Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

PycoTCP: Una libreria Python per l'Internet of Threads

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Federico Giuggioloni

Sessione di Ottobre
Anno Accademico 2015-2016

Introduzione

Internet e i protocolli TCP/IP erano inizialmente basati sull'interconnessione di computer, dunque le entità indirizzabili erano le interfacce di rete. A causa dell'evoluzione dei servizi di rete, questo tipo di comunicazione è diventato obsoleto. Da questa idea di fondo è nato il concetto di Internet of Threads (IoTh) [1], dal quale sono nate delle librerie che si pongono l'obiettivo di renderlo realtà.

Queste librerie, tra cui picoTCP e LWIPv6, sono scritte in C e offrono delle interfacce studiate ad hoc per l'IoTh, garantendo allo sviluppatore il controllo completo della virtualizzazione della rete che vanno a generare. Questo fa sì che siano la risorsa principale per lo sviluppo con questo nuovo paradigma.

Prendiamo ad esempio uno sviluppatore che ha già lavorato con il Python, e ha creato i suoi script che comunicano in rete attraverso il modulo *socket*. Quando questo sviluppatore si imbatte per la prima volta nel concetto dell'IoTh, è costretto a riscrivere le sue applicazioni in modo da renderle compatibili con questa nuova metodologia. Inoltre deve mantenere due rami dell'applicazione attivi allo stesso momento per affiancare il nuovo funzionamento al vecchio. Anche supponendo che lo sviluppatore in questione conosca già il linguaggio C, dovrà comunque studiare a fondo una di queste librerie per provarne il funzionamento e vedere se fa al caso suo.

Da questi problemi nasce PycoTCP. Attraverso la sua struttura orientata a fornire un'interfaccia semplice e immediata da comprendere, riesce a unificare tutte le attuali librerie per l'IoTh sotto una API identica a quella del

Python. Lo sviluppatore menzionato in precedenza dovrebbe essere in grado di prendere PycoTCP ed eseguire i suoi script nell'IoTh senza ulteriori passaggi, evitando anche di studiare il funzionamento e le differenze delle varie librerie.

Mediante questa semplificazione si vuole permettere una maggiore diffusione dell'IoTh rimuovendo qualsiasi barriera d'entrata per ogni sviluppatore interessato in questo nuovo paradigma di comunicazione di rete.

Indice

Elenco delle figure

Capitolo 1

Le reti per l'Internet of Threads

Per *Internet of Threads* si intende la capacità dei processi di essere indirizzabili come dei veri e propri nodi di rete in Internet, rendendo ogni processo un endpoint IP come un qualsiasi computer.

L'idea di assegnare un indirizzo IP a ogni processo invece che alle macchine fisiche è molto simile alla differenza tra telefoni fissi e cellulari. Infatti per contattare una persona tramite telefono fisso è necessario pensare a dove si trova, per poi chiamare il telefono che si trova più vicino. Al contrario, con i cellulari è possibile contattare direttamente la persona interessata.

Considerando il limitato spazio indirizzabile da parte dell'IPv4 è semplice vedere che l'IoTh non potrebbe esistere senza l'utilizzo sempre più diffuso dell'IPv6. Da questo punto di vista l'IoTh è una ovvia evoluzione delle vecchie tecnologie di interconnessione di rete resa possibile dall'IPv6 stesso.

1.1 Una rete di processi

Per interconnettere i processi non è possibile utilizzare switch o hub fisici: non hanno controller di rete hardware. È possibile però fornire dei controller di rete virtuali, che si vanno poi a collegare a switch a loro volta virtualizzati.

In questo modo si va creare la rete virtuale dei processi, nella quale ogni elemento connesso avrà il suo indirizzo IP.

Normalmente nei sistemi operativi possiamo trovare un unico stack di rete TCP/IP implementato al livello del Kernel, utilizzabile attraverso una Application Programming Interface (API) che solitamente è quella dei Berkeley Socket. Dunque il livello Network dello stack TCP/IP è unico per l'intera macchina, fornendo un solo indirizzo IP. Nell'IoTh ogni processo può scegliere quale stack TCP/IP utilizzare, portando l'intero stack all'interno dei processi stessi. Possono comunque condividere lo stesso stack di rete, ma questa sarebbe una scelta di sviluppo invece che una limitazione imposta dal sistema operativo.

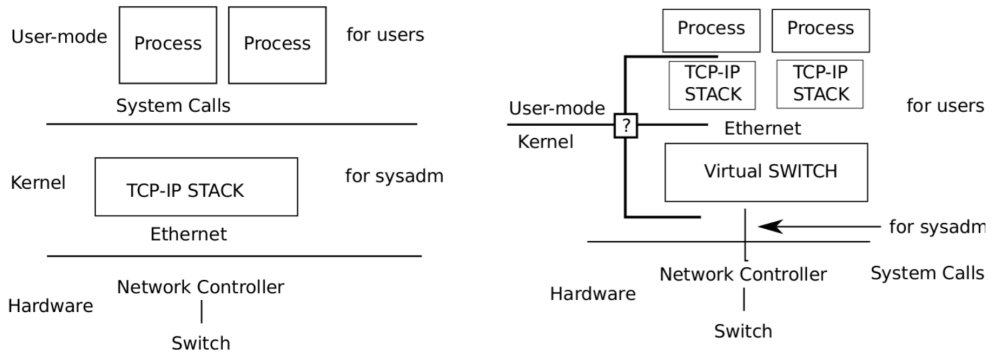

Figura 1.1: Prospettive differenti riguardo l'implementazione dei servizi di rete: Il supporto standard degli SO a sinistra, IoTh a destra.

