Implementation Of A Raptorq-Based Protocol For Peer To Peer Network

Yuzhu Bai
University of Mississippi

IMPLEMENTATION OF A RAPTORQ-BASED PROTOCOL FOR PEER TO PEER NETWORK

A Thesis
presented in partial fulfillment of requirements
for the degree of Master
in the Electrical Engineering
The University of Mississippi

by

Yuzhu Bai

Aug 2015

ABSTRACT

The object of this thesis is to develop and test a Ruby based implementation of the RaptorQP2P protocol. The RaptorQP2P protocol is a novel peer-to-peer protocol based on RaptorQ forward error correction. This protocol facilitates delivery of a single file to a large number of peers. It applies two levels of RaptorQ encoding to the source file before packet transmission. Download completion time using RaptorQP2P was found to be significantly improved comparing to BitTorrent.

We developed a Ruby interface to the Qualcomm proprietary RaptorQ software development kit library. Then we achieved the two levels of RaptorQ encoding and decoding with the Ruby interface. Our implementation uses 5 threads to implement RaptorQP2P features. Thread 1 runs as a server to accept the connection requests from new peers. Thread 2 works as a client to connect to other peers. Thread 3 is used for sending data (pieces) and thread 4 is to receive data from neighboring peers. Thread 5 manages the piece map status, the peer list, and choking of a peer.

We first tested communication modules of the implementation. Then we set up scheduled transmission tests to validate the intelligent symbol transmission scheduling design. Finally, we set up a multi-peer network for close to practical tests. We use 5 Raspberry Pi single-board computers to act as 1 seeder and 4 leechers. The seeder has the whole file and delivers the file to the 4 leechers simultaneously. The 4 leechers will also exchange part of the file with each other based on what they have received.

Test results show that our implementation attains all the features of RaptorQP2P: the implementation uses two levels RaptorQ encoding; a peer is able to download a piece from multiple neighbors simultaneously; and a peer can send the received encoded symbols of a piece to other peers even if the peer does not have the full piece yet.

DEDICATION

To my family.

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor, Prof. John N. Daigle, for his professional guidance and enormous patience throughout my graduate study at the University of Mississippi. My grateful appreciations also go to Prof. Feng Wang, Prof. Ramanarayanan Viswanathan, Dr. Xiao Di and Prof. Lei Cao for the many helpful comments and suggestions on my research.

I would like to thank Dr. Xiao Di, Prof. Lei Cao, Prof. Ramanarayanan Viswanathan and my family for supporting me to pursue a degree in Master of Science.

Finally, I am most grateful to my family and all my friends in Oxford, Mississippi for all their help in every way possible. A special gratitude must go to Dr. Xiao Di for his selfless support and invaluable suggestions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The objective of this thesis is to develop and test a Ruby based implementation of RaptorQP2P protocol. The RaptorQP2P protocol is a novel peer-to-peer protocol based on the RaptorQ Forward Error Correction (FEC) for reliable sharing of large files over the Internet [1]. The RaptorQ is a type of fountain code and belongs to the family of Raptor codes [2]. The primary application of RaptorQ is application layer forward error correction (FEC).

In this work, we utilize the Qualcomm proprietary RaptorQ software development kit library (RaptorQ SDK) to achieve RaptorQ encoding and decoding. We developed a collection of C functions that provide access to functions in the RaptorQ library. Then a Ruby interface to the collection of C functions was developed. This combination forms a rapid prototyping kit for implementing RaptorQ based content delivery. Run time for major subcomponents was quantified through a series of tests performed on Mac OS and Raspberry Pi platforms. Test results of RaptorQ library run times and peer to peer file distribution are presented.

The implementation in this thesis facilitates delivery of a single file to a large number of peers. We built a Raspberry Pi based research platform that enables us to

- download and upload data from any peer;
- trace and record data transmitted from any of the peers in the network; and
- calculate both the downloading and uploading time of a peer.

The remainder of this chapter gives an introduction to RaptorQ FEC and then briefly describes the RaptorQP2P protocol. An outline for the remainder of the thesis is also provided.

1.1 RaptorQ FEC

RaptorQ Forward Error Correction (FEC) belongs to the family of Raptor codes. It is a type of fountain code [3]. Fountain codes are also called rateless erasure codes, and the first practical fountain code is LT codes, which were invented by Luby in 1998 [4].

In a general LT coding scheme, the original file is divided into blocks and each block is then divided into source symbols. Each encoded symbol is generated as a linear combination of source symbols. The number of encoded symbols is theoretically limitless. In the decoding part, the decoder is able to recover the original block when a sufficient number of distinct symbols is received. These symbols can be any set of the generated encoding symbols as long as they are slightly longer in length than the original block [4]. LT codes are proved to be very efficient as the source data grows. That is, the average overhead required to recover source data decreases with increasing block size [5].

Following Luby's work, Shokrollahi [3] developed Raptor codes as an extension of LT codes. Raptor codes can be viewed as a combination of LT-code and LDPC code [6]. In the Raptor coding scheme, some intermediate symbols are generated from the source symbols by a high-rate LDPC code in the pre-coding stage. These intermediate symbols are then encoded by LT-code. The fountain property of Raptor codes is provided in the LT-coding stage. Raptor codes are proved to outperform LT codes on a wide variety of noisy channels [7].

RaptorQ code is a new variant of Raptor codes. Comparing to standardized Raptor codes, RaptorQ code has several improvements to the encoding and decoding processes. First, before the intermediate symbols are generated, RaptorQ augments FEC source blocks with additional padding symbols to ensure faster encoding and decoding [6]. Second, RaptorQ adopts a two stage pre-coding algorithm (LDPC and HDPC) to generate the intermediate symbols. Third, the RaptorQ uses a modified more efficient encoding process in the second encoding step. Finally, the RaptorQ code operates over the finite field GF(256) instead of the standardized Raptor code operating over Galois field GF(2) [3]. All the above features not only offer RaptorQ a better coding performance, but also allow RaptorQ to support larger source symbol block sizes. Specifically, RaptorQ supports up to 56,403 source symbols in a source block, and can generate as many as 16,777,216 encoded symbols for one source block [3].

The RaptorQ code also has a smaller decoding overhead requirement comparing to standardized Raptor codes. For example, the first widely adopted Raptor code, the Raptor 10 code, requires an overhead of 24 to achieve a failure probability of 10^{-6} while the RaptorQ code ensures a failure probability of less than 10^{-6} with an overhead of 2 symbols.

1.2 RaptorQP2P

The RaptorQP2P protocol facilitates reliable peer-to-peer sharing of large files over networks [1]. It utilizes two levels of RaptorQ encoding and an intelligent symbol scheduling algorithm to remedy some limitations of the BitTorrent protocol. The RaptorQP2P protocol can be viewed as an extension of the BitTorrent protocol, which was designed by Bram Cohen in 2001. In order to clarify the motivation for RaptorQP2P, we will review BitTorrent first.

BitTorrent is a peer-to-peer file sharing protocol that efficiently distributes large files over the Internet. In a traditional client-server protocol, the central server is responsible for delivering all the content to clients (peers). On the contrary, a peer-to-peer protocol enables the transfer of data from one peer to another. Some limitations of the servers and the network can be highly reduced in this way. Besides the peer-to-peer feature, BitTorrent protocol introduced a piece selection strategy to decide what data to request and a peer selection strategy to decide which peers to transfer pieces [8]. The piece selection strategy in BitTorrent begins by selecting pieces at random and then switches to rarest-first. The peer selection strategy, which is known as tit-for-tat, always unchokes peers with highest data rate every ten seconds. These two strategies guarantee good performance when the network is in steady state. However, in the real world, the peers and the network are always changing. The overall system performance may be significantly less than optimal [1].

With above findings, Su, Wang, Daigle and Shan proposed a novel protocol named RaptorQP2P, which is based on RaptorQ coding, to overcome the limitations of BitTorrent. The RaptorQP2P protocol encodes the source file with two levels of RaptorQ before transmission: the file is firstly RaptorQ encoded to generate a group of source blocks and repair blocks, and then each block is RaptorQ encoded to generate source symbols and repair symbols. These symbols are then distributed over the Internet. A peer is able to reconstruct a block after collecting enough distinct symbols for the block. When the peer possesses enough distinct blocks, the file can be recovered. By utilizing RaptorQ codes, the RaptorQ protocol allows multiple peers to send symbols (slices) of the same block (piece) to a peer simultaneously. This is a significant advantage over BitTorrent. Moreover, to maximize the utilization of peers' upload capacities, this protocol also allows opportunistic transmissions, where a peer can send the received encoded symbols of a piece to other peers even if the peer does not have all the slices of that piece. A more detailed description of RaptorQP2P is given in Chapter 3.

1.3 Organization of Thesis

The remainder of this thesis is structured as follows. Chapter 2 introduces the interface design and some tests for the RaptorQ libraries from Qualcomm. Chapter 3 gives a brief review of the RaptorQP2P. Chapter 4 describes the practical implementation of this protocol. Chapter 5 shows test results of the RaptorQP2P implementation. Chapter 6 concludes the thesis with a discussion on the future work.

CHAPTER 2

RUBY INTERFACE TO RAPTORQ LIBRARIES

This chapter provides a thorough discussion of our Ruby interface to the functions of the RaptorQ libraries which were provided by the Qualcomm Technologies, Inc. We begin by describing the Qualcomm RaptorQ software development kit in Section 2.1. In Section 2.2, we explain how the interface was designed for the library. In Section 2.3, we present a series of tests of our interface and the library.

2.1 The RaptorQ Library

The Qualcomm proprietary RaptorQ SDK is a specification-compliant implementation of the RaptorQ code specified in [3]. It consists of a RQ Encoder library and a RQ Decoder library, where each provides functions that enables us to build RaptorQ encoding/decoding applications.

Figure 2.1 illustrates a system-level view of communication between the RQ Encoder and RQ Decoder. This architecture is also adopted in our implementation of RaptorQP2P. In this architecture, the sender application retrieves and presents blocks of source data to an RQ Encoder to generate repair data (encoded data). Both the source data and repair data are passed to the sender transport layer for packetizing and transmission. At the receiver end, the receiver transport layer collects the packets and deliver them to the receiver application. When the RQ Decoder receives enough data (whether source or repair), it is able to recover the original source file. The benefit of using RaptorQ FEC in this architecture is that the RaptorQ FEC technology provides packet-level erasure protection against the loss of packets.

In the above RaptorQ scheme, the original source file is first divided into a number of source blocks. Each source block is further partitioned into equal-sized source symbols.

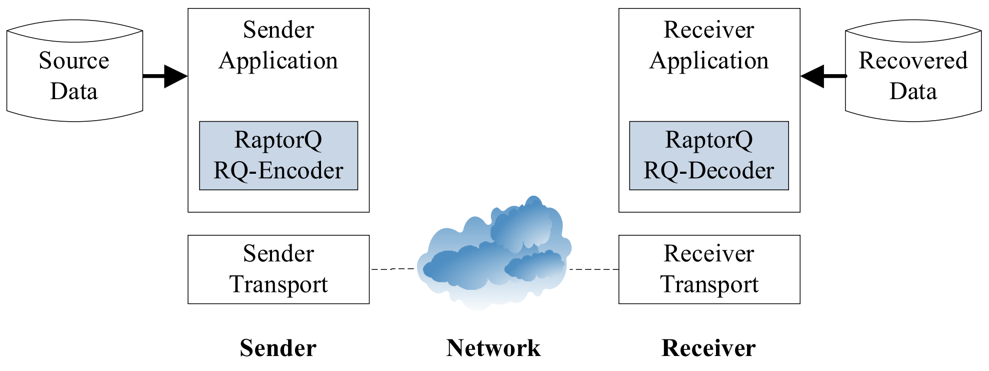


Figure 2.1. Communication Architecture with RQ Encoder and RQ Decoder.

Figure 2.2 shows a typical processing flow of the RQ Encoder and Decoder, which is also adopted in our implementation. The sender application passes source blocks to the RQ Encoder to generate intermediate blocks. The intermediate blocks are to be used to generate repair symbols. When the RQ Decoder have received enough symbols, whether source symbols or repair symbols, the original source blocks can be recovered. The Qualcomm RaptorQ SDK provides a series of functions to accomplish the above process. Our interface design is to wrap these functions in such a way that a single call in Ruby script operates the encoding or decoding process. Before discussing the interface design, we will introduce the RQ library first.

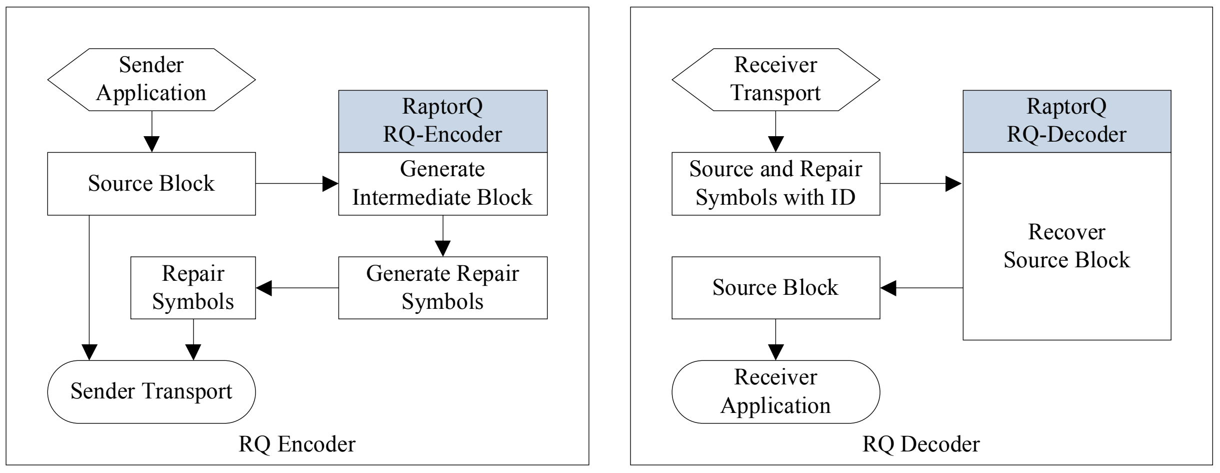


Figure 2.2. Processing Flow for RQ Encoder and Decoder.

The RQ Encoder Library consists of 11 functions, but only 7 of them are crucial to fulfill a RaptorQ encoding process. Our interface for the encoder part integrates the 7 functions into one single function StringSimpleSend(). This function takes in the source data as a string and returns encoded symbols. Table 2.1 gives a brief introduction to the 7 functions.

Table 2.1. Essential Functions in RQ Encoder library

Function	**Purpose**
DFRQEnclnit( )	Initialize the working block memory.
DFRQEncReset( )	Reset the working block memory.
DFRQEncPrepare( )	Prepare for the generation of an intermediate block.
DFRQEncInitSrcBlock( )	Initialize a source block for the generation of intermediate block.
DFRQEncGenIntermediateBlock( )	Generate an intermediate block.
DFRQEncGenRepairSymbols( )	Generate repair symbols.
DFRQEncGetSourceSymbols( )	Produce source symbols.

The RQ Decoder Library consists of 11 functions, whereas 7 of them are crucial to fulfill a RaptorQ decoding process. We also integrated the 7 functions into one single function, named FileSimpleDecode(). This function takes in the received symbols and tries to reconstruct the source data. Table 2.2 briefly introduces these functions.

2.2 Interface Design for the RaptorQ Library

In our implementation of the RaptorQP2P protocol, we achieve the two level RaptorQ encoding (which is an essential to the RaptorQP2P) via the RaptorQ libraries. These libraries provide a set of C functions (described in Section 2.1) that enables us to build our own RaptorQ encoding and decoding applications. While Ruby is well known for rapid prototyping, we choose to implement the RaptorQP2P in Ruby.

