Individual Network Programming Report

packet sniffer tools

Kiarash Vosough
Dr. Yaghmaee
April 2, 2021

1. Abstract

For this project, I chose to create a Desktop application for use on linux based operating systems.

The purpose of the application was to provide users with the ability to sniff on ports, packet, ping, etc. By adapting this functionality I could also provide users with a user-interface to engage with the application.

2. Introduction

As described in paper information provided by teacher assistant, packet sniffer project is an individual programming project which requires significant effort on the part of the student. Students are expected to formulate and specify a small programming project, develop an appropriate piece of software to carry out the stated purpose, conduct some experiments with the software and provide a report on the outcomes of their investigations.

This report aims to provide a detailed look at the resulting application and some analysis of the research conducted which influenced the design decisions made.

2.1 Background

At the beginning of the project, I had no previous experience with network programming, so a large portion of time was dedicated to investigating, understanding and testing smaller bits of functionality, as well as looking at alternative implementations to figure out what I thought worked best.

Ultimately, I would end up having to teach myself network programming, mostly relying on my ability to apply the knowledge I accumulated over the last two years at Ferdowsi University to different scenarios. There is also a small amount of code that was adopted from other sources - mostly from answers provided in StackOverflow, as well as various other tutorial sites/videos.

2.2 Project Brief

The main goal of this project was to create an application that would allow users to sniff on network with some information which would help them monitor their network statistics and floating data on it.

3. Application Details

3.1 Overview

The application provides users with three main functions named as sniffing mode (as seen in the figure below):

start of application preview

There are some arguments which you can pass in order to achieve your requested result. All of them are listed below:

- *Address* : the IP (v4 or v6) that you wish to sniff on. you can pass ip address or host name plus its domain.
- *Thread number* : number of threads (there is also a limit that we prompt before you enter your inputs) that the program can use to complete its tasks.
- *waiting time* : time (in second) that program wait to get response back from the server for each port.
- *sniffing modes* : port range for specific set of port that can be sniffed.
- *start* : start value of port interval that program will check(optional).
- *end* : end value of port interval that program will check(optional).
- *ip version* : ipV6 and ipV4 are supported(default is set on ipV4).

3.2 Socket Objects

Before going into details in regards to the functionality of the application it is import to get an understanding of Socket and how they are managed, as these are the primary objects on which this application is based.

Socket is an API required to create Connection between servers and clients, as well methods need to monitor the network. A detailed look at the composition of the class reveals more about what is contained in these objects and why that information is relevant.

3.2.1 Member Data Fields

These are most in use variable that we will pass it to socket class and we achieve our goals with focusing on them and changing them.

```
def __init__(self, family=-1, type=-1, proto=-1, fileno=None)
```

family : ip version which socket will use (ipV4 or ipV6 or Unix domain socket).
type : represent the socket types (SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_RDM, SOCK_SEQPACKET).
proto : The protocol number is usually zero and may be omitted or in the case where the address family is AF_CAN the protocol should be one of CAN_RAW, CAN_BCM, CAN_ISOTP or CAN_J1939.

def connect_ex*()* is used to establish a connection between client and server.It returns an error indicator (0 if the operation succeeded otherwise errno).

A pair *(host, port)* is used for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like ‘www.google.com’ or an IPv4 address like '100.50.200.5', and port is an integer.

For AF_INET6 address family, a four-tuple (host, port, flowinfo, scope_id) is used, where flowinfo and scope_id represent the sin6_flowinfo and sin6_scope_id members in struct sockaddr_in6 in C. We just pass last two parameter zero.

4. Managing Data

There are three class which help in order to model the data that application require to complete its tasks.

4.1 Port Scanning Model

This model is used on modeling the User input information. The init signature is :

def __init__(self, address, thread_number, portRange, waitingTime, sniffing_mode, ip_version=socket.AF_INET)

4.1.1 Member Data Fields

- *Address* : the IP (v4 or v6) that the user wishes to sniff on.
- *Thread number* : number of threads that the program can use to complete its tasks.
- *waiting time* : time (in second) that program wait to get response back from the server for each port.
- *sniffing modes* : port range for specific set of port that can be sniffed.
- *port Range : is a custom class named customRange which hold two value* ***start*** *and* ***end.***
- *ip version* : ipV6 and ipV4 are supported (default is set on ipV4=socket.AF_INET).

4.1.2 Methods

This class also implement 3 more method which are:

A. def get_connect_param(self, port) : It takes a port and return a itaratable tuple to create socket connection based on ip version.

