Python – Cours 4

Programmation réseau

Jean-Yves Thibon

IR3 2008–2009

1. Protocoles supportés nativement
   - Programmation web : HTTP
   - Courrier : SMTP, POP3, IMAP4 et NNTP
   - Protocoles basiques : TELNET et FTP
2. Socquettes
   - Modules
   - Exemples
     - Serveur d'écho
     - Version concurrente
     - Un client SNTP
     - Socquettes brutes : exemple ICMP
     - Accès couche 2 : une requète ARP
3. Bibliothèques de manipulation de paquets
   - dpkt
   - Impacket
   - Scapy

Modules pour HTTP et programmation web

- `BaseHTTPServer` : Serveur HTTP basique (classe dont dérivent `SimpleHTTPServer` et `CGIHTTPServer`)
- `cgi` : Utilitaires pour les scripts CGI
- `CGIHTTPServer` : Serveur de scripts CGI
- `Cookie, cookielib` : Gestion des cookies
- `htmllib, HTMLParser` : Traitement de documents HTML
- `httplib (2.6)/http.client (3.0)` : Client HTTP et HTTPS (bas niveau)
- `urllib, urllib2` : Client, ouvre n'importe quelle URL
- `urlparse, webbrowser ...`

Modules pour le courrier électronique

- SMTP
  - `smtpd` : Serveur SMTP
  - `smtplib` : Client SMTP
- POP3
  - `poplib` : Client POP3
- IMAP4
  - `imaplib` : Client IMAP4
- NNTP
  - `nntplib` : Client NNTP

Protocoles basiques

- TELNET
  - `telnetlib` : Client TELNET
- FTP
  - `ftplib` : Client FTP

Modules de bas niveau

Les modules précédents utilisent

- `socket` : interface bas niveau
- `SocketServer` : un cadre pour la programmation des serveurs
- `ssl` : Secure Socket Layer

La classe fondamentale est `socket.socket`, qui retourne un objet du type Socket :

```
socket([family[, type[, proto]]]) -> socket object
```

Les paramètres les plus courants sont

```
family=socket.AF_INET, type=socket.SOCK_STREAM
                            socket.SOCK_DGRAM
```

On peut accéder à la couche 2, mais c'est mal documenté (voir plus loin).

Sockets : étymologie et orthographe

Wikipédia prétend : francisation de socket ; ancien français : sochet, soket (petit soc de charrue)
Un article mieux informé (par Jacques Ardoino) donne :
Socle apparaît dans notre langue, vers 1674, emprunté à l'italien zoccolo (sabot), lui même dérivé du latin socculus, diminutif de soccus (sorte de pantoufle portée par les femmes et par les hommes à la maison, puis chaussures basses qu'utilisaient les acteurs comiques - ce dernier sens se retrouvera encore dans les mots français socque et socquette)...
On a pu croire, dans le passé, que le « soc » (pièce métallique tranchante d'une charrue) gardait quelque parenté avec le latin soccus et le français socque, dans la mesure ou la partie métallique vient littéralement « chausser » la pièce de bois qui en constitue le support. Mais cette supposition est maintenant tombée en désuétude : soc semblant dériver du gaulois et socle du latin.

On écrira donc **socquette**, comme dans le titre.

Serveur d'écho

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('',8888)) # '': toutes les interfaces disponibles
sock.listen(5)

try:
    while True:
        newSocket, address = sock.accept()
        print "Connected from", address
        while True:
            receivedData = newSocket.recv(1024)
            if not receivedData: break
            newSocket.sendall(receivedData)
        newSocket.close()
        print "Disconnected from", address
finally:
    sock.close()
```

Exemple de client

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect(('193.55.63.80',8888))
print "Connected to server"
data = """Le cheval de mon cousin
ne mange du foin que le dimanche."""
for line in data.splitlines():
    s.sendall(line)
    print "Sent:", line
    response = s.recv(1024)
    print "Received", response
s.close()
```

Résultat

```
jyt@monge ~/python $ python echoserv.py
Connected from ('82.225.166.14', 32801)
Disconnected from ('82.225.166.14', 32801)

#####################################
[jyt@liszt Python]$ python echocli.py
Connected to server
Sent: Le cheval de mon cousin
Received Le cheval de mon cousin
Sent: ne mange du foin que le dimanche.
Received ne mange du foin que le dimanche.
```

Commentaires

- Les fonctions et les constantes ont généralement les mêmes noms qu'en C
- Mais les nombres de paramètres et les valeurs de retour ne sont généralement pas les mêmes
- On ne trouve pas tout dans la doc de Python (cf. `man socket`)
- Et toutes les constantes ne sont pas définies (`PF_INET` n'est pas définie, alors qu'elle devrait être égale à `AF_INET`)
- Cela dit, on peut reprendre en Python les exercices standards du cours de réseau
- L'interpréteur permet d'expérimenter facilement

