TP4 : Couche liaison

Gilles Menez - UCA - UFR Sciences - Dépt. Informatique

L' objectif de ce TP est :

① . . . de survivre à une mission délicate.

---

1 La mission . . .

Notre agent 008, en mission pour l'UCA, a réussi à capturer des informations sur un câble Ethernet qui pourraient aider à l'obtention d'une bonne note.

La seule chose certaine est qu'il s'agissait d'un câble Ethernet.

Nous avons besoin de savoir à quoi correspond cet échange d'information :

- ➢ Que contient cette information ?
- ➢ Comment est structurée l'information ?
- ➢ Quels sont les interlocuteurs ?
- ➢ Pourquoi dialoguent t-ils ?
- ➢ . . .

Et de façon générale, toutes les informations et les explications qui permettront, à notre chef incompétent, de choisir la meilleure note.

- ➢ **Attention toutefois** à ne PAS LE PERDRE DANS DES DETAILS INUTILES !

Si vous acceptez votre mission, vous devrez réaliser un programme Python, exécutable sans avoir à être super-utilisateur, et qui permettra d'analyser les différentes informations sous forme synthétique et pédagogique.
N'oubliez pas que le chef n'est pas un agent de terrain !

- ➢ Si il faut ajouter des commentaires dans le code ou dans l'affichage . . . n'hésitez pas !
- ➢ Proposez une conclusion sur la sémantique des ces informations.

1.1 Documents Classifiés :

Nous allons vous confier les informations dans le fichier `XXX.txt`.

De plus, notre expert Qiou, a développé un début de solution qui pourrait peut être vous aider ?

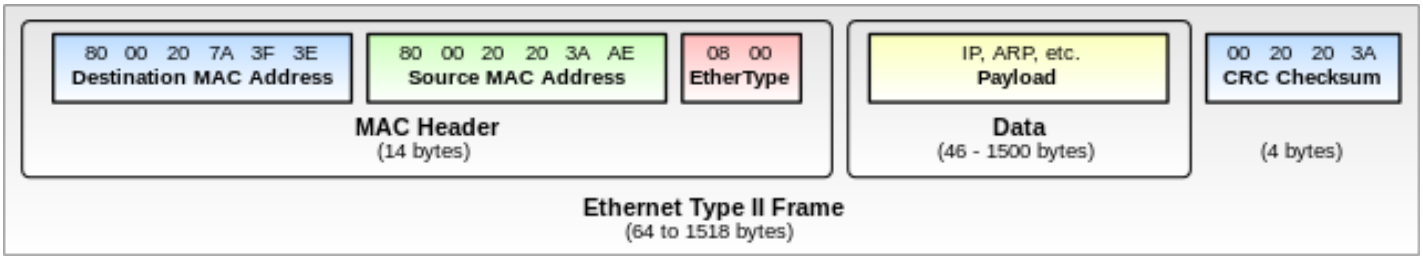
2 Analyse de trames Ethernet ...

2.1 Organisation de la trame Ethernet

Une trame Ethernet a "toujours" la même forme/organisation :

`https://fr.wikipedia.org/wiki/Ethernet`

- ✔ un entête (MAC header)
- ✔ puis les données / la charge (e.g. Data/Payload)



Lors de la capture, on n'a pas gardé la somme de contrôle (CRC checksum) dans le fichier. Ce n'était pas d'une grande utilité puisque si on l'a écrite c'est que la trame était correcte !

Le header de la trame contient un ensemble d'informations qui permettent aux entités Ethernet des différentes machines sur le réseau de dialoguer.

➢ Ces informations font partie du protocole Ethernet.

