ENGG 4920CQ Thesis II

Implementation of Network Coding Peer-to-Peer System

BY

YING, Xuhang (1009611842)

A FINAL YEAR PROJECT REPORT
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF BACHELOR OF INFORMATION ENGINEERING
DEPARTMENT OF INFORMATION ENGINEERING
THE CHINESE UNIVERSITY OF HONG KONG

May 20, 2013

Abstract

NeP2P protocol is a brand-new protocol that aims to deliver data more efficiently by using network coding powered by Peer-to-Peer techniques. To ensure its overall performance, it is necessary to design and implement an efficient Reliable UDP with NAT traversal capability. In addition, to conduct efficient testing for a P2P network, perform insightful data analysis and archive testing data, a good testing platform is highly desirable for NeP2P development. In our Final Year Project, we developed an efficient N-to-M bidirectional Reliable UDP, with NAT traversal capacity based on the hole punching technique. Simulation results and real-world experiments have demonstrated its functionality and efficiency. Furthermore, we built the NeP2P testing platform and performed sample testings to demonstrate its functionality and usability, and also pointed out evaluation dimensions that can be explored. So far, we have gathered a large amount of raw testing data, carried out first-order statistical data analysis, and reported NeP2P bugs to its developers. All data is well-archived and visually presented on the Online Demo Website.

Contents

1 Introduction

As a brand-new protocol, NeP2P aims to deliver data more efficiently by using network coding powered by Peer-to-Peer techniques. It provides reliable delivery, multicast and multipath support, and can be integrated with upper layer protocol to provide additional functions. The major innovation is the use of coding technology that brings significant benefits to file delivery.

NeP2P protocol ensures reliability in an innovative manner. Unlike TCP, the most heavily deployed reliable protocol that guarantees reliability with positive acknowledgement and retransmission, NeP2P adopts *erasure coding* [19] to achieve the same objective more efficiently. Moreover, network coding techniques are applied to provide a better support for multicasting and multi-path transmission compared to the file chunking used in other P2P systems. In particular, the rarest chunk problem inherent in any chunking based method is avoided. All in all, NeP2P protocol not only provides a reliable P2P transmission service, but also achieves higher efficiency in bandwidth utilization than other traditional retransmission-chunking based approaches.

As an extremely important part of NeP2P protocol, it is necessary to design and implement Reliable User Datagram Protocol (Reliable UDP) with Network Address Translation (NAT) traversal capability to guarantee successful reliable data delivery of control messages and unreliable data delivery of data messages among peers behind most NAT devices. Moreover, this RUDP has to support multicasting among peers efficiently and reliably.

More importantly, since NeP2P is still under development and its performance is not optimized yet, it is extremely worthwhile to build an efficient testing platform for debugging and performance analysis. As NeP2P targets on a real P2P network with a large number of remote nodes that are geographically separated, it is very challenging to perform centralized control, efficient code deployment, traffic monitoring, data analysis and archiving for a particular test. In addition, as NeP2P involves plently of optimization tasks, such as scheduling, congestion control, decoding etc., we believe a graphical and intuitive presentation toolkit is desperately needed for developers to understand protocol behaviors, and such toolkit can provide more insights for performance optimization.

Our contribution are listed as follows:

- In Section 3, we have:
  1. designed and implemented efficient and scalable N-to-M bidirectional RUDP using a single port in one process with well-defined interfaces;
  2. conducted simulation on an isolated peer to test basic functionalists of RUDP, and real experiments with multiple peers in PlanetLab;
  3. designed four operation modes of RUDP running, debug, logging and statistical analysis, and integrated it with NeP2P implementation smoothly.
- In Section 4, we have:
  1. studied the problem of NAT traversal and investigated existing solutions;
  2. designed and implemented Node List Updating based on the hole punching technique customized to NeP2P design;
  3. conducted experiments on both wireless and wired networks with CUHK campus to demonstrate the functionality of hole punching technique.
- In Section 5, we have:

1. designed and implemented a NeP2P testing platform consisting of Master Console, PlanetLab Nodes and Online Demo Websites, supporting centralized control, efficient code deployment, data logging and data visualization functions;
2. conducted sample testing on PlanetLab nodes to demonstrate its functionality, and pointed out possible evaluation dimensions that can be explored;
3. performed a large number of real experiments using this testing platform, and and reported existing NeP2P flaws to developers.

2 Background

2.1 Reliable UDP

As an essential building block of NeP2P protocol, a lower-level transport protocol is needed to provide both reliable and unreliable delivery service. Although there exist similar protocols or even open-source implementations, we are motivated to design our own protocol with the following reasons.

At the first place, TCP is not desirable in our NeP2P protocol. As TCP is connection-oriented, large overhead in connection establishment and termination is inefficient to deliver control messages of small size (e.g. around 1 KB) reliably and data messages unreliably (which is compensated by NeP2P protocol instead). In addition, TCP does not support multiple connections on a single port. Furthermore, maintenance of state information per-connection occupies memory and potentially degrades its performance.

At the second place, we choose UDP as our underlying protocol since it is the thinnest wrapper around IP. Though UDP lacks rate control, it has been considered using one more fundamental layer in NeP2P architecture to compensate these deficiencies.

Existing Reliable UDP documentation (e.g. RFC 1151 [13]) serves as a core reference for our initial design. Basically, we need a simple but efficient reliable protocol with least overhead. Therefore, we develop our own version of Reliable UDP to fulfill requirements imposed by NeP2P protocol.

2.2 NAT traversal

In order to alleviate the exhaustion of IPv4 address space, NAT devices allow multiple hosts to share a public IP address. Basically, NAT achieves its primary goal by performing modifications to TCP/IP address/port number of a packet and building mappings between private address/port number and public address/port number. Nowadays, one-to-many NAT devices are intensively deployed.

In practice, there are mainly four types of port translation, namely full cone, restricted cone, port-restricted cone and symmetric mapping [2]. In spite of extended IPv4 address spaces and additional security, hosts behind a NAT device cannot be reached by the outside world unless they initiate the connections. However, NAT has caused a lot of problems when it comes to peer-to-peer communication. For instance, it is necessary for media (audio and video) to flow directly between two end hosts in VoIP, since relaying is very costly in terms of bandwidth and latency.

In our context, an efficient NAT traversal solution is very essential for NeP2P protocol. First of all, NAT traversal is an unavoidable problem for any P2P systems and protocols, including

NeP2P. Second, its efficiency will significantly affect the overall performance of NeP2P which relies heavily on peer-to-peer data communication.

2.3 NeP2P Testing Platform

PlanetLab is "a global research network that supports the development of new network services" [15], which consists of up to 1167 nodes at over 500 sites. It is commonly used to create a realistic P2P networking environment, as remotes nodes are geographically separated with heterogeneous networking environment and computing capacities.

To realize remote control of PlanetLab nodes, Fabric is a very good choice. It is "a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks" [5]. Powered by this library, we can execute commends on multiple remote nodes in parallel.

An comprehensive form of data visualization speaks more than pure digits, and the nvd3.js [12] library is used to build an online demo website with four different types of interactive charts for better data interpretation and representation.

3 Reliable UDP

In this project, Reliable UDP (RUDP) is not a stand-alone protocol design or implementation to facilitate peer-to-peer communication. Instead, as an underlying layer for NeP2P, RUDP has to fit overall architecture design of NeP2P and provides reliable data transmission for light-weight control messages and unreliable delivery services for data messages.

Therefore, to meet all the requirements of NeP2P, we have devised our RUDP as a delicate wrapper of UDP with progressively refined functionalities. In addition, several simulations have been conducted to show that this RUDP could achieve reliable data transmission with great efficiency and robustness by introducing intentional demon peers transmitting invalid packets. After the basic implementation of RUDP, we keep on improving the design of this protocol to make it flexible enough for different transmission models, while preserving its simplicity by tearing down unnecessary features for large-scale bidirectional data transmissions in a real-world setting. At last, by integrating RUDP with the whole NeP2P project, it has been proven that this protocol module could be compatible with Peer-to-Peer network in a scalable manner.

Three different versions of RUDP have been developed with additional requirements listed as follows:

- **Version 1**: A One-to-One uni-directional RUDP built on top of UDP with minimized protocol overhead, connection-oriented operations and simple state managements, achieving basic efficiency and robustness.
- **Version 2**: A purified version of RUDP by separating the module as a packet interpreter, a packet processing function and a rudpConnection class to maintain each connection's transmission state for better applicability and flexibility. In addition, a One-to-N transmission model designed to meet single-server-multi-clients (S-MC) scenario for testing the functionality and the performance of RUDP.
- **Version 3**: A N-to-M bidirectional RUDP using a single socket within one process powered by multi-threading, in a simpler but more scalable manner. Apart from the normal

running mode, debug and logging modes are needed to help keep track of protocol operations for further development and analysis.

3.1 Objectives and Challenges

To achieve all the aforementioned requirements, we have encountered various obstacles in both protocol design and implementation issues.

For example, to fulfill the demand for inconspicuous packet-wise switch between reliable and unreliable data delivery, the receiver should have different behaviors in response to these two types of packets, but, meanwhile, keep it consistent for upper layer to read data from RUDP just like standard transport layer protocol.

Moreover, to design a protocol supporting one-to-many and even many-to-many connections, we should carefully eliminate redundant state data on each peer for dispatching packets to upper layer, and reduce the time complexity of the retransmission mechanism in case of packet loss. As a result, we abandoned the connection-oriented design that people used excessively in reliable data transmission protocols, but devised new data structures to realize state managements.

Last but not least, since only one socket, which means one single port, could be used to handle all data connections with many remote peers, a special data structure is necessary for quick referencing in terms of remote address. Also, multi-threading is inevitable for supervising incoming packet interpretation, response construction and retransmission mechanism as all the RUDP packets go through one external interface.

After combining all these three versions of RUDP, the main features of this protocol are as follows:

1. One peer should be able to (i) send data packets to multiple peers reliably and unreliably; (ii) receive data packets from multiple peers reliably and unreliably.

2. One peer should only occupy one port number for both sending and receiving data simultaneously, where the sending process is non-blocking.

3. For the scalability of usage, a RUDP socket is preferred with identical interfaces of original socket.

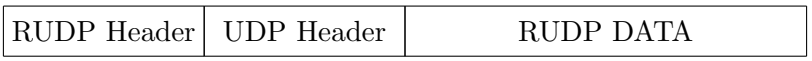
4. Four modes of RUDP are implemented: (i) Running mode for simply data communication; (ii) Debug mode for exception handling; (iii) Logging mode for protocol behavior observation; (iv) Stat mode for realtime performance monitor.

3.2 Protocol Segment Structure

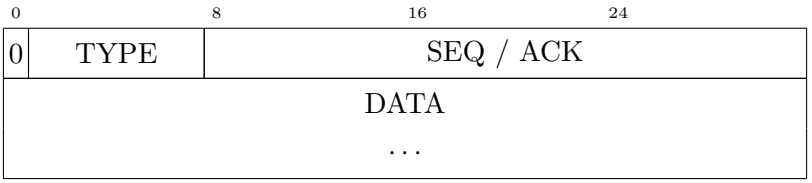
Inside the UDP packet, a RUDP header is included at the beginning of the UDP data field, as shown in Figure 1a. As the theoretical length limit of UDP is 65,535 bytes, the maximum size of RUDP is 65,507 bytes (65,535 - 8 byte UDP header - 20 byte IP header). As can be seen from the Figure 1b, RUDP has three main fields:

1. TYPE: Type is a 7-bit field to indicate the role of this packet. Three different types are shown in the Figure 2, and four more idle bits are reserved for further development:

    - DAT: A data packet.