Serveur d'écho (version concurrente)

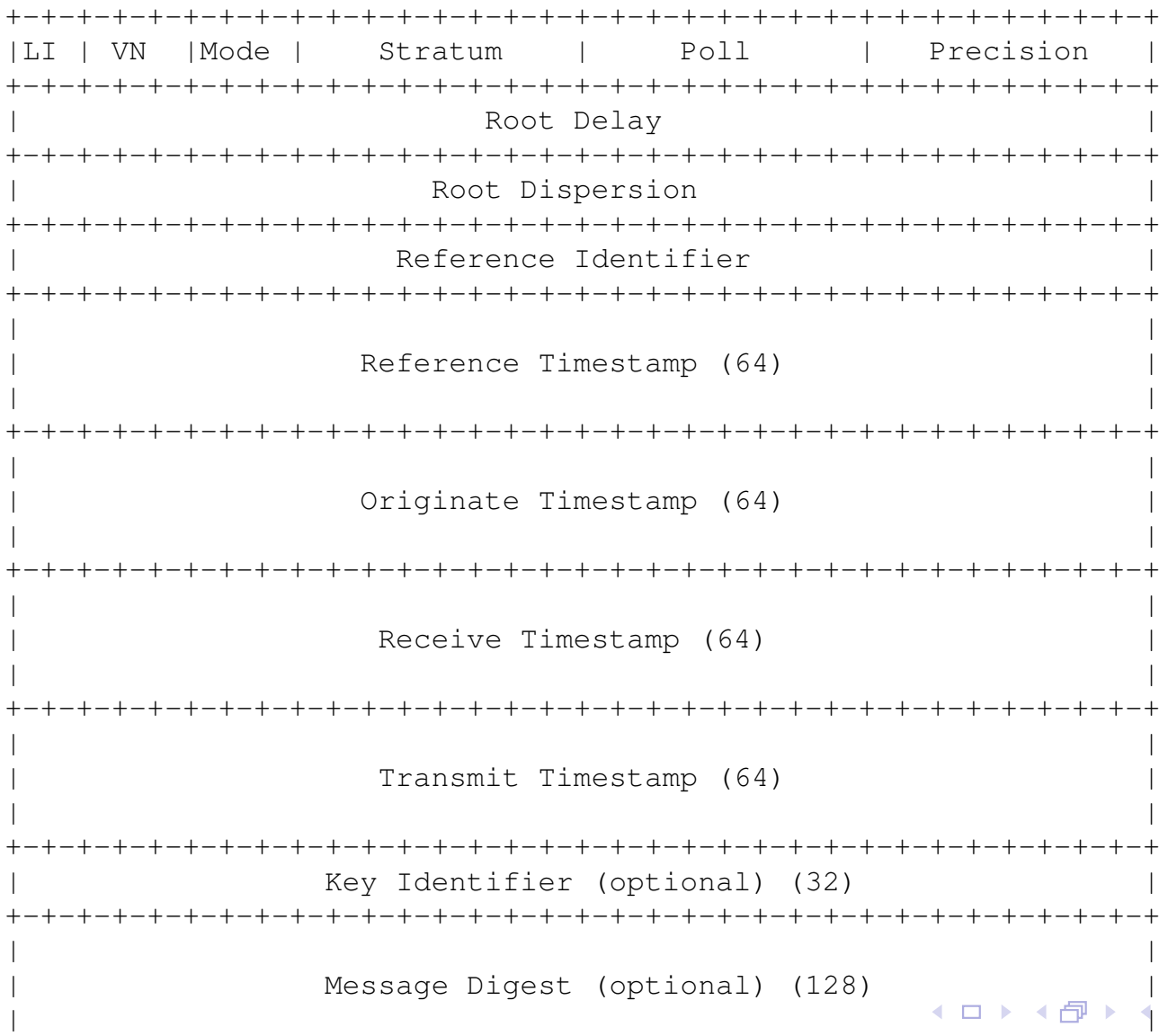
```
import thread, time
from socket import *
Host = ''
Port = 8888

s = socket(AF_INET, SOCK_STREAM)
s.bind((Host, Port))
s.listen(5)

def now():
    return time.ctime(time.time())

def handleClient(connection):
    time.sleep(5)
    while 1:
        data = connection.recv(1024)
        if not data: break
        connection.send('Echo=>%s at %s' % (data, now()))
    connection.close()

def dispatcher():
    while 1:
        connection, address = s.accept()
        print 'Server connected by', address,
        print 'at', now()
        thread.start_new(handleClient, (connection,))
```