Table 2.2. Essential Functions in RQ Decoder library

Function	**Purpose**
DFRQDecMemRequest( )	Request the memory needed for decoder operation.
DFRQDecInit( )	Initialize the decoder block.
DFRQDecReset( )	Reset the decoder block.
DFRQDecAddRcvSymbolIDs( )	Inform the RQ Decoder of the ESIs (Encoding Symbol ID) of received symbols.
DFRQDecPrepare( )	Prepare for the reconstruction of a source block.
DFRQDecInitRcvBlock( )	Initialize a receive block for the recovery of a source block.
DFRQDecRecoverSource( )	Recover a source block.

To simplify the prototyping, we developed a collection of C functions that provide access to functions in the RQ library. In the encoding part, function StringSimpleSend() was developed as a full RaptorQ encoding function utilizing the APIs. In the decoding part, function FileSimpleDecode() was developed to achieve the RaptorQ decoding function. We call these two functions *Interface Functions*. The interface functions are then wrapped by SWIG along with the APIs for Ruby access.

2.2.1 Interface Function Design

The API functions introduced in sections 2.1 contain a number of memory-level operations. This makes it complicate if we try to fulfill the RaptorQ encoding and decoding by calling these functions directly from Ruby. Therefore we developed interface functions in C to achieve the full RaptorQ encoding and decoding functions.

The first interface function developed by us is StringSimpleSend(). The formal variables for this function are as follows:

- **Symbol size**. This is an integer-valued variable that gives the length of every symbol in bytes. The idea is to give the user the freedom to partition the file according to his own objectives.

- **Source data pointer**. This is a pointer to the input data, namely the source data to be encoded. The source data is a binary string that is passed from another function.

- **Data size**. This integer-valued variable tells the length of the input string (source data) in bytes. It is essential for function StringSimpleSend() to obtain both the pointer and length of source data because the source data is treated as a binary string, and it can be complicate to find out the length of a binary string in C. Directly passing in the string length helps to simplify the development process.

- **Encode ratio**. This is an integer-valued variable no less than 100, typically set to be 200, 300, 400, etc. The literal meaning of encode ratio is the percentage protection to be applied to the source data. For example, Encode ratio = 100 stands for a 100% protection where all the output symbols will be source symbols and there is no repair symbols. Encode ratio = 200 means as many repair symbols as source symbols will be generated.

- **Output file name**. This is a character-valued variable indicating the output file name. The source symbols partitioned from source data will be written into file *Output file name.src*, and the generated repair symbol will be written into file *Output file name.rep*.

The work flow of StringSimpleSend() is shown in Figure 2.3. This figure also shows the corresponding API functions that are called in each step. At the very beginning, the program allocates the working memory and calls DFRQEncInit() to initialize the RQ Encoder. Then the RQ Encoder is reset by DFRQEncReset(). After that, the interface function calls DFRQEncPrepare() and DFRQEncIntSrcBlock() to prepare for and initialize intermediate block generation. When all the preparation jobs are finished, intermediate blocks are generated by DFRQEncGenIntBlock(). Source symbols and repair symbols can be generated by DFRQEncRepairSymbols() and DFRQGetSourceSymbols(). When the operation on one block of data is finished, the program moves on to next block if there are any.

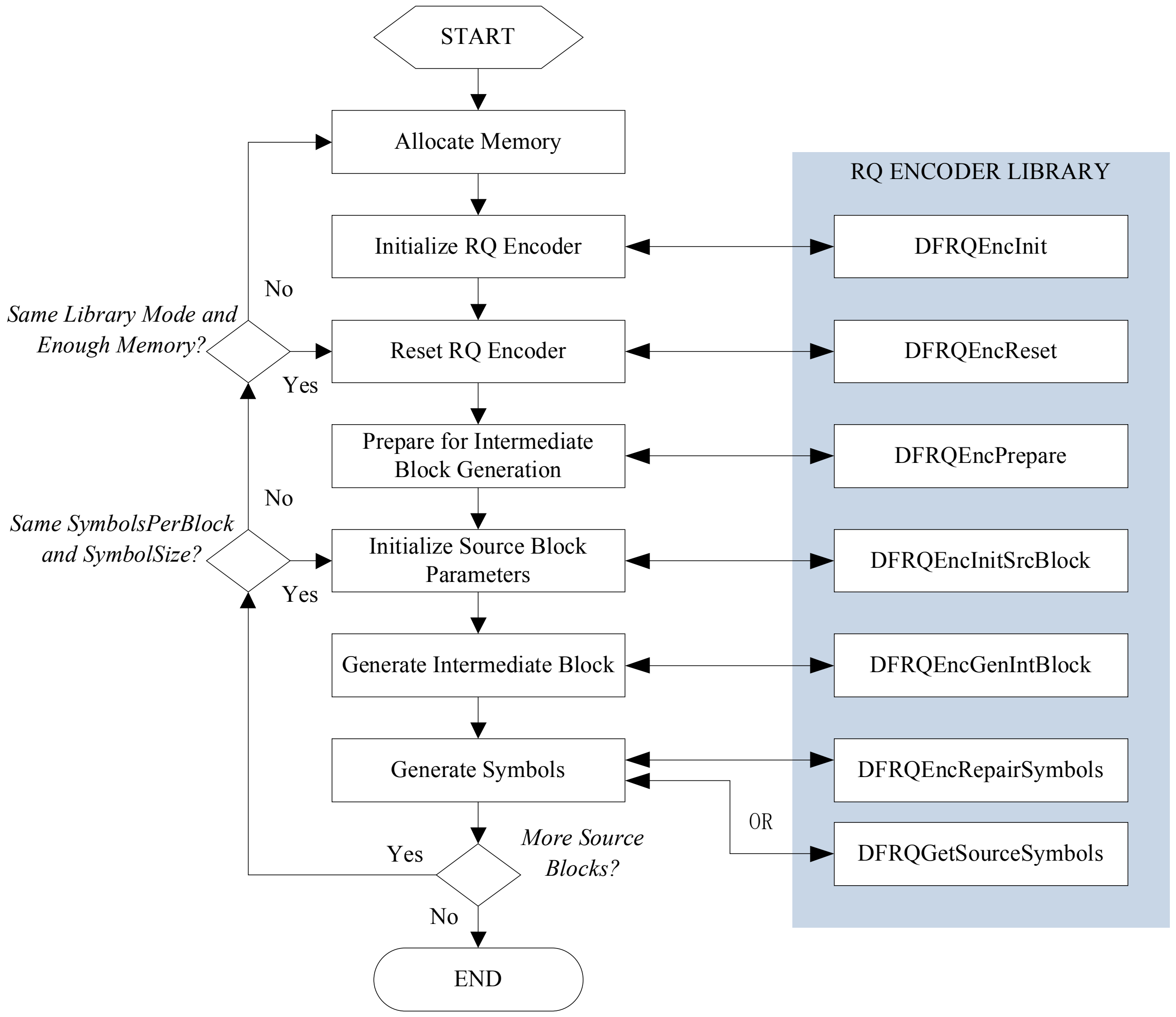


Figure 2.3. Workflow of RQ Encoder Interface Function StringSimpleSend() (Adopted from *Qualcomm RaptorQ RQ Encoder Developer's Guide*).

The decoding interface function developed by us is FileSimpleDecode(). This function tries to recover the original data from a bunch of symbols. The recovered data will be returned in a file after a successful decoding, or the function will return a failure status. The formal variables for this function are as follows:

- **Symbol size**. This integer-valued variable corresponds to the symbol size defined in the encoding process.

- **Input file name**. This is a character-valued variable. All the incoming symbols will be stored in a binary file in the name of *Input file name*.

- **Output file name**. This is also a character-valued variable indicating the output file name of the decoder. The recovered data will be written into file *Output file name* if the decoding process succeeds.

- **File size**. This integer-valued variable indicates the size of the original source data. It helps to determine the memory space and number of symbols needed for decoding operation.

- **Number of extra symbols**. This is an integer-valued variable no less than 0. The number of extra symbols stands for the difference of number of received symbols and the number of original source symbols. If the number of source symbols is k and we received $k + 1$ symbols, then the number of extra symbols equals 1.

The work flow of FileSimpleDecode() is shown in Figure 2.4. At the beginning, the API function DFRQDecMemRequest() gets memory block sizes so that the program is able to allocate memory. After that DFRQDecInit() initiates and DFRQDecReset() resets the decoder. Then some preparation work is done by DFRQDecPrepare(). The received block parameters is initialized by DFRQDecInitRcvBlock(). When the preparation is finished, the program tries to recover the source block. If the recovery succeeded, the recovered block will be written into an output file. The program moves on to next block if there is any.

2.2.2 Wrapping C Functions with Ruby Using SWIG

We use the software development tool Simplified Wrapper and Interface Generator (SWIG) to generate the interface between Ruby and the C functions. SWIG is a software development tool for building scripting language interfaces to C and C++ programs [9].

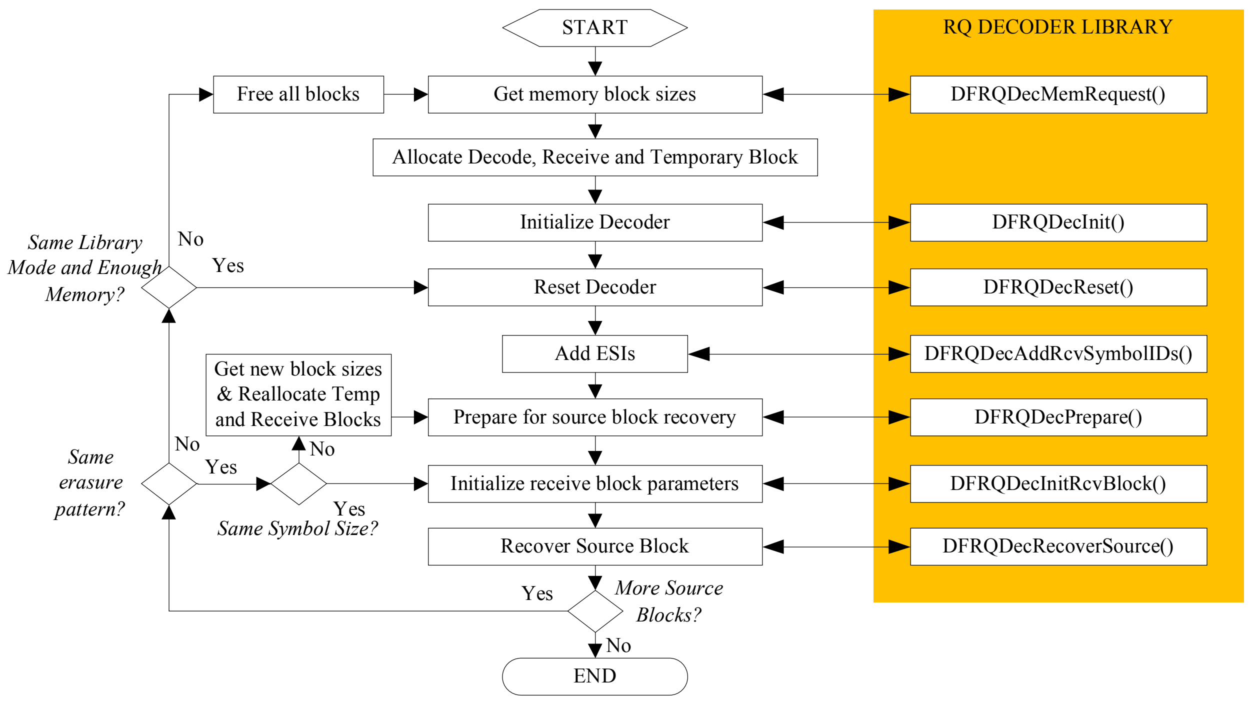


Figure 2.4. Workflow of RQ Decoder Interface Function FileSimpleDecode() (Adopted from *Qualcomm RaptorQ RQ Encoder Developer's Guide*).

It can be used to connect programs written in C and C++ with a variety of high-level programming languages, such as Perl, Python, Ruby, and Tcl. In this section, we'll show the process of wrapping function StringSimpleSend() along with RQ encoder library into dynamic libraries that can be used by Ruby. First we will create our own C library that includes the RQ encoder library, then we make an interface file for SWIG. Finally we will produce desired dynamic library (*.so) with SWIG. The process to wrap the other interface function FileSimpleDecode() is similar. With the help of SWIG, we are able to call the C functions directly from a Ruby script as long as the generated dynamic library is required. The detailed steps are as follows:

Step 1 Create the C library.

Before wrapping the RQ libraries we need to build our own C library first. Since our functions are already designed in Subsection 2.2.1, we list the sample header file as follows:

```c
/* File: StringSimpleSend.h */
#include "../include/RaptorRQEncoderAPI.h"
char *StringSimpleSend(int packetSize, int fileSize, char *inString,
    char *outFileName, int transferPercent);
```

Listing 2.1. Header file of our C library

and Listing 2.2 shows the sample C codes.

```c
/* File: StringSimpleSend.c */
#include "../include/RaptorRQEncoderAPI.h"
#include "StringSimpleSend.h"
char *StringSimpleSend(int packetSize, int fileSize, char *inString,
    char *outFileName, int transferPercent)
{
  ...
}
```

Listing 2.2. Sample codes of our C library

Notice that the RQ Encoder API is included by our own library in this way.

Step 2 Create the interface file.

An interface file as the input to SWIG is also needed. This interface file tells SWIG to create a Ruby module called StringSimpleSend which wraps functions designed in Step 1. All the functions listed in StringSimpleSend.h will be wrapped.

```
/* File: StringSimpleSend.i */
%module StringSimpleSend
%{
  #include "../include/RaptorRQEncoderAPI.h"
  #include "StringSimpleSend.h"
%}
```

Listing 2.3. Interface file to SWIG

Step 3 Wrap the C library.

Now we can use SWIG to wrap our C library so that function StringSimpleSend() can be called directly in Ruby. To run SWIG against the interface file, type:

```
$ swig -ruby StringSimpleSend.i
```

Listing 2.4. Generate wrap file with SWIG

This will generate StringSimpleSend_wrap.c, which can be compiled into a shared library that can be used in Ruby. This step will also create an extconf.rb which configures a Makefile to build the extension. To create the extension:

```
$ ruby extconf.rb
$ make
$ make install
```

Listing 2.5. Commands to generate dynamic library

After a successful make, we should be able to find a file named *StringSimpleSend.so*. This is the dynamic library containing function StringSimpleSend() that can be called by Ruby.

