Data Integrity and Proof of Service in BitTorrent-Like P2P Environments

by Jun Li

The success behind BitTorrent [1] or BitTorrent-like Peer-to-Peer (P2P) applications (such as those discussed in Stavrou, Rubenstein and Sahu [2]; Kong and Ghosal [3]; and Sherwood, Braud, and Bhattacharjee [4]) is their innovative use of collaboration among peer clients. While the scalability of client-server applications is often poor when a large number of clients access a single server, permitting a client to download data from other peer clients in addition to directly from its server has tremendously boosted the availability of data to a client and fundamentally addressed data-accessing scalability.

However, compared to receiving data directly from a server, receiving data from arbitrary, often less trustworthy peer clients is subject to much higher security risks. Two obvious concerns that warrant new study are data integrity and proof of service in this BitTorrent-like P2P environment. The integrity of data received from peer clients is more easily breached if malicious clients sneak into the system to corrupt peer communications. Mechanisms to reward peer clients for sharing data, such as crediting the providers, are also vulnerable, since clients may lie about peer-level service.

Conventional approaches designed for the client-server communication paradigm, such as Secure Sockets Layer (SSL) [5], cannot easily address these concerns. With thousands of clients sharing a single server and communicating with each other, simply setting up an SSL security channel between every client and the server and between every pair of peer clients is not only costly but also requires a great deal of extra work to make an SSL function in this environment.

In this article, we describe a feasible approach to addressing the data integrity and proof of service in a BitTorrent-like P2P environment. Both functionalities are included in a new protocol we designed that is called mSSL.

Data Integrity

Now that a client can receive its server's data from either the server or from its peer clients, the client must be able to verify the integrity of the data to ensure that the data has indeed originated from the server and has never been manipulated.

The mSSL protocol supports a block-based, on-demand data integrity solution. With a block-based solution that divides a data object into many blocks and verifies the data integrity at block level, a client can verify the integrity of every block once it is received. If a client has to verify the signature of an entire data object to verify its integrity, the penalty can be high; in particular, if signature verification fails, the entire data object—which can be a file with many gigabytes of data—has to be retransmitted. The integrity solution is also on demand, in that a client can determine the *integrity path* of a given block and then request the integrity path information to verify the integrity of the block. We explain the integrity path concept below.

To verify data integrity at block level, one solution is to bind a data signature to every block of a data object. A client must verify the block signature to determine its integrity. This method, however, involves encryption and decryption at block level and can lead to high computational overhead. Another approach is to build a superblock for a data object that contains a strong one-way hash result for every block. A client can first obtain the superblock (the superblock itself can carry a signature to prove its own integrity), then, whenever it receives a new block, it can calculate the hash of the block and compare it with the value contained in the superblock. Calculating the hash result is a much faster process than encryption operations, but, for large files, a superblock itself can be very large. In a 100 gigabyte video file, for example, if every block is 1 kilobyte, and every hash is 16 bytes, a superblock of 1.6 gigabytes has to be retrieved first, leading to a high startup latency in retrieving data blocks.

In fact, every data object can have a binary Merkle hash tree [7], and every block of the data object can have an authentication path obtained from that tree. (Merkle hash trees have been used in various contexts such as P2P media streaming [8] or in third-party distribution of integrity-critical databases [9] and XML documents. [10]) When a client receives a block, it can request the authentication path of that block to verify its integrity. The client does not have to download all hash values

beforehand as in the superblock-based approach. Figure 1 shows an example.

The problem with this approach, which is block based and on demand, as is mSSL, is its high traffic volume in receiving authentication paths. For example, if every block is 1kilobyte and every hash is 16 bytes, a 1 gigabyte data object can incur 320 megabytes of such traffic. (The object has 2^{20} blocks, and every block's authentication path consists of 20 hash values; *i.e.* 320 bytes).

The mSSL protocol greatly optimizes both on-demand requests of integrity verification information and the verification procedure itself. Unlike the authentication path above, in mSSL, every block is associated with an integrity path. An integrity path is only composed of those hash values that are not locally available for calculating the root of the Merkle hash tree of a data object. The question then is, *What hash values are locally available or unavailable?*

Denote a block *b*'s authentication path *A(b)* as $A(b) = H^m, H^{m-1}, ..., H^1$ (H^i is a hash value at level *i* of the Merkle hash tree and H^m is at the leaf level), and use *mip(b)* to denote *b*'s integrity path at a client. Our research has shown that, if the client already has H^{l-1}, but not yet H^l, then $mip(b) = <H^m, H^{m-1}, \cdots, H^l>$, with a total of $|mip(b)| = m - l + 1$ hash values. Also interestingly, a client only needs to specify the value of $|mip(b)|$ for its peer client or the server to determine what the integrity path is at the requesting client. With this approach, we have also proven that for a client to verify the integrity of all *n* blocks of a data object, the total number of hash values that the client needs to request is also *n*, if every block is received correctly. In the above example, in which every block is 1 kilobyte and every hash is 16 bytes, a 1–gigabyte data object will incur 16 megabytes of traffic overhead, only 1/20th of the authentication-path-based solution.

