mSSL: A Framework for Trusted and Incentivized Peer-to-Peer Data Sharing Between Distrusted and Selfish Clients

Jun Li

Abstract Conventional client-server applications can be enhanced by enabling peer-to-peer data sharing between the clients, greatly reducing the scalability concern when a large number of clients access a single server. However, for these "hybrid peer-to-peer applications," obtaining data from peer clients may not be secure, and clients may lack incentives in providing or receiving data from their peers.

In this paper, we describe our mSSL framework that encompasses key security and incentive functions that hybrid peer-to-peer applications can selectively invoke based on their need. In contrast to the conventional SSL protocol that only protects client-server connections, mSSL not only supports client authentication and data confidentiality, but also ensures data integrity through a novel exploit of Merkle hash trees, all under the assumption that data sharing can be between untrustworthy clients. Moreover, with mSSL's incentive functions, any client that provides data to its peers can also obtain accurate proofs or digital money for its service securely and reliably. Our evaluation further shows that mSSL is not only fast and effective, but also has a reasonable overhead.

Keywords Hybrid peer-to-peer application · Peer-to-peer security · Data sharing incentive · Data integrity · Proof of service · Atomic purchase · mSSL · SSL

Jun Li
University of Oregon
Computer and Information Science
Eugene, OR 97403-1202
Tel.: +1-541-3464424
Fax: +1-541-3464423
E-mail: lijun@cs.uoregon.edu

1 Introduction

Client-server applications often do not scale well when a server is serving a large number of clients. To address this, a recent trend is to allow the clients, such as those of a web server, to share data between themselves in a peer-to-peer fashion [1,2,3,4]. If a server has provided certain data to some clients, when other clients come to request the same data, the server can either still follow the conventional client-server model to directly provide data to these clients, or instead ask these clients to obtain data from their peers who already have the data. In this paper we also call this kind of application which integrates the conventional client-server paradigm and the peer-to-peer paradigm a *hybrid peer-to-peer (P2P) application.* It can make the data of a server—even if the server is under-provisioned—tremendously more available to thousands of clients.

However, hybrid P2P applications also face new, major challenges that to our knowledge no existent research or tools have systematically addressed. We look especially at the following three aspects:

1. **Securing data sharing between distrusted clients.** Compared to receiving data directly from a server, indirectly receiving data from arbitrary, often untrustworthy, and even malicious peer clients is subject to much higher security risks. Specifically, how should a server allow peer-to-peer data sharing without weakening client authentication and access control? How should data confidentiality be supported? Moreover, when a client retrieves data, whether from a server or its peer clients (which may manipulate data), can the integrity of the data be guaranteed—especially with low overhead and high speed?

2. **Building strong incentives for clients—who may be selfish and distrustful of one another—to share data.** Call a client who provides data to

others a *provider client* (or provider), and a client who receives data from others a *recipient client* (or recipient). A client may lack incentives to be either a provider or a recipient, a serious obstacle to realizing the benefits with peer-to-peer data sharing. For a client to be a recipient, unless it is certain that the data it receives from a provider will be *the same* as that from its server, and can be encrypted when needed, it probably has no motivation to contact its peer clients for data. Questions raised in securing data sharing above will also be raised here. On the other hand, for a client—which is probably selfish—to be a provider, it may also be unwilling to provide data to others without being able to obtain benefits from doing so. In another words, if a client provides data to others, can it obtain an accurate, authentic, and non-repudiable proof or digital money for its data provision service?

3. **Supporting hybrid P2P applications with different needs of security and incentives.** Hybrid peer-to-peer applications such as software distribution, video streaming, or critical information dissemination, may have different security and incentive policies. For example, for an application that disseminates early warnings of upcoming attacks to the public, it may choose not to enforce the information access control and confidentiality, and only enforces the data integrity. An audio/video sharing application, on the other hand, may decide client authentication is critical in order to allow a provision site to reliably track who has been requesting multimedia files. Clearly, a one-size-fits-all design may not work in accommodating the security and incentive needs of many different hybrid P2P applications.

Although there are many cryptography techniques already, simply applying them cannot effectively address these challenges. A new solution is needed. As we will present in this paper, we have designed, built, and evaluated a framework called *mSSL* (pronounced "missile"). The reason we name our work "mSSL" is simple: while conventional SSL (Secure Socket Layer) [5] involves one server and one client, mSSL involves one server and *multiple* clients. But note mSSL differs from SSL in much more fundamental ways: While SSL is mainly a security protocol for the secrecy and integrity protection between a server and a client, mSSL is not even a protocol but rather a framework that unifies multiple security and incentive functions specifically designed for hybrid P2P applications, as follows:

1. **It includes a set of security functions to protect both client-server and thousands of client-client communications**. mSSL departs from SSL in that mSSL targets a different model (hybrid P2P data sharing). It addresses both conventional security issues such as data integrity, client authentication, and data confidentiality and new security issues such as providing a trustworthy proof of data transferring service between clients, and ensuring atomic data purchase between clients. In designing these functions, mSSL not only leverages existing cryptography and security techniques, but also devises its own new algorithms by considering the pattern and scale of data sharing between clients in hybrid P2P applications, the fact that clients are often untrustworthy, and the storage and traffic overhead constraints.

2. **It includes a set of incentive functions to build and strengthen the incentives for peer clients to share data**. mSSL supports protected data sharing between selfish clients with either accurate and authentic proofs of service, or digital money through atomic peer-to-peer data purchase. Via mSSL, a recipient can be assured of the integrity and secrecy of any data it receives, thus allowing it to safely receive data either directly from its server or indirectly from one or multiple providers. A selfish provider also becomes willing to provide data to its peer clients: It can gain legitimate credits for itself based on its service, and it can prove to its server the service accurately, securely, using a small proof, and with minimal overhead; or, it can gain digital money after providing data to others via atomic peer-to-peer data purchase.

3. **It allows an application to invoke different mSSL functions based on its needs at its discretion.** In supporting hybrid P2P applications that sit on top of mSSL, mSSL can be treated as a middleware service, and is flexible in supporting the heterogeneous security and incentive needs of various hybrid P2P applications. An application can configure which security functions from mSSL to invoke, decide whether and how to leverage the incentive mechanisms from mSSL, and specify parameters of cryptography algorithms and the hybrid P2P environment that is specific to the application. Furthermore, while in this paper we will mainly discuss the security and incentive functions that we have already designed and implemented, the mSSL framework is extensible. In case that certain functions that an application needs is missing, an application can add its own new security and incentive functions, or extend the existing ones.

mSSL aims to be fast in executing its functions, secure in counteracting various attacks, and scalable in allowing thousands of clients and large quantities of data.

Its security functions may be used in different combinations for different applications, and its success will depend on its efficacy under all those combinations. Of particular concern is the overhead introduced by mSSL and how much an application could be slowed down while benefiting from the addition of mSSL. Using currently designed and implemented security and incentive mechanisms, as will be reported in this paper, our performance evaluation of mSSL shows that the overhead of mSSL is reasonable and generally very small.

The rest of this paper is organized as follows. We first provide an overview of mSSL in Section 2, introducing security and incentive functions of mSSL. We then describe our design of mSSL's five functions in Sections 3 to 7, covering data integrity, client authentication, data confidentiality, proof of service, and atomic peer-to-peer data purchase, respectively. In describing the proposed framework and its every function, this paper not only describes *how* they work, but also justifies *why* they work correctly, sometimes using theorems with proofs. We discuss attacks toward these functions of mSSL and countermeasures in Section 8. In Section 9 we discuss how a hybrid P2P application may invoke a combinination of mSSL functions. In Section 10 we evaluate the performance of mSSL. Section 11 describes the related work and we conclude the paper in Section 12.

2 mSSL Overview

In this section, we first introduce the two data access modes for hybrid P2P applications, and introduce security functions mSSL currently provides for them. We then describe our two economy models that provide incentives for peer-to-peer data sharing, and point out what mSSL does to secure the incentive mechanisms. We also describe how an application can invoke mSSL's functions.

2.1 Data Access Modes and mSSL's Security Functions

For a client to obtain its server's data, such as a video file, mSSL assumes two access modes: *direct access* and *indirect access*. In *both* modes, a client will first contact its server to request data—this step is necessary for the server to control and track the data downloading process, and can be helpful for security and incentive designs as we will see later. Assume the server determines that the client is permitted to obtain the data (such as after a client paid for purchasing a file). If in the direct access mode, the server will directly transfer the data to the client, but if in the indirect access mode, the client has to contact its peer clients for the data.

In either mode, a hybrid P2P application may need to protect its data sharing process. One goal of mSSL is to provide a suite of light-weight, robust and scalable security functions that hybrid P2P applications can rely on. These critical security functions currently include:

- **Data integrity function (Section 3)** to detect almost immediately any data fabrication or modification with low overhead;
- **Client authentication function (Section 4)** to ensure only authenticated clients can obtain a server's data, whether or not directly from the server; and
- **Data confidentiality function (Section 5)** to avoid data leakage to untrusted entities.

Regarding security functions, our main contribution is a new mechanism for ensuring data integrity. The design of client authentication and design of data confidentiality are both straightforward.

2.2 Incentive-Oriented Economy Models and mSSL's Trustworthy Incentive Functions

First, to provide incentives to every entity involved in data sharing, including the server, the provider clients, and the recipient clients, mSSL allows every application to choose a specific economy model such that recipients can "purchase" data and providers can gain credits or digital payments. It is assumed that applications will decide specific values of payments and credits. mSSL currently supports two different economy models. The main difference between these models is that one allows payments to go only to the server, whereas the other also allows payments to go to provider clients. The models are as follows:

- *Single-point-of-payment model.* Without losing generality, this model is as simple as follows:
    1. Every client will only pay the server directly for the data it receives, and does not pay any other client;
    2. A provider will receive credits for assisting the server;
    3. A recipient will pay less for the data if it does not directly utilize as much of the server's resources; and
    4. By offloading some tasks to provider clients, the server will likely serve more clients overall and thus make more profit, even though on average it charges each individual client less.
- *PPay model.* PPay (PeerPay) [6] is a cheat-proof micropayment protocol that allows nodes in a peer-to-peer system to receive coins from a broker, pay each other with such coins while exchanging data

or information, and cash the coins with the broker. The PPay model with mSSL can work as follows: When a client cannot obtain data from its server directly, the client can purchase digital money from the server (the server acts as a broker for its clients), and pay other provider clients with the money to purchase the data. In terms of benefiting and motivating every party involved, this model is the same as the single-point-of-payment model except that a recipient client can directly pay its provider clients.