(a) RUDP uses UDP as underlying transport layer protocol



(b) RUDP Protocol Segment Structure

Figure 1: RUDP Packet

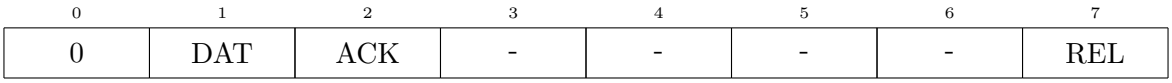


Figure 2: RUDP Packet Type

- ACK: A acknowledge packet corresponding to a reliable data packet or a retransmitted data packet.
- REL: A boolean value to specify if this data packet is transmitted reliably or unreliably.

2. SEQ / ACK: This is a 24-bit field to specify the sequence or the acknowledge number of the packet. For a sender, each RUDP packet it sends to the receiver has an accumulated sequence number, where the acknowledged number in the corresponding ACK transferred from the receiver is then this sequence number plus 1. Particularly, a sequence number 0 is equivalent to a new data connection, and a configuration parameter which could be modified by the upper layer identifies the maximum acceptable sequence number for terminating a data connection.

3. DATA: According to the requirement given by NeP2P team, the limit of this data field is stipulated as 1KB (1024 bytes).

3.3 Class and Interface Design

To resemble the interfaces of standard socket, we first devise a new class rudpSocket to combine the RUDP interpreter written in version 1 and also the packet processing operations refined in version 2. Moreover, multi-threading is needed to support concurrent data communication on the same port and non-blocking acknowledgment handling. More importantly, the peer using RUDP socket has to maintain and update RUDP status of all the connected remote peers. Therefore, the RUDP socket header is simplified and optimized for multiple senders and receivers data delivery, which is no longer connection-oriented and compatible with both reliable and unreliable transmission. The detailed rudp module is presented in Table 1.

Module Name	rudp
Dependencies	gevent, struct, time, Queue, collections and rudpException
Constants	MAX_DATA, MAX_RESND, RTO, SDR_PORT, RCV_PORT TYPE, DAT, ACK, REL, MAX_PKTID, MAX_CONN, ACK_LMT RUDP_DEBUG, RUDP_LOG, RUDP_STAT, STAT_PKTS
Functions	rudpPacket(pktType, pktId, isReliable, data) returns a RUDP packet encode(rudpPkt) returns a bit string decode(bitStr) returns a rudpPacket
Class	ListDict() a self-designed data structure to combine a list and a dictionary rudpSocket() rudpSocket.sendto(string, addr, isReliable) returns the number of bytes sent rudpSocket.recvfrom(isBlocking) returns duple (data, addr)

(a) rudp Module

Module Name	rudpException
Exception	NO_RECV_DATA no data is stored in the packet queue MAX_RESND_FAIL the sender fails to receive a ACK after resend three times ENCODE_DATA_FAIL, DECODE_DATA_FAIL codec failure WRONG_PKT a connection inside the receiver receives a wrong packet END_CONNECTION a connection has not received any data for a long time

(b) rudpException Module

Table 1: RUDP Version 3 Modules

3.4 Components

3.4.1 Data Structures

Efficiency or time complexity is always put in the first place throughout the design and implementation of our RUDP protocol. In order to manage multiple connected peers, receiving DAT and ACK packets from one single socket and handling multiple packet timeouts simultaneously, three different data structures are used to reduce the overall time complexity.

Queue

Queue is a First-In-First-Out (FIFO) linear data structure where the first element added in the queue will be the first one to be removed. Therefore, a queue can be used to store a set of tasks or objects where each of these elements should be handled or processed in sequence.

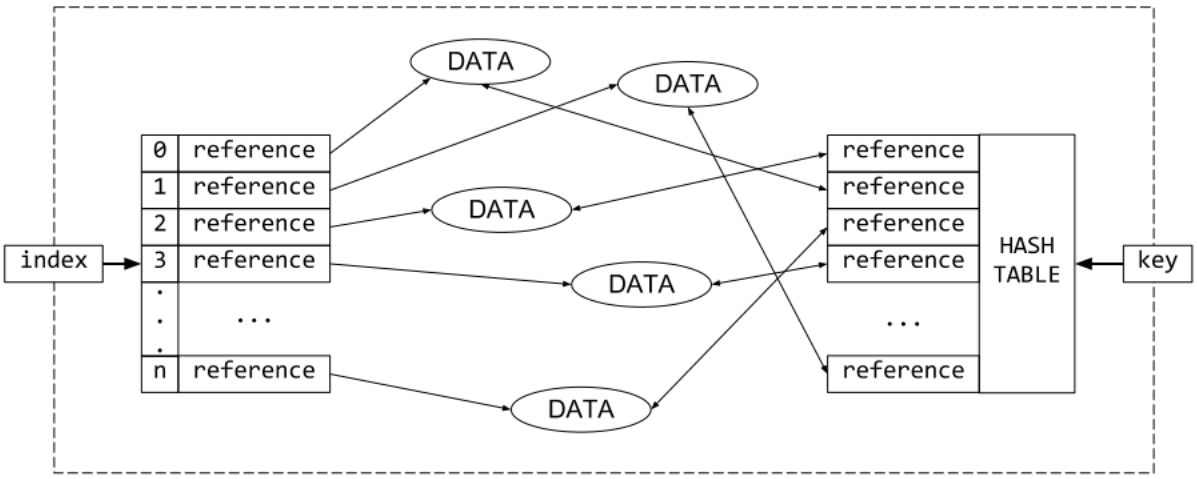


Figure 3: Data Sturcture: ListDict

OrderedDict

Collections.OrderedDict is a subclass of dict, a built-in dictionary data type given by Python [1]. Dictionary is an unordered set of data in which each element is indexed by a unique key and can be fetched within approximately O(1) time. Apart from the instant fetching advantage, OrderedDict preserves the order of all the keys so that the popping operation of the first or the last element takes exactly O(1) time to find the key and the corresponding (key, value) pair in the dictionary.

ListDict

As all the elements inside a list are in sequence, it takes O(1) time to select, add, delete and move one particular element by given the index number. We define our own data structure ListDict by combining the assets of dict and list. As shown in Figure 3, the reference of each data element is maintained twice in both a list and a dict. Moreover, once a data element is fetched based on a given key, the corresponding reference will be moved to the end of the list. So, the last reference in the list always refers to the latest operated data element. Also, each ListDict has a limited number of elements and the oldest data element will be removed when a new element is inserted into a full ListDict.

3.4.2 Peer State Objects

By taking advantages of these aforementioned data structures, the sets of senders, receivers and ACKs are collected accordingly and processed efficiently.

expId

A RUDP socket stores a **ListDict** typed collection of (address, expIdList) pairs of data where the address is uniquely identified by an IP address and a port number. And the expIdList presents all the packet sequence numbers that are acceptable for this RUDP socket sent from the address in the key, where the last ID is the next expected sequence number. In case of packet loss, other IDs kept in this list match with unreceived packet sequence numbers for accepting delayed or resent DAT packets. With the help of expIdList, a sender could send data in an unblocking manner and resend timeout packets adaptively. Figure 4 illustrates the updating operations of a (address, expIdList) pair.

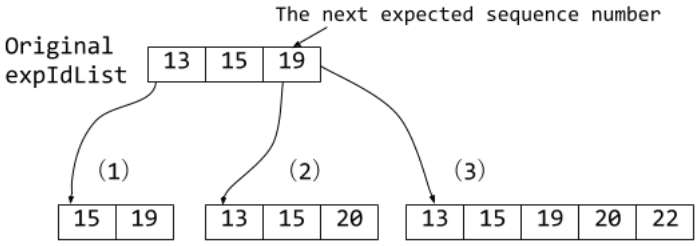


Figure 4: expId ListDict Operations: (1) A DAT packet with sequence number 13 is received, packet ID 13 is removed from the list. (2) A DAT packet with sequence number 19 is received. The last element of the list is increased by one to accept the next DAT packet with sequence number 20. (3) A DAT packet with sequence number 22 is received, but packet 19, 20 and 21 are lost. So the list is extended to store all the remaining unreceived packet IDs, and the next expected sequence number is incremented to 22.

nextId

A RUDP socket stores a **ListDict** typed collection of (address, id) pairs of data where the address is uniquely identified by an IP address and a port number. And the id should be included as sequence number inside the next DAT packets sent from this RUDP socket to the address in the key. By keeping this set of data, a RUDP socket could send packets to different remote peers.

notACKed

A RUDP socket stores an **OrderedDict** typed collection of ((pktId, address), (timestamp, resend number, resend packet)) pairs where the address is uniquely identified by an IP address and a port number. The pktId is corresponding to a ACK number that is one larger than the sequence number of the resend packet. The timestamp is used to keep track of the duration from the last retransmission of the packet and a MAX_RESND_FAIL exception is raised if resend number exceeds MAX_RESND. Therefore, the first element of this ordered dictionary is the next packet to be resent. After retransmission, timestamp is updated to the current time and the pair is moved to the end of the list.

datPkts

A RUDP socket stores an **Queue** typed packets. Each time the underlying UDP socket receives a RUDP packet, it will be added to the tail of the Queue if its TYPE is DAT. When the function rudpSocket.recvfrom() is invoked, a DAT packet from the top of the Queue is removed as the received data. By including this intermediate buffer, a RUDP socket could dispatch DAT and ACK packets separately.

3.4.3 Threads

As mentioned above, to support sending and receiving packets simultaneously on one port and within one process, three 'threads' or coroutines are needed as follows:

1. Main 'thread': this thread is used to communicate with upper layer. The two main interfaces are rudpSocket.sendto() and rudpSocket.recvfrom().

2. recv Loop: this thread continually ask the lower transport layer if UDP socket has received some data or not. If a UDP packet has been stored inside the socket buffer, it will be added to the **datPkts** queue if the packet TYPE is DAT and the corresponding list in **expId** is

updated. Otherwise, a ACK typed packet will be used to remove its corresponding entry inside the **notACKed** ordered dictionary.

3. ACK Loop: this thread handles timeout packets sequentially and periodically. The ordered dictionary **notACKed** stores all the not ACKed packets in the order of timestamp. Therefore, RUDP socket could resend packets from the top of notACKed and suspend or sleep a period of time until the next packet is timeout.

3.4.4 RUDP Modes

To implement an open source protocol that is easy for people to get started on, Running mode, Debug mode, Logging mode and Stat mode are defined for different purposes. To enable one or more modes, just configure the options inside the program: RUDP_DEBUG, RUDP_LOG, RUDP_STAT.

Running Mode
This mode simply run the program and work properly with pre-configured parameters. All of the errors or exceptions are triggered in Python built-in manner, and can be handled by the upper layer through try ... except ... statements.

Debug Mode
Apart from the functions of Running mode, this mode output every detail of a RUDP socket, such as connection flow, data transmission and error handling in stdout.

Logging Mode
This mode works similar to Debug mode, but record each events in the style of '[timestamp, type, message]', and message entries of packet-triggered events include information about packet length, packet id and packet address.

