Bankable Postage for Network Services

Martín Abadi[1], Andrew Birrell[2], Mike Burrows[3], Frank Dabek[4], Ted Wobber[2]

[1]University of California, Santa Cruz, CA
[2]Microsoft Research, Silicon Valley, CA
[3]Google, Mountain View, CA
[4]MIT Laboratory for Computer Science, Cambridge, MA

Abstract. We describe a new network service, the "ticket server". This service provides "tickets" that a client can attach to a request for a network service (such as sending email or asking for a stock quote). The recipient of such a request (such as the email recipient or the stockbroker) can use the ticker server to verify that the ticket is valid and that the ticket hasn't been used before. Clients can acquire tickets ahead of time, independently of the particular network service request. Clients can maintain their stock of tickets either on their own storage, or as a balance recorded by the ticket server. Recipients of a request can tell the ticket server to refund the attached ticket to the original client, thus incrementing the client's balance at the ticket server. For example, an email recipient might do this if the email wasn't spam. This paper describes the functions of the ticket server, defines a cryptographic protocol for the ticket server's operations, and outlines an efficient implementation for the ticket server.

1. Motivation

Several popular network services today have the characteristic that the person using the service doesn't pay for it, even though the provider of the service incurs real costs. In a variety of cases this largesse has led to abuse of the services. The primary example is email, where the result is the spam business. Other examples include submitting URL's to a web indexing service, or creating accounts at a free service such as Yahoo or Hotmail, or even in some cases reading a web site (which can be abused, for example, by parsing out the web site's contents and presenting them as part of another site, without permission).

To appreciate the scale and difficulty of the spam problem, consider some statistics. In February 2003 AOL reported that their spam prevention system was detecting, and suppressing, 780 million incoming spam emails per day for their 27 million users [5]. On August 4th 2003 Hotmail, with 158 million active user accounts, received 2.6 billion emails (plus 39 million from within Hotmail). 2.1 billion (78%) of the emails were mechanically classified as spam, roughly 24,000 per second. That same day Hotmail also had more than 1 million new accounts created—most likely

those were not all for normal users. Today, spam is extremely cheap for the sender. If you search at Google for "bulk email" you will find organizations willing to deliver your email to one million addresses (provided by them) for a total cost of $190 (i.e., 0.019 cents per message). Accordingly, there is a lot of current work on techniques for deterring spam [4, 9, 12, 13, 15, 20].

This paper is about one technology for preventing such abuse, which we call the "ticket server". For ease of exposition in this paper, and because spam was our primary motivation, we will mostly describe the application of this technology to email, though in general it applies equally well to other network services. To read what we say in a more general form, replace "email" with "network service"; replace "sender" with "client requesting the service"; replace "recipient" or "recipient's ISP" with "service provider"; and replace "email message" with "service request".

As was originally pointed out by Dwork & Naor [11], and subsequently used in several systems such as HashCash [6] and Camram [8], we can potentially reduce the abuse of email by forcing senders to attach to the email proof that the sender has performed a lengthy computation as part of the process of sending the email.

Straightforward computation is not the only plausible cost that we might impose on a sender. We might use a function that's designed to incur delays based on the latency of memory systems [2, 10] — this is more egalitarian towards people with low-powered computers. We might hope to force the sender to consult a human for each message by using a Turing test [3]. We might also rely on real money, and use a proof-of-purchase receipt as the proof. In all these cases, after the cost has been incurred the sender can assemble proof that it has been incurred. In this paper we call any such proof a "ticket". Our design is essentially independent of the particular form of cost that the ticket represents.

In all these schemes, it is critical that the sender can't use the same ticket for lots of messages. The way that this has been achieved in the past is by making the email message itself be involved in the creation of the proof. For example, you could require that the computation be parameterized by the message date, recipient, and some abstract or digest of the message body.

Unfortunately, this means that the sender must incur the cost after composing the message and before committing the message to the email delivery system. The sending human must wait for the cost to be incurred before knowing that the message has been sent. For example, the sender could not disconnect from the network until after the computation completes. In addition to causing an unfortunate delay, this synchronization limits how long a computation we can require, and thus limits the economic impact of the computational cost on spammers.

The ticket server design was created to avoid this problem. By introducing a stateful server, we allow the sender to acquire tickets independently of a particular email message. Instead, the ticket server maintains a database of tickets issued. When a recipient receives the email, he calls the ticket server to verify that the ticket is not being reused, and to update the ticket server's database to prevent subsequent reuse. The operation of the ticket server is reminiscent of how postage stamps work.

Introduction of a stateful server allow us to provide three key benefits:

(1) *Asynchrony:* senders can incur the cost (for example, perform the computation) well before composing or sending the email (perhaps overnight). This makes the whole mechanism much less intrusive on users' normal workflows, and therefore more likely to be acceptable to the non-spamming user community. It also makes it reasonable to demand much higher computational costs, since a non-spam sender can incur the cost at a convenient time.

(2) *Stockpiles:* since the tickets are now independent of what they'll be applied to, users can maintain a stockpile of tickets for future use. They could also acquire such a stockpile from elsewhere (for example, bundled with their purchase of email software or bundled with signing a contract to use a certain ISP, or presented to them by recipients who would welcome their email).

(3) *Refunds:* having a stateful server allows us to introduce the notion of an "account" for a user. This provides a small convenience for the sender, as a way for managing the user's stockpile of tickets. But it also enables a new feature: if a recipient receives a ticket with an email from a sender, and the recipient decides that the email didn't need to be paid for, then the recipient can refund the ticket to the sender, by telling the ticket server to do so.

