PyStack Documentation

Release 1

Robin David

February 28, 2013

CONTENTS

Pystack, is a python framework that allow to create small TCP/IP stacks in an easy manner in order to obtain a wanted behavior. The applications are multiples and there is currently no any module that provide similar functionnalities in python.

It is developped above the Scapy framework to bring some network stateful functionalities adding to it multiple protocols implementations themselves architectured into a stack which allow the different layers to communicate. All this **brings the IP stack in userland** and then allow to do anything on network packets like modifying the network stack behavior without patching the Linux kernel.

CHAPTER ONE

NEWS

1 March 2013 First version of the documentation generated using Sphinx

12 December 2012 First Alpha version of Pystack finalized

CHAPTER

TWO

DOCUMENTATION

2.1 Introduction

2.1.1 About PyStack

Pystack is a framework developped above the scapy framework to bring some network stateful functionality. Scapy is used in PyStack for input/output using raw sockets and more especialy `L2ListenSocket`. But also to craft and decode packets. Then pystack is built above multiple protocols implemen- tations themselves architectured into a sack which allow the different layers to communicate. It **brings the IP stack in userland** and then allow to do anything on network packets, modify the network stack behavior without patching the Linux kernel. PyStack took his first inspiration from the old project muXTCP presented at 22C3 and keep some of its principles.

Caution: PyStack is still at an experimental level of development. The stack want to stay minimalistic and then can become very unstable in case of specific event on the network. Moreover it can induce some side effect due to the usage of iptables.

2.1.2 How it works

PyStack has been implemented in python, and act as a subversive stack from the kernel point of view which have no control on it. To work simultaneously on the same host pystack need to alter the kernel stack behavior by blocking outgoing packets on the given ports using netfilter. Indeed when a subversive stack establish a connection to a remote host, any packet incoming packet from this host will be reset by the kernel which didn't instanciated the connection. That's why some host and port will be momentaneously blocked by the interfering stack. The only thing to remember is that pystack allow to personalize the TCP/IP stack behavior in a per/connection manner. The behavior can be different for any connection. Finally the overall advantage of it is that it run above the Linux Kernel stack so it does not disturb the stability of the whole system and the pystack is protected from Kernel panic because it works in userland.

2.1.3 What is the goal ?

The global goal of PyStack is to give the control of a network stack in userland. Basically in networking you can manipulate raw packets crafting them or manipulate packets at application layer with sockets modules. But in between no python module allow to manipulate from the Ethernet to Application layer. This is not a problem for protocol like UDP for which packets can be crafted easily but for stateful protocols like TCP it becomes far more complicated. Various things that can be accomplished with PyStack are:

Note: The main goal of this project is to provide a simple enough stack to allow anybody to hack into in order to obtain its own behavior.

- Modifying the protocols behavior (Ethernet, ARP, IP, TCP)
- Quickly prototyping protocols, or protocols functionnalities like (SYN Cookies, TCPCT, TFO TCP Fast Open) or any new fancy stuff
- Pentesting infrastructures playing with fragmentation, or weird behavior
- Fool fingerprinting tools or testing our own
- Or just get stack control from the top/bottom without hacking into the kernel ;)

2.2 Installation

2.2.1 Linux

PyStack has been especially designed to work on Linux. Here are the step to go through in order to get PyStack working:

1. Be sure the get a python version inferior to 3. Twisted does not work yet on Python 3.
2. Install all the dependencies which are Scapy and Twisted.
3. Download the code from Github and put it in your python libraries directory
4. Run PyStack as administrator

Note: All the tests were performed on python 2.7

2.2.2 Other OS

Windows

PyStack can not work on Windows with its current version because it uses netfilter to block certain packets. To get it working on Windows the only module to modify is kernel_filter and adapt it to work with the Windows Firewall. Once this is done. You just need to get scapy and twisted working and there is no reason that it wouldn't work on Windows. *This as not been tried*

Mac OS

The problem for Mac OS is similar to Windows (I think). But it might be simpler to get it working on because Mac OS is nearer than Windows of the Linux platform.

2.3 Framework Architecture

2.3.1 Modules/Submodules

All the class are located in a folder called *pystack*. Into this directory there is the following modules:

```
pystack
  |
  +-- kernel_filter
  |
  +-- pystack
```

```
  |
  +-- pystack_socket
  |
  +-- layers
        |
        + -- ...
```

- **kernel_filter**: contain a module that allow blocking incoming or outgoing packets of the kernel. Basically, this is just an interface with iptables. This should not require to be imported in your script directly because it is already ussed by the various layers.
- **pystack**: pystack is a module that provide a ready to use stack. It just create a stack assembling all the layers together and provide to right methods to add TCP or UDP application at the top of it.
- **pystack_socket**: This module is a tricky implementation of the python socket module. It reuse the exact same syntax than socket to make the usage of pystack similar to socket. So it implement the key functions to adapt to pystack. All the rest is kept from socket (so the socket module is imported in this module). Be careful pystack_socket does not support all the function,options and socket kinds. For new just SOCK_STREAM and SOCK_DGRAM socket are supported without options.
- **layers**: Contain all the different protocol implementation which is listed below.

layers contain all the protocol implementation and two associated class (scapy_io and layers) which will be discussed below. Any new protocol should be put in this folder. Feel free to add your own ;) For now the existing protocol implementations are:

```
layers
  |
  +-- layer
  +-- scapy_io
  +-- ethernet
  +-- arp
  +-- ip
  +-- tcp
  +-- udp
  +-- tcp_session
  +-- tcp_application
  +-- udp_application
  +-- dns
  +-- ...
```

- **layer**: layer is an abstract class. This is the mother class that give the layer structure than any child class have to implement. This is discussed in detail in the next section
- **scapy_io**: scapy_io cannot stricly be considered as a layer(does not extend layers) but it is a part of the stack because this class provide input/output functions. It takes an interface on which listening on and provides two ways of listening packets.
  - **reactor**: reactor is imported from twisted. A reactor is a special object on which we can register *Readers* and once the reactor launched it will try to read in all the readers. The main advantage is that on thread is needed for any reader. .. note:: reactor.run() is blocking. So once launched no any further instructions can be performed.
  - **thread**:This is an alternative method to reactor to be non-blocking. So if scapy_io is run with a thread the handle is given back and all the packet reception will be performed in the thread. Be careful this can lead thread access problems (improper reading etc).
- **ethernet**: Ethernet protocol implementation. ethernet module use scapy_io is "under layer" (hardcoded).
- **arp**: ARP protocol implementation. It manage all the MAC address resolution and hold by the way a cache of MAC/IP address association.

- **ip**: IP protocol implementation. It assure the routing of IP address to the associated protocol (tcp/udp) and assure also the fragmentation and reassembly at ip level. For this purpose this layer can temporarily hold ip fragments.
- **tcp**: This module just do the routing according to port source and port destination to the associated tcp_session.
- **udp**: Same as tcp but for udp segments.
- **tcp_session**: tcp_session contain tcp protcol behavior itself. It maintains the sequence and acknowledgement in addition to the state of the connection. The name "tcp_session" has nothing to do with the OSI session layer but the name fit perfectly the purpose of this module. Any TCP connection as client or server has its own tcp_session to keep the state of the connection. This module brings the stateful aspect to TCP.
- **tcp_application**: This class represent the layer 7 of the OSI protocol. This layer just deal with "string" object which are stacked when received. Protocol of layer 7 should inherit this class in order to use pystack.
- **udp_application**: Provide the same functionnalities than tcp_application but for UDP protocols. .. note:: There is no udp_session because this protocol is stateless, so udp_application are directly connected to udp layer.
- **dns**: DNS is a udp_application that allow to do DNS name resolution in a really basic manner. A name resolution is basically the only functionality provided by dns protocol(but it was needed for tcp for name resolution).