However, both models face serious problems with regards to the issue of trust. Both models must be bundled with security functions so that a recipient can be assured to receive data with integrity (and secrecy when needed) from its peers. Furthermore, in the single-point-of-payment model, a provider must be able to provide an accurate and authentic proof of its service so that its server can authenticate the proof and thus credit him for its service. In the PPay model, a provider must receive payment after providing service and a recipient must receive data after paying a provider; i.e., their transaction must be either complete or never started.

mSSL addresses these problems. In addition to designing security functions as described in Sections 4 to 3, we also designed the following two incentive functions to support both incentive-oriented economy models:

– **Proof of service (Section 6)** to allow a client in the single-point-of-payment model to obtain a precise, authentic, and non-repudiable proof that it has provided specific data-sharing service to other clients;
– **Atomic peer-to-peer data purchase (Section 7)** to ensure that the data purchase between a provider and a recipient in the PPay model must be either complete or never started.

2.3 Using mSSL

Hybrid P2P applications can invoke specific security and incentive functions of mSSL with specific parameters based on their needs. With the five functions mSSL currently provides, i.e., data integrity, client authentication, data confidentiality, proof of service, and atomic data purchase, every hybrid P2P application may have its own needs and thus choose a specific combination. For example, mSSL's data integrity function can be useful when a server is providing critical public information and all that is needed is to guarantee the integrity of the data. The combination of data integrity, client authentication, and data confidentiality may be chosen when encrypted data sharing using just the data confidentiality function is not sufficient; sometimes even if a

recipient can decrypt data from a provider successfully, it may not trust the provider and still want to verify the integrity to make sure the data is indeed unaltered from the server's original. Note that every hybrid P2P application is autonomous; i.e., it will only deal with its own server and set of clients, with no relevance to servers or clients from other applications (it is possible that a machine is involved in multiple hybrid P2P applications, but in this case the machine will be running multiple processes, each corresponding to a different hybrid P2P application that it is involved with). Also note that the mSSL functions are independent from each other but they do have compatibility issues. We detail in Section 9 how different mSSL functions may be combined when they are invoked.

Moreover, every application can choose specific parameters and configurations in invoking mSSL. An application can request specific algorithms with preferred key lengths for applying classical cryptography, public key cryptography, and secure hashing. An application can also pass every mSSL function arguments based on its context and preference. For instance, in invoking the data confidentiality function the application can indicate what confidentiality approach it selects (object-key-based or group-key-based as will discussed in Section 5); in invoking the data integrity function, it can notify mSSL of the size of a data object and the block size it chooses; and in invoking the proof of service or atomic data purchase function, it can specify its preferred level of parallelism for simultaneously handling multiple data blocks, as well as plug in credit or pricing values for its selected economy model.

3 mSSL Function 1: Data Integrity

In this section, we present our design of the data integrity function. When a hybrid P2P application is configured to support mSSL's data integrity function, every recipient client will be able to request from their server or provider clients not only data, but also data integrity information in order to verify the integrity of the data. The data integrity function of mSSL has the following features:

– A client can verify the integrity of every individual block of a data object *fast* and *promptly* (right after it receives the block). It does not have to wait until receiving an entire data object to detect if anything is wrong, nor does it need to verify a signature per block or preload all authentic block hashes.
– mSSL assumes a recipient may receive blocks of the same data object from multiple—perhaps even hundreds of—provider clients, and employs a *recipient-*

driven, on-demand data integrity solution. The recipient decides from whom to pull data blocks as well as integrity information.

- By introducing a new concept called *integrity path*, mSSL supports near-zero latency for receiving the first data block, allows a recipient to download much fewer hashes $(O(n))$ as opposed to that in a conventional Merkle-Hash-tree-based solution $(O(nlogn))$, and also greatly optimizes the integrity verification process.

mSSL assumes every data object can be divided into multiple blocks, and allows a client to verify the integrity at the *block* level. The size of a block is *application-specific*. Compared to signing an entire data object for integrity verification, a block-based solution allows a client to request a retransmission of an incorrectly received block immediately. This advantage is most apparent when the size of a data object is very large, such as an audio or video file of several hundred megabytes or gigabytes.

One simple approach to data integrity is to bundle a signature with every block of an object. However, signature verification block by block will lead to a high computational overhead, especially when compared to verifying the correct hash value for every block. One approach to block hash verification is to build a signed superblock for a data object that contains the strong one-way hash value of every block of the object and a signature of the entire superblock. The superblock will be transmitted first before transmitting data blocks. After a client receives an authentic superblock of a data object, it then uses the hash values from the superblock to verify the integrity of each block that it later receives. This solution avoids the need of per-block encryption or decryption.

Unfortunately, a superblock can be very large and can incur a high startup latency. For example, a 1-gigabyte file with 4-kilobyte blocks and a 64-byte hash per block will have a superblock with a total of 16 megabytes. As the size of a data object increases, the size of its superblock will also increase linearly. Worse, if the superblock itself is corrupted, the retransmission of the superblock itself can also be costly. As a result, before a client—especially a low-bandwidth client—receives the very first block, a large superblock can easily cause a significant delay, often unacceptable to applications in which users prefer prompt response, such as when downloading a large multimedia file. Increasing the block size can reduce the size of a superblock, but the retransmission cost of individual blocks will increase.

Below, we devise a new way of using a binary Merkle hash tree to achieve data integrity that is both fast and efficient. Except getting to know our notations, readers
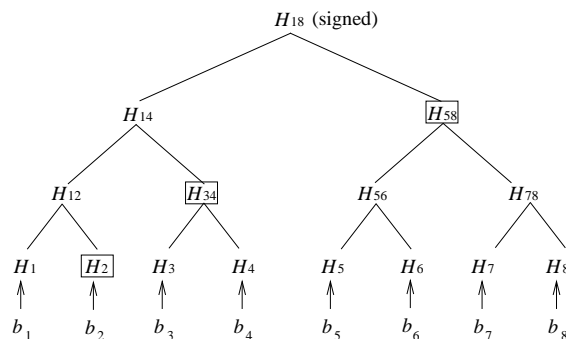

Fig. 1 Merkle hash tree of a data object with 8 blocks. The authentication path of block b_1 is $< H_2, H_{34}, H_{58} >$.

familiar with Merkle hash trees [7] can skip Sections 3.1 and 3.2.

3.1 Merkle Hash Tree of a Data Object; Notations

Every data object can have a Merkle hash tree (e.g., Figure 1). For a data object O divided into 2^m blocks, let $M(O)$ denote its binary Merkle hash tree. The height of $M(O)$ is then $m+1$, with the root at level 0 and leaf nodes at level m.

The following are notations for representing a node or its value on $M(O)$:

H^l: A node at level l.

H_i^l: The ith node (counting from left) at level l. The first node is H_1^l.

H_i: The ith leaf node counting from the left, i.e., H_i^m.

H_{ab}: The root node of the subtree containing leaf nodes H_a through H_b (inclusive).

$M(O)$ can be derived from the bottom up using a strong one-way hash function f as follows: (1) *Leaf nodes.* For every block b_i $(i = 1, ..., 2^m)$, $H_i = f(b_i)$. (2) *Non-leaf nodes.* For every two sibling nodes H_{2i-1}^l and H_{2i}^l, the value of their parent H_i^{l-1} is $f(H_{2i-1}^l, H_{2i}^l)$. If a node H_{2i-1}^l has no sibling, its parent H_i^{l-1} equals H_{2i-1}^l. Applying the parent calculation process recursively, we will obtain the value of every non-leaf node, including the root node H^0. An important property here is that it is computationally infeasible for an attacker to modify a hash value on $M(O)$ or a block of O without changing the value of H^0.

3.2 Conventional Merkle-Hash-Tree-Based Integrity Verification and Its Drawback

The conventional usage of a Merkle hash tree in data integrity verification is already advantageous in that

it does not require a client to download all the hash values beforehand as in a superblock-based solution. It does not require expensive encryption or decryption operations, either. To verify the integrity of every block of the data object O, a client can first request the certified (signed) value of H^0. Once it receives a block b, the client can also request b's *authentication path* $A(b) = \langle H^m, H^{m-1}, ..., H^1 \rangle$ to calculate H^0 by using the procedure in Section 3.1. $A(b)$ contains exactly one particular hash value from every level of $M(O)$, where H^{i-1} is the sibling of H^i's parent. H^{i-1} is also called H^i's *uncle*. If block b is modified (or any values on $A(b)$ is modified), the calculated H^0 will not equal the certified H^0, thus detecting an integrity violation. The client can then request the retransmission of block b. A similar solution is proposed in [8].

Unfortunately, this solution can lead to a high traffic overhead. For a data object with 2^m blocks, every block's authentication path will have m hash values. Assuming each hash value is 64 bytes, the overhead traffic will then be $64m * 2^m$ bytes, $\frac{64m}{|b|}$ of the data traffic ($|b|$ is the number of bytes of a block)! In Section 3.4 we will see that the verification procedure here is also *not* optimal.

3.3 mSSL *Integrity Path*

We apply a new usage of the Merkle hash tree in our recipient-driven data integrity solution. mSSL invents a concept called the "mSSL integrity path" with which a recipient—while receiving blocks from its server or one or many providers—can determine what hash values it needs for every block and quickly verify every block with low traffic and storage overhead. Some of our ideas are similar to those in [9]; we list our major differences from them in Section 11.

The integrity path of a block is only composed of those hash values from its authentication path *that are not locally available*. In fact, when a client needs to obtain an authentication path $A(b)$ of a newly received block b, it may not need to download every hash value h in $A(b)$. In fact, if $h \in A(b')$, where b' is another block and has been verified earlier, h must be already available locally. Moreover, certain hash values will have to be calculated during the local block integrity verification process, and can become available for verification along other authentication paths.

