Proof of Service in a Hybrid P2P Environment

Jun Li and Xun Kang

Department of Computer and Information Science,
University of Oregon,
{lijun, kangxun}@cs.uoregon.edu

Abstract. In a hybrid peer-to-peer environment, clients can either directly download data from their server, or share data with each other. In order to create incentives for clients to share data and decrease server load, an effective economy model is for the server to credit those *provider* clients who provide data to others and offer discounts to those *recipient* clients who download data from provider clients instead of the server. To realize this model, the proof of service between provider and recipient clients must be provided.

We design and investigate three different schemes and compare them in terms of scalability, effectiveness, and cost. We emphasize the issues of lessening the number of proofs which must be provided, avoiding a heavy load on the server, and ensuring the proof for *every* piece of data served. Our study shows our enhanced public-key-based scheme to be the most effective.

1 Introduction

Sharing data over the network can either be based on a client-server model, or on a peer-to-peer mechanism. Interestingly, these seemingly contradictory paradigms can actually complement each other. While a server can independently serve data toward multiple individual clients, its clients can form a peer-to-peer relationship to share data from the server among themselves [1,2,3], reducing the amount of data clients must directly download from the server.

The combination of peer-to-peer and client-server paradigms creates a new, hybrid peer-to-peer data service environment. Both a server and its clients can potentially benefit from this environment: the server does not need to serve every client directly, and a client may also be able to obtain data from other peer clients that are closer than the server.

An obstacle to realizing such benefits is whether clients are willing to share data among each other. In another words, how can a client have incentives to provide services to its peer clients?

To answer this question, a simple but generic economy model can be introduced as follows: (1) Every client will only pay the server directly, and does not pay any other client; (2) A client who helps others (a *provider client* providing data, also called *provider*) will receive credits for assisting the server. (3) A client who is helped by other clients (a *recipient client* receiving data, also called

G. Chen et al. (Eds.): ISPA Workshops 2005, LNCS 3759, pp. 64–73, 2005.

recipient) will pay less for the data, since it does not directly utilize as much of the server's resources; (4) By offloading some tasks to provider clients, the server will serve more clients overall and thus make more profit, even though it charges each individual client less. Note that a provider may have stronger but indecent incentives if it violates this model and directly "sells" data illegally to other clients. Our focus in this paper is to provide positive incentives that will benefit everyone.

This economy model provides incentives to every entity involved, including the server, the provider clients, and the recipient clients. However, it also faces serious challenges with regards to the issue of trust. For instance, what if a client masquerades as a provider client and cheats the server by claiming that it has offered other clients a large amount of data, thus requesting credits? What if a recipient client lies to the server by reporting that it received much less data than it really received, or that it never received any data at all? Can a recipient client receive data from hundreds of providers but never acknowledge the service they provided?

In order to harden the economy model above, a trustworthy, effective proof-of-service mechanism must be designed. With proof of service, a client can present to its server a proof about its service to others, a server can verify whether or not a client has indeed served others, and a recipient cannot deny, cheat or be cheated about its reception of data from others.

In this paper, we first briefly describe related work, then illustrate the design of three proof-of-service schemes. We also report our experimental results to show that under this hybrid environment, effective proof of service can be implemented with a reasonable cost.

2 Related Work

The proof of service toward a recipient could be realized by obtaining a non-repudiable receipt about the service from the recipient. So proof of service in this paper can be regarded as one particular case of non-repudiation service. In fact, there have been quite a few non-repudiation schemes designed in different contexts, focusing on non-repudiation of origin, receipt, submission, and delivery [4]. Louridas also provided guidelines for designing non-repudiation protocols [5]. Verification of non-repudiation schemes have also been studied [6,7,8].