2.3.2 layer architecture

layer provide the basic layer structure that any protocol should implement. Among this structure the more important are the way the way layers communicate with the two upperlayers and lowerLayers dictionnary, but also the way to register layers each other.

Code explanation

```
class Layer(object):

    name = ""

    def __init__(self):
        self.lowerLayers = {}
        self.upperLayers = {}
        self.register_upper_layer("default", Default())
```

Class layer has an attribute called name which has to be modified by child class with the appropriate name. "name" will be used as key identifier in upperLayers and lowerLayers. Eg: Ethernet layer receive an IP packet, it will then look for the "IP" layer in upperLayer to forward the packet to.

In init, a layer has both dictionnary lowerLayers and upperLayers which will respectively hold handlers for layers under and above. Within this dictionnaries layers are identified by their name (IP,TCP, Raw ..). A default upperLayer is added to handle packets that does not match any other layer. Default does nothing when a packet is received, but you can customize the behavior of Default like logging packets etc.

The IP layer has for instance the following layers: lowerLayers{"default":ethernet} upperLayers{"TCP":tcp,"UDP":udp,"default":Default}

```
def register_upper_layer(self, name, layer): #Register the given layer in upperLayers with the given
    self.upperLayers[name] = layer

def register_lower_layer(self, name, layer): #Register the given layer in lowerLayers with the gven n

def register_layer_full(self, name, layer): #Register the given layer in upperLayers and itself as th
```

```
    self.register_upper_layer(name, layer)
    layer.register_lower_layer("default", self)

def register_layer(self, layer): #Idem as register_layer_full but use the layer name attribute as key
    self.register_layer_full(layer.name, layer)

def unregister_upper_layer(self, name): #Unregister the layer identified by name in upperLayers
    self.upperLayers.pop(name)
```

All this method are really useful for registering layers together.

The following methods are really important because they define a default behavior for sending, and forwarding packets from on layer to another.

```
def send_packet(self, packet, **kwargs):
    """By default when calling send_packet it forge the packet calling
    forge_packet and forward it to the lower layer calling transfer_packet"""
    self.transfer_packet(self.forge_packet(packet), **kwargs)

def forge_packet(self, packet, **kwargs):
    """By default does nothing but should be overriden by child class"""
    pass

def transfer_packet(self, packet, **kwargs):
    """Define the default behavior when a packet should be transfered to the lower layer.
    The default behavior is to call the send_packet of the default lowerlayer. This method
    can be overriden by child layers"""
    self.lowerLayers["default"].send_packet(packet, **kwargs)
```

When you want to send a packet in a layer you should call send_packet. send_packet will by default call the method which should be overriden and then call transfert_packet which by default call the send_packet of the "default" in lowerLayers. This is the basic process of a packet within a layer. Then this packet goes through all the layers until it is sent by scapy_io.

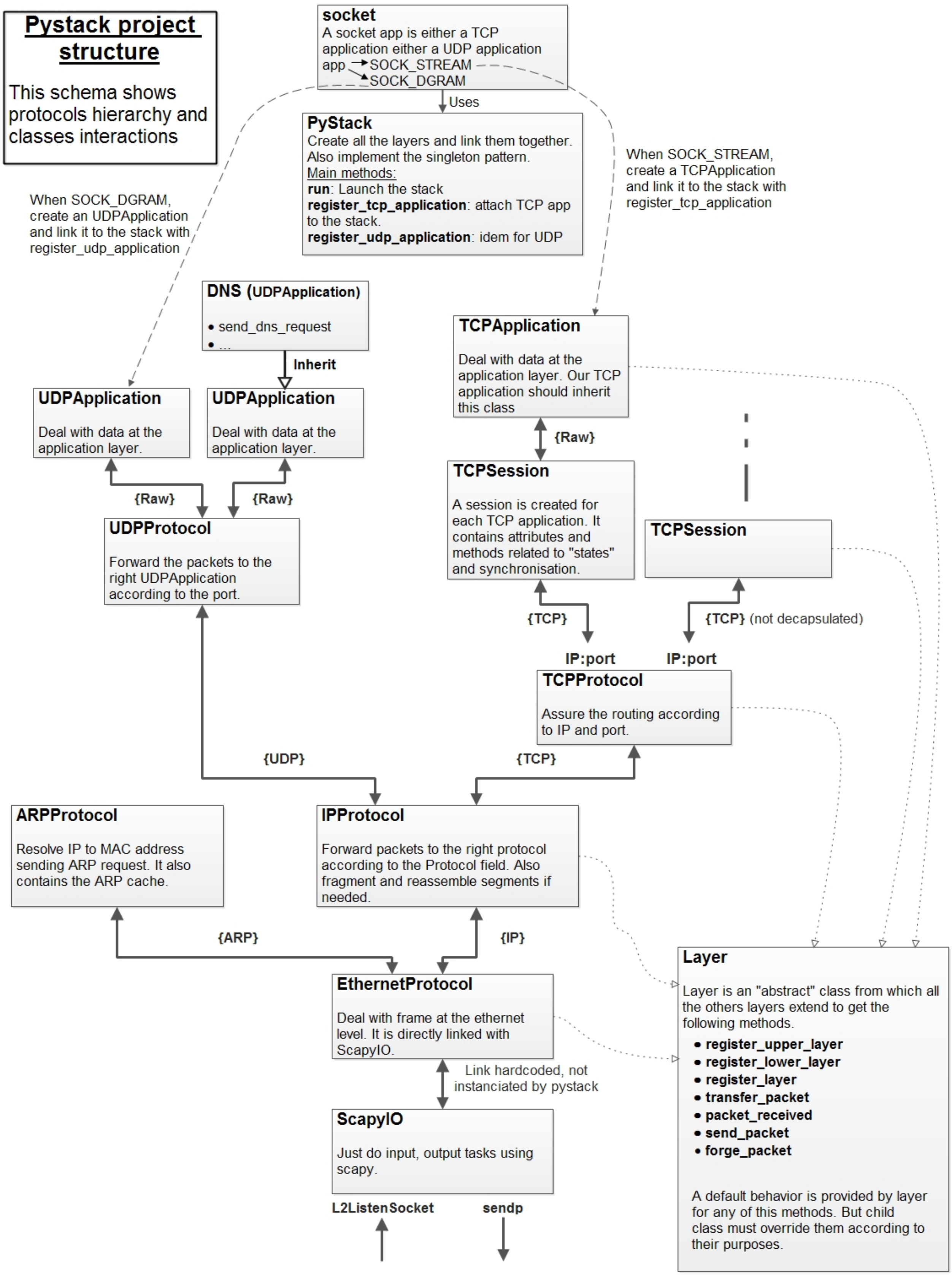
The second most important method after send_packet, is packet_received. It is called when a packet is received and should then contain all the packet processing. By default it "decapsulate the packet and send the payload to the upperlayer referenced by the payload name.

```
def packet_received(self, packet, **kwargs):
    target = self.upperLayers.get(packet.payload.name, self.upperLayers["default"])  #Get the handle
    kwargs[packet.name] = packet.fields
    target.packet_received(packet.payload, **kwargs)  #Call packet_received of the handler with the p
```

2.3.3 pystack

PyStack is a class that create a stack. It instanciate all the layers and register all them together. See the code for more. Another significant point about pystack is that the class implement the singleton pattern overriding the __*new*__ so that any component of the same program that will use pystack will manipulate the same stack "sharing" it (as it is the case with the real stack).