Refunds are potentially a powerful tool. If we can arrange that most tickets on non-spam email will be refunded, then most non-spamming users will end up having most of their tickets refunded, and have little need to acquire new ones. In that case it would be reasonable to make tickets even more expensive, and thereby make it even more likely that we can price the tickets in a way that will increase the sender's cost for sending spam to a level comparable to (or even exceeding) that of physical junk mail. We will consider the feasibility of this level of refunding in section 3.

The remainder of the paper is as follows. We provide an overview of our design in section 2. In section 3 we show how the ticket server can be applied to the particular case of spam reduction. Section 4 outlines a secure protocol for the ticker server operations. Section 5 describes an efficient implementation of the ticket server. Section 6 discusses the issues that would arise from trying to deploy the ticket server and apply it to spam reduction. Finally, section 7 discusses some related work, and we summarize our conclusions in section 8.

2. Overview

In its simplest form, the ticket server provides two operations to its customers: "Request Ticket" and "Cancel Ticket". A third operation, "Refund Ticket", is also useful.

The "Request Ticket" operation has no arguments. It returns a "ticket kit" (assuming the requestor has no account balance, see below). The ticket kit specifies one or more "challenges". The challenges can take a variety of forms, such as

computing a hard function (paying for the ticket by CPU or memory cycles), or passing a Turing test, or paying for the ticket by credit card (in some separate transaction not described here). The requestor decides which challenge to accept and does whatever it takes to respond to the challenge. The requestor can then take the response, and the rest of the ticket kit, and assemble them into a new, valid ticket. (Details of how this is achieved are described below).

The requestor can use this ticket, once, in any manner he chooses (for example, he can transmit it as a header line in an email message). Someone receiving a ticket (for example, the recipient of the email) can take the ticket and invoke the ticket server's "Cancel Ticket" operation. This operation verifies that the ticket is valid, and that it hasn't been cancelled before. It then records in the ticket server's database that the ticket has been cancelled, and returns to the caller (for example, the recipient of the email) indicating that the operation succeeded. Of course, the "Cancel Ticket" operation will return a failure response if the ticket is invalid or has previously been cancelled.

Finally, whoever received the ticket and successfully performed the "Cancel Ticket" operation can choose to refund the ticket to the originator by invoking the "Refund Ticket" operation at the ticket server. This causes the ticket server to credit the ticket to the original requestor's account, by incrementing the requestor's account balance. The ticket server also records in its database the fact that this ticket has been refunded. Of course, the ticket server rejects a "Refund Ticket" request for a previously refunded ticket.

When a requestor whose account has a positive balance calls the "Request Ticket" operation, instead of a ticket kit (with a challenge) he receives a new, valid, unused ticket, and his account balance is decremented.

The ticket server is designed to guarantee that tickets cannot be forged; that a ticket can be cancelled only if it's valid and hasn't previously been cancelled; and that a ticket can be refunded only after it has been cancelled, and only if authorized by the principal who cancelled the ticket, and at most once. We describe later a cryptographically protected protocol to achieve these guarantees.

As will be seen in the detailed protocol description, the use of tickets provides some additional benefits. The ticket identifies the requestor by an anonymous account identifier created by the ticket server. In addition to the ticket itself, the requestor is given a ticket-specific encryption key; the requestor is free to use this key to protect material that is shipped with the ticket, since the same key is returned to the caller of the "Cancel Ticket" operation.

3. Application to Spam Reduction

To use the ticket server for spam reduction, an email recipient (or his ISP) arranges that he will see only messages that either have a valid ticket attached, or are from a "trusted sender". This restriction can be implemented in the receiving ISP's mail

servers, or by the recipient's email viewing program. There is a wide range of options for the notion of "trusted sender", which we'll discuss later in this section.

3.1. Basic Scenario

The figure below shows the basic scenario for using tickets for spam prevention, in the absence of trusted senders. We describe later what happens if the ticket is omitted, and/or the sender is trusted.

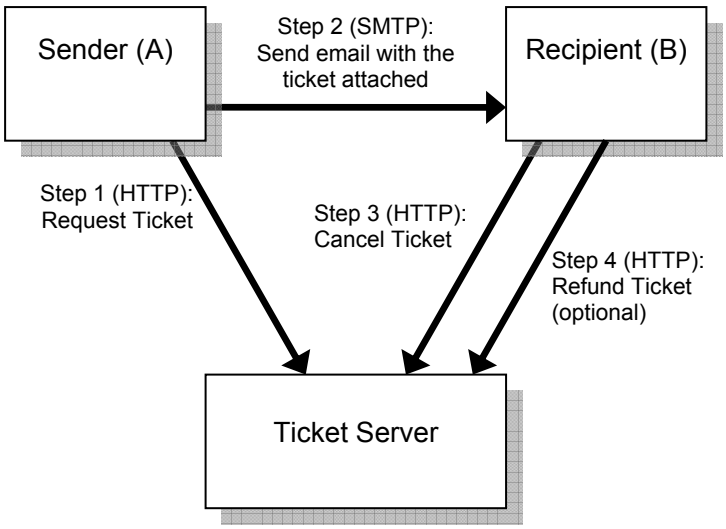

Fig. 1. The basic scenario for using tickets for spam prevention.

The prospective sender acquires a ticket kit by interacting with the ticket server by HTTP, constructs a ticket from the kit by responding to its challenge, and sends the ticket to the recipient with the email message by SMTP. The recipient (or the recipient's ISP) validates and cancels the ticket with the assistance of the ticket server by HTTP, and optionally uses HTTP once more to refund the ticket. It is of course possible for a frequent sender to acquire many ticket kits in a single HTTP request, and for a busy ISP to verify, cancel, and optionally refund many tickets in a single HTTP request.

