Crowds: Anonymity for Web Transactions

Michael K. Reiter
and
Aviel D. Rubin
AT&T Labs—Research

In this paper we introduce a system called Crowds for protecting users' anonymity on the world-wide-web. Crowds, named for the notion of "blending into a crowd", operates by grouping users into a large and geographically diverse group (crowd) that collectively issues requests on behalf of its members. Web servers are unable to learn the true source of a request because it is equally likely to have originated from any member of the crowd, and even collaborating crowd members cannot distinguish the originator of a request from a member who is merely forwarding the request on behalf of another. We describe the design, implementation, security, performance, and scalability of our system. Our security analysis introduces *degrees of anonymity* as an important tool for describing and proving anonymity properties.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*applications*; K.4.1 [**Computers and Society**]: Public Policy Issues—*privacy*; K.4.4 [**Computers and Society**]: Electronic Commerce—*security*

General Terms: Security

Additional Key Words and Phrases: anonymous communication, world-wide-web

1. INTRODUCTION

> *Every man should know that his conversations, his correspondence, and his personal life are private.* — Lyndon B. Johnson, president of the United States, 1963–69

The lack of privacy for transactions on the world-wide-web, or the Internet in general, is a well-documented fact [Brier 1997; Miller 1997]. While encrypting communication to and from web servers (e.g., using SSL [Hickman and Elgamal 1995]) can hide the content of the transaction from an eavesdropper (e.g., an Internet service provider, or a local system administrator), the eavesdropper can still learn the IP addresses of the client and server computers, the length of the data being exchanged, and the time and frequency of exchanges. Encryption also does little to protect the privacy of the client from the server. A web server can record the Internet addresses at which its clients reside, the servers that referred the clients to it, and the times and frequencies of accesses by its clients. With additional effort, this information can be combined with other data to invade the privacy of clients even further. For example, by automatically `finger`ing the client computer shortly after an access and comparing the idle time for each user of the client computer with the server access time, the server administrator can often deduce the exact user with high likelihood. Some consequences of such privacy abuses are described in [Miller 1997].

In this paper we introduce a new approach for increasing the privacy of web transactions and a system, called *Crowds*, that implements it. Our approach is based on the idea of "blending into a crowd", i.e., hiding one's actions within the actions of many others. To execute web transactions in our model, a user first joins a "crowd" of other users. The user's request to a web server is first passed to a random member of the crowd. That member can either *submit* the request directly to the end server or *forward* it to another randomly chosen member, and in the latter case the next member chooses to submit or forward independently. When the request is eventually submitted, it is submitted by a random member, thus preventing the end server from identifying its true initiator. Even crowd members cannot identify the initiator of the request, since the initiator is indistinguishable from a member that simply forwards a request from another.

In studying the anonymity properties provided by this simple mechanism, we introduce the notion of *degrees* of anonymity. We argue that the degree of anonymity provided against an attacker can be viewed as a continuum, ranging from no anonymity to complete anonymity and having several interesting points in between. We informally define these intermediate points, and for our Crowds mechanism described above, we refine these definitions and prove anonymity properties for our system. We expect these definitions and proofs to yield insights into proving anonymity properties for other approaches, as well.

An intriguing property of Crowds is that a member of a crowd may submit requests initiated by other users. This has both negative and positive consequences. On the negative side, the user may be incorrectly suspected of originating that request. On the positive side, this property suggests that the mere availability of Crowds offers the user some degree of deniability for her observed browsing behavior, if it is possible that she was using Crowds. Moreover, if Crowds becomes widely adopted, then the presumption that the computer from which a request is received is the computer that originated the request will become decreasingly valid (and thus decreasingly utilized).

The anonymity provided by Crowds is subject to some caveats. For example, Crowds obviously cannot protect a user's anonymity if the content of her web transactions reveals her identity to the web server (e.g., if the user submits her name and credit card number in a web form). More subtley, Crowds can be undermined by executable web content that, if downloaded into the user's browser, can open network connections directly from the browser to web servers, thus bypassing Crowds altogether and exposing the user to the end server. In today's browsers, such executable content takes the form of Java applets and ActiveX controls. Therefore, when using Crowds, it is recommended that Java and ActiveX be disabled in the browser, which can typically be done via a simple preferences menu in the browser.

The rest of this paper is structured as follows. In Section 2, we more precisely state the anonymity goals of our system and introduce the notion of *degrees* of anonymity. This gives us sufficient groundwork to compare our approach to other approaches to anonymity in Section 3. We describe the basic Crowds mechanism in Section 4 and analyze its security in Section 5. We describe the performance and scalability of our system in Sections 6 and 7, respectively. We discuss crowd membership in Section 8, the system's user interface in Section 9, and the obstacles that firewalls present to wide scale adoption of Crowds in Section 10. We conclude in Section 11.

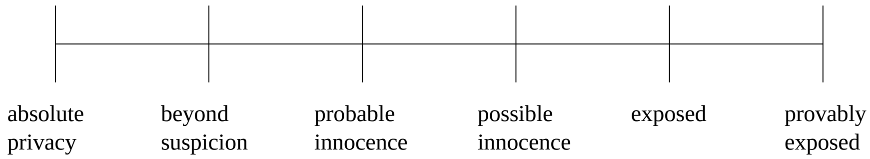


Fig. 1. **Degrees of anonymity:** Degrees range from *absolute privacy*, where the attacker cannot perceive the presence of communication, to *provably exposed*, where the attacker can prove the sender, receiver, or their relationship to others.

2. GOALS

2.1 Anonymity

As discussed in [Pfitzmann and Waidner 1987], there are three types of anonymous communication properties that can be provided: sender anonymity, receiver anonymity, and unlinkability of sender and receiver. *Sender anonymity* means that the identity of the party who sent a message is hidden, while its receiver (and the message itself) might not be. *Receiver anonymity* similarly means that the identity of the receiver is hidden. *Unlinkability of sender and receiver* means that though the sender and receiver can each be identified as participating in some communication, they cannot be identified as communicating *with each other*.

A second aspect of anonymous communication is the attackers against which these properties are achieved. The attacker might be an eavesdropper that can observe some or all messages sent and received, collaborations consisting of some senders, receivers, and other parties, or variations of these [Pfitzmann and Waidner 1987].

To these two aspects of anonymous communication, we add a third: the *degree* of anonymity. As shown in Figure 1, the degree of anonymity can be viewed as an informal continuum. For simplicity, below we describe this continuum with respect to sender anonymity, but it can naturally be extended to receiver anonymity and unlinkability as well. On one end of the spectrum is *absolute privacy*: absolute sender privacy against an attacker means that the attacker can in no way distinguish the situations in which a potential sender actually sent communication and those in which it did not. That is, sending a message results in no observable effects for the attacker. On the other end of the spectrum is *provably exposed*: the identity of a sender is provably exposed if the attacker cannot only identify the sender of a message, but can also prove the identity of the sender to others.

For the purposes of this paper, the following three intermediate points of this spectrum are of interest, listed from strongest to weakest.

—**Beyond suspicion**: A sender's anonymity is beyond suspicion if though the attacker can see evidence of a sent message, the sender appears no more likely to be the originator of that message than any other potential sender in the system.

—**Probable innocence**: A sender is probably innocent if, from the attacker's point of view, the sender appears no more likely to be the originator than to not be the originator. This is weaker than beyond suspicion in that the attacker may have reason to expect that the sender is more likely to be responsible than any other potential sender, but it still appears at least as likely that the sender is not responsible.

—**Possible innocence**: A sender is possibly innocent if, from the attacker's point of view, there is a nontrivial probability that the real sender is someone else.

It is possible to describe these intermediate points for receiver anonymity and sender/receiver unlinkability, as well. When necessary, we define these intermediate points more precisely.

