Stamping Out Spam

Aashin Gautam
Department of Computer Science
University of Texas at Austin

Advisors: Mohamed Gouda and Benjamin Kuipers
Department of Computer Science
University of Texas at Austin

10 May, 2004

Abstract

We describe a new service of spam control in this paper. This service starts by providing some k e-pennies to each user of a mail server enabling this user to send k emails. Every time an email is sent or received, the balance of e-pennies in the user's account is decremented or incremented respectively. Extra e-pennies can be attained from participating e-banks, which allow users to buy or sell e-pennies.

Table of Contents

1. Introduction

In today's fast-paced environment, email has become the most dominant form of communication. Huge amount of money is spent each year to make the email system faster, more secure and more reliable. This rapid growth of email has been stumbled by the fact that most of today's email servers are getting overloaded with spam. Spam, also known as Unsolicited Commercial Email, has become a big nuisance to Internet providers and users. According to anti-spam vendor Brightmail, over sixty percent of all email traffic in April 2004 was spam. This number was only eight percent in 2001. Thus, spam has grown considerably and congested Internet traffic and bandwidth over the last few years. Current spam solutions are working hard to address the issue, but just as fast as solutions are reached, spammers find new ways to deliver the spam to us.

Spam has also often proved to be offensive in nature. About 15% of all spam is in regard with adult material. This is a big problem in society even though it is involuntary. Other than being a nuisance, spam also has various costs associated with it. These costs include the costs of bandwidth, email servers and people's time. A lot of money also goes into research to fight spam. Most of these costs are borne by the Internet Service Providers (ISPs). This is why, in early April 2003, AOL filed lawsuits against five spammers, who collectively pushed a billion junk email messages through AOL's system. Various state and federal laws have also been passed in order to stop this activity. However, the ease with which an email identity can be attained and changed reduces the chances of finding the individual performing this fraudulent activity.

Spamming is an activity with no incremental cost to the spammer. The spammer can send one million email copies at the same cost as he would send one. Thus, the only way to cut down spamming is to turn around the economic strategy such that the burden of costs falls onto the spammer. There are a lot of questions that arise when you think of this idea. What about everyday users of email – should they pay for email? If so, how much will it cost to send an email?

2. Related Work

Various techniques have previously been used to cut down spam [SCT, OV, F, ABBDW03]. These techniques range from simple filters and email lists to complex algorithms used to decide the credibility of an email. Numerous anti-spam vendors market various sophisticated tools for individual users, ISPs and businesses. However, spammers always find ways to get around each of these.

Spam filters [SCT] are the most common form of protection. The complexities of the filters also range from simple subject line matching filters to highly complex Bayesian filters [OV]. The simple filters are easily tackled by spammers who have tailored their emails to get past them with little effort. Complex Bayesian filters are self learning filters based on input from users. Spammers, however, are constantly changing their spam content thereby making it difficult for such filters to be a step ahead of their adversaries.

Spam filters are also not always correct [F]. Marking a spam email as not spam is surely bad, but marking a non-spam email as spam is worse. This uncertainty of spam filters causes the users to constantly look at messages marked as spam for mistakes. This makes the spam filters useless from the end users perspective.

Another defense against spam is the use of "blacklists". This involves the checking of the sender's 'reputation' with a central database. If a user is blacklisted in a certain domain, it cannot send email to that domain. This is a very harsh approach to controlling spam since what is spam to one user might not be spam to another.

A similar, but less aggressive, approach is the "whitelist". This means that each user specifies who it wants to receive email from. Based on its preferences, those emails from whitelisted senders are delivered without scrutiny. All other email may be analyzed in many ways like filtering.

The key commonality in all the above ideas is the separation of email into two distinct categories: spam and not spam. Any email that is flagged off as spam is treated in a different way than one that is not. Thus, spammers have a specific goal to reach: make their email not look like spam. If they achieve that, they can break all defenses and make their way to the user.

Various economic ideas have also been suggested to make spammers think twice before sending bulk email. Ideas like the Ticket Server [ABBDW03] and Barry Shein's proposal. The Ticker Server proposes to charge a tiny amount for each network service requested, whether it be surfing the Internet or checking email. This is definitely an unreasonable idea from two stand points. Firstly, the indirect cost of assigning a stamp to each and every network service will surely slow down the Internet; and secondly, the direct cost to each user for surfing the Internet.

Another idea to turn the economics of email comes from Barry Shein from The World magazine, who suggested that if every email was charged a fractional cost a solution might be reached. This tiny cost would not mean a lot to the everyday users of the Internet who only send a nominal number of emails everyday, but this would largely affect spammers who send out thousands of emails everyday. Although this idea seems workable, it has not been accepted. After all, why would anybody want to pay money for a service that has been free thus far with no benefit to them? Acceptance of ideas usually requires benefits to the acceptor, and this is what this scheme does not offer.

3. Architecture

3.1 Overview

Our architecture for solving the spam problem revolves around direct benefit to end users. We achieve this by charging users for sending and rewarding them for receiving email. The money earned by the user can be used to send email or can be exchanged for money from a participating bank. Thus, the reward for receiving spam would outweigh the cost of sending email for a normal Internet user.