To determine what hash values are on a block's integrity path, we introduce Theorem 1 below:

Theorem 1 *Define $M(O)$ as the Merkle hash tree of object O. If a recipient client r has a hash value h from*

$M(O)$ *locally, then r also has $uncle(h)$ (h's uncle on $M(O)$) locally.*

Proof Define $T = \{h \mid h$ is a hash value on $M(O)\}$, $D = \{h \mid h \in T$ and h is a downloaded hash value$\}$, $C = \{h \mid h \in T$ and h is a locally calculated hash value$\}$, and L as the set of locally available hash values. Clearly, $L = D \cup C$. We now need to prove if $h \in L$, *i.e.*, $h \in D \cup C$, then $uncle(h) \in L$.

If $h \in D$, then $\exists$ an authentication path A_0 that recipient r obtained before, such that $h \in A_0$ (otherwise r will not have h locally). From the property of an authentication path, we know $uncle(h) \in A_0$. Now that A_0 is a path obtained before, $\forall H^l \in A_0, H^l \in L$. Therefore, $uncle(h) \in L$.

If $h \in C$, because it is impossible that all of h's descendants are calculated, there must be a descendant h_x of h such that $h_x \in D$. Denoting $A(h_x)$ as the authentication path that contains h_x. Then, from the property of an authentication path, $uncle(h) \in A(h_x)$. Now, since $A(h_x)$ was obtained before, and $\forall H^l \in A(h_x), H^l \in L$, therefore, $uncle(h) \in L$.

Theorem 1 further leads to Theorem 2 below that can help determine the integrity path of any given block.

Theorem 2 *If a block b's authentication path $A(b)$ is $\langle H^m, H^{m-1}, \cdots, H^1 \rangle$, and H^{l-1} is locally available, but H^l is not, then b's mSSL integrity path $mip(b)$ is $\langle H^m, H^{m-1}, \cdots, H^l \rangle$.*

Proof If H^{l-1} is locally available, but H^l is not, then according to Theorem 1, we know $H^{l-1}, H^{l-2}, \cdots, H^1$ at the upper levels are all locally available, and none of $H^m, H^{m-1}, \cdots, H^l$ at the lower levels are. Thus, $mip(b) = \langle H^m, H^{m-1}, \cdots, H^l \rangle$, with a total of $|mip(b)| = m-l+1$ hash values. $|mip(b)|$ is also the number of levels needed to be downloaded in order to have a complete authentication path.

3.4 mSSL's On-demand Data Integrity Solution via Integrity Path Concept

The recipient-driven, on-demand data integrity solution of mSSL contains three main components: obtaining the integrity path of a block, verifying the integrity of a block based on its integrity path, and maintaining the Merkle hash tree of a data object being downloaded. Figure 2 shows how a client c requests a data object O from its server S or a provider p and then verifies the data integrity block by block. We describe each component below.
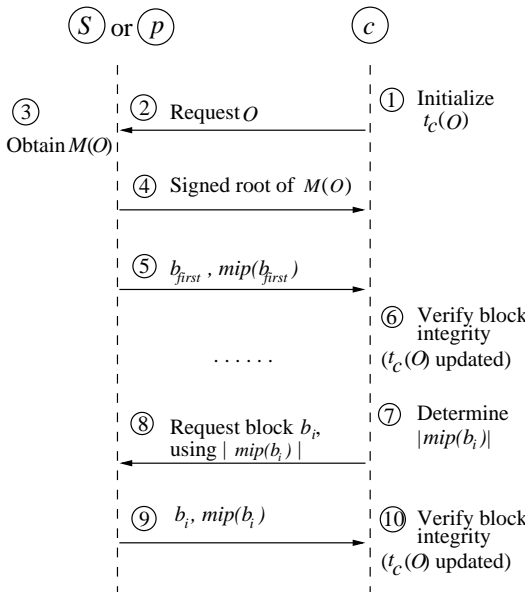
Fig. 2 The integrity solution of mSSL. A client c is requesting a data object O from its server S or a provider p. Except for the first block b_{first}, client c requests on demand the integrity path of every block along with the block itself. Client c maintains a Merkle hash tree of O, i.e., $t_c(O)$.

3.4.1 Obtaining Integrity Path

A client can use Theorem 2 to determine a block b's integrity path $mip(b)$. When a client requests block b from a server or a provider, it can also request $mip(b)$. Interestingly, we can simplify the request for $mip(b)$. The client can simply specify the length of $mip(b)$, and the server or the provider can then determine what $mip(b)$ is. The following Theorem 3 explains why.

Theorem 3 *If a node knows the length of the integrity path of block b, $|mip(b)|$, it can determine b's integrity path $mip(b)$.*

Proof Assume block b's authentication path is $A(b) = \langle H^m, H^{m-1}, \cdots, H^1 \rangle$. From Theorem 2, block b's integrity path $mip(b)$ is $\langle H^m, H^{m-1}, \cdots, H^l \rangle$ where $l = m + 1 - |mip(b)|$.

3.4.2 Integrity Verification

After a client receives the integrity path of a block b, i.e., $mip(b)$, the integrity verification of block b can also be simplified. On one hand, the client could use $mip(b)$ to derive block b's authentication path $A(b)$, and apply the procedure from Section 3.2 to calculate and check against the root of the Merkle hash tree to verify the integrity of block b. For a data object with a total of 2^m blocks, this will result in a total of $m \cdot 2^m$ hash calculations when no retransmission is needed for any

block. On the other hand, we can devise a much more efficient way for integrity verification that only requires $2^{m+1} - 1$ hash calculations when no retransmission is needed.

Theorem 4 *In verifying the integrity of a block b from a data object O, if the authentication path of block b is $A(b) = \langle H^m, H^{m-1}, \cdots, H^l, H^{l-1}, \cdots, H^1 \rangle$ and b's integrity path is $mip(b) = \langle H^m, H^{m-1}, \cdots, H^l \rangle$, the integrity of block b can be checked against the sibling node of H^{l-1} on the Merkle hash tree of O, instead of the root of the tree.*

Proof Clearly, H^{l-1} is already locally available. We can deduce that an earlier integrity verification that involves H^{l-1} must have used $sibling(H^{l-1})$—which is H^l's parent—to calculate all of H^{l-1}'s ancestors—including the root. An *authentic* $sibling(H^{l-1})$ must also be locally available already! As a result, the verification of block b actually only needs to compare a newly calculated $sibling(H^{l-1})$ with the current $sibling(H^{l-1})$ to decide if block b is correct; the additional calculations from level $l - 1$ up would be repeated calculations and thus unnecessary.

Note that when integrity verification of a block fails, there are two possibilities: One is that the block integrity is corrupted, the other is that some hash values from the block's integrity path are wrong. A recipient cannot know which, so it will have to request the block and its integrity path again. Using the integrity path—which is generally much shorter than the authentication path—will cause less chances to receive broken hash values, potentially avoiding more unnecessary retransmissions.

We can further summarize the computation overhead of this integrity verification method in the following theorem:

Theorem 5 *During the verification of blocks from a data object O with 2^m blocks, with the integrity verification method in Theorem 4, if every block is correct, every hash value on the tree will be calculated once and only once.*

Proof It is easy to see that every hash value on O's Merkle hash tree will be calculated *at least* once. We now show that when no retransmission is needed every hash value is calculated no more than once. First, any hash value at the leaf of the tree is the hash of a particular block of O; without retransmission, obviously it only needs to be calculated once per block. Second, for any intermediate hash h on the tree, if mSSL has checked against h once for verifying the integrity of a block according to Theorem 4, and the verification is successful,

then mSSL will know the authentic hash values of any child of h. Any checking against h can be replaced by checking against one of h's child hashes.

3.4.3 Maintaining Merkle Hash Tree

As described above, while a recipient client receives blocks of a data object from a provider client (or its server), both the recipient and the provider (or the server) must maintain the Merkle hash tree of the data object in order to verify the integrity of those blocks. Moreover, as the recipient receives and verifies new blocks, it also updates the tree: New hash values will be added, and old ones that will not be usable any more will be removed. (Note that if this recipient is going to act as a provider of this data object to others, it then will not remove hash values from the tree.)

Theorem 6 *For a block b from a data object with 2^m blocks, after a recipient verifies its integrity by checking against a hash h at level l, the number of hashes that will be changed is $\Delta = m - l - 1$.*

Proof First, when l equals to m, after verifying block b, the recipient will simply remove h from the tree since h is calculated once already and will not be calculated (thus used) later. The integrity path of b is empty and no new hashes are downloaded externally. Therefore, $\Delta = -1$, i.e., $\Delta = m - l - 1$.

Second, when $l < m$, the integrity path of b contains $m - l$ hash values downloaded externally. They now need to be added. Meanwhile, each of these downloaded hashes has a corresponding hash at the same level that is calculated during the verification; from Theorem 4, all these calculated hashes are now calculated once already and will not be needed later, thus will not be stored. Hash h will also be calculated and thus removed. As a result, verifying b will also result in adding $m - l - 1$ hashes.

In Section 10.4, we will further show that if a recipient sequentially receives blocks from a data object with 2^m blocks, it only needs to keep at most $m + 1$ hash values on the Merkle hash tree.

4 mSSL Function 2: Client Authentication

Whereas the data integrity function enables every client to verify the data it receives, in certain hybrid P2P applications the server also needs to ensure every client is authenticated. To do so, a hybrid P2P application can invoke mSSL's client authentication function.

The implementation of this function is straightforward. When data access from a client uses direct access mode, the client authentication is essentially the same as in the conventional client-server. If indirect access mode instead, mSSL further activates a ticket-based client authentication mechanism.

More specifically, in both direct access and indirect access modes, a client will first contact its server. If client authentication is required by the server, the client will run SSL to create a secure channel between itself and the server and then authenticate itself, such as using its account name and password or using an identity certificate. Once authenticated, the client can then request data from the server.

If the client needs to download data from its peer clients, i.e., the client is in indirect access mode, the client authentication needs an extra procedure. Basically, after the server authenticates the client, the server will provide a ticket for the client to contact other peer clients. The ticket includes the ID of the client, the ID of the data object that the client requests, the time that the ticket is issued, the validity period of the ticket, and a sequence number. The server also uses its private key to sign the ticket so that any client that knows the server's public key can verify the ticket. When the client presents the ticket to its peers, the ticket can prove that the server has authorized this client to download data from other clients. Any peer client can verify the ticket and authenticate the client in question before providing data.

5 mSSL Function 3: Data Confidentiality

In order to ensure that only authenticated clients can access data, a hybrid P2P application may further use mSSL's data confidentiality function. Mostly by leveraging existing cryptography methods, mSSL offers two different approaches to data confidentiality that a hybrid P2P application can choose from: an object-key-based approach, or a group-key-based approach.