Which degree of anonymity suffices for a user obviously depends on the user and her circumstances. Probable innocence sender anonymity should prevent many types of attackers from *acting* on their suspicions (therefore avoiding many abuses, e.g., cited in [Miller 1997]) due to the high probability that those suspicions are incorrect. However, if the user wishes to avoid any suspicion whatsoever—including even suspicions not sufficiently certain for the attacker to act upon—then she should insist on beyond suspicion sender anonymity.

The default degree of anonymity on the web for most information and attackers is *exposed*, as described in Section 1. All recent versions of Netscape Navigator and Internet Explorer are configured to automatically identify the client computer to web servers, by passing information including the IP address and the host platform in request headers.

2.2 What Crowds achieves

As described in Section 1, our system consists of a dynamic collection of users, called a *crowd*. These users initiate web requests to various web servers (and receive replies from them), and thus the users are the "senders" and the servers are the "receivers". We consider the anonymity properties provided to an individual user against three distinct types of attackers:

—A **local eavesdropper** is an attacker who can observe all (and only) communication to and from the user's computer.

—**Collaborating crowd members** are other crowd members that can pool their information and even deviate from the prescribed protocol.

—The **end server** is the web server to which the web transaction is directed.

The above descriptions are intended to capture the full capabilities of each attacker. For example, collaborating members and the end server cannot eavesdrop on communication between other members. Similarly, a local eavesdropper cannot eavesdrop on messages other than those sent or received by the user's computer. A local eavesdropper is intended to model, e.g., an eavesdropper on the local area network of the user, such as an administrator monitoring web usage at a local firewall. However, if the same LAN also serves the end server, then the eavesdropper is effectively global, and we provide no protections against it.

The security offered against each of these types of attackers is summarized in Table 1 and justified in the remainder of the paper. As indicated by the omission of an "unlinkability of sender and receiver" column from this table, our system serves primarily to hide the sender or receiver from the attacker. In this table, n denotes the number of members in the crowd (for the moment we treat this as static) and $p_f > 1/2$ denotes the probability of forwarding, i.e., when a crowd member receives a request, the probability that it forwards the request to another member, rather

Table 1. Anonymity properties provided by Crowds

Attacker	Sender anonymity	Receiver anonymity
local eavesdropper	exposed	$P(\text{beyond suspicion}) \underset{n\to\infty}{\longrightarrow} 1$
c collaborating members, $n \geq \frac{p_f}{p_f - 1/2}(c+1)$	**probable innocence** $P(\text{absolute privacy}) \underset{n\to\infty}{\longrightarrow} 1$	$P(\text{absolute privacy}) \underset{n\to\infty}{\longrightarrow} 1$
end server	**beyond suspicion**	N/A

than submitting it to the end server. (p_f is explained more fully in Section 4.) The boldface claims in the table—i.e., probable innocence sender anonymity against collaborating members and beyond suspicion sender anonymity against the end server—are guarantees. The probability of beyond suspicion receiver anonymity against a local eavesdropper, on the other hand, only increases to one asymptotically as the crowd size increases to infinity. Put another way, if the local eavesdropper is sufficiently lucky, then it observes events that expose the receiver of a web request, and otherwise the receiver is beyond suspicion. However, the probability that it views these events decreases as a function of the size of the crowd. Similarly, a sender's assurance of absolute privacy against collaborating members also holds asymptotically with probability one as crowd size grows to infinity (for a constant number of collaborators). Thus, if the collaborators are unlucky, users achieve absolute privacy. We provide a more careful treatment of these notions in Section 5.

Of course, against an attacker that is comprised of two or more of the attackers described above, our system yields degrees of sender and receiver anonymity that are the minimum among those provided against the attackers present. For example, if a local eavesdropper and the end server to which the user's request is destined collaborate in an attack, then our techniques achieve neither sender anonymity nor receiver anonymity. Another caveat is that all of the claims of sender and receiver anonymity in this section, and their justifications in the remainder of this paper, require that neither message contents themselves nor *a priori* knowledge of sender behavior give clues to the sender's or receiver's identity.

2.3 What Crowds does not achieve

Crowds makes no effort to defend against denial-of-service attacks by rogue crowd members. A crowd member could, e.g., accept messages from other crowd members and refuse to pass them along. In our system, such denial-of-service can result from malicious behavior, but typically does not result if (the process representing) a crowd member fails benignly or leaves the crowd. As a result, these attacks are detectable. More difficult to detect are active attacks where crowd members substitute wrong information in response to web requests that they receive from other crowd members. Such attacks are inherent in any system that uses intermediaries to forward unprotected information, but fortunately they cannot be utilized to compromise anonymity directly.

3. RELATED WORK

There are two basic approaches previously proposed for achieving anonymous web transactions. The first approach is to interpose an additional party (a *proxy*) between the sender and receiver to hide the sender's identity from the receiver. Examples of such proxies include the Anonymizer (`http://www.anonymizer.com/`) and the Lucent Personalized Web Assistant [Gabber et al. 1997] (`http://lpwa.com`). Crowds provides protection against a wider range of attackers than proxies do. In particular, proxy-based systems are entirely vulnerable to a passive attacker in control of the proxy, since the attacker can monitor and record the senders and receivers of all communication. Our system presents no single point at which a passive attack can cripple all users' anonymity. In addition, a proxy is typically a single point of failure; i.e., if the proxy fails, then anonymous browsing cannot continue. In Crowds, no single failure discontinues all ongoing web transactions.

A second approach to achieving anonymous web transactions is to use a *mix* [Chaum 1981]. A mix is actually an enhanced proxy that, in addition to hiding the sender from the receiver, also takes measures to provide sender and receiver unlinkability against a global eavesdropper. It does so by collecting messages of equal length from senders, cryptographically altering them (typically by decrypting them with its private key), and forwarding the messages to their recipients in a different order. These techniques make it difficult for an eavesdropper to determine which output messages correspond to which input messages. A natural extension is to interpose a *sequence* of mixes between the sender and receiver [Chaum 1981]. A sequence of mixes can tolerate colluding mixes, as any single correctly-behaving mix server in the sequence prevents an eavesdropper from linking the sender and receiver. Mixes have been implemented to support many types of communication, for example electronic mail (e.g., [Gulcu and Tsudik 1996]), ISDN service [Pfitzmann et al. 1991], and general synchronous communication (including web browsing) [Syverson et al. 1997].

The properties offered by Crowds is different from those offered by mixes. As described above, Crowds provide (probable innocence) sender anonymity against collaborating crowd members. In contrast, in the closest analog to this attack in typical mix systems—i.e., a group of collaborating mix servers—mixes do not provide sender anonymity but do ensure sender and receiver unlinkability [Pfitzmann and Waidner 1987]. Another difference is that mixes provide sender and receiver unlinkability against a global eavesdropper. Crowds does not provide anonymity against global eavesdroppers. However, our intention is for a crowd to span multiple administrative domains, where the existence of a global eavesdropper is unlikely. Another difference is that mixes typically rely on public key encryption, the algebraic properties of which have been exploited to break some implementations [Pfitzmann and Pfitzmann 1990].

Crowds' unique properties admit very efficient implementations in comparison to mixes. With mixes, the length of a message routed through a mix network grows proportionally to the number of mixes through which it is routed, and the mix network must pad messages to fixed lengths and generate decoy messages to foil traffic analysis. Moreover, in a typical mix implementation, routing a message through a sequence of n mixes incurs a cost of n public key encryptions and n private key decryptions on the critical path of the message, which are comparatively expensive operations. Thus, since the unlinkability provided by mixes is tolerant of up to $n-1$ mixes colluding, increasing n improves anonymity but hurts performance. Privacy in Crowds can similarly be enhanced by increasing the average number of times a request is forwarded among members before being submitted to the end server, but this should impact performance less because there are no public/private key operations, no inflation of message transmission lengths (beyond a small, constant-size header), and no decoy messages needed.