Proof of service is very similar to fair exchange of information, which can be either without or with a trusted third party (TTP) [4]. In the context of this paper, fairness would mean for a provider to receive a proof of its service and for a recipient to receive the desired data. (Note that unlike most fair exchange protocols, a recipient does not need to obtain the proof about the identity of the provider.) The most closely related to this paper is the fair exchange with an offline TTP. While leveraging current schemes, our solution in this hybrid environment has an important difference in that a server itself can act as a TTP for its provider and recipient clients. This is also an inherent advantage for enforcing fairness. Further note that the server is also the original source of the

data that a provider offers to a recipient, bringing another advantage: If needed, a server can verify the data without requesting them from other nodes, thus avoiding a drawback in many TTP-based solutions, especially when the data is of large size.

Rather than alleviating the load on a TTP as in other fair exchange solutions, ours will focus on keeping the server lightly loaded. A big challenge in our context is that there can be thousands of clients of a server, and every recipient client may be related to a large number of providers. Our solution must scale as the number of clients grows. Furthermore, in this hybrid environment clients may collude to attempt to gain illegal proof of service.

3 Overview

Assuming every data object (such as a file) is divided into multiple blocks our general approach is to enforce an interlocking block-by-block verification mechanism between every pair of provider and recipient. ("Block" in this paper is an abstract concept which can just be an application-specific data transmission unit.) For every block that a provider has sent to a recipient, the recipient will verify the integrity of the block (which is through an orthogonal mechanism that we will not cover in this paper), and send back an acknowledgment to the provider. On the other hand, the provider will verify the acknowledgment *before* providing the next block. Those verified acknowledgments can then be used to form the proof of the service that the provider has offered to other clients, and they must be non-repudiable and can be verified by the server.

Three severe problems arise in this basic solution and must be handled:

- **Proof Explosion Problem.** If a provider has to present a separate proof for *every* block it served, it can be a very large number of proofs for its server to handle. Note that there can also be a large number of providers. So, an acknowledgment should be able to aggregate the receipts of recent blocks.
- **Server Overload Problem.** If a recipient has to resort to its server for composing every non-repudiable acknowledgment, or a provider has to fall back on the server for verifying every acknowledgment, or if they frequently seek other help from the server, this will not be a scalable solution. Especially since a large number of clients may be sharing a single server.
- **Last Block Problem.** After a recipient receives the last block it needs from a provider, the recipient could deliberately decide not to send an acknowledgment for this last block. *Note that this last block is not necessarily the last block of a data object.* Except for some simple cases (such as providing the last block of a data object), the provider has no way of knowing whether a particular block will be a recipient's last block to request.

In the following, we first introduce a simple solution based on shared secret key cryptography. It is not scalable as it heavily relies on the server as an inline TTP. Then we introduce the second solution based on public key cryptography, which scales in that the server will be used as an offline TTP. But, while this

solution normally is effective, in some circumstances the last block problem is a concern, and we present a third solution.

We assume each client first contacts a server to establish a secure SSL channel between them. This is done by running the SSL protocol [9], which also helps the client obtain a certificate of the server's public key, and set up a secret key shared between the client and the server. The client then sends its request regarding particular data objects. While the server can directly serve the data, in this paper we assume that the server will issue a *ticket* to the client to authorize the client to retrieve data from other provider clients. The client then contacts one or more providers and presents its ticket to them. (A more detailed procedure on how clients find each other is out of the scope of this paper.) After verifying the ticket, a provider begins providing data to the client.

4 Shared-Secret-Key-Based Proof of Service

In this scheme, the server acts as an inline TTP. When a recipient sends back an acknowledgment for a block it received from a provider, the acknowledgment is protected using the secret key shared between the recipient and the server. The provider is not able to decrypt the acknowledgment, and it forwards the acknowledgment to the server. The server then verifies it and returns the verification result to the provider.

If the acknowledgment is verified as authentic, the server will also use it as a proof that the provider just sent a block to the recipient. Furthermore, only when the acknowledgment is verified as authentic will the provider go ahead and provide the next block to the recipient.