Once a client receives the integrity path of a block, it can also verify its integrity at a faster speed than the authentication-path-based approach. Instead of using the integrity path to determine the authentication path of the block and then calculating the root of the tree using the authentication path, mSSL will only calculate the hash value of an intermediary node on the tree, and compare it with that node's correct value, which has already been obtained earlier. More specifically, if a block *b*'s integrity path is $mip(b) = <H^m, H^{m-1}, \cdots H^l>$, a client only needs to calculate the sibling node of H^l's parent.

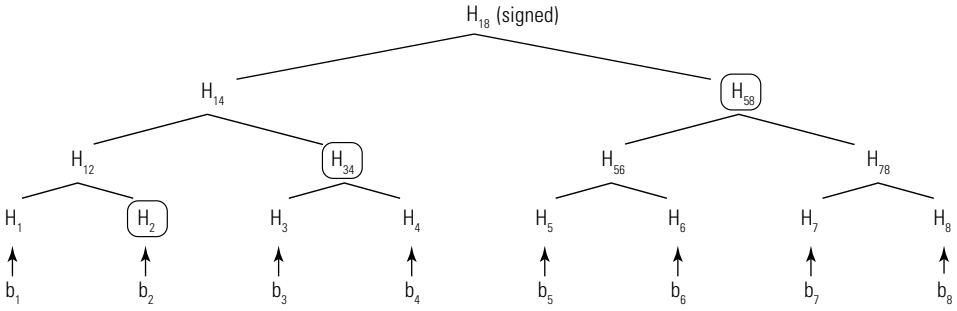

Figure 1 Merkle hash tree of a data object with eight blocks. Every leaf node is a hash value produced from a strong one-way hash function with the data of a corresponding block as input. Every intermediary node is a hash value that is calculated using the hash function with the values of its child nodes as input. Every block has an authentication path that can help calculate the root of the tree when knowing only the block itself. For example, the authentication path of block b_1 is $A(b_1) = < H2, H34, H58 >$, where b_1 can lead to H_1, H_1 and H_2 $[H_2 \in A(b_1)]$ can lead to H_{12}, *etc.* If a block is corrupted, it can be detected because the root calculated from the block will not be equal to the authentic value of the root of the tree.

Proof of Service

If a client can present to its server an accurate, verifiable, and non-repudiatable proof to describe its service by providing data to some of its peer clients, the server could offer this client certain credit; *e.g.*, assigning a higher priority in providing this client data or charging it a lower price for receiving the server's files. More importantly, the client will also have stronger incentive to continue to provide data to its peers.

A simple approach to obtaining proof of service is to enforce an interlocking block-by-block verification mechanism between every pair of provider and recipient clients. As shown in Figure 2, the recipient must acknowledge its receipt of the current block before it can receive the next block from the provider. The provider can then use the acknowledgment as proof of serving the block being acknowledged.

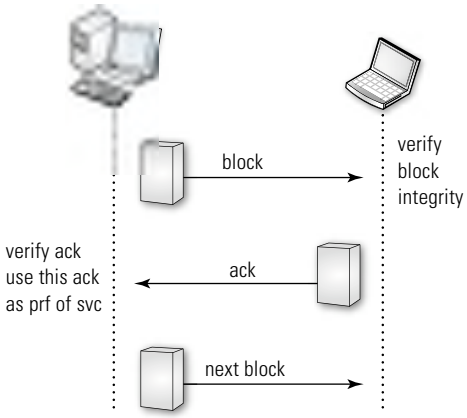

Figure 2 A basic proof of service solution

Unfortunately, this approach faces several serious problems. First, if a provider has to obtain a separate proof for *every* block it offered, it will face the proof-explosion issue when serving large files to its peer clients. Second, if a recipient decides *not* to acknowledge the receipt of a block after receiving it, the provider will not be able to obtain the proof of serving that block. Third, if the acknowledgment from a recipient is corrupted, the provider can be cheated in providing the next block.

The mSSL protocol supports a proof-of-service solution that addresses all these problems in a BitTorrent-like environment. This work is related to the strong, fair exchange of information in other contexts, and its design can be summarized as follows—

- A recipient will receive an encrypted block first and receive the decrypting block key after acknowledging the receipt of the block
- Acknowledgments are cumulative; thus the most recent one can replace previous ones as the proof of service. For example, it can be [1–66,68–128] to acknowledge the receipt of the first 128 blocks of a data object except block 67
- Every acknowledgment is signed and can be verified using the public key of the recipient.

