CNP3-Practice Documentation

Release 0.0

Olivier Bonaventure Mickael Hoerdt, Laurent Vanbever and Virginie van den Schriek

December 03, 2009

CONTENTS

This document contains the questions for the practical part of the INGI2141 course during the 2009-2010 academic year. The html and pdf versions will be updated every week. These exercises have been written by Olivier Bonaventure, Mickael Hoerdt, Laurent Vanbever and Virginie van den Schriek

CHAPTER

ONE

THE APPLICATION LAYER

This set of question covers several application layer protocols. We expect that each group will distribute the questions among the different members of the group. The best solution is to have two answers from different students for each question. This would allow a discussion during the synthesis phases if the students disagree. Most questions require to do some manipulations with simple networking tools or to read information on the web. Given the size of the groups, we do not expect that a student would spend more than 3 hours answering his/her allocated questions.

The deadline to provide the answers to these questions on the group's svn server is Tuesday, September 29th at 13.00. Please provide your answers in ASCII format.

1.1 The Domain Name System

The Domain Name System (DNS) plays a key role in the Internet today as it allows applications to use fully qualified domain names (FQDN) instead of IPv4 or IPv6 addresses. Many tools allow to perform queries through DNS servers. For this exercise, we will use dig which is installed on most Unix systems.

A typical usage of dig is as follows

```
dig @server -t type fqdn
```

where

- *server* is the IP address or the name of a DNS server or resolver
- *type* is the type of DNS record that is requested by the query such as *NS* for a nameserver, *A* for an IPv4 address, *AAAA* for an IPv6 address, *MX* for a mail relay, ...
- *fqdn* is the fully qualified domain name being queried

1. What are the IP addresses of the resolvers that the *dig* implementation you are using relies on [1] ?

1. What is the IP address that corresponds to *inl.info.ucl.ac.be* ? Which type of DNS query does *dig* send to obtain this information ?

1. Which type of DNS request do you need to send to obtain the nameservers that are responsible for a given domain ?

1. What are the nameservers that are responsible for the *be* top-level domain ? Where are they located ? Is it possible to use IPv6 to query them ?

[1] On a Linux machine, the *Description* section of the *dig* manpage tells you where *dig* finds the list of nameservers to query.

1. When run without any parameter, *dig* queries one of the root DNS servers and retrieves the list of the the names of all root DNS servers. For technical reasons, there are only 13 different root DNS servers. This information is also available as a text file from http://www.internic.net/zones/named.root What are the IP addresses of all these servers. Do they all support IPv6 [2] ?

1. Assume now that you are residing in a network where there is no DNS resolver and that you need to start your query from the DNS root.

   - Use *dig* to send a query to one of these root servers to find the IP address of the DNS server(s) (NS record) responsible for the *org* top-level domain
   - Use *dig* to send a query to one of these DNS servers to find the IP address of the DNS server(s) (NS record) responsible for root-servers.org‘
   - Continue until you find the server responsible for *www.root-servers.org*
   - What is the lifetime associated to this IP address ?

1. Perform the same analysis for a popular website such as *www.google.com*. What is the lifetime associated to this IP address ? If you perform the same request several times, do you always receive the same answer ? Can you explain why a lifetime is associated to the DNS replies ?

1. Use *dig* to find the mail relays used by the *uclouvain.be* and *gmail.com* domains. What is the *TTL* of these records (use the *+ttlid* option when using *dig*) ? Can you explain the preferences used by the *MX* records. You can find more information about the MX records in **RFC 974**

1. Use *dig* to query the IPv6 address (DNS record AAAA) of the following hosts

   - *www.sixxs.net*
   - *www.google.com*
   - *ipv6.google.com*

1. When *dig* is run, the header section in its output indicates the *id* the DNS identifier used to send the query. Does your implementation of *dig* generates random identifiers ?

```
dig -t MX gmail.com

; <<>> DiG 9.4.3-P3 <<>> -t MX gmail.com
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25718
```

1. A DNS implementation such as *dig* and more importantly a name resolver such as bind or unbound, always checks that the received DNS reply contains the same identifier as the DNS request that it sent. Why is this so important ?

   - Imagine an attacker who is able to send forged DNS replies to, for example, associate *www.bigbank.com* to his own IP address. How could he attack a DNS implementation that
     - sends DNS requests containing always the same identifier
     - sends DNS requests containing identifiers that are incremented by one after each request
     - sends DNS requests containing random identifiers

1. The DNS protocol can run over UDP and over TCP. Most DNS servers prefer to use UDP because it consumes fewer resources on the server. However, TCP is useful when a large answer is expected or when a large answer must Use *time dig +tcp* to query a root DNS server. Is it faster to receive an answer via TCP or via UDP ?

[2] You may obtain additional information about the root DNS servers from http://www.root-servers.org

1.2 Internet email protocols

Many Internet protocols are ASCII-based protocols where the client sends requests as one line of ASCII text terminated by *CRLF* and the server replies with one of more lines of ASCII text. Using such ASCII messages has several advantages compared to protocols that rely on binary encoded messages

- the messages exchanged by the client and the server can be easily understood by a developer or network engineer by simply reading the messages
- it is often easy to write a small prototype that implements a part of the protocol
- it is possible to test a server manually by using telnet Telnet is a protocol that allows to obtain a terminal on a remote server. For this, telnet opens a TCP connection with the remote server on port 23. However, most *telnet* implementations allow the user to specify an alternate port as *telnet hosts port* When used with a port number as parameter, *telnet* opens a TCP connection to the remote host on the specified port. *telnet* can thus be used to test any server using an ASCII-based protocol on top of TCP. Note that if you need to stop a running *telnet* session, Ctrl-C will not work as it will be sent by *telnet* to the remote host over the TCP connection. On many *telnet* implementations you can type *Ctrl-]* to freeze the TCP connection and return to the telnet interface.

1. Assume that your are sending an email from your @*student.uclouvain.be* inside the university to another student's @*student.uclouvain.be* address. Which protocols are involved in the transmission of this email ?

1. Same question when you are sending an email from your @*student.uclouvain.be* inside the university to another student's @*gmail.com* address

1. Before the advent of webmail and feature rich mailers, email was written and read by using command line tools on servers. Using your account on *sirius.info.ucl.ac.be* use the */bin/mail* command line tool to send an email to yourself *on this host*. This server stores local emails in the */var/mail* directory with one file per user. Check with */bin/more* the content of your mail file and try to understand which lines have been added by the server in the header of your email.

1. Use your preferred email tool to send an email message to yourself containing a single line of text. Most email tools have the ability to show the *source* of the message, use this function to look at the message that you sent and the message that you received. Can you find an explanation for all the lines that have been added to your single line email [3] ?

1. The first version of the SMTP protocol was defined in **RFC 821**. The current draft standard for SMTP is defined in **RFC 5321** Considering only **RFC 821** what are the main commands of the *SMTP* protocol [4] ?

1. When using SMTP, how do you recognise a positive reply from a negative one ?

1. A SMTP server is a daemon process that can fail due to a bug or lack of resources (e.g. memory). Network administrators often install tools [5] that regularly connect to their servers to check that they are operating correctly. A simple solution is to open a TCP connection on port 25 to the SMTP server's host [6] . If the connection is established, this implies that there is a process listening. What is the reply sent by the SMTP server when you type the following command ?

   telnet nostromo.info.ucl.ac.be 25

   Warning : Do *not* try this on a random SMTP server. The exercises proposed in this section should only be run on the SMTP server dedicated for these exercices : *nostromo.info.ucl.ac.be*. If you try them on a production SMTP server, the administrator of this server may become angry.

1. Continue the SMTP session that you started above by sending the greetings command (*HELO* followed by the fully qualified domain name of your host) and termine the session by sending the *QUIT* command.

[3] Since **RFC 821**, SMTP has evolved a lot due notably to the growing usage of email and the need to protect the email system against spammers. It is unlikely that you will be able to explain all the additional lines that you will find in email headers, but we'll discuss them together.

[4] A shorter description of the SMTP protocol may be found on wikipedia at http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol

[5] There are many monitoring tools available. nagios is a very popular open source monitoring system.

[6] Note that using *telnet* to connect to a remote host on port 25 may not work in all networks. Due to the spam problem, many ISP networks do not allow their customers to use port TCP 25 directly and force them to use the ISP's mail relay to forward their email. Thanks to this, if a software sending spam has been installed on the PC of one of the ISP's customers, this software will not be able to send a huge amount of spam. If you connect to *nostromo.info.ucl.ac.be* from the fixed stations in INGI's lab, you should not be blocked.

1. The minimum SMTP session above allows to verify that the SMTP is running. However, this does not always imply that mail can be delivered. For example, large SMTP servers often use a database to store all the email addresses that they serve. To verify the correct operation of such a server, one possibility is to use the *VRFY* command. Open a SMTP session on the lab's SMTP server (*nostromo.info.ucl.ac.be*) and use this command to verify that your account is active.

1. Now that you know the basics of opening and closing an SMTP session, you can now send email manually by using the *MAIL FROM:*, *RCPT TO:* and *DATA* commands. Use these commands to *manually* send an email to *INGI2141@nostromo.info.ucl.ac.be* . Do not forget to include the *From:*, *To:* and *Subject:* lines in your header.

1. By using SMTP, is it possible to send an email that contains exactly the following ASCII art ?

```
.
..
...
```

1. Most email agents allow you to send email in carbon-copy (*cc:*) and also in blind-carbon-copy (*bcc:*) to a recipient. How does a SMTP server supports these two types of recipients ?

1. In the early days, email was read by using tools such as */bin/mail* or more advanced text-based mail readers such as pine or elm . Today, emails are stored on dedicated servers and retrieved by using protocols such as POP or IMAP From the user's viewpoint, can you list the advantages and drawbacks of these two protocols ?

1. The TCP protocol supports 65536 different ports numbers. Many of these port numbers have been reserved for some applications. The official repository of the reserved port numbers is maintained by the Internet Assigned Numbers Authority (IANA) on http://www.iana.org/assignments/port-numbers [7] Using this information, what is the default port number for the POP3 protocol ? Does it run on top of UDP or TCP ?

1. The Post Office Protocol (POP) is a rather simple protocol described in **RFC 1939**. POP operates in three phases. The first phase is the authorization phase where the client provides a username and a password. The second phase is the transaction phase where the client can retrieve emails. The last phase is the update phase where the client finalises the transaction. What are the main POP commands and their parameters ? When a POP server returns an answer, how can you easily determine whether the answer is positive or negative ?

1. On smartphones, users often want to avoid downloading large emails over a slow wireless connection. How could a POP client only download emails that are smaller than 5 KBytes ?

1. Open a POP session with the lab's POP server (*nostromo.info.ucl.ac.be*) by using the username and password that you received. Verify that your username and password are accepted by the server.

1. The lab's POP server contains a script that runs every minute and sends two email messages to your account if your email folder is empty. Use POP to retrieve these two emails and provide the secret message to your teaching assistant.

1.3 The HyperText Transfer Protocol

1. What are the main methods supported by the first version of the HyperText Transfer Protocol (HTTP) defined in **RFC 1945** [8] ? What are the main types of replies sent by a http server [9] ?

1. System administrators who are responsible for web servers often want to monitor these servers and check that they are running correctly. As a HTTP server uses TCP on port 80, the simplest solution is to open a TCP connection on port 80 and check that the TCP connection is accepted by the remote host. However, as HTTP is an ASCII-based protocol, it is also very easy to write a small script that downloads a web page on the server and compares its content with the expected one. Use *telnet* to verify that a web server is running on host *rembrandt.info.ucl.ac.be* [10]

[7] On Unix hosts, a subset of the port assignments is often placed in */etc/services*

[8] See section 5 of **RFC 1945**

[9] See section 6.1 of **RFC 1945**

[10] The minimum command sent to a HTTP server is *GET / HTTP/1.0* followed by CRLF and a blank line

1. Instead of using *telnet* on port 80, it is also possible to use a command-line tool such as curl Use curl with the *–trace-ascii tracefile* option to store in *tracefile* all the information exchanged by curl when accessing the server.

   - what is the version of HTTP used by curl ?
   - can you explain the different headers placed by curl in the request ?
   - can you explain the different headers found in the response ?

1. HTTP 1.1, specified in **RFC 2616** forces the client to use the *Host:* in all its requests. HTTP 1.0 does not define the *Host:* header, by most implementations support it. By using *telnet* and *curl* retrieve the first page of the http://totem.info.ucl.ac.be webserver by sending http requests with and without the *Host:* header. Explain the difference between the two [11] .

1. By using dig and curl , determine on which physical host the http://www.info.ucl.ac.be, http://inl.info.ucl.ac.be and http://totem.info.ucl.ac.be are hosted

1. Use curl with the *–trace-ascii filename* to retrieve http://www.google.com . Explain what a browser such as firefox would do when retrieving this URL.

1. The headers sent in a HTTP request allow the client to provide additional information to the server. One of these headers is the Language header that allows to indicate the preferred language of the client [12]. For example, *curl -HAccept-Language:en http://www.google.be' will send to 'http://www.google.be* a HTTP request indicating English (en) as the preferred language. Does google provide a different page in French (fr) and Walloon (wa) ? Same question for http://www.uclouvain.be (given the size of the homepage, use diff to compare the different pages retrieved from www.uclouvain.be)

1. Compare the size of the http://www.yahoo.com and http://www.google.com web pages by downloading them with curl

1. What is a http cookie ? List some advantages and drawbacks of using cookies on web servers.

1. You are now responsible for the *http://www.belgium.be*. The government has built two datacenters containing 1000 servers each in Antwerp and Namur. This website contains static information and your objective is to balance the load between the different servers and ensures that the service remains up even if one of the datacenters is disconnected from the Internet due to flooding or other natural disasters. What are the techniques that you can use to achieve this goal ?

[11] Use dig to find the IP address used by *totem.info.ucl.ac.be*

[12] The list of available language tags can be found at http://www.loc.gov/standards/iso639-2/php/code_list.php Additional information about the support of multiple languages in Internet protocols may be found in **RFC 5646**

CHAPTER

TWO

THE ALTERNATING BIT PROTOCOL

The objective of this set of exercises is to better understand the basic mechanisms of a transport layer protocol by implementing them in an emulated environment. The deadline for this exercise is *Tuesday October 6th, 2009 at 13.00*.

2.1 scapy

For this exercise, and many of others, we will extend scapy. scapy is a packet injection tool developed by Philippe Biondi and many others. scapy is open-source and written in python. It was designed as a tool to easily create and process specially crafted TCP/IP packets. We will use scapy as a prototyping tool that allows to easily implement simple protocols and simple mechanisms.

scapy follows the layering principles used by networking protocols. It allows to easily create and send packets through the network. When using scapy, it is very easy to create any packet and send them. scapy creates packets and bypasses the networking stack. From a security viewpoint, such operations are privileged and can only be performed by *root* on Linux. scapy is run on the command line as an interactive tool

```
debian:~# sudo scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: No route found for IPv6 destination :: (no default route?)
DEBUG: Loading layer p3
Welcome to Scapy (2.0.1-dev)
>>>
```

You can ignore the INFO and WARNING lines that, depending on your installation may indicate that some optional packages have not been installed.

scapy is usually used to create special packets and send them. For example, the following line allows to create a reference p to a UDP segment containing “ABCD” as its payload

```
p=UDP()/"ABCD"
```

Note the utilization of the / character to stack the “ABCD” string on top of the UDP segment that was created with *UDP()* scapy contains classes that represent many of the protocols of the TCP/IP protocol suite, including IP, TCP, UDP, ... UDP is the easiest to understand. As you know, a UDP header contains a source port, a destination port, a checksum and a length field. This is exactly the same in scapy. scapy allows you to see all the fields of a given header by using *ls*

```
 >>> ls(UDP)
sport      : ShortEnumField       = (53)
dport      : ShortEnumField       = (53)
len        : ShortField           = (None)
chksum     : XShortField          = (None)
```

We find back all the fields of the UDP header. When creating a packet, most of these fields have a default value. In the case of UDP, the source and destination ports are set to 53, the length and the checksum are automatically computed based on the payload. You can change the source and destination ports of the UDP segment as follows

```
>>> p=UDP(sport=1234,dport=5678)
```

The *Packet* class contains several methods to inspect packets :

- pkt.summary() for a one-line summary
- pkt.show() for a developed view of the packet
- pkt.show2() same as show but on the assembled packet (checksum is calculated, for instance)

For example

```
>>> p=UDP(sport=1234,dport=5678)
>>> p.summary()
'UDP 1234 > 5678'
>>> p.show()
###[ UDP ]###
 sport= 1234
 dport= 5678
 len= None
 chksum= None
>>> p=IP()/UDP(sport=1234,dport=5678)
>>> p.show2()
###[ IP ]###
 version= 4L
 ihl= 5L
 tos= 0x0
 len= 28
 id= 1
 flags=
 frag= 0L
 ttl= 64
 proto= udp
 chksum= 0x7cce
 src= 127.0.0.1
 dst= 127.0.0.1
 \options\
###[ UDP ]###
    sport= 1234
    dport= 5678
    len= 8
    chksum= 0xe6db
>>>
```