5.1 Object-Key-Based Confidentiality

With this approach, every data object is associated with an object key for encrypting or decrypting the data object. An object key can have a life time and be replaced when it expires. Essentially an object-oriented approach, this scheme is able to enforce a fine-grained access control at the data object level. A server can encrypt any data object just once in advance for all potential clients, instead of once per client.

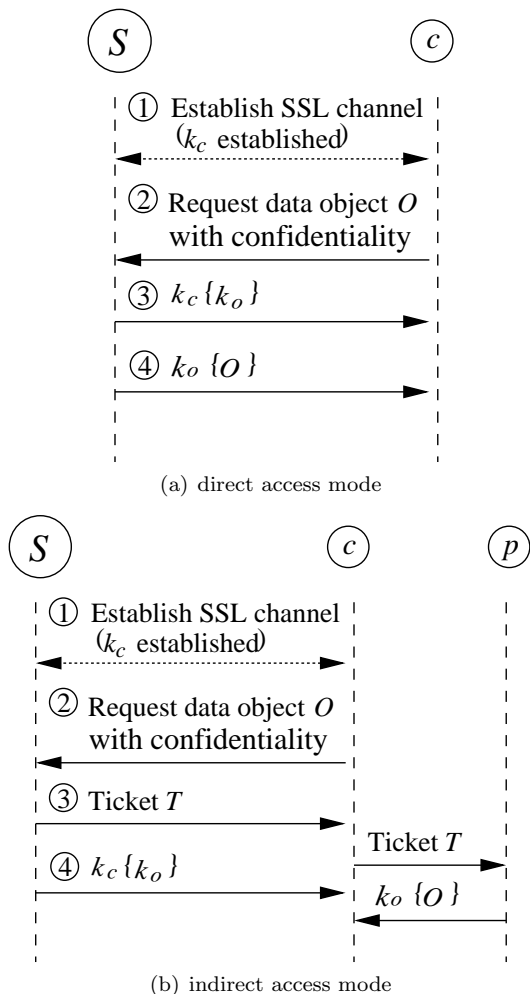As shown in Figures 3(a) and 3(b), when a client c requests a data object O in its encrypted form $k_o\{O\}$

(a) direct access mode

(b) indirect access mode

Fig. 3 Object-key-based confidentiality through mSSL.

from a server S, S can issue O's object key k_o to c immediately after authenticating c's request (or after c finishes downloading $k_o\{O\}$). After c receives $k_o\{O\}$ from either c's peers or S, c can use k_o to decrypt $k_o\{O\}$. Note that the distribution of k_o is protected by k_c, a secret key established between c and S over their SSL channel.

5.2 Group-Key-Based Confidentiality

Authenticated clients of a server can also form a group. In this situation, these clients can share a group key, and all data from the server to these clients can be encrypted using the group key.

Group key management is the core issue here. Fortunately, this topic has been well studied. In particular, we can have the server of the clients also act as the key server of the group! The server can distribute the key only to the authenticated clients in the group. Both the

data traffic from the server to these clients and traffic between these clients can be encrypted using this key. Research in [10] provides a detailed study of generating a symmetric group key, including handling the dynamic joins and departures of group members.

6 mSSL Function 4: Accurate and Authentic Proof of Service

When a hybrid P2P application is configured to use the single-point-of-payment model (Section 2), it provides an incentive for its clients to share data: While every client pays only its server for the data it receives, provider clients obtain credits for providing data to their peers. However, clients can lie about what data they provided or received. In this section, we illustrate the proof-of-service function we designed. In a hybrid P2P application with this function underneath, whenever a recipient receives data from a provider, the provider can present to its server an accurate proof of its service to others, a server can authenticate whether or not that service indeed happened, and no recipient can repudiate, cheat, or be cheated about its reception of data from others. The server can then accurately and confidently decide how to credit a provider, solidifying the incentives for provider clients to provide data to other clients.

A solid proof-of-service function must meet the following requirements:

- It must ensure trustworthiness of proofs;
- It must be scalable since a large number of clients of a single server may be involved and a data object can be composed of a large number of blocks;
- It must not overload the server; and
- It must not incur high traffic and storage overhead.

6.1 Basic Proof-of-Service Solution and Its Problems

In our basic proof-of-service solution, every data object is divided into multiple blocks and we enforce an interlocking block-by-block verification mechanism between every pair of provider and recipient. For every block that a provider has sent to a recipient, the recipient will verify the integrity of the block and sends back a *non-repudiable* acknowledgment. The provider will verify the acknowledgment *before* providing the next block. Those verified acknowledgments can then be used as the proof of the service that the provider has offered to other clients.

Problems arise in this basic solution, however. If a provider has to present a separate acknowledgment

as the proof for *every* block it served, there will be a proof explosion with large files. Also, after receiving a block from a provider, a recipient may run away without acknowledging the receipt of this block, leaving the provider unable to have a proof for serving the block. Finally, the server can be overloaded if requested by clients to compose or verify acknowledgments frequently.

6.2 Overview of mSSL's Proof-of-Service Solution

We address those problems from the basic proof-of-service solution. At a high level,

- A recipient will receive an encrypted block first, and then receive the decrypting "block key" *after* acknowledging the receipt of the block (*note that doing so will make it unnecessary to invoke mSSL's data confidentiality function between clients*);
- Acknowledgments are now *cumulative*—the most recent acknowledgment can replace previous ones as the proof of service;
- Every acknowledgment is signed and can be verified using the public key of the recipient; and,
- Most operations are between a provider and a recipient, and their server is seldom involved and thus will not be overloaded.

Figure 4 shows our proof-of-service solution between a provider p and a recipient r. Note that no matter how many clients a server may have, our solution can be employed by any given pair of provider and recipient. It includes four basic components that we will illustrate in Sections 6.3—6.6:

- *Handshake:* In steps 1 and 2, recipient r will present its authentication ticket to provider p (Section 4), and they will exchange their public key certificates and negotiate a session key.
- *Encrypted data delivery:* In step 3, r requests a block b_i from p, and in step 4 p sends block b_i encrypted using a secret block key to r.
- *Acknowledging encrypted delivery:* In step 5, r acknowledges its receipts of the encrypted b_i. p verifies the acknowledgment in step 6.
- *Data decryption and verification:* If r's acknowledgment is verified, in step 7 p sends the block key in its protected form to r. Then in step 8, r obtains the block key, uses it to decrypt the encrypted block, and verifies the integrity of the block.

6.3 Handshake

In addition to the ticket-based authentication, the handshake component is mainly to exchange public keys be-
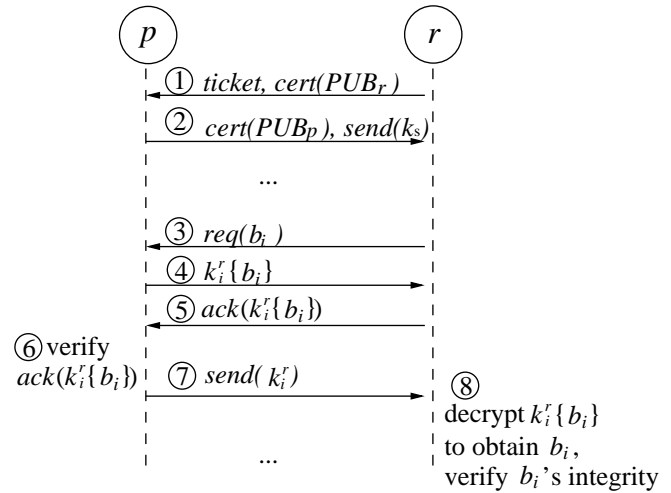

Fig. 4 mSSL's proof-of-service solution.

tween a recipient and a provider, and establish a session key for their communication. A prerequisite here is, how can clients obtain their public key certificates in the first place. This issue is important because clients are often just ordinary users without certified public keys.

We solve this issue when a client initially interacts with its server. Noticing that when a client runs the SSL protocol with its server, the server itself has a certificate for its own public key. We actually can use the server as a public key CA (certificate authority) to certify the public keys of its own clients! A client needs to generate beforehand a key pair by itself using a public key generation algorithm, then send a certificate request to the server. The server in turn will generate a certificate that is signed with the server's private key and send it back to the client. As a result, every client can have its own public key certified by the server, and every client is also able to verify the certificate of its peers.

After the provider and the recipient exchange their public keys, they can establish a session key k_s between them. In step 2, p generates k_s and then sends it to r through $send(k_s)$, which is:

$$PUB_r\{\ PRV_p\{k_s\}\ \}. \tag{1}$$

6.4 Encrypted Data Delivery

Encrypted delivery provides an interlocking mechanism between p and r. After r sends a request to receive block b_i from p, p will deliver an encrypted version of b_i to r, and hold on the encryption key until r acknowledges the receipt of the (encrypted) b_i. Step 4 shows that a secret block key is used for encrypting a block. When the provider pid sends the ith block b_i from a data

object *oid* to a recipient *rid*, it will generate a block key k_i^r as:

$$k_i^r = g(pid, rid, oid, i, k_p) \qquad (2)$$

where g is a one-way hash function and k_p is the secret key shared between the provider and the server. Note that this formula also allows the server to calculate the block key—which is useful when r needs to request the block key from its server instead of p (Section 6.6).

6.5 Acknowledging Encrypted Delivery

Recipient r must acknowledge its receipt of an encrypted block before it can receive its block key. In step 5, r acknowledges the receipt of the encrypted block $k_i^r\{b_i\}$. It forms the following acknowledgment $ack(k_i^r\{b_i\})$:

$$PRV_r\{pid,\ rid,\ oid,\ sack,\ timestamp,\ d(k_i^r\{b_i\})\}\ (3)$$

where PRV_r is the private key of r for signing the acknowledgment (so that r cannot deny it later), $timestamp$ records when the acknowledgment is issued, $d(k_i^r\{b_i\})$ is the digest of the encrypted block using a one-way hash function d.

To solve the proof explosion problem, we introduce cumulatively acknowledgments. Recipient r uses the *sack* field in $ack(k_i^r\{b_i\})$ to indicate what r has correctly received from p so far. Its format is similar to the SACK option for the TCP protocol [11], and can express *all* the blocks r has received from p, instead of just the most recent one. For example, it can be $[1 - 56, 58 - 128]$ to confirm the reception of the first 128 blocks of a data object except for block 57.