The following schema summarize the all structure of the project and what is built by pystack class.


Pystack project structure
This schema shows protocols hierarchy and classes interactions
socket
A socket app is either a TCP application either a UDP application
app
SOCK_STREAM
SOCK_DGRAM
Uses
PyStack
Create all the layers and link them together. Also implement the singleton pattern.
Main methods:
run: Launch the stack
register_tcp_application: attach TCP app to the stack.
register_udp_application: idem for UDP
When SOCK_DGRAM, create an UDPApplication and link it to the stack with register_udp_application
When SOCK_STREAM, create a TCPApplication and link it to the stack with register_tcp_application
DNS (UDPApplication)
• send_dns_request
• ...
Inherit
UDPApplication
Deal with data at the application layer.
UDPApplication
Deal with data at the application layer.
TCPApplication
Deal with data at the application layer. Our TCP application should inherit this class
{Raw}
{Raw}
{Raw}
UDPProtocol
Forward the packets to the right UDPApplication according to the port.
TCPSession
A session is created for each TCP application. It contains attributes and methods related to "states" and synchronisation.
TCPSession
{TCP}
{TCP} (not decapsulated)
IP:port
IP:port
TCPProtocol
Assure the routing according to IP and port.
{UDP}
{TCP}
ARPProtocol
Resolve IP to MAC address sending ARP request. It also contains the ARP cache.
IPProtocol
Forward packets to the right protocol according to the Protocol field. Also fragment and reassemble segments if needed.
{ARP}
{IP}
EthernetProtocol
Deal with frame at the ethernet level. It is directly linked with ScapyIO.
Link hardcoded, not instanciated by pystack
ScapyIO
Just do input, output tasks using scapy.
L2ListenSocket
sendp
Layer
Layer is an "abstract" class from which all the others layers extend to get the following methods.
• register_upper_layer
• register_lower_layer
• register_layer
• transfer_packet
• packet_received
• send_packet
• forge_packet
A default behavior is provided by layer for any of this methods. But child class must override them according to their purposes.

2.3.4 pystack_socket

pystack_socket intent to provide the same interface than socket but to use pystack. So the most critical functions had been reimplemented in a really really basic manner. All the rest is reused from socket. This will allow to use pystack in the same way than socket (but in more trivial). Indeed options, some functions and socket types are not supported. Only SOCK_STREAM, and SOCK_DGRAM are working. The __init__ methods show how tricky it is:

```
class socket:

    def __init__(self, family=AF_INET, type=SOCK_STREAM, proto=0, app=None):
        self.app = None
        self.blocking = True
        self.stack = PyStack()
        if family not in (AF_INET, AF_INET6, AF_UNIX):
            raise error("Address family not supported by protocol "+family)
        else:
            self.family = family
        if type not in (SOCK_DGRAM, SOCK_STREAM):#SOCK_RAW, SOCK_RDM, SOCK_SEQPACKET):
            raise error("Invalid argument "+type)
        else:
            self.type = type
            if app:
                self.app = app
            else:
                if type == SOCK_STREAM:
                    self.app = _TCPSocket()
                elif type == SOCK_DGRAM:
                    self.app = _UDPSocket()
        self.proto = proto
        if not app:
            if type == SOCK_STREAM:
                self.stack.register_tcp_application(self.app)
            elif type == SOCK_DGRAM:
                self.stack.register_udp_application(self.app)
            self.stack.run(doreactor=False)
```

Some comments about the code:

- Each time a socket is created a Pystack object is created but because it implement the singleton pattern only one among all the code will really be instanciated
- A family error is raised if not valid, but It is not taken in account anyway. (Only IPv4 is support for now)
- If the type not STREAM or DGRAM a type exception is raised whereas it should not but not implemented
- Depending of the type a _TCPSocket or a _UDPSocket is created. This two class just implement respectively TCPApplication and UDPApplication.
- Application are registered to the stack (so attached to UDP for _UDPSocket and linked to a TCPSession and attached to TCP for _TCPSocket)

All the other methods do the same taking the same kind of arguments than socket but dealing with it differently.

2.4 Usage

2.4.1 Stack crafting

Once a protocol modified or a new layer created changes must be applied to a network stack. There is two solution. The first solution is to modify directly the class **PyStack** located in *pystack.pystack*. The second solution is to recreate as small stack by hand according to our needs.

Rebuilding a stack by hand for a connection which is fairly simple. The following piece of code shows how to do it in the most easiest manner.

```
from pystack.layers.ethernet import EthernetProtocol
from pystack.layers.ip import IPProtocol
from pystack.layers.arp import ARPProtocol
from pystack.layers.tcp import TCPProtocol
from pystack.layers.tcp_session import TCPSession
from pystack.layers.tcp_application import TCPApplication

interface = "eth0"
eth = EthernetProtocol(interface)
ip = IPProtocol()
eth.register_layer(ip)
arp = ARPProtocol(interface)
eth.register_layer(arp)
tcp = TCPProtocol()
ip.register_layer(tcp)
```

Caution: The DNS layer has not been added, it not useful unless you intent to perfom name resolutions.

Then you can create a create for instance a tcp_application that will listen on port 7777. To do such, we should create a TCPSession a TCPApplication register them each other and then link the TCPSession to the TCP layer.

```
tcpsession = TCPSession(interface)
tcp.register_layer(tcpsession)
conn = TCPApplication()
tcpsession.register_layer(conn)
conn.bind(7777)
```

Finally we should start listening on the given interface using a reactor(True) or a thread(False)

```
eth.start_listening(doreactor=True)
```

2.4.2 Client

This section will show how to create a small TCP client using pystack. The usage of the PyStack class makes the client creation easy. The following program create a TCP client which connect to a web server. What is sent is not really important.

```
import time
from pystack.pystack import PyStack
from pystack.layers.tcp_application import TCPApplication

stack = PyStack() #Create a stack

conn = TCPApplication() #Create a TCPApplication
```

```
stack.register_tcp_application(conn) #Register the application to the stack which will manage to crea
stack.run(doreactor=False) #Run the stack to start listening using a thread to make it non-blocking

if conn.connect("myserver.com", 80): #Connect to the given server

    conn.send_packet("GET / HTTP/1.0\r\n.......\r\n\r\n") #Send the request to the server

    time.sleep(10) #Sleep to wait for an answer.

    conn.close() #Close the connection

stack.stop() #Stop the stack
```

Important:

There are important things to notice in this script:

- By default a tcp_application just store bytes received. To get the answer we should use *conn.fetch_data()*
- It is important at the end to stop the stack gently **otherwise iptables rules may remains in netfilter.**

2.4.3 Server

In order to make a server the bind method should be discussed because it differ from the socket bind approach.