Another performance advantage of Crowds is that since each user actively participates in the function of the crowd, the throughput of a crowd grows as a function of the number of users. In fact, we show in Section 7 that a crowd can scale almost limitlessly (in theory), in the sense that the load on each user's computer is expected to remain roughly constant as new users join the crowd. With a fixed network of mixes, the load of each server increases proportionally to the number of users, with a resulting linear decrease in throughput.

4. CROWD OVERVIEW

As discussed previously, a crowd can be thought of as a collection of users. A user is represented in a crowd by a process on her computer called a *jondo* (pronounced "John Doe" and meant to convey the image of a faceless participant). The user (or a local administrator) starts the jondo on the user's computer. When the jondo is started, it contacts a server called the *blender* to request admittance to the crowd. If admitted, the blender reports to this jondo the current membership of the crowd and information that enables this jondo to participate in the crowd. We defer further discussion of the blender and crowd membership maintenance to Section 8.

The user selects this jondo as her web proxy by specifying its host name and port number in her web browser as the proxy for all services. Thus, any request coming from the browser is sent directly to the jondo.[1] Upon receiving the first user request from the browser, the jondo initiates the establishment of a random *path* of jondos that carries its users' transactions to and from their intended web servers. More precisely, the jondo picks a jondo from the crowd (possibly itself) at random, and forwards the request to it. When this jondo receives the request, it flips a biased coin to determine whether or not to forward the request to another jondo; the coin indicates to forward with probability p_f. If the result is to forward, then the jondo selects a random jondo and forwards the request to it, and otherwise the jondo submits the request to the end server for which the request was destined. So, each request travels from the user's browser, through some number of jondos, and finally to the end server. A possible set of such paths is shown in Figure 2. In this figure, the paths are $1 \rightarrow 5 \rightarrow$ server; $2 \rightarrow 6 \rightarrow 2 \rightarrow$ server; $3 \rightarrow 1 \rightarrow 6 \rightarrow$ server; $4 \rightarrow 4 \rightarrow$ server; $5 \rightarrow 4 \rightarrow 6 \rightarrow$ server; and $6 \rightarrow 3 \rightarrow$ server. Subsequent requests initiated at the same jondo follow the same path (except perhaps going to a different end server), and server replies traverse the same path as the requests, only in reverse.

A pseudocode description of a jondo is presented in Figure 3. This figure describes a thread of execution that is executed per received request. This description uses client-server terminology, where one jondo is a *client* of its successor on the path.

[1]The services that must be proxied include Gopher, FTP, HTTP and SSL. Otherwise, e.g., FTP requests triggered by downloading a web page would not go through the crowd, and would thus reveal the user's IP address to the end server. Java and ActiveX should be disabled in the browser as well, because a Java applet or ActiveX control embedded in a retrieved web page could connect back to its server directly and reveal the user's IP address to that server.

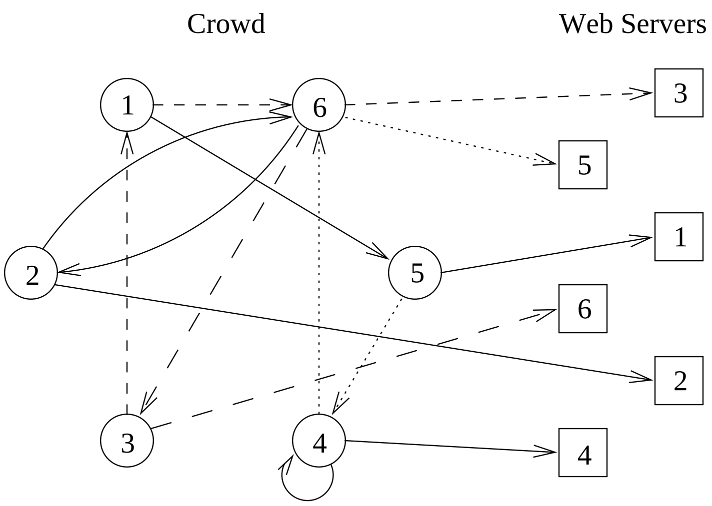


Fig. 2. Paths in a crowd (the initiator and web server of each path are labeled the same)

For each path, indicated by a *path id*, the value next[path_id] is the next jondo on the path. To assign next jondos for paths, each jondo maintains a set *Jondos* of jondos that it believes to be active (itself included). When it chooses to direct the path to another jondo, it selects the next jondo uniformly at random from this set (lines 6, 16, and 26); i.e., "$\leftarrow_R S$" denotes selection from the set S uniformly at random. Subsequent sections shed greater light on the operation of a jondo and the pseudocode description of Figure 3.

For technical reasons, it is convenient for the jondo at each position in a path to hold a different path identifier for the path. That is, if a jondo receives a request marked with path_id from its predecessor in a path, then it replaces path_id with a different path identifier stored in translate[path_id] before forwarding the request to its successor (if a jondo). This enables a jondo that occupies multiple positions on a path to act independently in each position: if the path_id remained the same along the path, then the jondo would behave identically each time it received a message on the path, resulting in an infinite loop. Path identifiers should be unique; in our present implementation, new_path_id() (lines 5 and 15) returns a random 128-bit value.

Omitted from the description in Figure 3 is that fact that all communication between any two jondos is encrypted using a key known only to the two of them. Encryption keys are established as jondos join the crowd, as is discussed in Section 8.

5. SECURITY ANALYSIS

In this section we consider the question of what information an attacker can learn about the senders and receivers of web transactions, given the mechanisms we described in Section 4. The types of attackers we consider were described in Section 2. Our analysis begins with the two attackers for which analysis is more straightfor-

```
client,request ← receive_request()
if (client = browser)
      sanitize(request)                           /* strip cookies and identifying headers */
      if (my_path_id = ⊥)                         /* if my_path_id is not initialized ... */
            my_path_id ← new_path_id()
            next[my_path_id] ←_R Jondos
      forward_request(my_path_id)
else                                              /* client is a jondo */
      path_id ← remove_path_id(request)           /* remove "incoming" path id */
      if (translate[path_id] = ⊥)                 /* "incoming" path id is new */
            coin ← coin_flip(p_f)                 /* tails with probability p_f */
            if (coin = heads)
                  translate[path_id] ← 'submit'
            else
                  translate[path_id] ← new_path_id()     /* set "outgoing" path id */
                  next[translate[path_id]] ←_R Jondos    /* select next jondo at random */
      if (translate[path_id] = 'submit')
            submit_request()
      else
            forward_request(translate[path_id])

subroutine forward_request(out_path_id)
      send out_path_id||request to next[out_path_id]
      reply ← await_reply(∞)                      /* wait for reply or recognizable jondo failure */
      if (reply = 'jondo failed')                 /* jondo failed */
            Jondos ← Jondos \ {next[out_path_id]}  /* remove the jondo */
            next[out_path_id] ←_R Jondos           /* assign a new random jondo for this path */
            forward_request(out_path_id)          /* try again */
      else                                        /* received reply from jondo */
            send reply to client

subroutine submit_request ()
      send request to destination(request)        /* send to destination web server */
      reply ← await_reply(timeout)                /* wait for reply, timeout, or server failure */
      send reply to client                        /* send reply or error message to client */
```

Fig. 3. Pseudocode description of a jondo

ward, namely a local eavesdropper and the end server. This is followed by an analysis of crowd security versus collaborating jondos.

5.1 Local eavesdropper

Recall that a local eavesdropper is an attacker that can observe all (and only) communication emanating from an individual user's computer. When this user initiates a request, the fact that she did so is exposed to the local eavesdropper, since we make no effort to hide correlations between inputs to and outputs from the initiating computer. That is, the local eavesdropper observes that a request output by the user's computer did not result from a corresponding input. Thus, we offer no sender anonymity against a local eavesdropper.