Stat Mode
This mode works like a throughput monitor to collect and output RUDP statistics of each node and the whole network. A class Recorder is used to keep updating these statistical data as shown in Table 2:

Class Name	`Recorder`
Attributes	`lastTime, perStat, ttlStat, timeStat, ttlTime, period, onperiod`

Methods	`updateSend(addr, length)`
	` update sending stat data.`
	`updateRecv(addr, length)`
	` update receiving stat data.`
	`updateLoss(addr, length)`
	` udpate lossing packet stat data.`
	`updateOnTime()`
	` udpate all statistical data all together every period of time.`
	`printPeriod()`
	` output statistical data in stdout.`
	`printTTL()`
	` output unit statistical data in stdout.`

Table 2: RUDP Recorder Class

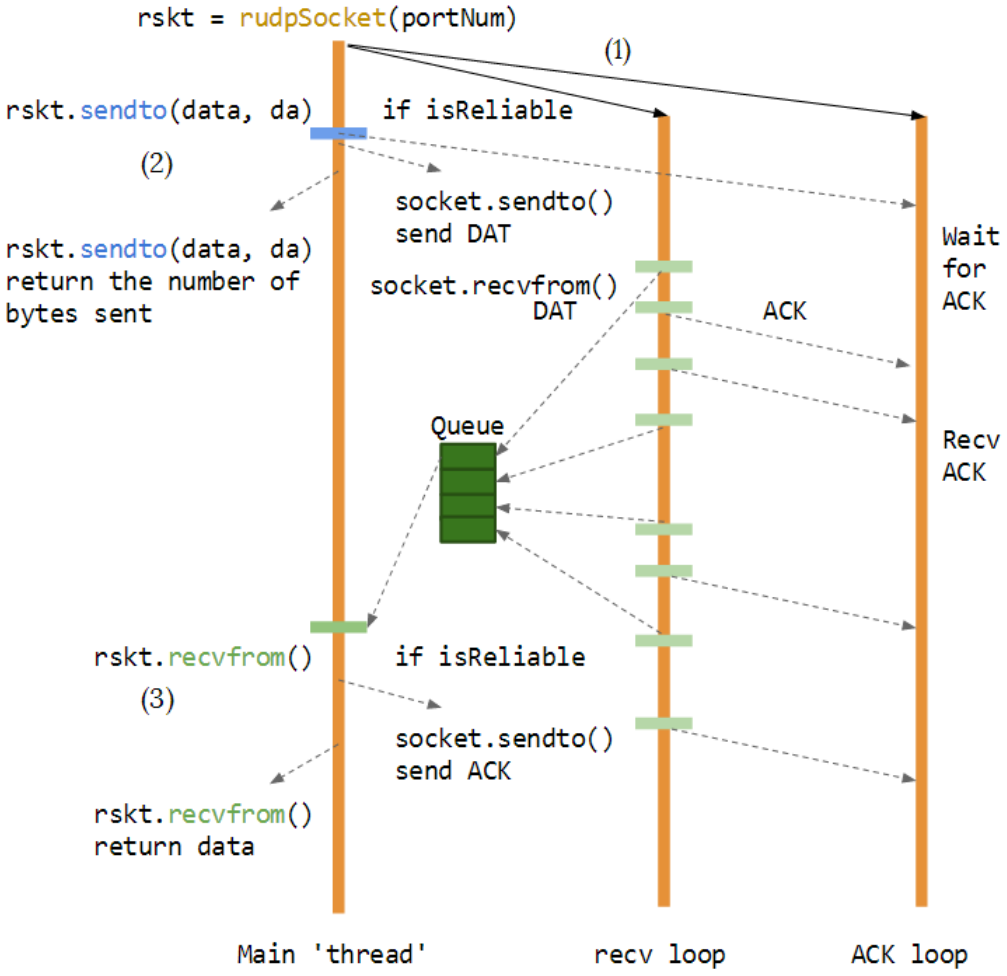


Figure 5: The workflow diagram of RUDP. (1) When a rudpSocket is initiated, the recv Loop and ACK Loop 'threads' are also started to handle receiving packets and ACK. (2) When function rudpSocket.sendto() is invoked, the underlying socket will directly send out the packet if reliability is not requested. Otherwise, the packets is stored inside **notACKed** to resend packets in case of timeout. (3) When function rudpSocket.recvfrom() is invoked, it will first query the **datPkts** queue if there has been data pacekts stored in this buffer. The function is blocked in blocking scheme if datPkts is empty, otherwise the data is returned. If reliability is required, a ACK would be sent accordingly.

3.5 Operations

Since each peer is both a sender and a receiver at the same time, two fundamental operations, shown in Figure 5, are needed to achieve this objective supported by multi-threading:+ i) sending operation; ii) and receiving operation. And Figure 6 depicts the flowchart of these two operations.

3.6 Evaluation

3.6.1 Testing Cases

A complete set of testing cases in provided by APPENDIX LINK. And all these testing scenarios are designed based on the basic behaviors of RUDP, where each of them is included inside a category listed below:

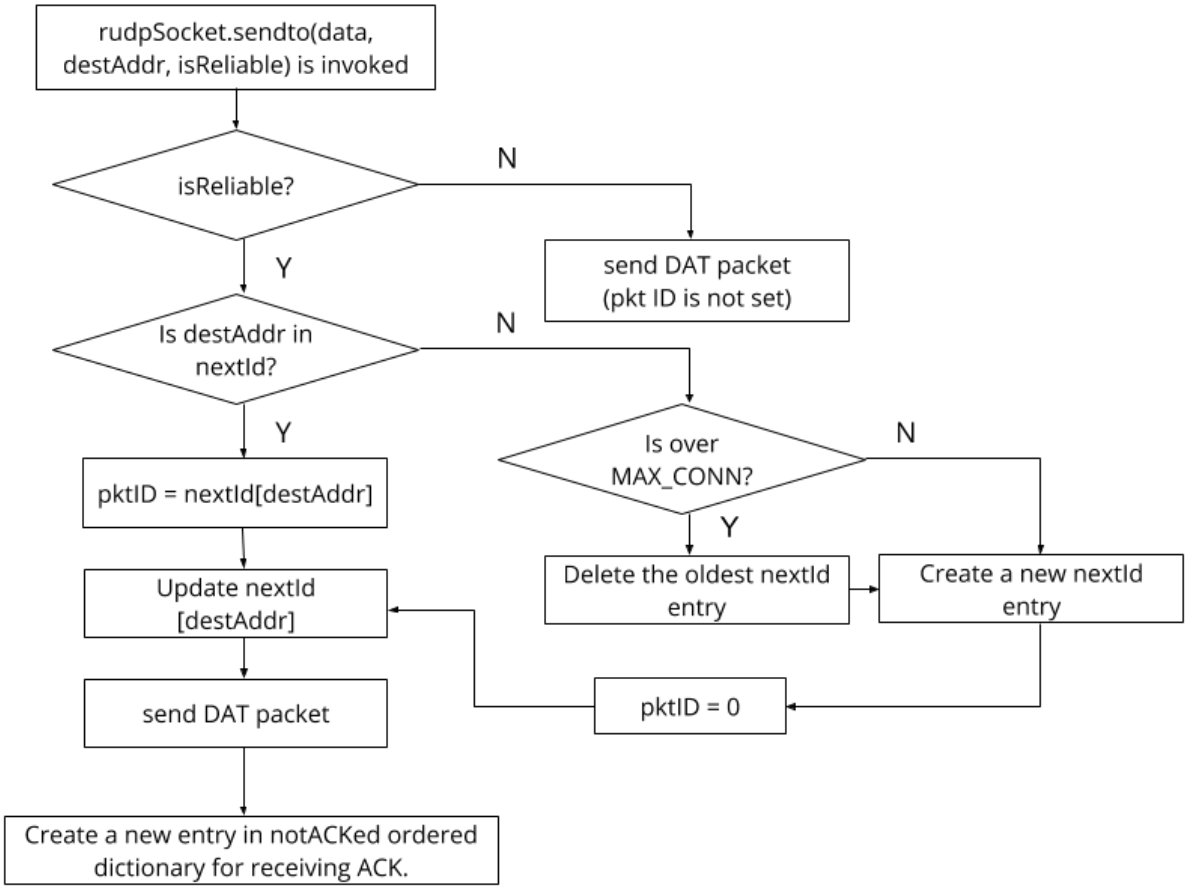
1. Socket errors: this category basically examines the operations with gevent.socket or a standard Socket given in Python library.
2. Sending errors: this category shows how a peer should run reacting to sending operations such as sending buffer is full or out-of-order packet ids.
3. Receiving errors: this category is similar to the Sending errors but dealing with receiving operations such as un-issued host address and invalid ACK packets.
4. Retransmission errors: this category includes operations when non-ACKed packet timeout or connection termination is triggered.

3.6.2 Single Peer

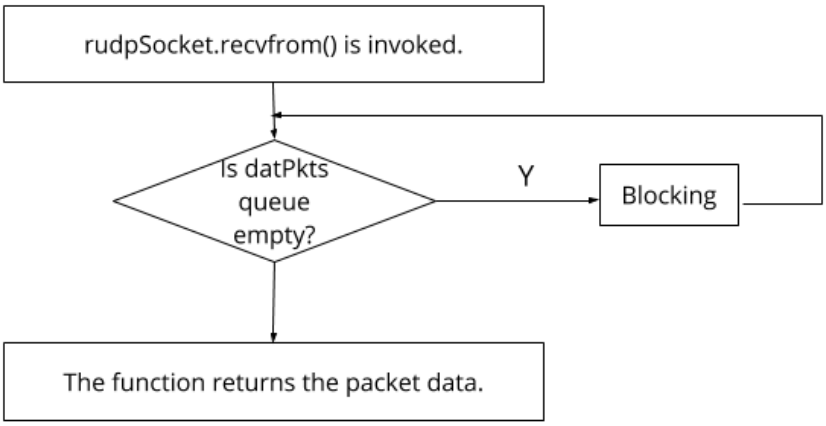
As a preliminary evaluation, an isolated peer is simulated to test the functionality of our protocol. As shown in Figure 7a, there are three major components in this experiment: the sending terminal, the receiving terminal and the target peer. The sending terminal is responsible for creating a MAX_ACTIVE_CONN number of sending peers, where each peer transmits a random number of data packets to the target peer. And a sending peer is randomly picked for sending DAT packets until all packets of sending peers are sent out.

The receiving terminal is used to create a bunch of peers listening to a range of port numbers and receiving data from the target peer. A single process is used to create receiving peers and call rudpSocket.recvfrom() periodically for each peer. Essentially, the target peer is the combination of sending and receiving terminals. Therefore, we specify the same settings for sender and receiver modules inside the target peer.

To test the basic functionality of N-to-M RUDP protocol, we conduct a simple experiment on a stand-alone Ubuntu machine. We specify μ, σ of number of data packets, and λ of packet delivery interval to be 100, 50 and 500 respectively for both the sending terminal and the sender module in the target peer. The target peers will send and receive data to and from 100 peers respectively in concurrent manner. Under this setting, the target peer works well without any MAX_RESND_FAIL exceptions. Sending and receiving terminals also work fine without any exceptions. Therefore, we believe that basic functionality of this protocol is achieved under such normal testing conditions.