classmethod **`bind`**(*self*, *port*[, *app=None*, *newinstance=False*])

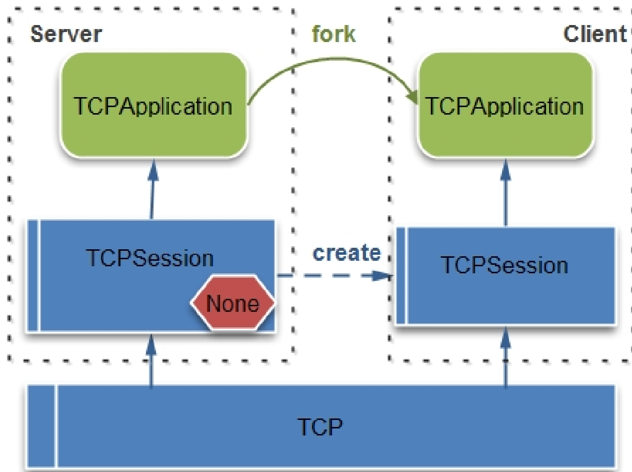
Call the bind method of the TCPSession with the given attributes.

Parameters

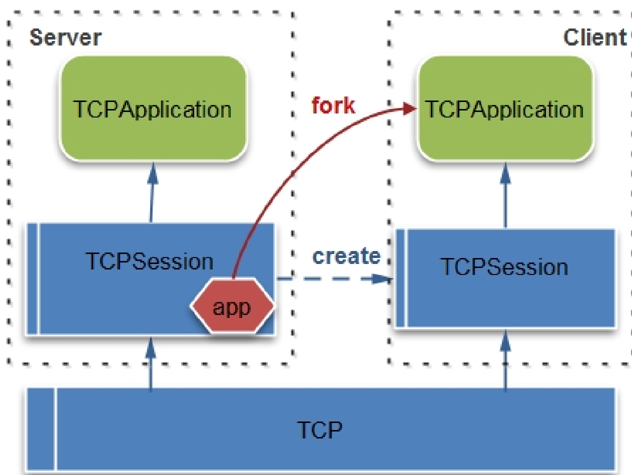
- **port** – port to listen on
- **app** – Should be a TCPApplication. All the clients connecting to the server will be attached to this application. If no app is provided the tcpapplication used is self !
- **newinstance** – Define if all the clients should be linked on the same tcpapplication (attribute app) or if it should be forked for each

The following schema summarize all the parameters combinations and the effect it produce when a client connect. We consider in this schema *conn = TCPApplication()*, and conn is the application that will be bind.

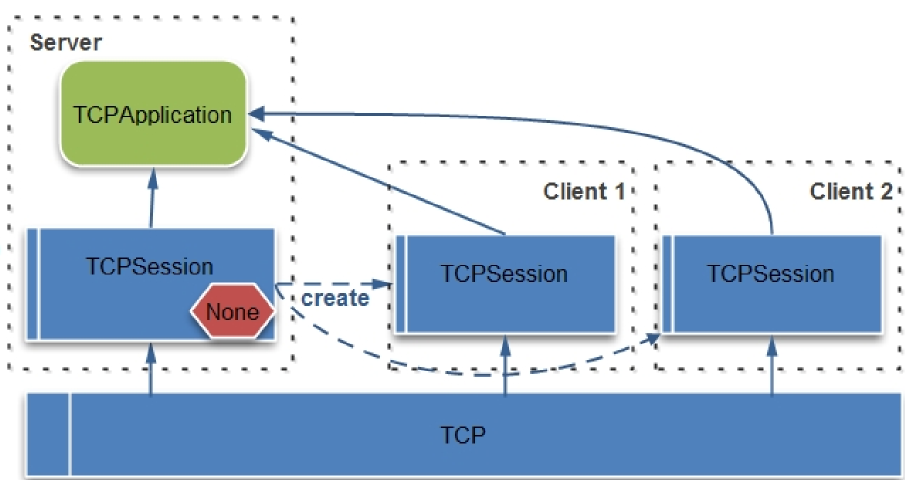
conn.bind(80,None,True)



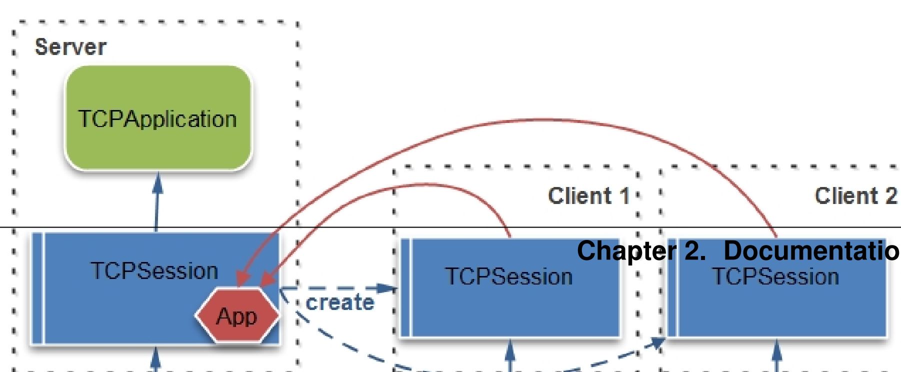
conn.bind(80,app,True)



conn.bind(80,None,False)



conn.bind(80,App,False)



In the end to make a server the code is similar to client. The only thing to modify is to change *conn.connect("myserver.com", 80)* by:

```
conn.bind(8888) #Without argument the TCPApplication use for new client is conn and the application
conn.listen(2)

s = conn.accept()
```

2.4.4 Modifying the stack behavior

There is no predefined way to modify the stack behavior. It depends of your needs of the protocols involved and of the requirements. Letting the user free of modifying anything in the code is the best solution.

The following example will override the _send_SYNACK method to modify the packet sent in response to a SYN. In this scenario we modify the sequence number to put it at an arbitrary value 5555.

```
class tcpsession_modified(TCPSession):

  def _send_SYNACK(self, packet):
      self.seqNo = 5555 #Change the sequence number
      self.nextAck = self.seqNo #Does not change the ack value
      self.send_packet(None, SYN+ACK) #Call send_packet without modifying flags
```

Then to make our clients and server to use this TCP session class instead of the classic one, either we recreate the stack by hand *Stack crafting* (above) or we modify the PyStack class to make it uses tcpsession_modified.

2.5 Examples

2.5.1 Echo server

The creation of an Echo server is really straightforward. It can be done with the following code:

```
from pystack.layers.tcp_application import TCPApplication
from pystak.pystack import PyStack

class EchoServer(TCPApplication):
    def packet_received(self, packet, **kwargs):
        self.send_packet(packet, **kwargs) #Just reply the exact same thing to the client

if __name__ =="__main__":
    stack = PyStack()

    echoserver = EchoServer()

    stack.register_tcp_application(echoserver)

    echoserver.bind(8888, echoserver, False)
    echoserver.listen(5)

    stack.run(doreactor=True)
```

Comments about the code:

- The EchoServer TCP application is fairly simple, we just override packet_received to reply to the client.

