Anonymizing Censorship Resistant Systems

Andrei Serjantov
Andrei.Serjantov@cl.cam.ac.uk

University of Cambridge Computer Laboratory,
William Gates Building,
JJ Thomson Avenue,
Cambridge CB3 0FD, UK

Abstract. In this paper we propose a new Peer-to-Peer architecture for a censorship resistant system with user, server and active-server document anonymity as well as efficient document retrieval. The retrieval service is layered on top of an existing Peer-to-Peer infrastructure, which should facilitate its implementation. The key idea is to separate the role of document storers from the machines visible to the users, which makes each individual part of the system less prone to attacks, and therefore to censorship.

1 Introduction

Many censorship resistant systems have been proposed recently, yet most still lack one crucial feature – protection of the servers hosting the content.

In the past this was not considered an issue. For instance, in Anderson's eternity service [And96], it was deemed sufficient to guarantee that a document was always available through the system. However, many examples indicate that servers hosting content are vulnerable to censorship, due to "Rubber Hose Cryptanalysis" – various kinds of pressure applied by attackers to shut down servers or remove files. Examples of documents subjected to censorship include DeCSS [DeC], the paper detailing the attack on SDMI [CMW$^+$01], and documents (or quotes from documents) which the Church of Scientology described as their secrets. In cases like this, the server administrators receive "cease and desist" letters when the censor finds the offending document on their server.

Most modern censorship resistant systems, for example Publius [WRC00] and Freenet [CSWH01], have not addressed this problem in a satisfactory way. It is possible for a malicious reader to find out which servers content is stored on, and subsequently try to pressure the server administrators to remove it. With Publius in particular, the situation is slightly more complicated as each document is encrypted and the key is split into n shares, any k of which are recombinable to form the document back. In this case, the attacker needs to remove content from $n - k + 1$ servers, all of which he can easily locate.[1] With the number of servers reasonably small and static, the job of censoring documents becomes easier than one might expect.

An alternative approach to dealing with censorship resistance is taken by systems like Dagster [SW01] and Tangler [WM01] which prevent removal of any single document from the system by entangling documents together. However, in our view, these are not effective enough at dealing with the problem either: if the offending document was entangled with the Declaration of Independence, Das Capital and the Little Red Book, censoring it (thereby removing all of the above from the system) would not be a major problem as all the other documents are readily available from other sources. Given the other documents are *not* available, the censor is unlikely to be discerning enough to want to keep some of them. Furthermore, if the system is run on a single server (eg. Dagster), then the censor may simply try to shut down the entire server.

In our system, we consider the storers of the files valuable entities, and protect them against "Rubber Hose Cryptanalysis". Furthermore, we protect the documents they are storing by providing *active-server document anonymity* (as first introduced in [DFM01]). This is a property which states that the storer should not be able to determine (parts of) which document it is storing, not even by retrieving the document from the system. We now proceed with a description of the system and then give an analysis and critique of it.

2 System Description

Our system consists of many identical peers, each of which can fulfil four different roles:

- Publisher P. The node which has a document and wishes to make it available and censorship resistant.
- Storer s. A node which stores part of a document.
- Forwarder a. A node which has an anonymous pointer to a node storing part of a document.
- Client c. A node which retrieves a document.
- Decrypter l. A node which decrypts part of a document and sends it off to the client.

The system is built on top of an existing Peer-to-Peer document storage service like PAST [DR01]. PAST can be viewed as a network of machines (peers), each with a unique identifier. Neighbouring machines (machines within a certain

[1] Indeed, if one server has been pressured into removal, the other server administrators may simply follow the precedent and remove the offending content themselves.

distance of each other within a logical name space) share state. The only thing required to send a message to a machine is its id, furthermore, PAST guarantees that the message takes no more than $\log N$ hops, where N is the number of nodes in the system. We also assume a public key infrastructure, so any peer is able to learn any other peer's public key. This is further discussed in the next section. Using an existing Peer-to-Peer architecture allows us to abstract from routing, clients leaving and joining the network, and other low level issues. An architecture like PAST also provides robustness, which is further discussed in the next section. We also make use of an anonymous connection system such as Onion Routing [GRS99] which is capable of handling reply blocks.

As usual in censorship resistant systems, the operations available to a node are publishing and retrieval. There is no search facility, therefore we rely on a broadcast mechanism like an anonymous newsgroup to transmit retrieval information to potential readers. We do not support content deletion or modification.

2.1 Publishing

The overall publishing process is illustrated in Figure 1. The main idea is to split the documents into many parts or shares h_i, and store them (encrypted) on machines s_i, while making them accessible through machines a_i which forward requests for the appropriate shares anonymously.

