Reti degli Elaboratori

Network Monitor

sessione autunnale 2012/13

Antonio Esposito / 253348
Michele Sorcinelli / 248412

Indice

1. Specifiche del problema....3
2. Scelte progettuali....4
GNU/Linux....4
Python....4
AJAX....5
JSON....5
Versionamento del software....5
Installazione delle dipendenze e avvio del programma....6
3. Implementazione....8
addresses.py – gestione degli indirizzi....8
scan.py – interfaccia ad alto livello a python-nmap....8
host.py – modulo che definisce la classe Host....9
netmapping.py –mappatura degli Host in Thread....9
netstat.py – lettura e decodifica delle informazioni sulle connessioni....10
Decodica delle informazioni....12
Decodifica degli IPv6....14
hardware.py – utilizzo delle risorse hardware....15
__main__.py – sorgente principale e webserver....16
Templates – rappresentazione delle informazioni (lato client)....17

1. Specifiche del problema

Sviluppare un semplice framework di monitoraggio di risorse di una rete LAN in grado di accorgersi delle più semplici modifiche che avvengono in essa (discovering dei dispositivi e accorgersi se non accesi o spenti) e delle connessioni (in entrata e uscita) effettuate sulla macchina ospite del sw (ad esempio l'uso di servizi come SMTP, FTP, Telnet, POP3 e HTTP).

Nel caso si riesca a sviluppare anche moduli di monitoraggio delle risorse hardwae (consumo CPU, Hard Disk, etc), la cosa costituirà motivi di maggior merito.

2. Scelte progettuali

Abbiamo realizzato un framework di monitoraggio multi-thread che raccoglie informazioni sulla macchina e la rete locale, come host e rispettivi servizi, connessioni locali e risorse hardware utilizzate. Quindi espone le informazioni tramite un server HTTP su una porta TCP locale.

Per lo sviluppo del software abbiamo scelto di utilizzare il linguaggio di programmazione *Python*, alcune librerie (standard e non) e un programma esterno per risolvere dei problemi intermedi utili a semplificare lo sviluppo del programma. In particolare, abbiamo usato i seguenti moduli del Python:

- *ipaddress* e' un modulo per la manipolazione degli indirizzi IPv4 e IPv6
- *python-nmap* fornisce un'API a *nmap*
- *psutil* e' un modulo per l'ottenimento di informazioni sui processi e sulle risorse del sistema operativo

Abbiamo inoltre utilizzato un modulo, *netstat*, per la lettura e la decodifica delle informazioni sulle connessioni *TCP* del sistema operativo, e lo abbiamo adattato per consentire la lettura anche per i protocolli *UDP*, *UDP (IPv6)* e *TCP (IPv6)*. Il modulo originale può essere reperito al seguente indirizzo:

- *http://voorloopnul.com/blog/a-python-netstat-in-less-than-100-lines-of-code/*

Di seguito descriviamo le scelte progettuali relative ai linguaggi, standard, sistemi operativi e descriveremo le versioni utilizzate.

Python

Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti e caratterizzato da una sintassi pulita ed intuitiva. Tuttavia consente l'utilizzo del paradigma procedurale permettendo una maggiore flessibilita' in fase di sviluppo.

Dispone di numerosi moduli di base che offrono soluzioni eleganti a molti problemi tipici della programmazione, come ad esempio la manipolazione delle stringhe. Diversi moduli sia delle librerie standard che esterni hanno delle parti di codice scritte in *C*, il che aumenta le prestazioni. Inoltre, lo stesso interprete di riferimento (*CPython*) è scritto in *C*.

L'interprete *Python* dispone di una modalità interattiva che consente di testare i moduli in maniera indipendente (vedere esempi nella sezione dell'implementazione).

Questo linguaggio ci ha permesso di creare in pochi passi un webserver funzionante capace di rispondere a richieste *HTTP* in maniera sufficientemente efficace al problema affrontato.

GNU/Linux

Nonostante il *Python* sia un linguaggio multipiattaforma e altamente portabile, a causa delle differenze a basso livello che caratterizzano i sistemi operativi oggi piu' diffusi, abbiamo scelto di non estendere la compatibilita' oltre le distribuzioni *GNU/Linux*.