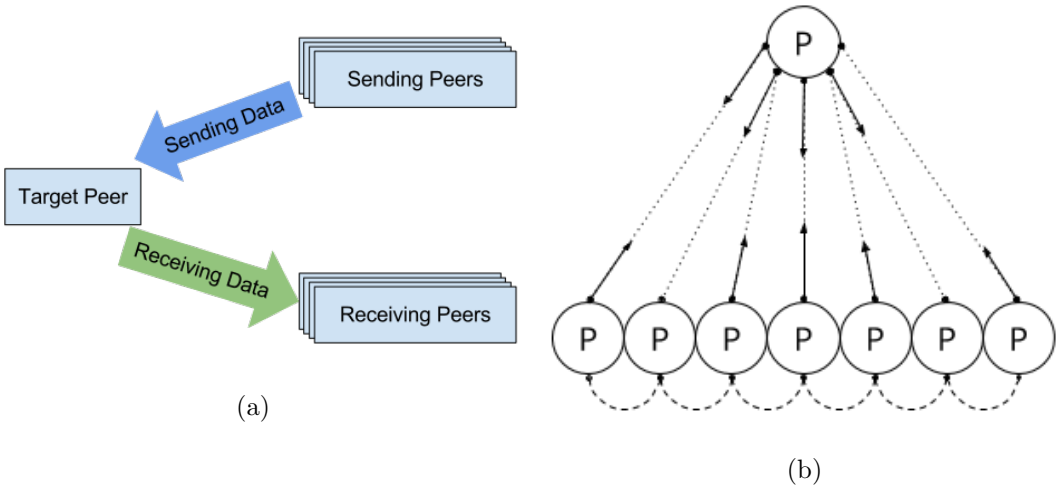


(a) rudpSocket.sendto() Operation



(b) rudpSocket.recvfrom() Operation

Figure 6: RUDP Version 3 Flowchart



(a) (b)

Figure 7: Single Peer and Multiple Peer Experiments. (a) RUDP Single Peer Experiment (b) RUDP Multiple Peers Experiment: the inter-sending time of peer is set to be exponentially distributed with λ, and a peer randomly select some peers to send data where the job size of is normally distributed with mean μ and standard deviation σ.

3.6.3 Multiple Peers

In addition to the single peer functionality evaluation, a more general multi-peer experiment is carried out to test the protocol in a practical P2P network. We first pre-install RUDP and the testing program on a number of peers like a pool, with each of them selecting a random portion of hosts in the pool for data delivery periodically. Figure 7b shows the workflow of this experiment.

A sample testing result 8 fetched from Stat Mode of RUDP shows that four different peers running on public hosts with addresses ('128.223.8.112', '128.111.52.63', '72.36.112.79' and '200.0.206.203') could communicate with each other based on the experiment settings: λ is 100, μ is 10240 and σ is 100. Essentially, a peer send out 100 packets per second and the average packet size is 10240 bytes. Also, the ratio of reliable data packet is 20%. Under this setting, the four peers work well without any MAX_RESND_FAIL exceptions, and all the lost or timeout packets are timely retransmitted.

3.7 Discussion

As the interfaces of rudpSocket is initially designed to be exactly the same with those in standard UDP sockets, integration with NeP2P is very smooth, by replacing Python built-in library sockets used in NeP2P with ours. For NeP2P version a16, we have combined the entire NeP2P system with RUDP for proper unreliable data transmission and reliable control message delivery, and every modification of NeP2P source codes has been recorded. However, as RUDP is connectionless and our design is not security-oriented, all the incoming packets will be processed no matter they are expected or not. So our RUDP is very vulnerable to intentional packet flooding attack in practice.

IP	Packets sent	Packets received	Packet Loss
'128.223.8.112'	8088 Packets 82194049 Bytes	3412 Packets 34670455 Bytes	137 Packets
'128.111.5263'	1140 Packets 11584276 Bytes	5360 Packets 54468243 Bytes	11 Packets
'72.36.112.79'	8098 Packets 82300724 Bytes	3402 Packets 34577004 Bytes	136 Packets
'200.0.206.203'	1485 Packets 15087665 Bytes	5515 Packets 56051361 Bytes	11 Packets

(1) Summary of each node.

KBps	'128.223.8.112'	'128.111.5263'	'72.36.112.79'	'200.0.206.203'	All
'128.223.8.112'	-	34.7 6.0	36.1 30.5	36.4 5.7	104.4 44.0
'128.111.52.63'	3.7 38.2	-	4.3 35.6	3.7 7.3	15.7 73.6
'72.36.112.79'	32.0 36.6	33.5 4.6	-	33.2 4.3	105.0 44.1
'200.0.206.203'	4.7 35.3	6.3 4.0	4.5 32.6	-	19.4 72.2

(2) Sending and receiving speed of each node in a period.

Figure 8: A sample testing result of four peers. '128.223.8.112', '128.111.52.63', '72.36.112.79' and '200.0.206.203'

4 Node List Updating

In NeP2P, a peer maintains a list of active nodes (or peers) for each transmission that it will send data to or receive data from. The lower-layer services in NeP2P need to ensure that all nodes in the list are active and reachable. Hence, a light-weight NAT traversal solution specifically designed for NeP2P is highly preferred, and its main objective is to ensure the reachability of nodes in the list. We call it Node List Updating. In practice, this service exploits hole punching techniques and is implemented as an extension of Reliable UDP.

4.1 Existing NAT Traversal Solutions

There are three categories of NAT traversal solutions, namely hole punching techniques, traffic relaying, and NAT-device-configuring protocols allowing hosts to alter the NAT devices' behaviors to allow desired connections on demand.

As a popular technique in P2P software and VoIP telephony, hole punching aims to establish bi-directional data communication between hosts in private networks behind NAT devices. It is commonly deployed on top of UDP and typically requires a third-party helper that is reachable by both two contacting peers. The main role of the third-party helper is to inform two contacting peers of each other's external address. Upon notification from the helper, the peer will send out a UDP packet to the external address of the target, which will surely be blocked by the NAT device at the destination private network. However, this packet will punch a hole on its own NAT device that allows incoming data packets from the target via its external address. By doing so, two peers in private networks are able to communicate with each other.

Nevertheless, this technique is based on the *Endpoint Independent Mapping (EIM)* assumption that an internal host with a particular private IP and port will reuse the same mapping for subsequent packets. Despite the fact that NAT operating characteristics are not standardized, the EIM assumption is very common in reality. A good example is Session Traversal Utilities for NAT (STUN) [20] that allows an end host to discover its public IP address if it is behind a NAT. The main merits of hole punching lie in its simplicity, efficiency and effectiveness.

For the traffic relaying approach, a public relay server is required to direct the traffic between two end hosts. Traversal Using Relays around NAT (TURN) [9] is a protocol achieving this goal. Even though it is able to address NAT traveral problems regardless of NAT behaviors, this method is very inefficient and undesirable in P2P communication.

NAT-device-configuring protocols are a set of networking protocols allowing hosts to configure NAT devices. Internet Gateway Device Protocol (IGD Protocol) is available on some NAT routers and implemented via Universal Plug and Play (UPnP) [23]. Local hosts are able to retrieve the external IP address of the device and control NAT mappings. Another good example is NAT Port Mapping Protocol (NAT-PMP) [3]. In practice, NAT configurations are very compilicated and differ from each other, and UPnP and NAT-PMP may not be avalabile

4.2 Existing NAT Traversal Implementations

There are a lot of available open-source implementations of previously mentioned protocols, including RESTUND [18], PJNATH [14] and TurnServer [22], with their own pros and cons. To be compatible with NeP2P, a few criteria need to be identified. First, redundant features and library dependencies should be minimized. Second, it is desirable to be implemented in programming languages supported by Python (e.g. C or C++) or Python itself. Third, existing implementations need good documentation in order to be adopted for our use. In the rest of this section, we would like to mention a few implementation and compare their pros and cons.

We checked a few implementations of STUN and TURN. Restund provides modular and flexible server for STUN and TURN, with IPv4 and IPv6 support [18]. However, it is not suitable for us due to its limited documentation, feature redundancy and heavy library dependency . TurnServer is an another implementation for TURN, which is out-of-date and incompatible with NAT configurations though. It also depends on other libraries heavily.Other implementations include STUNMAN [21], STUNT [7, 6], STUND [11], PyStun [8].

As for UPnP and NAT-PMP, a lot of open sources are available for different purposes. The MiniUPnP project [10] supports both UPnP and NAT-PMP on both client and server side. It is a lightweight and well-organized library with good documentation and maintenance. However, from the perspective of NeP2P, we do not integrate it with our implementation but recommend it to NeP2P developers for future performance enhancement.

4.3 Scenario Formulation

In this section, we formulate the scenario in the context of NeP2P. Figure 9 shows the basic scenario for NeP2P, where there is a sender and many receivers. In NeP2P, the sender will generate unique batches continuously, and send them to one receiver in each time slot according to some scheduling scheme. Receivers will cache and exchange batches with other receivers. After a peer receives enough batches for decoding, its NeP2P client will return the decoded data to the upper-layer application, while its NeP2P service keeps operating until all receivers get the data.

It is obvious that NeP2P requires full-mesh communication among the sender and all receivers within a NeP2P pool. All address information about other peers will be maintained in their own node lists. Our objectives are to update the list upon peers arrivals and departures, as well as maintain NAT mappings during transmission.

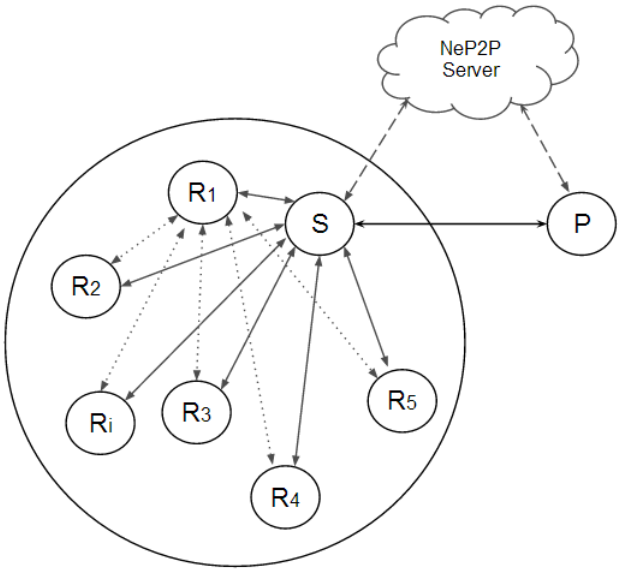


Figure 9: Basic scenario for NeP2P transmission. The big circle represents a NeP2P pool, a small circle denotes a peer (either a sender or a receiver). It contains a sender and multiple existing receivers. The sender will send out batches to existing receivers, and every receiver will exchange batches with other receivers. Hence, data communications within a NeP2P pool is full-mesh. When a peer arrives, it will first contact the sender via a NeP2P server. After that, the Node List Updating service will update the node lists accordingly and enable NAT traversal seamlessly.

4.4 Design

We base the Node List Updating on hole punching techniques. The main reason is that hole punching can be implemented in a decentralized and timely manner, which is highly desirable for NeP2P. For a particular NeP2P pool, the sender can serve as the third party to help establish connections among new receivers and exsiting ones.

4.4.1 Hole Punching

Figure 10 shows the basic operations of hole punching. It is assumed that NAT devices' behaviors on A and B's networks do not violate the EIM assumption, namely the same host with a particular internal IP and Port will always reuse the NAT mapping for subsequent packets. It is also assumed that Peers A and B are both connected with a mutual middle peer M. If not, a mechanism is needed to ensure it is happening, such as the existence of public tracker servers in BitTorrent to help a new peer to reach other peers with target resources.