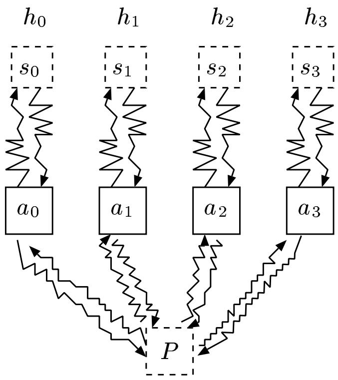

Fig. 1. Publishing

Publisher: To publish a document (see Figure 2), the publisher P splits it into $n+1$ shares h_i, any $k+1$ of which can be combined to form the whole document again. This can be done using one of the standard algorithms like Shamir's secret sharing [Sha79]. He then generates $n+1$ keys k_i and encrypts each share with the corresponding key. He now picks $n+1$ peers $a_0 \ldots a_n$ at random to act as forwarders and constructs onions[2] to send (via the anonymous connections layer) each of them the encrypted share $\{h\}_{k_i}$, the corresponding key k_i and a (large) random integer v_i together with a return address (reply onion)[3] r_P. The publisher can now wait for a confirmation to come back from each of the a_is (via the reply onion) saying whether the publishing has been successful or not. If the operation failed, the publisher should try different a_is.

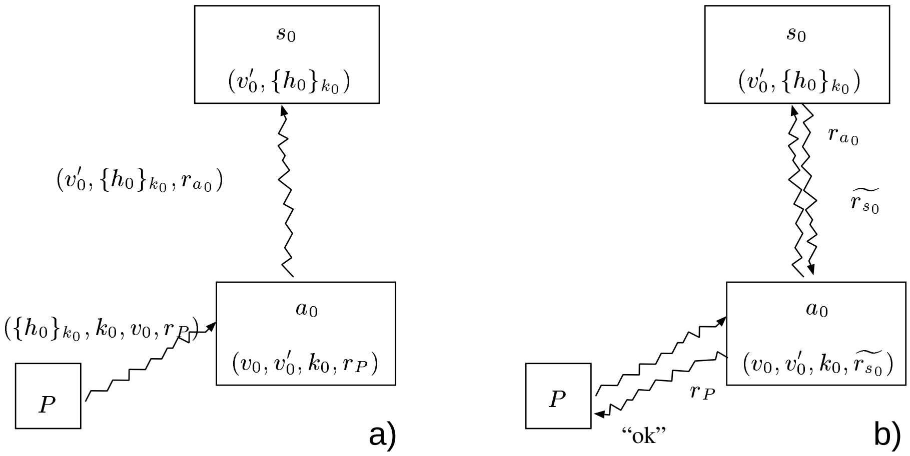

Fig. 2. Inserting share a_0. All communication is done via the anonymous connection system using randomly constructed onions. If the message is sent using a return address, it is displayed at the base of the arrow. Anonymous return addresses are denoted by r, eg. r_{a_0}

Share	h_0
Key	k_0
Storer	s_0
Random numbers	v_0, v_0'
Return addresses	r_P, $\widetilde{r_{s_0}}$

Forwarder: Each of the forwarders (take a_0 as an example) receives the message, finding an encrypted share $\{h_0\}_{k_0}$, a key k_0 and a random number v_0 and the publisher's return address. He then picks a storer s_0 to store the share and

[2] This is a technique first described in [Cha81]. A (standard) onion for destination d with message M and peer sequence $a_0 \ldots a_n$ is $\{a_1 \ldots \{d, \{M\}_{k_d}\}_{k_{a_n}} \ldots\}_{k_{a_0}}$ which is sent to a_0. a_i is the address of the server and k_{a_i} is its public key.

[3] A return address is a kind of onion which, if included in an anonymous message, can be used to reply to that message without revealing the original sender (see [Cha81] for details).

a number v'_0 which the storer would associate the share with. He constructs an onion for delivering these to the storer. Thus, he puts the encrypted share, v'_0, as well as its own anonymous return address r_{a_0} into the onion as the message and sends it off (see Figure 2a). He remembers v_0, v'_0, k_0 and r_P. If the onion is received by s_0, it stores the share and issues a number of different return addresses $\widetilde{r_{s_0}}$ (to be used for retrieval), sending them back to a_0 via the return address r_{a_0}. Now a_0 associates v_0, v'_0 and k_0 with the return addresses $\widetilde{r_{s_0}}$, forgets s_0, and replies "ok" to the publisher via r_P. Once all the shares have been stored, the publisher destroys them and announces the name of the file, together with the $n+1$ pairs (a_i, v_i) to potential users.