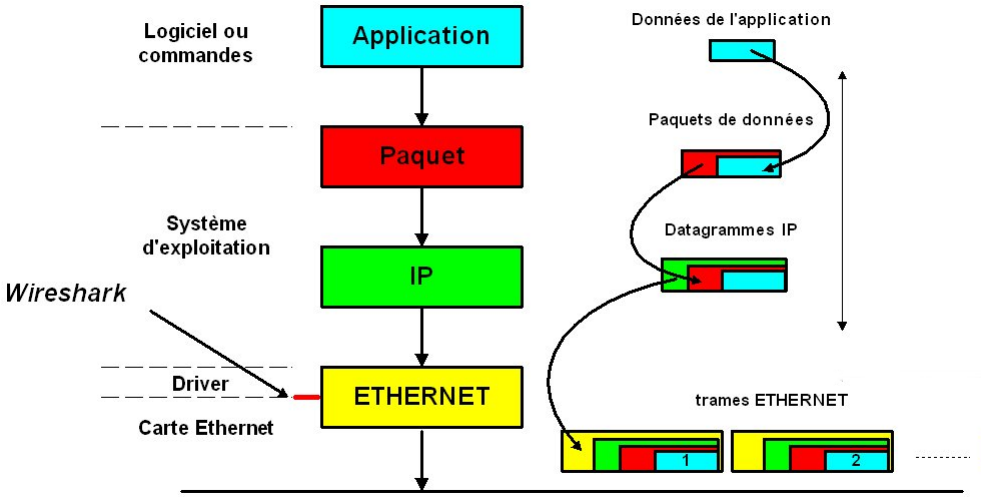
Il faut savoir retrouver ces informations et les analyser pour poursuivre l'analyse de la partie Data.

➢ **TOUTES LES INFORMATIONS** ne sont pas d'égales importances pour votre mission.

➢ **C'est grâce à l'EtherType que l'on sait ce que contient la partie Data** et donc comment aborder l'analyse de cette partie Data.

Cette partie "Data" contient les paquets échangés par les protocoles de niveaux supérieurs (IP, ARP, ...) qui se servent d'Ethernet comme d'un transporteur de charges (Payload).

➢ On appelle cela `l'encapsulation` .



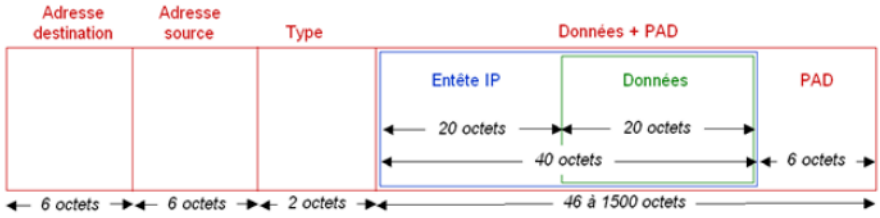
Dans la partie "Data" (en jaune sur la trame Ethernet), on trouvera un header et les datas du protocole de niveau supérieur.

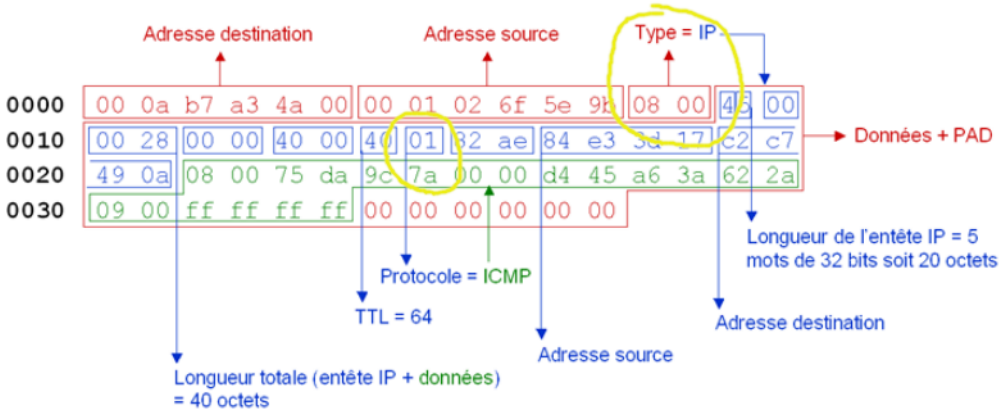
➢ Dans le header du protocole supérieur, il y aura **certainement** une information similaire à Ethertype qui **permettra de comprendre** ce qu'il transporte lui aussi.

On a entouré en jaune ces informations dans l'exemple qui suit, mais qui n'est pas celui que vous rencontrez dans le fichier XXX.txt.

➢ Dans le header d'Ethernet en rouge, le champ "Ethertype" indique que la data/charge vient d'un protocole IP. Du coup on sait comment elle est structurée !

➢ Dans le header d'IP en bleu, le champ "protocole" indique que la charge vient du protocole ICMP.