In step 6, p will verify the acknowledgment to decide whether or not to provide the block key to the recipient in step 7. This includes verifying the digest field $d(k_i^r\{b_i\})$ so that when p presents its server S this acknowledgment as the proof of its service, S 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 (2) to calculate k_i^r.

6.6 Data Decryption And Verification

Whereas the provider p has received an acknowledgment from r that p can use as a proof of its service to r, r is still holding an encrypted version of the most recently received block. Step 7 is the delivery of the block key in a protected form $send(k_i^r)$:

$$k_s\{pid, rid, oid, i, k_i^r\} \qquad (4)$$

where k_s is the session key between p and r. Since only p and r know k_s, r—and only r—can decrypt the message to obtain k_i^r.

In case p did not or could not provide a block key at all (*i.e.* no step 7 happening), r can retrieve it by asking S to apply Equation (2) to calculate the block key. When doing so, r must send S an acknowledgment as in Equation (3) so that S knows r has indeed received a correctly encrypted block from p. Also, since the occurrence of this query toward S means that r would probably decide that p should not be relied upon any more, for each provider-recipient pair, there should be only a very small number of such queries toward S.

Now with the receipt of the block key, Step 8 is for the recipient to finally obtain the plaintext block b_i. Recipient r can use the block key to decrypt $k_i^r\{b_i\}$ (received in step 4) to obtain b_i. Moreover, it can verify the integrity of b_i. Only if b_i is correct will the recipient continue with the provider for the next block. In case b_i appears to be corrupted, r knows that p cannot present $ack(k_i^r\{b_i\})$ to S as a correct proof of its service—S knows the correct b_i and k_i^r.

6.7 Further Performance Improvements Through Parallelism

The design so far requires a recipient to verify the integrity of a block b_i from a provider before acknowledging the next block b_{i+1}. (Otherwise, the provider may use the acknowledgment for b_{i+1} as a proof of sending blocks up to block b_{i+1}, even if b_i is corrupted.) This is then a slow, stop-and-go process.

The performance can be further accelerated, and we introduce parallelism in handling multiple blocks concurrently. We require every acknowledgment to include digests of the *last m* encrypted blocks, and the server will verify whether the recipient received the correctly encrypted blocks for the last m blocks (instead of just the last block). Meanwhile, upon the receipt of block key k_i^r and the encrypted block b_{i+1} (*i.e.* $k_{i+1}^r\{b_{i+1}\}$), r will first acknowledge the receipt of b_{i+1} *before* proceeding to step 8. With this design, it can repeat this prompt acknowledgment process for the next $m - 1$ blocks, greatly improving the performance. In case that block b_i is discovered to be corrupt, the provider will still not able to have a proof that it successfully delivered b_i—since the proof must contain correct digests of all of the last m blocks. Here, we want to select m such that it is small enough to be scalable, but large enough to maintain a high level of parallelism.

The server's load has been kept very light throughout the whole design. The only type of query that a

recipient can issue to its server is to verify or retrieve the block key of a block it received from a provider (see discussion on step 7 above), which happens rarely per provider-recipient pair. Also, a provider can wait until the end of serving a recipient to present a single proof of its service toward this client.

7 mSSL Function 5: Atomic Peer-to-Peer Data Purchase

A hybrid P2P application may instead support the PPay model. Different from the single-point-of-payment model, the PPay model allows every provider client to directly make money. A recipient client can deliver coins to its providers to purchase data. But, the PPay protocol is designed for transactions involving only a small amount of money, and the atomicity of data purchase is not critical. A provider can choose not to provide data after being paid, or a recipient may not pay after receiving data. When data shared between clients have a significant worth, it is critical to ensure the atomicity of peer-to-peer data purchase.

We design mSSL's atomic data purchase function under the PPay model that hybrid P2P applications can choose to use. The design principle is as following:

– The digital payment must be trustworthy;
– A recipient client must be able to obtain the data after it paid;
– A provider client must be able to obtain the payment after it delivered data; and
– A server should not be overloaded in helping out the atomic data purchase between its clients.

To design the atomic peer-to-peer data purchase solution, we revise the interlocking method from our proof-of-service solution. We rely on the PPay protocol to ensure that a digital payment between clients is trustworthy, and focus on the exchange of the payment and data below. For any pair of provider and recipient, initially both the payment from the recipient and the data from the provider will be encrypted. Then they try to get plaintext payment and data, respectively. Figure 5 shows the basic procedure of our solution between a provider p and a recipient r. It includes the following components:

– *Handshake:* In steps 1 and 2, recipient r authenticates itself to provider p (Section 4), and they exchange public key certificates and negotiate a session key.
– *Encrypted payment:* In step 3, r requests data from p, say block b_i from a data object, together with its payment for b_i. In order to prevent p from taking the money without providing b_i, the payment is
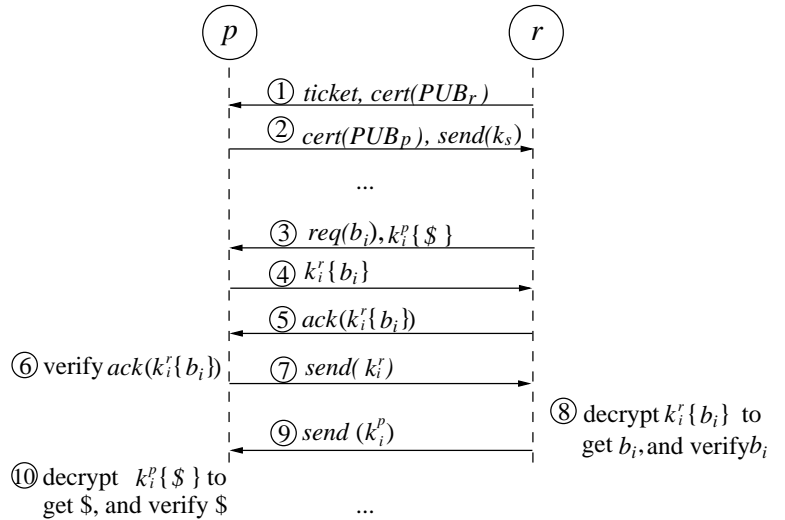

Fig. 5 mSSL's atomic peer-to-peer data purchase solution.

encrypted using a secret "payment key." r will not provide the payment key until r is certain it received b_i correctly.
– *Encrypted data delivery:* In step 4, p sends block b_i to r, encrypted using a secret block key. p will not provide the block key until it is certain that it can obtain the payment from r or its server. (*As in the proof-of-service function, doing so also makes it unnecessary to invoke mSSL's data confidentiality function between clients.*)
– *Acknowledging encrypted delivery:* In step 5, r acknowledges its receipt of the encrypted b_i, and in step 6 p verifies the acknowledgment. The acknowledgment can also help p resort to its server for payment recovery.
– *Data decryption and verification:* In step 7, p sends the block key in its protected form to r. In step 8, r obtains the block key, uses it to decrypt the encrypted block, and verifies the integrity of the block.
– *Payment verification:* In step 9, recipient r sends the payment key to p, and then in step 10 provider p verifies the payment to be correct.

Basically, in addition to the four components from the proof-of-service solution, the atomic peer-to-peer purchase solution adds two extra components. We now only need to focus on these two components:

7.1 Encrypted Payment

When recipient r pays provider p for data, to prevent p from running away with the payment without providing data, r encrypts its payment using a payment key as follows:

$$k_i^p = g(pid, rid, oid, i, k_r) \qquad (5)$$

where g is a one-way hash function and k_r is the secret key shared between r and its server. This formula also allows the server to calculate the payment key—which is useful when p needs to request the payment key from its server instead of r.

7.2 Payment Verification

After a recipient obtains data correctly, it finally delivers the payment key to its provider. As shown in step 9 from Figure 5, the payment key k_i^p is delivered in a protected form $send(k_i^p)$:

$$k_s\{pid, rid, oid, i, k_i^p\} \tag{6}$$

where k_s is the session key between p and r. Since only p and r know k_s, p—and only p—can decrypt the message to obtain k_i^p.

7.3 Ensuring Atomicity

We now show our solution achieves atomicity, i.e., we have either (a) the provider does not obtain payment and the recipient does not obtain data; or (b) the provider obtains payment and the recipient obtains data. In fact, if any step from steps 1 to 4 does not happen, we have (a). Otherwise, if the data is correctly encrypted and delivered, both the provider and the recipient can resort their server for help. The server needs to receive an acknowledgment of the encrypted block (i.e., $ack(k_i^r\{b_i\})$) to first check if the encrypted data delivery was indeed successful; if so, it can calculate and deliver the block key to the recipient and the payment key to the provider. Note that according to Equations (5) and (6), the server knows how to calculate both keys.

If the encrypted data was not delivered correctly, after decrypting the data and verifying its integrity the recipient will discover the incorrectness (step 8). In this case, it will not deliver the payment key to the provider. Since the provider cannot forge an authentic acknowledgment—it must be signed by the recipient—the provider will not be able to obtain the payment key from the server either.

In case that the data was correctly delivered but the payment from the recipient is wrong, again, the provider can resort the server for help by presenting the recipient's acknowledgment (i.e., $ack(k_i^r\{b_i\})$) to their server.

7.4 Parallel Purchase of Multiple Blocks

Whereas the purchase of every block of a data object can be made atomic, to improve performance mSSL allows multiple blocks to be purchased in parallel. This certainly will increase the storage overhead; and moreover, it will potentially increase the load of the server if a provider or a recipient resorts to their server for multiple block keys or payment keys. As a tradeoff between performance and scalability, when a hybrid P2P application invokes this mSSL function, it can specify the parallelism level, i.e., how many blocks can be purchased in parallel.

8 Attacks and Countermeasures

8.1 Overview

Every function of mSSL can be a target of attackers. Among the five functions we described, attacks toward the proof-of-service function and the atomic purchase function are probably the most complicated. In general, two types of attacks may occur: *individual cheating* in which an individual client cheats, such as by misreporting or denying the amount of services it has received or provided, and *colluded cheating* where multiple clients work in collusion, such as by forging false proofs or payments of service together. Because of the similarity between the proof-of-service solution and the atomic purchase solution, in the rest of this section we focus on the former.