AJAX

La tecnologia *AJAX* ha permesso l'aggiornamento asincrono delle 3 sezioni della pagina. Infatti, la sezione relativa all'hardware viene richiamata circa ogni secondo, quella degli host ogni 20 mentre le connessioni locali ogni 5 secondi. Questo diminuisce il carico complessivo e la quantita' di dati scambiati tra client e server, sebbene girino la stessa macchina.

JavaScript e jQuery

Invece di renderizzare la pagina lato server tramite un motore di template, abbiamo preferito affidare il compito al browser sfruttando il linguaggio di riferimento, *JavaScript*. La libreria *jQuery* poi rappresenta un potente ed efficace strumento per gestire ad alto livello sia le richieste GET che la renderizzazione della pagina.

JSON

Per lo scambio dei dati tra client e server abbiamo optato per *JSON* piuttosto che per l'*XML* per i seguenti motivi:

- maggiore flessibilita'
- maggiore semplicita' di portare dizionari *Python* in oggetti *JSON*
- oggetti nativi del *JavaScript* col quale vengono elaborati dal browser

Versionamento del software

Abbiamo optato per le ultime versioni stabili disponibili dei rispettivi software utilizzati, dall'interprete Python alle librerie, al framework jQuery. Lo sviluppo si e' quindi basato su Python 3.3, estendendo la compatibilita' all'ultima versione stabile del ramo 2 di Python, sia per la sua attuale vasta diffusione che per quei sistemi operativi che non forniscono l'ultima versione di Python 3.

Si consiglia pertanto di scegliere versioni relativamente recenti. In particolare, abbiamo riscontrato problemi con versioni di *nmap* piu' datate della 6.25, ma non escludiamo possano verificarsi problemi con versioni di altri pacchetti non testate.

Di seguito un elenco sufficientemente esaustivo dei software adottati e relative versioni.

Interpreti Python

- CPython 2.7.3, 2.7.5
- CPython 3.3.2

Pacchetti Python3

- PIP 1.3.1, 1.4
- python-nmap 0.3.1
- ipaddress 1.0.4
- psutil 1.0.1

Pacchetti Python2

- PIP 1.0, 1.4
- python-nmap 0.2.7
- ipaddress 1.0.4
- psutil 1.0.1

Altri software

- nmap 6.25, 6.40
- netstat (incluso nell'archivio)
- jQuery 2.0.3 (incluso nell'archivio)

Browser

- Firefox 23
- Chromium 28

OS/kernel

- Ubuntu GNU/Linux 12.04 LTS “Precise”, kernel Linux 3.2.0-51-generic
- Parabola GNU/Linux (derivata di Arch), kernel Linux 3.10.5-1-LIBRE

Installazione delle dipendenze e avvio del programma

Per installare python, nmap e pip su Debian, Ubuntu e derivate:

```
# apt-get install python python-pip python-dev nmap
```

Per installare python e pip su Arch e derivate:

```
# pacman -S python python-pip nmap
```

Questi tre moduli sono installabili tramite gestori di pacchetti python come *pip* o *easy_install*.

```
# pip install ipaddress python-nmap psutil
```

Per lanciare il programma e' sufficiente lanciare uno dei seguenti comandi a seconda se si vuole o meno specificare la rete in cui effettuare la ricerca e/o la porta del web server:

```
$ python network_monitor.zip
$ python network_monitor.zip --network 192.168.0.0/24
$ python network_monitor.zip --port 8888
```

Per visualizzare le informazioni, aprire il browser in uno dei seguenti indirizzi (sostituire eventualmente *8080* con la porta specificata dall'utente):

```
http://localhost:8080
http://127.0.0.1:8080
```

3. Implementazione

Descriviamo l'implementazione dei vari moduli che fanno parte del programma.

addresses.py – gestione degli indirizzi

Questo modulo fornisce due funzioni per la gestione degli indirizzi: *_get_ip()* e *get_network_address()*.