3.2. Who is a Trusted Sender?

We envisage several different ways of categorizing a sender as "trusted". Some senders should probably be trusted on a per-recipient basis. A user might reasonably decide to trust everyone in his address book. He might additionally trust everyone to whom he has previously sent email. There might also be a mechanism for a user to

add recipients to a "safe list" exempted from spam prevention. (The safe list would be stored by the ISP if the ISP was responsible for checking tickets, or by the user's email viewing program otherwise.) More broadly a user or his ISP might decide that some groups of senders are to be trusted. For example, if the ISP has a strong (and enforced!) policy of canceling spammers' accounts, then email from within the same ISP should be trusted. It might also be wise to trust senders from other ISP's with similarly acceptable anti-spam policies.

There is of course an issue about trusting senders, given the fact that most SMTP transactions do not authenticate the asserted sender name. The ISP can no doubt assure itself that asserted senders within the ISP's own domain name space occur only on messages originating within the ISP, and that those are authentic. However, there is always some suspicion about unauthenticated sender names arriving from outside. There are reasonable part-way measures available, though. For example, an ISP could accept as authentic a sender in "aol.com" if and only if the SMTP server transmitting the message is owned by AOL, as proved by reverse IP address lookup in DNS (and similarly for Hotmail, Yahoo, and numerous others). Overall we believe that with care it is possible to have a reasonable degree of confidence in the authenticity of a very large proportion of sender names. There are other groups currently exploring ideas for stronger authentication of sender names in email (see section 7.3).

3.3. Variations of the Basic Scenario

We can now explain what happens in all the variations of the basic scenario: messages with or without tickets, from trusted or untrusted senders. In the following, the actions described as being performed by the recipient's ISP could equally well be performed by the recipient's email program — either form of deployment would work.

Variation 1: Trusted Sender, but No Ticket

When a message arrives with no ticket attached, but from a trusted sender, the message appears in the recipient's inbox in the usual way. The ticker server is not involved at all.

Variation 2: Trusted Sender with a Ticket Attached

When a message arrives with a ticket attached from a trusted sender, it appears in the recipient's inbox in the usual way. Additionally, the ticket server is told that it should refund the ticket, crediting the sender's account. This step can be taken automatically by the ISP, and requires no explicit action from the recipient. In this case the "Cancel Ticket" and "Refund Ticket" operations at the ticket server can be combined for efficiency.

Variation 3: Untrusted Sender with a Ticket Attached

When a message arrives with a ticket attached from an untrusted sender, the recipient's ISP calls the ticket server's "Cancel Ticket" operation, to verify and cancel the ticket. If the ticket is invalid or previously cancelled, the message is silently discarded. Otherwise, it appears in the recipient's inbox. When the recipient sees the message, if the recipient decides that the sender should indeed pay for the message, he need do nothing more. However, if the recipient decides that this message wasn't spam, the recipient can choose to call the ticket server's "Refund Ticket" operation, to refund the ticket's value to the sender. (Note that there is an interesting human-factors decision to be made here: should "Refund Ticket" require an explicit action from the user, or should it be the default, which can be overridden by the user classifying the message as spam?)

Variation 4: Untrusted Sender and No Ticket

When a message arrives without a ticket attached and from an untrusted sender, the ISP might choose to respond in one of two ways. First, the ISP might treat the message as suspicious, and flag it but nevertheless deliver it to the recipient.

Alternatively, the ISP could hold the message (but invisibly to the recipient) and send a bounce email to the sender. The bounce email would offer the sender two choices: he can provide some previously acquired ticket, or he can acquire a new ticket by interacting with the ticket server.

In the case where the sender chooses to use a previously acquired ticket, he simply provides it to the ISP by passing it over HTTP to the ISP (perhaps through an HTML form provided as part of the bounce message). On receipt of this, the ISP calls the "Cancel Ticket" operation to verify and cancel the ticket, and provided this succeeds, makes the message available to the recipient's inbox.

Alternatively, if the sender wants to acquire a new ticket at this time, he must call the ticket server. To simplify doing so, the bounce email contains a link (URL) to the ticket server. Clicking on the link performs a "Request Ticket" operation at the ticket server. The result appears to the sender as a web page describing the available challenges. For example, for the computational challenge the web page will contain a link that would cause the sender to perform the computation (via an ActiveX control or a Java applet, perhaps). As another example, the web page resulting from the "Request Ticket" operation could include a Turing test such as those used by the Captcha system [3]. In either case, the result of the challenge is combined with the ticket kit data (also on the web page), and the resulting ticket is passed via HTTP to the recipient's ISP. The ISP now proceeds as if the message had originally arrived with the ticket attached, verifying and canceling the ticket and delivering the message to the recipient.

If a message remains in the "held" state too long without the sender responding to the bounce message, it is silently discarded by the ISP. The same happens if the

sender failed to provide an appropriate return address, or if the sender responds to the bounce message with an invalid ticket.

4. Protocol

This section describes a specific protocol for the ticket server's operations. The description is fairly abstract, and deliberately leaves a lot of flexibility: in how the data is represented and transported (for example, through HTML pages and HTTP in the spam scenario described above), in how the cryptography is implemented, and in some semantic choices (such as whether to encrypt data in transit or just protect it with a message authentication code or MAC [17]). While these are all important design decisions for an actual implementation, they are largely irrelevant to the overall ticket server concept. We do describe how to achieve appropriately transactional semantics in the presence of communication failures and retransmissions.

