Design

Description

LFTP using RDP Protocol, a UDP based protocol, helps **sending/getting large file between either server or client side**.
RDP implements the **reliability, flow control and congestion control** as TCP.

Usage

Before you use this program, on the server-side, you should make a folder named `data` under `code/` first to store the files to exchange with clients:

```
cd code
mkdir data
```

Then, you could run command below to run the program on the server:

```
# use python 3.x
python ./server.py [hostname port]
# default hostname: localhost
# default port: 8080
```

After that, you could run the command below to connect and exchange files with the server:

```
# use python 3.x
python ./client.py lget/lsend hostname[:port] filename
# default port: 8080
```

Hope you enjoy your time with it.

Transport Layer: RDP Protocol

> Author: Weiyuan Xu

Example of Using RDP

- Server

```
from rdp import *
server = RDP(addr=serverRunningAddress, port=serverRunningPORT)
# Set the max serving clients
# ATTENTION: Must run in a thread
server.listen(num)

# Retrieve a connected socket
# ATTENTION: Serve the client in a thread
while (connectSock = server.accept()) != None:
  data = connectSock.rdp_recv(bufferSize)
  # Analysis data command
  if getFileCommand:
    if connectSock.rdp_send(okCommand):
      connectSock.rdp_send(file)
  else if sendFileCommand:
    if connectSock.rdp_send(okCommand):
      connectSock.rdp_recv(bufferSize)
  else if quitCommand:
    addr = connectSock.release()
    # inform the listen server to release the port in addr
    server.releasePort(addr[1])
```

- Client

```
from rdp import *
client = RDP(Client=True)

client.makeConnection(addr=ServerAddress, port=serverPort)
if client.rdp_send(sendFileCommand):
  data = rdp_recv(bufferSize)
  if data == ok:
    client.rdp_send(file)
if client.rdp_send(GetFileCommand):
  data = client.rdp_recv(bufferSize)  # Get file
  # Once finish get the whole file, reset the recv state
  client.resetRecv()
  if data == ok:
    file = client.rdp_recv(bufferSize):  # Get file


client.rdp_send(data:quitCommand) # End connection with server

# Finally release socket
client.release()
```

RDP Design

RDP Protocol provides the **reliability, flow control, congestion control** like TCP based on UDP. Let's start introducing how RDP protocol is designed to retrieve the goal.

- **Design From Request**
  - **Fundament**
    According to the Application layer requests, *send data and receive data* function are fundamental and application needn't know the implementation. Thus, we first design two functions: `rdp_send(data)` to send data and `rdp_recv(size)` to get data. Furthermore, these two function should act like TCP, which means application just invokes functions and **knows** where it get/send data. So we need to make connection between server and client before invoking these functions with *handshake behavior*. `makeConnection(targetAddress)` is needed.
  - **Reliable**
    RDP is based on UDP, which is not reliable. We have to add something to make sure the reliability. Consider the rdt FSM shown in Chapter 3 in book(*Computer Networking, a top-down approach, six edition, James F.Kurose*), we get the idea. Set the data field in UDP packet as below:

UDP packet data field
Sequence Number
Acknowledgement Number
Flag Field (ACK, SYN, RST, FIN)
Data

    The usage of seqNum(Sequence Number) and ACKNum(Acknowledgement Number) are used to make sure pkt(packet) is truly sent to destination. Once we send a pkt labeled with seqNum, receiver must send a ACK pkt to sender with ACKNum equals the seqNum obtained in pkt received. The flag field is used to distinguish the kind of pkt(e.g. ACK means ACK pkt, SYN means handshake request pkt). Data is the real data we want to send.
  - **Multiple Client**
    Since server must support multiple client, the server application(host) must handle the clients at the same time. So **multiple thread** is needed. We provide each connected client a *server program* running in different *port*. So we design `listen(num)` function to *listen* the connection requests from clients and maximum number of client for server to serve is `num`. The listen function provides the *listening* and helps make connection

between server and client. Hence, sockets are created when connection successfully made in `listen`, we must export the serving socket for server application. **`accept()`** retrieve a serving socket and server application must run the socket in a thread and handle it.

- **Summary**

  `rdp_send(data)`, **`rdp_recv(size)`**, **`makeConnection(targetAddress)`**, **`listen(num)`** and **`accept()`** are the most important function designed in RDP. Following, we will introduce the implementation and the detail design of them.