The design of the data integrity, client authentication, and data confidentiality functions warrants a straightforward analysis of attacks and countermeasures toward these functions (E.g., to show that the data integrity function is robust against attacks, we can see that a client can obtain enough information to verify the integrity of data it receives. In order to verify the integrity of any block of a data object, including detecting any modification of the block by anyone, the client can obtain a certified root value of the Merkle hash tree of the data object, and the integrity path of the block.) We omit such analysis in this paper.

8.2 Individual Cheating

An individual client can cheat in the following ways:

1. A provider overstates its service for extra credit from the server. It claims that it sent another client certain data although it did not.
2. A recipient refuses to acknowledge its receipt of specific data blocks from a provider, such that the provider cannot show the proof of serving those blocks.

The proof-of-service design in mSSL addresses both methods of cheating above. For the first type of cheating, from Section 6 we know that every proof is a signed

acknowledgment from a recipient. A provider does not know the key other clients use to sign acknowledgments, so it cannot forge any. For the second type of cheating, if a recipient denies its reception of a data block, it will not generate an acknowledgment for the block; without the acknowledgment, it will not be able to receive the secret block key to decrypt the block. So this cheating is also not possible.

8.3 Colluded Cheating

In order to obtain extra non-deserved credits, multiple clients may collude to forge proofs of services that did not actually take place. We list several cases here: (1) A provider forges a proof that it provided itself certain data. (2) While a recipient r sends acknowledgments to its real provider p_1, r (or p_1) also sends a copy of every acknowledgment to its accomplices $a_1, a_2, \ldots, a_n$. Or, r (or p_1) can just copy the acknowledgment that confirms the largest number of received blocks. Then $a_1, a_2, \ldots, a_n$ can all claim that they delivered certain data blocks to r, even though they did not at all. (3) Clients r and p (or a set of providers) collude to forge a proof that p (or a set of providers) provided certain data to r, even though no data transmission ever occurred.

We first see how to counteract cases (1) and (2). Note that the proof-of-service design allows a server to determine whether a proof is indeed about the service from a specific client p to another specific client r. A proof must be signed by the recipient, and it contains the digest of the last one or several encrypted blocks that a provider has served (Equation 3), and the encryption uses a key specific to the provider (Equation 2). In case (1), the server will find out that it is a proof of self-service; in case (2), the server will detect only the proof from p_1 is trustworthy.

Case (3) is tricky in that the server cannot tell whether or not a service has occurred. An economical countermeasure may be the most effective: If the undeserved credit that the provider(s) may gain is always less than what r will be charged, there is no incentive for this colluded cheating.

9 Invoking Functions of mSSL

In this section, we show there are totally *eight* meaningful combinations of mSSL's security and incentive functions that a hybrid P2P application can choose from for its specific needs.

We use I, A, C, P, and $ respectively to represent the five functions mSSL currently provides: data integrity, client authentication, data confidentiality, proof of service, and atomic data purchase. We first study scenarios requiring just *one* of the five. Assuming there is a data object O at a server S:

- I: Every client is allowed to obtain O, but needs to ensure O's integrity. Here, S offers O for everyone, so no confidentiality or client authentication are needed.
- A: Some clients are allowed to access O, some are not. A client thus must authenticate itself.
- C: Confidentiality of O is required. If so, A is also required since encrypted data are only prepared for authenticated clients and the secret key for decrypting O must be delivered only to them. I.e., C implies AC.
- P: Proof of service is required. P must be provided together with I: in the proof of service design, a recipient client must verify the integrity of every received block. P must also be combined with A as any client involved in a proof-of-service transaction must be an authenticated client. As described in the design of proof of service, any client, if authenticated, can also obtain from its server a certificate for its public key, and its proof of service to others can be verified by the server. On the other hand, P includes its own method of encryption, thus P and C are exclusive of each other. Therefore, enforcing P implies enforcing P, I, and A all together, *i.e.* PIA.
- $: Requiring $ is similar to requiring P. Requiring $ also implies requiring $, I, and A all together, *i.e.* IA.

Now we consider possible combinations of *two* of the five functions. A two-function combination cannot have P or $, since they imply PIA or IA. C already implies AC. So the only new combination possible is IA. Furthermore, the only new combination with *three* of the five functions is IAC. We will not have a combination of four, $PIAC$ or IAC, since both P and $ imply not using C. Finally, P and $ are exclusive to each other, and an application can choose at most one of them, so we will not have a combination of all five.

Therefore, we have a total of seven different combinations of current mSSL functions: I, A, C (*i.e.* AC), P (i.e., PIA), $ (i.e., IA), IA, and IAC. With a *none* combination for which no mSSL function is actually needed, we have a total eight different combinations (Note that except for the *none* and I combination, every combination requires A, meaning every client must authenticate itself.)

10 Evaluation

10.1 Goal, Measured Cases, Metrics, and Methods

The goal of evaluating mSSL is to understand its performance and cost, and determine if mSSL has a low cost and good performance.

We evaluate mSSL under different combinations of mSSL functions that are introduced in Section 9. Note in the *none* combination, no mSSL functions are provided except that the mSSL infrastructure is in place. This combination is useful to measure the overhead of the mSSL framework itself and serve as the baseline for measuring the extra overhead introduced by other combinations. That is, we will evaluate mSSL under these eight different combinations: *none*, *I*, *A*, *C* (i.e., *AC*), *P* (i.e., *PIA*), *$* (i.e., *$IA*), *IA*, and *IAC*.

We implemented mSSL in Java and tested it in our laboratory. Furthermore, we have also implemented a file-sharing application as a representative hybrid P2P application that can invoke mSSL for its security and incentive needs. With this application, a client can either directly download a file from a server, or download it from providers that have a copy of the file. The client can ask the server for a list of provider clients, and can become a provider after it obtains the file (in general, a client can become a provider after obtaining just one block of the file). We adopted 3DES (112-bit key length) for classical cryptography, RSA (1024-bit key length) for public key cryptography, and MD5 for hashing algorithms.

The metrics we used to evaluate mSSL include:

- *Server capacity:* The number of client requests a server can process per time unit.
- *File downloading time:* The latency from the time a client initiates a connection with a server to the time that the client receives an entire file.
- *Storage overhead:* The space a client or a server needs in order to store mSSL-related information.
- *Control traffic volume:* The volume of control traffic when mSSL is in place.

Our evaluation of mSSL will focus on its indirect access mode when a client needs to download data objects from provider clients. (When mSSL is in direct mode, it will essentially incur the same cost as SSL.) We use experiments to collect results and perform analysis for server capacity and file downloading time, and analyze the storage overhead and control traffic volume through proofs and calculations.
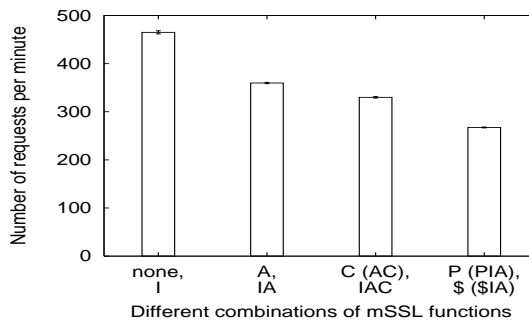

Fig. 6 Server capacity with different combinations of mSSL functions (95% confidence interval).

10.2 Server Capacity

To measure a server's capacity in handling client requests, we connected a client machine (a machine running Linux 2.4.20-20.9smp with a 2.4 GHz Pentium IV processor with Hyper-Threading and 1 GB of memory) and a server machine (an iBook running MacOS X with a 700 MHz processor and 384 MB memory) through a 100 Mbps link. Note that the server machine can simply be a low-end machine as most data sharing can be conducted between clients. Then for every one of the eight combinations to study, we had the client flood the server with file downloading requests, made sure the server reached its full capacity, and recorded how many such requests the server was able to serve over an extended period.

As shown in Figure 6, we can see that introducing new security functions does impact server capacity; however, such impact is acceptable. For example, the server capacity under the *none* combination averages 465 requests per minute. Adding data integrity does not affect the server capacity since no extra data-integrity-related operations are needed at the server. If requiring client authentication, the server capacity will become approximately 23% less. Server capacity decreases another 6% if also requiring confidentiality. Finally, if proof of service or atomic purchase is required, server capacity will decrease another 13%.

10.3 File Downloading Time

We measured mSSL's impact on downloading files as following. The server is the same machine as in Section 10.2, and every client is a Dell Latitude D810 machine running Linux 2.6.9-ck3 with a 1.73 GHz Pentium M processor and 512 MB memory, all connected over a 100 Mbps Ethernet. *We have found that increasing the number of providers for a recipient will linearly increase the downloading speed.* In the following, we report the
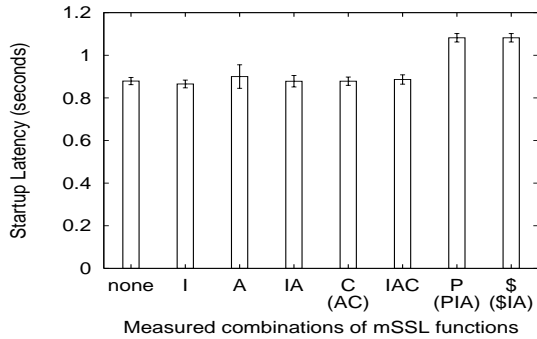
Fig. 7 File downloading startup latency (95% confidence interval).

Fig. 8 data transfer time under different combinations of mSSL functions.

measurement results for the case with one provider, and focus on the impact of all the different combinations of mSSL functions.

There are two components of a file downloading time: a *startup latency*, which is the time that a client spends in handshaking with the server before sending out a request for the first block of a file, and *data transfer time*, which is the rest of file downloading time. We report the results for each. (Note that although the proof of service function and the atomic data purchase function have a different design, when they have the same level of parallelism, i.e., the same number of blocks can be downloaded and handled in parallel, they will incur the same downloading time.)

The startup latency is not related to file size, as shown in our measurements, but different combinations of mSSL functions will have different startup latency. Figure 7 shows that compared to the *none* combination, further requiring data integrity, client authentication, data confidentiality, or their combinations, will not cause a measurable increase of the startup latency, but requiring proof of service or atomic purchase will lengthen the startup latency more significantly by about 23%.