B. def is_range_defined(self) : returns a boolean value which show whether user defined a range for port or not.

C. def __str__(self) : return string representation for ip version.

D. def get_all_ports(self) : returns a list of ports based on user constraint on start and end port interval (should be between 0-65535) otherwise it returns all ports in range 0-65535.

E. def get_reserved_ports(self) : filter the reserved ports list based on user selected port interval and return it plus application layer reserved ports otherwise it returns all reserved ports plus application layer reserved ports.

F. def get_application_port(self) : filter application layer reserved ports based on user selected port interval otherwise it returns all the application layer reserved ports.

G. def get_ports_by_mode(self) : decide to return one of the lists by calling one of three method explained above based on **sniffing_mode.** It also sort the list.

4.2 Port Model

In order to get in detail for every port, this Model help to achieve that. Some information about port and also port is hold by ***PortModel.*** The init signature is:

def __init__(self, description, port, status, tcp, udp)

4.2.1 Member Data Fields

- *description* : A string represent a brief information about that port .

- *port* : An integer or string which hold a port (it will be casted to integer if type is string.).
- *status* : It shows that if port is Registered in Reputable institution or not.
- *tcp* : A boolean value which show this port support ***tcp*** protocol or not.
- *udp : A boolean value which show this port support **udp** protocol or not.*

4.3 Reserved Port Services

The main purpose of this class is to map data but it is also a Enum. It holds application layer service popular ports and comes with one class method and one static methods :

4.3.1 Methods

A. @staticmethod def get_all_reserved_port() : return am array of port model from decoded JSON file.

B. @classmethod def get_port_model(cls) : returns an array of port which mapped from self (which is an Enum).

5. Functionality

The basic functionality of app on start was discussed and as it is not complicated, this section will show some functionality and details behind the scene.

5.1 App StartUp

When the user execute main.py (as it is the main file which should execute) an instance of ***AppStartManager*** class (inherited from StartManager class) will be created and that's where everything begin. It has various method which get things ready and also implements all of it super class.

5.1.1 Methods

A. def start(self) : this method start on first instance of ***AppStartManager,*** its main purpose is to find the maximum thread that user's computer can create and then prompt and ask for user inputs (both tasks are separated into two method). It also check for app mode by calling def detect_app_mode(self)

B. def detect_app_mode(self) : it checks for at least 2 argument from argv if not the app will exit. if the condition is satisfied, the method def validate_app_mode(self) from Validation manager will be invoked and it detect if the user request for port sniffer or ping service. On each result the StartManager related to that service will be instantiated and start.

// these methods moved to PortScannerStartManager

C. def run_on_terminal(self) : Used to parse argument passed in-front of the main.py when user try to run program in terminal.

D. def run_in_(self) : This method asks user for inputs and after that validate the inputs with **ValidationManager** class. if the inputs are in valid an error throws and prints for user (user can still enter his/her inputs from the first step). If the inputs were correct an instance of **PortScanningModel** will be created. This instance will pass to **portSniffer** instance to start sniffing.

E. def start_port_sniffing(self) : After the user input being processed, this method will start sniffing procedure by create a **portSniffer** and pass a **PortScanningModel** to it.

5.2 Threading Utilities

A singleton class which contains some useful functions for better threading handling and do specific tasks.

5.2.1 Data Field

- *should_end_main_thread* : A boolean indicates that the main thread should be finished or being kept alive by def keep_main_thread_alive(self) method.
- end_printing_thread : A boolean indicates that the printing thread should be finished or being kept alive.
- *lock* : A mutex lock used to acquire if *should_end_main_thread wants to be changed.*

5.2.2 Methods

A. def start_new_procces_to_get_max_threads(self) : This is the first method that start call. In order to realize the thread creation limit, A new process will be created and pass get_max_thread_on_machine as its target and child_conn as a parameter (child_conn is a pipe and it is used to passing data between two process). The process start and the program wait until it sends our requested data back through child_conn, then the process will be terminated and run_in_interactive method get called.

B. def get_max_thread_on_machine(pipe) : it take a pipe for parameter and will send data back through that pipe to parent process. The method used here to get maximum thread is that the method create a for-loop with a counter inside it that repeat 1 million times, each time there is a thread (which sleep for 1000 second as it starts) created in for-loop and starts. This will continue until the operating system stop the process to create thread and that's were an error throws and break the loop. This way the maximum number of thread that could be created is obtained. This limit is shown before the user is asked for inputs. And an error is thrown if user hit the limit.