To illustrate, we divide the hole punching into six basic operations. To start with, every peer needs to be assigned with a unique ID and register itself on some well-known peers or servers, namely Peer M in this case, once the service (*e.g.* NeP2P service) is started. If A

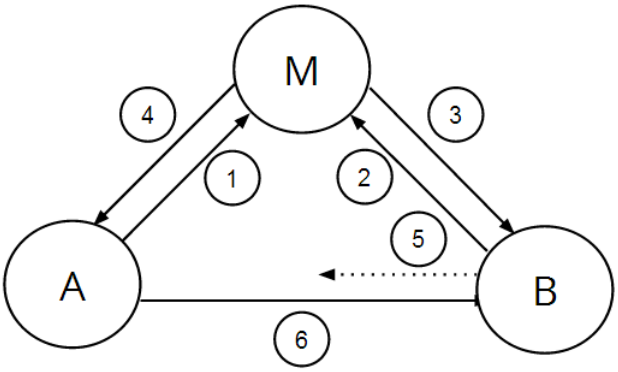


Figure 10: Basic operations of hole punching. (1) Peer A registers itself in Peer M, one of the public peers or servers, waiting to be connected by other peers. (2) Peer B sends a message to M requesting Peer A's external address. (3) With matching information, Peer M will reply Peer B with Peer A's external address. (4) Peer M also notifies Peer A with Peer B's external address. (5) Peer B sends a packet to Peer A's external address, creating a NAT mapping on NAT device. (6) Peer A sends a packet to Peer B, also creating NAT mappings for B. This packet will reach B successfully, so do their subsequent packets. Note that the ordering is not fixed in reality and vary from case to case.

wants to connects to B, it will request to be introduced to B via M. If B is also registered in M, Peer M will immediately reply A with B's external IP address and port number, and also notifies B of A's external IP address and port number. Once B receives notification, it will send out a packet to Peer A's address, and the packet is very likely to be blocked at A's NAT device. This is because A has never send a packet to Peer B at its external IP address, and no NAT mappings exist in A's NAT device for this communication. However, this packet will create a NAT mapping at B's NAT device, and A's packets will reach Peer B successfully. Once both A and B receive each other's data packets, hole punching is finished and bi-directional communication is established between A and B.

4.4.2 Procedures

To make it useful, we slightly adjust the hole punching to fit NeP2P scenarios. We use S to represent the sender, R_i the i-th existing receiver, and P the new peer. There are three basic stages:

1. **Receiver Arrival**: S acts as the middle peer, and R_i are already registered in S's node list. After receiving the request from P, S will reply with its own node list and also notifies R_i of P's arrival. Then, exactly as what will happen in hole punching, P will punch a hole for all receivers, and so will receivers. When every one is able to contact each other, they will update their node lists accordingly.

2. **Keep NAT Mapping Alive**: to avoid the expiration of NAT mappings, we need to let the internal host send out whatever messages (either data packets or Keep-Alive messages). For simplicity, we can set the duration to a relatively small value (*e.g.* 10 seconds). In addition, data messages will be tracked to avoid unnecessary Keep-Alive messages.

3. **Receiver Departure**: when a peer cannot receive either data messages or Keep-Alive messages from a remote peer for certain duration, the remote peer may be gone and there

0	1	2	3	4	5	6	7
0	DAT	ACK	**NAT**	-	-	-	REL

Figure 11: RUDP Packet Type with NAT Flag

is no need to continue punching the hole. In this case, the node list will be updated accordingly.

4.4.3 NAT Loopback

NAT loopback is a feature in many NAT devices that allow a user to connect to its own public IP address from inside the LAN-network. This feature is useful in NAT traversal, which enables hole punching to help two peers on the same private network to communicate with each other via the same external IP address. However, in practice, some NAT devices may not support NAT loopback. To address this problem, we require peers to disseminate their private addresses. If two contacting peers see the same external address, they can touch each others via privates addresses instead of external ones.

4.5 Implementation

We consider Node List Updating as a service offered to NeP2P clients, which is implemented as an extension complementary to Reliable UDP. We add the NAT Flag to the RUDP Header as shown in Figure 11.

Table 3 illustrates the class and interfaces of rudpNATSocket and natModule. The rudpNATSocket class is inherited from rudpSocket with simple interfaces provided for NeP2P clients. Since a host may launch multiple NeP2P clients, the appSocket ID is used to identify one NAT service from another, with respective roles (*e.g.* the middle peer/sender or the peer) and node lists.

The NAT module has four types of messages. MSG_NAT_CONNECT is used in the request message from the new peer. The sender will reply the new peer with MSG_NAT_REPLY and notifies other peers with MSG_NAT_ARRIVAL. MSG_NAT_KEEPALIVE indicates a Keep-Alive message.

4.6 Experiments

To test the functionality of hole punching, we try out the scenario in Figure 9. This experiments covers 6 networks including CUHK Hostel Wired LAN, CUHK Wi-Fi, PCCW Wi-Fi, eduroam, Univ. Wi-Fi and IE Wired LAN. Both wireless and wired networks near CUHK campus. We use a laptop with a public IP address as the middle peer.

Table 4 shows some testing results of hole punching near CUHK campus. We can see that this simple technique works most of the time, whereas fails for the peers behind the same IE wired LAN. This is because that the NAT device on IE wired LAN does not support NAT loopback feature, and blocks data from an internal host to another internal host via the external IP address. If we use private IP addresses instead of the public one, we will easily address this problem.

Module Name	rudpNATSocket
Super Class	rudpSocket
Overridden Methods	sendto, recvLoop
Added Methods	sendto, sendNATMsgto, proNAT
Added Interfaces	getNATService, startNATService, delNATService, getNodeList

(a)

Module Name	natModule
Dependencies	Socket, json, rudp
Roles	NAT_MIDPEER, NAT_PEER
Message Types	MSG_NAT_CONNECT, MSG_NAT_REPLY, MSG_NAT_KEEPALIVE, MSG_NAT_ARRIVAL
Class	natService(rudpSkt, peerRole, port) Methods: getNodes, delNodes, addNodes, connect, request, notify

(b)

Table 3: (a) rudpNATSocket (b) natModule

Host A	Host B	Peer Exchange Status
CUHK Hostel	CUHK Hostel	OK
CUHK Hostel	IE Wired LAN	OK
CUHK Wi-Fi	CUHK Wi-Fi	OK
CUHK Wi-Fi	IE Wired LAN	OK
PCCW Wi-Fi	PCCW Wi-Fi	OK
PCCW Wi-Fi	IE Wired LAN	OK
eduroam	eduroam	OK
eduroam	IE Wired LAN	OK
University Wi-Fi	University Wi-Fi	OK
University Wi-Fi	IE Wired LAN	OK
IE Wired L	IE Wired LAN	**Fail***

Table 4: Some testing results of the simple hole punching technique.

4.7 Discussion

The basic NAT traversal functionality is successfully realized in most network environments. Unfortunately, due to the limited diversity of real-world networks, we are not able to fully examine our light-weight NAT traversal solution. For non-EIM/symmetric NAT private networks, we may use TURN as the last resort. As for integration with NeP2P, we will further discuss with NeP2P developers to apply the Node List Updating service to NeP2P.

5 NeP2P Testing Platform

NeP2P is a brand new P2P system powered by Network Coding. From the perspective of software development, a good testing platform is very necessary and extremely meaningful for NeP2P developers. However, the testing of a P2P protocol is not that trivial as it needs to coordinate a large number of peers to create a realistic P2P environment, and also requires great efforts to perform the data analysis. In this project, we focus on the implementation of a flexible, simple, and scalable testing platform. Moreover, as NeP2P is still under development,

we aim to provide a useful and intuitive presentation toolkit to demonstrate peer behaviors under NeP2P.

5.1 Components

The testing platform consists of three components: **Master Console** (MC), **PlanetLab Nodes** and **Online Demo Website**. Figure 12 illustrates the architecture of the testing platform.

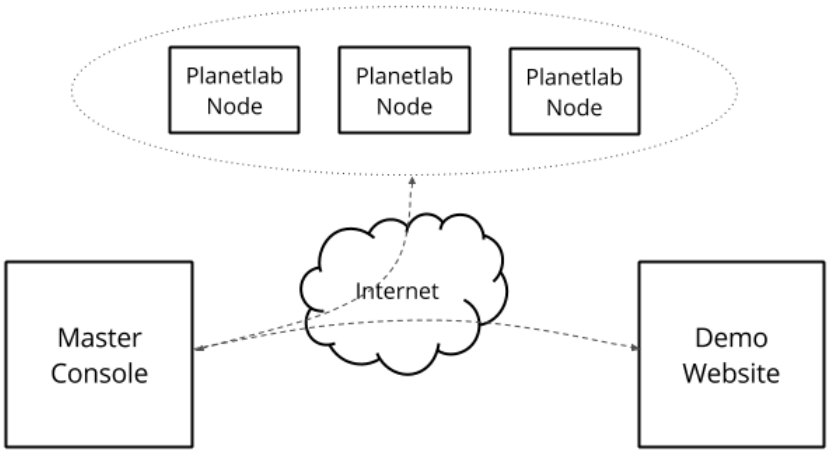


Figure 12: Architecture of NeP2P Testing Platform. The Master Console maintains virtual bi-directional connections with the PlanetLab network, controlling and monitoring the process of a test. Log data is generated and stored locally in every remote node for each test. The Master Console will fetch the log data from all nodes, and push it to the Demo Server for archiving and data analysis.

5.1.1 Master Console

The master console (MC) is no more than a terminal on any Unix machine (*e.g.* Ubuntu or Fedora) with public Internet connectivity. Its main job is to provide a simple interface for the developer to configure necessary parameter inputs for a test, provide centralized control over all remote PlanetLab nodes and synchronize fetched log files with the server hosting the Demo Website.

To realize the functionality of MC, a large number of Python and bash scripts have been written and deployed on every remote node beforehand. These scripts are sufficient to control the behaviors of an individual node, including library installation, code deployment, transmission with certain version of NeP2P. After a test is configured, MC will first validate the configuration files, generate separate configuration files for nodes with different roles (*e.g.* either a sender or a receiver), push files to individual nodes and execute commands in serial or parallel.

5.1.2 PlanetLab

Though PlanetLab is prevalent for research use, we have encountered several challenges in dealing with this network. One big challenge is to solve the heavy library dependencies and incompatibility problems, since the operating system of PlanetLab nodes is Fedora Release 8.

In addition, as its bit version (32-bit) is not consistent with MC and our own lab server, we also need to prepare some pre-compiled binary libraries with different bit versions beforehand. Logging onto a remote node is very simple, whereas nodes may not be online from time to time. Hence, for a large-scale test, we need to confirm the availability of every node before the test.

5.1.3 Online Demo Website

To better understand NeP2P performance, an intuitive online demo platform is highly desirable to present numerical information in a pictorial or illustrative form, allowing better comprehension of the data. More importantly, such a website can achieve the goal of data archiving with global access.

The architecture of the Online Demo Website is illustrated in Figure 13. It consists of three basic components: i) Express.js framework: a Node.js web application platform supports HTTP request handling, url routing and Connect middleware; ii) Jade template engine: a Node template engine for quick HTML markup generation; and iii) nvd3.js charting engine: a re-usable charting components and charting toolkit based on d3.js.

There are three main features of this Online Demo Website. First of all, it is very **interactive**. Thanks to the charting engine, all charts are handled dynamically by Javascript event handlers, and respond to viewers' actions interactively. For instance, the viewer can compare the sending bandwidth use among any peers in a line chart by hiding or showing others, switch between various chart modes to dig out more useful information and even zoom up the line chart for more details. Figure 14 demonstrates how the interactive webpages are generated.

Second, it is very **efficient**. Since statistical log data is independently stored, users can either download raw data through url link directly or browse well-illustrated charts where data is loaded via Data Retrieval AJAX requests on demand. Also, all log data is encoded in standard JSON format and can be easily handled by Javascript program.

Third, it is very **flexible**. As the log data is well formatted, adding new charts is as simple as combining required data with various chart plotting methods. Such flexibility can definitely help NeP2P developers to generate other meaningful charts for data analysis.