RMQ : vous remarquez que pour "avancer" dans la trame, vous n'avez pas besoin de comprendre **TOUS** les champs.

2.2 Lire ces trames

L'information dont on dispose, c'est à dire le fichier, contient probablement plusieurs trames échangées entre les participants.

Le premier problème est d'accéder à ces trames. C'est un problème parce que les trames sont réparties sur plusieurs lignes de texte dans le fichier.

➢ On va "assembler" les lignes pour former chaque trame.

➢ Le fichier de texte a une structure syntaxique qui facilite la chose, avec une ligne vide entre chaque trame.

C'est la fonction `readtrames()` qui réalise cela.

2.3 Interpréter ces trames

L'interprétation de ces trames ne peut pas se faire au niveau d'une chaine de caractères car la granularité des informations formant les entêtes des protocoles n'est pas forcément l'octet :

- Les informations sont représentées par des groupements de bits de tailles possiblement différentes de celle d'un caractère (8 bits).

On va donc utiliser quelques fonctions du module `binascii` pour convertir cette chaine de caractère en des objets "bytes-like" qui permettront une interprétation plus fine que la notion de caractère.

Les thèmes suivant seront sans doute utiles :

- Manipulations de caractères
  - `http://stackoverflow.com/questions/227459/ascii-value-of-a-character-in-python`
- Format strings : pack et unpack

  `https://docs.python.org/2/library/struct.html`
- binascii

  `http://stackoverflow.com/questions/32461630/convert-from-mac-address-to-hex-string-and-vice-versa-both-python-2-and-3`