The following datatypes and messages implement the ticket server's three operations. "Request Ticket" is message (1) and its response (2); "Cancel Ticket" is message (4) and its response (5); "Refund Ticket" is message (6) and its response (7). The participants in the protocol are the ticket server itself, client "A" who requests a ticket, and client "B" who receives the ticket from A and uses it in messages (4) through (7). In the case of email spam prevention, A is the sender (or perhaps the sender's ISP) and B is the recipient's ISP (or perhaps the recipient mail user-agent software).

The functions and values involved are as follows. See section 5.3 for further discussion of how the items related to the challenge (P, X, F, and C) are represented.

- S is a unique identifier for a ticket (in practice, a sequence number issued by the ticket server).
- K_T, K_A, K_B, and K_S are secret keys (for ticket T, for A, for B, and for the ticket server).
- I_A identifies A (and K_A) to the ticket server.
- I_B identifies B (and K_B) to the ticket server.
- $TransID_A$ is an identifier chosen by A to identify a "Request Ticket" transaction.
- $TransID_B$ is an identifier chosen by B to identify a "Use Ticket" transaction.
- H(D) is a secure hash of some data D (such as might be obtained by applying the SHA1 algorithm); H(K, D) is a keyed secure hash of D using a key K.

- K(D) uses key K to protect some data D in transit. This might provide a secure MAC for D, and/or it might encrypt D using K, and/or it might prove the timeliness of D by including a secured real-time clock value.
- P is a Boolean predicate. It occurs in a ticket kit (see "TK", below), where it specifies a particular, ticket-specific, challenge. It will be represented by an integer or a URL.
- X is the answer to a challenge. It will be represented by an integer or short string. If a particular ticket kit contains predicate P, and if P(X), then X is an appropriate value for constructing a valid ticket from the ticket kit. In other words, the challenge for A is to find an X such that P(X) is true.
- F is a function that the ticket server will use in verifying that a ticket has a correct value of X. It will be represented by an integer. Note that F is visible to the ticket requestor.
- C is a secret that assists the ticket server in verifying a ticket's X value. It will be represented by an integer or short string. For any valid ticket, F(X) = C.
- M is a message, or other request for a service that might require a ticket.

The ticket server maintains the following state, in stable storage:

- Its own secret key, K_s.
- The largest value of S that it has ever issued.
- State(S) = "Issued" or "Cancelled", for each ticket S that has been issued (subject to a maximum ticket lifetime, not specified in this paper).
- Balance(I_A) = an integer, the account balance for A's account.
- Result($TransID_A$) = result of the most recent "Request Ticket" operation that contained $TransID_A$ (maintained by the ticket server only for recently requested tickets, then discarded).
- Canceller(S) = $TransID_B$, the identifier used by B in a recent "Use Ticket" request for ticket S (maintained by the ticket server only for recently used tickets, then discarded).
- Refunded(S) = a Boolean indicating whether the ticket S has been refunded to an account (subject to a maximum ticket lifetime, not specified in this paper).

The following are trivially derived from other values:

- TK = (S, P, F, I_A, H(K_s, ("Hash for T", S, F, C, I_A))), a ticket kit.
- T = (S, X, F, I_A, H(K_s, ("Hash for T", S, F, C, I_A))), a ticket issued to A.
- T.S is the S used in forming T; similarly for the other components of T, and for components of TK.

- $K_T = H(K_S, (\text{“Hash for KT”}, T.S))$, the requestor's secret key for T.
- $K_T' = H(K_S, (\text{“Hash for KT prime”}, T.S))$, the canceller's secret key for T.
- $K_A = H(K_S, (\text{“Hash for KA”}, I_A))$, the secret key identified by I_A.
- $K_B = H(K_S, (\text{“Hash for KB”}, I_B))$, the secret key identified by I_B.

A ticket T is "valid" if and only if T.S has been issued by the ticket server, and $H(K_S, (T.S, T.F, Y, T.I_A)) = H_T$, where H_T is the keyed hash in T and Y = T.F(T.X). Note that this definition includes valid but cancelled tickets (for ease of exposition).

The ticket server creates each ticket kit TK in such a way that the ticket constructed by replacing TK.P with X, for any X for which TK.P(X) is true, will be a valid ticket. Thus, A should find an X such that P(X) is true. This arrangement turns out to be highly flexible, and usable for a wide variety of challenges. Moreover, this arrangement has the property that the ticket server does not need to compute X — it only verifies it. This property is important in the case where computing X is hard, for example when doing so is expensive in CPU time or memory cycles. See the commentary section for discussion and some examples.

When client A wishes to acquire a new ticket, it chooses a new $TransID_A$, and calls the ticket server:

(1) A → ticket server: I_A, K_A(“Request”, $TransID_A$)

The ticket server uses I_A to compute K_A, and verifies the integrity and timeliness of the message (or else it discards the request with no further action). Now, there are three possibilities:

- If $Result(TransID_A)$ is already defined, then it is left unaltered.
- If $Balance(I_A) > 0$, then it is decremented and $Result(TransID_A)$ is set to a new valid ticket T such that State(T.S) = "Issued". Note that in this case the sender does not need to deal with a challenge.
- If $Balance(I_A) = 0$, then $Result(TransID_A)$ is set to a ticket kit TK for a new valid ticket, such that State(TK.S) = "Issued". Note that the ticket server does not compute the response to the challenge implicit in TK: it's up to A to do that.