5.2 Functions

5.2.1 Centralized Control

The testing of NeP2P requires complete centralized control over remote peers, including testing setup, starting and ending of a test, code updating and data fetching. This control functionality is realized at the Master Console.

Our control commands are very simple to use. Different actions are implemented in terms of roles (*e.g.* a sender or a peer), tasks (*e.g.* start, check and end etc.) as well as execution modes (*e.g.* serial or parallel). By doing so, we can easily specify any task that needs to be done by certain nodes. Besides, all functions are implemented in fabfile.py as shown in Table 5b. Their interfaces are well-defined and support future extension.

5.2.2 Efficient Deployment

It is challenging to deploy updated codes and push specific configuration files onto a large number of remote nodes for each test. To address this problem, one possible solution is to

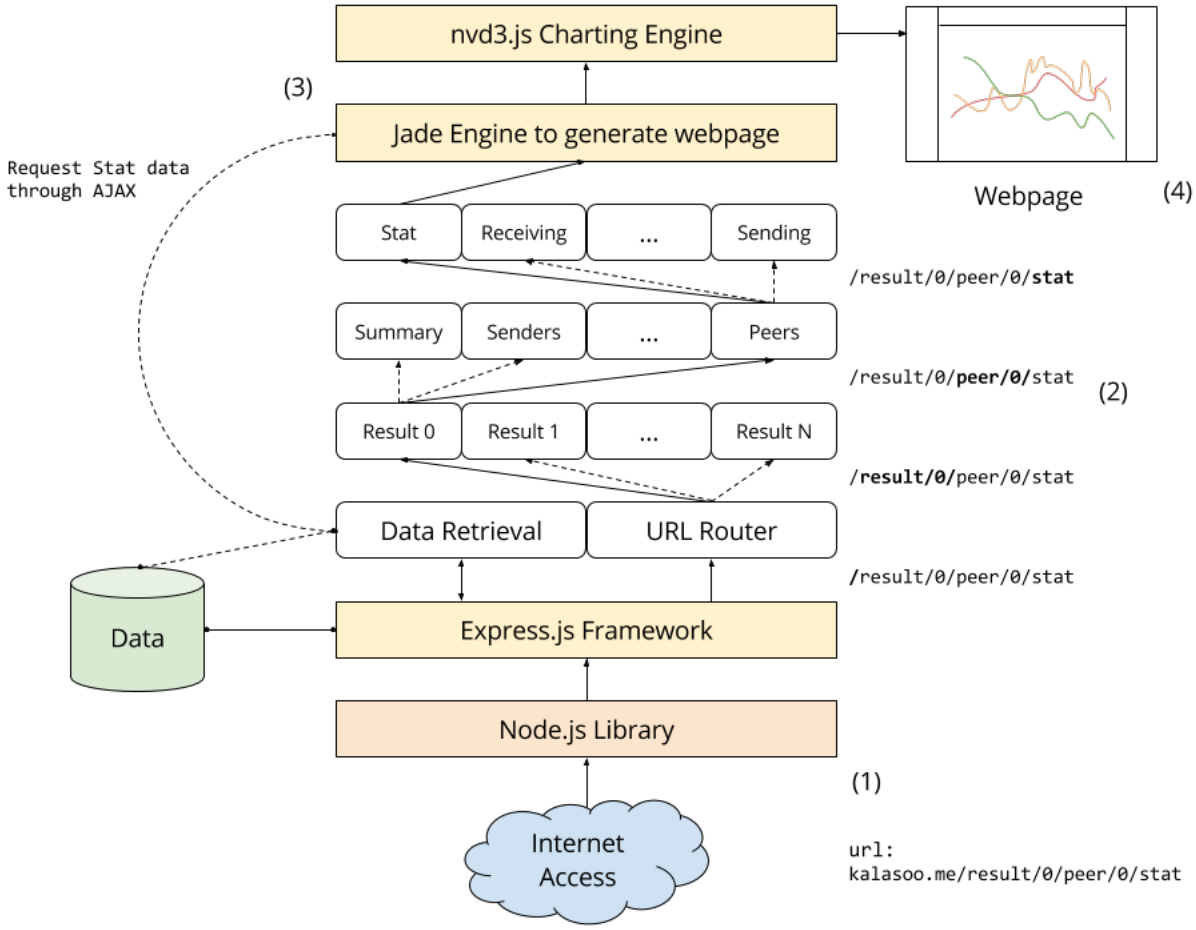


Figure 13: The workflow of Online Demo Website. (1) The website is programed in node.js and run on YIN MING's Amazon EC2 server with domain name `kalasoo.me`. (2) The Express.js is a node.js web application framework and redirect the url with a router module. As shown in the picture, [result/0] shows that the web user is going to find the Result 0 experiment data set and [peer/0/stat] indicates that statistical results of Peer 0 is shown to the user. (3) After redirection, jade template engine is used to compose a HTML website, while all the real data is loaded through AJAX for quick webpage rendering. (4) At last, the chart is plotted on website with nvd3.js charting engine.

compress the most updated codes and make it available for every node to download. However, as there may only be one or two files changed for debugging, downloading every file is not that efficient.

An alternative solution is to maintain the source codes on Github and instruct nodes to execute "git pull" every time. Nevertheless, git commands would also fetch unnecessary files such as commit history for tracking purpose. Hence, to avoid the above problems, we prepare a bash script that instructs nodes to download only changed files for each test. The Master Console just needs to tell every node to fetch and execute the script in parallel.

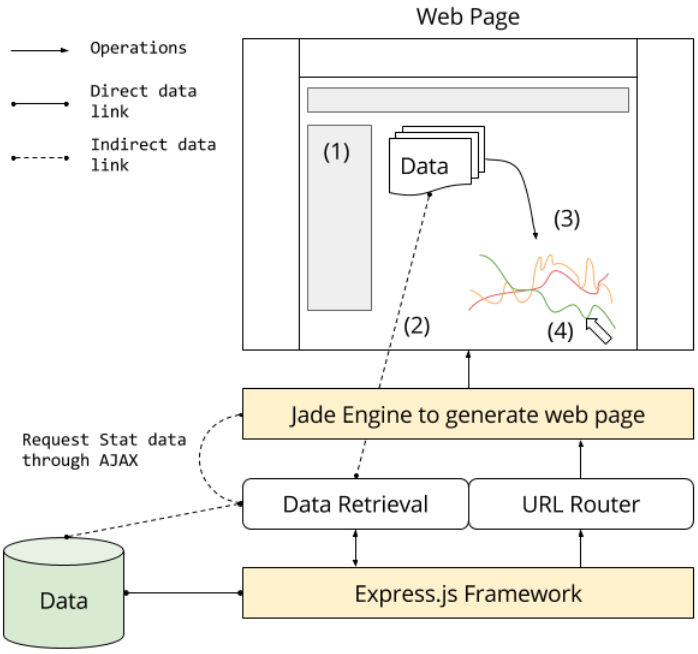


Figure 14: The workflow of Web page construction. (1) The basic grid structure, native components and fixed data are rendered with HTML and CSS in browser. (2) The statistical data is retrieved accordingly through AJAX requests to server-side Data Retrieval interfaces.(3) Once the data is loaded successfully, the plotting toolkit starts to interpret data sets and plot charts piece by piece as svg nodes. (4) Each svg object is bound with one or more interactivity events so that users could operate with the chart.

5.2.3 Data Logging

Data logging is essential for performance evaluation. In accordance with [16] and [17], logging data are discreetly selected to draw outcomes in the following testing dimensions: i) Scalability: time needed for a group of peers to download a file. ii) File-sharing efficiency: how each peer inside the system could efficiently utilize its bandwidth. iii) Protocol efficiency: the ratio between control messages and unit data throughput. And We aim to implement data logging function in an effective and efficient way, with focus on the bandwidth use, downloading and decoding performance.

The most important information for a peer is *the number of packets sent to and received from a particular peer per second*. Such information is extremely useful in the following aspects. First, since data packets are transmitted unreliably via UDP, we need to compute *packet loss rate* for any two peers by synthesizing this information from all peers in a test, so as to evaluate *congestion control* in NeP2P. Second, we can evaluate *scheduling scheme* by analyzing the pattern of *outgoing data traffic*. Third, by analyzing *incoming data traffic*, we can get the distribution of different packets from different sources.

It is also important to log down *timing and decoding information* for an app worker. We can easily extract *downloading duration* based on the timing information, and understand *decoding performance* by logging down the number of bytes received and decoded per second. The above first-order statistics are very essential for our data analysis.

Figure 15: Four different types of charts. (1) A bar chart shows relationships between different data series. (2) A stack graph is meant for visualizing the total change over time of a group of quantities. (3) A pie chart compares parts to a whole, and the how each segment dominates among others. (4) A line graph helps audience understand the trend each data series and how each of them correlate with each other.

5.2.4 Data Visualization

"A picture is worth a thousand words", we have utilized data visualization tools to present data as graphs and charts because that they could show more information than pure quantities comprehensively. Figure 15 shows four types of charts used in this testing platform.

Specifically, a bar chart is used to compare the total receiving and sending packets for a peer with another involved NeP2P host which may be a sender or also a receiving peer. And the stack graph shows the unit bandwidth use over time of each host which represents the peer behaviors. Moreover, to summarize all these packets communication, the receiving and sending packets are categorized by IP addresses to show the weight of each part in pie charts. Finally, the sending speed of the sender towards different receivers is plotted inside a line graph so that the scheduling behavior of NeP2P service is thoroughly depicted.

5.3 Implementation

5.3.1 Class and Interface Design

There are three core modules for testing: test_obj module, fabfile module and node control module. Table 5b shows the class and interface design for Fabric Master Console Module. For easy comparison between NeP2P and other P2P system (*e.g.* BitTorrent), we make it a base

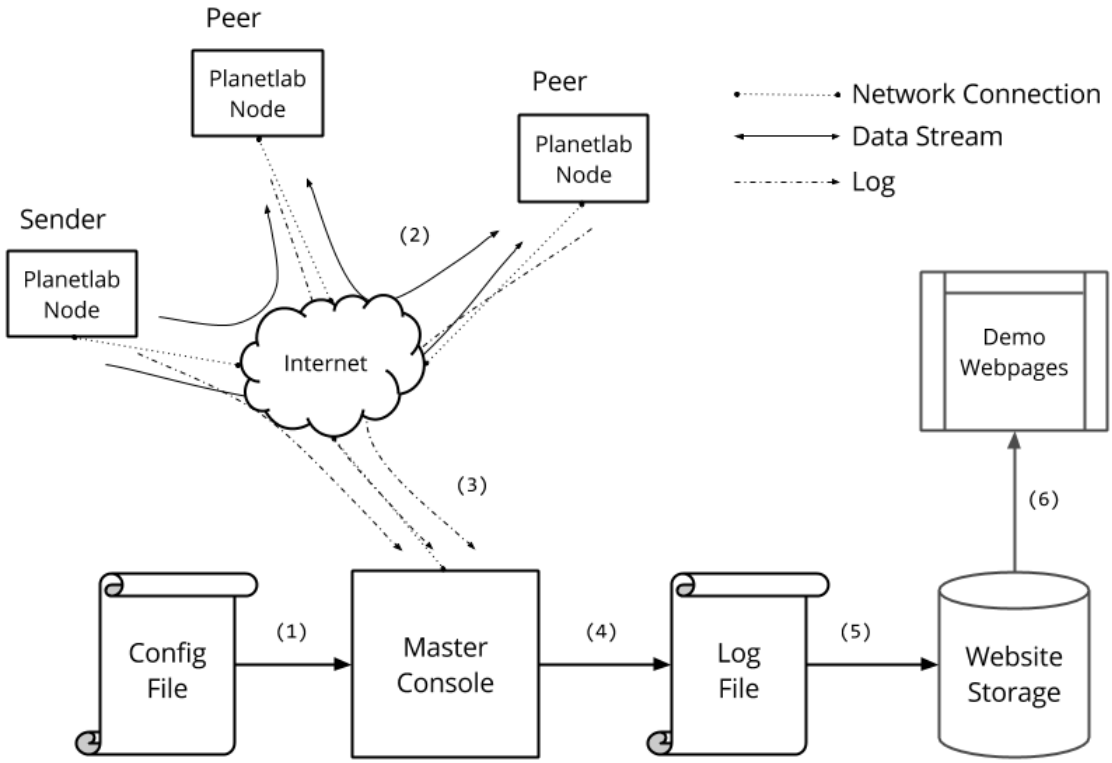


Figure 16: Workflow of the testing platform. (1) Specify the configuration file for a test; (2) the Master Console checks the configuration inputs, pushes files onto remote nodes, deploys latest and starts the transmission; (3) when the transmission in finishes, the Master Console get log files back; (4) all log files are well-archived; (5) the platform synchronizes log files with the website server; (6) the Online Demo Website loads data and generate reports online.

class with proper interfaces including common tasks of a P2P test. For example, it has to check if the test configuration is valid or not, add testing hosts to this master console and then execute a full workflow of testing. As can be shown in Table 5a, Nep2pTest subclass and BtTest subclass are designed to handle tests respectively, and the TObj base interface could be easily extended for other P2P systems.