- Line 15 the bind arguments are very important, there is no need to keep information about the client and the processing is the same for all that's why all the client will have the same tcp application (echoserver).
- Default arguments for bind are *app=None, newinstance=False* so we could have call *echoserver.bind(8888)* because when no app is provided self is used.
- Line 18 we start the stack usin reactor, so that it keep the handle and the script wait for Ctrl+C. (If we had use thread the script would have ended up directly)
- Also line 18 when the user type Ctrl+C the stack.stop() is called automatically.

2.5.2 Client/Server using pysocket

To make socket in a similar way than the official "socket module", pystack_socket provides an interface to socket. The only thing to do is

```
import psytack.pystack_socket as socket
```

Then a client or a server can be done in a similar way than with socket except that, at the end the stack should be stopped. The following sample shows a basic example:

```
import pystack.pystack_socket as socket
import time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

if s.connect(("myserver.com", 80)):
    s.sendall('Hello, world\n')
    data = s.recv(1024)
    print('Received '+ repr(data))
    s.close()
else:
    print("Not connected")

time.sleep(4) #Wait a little to avoid to get the stack destroyed before the socket is gently closed.
s.stop() #To stop the stack
```

Like in socket module a server can be written replacing the connect by:

```
s.bind(("localhost",PORT))
s.listen(2)
cli, addr = s.accept()
```

Important: The first parameter sent by bind is ignored by pystack. The server will only listen on the interface on which the stack is listening on. By default the stack use the default interface. For instance if a server is listening on the address 192.168.0.1 on port 80 trying to access the server locally will certainly fail because the system may resolve 192.168.0.1 as 127.0.0.1.

2.5.3 socket module hijacking

Thank's to pystack_socket it can be interesting to force certain scripts or program to use pystack instead of socket without modifying the source code. As soon as no extra socket functionalities are used this might succeed. This section shows how to do it. This is tricky and it does not work all the time but it might fit in simple cases. Let's take the following code that use socket.

```
import socket

class Client():
    def __init__(self):
        pass

    def run(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(("myserver.com", 5555))
        s.sendall('Hello, world\n')
        data = s.recv(1024)
        s.close()
        print('Received'+ repr(data))
```

What we will try to do is to make *Client* to use pystack instead of socket. The following script does it:

```
1  import pystack.pystack_socket
2  import sys
3  sys.modules["socket_original"] = sys.modules["socket"]
4  sys.modules["socket"] = pystack_socket
5
6  from test_client import Client
7  c =Client()
8  c.run()
9
10 from pystack.pystack import PyStack
11 s = PyStack() #Retrieve the pystack instance to stop it
12 s.stop()
13
14 sys.modules["socket"] = sys.modules["socket_original"]
15 sys.modules.pop("socket_original")
```

Comments about the script:

- **Line 1-5**: Replace the socket module in sys by pystack_socket
- **Line 7-9**: Import and launch the Client
- **Line 11-13**: Import pystack create a PyStack object but because it implement the singleton pattern we retrieve the only instance of PyStack and we stop it.
- **Line 15-16**: Put back the genuine socket module in sys.modules

2.5.4 Web server

Making a Web server is a complex tasks. We will tkae advantage here of *twisted* functionalities. We will only manage to receive and send request. All there serving content part is delegated to twisted. The method used below is inspired from the one used in muXTCP . We will use the class Site taken from twisted.web.server that allow to serve a given static directory as web server. The the trick is located in the Site object which have an attribute called *transport* which take care of input/output tasks. We will define the transport attribute to be our TCPApplication which implies to implement additional methods. The following class **WebServer** inherit from both TCPApplication for the pystack part and _ConsumerMixin for the twisted part.

> **Warning:** There is certainly a nicer way to do it, but this not the purpose of this example.

```
from pystack.layers.tcp_application import TCPApplication
from twisted.web.server import Site
```

```
from twisted.web import static
from twisted.internet.abstract import FileDescriptor
from twisted.internet.abstract import _ConsumerMixin
import os

class WebServer(TCPApplication, _ConsumerMixin):
    disconnecting = False #Required by twisted
    connected = True
    disconnected = False

    def __init__(self):
        TCPApplication.__init__(self)
        _ConsumerMixin.__init__(self)
        self.app = Site(static.File(os.path.abspath("./sitetest"))).buildProtocol("test")
        self.app.transport = self #Because we define self as transport we have to implement function

    def packet_received(self, packet, **kwargs): #Override TCPApplication packet_received to call the
        self.lastclient = kwargs["id"]
        try:
            print("Request received")
            self.app.dataReceived(packet)
        except Exception, e:
            print("Something is wrong in the request:"+ str(e))

    def write(self, data):
        print "data to send"
        while len(data) > 0:
            x = data[0:1000]
            data = data[1000:]
            #self.send_data(x)
            self.send_packet(x, **{"id":self.lastclient})

    def getPeer(self):
        class X:
            host = "myHost"
            port = "myPort"
        return X()

    def getHost(self):
        return self.getPeer()

    def writeSequence(self, iovec):
        self.write("".join(iovec))

    def loseConnection(self):
        pass

    def getvalue(self):
        pass
```

getPeer, getHost, loseConnection and getvalue should be present to work even though we didn't implemented them. This is for the TCPApplication, the instanciation and registration toward the stack is classic.

```
if __name__ =="__main__":
    from pystack.pystack import PyStack
    stack = PyStack()

    webserver = WebServer()
```

```
stack.register_tcp_application(webserver)

webserver.bind(80, app=webserver, newinstance=True)
webserver.listen(2)

stack.run(doreactor=True)
```

Error: A new webserver instance should be instanciated for each new client because twisted does not accept multiples request on the same _ConsumerMixin more than once. (Which is also problematic for a single client)

2.5.5 SSH server

Creating an SSH server is made quite simple thank's to all the twisted functionalities. It have globaly the same structure than WebServer except that we will create a *unix.UnixSSHRealm* instead of a Site.