The above process is summarized in Figure 2.5. An example of using function StringSimpleSend() in Ruby to encode a file is as follows:

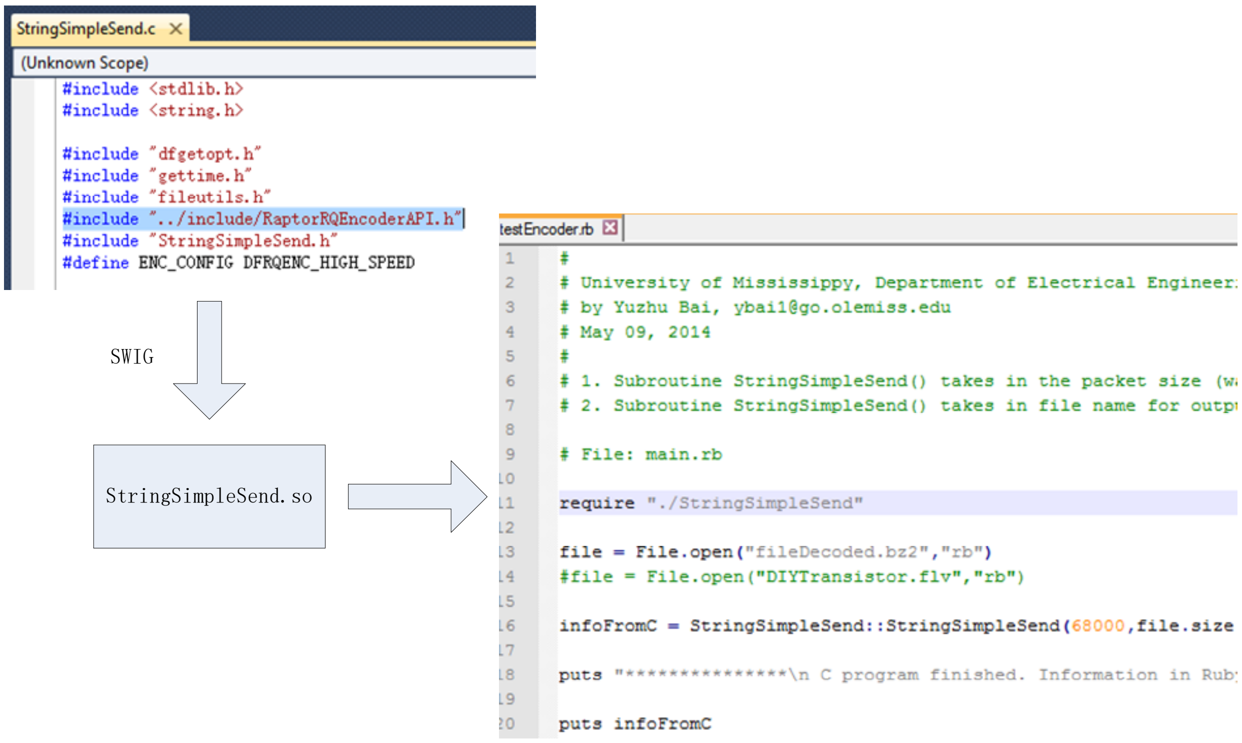
```
# File: Encode_Example.rb
require "./StringSimpleSend"
inFile = File.open("IMG_0325.jpg","rb")
result = StringSimpleSend::StringSimpleSend(6000, inFile.size, inFile.read
    , "Encoded", 200)
```

Listing 2.6. Using the interface in Ruby

In the above example, Ruby passes source data *inFile.read* along with necessary variables to function StringSimpleSend(). All the encoding process is then finished within the C function. The generated source symbols and repair symbols will be written into file *Encoded.src* and *Encoded.rep*, respectively.

2.3 Functional Verification Test of Interface

The first sets of tests we made was to verify the function of our interface. The main idea is to read the source data from a file via Ruby, then pass the source data as a binary string into our interface function to generate source symbols as well as repair symbols. These symbols will be exported in the form of a source symbol file (**.src*) and a repair symbol file (**.rep*). If our interface design was correct, we should be able to recover the source file from either the source symbol file or the repair symbol file via our decoding interface function



```c
#include <stdlib.h>
#include <string.h>

#include "dfgetopt.h"
#include "gettime.h"
#include "fileutils.h"
#include "../include/RaptorRQEncoderAPI.h"
#include "StringSimpleSend.h"
#define ENC_CONFIG DFRQENC_HIGH_SPEED
```

```ruby
#
# University of Mississippy, Department of Electrical Engineer
# by Yuzhu Bai, ybai1@go.olemiss.edu
# May 09, 2014
#
# 1. Subroutine StringSimpleSend() takes in the packet size (w
# 2. Subroutine StringSimpleSend() takes in file name for outp

# File: main.rb

require "./StringSimpleSend"

file = File.open("fileDecoded.bz2","rb")
#file = File.open("DIYTransistor.flv","rb")

infoFromC = StringSimpleSend::StringSimpleSend(68000,file.size

puts "***************\n C program finished. Information in Rub

puts infoFromC
```

Figure 2.5. Wrapping the RaptorQ library with SWIG.

FileSimpleDecode(). Decoding from the source symbol file can be viewed as the best case in a real communication, meaning the receiver end received all the source symbols. Instead, decoding from the repair symbol file is similar to the worst case in real communication, meaning the receiver end didn't receive any source symbol so that all the received symbols are repair symbols. Figure 2.6 illustrates the data flow of encoder interface tests. The test code below gives an example of encoding file IMG_0325.jpg.

```ruby
# File: Encode_Example.rb
require "./StringSimpleSend"
inFile = File.open("IMG_0325.jpg","rb")
result = StringSimpleSend::StringSimpleSend(6000, inFile.size, inFile.read
    , "Encoded", 200)
```

Listing 2.7. Encoding a file in Ruby

In the above test code, Ruby reads source data from file IMG_0325.jpg and pass it into function StringSimpleSend() for RaptorQ encoding. The coding ratio is set to 200%, which means if the source data was divided into k source symbols, another k repair symbols will be generated. After the program finished executing, we get two files IMG_0325.jpg.src and IMG_0325.jgp.rep containing the source symbols and repair symbols, respectively.

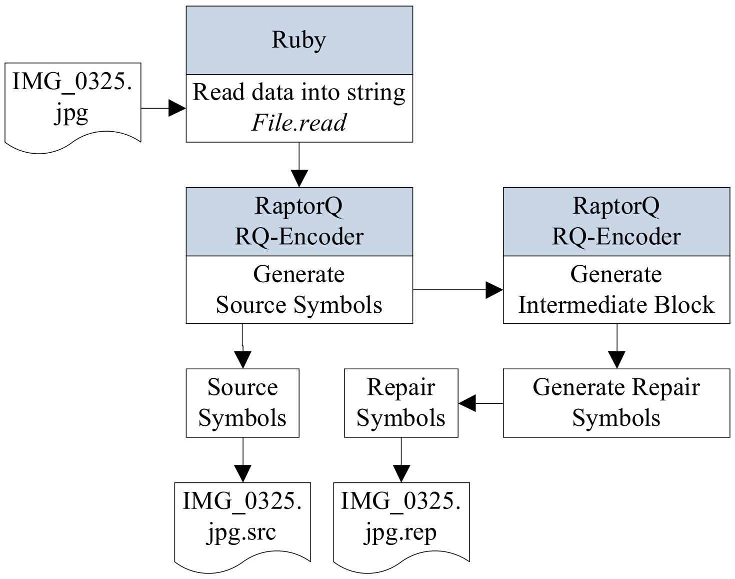


Figure 2.6. RQ Encoder Interface Test.

Figure 2.7 shows the data flow of decoder interface test. In this test example, Ruby passes the repair symbol file IMG_0325.jpg.rep along with several parameters into function FileSimpleDecode(). This function recovers source symbols from the repair symbols and writes them into file *IMG_0325.jpg(Recovered)*. We made a bit-to-bit comparison between the original file *IMG_0325.jpg* and the recovered file *IMG_0325.jpg(Recovered)* and the result turned out to be exactly the same, meaning that both the encoder and decoder interfaces work properly. The test code is as follows:

```
# File: Decode_Example.rb
require "./FileDecode"
status = FileDecode::FileSimpleDecode(5998, "IMG_0325.jpg.rep", "IMG_0325.
    jpg(Recovered)", 5982057, 0, 1)
```

Listing 2.8. Decoding a file in Ruby

A few more tests with different data formats were made afterwards. We tested ASCII files (*.txt), video files (*.flv, *.rmvb, etc), compressed files (*.zip, *.rar, etc). The interface functions were proved to be correctly designed.

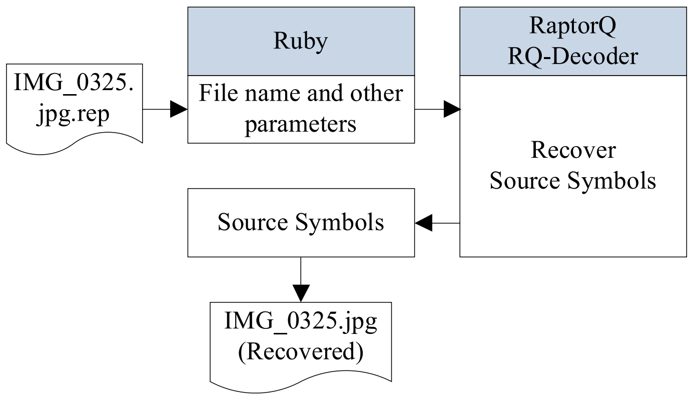


Figure 2.7. RQ Decoder Interface Test.

2.4 Encoding/Decoding Time Test

One of the most significant properties of RaptorQ is linear encoding and decoding time. With the help of the interface designed in Section 2.2, we made a series of tests in Ruby to verify the linearity of the RaptorQ libraries. The testbed is a Raspberry Pi, a single-board computer with ARM1176JZF-S 700 MHz processor and 512 MB RAM.

The first collection of tests were made to test the encoding time of the RQ library. We chose 6 files at the size of 0.14 MB, 2 MB, 6 MB, 12 MB, 18 MB and 24 MB. For each file, we generated 200% repair symbols and recorded the running time. Each test was repeated 100 times and taken average. The pseudo code of our test code is listed as Algorithm 1. The source code can be found in the Appendices. Figure 2.8 shows that the run time of the RQ Encoder Library is linear according to the file size.

Algorithm 1 Encoding Time Test

```
1: for $fileNumber = 1..6$ do
2:     $time = 0$
3:     for $i = 0..99$ do
4:         Encode file with the number of $fileNumber$
5:         $time \leftarrow time +$ encoding time
6:     end for
7:     $runTime(fileNumber) = time/100$
8: end for
```

We also made a comparison between the run time of Ruby and that of the C library. In

Figure 2.8, the blue dashed line presents the run time in C while the red dashed line presents the run time in Ruby. They both appear to be linear according to the file size. Moreover, we found a small gap between the run time in Ruby and the library. In order to determine the cause of this gap, we created a variation of interface function StringSimpleSend(). The variation takes in the name of the source file instead of the source data string. In the comparison of encoding time between Ruby and that of the variation interface function, the gap disappeared. This indicates that the gap between the run time in Ruby and C is caused by passing the data string from Ruby to C.

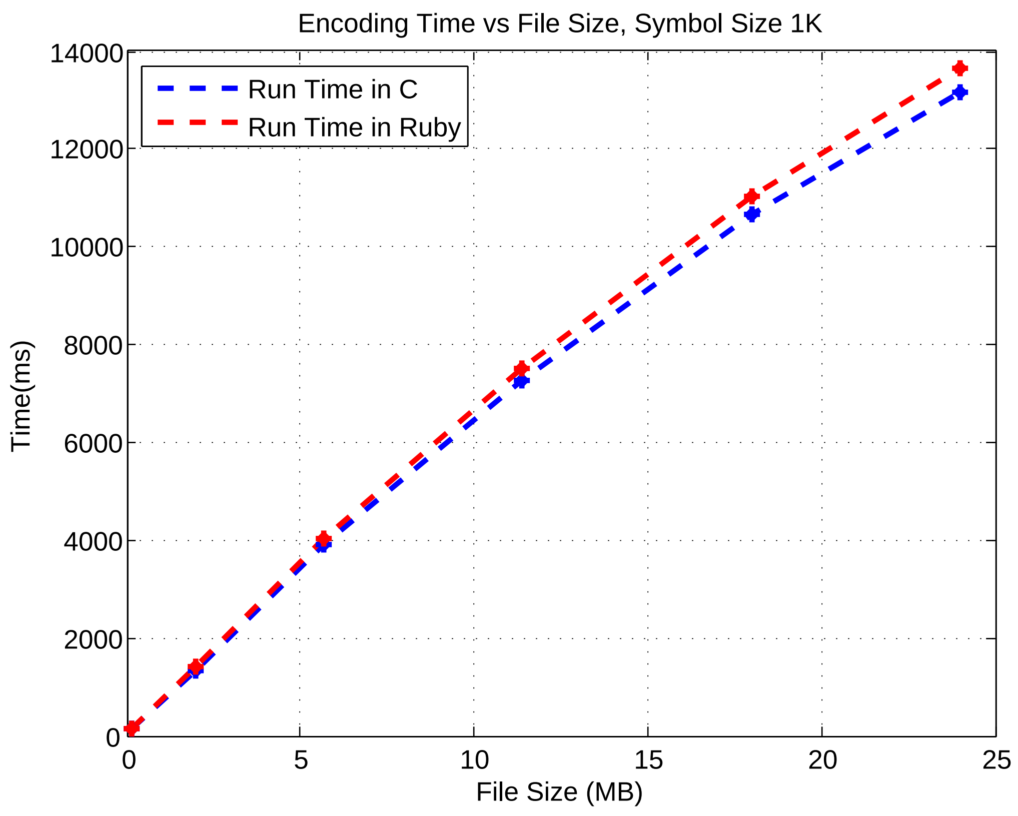


Figure 2.8. Run Time of RQ Encoder on Raspberry Pi.

Another test of the encoding time was made for different symbol size settings. Different symbol sizes also lead to different number of symbols for the same file. For example, a 6 MB file is partitioned into 6,000 symbols when symbol size is set to 1 KB, but is only partitioned into 100 symbols at symbol size of 60 KB. In our test, we chose 5 files at the size of 2 MB, 6 MB, 12 MB, 18 MB and 24 MB. For each file, run time was obtained at the symbol size of 1 KB, 4 KB, 16 KB and 60 KB, respectively. Figure 2.9 shows the run time of RQ Encoder at different symbol size settings. The blue dashed line presents the run time of each file when symbol size is 1 KB. The red dashed line presents the run time of symbol size 4 KB, the green dashed line stands for symbol size of 16 KB and black is 60 KB. We found that there is no significant difference of run time when symbol size varies. In other words, the encoding time of RQ Encoder is dominated by the source data size, and both Figure 2.8 and Figure 2.9 show the linearity of the RQ library's encoding time.

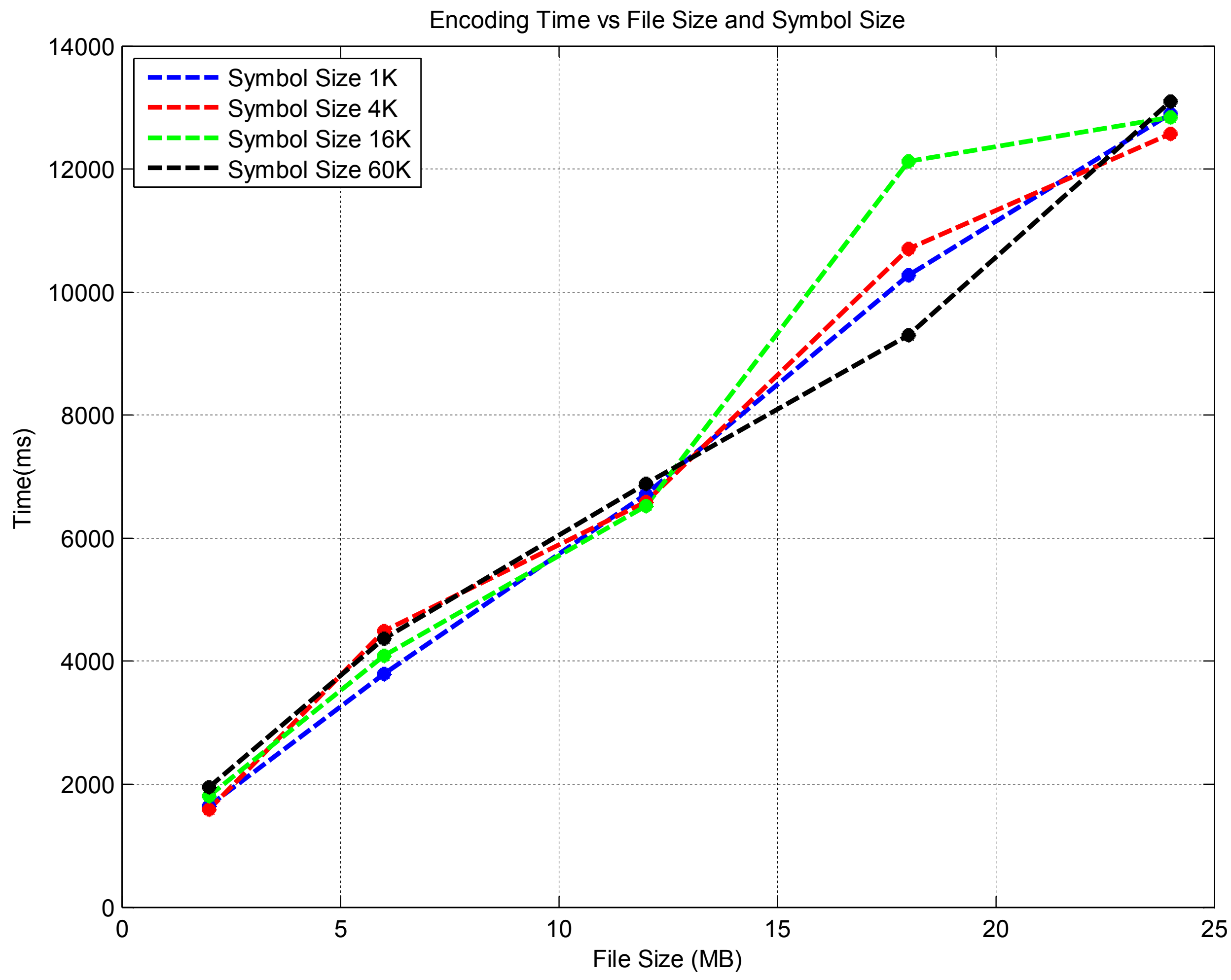


Figure 2.9. Run Time of RQ Encoder at Different Symbol Sizes on Raspberry Pi.

