# Rateless Codes and Big Downloads - Implementation

Petar Maymounkov and Chris Sun

## 1 Introduction

This report details the implementation of the multi-source file download algorithm, described in "Rateless Codes and Big Downloads" by Maymounkov and Mazieres. The implementation consists of a single application which acts both as a server of the shared files on the local machine as well as a client for downloading files, requested by the local user.

The interface of the application to the user consists of two functions (accessed through a prompt or alternate user interface):

1. `share_file (str filename)`
2. `request_file (sha1 file_id)`

All operations on files are asynchronous and interleaved in a way that tries to achieve a reasonable level of fairness in terms of CPU, disk and network resources spent on any file.

When a file is declared as shared, the program (1) reads the entire file from disk; (2) splits it logically into blocks (for purposes of the coding algorithm); (3) computes hashes of all blocks, and the file's ID (hash of all blocks' hashes); (4) computes the auxiliary blocks, necessary for encoding the file using Online Codes; and finally (5) stores all hashes and auxiliary blocks in private (to the application) directory.

When a file is requested for download, the application acquires a list of IP addresses of hosts where the file is present in full or in part. It then proceeds to download it by managing an optimal number of concurrent connections to some of the given hosts. Over the course of the download, the application makes its partial konwledge of the file available to other downloading nodes.

The implementation consists of two main parts: *network* and *local*. The former part takes care of managing connections to other nodes (as a server and/or a client). When used as a server, it is responsible for communicating the local node's file knowledge and consequently serving any requested blocks from files' encoding. When used as a client, the network part is responsible for concurrently connecting to sufficiently many server nodes, asking them for their knowledge of the file, and requesting a set of blocks from each server node which will maximize the benefit of the concurrent connections.

The local part takes care of storing and retrieving file and auxiliary blocks from disk, computing encoding blocks and decoding from incoming check blocks. In doing this, it takes care of optimizing disk access by batching together groups of disk IO operations, as well as applying write-ahead logging techniques to prevent loss of information (and hence waste the used bandwidth and opportunity).

The remaining part of the report, describes the implementation of the two parts in more detail.

## 2 Specifics

The interface between the local and the network part essentially entails two asynchronous function calls made from the netowrk park into the local part:

1. `void store_block (block, callback)`
2. `void retrieve_checkblock (block_id, ready_callback)`

The following two sections describe how the local part implements these functions, and how the network part uses them.

## 3 Local part

We described the design of the local part by exposing it gradually from bottom to top.

### 3.1 Checkpointing

Most of this part is based around the asynchronous interface for checkpointed disk access. It consists of the operations:

1. `void add_disk (disk_id)`
2. `void remove_disk (disk_id)`
3. `void export_disk (disk_id, filename)`
4. `void truncate (disk_id, nbytes)`
5. `void read (offset, nbytes, callback)`
6. `void write (offset, nbytes, callback)`
7. `void checkpoint ()`
8. `void sync (callback)`
9. `size_t outstanding ()`

The above functions are pretty much self-descriptive. The first six of them represent the operations that fall within the checkpointing control. The last function returns the amount of bytes in outstanding operations that have not been written out to the log yet. A call to `sync` will flush them to the log first, then will execute them, and if successful, will finally clear the log.

### 3.2 Persistent XDR Memory

The encoding and decoding algorithms of Online Codes require non-trivial data structures for representing the state of the decoding and/or encoding process. As the life-cycle of the decoding process usually extends past the life-time of a single program execution, it is crucial that we have a mechanism for storing and retrieving the coding data structures to and from disk, respectively. For these purposes, we have developed a persistent memory manager (somewhat similar to the standard C/C++ memory manager), which is specifically designed to store XDR structures. XDR strucutres represent the necessary level of generality for our needs while at the same time providing the benefit of being well-understood, standardized and highly portable. The data structures are communicated to disk, through the checkpointing interface, and each structure is stored/retrieved, using its XDR encoding and, respectively, decoding specification. The interface is as follows:

1. `void alloc (type, pptr)`
2. `void free (pptr)`
3. `ptr<type> map (pptr)`
4. `void unmap (pptr)`
5. `void test (pptr)`
6. `void sync ()`

The `type` argument to the first function is implemented using C++ templates. `pptr` is a 64-bit number that represents the ID of the data structure in the virtual space of data structure IDs. An allocated data structured can be mapped, which returns a pointer to the corresponding XDR type's C++ object, which provides the necessary manipulation methods.

A call to `sync` writes the entire persistent memory's image to checkpointed disk, using the checkpointed disk access class. Note that the persisten't memory's `sync` function doesn't actually, call the checkpointed disk access `checkpoint` or `sync` functions. This is left to the user. This behavior is crucial to our design, because it allows the user to combine different interdependent persistent memory managers or other disk access objects and checkpoint only when all of them are in a compatible state.