2.2 Retrieval

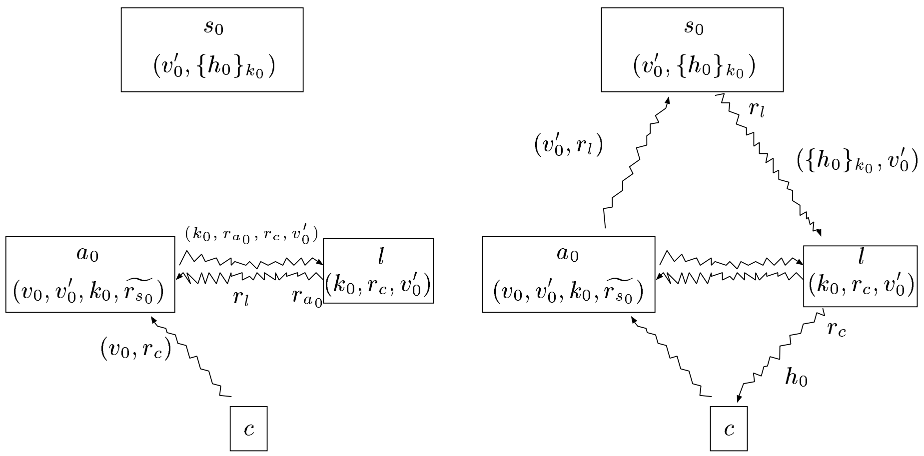

Fig. 3. Retrieval

Share	h_0
Key	k_0
Storer	s_0
Client	c
Decrypter	l
Random numbers	v_0, v'_0
Return addresses	$r_{a_0}, r_{s_0}, r_l, \widetilde{r_{s_0}}, r_c$

To retrieve a document (see Figure 3), the client c asks the forwarder a_0 (and each a_i in the same way) to retrieve the share h_0 by sending them an anonymous message with v_0 and their anonymous return address r_c. The forwarder a_0 then picks a random server l to act as a decrypter and sends it k_0, the key it is storing which decrypts the stored share, v'_0, r_c, and a return address r_{a_0}, getting back a return address for l. Now a_0 forwards r_l and v'_0, which identifies the share, to s_0 via one of the $\widetilde{r_{s_0}}$ (r_{s_0}). Now s_0 looks up the encrypted share corresponding to v'_0 and forwards it and v'_0 to l, which decrypts the share and sends it to the client via r_c. The process continues until c has accumulated enough shares to reconstruct the document.

We note that when the forwarder starts running out of return addresses for the storer (you can use each one only once), all he needs to do is request some more via one of the return addresses it still has.

The other important detail which we have so far left out of the description of the system is that we use the P2P layer (PAST) to replicate state among neighbouring nodes. This enables requests to be routed to any of the nodes which contain the replicated state. In particular, the forwarder shares $(v_0, v'_0, k_0, \widetilde{r_{s_0}})$ with neighbouring nodes which can therefore also answer requests. Similarly, the storer shares $(v'_0, \{h_0\}_{k_0})$. The decrypter does not need to share anything as he will only get one request to decrypt the share and will then give up this role.

3 Commentary on the Protocol

In the last section we presented the protocol. Here we will try to explain some of our design decisions and show how they relate to the properties we want our system to satisfy.

There are several novel aspects in the design of our censorship resistant system as compared to existing architectures:

- Replication. The use of a P2P layer like PAST to replicate state in forwarders and storers.
- Forwarders. The use of forwarders to provide an extra layer of indirection and prevent the storers from being visible by clients.
- Encryption of shares. Storing the shares in an encrypted form and keeping the keys at the forwarders.
- Decrypters. The use of separate nodes (as opposed to, for instance, forwarders) to decrypt shares.

3.1 Replication

Replication of state in the system provides fault tolerance, efficiency and prevents several kinds of attacks.

Firstly, it makes denial of service attacks and simple efforts to take down individual forwarders ineffective because there are always a number of other hosts ready to forward requests. Furthermore, even if the attacker succeeds in taking down a particular forwarder, state will be replicated onto a new node which will also start forwarding requests.

More subtly, it reduces the link between any particular forwarder and the share which has been retrieved. This is because the address for a particular forwarder (for instance, a_0) which is published in the anonymous newsgroup denotes a dynamic set of physical hosts rather than a single machine. This is due to the behaviour of the underlying P2P system (PAST): if asked to route a request to a node (a_0), it does not necessarily forward it to that specific node, but instead to any node which shares state with a_0. Therefore, it is not easy to establish precisely which physical machines the address a_0 represents, indeed, this set changes as machines go down and come back up.