Un client SNTP

```
import struct, sys
from socket import *
from time import ctime

TIME1970 = 2208988800 # sec depuis 01/01/1900 00:00

if len(sys.argv) < 2:
    srv = '150.254.183.15'
else:
    srv = sys.argv[1]

s = socket(AF_INET, SOCK_DGRAM)
data = '\x1b' + '\0'*47
s.sendto(data, (srv, 123))
data, addr = s.recvfrom(1024)
if data:
    print "Received from: ", addr
    try:
        t = struct.unpack('!12I', data)[10]
        t -= TIME1970
        print '\tTime = %s' % ctime(t)
    except: print data
```

Commentaires

- 123 = port SNTP (Simple Network Time Protocol)
- On interroge le serveur en envoyant un datagramme de 48 octets commençant par `0x1b`
- Il renvoie 48 octets (12 mots de 32 bits), le 11ème contient le nombre de secondes depuis le 1er janvier 1900, 0 h.
- Le 12ème donne les microsecondes (cf. transparent suivant)
- On le décode au moyen du module `struct`
- La RFC 2030 décrit les différents champs
- la chaîne `'!12I'` décode 12 entiers longs non signés (I) en big-endian (! = network order)


LI
VN
Mode
Stratum
Poll
Precision
Root Delay
Root Dispersion
Reference Identifier
Reference Timestamp (64)
Originate Timestamp (64)
Receive Timestamp (64)
Transmit Timestamp (64)
Key Identifier (optional) (32)
Message Digest (optional) (128)

Socquettes brutes (raw sockets)

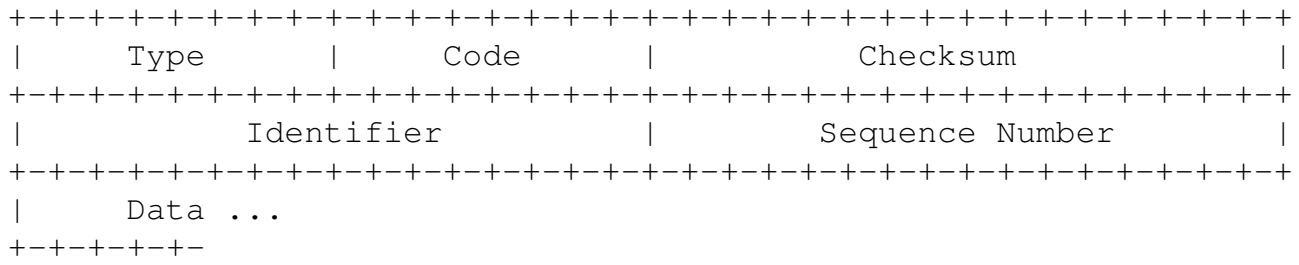
Permettent l'implantation de protocoles de plus bas niveau.
Exemple : ICMP – envoi d'une demande d'écho (cf. `ping`, `traceroute`)

```
sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)
# On positionne les options avec setsockopt
# SOL = Socket Option Level, SO = Socket Option
sd.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)    # autorise l'adresse
                                              # de diffusion (ping -b)
sd.setsockopt(SOL_SOCKET, SO_RCVBUF, 60*1024) # augmente la taille du
                                              # tampon en consequence

sd.settimeout(5) # les socquettes sont bloquantes par defaut
```

Construction manuelle d'un paquet

```
class Icmp_ER():                          # Echo request
    def __init__(self,ident,seqnum):      #
        self.id = ident                   # identifiant (ex.: os.getpid())
        self.seq = seqnum                 #
        self.type = '\x08'                # type ECHO REQUEST
        self.code = '\x00'                # seul code possible ici
        self.chks = 0                     # checksum

    def __str__(self):                    # assemble le paquet
        tc = struct.pack('!cc', self.type,self.code) #
        idseq = struct.pack('!HH', self.id, self.seq)#
        s = checksum(tc + idseq)                     # calcule la
        self.chks = s                                # checksum
        return tc + struct.pack('!H',s)+idseq        #
```

Exemple



Par exemple (`checksum` est définie plus loin)

```
>>> p=Icmp_ER(1,2)
>>> s=str(p)
>>> s
'\x08\x00\xf7\xfc\x00\x01\x00\x02'
>>> struct.unpack('!ccHHH',s)
('\x08', '\x00', 63484, 1, 2)
>>> checksum(s[:2]+s[4:])
63484
>>> checksum(s)
0
```