C. def wait_for_threads(threads, should_sleep=0) : A static method which take threads list and keep the current thread alive until they are terminated or finished.

D. def keep_main_thread_alive(self) : First acquire lock and change *should_end_main_thread* to true.

5.3 Validation Manager

This class has three static method which help to validate user inputs.

A. @staticmethod def validate_ip_version(ip) : Check whether the ip string which passed, is ipV6 or ipV4, otherwise an error is thrown.

B. @staticmethod def validate_app_mode(mode) : Check whether the mode from user input and return AppMode enum shows the requested service. if any thing goes wrong exit(0) will be invoked.

C. @staticmethod def get_ip_from_address(address) : Check whether the string passed to it is a valid ip or if it is host address , the ip of the host address will be returned, if not, it raise an exception.

D. @staticmethod def validate_ip_address(mode) : Check the address passed to it by def *inet_aton(string) and return tru if success*.

E. @staticmethod def validate_ip_version(ip) : return socket.AF_INET or socket.AF_INET6 based on string passed to it (string should be '4' or '6' otherwise an exception raise).

F. @staticmethod def valid_ip_format(address, ip_version) : Check for address validation and availability of ipV6 address. If there is no internet connection or unavailability, An error is thrown.

G. @staticmethod def validate_userInput(address, thread_num, waiting_time, start, end, sniffing_mode, ip_version, max_thread) : With the help of two other method, and some type casting, this method would be able to validate user inputs and if anything goes wrong, it will throw an error with suitable message.

H. @staticmethod def validate_ping_userInput(max_thread, address, waiting_time, packet_size) : waiting_time should be greater than 1000, packet size should be greater than 0, all ip addresses must be in valid form, as every ip address will has its own thread to send and receive packet, the number of addresses should be less than max thread. If any of above conditions are not satisfied an exception will be raised with exit_0 action.

5.4. Port Sniffer

This class puts together all the inputs and other data and create socket to recognize which ports are open. It takes **PortScanningModel** in init.

def __init__(self, port_data_model)

5.4.1. Member Data Fields

executing_thread : An array of threads that are running and checking a port.

ports_queue : An array of **portModels** which has been created after user entered some inputs.

sniffing_mode : The mode that user selected.

open_ports : A counter which holds the number of ports that recognized as open.

print_thread : An object of type **PrintThreading** which can print messages in separate thread concurrently.

5.4.2. Methods

A. def start(self) : It first assign ports by calling get_ports_by_mode() from **port_data_model** instance, then it start printing thread by calling start_print_thread() from print_thread instance. For the next move a while loop start until the **ports_queue** become empty, it first count the number of alive thread in **executing_thread** list, if it is zero, list is cleared and execute_check_procedure() gets called. After that all ports were checked, prompt_after_compeletion() will be called and show overall result for user.

B. def execute_check_procedure(self) : It first pop a port if available and create thread, the poped port will be passed to created thread and thread starts. These steps repeat **port_data_model.thread_number** times in order to satisfy the limit which user defined.

C. def check(self, port_model) : This is the target method for check procedure thread, it takes a **port_model** as its argument, then it create a socket base on ipVer, address, and ports that user defined and call connect_ex(). If the flag that connect_ex() returns is zero, it means that socket successfully connected, and a message will be shown to user with **print_thread** class and the socket will be closed.

6. ByteOrder

An enum class which represent two type of byte order in different systems

- little_endian = 1
- big_endian = 2

7. IPVersion

An enum class which represent two type of ip.

- v4 = socket.AF_INET
- v6 = socket.AF_INET6

8. RunMode

An enum class which represent two way of running the app.

- non_interactive = 0
- interactive = 1

8. Constant

A class which contains constant values.

9. ICMP Header

A data class which model the 8 bytes of header and data section if needed.

9.1. Data Field

code : It gives additional info for the message.

checksum : Used for error checking.

packet_id : The id of packet which was sent (or going to be sent) to destination.

sequence_number : Used for send and receiving acknowledgement.

format : The format which the header can parsed with.

9.2. Methods

A. def *get_icmp_header_from_packet(self, packet, unpack_format)* : Extract icmp header from packet and return icmpHeader data model.

10. IP Header

A data class that model the ip header of packet.

10.1. Data Field

ver :version of ip.

type : This field is provided features related to the QOS for data streaming or VOIP.

length : Size of ip datagram which normally is 20 bytes.

id : Used for identify fragments of an ip datagram.

flags : Three bit which can help to identify fragments.

ttl : Maximum time that the datagram will be live in the internet system.

protocol : Denotes that internet protocol is used in the latter portion of the datagram.

checksum : Used to check header for any error.

source_ip : A 32 bit address of source used for packet.