To build a new protocol with scapy, you first need to define the format of the header. Taking UDP as an example and looking at file *scapy/layers/inet.py*, we see that UDP is defined as follows

```
class UDP(Packet):
   name = "UDP"
   fields_desc = [ ShortEnumField("sport", 53, UDP_SERVICES),
                   ShortEnumField("dport", 53, UDP_SERVICES),
                   ShortField("len", None),
                   XShortField("chksum", None), ]
```

The UDP header is composed of 4 short (i.e. 16 bits long) fields : the source port [1], the destination port, the length field and the checksum. To design your own segment header, you need to define a similar class. scapy knows

[1] The specification of the UDP source port field in scapy contains three parameters. The first one is the name of the field, which can be used to set the value of this field when creating a packet. The second value is the default. The source port is set to *53* when not specified at packet creation time. The last parameter for an enumerated field is the list of names for all UDP services. This list is automatically created by scapy from the */etc/services* file on a Linux host so that you can specify *dns* as source port instead of 53.

many basic types of fields that can be included in a header, such as :

- *BitField(name, default, n)* where *name* is the *name* of the field, *default* the default value when not specified and *n* the length of the field in bits
- *FlagsField(name, default, n, [”A”,...])* where *name* is the *name* of the field, *default* the default value when not specified, *n* the number of bits of the field and the array *[“A”, ...]* contains the names of the flags. Compared to *BitField*, the advantage of the *FlagsField* is that each bit is a flag that you can set or reset.
- *BitEnumField(name, default, n, {0:”A”, ...})* where *name* is the

  name of the *n* bits field, *default* the default value when not specified and the *{0:”A”, ...}* provides the names associated to the different bit values. Compared to the *BitField*, the advantage of the *BitEnumField* is that you can specify the values of the field as an enumeration.
- *ShortField(name, default)* where *name* is the *name* of the 16 bits field, *default* the default value when not specified
- *ByteField(name, default)* where *name* is the *name* of the 8 bits field, *default* the default value when not specified

To create a header for MyProtocol, you first need to define the class that extends the Packet class. This can be done as follows

```
class MyProtocol(Packet):
   name = "MyProtocol"
   fields_desc = [ BitEnumField("first", 0, 4, {0:"A", 1:"B"} ),
                   BitField("second",0,4),
                   ShortField("len", None),
                   ByteField("padding",0),
                 ]
```

The header of this protocol is composed of 32 bits and contain three field. A first group of 4 bits, then a second one, a 16 bits *len* field and one byte of padding to ensure that the header takes 32 bits. The description of this header will be placed in file *scapy/layers/myprotocol.py* in the scapy source distribution.

Besides creating the header, you also need to explain to scapy how to build a packet containing this header. This is done by defining in the *MyProtocol* class the *post_build* method. The simplest implementation of this method is to concatenate the header with the payload as follows

```
def post_build(self, p, pay):
    p += pay
    return p
```

If you want to use a length field as in the example above, you need to compute the length of the packet and include it in the header. In the example above, the first byte of the header containing the flags, the second and third the length and the fourth byte contains the padding. The length field can be computed automatically and placed in the appropriate location in the header as follows (the *struct.pack(“!H”,l)* call converts the *l* value in unsigned network byte order format which is the standard format used to send 16 bits numbers over an IP network - you can reuse this method as it is if you have a length field in your protocol)

```
def post_build(self, p, pay):
    p += pay
    l = self.len
    if l is None:
            l = len(p)
            p = p[:1]+struct.pack("!H",l)+p[4:]
    return p
```

If you want to add a checksum to your protocol, this is also in the *post_build* that you would add the checksum computation. However, this requires some advanced knowledge of scapy that goes beyond this exercise.

Once the protocol header has been created, you need to tell scapy how it can intercept the packets using this protocol. As we are building a transport layer protocol, it will run on top of a networking protocol. For this exercise, you will run your protocol above **IP_** Although we haven't yet described how IP works, you need to know that IP uses a field called *protocol* to indicate the transport protocol that was used to create the payload of an IP packet. IP protocol numbers are allocated by IANA and the registry can be found at http://www.iana.org/assignments/protocol-numbers/ For this exercise, we can reuse protocol *210* which is unallocated. To tell scapy to bind *MyProtocol* to IP protocol 210, you need to add in *scapy/layers/myprotocol.py* the following line

```
bind_layers(IP,MyProtocol, proto=210)
```

Once your protocol has been defined and bound, you can use scapy in interactive mode to send and receive packets containing your protocol. To send a packet, you can simply use the *send* method provided by scapy. For example

```
>>> send(IP(src="1.2.3.4",dst="5.6.7.8")/MyProtocol(first="A")/"ABC")
.
Sent 1 packets.
```

Defining the header of the protocol is simple in scapy with simple protocols. Besides specifying the format of the header, you also need to define the behavior of a sender and a receiver implementing your protocol. scapy defines a set of base classes that can be used to implement simple Finite State Machines. You will use these FSMs to implement your simple transport protocol. A FSM is implemented as a python class that extends the Automaton class provided by scapy A FSM contains :

- implementation of the *parse_args* method that is used to parse the arguments provided when creating the FSM
- implementation of the *master_filter* method that specifies how scapy will capture on the network the packets that must be processed by the FSM.
- states, conditions and timeouts that implement the FSM

The *parse_args* allows you to specify parameters for your FSM when creating it. For a sender, you can use something like

```
class MyProtocolSender(Automaton):
        def parse_args(self, payloads, receiver,**kargs):
                Automaton.parse_args(self, **kargs)
                self.receiver = receiver
                self.q = Queue.Queue()
                for item in payloads:
                        self.q.put(item)

        def master_filter(self, pkt):
                  return (IP in pkt and pkt[IP].src == self.receiver
                  and MyProtocol in pkt)
```

This method specifies a set of messages (e.g. a list of Strings) to be sent and the address of the receiver as a String. It places all the messages in a Queue to be processed later and remembers the address of the receiver. The *master_filter* method instructs scapy to capture all the IP packets that it receives from *self.receiver* and contain a MyProtocol segment.

For a receiver, you could use something like

```
class Receiver(Automaton):
        def parse_args(self, sender, **kargs):
                Automaton.parse_args(self, **kargs)
                self.sender = sender

        def master_filter(self, pkt):
                return (IP in pkt and pkt[IP].src == self.sender and
                MyProtocol in pkt)
```

In this case, we only specify the address of the sender as a String. The *master_filter* method instructs scapy to capture all the IP packets that it receives from *self.sender* and contain a MyProtocol segment.

In scapy, an automaton has different states. A *start* state and possibly some *end* and *error* states. There are transitions from one state to another. These transitions can depend on a specific condition, the reception of a specific packet or the expiration of a timeout. When a transition is taken, scapy will run the actions that have been bound to this transition. It is possible to pass parameters from states to transitions and the opposite. From a programmer's point of view, states, transitions and actions are methods from an Automaton subclass. They are decorated to provide meta-information needed in order for the automaton to work.

A first and simple FSM is to capture all the packets using MyProtocol and display them. This can be achieved by defining one state which is the initial one and one receive condition that is associated to this state and is triggered every time the FSM is in this state and a packet is received by the *master_filter* specified above

```
@ATMT.state(initial=1)  ## initial state
def WAIT_PDU (self):
        print "State: WAIT_PDU"
        pass

@ATMT.receive_condition(WAIT_PDU)
def wait_pdu (self, pkt):
        print "received ",pkt.show()
        raise self.WAIT_PDU()
```

This FSM is always in the *WAIT_PDU* state. The *wait_pdu()* method is attached to the *WAIT_PDU* state as a receive condition by using the *@ATMT.receive_condition(WAIT_PDU)* decorator. The *wait_pdu()* method simply prints all packets using the *MyProtocol* protocol that it receives by using the *show()* method of the Packet class. The command *raise* forces the transition to the specified state of the FSM. If you want to extract the payload of the received packet and print it, you can use

```
payload = pkt.getlayer(MyProtocol).payload.load
print "data received [[",payload,"]]"
```

Transport protocols often use timers and the scapy automatons allow you to easily define timers to for example retransmit an unacknowledged segment. This is done by using the *@ATMT.timeout()* and *@ATMT.action()* decorators.

For example, let us consider a simple sender that uses a timer to protect the segments sent. In the *MyProtocolSender* class shown above, we could add the following states and transitions

```
@ATMT.state(initial=1)
def WAIT_SDU (self):
        try:
                self.payload = self.q.get()
                print "data to be transmitted [[",self.payload,"]]"
                self.buffer=IP(dst=self.receiver)/MyProtocol(first="A")/self.payload
                self.send(self.buffer)
               raise self.WAIT_CTRL()
        except Queue.Empty:
                sys.exit(0)

@ATMT.state()
def WAIT_CTRL(self):
        print "State: WAIT_CTRL"

# other states/transitions not shown

@ATMT.timeout(WAIT_CTRL, 2)
def timeout_waiting_for_ctrl(self):
        raise self.WAIT_CTRL()

@ATMT.action(timeout_waiting_for_ctrl)
```

```
def retransmit(self):
        self.send(self.buffer)
```

The initial state simply processes the Queue containing all the SDUs as defined in *MyProtocolSender* above. It sends one SDU at a time then goes to the *WAIT_CTRL* state where the acknowledgments will be processed (this part is not shown here). Note that the last segment sent is stored in *self.buffer*. The timeout *timeout_waiting_for_ctrl* is associated to the *WAIT_CTRL* state and has a duration of 2 seconds [2]. When the timeout expires, the action associated to it is executed. In this case, the FSM will retransmit the packet and the FSM will transition to the *WAIT_CTRL* state as indicated in the timer definition.

Once a FSM has been written, you can use it from scapy by creating an instance of your FSM and start its *run()* method

```
Welcome to Scapy (2.0.1-dev)
>>> s=Sender(("A","B","C"),"192.168.56.102")
>>> s.run()
```

You can also use the *next()* method to execute the transitions one by one.

You can find additional information about scapy in its official documentation : http://www.secdev.org/projects/scapy/doc/ However, note that this document was written mainly for security specialists who already know the bit-level details of many protocols. Most of the students following this course are unlikely to have already this experience... The only part of the scapy documentation that could be useful for this exercise is the description of the automatons : http://www.secdev.org/projects/scapy/doc/advanced_usage.html#automata

2.2 Implementing the Alternating Bit Protocol in scapy

The Alternating Bit Protocol (ABP) is the simplest protocol that provides a reliable data transfer in the transport layer. Your objective for this week is to implement, by teams of two students, either the Sender or the receiver part of the ABP. For this, you should probably work as follows :

- define in the entire group a common format for the ABP header and specify the corresponding ABP class that extends the Packet class
- divide the group in teams of two students (a team of 3 is allowed if the number of students in the group is odd)
- schedule an interoperability meeting with another team during which you can test your Sender with their Receiver (or the opposite)
- write your implementation in python by changing the *abp.py file in 'scapy/layers* in the scapy distribution provided
- perform the interoperability test and verify that your implementation works correctly

To ease the implementation, we will ignore the utilization of a checksum in the Alternating Bit Protocol. In the emulated environment that you will use, losses are unlikely to occur, but you can of course easily create them inside your sender or receiver FSM. For this, you can reuse the standard python classes, such as :

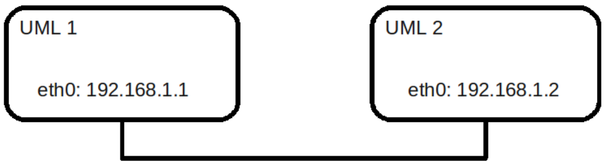
- in the *random* class, you can use the *random.random()* method to generate a random number and use it to probabilistically drop a packet before sending it in the sender or receiving it in the receiver. The drop probability could be a parameter that you specify when creating your FSM. This would allow you to easily simulate the effect of transmission errors on your protocol.
- in the *time* class, you can use the *time.sleep()* class to create delays when processing messages

[2] For this exercise, consider a fixed timeout value. We will discuss later on how to correctly choose a timeout value.

2.3 Running scapy in the lab

To allow you to write the required scapy extensions without losing too much time with practical problems in installing and configuring software, we have prepared an emulated environment that runs in the computing labs. This environment is composed of two User Mode Linux (UML) images. A UML image is an emulated machine containing a Linux kernel, a complete filesystem and the necessary utilities. This UML image runs as a process on a Linux workstation such as those in the computing lab.

Warning: Although scapy can work on MacOS and Windows, we recommend that you use the special version of User Mode Linux that we have prepared instead of spending time to install scapy on your own machines. Please note also that the fact that we use scapy for the basic networking course cannot be considered as an authorization to send specially crafted packets in the campus network or on the Internet. Such specially crafted packets may be considered by system administrators as a security attack to which they may react.



The filesystems and the kernel of the virtual machines are stored in the directory */etinfo/applications/INGI2141/uml*. To create the virtual setup illustrated on the figure execute the following procedure:

1. Copy the boot script of each virtual machines in your working directory

```
cp /etinfo/applications/INGI2141/uml/start_uml1 ~/INGI2141/UML
cp /etinfo/applications/INGI2141/uml/start_uml2 ~/INGI2141/UML
```

2. Open 2 terminals, 1 for each virtual machine
3. In the first terminal, start the virtual machine UML1 by executing the script *start_uml1*. In the second terminal, start the virtual machine UML2 by executing the script *start_uml2*.
4. Once the machines will have finished their boot process, you can login on each machine as root. Each UML machine has an Ethernet interface named *eth0* that is directly connected to the other UML machine. To check their virtual IP connectivity by pinging the IP of the interfaces on the other end of the virtual link by executing the following commands

```
ping 192.168.1.1
ping 192.168.1.2
```

You can share files between the virtual machines and the machine where you are running the User Mode Linux processes in the directory */mnt/host* of the virtual machines.

You will also find in the */root* directory of the emulated machines the *scapy-2.01* directory that contains a modified scapy source code. This version of scapy has been modified by adding a new protocol called ABP (Alternating Bit Protocol) and configuring scapy with the necessary hooks to understand this protocol. ABP is located in file *scapy/layers/abp.py* of the scapy archive. You should modify this file (and only this file) to create :

- your own header by extending the scapy Packet class in file *scapy/layers/abp.py*
- your sender [3] FSM
- your receiver [4] FSM

[3] If your team of two students implements a sender FSM, you can test it by using on the other UML machine a simple FSM that has a single state and prints all received segments (see the example with the *WAIT_PDU* state)

[4] If your team of two students implements a receiver FSM, you can test it by manually sending segments using *send()* from scapy

Once you have modified *abp.py*, you should recompile scapy and reinstall it on the two emulated Linux machines. This can be done by using

```
python setup.py  clean
python setup.py  install
```

These commands should be run in the root of the scapy distribution. The second command will reinstall your modified version of scapy. Check its output to verify that there are no compilation errors. As python is an interpreted language, you may also detect errors at runtime.

Note: As scapy does not completely support the loopback interface, you should only use it on the Ethernet interfaces of the UML machines. Do not waste your time trying to run scapy on a loopback interface.

CHAPTER

THREE

GO-BACK-N AND SELECTIVE REPEAT

Go-back-n and selective repeat are the basic mechanisms used in reliable window-based transport-layer protocols. During this exercise, you will implement a go-back-n (GBN) sender using scapy. In each group, you will write one sender per team of two students.

The deadline for this exercise is Tuesday October 13th, 2009 at 13.00.

3.1 Partial implementation of a simplified GBN receiver in scapy

Our go-back-n protocol relies on a very simple header that can be implemented by the following scapy class

```
NBITS=4      # number of bits used to encode the sequence number field
assert NBITS<=8 # precondition

# the header of the simplified Go-Back-N protocol

class GBN(Packet):
   name = "Go-Back-N (INGI2141) "
   fields_desc=[ BitEnumField("type",0,1, {0:"data", 1:"ack"}), # type of segment : data or ack
                 BitField("padding" , 0 , 7), # padding
                 ShortField("len", None), # segment length in bytes
                 ByteField("num",0),  # sequence number in data, ack number in ack segments, incr
                 ByteField("win",0) # receiving window (ignored in data) in number of segments
                 # real protocol would contain a checksum
                 ,]

   # automatic computation of the length
   def post_build(self, p, pay):
       p += pay
       l = self.len
       if l is None:
               l = len(p)
               p = p[:1]+struct.pack("!H",l)+p[3:]
       return p

bind_layers(IP,GBN,proto=222)
```

Our header contains the following information :

- the first bit indicates whether the segment contains data or only an acknowledgment
- the second and third bytes contain the segment length. This length is automatically computed by the *post_build* method

- the fourth byte contains an 8 bits integer. When used in a *data* segment, this integer is the sequence number of the segment. In our simple go-back-n protocol, the sequence number is incremented by one every time a segment is sent. When used in an *ack* segment, the fourth byte contains the sequence number of the next segment that is expected by the receiver. The constant NBITS allows you to change the number of bits used to encode the sequence number. Of course, as the sequence number is encoded as a one byte field, the NBITS cannot be larger than 8. Note that the size of the *num* field is not adjusted if you change the NBITS constant.
- the last byte contains the receiver's window.