Le reste du code

```
import struct, os, sys, array
from socket import *

def checksum(pkt):
    if len(pkt) % 2 == 1:
        pkt += "\0"
    s = sum(array.array("H", pkt))
    s = (s >> 16) + (s & 0xffff)
    s += s >> 16
    s = ~s
    return (((s>>8)&0xff)|s<<8) & 0xffff

PID = os.getpid()              # Par exemple ..
SEQ = 0                        # idem
srv = ('192.168.2.1', 0)       # Port 0 sans importance
data = str(Icmp_ER(PID,SEQ))   # assemblage du paquet
sd.sendto(data, srv)
# Pour tester:
data, addr = sd.recvfrom(1500)
x =(len(data),)+addr           # On doit recevoir 28 octets
print "Received %d bytes from %s, port %d" % x
```

Accès couche 2 : une requète ARP

On peut accéder au niveau 2 (ex. ethernet) avec les socquettes brutes de la famille `PF_PACKET`

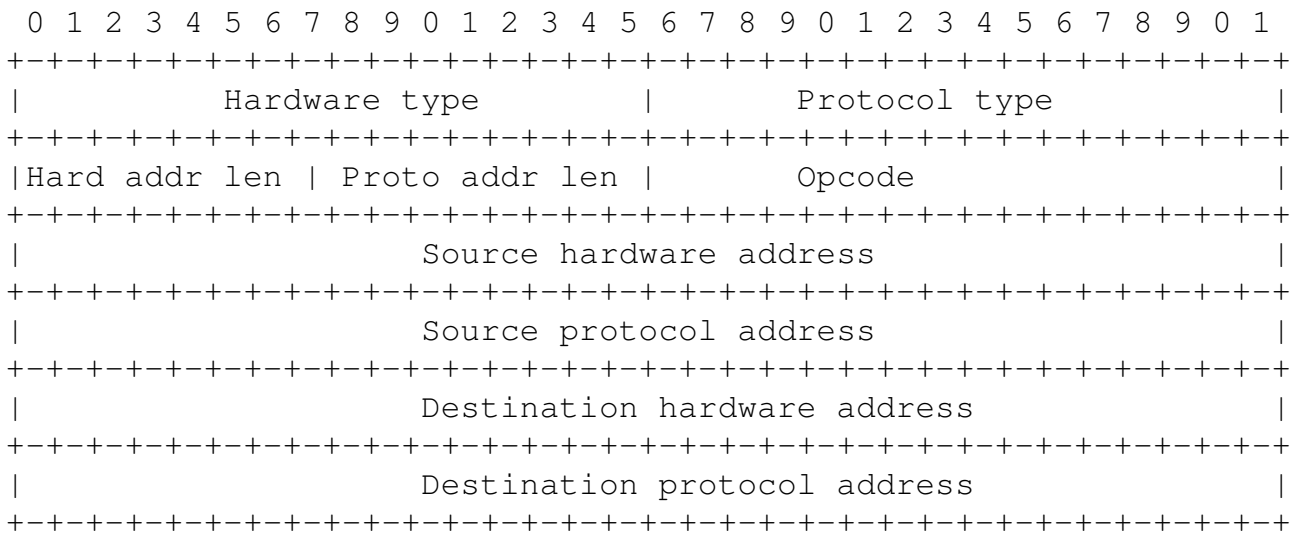
```
sd = socket(PF_PACKET, SOCK_RAW)
sd.bind(('eth1', 0x806)) # 0x806 = paquets ARP
```

On va le tester avec une requète ARP (who-has). La documentation n'indique pas clairement s'il faut inclure l'en-tête ethernet. En expérimentant, on voit qu'il le faut ...

```
data = ether + arp
sd.send(data)
ans = sd.recv(1024)
```

Reste à construire les chaînes `arp` et `ether`.

L'en-tête ARP


0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Hardware type
Protocol type
Hard addr len
Proto addr len
Opcode
Source hardware address
Source protocol address
Destination hardware address
Destination protocol address

L'en-tête ETHERNET

C'est le plus simple, il montre la voie à suivre ... Dans `/usr/src/linux/include/linux/if_ether.h`, on trouve

```
struct ethhdr
{
        unsigned char   h_dest[ETH_ALEN];       /* destination eth addr */
        unsigned char   h_source[ETH_ALEN];     /* source ether addr    */
        unsigned short  h_proto;                /* packet type ID field */
} __attribute__((packed));
```

(`ETH_ALEN` vaut 6), d'où en Python, quelque chose du genre

```
class Ether():
    def __init__(self,dhw,shw, ptype):
        self.shw = shw      # ex:
        self.dhw = dhw      #'ff:ff:ff:ff:ff:ff'
        self.type = ptype   # ex: 0x806
    def __str__(self):
        return (mac2str(self.dhw)
                + mac2str(self.shw) + struct.pack('!H',self.type))
```

où `mac2str` fait ce qu'on imagine.