Decoding time of the RQ library was also tested. With the repair symbols generated in the encoding test, we examined the run time of the RQ decoder. Figure 2.10 shows the run time of RQ Decoder at different symbol size settings. The blue dashed line presents the decoding time of each file when symbol size is 1 KB. The red dashed line shows the decoding time of symbol size 4 KB, the green dashed line stands for symbol size of 16 KB and black is 60 KB. We can see that the run time of the RQ decoder is also linear regarding to the file size.

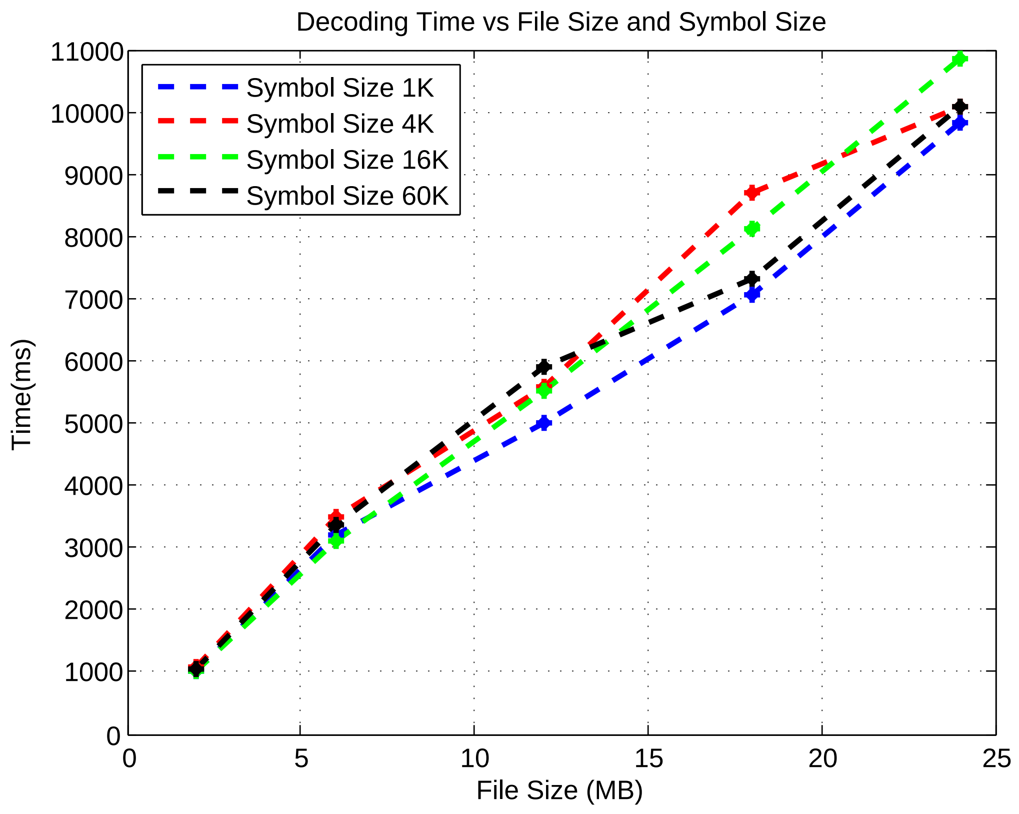


Figure 2.10. Run Time of RQ Decoder on Raspberry Pi.

2.5 Decoding Overhead-Failure Probability Test

According to the specification of the RQ code provided in [3], the RQ code overhead-failure curve mimics that of a random fountain code over GF(256) [2] but is steeper than that of all the other fountain codes. e.g. When the number of source symbols K is set to 200 over a channel with up to 10% packet loss, R10 codes need an extra 24.5% repair symbols to ensure a 10^{-6} failure probability, but RaptorQ only need 12.5% [2].

Theoretically, when the number of symbols received is only slightly larger than the number of source symbols, the source file can be recovered. In [2], the failure probability of RaptorQ code was simulated with disparate overhead and turned out to be better than that of the random fountain over GF(256). More specifically, RaptorQ code with zero overhead (the number of received symbols equals the number of source symbols) has a failure probability no more than 1%, and with one overhead the failure probability is anticipated to be less than 0.003%. Since these results were given by simulations, we decide to test the failure probability with the RQ library in practice.

For the setup of this test, we choose a file at the size of 6.0 MB and set the symbol size to be 60 KB. Thus we have 100 source symbols for this file. Next, we generate 900 repair symbols using the RaptorQ Encoder library so that we have 1000 symbols in total. As shown in Figure 2.11, the ESIs (Encoded Symbol Identifier) are arranged from 0 to 999, where the first 100 symbols with ESIs from 0 to 99 are source symbols, and the rest are repair symbols.

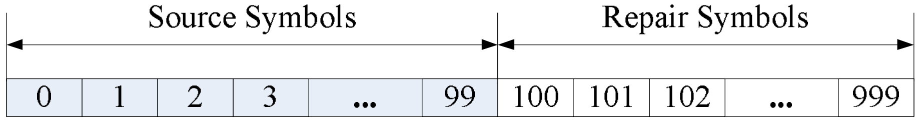


Figure 2.11. Encoded Symbols for Decoding Overhead-Failure Probability Test.

We designed three packet selection strategies to produce different collections of symbols for decoding. First selection mode is *Continuously-Select*. For example, we choose 100 symbols with ESIs from 0 to 99, or from 1 to 100. In this scenario, we can have 900 different combinations with the 1000 symbols. The second mode is *Modulo-Select*, where we select symbols with ESIs that modulo a certain number. For example, if we are doing a *Modulo 5* selection, symbols with ESIs of 0, 5, 15, 20, ..., etc, will be selected. The last mode is *Random-Select* that randomly select symbols from the 1000 symbols. Table 2.3 shows our test results with above three selection modes. Of all the symbol selection modes, the probability of success decoding turns out to be greater than 99.99% when decoding with one overhead symbol.

Table 2.3. Decoding Overhead-Failure Probability Test

Selection Mode	**Overhead**	**Tries**	**Failures**	**Failure Probability**
Continuously Select	0	900	4	0.44%
Continuously Select	1	900	0	0
Modulo 5	0	500	3	0.6%
Modulo 4	0	600	5	0.83%
Modulo 3	0	700	7	1%
Modulo 2	0	800	3	0.375%
Random Select	0	1000	2	0.2%
Random Select	1	100000	1	0.001%

CHAPTER 3

RAPTORQP2P

This chapter reviews the RaptorQP2P protocol. The RaptorQP2P protocol was designed by Z. Su, F. Wang and J. Daigle [1] in 2014. Different from BitTorrent, which was the most popular peer-to-peer protocol, RaptorQP2P utilizes a two level RaptorQ encoding and an intelligent symbol scheduling algorithm to overcome some shortages of the BitTorrent protocol. To illustrate the protocol clearly, we first briefly introduce terminology and then give a detailed review on the RaptorQP2P protocol.

3.1 RaptorQP2P Terminology

The downloading of RaptorQP2P is initiated from a *torrent file.* The torrent file provides information of the object file and the IP address of the *tracker.*

The *tracker* is a server that provides the information of all the *peers* currently downloading/uploading the file. Each peer contacts the tracker before joining the network. The tracker returns a list of current peers and puts that peer into the *peer list.*

Peers are the users who participate in distributing a file. The peers may be divided into two types. One is the *leecher*, which does not have the whole file yet. The other is *seeder*, which has finished downloading but stays in the network to further help the file distribution. Only leachers will download the file content from other peers but both leachers and seeders can upload the file content to other peers. Each peer will maintain several peers as its *neighbors.* A *neighbor* is a peer that has exchanged a *piecemap* with current peer.

Piecemap is a data structure that keeps the information of the *pieces* that the peer has already downloaded. It indicates the downloading progress of this peer. In RaptorQP2P, piece sizes can be selected. For the comparison to BitTorrent, we divide the original file into *pieces* in the length of 1600 KB. Each piece is further divided into 100 *source symbols*. A *symbol* is the smallest unit of transmission in RaptorQP2P.

A *swarm* is the set of all the peers that participate in the file distribution.

If peer A does not have a certain piece but peer B has, then we say peer A is *interested* in peer B.

If peer A decides not to send data to peer B, peer A *chokes* peer B. If peer A decides to send data to peer B, peer A *unchokes* peer B.

After a peer receives the peer list from tracker, it tries to establish connections with the peers in the peer list. When a connection is established, the two peers exchange piecemaps with each other. A peer considers the collection of peers with whom it has exchanged piecemaps its *potential neighbors*. Among these potential neighbors, a peer uploads symbols to at most five of them. These five peers are *neighbors* of this peer.

3.2 RaptorQP2P Protocol Review

The RaptorQP2P protocol features two levels of RaptorQ encoding. Figure 3.1 shows a demonstration of the two level RaptorQ encoding. At the top layer, the entire file is RaptorQ encoded to yield a collection of source blocks and repair blocks, and then each source and repair block is RaptorQ encoded independently to yield a collection of source symbols and repair symbols for the block. The symbols are independently transferred among the peers and when a sufficient number of distinct symbols for a particular block have been received, whether source or repair, the block can be reconstructed. The file can be reconstructed using a sufficient arbitrary number of distinct blocks.

The mechanism of RaptorQP2P is as follows: a leecher processes the .torrent file to get necessary information, including the IP address of the tracker. Then the leecher will contact the tracker to obtain a list of peers currently in the swarm. Next, the leecher selects a number of peers from the list and tries to connect to them. When peers are connected

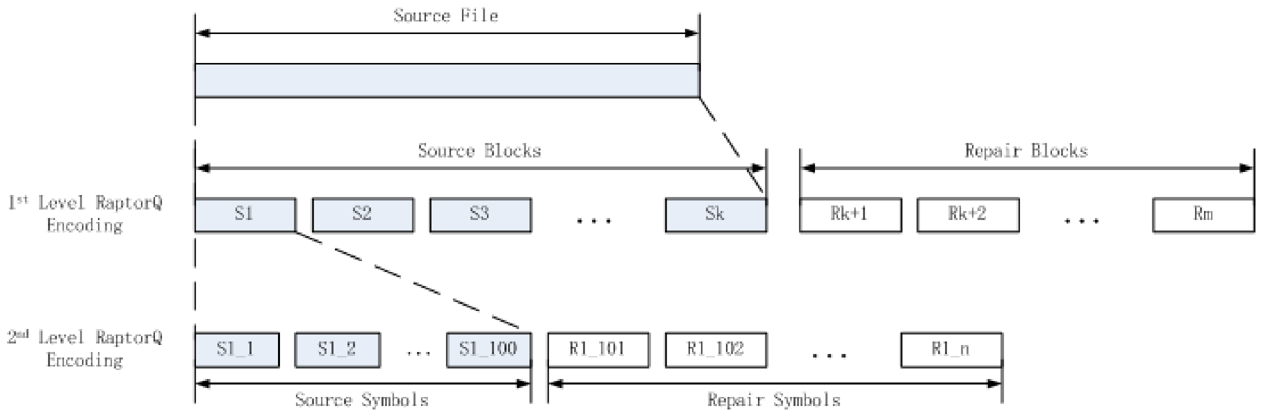


Figure 3.1. Two Level RaptorQ Encoding.

to each other, they will exchange their piecemaps to determine whether or not they have interest so that peer can request missing pieces from others.

By utilizing RaptorQ codes, the RaptorQP2P protocol allows multiple peers to send the same piece to a peer simultaneously, which can be automatically optimized by the RaptorQ coding as discussed in Chapter 1. Moreover, to maximize the utilization of peers' upload capacities, RaptorQP2P also allows opportunistic transmissions, where a peer can send the received encoded symbols of a piece to other peers even if the peer does not have the full piece yet. These two properties helps the RaptorQ to surpass BitTorrent under most dynamic situations.

In order to ensure that different neighbors generate and send different encoded symbols so as to avoid duplications, RaptorQP2P introduced an intelligent symbol scheduling algorithm. In this algorithm, when a peer connects to a new neighbor, it will assign an empty neighbor slot to the neighbor and inform the neighbor its neighbor slot number during the initial piecemap exchange. If the neighbor has a full piece that the peer does not have, the neighbor will only send those symbols encoded from that piece whose symbol numbers must be equal to the neighbor's slot number after taking mod on the number of the peer's neighbor slots. By doing this can guarantee that the encoded symbols received from one neighbor are different from those received from another neighbor. In addition, to deal with peer churns, when requesting a piece from a neighbor, a peer includes the maximum symbol number that it has received so far for the piece, so that if a neighbor leaves after sending some encoded symbols, the new neighbor can continue sending later symbols starting from the maximum symbol number indicated during the piece request.

CHAPTER 4

IMPLEMENTATION OF RAPTORQP2P

This chapter discusses the implementation of RaptorQP2P in detail. The program is developed in Ruby. Section 4.1 discusses the essential features required in the implementation. Section 4.2 introduces our architectural design for the implementation. Section 4.3 introduces some internal protocols. Section 4.4 discusses our implementation in detail.

4.1 Implementation Requirement

As an implementation of the RaptorQP2P, our program is able to distribute a file to a number of peers over the internet. Below are the basic features required in our implementation.

- A peer processes a torrent file to get necessary information. Since the downloading is initiated from a torrent file, a peer should be able to achieve all the information needed to start downloading from the torrent file. The torrent file contains the IP address of the tracker, the number of blocks, number of symbols in a block and symbol size.

- A tracker keeps the IP addresses of all the peers in the swarm. When a peer joins the swarm, the tracker acknowledges the IP address of this peer, puts the IP address in the peer list, and monitors the activeness of this peer. A peer needs to send the tracker a signal every few seconds to keep "alive". If a tracker does not receive the "alive" signal from a peer for a certain time, this peer will be eliminated from the peer list.

- The source file is applied a 2-level RaptorQ encoding. The first-level encoding partitions the source file into *blocks* and generate some repair blocks. The second-level encoding is applied to each block independently producing source symbols and repair symbols. Symbols are the smallest unit of transmission in RaptorQP2P.