- **Inside Function**
  - **`makeConnection(targetAddress)`** and **`listen(num)`**

    `makeConnection` helps make connection with client and server. We only invoke it in client side. First, it send SYN pkt and waits server to send ACK pkt. After receiving the ACK from server, client send ACK to client and the *handshake* or connection is established.

    Though, it helps us make connection, the socket keeps the targetAddress is a problem. Because the targetAddress is linked to the server listening application. We need to bind the client with the correct port of serving application(*Client just need to know the server running at the 8080 port and makeConnection with it, but server using different port to handle multiple clients. So it is important for client socket to know which port in server is serving it and* ***communicate*** *with serving application through that port*).

    ```
    function makeConnection(targetAddr) {
      pkt_send(SYN_pkt)
      while True {
        data = pkt_recv()
        if data.source_address == targetAddr && data.pkt_kind == ACK {
          save_server_port(data.port)
          pkt_send(ACK_pkt)
        }
      }
    }
    ```

    We deal with the problem with the idea of server sending the port to client in ACK pkt and waiting for the ACK to truly create a socket to server the client. Why creation happens when server receive from client? Because the *SYN flood* attack. If we create a socket while receiving a SYN pkt, server will soon run out of resource for serving the true client when somebody just send SYN to sever but not response ACK to server's ACK pkt. Pseudocode is below:

```
function listen(max) {
  while True {
    if serving_cnt < max {
      data = pkt_recv()
      if data.pkt_kind == SYN && !data.source_address in SYN_dict {
        port = retrieve_available_port()
        send(ACK_pkt and Port)
        SYN_dict.save(data.source_address)
      }
      else if data.pkt_kind == ACK && data.source_address in SYN_dict {
        socket = create_socket()
        active_sockets.save(socket)
        serving_cnt += 1
      }
    }
  }
}
```

- **`rdp_send(data)`**

  Because we must implement the flow control and congestion control. We need to add some variables to help. For controlling the sending rate, we use `sendWindowSize` to help. It acts like the size of the receive window which tells the most size of the send pkt. For flow control, we use `buffer` and `bufferSize` to help. In our design, once the rdp_recv(bufferSize) function is invoked, at most bufferSize data is returned. So the lastByteRead is always zero and the len(rcv_buffer) is the lastByteRecv. Then the rwnd = BufferSize - (lastByteRecv - lastByteRead) = BufferSize - len(rcv_buffer). The receiver include the receive window size inside the ACK pkt, so we add **`rwnd`** field into the UDP data field. Also, when the rwnd is zero, sender must wait. We design a special flag **`WRW`** (wait receive window) to label a pkt, which is used to loop asking the receiver the rwnd when rwnd is zero. This prevents sender not knowing the receiver have buffer to get data after last ACK pkt with rwnd equals zero. And sender does not send any data to receiver because it is told receiver have no space for data and receiver will never send a new rwnd to sender.

```
function rdp_send(data) {
  data_pkt = fragment(data)
  pipeline_send(data_pkt, sendWindowSize)
  lastACK = 0
  lastSend = sendWindowSize + 1
  while True {
    d = pkt_recv()
    if d.pkt_kind == WRW {
      do {
        send(WRW_pkt)
      } while pkt_recv().rwnd == 0
    } else if d.pkt_kind == ACK {
      if lastACK < d.ACK_number < lastSend {
        sendWindow[d.ACK_number - lastACK] = True
        sendWindowSize = d.rwnd
      } else if d. ACK_number == lastACK {
        for win in sendWindow {
          if win == True {
            lastACK += 1
          }
        }
        sendWindowSize = d.rwnd
        index = lastSend
        pipeline_send(data_pkt, sendWindowSize)
      }
    }
  }
}
```

For congestion control, we need to trace a variable **`cwnd`** in sender. Also, `ssthresh` and `dupACK` helps turning the state of the sender to control the sending rate in case of the congestion in network. We set slow start state, congestion avoidance state and fast recovery state as integer number to distinguish. We handle different cwnd behavior based on the state code.