Une classe pour les requêtes ARP

On pourrait définir tout de suite la classe `Arp()` mais pour faire simple on va préremplir les attributs correspondant à une requête.

```
class Arp_who_has():
    def __init__(self, hw_src, ip_src, hw_dst, ip_dst):
        self.hwdst = mac2str(hw_dst)
        self.hwsrc  = mac2str(hw_src)
        self.ipsrc = ip2str(ip_src)
        self.ipdst = ip2str(ip_dst)
        self.hwtype = 0x1  # ARPHRD_ETHER    1 (Ethernet 10Mbps)
        self.ptype = 0x800 # IP
        self.hwlen= 6
        self.plen = 4
        self.op= 1         # ARPOP_REQUEST   1  (ARP request)

    def __str__(self):
        w = struct.pack('!HHBBH', self.hwtype, self.ptype,
                                  self.hwlen, self.plen, self.op)
        return w + self.hwsrc + self.ipsrc + self.hwdst + self.ipdst
```

Le reste du code

```
import struct, os, sys
from socket import *

ETH_BROADCAST = 'ff:ff:ff:ff:ff:ff'
ETH_UNSPECIFIED = '00:00:00:00:00:00'

def getMacAddress(iface):
    for line in os.popen("/sbin/ifconfig "+iface):
        if line.find('Ether') > -1:
            return line.split()[4]

def ip2str(ip):
    return ''.join([chr(int(i)) for i in ip.split('.')])

def mac2str(mac):
    return ''.join([chr(int(i,16)) for i in mac.split(':')])

def str2ip(s):
    return '.'.join([str(ord(i)) for i in s])

def str2mac(s):
    return '%02x:%02x:%02x:%02x:%02x:%02x' % tuple(map(ord,list(s)))
```

Et un test ...

```
HOST = gethostbyname(gethostname())
MAC = getMacAddress('eth1')

ether = str(Ether(ETH_BROADCAST, MAC, 0x806))
arp = str(Arp_who_has(MAC, HOST, ETH_UNSPECIFIED, '192.168.2.148'))

data = ether + arp
sd.send(data)
ans = sd.recv(1024)

rarp = struct.unpack('!HccBBH6s4s6s4s',ans[14:42])
print '%s is at %s' % (str2ip(rarp[7]), str2mac(rarp[6]))
```

On obtient

```
>>>
192.168.2.148 is at 00:c0:ca:1a:06:75
```

Conclusions

- On peut ...
- Mais c'est pas la peine ...
- ... parce que d'autres l'ont déjà fait (et mieux)
- Par exemple :
  - `dpkt` par Dug Song (`dsniff` etc.)
  - `impacket` de Core Security Technologies
  - `scapy` par Philippe Biondi

Conclusions

- On peut ...
- Mais c'est pas la peine ...
- ... parce que d'autres l'ont déjà fait (et mieux)
- Par exemple :
  - `dpkt` par Dug Song (`dsniff` etc.)
  - `impacket` de Core Security Technologies
  - `scapy` par Philippe Biondi

Conclusions

- On peut ...
- Mais c'est pas la peine ...
- ... parce que d'autres l'ont déjà fait (et mieux)
- Par exemple :
  - `dpkt` par Dug Song (`dsniff` etc.)
  - `impacket` de Core Security Technologies
  - `scapy` par Philippe Biondi

Conclusions

- On peut ...
- Mais c'est pas la peine ...
- ... parce que d'autres l'ont déjà fait (et mieux)
- Par exemple :
  - `dpkt` par Dug Song (`dsniff` etc.)
  - `impacket` de Core Security Technologies
  - `scapy` par Philippe Biondi

dpkt

`http ://code.google.com/p/dpkt/`

Récupérer avec

`svn checkout http://dpkt.googlecode.com/svn/trunk/ dpkt-read-only`

On trouve une classe

```
class Packet(object):
    """Base packet class, with metaclass magic to generate members from
    self.__hdr__.
```

dont dérivent tous les types de paquets.

Requête ARP avec dpkt

Par exemple, `arp.py` contient diverses constantes (du genre `ARP_PRO_IP = 0x0800`) et une classe traduisant la définition de `struct arphdr` dans `if_arp.h` (avec les paramètres pour une requête) :

```
class ARP(dpkt.Packet):
    __hdr__ = (
        ('hrd', 'H', ARP_HRD_ETH),
        ('pro', 'H', ARP_PRO_IP),
        ('hln', 'B', 6),        # hardware address length
        ('pln', 'B', 4),        # protocol address length
        ('op', 'H', ARP_OP_REQUEST),
        ('sha', '6s', ''),
        ('spa', '4s', ''),
        ('tha', '6s', ''),
        ('tpa', '4s', '')
        )
```