Error: During the test I also had a problem with OpenSSHFactory which failed to read my keys. This problem is independant of pystack and is certainly due to twisted itself. This led me to create my own OpenSSHFactory fixing to problem which was located in getPrivateKeys.

```
from pystack.layers.tcp_application import TCPApplication
from twisted.internet.abstract import _ConsumerMixin
from twisted.conch import checkers, unix
from twisted.conch.openssh_compat import factory
from twisted.conch.openssh_compat.factory import OpenSSHFactory
from twisted.cred import portal, checkers as chk

class MyFactory(OpenSSHFactory):
    '''I need to create my factory because OpenSSHFactory fail when reading /etc/ssh and all keys
    Because some are not recognised it return None but no test is made
    So I just added "if key:" at the fourth last line of getPrivateKeys'''

    def getPrivateKeys(self):
        from twisted.python import log
        from twisted.python.util import runAsEffectiveUser
        from twisted.conch.ssh import keys
        import os, errno
        privateKeys = {}
        for filename in os.listdir(self.dataRoot):
            if filename[:9] == 'ssh_host_' and filename[-4:]=='_key':
                fullPath = os.path.join(self.dataRoot, filename)
                try:
                    key = keys.Key.fromFile(fullPath)
                except IOError, e:
                    if e.errno == errno.EACCES:
                        # Not allowed, let's switch to root
                        key = runAsEffectiveUser(0, 0, keys.Key.fromFile, fullPath)
                        keyType = keys.objectType(key.keyObject)
                        privateKeys[keyType] = key
                    else:
                        raise
                except Exception, e:
                    log.msg('bad private key file %s: %s' % (filename, e))
                else:
                    if key: #Just to add this fucking Line !
                        keyType = keys.objectType(key.keyObject)
```

```
                        privateKeys[keyType] = key
        return privateKeys

class SSHServer(TCPApplication, _ConsumerMixin):
    disconnecting = False #Required by twisted
    connected = True
    disconnected = False

    def __init__(self):
        TCPApplication.__init__(self)
        _ConsumerMixin.__init__(self)

        #t = factory.OpenSSHFactory()
        t = MyFactory() #Use my factory instead of the original one
        t.portal = portal.Portal(unix.UnixSSHRealm())
        t.portal.registerChecker(checkers.UNIXPasswordDatabase())
        t.portal.registerChecker(checkers.SSHPublicKeyDatabase())
        if checkers.pamauth:
            t.portal.registerChecker(chk.PluggableAuthenticationModulesChecker())
        t.dataRoot = '/etc/ssh'
        t.moduliRoot = '/etc/ssh'

        t.startFactory()
        self.app = t.buildProtocol("test")
        self.app.transport = self

    def connection_made(self):
        self.app.connectionMade()

    def packet_received(self, packet, **kwargs): #Override TCPApplication packet_received to call the
        try:
            print("Request received")
            self.app.dataReceived(packet)
        except Exception, e:
            print("Something is wrong in the request:"+ str(e))

    def write(self, data):
        print("Write data")
        while len(data) > 0:
            x = data[0:1000]
            data = data[1000:]
            #self.send_data(x)
            self.send_packet(x)

    def getPeer(self):
        class X:
            host = "myHost"
            port = "myPort"
        return X()

    def getHost(self):
        return self.getPeer()

    def writeSequence(self, iovec):
        self.write("".join(iovec))

    def logPrefix(self):
        return "pystackSSHServer"
```

```
    def setTcpNoDelay(self, tog):
        pass

    def loseConnection(self):
        pass

    def getvalue(self):
        pass
```

Then starting the server works the exact same manner than WebServer

Caution: The close of a session does not always work fine.

```
if __name__ =="__main__":
    from pystack.pystack import PyStack
    stack = PyStack()

    sshserver = SSHServer()
    stack.register_tcp_application(sshserver)

    sshserver.bind(80, app=sshserver, newinstance=True)
    sshserver.listen(2)

    stack.run(doreactor=True)
```

2.6 PyStack development

2.6.1 About the project

PyStack use Github as repository and is available at https://github.com/RobinDavid/pystack .

Because I am the only developer of the project I use Github to host it. Feel free to fork it and to contribute enhancing the stack functionalities or by adding new protocols. I will be happy to merge them into the main repo.

2.6.2 How to Contribute

- Fork the project and add new functionalities
- Add explanation or examples to this documentation
- Found a bug, report it on issues

2.6.3 TODO list

Here is the list of all the enhancement that are planned to be added to the project:

- Writing support for IPv6 protocol
- In PyStack when listening, listening on all available interface (lo) included and ensure the routing between the different interfaces. *This represent a lot of work*
- Writing a filter for ARP which packets are not blocked
- Writing a kernel module to decide either to keep a packet or not (better than using iptables but less portables)

- Manage manual fragmentation (now use fragment function of scapy)

Technical todo:

- Send ICMP time exceeded on reassembly timeout
- In send_packet of TCPSession put ack value to 0 if ack flag not set
- Create a timer hypervisor, which can call a callback method when a timer exceeded.
- Add a timestamp to each ARP entries in the cache to make them expire after 20 minutes
- Manage simultaneous passive and active close deadlock.

2.6.4 Extending Pystack adding layers/protocols

If you create a new layer/protocol feel free to contact me, and I will manage to merge it with the current to make it available for all.

To achieve such the best is to fork the project on Github apply our changes and then I merge the modification in the main branch or in a devel branch.

2.7 IP Stack compliance

2.7.1 RFC1122 Compliance

Note: This page intent to provide a brief overview of the functionalities provided regarding the RFC1122 about IP stack requirements. The informations provided are not 100% accurate.

Requirements for Internet Hosts

Communication Layers

Link Layer requirements

May	Trailer encapsulation	No
Must	Not send trailer by default	Yes
Must	be able to send and receive RFC 894 Ethernet	Yes
Should	receive RFC 1042 (IEEE 802) encapsulation	No
May	Send RFC 1042 encapsulation	No
Must	report link-layer broadcasts to the IP layer	No
Must	pass the IP TOS value to the link layer	Yes (p.1175)

IP Requirements

Must	implement IP and ICMP	No (no ICMP yet)
Must	handle remote multihoming in application layer	No (I don't think so)
May	support local multihoming	No
Must	meet router specifications if forwarding datagrams	No
Must	provide configuration switch for embedded router functionality	No
Must	not enable routing based on number of interfaces	Yes
Should	log discarded datagrams, including the contents of the datagram	No
Must	silently discard datagrams that arrive with an IP version other than 4	Yes
Must	verify IP checksum and silently discard an invalid datagram	No (not discarded)
Must	support subnet addressing (RFC 950)	No
Must	transmit packets with host's own IP address as the source address	Yes
Must	silently discard datagrams not destined for the host	Yes
Must	silently discard datagrams with bad source address	Yes
Must	support reassembly	Yes (Hellyeah !)
May	retain same ID field in identical datagrams	No
Must	allow the transport layer to set TOS	No
Must	pass received TOS up to transport layer	Yes
Should	not use RFC 795 [Postel 1981d] link-layer mappings for TOS	Yes (no mapping is made :p)
Must	not send packet with TTL of 0	No (no verification made)
Must	not discard received packets with a TTL less than 2	Yes
Must	allow transport layer to set TTL	Yes
Must	enable configuration of a fixed TTL	No

Multihoming

Should	select, as the source address for a reply, the specific address received as the destination address of the request	Yes
Must	allow application to choose local IP address	No (only listen 1 interface)
May	silently discard datagrams addressed to an interface other than the one on which it is received	**Yes (cause it listen on** one interface)
May	require packets to exit the system through the interface...	No

Broadcast

Must	not select an IP broadcast address as a source address	Yes (no broadcast)
Should	accept an all-0s or all-1s broadcast address	Yes (does not make the difference)
May	support configurable option to send all 0s or all 1s as the broadcast	No
Must	recognize all broadcast address formats	No
Must	use an IP broadcast or IP multicast destination address in a link-layer broadcast	No
Should	silently discard link-layer broadcasts when the packet does not specify an IP broadcast address as its destination	No
Should	use limited broadcast address for connected networks	No

IP Interface

Must	allow transport layer to use all IP mechanisms	Yes (giving transport layer access to all ip headers)
Must	pass interface identification up to transport layer	No (but quite important)
Must	pass all IP options to transport layer	No (but easily doable)
Must	allow transport layer to send ICMP port unreachable and any of the ICMP query messages	**No (but easily doable with transversal_layer_acces** if icmp implemented)
Must	pass the following ICMP messages to the transport layer:destination unreachable, source quench, echo reply, timestamp, reply, and time exceeded	No (but like previous point)
Must	include contents of ICMP message in ICMP message passed to the transport layer	No (idem)
Should	be able to leap tall buildings at a single bound	No (what's that ?)