_get_ip() cerca di ottenere l'indirizzo *IP* della macchina aprendo una socket con *google.com* e leggendo il nome della socket, infatti una socket ha la forma *indirizzo_ip:porta*, e da quelle informazioni ricaviamo l'indirizzo della macchina. In caso non fosse possibile ottenere l'indirizzo *IP* (perché ad esempio non si è collegati alla rete o improbabilmente *google.com* sia offline) restituiamo un *False*.

get_network_address() richiama *_get_ip()* per ottenere l'indirizzo *IP* della macchina, e prova a calcolare l'indirizzo della rete (in notazione CIDR) sostituendo all'ultimo gruppo di cifre indirizzo *IP* della macchina *0/24*: il procedimento funziona se la maschera della rete è *255.255.255.0*, come spesso si verifica in una *LAN* domestica.

Naturalmente il programma offre la possibilità di inserire manualmente l'indirizzo della rete, oltre che il numero della porta *TCP* su cui il webserver sarà in ascolto.

scan.py – interfaccia ad alto livello a python-nmap

Questo modulo offre due metodi per la scansione della rete.

host_discovery() esegue una scansione (tramite *nmap*) all'indirizzo di rete passato come parametro, e alla fine della scansione (che impiega circa 30 secondi) inserisce in una lista tutti gli host trovati e ritorna una lista degli *IP* attivi.

Alcuni host potrebbero non venire trovati se protetti da firewall come *iptables*. Il tipo di ricerca effettuato è una *ping scan*, cioè non va oltre alla determinazione dello stato dell'host.

scan_host() esegue una scansione sull'host passato come parametro, utilizzando *nmap*. La scansione è del tipo *probing*, cioè consiste nel provare ad effettuare richieste sulle varie porte dell'host per

determinare il tipo di servizio disponibile su quella porta. Quindi la funzione ritorna l'oggetto *PortScanner* che contiene le informazioni sulla scansione. La ricerca impiega circa 30 secondi.

host.py – modulo che definisce la classe Host

In questo modulo creiamo la classe *Host* come figlia della classe *Thread*. L'oggetto Host, una volta istanziato, è in grado di ottenere in maniera autonoma (essendo un *Thread*) le informazioni su un *Host* (attraverso il metodo *scan_host()* di *scan.py*) e di registrare queste informazioni sull'attributo *info*. Un esempio:

```
>>> from host import Host
>>> h = Host('192.168.0.100')
>>> h.start()
>>> Checking services on 192.168.0.100
print(h.info)
{'services': {'ip': [], 'udp': [], 'sctp': [], 'tcp': [[22, u'open', u'ssh', u'OpenSSH',
u'5.9p1 Debian 5ubuntu1.1'], [111, u'open', u'rpcbind', '', u'2-4'], [631, u'open', u'ipp',
u'CUPS', u'1.5']]}, 'ip': '192.168.0.100', 'isUp': True}
```

L'attributo *info* contiene dunque un dizionario (struttura dati che contiene coppie di chiavi-valori, simile alla hash table in Java) che a sua volta contiene i servizi dell'host (sottoforma di liste multidimensionali), lo stato, e l'indirizzo *IP* stesso del'host.

Finché il thread è attivo continua ad aggiornare queste informazioni a intervalli di 30 secondi. Quando l'host non è più attivo la voce *isUp* diventa *False* ma il thread continua comunque a verificarne lo stato e ricercarne i servizi a intervalli di 30 secondi.

netmapping.py –mappatura degli Host in Thread

Il modulo *netmapping* definisce la classe *NetMapper* (figlia dela classe *Thread* anche essa). L'oggetto *NetMapper*, una volta istanziato, si occupa di controllare a intervalli di 30 secondi la presenza di host nella rete (tramite il metodo *host_discovery()* del modulo *scan.py*) e si occupa di inserire i nuovi host trovati in una lista, di istanziare gli oggetti *Host* (ossia dei *Thread*, vedi sopra) e avviarli.