3.2 Motivation

To make a proposal for any change in a system acceptable, it has to have a two fold benefit. Firstly, it should tackle the problem at hand; and secondly, it should provide an incentive for the end user to make the change.

In this paper, we discuss such a solution for the spam problem. Spam is a common enemy to two parties: the ISPs and the end users. To satisfy both these parties, we must look at the problem from the view point of each of these separately. The ISPs are spending millions of dollars in making the Internet faster and annoyed at the spammers for clogging their networks. End users are irritated because spammers are filling their email boxes to an unbearable extent. We need to satisfy both these parties with our solution in order to make it acceptable.

ISPs would have a direct benefit if the number of spammers decreased. This would free up a lot of bandwidth and reduce a bulk of their costs. End users also need a direct benefit from receiving spam. To an end user, receiving one spam message or ten is just as annoying. Thus, simply blocking out a percentage of the messages is not a feasible solution. Instead, if users are paid to receive this email, they would happily accept email from these spammers.

Thus, our motivation is to put a burden on the spammers thereby being advantageous to the ISPs and end users.

3.3 Properties

We describe a new service of spam control in this paper. This service starts by providing some k e-pennies to each user of a mail server enabling this user to send k emails. Every time an email is sent or received, the balance of e-pennies in the user's account is decremented or incremented respectively. Extra e-pennies can be attained from participating e-banks, which allow users to buy or sell e-pennies. This basic scheme has the following core properties:

Zero-sum:
Any complete transaction in this model is zero-sum i.e. the number of e-pennies decremented on one side is equal to the number of e-pennies incremented on the other. Each transaction involves two sides.

1. When an email is sent, n e-pennies are decremented from the sender's balance and n e-pennies are incremented in the receiver's balance.
2. When e-pennies are bought or sold to an e-bank, then for every n dollars paid to the bank, you receive n e-pennies and for every n e-pennies redeemed, you receive n dollars.

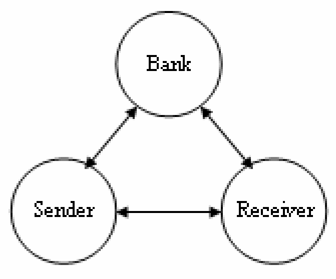
Returnable e-pennies:
When an email is received, the receiver's balance of e-pennies is incremented and the sender's balance is decremented. The receiver then has the option of returning the e-penny if the sender is 'trusted', by sending a reply to the original sender thereby restoring the original balance of e-pennies.

Reusable e-pennies:
The e-pennies attained by receiving email can be reused to send email. Thus, users can set up a small initial balance of pennies and have no need to recharge this balance if they receive enough email.

Convertibility to cash:
The e-pennies attained by receiving email can be converted to cash by trading them at an e-bank. Thus, users are paid directly for their time wasted in dealing with spam.

Invisibility to Users:
Everyday users of email should not see a substantial change in the email system. Thus, we provide f free emails to each user everyday. This allows the user to send f emails without affecting its balance. This f does not carry forward to the next day, thus you cannot save your free emails over time. If the user receives a 'free' email from someone (out of the f free emails the sender has), it does not receive any credit for it.

Using these properties, we design a system involving three entities: the sender, the receiver and the bank. The sender and the receiver are compliant email servers agreeable by the bank.



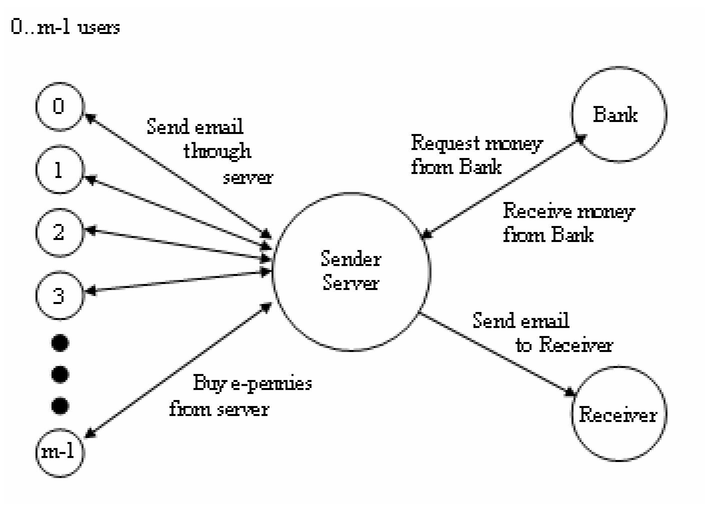
[Figure 1: The 3 entities]

The sender server maintains a deposit which he attains from the bank and which each of its users can use. When this deposit runs out, the server can buy more from the bank. Users are provided some free emails at the start of each day. The server keeps track of the number of emails sent by each user and starts charging the user for every email sent after the limit has been crossed. The server also keeps track of the balance of each user. This balance is attained by the user by buying e-pennies from the server's deposit. Whenever this balance runs out, the user can purchase more from his server. Thus, the communications in the sender server look like Figure 2.