The class *GBNReceiver* below is a simple FSM of a receiver for our simple go-back-n protocol

```
## This class implements the receiver side of the GBN protocol by using scapy Automaton facilitie

class GBNReceiver(Automaton):
       def parse_args(self, window, pdata, pack, debug, sender, **kargs):
               Automaton.parse_args(self, **kargs)
               self.win=window       # window size advertised by receiver in segments
               assert self.win <= pow(2,NBITS)
               self.pdata=pdata      # loss probability for data segments 0<= proba < 1
               assert 0<=pdata and pdata<1
               self.pack=pack        # loss probability for acks  0<= proba < 1
               assert 0<=pack and pack<1
               self.dbg=debug        # True if debug output is requested, false otherwise
               self.sender = sender  # ip address of sender
               self.next=0           # next expected sequence number

       def master_filter(self, pkt):
               return (IP in pkt and pkt[IP].src == self.sender and GBN in pkt)

       @ATMT.state(initial=1)
       def WAIT_SEGMENT (self):
           if self.dbg:
               print "Waiting for segment ",self.next
           pass

       @ATMT.receive_condition(WAIT_SEGMENT)
       def wait_segment (self, pkt):
           if random.random() < self.pdata :
               # received segment was lost/corrupted in the network
               if self.dbg:
                   print " Lost : [type=",pkt.getlayer(GBN).type,",
       num=",pkt.getlayer(GBN).num," ,win=",pkt.getlayer(GBN).win,"]"

               raise self.WAIT_SEGMENT()

           else:
               # segment was received correctly
               if self.dbg:
                   print " Received : [type=",pkt.getlayer(GBN).type,", num=",pkt.getlayer(GBN).nu

               # check if segment is a data segment
               ptype = pkt.getlayer(GBN).type
               if ptype==0:
                   if pkt.getlayer(GBN).num==self.next:
                       # this is the segment with the expected sequence number
                       print "data received :",pkt.getlayer(GBN).payload
                       self.next=int((self.next+1) %  pow(2,NBITS))
                   else:
                       # this was not the expected segment
                       if self.dbg:
                           print "Out of sequence segment [",pkt.getlayer(GBN).num,"] received "
               else:
                   # we received an ack while we are supposed to receive only data segments
                   print "ERROR: Received ack segment : ",pkt.show()
                   sys.exit(-1)

               # send ack back to sender
               if random.random() < self.pack :
                   # the ack will be lost, discard it
                   if self.dbg:
                       print " Lost ack:",self.next
               else:
                   # the ack will be received correctly
                   if self.dbg:
                       print " Sending ack :",self.next
                   send(IP(dst=self.sender)/GBN(type="ack",num=self.next,win=self.win))
               # transition to WAIT_SEGMENT to receive next segment
               raise self.WAIT_SEGMENT()
```

The *GBNReceiver* constructor takes the following arguments :

- the receiving window (in segments)
- the loss probabilities for the *data* and *ack* segments. In our emulated network, the *GBNReceiver* models segment losses by probabilistically discarding the received *data* segments or probabilistically discarding the *ack* segments before sending them
- a debug flag that causes the FSM to print debugging information when set to *True*
- the IP address of the sender

The *GBNReceiver* is implemented as a single state FSM. This FSM receives data segments in the *WAIT_SEGMENT* state. These segments are handled by the *wait_segment* receive condition that operates as follows.

- the receive condition first checks whether the received segment should be probabilistically discarded.
- the segment is verified to check that it is a *data* segment
- the FSM verifies that the received segment has the expected sequence number. This is done thanks to the *self.next* state variable that is maintained by the FSM. If the expected sequence number was received *self.next* is incremented by $1 \ mod \ 2^{NBITS}$
- the acknowledgment is sent. In our simple protocol, the *num* field in the *ack* segment always contains the sequence number of the next expected segment (i.e. the sequence number of the last segment received in sequence $+1 \ mod \ 2^{NBITS}$)

3.2 Preliminary questions

The following questions should help you to prepare your implementation of the go-back-n sender FSM.

1. As the sequence number are encoded by using NBITS, there are only 2^{NBITS} different sequence numbers. How do you compute the sequence number that follows sequence number *x* ?
2. A go-back-n sender must contain a sending buffer where it stores the segments that have been sent without having already been acknowledged. Which kind of python data structure [1] will you use to build this buffer ? (do not develop your own optimized data structure, reuse a data structure that python already supports - the chosen data structure should allow you to easily associate a sequence number with a corresponding segment/payload, count the number of segments/payloads stored and check whether a given sequence number is already in the sending buffer or not)
3. The maximum capacity of your sending buffer is the window size of your FSM. How do you check whether your sending buffer is full ?

[1] The basic data structures of the python library are described in http://docs.python.org/tutorial/datastructures.html and http://docs.python.org/library/stdtypes.html

4. When there are no losses, you will need to remove from your sending buffer a *data* segment every time an *ack* segment is received. How do you remove from the chosen data structure *the* segment that is acknowledged by the received *ack* ?
5. *data* segments are protected against losses by using the retransmission timer and the go-back-n mechanism. *ack* segments are protected against losses due to the fact that when an *ack* segment with *num* set to *x* is received, it acknowledges all *data* segments that were sent before segment *x*. Thus, when you receive an *ack*, you may need to remove *0*, *1* or more acknowledged *data* segments from your sending buffer. How do you decide which data segments must be removed from the sending buffer when you receive an acknowledgment ?
6. How can you detect that all the data segments that you have sent have been acknowledged ?
7. When the timer expires, what are the segments that should be retransmitted ? How do you retransmit them ?

3.3 Implementation of the sender FSM

To implement the FSM for your go-back-n sender, you can start from the skeleton FSM below that is composed of two states, one timeout and one receive condition

```
## This class implements the sender side of the go-back-n Protocol by
  using scapy Automaton facilities

TIMEOUT=1  # default timeout in seconds

class GBNSender(Automaton):
       def parse_args(self, sdus, win, receiver,**kargs):
               Automaton.parse_args(self, **kargs)
               self.win=win # the maximum window size of the sender
               assert self.win<pow(2,NBITS)
               self.receiver = receiver # the IP address of the receiver
               self.q = Queue.Queue()
               for item in sdus:
                       self.q.put(item)

               ## Additional state variables to be maintained

       # filter for GBN segments
       def master_filter(self, pkt):
               return (IP in pkt and pkt[IP].src == self.receiver and GBN in pkt)


       @ATMT.state(initial=1)
       def BEGIN (self):
           raise self.WAIT_ACK()


       @ATMT.state()
       def WAIT_ACK(self):
             # to be completed

       @ATMT.receive_condition(WAIT_ACK)
       def wait_for_ack(self,pkt):
           # to be completed, wait for acks

       @ATMT.timeout(WAIT_ACK, TIMEOUT)
       def timeout_wait_ack(self):
           # to be completed
           # retransmit unacked segments
```

The constructor of the *GBNSender* takes a list of SDUs as arguments and places them in a Queue . This can be used to model the *data.request()* primitives from the sending user. The *payloads* list should be a list of Strings. You can easily create a long list of Strings by using

```
l=[]
for f in range(1000) : l.append(str(f))
```

To extract the next SDU from the Queue, you can use e.g.

```
try:
     # extract one additional SDU
     payload = self.q.get()
     ...
except Queue.Empty:
     # All SDUs have been sent
     # stop sender only once all data has been acknowledged
```

To test your *GBNSender*, you can use the *GBNReceiver* shown above. The *GBNReceiver* expects to receive a first *data* segment with sequence number *0*. Your *GBNSender* must thus start to number the segments that it sends at *0*. You should first try to test it with loss probabilities set to 0 for both *data* and *ack* segments. Then, try to see how your *GBNSender* works when *data* segments are lost and after that add losses for the *ack* segments as well. Do not forget to send a list of SDUs that contains more than 2^{NBITS} Strings.

To implement the go-back-n receiver, you can reuse the UML machines that you used for the Alternating Bit Protocol. A new protocol can be added to the scapy distribution as follows :

- create the file *scapy/layers/gbn.py* that will contain your protocol. Reuse the python imports that were defined for the Alternating Bit Protocol and copy the code of the go-back-receiver shown above
- change the *scapy/config.py* and add “gbn” at the end of the *load_layers* line

  ```
  load_layers = ["l2", "inet", "dhcp", "dns", "dot11", "gprs", "hsrp", "inet6", "ir", "isakmp",
  ```

- recompile scapy (*python setup clean* and *python setup install*) and you are ready to implement your go-back-n sender.

3.4 Deliverables

For this week, you need to provide in your group’s subversion repository :

- one implementation in python per team of two students (if the number of students in the group is odd, one team can be composed of three students). The code should be readable, of course
- a short description in ASCII of the tests that you performed with your implementation and its limitations

CHAPTER

FOUR

THE TRANSMISSION CONTROL PROTOCOL

The Transmission Control Protocol plays a key role in the TCP/IP protocol suite by providing a reliable byte stream service on top of the unreliable connectionless service provided by IP. During this exercise, you will learn how to establish correctly a TCP connection by playing either the role of a client or the role of a server and also to reliably exchange one data segment over this connection.

The deadline for this exercise is Tuesday October 20th, 13.00.

4.1 TCP support in scapy

scapy includes all the python classes that implement the TCP segment format. These classes are part of the standard scapy distribution and can be found in file *scapy/layers/inet.py*. A TCP segment in scapy contains the following fields (with the default value of the field between ()):

```
>>> ls(TCP)
sport      : ShortEnumField      = (20)
dport      : ShortEnumField      = (80)
seq        : IntField            = (0)
ack        : IntField            = (0)
dataofs    : BitField            = (None)
reserved   : BitField            = (0)
flags      : FlagsField          = (2)
window     : ShortField          = (8192)
chksum     : XShortField         = (None)
urgptr     : ShortField          = (0)
options    : TCPOptionsField     = ({})
>>>
```

The TCP flags are defined in scapy as :

- *S* for *SYN*
- *F* for *FIN*
- *R* for *RST*
- *P* for *PSH*
- *A* for *ACK*
- *U* for *URG*

scapy allows you to easily create a TCP segment. For example, to create a *SYN* segment with sequence number set to *1234*, a window size of 4096 bytes, an acknowledgement field set to *0*, a source port set to *9999* and *80* as destination port, you would write

```
p=IP(dst="1.2.3.4",src="5.6.7.8")/TCP(sport=9999,dport=80,flags="S",window=4096,seq=1234,ack=0)
```

The other fields of the TCP header are set to a default value (*urgptr*) or automatically computed (*chksum*). In this segment, since the *ACK* flag is not set, the receiver will not look at the content of the *ack* field. A *SYN+ACK* segment sent in reply to the previous *SYN* segment could be

```
p=IP(dst="5.6.7.8",src="1.2.3.4")/TCP(dport=9999,sport=80,flags="SA",
                                      window=4096,seq=56789,ack=1235)
```

Note that in the reply, the source and destination addresses and the source and destination ports have been swapped. When establishing a TCP connection, the utilisation of the *SYN* flag consumes one sequence number. This explains why the *ack* field of the reply segment is set to *1235* while the sequence number of the *SYN* segment was *1234*.

scapy allows you to easily set a flag in the header of a TCP segment that you create. However, scapy is not as user friendly to check the value of a flag in a received segment. For this, you can use the following function

```
def is_set(flag, pkt):
   '''
   Verifies whether flag was set in segment pkt

   :param pkt: segment to compare flag values
   :type pkt:  scapy.TCP
   :param flag: character corresponding to the flag
   :type flag: char
   :return: True if flag was set in pkt, False otherwise
   :rtype: boolean
   '''
   flag_vals = {"F":0x1, "S":0x2, "R":0x4, "P":0x8, "A":0x10, "U":0x20, "E":0x40, "C":0x80 }
   if flag_vals.has_key(flag) :
       return pkt[TCP].flags & flag_vals[flag]
   else:
       return False
```

Note that the *pkt[TCP]* is used to force scapy to interpret *pkt* as a TCP segment. *pkt[TCP].flags* returns a byte containing the TCP flags of a received segment, while *pkt.flags* returns the *IP* flags of the received segments.

Conversely, you may also want to convert the flags of a received segment in a string. This can be done with the following function

```
def flags2string(flags):
   '''
   Converts the [TCP].flags field of a received segment in a string

   :param flags: the received flags (one byte)
   :return: a string containing the letters corresponding to the set flags
   :rtype: string
   '''
   flag_vals = {"F":0x1, "S":0x2, "R":0x4, "P":0x8, "A":0x10, "U":0x20, "E":0x40, "C":0x80 }
   flagstring=''
   for f in flag_vals.keys():
       if flags & flag_vals[f] :
           flagstring+=f

   return flagstring
```

scapy allows you to easily access the *seq*, *ack* or *window* fields of a received segment. To print the main header fields of a received TCP segment, you can use

```
def print_segment(pkt):
   '''
   prints a TCP segment on stdout
```

```
    :param pkt: segment to compare flag values
    :type pkt:  scapy.TCP
    '''
    if not (TCP in pkt):
       print "Not a TCP segment"
    else:
       print "sport=",pkt[TCP].sport," dport=",pkt[TCP].dport,
             " seq=",pkt[TCP].seq," ack=",pkt[TCP].ack,
             " len=",len(pkt[TCP].payload),
             " flags=",flags2string(pkt[TCP].flags),
             " win=",pkt[TCP].window
```

Note that *len(pkt[TCP].payload)* allows you to easily extract the length of the payload of a received TCP segment.

When implementing a Finite State Machine in scapy, it can sometimes be useful to add some debugging information about the segments that are received. For this, it is interesting to note that scapy allows you to define multiple *receive_conditions* for a given state. For example, you can write the following code

```
@ATMT.state()
def ESTABLISHED(self):
    pass

@ATMT.receive_condition(ESTABLISHED,prio=0)
def printpkt(self,pkt):
    print_segment(pkt)

@ATMT.receive_condition(ESTABLISHED,prio=1)
def data_received(self,pkt):
# processing of the received segment
```

You can also specify a *priority* to the *receive_condition*. The default value of the priority is *0* and scapy will first run the conditions having the lowest priority value. In the example above, scapy will first evaluate the *printpkt* condition and then the *data_received* condition.

4.2 Implementing TCP in scapy

A complete implementation of TCP in scapy is, of course, outside the scope of this exercise. However, even for a simplified implementation such as this one, it is useful to consider some of the problems that must be solved in a real TCP implementation.

A TCP implementation maintains a Transmission Control Block (TCB) for each TCP connection. This TCB is a data structure that contains the complete "*state*" of each TCP connection. The TCB is described in **RFC 793**. This TCB contains first the identification of the TCP connection. As the IP address of the local host is already known by scapy, your implementation will need to store :

- *self.remoteip* : the IP address of the remote host
- *self.remoteport* : the TCP port used for this connection on the remote host
- *self.localport* : the TCP port used for this connection on the local host. Note that when a client opens a TCP connection, the local port will be chosen in the ephemeral port range (49152 <= localport <= 65535).

Your implementation also needs to store information about the segments that it has sent and the segments that it has received. **RFC 793** defines the following variables :

- *self.sndnxt* : the sequence number of the next byte in the byte stream (the first byte of a new data segment that you send will use this sequence number)
- *self.snduna* : the earliest sequence number that has been sent but has not yet been acknowledged

- *self.rcvnxt* : the sequence number of the next byte that your implementation expects to receive from the remote host. For this exercise, you do not need to maintain a receive buffer and your implementation can discard the out-of-sequence segments that it receives

For this exercise we will limit the number of unacknowledged segments to *one*. You can thus implement the sending buffer by storing the last unacknowledged segment :

- *self.lastsegment* : last unacknowledged segment

The TCB also needs to contain the current size of the windows :

- *self.sndwnd* : the current sending window
- *self.rcvwnd* : the current window advertised by the receiver

For this exercise, you do not need to process the *window* field of the received segments. You can assume that the receiver is always advertising a window of a least one segment.

4.3 Practical issues

For this exercise, you will reuse the UML virtual machines that you have used for the two previous exercises.

As you are implementing a protocol that is already supported by the Linux kernel, you need to ensure that the Linux kernel will not reply to the TCP segments that your implementation receives. For this, the easiest solution is to configure the firewall [1] on the UML machine where you are running scapy to block all TCP segments. TCP should only be blocked on the interface between the two UML machines. For example, if you run scapy on the UML1 and test it against UML2, you could configure UML1's firewall as follows

```
iptables -A INPUT -p tcp -i eth0 -j DROP
iptables -A OUTPUT -p tcp -o eth0 -j DROP
```

The first line configures the firewall to drop all TCP segments destined to the Linux kernel on interface *eth0* while the second line drops all TCP segments sent by the Linux kernel on interface *eth0*. The Linux firewall does not interfere with scapy, but of course it prevents you from using any TCP-based client or server such as telnet on the *eth0* interface. If you run into problems, you can remove all the rules configured in the firewall with

```
iptables --flush
```

Additional information about the netfilter firewall used on the Linux kernel may be found at http://www.netfilter.org/documentation/index.html or in the `iptables(8)` man page.