The mechanisms we described do, however, typically prevent a local eavesdropper from learning the intended receiver of a request, because every message forwarded on a path, except for the final request to the end server, is encrypted. Thus, while the eavesdropper is able to view any message emanating from the user's computer, it only views a message submitted to the end server (or equivalently a plaintext message containing the end server's address) if the user's jondo ultimately submits the user's request itself. Since the probability that the user's jondo ultimately submits the request is $1/n$ where n is the size of the crowd when the path was created, the probability that the eavesdropper learns the identity of the receiver decreases as a function of crowd size. Moreover, when the user's jondo does not ultimately submit the request, the local eavesdropper sees only the encrypted address of the end server, which we suggest yields receiver anonymity that is (informally) beyond suspicion. Thus, $P(\text{beyond suspicion}) \xrightarrow[n\to\infty]{} 1$ for receiver anonymity.

5.2 End servers

We now consider the security of our system against an attack by the end server only. Because the web server is the receiver, obviously receiver anonymity is not possible against this attacker. However, the anonymity for the path initiator is quite strong. In particular, since the path initiator first forwards to another jondo when creating its path (see Section 4), the end server is equally likely to receive the initiator's requests from any crowd member. That is, from the end server's perspective, all crowd members are equally likely to have initiated the request, and so the actual initiator's sender anonymity is beyond suspicion. It is interesting to note that this result, as opposed to that for collaborating jondos below, does not depend on p_f (the probability of forwarding; see Section 4). Indeed, increasing expected path length offers no additional assurance of anonymity against an end server.

5.3 Collaborating jondos

Consider a set of collaborating (corrupted) jondos in the crowd. A single malicious jondo is simply a special case of this attacker, and our analysis applies to this case as well. Because each jondo can observe plaintext traffic on a path routed through it, any such traffic, including the address of the end server, is exposed to this attacker. The question we consider here is if the attacker can determine who initiated the path.

To be precise, consider any path that is initiated by a non-collaborating member and on which a collaborator occupies a position. The goal of the collaborators is to determine the member that initiated the path. Assuming that the contents of the communication do not suggest an initiator, the collaborators have no reason to suspect any member other than the one from which they immediately received it, i.e., the member immediately preceding the first collaborator on the path. All other noncollaborating members are each equally likely to be the initiator, but are also obviously less likely to be the initiator than the collaborators' immediate predecessor. We now analyze how confident the collaborators can be that their immediate predecessor is in fact the path initiator.

Let H_k, $k \geq 1$, denote the event that the first collaborator on the path occupies the kth position on the path, where the initiator itself occupies the 0th position (and possibly others), and define $H_{k+} = H_k \vee H_{k+1} \vee H_{k+2} \vee \ldots$ Let I denote the event that the first collaborator on the path is immediately preceded on the path by the path initiator. Note that $H_1 \Rightarrow I$, but the converse is not true, because the initiating jondo might appear on the path multiple times. Given this notation, the collaborators now hope to determine $P(I|H_{1+})$, i.e., given that a collaborator is on the path, what is the probability that the path initiator is the first collaborator's immediate predecessor? Refining our intuition from Section 2, we say that the path initiator has probable innocence if this probability is at most 1/2.

Definition 5.1. *The path initiator has* probable innocence *(with respect to sender* *anonymity) if* $P(I|H_{1+}) \le 1/2$.

In order to yield probable innocence for the path initiator, certain conditions must be met in our system. In particular, let $p_f > 1/2$ be the probability of forwarding in the system (see Section 4), let c denote the number of collaborators in the crowd, and let n denote the total number of crowd members when the path is formed. The theorem below gives a sufficient condition on p_f, c, and n to ensure probable innocence for the path initiator.

THEOREM 5.2. *If* $n \ge \frac{p_f}{p_f - 1/2}(c+1)$, *then the path initiator has probable innocence against* c *collaborators.*

PROOF. We want to show that $P(I|H_{1+}) \le 1/2$ if $n \ge \frac{p_f}{p_f-1/2}(c+1)$. First note that

$$P(H_i) \;=\; \left(\frac{p_f(n-c)}{n}\right)^{i-1}\left(\frac{c}{n}\right)$$

This is due to the fact that in order for the first collaborator to occupy the ith position on the path, the path must first wander to $i-1$ noncollaborators (each time with probability $\frac{n-c}{n}$), each of which chooses to forward the path with probability p_f, and then to a collaborator (with probability $\frac{c}{n}$). The next two facts follow immediately from this.

$$P(H_{2+}) \;=\; \frac{c}{n}\sum_{k=1}^{\infty}\left(\frac{p_f(n-c)}{n}\right)^k \;=\; \left(\frac{c}{n}\right)\left(\frac{\frac{p_f(n-c)}{n}}{1-\frac{p_f(n-c)}{n}}\right) \;=\; \frac{p_f c(n-c)}{n^2 - p_f n(n-c)}$$

$$P(H_{1+}) \;=\; \frac{c}{n}\sum_{k=0}^{\infty}\left(\frac{p_f(n-c)}{n}\right)^k \;=\; \left(\frac{c}{n}\right)\left(\frac{1}{1-\frac{p_f(n-c)}{n}}\right) \;=\; \frac{c}{n - p_f(n-c)}$$

Other probabilities we need are $P(H_1) = \frac{c}{n}$, $P(I|H_1) = 1$, and $P(I|H_{2+}) = \frac{1}{n-c}$. The last of these follows from the observation that if the first collaborator on the path occupies only the second or higher position, then it is immediately preceded on the path by any noncollaborating member with equal likelihood. Now, $P(I)$ can be captured as

$$P(I) \;=\; P(H_1)P(I|H_1) + P(H_{2+})P(I|H_{2+}) \;=\; \frac{c(n - np_f + cp_f + p_f)}{n^2 - p_f n(n-c)}.$$

Then, since $I \Rightarrow H_{1+}$ we get

$$P(I|H_{1+}) \;=\; \frac{P(I \wedge H_{1+})}{P(H_{1+})} \;=\; \frac{P(I)}{P(H_{1+})} \;=\; \frac{n - p_f(n-c-1)}{n}$$

So, if $n \ge \frac{p_f}{p_f-1/2}(c+1)$, then $P(I|H_{1+}) \le \frac{1}{2}$. □

As a result of Theorem 5.2, if $p_f = \frac{3}{4}$, then probable innocence is guaranteed as long as $n \ge 3(c+1)$. More generally, Theorem 5.2 implies a tradeoff between the length of paths (i.e., performance) and ability to tolerate collaborators. That is, by making the probability of forwarding high, the fraction of collaborators that can be tolerated approaches half of the crowd. On the other hand, making the probability of forwarding close to one-half decreases the fraction of collaborators that can be tolerated.

The value of $P(H_{1+})$ derived in the proof of Theorem 5.2 shows that $P(H_{1+}) \to 0$ as $n \to \infty$ if c, p_f are held constant. Assuming that collaborators cannot observe a path on which they occupy no positions, it follows that $P(\text{absolute privacy}) \underset{n\to\infty}{\longrightarrow} 1$ for sender anonymity and receiver anonymity. The rate of this growth, however, can be slow if p_f is large.

5.3.1 *Timing attacks.* So far the analysis of security against collaborating jondos has not taken timing attacks into account. The possibility of timing attacks in our system results from the structure of HTML, the language in which web pages are written. An HTML page can include a URL (e.g., the address of an image) that, when the page is retrieved, causes the user's browser to automatically issue another request.[2] It is the immediate nature of these requests that poses the greatest opportunity for timing attacks by collaborating jondos. Specifically, the first collaborating jondo on a path, upon returning a web page on that path containing a URL that will be automatically retrieved, can time the duration until it receives the request for that URL. If the duration is sufficiently short, then this could reveal that the collaborator's immediate predecessor is the initiator of the request.

