RaptorQ-based File Transfer Protocol

Yilong Li
yilongl@cs.stanford.edu

Raejoon Jung
raejoon@stanford.edu

Yu Yan
yuyan0@stanford.edu

1. INTRODUCTION

2. IMPLEMENTATION

2.1 Reliable file transfer

In Tornado transfer, the reliability guarantee is achieved at the application level by the use of RaptorQ code. This eliminates the need for retransmission at the file transfer application level, which greatly simplifies the design and implementation of a reliable file transfer protocol. In Tornado transfer, the sender has no need to keep track of exactly what symbols are received by the receiver. Therefore, the ACK message is essentially just a bitmask of 256 bits (256 is the maximum number of blocks specified in RFC6330) that records which blocks have been successfully decoded by the receiver. Tornado transfer protocol has a handshake procedure similar to TCP to establish the connection. Once the handshake succeeds, the sender starts sending source symbols of each block in order. Furthermore, in order to compensate for the potential lost symbols, the sender sends one repair symbol for each previously un-ACK'ed block after every X source symbols. After all source symbols have been sent, the sender simply sends repair symbols for each un-ACK'ed block in a round-robin fashion. Ideally, the repair symbol transmission interval X should be set to a value such that after all source symbols of block n has been sent, the receiver has received enough symbols for block n-1 for decoding. This way, the receiver only needs to keep roughly one block in memory for decoding at a time. The receiver sends back a heartbeat ACK message constantly to compensate for potentially lost ACK messages. Besides, it immediately sends back an ACK message once it decodes a new block to reduce the probability of sender sending more symbols for the decoded blocks. Once the receiver decodes the entire file, it simply exits. The sender will also terminate once it figures out that the receiver exits. This can be done either by relying on the shutdown mechanism of DCCP socket or through an ICMP destination unreachable message generated by the receiver.

Parameter setting There are two most important parameters that we can pass on to the RaptorQ library: symbol size and number of symbols per block.

1. Symbol size In our current implementation, we choose the symbol size to be 1400 bytes to avoid IP fragmentation. We could potentially choose a larger number to, say, reduce the number of symbols for performance reason described in the section below. However, the downside of a larger symbol size is that each symbol may be fragmented at the IP layer and the loss of each fragment results in the loss of the entire symbol. In other words, the nice property of digital fountain that every packet received contributes to the decoding of the entire file is no longer preserved. Currently, we have not quantified the effect of a larger symbol size.

2. Number of symbols per block The number of symbols per block is critical to the performance of encoding and decoding. Generally speaking, we would like to keep it as small as possible. RFC 6330 does not allow us to explicitly set this value. Instead, we provide a parameter WS, the maximum size of a block that can be efficiently decoded in the working memory of the receiver, and RFC 6330 describes the procedure for deriving the number of symbols per block based on it. This parameter derivation algorithm involves lookups into the hardcoded RaptorQ matrices and is not very straightforward. Therefore, our current implementation enumerates parameter WS starting from a small number and increase it by one each time to search for the smallest legal value of the number of symbols per block. In practice, this search procedure is fast enough to be hardly noticeable.

Performance bottlenecks Our current implementation of Tornado transfer has two performance bottlenecks which limits its practicality. We briefly describe the problems here and leave the solutions as future work.

1. Precomputation The most computational expensive in RaptorQ encoding/decoding process is the process of precomputing intermediate symbols for each block. The time complexity of the precomputation is cubic in the number of symbols per block. We currently use a background thread for precomputing intermediate symbols while transmitting symbols. However, this has be-

come a bottleneck for larger file size. For instance, for a file of size 100MB, the smallest number of symbols per block that is legal with respect to RFC 6330 is 296.

2. Decoding Once intermediate symbols have been precomputed, even though both encoding and decoding are linear in the number of symbols, decoding tends to fall behind encoding for two reasons. First, encoding is a stream operation that takes constant time to generate the next symbol, while decoding is a batch operation that only happens after enough symbols of a block have been received. Second, decoding is inherently slower than encoding in the current implementation of libRaptorQ. To resolve this bottleneck,

2.2 Congestion control

Tornado Transfer utilizes DCCP, the Datagram Congestion Control Protocol, to provide congestion control mechanism for unreliable datagrams. DCCP is designed to make it easy to deploy delay-sensitive applications, such as streaming media, which prefer timeliness to reliability. It is also appropriate for Tornado Transfer because we have offered reliability using RaptorQ in the application layer.

There are mainly two categories of congestion control algorithms in DCCP— CCID 2 and CCID 3. CCID 2 denotes TCP-like congestion control that describes Additive Increase Multiplicative Decrease (AIMD) congestion control mechanism including several other features in TCP. CCID 2 is suitable for applications like Tornado Transfer that achieve maximum throughput over long term. CCID 3 denotes TCP-Friendly Rate Control that describes an rate-controlled congestion control mechanism. CCID 3 is suitable for streaming applications due to its lower variation in terms of throughput.

We choose to adopt DCCP as transport layer protocol with CCID 2 enabled for the sake of its simplicity. DCCP has been in the Linux kernel and creating a DCCP socket to send and receive is much like creating a TCP socket. We simply replace UDP with DCCP, without having to add congestion control mechanism on top of UDP by ourselves.

The drawbacks of using DCCP sockets are notable as well. According to our experiments in mahimahi and RFC 5597, NAT (Network Address Translation) support for DCCP is not functioning properly, which results in very little use of DCCP. Besides, its congestion control mechanism, such as replying ACKs, is hidden from us. The black box makes it harder for us to debug when running into problems.

In the future work, we plan to implement congestion control algorithm, e.g. LEDBAT (Low Extra Delay Background Transport), above UDP on our own.

3. EVALUATION

4. CONTRIBUTION

5. REFLECTION

6. CONCLUSION