Nouvelle version

```
import struct, sys, os
from socket import *
from dpkt import ethernet, arp  # seul changement jusqu'ici
def getMacAddress(iface):
    for line in os.popen("/sbin/ifconfig "+iface):
        if line.find('Ether') > -1:
            return line.split()[4]

def ip2str(ip):
    return ''.join([chr(int(i)) for i in ip.split('.')])

def mac2str(mac):
    return ''.join([chr(int(i,16)) for i in mac.split(':')])

def str2ip(s):
    return '.'.join([str(ord(i)) for i in s])

def str2mac(s):
    return '%02x:%02x:%02x:%02x:%02x:%02x' % tuple(map(ord,list(s)))

ETH_BROADCAST = 'ff:ff:ff:ff:ff:ff'
ETH_UNSPECIFIED = '00:00:00:00:00:00'
HOST = gethostbyname(gethostname())
MAC = getMacAddress('eth1')
```

```
ar = arp.ARP()                      # construction des en-etes
ar.sha = mac2str(MAC)               # plus simple
ar.tha = mac2str(ETH_UNSPECIFIED) # toutes les constantes
ar.spa = ip2str(HOST)               # difficiles a trouver
ar.tpa = ip2str( '192.168.2.148') # sont predefinies
eth = ethernet.Ethernet()
eth.src = mac2str(MAC)
eth.dst = mac2str(ETH_BROADCAST)
eth.data = ar
eth.type = ethernet.ETH_TYPE_ARP

sd = socket(PF_PACKET, SOCK_RAW) # mais on doit encore
sd.bind(('eth1', 0x806))         # se debrouiller
                                 # avec les socquettes
data = str(eth) + str(ar)        # en particulier, comprendre
sd.send(data)                    # qu'on doit contruire
ans = sd.recv(1024)              # l'en-tete ethernet ...
r = struct.unpack('!HccBBH6s4s6s4s',ans[14:42]) # ... et decoder
print '%s is at %s' % (str2ip(r[7]), str2mac(r[6]))
```

Impacket I

Développé par Core Impact Technologies :
`http://oss.coresecurity.com/projects/impacket.html`
Assemblage de paquets et décodage. Utilisation avec `Pcapy` recommandée (interface Python/libpcap, aussi par Core Impact).

```
from socket import *
from impacket import ImpactDecoder, ImpactPacket
arp = ImpactPacket.ARP()
arp.set_ar_hln(6)
arp.set_ar_pln(4)
arp.set_ar_op(1)
arp.set_ar_hrd(1)
arp.set_ar_spa((192, 168, 2, 171))
arp.set_ar_tpa((192, 168, 2, 148))
arp.set_ar_sha((0x00, 0x0f, 0xea, 0xaf, 0x79, 0x15))
arp.set_ar_pro(0x800)
```

Impacket II

```
eth = ImpactPacket.Ethernet()
eth.contains(arp)
eth.set_ether_shost((0x00, 0x0f, 0xea, 0xaf, 0x79, 0x15))
eth.set_ether_dhost((0xff, 0xff, 0xff, 0xff, 0xff, 0xff))
```

Même principe que dans `dpkt` et les exemples forgés à la main. On doit encore gérer les socquettes.

```
sd = socket(PF_PACKET, SOCK_RAW)
sd.bind(('eth1', 0x806))
sd.settimeout(2)
sd.send(eth.get_packet())
ans = sd.recv(1024)
```

Impacket III

La suite est plus simple avec le module `ImpactDecoder`.

```
reth = ImpactDecoder.EthDecoder().decode(ans)
print reth # juste pour voir

rarp = reth.child()
print rarp # c'est comme on pense

fmt = '%d.%d.%d.%d is at %02X:%02X:%02X:%02X:%02X:%02X'
# mais la deniere ligne suffit
print fmt % (tuple(rarp.get_ar_spa())+tuple(rarp.get_ar_sha()))
```

Impacket IV

Les objets `reth` et `rarp` sont des paquets, `print` imprime leur `str`, et on accède aux champs intéressants par `get_ar_XXX` :

```
Ether: 0:c0:ca:1a:6:75 -> 0:f:ea:af:79:15
ARP format: ARPHRD ETHER opcode: REPLY
0:c0:ca:1a:6:75 -> 0:f:ea:af:79:15
192.168.2.148 -> 192.168.2.171
0000 0000 0000 0000 0000 0000 0000 bf86    ................
8550                                       .P
ARP format: ARPHRD ETHER opcode: REPLY
0:c0:ca:1a:6:75 -> 0:f:ea:af:79:15
192.168.2.148 -> 192.168.2.171
0000 0000 0000 0000 0000 0000 0000 bf86    ................
8550                                       .P
192.168.2.148 is at 00:C0:CA:1A:06:75
```