Esempio:

```
>>> from netmapping import NetMapper
>>> nm = NetMapper()
>>> nm.start()
>>> Searching for hosts on 192.168.0.0/24...
```

```
Found 3 host(s)
[192.168.0.1', '192.168.0.100', '192.168.0.103']
Starting thread for host 192.168.0.1
Starting thread for host 192.168.0.100
Starting thread for host 192.168.0.103
Checking services on 192.168.0.1
Checking services on 192.168.0.100
Checking services on 192.168.0.103
```

Inoltre, il modulo si occupa anche di fare il parsing del comando di lancio per controllare se sia stato fornito un indirizzo per la rete e in quel caso utilizza quell'indirizzo (invece dell'indirizzo ottenibile da *get_network_address()* del modulo *addresses.py*).

netstat.py – lettura e decodifica delle informazioni sulle connessioni

Questo modulo si occupa della lettura delle connessioni dai file forniti dal kernel su */proc/net/*

La funzione principale di questo modulo è *netstat()* che fornisce le infomazioni sul protocollo passato come parametro (TCP se non viene passato niente). Un esempio:

```
>>> from netstat import netstat
>>> for x in netstat():
...   print x
...
['0:', 'root', '0.0.0.0:111', '0.0.0.0:0', 'LISTEN', '', '']
['1:', 'statd', '0.0.0.0:51730', '0.0.0.0:0', 'LISTEN', '', '']
['2:', 'root', '127.0.0.1:53', '0.0.0.0:0', 'LISTEN', '', '']
['3:', 'root', '0.0.0.0:22', '0.0.0.0:0', 'LISTEN', '', '']
['4:', 'root', '0.0.0.0:631', '0.0.0.0:0', 'LISTEN', '', '']
['5:', 'michele', '0.0.0.0:17500', '0.0.0.0:0', 'LISTEN', '2146', '/home/michele/.dropbox-
dist/dropbox']
['6:', 'root', '0.0.0.0:36350', '0.0.0.0:0', 'LISTEN', '', '']
['7:', 'root', '192.168.0.100:45409', '166.84.136.101:80', 'TIME_WAIT', '2000', '/usr/bin/gnome-
session']
['8:', 'michele', '192.168.0.100:44016', '173.194.78.16:993', 'ESTABLISHED', '2811',
'/usr/lib/thunderbird/thunderbird']
['9:', 'michele', '192.168.0.100:56450', '173.194.70.189:443', 'ESTABLISHED', '2338',
'/usr/lib/chromium-browser/chromium-browser']
['10:', 'michele', '192.168.0.100:39961', '208.68.163.220:5222', 'ESTABLISHED', '2750',
'/usr/bin/pidgin']
['11:', 'root', '192.168.0.100:57800', '82.57.200.129:110', 'TIME_WAIT', '2000', '/usr/bin/gnome-
session']
['12:', 'michele', '192.168.0.100:59608', '91.189.94.25:80', 'CLOSE_WAIT', '2332',
'/usr/lib/ubuntu-geoip/ubuntu-geoip-provider']
['13:', 'michele', '192.168.0.100:45515', '166.84.136.101:80', 'ESTABLISHED', '2338',
'/usr/lib/chromium-browser/chromium-browser']
['14:', 'michele', '192.168.0.100:45506', '166.84.136.101:80', 'ESTABLISHED', '2338',
'/usr/lib/chromium-browser/chromium-browser']
['15:', 'michele', '192.168.0.100:35580', '108.160.163.43:80', 'ESTABLISHED', '2146',
```

```
'/home/michele/.dropbox-dist/dropbox']
['16:', 'michele', '192.168.0.100:44017', '173.194.78.16:993', 'ESTABLISHED', '2811',
'/usr/lib/thunderbird/thunderbird']
['17:', 'michele', '192.168.0.100:38118', '31.13.64.81:443', 'ESTABLISHED', '2338',
'/usr/lib/chromium-browser/chromium-browser']
['18:', 'michele', '192.168.0.100:54933', '69.171.235.16:443', 'ESTABLISHED', '2338',
'/usr/lib/chromium-browser/chromium-browser']

>>> for x in netstat('tcp6'):
...   print(x)
...
['0:', 'root', ':::44110', ':::0', 'LISTEN', '', '']
['1:', 'root', ':::111', ':::0', 'LISTEN', '834', '/sbin/rpcbind']
['2:', 'root', ':::22', ':::0', 'LISTEN', '1023', '/usr/sbin/sshd']
['3:', 'root', ':::631', ':::0', 'LISTEN', '1104', '/usr/sbin/cupsd']
['4:', 'statd', ':::40739', ':::0', 'LISTEN', '1114', '/sbin/rpc.statd']
```