And we adjust the sending window size with the formula: $size = min\{cwnd, rwnd\}$. The size helps determine the pkt numbers to send in case overflow the buffer in receiver and congestion in network.

```
function rdp_send(data) {
  data_pkt = fragment(data)
  pipeline_send(data_pkt, sendWindowSize)
  lastACK = 0
  lastSend = sendWindowSize + 1
  while True {
    d = pkt_recv()
    if d.sourceAddr == clientAddr {

      if recv_timeout || recv_dupACK || recv_normal{
        update_cwnd()
      }
      if d.pkt_kind == WRW {
        do {
          send(WRW_pkt)
        } while pkt_recv().rwnd == 0
      } else if d.pkt_kind == ACK {
        if lastACK < d.ACK_number < lastSend {
          sendWindow[d.ACK_number - lastACK] = True
          sendWindowSize = d.rwnd
        } else if d. ACK_number == lastACK {
          for win in sendWindow {
            if win == True {
              lastACK += 1
            }
          }
          sendWindowSize = d.rwnd
          index = lastSend
          pipeline_send(data_pkt, sendWindowSize)
        }
      }
    }
  }
}
```

- **rdp_recv(size)**

  It receive data from sender and return data in buffer to application. To match the flow control, we must determine if receive data from sender. So we must see the data size application wants which is the parameter `size` make space for further data. And once we send ACK pkt, the rwnd is calculated and sent, too. More interesting thing is that, we design a *buffer for buffer*. When rwnd is not zero, sender sends data. But the data sender sends makes buffer overflow. In this case, we still buffer this data, because it is a waste to abandon the pkt if we just set a *buffer for buffer*. We hide the real rwnd because it is less than zero in this case, and just send zero as rwnd to sender. And after, application invoke `rdp_recv(size)` , we will determine if the size of data make space to receive more data from sender. If yes, we get data from remote. Otherwise, we just

return application the data in buffer and not get data from remote.

```
function rdp_recv(size) {
  if size > buffer_size {
    throw value_error
  }
  data = retrieve_from_buffer(size)
  rwnd = calc_rwnd(buffer_size, read_pos, recv_pos)
  if rwnd < 0 {
    return data
  }

  timeout_cnt = 0
  while True {
    data = pkt_recv()
    if timeout {
      if timeout_cnt < 5 {
        timeout_cnt += 1
      } else {
        break
        // End of the whole data transfer
      }
    }
    if data.sourceAddr == clientAddr {
      if data.pkt_kind == WRW {
        rwnd = calc_rwnd(buffer_size, read_pos, recv_pos)
        pkt_send(rwnd)
      } else {
        back_index = rcv_base - rcv_window_size < 0 ? 0 : rcv_base - rcv_w
        if back_index <= data.seq_num < rcv_base {
          pkt_send(ACK)
        } else if rcv_base < data.seq_num < rcv_base + rcv_window_size {
          buffer(data.data)
          pkt_send(ACK)
        } else if data.seq_num == rcv_base {
          pkt_send(ACK)
          update_rcv_base()
          buffer(data.data)
        }
      }
    }
  }
  return data = retrieve_from_buffer(size)
}
```

- **Packet Structure**

  This is the final pkt structure after taking the consideration above.

UDP packet data field

Sequence Number
Acknowledgement Number
Flag Field (ACK, SYN, RST, FIN, WRW)
rwnd
Data

- **Summary**
  Some other designs are not introduced because above functions are the most important to retrieve the goal, like release the socket. More details could be found in code with detail comments and in book, ***Computer Networking, a top-down approach, six edition, James F.Kurose***. Most of the idea are inspired from the book for it gives a detail picture.
  Moreover, thanks to my group member, **Yongqi Xiong(SiskonEmilia)**, who listens to my complaint and forgives my idiot fault.

Application Layer: LFTP

> Author: Yongqi Xiong

Server side

Target

1. Support multiple client by multiple thread function
2. Implement the data sending(client getting file) by calling the function `RDP.rdp_send(data)`
3. Implement the data downloading(client uploading file) by calling the function `RDP.rdp_get(size)`

Multiple User

In order to implement multiple user support, we introduce multi-thread architecture to this project. With multi-thread technique, we handle each user's request with a single thread, which could run concurrently with other threads. Thus we can handle plushy users' request 'at the same time':

```
listenThread = threading.Thread(target=listen, args=(hostname, port), name="Lister
listenThread.start()
console()
```

Further more, multi-thread architecture also makes it possible for us to response to the requests of users to make connection and handle the requests of users who has made connection concurrently:

```
server = RDP.RDP(addr=hostname, port=int(port))
listenThread = threading.Thread(target=server.listen, args=(10,), name="basic list
listenThread.start()
while True:
  time.sleep(0.1)
  # If the user is trying to exit
  if exit:
    # Stop the listen thread
    RDP.exit = True
    if threading.active_count() == 2:
      break
    else:
      time.sleep(0.5)
      continue

  serverLock.acquire()
  clientSocket = server.accept()
```

What worth being paying attention to is that, every dead loop in a thread may affects the speed of other threads to execute code. A good practice is to introduce `time.sleep` into each thread to balance the computing resources given to each thread.