As most Unix variants, Linux supports the `netstat(8)` command. This command allows you to extract various statistics from the networking stack on the Linux kernel. For TCP, *netstat* can list all the active TCP connections with the state of their FSM. *netstat* supports the following options that could be useful during this exercices :

- *-t* requests information about the TCP connections
- *-n* requests numeric output (by default, *netstat* sends DNS queries to resolve IP addresses in hosts and uses */etc/services* to convert port number in service names, *-n* is recommended on the UML machines)
- *-e* provides more information about the state of the TCP connections
- *-o* provides information about the timers
- *-a* provides information about all TCP connections, not only those in the Established state

[1] A firewall is a software or hardware device that analyses TCP/IP packets and decides, based on a set of rules, to accept or discard the packets received or sent. The rules used by a firewall usually depend on the value of some fields of the packets (e.g. type of transport protocols, ports, ...). We will discuss in more details the operation of firewalls in the network layer chapter.

4.4 Deliverables

Each team of two students will either implement [2]: :

1. One FSM for a TCP client that connects to a TCP server (e.g. a python server using the socket API on a Linux kernel), sends reliably one segment of data, receives one segment, acknowledges it and prints it on stdout
2. One FSM for a TCP server that accepts one connection from a client (this client could be a python client using the socket API on top of a Linux kernel), receives one data segment, prints it on stdout, acknowledges it and sends reliably one data segment in response

Each group must ensure that at least one team implements a client and one team implements a server.

The two FSMs must correctly process the TCP segments with the *SYN*, *RST* and *ACK* flags. The other flags (notably *FIN*, *PSH* and *URG*) can be ignored for this exercise.

To simplify the TCP connection establishment, you can ignore the options in the *SYN* segment and assume the default **MSS_** size of 536 bytes as defined in **RFC 793**.

For the TCP client, your implementation should be structured as follows

```
class TCP_simpleclient(Automaton):

   def parse_args(self, remoteip, remoteport, data, *args, **kargs):
       Automaton.parse_args(self, **kargs)
       # to be completed

   def master_filter(self, pkt):
       return (IP in pkt
              # to be completed
              )

   @ATMT.state(initial=1)
   def INIT(self):
   # to be completed

   @ATMT.state()
   def SYN_SENT(self):
   # to be completed

   @ATMT.state()
   def SYN_RCVD(self):
   # to be completed

   @ATMT.state()
   def ESTABLISHED(self):
       # to be completed

   @ATMT.state(final=1)
   def CLOSED(self):
       # send RST segment
       p=IP(dst=self.remoteip)/TCP(seq=self.sndnxt,sport=self.localport,
                                   dport=self.remoteport,ack=self.rcvnxt,
                                   window=self.sndwnd,flags='R')
       print "Done !"

   # receive_conditions and timeouts to be added
```

[2] The astute reader or expert googler might notice that some implementations of the TCP FSM have already been written. For example, the scapy distribution contains in the file *scapy/layers/inet.py* a *TCP_client* Automaton. Although this Automaton can work in some cases, it does not completely implement the TCP FSM. A more detailed TCP FSM in scapy has been implemented by Adam Pridgen http://www.thecoverofnight.com/projects/code/basic_tcp_sm.py, but this FSM goes beyond this simple exercise.

For the TCP server, your implementation should be structured as follows

```
class TCP_simpleserver(Automaton):

   def parse_args(self, remoteip, localport, *args, **kargs):
       Automaton.parse_args(self, **kargs)
       # to be completed

   def master_filter(self, pkt):
       return (IP in pkt
               # to be completed
               )

   @ATMT.state(initial=1)
   def INIT(self):
   # to be completed

   @ATMT.state()
   def SYN_RCVD(self):
   # to be completed

   @ATMT.state()
   def SYN_SENT(self):
   # to be completed

   @ATMT.state()
   def ESTABLISHED(self):
   # to be completed

   @ATMT.state(final=1)
   def CLOSED(self):
       # send RST segment
       p=IP(dst=self.remoteip)/TCP(seq=self.sndnxt,sport=self.localport,
                                   dport=self.remoteport,ack=self.rcvnxt,
                                   window=self.sndwnd,flags='R')
       print "Done !"

   # receive_conditions and timeouts to be added
```

You do not need to implement the TCP connection release. To release the TCP connection, you can set a long timeout in the *ESTABLISHED* and enter the *CLOSED* state when this timer expires

```
@ATMT.timeout(ESTABLISHED,30)
def reset_connection(self):
    raise self.CLOSED()
```

Remember that the initial TCP specification **RFC 793** defines a go-back-n mechanism for TCP. This implies that your implementation can behave as a go-back-n receiver (e.g. ignore out-of-sequence segments) or sender. To simplify the retransmission of segments, consider that you send a single segment at a time. You can advertise a TCP window size of one segment (536 bytes) to limit the number of segments that the remote TCP implementation will send you.

Do not forget to document your code and provide some explanations about the tests you perform with your FSM to verify its interoperability with the TCP implementation of the Linux kernel.

To test your implementation, you can use a simple client or server written by using the socket API.

4.4.1 The socket API

Network applications are often written by using the socket API. This API is a relatively low-level API that allows to develop servers and clients. The documentation of the socket API in python may be found at http://docs.python.org/library/socket.html

For example, the code below [3] provides a simple socket client written in python that opens a TCP connection to a remote server, sends *Hello, world*, prints the received answer on stdout and waits for 10 seconds before closing the connection

```
#! /usr/bin/env python

import sys
import socket    # the socket API in python
import time

if len(sys.argv)!=3:
  print "Usage  rcv-tcp.py <host> <port>"
  sys.exit(-1)

HOST = sys.argv[1]               # any IP address of the server
PORT = int(sys.argv[2])          # server port

# creation of the socket.
# socket.AF_INET indicates that we will use IPv4
# socket.SOCK_STREAM indicates that we will use TCP over this socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# open a TCP connection to HOST:PORT
s.connect((HOST, PORT))

print "Connected to", HOST

# sends a string over the TCP connection
s.send("Hello,world")

# wait for data, a default buffer of 1024 bytes is provided
data = s.recv(1024)
if not data:
  print "Error : no data received"
else:
  print "Received: ",data

time.sleep(10)

print "Closing connection"

# release of the connection
s.close()
```

The code below is for a server that receives a string, prints it on stdout, echoes it back to the client and releases the TCP connection after a delay of 10 seconds

```
#! /usr/bin/env python

import sys
import socket      # the socket API

if len(sys.argv)!=2:
  print "Usage  rcv-tcp.py <port>"
  sys.exit(-1)

HOST = ''                        # any IP address of the host
PORT = int(sys.argv[1])          # server port
# creation of the socket.
# socket.AF_INET indicates that we will use IPv4
# socket.SOCK_STREAM indicates that we will use TCP over this socket
```

[3] This code is adapted from the python documentation available at http://docs.python.org/library/socket.html

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# socket is bound to PORT and will wait for TCP connections
s.bind(('', PORT))
# socket listens for maximum 1 TCP connection in the queue
s.listen(1)
# s.accept() succeeds when a TCP connection is established
conn, addr = s.accept()
print 'Connected by', addr
# wait for data, a default buffer of 1024 bytes is provided
data = conn.recv(1024)
if not data:
   print "Error : no data received"
else:
  print "Received: ",data
# echoes the received data to client
conn.send(data)
time.sleep(10)
print "Closing connection"
# release of the connection
conn.close()
```

CHAPTER

FIVE

TCP CONGESTION CONTROL

The TCP congestion control mechanisms, defined in **RFC 5681** plays a key role in today's Internet. Without this mechanism that was first defined and implemented in the late 1980s, the Internet would not have been able to continue to work until now. The objective of this exercise is to allow you to have a better understanding of the operation of TCP's congestion control mechanism by analysing all the segments exchanged over a TCP connection.

The deadline for this exercise is Tuesday October 27th, 13.00.

5.1 Experimental setup

For this exercise, we have performed measurements in the emulated [1] network shown below.

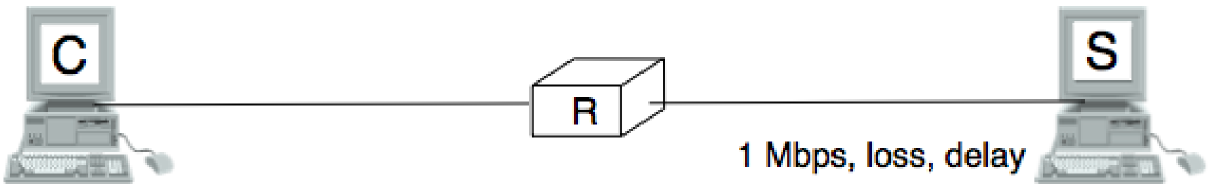


Figure 5.1: Emulated network

The emulated network is composed of three UML machines [2]: a client, a server and a router. The client and the server are connected via the router. The client sends 1 MBytes of data to the server by using iperf. The link between the router and the client is controlled by using the netem Linux kernel module. This module allows us to insert additional delays, reduce the link bandwidth and insert random packet losses.

We used netem To perform three measurements :

1. no losses on the link between *R* and *S*, 100 milliseconds of delay
2. 1% of segment losses on the link between *R* and *S*, 100 milliseconds of delay
3. 10% of segment losses on the link between *R* and *S*, 100 milliseconds of delay

Note that due to the way netem has been configured, the delays and the losses are only applied on packets received by *S*, not on packets sent by *S*.

[1] With an emulated network, it is more difficult to obtain quantitative results than with a real network since all the emulated machines need to share the same CPU and memory. This creates interactions between the different emulated machines that do not happen in the real world. However, since the objective of this exercise is only to allow the students to understand the behaviour of the TCP congestion control mechanism, this is not a severe problem.

[2] For more information about the TCP congestion control schemes implemented in the Linux kernel, see http://linuxgazette.net/135/pfeiffer.html and http://www.cs.helsinki.fi/research/iwtcp/papers/linuxtcp.pdf or the source code of a recent Linux. A description of some of the sysctl variables that allow to tune the TCP implementation in the Linux kernel may be found in http://fasterdata.es.net/TCP-tuning/linux.html. For this exercise, we have configured the Linux kernel to use the NewReno scheme **RFC 3782** that is very close to the official standard defined in **RFC 5681**

For each measurement, we have collected a packet trace that can be analysed by using wireshark and two tcpprobe traces. You can download these traces from the links below :

- **traces/0pc_100msec.tar.gz** : 100 milliseconds delay, no packet losses
- **traces/1pc_100msec.tar.gz** : 100 milliseconds delay, 1% packet losses
- **traces/10pc_100msec.tar.gz** : 100 milliseconds delay, 10% packet losses

Each team of two students will analyse these three traces, first by using wireshark and then by looking at the tcpprobe trace to find more detailed explanation. For each trace, you must :

1. identify the TCP options that have been used on the TCP connection
2. try to find explanations for the evolution of the round-trip-time on each of these TCP connections. For this, you can use the *round-trip-time* graph of wireshark

3. verify whether the TCP implementation used implemented *delayed acknowledgements* 3. analyse the packet trace without packet losses and explain the behaviour of TCP congestion control scheme by looking at the tcpprobe traces 4. inside the traces with packet losses, find :

1. one segment that has been retransmitted by using *fast retransmit*. Explain this retransmission in details.
2. one segment that has been retransmitted thanks to the expiration of TCP's retransmission timeout. Explain why this segment could not have been retransmitted by using *fast retransmit*.

1. wireshark contain several two useful graphs : the *round-trip-time* graph and the *time sequence* graph. Explain how you would compute the same graph from such a trace
2. When displaying TCP segments, recent versions of wireshark contain *expert analysis* heuristics that indicate whether the segment has been retransmitted, whether it is a duplicate ack or whether the retransmission timeout has expired. Explain how you would implement the same heuristics as wireshark.

5.2 Packet trace analysis tools

When debugging networking problems or to analyse performance problems, it is sometimes useful to capture the segments that are exchanged between two hosts and to analyse them.

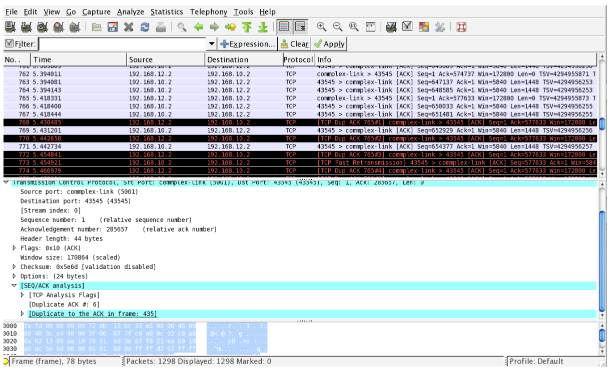
Several packet trace analysis softwares are available, either as commercial or open-source tools. These tools are able to capture all the packets exchanged on a link. Of course, capturing packets require administrator privileges. They can also analyse the content of the captured packets and display information about them. The captured packets can be stored in a file for offline analysis.

tcpdump is probably one of the most well known packet capture software. It is able to both capture packets and display their content. tcpdump is a text-based tool that can display the value of the most important fields of the captured packets. Additional information about tcpdump may be found in `tcpdump(1)`. The text below is an example of the output of tcpdump for the first TCP segments exchanged on an scp transfer between two hosts

```
21:05:56.230737 IP 192.168.1.101.54150 > 130.104.78.8.22: S 1385328972:1385328972(0) win 65535 <m
21:05:56.251468 IP 130.104.78.8.22 > 192.168.1.101.54150: S 3627767479:3627767479(0) ack 13853289
21:05:56.251560 IP 192.168.1.101.54150 > 130.104.78.8.22: . ack 1 win 65535 <nop,nop,timestamp 27
21:05:56.279137 IP 130.104.78.8.22 > 192.168.1.101.54150: P 1:21(20) ack 1 win 49248 <nop,nop,tim
21:05:56.279241 IP 192.168.1.101.54150 > 130.104.78.8.22: . ack 21 win 65535 <nop,nop,timestamp 2
21:05:56.279534 IP 192.168.1.101.54150 > 130.104.78.8.22: P 1:22(21) ack 21 win  65535 <nop,nop,t
21:05:56.303527 IP 130.104.78.8.22 > 192.168.1.101.54150: . ack 22 win 49248 <nop,nop,timestamp 1
21:05:56.303623 IP 192.168.1.101.54150 > 130.104.78.8.22: P 22:814(792) ack 21 win 65535 <nop,nop
```

You can easily recognise in the output above the *SYN* segment containing the *MSS*, *window scale*, *timestamp* and *sackOK* options, the *SYN+ACK* segment whose *wscale* option indicates that the server uses window scaling for this connection and then the first few segments exchanged on the connection.

wireshark is more recent than tcpdump. It evolved from the ethereal packet trace analysis software. It can be used as a text tool like tcpdump. For a TCP connection, wireshark would provide almost the same output as tcpdump. The main advantage of wireshark is that it also includes a graphical user interface that allows to perform various types of analysis on a packet trace.


File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply

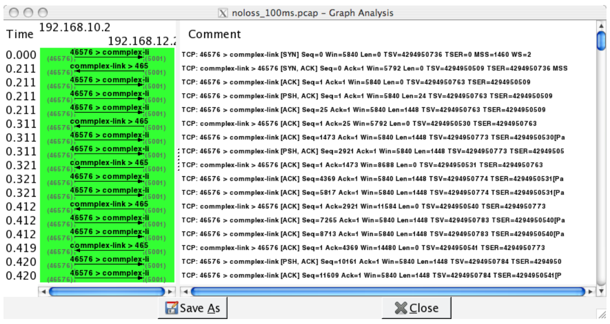
```
No. . Time        Source         Destination    Protocol Info
  762 5.394011    192.168.12.2   192.168.10.2   TCP      commplex-link > 43545 [ACK] Seq=1 Ack=574737 Win=172800 Len=0 TSV=4294955871 T
  763 5.394081    192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=647137 Ack=1 Win=5840 Len=1448 TSV=4294956253
  764 5.394143    192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=648585 Ack=1 Win=5840 Len=1448 TSV=4294956253
  765 5.418331    192.168.12.2   192.168.10.2   TCP      commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Len=0 TSV=4294955873 T
  766 5.418400    192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=650033 Ack=1 Win=5840 Len=1448 TSV=4294956255
  767 5.418444    192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=651481 Ack=1 Win=5840 Len=1448 TSV=4294956255
  768 5.430485    192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#1] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le
  769 5.431201    192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=652929 Ack=1 Win=5840 Len=1448 TSV=4294956256
  770 5.442658    192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#2] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le
  771 5.442734    192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=654377 Ack=1 Win=5840 Len=1448 TSV=4294956257
  772 5.454841    192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#3] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le
  773 5.454921    192.168.10.2   192.168.12.2   TCP      [TCP Fast Retransmission] 43545 > commplex-link [ACK] Seq=577633 Ack=1 Win=584
  774 5.466979    192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#4] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le

Transmission Control Protocol, Src Port: commplex-link (5001), Dst Port: 43545 (43545), Seq: 1, Ack: 285657, Len: 0
    Source port: commplex-link (5001)
    Destination port: 43545 (43545)
    [Stream index: 0]
    Sequence number: 1    (relative sequence number)
    Acknowledgement number: 285657    (relative ack number)
    Header length: 44 bytes
  Flags: 0x10 (ACK)
    Window size: 170864 (scaled)
  Checksum: 0x5e6d [validation disabled]
  Options: (24 bytes)
  [SEQ/ACK analysis]
    [TCP Analysis Flags]
     [Duplicate ACK #: 6]
    [Duplicate to the ACK in frame: 435]

0000  fe fd 00 00 00 00 72 eb  15 bc 33 e6 08 00 45 00   ......r. ..3...E.
0010  00 40 3c e4 40 00 3f 06  67 7f c0 a8 0c 02 c0 a8   .@<.@.?. g.......
0020  0a 02 13 89 aa 19 70 51  ed 3e 6f f8 21 ea b0 10   ......pQ .>o.!...
0030  a6 dc 5e 6d 00 00 01 01  08 0a ff ff d2 67 ff ff   ..^m.... .....g..