Le informazioni vengono registrate su liste a più livelli, la lista di livello inferiore (che riguarda una singola connessione) contiene in ordine:

- utente
- indirizzo:porta locale
- indirizzo:porta remoto
- stato della socket
- PID del processo (se disponibile)
- nome del processo (se disponibile)

Per quanto riguarda i processi potrebbe verificarsi quanto segue:

- non si hanno privilegi di root e quindi alcuni processi potrebbero non essere mostrati
- il programma restituisce init o simili (si verifica con le socket in *TIME_WAIT*)

Spiegazione degli stati delle socket (da *man netstat*):

```
   State
       The state of the socket. Since there are no states in raw mode and usually no states
used in UDP, this column may be left blank. Normally this can be one of several values:

       ESTABLISHED
              The socket has an established connection.

       SYN_SENT
              The socket is actively attempting to establish a connection.
```

```
       SYN_RECV
              A connection request has been received from the network.

       FIN_WAIT1
              The socket is closed, and the connection is shutting down.

       FIN_WAIT2
              Connection is closed, and the socket is waiting for a shutdown from the remote
end.

       TIME_WAIT
              The socket is waiting after close to handle packets still in the network.

       CLOSE  The socket is not being used.

       CLOSE_WAIT
              The remote end has shut down, waiting for the socket to close.

       LAST_ACK
              The remote end has shut down, and the socket is closed. Waiting for
acknowledgement.

       LISTEN The socket is listening for incoming connections.  Such sockets are not included
in the output unless you specify the --listening (-l) or --all (-a) option.

       CLOSING
              Both sockets are shut down but we still don't have all our data sent.

       UNKNOWN
              The state of the socket is unknown.
```

Come è possibile notare, le socket in *TIME_WAIT* sono socket chiuse ma che hanno pacchetti ancora da gestire. Di solito rimangono in memoria al massimo per 60 secondi (può variare in base alle impostazioni del kernel). Nei sistemi *GNU/Linux* è normale che ci siano anche migliaia di socket in *TIME_WAIT* se la frequenza di apertura delle socket è molto alta (come si verifica ad esempio con il nostro client che richiede ogni secondo al webserver lo stato dell'hardware tramite una *GET*). Le socket in *TIME_WAIT* vengono gestite dal demone init o da altri processi padri (come *gnome-session*, *upstart*, *systemd*, ecc).

Naturalmente la funzione *netstat()* richiama altre funzioni tra cui quelle per la lettura e il parsing dei file e per la decodifica delle informazioni.