Il FAUT que vous compreniez les manipulations illustrées dans la fonction `manipulation_binascii` du programme qui suit : C'est du python 3 et le code est fourni sur le site !

```
import binascii
def manipulation_binascii():
    """
    Jouons "juste ce qu'il faut" avec les octets ...
    pour marquer la difference entre str et bytes
    """
    print("="*40)
    # En Python, ll n'y a pas de type char comme en C ...
    # => un char est une string de 1 char
    # Deux facons de creer un tel objet :
    c = chr(97) # 97 est le code de 'a'
    print("c = {} de type {}".format(c,type(c)))
    c = 'a'     # est de type 'str' en python
    print("c = {} de type {}".format(c,type(c)))
    v = ord(c) # code ascii de c = i.e representation numerique de c
    # impossible de passer par c[0] comme en C pour acceder
    # a ce nombre/code => faut utiliser ord()
    print("c = {} est code par {} = 0x{:x}".format(c,v,v))

    #=============================================================
    print("="*40)
    s = "aA19"
    # s est codee sur 4 octets
    # for i in [0,3], s[i] vaut le code ascii d'un caractere
    print("s is {} of type {} and len = {} {}".format(s, type(s), len(s), type(s[0])))
    print("Caracteres de la chaine : {}/{}/{}/{}".format(s[0],s[1],s[2],s[3]))
    print("Codes Ascii dans la 'char' chaine : {:x}/{:x}/{:x}/{:x}".format(ord(s[0]),ord(s[1]),ord(s[2]),ord(s[3])))

    #=============================================================
    print("="*40)
    """
    binascii.unhexlify(s)
    Return the binary data represented by the hexadecimal string

    val = unhexlify(s) : supposons que s soit une chaine de caracteres
                         representant une valeur hexa
                         =>  on voudrait recuperer cette valeur ou ces
                             valeurs (selon la taille de s)
    """
    s = "ff00" # une string representant un hexadecimal
    # => Avec une string de 4 chiffres hexa on va obtenir
    # un tableau de 2 octets : on rappelle qu'un octet c'est deux chiffres hexa.
```

```
43      print("s is {} of type {} and len = {}*{}".format(s,type(s),len(s),type(s[0])))
44
45      bs = binascii.unhexlify(s)
46      print("unhexlify(s={}) => bs = {} of type {} and len = {}*{}".format(s,bs,type(bs),len(bs),type(bs[0])))
47      # VOUS REMARQUEZ le petit b a l'affichage ?!
48      # bs est une string d'octets => "bytes object" !
49      # bs est codee sur 2 octets
50      # bs[0] est un octet ... dont le type est int
51      print("bs[0] is {} = {:02x}".format(type(bs[0]),bs[0]))    # octet 0 ecrit en hexa
52      print("bs[1] is {} = {:d}".format(type(bs[1]),bs[1]))      # octet 1 ecrit en decimal
53
54      # Une string raw (avec le b pour "byte") est un tableau d'octets
55      # qui ne representent pas forcement des codes ASCII :
56      # et du coup on peut y mettre des valeurs hors ASCII !
57
58      #============================================================
59      print("="*40)
60      s = "5354" # dans ce cas les octets produits par unhexlify von etre ASCII compatibles
61      bs = binascii.unhexlify(s)
62      print("unhexlify(s={}) => bs = {} of type {} and len = {}*{}".format(s,bs,type(bs),len(bs),type(bs[0])))
63      print("bs[0] is {} = {:02x}".format(type(bs[0]),bs[0]))
64      print("bs[1] is {} = {:02x}".format(type(bs[1]),bs[1]))
65      print("="*40)
66      s = bs.decode("ascii") # pour faire de ce tableau de 2 octets
67                             # une string classique de 2 caracteres
68                             # dont les codes ASCII sont bs[0] et bs[1]
69      print("decode(bs={}) = ASCII interpretation de bs => s = {} ({}/{}*{})".format(bs,s,type(s),len(s),type(s[0])))
70      print("Le b a disparu ! => la string n'est plus raw/bytes !")
71
72      # la conclusion de l'histoire : une chaine de caracteres sur 4
73      # octets representant un nombre hexa peut etre represente comme
74      # une chaine raw/bytes de 2 octets
75
76      #============================================================
77      """
78      hexlify(val)
79      => En sens inverse : de la valeur (tableau de bytes) vers la
80      chaine des caracteres en bytes.
81      => Attention : hexlify() rend une chaine de bytes
82      """
83      print("="*40)
84      bs = b"ST" # Avec cette initialisation, on obtient le tableau de bytes : bs[0]=ord('S')
85      print("bs is {} of type {} and len = {} bytes".format(bs, type(bs), len(bs)))
86      print("{:x}/{:x} == {:x}/{:x}".format(bs[0],bs[1],ord(s[0]),ord(s[1])))
87
88      s = binascii.hexlify(bs)
89      print("hexlify(bs={}) => s = {} of type {} and len = {}*{}".format(bs,s,type(s),len(s),type(s[0]))) #
90      # VOUS REMARQUEZ le petit b a l'affichage ?! s est une string d'octets => "bytes object" !
91      print("Les elements de la 'bytes' chaine s sont les codes ASCII des quartets/nibbles de la chaine bs")
92      print("Caracteres de la 'bytes' chaine s : {}/{}/{}/{}".format(chr(s[0]),chr(s[1]),chr(s[2]),chr(s[3])))
93      print("Code ascii des octets de la chaine s : {:x}/{:x}/{:x}/{:x}".format(s[0],s[1],s[2],s[3]))
94
95
96      #============================================================
97      # s est bien une raw/bytes string de 4 octets
98      ss=b'\x35\x33\x35\x34'
99      print(s==ss)
100
101     #============================================================
102     print("="*40)
103     s = s.decode("ascii")
104     print("decode(bs={}) = ASCII interpretation de bs => s = {} ({}/{}*{})".format(bs,s,type(s),len(s),type(s[0])))
105     print("Le b a disparu ! => la string n'est plus raw/bytes !")
106
107     print("="*40)
108
109 #============================================================
110 if __name__ == '__main__':
111
112     # Pour comprendre la difference entre chaine de caracteres <'str'>
113     # et chaine d'octets <'bytes'>
114     manipulation_binascii()
```

2.4 Analyse sémantique d'une trame

Vous allez devoir interpréter le contenu et la sémantique de trames Ethernet échangées entre des machines.