In our present implementation, we eliminate such timing attacks as follows. When a jondo receives an HTML reply to a request that it either received directly from a user's browser or submitted directly to an end server—i.e., the jondo is either the user's (i.e., the path initiator) or the last jondo on the path—it parses the HTML page to identify all URLs that the user's browser will automatically request as a result of receiving this reply. The last jondo on the path requests these URLs and sends them back along the same path on which the original request was received. The user's jondo, upon receiving requests for these URLs from the user's browser, does *not* forward these requests on the path, but rather simply waits for the URLs' contents to arrive on the path and then feeds them to the browser. In this way, other jondos on the path never see the requests that are generated by the browser, and thus cannot glean timing information from them. Note that misbehavior by the last jondo on the path (or any intermediate jondo) can result only in a denial of service, and not in a successful timing attack. In particular, if an attacking jondo inserts an embedded URL into the returning page, the user's jondo will identify it and expect the URL contents to arrive, but will not forward the request for the URL that the user's browser initiates.

This mechanism prevents jondos other than the user's from observing requests automatically generated due to the retrieval of a page. Therefore, all requests observable by attacking jondos are generated by explicit user action. It is conceivable that a user's response to a page (e.g., clicking on a contained URL), if sufficiently rapid, could reveal to the jondo in the first position on the path that its predecessor is the initiator of the path, in a way similar to how an automatic request might. However, the user's response would need to be extremely fast—typically within a

[2] These URLs are contained in, for example, the `src` attributes of `<embed>`, `<frame>`, `<iframe>`, `<img>`, `<input type=image>`, and `<script>` tags, the `background` attributes of `<body>`, `<table>`, `<tr>` and `<td>` tags, the `content` attributes of `<meta>` tags, and others.

fraction of a second of viewing the page—to risk revealing this information. We expect that such response times are uncharacteristic of human browsing, and can be made even less so by educating users of this risk. If, however, this presumption turns out to be incorrect, the user's jondo could insert a random delay per user-generated request, thereby decreasing the chances of revealing this information to virtually zero.

The primary drawback of our present approach to defending against timing attacks is that it is not easily compatible with some web technologies. For example, web pages that contain executable scripts, e.g., written in JavaScript, can make it difficult for a jondo to identify in advance the URLs that a browser will automatically request as a result of interpreting those pages. One way to address this is for the user's jondo to delay requests received from the browser immediately after feeding the browser a page containing JavaScript. A more foolproof defense, which we recommend, is for the user to disable JavaScript in the browser when browsing via Crowds; this can be done easily via a preference menu in most browsers. Another technology that presents some difficulties is SSL, a protocol by which web pages can be encrypted during transport. To enable both the user's jondo and the last jondo on the path to parse SSL-retrieved pages, the SSL connection to the web server must be made by the last jondo on the path. In this case, HTTP communication is not protected from jondos on the path, but is protected from other eavesdroppers because all communication between jondos is encrypted. At the time of this writing, however, SSL is not supported by Crowds.

5.3.2 *Static paths.* Early in the design of Crowds, we were tempted to make paths much more dynamic than they are in the present system, e.g., by having a jondo use a different path for each of its users, per time period, or even per user request. The advantages of more dynamic paths include the potential for better performance via load balancing among the crowd. In this section, however, we caution that dynamic paths tends to decrease the anonymity properties provided by the system against collaborating jondos. The reason is that the probable innocence offered by Theorem 5.2 vanishes if the collaborators are able to link many distinct paths as being initiated by the same jondo. Collaborating jondos might be able to link paths initiated by the same unknown jondo based on related path content or timing of communication on paths. To prevent this, we made paths static, so the attacker simply does not have multiple paths to link to the same jondo.

To see why multiple linked paths initiated by the same jondo could compromise its user's anonymity, note that collaborating jondos have a higher probability of receiving each path initiation message (i.e., the first request on the path) from the initiator of the path than from any other individual member (see the proof of Theorem 5.2). Multiple paths initiated by the same user's jondo therefore pinpoint that jondo as the one from which the collaborators most often receive the initiating messages. Put another way, if the collaborators identify paths $P_1, \ldots, P_k$ from the same (unknown) initiator, then the expected number of paths on which the first collaborator is directly preceded by the path initiator is $\mu = k(\frac{n-p_f(n-c-1)}{n})$. By Chernoff bounds, the probability that the first collaborator is immediately preceded by the initiator on substantially fewer of these paths is small: the first collaborator is immediately preceded by the path initiator on fewer than $(1-\delta)\mu$ paths with probability only $e^{-\mu\delta^2/2}$ (see [Motwani and Raghavan 1995, Theorem 4.2]). Thus, the initiator would be identified with high probability.

Again, it is for this reason that a jondo sets up one path for all its users' communications, and this path is altered only under two circumstances. First, a path is altered when failures are detected in the path. More specifically, paths are only rerouted when the failure of a jondo is unmistakenly detected, i.e., when the jondo executes a *fail-stop* failure [Schlichting and Schneider 1983]. In our present implementation, such failures are detected by the TCP/IP connection to the jondo breaking or being refused; a jondo does not reroute a path based on simply timing out on the subsequent jondo in the path (see line 23 of Figure 3). While this increases our sensitivity to denial-of-service attacks (see Section 2.3), it strengthens our promise of anonymity to the user.

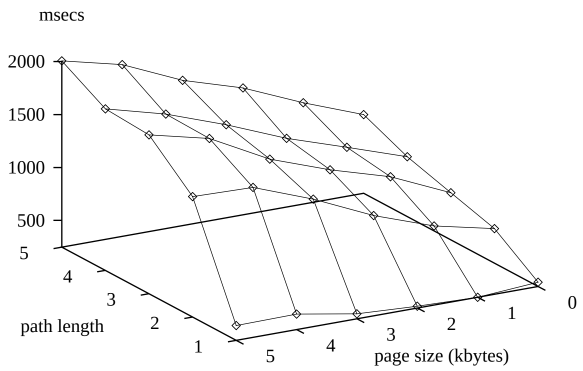
A reasonable question, however, is whether a malicious jondo on a path can feign its own failure in hopes that the path will be rerouted through a collaborator, yielding information that incriminates the path initiator. Fortunately, the answer is "no." If a jondo in a path fails (or appears to fail), the path remains the same up until the predecessor of that faulty jondo, who reroutes the remainder of the path randomly (line 26 of Figure 3). Since the collaborating jondos cannot distinguish whether that predecessor is the originator or not, the random choices made by that predecessor yield no additional information to the collaborators.

The second circumstance in which paths are altered is when new jondos join the crowd. The motivation for rerouting paths is to protect the anonymity of a joining jondo: if existing paths remained static, then the joiner's new path can be easily attributed to the new jondo when it is formed. Thus, to protect joiners, all jondos "forget" all paths after new jondos join, and re-establish paths from scratch. To avoid exposing path initiators to the attack described previously in this section, joins are grouped into infrequent scheduled events called *join commits* (see Section 8). Once a join commit occurs, existing paths are forgotten, and the newly joined jondos are enabled to participate in the crowd. Batching many joins into a single join commit limits the number of times that paths are rerouted and thus the number of paths vulnerable to linkage by collaborators. Moreover, each user is alerted when a join commit occurs and is cautioned from continuing to browse content related to what she was browsing prior to the commit, lest collaborators are attempting to link paths based on that content.

6. PERFORMANCE

In this section we describe the performance of Crowds 1.0. As discussed in Section 3, performance is one of the motivating factors behind the design of Crowds and, we believe, a strength of our approach relative to mixes [Chaum 1981] (though there are few published performance results for mix implementations to which to compare our results). And, while Crowds performance is already encouraging, it could be improved further by re-implementing it in a compiled language such as C. Crowds 1.0 is implemented in Perl 5 (a partially interpreted language), which we chose for its rapid prototyping capabilities and its portability across Unix and Microsoft platforms.