Figure 1 shows the procedure of this scheme. The recipient r has received block b_i. It verifies the integrity of the block, sends an acknowledgment $ack(b_i)$ to the provider p, and requests the next block b_{i+1}. The acknowledgment is in the following format:

$$k_r\{pid,\ rid,\ oid,\ i,\ timestamp\} \tag{1}$$

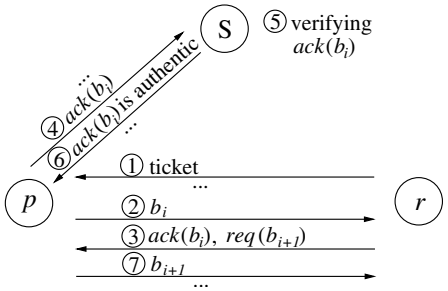

Fig. 1. Shared-secret-key-based proof of service

where k_r is the shared secret key between r and the server S for signing the acknowledgment; *pid* and *rid* are the ids of p and r, respectively; *oid* and i specifies which blocks of which data object are being acknowledged; *timestamp* records when the request for b_i was issued.

This scheme is subject to the proof explosion problem, since a separate proof must be presented for every block that a provider served. It is also subject to the server overload problem in that the server has to verify the authenticity of *every* acknowledgment resulting from the interlocking verification process, and every block that a recipient receives will lead to a new acknowledgment. Lastly, it also does not address the last block problem.

5 Public-Key-Based Proof of Service

In this solution, when a recipient wants to receive data from a provider, it will not only present a ticket, which is prepared by the server to authorize the recipient to receive data from other providers, but it will also present the certificate of its own public key. When a recipient receives a data block, it will then apply the same procedure as in Section 4, except that it will (1) sign the acknowledgment using its private key, and (2) include a *sack* field instead of the index of the most recently received block. The following shows the format of the acknowledgment:

$$p_r\{pid,\ rid,\ oid,\ sack,\ timestamp\} \tag{2}$$

where p_r is the private key of the recipient client r.

The *sack* field solves the proof explosion problem. It is in a format similar to the SACK options for the TCP protocol [10]. It can express *all* the blocks that the recipient has received from the provider, instead of just the most recent one. For example, it can be $[0-56, 58-99]$ to confirm the reception of the first 100 blocks of a data object except for the block 57. This way, a provider only needs to keep the most recent—thus also the most complete—acknowledgment as the proof of its service to indicate all the blocks it provided.

Figure 2 shows an example when r acknowledges its receipt of blocks up to block b_i, and requests the next block b_{i+1}. Different from the shared-secret-key-based solution, here the provider verifies the acknowledgment by itself before sending block b_{i+1}. Recall it has the public key of the recipient and is thus able to verify it. Therefore, this solution also solves the server overload problem.

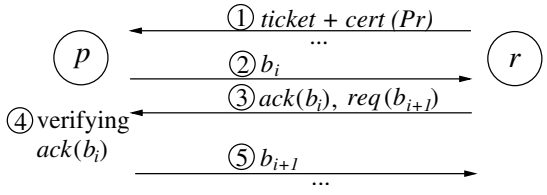

Fig. 2. Public-key-based proof of service

6 Enhanced Public-Key-Based Proof of Service

6.1 The Basic Idea and Its Issues

Neither of the above two solutions solves the problem of *last block cheating*. We now introduce a *block-key-based* mechanism. If a block is *not* the last block, we can just use the solution from Section 5. Otherwise, if the provider knows that the ith block b_i from a data object *oid* is the last block to send to a recipient *rid*, it will generate a secret *block key* k_i^r:

$$k_i^r = f(pid, rid, oid, i, k_p) \tag{3}$$

where f is a one-way hash function and k_p is the secret key shared between the provider and the server. Then, as shown in Figure 3, instead of delivering the original block b_i to the recipient, the provider will encrypt the block with this block key, and send the encrypted block to the recipient. Upon the receipt of the encrypted block, the recipient will acknowledge its receipt of this block, using the same format as in Equation (2). Here, the recipient *must* acknowledge the receipt of this block in order to receive the block key k_i^r to decrypt $k_i^r\{b_i\}$.