- Download is initiated by a single peer. Though there might be multiple seeders in practical downloading, RaptorQP2P has the ability to initiate and accomplish a file distribution from a single seeder.

- Peers can join and leave the swarm dynamically. The status of the whole swarm is monitored by the tracker.

- Peers in swarm download data from each other. This is a basic feature of any peer-to-peer protocol. A peer is able to download data from and upload data to other peers in a swarm. Moreover, a peer is also able to refuse to upload data to a certain peer in the swarm, known as *choking* that peer.

- Download at any given peer is completed when a sufficient number of distinct blocks have been received. Because of the fountain property of RaptorQ, a peer is able to recover a block when it collected enough distinct symbols for that block. Next, when the peer possesses enough blocks, it can recover the whole source file and complete the download. This peer will then be able to generate any symbol of any block. If this peer keeps staying in the swarm, it will become a seeder.

- Multiple peers can send symbols from the same piece to a peer simultaneously. This property is supported by utilizing RaptorQ codes.

- A peer can send the received symbols of a piece to other peers even before the peer has the full piece.

- Duplicate reception of pieces is avoided by using the symbol scheduling algorithm of RaptorQP2P. In the RaptorQP2P, a peer assigns distinct slot numbers to its neighbors, and the neighbors send certain symbols according to that slot number.

We use Ruby for the implementation because it has good interface for socket programming and we have already used it successfully for experiment with RaptorQ libraries.

4.2 Systematic Design

Based on the analysis in Section 4.1, we are able to delineate the logic model of our program. We will discuss the systematic design for the implementation in this section, including the program workflow, program organization and multi-threading.

4.2.1 Program Workflow

The workflow of our program can be easily summarized from Section 4.1. The flowchart can be as simple as Figure 4.1. In this flowchart, the program reads a torrent file to get the IP address of the tracker and some necessary information of the object file, e.g. file name, file size, number of pieces, etc. Then the peer connects to the tracker to get the IP address and port numbers of other peers and then starts downloading or uploading.

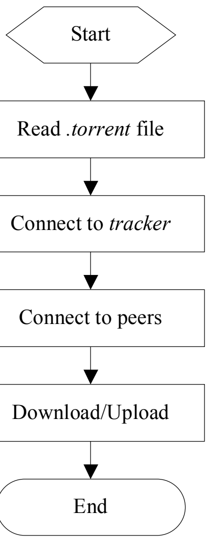


Figure 4.1. Simplified Flowchart of RaptorQP2P.

Figure 4.1 is simple and seems easy to be implemented. However, when we get to the detailed design, things become complicated. We want the program to be able to accept connection requests from other peers even when itself is trying to connect to a peer; we want the program to upload pieces when itself is downloading; we want a peer to choke the neighbor with lowest upload rate and maximize its own download rate. In additional, we want the program be able to do all the above tasks simultaneously. Considering all the above factors, we introduce a multi-threading into our implementation.

4.2.2 Multi-threading

As discussed in Section 4.1 and Section 4.2.1, our implementation is expected to be able to perform (at least) the following 7 tasks: communicate with the tracker, connect to other peers, accept connection requests from other peers, download pieces, upload pieces, encode and decode pieces, and management related tasks. We re-organize these tasks as follows:

- Thread 1 runs as a server. This thread accepts connection requests from other peers and returns objects of the sockets.
- Thread 2 connects to other peers. The input of this thread is a list of peers (IP addresses) to connect to, and output is objects of the sockets to the peers.
- Thread 3 sends data in the outgoing buffer.
- Thread 4 receives data from the sockets with high priority and writes them into the incoming buffer.
- Thread 5 performs all the management related tasks.

Thus we have 5 threads for different tasks, as shown in Figure 4.2. Thread 1 runs as a server to accept the connection requests from other peers. Thread 2 works as a client to connect to other peers. Both threads 1 and 2 return an object that contains the socket identification of the corresponding peer. Thread 3 is used for sending data (pieces) and thread 4 is to receive data from neighbors. Thread 5 manages the piece map status, updates the peer list, prepare the pieces, and decides whether to choke a peer or not.

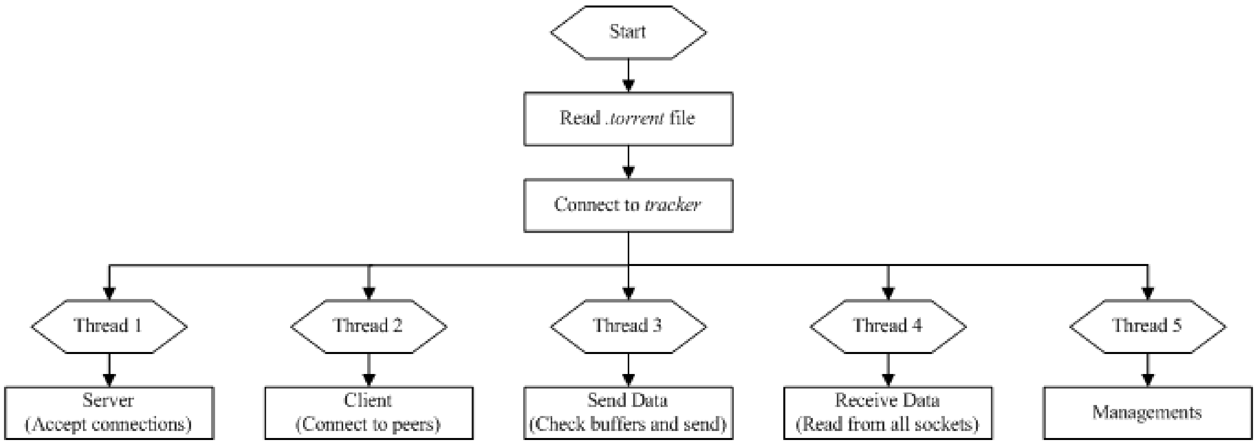

Start
Read .torrent file
Connect to tracker
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Server (Accept connections)
Client (Connect to peers)
Send Data (Check buffers and send)
Receive Data (Read from all sockets)
Managements

Figure 4.2. Multi-threading The Implementation.

4.2.3 Program Organization

Now we have the basic workflow and thread arrangement for the implementation. Considering the task similarity and co-operation between the threads, we organize the program as 4 modules. Each module accomplishes a task required by the features of protocol. We also introduced a standalone program, the tracker, to maintain the information of the swarm. Figure 4.3 presents the RaptorQP2P system structure. In the first level, there are four modules: *torrent file processing*, *communication with tracker*, *membership management* and *file transmission.* The *file transmission* module can be further divided into four parts as *piecemap management*, *neighbor management*, *strategy management* and *data transmission.* Specifically, the *strategy management* consists of *piece selection* and *choke/unchoke strategy.* We discuss the details (function, input and output) of these modules as follows:

Torrent File Processing This module initiates the downloading/uploading. It is the first module to be called in the program. The input of this module is the torrent file. It reads this torrent file to get essential information for downloading. The output of this module is the information from the torrent file, such as the IP address of the tracker, file name, file size, number of pieces, etc.

Communication With Tracker This module is called after module *Torrent File Process-*

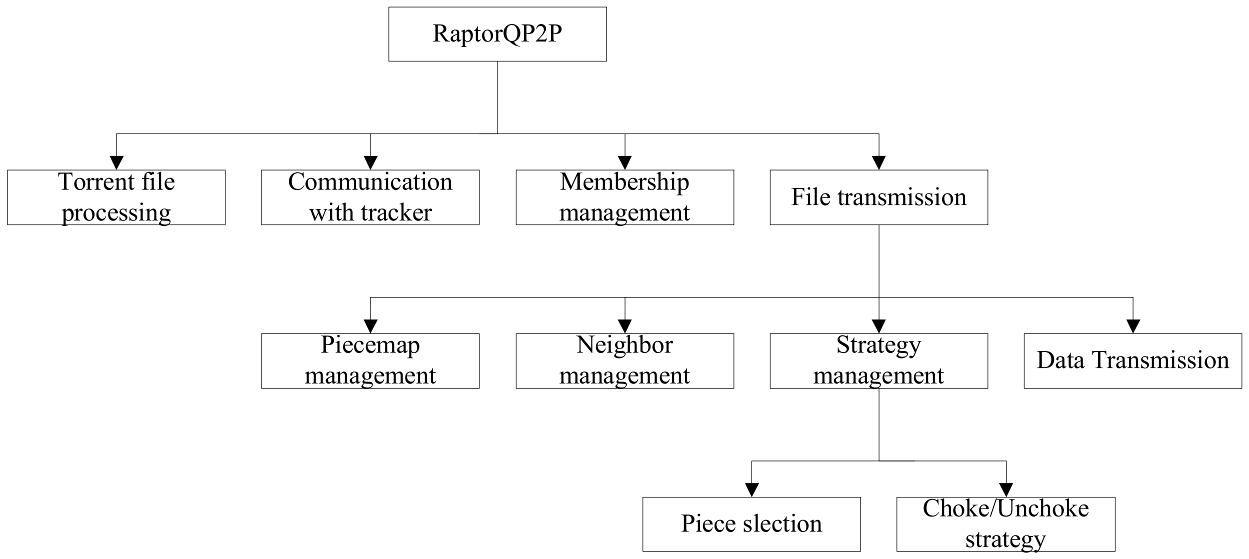


Figure 4.3. System Module Structure of RaptorQP2P.

ing. The first task of this module is to connect to the tracker and fetch a peer list of the swarm, then this module keeps sending a *live* signal to the tracker every 10 seconds. The input of this module is the IP address of the tracker and the output is an array of the peers (in the swarm).

Potential Neighbor Management Module This module simply tries to make connection with all the peers in the peer list (from module *Communication with tracker*). It sends a connection request to every peer and wait for connection acknowledgement. The input of this module is an array of IP addresses (of the peers in the swarm), and the output is TCP sockets of the peers.

File Transmission Module This core module fulfills the data transmission (download and upload) in RaptorQP2P. It keeps track of the piecemap status of the file being downloaded, decides which peer becomes a neighbor, performs piece selection and choke/unchoke decision and finally send/receive data from other peers.

The *File Transmission Module* is the very core and most complicate one in the system. We divide it into following 4 sub-modules:

Piecemap Management This sub-module manages the piecemap of object file. At the very beginning, it examines the buffer memory of downloaded pieces. During the runtime of the program, this module keeps track of every downloaded piece and updates the piecemap. The piecemap is a data structure indicating which pieces are already downloaded.

Neighbor Management Any peer in the swarm that has established a socket connection with local peer is considered as a potential neighbor. A peer often keeps a large set of potential neighbors, but only simultaneously upload/download to/from a small subset of them, which are called active neighbors [10]. Neighbor management not only determines which peers (from the peer list) are to become potential neighbors, but also chooses active neighbors from the potential neighbors. This sub-module operates on an array of peers, and the membership management module makes socket connections accordingly.

Strategy Management There are basically two strategies in the RaptorQP2P. One is the tit-for-tat strategy, which chokes an active neighbor with lowest transimission rate. The other strategy is the intelligent symbol scheduling, which allocate slot number to active neighbors and request symbols accordingly.

Data Transmission This sub-module takes care of the data transmission part. For download, it periodically check and read from the sockets; for upload, it writes the data from the buffer to destination sockets.

4.2.4 Internal Communication

This subsection discusses the internal communication of RaptorQP2P. The input and output of each module is shown in Figure 4.4. The red lines indicates the data direction with arrows, and the comments beside red lines summarize the data content. We can see the initiate input is a torrent file to the torrent file processing module. The torrent file processing module outputs the tracker IP address to communication with tracker, and other information to piecemap management module. The output of the piecemap management is piecemap, and the output of Tracker Communication module is peer list. Both of these two outputs go to the membership management module. The membership management module then returns member list, piecemap, IP address and port number to the neighbor management module. Neighbor management module outputs the piecemap to piece selection module. After piece selection module makes decision, it tells the data transmission module to request and receive symbols from other peers. The choke/unchoke module monitors the transmission rate, decides which peers to choke or unchoke, and sends the decision to the neighbor management module.

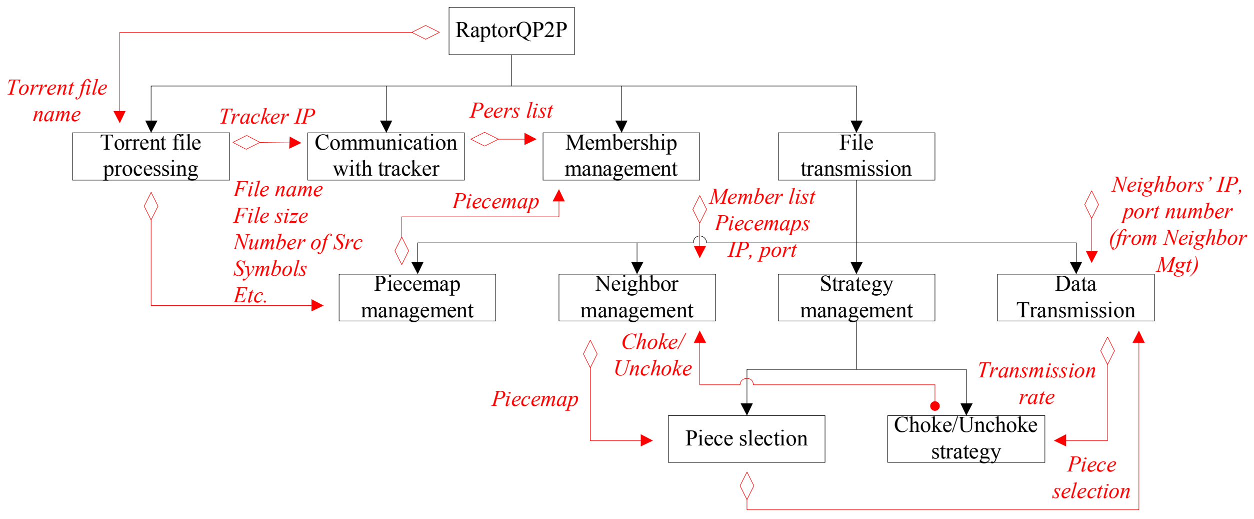


Figure 4.4. Internal Communication.

4.3 Internal Protocols Design

In the previous section, we divided the program into 4 modules and discussed the input and output of each module. Now we consider the data structure and communication protocol between peers. This section discusses the internal communication protocols and data structures that are vital to the implementation.

- Packet Format:

The basic communication unit between peers is a packet. All the packets transferred between peers must be of the same format. Herein we define a packet format that consists of three parts: data type part, length part, and payload. As shown in Figure 4.5, the data type part is 3 Bytes and the data length part is 8 Bytes, the rest are for the payload. The data type part specifies the type of this packet. For example, DAT means this packet contains a data symbol, REQ means this packet is a request message. Data length part points out the length of the payload in this packet.

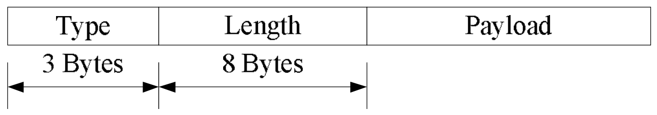


Figure 4.5. Packet Format.

- Piecemap Format:

In the traditional BitTorrent protocol, piecemap uses 1 bit for each piece to indicate whether this piece is possessed or not. But in RaptorQP2P, it is far from enough just know the piece level. In the intelligent symbol scheduling algorithm of RaptorQP2P, a peer downloads symbols with respect to their symbol IDs (also know as ESI, Encoded Symbol ID). This drives us to design an appropriate data structure for the piecemap which could efficiently record which and how many symbols are currently received. Figure 4.6 is the piecemap data structure designed by us. In this structure, every contiguous 6 Bytes indicates the status of a single piece. Of these 6 Bytes, the first byte has values among 0, 1 and 2. 0 means no symbols of this piece has been received yet, and 1 means the whole piece has been received. The first byte equals 2 means part of this piece was received. The following 5 bytes indicate how many symbols with that particular ESI was received. For example, $2, 30, 55, 0, 0, 0$ means for this particular piece, 30 symbols with ESI modulo 5 and 55 symbols with ESI modulo 5 remaining 1 has been received.