10.2. Methods

A. def *get_ip_header_from_packet(packet, unpack_format)* : Extract ip header from packet and return ipHeader data model.

11. Packet Model

A data class which model the packet data.

11.1. Data Field

address : IP address of source.

destination_address : IP address of destination.

waiting_time : An amount of time which app wait to receive response from sent packet.

icmp : ICMP Header of packet.

ipVer : The ip version that is used.

packet_format : The format used to extract packet data.

id :Used for identify the packet.

packet_size : The size of packet which will be sent to destination address.

sequence_number : Used for send and receiving acknowledgement.

11.2. Methods

A. def *get_packet_id_based_on_threads(self)* : Each ping operation is executed in one thread and we use that thread id to identify the packet sent and received. This function return the unique id for packet.

B. def *create_pakcet(self)* : create and return a packet from given data in model that can be sent to destination address.

12. PacketResponseModel

A data class which model the packet received from destination.

12.1. Data Field

receive_time : An amount of time which was taken that the response of the sent packet received.

received_packet_size : size of the packet which received.

ip_header :The IPHeader of received packet.

icmp : ICMP Header of received packet.

13. Ping Inputs Model

A data class which model the user inputs for ping service.

13.1. Data Field

host_address : IP address of destinations.

destination_address : IP address of destination.

waiting_time : An amount of time which app wait to receive response from sent packet.

icmp_max_received : ICMP Header of packet size limit.

packet_models : Used to hold the generated packet model from host addresses.

13.2. Methods

A. def *get_waiting_time_in_sec(self)* : Convert the waiting time to second unit.

B. def *get_packet_models(self)* : Generate packet model and store it from host addresses.

14. Ping Response Model

A data class which model the Ping overall statistics.

14.1. Data Field

identifier : A unique id that each thread has and each packet will identified by that .

packet_model : Used to temporary store the packet model that has been sent .

delays :Total delay to received response from the sent packet.

14.2. Methods

A. def *print_final_results(self)* : Print final statistics of ping operation .

B. def *print_sequential_result(self)* : Print statistics for every packet that was received.

C. def *print_timeout(self)* : Print the information about a sent packet that did not received response after passing timeout limits.

D. def *print_sending_error(self, message)* : Print the information about a packet that supposed to be sent but failed.

E. def *print_not_received(self)* : Print the information about a packet that supposed to be received but failed.

F. def *get_received_packet(self)* : Return the total number of packet that were received.

G. def *save_delay(self, sending_time, receiving_time)* : Calculate the delay and store it in ***delays***.

H. def *set_next_packet_model_delay(self, packet_model)* : Sets the next packet that should be sent.

I. def *get_analyzes(self)* : store and return max_rtt, min_rtt and avg_rtt from packets that has been sent.

15. CheckSum Factory

A factory class.which generate checksum for packets.

15.1. Data Field

raw_packet_data : The generated data for packet.

raw_packet_length : Length of the generated data for packet.

15.2. Methods

A. def *get_checksum(self)* : generate and return checksum (with network byte order) .

B. def *set_bytes(self)* : Return low and high bytes for checksum data.

C. def *check_odd_bytes(self)* : Check if the length of bytes are odd, if it is odd, it return last bytes and add to checksum byte to make it even.

D. def *get_packet_loss(self, sequence_number, packet_received)* : calculate packet loss percentage from given arguments.

16. Padding Factory

A factory class.which generate padding data for packet's checksum.

16.1. Data Field

packet_size : Size of packet.

16.2. Methods

A. def *get_padding_bytes(self)* : generate and return padding byte array.

17. Ping StartManager

When ***AppStartManager*** detected the AppMode, an instance of this class (inherited from StartManager class) will be created and that's where everything begin for ping service.

17.1. Data Field

max_threads : Maximum threads that can be created in order to respond to user requests.

17.2. Methods

A. def *run_in_non_interactive(self)* : an overriden method that parse argv and start ping operation by creating an instance of ping class and pass the user inputs to it.

17. Ping

when ***PingStartManager*** parsed user inputs this class will be created and start the procedure of ping by creating a thread for every host addresses or IPs.

17.1. Data Field

threads : Threads that was created to send and receive packets.

stop: A shared bool property used to signal the threads that they should keep running or continuing.

17.2. Methods

.....

References

1. python.org/
2. stackoverflow.com/