Results of performance tests on our implementation are shown in Figures 4–5. In these tests, the source of requests was a Netscape 3.01 browser configured to



Path length	Page size (kbytes) 0	1	2	3	4	5
1	288	247	264	294	393	386
2	573	700	900	1157	1369	1384
3	692	945	1113	1316	1612	1748
4	814	1004	1191	1421	1623	1774
5	992	1205	1446	1620	1870	2007

Fig. 4. Response latency (msecs) as a function of path length and page size

allow a maximum of 4 simultaneous network connections. The crowd consisted of four jondos, each executing on a separate, moderately loaded 150 MHz Sparc 20 running SunOS 4.1.4. The web server was a fairly busy 133 MHz SGI workstation running Irix 5.3 and an Apache web server. All of these computers are located in AT&T Labs, and thus are in close network proximity to one another.

Figure 4 shows the mean latency in milliseconds of retrieving web pages of various sizes (containing no embedded URLs) for various path lengths. Each number indicates the average duration beginning when the user's jondo receives the request from the browser and ending when the page has been written back to the browser. In this figure, the path length is the number of *appearances* of jondos on the path. That is, if a jondo appears k times on a path, then this jondo contributes k to the total path length. So, for example, in Figure 2, the paths initiated by jondos 1, 4, and 6 are each of length two, and the paths initiated by 2, 3, and 5 are each of length three.

One observation we can make from Figure 4 is that the latency sharply increases when the path length increases from one to two. The primary reason for the sharp increase is that a path length of two is the first length at which encryption of page contents takes place. In a path of length one (which would be employed only if there were one crowd member), the user's jondo acts as a simple proxy between the browser and end server, to strip away identifying information from HTTP headers. In a path of length two, however, both the request and reply are passed, and encrypted, between the jondos on the path. To slow the growth of this latency as the path gets longer, this encryption is performed using a *path key*, which is a key shared among all jondos on a path. A path key is created by the jondo initiating the path, and each jondo on a path forwards it to the next jondo by encrypting the path key with a key it shares with the next jondo (see Section 8). The existence of a path key enables requests to be encrypted at the jondo initiating the path, decrypted by the last jondo in the path, and passed by intermediate jondos without encrypting or decrypting the requests. Similarly, replies are encrypted at the last jondo in the path, and decrypted only at the jondo where the path was initiated. The cryptographic operations are performed using an efficient stream cipher, allowing some of the encrypting and decrypting streams for the reply to be generated while the jondos are waiting for the reply from the web server. However, since even this cipher is implemented in Perl for portability, it remains a bottleneck in our implementation.

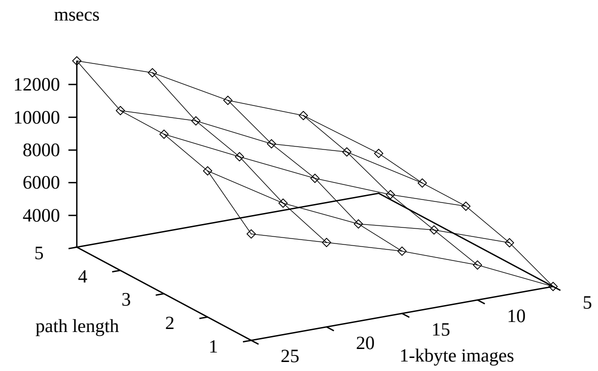
Figure 5 shows the mean latency in milliseconds of retrieving, via paths of various different lengths, pages containing URLs that are automatically retrieved by the browser (see Section 5.3.1). In these tests, each embedded URL is the address of a 1-kilobyte image resident on the same server as the page that referenced it. Each number indicates the average duration beginning when the user's jondo receives the initial request from the browser and ending when the jondo finishes writing the page and all of the images on the page to the browser. It is clear from Figure 5 that the number of images considerably impacts the latency of responses. Though this is to be expected in general, this effect is particularly pronounced in our implementation, and is due primarily to encryption costs. Moreover, returning images on the path has the effect of serializing their retrieval, which further increases the latency over that achieved by modern browsers alone (which use several network connections to retrieve multiple images concurrently).

Because paths (and thus path lengths) are established randomly at run time, the user cannot choose her path length to predict the request latency she experiences. However, the expected path length can be influenced by modifying the value p_f—i.e., the probability that a jondo forwards to another jondo versus submitting to the end server—at all jondos. Specifically, if $n > 1$, the expected length of a path is

$$\begin{aligned}(1-p_f)\sum_{k=0}^{\infty}(k+2)(p_f)^k &= (1-p_f)\left[\sum_{k=0}^{\infty}k(p_f)^k + 2\sum_{k=0}^{\infty}(p_f)^k\right]\\ &= (1-p_f)\left[\frac{p_f}{(1-p_f)^2}+\frac{2}{1-p_f}\right]\\ &= \frac{p_f}{1-p_f}+2\end{aligned}$$

This suggests that multiple types of crowds should exist: those employing a small p_f for better performance but less resilience to collaborating jondos (see Theorem 5.2), and those using a large p_f to increase security with a cost to performance.

Performance seen in practice may differ from Figures 4 and 5, depending on the platforms running jondos and the speed of network connectivity between jondos. In particular, a jondo connected to the Internet via a slow modem link considerably



Path length	Number of 1-kbyte images 5	10	15	20	25
1	2069	4200	5866	7219	8557
2	3313	4915	6101	8195	10994
3	4127	5654	7464	9611	11809
4	4122	6840	8156	10380	11823
5	4508	7644	9388	11889	13438

Fig. 5. Response latency (msecs) as a function of path length and number of embedded images

impacts latencies on paths that use it. Again, this suggests multiple types of crowds, namely ones containing only jondos connected via fast links, and ones allowing jondos connected via slower links.

7. SCALE

The numbers in Section 6 give little insight into how performance is affected as crowd size grows. We do not have sufficient resources to measure the performance of a crowd involving hundreds of computers, each simultaneously issuing requests. However, in this section we make some simple analytic arguments to show that the performance should scale well.

The measure of scale that we evaluate is the expected total number of appearances that each jondo makes on all paths at any point in time. For example, if a jondo occupies two positions on one path and one position on another, then it makes a total of three appearances on these paths. Theorem 7.1 says that the each jondo's expected number of appearances on paths is virtually constant as a function of the size of the crowd. This suggests that crowds should be able to grow quite large.

THEOREM 7.1. *In a crowd of size n, the expected total number of appearances that any jondo makes on all paths is* $O\left(\frac{1}{(1-p_f)^2}(1+\frac{1}{n})\right)$.

PROOF. Let n be the size of the crowd. To compute the load on a jondo, say J, we begin by computing the distribution of the number of appearances made by J

on each path. Let R_i, $i > 0$, denote the event that this path reaches J exactly i times (not counting the first if J initiated the path). Also, define R_0 as follows:

$$P(R_0) = (1-p_f)\sum_{k=0}^{\infty}(p_f)^k\left(\frac{n-1}{n}\right)^k = (1-p_f)\left(\frac{1}{1-p_f\frac{n-1}{n}}\right)$$

Intuitively, $P(R_0)$ is the probability that the path, once it has reached J, will never reach J again. Then, we have

$$P(R_1) = \frac{1}{n}P(R_0)\sum_{k=0}^{\infty}(p_f)^k\left(\frac{n-1}{n}\right)^k = (1-p_f)\left(\frac{1}{n}\right)\left(\frac{1}{1-p_f\frac{n-1}{n}}\right)^2$$

$$P(R_2) = \frac{1}{n}p_f P(R_1)\sum_{k=0}^{\infty}(p_f)^k\left(\frac{n-1}{n}\right)^k < (1-p_f)\left(\frac{1}{n}\right)^2\left(\frac{1}{1-p_f\frac{n-1}{n}}\right)^3$$

$$\vdots$$

$$P(R_i) = \frac{1}{n}p_f P(R_{i-1})\sum_{k=0}^{\infty}(p_f)^k\left(\frac{n-1}{n}\right)^k < (1-p_f)\left(\frac{1}{n}\right)^i\left(\frac{1}{1-p_f\frac{n-1}{n}}\right)^{i+1}$$