When the persistent memory is loaded from disk, it is read into memory in its entirety. This simplifies the disk format, avoiding the need for a complicated B-tree-like file format, while remaining practical for our purposes. In specific, the amount of technical information the coding mechanism stores in persistent

XDR structures is typically no bigger than 1% of the size of the file being manipulated.

The persistent memory manager is additionally used for storing a small amount of ordering data by the block store manager, described next.

### 3.3 Block store

During the life-time of a download, the coding scheme needs to keep track of all currently known file data. This usually consists of some number of message, auxiliary and check blocks. When downloading big files (to the tune of GBs), the total amount of data in blocks can grow considerably big, albeit always bounded within 5% of the file's size itself. This makes it practically impossible to fit all blocks in memory and, hence, requires a special treatment, provided by the block store module.

The block store module provides functions for reading and writing blocks named by a 64-bit ID, as well as an auxiliary function for renaming blocks, used for optimization purposes:

1. `void write (block_id, block, callback)`
2. `void read (block_id, callback)`
3. `void remove (block_id)`
4. `void rename (old_block_id, new_block_id)`
5. `bool test (block_id)`

Internally, the block store is implemented in terms of two independent objects – a checkpointed file, which contains the blocks themselves in order of their creation, and a persistent memory which contains the mappings between block IDs and their order number in that file.

Blocks' contents is only fetched from disk when the blocks are required. There is no caching scheme, because the nature of Online Code's disk access is purely random and there is no expectation for any pattern in it.

When a file is completely downloaded, the decoded file contents is in the block store in random order. An ordering algorithm on the blocks is performed, using block renaming and copying. When the message blocks are ordered, the leftover auxiliary blocks at the end of the file are removed by truncating the file. Finally, the block file is exported to user space as the newly downloaded file.

After a file's download has completed, its contents as well as its auxiliary blocks are still useful for serving it to future requests. For purposes of supporting a file for serving after it has been downloaded, we provide a secondary block store object, which carries the same interface as the original one. This block store fetches the file blocks (also known as the message blocks in code terms) from the file itself, and the auxiliary blocks from a block store of the first kind. Additionally, it enforces read-only semantics on these blocks, which is useful for debug purposes.

Both block store objects set aside a part of the virtual address space of block IDs for temporary blocks which exist only in memory and are readable and writable. This functionality is useful to the internals of the code implementation. In the next section, we described how the encoding and decoding algorithm use the block store.

### 3.4 Coding implementation

We felt that the coding implementation should be as transperant as possible to the mechanism that stores and retrieves the message, auxiliary and check blocks. In specific, it shouldn't be aware whether this mechanism is asynchronous, multi-threaded, single-threaded or otherwise. To achieve this goal, we implemented the codes as a class that assumes a storage layer, with simple C-style (synchronous) block manipulation interface functions. Assuming that the incoming check blocks (or alternatively message blocks) are already in the store, the only functions that the coding algorithms need are:

1. `void copy (block_id1, block_id2)`
2. `void xoronto (block_id1, block_id2)` (Reads the contents of the first block, XORs it with the contents of the second block, and stores the result in place of the second block.)

3. `void rename (old_block_id, new_block_id)`

Consequently, in order to make the connectrion between this abstract storage layer and our asynchronous block store layer, we introduce an intermediate *block sequencing* layer.

The block sequencing layer is represented by an object that implements the above interface in addition to the following:

1. `void read (block_id, callback)`
2. `void write (block_id, ptr<contents>)`
3. `void sync (callback)`

Upon a call to any of the above functions with the exception of the `sync` function, the sequencing layer only records the function call and its parameters, without actually executing. When `sync` is invoked, the recorded sequence is executed in its proper order, by translating the operation to the underlying block store layer and waiting to ensure that each of them completes successfuly.

As a by-product of this design, we benefit from the fact that the sequencing layer has the power and necessary information to make intelligent decisions about local reordering and pipelining of block operations with intention reduce disk seek times. Of course, there is a limit to how much the recorded sequence can grow before it is flushed, and hence a limit on the benefit we can achieve from reordering block operations. This limit stems from the fact that flushing a sequence will generally require to execute all operations with the block store, before we can invoke the checkpointing `sync` function. Which in turn means that all involved blocks will be present in memory at the same time. In order to make intelligent decisions as to when to flush the sequence's contents, the sequencing layer provides one more function:

1. `size_t outstanding ()`,

which returns an estimate of the number of bytes of memory that will be involved in the process of syncing the outstanding sequence of block operations to disk.