The ticket server computes K_T from TK.S or T.S, and sends it and $Result(TransID_A)$ to A:

(2) ticket server → A: K_A(“OK”, $Result(TransID_A)$, K_T, $TransID_A$)

A verifies the integrity of this message (or else discards it). Note that if message (2) is lost or corrupted, A can retransmit (1), causing the ticket server to retransmit an identical copy of (2). If the result is a ticket kit TK, not a complete ticket, then A can take TK, solve the challenge by determining some X such that P(X), and assemble the complete ticket T from the elements of TK.

When A wants to use T to send B the message M (or other request for service), A sends:

(3) A → B: T, K_T(M)

Note that B does not yet know K_T. B now asks the ticket server to change the state of T.S to "Cancelled". B chooses a new $TransID_B$, and sends:

(4) B → ticket server: I_B, K_B("Cancel", T, $TransID_B$).

The ticket server verifies the integrity of this message (or discards it). If T is not "valid" (as defined above), or if State(T.S) = "Cancelled" and Canceller(T.S) is not $TransID_B$, then the result of this call is "Error". Otherwise, the ticket server sets the state of T.S to "Cancelled", sets Canceller(T.S) to $TransID_B$, computes K_T, and sends it back to B:

(5) Ticket server → B: K_B(("Error" | ("OK", K_T, K_T')), $TransID_B$).

B verifies the integrity of this message (or else discards it). Note that if message (5) is lost or corrupted, B can retransmit (4), causing the ticket server to retransmit an identical copy of (5). B can now use K_T to verify the integrity of (3), and to extract M from it if it was in fact encrypted with K_T. The key K_T' will be used to authenticate B if B decides to refund the ticket.

In the spam-prevention application, when B is the recipient's ISP, the ISP will now proceed to make the email visible to the recipient. If the recipient decides that M should be accepted without payment, then the recipient (or the recipient's ISP) tells the ticket server to recycle this ticket:

(6) B → ticket server: T.S, K_T'("Refund", T).

The ticket server verifies the integrity of this message (or discards it). Note that in doing so it computes K_T' from T.S, so the verification will succeed only if B truly knew K_T'. If Refunded(T.S) is false, the ticket server sets Refunded(T.S) and increments Balance($T.I_A$). Regardless of the previous value of Refunded(T.S), the ticket server then reports completion of the operation:

(7) Ticket server → B: T.S, K_T'("OK").

Some aspects of the protocol call for further explanation:

- The transaction ID in step (1) allows for the error case where the response (2) is lost. Because the ticket server retains Result($TransID_A$), A can retransmit the request and get the ticket that he's paid for (in the case where A's account was decremented) without needing to pay again. The key K_A authenticates A, so that nobody else can acquire tickets charged to A's account.
- Since I_A is embedded in T and in the secure hash inside T, it authenticates the ticket as having been issued to A (or more precisely, to a principal that knows I_A and K_A).
- The key K_T returned to A in step (1) allows A to construct K_T(M); in other words, it authorizes A to use T for a message of A's choosing.

- When B receives (3), B does not know K_T so B cannot reuse T for its own purposes. B acquires K_T only after canceling the ticket.
- The transaction ID used in request (4) allows B to retransmit this request if the response is lost. Because the ticket server has recorded Canceller(T.S), it can detect the retransmission and return a successful outcome even though the ticket has already been cancelled in this case.
- If the result (7) is lost, B can retransmit the request. Because of the ticket server's "Refunded" data structure, this will not cause any extra increment of A's account balance.
- The use of K_T' in messages (6) and (7) authenticate the principal authorized to re-use T. Only the principal that cancelled T can credit A's account. It's fine for A to do this himself, but if he does so he can't use T for any other purpose – it's been cancelled.

5. Implementation

We have implemented the ticket server, and applied it in a prototype spam-prevention system. In this section we outline the techniques that we used, which provides a simple, robust, and highly efficient server. At the end of this section we discuss the performance of this design.

5.1. Ticket State

The ticket server uses sequential serial numbers when creating the field S for a new ticket. The ticket server maintains in stable storage an integer which indicates the largest serial number that it has used in any ticket. It's easy to implement this efficiently, if you're willing to lose a few serial numbers when the ticket server crashes. The ticket server maintains in volatile memory the largest serial number that has actually been used (S_{used}), and in stable storage a number always larger than this (S_{bound}). When enough serial numbers have been used that S_{used} is approaching S_{bound}, the ticket server rewrites S_{bound} with the new value S_{used}+K, where K is chosen to make the number of stable storage writes for this process insignificant (K = 1,000,000 would suffice). If the ticket server crashes, up to K serial numbers might be lost. In no case will serial numbers be reused.

The ticket server maintains 2 bits of state for each ticket S (with $S \leq S_{used}$). One bit is set iff the ticket has been cancelled, and the other is set iff the ticket has been refunded. The "truth" for these bits necessarily resides on disk, and the server must read it from there after any cold restart. However, during normal running the ticket server can maintain this in volatile memory with an array indexed by ticket serial number.

When a ticket changes state (from "issued" to "cancelled" or from "cancelled" to "refunded") the ticket server updates this array, and synchronously with the operation records the transition in stable storage. All it needs to record is the ticket serial number, and the new state.

5.2. Ticket Request and Cancellation Logs

It is possible that the ticket server will crash in the vicinity of responding to a "Request Ticket" operation. However, the worst that this does to A is to cause him to lose a single ticket, which we believe is acceptable. We saw no need, in this application, to keep a log in stable storage that would allow A to replay his request after the server restarts.

