LFTP Document

reliableUDP

reliableUDP 模块具体如下：

1. `rUDPConnection`

   `rUDPConnection` 类包含于connection.py 文件中，作用为创建 socket 供 rUDP 连接使用。

   ```
   def __init__(self, ip=None, port=None)
   ```

   使用需绑定的 ip 地址与端口号创建对象。

2. utilities.py

   该文件包含了rUDP 的服务端与客户端都可能使用的变量，函数与类，具体如下：

   - 服务端状态集

     ```
     RecvStates = Enum('RecvStates', ('CLOSED', 'LISTEN', 'SYN_REVD',
                                      'ESTABLISHED', 'CLOSE_WAIT',
     'LAST_ACK'))
     ```

   - 客户端状态集

     ```
     SendStates = Enum('SendStates', ('CLOSED', 'SYN_SENT',
     'ESTABLISHED',
                                      'FIN_WAIT_1', 'FIN_WAIT_2',
     'TIME_WAIT'))
     ```

   - 拥塞控制状态集

     ```
     CwndState = Enum('CwndState', ('SLOWSTART', 'CONGAVOID',
     'SHAKING'))
     ```

   - 表头字段集合

     ```
     Sec = Enum('headerStruct', ('sPort', 'dPort', 'seqNum', 'ackNum',
                                           'offset', 'NS', 'CWR',
     'ECE', 'URG',
                                           'ACK', 'PSH', 'RST', 'SYN',
     'FIN',
                                           'recvWin', 'checksum',
     'urgPtr'))
     ```

     对于表头数据进行转换时需要使用包含以上所有字段的 dict 进行。

- 将表头数据转换为 python 中的 dict 表：

```
def header_to_dict(headerData: bytearray)
```

- 将一个 dict 表转换为表头二进制数据(checksum 段为0)：

```
def dict_to_header(headerDict: dict)
```

- 传入表头与数据，计算 checksum 并填入表头数据：

```
def fill_checksum(header: bytearray, data: bytearray)
```

- 传入接收到的数据，检验 checksum 正确性：

```
def check_header_checksum(data: bytearray)
```

- 接收缓冲区，使用环形队列实现：

```
class rcvBuffer
```

  各方法如下：

  - 将数据加入缓冲区，若已满则返回 False：

```
def add(self, data: bytearray)
```

  - 获取缓冲区剩余空间大小：

```
def get_win(self)
```

  - 获取缓冲区中的第一个数据：

```
def peek(self)
```

  - 获取缓冲区中第一个数据并将其移出：

```
def pop(self)
```

- 发送缓冲区，使用环形队列实现：

```
class sndBuffer
```

  各方法如下：

  - 发现超时，启动拥塞控制：

```
def find_cong(self)
```

- 获取发送窗口中的数据：

```
def get_data(self)
```

- 设置发送窗口(最大20)：

```
def set_win(self, win)
```

- 添加新数据至缓冲区：

```
def add(self, data: bytearray)
```

- 收到 ack 消息时对缓冲区中数据进行调整：

```
def ack(self, mess)
```

- 忽略流量或拥塞控制直接发送消息后将消息推入缓冲区进行 ack 管理：

```
def send(self, mess)
```

- 消息类

```
class message
```

包含的方法如下：

- 创建消息，传入消息数据， connection 对象，以及发送缓冲区的引用，消息将调用 connection 进行发送

```
def __init__(self, data, conn: rUDPConnection, sendBuf: sndBuffer=None)
```

- 返回是否已收到大于消息 seq 的 ack 信息

```
def is_acked(self)
```

- 发送消息至指定地址并启动计时器，到时时若仍未收到对应的 ack 则进行重传：

```
def send_with_timer(self, destAddr)
```