Frame (frame), 78 bytes    Packets: 1298 Displayed: 1298 Marked: 0    Profile: Default
```

Figure 5.2: Wireshark : default window

The wireshark window is divided in three parts. The top part of the window is a summary of the first packets from the trace. By clicking on one of the lines, you can show the detailed content of this packet in the middle part of the window. The middle of the window allows you to inspect all the fields of the captured packet. The bottom part of the window is the hexadecimal representation of the packet, with the field selected in the middle window being highlighted.

wireshark is very good at displaying packets, but it also contains several analysis tools that can be very useful. The first tool is *Follow TCP stream*. It is part of the *Analyze* menu and allows you to reassemble and display all the payload exchanged during a TCP connection. This tool can be useful if you need to analyse for example the commands exchanged during a SMTP session.

The second tool is the flow graph that is part of the *Statistics* menu. It provides a time sequence diagram of the packets exchanged with some comments about the packet contents. See blow for an example.



```
0.000  TCP: 46576 > commplex-link [SYN] Seq=0 Win=5840 Len=0 TSV=4294950736 TSER=0 MSS=1460 WS=2
0.211  TCP: commplex-link > 46576 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 TSV=4294950509 TSER=4294950736 MSS
0.211  TCP: 46576 > commplex-link [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=4294950763 TSER=4294950509
0.211  TCP: 46576 > commplex-link [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=24 TSV=4294950763 TSER=4294950509
0.211  TCP: 46576 > commplex-link [ACK] Seq=25 Ack=1 Win=5840 Len=1448 TSV=4294950763 TSER=4294950509
0.311  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=25 Win=5792 Len=0 TSV=4294950530 TSER=4294950763
0.311  TCP: 46576 > commplex-link [ACK] Seq=1473 Ack=1 Win=5840 Len=1448 TSV=4294950773 TSER=4294950530[Pa
0.311  TCP: 46576 > commplex-link [PSH, ACK] Seq=2921 Ack=1 Win=5840 Len=1448 TSV=4294950773 TSER=42949505
0.321  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=1473 Win=8688 Len=0 TSV=4294950531 TSER=4294950763
0.321  TCP: 46576 > commplex-link [ACK] Seq=4369 Ack=1 Win=5840 Len=1448 TSV=4294950774 TSER=4294950531[Pa
0.321  TCP: 46576 > commplex-link [ACK] Seq=5817 Ack=1 Win=5840 Len=1448 TSV=4294950774 TSER=4294950531[Pa
0.412  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=2921 Win=11584 Len=0 TSV=4294950540 TSER=4294950773
0.412  TCP: 46576 > commplex-link [ACK] Seq=7265 Ack=1 Win=5840 Len=1448 TSV=4294950783 TSER=4294950540[Pa
0.412  TCP: 46576 > commplex-link [ACK] Seq=8713 Ack=1 Win=5840 Len=1448 TSV=4294950783 TSER=4294950540[Pa
0.419  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=4369 Win=14480 Len=0 TSV=4294950541 TSER=4294950773
0.420  TCP: 46576 > commplex-link [PSH, ACK] Seq=10161 Ack=1 Win=5840 Len=1448 TSV=4294950784 TSER=4294950
0.420  TCP: 46576 > commplex-link [ACK] Seq=11609 Ack=1 Win=5840 Len=1448 TSV=4294950784 TSER=4294950541[P
```

Figure 5.3: Wireshark : flow graph

The third set of tools are the *TCP stream graph* tools that are part of the *Statistics menu*. These tools allow you to plot various types of information extracted from the segments exchanged during a TCP connection. A first interesting graph is the *sequence number graph* that shows the evolution of the sequence number field of the captured segments with time. This graph can be used to detect graphically retransmissions.

A second interesting graph is the *round-trip-time* graph that shows the evolution of the round-trip-time in function of time. This graph can be used to check whether the round-trip-time remains stable or not. Note that from a packet trace, wireshark can plot two *round-trip-time* graphs, One for the flow from the client to the server and the other one. wireshark will plot the *round-trip-time* graph that corresponds to the selected packet in the top wireshark window.

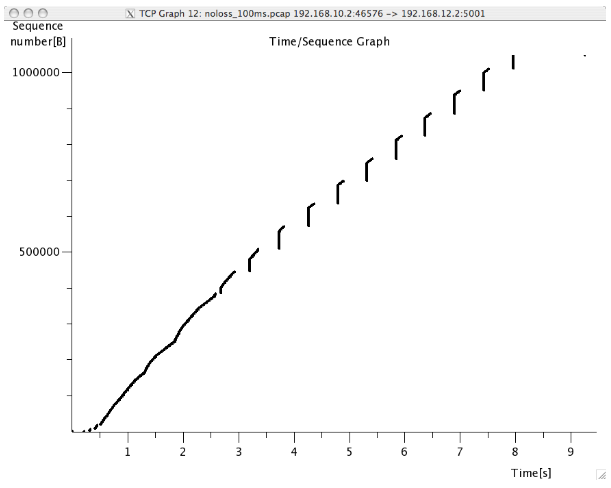


Figure 5.4: Wireshark : sequence number graph

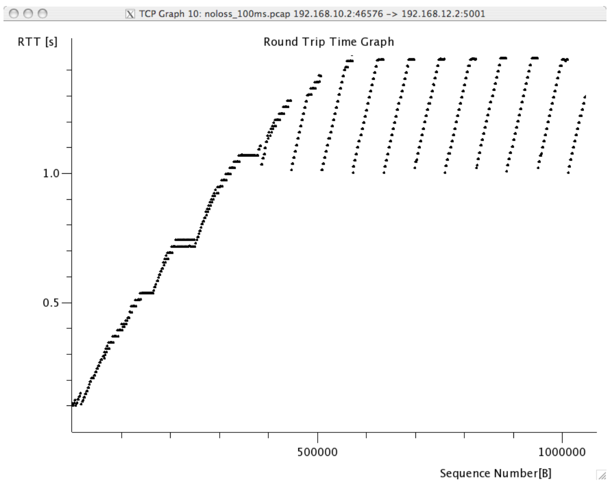


Figure 5.5: Wireshark : round-trip-time graph

5.2.1 scapy

In addition to allowing you to send and receive packets, scapy also allows you to easily read packet traces captured by tools such as tcpdump or wireshark provided that they are in the libpcap format. The *rdpcap* method allows you to read an entire trace in memory and convert it into a list of scapy packets

```
>>> l=rdpcap("/mnt/host/tcp.pcap")
>>> l
<tcp.pcap: TCP:2863 UDP:48 ICMP:0 Other:14>
```

This packet trace contains 2863 TCP segments, 48 UDP segments and 14 other packets. You can easily access any of the TCP segments of the trace

```
>>> l[1234]
<Ether  dst=00:19:e3:d7:12:04 src=00:0f:66:5b:53:9a type=0x800 |<IP  version=4L ihl=5L tos=0x0 le
>>> ls(l[1234][TCP])
sport      : ShortEnumField       = 22              (20)
dport      : ShortEnumField       = 54150           (80)
seq        : IntField             = 3627769700L     (0)
ack        : IntField             = 1386377058      (0)
dataofs    : BitField             = 11L             (None)
reserved   : BitField             = 0L              (0)
flags      : FlagsField           = 16L             (2)
window     : ShortField           = 49248           (8192)
chksum     : XShortField          = 51890           (None)
urgptr     : ShortField           = 0               (0)
options    : TCPOptionsField      = [('NOP', None), ('NOP', None), ('Timestamp', (1212095749, 274
>>> l[1234][TCP].window
49248
```

You can write python scripts to extract information from a libpcap trace. When writing such a script, do not forget that the trace contains the segments sent and received by a host. For this exercise, scapy is not required. You can answer the questions without using scapy

5.2.2 tcpprobe

During the previous exercise, you have used `netstat(8)` To lookup the state of the TCP connections on a given host. On the Linux kernel, there are tools that can provide more information than `netstat(8)`. One of these tools is the TCPProbe kernel module. When installed on a Linux kernel, this kernel module prints one ASCII line containing the following information upon the arrival and the transmission of each TCP segment :

1. The timestamp (seconds.nanoseconds)
2. The source endpoint (address:port)
3. The destination endpoint (address:port)
4. This column should be ignored
5. This column should be ignored
6. The current value of *snd.nxt*
7. The current value of *snd.una*
8. The current value of the congestion window *snd.cwnd* (in segments)
9. The current value of the slow-start threshold *snd.ssthresh* (in segments)
10. The current size of the sending window *snd.wnd* (in bytes)
11. The current value of the smoothed round-trip-time *srtt* (in multiples of 10 milliseconds)
12. The current value of *rcv.nxt*

13. This column should be ignored
14. The current value of the receive window *rcv.wnd*

A sample TCPProbe trace is shown below

```
14.449378000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6fed8c49 0x6fed8689 2 2147483647 5792 8
14.459272000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6fed9799 0x6fed86a1 3 2147483647 5792 7
14.471374000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6feda2e9 0x6fed8c49 4 2147483647 8688 6
14.483485000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6fedae39 0x6fed91f1 5 2147483647 11584 6
14.495677000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6fedb989 0x6fed9799 6 2147483647 14480 5
14.507770000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6fedc4d9 0x6fed9d41 7 2147483647 17376 5
14.519939000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6fedd029 0x6feda2e9 8 2147483647 20272 5
14.532096000 192.168.10.2:48044 192.168.12.2:5001 0 32 0x6feddb79 0x6feda891 9 2147483647 23168 5
```

As you can see from the trace, the values of *snd.nxt* and *snd.una* are in hexadecimal. The congestion window and the slow-start threshold are expressed in MSS-sized segments. We see clearly in the trace above the congestion window that increases. The slow-start threshold is initialised at *2147483647* when the TCP connection starts. Its value will be updated after the first congestion event. Information from a tcpprobe trace can be easily plotted by using a tool such as gnuplot. See http://www.linuxfoundation.org/en/Net:TcpProbe for an example gnuplot script to plot the evolution of *snd.cwnd* and *snd.ssthresh*.

CHAPTER

SIX

ROUTING PROTOCOLS

The network layer contains two types of protocols :

- the *data plane* protocols such as IP that define the format of the packets that are exchanged between routers and how they must be forwarded
- the *routing protocols*, that are part of the *control plane*. Routers exchange routing messages in order to build their routing tables and forwarding tables to forward the packets in the data plane

Several types of routing protocols are used in IP networks. In this set of exercises, you will study intradomain routing protocols. More precisely, you will analyse the operation of a routing protocol that uses distance vectors with split horizon.

The deadline for this exercise is Tuesday November 3rd, at 13.00. Inside each group, each team of two students will write a small report in ASCII or pdf format containing the answers to the questions below.

6.1 Questions

1. Routing protocols used in IP networks only use positive link weights. What would happen with a distance vector routing protocol in the network below that contains a negative link weight ?

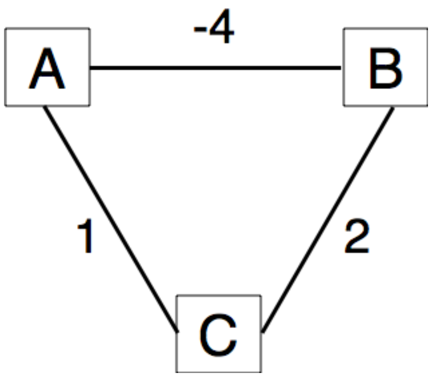


Figure 6.1: Simple network

1. When a network specialist designs a network, one of the problems that he needs to solve is to set the metrics the links in his network. In the USA, the Abilene network interconnects most of the research labs and universities. The figure below shows the topology [1] of this network in 2009.

In this network, assume that all the link weights are set to 1. What is the paths followed by a packet sent by the router located in *Los Angeles* to reach :

- the router located in *New York*

[1] This figure was downloaded from the Abilene observatory http://www.internet2.edu/observatory/archive/data-views.html. This observatory contains a detailed description of the Abilene network including detailed network statistics and all the configuration of the equipment used in the network.

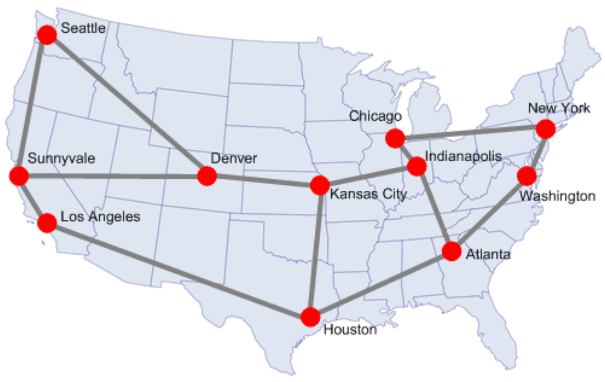


Figure 6.2: The Abilene network

- the router located in *Washington* ?
- Is it possible to configure the link metrics so that the packets sent by the router located in *Los Angeles* to the routers located in respectively *New York* and *Washington* do not follow the same path ?
- Is it possible to configure the link weights so that the packets sent by the router located in *Los Angeles* to router located in *New York* follow one path while the packets sent by the router located in *New York* to the router located in *Los Angeles* follow a completely different path ?
- Assume that the routers located in *Denver* and *Kansas City* need to exchange lots of packets. Can you configure the link metrics such that the link between these two routers does not carry any packet sent by another router in the network ?

1. In the five nodes network shown below, can you configure the link metrics so that the packets sent by router *E* to router *A* use link *B->A* while the packets sent by router *B* use links *B->D* and *D->A*?

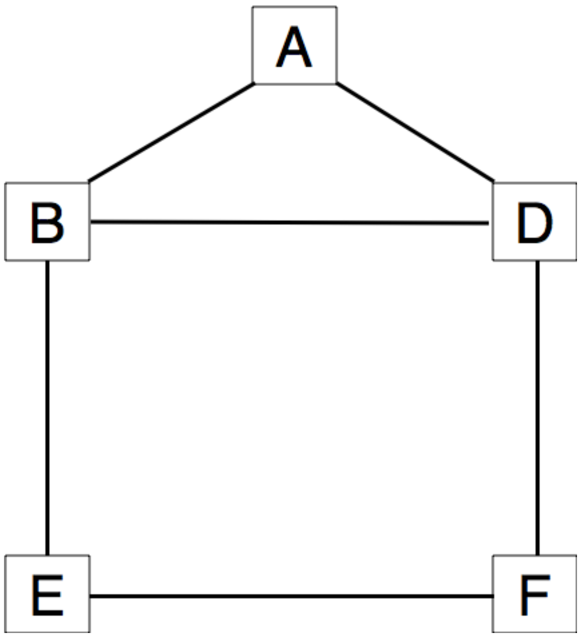


Figure 6.3: Simple five nodes network

1. In the five nodes network shown above, can you configure the link weights so that the packets sent by router *E* (resp. *F*) follow the *E->B->A* path (resp. *F->D->B->A*) ?
2. In the above questions, you have worked on the stable state of the routing tables computed by routing protocols. Let us now consider the transient problems that main happen when the network topology changes

[2]. For this, consider the network topology shown in the figure below and assume that all routers use a distance vector protocol that uses split horizon.

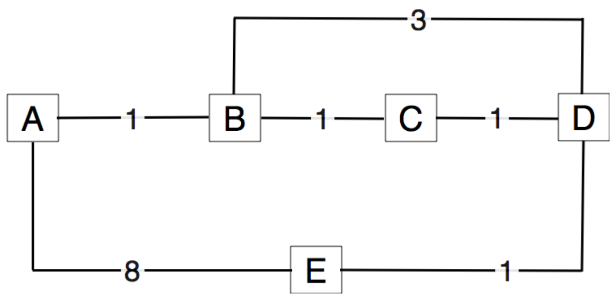


Figure 6.4: Simple network

If you compute the routing tables of all routers in this network, you would obtain a table such as the table below :

Destination	Routes on A	Routes on B	Routes on C	Routes on D	Routes on E
A	0	1 via A	2 via B	3 via C	4 via D
B	1 via B	0	1 via B	2 via C	3 via D
C	2 via B	1 via C	0	1 via C	2 via D
D	3 via B	2 via C	1 via D	0	1 via D
E	4 via B	3 via C	2 via D	1 via E	0

Distance vector protocols can operate in two different modes : *periodic updates* and *triggered updates*. *Periodic updates* is the default mode for a distance vector protocol. For example, each router could advertise its distance vector every thirty seconds. With the *triggered updates* a router sends its distance vector when its routing table changes (and periodically when there are no changes).

- Consider a distance vector protocol using split horizon and *periodic updates*. Assume that the link *B*-*C* fails. *B* and *C* update their local routing table but they will only advertise it at the end of their period. Select one ordering for the *periodic updates* and every time a router sends its distance vector, indicate the vector sent to each neighbor and update the table above. How many periods are required to allow the network to converge to a stable state ?
- Consider the same distance vector protocol, but now with *triggered updates*. When link *B*-*C* fails, assume that *B* updates its routing table immediately and sends its distance vector to *A* and *D*. Assume that both *A* and *D* process the received distance vector and that *A* sends its own distance vector, ... Indicate all the distance vectors that are exchanged and update the table above each time a distance vector is sent by a router (and received by other routers) until all routers have learned a new route to each destination. How many distance vector messages must be exchanged until the network converges to a stable state ?

[2] The main events that can affect the topology of a network are : - the failure of a link. Measurements performed in IP networks have shown that such failures happen frequently and usually for relatively short periods of time - the addition of one link in the network. This may be because a new link has been provisioned or more frequently because the link failed some time ago and is now back - the failure/crash of a router followed by its reboot. - a change in the metric of a link by reconfiguring the routers attached to the link See http://totem.info.ucl.ac.be/lisis_tool/lisis-example/ for an analysis of the failures inside the Abilene network in June 2005 or http://citeseer.ist.psu.edu/old/markopoulou04characterization.html for an analysis of the failures affecting a larger ISP network

CHAPTER

SEVEN

A PROTOTYPE IPV4 ROUTER IN SCAPY

During this exercise, you will implement a subset of a prototype IPv4 router by using scapy. Your router will be able to :

- forward IPv4 packets
- reply to ICMP *Echo request* messages produced by `ping(8)`
- support `traceroute(8)`

The deadline for this exercise is Tuesday November 10th, 13.00.

7.1 Preparation of the lab

The first step of this exercise is to prepare the configuration of the emulated network that you will use. Your emulated netwo

- your router with three interfaces : *eth0*, *eth1* and *eth2*
- three clients : *client1*, *client2* and *client3*. Each client has an *eth0* interface

The different interfaces have been connected as follows :

- *eth0* on the router is connected to *eth0* on *client1*
- *eth1* on the router is connected to *eth0* on *client2*
- *eth2* on the router is connected to *eth0* on *client3*

Note: The virtual machines will be placed in the */etinfo/applications/uml2/* directory. You will find in the README file additional information about their installation.

The clients will use the IPv4 implementation of the Linux kernel while the router will use your implementation written in scapy. As you will use the emulated network to send and receive IPv4 packets, you need to configure the interfaces on all clients. On Linux, the IP addresses assigned on an interface can be configured by using `ifconfig(8)`. When `ifconfig(8)` is used without parameters, it lists all the existing interfaces of the host with their configuration. A sample `ifconfig(8)` output is shown below