IP Options Requirements

Must	allow transport layer to send IP options	No (but why not)
Must	pass all IP options received to higher layer	No (but why not)
Must	silently ignore unknown options	Yes (ignore them all :p)
May	support the security option	No
Should	not send the stream identifier option and must ignore it in received datagrams	Yes (??)
May	support the record route option	No
May	support the timestamp option	No
Must	support originating a source route and must be able to act as the final destination of a source route	No
Must	pass a datagram with completed source route up to the transport layer	Yes (Why won't it)
Must	build correct (nonredundant) return route	No
Must	not send multiple source route options in one header	No

Source route forwarding

May	support packet forwarding with the source route option	No
Must	obey corresponding router rules while processing source routes	No
Must	update TTL according to gateway rules	No
Must	generate ICMP error codes 4 and 5	No
Must	allow the IP source address of a source routed packet to not be an IP address of the forwarding host	No
Must	update timestamp and record route options	No
Must	support a configurable switch for nonlocal source routing	No
Must	satisfy gateway access rules for nonlocal source routing	No
Should	send an ICMP destination unreachable error (when forwarding fail)	No

IP Fragmentation and Reassembly

Must	be able to reassemble incoming datagrams of at least 576 bytes	Yes
Should	support a configurable or indefinite maximum size for incoming datagrams	Yes (infinite)
Must	provide a mechanism for the transport layer to learn the maximum datagram size to receive.	Yes
Must	send ICMP time exceeded error on reassembly timeout	No (but doable)
Should	support a fixed reassembly timeout value	Yes
Must	provide the MMS_S to higher layers	No
May	support local fragmentation of outgoing packets	Yes
Must	not allow transport layer to send a message larger than MMS_S	No (but fragment it in ip layer in this case)
Should	not send messages larger than 576 bytes to a remote destination in the absence of other information regarding the path MTU to the destination	No (I don't care :p)
May	support an all-subnets-MTU configuration flag	No

ICMP Requirements

Must	silently discard ICMP messages with unknown type	No
May	include more than 8 bytes of the original datagram	No
Must	return the header and data unchanged from the received datagram	No
Must	demultiplex received ICMP error message to transport protocol	No
Should	send ICMP error messages with a TOS field of 0	No
Must	not send an ICMP error message caused by a previous ICMP error message	No
Must	not send an ICMP error message caused by an IP broadcast or IP multicast datagram	No
Must	not send an ICMP error message caused by a link-layer broadcast	No
Must	not send an ICMP error message caused by a noninitial fragment	No
Must	not send an ICMP error message caused by a datagram with nonunique source address	No
Must	return ICMP error messages when not prohibited	No
Should	generate ICMP destination unreachable	No
Must	pass ICMP destination unreachable to higher layer	No
Should	respond to destination unreachable error	No
Must	interpret destination unreachable as only a hint, as it may indicate a transient condition	No
Must	not send an ICMP redirect when configured as a host.	No
Must	update route cache when an ICMP redirect is received	No
Must	handle both host and network redirects	No
Should	discard illegal redirects	No
May	send source quench if memory is unavailable	No
Must	pass source quench to higher layer	No
Should	respond to source quench in higher layer	No
Must	pass time exceeded error to transport layer	No
Should	send parameter problem errors	No
Must	pass parameter problem errors to transport layer	No
May	report parameter problem errors to process	No
Must	support an echo server and should support an echo client	No
May	discard echo requests to a broadcast address	No
May	discard echo request to multicast address	No
Must	use specific destination address as echo reply source	No
Must	return echo request data in echo reply	No
Must	pass echo reply to higher layer	No

Continued on next page

Table 2.1 – continued from previous page

Must	reflect record route and timestamp options in ICMP echo request message	No
Must	reverse and reflect source route option	No
Should	not support the ICMP information request or reply	No
May	implement the ICMP timestamp request and timestamp reply messages	No
Must	minimize timestamp delay variability	No
May	silently discard broadcast timestamp request	No
May	silently discard multicast timestamp requests	No
Must	use specific destination address as timestamp reply source address	No
Should	reflect record route and timestamp options in an ICMP timestamp request	No
Must	reverse and reflect source route option in ICMP timestamp request	No
Must	pass timestamp reply to higher layer	No
Must	obey rules for standard timestamp value	No
Must	provide a configurable method for selecting the address mask selection method for an inter	No
Must	support static configuration of address mask	No
May	get address mask dynamically during system initialization	No
May	get address with an ICMP address mask request and reply messages	No
Must	retransmit address mask request if no reply	No
Should	assume default mask if no reply is received	No
Must	update address mask from first reply only	No
Should	perform reasonableness check on any installed address mask	No
Must	not send unauthorized address mask reply message and must be explicitly configured as agent	No
Should	support an associated address mask authority flag with each address mask configuration	No
Must	broadcast address mask reply when initialized	No

Multicasting requirements

Should	support local IP multicasting (RFC 1112)	No
Should	join the all-hosts group at start-up	No
Should	provide a mechanism for higher layers to discover an interface's IP multicast capability	No

IGMP Requirements

May	support IGMP (RFC 1112)	No

Routing Requirements

Must	use address mask in determining whether a datagram's destination is on a connected network	Yes (but made by scapy)
Must	operate correctly in a minimal environment when there are no routers	Yes (it should I hope)
Must	keep a "route cache" of mappings to next-hop routers	Yes
Should	treat a received network redirect the same as a host redirect	No
Must	use a default router when no entry exists for the destination in the routing table	Yes
Must	support multiple default routers	No (i don't think so)
May	implement a table of static routes	Yes
May	include a flag with each static route specifying whether or not the route can be overridden by a redirect	No
May	allow the routing table key to be a complete host address and not just a network address	Yes
Should	include the TOS in the routing table entry	No (I do not hold the routing table directly)
Must	be able to detect the failure of a next-hop router that appears as the gateway field in the routing table and be able to choose an alternate next-hop router	No
Should	not assume that a route is good forever	Yes (for sure)
Must	not ping routers continuously (ICMP echo request)	Yes
Must	use pinging of a router only when traffic is being sent to that router	No
Should	allow higher and lower layers to give positive and negative advice	No
Must	switch to another default router when the existing default fails	No
Must	**allow the following information to be configured manually in the routing table: IP addre** mask, list of defaults	No

ARP Requirements

Must	provide a mechanism to flush out-of-date ARP entries	No (but easily doable)
Must	include a mechanism to prevent ARP flooding	No
Should	save (rather than discard) at least one (the latest) packet of each set of packets destined to the same unresolved IP address	Yes

UDP Requirements

Should	send ICMP port unreachable	No
Must	pass received IP options to application	No (just ipsrc, ipdst)
Must	allow application to specify IP options to send	No
Must	pass IP options down to IP layer	No
Must	pass received ICMP messages to application	No
Must	be able to generate and verify UDP checksum	Yes (made by scapy)
Must	silently discard datagrams with bad checksum	No
May	allow sending application to specify whether outgoing checksum is calculated, but must default to on	No
May	allow receiving application to specify whether received UDP datagrams without a checksum	No
Must	pass destination IP address to application	Yes
Must	allow application to specify local IP address to be used when sending a UDP datagram	No
Must	allow application to specify wildcard local IP address	No
Should	allow application to learn of the local address that was chosen	No
Must	silently discard a received UDP datagram with an invalid source IP address	No
Must	send a valid IP source address	Yes
Must	provide the full IP interface from Section 3.4 of RFC 1122	??
Must	allow application to specify TTL, TOS, and IP options for output datagrams	No
May	pass received TOS to application	No