- 发送消息，不启动计时器（用于发送 ack 等信息）

```
def send(self, destAddr)
```

- 消息管理池：

  ```
  class msgPool
  ```

  方法如下：

  - 添加消息，需要同时传入预期的最小 ack 序号：

    ```
    def add_msg(self, msg: message, expectACK)
    ```

  - 使用预期 ack 序号获取对应消息对象：

    ```
    def get_mess(self, ackNum)
    ```

  - ack 对应的消息：

    ```
    def ack_msg(self, ackNum)
    ```

  - 收到ack 后对所有小于其 ack 序号的消息进行 ack 操作（关闭重传计时器）：

    ```
    def ack_to_num(self, ackNum)
    ```

3. client.py

   该文件是 rUDP 的客户端代码，包含一个客户端类：

   ```
   class rUDPClient
   ```

   包含的方法如下：

   - 应用层调用该方法从接受缓冲区中获取一个数据包：

     ```
     def consume_rcv_buffer(self)
     ```

   - 应用层调用该方法向发送缓冲区添加一个数据包等待发送：

     ```
     def append_snd_buffer(self, data: bytearray)
     ```

   - 内部调用方法，检测拥塞状态并根据具体状态发送数据：

     ```
     def check_cong_and_send(self)
     ```

   - 更新状态并记录日志：

     ```
     def update_state(self, newState)
     ```

   - 发送第一次握手消息尝试与服务器建立连接：

```
def establish_conn(self)
```

- 发送第三次握手信息：

```
def third_handshake(self)
```

- 进行完整的三次握手操作：

```
def handshake(self)
```

- 检测收到的消息是否为正确的第二次握手信息：

```
def check_establish_header(self, headerDict: dict)
```

- 应用层调用该方法进行连接握手操作：

```
def connect(self, destIP, destPort)
```

- 不使用发送缓冲区直接发送消息：

```
def send_msg(self, data)
```

- 应用层调用该方法表示结束连接，发送挥手消息：

```
def finish_conn(self)
```

- 发送第一次挥手信息：

```
def first_wavehand(self)
```

- 检测接收的消息是否为正确的第二次挥手消息：

```
def check_second_wave(self, headerDict: dict)
```

- 检测收到的消息是否为第三次挥手信息：

```
def check_third_wave(self, headerDict: dict)
```

- 接收到第二次挥手消息后更新连接状态：

```
def second_wavehand(self)
```

- 接收到第三次挥手消息后更新状态并发送第四次挥手：

```
def third_wavehand(self)
```

- 关闭 socket：

```
def close(self)
```

- 发送第四次挥手消息：

```
def fourth_wavehand(self)
```

- 对接收到的信息进行处理：

```
def process_msg(self, data)
```

- 对接收到的信息进行 ack 消息的发送：

```
def ack_msg(self)
```

- 连接建立后建立该函数的子线程进行套接字接收消息的处理：

```
def listen_msg(self)
```

4. server.py

   该文件包含 rUDP 的服务端的实现，共有两个类：

   1. 服务端与单个用户的连接类：

      ```
      class serverConn
      ```

      方法如下：

      - 应用获取接收缓冲区数据：

        ```
        def consume_rcv_buffer(self)
        ```

      - 应用发送数据至发送窗口：

        ```
        def append_snd_buffer(self, data: bytearray)
        ```

      - 检测拥塞状态并根据实际情况发送数据:

        ```
        def check_cong_and_send(self):
        ```

      - 更新状态并记录日志：

        ```
        def update_state(self, newState)
        ```

      - 接收到创建连接消息后发送第二次握手消息：

        ```
        def handshake(self)
        ```

- 发送第三次挥手消息：

```
def response_FIN(self)
```

- 根据接收的数据和表头处理消息：

```
def process_data(self, data, headerDict: dict)
```

- 不使用缓冲区直接发送数据：

```
def send_msg(self, data)
```

- 对接收到的数据消息发送 ack 消息：

```
def ack_message(self)
```

- 将自身从下述的服务器连接集合中移除：

```
def removeSelf(self)
```

2. rUDP 服务器类，用于管理所有服务器的连接：

```
class rUDPServer
```

方法如下：

- 构建服务器时将服务器所监听的地址，端口号及应用对象传入：

```
def __init__(self, ip, port, app)
```

- 服务器创建后将对下面的函数建立子线程，监听端口接收的消息：

```
def recv_msg(self)
```

- 根据接收到的信息将数据传至对应的用户连接进行处理，若用户不存在且该连接为握手消息，则创建新连接进行处理：

```
def process_recv_msg(self, data, addr)
```