Scapy I

Développé par Philippe Biondi : `http://www.secdev.org/`
Beaucoup plus puissant que les précédents. L'exemple ARP se résume à

```
>>> a=ARP()
>>> a.pdst='192.168.2.148'
>>> b=Ether()
>>> b.src=a.hwsrc
>>> ans, unans = srp(b/a)
Begin emission:
*Finished to send 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
>>> print '%s is at %s'%(ans[0][1].payload.psrc,
                         ans[0][1].payload.hwsrc)
192.168.2.148 is at 00:c0:ca:1a:06:75
```

Scapy II

Examinons les détails.

```
[root@liszt scapy]# scapy
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.0.0.11 beta)
>>> a=ARP()
>>> a.show()
###[ ARP ]###
  hwtype= 0x1
  ptype= 0x800
  hwlen= 6
  plen= 4
  op= who-has
  hwsrc= 00:0f:ea:af:79:15
  psrc= 192.168.2.171
  hwdst= 00:00:00:00:00:00
  pdst= 0.0.0.0
>>> a.pdst='192.168.2.148'
>>> b=Ether()
>>> b.show()
###[ Ethernet ]###
WARNING: Mac address to reach destination not found. Using broadcast.
  dst= ff:ff:ff:ff:ff:ff
  src= 00:00:00:00:00:00
  type= 0x0
```

Scapy III

```
>>> b.src=a.hwsrc
>>> ans, unans = srp(b/a)
Begin emission:
*Finished to send 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
>>> ans
<Results: TCP:0 UDP:0 ICMP:0 Other:1>
>>> ans[0]
(<Ether  src=00:0f:ea:af:79:15 type=0x806 |<ARP  pdst=192.168.2.148 |>>,
 <Ether  dst=00:0f:ea:af:79:15 src=00:c0:ca:1a:06:75 type=0x806 |
 <ARP  hwtype=0x1 ptype=0x800 hwlen=6 plen=4 op=is-at hwsrc=00:c0:ca:1a:06:75
  psrc=192.168.2.148 hwdst=00:0f:ea:af:79:15 pdst=192.168.2.171 |
 <Padding  load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xbf\x86\x85P' |>>>)
>>> ans[0][1].payload
<ARP  hwtype=0x1 ptype=0x800 hwlen=6 plen=4 op=is-at hwsrc=00:c0:ca:1a:06:75
 psrc=192.168.2.148 hwdst=00:0f:ea:af:79:15 pdst=192.168.2.171 |
 <Padding  load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xbf\x86\x85P' |>>
>>> print '%s is at %s' % (ans[0][1].payload.psrc, ans[0][1].payload.hwsrc)
192.168.2.148 is at 00:c0:ca:1a:06:75
```

Scapy IV

Scapy connaît un grand nombre de protocoles, et définit une classe pour chaque type de paquet (même logique que précédemment). Les instances sont créées avec des valeurs par défaut et sont dès le début des paquets valides.
On visualise les attributs avec la méthode `show()` et on les modifie à volonté.
On peut ensuite empiler les protocoles avec l'opérateur / :

```
>>> a=TCP()
>>> b=IP()
>>> c=Ether()
>>> p = c/b/a
>>> p
<Ether  type=0x800 |<IP  frag=0 proto=tcp |<TCP  |>>>
```

Scapy V

On accède aux différentes couches avec une syntaxe de type dictionnaire, ou avec l'attribut `payload` :

```
>>> p[IP].dst = '192.168.2.148'
>>> p
<Ether  type=0x800 |<IP  frag=0 proto=tcp dst=192.168.2.148
                                                |<TCP  |>>>
>>> p[TCP]
<TCP  |>
>>> p.payload
<IP  frag=0 proto=tcp dst=192.168.2.148 |<TCP  |>>
>>> p.payload.payload
<TCP  |>
>>> p.haslayer(TCP)
1
>>> p.haslayer(ARP)
0
```

Scapy VI

Par exemple, l'interrogation du serveur de temps pourrait se faire avec

```
>>> p = IP(dst='150.254.183.15')/UDP(sport=11111, dport=123)/('\x1b' + '\0'*47)
>>> x,y = sr(p)
```

On n'a plus à gérer les socquettes. Les paquets sont envoyés par la fonction `send` (couche 3) ou `sendp` (couche 2).
Si on attend une réponse, on utilise `sr, srp, sr1, srp1`. Ces fonctions retournent un couple de listes `(ans, unans)`. La première est une liste de couples (stimulus, réponse). La seconde contient les paquets sans réponse.

```
>>> x
<Results: TCP:0 UDP:1 ICMP:0 Other:0>
>>> x[0][1][UDP][Raw].load
'\x1c\x01\x00\xf0  ... [snip] ...\xcdB\xa3\xfd\xdbJ\xba8'
```