From this, the expected number of appearances that J makes on a path formed by another jondo is bounded from above by:

$$\begin{aligned}\left(\frac{1-p_f}{1-p_f\frac{n-1}{n}}\right)\left[\sum_{k=0}^{\infty}k\left(\frac{1}{n-p_f(n-1)}\right)^k\right] &= \left(\frac{1-p_f}{1-p_f\frac{n-1}{n}}\right)\left(\frac{n-p_f(n-1)}{(1-n+p_f(n-1))^2}\right)\\ &< \frac{n-p_f(n-1)}{(1-n+p_f(n-1))^2}\\ &= \frac{1}{(1-p_f)(n-1)}+\frac{1}{((1-p_f)(n-1))^2}\\ &< \frac{2}{(1-p_f)^2(n-1)}\end{aligned}$$

Therefore, the expected number of appearances that J makes on all paths is bounded from above by:

$$\frac{2n}{(1-p_f)^2(n-1)} = \frac{2}{(1-p_f)^2}\left(1+\frac{1}{n-1}\right)$$

□

8. CROWD MEMBERSHIP

The membership maintenance procedures of a crowd are those procedures that determine who can join the crowd and when they can join, and that inform members of the crowd membership. We discuss mechanisms for maintaining crowd membership in Section 8.1, and policies regarding who can join a crowd in Section 8.2.

8.1 Mechanism

There are many schemes that could be adopted to manage membership of the crowd. Existing *group membership protocols*, tolerant either of benign (e.g., [Cristian 1991;

Ricciardi and Birman 1991; Moser et al. 1991]) or malicious [Reiter 1996b] faults, can be used for maintaining a consistent view of the membership among all jondos, and the members could use voting to determine whether an authenticated prospective member should be admitted to the crowd. Indeed, a similar approach has been adopted in prior work on secure process groups [Reiter et al. 1994]. While providing robust distributed solutions, these approaches have the disadvantages of incurring significant overhead and of providing semantics that are arguably too strong for the application at hand. In particular, a hallmark of these approaches is a guaranteed consistent view of the group membership among the group members, whereas it is unclear whether such a strong guarantee is required here.

In our present implementation we have therefore opted for a simpler, centralized solution. Membership in a crowd is controlled and reported to crowd members by a server called the *blender*. To make use of the blender (and thus the crowd), the user must establish an *account* with the blender, i.e., an account name and password that the blender stores. When the user starts a jondo, the jondo and the blender use this shared password to authenticate each other's communication. As a result of that communication (and if the blender accepts the jondo into the crowd; see Section 8.2), the blender adds the new jondo (i.e., its IP address, port number, and account name) to its list of members, and reports this list back to the jondo. In addition, the blender generates and reports back a list of shared keys, each of which can be used to authenticate another member of the crowd. The blender then sends each key to the other jondo that is intended to share it (encrypted under the account password for that jondo) and informs the other jondo of the new member. At this point all members are equipped with the data they need for the new member to participate in the crowd. However, to protect itself from attacks described in Section 5.3.2, the new member refrains from doing so until it receives a join "commit" message from the blender. This is discussed further in Section 8.2.

Each member maintains its own list of the crowd membership. This list is initialized to that received from the blender when the jondo joins the crowd, and is updated when the jondo receives notices of new or deleted members from the blender. The jondo can also remove jondos from its list of crowd members, if it detects that those jondos have failed (see line 25 of Figure 3). This allows for each jondo's list to diverge from others' if different jondos have detected different failures in the crowd. This appears to have little qualitative effect on our security analysis of Section 5, unless attackers are able to prevent communications between correct jondos to the extent that each removes the correct jondos from its list of members.

A disadvantage of this approach to membership maintenance is that the blender is a trusted third party for the purposes of key distribution and membership reporting. Techniques exist for distributing trust in such a third party among many "third party replicas", in a way that the corruption of some fraction of the replicas can be tolerated (e.g., [Deswarte et al. 1991; Gong 1993; Reiter 1996a]). In its present, non-replicated form, however, the blender is best executed on a secure computer, e.g., with login access available only at the console. Even though it is a trusted third party for some functions, note that users' HTTP communication is *not* routed through the blender, and thus a passive attack on the blender does not immediately reveal users' web transactions (unlike the Anonymizer; see Section 3). Moreover, the failure of the blender does not interfere with ongoing web transactions (again unlike the Anonymizer). We anticipate that in future versions of Crowds, jondos will establish shared keys using Diffie-Hellman key exchange [Diffie and Hellman 1976], where the blender serves only to distribute the Diffie-Hellman public keys of crowd members. This will eliminate the present reliance on the blender for key generation.

8.2 Policy

It is important in light of Section 5 that some degree of control over crowd membership be maintained. First, if anyone can add arbitrarily many jondos to a crowd, then a single attacker could launch enough collaborating jondos so that $n < \frac{p_f}{p_f - 1/2}(c+1)$, at which point Theorem 5.2 no longer offers protection. Second, since joins cause paths to be re-routed (see Section 5.3.2), if joins are allowed to occur frequently and without controls, then paths may be re-routed sufficiently frequently to allow collaborating jondos to mount the correlation attack described in Section 5.3.2. In our present implementation, the blender serves as the point at which joins to the crowd are controlled.

To address the latter concern, the blender batches joins together so they occur in one scheduled, discrete event called a *join commit*. The schedule of join commits is a configurable parameter of the blender, but we envision that one commit per day should typically suffice. The blender informs all crowd members of the join commit, at which point all newly joined members are enabled to participate in the crowd and all old members reset their paths, as described in Section 5.3.2.

The need to limit the number of collaborators that join the crowd suggests that two different types of crowds will exist. The first type would consist of a relatively small (e.g., 10–30) collection of individuals who, based on personal knowledge of each other, agree to form a crowd together. Each member would be allowed to include at most one jondo in the crowd. More precisely, each person would be given one account, and only one jondo per account would be allowed. Each member's personal knowledge of the other members enables her to trust that sufficiently few members collaborate to ensure that $n \geq \frac{p_f}{p_f - 1/2}(c+1)$.

The second type of crowd would be a much larger "public" crowd, admitting members that might not be known to a substantial fraction of the present membership. The privacy offered by the crowd against collaborating members would rely on the size of the crowd being so large that an attack aimed at making $n < \frac{p_f}{p_f - 1/2}(c+1)$ would require considerable effort to go undetected. That is, by limiting each user to one account (e.g., the blender administrator sets up an account for a user only after receiving a written, notarized request from that user) and each account to one jondo, and by monitoring and limiting the number of jondos on any one network (using IP address), the attacker would be forced to launch jondos using many different identities and on many different networks to succeed.

9. USER INTERFACE

In our present implementation, there are several ways in which a user interacts with her jondo, i.e., the jondo that serves as the HTTP proxy of her browser.