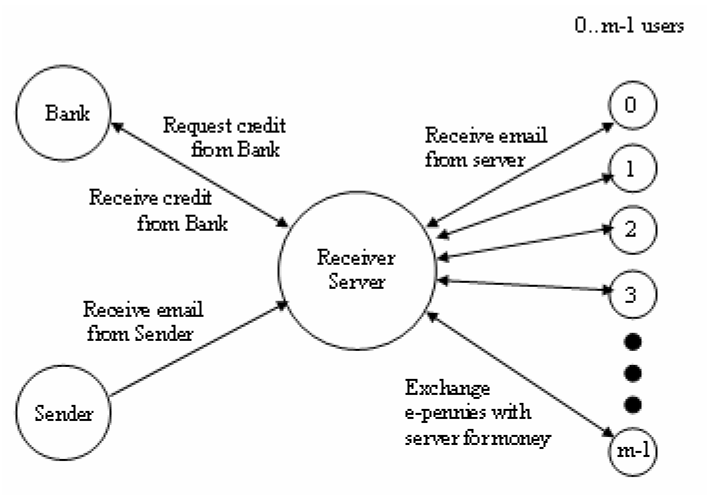
The receiver on the other hand maintains its income. The server keeps track of the number of paid emails received from the sender. When an email is received for a particular user of the server, the following two actions take place:

1. The e-penny attained from the email is added to the income of the server.
2. The server pays the intended receiver (the user) an e-penny from its deposit.

Thus, the zero-sum property is still maintained. Since one e-penny was also reduced from the senders balance.
Whenever the receiving server wants to exchange the deposit for money, it can do so with the bank. Also, whenever a user's balance is high, the user can exchange it with the server for money. Thus, the communications in the receiver server look like Figure 3.

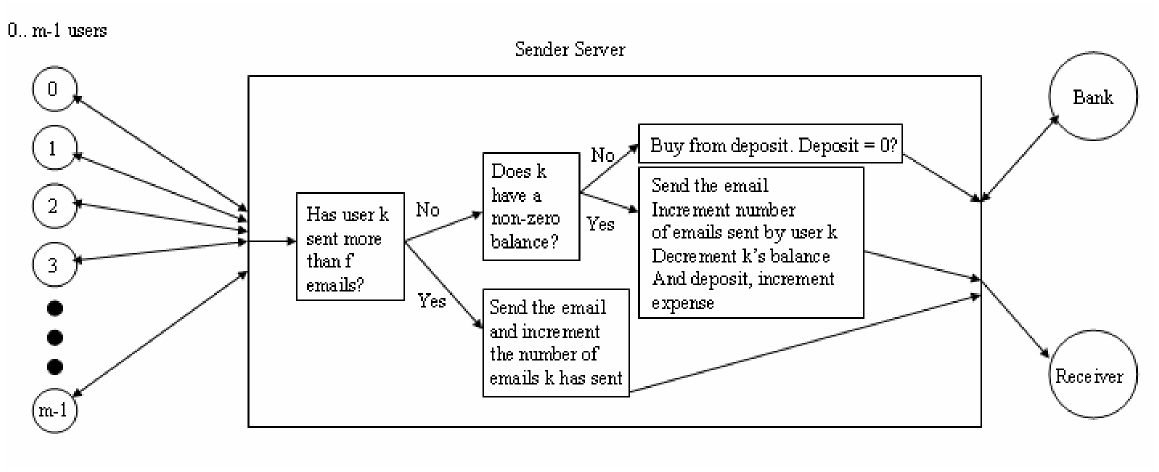


[Figure 2: Communications of the sender server]



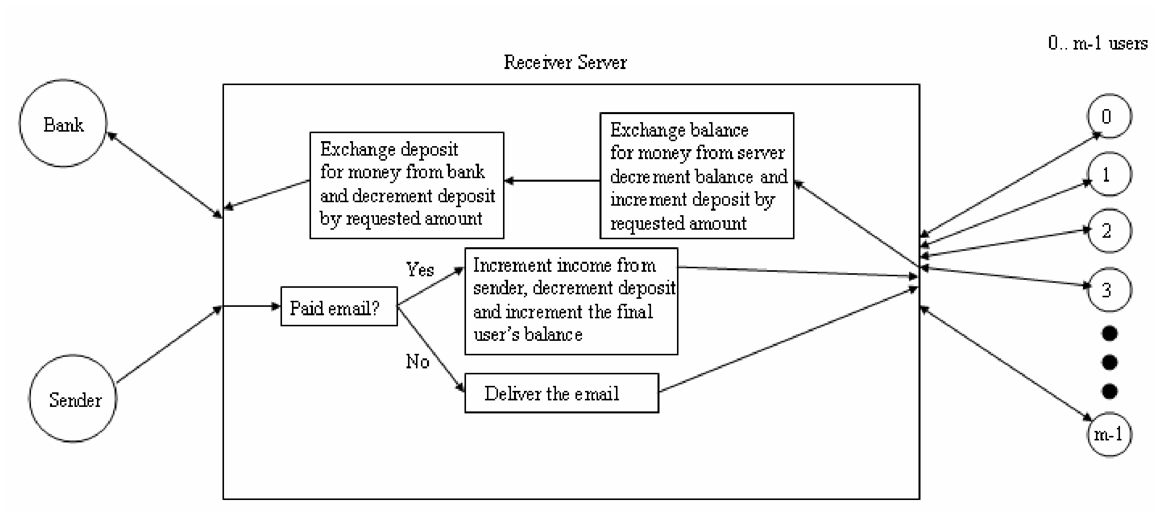
[Figure 3: Communications of the receiver server]