Decodica delle informazioni

Di seguito il contenuto di */proc/net/tcp6*

```
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout
inode
   0: 00000000:006F 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 9539 1
0000000000000000 100 0 0 10 -1
   1: 00000000:CA12 00000000:0000 0A 00000000:00000000 00:00000000 00000000   125        0 8864 1
0000000000000000 100 0 0 10 -1
   2: 0100007F:0035 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 296320 1
0000000000000000 100 0 0 10 -1
   3: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 1655 1
0000000000000000 100 0 0 10 -1
   4: 00000000:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 12509 1
0000000000000000 100 0 0 10 -1
   5: 00000000:445C 00000000:0000 0A 00000000:00000000 00:00000000 00000000  1000        0 18329 1
0000000000000000 100 0 0 10 -1
   6: 00000000:8DFE 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 152064 1
0000000000000000 100 0 0 10 -1
   7: 6400A8C0:C754 41400D1F:01BB 06 00000000:00000000 03:000000B5 00000000     0        0 0 3
0000000000000000
   8: 6400A8C0:ABF0 104EC2AD:03E1 01 00000000:00000000 00:00000000 00000000  1000        0 295281 1
0000000000000000 42 4 18 4 3
   9: 6400A8C0:DC82 BD46C2AD:01BB 01 00000000:00000000 02:000009D2 00000000  1000        0 296723 2
0000000000000000 46 4 30 5 3
  10: 6400A8C0:9C19 DCA344D0:1466 01 00000000:00000000 00:00000000 00000000  1000        0 298263 1
0000000000000000 57 4 26 2 2
  11: 6400A8C0:E8D8 195EBD5B:0050 08 00000000:00000001 00:00000000 00000000  1000        0 296332 1
0000000000000000 86 4 20 2 -1
  12: 6400A8C0:B363 658854A6:0050 01 00000000:00000000 02:00000BB2 00000000  1000        0 403067 2
0000000000000000 61 4 31 3 2
  13: 6400A8C0:C753 41400D1F:01BB 01 00000000:00000000 02:000006B2 00000000  1000        0 402144 2
0000000000000000 72 4 20 3 7
  14: 6400A8C0:8AFC 2BA3A06C:0050 01 00000000:00000000 00:00000000 00000000  1000        0 297665 1
0000000000000000 60 4 31 5 3
  15: 6400A8C0:ABF1 104EC2AD:03E1 01 00000000:00000000 00:00000000 00000000  1000        0 295282 1
0000000000000000 44 4 22 2 2
  16: 6400A8C0:B36C 658854A6:0050 01 00000000:00000000 02:00000232 00000000  1000        0 403132 2
0000000000000000 59 4 30 5 3
  17: 6400A8C0:D695 10EBAB45:01BB 01 00000000:00000000 02:00000C99 00000000  1000        0 296362 2
0000000000000000 62 4 25 4 2
```

Appare evidente che gli indirizzi sono in esadecimale, ma sono anche scritti al contrario. Prendiamone uno come esempio e trasformiamo ogni coppia in decimale:

```
6400A8C0:ABF0

>>> int('C0', 16)
192

>>> int('A8', 16)
168

>>> int('00', 16)
0

>>> int('64', 16)
100
```

```
>>> int('ABF0', 16)
44016
```

In sintesi, troviamo l'indirizzo *IP*:

```
>>> _ip('6400A8C0', False)  # il False indica che è un IPv4 e non un IPv6
'192.168.0.100'
```

Troviamo la porta:

```
>>> int('ABF0', 16)
44016
```

L'algoritmo (implementato da *Ricardo Pascal*) esegue la conversione nel modo illustrato sopra.

Per quanto riguarda gli stati, è sufficiente definire un dizionario (hash table):

```
STATE = {
    '01':'ESTABLISHED',
    '02':'SYN_SENT',
    '03':'SYN_RECV',
    '04':'FIN_WAIT1',
    '05':'FIN_WAIT2',
    '06':'TIME_WAIT',
    '07':'CLOSE',
    '08':'CLOSE_WAIT',
    '09':'LAST_ACK',
    '0A':'LISTEN',
    '0B':'CLOSING'
}
```

A questo punto la notazione esadecimale è la chiave per accedere alla stringa contenente lo stato.

Per trovare il processo che usa la connessione, l'algoritmo prevede di ricercare quel processo che sta utilizzando l'inode relativo alla connessione.

Decodifica degli IPv6

Prendiamo ad esempio l'indirizzo come risulta in */proc/net/tcp6* (oppure */proc/net/upd6*)

```
0000000000000000FFFF00000100007F
```

La funzione lo converte come segue:

```
>>> from netstat import _ip
```

```
>>> _ip('0000000000000000FFFF00000100007F', True)
'::ffff:7f00:1'
```

Dividiamo l'*IP* in blocchi da 4:

```
 0000:0000:0000:0000:FFFF:0000:0100:007F
```

Scambiamo i blocchi a due a due:

```
0000:0000:0000:0000:0000:FFFF:007F:0100
```

Scambiamo le coppie dentro i blocchi:

```
0000:0000:0000:0000:0000:FFFF:7F00:0001
```