The ticket server does need to maintain a volatile data structure containing Result($TransID_A$). However, this is only relevant for as long as it might receive a transmission of the corresponding "Request Ticket" operation. The ticket server can apply a relatively short limit to this time interval, to keep the data structure small.

5.3. Representing the Challenge

The protocol values P, X, F, and C are used to represent the challenge and the proof that A has responded appropriately, and to assist the ticket server in verifying this response. It's a somewhat complex mechanism, because it's designed to deal with a variety of challenge styles (computational, Turing test, perhaps others). Here are some examples of how these values can be used.

In all cases F is represented by a small integer, indicating to the ticket server what it should do to verify that a ticket has an appropriate value of X, i.e., that F(X) = C.

For a computational test, P describes the test. It can be represented by a small integer P_i and a parameter P_n. The small integer selects a function P_f to be computed, from a list known to the participants in the protocol. The challenge for A is to find a value X such that $P_f(P_n, X) = 0$. For this case, C is always 0 and the ticket server implements F(X) by computing $P_f(P_n, X)$.

For a Turing test that requires recognizing a word from a distorted image, P specifies the image. We have not implemented this case, but it seems straightforward. We could use the image itself as P, but more likely we would use a URL at which the image resides. The challenge for A is to find a string X such that X is embedded in the image. For this case C is the correct string, and the ticket server implements F(X) by comparing the strings X and C.

5.4. Performance

Although we have not fully tuned our implementation, we believe that this ticket server design, implemented on a single inexpensive PC, could handle as many ticket operations as could be accommodated on a 100 Mb/sec network connection.

- In terms of memory usage, this implementation uses approximately 2 bits of main memory for each outstanding uncancelled ticket. So each GByte of DRAM suffices to keep the state for 4 billion tickets, roughly one week's email traffic at Hotmail (after we ignore the messages that Hotmail already categorizes as spam, and before allowing for the fact that email messages from trusted senders don't need tickets).
- In terms of disk performance, each operation (Request, Cancel, and Refund) requires a disk write. However, this doesn't need to overload the disk channel. We use a simple "group commit" design to keep the delays caused by the disk transfers negligible. To commit one of the operations we need to write only 16 bytes to disk: 8 bytes for S and 8 bytes for either ID_A (for Request or Refund) or $TransID_B$ (for Cancel). If the operation waits until any previous disk request has completed, and meanwhile we accumulate the transaction records for multiple operations into a 4 KByte buffer, we can record 256 operations with a single 4 KByte disk write. Writing 4 KBytes takes about 16 milliseconds even on an ATA disk, allowing us to record about 16,000 ticket transactions per second on a single disk. So while each individual operation incurs the latency of a 4 KByte disk request, the overall throughput of the ticket server is not limited by the disk.
- In terms of CPU performance, our current implementation is still unsatisfactorily slow. Most of the time is spent in the overheads of communicating through our TCP stack, which has not yet been optimized for this application. At the moment, the protocol and its cryptography are not significant in limiting the server's performance.

Of course, full deployment of this design would require multiple ticket servers, and arranging cooperation between them is a significant design problem. We discuss this, briefly, in the next section.

6. Deployment Issues

If everyone's email system used the ticket server, it seems highly likely that we would have solved the spam problem. We can tune the effective cost of sending email so as to increase it, in real dollar terms, to a level comparable to physical mail: a few cents per message instead of 0.019 cents. This would clearly cause the demise of the more absurd sorts of spam, since the economic model would no longer support mailings with extremely low response rates. As with physical mail, a small level of junk traffic would necessarily remain. However, there are several problems with the ticket server idea, and any of them might prevent it succeeding. We address them next.

6.1. User Acceptance

Certainly any scheme based on making the sender incur costs will make email marginally less pleasant for normal users. The question here is how large that margin is. We believe it's small. Two aspects of the design encourage us in this belief. First, most senders will fall into the category of trusted senders for almost all of their recipients: in general we send email to people who know us. We'll be in their address book, in the same organization as them, or they'll accept our email as being non-spam and they'll refund our ticket. The net effect is that most users will actually consume very few tickets. If we start everyone out with a moderate stockpile (for example, 1000 tickets bundled with a software purchase), they most likely will never be inconvenienced by needing to perform a lengthy computation.

There is a separate issue about users of low-powered devices: PDA's, phones, and old computers. One possibility we have already mentioned is use of memory-bound functions instead of CPU-bound computations. Since memory speed is relatively uniform across devices, there is not a lot of disparity in the cost of tickets. Additionally, email access from PDA's and phones is always through an intermediary with whom the user has a service contract. It would be perfectly reasonable for that intermediary to provide a moderate number of tickets to each user as part of that contract.

The most intrusive part of the scheme occurs in a transition period, where some recipients are requiring use of the scheme and some senders have not yet adopted it. This will produce irritating bounce messages demanding tickets. This is a problem of "you can't get there from here", and might well be a fatal flaw (although it's one shared by virtually all other anti-spam technologies apart from ever-smarter filtering rules). The most plausible way to avoid the flaw would be to get the scheme distributed, but disabled, in a wide range of email software, and once it's widely deployed then starting to actually use it. We don't know if this is feasible.

6.2. Trust

The scheme depends on everyone having some level of trust in the ticket server. While it's easy to see that, for example, a Hotmail user would be willing to trust a ticket server run by Microsoft, it's much less clear why anyone else should. So while an organization such as Hotmail could unilaterally start requiring tickets, it is less likely that any one organization could provide a universal ticket service (nor would they want to).