```
UML1:~# ifconfig
eth0      Link encap:Ethernet  HWaddr FE:3A:59:CD:59:AD
         Inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
         inet6 addr: fe80::fc3a:59ff:fecd:59ad/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:3 errors:0 dropped:0 overruns:0 frame:0
         TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:216 (216.0 b)  TX bytes:258 (258.0 b)
```

```
          Interrupt:5
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

This UML host has two interfaces : the loopback interface (*lo* with IPv4 address *127.0.0.1* and IPv6 address *::1*) and the *eth0* interface. The *192.168.1.1/24* address and a link local IPv6 address (*fe80::fc3a:59ff:fecd:59ad/64*) have been assigned to interface *eth0*. The broadcast address is used in some particular cases, this is outside the scope of this exercise. `ifconfig(8)` also provides statistics such as the number of packets sent and received over this interface.

Another important information that is provided by `ifconfig(8)` is the hardware address (HWaddr) used by the datalink layer of the interface. On the example above, the *eth0* interface uses the 48 bits *FE:3A:59:CD:59:AD* hardware address.

You can configure the IPv4 address assigned to an interface by specifying the address and the netmask

```
ifconfig eth0 192.168.1.2 netmask 255.255.255.128
```

or you can also specify the prefix length

```
ifconfig eth0 192.168.1.2/25
```

In both cases, *ifconfig eth0* allows you to verify that the interface has been correctly configured

```
eth0      Link encap:Ethernet  HWaddr FE:3A:59:CD:59:AD
          inet addr:192.168.1.2  Bcast:192.168.1.127  Mask:255.255.255.128
          inet6 addr: fe80::fc3a:59ff:fecd:59ad/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:216 (216.0 b)  TX bytes:258 (258.0 b)
          Interrupt:5
```

`ifconfig` also allows you to configure several IPv4 addresses over a single physical interface. This is useful for some particular configuration and we will use these virtual interfaces to reduce the number of interfaces in our emulated network. The first virtual interface of *eth0* is *eth0:0*, the second *eth0:1*, ... You can configure a different IPv4 address on each virtual interface

```
ifconfig eth0:1 10.0.0.1/8
ifconfig eth0:1
eth0:1    Link encap:Ethernet  HWaddr FE:3A:59:CD:59:AD
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:5
```

Use `ifconfig(8)` to configure the following IPv4 addresses :

- *192.168.0.2/24* on *eth0* on *client1*
- *10.1.0.2/30* on *eth0* on *client2*
- one IPv4 address inside *10.0.0.0/25* on *eth0:0* on *client2*
- one IPv4 address inside *8.0.0.0/13* on *eth0:1* on *client2*
- one IPv4 address inside *9.0.0.0/24* on *eth0:2* on *client2*

- *10.1.0.6/30* on *eth0* on *client3*
- one IPv4 address inside *10.0.0.128/25* on *eth0:0* on *client3*
- one IPv4 address inside *8.8.0.0/16* on *eth0:1* on *client3*
- one IPv4 address inside *9.0.0.0/23* on *eth0:2* on *client3*

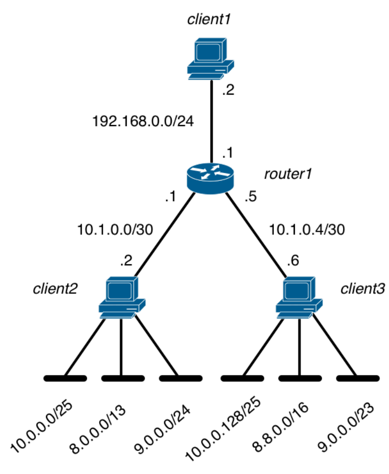


Figure 7.1: Testbed configuration

Concerning the addresses of the virtual interfaces on *client2* and *client3*, when there are overlapping prefixes, ensure that the address assigned in largest subnet on one client does not belong to the more specific subnet allocated to the other client. Verify the configuration of these addresses with `ifconfig(8)`. For each interface, note its hardware address. Then, use `ping(8)` on each virtual machine to verify that the locally assigned IPv4 addresses are working correctly.

The last point that you need to configure is the mapping between the IP addresses and the hardware addresses on the clients. When a client sends an IP packet inside on an Ethernet LAN, it places the packet inside an Ethernet frame that contains the hardware address of the source Ethernet interface, the hardware address of the destination Ethernet interface and the entire IP packet. In a real network, a special protocol [1] is used for this, but implementing this protocol is outside the scope of this exercise. It is also possible to configure these mappings manually by using the `arp(8)` software. `arp(8)` inserts in the Linux kernel a static mapping between an IP address and an Ethernet address. `arp(8)` can be used as follows

```
UML2:~# arp -i eth0 -s 192.168.1.1 C6:41:03:34:F4:98`
Uml2:~# arp -v -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   c6:41:03:34:f4:98   CM                    eth0
Entries: 1     Skipped: 0      Found: 1
```

The first line adds a mapping between IPv4 address *192.168.1.1* and hardware address *C6:41:03:34:F4:98* on interface *eth0*. The *arp -v -n* command allows you to see the content of the mapping table maintained by the kernel.

Use `arp(8)` to insert the following mappings :

- on *client1* insert a mapping between *192.168.0.1* and the hardware address of interface *eth0* on *router1*
- on *client2* insert a mapping between *10.1.0.1* and the hardware address of interface *eth1* on *router1*
- on *client3* insert a mapping between *10.1.0.5* and the hardware address of interface *eth2* on *router1*

Verify each mapping with *arp -v -n*.

As your router will use scapy to process IPv4 packets, you need to disable IPv4 on the Linux kernel of the *router* virtual machine. This can be done by installing an `iptables(8)` on each interface on your *router*

[1] In practice, IPv4 hosts and routers use the Address Resolution Protocol (ARP) specified in **RFC 826** to find the hardware address that corresponds to an IPv4 address. However, implementing this protocol is outside the scope of this exercise.

```
iptables -A INPUT -p ip -i eth0 -j DROP
iptables -A OUTPUT -p ip -o eth0 -j DROP
```

7.2 Implementation of a prototype IPv4 router in scapy

scapy was optimised to allow hosts to easily send crafted IP packets, but it was not designed to implement a router that need

- each Ethernet interface has a unique 48 bits hardware address
- each Ethernet frame contains the hardware address of the interface on the host that sent the frame and the hardware address of the host that will receive the frame on this Ethernet network
- Ethernet frames can contain up to 1500 bytes of data

You can obtain the 48 bits hardware addresses of the Ethernet interfaces on the emulated testbed by using `ifconfig(8)` on each UML.

You can use scapy to easily build an Ethernet frame that contains an IPv4 packet and send it over a particular interface. For example, assuming that *client1* uses the *AA:AA:AA:AA:AA:AA* hardware address while *router1* uses address *BB:BB:BB:BB:BB:BB*, you can proceed as follows to send one IP packet from *client1* to the router

```
# build an Ethernet frame containing an IP packet
frame=Ether(src="AA:AA:AA:AA:AA:AA", dst="BB:BB:BB:BB:BB:BB")/IP(src="192.168.0.2",dst="192.168.0
# send the frame over the eth0 interface
sendp(frame, iface="eth0")
```

Note that the code above uses *sendp* an not *send* to send the Ethernet frame. *send* uses the IP implementation and the routing tables on the local host to decide on which interface the packet should be forward. We will not use *send* in this exercise as your will use the routing tables that you implement in scapy and not the kernel's routing tables.

scapy allows you to easily create and parse IPv4 packets and ICMP messages. For reference, here are the fields of the IPv4 header and of the ICMP header

```
>>> ls(IP)
version    : BitField             = (4)
ihl        : BitField             = (None)
tos        : XByteField           = (0)
len        : ShortField           = (None)
id         : ShortField           = (1)
flags      : FlagsField           = (0)
frag       : BitField             = (0)
ttl        : ByteField            = (64)
proto      : ByteEnumField        = (0)
chksum     : XShortField          = (None)
src        : Emph                 = (None)
dst        : Emph                 = ('127.0.0.1')
options    : IPoptionsField       = ('')
>>> ls(ICMP)
type       : ByteEnumField        = (8)
code       : ByteField            = (0)
chksum     : XShortField          = (None)
id         : ConditionalField     = (0)
seq        : ConditionalField     = (0)
ts_ori     : ConditionalField     = (75208778)
ts_rx      : ConditionalField     = (75208778)
ts_tx      : ConditionalField     = (75208778)
gw         : ConditionalField     = ('0.0.0.0')
ptr        : ConditionalField     = (0)
```

```
reserved   : ConditionalField     = (0)
addr_mask  : ConditionalField     = ('0.0.0.0')
unused     : ConditionalField     = (0)
```

To implement your prototype IPv4 router in scapy, proceed as follows :

First, define several python data structures that allow you to store the configuration of your router, namely :

- the list of all hardware addresses of your router
- the list of all IPv4 addresses that have been assigned to your router
- for each interface of your router, the hardware address of the *client* attached to this interface (we assume that the Ethernet links used in the emulated testbed model point-to-point links and ignore the issues that arise with local area networks)

Second, in addition to its list of IPv4 and hardware addresses, a router needs a routing table where it can easily perform a longest prefix match. Many data structures could be used to implement such a longest prefix match. For this exercise, use the py-radix library available from http://www.mindrot.org/projects/py-radix/ . py-radix has been installed on the virtual machines in the lab. The README file from the py-radix distribution contains some information about the utilisation of this data structure. For this exercise, you need to :

- store in the radix tree the forwarding table that the router shown in the figure above would use. For this, insert each of the prefixes shown in the figure above in the radix tree and associate the appropriate interface as the *data* of each node
- initialise a data structure that maps with each outgoing interface the hardware address of the nexthop router attached to this interface (*client1*, *client2* or *client3*)

Third, start your implementation with the skeleton below

```python
import Queue,sys,radix
from scapy.packet import *
from scapy.fields import *
from scapy.automaton import *
from scapy.layers.inet import *
from scapy.sendrecv import *

TIMEOUT = 2

class InternetRouter(Automaton):
     # IP addresses local to the router
     local_ips = ...

     # hardware addresses local to the router corresponding to each interface
     hard_addr_src = ...
     # Remote hardware addresses corresponding to each interface
     hard_addr_dst = ...

     def parse_args(self, **kargs):
         Automaton.parse_args(self, **kargs)
         # build the radix tree
         ...

     # the master filter ensures that the router captures correctly
     # all Ethernet frames containing IP packets addresses to one
     # of its Ethernet interfaces

     def master_filter(self, pkt):
         return (IP in pkt and pkt.dst in self.mac_addr_src.values() )

     # Scapy IPv4 Forwarding Automata

     @ATMT.state(initial=1)
```

```
    def WAIT_FOR_PACKET_TO_FORWARD(self):
        pass

    @ATMT.timeout(WAIT_FOR_PACKET_TO_FORWARD, TIMEOUT)
    def timeout_waiting_for_packet_to_forward(self):
            print "<WAIT_FOR_IP_PACKET/timeout>: Nothing to forward..."
            raise self.WAIT_FOR_PACKET_TO_FORWARD()
```

Fourth, start to build your prototype router in scapy . Your first objective will be to support `ping(8)`. A client, such as *client1* must be able to *ping* the IP address of your router (i.e. *192.168.0.1*) and receive the correct ICMP message in response. For this, define a first receive condition that process any IP packet whose destination address is one of the IPv4 addresses assigned to the router

```
@ATMT.receive_condition(WAIT_FOR_PACKET_TO_FORWARD)
def received_local_ICMP(self, pkt):
    if (ICMP in pkt and pkt[IP].dst in self.local_ips and pkt[ICMP].type==‘echo-request‘):
    ...
```

When building an ICMP message, scapy supports all the ICMP types and codes. An ICMP message is always sent inside an IPv4 packet. Note that when `ping(8)` sends a ICMP echo request, it sets the value of the *seq* and ìd‘ fields. These two fields must be echoed in the returned *ICMP Echo reply* message. A typical ICMP *Echo reply* in response to a received ICMP *Echo request* will be built as follows

```
IP(src=myIP,dst=pkt[IP].src)/ICMP(type='echo-reply', seq=pkt[ICMP].seq,id=pkt[ICMP].id)/pkt[Raw]
```

Fifth. Now that your router can receive packets on one of your its local addresses, the next step is to forward IPv4 packets. You will implement this forwarding as another receive condition

```
@ATMT.receive_condition(WAIT_FOR_PACKET_TO_FORWARD, prio=2)
def received_fwd_IP(self, pkt):
    if not (pkt[IP].dst in self.local_ips):
        ...
```

To forward the packet, remember that a router needs to :

- update the TTL (assume first that the decremented TTL is >0)
- lookup the packet's destination address in the router's forwarding table using the radix tree (assume first that the packet destination is found in the forwarding table)
- forward the packet over the appropriate outgoing interface

A real router would update the TTL of the received packet and recomputes the packet's checksum incrementally. In scapy, it is easier to extract the payload and the important fields of the received packet and create a new IPv4 packet as in this case scapy automatically computes the checksum. Once the IPv4 packet to be forwarded has been built, add it to an *Ether* frame whose source (resp. destination) hardware address is the hardware address of the router (resp. nexthop router - i.e. *client1* or *client2* in this exercise) on the outgoing link.