Looking at the sender server in more detail, we have a look at the data it has to manage. The sender needs to keep track of the number of emails sent by its user's everyday and the balance of each user. It also has to maintain a sufficient deposit to allow its users to send email whenever needed. These three variables determine the user's ability to free or paid email. When a user wants to send an email, first the server checks the number of emails sent by that user on that day. If the user is still within its free email limit (f), the email is sent without any changes to its balance. Otherwise, the server checks if the user's balance will allow it to send the email. If the user has sufficient balance, one e-penny is decremented from the user's balance and the email is sent. Also, the server's expense is incremented and its deposit is decremented. Whenever a user's balance is zero, it can obtain e-pennies from its server. When the server wants to increase its balance, it can do so by buying e-pennies from the bank and increasing its deposit. We can see this clearly in Figure 4.


0.. m-1 users
Sender Server
0
1
2
3
m-1
Has user k sent more than f emails?
No
Yes
Does k have a non-zero balance?
No
Yes
Buy from deposit. Deposit = 0?
Send the email Increment number of emails sent by user k Decrement k's balance And deposit, increment expense
Send the email and increment the number of emails k has sent
Bank
Receiver

[Figure 4: Sender server in detail]

The receiver server also has a set of data it must manage. The receiver needs to keep track of the number of emails it receives from the sender server, the balance of each user and the amount of money the server has earned from receiving email. These variables put together decide the ability of a user to earn money. Whenever an email is received, the server checks whether it was a free or a paid email. If it was a free email, no action is taken and the email is delivered to the intended user. If the email is paid, the e-penny earned from that email is added to the income of the server. Now the server needs to pay the intended user an e-penny from its deposit. Thus the user earns an e-penny and the email is delivered to it. The user can exchange these earned e-pennies for money from the server. The server can also exchange its deposit for money from the bank. We can see this clearly in Figure 5.


0.. m-1 users
Receiver Server
Bank
Exchange deposit for money from bank and decrement deposit by requested amount
Exchange balance for money from server decrement balance and increment deposit by requested amount
Yes
Increment income from sender, decrement deposit and increment the final user's balance
Paid email?
No
Deliver the email
Sender
0
1
2
3
m-1

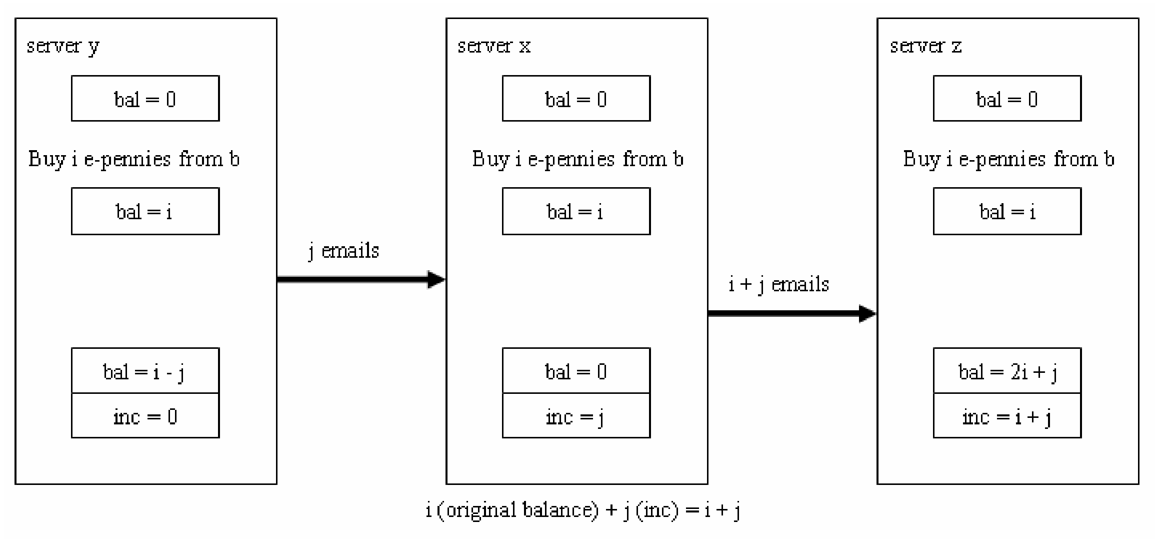
[Figure 5: Receiver server in detail]

Each server in the protocol is also provided with a certificate by the bank. This certificate is encrypted with the private key of the bank and it has the public key of the server and the server's identity. This certificate can only be attained by compliant servers. Thus, a spammer program would not have such a certificate and thus would be unable to send messages to servers using this protocol. The way in which this certificate is used in sending the mail message is discussed in section 4.1.