Equally, if several organizations (Hotmail, AOL, Yahoo, etc.) independently started using ticket servers, senders would be distressed at the complexity of acquiring all the appropriate forms of ticket. It would be like the postal system without the international agreements on mutual acceptance of foreign postage stamps.

So it is more likely that a set of ticket servers would be run cooperatively several organizations, with contractual agreements about accepting each other's tickets. This is less unlikely than it might seem: there are some extremely large, but highly

competitive, players in the email business (AOL, Hotmail, Yahoo, and MSN). A system run jointly by them would have a reasonably appearance of neutrality (at least within the U.S.), good credibility, and would automatically be trusted by a very large community of users. There would be many technical (and business) issues in how such a federation would work, and we do not explore them in this paper.

6.3. Reliability

Introduction of the ticket server adds a new failure mode to the email system. No doubt we could build a ticket server that had negligibly few programming bugs, but of course that isn't enough. Certainly we would need multiple servers, for tolerance of physical and environmental failures, and to handle the total load. Some such replication would come from a federation created to handle the trust issues, but probably more would be needed.

A more worrisome problem is that the ticket server would be an appealing target for a denial-of-service attack. But the large email services are already that appealing. It's not terribly hard to engineer the ticket server to be able to handle anything that its network connection can handle, and if we assume that the servers are co-resident with existing large data centers such as those of Hotmail and AOL, then the problem reduces to the (admittedly unsolved) one of protecting those data centers.

Finally, note that the ticket server need not be essential, and the consequences of a successful attack need not be dire. If a sender cannot contact the ticket server, and his local stockpile of tickets is empty, he can send the email anyway and just hope that he doesn't get a "ticket needed" bounce message. If a recipient cannot contact the ticket server, he can just read the email regardless.

6.4. Cheating

The ticket server protocol is designed to prevent obvious attacks, like forging or re-using tickets. There is one behavior pattern that might be considered at least "misuse": ticket farming. A spammer might, for example, acquire tickets by running a popular web site (such as a gamer site or a porn site) and requiring that customers perform the challenge part of ticket construction before being rewarded by the web site (for example, being allowed to play or to see a picture). This isn't really a violation of the protocol. We make the sender incur a cost, but fundamentally there's no way we can prevent the sender delegating the required work. A similar criticism applies to using Turing tests: there are, unfortunately, parts of the world where labor is so cheap as to make the cost of the tests exceedingly small.

7. Related Work

There are several other systems aimed at causing bulk emailers to incur costs when sending their email. The earliest proposal we know is that of Dwork and Naor [11], which we used as one of the starting points for the current work. Actual deployed systems of this sort include Back's HashCash system [6]. The Camram system [8] adds mechanisms that in effect automate the accumulation of a safe list. None of these systems provide the benefits of the ticket server that we outlined in section 1 of this paper.

The ticket server is reminiscent of several other classes of system, although in aggregate it has distinctly different properties than any of them. There are authentication systems that work by issuing "tickets"; there are numerous systems aimed at offering and enforcing small-scale payments ("micro-payments"); there are other systems for incurring cost when sending email; and there are other systems intended to reduce spam. We consider here a few examples from those classes of system.

7.1. Authentication & Authorization Systems

Systems such as Kerberos [16] produce "tickets" as the result of an authentication. These tickets, like those from the ticket server, enable the clients to perform some operations (such as accessing a file system or a network service). However, the functionality of the tickets is quite different: there is no notion of tickets being consumed or cancelled, nor being refunded. The only way in which tickets become invalid is by expiry after a timeout (or conceivably through a revocation mechanism).

Note that the clients of the ticket server, despite being identified by an ID, can be anonymous. We verify that a client is the same client as one who made a previous request, but we have no knowledge whatsoever of the client's identity. Clients can casually acquire or discard identities.

Our tickets are also reminiscent of capability systems such as Amoeba [19], except that they are explicitly single-use and the ticket server enforces this. We are unaware of any capability system that provides such a feature. Note that the enforcement of single-use necessarily produces a protocol structure involving more communication with the ticket server (just as a revocation mechanism does in capability systems).

It is conceivable that one could extend Kerberos or Amoeba to make them into tools for preventing spam and similar abuses, much like the ticket server, but the extensions would be quite substantial, and it seems preferable to start from scratch rather than attempt to build a more complex system on the prior substrates.

7.2. Micro-Payment Systems

Micro-payment systems come closest in functionality to the ticket server. They achieve very similar goals: the purchaser incurs a cost. It might conceivably be

feasible to take an existing micro-payment system and rework it to use some form of virtual currency instead of real cash. But again, we believe that we will produce a more satisfactory system by building it from scratch with the present applications in mind. The differences are substantial, as we show next.

One such system was Millicent [14], which is a convenient example (partly because the authors are familiar with it, but primarily because it was developed into a real, deployed system with transactions involving real cash). In Millicent, as in the ticket server, there is protection against double spending. However, to avoid excessive interactions with an online authority, that's done by making currency ("scrip" in Millicent terminology) vendor-specific. For an application such as spam prevention, this would correspond to having scrip specific to the recipient's ISP. The ticket server, by providing a recipient-independent online verification system, avoids this. Also, the simplicity of the ticket server protocols and data structures (and the fact that an ISP can batch its requests to the ticket server) makes using the ticket server as an online authority acceptably efficient for our applications.

There are numerous other issues that spring to mind when considering the use of a straight micro-commerce system. For example, sending email from your email account at your employer to your personal account at home would in effect steal money from your employer.