In order to take advantage of the power that the sequencing object provides, we need to make sure that encoding and decoding functions in the codes layer return after a constant number of block operations has been performed. This will give us a fine-grained control over the disk operations, which are the main bottleneck in our scenario. In the context of Online Codes, this is easy to achieve as we can provide functions that represent single edge operations. In the case of encoding, an operation is *"delete edge"*, which corresponds to XOR-ing to blocks together. In the case of decoding, the operation is *"create edge"*, which corresponds to the same block operation:

1. `bool encode_poll ()`
2. `bool decode_poll ()`

The `decode_poll` return TRUE, if the file has been completely decoded. The `encode_poll` function returns TRUE, if the requested block has been computed. In order to initiate generatoin of an encoding block, or to announce the reception of a new check block, the codes class provides two more methods:

1. `void gen_block (block_id)`
2. `void received_block (block_id)`

The former instructs the codes class to enqueue the requested block in its internal block generation queue. The latter inserts the received block's ID in the internal decoding queue. Before this method is invoked, the calling class must ensure that the received block is registered with the sequencing layer, which is achieved by calling the `write` function.

This completes the description of our implementation higher design.

# 4 Network part

The network module of the peer to peer application is reponsible for transfering and receiving blocks of data between peers. It exchanges control messages, reconciles file differences, and manages connections. The network module utilizes the aforementioned `store_block ()` and `retrieve_checkblock ()`

functions to store or get data received over the network.

The top level class of the network module is the `peer` class. It provides the following public interface to the application:

1. void share_file (str filename)
2. void request _file (sha1 file_id)

The `share_file ()` and `request_file ()` functions rely on private methods of the `peer` class which exchange control messages between nodes in the system. These methods include:

1. `void get_state_table (sha1 file_id, u_int64_t node_id)`
2. `void get_stream (sha1 file_id, u_int64_t stream_id, u_int32_t block, conn c)`
3. `void send_state_table (sha1 file_id, conn c)`
4. `void send_block (sha1 file_id, u_int64_t stream_id, u_int32_t block, conn c)`
5. `void send_void (conn c)`

The `get_state_table ()` method sends a query message to another peer asking whether it has a file and how much of the file. A peer that receives a query message will check if it has the file. If it has the entire file, the peer returns a file ID and stream ID for the corresponding file. The stream ID will be used by both peers to help encode/decode blocks of the file. If a peer has only partial knowledge of file, it will send back the file ID and a series of tuples of the form ¡stream ID, last block¿. Each tuple describes a stream the peer has knowledge of and the last block of that stream which it received. A peer maintains a table of these tuples for each file that it has partial knowledge of. When a peer gains full knowledge of a file this table can be removed. Internally, this table is implemented as hash table whose keys are stream IDs.

When a peer A has received the state table of another peer B, it can determine whether it wants to retrieve any of the blocks stored on B. If it has no knowledge of a file, then it will request all the streams that B has starting with block 0. If peer A has some knowledge of a file, it can easily compare state tables. Peer A should request any stream it does not have that B has. For streams that both A and B have, A simply checks if its last block for that stream is less than the last block B has.

When a peer wishes to retrieve blocks of a file it calls the get_stream() function which requests blocks corresponding to a particular stream. This stream most likely corresponds to one returned from the call to get_state_table(). The ”block” argument of the get_stream() function specifies which block number should be returned first. A peer that shares a file will call send_block() to return blocks of a file to another requesting peer starting at a particular block number. If for some reason a requesting peer wishes to stop receiving blocks, it can call send_void(). The peer that was sending blocks will cease to do so after receiving such a message.

When a peer receives a data block, its computes the SHA1 hash of its block number and stream ID. The yields the check block ID of the data block. The check block ID along with the raw data is passed to store_block() function, which can determine whether the file is completely downloaded or not.

Similarly, a sending peer computes the check block ID (using SHA1 of a block number and stream ID) to obtain a data block using the retrieve_block() function. This data can then be sent back to the requesting peer.

The conn argument used in get_stream(), send_block(), and send_void() is a structure representing a connection between two peers. This structure includes a unique connection ID, a connection state variable, and a connection object. The state of connection can be depends on the control messages transferred using a connection. For example, the state might reflect if a connection is waiting for an incoming state table or sending blocks of data to another peer.

A connection object is implemented by the XPRT class. The XPRT class sends control messages, also known as tokens, over a TCP connection. Tokens are marshaled and unmarshaled by peers using XDR routines. Callbacks can be registered with an XPRT object to notify when data has been sent/received or when the TCP connection has been closed. Statistics regarding the bandwidth of usage of an XPRT object can also be requested. A peer collectively manages all XPRT objects through a class called XPRT_MGR. Any new TCP connections are accepted by the XPRT_MGR, wrapped into XPRT objects, and then stored in a linked list.