In the very following of the execution of the server-side program, we will deploy a listener at the given hostname and port number, then internally accept the requests of users to make connection, and we will allocate a new port for the user to access the server (and exchange data with it on). After that, we will construct a socket to handle this connection, then create a new thread to handle these commands sent from users.

A critical issue lead by multiple users support is the **Writer-Reader Problem**, which means concurrent writing and reading on the same file may results in an unexpected error. To get rid of this trouble, we introduce `Thread Lock` to our program. Any writer needs to acquire a writer lock (there's only one such lock for each file) before they begin to write while making sure no reader is reading the file, and every reader needs to make sure no writer is writing before they begin to read. In practice, taking writing method as an example, we implement it as below:

```
print("Receiving %s: Acquiring Dictionary Lock..." % filename)
dictLock.acquire()
print("Receiving %s: Dictionary Lock Acquired" % filename)
# Try to get the writer lock
if filename not in wLockDict:
  print("Receiving %s: Initializing File Lock..." % filename)
  # If no lock is initialized, create one
  wLockDict[filename] = threading.Lock()
  # Get the writer lock
  print("Receiving %s: Acquiring file writing lock..." % filename)
  wLockDict[filename].acquire()
  print("Receiving %s: File writing lock Acquired." % filename)
  rLockDict[filename] = threading.Lock()
  rCountDict[filename] = 0
  dictLock.release()
  print("Receiving %s: Dictionary Lock Released." % filename)
else:
  dictLock.release()
  print("Receiving %s: Dictionary Lock Released." % filename)
  # Get the writer lock
  print("Receiving %s: Acquiring file writing lock..." % filename)
  wLockDict[filename].acquire()
  print("Receiving %s: File writing lock Acquired." % filename)
  while rCountDict[filename] != 0:
    time.sleep(0.5)

####################
# Write file here #
####################

# End of writing
print("Receiving %s: File Lock Released." % filename)
wLockDict[filename].release()
```

Sending Data

When the user sends a command like `lget filename`, we will try to find a file of filename and send it to the user:

First, we will check the existence of target file with `os.path.exists()`, if no such file exists, an error message will be sent to the client.

```
if not os.path.exists("data/" + filename):
  socket.rdp_send("NO")
  print("Sending file %s: No such file." % filename)
  return
```

Then, we'll get the length of the file with `os.stat().st_size`, pack it with an OK message, and send them to the client.

```
if not socket.rdp_send("OK\n" + str(length)):
  print("Sending file %s: No such file." % filename)
  return
```

After we get the confirm of client of receiving the message above, we'll start to read some data from the file and send them to the client internally before the length of data sent successfully equals to the total length of the whole file.

```
with open("data/" + filename, "rb") as f:
  start_time = time.time()
  while sentLength != length:
    line = f.read(20480)
    if not socket.rdp_send(base64.b64encode(line).decode("ASCII")):
      print("Error while sending file %s." % filename)
      return
    sentLength += len(line)
    print("Sending file %s: %d%% done." % (filename, sentLength / length * 100))
    print("Speed: %d KB/s" % (sentLength / (time.time() - start_time + 0.01) / 100
  print("Sending done.")
```

To better pack our RDP packet, we need to convert every byte in the binary file to base64 format, and use ASCII to format it into a literal string.