Figure 3 shows the detailed steps for a provider *p* to obtain a proof, while providing data blocks to a recipient *r*. More specifically, it contains the following major steps—

1. **Public Key Exchange**—*p* and *r* will first exchange public key certificates so that they can know each other's public key. In mSSL, when a client (such as *p* or *r*) first contacts its server, it will still set up an SSL channel with the server and obtain both the server's public key certificate and, at the same time, the server's public key. Through this SSL channel, the client can also request the server to sign the client's public key, thus generating a public key certificate signed by the server. This kind of certificate is exactly what *p* and *r* exchange, and each can verify the certificate from the other side.
2. **Encrypted Block Request and Transmission**—*r* then requests a block *b* from *p*, which sends an encrypted block back to *r*. Here, the key used for block encryption, also called **block key**, is generated by *p* using a strong one-way hash function, *f*: $k = f(p, r, file, blockid, k_p)$, in which *p, r, file, blockid* are the identifiers of *p, r,* the data object that the block belongs to, and the block itself. k_p is the secret key shared between the provider and the server. Note: The server can also apply this formula to calculate the block key.
3. **Cumulative Acknowledgment**—*r* then sends back an acknowledgment and its signature. Instead of acknowledging only the receipt of the encrypted block, *r* uses a *sack* field to acknowledge the receipt of all blocks it has received from *p*. It also includes the digest of the encrypted block. *r* signs the acknowledgment with its private key, PRV_r. On the receipt of the acknowledgment, *p* then can verify it. Only if the verification is successful will *p* use the acknowledgment as its proof of service to *r* so far—and begin to deliver the block key to *r*.
4. **Block Key Delivery**—*p* delivers the block key to *r,* which is encrypted with *r*'s public key, PUB_r, and signed with *p*'s private key, PRV_p. The delivery must be secure so that only *r* can obtain the key and *r* can verify it is from *p*. After *r* receives the block key, it then can decrypt the block and verify the integrity of the block. If *p* refuses or fails to deliver the block key, *r* can request its server to calculate the block key and deliver the key to *r*. Note: This operation implies that *r* probably will stop using *p* as its data provider and is therefore an infrequent operation that will not overload the server. If the block integrity is found to be corrupted, *r* will not acknowledge the receipt of the next block. By doing so, when *p* presents the current acknowledgment to the server as proof of service, the server can calculate the digest of the current block in its encrypted form to detect that *r* did not correctly receive the current block. This design, however, leads to a slow, stop-and-go data-transmission process, which we revisit below.

The above process handles one block at a time. To improve performance, mSSL further introduces parallelism to concurrently handle multiple blocks. Every acknowledgment can actually include digests of the last *m* encrypted blocks (*m* = 1 in the above step-by-step description). When an acknowledgment is used as a proof of service and presented to a server, the server will verify the correctness of all *m* digests to ensure that the last *m* encrypted blocks are all delivered correctly. This way, instead of first using a block key to decrypt an encrypted block and verify its integrity before acknowledging data reception, a recipient can always first acknowledge the receipt of up to *m*-1 encrypted blocks.

Two types of attacks may occur against the proof-of-service design:

individual cheating, during which an individual client misreports the amount of service it received or provided, and colluded cheating, during which multiple clients forge proofs of service together. mSSL addresses both attacks.

For individual cheating, a provider cannot overstate its service, since every proof is provided and signed by its peer clients after they receive data from the provider; and a recipient cannot deny the service it receives since, in doing so, the recipient will not be able to receive block keys to decrypt the encrypted data blocks it receives.

Colluded cheating can happen in various ways, especially in the following two scenarios—

- In one scenario, a recipient client sends acknowledgments to its real provider and then may try to copy these acknowledgments to its accomplices, who, in turn, may try to use these acknowledgments to claim that they each also have delivered data to the recipient. This colluding scenario can be detected by the server of these clients when verifying the digest(s) of the acknowledgment, which must be the digest(s) of the block(s) encrypted by the real provider using a block key (or block keys) specific to that provider. This mechanism can also help detect proofs that a client forges by simply providing data to itself.
- In another scenario, even if no data transmission *ever* occurred, clients can collude to forge a proof that one or several of them have provided data to a recipient client. Here, an economical countermeasure may be the most effective: Because the server needs only to make sure that the undeserved credits that fake providers may obtain are always less than the cost that the colluding recipient has to pay, these colluders will lack incentives to proceed.

