Relazione del progetto di Programmazione di Reti
Traccia 3: Monitoraggio di Rete

Leonardo Grimaldi

13 giugno 2024

Indice

Capitolo 1

Consegna

Realizzare uno script Python per monitorare lo stato di una rete, controllando la disponibilità di uno o più host tramite il protocollo ICMP (ping). Lo script deve consentire all'utente di specificare gli indirizzi IP degli host da monitorare e deve visualizzare lo stato (online/offline) di ciascun host.

Capitolo 2

Funzionamento

All'avvio dello script `ping.py` verrà chiesto all'utente di inserire in console l'hostname (es. `google.com`) oppure l'indirizzo IP (es. `8.8.8.8`) della macchina della quale si vuole sapere la disponibilità. Qualora arrivi la risposta del destinatario, si otterrà un messaggio del formato: `17 byte da 216.58.204.238: icmp_seq=0 ttl=111 tempo=16 ms`, dove il numero di byte è dato dalla dimensione della sezione DATA (nel nostro caso `DATA = "Buongiorno mondo!"`) e il tempo indica il delay, ovvero quanto ci ha messo a ricevere la risposta dal tempo di invio. Alla fine dell'invio dei `times = 4` pacchetti verrà mostrato lo stato 'Offline'/'Online' dell'host.
Nel caso in cui non si riceva una risposta, a schermo verrà stampato quanti byte sono stati inviati, oppure in casi eccezionali un messaggio di errore. Per terminare correttamente lo script su `Windows` bisogna utilizzare la combinazione di tasti `CTRL + Break`.

Capitolo 3

Codice

Il sorgente è composto da tre sezioni principali necessarie per l'implementazione base del `ping`, elencate e spiegate qui sotto in dettaglio.

3.1 `ICMPchecksum(packet)`

Il codice scritto in questo metodo contiene la logica di calcolo di un checksum ICMP. Prende in input un oggetto `bytes()` e restituisce in output un `int` con soli 16 bit meno significativi utili.

```
1   def ICMPchecksum(packet):
2   temp = packet
3   if len(temp) % 2 != 0:
4       temp += bytes([0])
```

Il pacchetto viene spostato in una variabile di appoggio `temp` per evitare di modificare il parametro in input e causare inconsistenze nel codice.
Successivamente viene eseguito un controllo sulla parità: nel caso in cui il numero di byte del pacchetto è dispari, viene aggiunto alla fine un byte `x00` di padding.

```
5   first = int.from_bytes(temp[0:2], byteorder='big')
6   sum = first
7   for i in range(2, len(temp) - 1, 2):
8       next = int.from_bytes(temp[i:i+2],byteorder='big')
9       sum += next
```

In questa porzione di codice vengono presi 2 byte e convertiti in `int` e sommati con i due successivi finché non si raggiunge la fine.

```
10    overflow = sum >> 16
11    checksum = ~(sum + overflow) & 0xFFFF
12    return checksum
```

Si identifica l'`overflow` prendendo i 16 bit più significativi e lo si aggiunge a `sum`. Si effettua il complemento a 1 di tale valore con il carattere `~` e poi si azzerano i primi 16 bit (ridondanti dato che il checksum è a 16 bit mentre l'intero a 32).
Notare che **il metodo `ICMPchecksum` non azzera in alcun modo il campo 'checksum' di un pacchetto**, sarà infatti compito del chiamante farlo.

3.2 ping(mySocket, destinationHost, identifier, sequenceNumber)

Viene usato per effettuare un messaggio di Echo request a un destinatario. Riceve in input:

- `mySocket`: oggetto `socket` da usare per l'invio del pacchetto
- `destinationHost`: indirizzo IP oppure host name del destinatario
- `identifier`: identificatore del pacchetto da inviare
- `sequenceNumber`: numero di sequenza del pacchetto

```
1   def ping(mySocket, destinationHost, identifier,
    ↪  sequenceNumber):
2   checksum = 0
3   header = struct.pack('!BBHHH', TYPE_ECHO_REQUEST,
    ↪  CODE_ECHO_REQUEST, checksum, identifier, sequenceNumber)
4   # len(data) Ci dà il numero di caratteri nella stringa
5   # esempio: len(data) = 10
6   # struct.pack('!10s', ...) dice quindi di usare 10 byte
7   # encode() usa l'encoding UTF-8 di default.
8   data = struct.pack('!' + str(len(DATA)) + 's', DATA.encode())
9   packet = header + data
```