Scapy VII

Chaque champ d'un paquet peut être un ensemble. On crée ainsi un ensemble de paquets ayant toutes les combinaisons de valeurs possibles dans ces ensembles.
Pour détecter les machines ayant un serveur web sur le réseau de classe C contenant le serveur de l'IGM (FR-UMLV-8), on peut par exemple émettre

```
>>> a=IP(dst="igm.univ-mlv.fr/24")/TCP(dport=80)
>>> ans, unans = sr(a,timeout=3)
Begin emission:
.***Finished to send 256 packets.

Received 4 packets, got 3 answers, remaining 253 packets
```

Scapy VIII

La méthode `ans.nsummary()` permet de visualiser les réponses. On peut les filtrer, et choisir ce que l'on veut voir :

```
>>> ans.nsummary(lfilter = lambda(s,r): r[TCP].flags & 2,
                 prn = lambda(s,r): r[IP].src)
0001 193.55.63.80
0002 193.55.63.149
```

La clause `flags & 2` sélectionne les paquets qui ont `SYN = 1`. Ici le butin est modeste, on n'a trouvé qu'un deuxième serveur.

Scapy IX

Un `traceroute -I` (envoie des paquets ICMP) pourrait s'écrire

```
>>> ans,unans=sr(IP(dst='193.55.63.80',
                    ttl=(1,25),
                    id=RandShort())/ICMP(),timeout=5)
Begin emission:
*****************Finished to send 25 packets.
*****
Received 23 packets, got 23 answers, remaining 2 packets
>>> for snd,rcv in ans:
...   print snd.ttl, rcv.src, isinstance(rcv.payload, ICMP)
...
1 192.168.2.1 True
2 82.225.166.254 True
3 78.254.3.126 True
...
```

Scapy X

On obtient la liste des protocoles supportés avec la commande `ls()` :

```
>>> ls()
ARP         : ARP
ASN1_Packet : None
BOOTP       : BOOTP
CookedLinux : cooked linux
DHCP        : DHCP options
... (plus de 300)
```

et la liste des fonctions avec `lsc()` :

```
>>> lsc()
arpcachepoison      : Poison target's cache with (your MAC,victim's IP) couple
arping              : Send ARP who-has requests to determine which hosts are up
bind_layers         : Bind 2 layers on some specific fields' values
corrupt_bits        : Flip a given percentage or number of bits from a string
...
```

Scapy XI

Il y a beaucoup d'outils précodés. Par exemple `sniff`, qui capture le traffic, accepte des filtres et des fonctions de présentation. Le code suivant espionne le courrier, et capture les mots de passe :

```
a=sniff(filter="tcp and (port 25 or port 110)",
        prn=lambda x:
                x.sprintf("%IP.src%:%TCP.sport%
                          -> %IP.dst%:%TCP.dport%
                          %2s,TCP.flags% : %TCP.payload%"))
```

C'est la méthode `sprintf()` qui permet une présentation claire.

Scapy XII

`sprintf(self, fmt, relax=1)` méthode des instances de scapy.layers.inet.IP.
`sprintf(format, [relax=1]) -> str`
où format est une chaîne qui peut inclure des directives.
Une directive commence et finit par % : `%[fmt[r],][cls[:nb].]field%`.
fmt est une directive de printf, "r" est pour "raw substitution" (ex : IP.flags=0x18 au lieu de SA), nb est le numéro de la couche voulue (ex : pour les paquets IP/IP, IP :2.src est src de la couche IP supérieure). Cas particulier : `"%.time%"` est la date de création. Ex :

```
p.sprintf("%.time% %-15s,IP.src% -> %-15s,IP.dst% %IP.chksum% "
                  "%03xr,IP.proto% %r,TCP.flags%")
```

Le format peut inclure des conditionnelles : layer :string string est la chaîne à insérer si layer est présent. Si layer est precedée de "!", le result est inversé. Les conditions peuvent être imbriquées :

```
p.sprintf("This is a{TCP: TCP}{UDP: UDP}{ICMP:n ICMP} packet")
p.sprintf("{IP:%IP.dst% {ICMP:%ICMP.type%}{TCP:%TCP.dport%}}")
```

Pour obtenir `"{"` et `"}"`, utiliser `"%(" et "%)"`.

Scapy XIII

Scapy permet de belles présentations graphiques, s'il est installé avec les dépendances idoines :

- plot() : demande Gnuplot-py, qui demande GnuPlot et NumPy
- graphiques 2D : psdump() pdfdump() demandent PyX
- graphes : conversations() demande Graphviz et ImageMagick
- graphiques 3D : trace3D() demande VPython

Pour une présentation détaillée, voir

`http://www.secdev.org/conf/scapy_pacsec05.pdf`