# Peer-to-Peer File Sharing

***Release 1.0***

**Joshua Talbot and Ethan Nunez**

**Apr 14, 2025**

# MODULES:

Welcome to the documentation for the Python-based peer-to-peer file sharing system.

This project mimics basic BitTorrent-style behavior using a custom protocol.

CHAPTER

# ONE

# CONTENTS

## 1.1 Modules

### 1.1.1 p2p_command

p2p_command.**exchange_data**(*peers*, *peer_name*, *file_id*, *receiver*, *address*)

Sends an EXCH_REQ to the remote peer. The remote peer's receiver will queue a file-sending job in 'exch_req_queue'.

p2p_command.**get_index_path**(*exch_id*)

Given a file ID, return the absolute file path.

**Parameters**

- **exch_peer** – the peer requesting the file (not used currently)
- **exch_id** – the file ID being requested

**Returns**

the file path as a string, or empty string if not found

p2p_command.**p2p_command_line**(*name*, *port*)

Main interface for the P2P system. Handles user input and executes commands.

p2p_command.**peer_discovery**(*my_port*, *my_name*)

Uses the tracker to discover other peers in the network. Returns a dictionary mapping 'peer_id' to (ip, port).

p2p_command.**print_index**(*peer_addr=None*, *soc=None*)

Displays the list of available files from this peer. Uses a separate socket to avoid interference with receiver.

p2p_command.**print_menu**()

Prints the command menu for the user.

p2p_command.**process_exchange_requests**(*exch_req_queue*, *address*, *soc*)

Continuously monitors 'exch_req_queue' for (file_id, peer_address) tuples and starts a Sender to serve that file to 'peer_address'.

p2p_command.**register_with_tracker**(*tracker_host*, *tracker_port*, *peer_host*, *peer_port*, *peer_name*)

Connects to the tracker and registers the current peer. Returns a list of other peers in the network.

p2p_command.**start_tracker**(*host='0.0.0.0'*, *port=9000*)

Starts the tracker server that listens for incoming peer registrations. It adds peers to a global set and returns the list of known peers (excluding the caller).

### 1.1.2 receiver_rdt

**class** receiver_rdt.**Receiver**(*soc*, *peer_files=None*)

Bases: object

Receiver class that can handle multiple files simultaneously.

For each inbound file, we store its 'base_seq', 'max_seq', and 'packets' inside self.active_files[file_id], for example:

**self.active_files[file_id] = {**
'base_seq': -1, 'max_seq': -1, 'packets': []

}

When a new chunk arrives for 'file_id', we place it at index [seq - base_seq]. Once we see seq == -1, we know the sender is done sending that file, and we call finalize_file(file_id) to write out the chunks and remove the entry.

**packets**

Array of received decoded data

**soc**

socket that receiver uses to bind and receive data over

**ip**

ip address to receive data from

**port**

port number to receive data from

**base_seq**

the lowest sequence number to index by

**max_seq**

the highest sequence number known to the receiver

**add_packet**(*file_id*, *seq_num*, *data_str*, *expand_pkts*)

Given file_id, seq_num, data_str, place the data into the correct spot in 'info["packets"]'. If expand_pkts is True, we enlarge 'info["packets"]' up to seq_num.

**Parameters**

- **file_id** (*String*) – the identifier of the inbound file
- **seq_num** (*int*) – sequence number of this chunk
- **data_str** (*String*) – the chunk contents
- **expand_pkts** (*bool*) – True if seq_num >= info['max_seq'], meaning we may need to extend the packets array

**base_seq = -1**

**finalize_file**(*file_id*)

Writes out the collected packets for 'file_id' to <file_id>_torrent.txt, then clears them from self.active_files.

**Parameters**
**file_id** (*String*) – the unique identifier for the file being transferred

**listen_for_requests**(*exch_req_queue*)

Waits for request from other peers from self.soc, verifies data and verifies requests Runs as long as the p2p_command is running

**max_seq = -1**

**packets = []**

**rebase_packets**(*file_id*, *seq_num*, *data_str*)

Given file_id and a chunk's sequence number is smaller than base_seq, rebase so that 'seq_num' becomes the new base_seq and put 'data_str' at index 0.

**Parameters**

- **file_id** (*String*) – the identifier of the inbound file
- **seq_num** (*int*) – the inbound packet's sequence number
- **data_str** (*String*) – the actual file data chunk

**set_timeout**()

Optional method to signal that this receiver should time out.