There are still a few issues to consider, however, given that both the provider and the recipient may be dishonest. First, this solution assumes that the provider knows block b_i is the last block that the recipient wants to receive. Second, this solution allows a provider to obtain an acknowledgment of the last block even if the provider did not provide the recipient with a correctly encrypted block or an authentic block key. Third, it is possible that the server may be overloaded with too many (honest or dishonest) requests related to the last block issues—for example, what if a recipient complains to the server frequently that it did not receive the block key even if it did? We address these issues in the following.

6.2 Determine the Last Block

The provider can treat every block as potentially the last block that a recipient would receive from it, and apply to every block the approach shown in Figure 3. Doing so, the provider will obtain the proof of its service of every block, including the last block. In case the provider is certain that the current data block is not the last block (such as via out-of-band knowledge), or if the provider does not mind missing the proof of just one last block, it can simply apply the original public-key-based proof-of-service scheme.

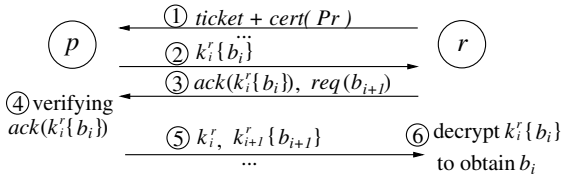

Fig. 3. Enhanced public-key-based proof of service

6.3 Ensure the Correctness of Encrypted Block

For acknowledging an encrypted block $k_i^r\{b_i\}$, we enhance Equation (2) to also include the digest (using a one-way hash function) of $k_i^r\{b_i\}$, denoted as $d(k_i^r\{b_i\})$, in the acknowledgment $ack(k_i^r\{b_i\})$:

$$p_r\{pid,\ rid,\ oid,\ sack,\ timestamp,\ d(k_i^r\{b_i\})\}. \tag{4}$$

The provider p will verify $ack(k_i^r\{b_i\})$, including the digest, to decide whether or not to provide the block key k_i^r to the recipient r. This way, when p presents the server this acknowledgment as the proof of its service, the server can verify whether r received the correctly encrypted last block, i.e., correct block b_i encrypted using correct block key k_i^r. Note that the server can use Equation (3) to calculate the correct block key k_i^r.

Also, after r receives k_i^r, it can decrypt $k_i^r\{b_i\}$ to obtain b_i. Furthermore, it can verify the integrity of b_i. Only if b_i is integral will r continue with p for the next block. In case b_i appears to be corrupted, r knows that p cannot use $ack(k_i^r\{b_i\})$ as a proof of its service. r will not send $ack(k_{i+1}^r\{b_{i+1}\})$ to p; otherwise p can use it as the proof of its service, hiding the corrupted delivery of block b_i. The recipient r can either request b_i from p again, or decide not to continue with p.

Unfortunately, this stop-and-go process can have poor performance. To remedy this, we require every acknowledgment to include digests of last m encrypted blocks, and the server will verify whether the recipient received the correctly encrypted blocks for last m blocks (instead of just last one block). In step 5 of Figure 3, upon the receipt of k_i^r and the encrypted block b_{i+1} (i.e. $k_{i+1}^r\{b_{i+1}\}$), r will first immediately acknowledge the receipt of b_{i+1}, then invoke a separate process for decrypting block b_i and verifying its integrity. This process can be repeated for the next $m-1$ blocks, greatly improving the performance; in case that block b_i is discovered corrupted, p will still not be able to have a proof that it successfully delivered b_i—since the proof must show correct digests of all last m blocks. Here, we want to select m carefully such that it is small enough to be scalable, but large enough to keep a high level of parallelism.

