ID in the LOOKUP database, that there exists a corresponding e-coin in the broker database, if the location LOOKUP$[y][1]$ is the broker ID.
- Property 3 shows for each e-coin ID held by the customer, that if the location of the e-coin is not the broker ID and the customer status is IDLE or BUYCOIN, then there exists a corresponding e-coin at that location.
- Property 4 shows for each e-coin ID held by the customer, that if the location of the e-coin is not equal to the broker ID and the customer status is SPENDING, then there exists a corresponding e-coin at that location or at a location stored in variable *eloc_V*. This means that while the broker may have information that is temporarily not valid, the correct location is stored in an auxiliary variable.

The properties 1 - 4 were verified using the PAT model checker.

4.2 Double Spending

The main goal of this property is to prevent customers from spending an e-coin more than once. Note, that because we abstract the exact amount of an e-coin it can be spent as long as the amount is ISPOSITIVE. This assumes that the vendor updates the amount correctly. However, double spending could still

Property 1. Chain of Trust - Comparison of customer and broker databases

```
#define Chain_of_Trust_1(&&y:{0..(MAXCOINS-1)};x:{0..(CUSTOMERS-1)}
        @(DB_C[x][y][0]==-1 ||(||z:{0..(BROKER_SIZE-1)}
        @(DB_C[x][y][0] == DB_B[z][0]))));

#assert Netpay |=[] Chain_of_Trust_1 ;
```

Property 2. Chain of Trust - Comparison of broker and lookup databases

```
#define Chain_of_Trust_2(&&y:{0..BROKER_SIZE-1}@(LOOKUP[y][1]!= BROKERID
        ||(||z:{0..BROKER_SIZE-1}@(DB_B[z][0] == LOOKUP[y][0]))));

#assert Netpay |=[] Chain_of_Trust_2 ;
```

Property 3. Chain of Trust - Comparison of vendor, customer and lookup databases

```
#define Chain_of_Trust_3(&&y:{0..(MAXCOINS-1)};a:{0..(CUSTOMERS-1)}
        @(||z:{0..(BROKER_SIZE-1)}@(!(DB_C[a][y][0] == LOOKUP[z][0]
        && LOOKUP[z][1]!= BROKERID &&(STATUS_C[a] == IDLE ||
        STATUS_C[a] == BUYCOIN))) ||  (||x:{0..(VENDORS-1)}
        @(||b:{0..(EWALLET_SIZE-1)}@(DB_C[a][y][0]==Ewallet[x][b][0])))));

#assert Netpay |=[] Chain_of_Trust_3;
```

Property 4. Chain of Trust - Comparison of customer, lookup and different vendor databases

```
#define Chain_of_Trust_4(&&y:{0..(MAXCOINS-1)};a:{0..(CUSTOMERS-1)}
        @(||z:{0..(BROKER_SIZE-1)}@(DB_C[a][y][0] != LOOKUP[z][0]
        || STATUS_C[a] != SPENDING || (&&x:{0..(VENDORS-1)}
        @(LOOKUP[z][1]!= x || eloc_V[x]==-1 || eloc_V[x]==BROKERID)
        ||(||b:{0..(EWALLET_SIZE-1)}@(DB_C[a][y][0] == Ewallet[eloc_V[x]][b][0]
        || DB_C[a][y][0] == Ewallet[x][b][0]))))) )  ;

#assert Netpay |=[] Chain_of_Trust_4;
```

occur, if there would be e-wallets with same e-coin ID and a positive amount. In that case the customer could spend the same e-coin at two different vendors. We prove that all e-coins exists only once in one e-wallet at any time. This is expressed in Property 5: No two e-wallets have the same e-coin ID whose amount is IsPositive. This means that the coin cannot be spent twice. A Windows 7, i5 processor, 3.2 GHz and 6 GB RAM machine took about eleven minutes to verify all 5 properties.