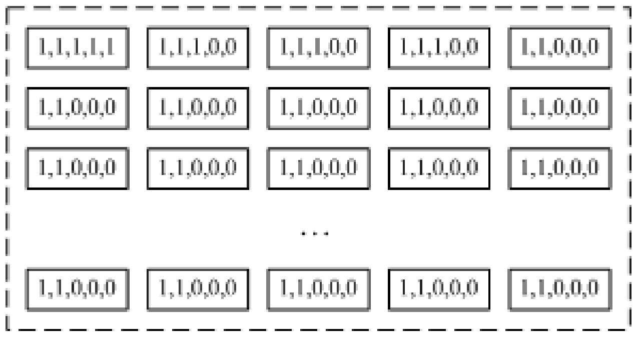


Figure 4.6. Piecemap Format.

- Request Packet Format:

  RaptorQP2P operates on the symbol-level requesting, so we need to design the request packet format. Figure 4.7 gives an example of a request packet. The first 11 bytes are for the header. In the payload part, the request for a piece is fulfilled by 3 contiguous bytes. The first byte tells which piece, the second byte tells which slot, and the third byte tells which symbol to begin.

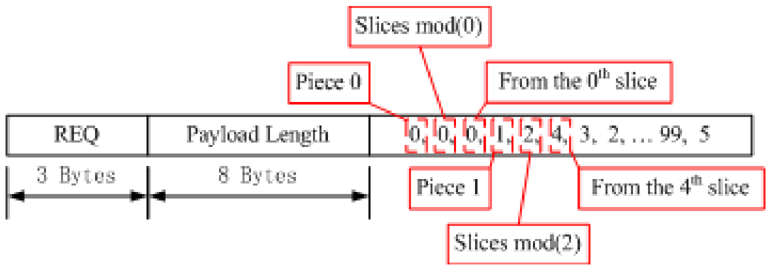


Figure 4.7. Piece Request Format.

4.4 Implementation of Modules

4.4.1 Torrent File Processing Module

The Torrent File Processing is realized by function read_torrent_file(). This function takes in the torrent file name and returns 5 variables: tracker IP, file name, file size, symbols size and number of source symbols.

4.4.2 Server Module

The server module acts as a server accepting connection requests from other peers. It utilizes the function listen_to_peers() to fulfill this task. The usage of the server module is listed below. Function listen_to_peers() is called in the new started thread *listen*. There is a loop in this function listening to port 4481. If the thread receives a connection request, the request will be accepted immediately and an object of the TCP socket is returned.

```
listen = Thread.new {
  server = TCPServer.new(4481)
  listen_to_peers(server, members, lock, local_ip)
}
```

Listing 4.1. Ruby peseudo code for server module

4.4.3 Connect Module

The connect module acts as a client requesting connections requests to other peers. It utilizes the function connect_to_peers() to fulfill this task. The usage of the connect module is listed below:

```
connect = Thread.new {
  connect_to_peers(peerList, members, lock, local_ip)
}
```

Listing 4.2. Ruby peseudo code for connect module

4.4.4 Send Module

The send module checks the send buffer of each peer in a loop. The buffer is a queue structure. If there is data to be sent, this module writes the first 1024 bytes of the buffer into corresponding socket. These data will then be removed from the buffer to avoid duplication. The Ruby codes in Listing 4.3 fulfill the function of a send module. We create a new thread (line 1) to repeatedly check the data length of the send buffer of each peer (lines 3-13). If the length of the buffer is greater than 0, a packet will be written into the corresponding socket (line 7). This packet will then be removed from the buffer (line 9).

```
1 send = Thread.new {
2   loop do
```

```
    for i in 0..(neighbors.peers.length-1)
      if neighbors.peers[i].sendBuf.length > 0
        lock.synchronize {
          # Send one packet each time.
          neighbors.peers[i].socket.write(neighbors.peers[i].sendBuf
              [0..(10+pLength)])
          # Remove processed data from buffer
          neighbors.peers[i].sendBuf = neighbors.peers[i].sendBuf[(11+
              neighbors.peers[i].sendBuf[3..10].to_i)..-1]
        }
      else
        puts "No data to send to socket #{neighbors.peers[i].ip}"
      end
    end
  end
}
```

Listing 4.3. Ruby peseudo code for send module

4.4.5 Receive Module

The receive module checks the socket of each peer in a loop. If there were data coming, this module reads 1024 bytes from the socket and hands the data to the message process module. The Ruby codes in Listing 4.4 fulfill the function of a receive module. As Listing 4.4 shows, the loop (lines 2-12) in the receiving thread repeatedly checks the status of each socket. If there were data in the socket (line 4), the module reads 1024 bytes and write them into the corresponding buffer (line 6).

```
 1  receive = Thread.new {
 2    loop do
 3      for i in 0..(neighbors.peers.length-1)
 4        if neighbors.peers[i].socket.ready?
 5          lock.synchronize {
 6            neighbors.peers[i].rcvBuf << (neighbors.peers[i].socket.
                  readpartial(1024) rescue nil)
 7          }
 8        else
 9          puts "No data from socket #{neighbors.peers[i].ip}"
10        end
11      end
12    end
13  }
```

Listing 4.4. Ruby peseudo code for receive module

4.4.6 Message Process Module

The message process module checks the receiving buffer of each peer periodically. Whenever there is something in the buffer, these data are processed as Figure 4.8 shows. There is a public buffer shared by both the message process module and the receive module. The structure of this buffer is a queue. Each time the receive module writes the received packet into this buffer, the received packet is always added to the tail of the queue. In contrast to that, the message process module always takes packets from the head of the queue. The message process module is able to find the length of each packet from the first 11 bytes of this packet. So when the buffer length is less than the packet length, the message process module will skip current cycle.

```ruby
msgproc = Thread.new {
  loop do
    if peerObj.rcvBuf.length > 0
      if packet_process(peerObj.rcvBuf, fileName, peerObj, lock) == 0
        lock.synchronize {
          # Remove processed data from buf
          peerObj.rcvBuf = peerObj.rcvBuf[11+peerObj.rcvBuf[3..10].to_i
              ..-1]
        }
      end
    else
      puts "No data from socket #{peerObj.ip}"
    end
  end
}
```

Listing 4.5. Ruby peseudo code for message process module

4.4.7 Piece Request Module

A peer analyzes the piece maps received from its K neighbors and allocates each of its neighbors a slot number. The slot numbers are assigned 0 to K-1. If the peer requests a piece that the neighbor has, the neighbor will only send the symbols whose ID number mod K is equal to the slot number.

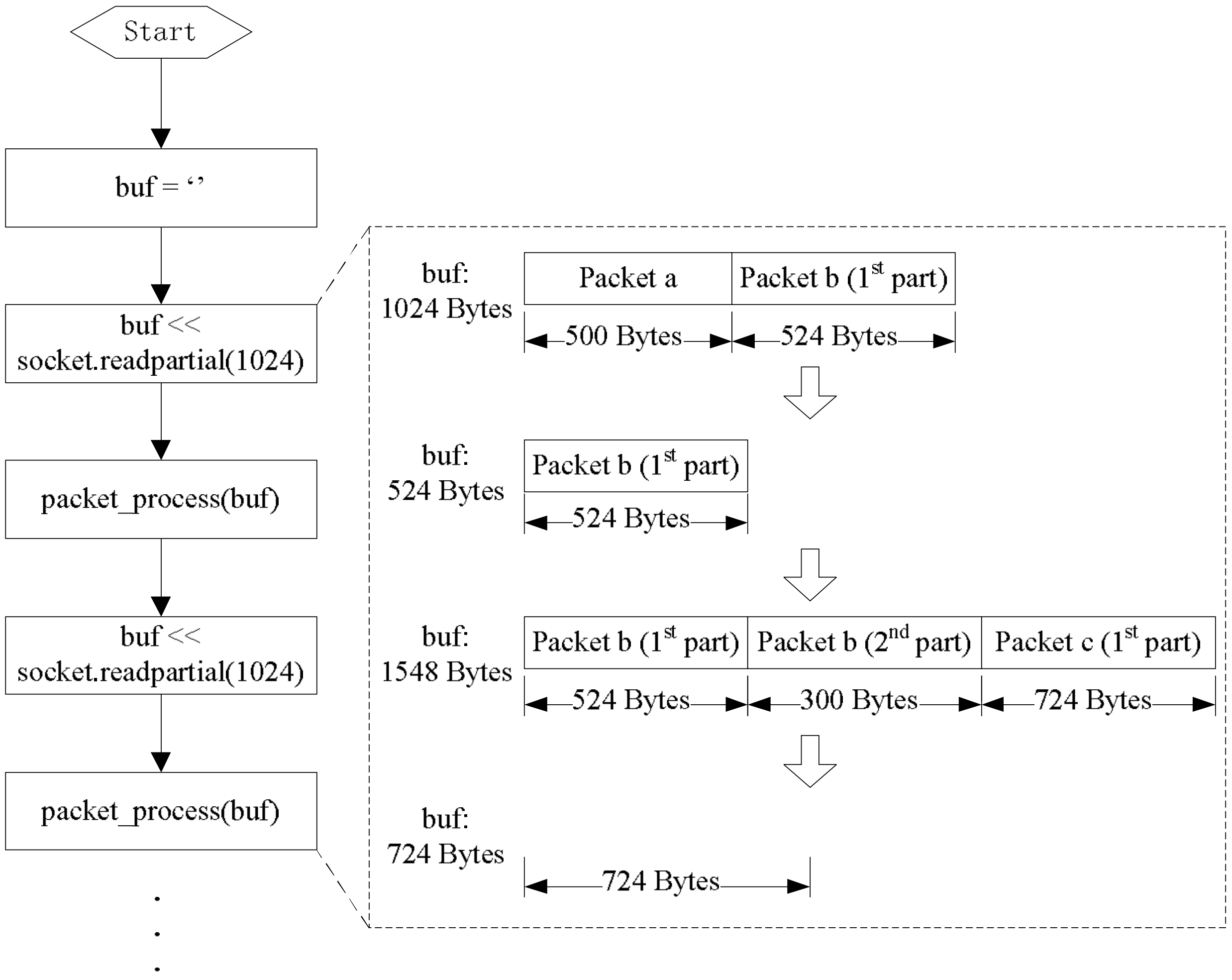


Figure 4.8. Message process.

4.5 Summary

In this chapter, we provided a detailed introduction to our Ruby implementation of RaptorQP2P, including required features, program workflow, multi-threading, and program organization. We organized the program as 4 main modules and 4 sub-modules. We also discussed the internal communication and internal protocols design. We explained the module design at code level. All the source codes can be found in appendices. In the next chapter, we will introduce some tests for our implementation.

CHAPTER 5

TEST RESULTS OF RAPTORQP2P

We made a series of tests for our implementation of the RaptorQP2P. This chapter introduces these tests in detail. In Section 5.1, we describe the testbed setup. In Section 5.2, we tested communication modules of our implementation. In Section 5.3, we discussed scheduled transmission test, which validates our intelligent symbol scheduling design. Finally, Section 5.4 introduces our multi-peer test, which is the most close to practice test.

5.1 Testbed Setup

Our testbed is Raspberry Pi, a sigle-board computer with ARM1176JZF-S 700 MHz processor and 512 MB RAM [11]. Figure 5.1 shows the Raspberry Pi 1 model B+ released in February 2012. We choose the Raspbian operation system, which is a free operating system based on Debian [12]. During our test, we need to operate on up to 5 Raspberry Pis at one time. We use VNC (Virtual Network Computing) to remotely control the Raspberry Pis from another computer [13].

5.2 Communication Test

The first test is made to validate the server module, connect module and message process module of our program. We set up two peers, peer A and peer B. As Figure 5.2 shows, a tracker and the two peers are connected to the same network. The test takes place as follows:

In the beginning, there is only a tracker online until Peer A joins. Peer A gets the Tracker's IP address from a torrent file, so that it is able to connect to the tracker and get the peer list. However, the peer list from the tracker is null because there was no other

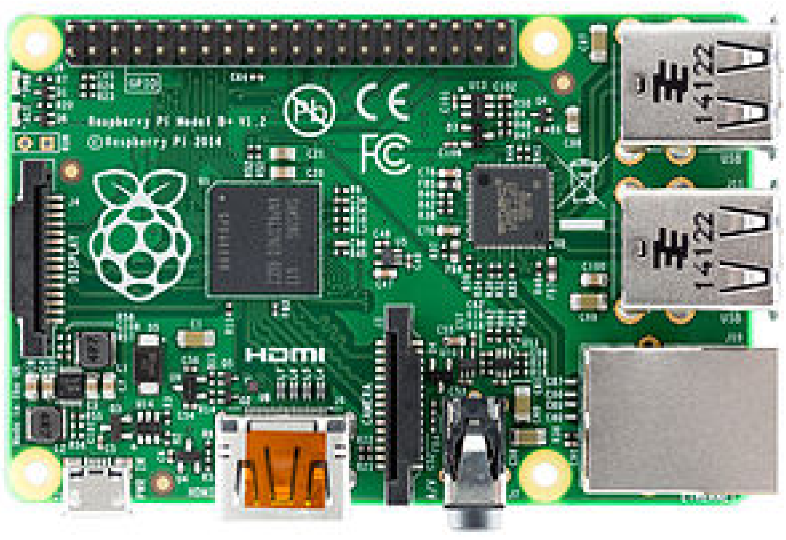


Figure 5.1. Raspberry Pi (Adopted from *Wikipedia*).

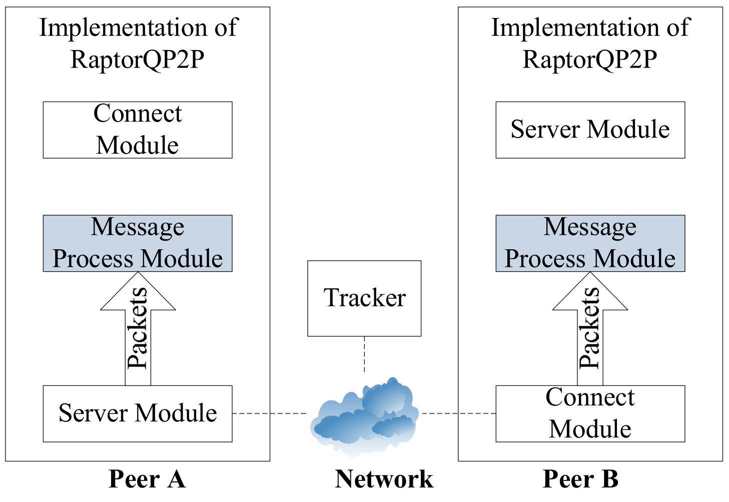


Figure 5.2. Communication Test.

peer in the swarm at that time. But after Peer A's communication with tracker, the tracker records the IP address of Peer A and puts it into the peer list. Peer A now stays in the swarm as a server because its peer list is empty, making Peer A has no peer to connect to. Then Peer B joins the network. Peer B also gets the Tracker's IP address from a torrent file. After the communication between Peer B and the Tracker, Peer B acquires a peer list containing Peer A's IP address. Thus the next move of Peer B is to send a connection request to Peer A, according to the IP address from the Tracker. Peer A will accept Peer B's request and a TCP socket communication is established. In our test, Peer B will send a packet containing its piece map. When Peer A receives the packet, it passes it to the message process module. The message Process Module will find out that a piece map is within the packet. It will update Peer B's piece map and send its own piece map to Peer B in return. Peer B also updates its information of Peer A with the help of the Message Process Module. Communication between Peer A and Peer B is now established, and they have each other's piece map information.