- 移除指定的 用户连接：

```
def removeConn(self, addr)
```

5. application/app.py

application子模块中包含一个 app 类，应用层通过实现该类中的4个方法来使用 rUDP。

```
class app
```

4个方法如下：

1. 处理信息函数，函数接收的参数为接收消息的用户地址，应用层在函数中调用 rUDP 连接的`consume_rcv_buffer`方法获取接收的消息。

```
@abstractmethod
def process_data(self, user)
```

2. 下一步函数，该函数一般在收到先前消息的 ack 消息后触发，作用为通知应用层进行下一步操作（如发送文件大小信息，文件下一个数据段等）。

```
@abstractmethod
def next(self, user)
```

3. 移除用户函数，该函数通知应用层一个用户的连接已释放，可以将该连接的用户删除，仅服务端应用实现该函数：

```
@abstractmethod
def remove_user(self, user)
```

4. 连接关闭函数，该函数通知应用层连接已关闭，仅客户端应用实现此函数：

```
@abstractmethod
def notify_close(self)
```

6. lftplog.py

   该文件定义了一个Logger 类型的 logger 变量，rUDP 使用该变量进行日志的记录

应用层

应用层的代码实现位于代码的根目录，包含服务端和应用端的两个代码文件，具体介绍如下:

1. client.py

   文件中包含了一个 client 类，包含了服务端应用的主要逻辑，客户端应用执行过程中共有三个状态：

```
clientStates = Enum('clientStates', ('CLOSED', 'SENDREQUEST', 'DATA'))
```

   - CLOSED 状态为客户端未建立时的状态
   - 客户端与服务端连接建立后应用进入SENDREQUEST 状态，发送请求至服务端
   - 服务端响应请求后若请求有效则进入 DATA 状态，在该状态下进行文件数据的传输，若请求无效则客户端程序释放连接并退出。

   client 类中在实现 app 基类中的4个抽象方法后实现了如下的`send_data`方法：

```
def send_data(self, data, useBuffer)
```

该方法的作用是对输入的数据进行发送前的填充并进行发送，在数据前的4个字节中填充入以big endian 编码的数据长度，若该长度与数据合并后长度仍未达到 rUDP 的数据包默认大小（5120 字节）则在末尾补0至要求的包大小。useBuffer 变量的作用是制定本次发送是否使用发送缓冲区。

2. server.py

   该文件进行了服务端应用的实现，主要包含两个类：

   1. `serverSession` 是每个客户端对应的服务端应用，在用户创建连接并发送请求后根据请求构建。

      包含的方法如下：

      - 更新状态并记录日志：

        ```
        def update_state(self, newState)
        ```

      - 进行下一步操作（发送文件下一段等）：

        ```
        def next(self)
        ```

      - session 启动后对用户的请求进行响应：

        ```
        def response_req(self)
        ```

      - 处理用户发送的数据（文件大小，文件数据段等）：

        ```
        def process_data(self, data)
        ```

   2. `server(app)` 对 session 进行管理，实现了 app 基类的4个抽象方法，使用一个 dict 实现用户地址到 session 的映射。

客户端与服务端应用层交互命令

1. client->server: `lLIST`

   客户端向服务端发送列出文件请求，服务端将向服务端返回 JSON 格式的数据以字符串数组的形式来表示服务器中存在的文件的列表

2. client->server: `lGET filename`

   客户端向服务器请求文件，`lGET` 与文件名间以空格字符作为分隔符

3. client->server: `lSEND filename`

   客户端向服务器请求上传一个文件

4. server->client: `NOTEXIST filename`

   服务端对请求下载文件命令进行命令无效响应，原因是文件不存在

5. server->client: `EXISTED filename`

   服务端对请求上传文件的命令进行无效响应，原因是同名文件已存在

6. server->client: `WAITING filename`

   表示对服务器请求有效，服务器等待客户端传输数据

7. client<->server: `SIZE filesize`

   文件发送方发送前发送文件大小，文件大小以 little endian 编码，当服务器收到下载请求时返回该命令表示请求有效。

8. server->client: `DONE`

   服务器完成文件的接收或发送后发送该指令表示当前可以关闭连接。