4.3 Non-Blocking Behavior

The CSP model described in Section 3 uses channels for communication between different processes. A sender process will be blocked on an output channel if no

Property 5. Double Spending

```
#define DoubleSpending(&&x:{0..VENDORS-1};a:{0..VENDORS-1}
        @(&&y:{0..EWALLET_SIZE-1};z:{0..EWALLET_SIZE-1}
        @((Ewallet[x][y][0]==-1||Ewallet[a][z][0]==-1
        ||Ewallet[x][y][1]==ISZERO||Ewallet[a][z][1]==ISZERO || a==x)
        ||(Ewallet[x][y][0]!=Ewallet[a][z][0]))));

#assert Netpay |=[] DoubleSpending;
```

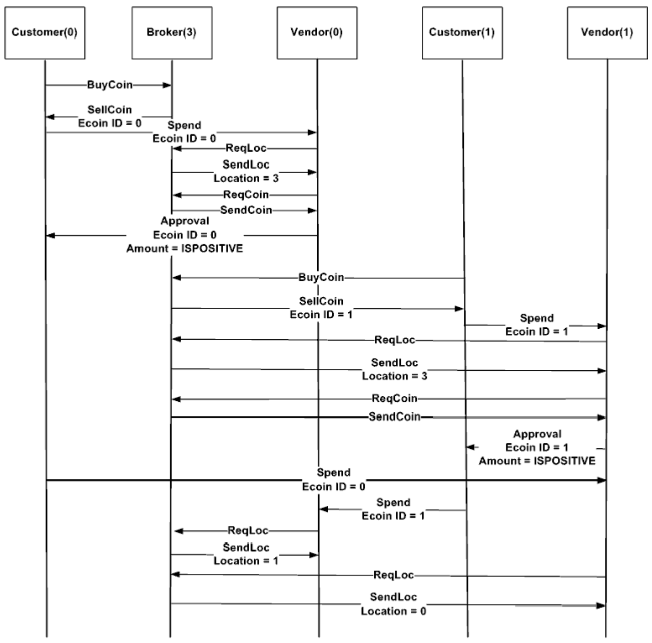

Fig. 1. Deadlock in Netpay protocol

other process has a matching input channel that is enabled. If a channel is blocked indefinitely, then there is a deadlock in the protocol. We use PAT to check for absence of deadlocks.

PAT did identify a deadlock in the protocol. Figure 1 depicts the trace that was generated by PAT model checker. The deadlock occurs as follows: Customer(0) buys an e-coin with Broker(3) and spends e-coins at vendor(0). Vendor(0) requests the e-coin location from the broker. The broker provides its own ID as e-coin location and updates the location for this e-coin to 0. Vendor(0) requests and receives the e-coin from the broker. Vendor(0) approves payment for Customer(0), but the amount remains positive. Likewise, Customer(1) buys an e-coin and spends it at Vendor(1), with a positive amount remaining. Now Customer(0) spends e-coins with ID 0 at vendor(1) while Customer(1) spends e-coins with ID 1 at Vendor(0). Vendor(1) requests and receives e-coin location as

0 from broker. Vendor(1) requests for e-coin location and receives 1 from broker. Now they ask request the e-coin from each other, and enter a circular wait.

This model of the protocol has been corrected by adding a separate process for the vendor side to handle reply for e-coins request. This process can be found in [4]. With these changes PAT can verify deadlock freedom. The description of the protocol does not specify how many processes a vendor should have [7], although it is presented as if it were a single process. A prototype implementation also used a single process. Our results suggest that the vendor should be split instead into two parts to avoid a deadlock.

4.4 Non-Cooperative Customers and Vendors

A system is not only composed of cooperative parties but there will be some parties who will try to cheat the protocol. In such a case, the protocol may not be correct anymore. The current protocol does deal with customers who try to spend e-coins that have zero amount, and also with customers who want to spend the same e-coin twice. However, if we would enable a customer to send e-coin IDs that do not exists in the broker or vendor databases, it will cause currently a deadlock. The current protocol provides no way for a broker to communicate back to vendor that an e-coin does not exist. This problem is, however, easily addressed by adding one more case for a declined payment.