### 4.1 Test results

When a node has partial knowledge of a file, it maintains a table of the streams it has received along with the last block in the stream it received. When nodes receive a new stream they always receive block 0 first, then block 1, and so on. Therefore a node only needs to remember that last block it received to record its partial knowledge of a file.

An interesting open problem is to determine how the large the state table, containing ¡stream ID, last block¿ pairs can get under certain circumstances. A test scenario was set up where N peers with no knowledge of a file (clients) tried to obtain the file from 1 source node with full knowledge of the file. The N clients with no knowledge would choose a random peer and ask for its state table. In the early portion of the test scenario, many clients would query other clients requesting their state table. However since only the source node had full knowledge of the file, the result of these queries resulted in no sharing of knowledge. Only when one client randomly contacted the source node, did knowledge of the file become more distributed.

If network connections are never interrupted or closed prematurely, then a client who contacts the source node or another client with full knowledge of the file, can itself receive full knowledge of the file via one connection. Therefore the maximum number of entries in the client's state table will always be 1, since this entry corresponds to the peer which gave you knowledge of the file.

A more interesting scenario occurs when network connections are randomly closed, a situation that frequently occurs in real peer to peer communication. Tests were run where a certain percentage of network connections were dropped when a peer with full knowledge of a file was communicating with another peer having partial or no knowledge of a file. Connections that were closed prematurely, were closed a random point during communication. For example, a client may have received 0, 1, 50, or 99 percent of the file before the connection was closed. Tests were run where the the percentage of connections dropped ranged from 0 to 100, in increments of 10 (i.e 0, 10, 20, 30,... 100). The source node maintained a 10 MB file, and individual portions of the file were sent in 1 KB blocks. The average of 5 runs was recorded.

As expected, the average number of entries in a client's state table increased as the percentage of dropped connections increased. There is a large increase when the percentage of dropped connections goes from 80 to 90 percent and also from 90 to 100 percent. These large changes are probably due the fact that less information is being shared with each connection. When connections are frequently dropping, a peer is likely to get a small piece of knowledge about a file then get cut off. Therefore, the peer will probably end up contacting many other peers, who in turn will give it some knowledge before cutting it off. Note that the maximum number of entries in a peer's state table will always have an upper bound based on the number of peers in the system. This is because stream IDs are based on the IDs of peers.

It is uncertain how the number of clients in the system would affect the maximum number of entries in a peer's state table. Tests using the same scenario as mentioned previously were run, while tuning the the number of peers in the system. The rate at which connections were dropped was 50%. We were unable to simulate a large number of peers in the system due to the hardware constraints. The maximum number of peers tested was 30. When tests with more peers were run, the "accept" system call kept returning errors, complaining "too many open files" existed.

Results for all the data points show that the maxi-

Testing max number of entries in state table versus % of connections dropped. Sending a 10 MB file using 1KB block buffer. 10 clients and 1 source node. Average of 5 runs.

| Max entries | % connections dropped |
|---|---|
| 1.0 | 0 |
| 1.04 | 10 |
| 1.10 | 20 |
| 1.26 | 30 |
| 1.48 | 40 |
| 1.60 | 50 |
| 2.04 | 60 |
| 2.30 | 70 |
| 2.62 | 80 |
| 4.16 | 90 |
| 8.66 | 100 |

mum number of entries was around 1.6. Therefore it appears that the number of clients in the system does not affect the size of a peer's state table. This implies that the connection drop rate, which determines how much knowledge of a file can be transferred per connection, is the real source of variance for the number of entries in the state table. No matter how many peers exist in the system, the amount of information that can be exchanged over a given connection, determines how many peers a node will contact to gain full knowledge of a file.

We would have liked to evaluate the affect of file size on the number of entries in a peer's state table. However due to time constraints this was not done. We expect the number of entries in the state table to increase with file size since more data is required for full knowledge of a file and the average amount of data shared per connection is fixed.

## 5 Conclusion

We believe that our implementation is on a promising path, but there is further work to be done. In particular, the networking algorithm needs to be upgraded to handle multiple concurrent connections as well as different simultaneous downloads.

Testing max number of entries in state table versus number of clients (no knowledge of file). Sending 10MB file using 1KB block buffer. 1 % of connections dropped. Average of 5 runs.

| Max entries | Num. clients |
|---|---|
| 1.6 | 10 |
| 1.61 | 15 |
| 1.68 | 20 |
| 1.67 | 25 |
| 1.65 | 30 |