The data transfer time increases when the size of a file increases and also depends on which combination of mSSL function is used. More specifically, our measurement shows that data transfer time is proportional to the size of a file, as shown in Figure 8. The figure also shows the difference between data transfer times in different combinations, indicating different mSSL functions have different impacts on downloading speed. The impact from client authentication and/or confidentiality is almost negligible; except for a few extra operations, transferring encrypted data or non-encrypted data does not differ in terms of transferring time, and data encryption and decryption operations can be done off-line. There is a slight decrease of downloading speed when adding data integrity verification, because a recip-
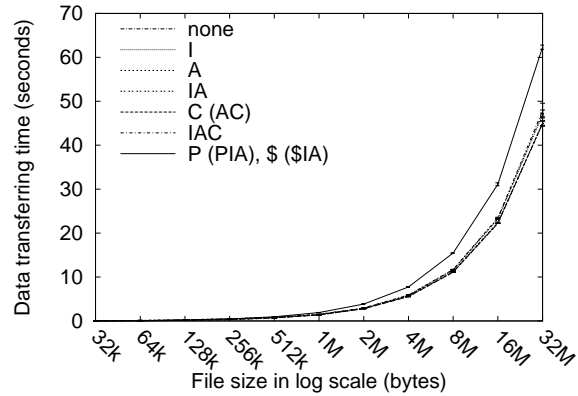
ient needs to request integrity path information (Section 3). The proof of service function and the atomic data purchase function have a more significant impact; e.g., with such a function turned on, downloading a 32 MB file will require 62s compared to 45s in the *none* combination.

10.4 Storage Overhead

While the *none* combination will incur the same storage overhead as in conventional SSL, adding more security and incentive functions to mSSL leads to extra storage overhead. Assuming the server is S and a recipient r is obtaining a data object O from a provider p, the extra storage overhead related to each function is:

- Integrity: Both p and r need to store the Merkle hash tree of O. If O is $1GB$, each block is $8KB$, and every hash value is 16 bytes, the tree will need approximately 4 MB of space—if it stores all hash values. Below we further show that the storage space needed could be as small as 288 bytes (as demonstrated by Theorem 7).
- Client authentication: r needs to store a ticket for O. In our implementation, a ticket is approximately 160 bytes.
- Confidentiality: p needs to store an encrypted copy of O (unless p encrypts O on the fly) and S needs to store the decryption key (either an object key for the object-key-based confidentiality approach, or a group key for the group-key-based approach).
- Proof of service: p and r need to store certificates of their public keys, and p also needs to store acknowledgments from r. Due to the aggregation feature of the acknowledgment mechanism, only a small number of acknowledgments are needed and each acknowl-
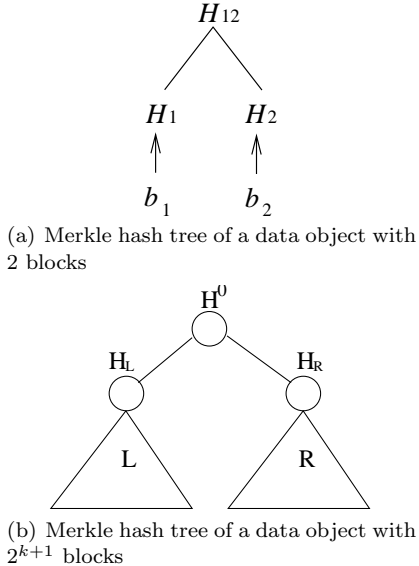
(a) Merkle hash tree of a data object with 2 blocks

(b) Merkle hash tree of a data object with 2^{k+1} blocks

Fig. 9 Merkle hash trees in the proof of Theorem 7.

edgment in our implementation is typically $150-200$ bytes.

- Atomic data purchase: Same storage overhead as the proof of service function, except that p and r also need to store the state information for running the PPay protocol [6].

Now we look more closely on the storage space needed for the data integrity function. While for a data object with 2^m blocks a full Merkle hash tree will have $2^{m+1} - 1$ hash values, in fact mSSL's data integrity function needs much less space, as described in the following theorem:

Theorem 7 *If a recipient sequentially receives blocks from a data object O with 2^m blocks, it only needs to keep at most $S(m)=m + 1$ hash values on O's Merkle hash tree.*

Proof We first show when $m = 1$,

$$S(m) = m + 1 = 2. \tag{7}$$

Refer to Figure 9(a) that shows the Merkle hash tree of a data object with two blocks b_1 and b_2. To verify b_1, the recipient needs to know H_{12} and H_2. After verifying b_1, the recipient will not need H_{12} and H_1 since verifying b_2 only needs H_2 according to Theorem 4. At any moment at most two hash values are necessary, i.e., $S(1) = 2$.

Now, assume when $m = k$, $S(m) = k + 1$, all we need to prove now is that when $m = k + 1$, we have $S(m) = k + 2$.

Refer to Figure 9(b), where the Merkle hash tree of the data object is rooted at H^0. It has a subtree $M(L)$ on the left rooted at H_L, and another $M(R)$ on

the right rooted at H_R. If a recipient is verifying blocks from the first half of the data object, it will need to store hash values in $M(L)$, plus H_R. Note that the number of hash values in $M(L)$ is the same as that in a Merkle hash tree for a data object with 2^k blocks. Therefore, $S(k + 1) = k + 2$. On the other hand, if the recipient moves to verifying blocks from the second half of the data object, all hash values it needs will be in $M(R)$, which also has the same number of hashes as that in a Merkle hash tree for a data object with 2^k blocks. Here, $S(k + 1) = k + 1$. Overall, at most $k + 2$ hashes need to be stored, i.e., $S(m) = k + 2$.

10.5 Volume of Control Traffic

Except for the *none* combination that will incur the same volume of control traffic as in conventional SSL, adding security and incentive functions lead to extra control traffic. Again, assuming the server is S and a recipient r is obtaining a data object O from a provider p, extra traffic related to each of the mSSL function is:

- Integrity: assuming O has n blocks, the extra traffic will be n hash values from p to r (see Theorem 8 below) and a small amount of request traffic from r to p.
- Client authentication: the delivery of a ticket from S to r and from r to p.
- Confidentiality: S needs forward the decryption key to r.
- Proof of service: p and r need to send each other a certificate of their own public keys. r also needs to send an acknowledgment for each encrypted block of O. p also needs to send a protected block key to r (see Section 6). p may also contact S to present the proof of its service, which will be the size of an acknowledgment (recall it is typically 150-200 bytes in our implementation). r may also contact S with a small amount of traffic when r has trouble with block keys.
- Atomic data purchase: Except incurring the same amount of traffic volume as the proof of service function, r needs to deliver payment to p, and running PPay protocol for handling payments also incurs extra traffic (an insignificant amount as reported in [6]).

Theorem 8 *Define $M(O)$ as the Merkle hash tree of object O. For a recipient r to verify the integrity of all n blocks of O with mSSL's integrity solution, if every block is received correctly, the total number of hash values on $M(O)$ that r needs to download (including the root of $M(O)$) is n.*
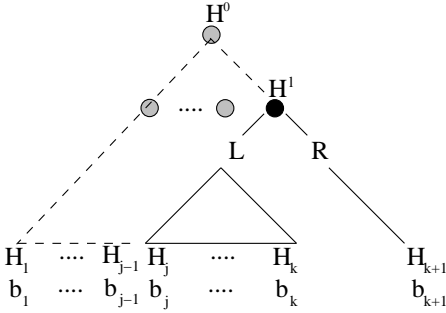
Fig. 10 Merkle hash tree of an object with $k + 1$ blocks. (Note that $j = 1$ if $k = 2^m$ (m is an integer), and $j = k$ if H_k and H_{k+1} are siblings.)

Proof We use $\gamma(n)$ to denote the total number of hash values to download for a data object with n blocks. Thus what we need to prove is that $\gamma(n) = n$. In the following, we use induction over n.

We first prove that $\gamma(1)=1$. In fact, when $n = 1$, O has just one block, say b_1. To verify b_1's integrity, r only needs to obtain the (signed) hash value of b_1. Clearly,

$$\gamma(1) = 1. \tag{8}$$

Now, let us assume:

$$\gamma(k) = k \quad (k \geq 1). \tag{9}$$

We then need to prove that

$$\gamma(k + 1) = k + 1. \tag{10}$$

If $n = k + 1$, O will have $k + 1$ blocks, and $M(O)$ will have $k + 1$ leaf nodes. Use H_k and H_{k+1} to denote hash values of b_k and b_{k+1}, respectively. And we further introduce the following notations, as also shown in Figure 10:

- H^l: The closest common ancestor of H_{k+1} and H_k on $M(O)$. A hash value at level l. (Note that when H_{k+1} and H_k are siblings, H^l is their parent.)
- L: The left child of H^l. The ancestor of H_k or H_k itself.
- R: The right child of H^l. The ancestor of H_{k+1} or H_{k+1} itself. Note while L and R are siblings when O has $(k + 1)$ blocks, L has no sibling when O has k blocks.
- $M(H^l)$, $M(L)$, $M(R)$: The subtrees of $M(O)$ that are rooted at H^l, L, R, respectively. Note that every non-leaf node on $M(R)$ has *only* one child.
- $H_j, H_{j+1}, ..., H_k$: Leaf nodes of $M(L)$.
- $b_j, b_{j+1}, ..., b_k$: Blocks whose hash values are leaf nodes of $M(L)$, i.e., $H_j, H_{j+1}, ..., H_k$, respectively.
- b_x: The first block that r receives among $b_j, b_{j+1}, ..., b_k$.

Assume r receives b_x (it has *not* received b_{k+1} yet) and needs to verify its integrity. Compared to the hash values that r would need for b_x's mSSL integrity path when $n = k$, now r will need one more hash value: R. For all the rest of the blocks of O except b_x and b_{k+1}, each of them will request the same number of hash values as that when $n = k$. No hash value is needed to download for b_{k+1}'s mSSL integrity path. Therefore, $\gamma(k+1) = \gamma(k)+1$. According to Equation 9, $\gamma(k+1) = k + 1$.

If r receives block b_{k+1} prior to receiving any blocks from $b_j, b_{j+1}, ..., b_k$, with a procedure similar to the above, we can also show $\gamma(k + 1) = k + 1$.

Combining Equations 8, 9, and 10, we finally prove Theorem 8.