6.4 Ensure the Correctness and Availability of Block Key

To ensure the correctness of a block key, the provider p can sign it with its private key p_p, replacing k_i^r in step 5 of Figure 3 with a *protected block key*:

$$p_p\{pid, rid, oid, i, k_i^r\}. \tag{5}$$

The recipient r can decrypt it to obtain k_i^r. If r cannot decrypt $k_i^r\{b_i\}$ correctly, it can forward the protected block key to the server so that the server can verify if p sent a wrong block key. In case p did not provide a block key at all (i.e., no step 5 in Figure 3), r can retrieve it by asking the server to apply Equation (3) to calculate the block key.

6.5 Keep Server Lightly Loaded

Throughout the whole design, the server's load has been kept very light. A provider can wait until the end of serving a recipient to present a single proof of its service toward this client. Also, in addition to obtaining the ticket and certificate at the beginning, the only type of query that a recipient can issue is to verify or retrieve the block key of a block it receives from a provider (see Section 6.4). Because there can only be one last block between a recipient and a provider, this kind of query happens only once per recipient-provider pair.

7 Experimental Results

We have implemented a framework to support peer-to-peer data sharing among clients of a common server, and used this framework to evaluate the cost and performance when enforcing the public-key-based proof of service solution and its enhanced version, which we will denote as P and P_e, respectively. (Note that we do not evaluate the shared-secret-key-based solution since it is not feasible.) In addition, our framework can be configured to enforce a subset of three orthogonal security functions: client authentication, data integrity protection, and data confidentiality protection, denoted as A, I, C, respectively.

We measured several metrics with and without proof of service for comparisons. We compared the scenarios AI vs. AIP and AIP_e, and the scenarios AIC vs. $AICP$ and $AICP_e$. (Note that P or P_e always needs to be enforced together with A and I.) Cryptography parameters are: 112-bit 3DES for secret key, 1024-bit RSA for public key, and MD5 for message digest.

7.1 Server Capacity

Server capacity is defined as the number of client requests that a server can serve per time unit. Measuring the server capacity before and after applying a proof-of-service solution can test whether the solution would significantly impact the server. For both P and P_e, our design requires a server to perform the same operations; therefore, the server capacity will be exactly the same. We measured the server capacity of an iBook machine running MacOS X with a 700 MHz processor and 384 MB of memory. While the server capacity for the AI scenario is 360 requests per minute on average, it decreases by 93 (26%) to be 267 after adding P or P_e. At the same time, a 23% decrease (330 to 253 requests per minute) occurs when enforcing P or P_e on top of the AIC scenario.

7.2 File Downloading Time

File downloading time is the latency from the time that a client establishes a connection with a server for requesting a file to the time that it receives the whole file. It consists of a startup latency, which is the time that the client spends in handshaking with the server before sending out a request for the first block of

a file, and data transferring time, which is the rest of file downloading time. Measuring file downloading time with and without P or P_e can indicate how a user may feel the impact of P or P_e when downloading a file.

In our experiment, we use the same server as in our server capacity measurement, and every client machine (either provider or recipient) is a Dell Latitude D810 machine running Linux 2.6.9-ck3 with a 1.73 GHz Pentium M processor and 512 MB of memory. We connect all of them on an isolated subnet to avoid background noise so that our comparison of file downloading times under different scenarios can be accurate and consistent.

Regarding the startup latency with and without proof of service, adding P or P_e on top of the AI scenario will increase the startup latency by approximately 0.2 and 0.3 seconds, respectively. The results for the AIC scenario is similar. Both are acceptable to users downloading files.