Le code qui suit vous donne une base pour démarrer ce travail : Le fichier est sur le site Web !

```
'''
Created on 19 avr. 2016
@author: menez

Draft !!!!

'''
import socket
from struct import *
import binascii

#=========================================================

def readtrames(filename):
    """
    Cette fonction fabrique une liste de chaines de caracteres a partir du
    fichier contenant les trames.

    Chaque chaine de la liste rendue est une trame du fichier.


    return : liste des trames contenues dans le fichier
    """
    file = open(filename)
    trames = [] # List of frames (= trames)
    trame = ""  # Current frame .. string vide
    i = 1
    for line in file : # acces au fichier ligne par ligne
        line = line.rstrip('\n') # on enleve le retour chariot de la ligne
        line = line[6:53]        # on ne garde que les colonnes interessantes
        print ("lig {} : {}".format(i,line))
        trame = trame + line

        if (len(line) == 0): # Trame separator
            #print "Ligne vide :", (len(line))
            trame = trame.replace(' ','') # on enleve les blancs

            # Faut que la trame contienne un nombre pair de chiffres pour
            # pouvoir etre "unhexlifier" : 2 chiffres hexa => 1 octets
            if (len(trame) % 2) != 0 :
                trame = trame + "0"
            trames.append(trame) # on ajoute la trame a la liste
            trame = ""        # reset trame

        i = i+1

    if len(trame) != 0 : # Last frame
        trame = trame.replace(' ','') # on enleve les blancs

        # Faut que la trame contienne un nombre pair de chiffres pour
        # pouvoir etre "unhexlifier" : 2 chiffres hexa => 1 octets
        if (len(trame)%2) != 0 :
            trame = trame + "0"
        trames.append(trame) # on ajoute la trame a la liste

    return trames

#=========================================================

def decodageEthernet(trame):
    """
    Analyse une trame Ethernet :  cf https://fr.wikipedia.org/wiki/Ethernet
    Input : trame est une chaine de caracteres

    Je vous donne deux facons d'aborder le probleme !
    """
    print("-"*60)
    print ("\nTrame Ethernet en cours d'analyse : \n{}\n".format(trame))
    print(type(trame))
    trame = binascii.unhexlify(trame) # Les octets representes par cette chaine
    print(type(trame))
```

```
72
73      # Parse ethernet header
74      print ("Header Ethernet :")
75      eth_length = 14
76      eth_header = trame[:eth_length]   # Get the bytes of the header
77      print (type(eth_header))
78
79      # Soit on recupere les adresses MAC en UTILISANT le slicing Python.
80      mac_dest = eth_header[0:6]
81      mac_src = eth_header[6:12]
82      print ('Destination MAC : {}'.format(binascii.hexlify(mac_dest)))
83      print ('Source MAC \t: {}'.format(binascii.hexlify(mac_src)))
84
85      # Soit on recupere les adresses MAC en UTILISANT unpack
86      # For more information on format strings and endiannes, refer to
87      # https://docs.python.org/3.5/library/struct.html
88      ethfields = unpack('!6s6sH' , eth_header)
89      mac_dest = ethfields[0]
90      mac_src = ethfields[1]
91      print ('Destination MAC : {}'.format(binascii.hexlify(mac_dest)))
92      print ('Source MAC \t: {}'.format(binascii.hexlify(mac_src)))
93
94
95      # Maintenant il faut analyser la suite de la trame !!!
96      #
97      #
98
99  #=====================================================================
100
101 if __name__ == '__main__':
102
103     filename = "XXX.txt"
104     filename = "ping_req.txt"
105     filename = "dhcp_req.txt"
106     filename = "httpbin_org.txt"
107     # Transformation des echanges contenus dans le fichier
108     # vers une liste de strings : une string = une trame
109     print("="*60)
110     trames = readtrames(filename)
111
112     print("="*60)
113     print("\nTrames obtenues :\n")
114     for i,t in enumerate(trames):
115         print("trame #{} : {}".format(i,t))
116
117     # Analyse de chaque trame de la liste
118     for trame in trames:
119         decodageEthernet(trame)
```

Ce programme travaille par défaut sur le fichier `XXX.txt` qui contient cinq trames et qui est fourni.

2.5 TODO :

J'espère de vous une amélioration du code qui vous permettra de valider la syntaxe de la trame **jusque dans les protocoles encapsulés**.

Vous en déduirez la sémantique de ces trames et conclurez en expliquant qui communique et pourquoi ?
On espère grâce à votre programme Python voir s'afficher :

- ✔ Adresses physiques des machines
- ✔ Adresses logiques (IP)
- ✔ Nom des protocoles utilisés
- ✔ Information contenue dans les charges des trames du protocole le plus haut.