Prima di inviare il pacchetto è necessario costruirlo: usando la funzione `struct.pack()` è possibile convertire le variabili di Python in oggetti `bytes()` in modo da manipolarli più facilmente. Passando ad essa la format string `'!BBHHH'` si può indicare come verranno rappresentati i dati letti nei successivi parametri:

- `!`: ordine byte network (big endian)
- `B`: unsigned char (1 byte)
- `H`: unsigned short (2 byte)

Elencando successivamente le variabili che formano il pacchetto ICMP (ricordando che il checksum deve essere posto a `x00` prima che venga calcolato), è possibile ottenere una rappresentazione in `byte()` del header. Alla riga `9` viene formato il pacchetto accodando i byte di data su quelli di header.

```
10    chk = ICMPchecksum(packet)
11    header = struct.pack('!BBHHH', TYPE_ECHO_REQUEST,
      ↪  CODE_ECHO_REQUEST, chk, identifier, sequenceNumber)
12    packet = header + data
```

In questa porzione si calcola il checksum del pacchetto e lo si ricostruisce inserendone il valore ottenuto.

```
13    sentTime = time.time()
14    bytesSent = mySocket.sendto(packet, (destIP, 1))
15    return bytesSent, sentTime
```

Infine viene inviato il pacchetto usando `socket.sendto()` e restituito il tempo di invio e il numero di byte inviati

3.3 `receive_reply(mySocket, myID, timeout)`

Questo metodo gestisce l'arrivo del pacchetto di risposta ICMP (Echo reply). In input riceve l'oggetto `socket` su cui leggere i pacchetti, l'identificatore `myID` per verificare che il pacchetto sia quello cercato e il tempo di attesa `timeout`.

```
1     def receive_reply(mySocket, myID, timeout):
2     timeLeft = timeout
3     while True:
4         startedSelect = time.time()
5         # aspetto che il 'mySocket' sia in stato read ovvero
          ↪  ricezione pachetti
6         readable, writeable, exceptional =
          ↪  select.select([mySocket], [], [], timeout)
7         selectTime = (time.time() - startedSelect)
```

Creo un ciclo 'infinito' che dovrà ovviamente fermarsi quando è scaduto il mio `timeout`. Aspetto che il socket sia in modalità lettura e in `selectTime` inserisco il tempo che ci ha messo ad attendere.

```
8   # select timeout
9   if not (readable or writeable or exceptional):
10      return None, None, None, None, None
11  timeReceived = time.time()
12  packet, address = mySocket.recvfrom(ICMP_MAX_RECV)
```

Gestisco il caso di non leggibilità e continuo estraendo il pacchetto ricevuto con `socket.recvfrom()`. Il parametro `ICMP_MAX_RECV` indica la dimensione del buffer, in questo caso 2048 bytes, oltre i quali il dato verrà spezzato.

```
13  ipHeader = packet[:20]
14  (iphVersion, iphTypeOfSvc, iphLength, iphID, iphFlags,
    ↪  iphTTL,
15   iphProtocol, iphChecksum, iphSrcIP, iphDestIP) =
     ↪  struct.unpack("!BBHHHBBHII", ipHeader)
16  icmpHeader = packet[20:28]
17  icmpType, icmpCode, icmpChecksum, \
18  icmpPacketID, icmpSeqNumber = struct.unpack("!BBHHH",
    ↪  icmpHeader)
```

Usando `struct.unpack()` converto i `bytes()` del `packet` in variabili Python che potrò tornare e visualizzare più facilmente.

```
19  if icmpPacketID == myID: # Nostro pacchetto
20    dataSize = len(packet) - 28
21    return timeReceived, dataSize, iphSrcIP, icmpSeqNumber,
       ↪  iphTTL
22  timeLeft = timeLeft - selectTime
23  if timeLeft <= 0:
24    return None, None, None, None, None
```

Confronto l'ID del pacchetto con il mio (quello per cui devo ricevere la risposta): se la condizione è vera ritorno alcune informazioni relative ad esso. Altrimenti, riciclo lo stesso procedimento se non ho ancora superato il tempo `timeout`.