3.4 The Bank

The bank acts as a governing body over the email servers. It keeps track of the e-pennies bought and sold by servers. Whenever a server buys e-pennies from the bank, the bank makes a record of the purchase for future review. When a server sells e-pennies back to the bank, it also sends to the bank how it got its income, which is recorded by the bank. Any fraudulent activity performed by a server can be tracked down using this information since the bank can now see who has sent emails to a particular mail server and narrow down any misconduct to those two mail servers.

Assume servers x, y, z initially purchase i e-pennies from the bank. Thus, the bank knows that each of the servers' balances is at least i. Now the server x receives j paid emails from server y. Thus, their balances are now i+j and i-j respectively. Then, server x sends i+j paid emails to server z. Balances of the three servers now are 0, i-j and 2i+j respectively. If server z exchanges its e-pennies with the bank, the bank sees its income and notices that z received i+j e-pennies from x. If the banks suspects fraud, it can ask x for its income and balance and check if they, along with the original balance of x, add up to at least i+j. It can also ask for x's expense if needed. If this does not hold, one of the two servers is lying. This is illustrated in Figure 5.



[Figure 5: Fraud Protection]

Thus, the bank acts much like a real bank would in the case of deceitful activity. Instead of pointing a finger at an individual, it narrows down its suspects using the data available and past reputation.

3.5 Assumptions

As mentioned earlier, the sender and receiver servers are compliant with the bank. This means that based on past behavior, servers are divided as either truthful or not truthful. Servers that are not truthful cannot communicate with the truthful servers and vice versa. The bank scrutinizes the servers' behavior and regularly designates servers as truthful or not truthful. Thus, it is in the servers' benefit to behave and obey the rules of the protocol. Any errant behavior causes the server to be flagged as not truthful by the bank thereby removing its privileges to send email to other truthful servers. Also, we assume that the channels of communication between the servers and the bank are fault free. Thus, we discard the possibility of message loss, corruption and replay.

Putting these assumptions together, we can see that the only attacks possible are through adversaries outside of the truthful servers and the bank. Defenses against these attacks are addressed in Section 5.

4. Protocol

We now formalize our ideas using the Abstract Protocol (AP) Notation. First, we look at a simple system with only one server sending email, one server receiving email and the bank. Each of these processes has distinct functions. The sender knows only how to send mail and the receiver knows only how to receive mail. Thus, we separate the two properties of an email server for simplicity and understanding.

Consider a network which has three processes *s*, *r* and *b*. These correspond to the sender, the receiver and the bank respectively. Process *s* and *r* both have *m* users on their server. The processes also have some data used for security purposes. Each process has certain keys used for encryption.

1. Sender and Receiver: *bkb* and *rk* which are the public key of the bank and the server's own private key
2. Bank: *rkb*, *bks* and *bkr* which are the private key of the bank, the public key of the sender and the public key of the receiver.

Other than these, the servers also have variables for nonces which are used to prevent message replay. These variables are *nc* and *nd* in the sender and receiver; and *ncs*, *nds*, *ncr* and *ndr* in the bank.

Each user on the sender's server is allowed to send *f* free emails everyday. Since users are allowed to send these free emails, the sender has an integer array *sent[0..m-1]* which counts the number of emails sent by each user on a given day. The sender also has another integer array *bal[0..m-1]*, which keeps track of the balance of each user and an integer *exp* which keeps track of the servers expenses towards sending email to the other server. Other than maintain the users' data, the sender also maintains an integer *dpst* which is the amount of deposit the server has attained from the bank. When the server requests more e-pennies from the bank, it flags off a boolean variable *rqstd* so that it knows not to ask the bank for more e-pennies until it receives the bank's reply.

```
process s

const   m,                                       {number of users on each server}
        f                                        {free emails per day}

inp     bkb    :    integer                      {public key of bank}
        rk     :    integer                      {private key of s}

var     x, y   :    0..m-1                       {x: receiver ID, y: sender ID}
        bal    :    array [0..m-1] of integer    {balance of each user: init 0}
        exp    :    integer                      {number of pennies sent to each server: init 0}
        dpst    :    integer
        z       :    integer
        rqstd   :    boolean                          {init false}
        sent    :    array [0..m-1] of integer        {init 0 at the start of each day}
        nc, nd  :    integer                          {nonces init 0}

par     k       :    0..m-1                           {users}


begin

        ¬rqstd                      ->   z, rqstd, nc := any, false, NNC;
                                         send rqstm(NCR (bkb, (NCR (rk, z), s, nc))) to b;

        | rcv rplym(z) from b       ->   z, nd := DCR(bkb, (DCR(rk, z)));
                                         if nd > nc
                                                 nc, dpst, rqstd := nd, dpst +  z, true;
                                          | nd <= nc
                                                 skip;
                                         fi

        | bal[k]=0                  ->   z := any;
                                         if z<= dpst
                                                 dpst, bal[k] := dpst - z, z;
                                          | z > dpst
                                                 skip;
                                         fi

        | bal[k]>0 V sent[k]<=f     ->   if sent[k]> f
                                                 sent[k], bal[x], y := sent[k] + 1, bal[x] - 1, any;
                                                 exp, dpst := exp + 1, dpst - 1;
                                                 send mail(k, y, 1) to r;
                                          | sent[k]<=f
                                                 sent[k], y := sent[k] + 1, any;
                                                 send mail(k, y, 0) to r;
                                         fi
end
```

Process *s* has four actions. In the first action, *s* requests the bank for more e-pennies upon checking that it has not already sent the request. In the second action, *s* receives the requested e-pennies from the bank which it adds to its deposit. In the third action, any user of *s* with a zero balance requests the sender for e-pennies. Thus, the server provides the user e-pennies from its deposit. In the forth action, any user of *s* with a sufficient balance or pending free emails sends an email to a user of server *r*. The sender checks how many emails this user has sent on that day and accordingly sends the mail message with or without charging the user.

