Open-Ended BitTorrent: Efficient Peer to Peer Open File Stream Distribution

Bilal Karim Mughal
Isra University
Hyderabad, Sindh
bilalkmughal@gmail.com

Aisha Syed
University of Utah
Salt Lake City, UT 84112
aisha.syed@utah.edu

ABSTRACT

We present a scheme for distributing a file over a peer-to-peer network while supporting appending of additional content generated in the future. We do this by modifying the BitTorrent protocol. As content can always be updated and a hash of all content is not available, we use public key cryptography to maintain file integrity. This scheme can be used to distribute log structured storage or arbitrary pure functional data structures across a network of nodes, and also enables future updates to these structures. We describe how such a scheme would be useful for anything from peer-to-peer OS package updates to video streaming.

General Terms

Algorithms, Design.

Keywords

BitTorrent; log structured storage; append-only databases; pure functional data structures; peer-to-peer; p2p

1. INTRODUCTION

A number of technologies exist for peer-to-peer data distribution, BitTorrent [1, 2] being the most popular. BitTorrent is used with either centralized trackers or with distributed hash tables (DHTs) for peer discovery. The content is frozen when shared, with no further modification allowed after it has been published. What we propose is a variation of the BitTorrent model which allows append operations on the data after the initial publish event.

The specific use-case we would like to highlight: We can fully implement arbitrary pure functional data structures based on this primitive, with support for arbitrary updates via the append operation. This effectively extends to peer-to-peer distribution of an append-only database.

2. SUPPORTING APPEND OPERATIONS IN BITTORENT PROTOCOL

In a typical BitTorrent protocol [2] based file transfer session, everything starts with a torrent file (also called a metafile). When a DHT is in use, the file may be obtained from peers via a content addressable scheme. The torrent file includes a list of hashes mapping to blocks of the file being transferred. Individual blocks of the file can be requested from peers, and verified using the hashes.

In our scheme, the full file stream is unavailable and unknown at the start, and only an initial (possibly zero length) portion is available. Given this fact, we cannot use hashes, which also implies we cannot verify the content provided by any peers.

A simple solution is using public key crypto. We generate a public-private key pair for our stream. We modify the torrent file to include the public key instead of the list of hashes. The torrent remains content addressable, we just use the public key (or its hash) as the address instead of a hash of the content.

The BitTorrent peer protocol [2] is a fairly simple message based protocol. Nodes broadcast a *have* notification message when they receive new chunks of content. Nodes can request chunks of content from peers using a *request* message, and peers can respond to that with a *piece* message. These basic primitives enable replication of content amongst peers. There are additional message types defined for bandwidth management.

We define additional messages to support a modified variant of the protocol which supports the append operation on the stream.

The first message we define is a *length* message, which includes an unsigned integer, along with a signature (signed using the private key for verification). A node broadcasts this to its peers in its initial handshake once it connects to a peer initially. If a node receives a newer (longer) length, it broadcasts this to all peers with a smaller length. This simple operation causes new values of length to propagate efficiently throughout the network.

The *request* and *piece* messages work as they do for a normal torrent, but one missing feature is verification of the received pieces as we do not have a list of hashes available. To get around this, we define additional messages: *hash-*

request, *hash-pieces* and *hash-have*. We are essentially defining a second stream of data just for the list of hashes, with exactly the same semantics as the normal stream. The contents of this stream are a list of tuples, each tuple containing a hash of the associated block in the data stream, the length of the associated block (which is equal to the block size, except for the last block), and a signature derived via signing *concat(block_hash, block_size, block_number)* with the private key. We are assuming a signature algorithm with a fixed output size (a size based on the key size, and not on the input value). Given this, as all tuples are fixed sized, the format can support random access, and each tuple can be independently verified. No *length* message is necessary for the hash stream, as this can be trivially derived from the data stream.

The above described scheme and protocol allows all the properties of traditional torrents, but also allows for an increase in data size. Care needs to be taken around the last pieces in the data and hash streams, as they can operate on truncated data when the data size is not a multiple of the block size, but handling this is straightforward: after an updated length, the former last block's hash needs to be recalculated as the newly added pieces become available. An old hash tuple at the end is replaced by one with longer size.

One interesting aspect of our defined format is that both the data and hash streams support random access, i.e., obtaining and verifying content in one part of the streams is independent of other parts of the stream. If newer parts of the stream make older parts irrelevant, nodes are free to discard older data, and holding on to an ever-growing log of data is not required.

3. USE CASES

An obvious use case is log structured storage [6]. Such a storage system allows implementing a complete file system simply on the basis of the append operation. This can be upgraded into a complete versioning file system, where not only is the current state of the file tree accessible, but all previous states are also accessible. Nodes could potentially be made to discard older states if only the current version is required, to conserve storage.

Indeed, our scheme can support copy-on-write B-trees (as well as modern variations and equivalents), which are the structure underlying many of today's file systems and databases, including WAFL [3], ZFS [4], Btrfs [5] and more.

An example where this scheme could be useful is operating system package updates. Operating systems consist of a large number of separately versioned packages. Most package update systems currently in use are completely centralized, but with our scheme all new packages can be added as part of a versioning file system stream. Nodes are able to request the data pieces that they need from the peer swarm, and can discard older data as newer packages become available, conserving space.

A simpler example is live peer-to-peer video streaming. An append-only stream fully allows for live encoding of a video feed, which gets distributed across a network of peers. Nodes can either keep older parts of the stream (if they wish to keep a recording), or discard them while keeping just a small buffer of recent data.

4. CONCLUSION

BitTorrent combined with DHTs provided existence proof that large, fully distributed content distribution networks were practical. Our work builds on that to describe how with just a few simple additions, we can allow arbitrary updates to content over such a swarm of nodes. Our protocol can be implemented with fairly minor modifications to existing BitTorrent clients.

5. FUTURE WORK

The initial peer handshake in the BitTorrent protocol also involves peers notifying each other of the blocks they already have. This is done via sending a *bitfield* message, which contains a set of bits, one for each block, describing which blocks a peer already has. For a never-ending stream, this bit field will grow forever, even if the nodes have discarded older pieces. Allowing a separate stream for the bit field data would avoid this issue.

Our scheme depends on a public key cryptography for integrity. It is expected that the stream author will only generate a sequential stream. However, the signature scheme in theory allows the author to split the stream into multiple conflicting appended streams. This could be exploited to allow for branching streams, each describing a separate history of data, allowing structures like the Git version control system [7] to be directly implemented.

6. REFERENCES

[1] BitTorrent. http://www.bittorrent.com/

[2] Cohen, Bram. "The BitTorrent protocol specification." (2008): 28.

[3] Hitz, David, James Lau, Michael Malcolm, and Byron Rakitzis. "Write anywhere file-system layout." U.S. Patent 5,963,962, issued October 5, 1999.

[4] Bonwick, Jeff, and Bill Moore. "Zfs: The last word in file systems." (2007): 4-7.

[5] Rodeh, Ohad, Josef Bacik, and Chris Mason. "Btrfs: The linux b-tree filesystem." *ACM Transactions on Storage (TOS)* 9, no. 3 (2013): 9.

[6] Rosenblum, Mendel, and John K. Ousterhout. "The design and implementation of a log-structured file system." *ACM Transactions on Computer Systems (TOCS)* 10, no. 1 (1992): 26-52.

[7] Git. http://git-scm.com/