Getting Data

Getting data from clients is like an inverse progress of sending data to them. Thus after we get command like `lsend filename length` from clients, we will send `OK` message to confirm that we're ready to receive the file:

```
if not socket.rdp_send("OK"):
  print("Receiving %s: Connection Error: Fail when asking client to send file." %
  wLockDict[filename].release()
  return
```

After we receive the ACK packet from clients, we begin to receive those data that clients sends till the length of data we accepted equals to the length that the client claims the file to have:

```
with open("data/" + filename, "wb") as f:
  start_time = time.time()
  while True:
    # User want to exit
    if exit:
      print("Receiving %s: Server is exiting..." % filename)
      break
    # Finish
    if acLength == length:
      print("Receiving %s: Done" % filename)
      break
    # Receive some data
    metadata = socket.rdp_recv(30720)
    while len(metadata) % 4 != 0:
      temp = socket.rdp_recv(30720)
      if len(temp) == 0 :
        metadata = ""
        break
      metadata += temp

    data = base64.b64decode(metadata.encode("ASCII"))
    if len(data) == 0:
      print("Receiving %s: Connection Error: Timeout when receiving data." % filer
      break
    acLength += len(data)
    print("Receiving %s: %d%% data received..." % (filename, acLength / length * 1
    # Write to file
    f.write(data)
    print("Speed: %d KB/s" % (acLength / (time.time() - start_time + 0.01) / 1000)
```

There's an issue you need to pay sight on: base64 code must be a multiplex of 4 (actually, it encode every 3 bytes into 4 ASCII code), which means the existence of invalid length of data. Thus we need to judge if the data buffered is valid, and only decode the valid ones.

Client side

1. Implement the data sending(uploading file) by calling the function `RDP.rdp_send(data)`
2. Implement the data receiving(downloading file) by calling the function `RDP.rdp_get(size)`

Make Connection

Client side program needs to corporate with the server side one to make the whole system work as expected. So, before we do any operation, we need to make connection with the server:

```
client = RDP.RDP(client=True)

if not client.makeConnection(addr=hostname, port=port):
  print("Error while connecting server.")
  return
```

As in each session of the program, only one operation will be do individually, multi-thread architecture is no longer necessary for client side.

Sending Data

Before sending the real content of file, we need to check the existence of file and inform server of the information of the file. And our sending can only begin after the server gets ready (and sends us a OK message).

```
# Check if file exists
if not os.path.exists(sys.argv[3]):
  print("Error: No such file.")
  return

filename = os.path.basename(sys.argv[3])

# Get the length of file and the file itself
length = os.stat(sys.argv[3]).st_size

if not client.rdp_send("lsend\n" + filename + "\n" + str(length)):
  print("Error while sending command.")
  return

response = client.rdp_recv(1024)
if response != "OK":
  print("Error while waiting for response! " + response)
  return
```

After we get server's permission to transport, we will do like server does before, read data internally and send them to the server:

```
start_time = time.time()
while sentLength != length:
  line = f.read(20480)
  if not client.rdp_send(base64.b64encode(line).decode("ASCII")):
    print("Error while sending file %s." % filename)
    return
  sentLength += len(line)
  print("Sending file %s: %d%% done." % (filename, sentLength / length * 100))
  print("Speed: %d KB/second" % (sentLength / (time.time() - start_time + 0.01) /
print("Sending done.")
```

Getting Data

Like what sending data does, when we want to get data from server, we need to inform it with the information of the file we want to get and wait for its response:

```
if not client.rdp_send("lget\n" + filename):
  print("Error while sending command.")
  return

response = client.rdp_recv(1024)
if response == "NO":
  print("No Such File")
  return
elif response == "":
  print("Error while waiting for response!")
  return
```

As server has sent us some information about the file to be sent, we need to use them to config the process followed:

```
response = response.split("\n")
print("Response from server: ", "".join(response))
if response[0] != "OK" or len(response) != 2:
  print("Error while analysising response!")
  return
```

At last, just get data like what server does above:

```
while True:
  if acLength == length:
    print("Receiving %s: Done" % filename)
    break
  # Receive some data
  metadata = client.rdp_recv(40960)
  while len(metadata) % 4 != 0:
    temp = client.rdp_recv(40960)
    if len(temp) == 0 :
      metadata = ""
      break
    metadata += temp

  data = base64.b64decode(metadata.encode("ASCII"))
  if len(data) == 0:
    print("Receiving %s: Connection Error: Timeout when receiving data." % filenam
    break
  acLength += len(data)
  print("Receiving %s: %d%% data received..." % (filename, acLength / length * 100
  # Write to file
  f.write(data)
  print("Speed: %d KB/s" % (acLength / (time.time() - start_time + 0.01) / 1000))
```