(1) The user can issue a *crowd query* by appending `?crowd?` to the end of any URL that she requests. This returns a list of all of the active jondos in the crowd,

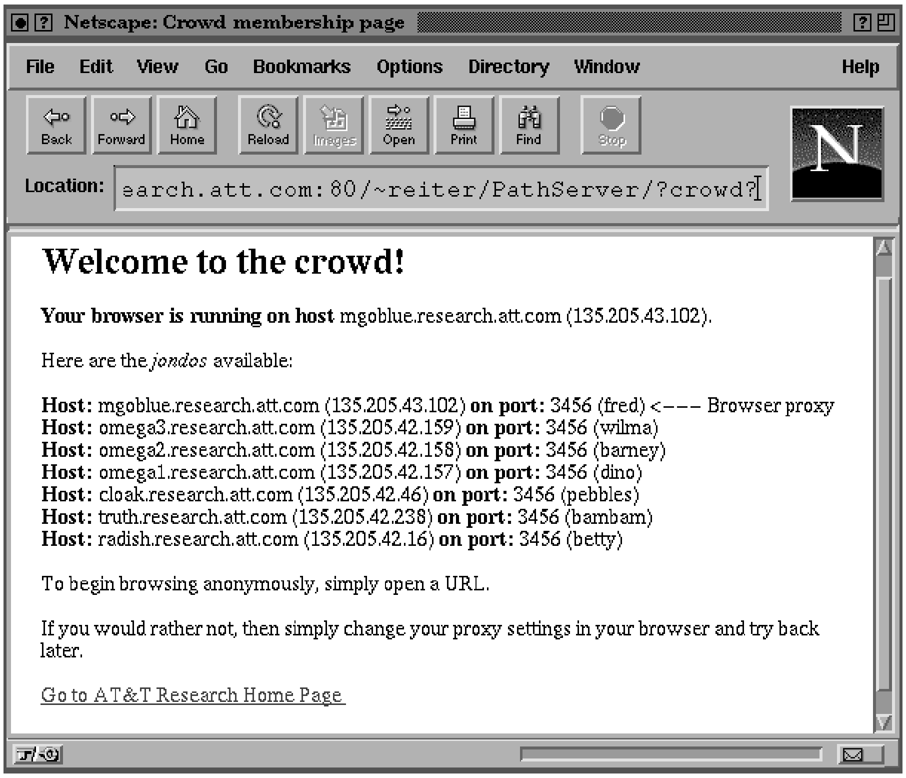


Fig. 6. **Crowd query:** A crowd query shows the jondos that are available, and indicates which one is acting as the user's HTTP proxy for the browser.

according to her jondo. The information includes each jondo's account name, its IP address and its port number. An example is shown in Figure 6.

(2) When a user is browsing via the crowd, the word *Crowd:* is prepended to the title of each page. Thus, a user can check whether or not she is using the crowd by looking at the title of the documents in the browser. Of course, a web server could add this word to the title of any document to fool the user, and so this alone should not be relied upon.

(3) Crowds offers other informational pages to the user via the browser, similar to Figure 6. For example, Crowds alerts the user when a join commit occurs.

The savvy Crowds user can also fine-tune her jondo's behavior by way of a configuration file that defines the behavior of her jondo. This configuration file includes, for example, parameter settings to allow or disallow the passage of *cookies*, i.e., data that a web server can download to the user's browser and that the user's browser will include in subsequent requests to that server. By default, a jondo strips all cookies out of requests it receives from browsers in order to better protect its users' privacy, but the jondo can be configured to let cookies pass. Other configurable parameters of a jondo include, for example, the host and port of the crowd blender, and the account name and password under which the jondo requests admission to the crowd. In the future, other configuration options may be added to give the user further control over her jondo. For example, a parameter could be included to define a threshold so that if the number of crowd members drops below this value, then the user is alerted to this fact. Other parameters could be included that specify which HTTP headers are allowed to pass in requests (presently a jondo strips away any that contain information characterizing the user or her platform) or what types of content (e.g., Java, JavaScript) are allowed to pass into the browser.

10. FIREWALLS

Firewalls present a problem for Crowds. Like all network servers, jondos are identified by their IP address and port number. Most corporate firewalls do not allow incoming connections on ports other than a few well-known ones. Thus, a firewall will generally prevent a jondo outside the firewall from connecting to another behind the firewall. For this reason, firewalls represent a barrier to wide-scale inter-corporation adoption of Crowds.

Since most firewalls are configured to allow outgoing connections on any port, it is still possible for a jondo to initiate a path that goes outside the firewall and eventually to web servers. However, the firewall gives the first jondo on the path outside that domain a way to verify that the initiating computer resides within the domain: it simply tries to open a connection back to its predecessor on the path, and if that fails, then the path must have originated in the predecessor's domain. Thus, a crowd member behind a firewall is not offered the same anonymity as those that are not.

It is conceivable that if Crowds becomes widespread, and there is demand for a special reserved port, that firewalls can open this port and allow jondos to communicate. Until then, Crowds will be most useful across academic institutions, as a service provided by Internet service providers, and within large corporations.

11. CONCLUSION

In this paper we have presented a novel approach to protecting users' privacy while retrieving information on the world-wide-web, and a system that implements it. Our approach works by grouping web users into a geographically diverse collection, called a *crowd*, which retrieves information on its users' behalf by way of a simple randomized routing protocol. Using *degrees of anonymity*, we have characterized the anonymity properties provided by this protocol against several classes of attackers. We have also described the Crowds system that we have implemented, the measures it takes to defend against various attacks resulting from the way the web works today, and the performance, scalability, and limitations of our system. The principles behind our system can be more broadly applied for anonymizing other forms of communication.

At the time of this writing, we have distributed over 450 copies of the Crowds code free-of-charge in response to user requests, and we are maintaining the blender for an active crowd on the Internet. Information about obtaining the Crowds code can be found at `http://www.research.att.com/projects/crowds`.

Acknowledgements

We thank Gerrit Bleumer, Marc Briceno, Hal Finney, Ian Goldberg, David Goldschlag, Raph Levien, Jim McCoy, Fabian Monrose, Michael Reed, Paul Syverson, and David Wagner for many valuable suggestions regarding Crowds and this paper.

REFERENCES

BRIER, S. 1997. How to keep your privacy: Battle lines get clearer. *The New York Times*, January 13, 1997.

CHAUM, D. 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (February), 84–88.

CRISTIAN, F. 1991. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing 4*, 175–187.

DESWARTE, Y., BLAIN, L., AND FABRE, J. 1991. Intrusion tolerance in distributed computing systems. *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, 110–121.

DIFFIE, W. AND HELLMAN, M. E. 1976. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6.

GABBER, E., GIBBONS, P., MATIAS, Y., AND MAYER, A. 1997. How to make personalized web browsing simple, secure, and anonymous. *Proceedings of Financial Cryptography '97*.

GONG, L. 1993. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications 5*, 11 (June), 657–662.

GULCU, C. AND TSUDIK, G. 1996. Mixing e-mail with BABEL. *Symposium on Network and Distributed System Security*, 2–16.

HICKMAN, K. E. B. AND ELGAMAL, T. 1995. The SSL protocol. *Internet draft draft-hickman-netscape-ssl-01.txt*.

MILLER, L. 1997. No solitude in cyberspace. *USA Today*, June 9, 1997.

MOSER, L. E., MELLIAR-SMITH, P. M., AND AGRAWALA, V. 1991. Membership algorithms for asynchronous distributed systems. *Proceedings of the 11th International Conference on Distributed Computing Systems*, 480–488.

MOTWANI, R. AND RAGHAVAN, P. 1995. *Randomized Algorithms*. Cambridge University Press.

PFITZMANN, A. AND PFITZMANN, B. 1990. How to break the direct RSA-implementation of mixes. *Advances in Cryptology—EUROCRYPT '89*, 373–381.

PFITZMANN, A., PFITZMANN, B., AND WAIDNER, M. 1991. ISDN-mixes: Untraceable communication with very small bandwidth overhead. *GI/ITG Conference: Communication in Distributed Systems*, 451–463.

PFITZMANN, A. AND WAIDNER, M. 1987. Networks without user observability. *Computers & Security 2*, 6, 158–166.

REITER, M. K. 1996a. Distributing trust with the Rampart toolkit. *Communications of the ACM 4*, 39 (April), 71–74.

REITER, M. K. 1996b. A secure group membership protocol. *IEEE Transactions on Software Engineering 1*, 22 (January), 31–42.

REITER, M. K., BIRMAN, K. P., AND VAN RENESSE, R. 1994. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems 4*, 12 (November), 340–371.

RICCIARDI, A. M. AND BIRMAN, K. P. 1991. Using process groups to implement failure detection in asynchronous environments. *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, 341–351.

SCHLICHTING, R. D. AND SCHNEIDER, F. B. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems 1*, 3 (August), 222–238.

SYVERSON, P. F., GOLDSCHLAG, D. M., AND REED, M. G. 1997. Anonymous connections and onion routing. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*.