11 Related Work

As a framework with a set of security and incentive functions, mSSL allows hybrid peer-to-peer applications to benefit from motivated and secure data sharing between peer clients. One could argue that some security protocols (such as SSL or Kerberos) can be used to support mSSL functions. There have also been data integrity mechanisms, including those leveraging Merkle hash tree. Obtaining proofs or digital payment during data sharing is also related to and can benefit from research on fair exchange of data. In this section, we go through these related work and compare them with mSSL.

We first compare mSSL and SSL [5]. SSL (or SSL/TLS) is the most common security scheme today for securing web-based services and has also been used for many other services. While SSL mainly provides data protection between a client and a server, it cannot address well the security challenges arising from data sharing among clients; applying SSL separately to every client-server and client-client connection, for example, will incur a high overhead and face deployment difficulties. SSL does not provide incentives for clients to share data, either. In contrast, mSSL not only secures data sharing between peer clients, but also provides incentive mechanisms for them to share. On the other hand, while a SSL channel enables two-way, interactive data transfer between a client and a server, mSSL is designed for peer clients to share data from their server. Therefore, mSSL and SSL are complementary to each other. A hybrid P2P application can have its clients first use SSL to securely interact with their server, and then use mSSL to securely obtain data through motivated peer clients.

Related to mSSL's data integrity design, existing peer-to-peer file-sharing applications also support certain data integrity functions. PROOFS [4] and Slurpie

[2] recommend the use of MD5 or similar checksum algorithms to protect sensitive data. BitTorrent [12] adopts a superblock-based mechanism (see our comments on this mechanism in Section 3). However, more related to mSSL are solutions leveraging Merkle hash tree, such as those used in peer-to-peer media streaming [13], third-party distribution of integrity-critical databases [14] and XML documents [15], and updating/downloading the code blocks for SmartCards [9]. The work in [9] achieved the same magnitude as mSSL for shortening the transmission, storage, and verification overhead, but it differs from mSSL in key aspects: (1) It relies on the *sender* to determine what hash values to send so that the receiver can verify a block's integrity. This method will not scale when there are a large number of receivers per sender. On the contrary, mSSL relies on the receiver to decide, which is more scalable. (2) A sequential downloading process is described in [9], while mSSL allows a recipient to receive data blocks out of order from multiple senders. (3) Compared to [9], which mainly describes the procedure for handling integrity, our paper provided formal proofs of eight theorems.

Kerberos protocol [16] is also related to mSSL, mainly the client authentication function of mSSL. It allows a client to contact a trusted third party—a Key Distribution Center—to obtain a ticket-granting ticket (TGT), and then use the TGT to obtain a ticket related to a particular service. mSSL's authentication mechanism is also ticket-based, but it does not rely on a trusted third party. It employs a simplified design by allowing the server to issue a ticket directly to a client.

Both the proof of service function and the atomic data purchase function in mSSL are related to strong fair exchange of information or goods, where at the end of an exchange, either each party obtains what it expects or neither of them does. Fair exchange has been studied in the context of electronic payments [17,18], certified e-mail systems [19,20], contract signing protocols [21,22,23], and non-repudiation protocols [24,25]. In the context of this paper, fairness would mean for a provider to receive a proof or digital payment of its service and for a recipient to receive the requested data. Solutions with a trusted third party (TTP) can be created using an *inline* TTP (such as [26,20] where the TTP is required to mediate every communication between a sender and a receiver), using an *online* TTP (such as [27,28] where the sender and the receiver can directly communicate, but still need the TTP to store and fetch information), and using an *offline* TTP (such as [29,30], where the TTP will be involved only when a problem occurs). The most closely related to mSSL is fair exchange with an offline TTP. While leveraging current schemes, our solution has an important differ-

ence in that a server itself can act as a TTP for its own provider and recipient clients—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. This brings us another advantage (especially when the amount of data is large): If a server needs to verify data, it does not have to request them from other sources, avoiding a drawback in many TTP-based solutions.

12 Conclusions

By enhancing the conventional client-server communications with peer-to-peer data sharing among clients, hybrid peer-to-peer applications significantly improve the scalability in handling a large number of clients. However, these applications also face severe challenges: Receiving data from arbitrary, often less trustworthy peer clients is subject to much higher security risks than receiving data directly from a server; mechanisms to reward peer clients for sharing data, such as crediting or paying those providing data to others, are also vulnerable since clients may lie about peer-level service and even collude.

In this paper, we designed and evaluated the extensible mSSL framework that currently includes five security and incentive functions that applications can invoke based on their need. Data sharing between peer clients via mSSL is not only as secure as that between client and server via SSL, but is also motivated through accurate and authentic proofs of service or digital money.

The low cost, high performance, and scalability of mSSL functions further makes mSSL an attractive solution. The extra storage overhead from mSSL and its control traffic volume are low, and mSSL's impact on server capacity and file downloading time are both acceptable. This result is even more encouraging if considering that with mSSL every server can enlist a large pool of possibly selfish but motivated clients to help distribute its data securely, allowing even an under-provisioned site to distribute data to many with ease.

References

1. BitTorrent Inc., "BitTorrent," `http://bittorrent.com`, 2005.
2. Rob Sherwood, Ryan Braud, and Bobby Bhattacharjee, "Slurpie: A Cooperative Bulk Data Transfer Protocol," in *IEEE INFOCOM*, 2004.
3. Keith Kong and Dipak Ghosal, "Mitigating Server-Side Congestion in the Internet through Pseudoserving," *IEEE/ACM Trans. Netw.*, vol. 7, no. 4, pp. 530–544, 1999.
4. Angelos Stavrou, Dan Rubenstein, and Sambit Sahu, "A lightweight, robust P2P system to handle flash crowds," in

Proceedings of ICNP, Washington, DC, USA, 2002, pp. 226–235, IEEE Computer Society.

5. Eric Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, Boston, MA, USA, 2001.

6. Beverly Yang and Hector Garcia-Molina, "PPay: micropayments for peer-to-peer systems," in *Proceedings of the Conference on Computer and Communications Security*, New York, NY, 2003, pp. 300–310, ACM Press.

7. Ralph Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Privacy and Security*, April 1980, pp. 122–134.

8. Chung Kei Wong and Simon S. Lam, "Digital signatures for flows and multicasts," *IEEE/ACM Transactions on Networking*, vol. 7, no. 4, pp. 502–513, 1999.

9. Luke O'Connor and Gunter Karjoth, "Efficient downloading and updating applications on portable devices using authentication trees," in *IFIP TC8/WG8.8 4th Working Conference on Smart Card Research and Advanced Applications*, Norwell, MA, September 2002, Kluwer Academic Publishers.

10. Yang Richard Yang, X. Steve Li, X. Brian Zhang, and Simon S. Lam, "Reliable Group Rekeying: A Performance Analysis," in *Proceedings of ACM SIGCOMM*, San Diego, California, 2001, pp. 27–38, ACM Press.

11. Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow, "IETF RFC 2018: TCP Selective Acknowledgement Options," 1996.

12. Bram Cohen, "Incentives build robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer Systems*, 2003.

13. Ahsan Habib, Dongyan Xu, Mikhail Atallah, Bharat Bhargava, and John Chuang, "Verifying Data Integrity in Peer-to-Peer Media Streaming," in *Twelfth Annual Multimedia Computing and Networking (MMCN '05)*, 2005.

14. Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine, "Authentic third-party data publication," in *Proceedings of the IFIP TC11/ WG11.3 14th Annual Working Conference on Database Security*, Deventer, The Netherlands, 2001, pp. 101–112, Kluwer, B.V.

15. Elisa Bertino, Barbara Carminati, Elena Ferrari, Bhavani M. Thuraisingham, and Amar Gupta, "Selective and Authentic Third-Party Distribution of XML Documents.," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 10, pp. 1263–1278, 2004.

16. B. Clifford Neuman and Theodore Ts'o, "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications*, vol. 32, no. 9, pp. 33–38, September 1994.

17. Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich, "Endorsed e-cash," in *Proceedings of the IEEE Symposium on Security and Privacy*. May 2007, pp. 101–115, IEEE Computer Society.

18. Benjamin Cox, J. D. Tygar, and Marvin Sirbu, "NetBill Security and Transaction Protocol," in *The First USENIX Workshop on Electronic Commerce*, July 1995, pp. 77–88.

19. Jianying Zhou and Dieter Gollmann, "Evidence and non-repudiation," *J. Netw. Comput. Appl.*, vol. 20, no. 3, pp. 267–281, 1997.

20. A. Bahreman and J.D. Tygar, "Certified Electronic Mail," in *Proc. of Symposium on Network and Distributed Systems Security*, San Diego, 1994, pp. 3–19, Internet Society.

21. Guilin Wang, "An Abuse-Free Fair Contract Signing Protocol Based on the RSA Signature," in *WWW 2005*, New York, NY, USA, May 2005, pp. 412–421, ACM Press.

22. G. Ateniese, "Efficient Verifiable Encryption (and Fair Exchange) of Digital Signature," in *Proceedings of the Conference on Computer and Communications Security*, New York, NY, 1999, pp. 138–146, ACM Press.

23. M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest, "A Fair Protocol for Signing Contracts," *IEEE Transactions on Information Theory*, vol. 36, no. 1, pp. 40–46, 1990.

24. Steve Kremer, Olivier Markowitch, and Jianying Zhou, "An Intensive Survey of Fair Non-Repudiation Protocols," *Computer Communications*, vol. 25, no. 17, 2002.

25. Panagiotis Louridas, "Some Guidelines for Non-Repudiation Protocols," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 5, pp. 29–38, 2000.

26. Tom Coffey and Puneet Saidha, "Non-Repudiation with Mandatory Proof of Receipt," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 1, pp. 6–17, 1996.

27. N. Zhang and Q. Shi, "Achieving Non-Repudiation of Receipt," *The Computer Journal*, vol. 39, no. 10, pp. 844–853, 1996.

28. Jianying Zhou and Dieter Gollmann, "A Fair Non-Repudiation Protocol," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996, pp. 55–61, IEEE Computer Society Press.

29. N. Asokan, V. Shoup, and M. Waidner, "Asynchronous Protocols for Optimistic Fair Exchange," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1998, pp. 86–99.

30. Steve Kremer and Olivier Markowitch, "Optimistic Non-Repudiable Information Exchange," in *Proceedings of the 21st Symposium on Information Theory in the Benelux*, Wassenaar, The Netherlands, 2000, pp. 139–146.