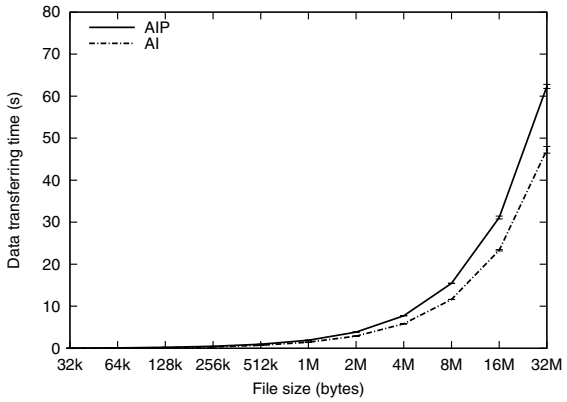

Fig. 4. Data transferring time (with 95% confidence interval)

We now analyze the data transferring time with and without proof of service. Since in our design whether or not the data is encrypted does not affect the data transferring time, we only look at scenarios without confidentiality. Furthermore, our design makes P and P_e have the same amount of data transferring time—in P_e, block encryption and decryption operations at a provider or a recipient are conducted out of band, and is not part of the data transferring time. As a result, we can just compare the data transferring times between the AI and AIP scenarios. Figure 4 shows the results. We can see that for each scenario, while the size of a file doubles, so does the data transferring time. Furthermore, for the same file size, adding P will increase the data transferring time, but within a reasonable range. For instance, adding proof of service can increase the data transferring time of a 4 MB file from 5.8 seconds to 7.7 seconds.

8 Conclusions

If clients of a server share data among themselves, great benefits can be obtained in that the server will have a reduced load and each client can choose close peer clients to obtain data. With such a hybrid peer-to-peer environment, an economy model can further be defined to create incentives for peer-level sharing: provider clients can gain credits, recipient clients can pay less, and the server can still make profit as it can afford to serve a larger population.

However, in order for the above to be true, a critical prerequisite is to enable providers to obtain non-repudiable, trustworthy proofs of the service they have given recipient clients. In this paper, we presented three different proof-of-service schemes—a shared-secret-key-based scheme, a public-key-based scheme, and an enhanced public-key-based scheme. While the shared-secret-key-based scheme is not scalable, we have shown that with the latter two schemes, a server will only be lightly loaded. The enhanced version can further ensure that a provider can obtain the proof for *all* the data it served. Experimental results have also shown that the final solution is effective and can be applicable for real usage with a reasonable cost. Future work of this research includes obtaining a formal verification of the protocol and conducting further studies on actual deployment.

Acknowledgements

We thank the anonymous reviewers for their comments. This research also benefited from discussions with Toby Ehrenkranz and Eric Anderson. Toby also helped prepare the camera-ready version of this paper.

References

1. BitTorrent, Inc.: BitTorrent. `http://bittorrent.com` (2005)
2. Stavrou, A., Rubenstein, D., Sahu, S.: A Lightweight, Robust P2P System to Handle Flash Crowds. In: ICNP. (2002) 226–235
3. Sherwood, R., Braud, R., Bhattacharjee, B.: Slurpie: A Cooperative Bulk Data Transfer Protocol. IEEE INFOCOM (2004)
4. Kremer, S., Markowitch, O., Zhou, J.: An Intensive Survey of Fair Non-Repudiation Protocols. Computer Communications **25** (2002)
5. Louridas, P.: Some Guidelines for Non-Repudiation Protocols. Computer Communication Review **30** (2000) 29–38
6. Zhou, J., Gollmann, D.: Towards Verification of Non-Repudiation Protocols. In: Proceedings of 1998 International Refinement Workshop and Formal Methods Pacific, Canberra, Australia (1998) 370–380
7. Schneider, S.: Formal Analysis of a Non-Repudiation Protocol. In: CSFW, Washington, DC, USA (1998) 54
8. Kremer, S., Raskin, J.F.: A Game-Based Verification of Non-Repudiation and Fair Exchange Protocols. Lecture Notes in Computer Science **2154** (2001) 551+
9. Freier, A.O., Karlton, P., Kocher, P.C.: SSL Protocol Version 3.0. `http://wp.netscape.com/eng/ssl3/ssl-toc.html` (1996)
10. Mathis, M., Mahdavi, J., Floyd, S., Romanow, A.: IETF RFC 2018: TCP Selective Acknowledgement Options (1996)