Process *r* needs to maintain its own set of data too. It maintains its income using integer variable *inc* which it increments each time it receives a paid email from *s*. Process *r* also maintains the balance of each of its user using an array of integers *bal[0..m-1]*. Like the sender, it also users an integer variable *dpst* to maintain its deposit and boolean variable *rqstd* to remember whether it has already asked the bank for credit.

```
process r

const   m                                                   {number of users on each server}

inp     bkb     :       integer                             {public key of bank}
        rk      :       integer                             {private key of r}

var     x, y    :       0..m-1                              {x: receiver ID, y: sender ID}
        inc     :       integer                             {number of pennies from each server: init 0}
        bal     :       array [0..m-1] of integer           {balance of each user: init 0}
        dpst    :       integer
        z       :       integer
        rqstd   :       boolean                             {init false}
        nc, nd  :       integer                             {nonces init 0}

par     k       :       0..m-1                              {users}

begin

        | ¬rqstd                    ->      z := any;
                                            if z<=dpst
                                                    rqstd, nc := true, NNC;
                                                    send
                                                    rqstc (NCR(bkb, (NCR (rk, (z, inc, dpst-z)), r, nc)))
                                                    to b;
                                             | z > dpst
                                                    skip;
                                            fi

        | rcv rplyc(z) from b       ->      z, nd := DCR(bkb, (DCR(rk, z)));
                                            if nd > nc
                                                    nc, rqstd, dpst := nd, false, dpst - z;
                                                    {init inc 0}
                                             | nd <= nc
                                                    skip;
                                            fi

        | true                      ->      z := any;
                                            if z <= dpst
                                                    bal[k], dpst := bal[k] - z, dpst + z;
                                                    {server pays user}
                                             | z > dpst
```

```
                                        skip;
                                    fi

    | rcv mail(x, y, z) from s     ->   {deliver mail to my user x from user y of server s}
                                    bal[y], dpst, inc := bal[y]  + z, dpst  - z, inc  + z;


end
```

Process *r* also has four actions. In the first action, *r* requests the bank for credit on its deposit upon checking that it has not already made this request. In the second action, *r* receives a reply from the bank to its request for credit. On receiving this, *r* decrements its deposit by the amount requested. In the third action, any user of *r* exchanges its excess balance for credit. Thus, the server reduces the user's balance and increases its deposit. In the forth action, *r* receives a mail message from *s*. Upon checking whether the email is paid or not, it increments its income from *s*, decrements its deposit and increments the intended user's balance.

The bank maintains data for both the servers, not the users of these servers. At every message exchange between the servers and *b*, the balance of the server is cross checked. Thus, *b* maintains the balance of *s* and *r* using integers *acnts* and *acntr*. It also uses the security variables mentioned above.

```
process b

inp     rkb      :      integer                 {private key of b}
        bks      :      integer                 {public key of server s}
        bkr      :      integer                 {public key of server r}

var     acnts    :      integer                 {init 0}
        acntr    :      integer                 {init 0}
        inc      :      integer
        l, m     :      integer
        ncs,nds  :      integer                 {nonces for s}
        ncr,ndr  :      integer                 {nonces for r}

par     x        :      0..p-1
        j        :      0..p-1

begin

        rcv rqstm(m) from s     ->      m, j, nds := DCR(rkb, m);
                                        if j != x
                                                {authorization failed}
                                         | nds <= ncs
                                                skip;
                                         | j = x
                                                m := DCR(bks, m);
                                                ncs := nds;
                                                acnts, ncs := acnts  + m, NNC;
                                                send NCR(bks, (NCR(rkb, rplym(m, ncs)))) to s;
                                        fi

        | rcv rqstc(m) from r           ->      m, j, ndr := DCR(rkb, m);
                                        if j != x
                                                {authorization failed}
                                         | ndr <= ncr
                                                skip;
                                         | j = x
                                                m, inc, l := DCR(bkr, m);
                                                ncr := ndr;
                                                {Check for validity of inc}
                                                acntr, ncr := l, NNC;
                                                send NCR(bkr, (NCR(rkb, rplyc(m, ncr)))) to r;
                                        fi

end
```

Process *b* has two actions. In the first action, *b* receives a request for e-pennies from *s*. It validates the message and sends *s* the requested e-pennies. It tallies this with its *acnts* variable. In the second action, *b* receives a request for credit from *r*. Again, it validates this message and sends *s* the requested credits after tallying *r*'s *acntr.*