Test that your router works by performing a `ping(8)` from *client1* to the addresses configured on *client2* and *client3*.

Finally, a real router should also send ICMP messages in case of errors. Modify your code so that your router :

- sends an ICMP *TTL Exceeded* messages when it receives a packet whose *TTL* is set to *1*. This ICMP message can be built with *type=time-exceeded* and *code=ttl-zero-during-transit* with scapy
- sends an ICMP *Destination unreachable* when it receives a packet whose destination address cannot be found in its forwarding table. This ICMP message can be built with *type=dest-unreach* and *code=network-unreachable*

If these ICMP messages are implemented correctly, your should be able to use `traceroute(8)` in your emulated network.

CHAPTER

EIGHT

A NETWORK ADDRESS TRANSLATOR

The global Internet is a fully distributed and uncoordinated network. Historically IP addresses were allocated in a first-come first-served basis. As a consequence a quick shortage of IPv4 prefixes happened and people started to have difficulties to obtain an address block. As a temporarily solution, IP Network Address Translation (NAT) was designed to use private IPv4 addresses in corporate or home networks and to map and route them with public source addresses when packets are forwarded over the global Internet.

In this exercise you'll implement a gateway that allows communications between hosts that are using **RFC 1918** private addresses in a private network and hosts that are using public IPv4 addresses. Your gateway will be a simplified version of the NAT box described in **RFC 1631**. Compared to a real NAT, your NAT will ignore packet fragmentation, DNS and only support TCP.

The deadline for this exercise is Tuesday November 17th, 13.00.

8.1 Lab preparation

Before implementing a NAT in scapy, you need to setup the emulated environment. You will use three virtual machines.

- Your NAT box with 2 interfaces: *eth0* with a private IP address and *eth1* with a public IP address.
- Host *H1* uses a private IPv4 address on *eth0*
- Host *H2* uses a public IPv4 address on *eth0*

The different interfaces have been connected as follows :

- *eth0* on NAT is connected to *eth0* on *H1*
- *eth1* on NAT is connected to *eth0* on *H2*

H1 and *H2* will use the standard IP implementation of the Linux kernel while *NAT* will use your implementation written in scapy. You will use the emulated network to send and receive public and private IP packets.

The filesystems and the kernel of the virtual machines are stored in the directory */etinfo/INGI2141/uml/nat*. To create the virtual setup illustrated on the figure execute the following procedure:

1. Copy the boot script of *H1*,'H2' and *NAT* virtual machines in your working directory

```
cp /etinfo/INGI2141/uml/nat/start_h1 ~/INGI2141/UML/NAT
cp /etinfo/INGI2141/uml/nat/start_h2 ~/INGI2141/UML/NAT
cp /etinfo/INGI2141/uml/nat/start_nat ~/INGI2141/UML/NAT
```

2. Open 3 terminals, 1 for each virtual machine

3. In the first terminal, get your student id and then start the virtual machine *H1* by executing the script *start_h1*. In the second terminal, start the virtual machine *H2* by executing the script *start_h2*. In the third terminal start the virtual machine *NAT* by executing the script *start_nat*
4. The interfaces of your boxes have already been configured with the following IPv4 addresses :

- *192.168.1.2/24* on *eth0* on *H1*
- *192.168.1.1/24* on *eth0* on *NAT*
- *216.239.59.104/24* on *eth1* on *NAT*
- *216.239.59.103/24* on *eth1* on *H2*

1. Check the IP connectivity between *H1* and *NAT* and *H2* and *NAT* by using `ping(8)`

You need to configure the routing table on *H1* so that all packets with a destination in the public network will be sent via *NAT*. This is can be achieved by adding a default route. To do this use the following command on *H1*

```
route add default gw 192.168.1.1
```

Your NAT will only support TCP. To ensure that the Linux kernel on *NAT* does not intercept the packets that you will process in scapy, you need to configure the following filters on *NAT*:

```
iptables -A INPUT -p ip -i eth1 -j DROP
iptables -A OUTPUT -p ip -o eth1 -j DROP
iptables -A INPUT -p ip -i eth0 -j DROP
iptables -A OUTPUT -p ip -o eth0 -j DROP
```

To check that your NAT implementation is functioning you need to start on a TCP server on *H2*. For this, you can use the simple python server in file */root/srv-tcp.py* on *H2*. This server waits for a TCP connection on the port specified as the first argument.

On *H1* you can use the python client in file */root/cl-tcp.py*

8.1.1 Implementation issues in scapy

To implement a NAT in scapy, you already know how to process IPv4 packets and TCP segments. To ease your implementation, assume none of the IPv4 packets that you will translate need to be fragmented.

The main difficulty when implementing a NAT is to select a data structure that contains the mapping between a *(private IPv4 address, TCP port)* and a *(public IPv4 address, TCP port)*. This data structure will be accessed every time you receive a packet from *H1* destined to a public IPv4 address, but also when a packet is received from *H2*. Note that, although the emulated network only contains two hosts, your implementation must support any number of hosts on both the private and the public sides.

To design your data structure, consider what happens when :

- client *A* opens 100 TCP connections with different source ports to server *C* on destination port *80*
- client *A* opens a TCP connection with *source port=1234* to servers *S1* and *S2* on destination port *1234*
- clients *A* and *B* using private addresses open a TCP connection with *source port=1234* to servers *S1* on destination port *1234*

When implementing your NAT, you will need to solve three problems by using your data structure :

1. When a packet is received from a private host, how do you translate it ? What happens if there is no suitable mapping entry in your NAT ?
2. When a packet is received from a public host, how do you translate it ? What happens if there is no suitable mapping entry in your NAT ?
3. A real NAT will have a finite memory. When do you remove a mapping entry in your data structure ? Discuss your design choice as comments in your code

As the *NAT* virtual machine has been configured with the appropriate IP addresses and routes, you can use the *send()* method to send IP packets in your NAT.

scapy allows you to modify any field in an IPv4 packet or TCP segment. For example, if *p* is a packet, you can use

```
p[IP].dst='1.2.3.4'   # change destination address
p[IP].chksum=None     # remove IP checksum, scapy_ will recompute it
                      # when sending it with send()
p[TCP].dport=1234     # change destination port number
p[TCP].chksum=None    # remove TCP checksum, scapy will recompute it
                      # when sending it with send()
```

To capture the packets to be processed by the NAT, you can use a *master_filter* such as the following one

```
mac_addr_src = { "00:00:E3:00:30:04":'eth0', "00:00:E3:00:30:03":'eth1'}
mac_addr_dst = { "bcastv4":'ff:ff:ff:ff:ff:ff'}
# verify that these are the hardware addresses of your NAT
# with ifconfig

def master_filter(self, pkt):
    return( (TCP in pkt) and
           (not pkt.src.upper() in self.mac_addr_src.keys()) and
           (not pkt.dst.upper() in self.mac_addr_dst.values()) )
```

Note: If you create a new IP packet with scapy, scapy sets the unspecified fields to a default value. When translating a packet, you should only change the fields of the IPv4 header and TCP header that need to be translated and leave the other fields unmodified.

You have written enough scapy prototypes to be able to write your implementation without a skeleton.

CHAPTER

NINE

OSPF AND BGP

In this set of exercices, you will explore in more details the operation of a link-state routing protocol such as OSPF and the operation of a path vector protocol such as BGP.

The deadline for this set of exercises is Tuesday December 1st, 13.00.

9.1 Link state routing

1 Consider the network shown below. In this network, the metric of each link is set to *1* except link *A*-*B* whose metric is set to *4* in both directions. In this network, there are two paths with the same cost between *D* and *C*. Old routers would randomly select one of these equal cost paths and install it in their forwarding table. Recent routers are able to use up to *N* equal cost paths towards the same destination.

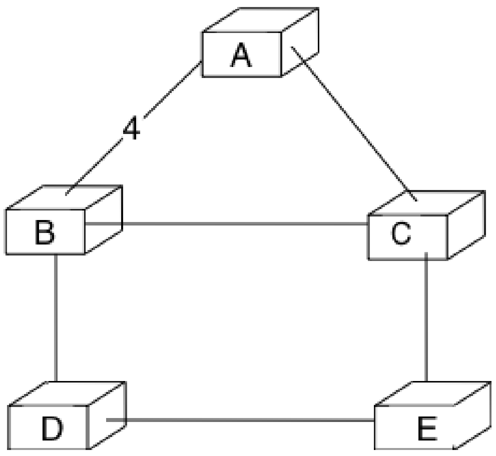


Figure 9.1: A simple network running OSPF

On recent routers, a lookup in the forwarding table for a destination address returns a set of outgoing interfaces. How would you design an algorithm that selects the outgoing interface used for each packet, knowing that to avoid reordering, all segments of a given TCP connection should follow the same path ?

2 Consider again the network shown above. After some time, OSPF converges and all routers compute the following routing tables :

Destination	Routes on A	Routes on B	Routes on C	Routes on D	Routes on E
A	0	2 via C	1 via A	3 via B,E	2 via C
B	2 via C	0	1 via B	1 via B	2 via D,C
C	1 via C	1 via C	0	2 via B,E	1 via C
D	3 via C	1 via D	2 via B,E	0	1 via D
E	2 via C	2 via C,D	1 via E	1 via E	0

An important difference between OSPF and RIP is that OSPF routers flood link state packets that allow the other routers to recompute their own routing tables while RIP routers exchange distance vectors. Consider that link *B*-*C* fails and that router *B* is the first to detect the failure. At this point, *B* cannot reach anymore *A*, *C* and 50% of its paths towards *E* have failed. *C* cannot reach *B* anymore and half of its paths towards *D* have failed.

Router *B* will flood its updated link state packet through the entire network and all routers will recompute their forwarding table. Upon reception of a link state packet, routers usually first flood the received link-state packet and then recompute their forwarding table. Assume that *B* is the first to recompute its forwarding table, followed by *D*, *A*, *C* and finally *E*

1. After each update of a forwarding table, verify which pairs of routers are able to exchange packets. Provide your answer using a table similar to the one shown above.
2. Can you find an ordering of the updates of the forwarding tables that avoids all transient problems ?

9.2 BGP

1 Consider the network shown in the figure below and explain the path that will be followed by the packets to reach *194.100.10.0/23*

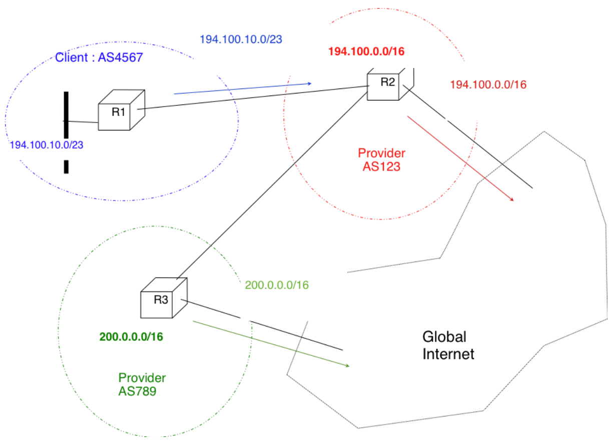


Figure 9.2: A stub connected to one provider

2 Consider, now, as shown in the figure below that the stub AS is now also connected to provider *AS789*. Via which provider will the packets destined to *194.100.10.0/23* will be received by *AS4567* ? Should *AS123* change its configuration ?

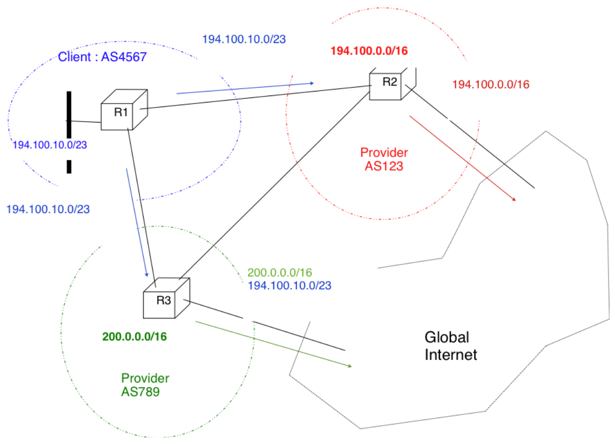


Figure 9.3: A stub connected to two providers

3 Consider that stub shown in the figure below decides to advertise two */24* prefixes instead of its allocated */23* prefix.

1. Via which provider does *AS4567* receive the packets destined to *194.100.11.99* and *194.100.10.1* ?
2. How is the reachabilty of these addresses affected when link *R1-R3* fails ?
3. Propose a configuration on *R1* that achieves the same objective as the one shown in the figure but also preserves the reachability of all IP addresses inside *AS4567* if one of *AS4567*'s interdomain links fails ?

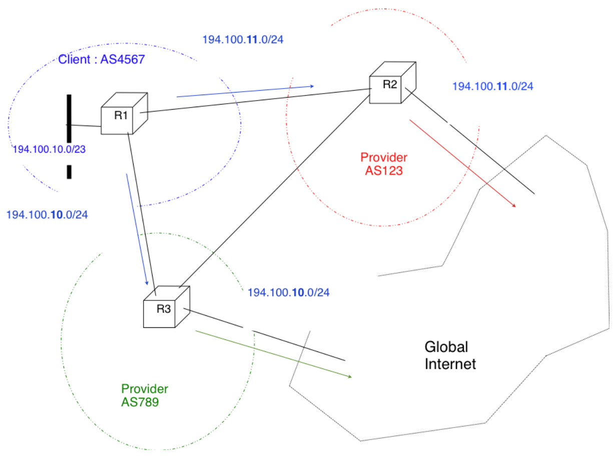


Figure 9.4: A stub connected to two providers

4 Researchers and network operators collect and expose lots of BGP data. For this, they establish eBGP sessions between *data collection* routers and production routers located operationnal networks. Several *data collection* routers are available, the most popular ones are :

- http://www.routeviews.org
- http://www.ripe.net/ris

For this exercice, you will use one of the *routeviews* BGP routers. You can access this router by using telnet. Once logged on the router, you can use the router's command line interface to analyse its BGP routing table

```
telnet route-views.routeviews.org
Trying 128.223.51.103...
Connected to route-views.routeviews.org.
Escape character is '^]'.
C
**********************************************************************

                   Oregon Exchange BGP Route Viewer
         route-views.oregon-ix.net / route-views.routeviews.org

route views data is archived on http://archive.routeviews.org

This hardware is part of a grant from Cisco Systems.
Please contact help@routeviews.org if you have questions or
comments about this service, its use, or if you might be able to
contribute your view.

This router has views of the full routing tables from several ASes.
The list of ASes is documented under "Current Participants" on
http://www.routeviews.org/.
```

```
                          **************

route-views.routeviews.org is now using AAA for logins.  Login with
username "rviews".  See http://routeviews.org/aaa.html

**********************************************************************


User Access Verification

Username: rviews
route-views.oregon-ix.net>
```

This router has eBGP sessions with routers from several ISPs. See http://www.routeviews.org/peers/route-views.oregon-ix.net.txt for an up-to-date list of all eBGP sessions maintained by this router.

Among all the commands supported by this router, the *show ip bgp* command is very useful. This command takes an IPv4 prefix as parameter and allows you to retrieve all the routes that this routers has received in its Adj-RIB-In for the specified prefix.

1. Use *show ip bgp 130.104.0.0/16* to find the best path used by this router to reach UCLouvain
2. Knowing that *130.104.0.0/16* is announced by belnet (AS2611), what are, according to this BGP routing tables, the ASes that peer with belnet
3. Do the same analysis for one of the IPv4 prefixes assigned to Skynet (AS5432) : *62.4.128.0/17*. The output of the *show ip bgp 62.4.128.0/17* reveals something strange as it seems that one of the paths towards this prefix passes twice via *AS5432*. Can you explain this ?

   2905 702 1239 5432 5432

   196.7.106.245 from 196.7.106.245 (196.7.106.245) Origin IGP, metric 0, localpref 100, valid, external

5 Consider the network shown in the figure below and assume that R1‘ advertises a single prefix. R1‘ receives a lot of packets from *R9*. Without any help from *R2*, *R9* or *R4*, how could *R1* configure its BGP advertisement such that it receives the packets from *R9* via *R3* ? What happens when a link fails ?