Summary

While BitTorrent or BitTorrent-like P2P applications have been very successful in sharing among peer clients data that are traditionally only available by directly downloading from a server, security concerns in these applications are severe, especially ensuring data integrity and obtaining trustworthy proof of service that a client has provided to its peers.

The mSSL protocol offers a security solution in this environment, and in this article we described its approach to data integrity and proof of service. In ensuring data integrity, mSSL introduces an integrity path concept to support a prompt, block-based, and on-demand integrity verification mechanism that permits both low traffic and low computational overhead. The proof-of-service design is also advantageous in BitTorrentlike P2P environments. It not only minimizes server overhead and data-transmission slowdown but also ensures that the proofs are accurate and of small size. ■

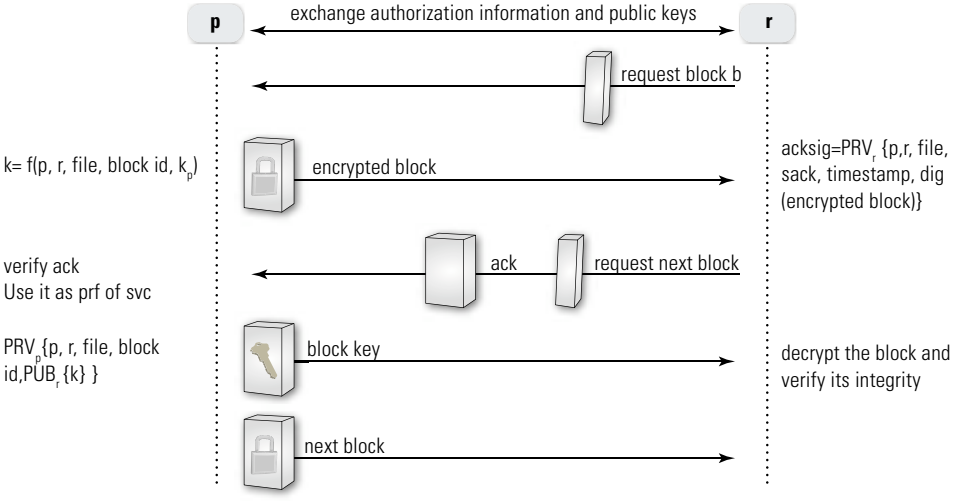

Figure 3 mSSL's proof-of-service design

References

1. BitTorrent, *http://bittorrent.com, 2005.*
2. Stavrou, Angelos; Rubenstein, Dan & Sahu, Sambit. A Lightweight, Robust P2P System to Handle Flash Crowds. *10th* ICNP, 2002 (pp. 226–235).
3. Kong, Keith & Ghosal, Dipak. (1999) Mitigating Server-Side Congestion in the Internet through Pseudoserving. *IEEE/ACM Trans. Netw., vol. 7, no. 4* (pp. 530–544).
4. Sherwood, Rob; Braud, Ryan & Bhattacharjee, Bobby. (2004). Slurpie: A Cooperative Bulk Data Transfer Protocol. *IEEE INFOCOM*
5. Rescorla, Eric. *SSL and TLS: Designing and Building Secure Systems. Addison-Wesley*, 2000.
6. Li, Jun & Kang, Xun (December 2005). mSSL: Extending SSL to support data sharing among collaborative clients. *Annual Computer Security Applications Conference, Tucson, Arizona* (pp. 357–368).
7. Merkle, Ralph. (April 1980). Protocols for public key cryptosystems. *IEEE Symposium on Privacy and Security* (pp. 122–134).
8. Habib, Ahsan; Xu, Dongyan; Atallah, Mikhail; Bargava, Bharat & Chuang, John. (2005) Verifying Data Integrity in Peer-to-Peer Media Streaming." *Twelfth Annual Multimedia Computing and Networking (MMCN '05).*
9. Devanbu, Premkumar T.; Gertz, Michael; Martel, Chip & Stubblebine, Stuart G. (2000) Authentic Third-party Data Publication. *IFIP Workshop on Database Security* (pp. 101–112).
10. Bertino, Elisa; Carminati, Barbara; Ferrari, Elena; Thuraisingham, Bhavani M. & Gupta, Amar (2004). Selective and Authentic Third-Party Distribution of XML Documents. *IEEE Trans. Knowl. Data Eng., vol. 16, no. 10.* (pp. 1263–1278).

About the Author

Jun Li | is an assistant professor at the University of Oregon. He received his PhD from the University of California, Los Angeles, in 2002. His current research includes Internet worm defense, Internet routing forensics, Internet Protocol (IP) source address validity enforcement, and P2P security in BitTorrent-like environments. Dr. Li is a member of the Institute of Electrical & Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). His research is being funded by National Science Foundation and Intel Corporation. He may be reached at lijun@cs.uoregon.edu.