RaptorQ-based File Transfer Protocol

Yilong Li
yilongl@cs.stanford.edu

Raejoon Jung
raejoon@stanford.edu

Yu Yan
yuyan0@stanford.edu

1. INTRODUCTION

The File Transfer Protocol (FTP) has been widely used to transfer computer files between hosts for quite a long time, especially within the internal networks of organizations. It has many advantages such as no size limitation on single transfers, the ability of transfer resumption if the connection is lost, faster transfer speed, etc. Even though with disadvantages like inadequate security, FTP still works better than HTTP in some specific scenarios.

To get the most out of FTP, we propose to make a RaptorQ-based FTP. The original FTP is based on TCP and thus suffers the shortcomings of TCP. For example, it has to send acknowledgement for each datagram to provide reliability, leading to limitation on the goodput. Indeed, consider the case where there is packet loss, retransmission wastes at least a round-trip time.

However, RaptorQ can provide an alternate method for achieving reliability, with only one acknowledgement required. More specifically, the receiver can retrieve the original file by performing calculations on any enough subset of the data flow encoded by RaptorQ, and send back only one acknowledge to indicate success to the sender. The sender does not need any responses in the middle of generating arbitrary redundancy and sending datagrams.

In addition, we suppose that RaptorQ-based FTP can further benefit from a customized congestion control algorithm. Therefore, we will design and implement our own congestion control algorithm as well to boost the performance, but avoid congestion collapse in the meantime.

Finally, we will evaluate the performance of RaptorQ-based FTP and compare the results with TCP-based FTP.

2. DESIGN GOAL

1. To improve the goodput of data transfer.
2. To maintain the reliability.
3. To optimize the performance but avoid congestion collapse meanwhile.

3. APPLICATION OF RAPTORQ TO FILE TRANSFER PROTOCOLS

RaptorQ code is a state-of-the-art fountain code. Fountain codes are promising forward error correction codes due to the following properties,

(a) **Arbitrary amount of redundancy can be added on the fly.** Fountain codes can generate an arbitrary number of encoded data for a given original message. It provides arbitrary amount of redundancy to the encoded data which allows to be resilient to arbitrary amount of data lost. As a result, the receiving end can retrieve the original message without any retransmission.

(b) **Original message can be retrieved by any subset of the encoded data of size of at least the size of the original message.** Fountain codes are a class of erasure codes which is reliable in transmission over channels where subset of the transmitted data can be randomly erased. Therefore it can provide robustness in applications over networks with packet loss. Fountain codes including RaptorQ can achieve this as long as the receiver receives the amount encoded data which is the same as the size of the original message regardless of which encoded data the receiver has failed to receive.

These properties can be beneficial in file transfer applications in two ways.

(a) **Transport protocol can be simplified without losing reliability.** Unlike how TCP provides reliability by tracking successful transmission using sequence numbers and acknowledgement, the sender can encode the source file with RaptorQ and send arbitrary number of datagram until the receiver received enough packets to decode the original file. Only one acknowledgement is required and it is to avoid the sender sending excessive amount of datagrams, not for reliability guarantees.

(b) **Goodput of the receiving end can be improved.** Eliminating acknowledgements can also provide goodput benefits for short flows in a network with packet losses. If the transport protocol requires an acknowledgement followed by a retransmission, it has to wait for at least a round-trip time whenever there is a packet loss for retransmission. However, a RaptorQ-applied sender does not need to wait for any response and send a packet that can contain the same amount of information as the lost packet.

We expect to experience these benefits once we develop a file transfer application using RaptorQ.

4. FEASIBILITY AND POTENTIAL OBSTACLES

For the file transfer application, three modules are required; (1) RaptorQ encoding/decoding module, (2) transport module, and (3) congestion control module.

There are open source implementations for RaptorQ [1], [2]. `libRaptorQ` project [2] based on C++ seems to be a promising implementation that we can rely on for the coding portion of the application. The project claims that it can encode and decode within the throughput range of ∼5 Mbps to ∼5 Gbps, depending on the encoding parameters. In our project, we would need to tune the parameters to avoid the coding module being the bottleneck of the file transfer. Dynamic tuning based on packet loss statistics in the network would be required for optimization. However, we can start with finding a rough range of parameter values for a single test case, for instance testing in the Stanford network. If it is impossible to get sufficient throughput for on-the-fly encoding, we can limit our application to (1) content providing with already encoded contents or (2) peer-to-peer file transfer where multiple senders are involved for a single file transfer.

Transport module is responsible of transporting RaptorQ-encoded payload. It guarantees sufficient payload reception in the receiving host for following the following decoding procedure. The module should not only initiate the flow, but also should know when to terminate the flow. This prevents the application from keep sending datagrams after the receiving end has correctly retrieved the file. The rateless property of RaptorQ can simplify the transport protocol, since it only requires a single acknowledgement when the receiving host retrieved sufficient number of datagrams in order to retrieve the file. We expect this part to be straightforward.

Congestion control module provide fairness over the application and other existing applications, which mostly are using TCP congestion control. The application will avoid occupying the majority of the bottleneck link bandwidth in the transmission path. We can develop our own congestion control algorithm if there is room for optimization. If this is not feasible, using existing congestion control algorithms in TCP is also an option, even though we do not use TCP as our transport protocol. We expect most of the work will be related to this module since we cannot rely on the existing kernel module for TCP.

5. INTELLECTUAL CONTRIBUTION

This project will make several contributions:

1. The first open source implementation of file transfer protocol based on RaptorQ.
2. A (potentially) novel congestion control module for RaptorQ-based file transfer application.
3. A comparative study of RaptorQ and TCP/IP for file transfer.
4. Upstream patches and optimizations to help push the `libRaptorQ` project towards a mature open source RaptorQ implementation.

6. SCHEDULE

The following tentative schedule distributes the work of this project over 8 weeks.

- Experiment with the encoding/decoding module of `libRaptorQ`. (1 week)
- Design and implement the transport module based on RaptorQ. (1 week)
- Design and implement the congestion control module. (4 weeks)
- Performance evaluation and comparative study. (1 week)
- Final write-up. (1 week)

7. REFERENCES

[1] Openrq: an open-source raptorq implementation. `http://openrq-team.github.io/openrq`.

[2] Raptorq rfc6330 c++11 implementation. `http://github.com/LucaFulchir/libRaptorQ`.