**timeout = None**

receiver_rdt.**convert_sender_payload**(*data*)

Decodes packet payload to retrieve sequence number and message of packet

**Parameters**
**data** (*Bytes*) – sequence of Bytes to decode

**Returns**
send_seq, sequence number of packet

**Return type**
Bytes

**Returns**
msg, data from packet

**Return type**
String

receiver_rdt.**make_checksum**(*data*)

Forms checksum from data using crc32 function from zlib library

**Parameters**
**data** (*Bytes*) – sequence of Bytes to calculate checksum

**Returns**
checksum of data

**Return type**
Bytes

receiver_rdt.**make_packet**(*seq_num*, *msg*)

Forms packet by combining calculated checksum and formed payload

**Parameters**

- **seq_num** (*int*) – int to convert to bytes
- **msg** (*String*) – characters to encode

**Returns**
payload, sequence of bytes containing seq_num and msg

**Return type**
Bytes

receiver_rdt.**make_receiver_payload**(*seq_num*, *msg*)

Forms packet payload by encoding sequence number and message of packet

**Parameters**

- **seq_num** (*int*) – int to convert to bytes
- **msg** (*String*) – characters to encode

**Returns**
payload, sequence of bytes containing seq_num and msg

**Return type**
Bytes

receiver_rdt.**verify_integrity**(*sent_chksum*, *data*)

Verifies checksum from received packet

**Parameters**

- **sent_chksum** (*Bytes*) – received checksum with length of 8 bytes
- **data** (*Bytes*) – sequence of bytes to calculate checksum with

**Returns**
if sent_chksum is the exact same as calculated checksum

**Return type**
Boolean

### 1.1.3 sender_rdt

**class** sender_rdt.**Sender**(*soc*, *ip*, *port*, *file_id*)

Bases: object

Sender, a class with defined behavior to send data to a receiver

**packets**

Array of 3 object arrays containing:

**[formed byte packet, boolean ack, Timeout retransmission thread]**

**soc**

socket that sender uses to send data over

**ip**

ip address to send data to

**port**

port number to send data to

**base_seq**

the lowest sequence number to index by

**arrange_pkts**(*data*)

Given chunks of data, populate each entry of Sender packets with packet, False (for acknowledgement), thread.Timer for timeout and retransmit

**Parameters**
**data** (*Array of Strings*) – array of chunks of data

**find_recv_base_window**(*window_size*)

Given window size and Sender packets, find the closest unacknowledged packet and calculate the window

**Parameters**
**window_size** (*int*) – size of window

**make_packets**(*exch_path*, *chunk_size*)

Forms packets from file by splitting file into chunks

**Parameters**

- **file_name** (*String*) – String containing name of file to send
- **chunk_size** (*int*) – number of characters to fit in a chunk from file

**Returns**
pkts, array of character chunks from file

**Return type**
Array

**packets = None**

**run_sender**()

Sends packets using Selective Repeat. Only creates timers once per packet.

**send_pkt**(*seq_num*)

Retransmits packet after timeout by thread.Timer and resets timeout

**Parameters**
**seq_num** (*int*) – sequence number to retransmit

**setup_exchange**(*exch_path*)

sender_rdt.**convert_ack_payload**(*data*)

Parses a receiver ACK payload (just a sequence number + "ACK")

**Parameters**
**data** – sequence of bytes

**Returns**
seq_num, message

sender_rdt.**convert_receiver_payload**(*data*)

Decodes packet payload to retrieve sequence number and message of packet

**Parameters**
**data** (*Bytes*) – sequence of Bytes to decode

**Returns**
send_seq, sequence number of packet

**Return type**
Bytes

**Returns**
msg, data from packet

**Return type**
String

sender_rdt.**make_checksum**(*data*)

Forms checksum from data using crc32 function from zlib library

**Parameters**
**data** (*Bytes*) – sequence of Bytes to calculate checksum

**Returns**
checksum of data

**Return type**
Bytes

sender_rdt.**make_packet**(*seq_num*, *msg*, *file_id*)

Forms packet by combining calculated checksum and formed payload

**Parameters**

- **seq_num** (*int*) – int to convert to bytes
- **msg** (*String*) – characters to encode

**Returns**
payload, sequence of bytes containing seq_num and msg

**Return type**
Bytes

sender_rdt.**make_sender_payload**(*seq_num*, *msg*, *file_id*)

Forms packet payload by encoding sequence number and message of packet

**Parameters**

- **seq_num** (*int*) – int to convert to bytes
- **msg** (*String*) – characters to encode

**Returns**
payload, sequence of bytes containing seq_num and msg

**Return type**
Bytes

sender_rdt.**verify_integrity**(*sent_chksum*, *data*)

Verifies checksum from received packet

**Parameters**

- **sent_chksum** (*Bytes*) – received checksum with length of 8 bytes
- **data** (*Bytes*) – sequence of bytes to calculate checksum with

**Returns**
if sent_chksum is the exact same as calculated checksum

**Return type**
Boolean

# INDEX

## T



## V