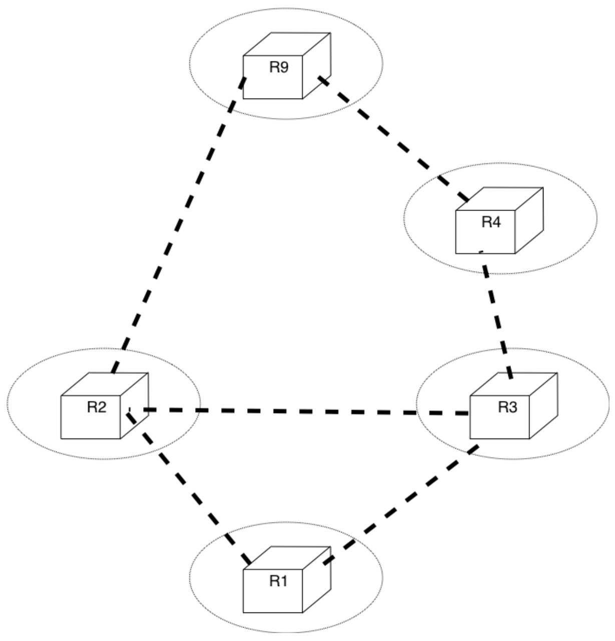


Figure 9.5: A simple internetwork

6 Consider the network below.

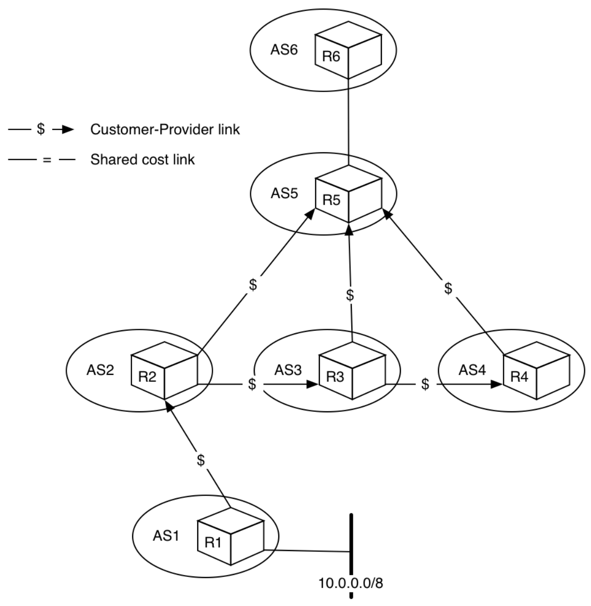


Figure 9.6: A simple internetwork

1. Show which BGP messages are exchanged when router *R1* advertises prefix *10.0.0.0/8*.
2. How many and which routes are known by router *R5* ? Which route does it advertise to *R6*?
3. Assume now that the link between *R1* and *R2* fails. Show the messages exchanged due to this event. Which BGP messages are sent to *R6* ?

7 Consider the network shown in the figure below where *R1* advertises a single prefix. In this network, the link between *R1* and *R2* is considered as a backup link. It should only be used only when the primary link (*R1-R4*) fails. This can be implemented on *R2* by setting a low *local-pref* to the routes received on link *R2-R1*

1. In this network, what are the paths used by all routers to reach *R1* ?
2. Assume now that the link *R1-R4* fails. Which BGP messages are exchanged and what are now the paths used to reach *R1* ?
3. Link *R1-R4* comes back. Which BGP messages are exchanged and what do the paths used to reach *R1* become ?

8 On February 22, 2008, the Pakistan Telecom Authority issued an order to Pakistan ISPs to block access to three IP addresses belonging to youtube: *208.65.153.238*, *208.65.153.253*, *208.65.153.251*. One operator noted that these addresses were belonging to the same */24* prefix. Read http://www.ripe.net/news/study-youtube-hijacking.html to understand what happened really.

1. What should have done youtube to avoid this problem ?
2. What kind of solutions would you propose to improve the security of interdomain routing ?

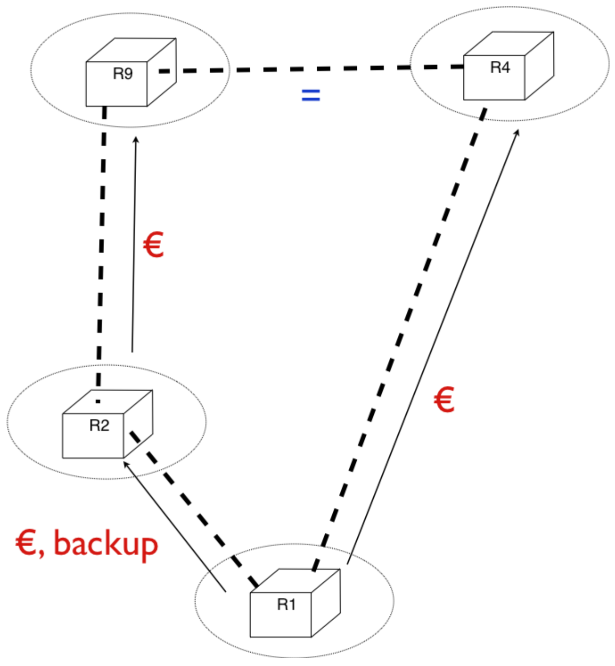


Figure 9.7: A simple internetwork with a backup link

CHAPTER

TEN

BGP AND IP OVER ETHERNET

In this set of exercises, we will discuss about the BGP decision process and analyse in more details the operation of IP in an Ethernet network. For each exercise, try to provide a detailed justification for your answer.

The deadline for this exercise is Tuesday December 8th, 13.00.

10.1 BGP

1. In the network shown below, draw the iBGP and eBGP sessions that must be established among the routers.

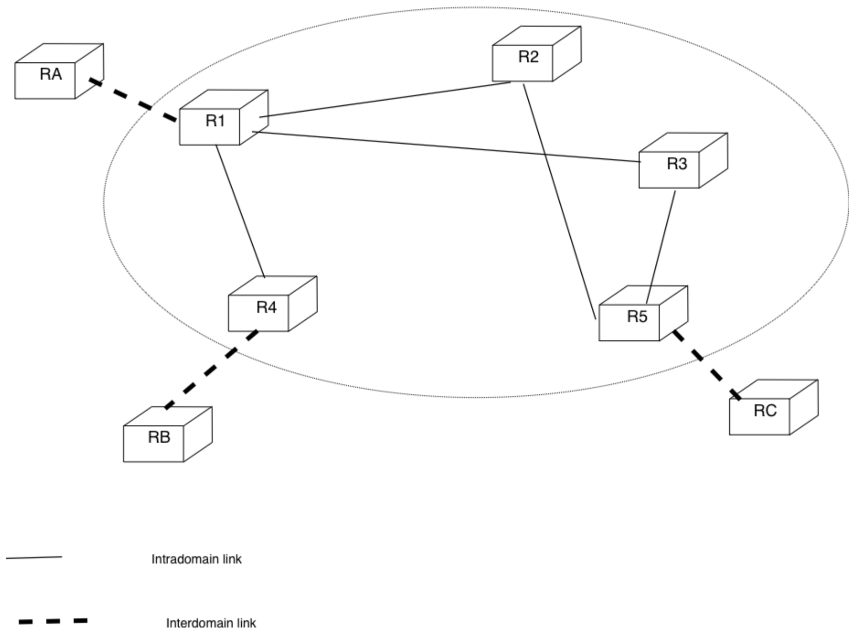


Figure 10.1: A small domain

1. Consider the network shown below. Assume that *R3* and *R5* use the *MED* attribute when advertising a route towards *R1* and *R2*. The *MED* attribute that they advertise is the IGP cost to their nexthop.
   1. Consider that *R5* is attached to prefix *p1* and advertises this prefix. What are the routes received and chosen by *R1* and *R2* ?
   2. Consider that *R9* is attached to prefix *p2* and advertises this prefix. What are the routes received and chosen by *R1* and *R2* ?
   3. Consider that *R6* is attached to prefix *p3* and advertises this prefix. What are the paths received and chosen by *R1* and *R2* ?

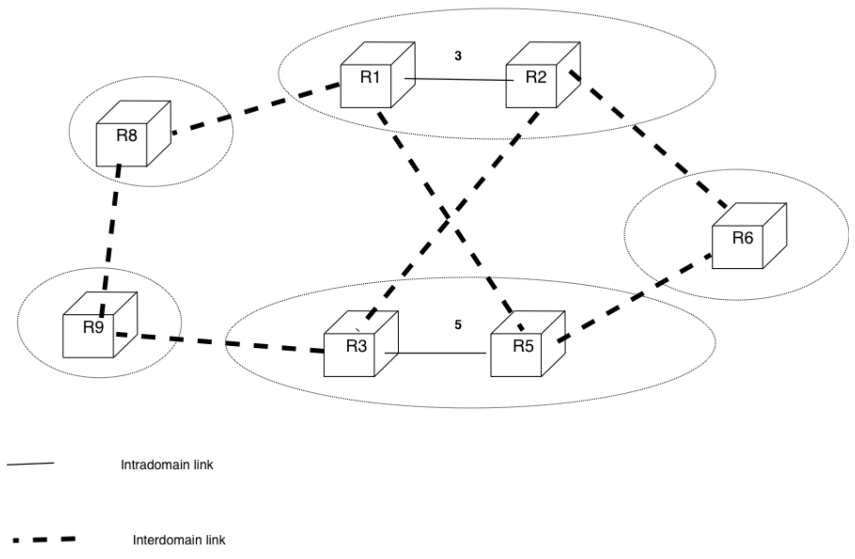


Figure 10.2: A small internetwork

1. Consider the network shown below. In this network, *RA* advertises prefix *p*. Shown the iBGP sessions in this network and all the BGP messages that are exchanged when :

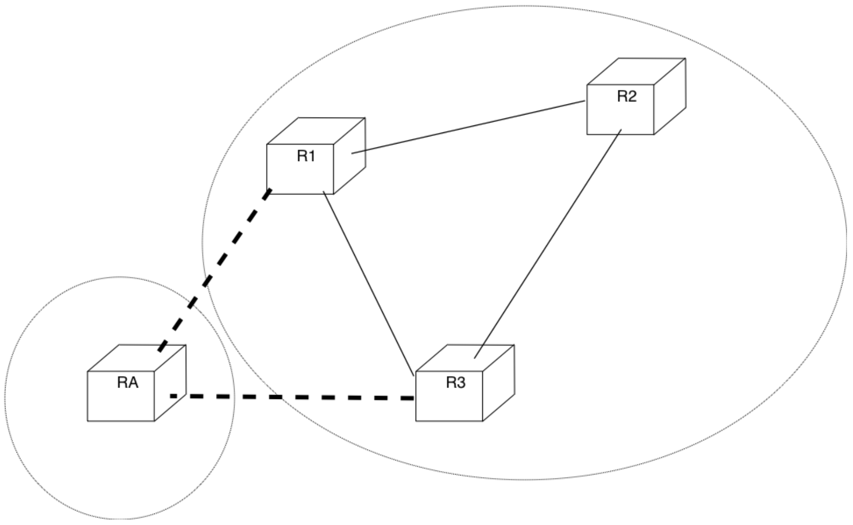


Figure 10.3: A small internetwork

- *RA* advertises *p* first on link *RA-R1* and later on link *RA-R3*
- Link *RA-R3* fails
- Perform the same analysis again by considering now that *R3* inserts a *local-pref* value of *100* in the routes received from *RA* while *R1* inserts a *local-pref* value of *50*.

1. Consider now the network shown below where the link costs are specified. Compute the routing tables on all routers when :

- prefix *p1* is advertised by *RA*, *RB* and *RC* with an AS Path of length 1
- prefix *p2* is advertised by *RA* with an AS Path length of 2 and by *RB* and *RC* with an AS Path of length 1
- What happens in these two situations when the link *R3-R5* fails ?

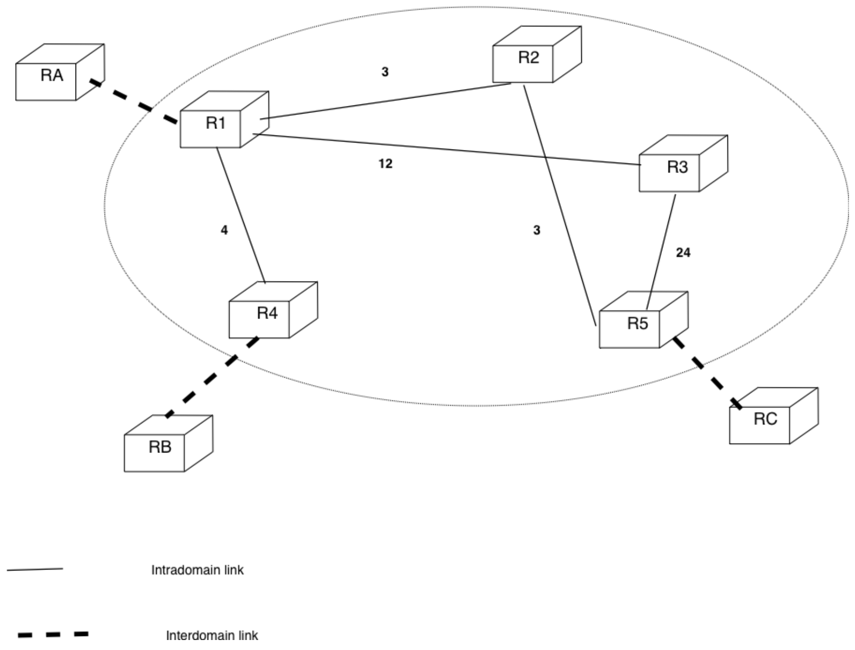


Figure 10.4: A small internetwork

10.2 IP over Ethernet

1. Consider the network shown below. In this network, write the routing tables that need to be configured on all routers. Assume that host *154.112.3.5* sends an IP packet towards host *154.112.0.15* and that this host sends a reply. Show all Ethernet frames that are exchanged. Provide your answer in a table such as the following that lists for each frame :
   - the link on which the frame is sent
   - the Ethernet source and destination addresses
   - the IP source and destination addresses (if any)
   - indicate in the explanation any change to the state of ARP tables on hosts and routers and port/station tables on switches

Link	Ethernet source	Ethernet dest.	IP source	IP dest.	Explanation
...	...	...	...	...	...

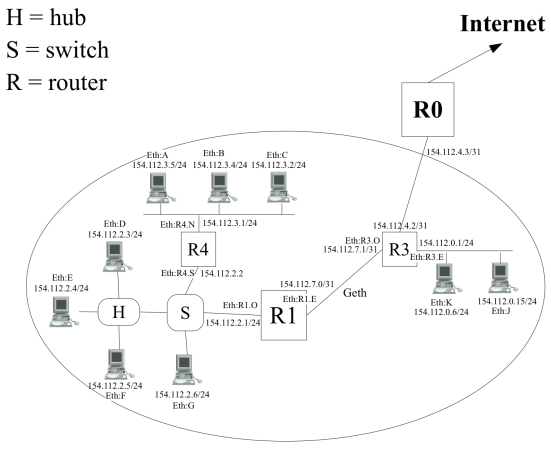


Figure 10.5: A small enterprise network

1. Many TCP/IP implementations today send an ARP request for their own IP address before sending their first IP packet using this address. Can you explain why this is useful in practice ?
2. Consider now the transmission of IPv4 packets. One security issue with IPv4 is the Address Resolution Protocol (ARP). Can you explain what ARP spoofing or ARP poisoning is and show how host *A* in the network below could intercept all the IP packets sent by host *B* via the default router ?

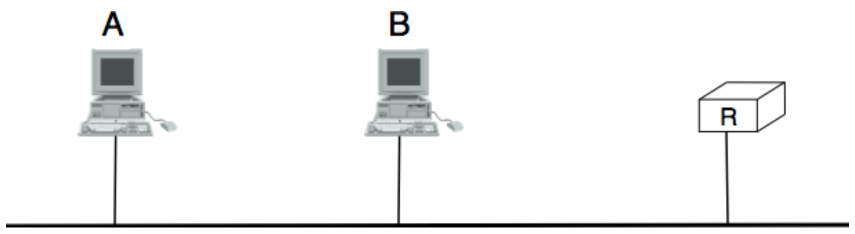


Figure 10.6: A small Ethernet

1. Same question as above, but now consider that the Ethernet network is not a coaxial cable but an Ethernet switch. Is it possible to do something on the switches against these ARP spoofing attacks ? If so how ?

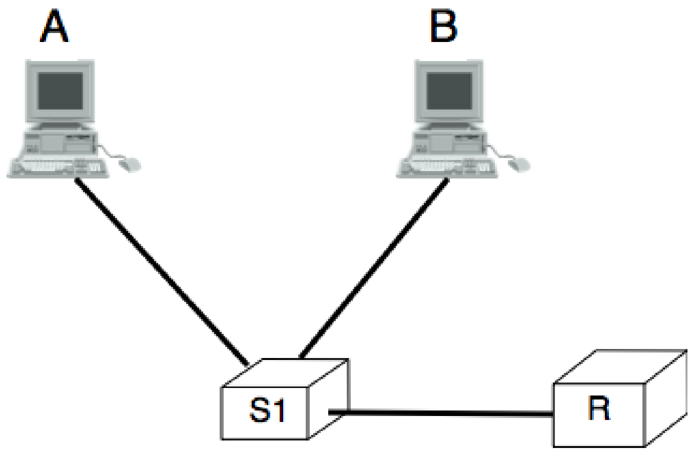


Figure 10.7: A small switched Ethernet

CHAPTER

ELEVEN

INDICES AND TABLES

* *Index*
* *Search Page*

INDEX