5.3 Scheduled Transmission Test

The scheduled transmission test was made for functional verification of our implementation. This test examines the slot assignment algorithm and symbol-level transmission design. We introduced a time scheduling method to control the time slot that each peer joins the swarm. In this test, we omitted the torrent file part.

The time scheduling for this test is designed as Figure 5.3 shows. In time slot 0, there is a Seeder in the swarm. This seeder has all the pieces of the object file. At time slot 1, Leecher A joins the swarm and requires symbols from Seeder. Since the Seeder is the first neighbor of Leecher A and has all the symbols, it is assigned slot number 0 by Leecher A. Thus the Seeder uploads symbols with ESIs that modulo 5 to Leecher A. We call these symbols 0-symbols for short. In time slot 2, Leecher B joins the swarm. At this time, Leecher A already has some 0-symbols of piece 0, so that Leecher B can require piece

0 from both Leecher A and Seeder. Leecher B finds out that it can acquire any symbols (of piece 0) from Seeder but only can acquire 0-symbols from Leecher A, so Leecher B assigns slot number 0 to Leecher A and slot number 1 to Seeder. With slot number 0 from Leecher B, Leecher A uploads the 0-symbols it currently owns to Leecher B. On the other side, the Seeder sends the symbols with ESIs that divided by 5 with remainder of 1 to Leecher B. We call these symbols the 1-symbols (Similarly, we have 2-symbols whose ESIs divided by 5 with the remainder of 2). In this way, Leecher B downloads symbols for piece 0 from both Seeder and Leecher A simultaneously. The same thing happens if another peer Leecher C joins the swarm after time slot 2, which was not shown in Figure 5.3. Leecher C will find that it could download 0-symbols from Leecher A, 1-symbols from Leecher B and 2-symbols from Seeder, simultaneously.

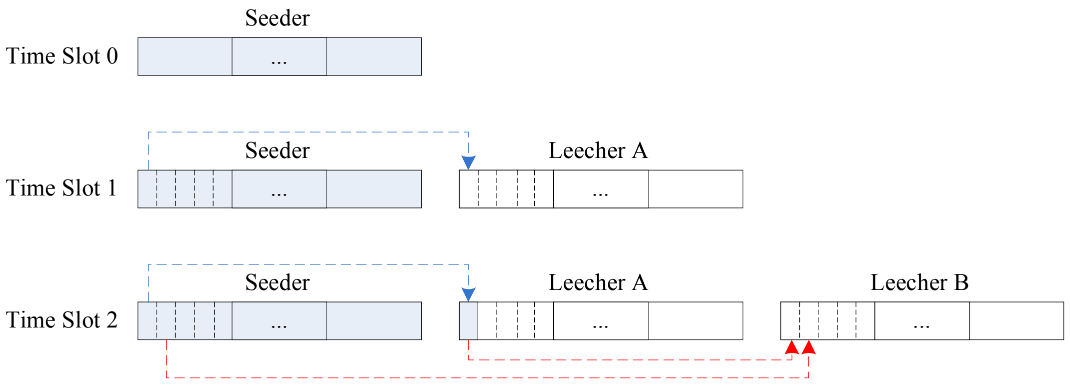


Figure 5.3. Scheduled Transmission Test.

We use the *Time.parse* and *sleep* method in Ruby to achieve time synchronization and scheduling. Before Leecher A and Leecher B join the swarm, they receive the Seeder's time and sleep for a certain period before requiring symbols. Listing 5.1 below gives the example. In this code, the time of Seeder is read from the socket, and a time to begin download is read from file *StartTime.txt*. For Leecher A, we set the variable bufferTime to be 10, so Leecher A will begin download at startTime+10. For Leecher B, we set bufferTime to 10.1, which means Leecher B will begin download 100 ms after Leecher A.

```
startTime = File.read("StartTime.txt")
bufferTime = 10
```

```
rcv_text , sender = socket.recvfrom(300)
seederTime = Time.pase(rcv_text)
puts "going to sleep"
sleep(startTime - seederTime + bufferTime)
puts "woke up now"
seeder.requireSymbols()
```

Listing 5.1. Time Scheduling in Ruby

5.4 Multi-peer Test

The multi-peer test is close to a real world model. In this test, we have 5 Raspberry Pis performing 5 peers (1 seeder and 4 leechers).

At the beginning of the test, only the seeder has the whole source file. Each leecher has a torrent file that contains the IP address of the tracker. The Ruby control program manages the join and leave time for group of peers. The swarm is initiated by a single leecher, which connects to the seeder, and starts the download. After a waiting period the second leecher joins and gets the IP address of the seeder and the 1st peer from the tracker. The 2nd leecher then connects to both the seeder and 1st leecher. This goes on until all the 5 peers joins the network.

At the end of the transmission, all the peers in the swarm have enough pieces to recover the source file. We monitored the downloading time, which is from a peer joins the swarm till it collects enough symbols to recover the source file. Figure 5.4 presents the test results. For a file at the size of 6.0 MB, we compared the average downloading time of each peer at different situations. As Figure 5.4 shows, the average download finishing time of all the peers decreases as the number of peers increases.

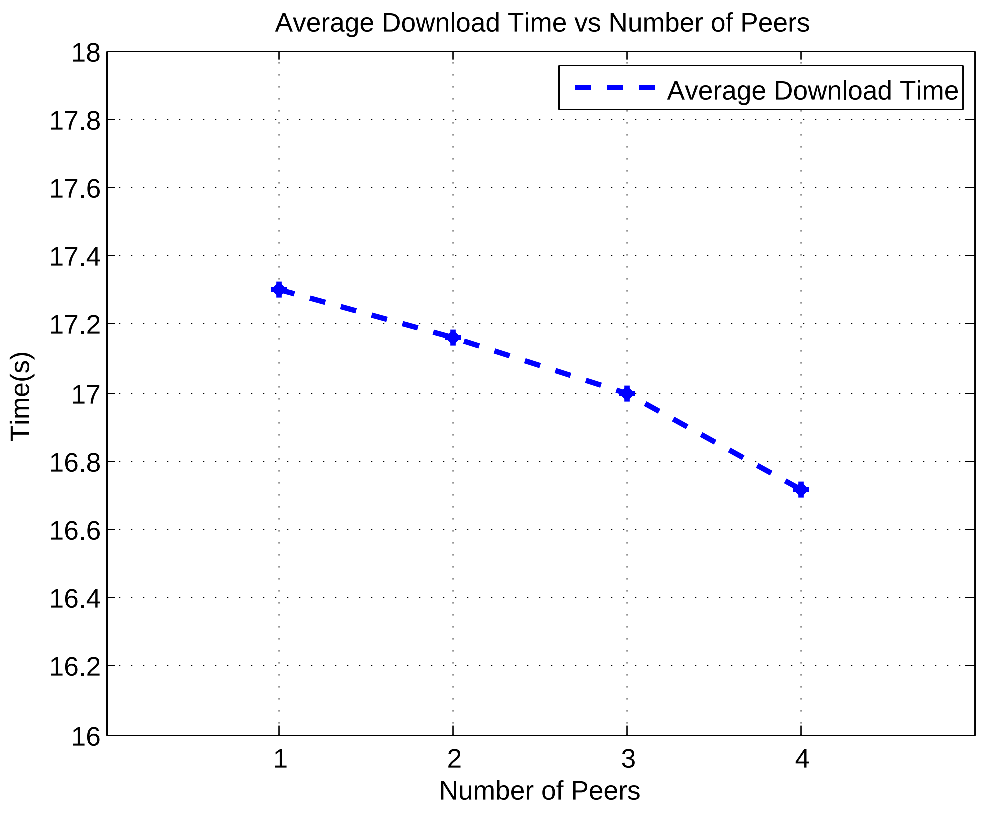


Figure 5.4. Download Time Test.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The Ruby interface development effort has resulted in an effective tool to rapidly prototype RaptorQ-based protocols. We have used the interface successfully to implement and test RaptorQP2P on a small network. Tests with a larger number of peers are needed to identify potential problems with a full-scale deployment.

There are still some drawbacks when using the RaptorQ SDK. We didn't find a method to allow certain symbols/pieces to be generated. That is, if a peer receives a request for the symbols with ESIs between m and $m+n$, it has to generate all the symbols from 0 to $m+n$, instead of just produce symbols m to n. This will clearly reduce the transmission efficiency. For the future work, we definitely need to find a way for certain symbols' generation. A way to do this is to use the open source code.

We didn't apply any selection algorithm in the potential neighbor management. Also, our tracker only keeps the IP address of peers in the swarm. In the future, we plan to allow the tracker monitor the bandwidth and piecemap of each peer. We believe some improvements can be made to the potential neighbor selection with these information.

Limited by our program and hardware, we couldn't make a fair comparison between our implementation of RaptorQ and BitTorrent. We plan to design an approach to compare the transmission efficiency and patterns between RaptorQP2P and BitTorrent in the future.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Z. Su, F. Wang, J. Daigle, and H. Wang, “RaptorQP2P: Maximize the performance of P2P file distribution with RaptorQ coding,” in *Proceedings of IEEE ICC 2015*, June 2015.

[2] A. Shokrollahi1 and M. Luby, “Raptor codes,” *Foundations and Trends in Communications and Information Theory*, vol. 6, pp. 213–322, 2009.

[3] M. Luby, A. Shokrollahi1, M. Watson, T. Stockhammer, and L. Minder, “Raptorq forward error correction scheme for object delivery,” *Internet Engineering Task Force*, August 2010. [Online]. Available: http://tools.ietf.org/html/draft-ietf-rmt-bb-fec-raptorq-03

[4] M. Luby, “LT codes,” *Proceedings 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.

[5] S. Puducheri, J. Kliewer, and T. Fuja, “The design and performance of distributed LT codes,” *IEEE Transactions On Information Theory*, vol. 53, no. 10, October 2007.

[6] C. Bouras, N. Kanakis, V. Kokkinos, and A. Papazois, “Enhancing reliable mobile multicasting with RaptorQ FEC,” *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pp. 000 082–000 087, 2012.

[7] R. Palanki and J. S. Yedidia, “Rateless codes on noisy channels,” *Information Theory, 2004. ISIT 2004*, 2004.

[8] P. Sandvik and M. Neovius, “The distance-availability weighted piece selection method for bittorrent: A bittorrent piece selection method for on-demand streaming,” *Advances in P2P Systems, 2009. AP2PS ’09*, pp. 198 – 202, 11-16 Oct. 2009.

[9] “Swig-1.3 documentation.” [Online]. Available: http://web.mit.edu/ghudson/trac/src/swig-1.3.25/Doc/Manual/SWIGDocumentation.html

[10] H. Zhang, Z. Shao, M. Chen, and K. Ramchandran, “Optimal neighbor selection in bittorrent-like peer-to-peer networks,” *SIGMETRICS11*, June 711, 2011.

[11] “Raspberry pi,” https://en.wikipedia.org/wiki/Raspberry_Pi, [Online; accessed 15-July-2015].

[12] “About Raspbian,” https://www.raspbian.org/RaspbianAbout, [Online; accessed 15-July-2015].

[13] “VNC (virtual network computing),” https://www.raspberrypi.org/documentation/remote-access/vnc/, [Online; accessed 15-July-2015].

APPENDICES

APPENDIX A

SOURCE CODES

A.1 Ruby codes for run time test

```ruby
# File: testRunTime.rb

require "./StringSimpleSend"
nRuns = 100      # nRuns denotes how many runs are taken out
fileNo = 0
while fileNo < 6
  i = 0
  totalTime_C = 0
  totalTime_Ruby = 0

  case fileNo
  when 0
    testFile = "StringSimpleSend_wrap.c"         # 143 KB
  when 1
    testFile = "IMG_0231.jpg"    # 2 MB
  when 2
    testFile = "24MB.bz2"        # 24 MB
  when 3
    testFile = "12MB.bz2"        # 12 MB
  when 4
    testFile = "18MB.bz2"        # 18 MB
  when 5
    testFile = "IMG_0325.jpg"    # 6 MB
  end

  outFile = File.open("./RunTimeTest/RunTimeTest_#{testFile}.txt","w")
  outFile.puts("Test Time: #{Time.now}")
  outFile.puts("File Size: #{(File.size(testFile).to_f/2**20).round(1)} MB
     ")
  outFile.puts("Number of Runs: #{nRuns}")

  while i < nRuns
    puts "==================================="
    file = File.open(testFile,"rb")
    t1 = Time.now.to_f
    infoFromC = StringSimpleSend::StringSimpleSend(1000,file.size, file.
        read, testFile, 200)
    t2 = Time.now.to_f
    var = infoFromC.split(",")
    runTime_C = var[7].to_i*0.001
    totalTime_C += runTime_C
    totalTime_Ruby += (t2-t1)
    puts "Run Time in C is #{runTime_C} ms"
    puts "Run Time in Ruby is #{(t2-t1)*1000} ms"
    puts "Run Count:#{i +=1}"
    file.close
  end

  avgTime_C = totalTime_C/nRuns
  avgTime_Ruby = totalTime_Ruby/nRuns
  outFile.puts("Average Run Time in C: #{avgTime_C} ms")
  outFile.puts("Average Run Time in Ruby: #{avgTime_Ruby*1000} ms")
  puts "Average Run Time in C is #{avgTime_C} ms"
  puts "Average Run Time in Ruby is #{avgTime_Ruby*1000} ms"
  outFile.close

  fileNo +=1
end
```