TCP Requirements

PSH

May	aggregate data sent by the user without the PSH flag	No
May	queue data received without the PSH flag	No
Sender	should collapse successive PSH flags when it packetizes data	No
May	implement PSH flag on write calls	Yes
Since	PSH flag is not part of the write call, must not buffer data indefinitely	No
May	pass received PSH flag to application	Yes
Should	send maximum-sized segment whenever possible, to improve performance	No

Window

<table>
<tr><td>Must | treat window size as an unsigned number.</td><td>Yes</td></tr>
<tr><td>Receiver| must not shrink the window</td><td>Yes</td></tr>
<tr><td>Sender | must be robust against window shrinking</td><td>No</td></tr>
<tr><td>May | keep offered receive window closed indefinitely</td><td>No</td></tr>
<tr><td>Sender | must probe a zero window</td><td>No</td></tr>
<tr><td>Should | send first zero-window probe when the window has been closed for the RTO</td><td>No</td></tr>
<tr><td>Should | exponentially increase the interval between successive probes</td><td>No</td></tr>
<tr><td>Must | allow peer's window to stay closed indefinitely</td><td>No</td></tr>
<tr><td>Sender | must not timeout a connection just because the other end keeps advertising a zero window</td><td>No</td></tr>
</table>

Urgent Data

Must	have urgent pointer point to last byte of urgent data	No
Must	support a sequence of urgent data of any length	Yes
Must	inform the receiving process (1) when TCP receives an urgent pointer and there was no previously pending urgent data	No
Must	be a way for the process to determine how much urgent data remains	No

TCP Options

Must	be able to receive TCP options in any segment	Yes
Must	ignore any options not supported	Yes
Must	cope with an illegal option length	No
Must	implement both sending and receiving the MSS option	Yes
Should	send an MSS option in every SYN when its receive MSS	yes
Should	If an MSS option is not received with a SYN, must assume a default MSS of 536	No
Must	calculate the "effective send MSS."	Yes

TCP Checksums

Must	generate a TCP checksum in outgoing segments and must verify received checksums	Yes

ISN

Must	use the specified clock-driven selection from RFC 793	No

Opening connections

Must	support simultaneous open attempts	No
Must	keep track of whether it reached the SYN_RCVD state from the LISTEN or SYN_SENT states	No (but could be interesting to know)
Must	A passive open must not affect previously created connections	Yes
Must	allow a listening socket with a given local port at the same time that another socket with the same local port is in the SYN_SENT or SYN_RCVD state	Yes
Must	ask IP to select a local IP address to be used as the source IP address when the source IP address is not specified by the process performing an active open on a multihomed host	No
Must	continue to use the same source IP address for all segment sent on a connnection	Yes
Must	not allow an active open for a broadcast or multicast foreign address	No
Must	ignore incoming SYNs with an invalid source address	No

Closing Connections

Should	allow an RST to contain data	Yes
Must	inform process whether other end closed the connection normally (FIN normal, RST aborted)	No
May	implement a half-close	No
Must	If the process completely closes a connection TCP should send RST indicating data was lost	Yes
Must	linger in TIME_WAIT state for twice the MSL	No
May	accept a new SYN from a peer to reopen a connection directly from the TIME_WAIT state	No (cause TIME_WAIT not implemented)

Retransmission

Must	implement Van Jacobson's slow start and congestion avoidance	No
May	reuse the same IP identifier field when a retransmission is identical to the original packet	Yes may
Must	implement Jacobson's algorithm for calculating the RTO and Karn's algorithm for selecting the RTT measurements	No
Must	include an exponential backoff for successive RTO values	No
Must	Retransmission of SYN segments should use the same algorithm as data segments	Yes
Should	initialize estimation parameters to calculate an initial RTO of 3 seconds	Yes (static)
Should	have a lower bound on the RTO measured in fractions of a second and an upper bound of twice the MSL	No

Generating ACKs

Should	queue out-of-order segments	Yes
Must	process all queued segments before sending any ACKs	No
May	generate an immediate ACK for an out-of-order segment	Yes
Should	implement delayed ACKs and the delay must be less than 0.5 second	No
Should	send an ACK for at least every second segment	No
Must	include silly window syndrome avoidance in the receiver	No

Sending Data

Must	The TTL value for TCP segments must be configurable	No
Must	include sender silly window syndrome avoidance	No
Should	implement the Nagle algorithm	No
Must	allow a process to disable the Nagle algorithm on a given connection	No

Connection Failures

Must	pass negative advice to IP when the number of retransmissions for a given segment exceeds some value R1	No (Figure 25.26)
Must	close a connection when the number of retransmissions for a given segment exceeds some value R2	No (Figure 25.26)
Must	allow process to set the value of R2	No (Figure 25.26)
Should	inform the process when R1 is reached and before R2 is reached	No
Should	default R1 to at least 3 retransmissions and R2 to at least 100 seconds	No
Must	handle SYN retransmissions in the same general way as data retransmissions	No
Must	set R2 to at least 3 minutes for a SYN	No

Keepalive Packets

May	provide keepalives	No
Must	allow process to turn keepalives on or off, and must default to off	No
Must	send keepalives only when connection is idle for a given period	No
Must	allow the keepalive interval to be configurable and must default to no less than 2 hours	No
Must	not interpret the failure to respond to any given probe as a dead connection	No

IP Options

Must	ignore received IP options it doesn't understand	Yes
May	support the timestamp and record route options in received segments	No (but easily doable)
Must	save a received source route in a connection that is passively opened and use the return route for all segments sent on this connection	No

Receiving ICMP Messages from IP

Receipt of an ICMP source quench should trigger slow start	
Receipt of a network unreachable, host unreachable, or source route failed must not cause TCP to abort the connection and the process	
Receipt of a protocol unreachable, port unreachable, or fragmentation required and DF bit set should abort an existing connection	
Should	handle time exceeded and parameter problem errors

Application Programming Interface

Must	be a method for reporting soft errors to the process normally in an asynchronous fashion	No
Must	allow process to specify TOS for segments sent on a connection	No
May	pass most recently received TOS to process	No
May	implement a "flush" call	No
Must	allow process to specify local IP address before either an active open or a passive open	Yes (called by bind/accept or connect

2.8 Troubleshooting

2.8.1 FAQ

Pystack let iptables rules in filter table

This mean the stack is been shutdown suddently or that as socket has not been closed. Iptables is a weakness of pystack because it induce this kind of side effects.

2.8.2 Getting Help

For any help you can open an issue on Github I will be directly notified.

genindex

CHAPTER

THREE

DOWNLOAD

Stable 1.0:

- Source code
- Source .gz

Latest dev:

- Source code

CHAPTER

FOUR

CHANGES

1.0:

- Initial version with Ethernet, ARP, IP, TCP, UDP, DNS support
- Relative stability for TCP protocol sessions
- Code comments with docstring
- pystack module which assemble a fully working stack
- pysocket which recreate socket module but using pystack

CHAPTER

FIVE

CONTACTS

For any questions I recommend you to contact me on Github.

As I am the only maintainer of the project I might take some time to reply but be sure I will find a minute to reply you as soon as I can ;)

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX

B