It only makes sense to put processes *s* and *r* together so that a server can send and receive email. It also makes sense to have more than one such server in the Internet. We achieve this by using process arrays. Each of the servers on the Internet run on the same protocol and thus can be described using the process array *s[0..n-1]* which says that there are *n* such servers. For simplicity we assume that all these servers have the same number of users, *m*.

```
process s[i:0..n-1]

const   n,                                      {number of servers}
        m,                                      {number of users on each server}
        f                                       {free emails per day}

inp     bkb     :       integer                 {public key of bank}
        rk      :       integer                 {private key of i}

var     x, y    :       0..m-1                  {x: receiver ID, y: sender ID}
        inc     :       array [0..n-1] of integer   {number of pennies from each server: init 0}
        exp     :       array [0..n-1] of integer   {number of pennies from each server: init 0}
        bal     :       array [0..m-1] of integer   {balance of each user: init 0}
        dpst    :       integer
        z       :       integer
        rqstd   :     boolean                    {init false}
        sent    :     array [0..m-1] of integer  {init 0 at the start of each day}
        nc, nd  :     integer                    {nonces init 0}

par     j       :     0..n-1                     {servers}
        k       :     0..m-1                     {users}

begin

        ¬rqstd                    ->    z, rqstd, nc := any, false. NNC;
                                        send rqstm(NCR (bkb, (NCR (rk, z), I, nc))) to b;

        | rcv rplym(z) from b     ->    z, nd := DCR(bkb, (DCR(rk, z)));
                                        if nd > nc
                                                dpst, rqstd, nc := dpst +  z, true, nd;
                                         | nd <= nc
                                                skip;
                                        fi

        | bal[k]=0                ->    z := any;
                                        if z<= dpst
                                                dpst, bal[k] := dpst - z, z;
                                         | z > dpst
                                                skip;
                                        fi

        | bal[k]>0 V sent[k]<=f   ->    if sent[k]> f
                                                sent[k], bal[x], y := sent[k] + 1, bal[x] – 1, any;
                                                exp[k], dpst := exp[k] + 1, dpst - 1;
                                                send mail(k, y, 1) to s[j];
                                          | sent[k]<=f
                                                sent[k], y := sent[k] + 1, any;
                                                send mail(k, y, 0) to s[j];
                                        fi

        | ¬rqstd                  ->    z := any;
                                        if z<=dpst
                                                rqstd, nc := true, NNC;
                                                send
                                                rqstc(NCR (bkb,(NCR (rk,(z, inc, dpst-z)), j, nc)))
                                                to b;
                                         | z > dpst
                                                skip;
                                        fi

        | rcv rplyc(z) from b     ->    z, nd := DCR(bkb, (DCR(rk, z)));
                                        if nd > nc
                                                rqstd, dpst, nc := false, dpst – z, nd;   {init inc 0}
                                         | nd <= nc
```

```
                                                    skip;
                                            fi

                | true                    ->        z := any;
                                            if z <= dpst
                                                    bal[k], dpst := bal[k] - z, dpst + z;
                                                    {server pays user}
                                             | z > dpst
                                                    skip;
                                            fi

                | rcv mail(x, y, z) from s[j]   ->  {deliver mail to my user x from user y of server j}
                                            bal[y], dpst, inc[j] := bal[y] + z, dpst - z, inc[j] + z;


end




process b

const   p,                                          {number of servers}
        r                                           {r-1 is the largest value for a key}

inp     rkb     :       0..r-1                      {private key of b}
        bk      :       array [0..p-1] of 0..r-1    {public keys of servers}

var     acnt    :       array [0..p-1] of integer   {init 0}
        inc     :       array [0..p-1] of integer
        l, m    :       integer
        nc, nd  :       array [0..p-1] of integer   {nonces init 0}

par     x       :       0..p-1
        j       :       0..p-1

begin


        rcv rqstm(m) from s[x]    ->        m, j, nd[x] := DCR(rkb, m);
                                            if j != x
                                                    {authorization failed}
                                             | nd[x] <= nc[x]
                                                    skip;
                                             | j = x
                                                    m := DCR(bk[j], m);
                                                    nc[x] := nd[x];
                                                    acnt[x], nc[x] := acnt [x] + m, NNC;
```

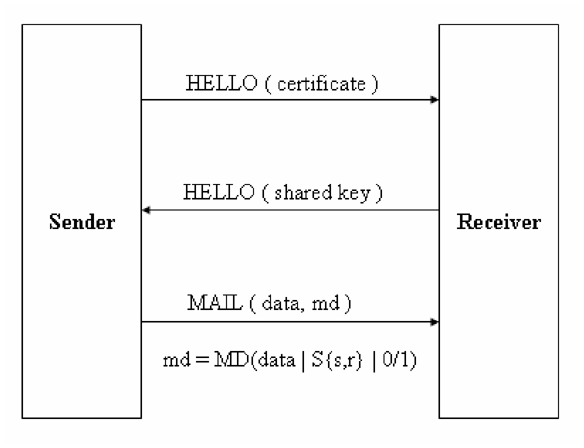
```
                                        send
                                        NCR(bk[j], (NCR(rkb, rplym(m, nc[x])))) to s[x];
                                fi

    | rcv rqstc(m) from s[x]    ->      m, j, nd[x] := DCR(rkb, m);
                                        if j != x
                                                {authorization failed}
                                         | nd[x] <= nc[x]
                                                skip;
                                         | j = x
                                                m, inc, l := DCR(bk[j], m);
                                                nc[x] := nd[x];
                                                {Check for validity of inc}
                                                acnts[x], nc[x] := l, NNC;
                                                send
                                                NCR(bk[j], (NCR(rkb, rplyc(m, nc[x])))) to s[x];
                                        fi

end
```