Infine usiamo il modulo *ip_address* per abbreviarlo:

```
>>> from ipaddress import ip_address
>>> str(ip_address('0000:0000:0000:0000:0000:FFFF:7F00:0001'))
'::ffff:7f00:1'
```

hardware.py – utilizzo delle risorse hardware

Questo modulo sfrutta la libreria *psutil* per ottenere dal sistema operativo l'attuale utilizzo delle principali risorse hardware:

- utilizzo CPU
- utilizzo della memoria principale
- utilizzo della memoria di swap
- utilizzo del disco in lettura e scrittura
- utilizzo della banda di rete in entrata e uscita

La classe *Hardware* che estende *Thread* viene istanziata e lanciata dal web server. Dopo aver inizializzato le variabili, ripete continuamente le seguenti operazioni:

1. memorizza in variabili temporanee i valori relativi a letture/scritture e traffico in entrata/uscita
2. calcola il carico medio dei core della CPU nell'intervallo di un secondo
3. ricalcola i valori del punto (1) ed effettua la differenza con quelli memorizzati in precedenza. il valore ottenuto rappresenta un'approssimazione dell'utilizzo di disco e traffico di rete in un

secondo, dal momento che il punto (2) impiega un secondo ad essere compiuto
4. calcola l'utilizzo della memoria RAM e dello swap
5. attende un secondo, per poi riprendere il ciclo

Infine **get_results()** restituisce un dizionario contenente tutti i valori calcolati e viene richiamato circa ogni secondo tramite una richiesta *GET* del browser.

__main__.py – sorgente principale e webserver

Il programma inizia con i seguenti passi:

1. viene instanziato un oggetto della classe *HTTPServer* a cui viene passata la classe *MyHandler* personalizzata che gestira' le richieste *HTTP*
2. vengono istanziate e lanciate le classi *NetMapper* e *Hardware*
3. viene aperta una socket su una porta *TCP* locale (eventualmente specificata dall'utente da linea di comando)

La classe **MyHandler** (oltre al proprio costruttore di default che richiama quello di *BaseHTTPRequestHandler*) contiene degli attributi che una volta definiti rimangono costanti:

- definizione del metodo *write()* che gestisce correttamente la codifica del file da restituire, tenendo presente la versione del Python. Vengono poi aperti i 3 file necessari e vengono caricari su 3 differenti variabili. In questo modo si evita che il webserver vada a riaprire gli stessi file piu' volte ad ogni richiesta di caricamento dell'intera pagina web.
- il dizionario *request_to_response* abbina le richieste dei file da parte del browser alle variabili dove avevamo precedentemente salvato i file
- il dizionario *ext_to_content_type* invece associa le estensioni dei file al *Content-type* relativo nell'header delle risposte *HTTP*

Il metodo **do_GET()** della classe *MyHandler* viene invocato ad ogni richiesta *HTTP* di tipo *GET* da parte del browser. Innanzitutto suddivide la path richiesta in due variabili che rappresentano il file o il contenuto richiesto (*request*) e l'estensione relativa (*ext*).

L'estensione determina il *Content-type* da inviare nell'header della risposta *HTTP* al browser, mentre il contenuto puo' essere un file statico (precedente catturato in una variabile) oppure un dizionario generato con i valori ottenuti.

In caso di altre richieste non previste, viene generata e catturata l'eccezione *KeyError*, quindi viene restituito l'errore *404* relativo ad una pagina non trovata dal web server.

Templates – rappresentazione delle informazioni lato client

Questa cartella contiene:

- *index.html*, ovvero la struttura della pagina web e gli script *JavaScript/jQuery*
- *style.css* per migliorare la leggibilita' della pagina
- *jQuery-2.0.3-min.js*, l'ultima versione della famosa libreria *jQuery*

Il file *index.html* contiene una struttura semplice e pulita che funge da interfaccia al programma, composta da un header, titoli e tabelle. Gli script hanno principalmente due ruoli:

- sfruttando la tecnologia AJAX, effettuare a intervalli regolari (diversi per ogni sezione) delle richieste *HTTP GET*, in particolare delle *XMLHTTPRequest*
- generare dinamicamente codice *HTML* partendo dagli oggetti *JSON* ottenuti dal web server, e sostituirlo di volta in volta alle relative sezioni