5.3.2 Operations

Figure 16 provides a good illustration for the workflow of our testing platform and Figure 17 gives detailed flowchart of each execution option. To conduct a test, we need to follow the following steps. First, we need to specify necessary parameters in config.json, including the NeP2P version, the file size, the path for log files, IDs of peers and senders, etc. Second, instruct remote nodes to switch to a specific version of NeP2P, or fetch the latest codes from Github. Third, start the NeP2P service at remote nodes in parallel. After that, we can create NeP2P clients and start the transmission immediately. By observing outputs at the sender and peer terminals, we can spot any possible exceptions or errors for debugging. When the transmission is finished, the sender will be notified via NeP2P control messages, and we can stop NeP2P services to save logged statistics. At last, we need to instruct every nodes to merge all log files into a standard Json format and fetch them remotely. Up to this point, a test is completely finished.

Module Name	test_obj.py
Dependencies	json, fabric.api
Base class:	TObj Class: an interface to define common testing methods. Parameter: config => a dictionary to store configuration values. Methods: check_config, add_hosts, setup, start, end, getlog.
Sub class:	Nep2pTest Class: a Nep2pTest subclass of TObj to define and run a NeP2P Test. Parameter: config => a dictionary to store configuration values. Added Public Methods: check, clean. Private Methods: gen_nep2p_files, gen_ndoes, gen_nep2p, gen_config
Sub class:	BtTest Class: a BtTest subclass of TObj to define and run a BitTorrent Test. Parameter: config => a dictionary to store configuration values. Added Public Methods: show, show_daemon, show_bt, check, clean_daemon, clean_file. Private Methods: gen_config

(a) Fabric Test Class

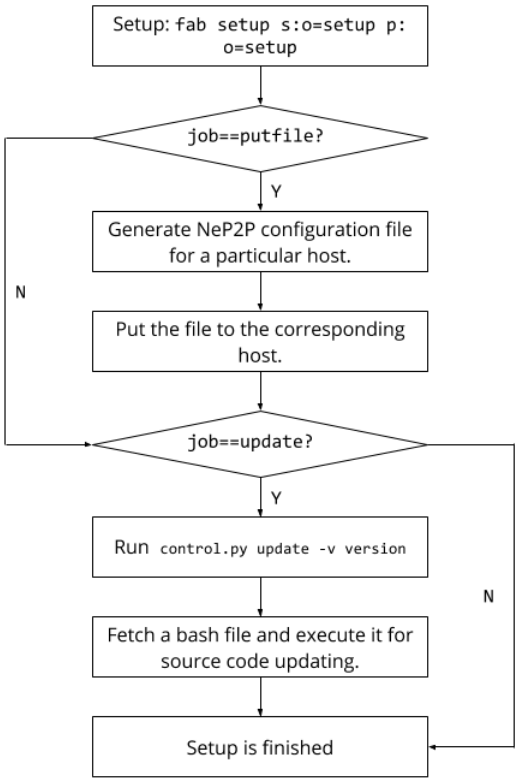
Module Name	fabfile.py
Dependencies	json, test_obj
setup	setup(t, cf) Method: to setup a p2p testing. Parameter: t => P2P system type ('bt, 'nep2p') Parameter: cf => configuration file name.
sender	s(o, job) Method: to start an operation of senders. Parameter: o => To invoke tObj.o(True, job) for certain options. Parameter: job => To assign jobs for option specified by o.
peer	p(o, job) Method: to start an operation of peers. Parameter: o => To invoke tObj.o(False, job) for certain options. Parameter: job => To assign jobs for option specified by o.

(b) Fabric Master Console Module

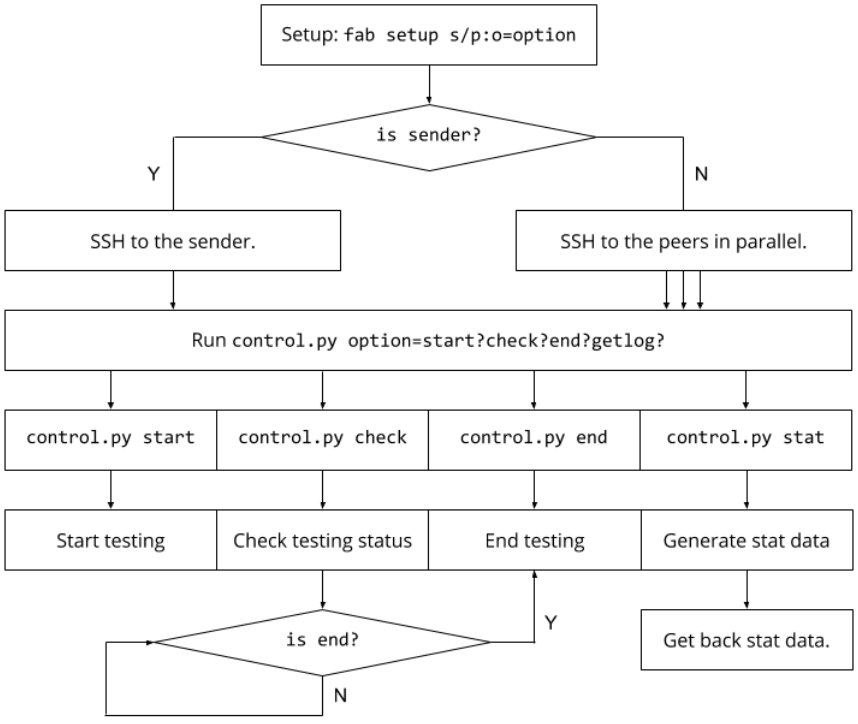
Module Name	control.py
Dependencies	json, argparse, subprocess
usage	control.py [-h] [-f DATAFILE] [-r s,c] [-v a13, a16]
positional arguments	update,start,check,status,end,quit,stat,clean Control the actions of a node.
optional arguments	-h, --help : show this help message and exit -f DATAFILE, --dataFile DATAFILE : test file to send -r s,c, --role s,c : NeP2P service or client -v a13,a16, --version a13,a16: version codes

(c) Node Control Module

Table 5: Class and interface design.



(a) The flowchart of Sender and Peer setup.



(b) The flowchart of Sender and Peer Testing Operations.

Figure 17: Sample results of statistical data.

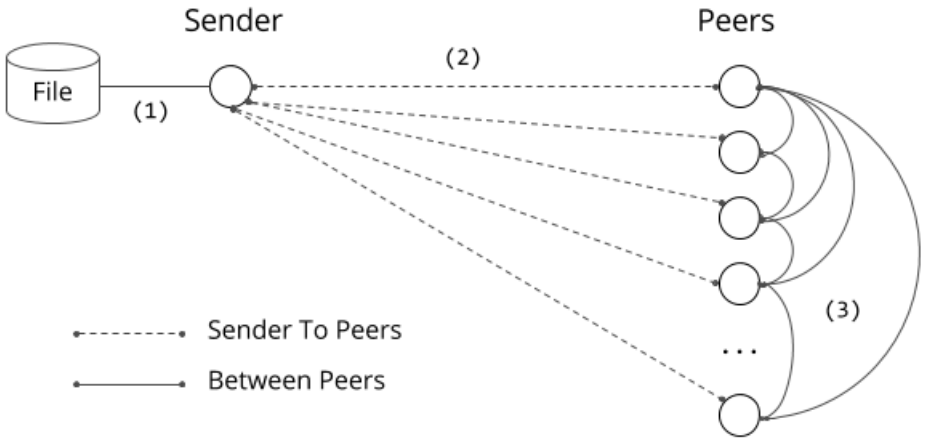


Figure 18: File Sharing Testing Scenario. (1) A file is deployed to the sender, and all peers prepare to download the file. (2) The requests are initiated, the sender construct data connections with each peer. (3) Peers start to communicate with other for propogating more decodable data.

5.4 Sample Testing Results

In this section, we use our testing platform to conduct sample tests of File Sharing [4] to demonstrate its functionality, and also point out various evaluation dimensions that can be potentially explored based on the collected logging data. As NeP2P is still under development, we hope this platform would help developers to understand peer behaviors under NeP2P for performance optimization. Figure 18 shows the workflow the File Sharing scenario.

5.4.1 First-Order Statistics

Current first-order statistics contain packet loss rate, downloading time and the ratio between control messages and data messages (C/D Ratio).

Figure 19a is a sample table of packet loss rate. In this sample test, a file of 100M size is transmitted and we can observe very high packet loss rate in general, whereas the rate differs from each other. It indicates that the congestion control algorithm in NeP2P can be further improved, and to achieve better performance, developers may need to take into account of the heterogeneity of real networking environments and think of a good way to estimate the networking environment of a peer.

Figure 19b shows both C/D Ratio and downloading time. Here, we count all messages other than data messages as control messages, which in fact indicates the internal communication between NeP2P service and clients when there is more than one receiver. As we can see from the sample result, the C/D Ratio ranges from 2% to around 6.5%, and we observed higher values when there are more receivers. This indicates that developers may need to make internal communication more efficient. The second row clearly lists out the transmission time for each receiver/peer with the minimal and maximal values highlighted.

p\s	s1	s2	s3	s4	s6	s7	s8	s9	all
s2	12.86%	-	61.07%	38.06%	35.31%	44.38%	50.65%	63.37%	40.00%
s3	0.20%	5.95%	-	17.20%	1.18%	26.54%	36.44%	38.23%	16.62%
s4	11.73%	37.58%	57.83%	-	36.22%	35.02%	50.30%	65.09%	37.45%
s6	22.21%	38.82%	43.96%	35.25%	-	44.29%	44.64%	58.75%	38.26%
s7	4.03%	5.02%	28.31%	14.78%	8.37%	-	27.45%	32.81%	16.37%
s8	12.80%	27.74%	25.33%	18.25%	27.57%	35.18%	-	65.04%	28.18%
s9	14.49%	2.66%	52.43%	7.43%	4.07%	51.25%	32.10%	-	18.81%

(a) Sample table for packet loss rate in percentage.