We also considered cheating vendors in models that do not abstract from the exact amount an e-coin is worth. Since vendors in the Server-side Netpay protocol handle the e-wallet they can cheat by deducting or redeeming the wrong amount. It will be very difficult for the customer (or broker) to detect such behavior. In the client-side protocol the customer owned the payword, and could use this to verify correct behavior of the vendor.

5 Conclusions and Future Research

This paper modeled the server-side Netpay protocol used for micro-payment using CSP, and verified three important properties using the PAT model checker. The first is that the broker keeps track of e-coins throughout, even if it is transferred from vendor to vendor. The second is that a customer cannot spend more than an e-coin is worth. The protocol prevents double spending. The last is that the protocol has deadlock when two vendors request e-coins from each other. This has been rectified by adding a separate process to deal with transfer of e-coins from one to another.

The verification was done based under the assumption that the broker, vendors, and customers adhere to the protocol. There have been few restrictions on the behavior of the customer though, who is able use the same e-coin again and again. Future research involves working on proving validity of e-coins and double spending when the customer and vendors are not cooperative, and when the vendors cannot be fully trusted.

References

1. Cai, Y., Grundy, J., Hosking, J., Dai, X.: Software Architecture Modeling and Performance Analysis with Argo/MTE. In: SEKE 2004 (1990)
2. Chaudhary, K., Dai, X.: P2P-NetPay: An off-line Micro-payment System for Content Sharing in P2P-Networks. JETWI 1(1), 46–54 (2009)
3. Chaudhary, K., Fehnker, A.: Modeling and Verification for the Micropayment Protocol Netpay. In: WASET 2012, vol. 72 (2012)
4. Chaudhary, K., Fehnker, A.: Server-Side Netpay Protocol Models (2015), http://repository.usp.ac.fj/id/eprint/8165
5. Dai, X., Grundy, J.: Architecture for a Component-Based, Plug-In Micro-payment System. In: Zhou, X., Zhang, Y., Orlowska, M.E. (eds.) APWeb 2003. LNCS, vol. 2642, pp. 251–262. Springer, Heidelberg (2003)
6. Dai, X., Grundy, J.: Three Kinds of E-wallets for a NetPay Micro-Payment System. In: Zhou, X., Su, S., Papazoglou, M.P., Orlowska, M.E., Jeffery, K. (eds.) WISE 2004. LNCS, vol. 3306, pp. 66–77. Springer, Heidelberg (2004)
7. Dai, X., Lo, B.: NetPay - An Efficient Protocol for Micropayments on the WWW. In: AusWeb 1999, Australia (1999)
8. Glassman, S., Manasse, M., Abadi, M., Gauthier, P., Sobalvarro, P.: The Millicent Protocol for Inexpensive Electronic Commerce. In: WWW 1995 (December 1995)
9. Hauser, R., Steiner, M., Waidner, M.: Micro-payments Based on ikp. In: SECURICOM 1996. LNCS (1996)
10. Herzberg, A., Yochai, H.: Mini-pay: Charging Per Click on the Web (1996)
11. Nisan, N., London, S., Regev, O., Camiel, N.: Globally Distributed Computation Over the Internet. The POPCORN project. In: ICDCS 1998. IEEE (1998)
12. Rivest, R., Shamir, A.: PayWord and MicroMint: Two Simple Micropayment Schemes. In: Crispo, B. (ed.) Security Protocols 1996. LNCS, vol. 1189, pp. 69–87. Springer, Heidelberg (1997)
13. Sun, J., Liu, Y., Dong, J.: Protocol Analysis Toolkit, http://www.comp.nus.edu.sg/~pat/
14. Wei, K., Smith, A., Chen, Y., Vo, B.: WhoPay: A Scalable and Anonymous Payment System for Peer-to-Peer Environments. In: Distributed Computing Systems. IEEE (2006)
15. Yang, B., Garcia-Molina, H.: PPay: Micro-payments for Peer-to-Peer Systems. In: CSS 2003, pp. 300–310 (2003)
16. Zou, E., Si, T., Huang, L., Dai, Y.: A New Micro-payment Protocol Based on P2P Networks. In: ICEBE 2005 (2005)