4.1 The mail Message

The mail message sent by the sender to the receiver is different from the current email protocols. In the mail message the sender needs to:

1. Authenticate itself
2. Tell the receiver whether the email is paid of free

Whenever two servers communicate to send email, the sender first sends the receiver a HELLO message which has the certificate it obtained from the bank attached to the message. Then, the receiver server authenticates the sender by decrypting the certificate using the bank’s public key and checking the server’s public key. The receiver now creates a shared key for the sender and the receiver and sends it back to the sender in another HELLO message. All future communication in this transaction will use this shared key. Now, when the sender sends the data, it does so by appending to the data a message digest of the same data appended with the shared key and a bit 1 or 0 signifying whether the email is paid or free respectively. This transaction is shown clearly in *Figure 6*.



[Figure 6: The mail Message]

5. Security

The above protocol runs on top of the current email protocols. Thus, when one server sends an email to another, we do not need to worry about those messages getting delivered correctly. Also, according to our assumptions, the servers and the channels of communications are fault free and thus we do not need to worry about cheating by servers or errors in message delivery by the channels. Our biggest enemy in this scheme would be an adversary who steals, modifies or inserts messages from the channel.

To defend against adversary action between the servers and the bank, we use encryption using asymmetric keys. Two keys K and L are asymmetric iff K != L. Each server in the system has a public and private key. The bank has a list of public keys of each email server. The bank also has a public and private key. Each email server knows the public key of the bank.

Let *b* be the bank and *s* be one email server. Let *B.b* and *R.b* be the public and private keys of the bank. Let *B.s* and *R.s* be the public and private keys of the email server. Now, whenever the server communicates with the bank, it first encrypts the message with its private key and then with the public key of the bank.

encrypted_message = (NCR (B.b, (NCR (R.s, (message)))))

In order to decrypt this message, one needs the private key of the bank and the public key of the email server that sent the message.

message = (DCR (R.b, (DCR (B.s, (encrypted_message)))))

Since an adversary would not know the private key of the bank, this message could not be intercepted and read. Also, since the adversary does not know the private key of the email server that sent the message, it cannot insert such a message into the channel.

Another potentially dangerous adversary action is message replay. If the adversary intercepts a message from an email server to the bank asking for credit and sends that message to the bank multiple number of times, the server will be paid more than it had asked for.

In order to protect the system from this attack we use nonces. Nonces are constantly increasing positive integers which we attach with each message. Every message received from a particular email server must have a nonce attached with it which has a value higher than the previous nonce. If not, the message is discarded. Thus, in the above example of encrypting messages, the format for the message would include a positive integer (nonce) attached by the server and the bank whenever a message is sent.

encrypted_message = (NCR (B.b, (NCR (R.s, (message, nonce)))))

These methods of defense against adversary action protect us from messages that have no credibility. Using these schemes, we can be assured that the message we receive is indeed from whom we expect it to be.

An adversary can also send email by inserting a message in the channel between two servers without paying for it. To avoid this, each server in the protocol is provided with a certificate by the bank. This certificate is encrypted with the private key of the bank and it has the public key of the server and the server's identity. This certificate can only be attained by compliant servers. Thus, a spammer program would not have such a key thus would be unable to send messages to servers using this protocol.

6. Conclusion

The protocol mentioned in this paper revolves around reversing the economics of email such that the bulk of the cost for sending spam falls on the spammers. The individual whose time and effort is wasted in receiving and reading spam is now compensated. The ISP that spends incredible amount of money to keep the Internet fast is also satisfied that the people who are using the most bandwidth are being charged for it. The Internet is not a free advertising machine and thus spammers are liable for their usage.

This scheme does not promise to stamp out spam, because what is spam for one person is important information for another. Instead, this scheme makes the spammers think twice before sending bulk email. Spam should only be sent to individuals who are interested in knowing about that subject. This idea forces spammers to send email to only those people who are actually going to read it. Thus, junk mail will still be around, but you will only receive the junk you asked for.

7. Acknowledgments

I would like to sincerely thank Dr. Mohamed Gouda, Dr. Benjamin Kuipers and Alex Liu for their support and help in guiding me through writing this paper and for giving me this opportunity.

8. References

[ABBDW03] Abadi, M., Birrell, A., Burrows, M., Dabek, F., Wobber, T., Bankable Postage for Network Services, 2003

[SCT] Soonthornphisaj, N., Chajkulseriwat, K., Tang-On, P., Anti-Spam Filtering: A Centroid-Based Classification Approach

[F] Fawcett, T., "In vivo" spam filtering: A challenge problem for KDD

[OV] O'Brien, C., Vogel, C., Spam Filters: Bayes vs. Chi-squared; Letters vs. Words