However, this introduces a slight complication. Although the forwarders share state, they cannot share private keys (it would be impossible to keep these keys secret because the set of forwarders constantly changes as nodes go up and down). Therefore requests addressed to them must be delivered in plaintext[4]. This turns out not to be problem here as the attacker who watches traffic arriving at the forwarders sees v_0 and r_c. The former is public anyway, and the latter gives away no information about c itself (but enables him to send a fake share to c[5]).

3.2 Forwarders

The use of forwarders serves several purposes. First of all, they help protect the storers against "Rubber Hose Cryptoanalysis" by hiding them from the clients. Secondly, they can help provide active-server document anonymity by randomly introducing new dummy requests into the system and dropping some of the valid ones. Thus, it will make it hard for the storer to find out (part of) which document it is storing, even by acting as a client in the system. Finally, we use the forwarders to store keys which decrypt the shares and forward them to the decrypters.

3.3 Encryption of Shares

We have argued that the storers should not be able to see the content they are storing to prevent the possibility of them being pressured into censorship. Therefore, we must make them store the shares in an encrypted form and stop them from getting hold of the keys which would decrypt the shares. Thus these keys

[4] This means that the last layer of the onion is not encrypted. Therefore, the message is still anonymous but not secret.

[5] This does not constitute an attack as the adversary would have to perform this active and therefore expensive operation for every share every client requests.

cannot be published as this would enable the storers to retrieve all of them and see which one decrypts each of the shares. Hence they are stored by the forwarders.

3.4 Decrypters

Some would argue that the use of decrypters is superfluous. The storer could just send the shares back to the forwarder who would send them back to the client. However, this would expose the forwarder to the risk of being caught red-handed with the share. Furthermore, they might be pressured into installing a filter to censor shares which the attacker does not like. As the forwarder is the publically visible part of the system (and therefore most vulnerable), we decided to delegate the task of decrypting the share to a completely different entity who does not have any information about what it is decrypting.

4 Discussion

In this section we discuss the limitations of our system.

First, one should question the validity of assuming a public key infrastructure on a P2P network. We need each peer to be able to retrieve the public key of any other peer and verify that the key belongs to that particular peer. The simple solution is to use a global repository. However, such a scheme would limit the scalability of the system. We believe that better solutions exist, and are actively working on this problem. A related issue is the fact that working on top of a peer to peer system may result in attacks. For instance, if the attacker is able to arrange requests not to reach a particular set of nodes corresponding to a forwarder at this level (by, for instance, modifying routing tables in some of the nodes of the P2P system), he will effectively censor the corresponding share. We do not address these problems here, but merely point out that this is an active research area and one which we need to pay close attention to. A nice summary of the problems in security of Peer to Peer systems and some solutions to them can be found in [SM02,Dou02].

Secondly, we should consider how likely the forwarders are to suffer from "Rubber Hose Cryptoanalysis" – they are certainly visible to the attacker and contain information which is necessary for share retrieval. However, we argue that they are much less likely to be subjected to such pressure than, for example, Publius servers for the following reasons:

- They are not storing the offending document, not even in an encrypted form, so their connection with it is somewhat indirect.

- They do not store the identities of the s_is, so an attack to try to get it out of them will not succeed.
- The share does not actually go through the forwarders a_i after publication is completed.
- The requests addressed to the forwarder a_i are likely to end up being handled by a number of different physical hosts.

A slight modification to the protocol (which is beyond the scope of this paper) can be introduced to further reduce the role the a_i play in the protocol, and therefore reduce the potential for them to be attacked.

We must also consider the number of compromised peers it takes to remove a document from the system. A possible attack is as follows: each forwarder a_i remembers (rather than forgetting) the corresponding storer s_i at the time of publication in the hope of exposing them later, if the content turns out to be offending. Once the document has been successfully published, a_i notes the correspondence between the random integer (v_i) published with the document and one in its lookup table, and works out the fact that s_i is storing a share of a particular document. It can now pressure s_i into removing the share. However, the chances of the $n - k + 1$ peers picked as a_i being compromised are small and the peers have to be compromised at the time of publication, otherwise the attack fails.

Having stated that we provide active server anonymity, we must pay attention to the amount of information the storer can gain by repeatedly requesting the document and noticing the requests coming in for the share it is storing. However, a suitable number of random requests generated by the forwarders should weaken this attack. Again, the precise details are beyond the scope of this paper.