Type	s2	s3	s4	s6	s7	s8	s9	all
C/D Ratio	5.86%	3.15%	6.42%	6.17%	2.54%	4.62%	3.25%	4.44%
Trans Time	137.0706	84.0038	138.7851	136.4007	82.7125	100.7151	89.0879	109.8251

(b) Sample table for Control Message over Data Message ratio and Downloading time.

Figure 19: Sample results of statistical data.

5.4.2 Data Traffic

For a large P2P system, it is essential to understand data traffic patterns while our testing platform can visualize the traffic to offer greater insights. Figure 20 shows the visualized data traffic.

Figure 20a illustrates the outgoing traffic for the sender. Each color represents the traffic to a particular peer. It can be seen that sender's scheduler is sending out batches to all receivers at the very beginning, but send to a single peer after a certain duration. The total sending rate is well controlled.

If we can also select data traffic to peer s2 and s4 as shown in Figure 20b, it is very straightforward to understand what is happening. In line chart mode in Figure 20c, we select a portion of data traffic to display. The y-axis is the number of data packets, and x-axis the lapsed transmission time. It clearly indicates the scheduler is sending almost equal amount of data packets to all receivers initially.

For the receiver, Figure 20d shows how the peer's scheduler works in practice. As long as receiving initial batches from the sender, it immediately starts exchanging packets with other peers. The sending process is not continuous, maybe because the peer needs some time to do the data receiving and decoding. In practice, our testing platform is flexible enough to add more meaningful figures and charts to enhance data visualization.

5.4.3 Decoding Efficiency

For the decoding performance, we also log down relative information. Figure 21 is a sample figure showing the decoding scenario for a receiver. The red line indicates the accumulated size of decoded data. Typically, it is rather flat at the beginning and the decoding speed is relatively

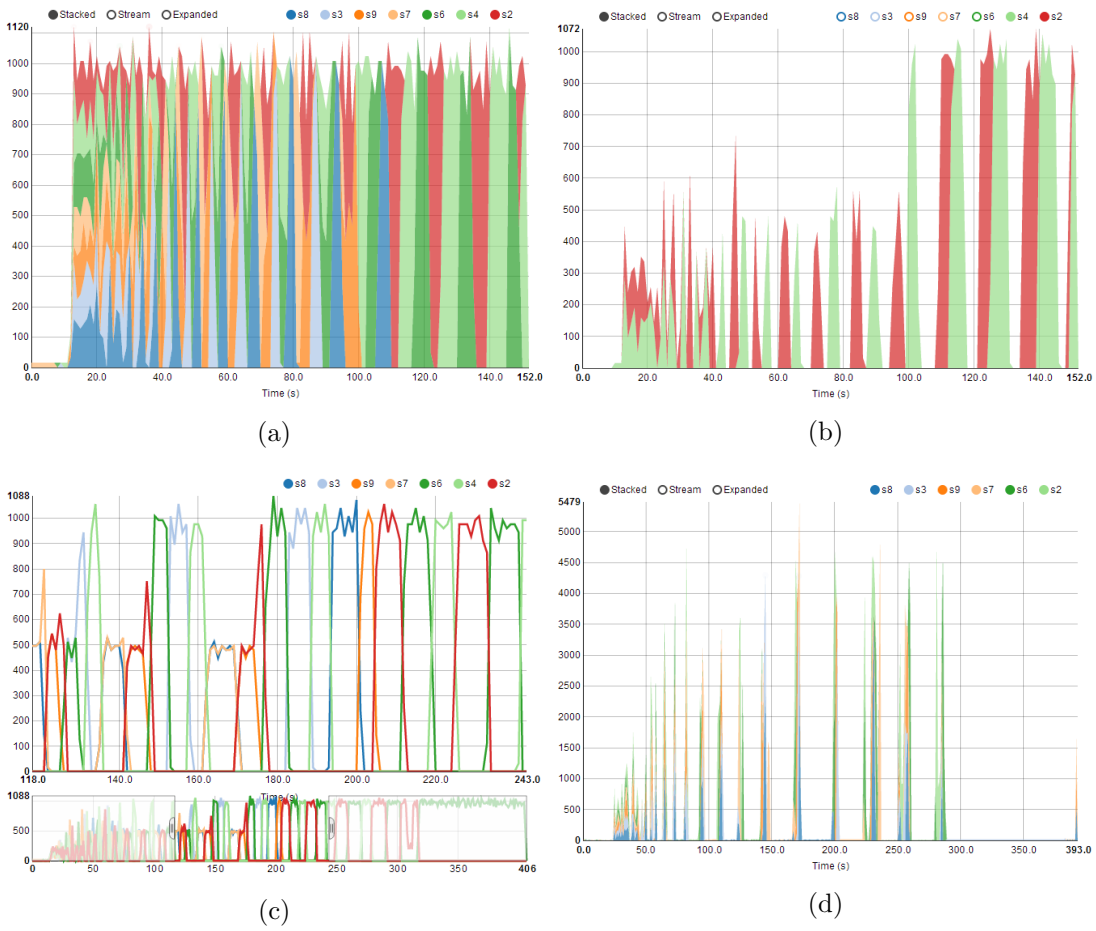


Figure 20: Sample results of data traffic. (a) Stack area chart showing sender's outgoing data traffic to all receivers. (b) Stack area chart showing sender's outgoing data traffic to peer s2 and s4. (c) Line chart showing sender's outgoing traffic speed with part of duration selected. (d) Stack area chart showing a peer's outgoing data traffic to other peers.

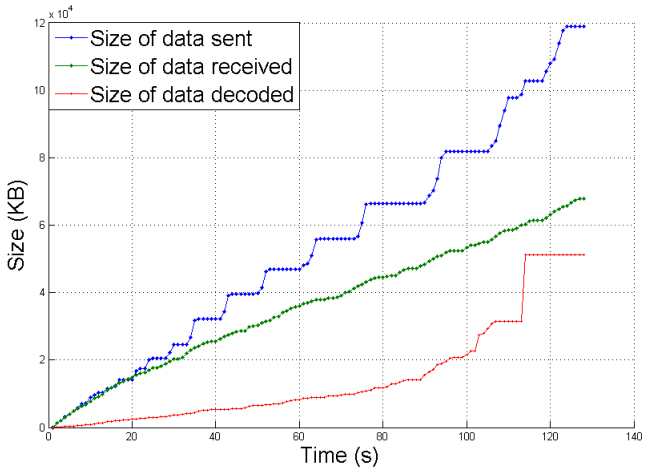


Figure 21: Sample figure illustrating the decoding performance.

low. The speed climbs up immediately between 80 to 100 seconds, but nothing more can be decoded between 105 to 115 seconds. Then, after receiving enough packets, it is able to decode the rest of the data all of a sudden and complete the transmission. The figure of decoding performance can definitely help developers better understand the decoding part of a receiver.

5.5 Discussion

So far, we are able to gather sufficient testing results via our testing platform for the case of file sharing, where there is one sender and multiple receivers with only one NeP2P client launched in every peer. As our testing platform is designed to be flexible, we can slightly adjust our platform settings to support more testing scenarios.

6 Conclusion

NeP2P protocol is a brand-new protocol powered by Peer-to-Peer techniques. To ensure its overall performance, it is necessary to design and implement an efficient Reliable UDP with NAT traversal capability. In addition, to conduct efficient testing for a P2P network, perform insightful data analysis and archive testing data, a good testing platform is highly desirable for NeP2P development.

In our Final Year Project, we developed an efficient N-to-M bidirectional Reliable UDP. Simulation results and real-world experiments have demonstrated its functionality and efficiency. To fit the design of NeP2P, we devise the Node List Updating service based on hole punching technique as a light-weight NAT traversal solution. Testing results have demonstrated its functionality in general cases. Furthermore, we built the NeP2P testing platform and performed sample testings to demonstrate its functionality and usability, and also pointed out evaluation dimensions that can be explored. By visualizing and analyzing collected data, the testing platform will definitely help developers understand how NeP2P protocol behaves. Through our Final Year Project, we have familiarized ourselves with Linux developing environments, and also gained excessive experience of networking programming as well as protocol design in the process of software development.

References

[1] *Adding an ordered dictionary to collections*. URL: http://www.python.org/dev/peps/pep-0372/.

[2] F. Audet and C. Jennings. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. RFC 4787 (Best Current Practice). Internet Engineering Task Force, Jan. 2007. URL: http://www.ietf.org/rfc/rfc4787.txt.

[3] S. Cheshire, M. Krochmal, and K. Sekar. "Nat port mapping protocol (nat-pmp)". In: *draft-cheshire-nat-pmp-03 (work in progress)* (2008).

[4] Gustavo De Veciana and Xiangying Yang. "Fairness, incentives and performance in peer-to-peer networks". In: *Seeds* 250.300 (2003), p. 350.

[5] *Fabric.org*. URL: http://docs.fabfile.org/en/1.6/ (visited on 05/19/2013).

[6] S. Guha, Y. Takeda, and P. Francis. "NUTSS: A SIP-based approach to UDP and TCP network connectivity". In: *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*. ACM. 2004, pp. 43–48.

[7] Saikat Guha and Paul Francis. *Simple traversal of UDP through NATs and TCP too (STUNT)*. URL: http://nutss.gforge.cis.cornell.edu/.

[8] Jtriley. *Github - pystun*. URL: https://github.com/jtriley/pystun.

[9] R. Mahy, P. Matthews, and J. Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766 (Proposed Standard). Internet Engineering Task Force, Apr. 2010. URL: http://www.ietf.org/rfc/rfc5766.txt.

[10] *MiniUPnP Project HomePage*. URL: http://miniupnp.free.fr/.

[11] Miserlou. *Github - stund*. URL: https://github.com/Miserlou/stund.

[12] *nvd3.js re-usable charts for d3.js*. URL: http://nvd3.org/ (visited on 05/19/2013).

[13] C. Partridge and R.M. Hinden. *Version 2 of the Reliable Data Protocol (RDP)*. RFC 1151 (Experimental). Internet Engineering Task Force, Apr. 1990. URL: http://www.ietf.org/rfc/rfc1151.txt.

[14] *PJNATH - Open Source ICE, STUN, and TURN Library*. URL: http://www.pjsip.org/pjnath/docs/html/.

[15] *Planetlab.org*. URL: http://www.planet-lab.org/ (visited on 05/10/2013).

[16] Johan A Pouwelse et al. *A measurement study of the bittorrent peer-to-peer file-sharing system*. Tech. rep. Technical Report PDS-2004-003, Delft University of Technology, The Netherlands, 2004.

[17] Dongyu Qiu and Rayadurgam Srikant. "Modeling and performance analysis of BitTorrent-like peer-to-peer networks". In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 367–378.

[18] *restund*. URL: http://www.creytiv.com/restund.html.

[19] Luigi Rizzo. "Effective erasure codes for reliable computer communication protocols". In: *SIGCOMM Comput. Commun. Rev.* 27.2 (Apr. 1997), pp. 24–36. ISSN: 0146-4833. DOI: 10.1145/263876.263881. URL: http://doi.acm.org/10.1145/263876.263881.

[20] J. Rosenberg et al. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489 (Proposed Standard). Obsoleted by RFC 5389. Internet Engineering Task Force, Mar. 2003. URL: http://www.ietf.org/rfc/rfc3489.txt.

[21] John Selbie. *STUNTMAN - Open source STUN server software*. URL: http://www.stunprotocol.org/.

[22] *TurnServer project - open-source TURN server implementation*. URL: http://turnserver.sourceforge.net/.

[23] *UPnP Forum*. URL: http://www.upnp.org/.

7 Appendix