A.2 Main Program

```
#
# University of Mississippi, Department of Electrical Engineering
# by Yuzhu Bai, ybai1@go.olemiss.edu
#
# Model 3: This model is able to download and upload simultaneously.
# 1. Add a thread to connect to peers (in the peer list).
# 2.
#

# File: test_Model3.rb

require "./FileRcv/src/FileDecode.so"
require "./P2Putils.rb"
require 'fileutils'
require 'thread'
require 'socket'
require 'io/wait'
puts "=============================================================="
puts "*                                                            *"
puts "*                   File: test_Model3.rb                     *"
puts "*                                                            *"
puts "*                University of Missisippi                    *"
puts "*                                                            *"
puts "=============================================================="

# Preparation
lock = Mutex.new                # Mutex for synchronizing public data.
neighbors = Membership.new
local_ip = get_local_ip         # Gets the local IP address
fileName = "IMG_0325.jpg"
File.new(fileName+'.pieces','w+')

# Connect to seeder
```

```

seeder = TCPSocket.new('192.168.0.100',4481)
puts "Connected to seeder: #{seeder}"
sock_domain, remote_port, remote_hostname, remote_ip = seeder.peeraddr
peerObj = Peer.new(remote_ip, seeder)
neighbors.add_peer(peerObj)

# Send piece request

reqPieces = Array.new
for i in 0..99
  reqPieces << i
  reqPieces << 0
  reqPieces << 0
end
puts reqPieces.to_s
sLength = reqPieces.to_s.bytesize.to_s.rjust(8,'0')
peerObj.socket.write("REQ" + sLength + reqPieces.to_s)


# Start a thread listening to other peers.
listen = Thread.new {
  puts "Server thread is up. Listening to port 4481.\n"
  server = TCPServer.new(4481)
  listen_to_peers(server, neighbors, lock, local_ip)
}

# Start a new thread for receiving.
receicve = Thread.new {
  puts "New thread is up for reading from client.\n"
  sleep(4)
  loop do
    puts '=========================================='
    puts 'Receiving thread is reading from socket.'
    for i in 0..(neighbors.peers.length-1)
      puts neighbors.peers[i].socket.ready?
      if neighbors.peers[i].socket.ready?
#          tempData = (neighbors.peers[i].socket.readpartial(10240) rescue
    nil)
        lock.synchronize {
          neighbors.peers[i].rcvBuf << (neighbors.peers[i].socket.
              readpartial(10240) rescue nil)
        }
        puts "receive buffer length: #{neighbors.peers[i].rcvBuf.length}"
        sleep(0.01)
      else
        puts "No data from socket #{neighbors.peers[i].ip}"
        sleep(4)
      end
    end
  end
}

# Start a new thread for message processing.
msgproc = Thread.new {
  puts "New thread is up for processing message.\n"
```

```ruby
  sleep(6)
  loop do

    puts '---------------------------------------------'
    puts 'Message processing thread is active.'
    for i in 0..(neighbors.peers.length-1)
      if neighbors.peers[i].rcvBuf.length > 0
        lock.synchronize {
          if packet_process(neighbors.peers[i].rcvBuf, fileName, neighbors.peers[i], lock) == 0
            # Remove processed data from buf
            neighbors.peers[i].rcvBuf = neighbors.peers[i].rcvBuf[11+neighbors.peers[i].rcvBuf[3..10].to_i..-1]
          end
        }
      else
        puts "No data from socket #{neighbors.peers[0].ip}"
      end

    end
    sleep(0.1)
  end
}

# Start a new thread to send data.
send = Thread.new {
  puts "New thread is up for sending data."
  sleep(4)
  loop do
      puts '***************************************'
      puts 'Data transmitting thread is active.'

    for i in 0..(neighbors.peers.length-1)
      if neighbors.peers[i].sendBuf.length > 0
        lock.synchronize {
          puts "#{neighbors.peers[i].sendBuf.length} bytes data to send."
#IO.write("bufferInMain", neighbors.peers[i].sendBuf)   # for debug
          puts pLength = neighbors.peers[i].sendBuf[3..10].to_i
          puts neighbors.peers[i].sendBuf[0..10]

          # Send one packet each time.
          neighbors.peers[i].socket.write(neighbors.peers[i].sendBuf[0..(10+pLength)])
          puts "Data sent to #{neighbors.peers[i].ip}"
#IO.write("bufferInMain1", neighbors.peers[i].sendBuf)  # for debug
          # Remove processed data from buf
          neighbors.peers[i].sendBuf = neighbors.peers[i].sendBuf[(11+neighbors.peers[i].sendBuf[3..10].to_i)..-1]
#IO.write("bufferInMain2", neighbors.peers[i].sendBuf[(11+pLength)..-1])  # for debug
#IO.write("bufferInMain3", neighbors.peers[i].sendBuf)  # for debug
        }
      else
        puts "No data to send to socket #{neighbors.peers[i].ip}"
      end
    sleep(0.001)
    end
    sleep(0.1)
  end
}

# We make main thread doing nothing here.
loop do
  puts 'Main thread is doing nothing.'
  sleep(20)
end
```

A.3 Utility Functions for RaptorQP2P

```
#
# University of Mississippy, Department of Electrical Engineering
# by Yuzhu Bai, ybai1@go.olemiss.edu
#
# Sept 10, 2014
#
#

# File: P2Putilss.rb

class Peer
  def initialize(ip, socket)
    @ip = ip
    @socket = socket
    @req = Array.new
    @rcvBuf = ''
    @sendBuf = ''
  end
  attr_accessor :ip, :piecemap, :state, :socket, :sRate, :rcvBuf, :
      am_choking, :am_interested, :peer_choking, :peer_interested
attr_accessor :req, :sendBuf
  # @state: handshaked, am_choking, am_unchoking
end

class Membership
  def initialize()
    @peers = Array.new
    @sockets = Array.new
  end

  def peers
    @peers
  end

  def add_peer (peer)
    @peers.push(peer)
  end

  def add_socket (socket)
    @sockets.push(socket)
  end

public :add_peer, :add_socket
end

def read_torrent_file(torrentFileName)
# Input: .torrent file name
# Outputs: Tracker IP, file name, symbol  size, number of source symbols
  torrentFile = File.open(torrentFileName)
  trackerIP, fileName, fileSize, symbolSize, nSrcSymbols = torrentFile.
      readlines
  torrentFile.close
  puts "————————————————————————————————————————"
  puts "Method: read_torrent_file()\n\n"
  puts "Tracker IP: #{trackerIP}"
  puts "File Name: #{fileName}"
  puts "File Size: #{fileSize}"
  puts "Symbol Size: #{symbolSize}"
  puts "Number of Source Symbols: #{nSrcSymbols}"
  puts "————————————————————————————————————————\n\n"
  return trackerIP, fileName, fileSize, symbolSize, nSrcSymbols
end

def get_piecemap(fileName, nSrcSymbols, symbolSize)
# Input: file name, number of source symbols, symbolsize
# Output: the piecemap of the file
  puts "————————————————————————————————————————"
  puts "Method: get_piecemap()\n\n"
  if File.exist?(fileName+".pieces")
    puts "Getting piecemap..."
    get_symbol_tags(fileName+".pieces", symbolSize)
    pieceMap = File.read(fileName+".piecemap")
  else
    pieceMap = nil
  end
  puts "Gets the piecemap of file #{fileName}."
  puts "————————————————————————————————————————\n\n"
  return pieceMap
end

def bin_to_hex(s)
# Converts binary string into hexadecimal.
  s.each_byte.map{|b| b.to_s(16)}.join
end

def get_symbol_tags(fileName, symbolSize)
# Gets the symbol tags in a file.
# The structure of a symbol is as follows:
# (Data)(Data)(Data) ... (Data)(Tag Byte 1)(Tag Byte 2)(Tag Byte 3)(Tag
    Byte 4)
  i = 1
  bTag = 0
  file = File.open(fileName+".piecemap", "w")
  while true
    bTag = IO.binread(fileName, 4, (symbolSize * i -4))
    if bTag != nil
```

```
      symbolTag = bin_to_hex(bTag[0]).to_i(16) + bin_to_hex(bTag[1]).to_i
          (16)*256 + bin_to_hex(bTag[2]).to_i(16)*256*256 + bin_to_hex(
          bTag[3]).to_i(16)*256*256*256
      file.puts(symbolTag)
      i +=1
    else
      break
    end
  end
  file.close
end

def listen_to_peers(server, members, lock, local_ip)
  loop do
    Thread.start(server.accept) do |s|
      sock_domain, remote_port, remote_hostname, remote_ip = s.peeraddr
      puts "Connection request from peer #{remote_ip} is accepted."
      lock.synchronize {
        peer = Peer.new(remote_ip, s)
        members.add_peer(peer)
      }
      puts "Current members:\n#{members.memberlist}"
=begin
      while data = s.recvfrom(40)[0].chomp do
        puts "Data from #{remote_ip}:\n  #{data}"
        if data.include?("bye")
          puts "Communication with #{remote_ip} closed!"
          s.close
        end
      end
=end
    end
  end
end

def connect_to_peers(peerList, members, lock, local_ip)
  i = 0
  while i < peerList.length do
    unless members.memberlist.include? peerList[i]
      puts "Connecting to #{peerList[i]}"
      begin
        newpeer = TCPSocket.new(peerList[i],4481)
        sock_domain, remote_port, remote_hostname, remote_ip = newpeer.
            peeraddr
        puts "Connected to #{remote_ip}"
#         newpeer.write("PieceMap:"+pieceMapString)
        peer = Peer.new(remote_ip)
        lock.synchronize {
          peer.socket = newpeer
          members.add_peer(peer)
        }
        puts "Current members:\n#{members.memberlist}"
      rescue
        puts "Connecting to #{peerList[i]} failed!"
      end
    end
    i +=1
  end
end

def send_data(members, lock)
# Sends the data in each peer's buffer
  loop do
    i = 0
    while members.memberlist.length > 0 and i < members.memberlist.length
      lock.synchronize {
        members.memberlist[i].socket.write(members.memberlist[i].buffer)
      }
      i +=1
    end
  end
end

def peer_mgmt_data_trans(piecemap, members, lock, peerList)
  nMembers = 0
  loop do
    puts "Main is up."
    sleep (3)
    lock.synchronize {
      nMembers = members.memberlist.length
    }
    puts "Current number of active members: #{nMembers}"
    if nMembers > 0
      puts "#{members.memberlist[0].socket.class}"
      data = members.memberlist[0].socket.recvfrom(40)[0].chomp
      puts "Data from #{members.memberlist[0].ip}:\n  #{data}"
    end
  end
=begin

    if member.memberlist.length > 0
      data = member.memberlist[0].socket.recvfrom(40)[0].chomp
      puts "Data from #{remote_ip}:\n  #{data}"
    else
      puts"Main: No members connected."
      sleep(5)
    end
  end
=end
end

def get_local_ip
  orig, Socket.do_not_reverse_lookup = Socket.do_not_reverse_lookup, true
      # turn off reverse DNS resolution temporily
  UDPSocket.open do |s|
    s.connect '64.233.187.99', 1
    s.addr.last
  end
ensure
  Socket.do_not_reverse_lookup = orig
end

```

```
def select_neighbors(members, neighbors, lock)
# Keep 5 peers as neighbors. Returns a list of neighbors.
  neighbors = members

# Change the states of neighbors to "unchoke".
  if neighbors.memberlist.length > 0
  lock.synchronize {
    0.upto(neighbors.memberlist.length) { |i| neighbors.memberlist[i].
        state = "unchoke" }
  }
  puts "Neighbors selected.\n\n"
  end
end

def update_neighbors(members, neighbors, lock)
# Keep 5 peers as neighbors. Returns a list of neighbors.
  neighbors = members
# Change the states of neighbors to "unchoke".
#  puts neighbors.memberlist[0].ip
  lock.synchronize {
    0.upto(neighbors.memberlist.length - 1) { |i| neighbors.memberlist[i].
        state = "unchoke" }
  }
  puts "Neighbors updated.\n\n"
end

def packet_process(inString, fileName, peerObj, lock)
  if inString[0..2]['DAT']
    puts "It\'s data. I\'m going to write this into #{fileName}.pieces"
    if inString.length < inString[3..10].to_i
      puts "Waiting for whole piece finished."
      return 1
      exit
    else
      data = inString[11..(10+inString[3..10].to_i)]
      puts "data length = #{data.length}"
      File.write(fileName+'.pieces', data, File.size(fileName+'.pieces'),
          mod: 'a')
      return 0
    end
  elsif inString[0..2]['PMP']
    if inString.length < inString[3..10].to_i
      return 1
      exit
    else
      puts 'It\'s piecemap. Updating piecemap for peer #{peerObj}'
      puts data = inString[11..(10+inString[3..10].to_i)]
      peerObj.piecemap = data.split(",").map{|s| s.to_i}
      puts peerObj.piecemap.class
      return 0
    end
  else
    if inString.length < inString[3..10].to_i
      return 1
    else
      puts 'It\'s message. I\'m going to call msg_process'
      msg_process(inString, peerObj, fileName, lock)
      return 0
    end
  end
end

def test_msgprocess(inString, i)
  inString << i.to_s
end

def msg_process(inString, peerObj, fileName, lock)
#  puts 'This is msg_process():'
#  puts inString
  case inString[0..2]
  when 'REQ'
    puts 'It\'s request.'
    data = inString[12..(9+inString[3..10].to_i)]
    peerObj.req = data.split(",").map{|s| s.to_i}
    puts "peerObj.req = #{peerObj.req}"
    prepare_data(peerObj, fileName, lock)
#    puts peerObj.sendBuf.length
  when 'CTL'

  else

  end
end

def prepare_data(peerObj, fileName, lock)
# Prepares pieces requested by peerObj, and write into peerObj.sendBuf
  puts "Now preparing data for #{peerObj.ip}"
  for i in 1..peerObj.req.length/3
    peerObj.sendBuf << pack_piece(fileName, peerObj.req[(i-1)*3])
#    puts "Piece #{peerObj.req[i*3]} loaded."
  end
  puts "Piece #{peerObj.req[(i-1)*3]} loaded."  # for debug
  IO.write("buffer", peerObj.sendBuf)           # for debug
  puts "Send buffer preparation accomplished."
end

def pack_piece(fileName, pieceNum)
  begin
#    puts "temp/" + fileName + ".P" + pieceNum.to_s     # for debug
    data = IO.binread("temp/" + fileName + ".P" + pieceNum.to_s)
    sLength = data.length.to_s.rjust(8,'0')      # Data length
#    sLength = data.bytesize.to_s.rjust(8,'0')  # Data length
    return "DAT" + sLength + data
  rescue
    puts "Error reading from file #{"temp/" + fileName + ".P" + pieceNum.
        to_s}"
    exit
  end
end

def divide_pieces(fileName, symbolSize, outfileName)
```

```
# The input file (fileName) should be encoded files (*.src or *.rep)that
    just generated by encoder.
# This subroutine divides encoded files into many files , each containing
    only one symbol(piece).
  inFile = File.open(fileName, "rb")
  content = inFile.read
  while content.length > 0
    symbol = content[0..(symbolSize-1)]
    puts symbol.length
    content = content[symbolSize..-1]
    puts symbolID = bin_to_hex(symbol[symbolSize-4]).to_i(16) + bin_to_hex
        (symbol[symbolSize-3]).to_i(16)*256 + bin_to_hex(symbol[symbolSize
        -2]).to_i(16)*256*256 + bin_to_hex(symbol[symbolSize-1]).to_i(16)
        *256*256*256
    outFile = File.open(outfileName+".P"+symbolID.to_s,"w")
    outFile.write(symbol)
    outFile.close
  end
  inFile.close
end

def continuously_select(start, n)
  inFile = File.open("Symbols.all","rb")
  outFile = File.open("SelectedSymbols","w")
  i = start
  while i < start+n
    outFile.write(IO.binread("Symbols.all",59998, 59998*i))
    i +=1
  end
  inFile.close
  outFile.close
end

def mod_M_select(k,m,n)
# Starts at k_th symbol
  inFile = File.open("Symbols.all","rb")
  outFile = File.open("SelectedSymbols","w")
  i = 0 # Number of symbols
  j = k # Symbol ID
  while j < 1000
    if i < n
      if (j-k)%m == 0
        outFile.write(IO.binread("Symbols.all",59998, 59998*j))
        i +=1
      end
    end
    j +=1
  end
  inFile.close
  outFile.close
end
```

A.4 Tracker

```ruby
#
# University of Mississippy, Department of Electrical Engineering
# by Yuzhu Bai, ybai1@go.olemiss.edu
#
# Tracker for RaptorQP2P.
# Accepts requests from clients and returns peer list.
# Sept. 2 2014

# File: Tracker.rb

require 'socket'

# create a new TCP socket
server = TCPServer.new(4481)

loop do
  # Wait until a client connects.
  puts 'Ruby: Tracker waiting for client.'
  connection, _ = server.accept
  puts 'Ruby: Client connected.'

  # Return peer list to client.
  peerlist = File.open("PeerList.txt","r+")
  connection.write( peerlist.read )

  # Add new client into peer list.
  print 'Ruby: Client IP'
  sock_domain, remote_port, remote_hostname, remote_ip = connection.
      peeraddr
 # peerlist.puts(remote_ip)

  peerlist.close
  connection.close

end
```

VITA

Yuzhu Bai received his Bachelor of Engineering degree in Communication Engineering in 2007 at Nanjing University of Science and Technology, Nanjing, China, and Master of Engineering degree in Signal and Information Processing in 2011 at North China University of Technology, Beijing, China. In August 2011, he joined the Department of Electrical Engineering at the University of Mississippi as a graduate student emphasizing in Telecommunications, where he was also a research assistant from August 2011 to June 2012. Since January 2012, he has been a research assistant in National Center for Physical Acoustics, University of Mississippi. His research interest includes wireless communication, signal processing and network programming.