We also note that we are presenting just one part of a design of a system. Many questions are left unanswered and a few attacks are not addressed. For example, the attacker can simply flood the censorship resistant system with random data so that "real" documents cannot be inserted. This design does not include protection against such an attack.

Neither do we deal with accountability in any systematic way. Consider a scenario where the attacker is powerful enough to insert many nodes into the system. Each of these, when asked to act as a forwarder, replies "ok", but drops the share and fails to answer subsequent requests. In this design, we are relying on the inability of the attacker to insert enough malicious nodes to censor documents in this way. We felt that although standard methods which are summarised in [DFM00] can be used in this system, they are inappropriate in this context or do not provide adequate protection. Therefore, we leave this for future work.

5 Related Work

A variety of censorship resistant systems have been designed, some of which have also been implemented. We have already discussed Publius, Dagster and Tangler but, perhaps the system closest to ours in terms of the aims it tries to achieve is Free Haven [DFM01].

It is built on top of an underlying network of anonymous remailers and deals with reader anonymity, server anonymity and censorship resistance. However, it uses a Gnutella-like search for retrieval of shares of the document. More specifically, a user request to retrieve a particular document gets broadcast from the user node to all the neighbouring nodes, and so on. When a request arrives at a peer which has a matching share, it gets sent off to the requester via a chain of remailers. This scheme for locating files is rather inefficient, and, as the Gnutella experience has shown, does not scale for large numbers of peers. Furthermore, we note that peers frequently exchange shares with each other. This is costly in terms of network bandwidth and makes it hard to provide guarantees that a document will be located. Our system aims to be more efficient both in terms of bandwidth and share retrieval. Finally, Free Haven has not been implemented, perhaps because it contains complex notions of share trading, reputation, etc. We note, however, that moving the individual shares around in the system is an interesting technique for increasing censorship resistance which may be incorporated into our system. For example, the shares may be moved periodically, and the state in the forwarders updated.

6 Conclusion

We have presented a design of a system which deals with censorship resistance and satisfies strong anonymity requirements. Although (like many other similar systems) it has not yet been implemented, we hope that the fact that it is based on top of an existing infrastructure will make the job easier.

We have argued for building an anonymous censorship resistant system on top of a peer to peer architecture and demonstrated the feasibility of doing so to provide strong anonymity and robustness guarantees.

We have two main future objectives. Firstly, having described an anonymity system and claimed that it satisfies some properties, we consider it worthwhile to formalise those and prove them rigorously.

Finally, we would like to build a prototype implementation on top of a P2P layer like PAST and a suitable anonymity system.

Acknowledgements

I acknowledge support from EPSRC grant GRN24872 *Wide-area Programming* and EU grant PEPITO. A variety of ideas have resulted from conversations with Richard Clayton, George Danezis, Peter Pietzuch, Peter Sewell and Keith Wansbrough and from comments by various members of the Cambridge Security Group.

References

[And96] R. J. Anderson. The eternity service. In *Pragocrypt*. 1996. `http://www.cl.cam.ac.uk/users/rja14/eternity/eternity.html`.

[Cha81] D. Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Communications of the A.C.M.*, 24(2):84–88, 1981.

[CMW+01] S. A. Craver, J. P. McGregor, M. Wu, B. Liu, A. Stubblefield, B. Swartzlander, D. S. Wallach, D. Dean, , and E. W. Felten. Reading between the lines: lessons from the SDMI challenge. In *Information Hiding Workshop*. 2001.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Federrath [Fed01], pages 46–66. `http://freenet.sourceforge.net`.

[DeC] Gallery of CSS descramblers. `http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/`.

[DFM00] R. Dingledine, M. J. Freedman, and D. Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter 16. O'Reilly, 2000.

[DFM01] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven Project: Distributed anonymous storage service. In Federrath [Fed01], pages 67–95. `http://freehaven.net`.

[Dou02] J. R. Douceur. The sybil attack. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*. Cambridge, MA, 2002.

[DR01] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *The 8th Workshop on Hot Topics in Operating Systems*. 2001.

[Fed01] H. Federrath, editor. *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. ISBN 3-540-41724-9.

[GRS99] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM (USA)*, 42(2):39–41, 1999.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[SM02] E. Sit and R. T. Morris. Security considerations for peer-to-peer distributed hash tables. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*. Cambridge, MA, 2002.

[SW01] A. Stubblefield and D. Wallach. Dagster: Censorship-resistant publishing without replication. Technical report, Rice University, 2001.

[WM01] M. Waldman and D. Mazieres. Tangler: A censorship resistant publishing system based on document entanglements. In *8th ACM Conference on Computer and Communcation Security (CCS-8)*. 2001.

[WRC00] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*. 2000.