7.3. Email Systems with Stronger Semantics

There are many email enhancements that provide stronger semantics than basic RFC822 and SMTP email. For example, one can use PGP, S/MIME, and many other elaborate systems to provide features such as certified, authenticated, or secure email. In particular, properly authenticated email would most likely be a powerful tool for reducing spam, since recipients would at least be able to filter out email from repeat offenders or groups. But authentication is not by itself a solution: knowing the author of an email message isn't sufficient to classify the message as being spam or not. Authentication does not address the basic problem, namely that the sender incurs too small a cost when sending email, with the result that vast quantities of worthless email flood our inboxes. (See [1, 7, 21, 22] for some design possibilities in this area, and for numerous further references.)

8. Conclusion

The ticket server is a new tool that we can use to control or limit the use of otherwise free and open services, such as email. By adding a shared, stateful server to the earlier proposals, we get significant benefits. In particular, the cost to be incurred for a service can be independent and prior to any particular requests. Moreover, clients may have account balances and may receive refunds. Therefore, we can escalate the cost of the service to a level where we can be certain that it will deter excessive use.

We have designed a protocol that will allow the ticket server to guarantee the correct functioning of the tickets, and prevent or make impractical any attempts to cheat by forging, stealing, or otherwise maltreating tickets. We have also designed and implemented a concrete mechanism to invoke this protocol with a combination of email messages and HTTP requests. Furthermore, we have outlined how the ticket server can be implemented simply and cheaply.

There remain numerous questions about the ticket server. As discussed earlier, deployment of the service raises several tricky issues. Many of the same issues arise with other schemes for email payment: in all cases, deployment is difficult, involving fundamental and disruptive changes to the way that Internet email works. It's not at all clear that we can achieve such changes. On the other hand, removing spam would be a significant financial benefit to the major email service providers, and a convenience for all of us.

9. Acknowledgements

The work described here arose from discussions with Cynthia Dwork and Mark Manasse about computational techniques for spam prevention. Most of Martín Abadi's and Mike Burrows's work was done at Microsoft Research, Silicon Valley (where the techniques described in this paper were invented), with Microsoft's support. Martín Abadi's work is also partly supported by the National Science Foundation under Grant CCR-0208800.

References

1. Abadi, M., Glew, N., Horne, B. and Pinkas, B., Certified Email with a Light On-Line Trusted Third Party: Design and Implementation. In *Proceedings of the Eleventh International World Wide Web Conference* (May 2002), 387-395.
2. Abadi, M., Burrows, M., Manasse, M. and Wobber, E., Moderately Hard, Memory-bound Functions. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium* (February 2003), 25-39.
3. Von Ahn, L., Blum, M., Hopper, N. and Langford, A, Captcha. At http://www.captcha.net.
4. Androutsopoloulos, A. *et al*. An experimental comparison of naïve Bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of SIGIR-00, 23rd ACM International Conference on Research and Development in Information Retrieval.* ACM Press (2000), 160-167.
5. AOL press release on Business Wire, as reported by the Washington Post http://sites.stockpoint.com/wpost/newspaperbw.asp?dispnav=&Story=20030220/051b1753.xml (Feb. 2003)
6. Back, A., *HashCash*. http://www.cyberspace.org/~adam/hashcash (1997).

7. Bahreman, A. and Tygar, J.D., Certified electronic mail. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security* (February 1994), 3-19.
8. *Camram*. At http://www.camram.org.
9. Cranor, L. and LaMacchia, B., Spam!. *Commun. ACM 41,* 8 (August 1998), 74-83.
10. Dwork, C., Goldberg, A. and Naor, M., On memory-bound functions for fighting spam. In *Proceedings of CRYPTO 2003* (to appear).
11. Dwork, C. and Naor, M., Pricing via processing or combating junk mail. In *Advances in Cryptology – CRYPTO '92*. Springer (1992), 139-147.
12. Fahlman, S., Selling interrupt rights: a way to control unwanted e-mail and telephone calls. *IBM Systems Journal 41,* 4 (2002), 759-766.
13. Gabber, E. *et al*, Curbing junk e-mail via secure classification. In *Financial Cryptography,* (1998), 198-213.
14. Glassman, S., Manasse, M., Abadi, M., Gauthier, P. and Sobalvarro, P., The Millicent protocol for inexpensive electronic commerce. In *Proceedings of the Fourth International World Wide Web Conference*. O'Reilly and Associates (1995), 603-618.
15. Ioannidis, J., Fighting spam by encapsulating policy in email addresses. In *Proceedings Symposium on Network and Distributed Systems Security 2003* (February 2003) 17-24.
16. Kohl, J. and Neuman, C., The Kerberos network authentication service. At http://www.rfc-editor.org/rfc/rfc1510.txt.
17. Menezes, A., Van Oorschot, P. and Vanstone, S. *Handbook of applied cryptography*. CRC Press (1996).
18. McCurley, K., Deterrence measures for spam, slides presented at the RSA conference. http://www.almaden.ibm.com/cs/k53/pmail/ (January 1998).
19. Mullender, S. and Tannenbaum, A., The design of a capability-based operating system. *Computer Journal 29*, 4 (August 1986), 289-299.
20. Templeton, B., *E-stamps*. http://www.templetons.com/brad/spume/estamps.html (undated)
21. Tygar, J.D., Yee, B. and Heintze, N., Cryptographic Postage Indicia. In *ASIAN 1996, Lecture notes in computer science 1179* (1996), 378-391.
22. Zhou, J. and Gollmann, D., Certified electronic mail. In *Computer Security– ESORICS '96 proceedings*. Springer-Verlag (1996), 160-171.