1.2 Il nuovo paradigma

Attraverso l'IoTh, le comunicazioni di rete in TCP/IP diventano automaticamente il mezzo per un servizio di *Inter Process Communication* (IPC), rendendo questi protocolli di comunicazione indipendenti dall'host su cui il processo è effettivamente in esecuzione. Infatti non è necessario sapere dove si trova il processo con cui si vuole comunicare: basta utilizzare il suo indirizzo IP. Questo semplifica enormemente la migrazione di un servizio da un host a un altro, dato che è sufficiente fermare il processo dal primo host ed eseguirlo nel secondo, senza ulteriore configurazione.

Inoltre grazie all'IoTh è possibile eseguire un qualsiasi numero di processi che offrono un certo tipo di servizio all'interno dello stesso host. Ad esempio, si possono eseguire multipli daemon POP, IMAP, DNS, HTTP, etc.. ed ognuno potrebbe operare con il suo stack e con la porta standard per quel servizio.

Alcuni progetti hanno fornito delle implementazioni dello switch Ethernet virtuale, uno dei quali è quello utilizzato durante lo sviluppo di PycoTCP. Si tratta di VDE (Virtual Distributed Ethernet), che fornisce gli switch virtuali (`vde_switch`) e i cavi con cui collegarli (`vde_plug`). [2]

Per l'implementazione dello stack TCP/IP a livello utente esistono due librerie principali: PicoTCP e LWIPv6. Esiste una terza libreria nota come FDPicoTCP, il cui obiettivo è quello di fornire le stesse funzionalità di PicoTCP ma mediante un'API conforme con i berkeley socket. [7]

PicoTCP La libreria da cui PycoTCP trae il nome. Effettua la simulazione dei vari componenti di rete dividendola in "tick" temporali. Ogni "tick" corrisponde a un passo della simulazione, e richiede la registrazione di Callback per ottenere i dati dai Socket. Ha una API completamente diversa dai Berkeley socket. Originariamente questa doveva essere l'unica libreria supportata da PycoTCP. [5]

LWIPv6 Libreria sviluppata come parte del progetto Virtual Square per aggiungere il supporto all'IPv6 nella libreria LWIP. Nel tempo si è

evoluta indipendentemente, differenziandosi sempre più dall'originale LWIP. Implementa i Socket dell'IoTh con possibilità di usare (per la maggior parte) la stessa interfaccia dei Berkeley Socket (effettivamente sostituendosi a loro). [3]

FDPicoTCP Libreria creata recentemente per rendere l'interfaccia fornita da PicoTCP conforme con quella dei Berkeley Socket. Per questo genera e restituisce File Descriptor validi virtuali o reali. [4]

Questo nuovo paradigma sembra avere già le fondamenta per funzionare correttamente: Il concetto è ben definito, alcuni progetti forniscono la virtualizzazione della rete, mentre delle librerie forniscono lo stack TCP/IP su cui i processi possono lavorare. Resta comunque un grave problema che ne impedisce la diffusione: per integrare lo stack all'interno di un programma già scritto è necessario modificare la parte del codice che riguarda le comunicazioni di rete, studiando il funzionamento di queste librerie, per poi ricompilare il tutto. Inoltre, le API fornite da queste librerie sono eterogenee, e differenti dalle API dei Berkeley Socket.

PycoTCP risolve il problema della riscrittura del codice, in modo che qualunque script possa essere eseguito nell'IoTh senza alcuna configurazione; inoltre, unifica le API di tutte queste librerie sotto una unica API identica a quella dei Berkeley Socket in Python, evitando il passo di studio delle librerie e di come devono essere utilizzate.

Se un giorno verrà creata una nuova libreria C che fornisca il funzionamento di uno stack TCP/IP, PycoTCP ne permetterà l'integrazione in maniera semplice e senza dover riscrivere il cuore del pacchetto.

Con queste enormi semplificazioni, creare un programma che opera all'interno dell'Internet dei Thread è semplice come scrivere uno script Python mediante l'utilizzo del modulo `socket`, mentre rendere uno script esistente compatibile con l'IoTh è triviale.

Tutto questo non sarebbe possibile senza l'utilizzo del Python, uno dei linguaggi più adatti alla sperimentazione, del quale è stata scelta la versione 2.7 perchè è ancora la versione predefinita in molti sistemi.

Capitolo 2

Funzionamento

2.1 Tipologie

Esistono due metodi principali di utilizzo di PycoTCP, il metodo Implicito e quello Esplicito. La differenza principale è che, mentre in quello esplicito si devono aggiungere delle componenti estranee al funzionamento del modulo `socket` standard, in quello implicito non si richiede alcuna modifica del codice già esistente. Per la stessa motivazione, soltanto l'utilizzo del metodo Implicito consente di mantenere la compatibilità con i `socket` standard del Python.

2.1.1 Funzionamento esplicito

Questo è il metodo base di sviluppo con PycoTCP. È utile soltanto se lo sviluppatore conosce già l'IoTh e vuole averne il controllo completo, pur usando il Python come linguaggio di programmazione.

Così facendo è necessario inizializzare manualmente il contesto IoTh fornito dalle librerie sottostanti, oltre a scegliere quale libreria utilizzare. In questo caso bisogna importare i vari moduli manualmente dal pacchetto `pycotcp`. Mentre si importa il modulo `socket` del `pycotcp` è possibile usare `as socket` per rendere lo sviluppo il più simile possibile al normale sviluppo in Python.

Dopo l'inizializzazione del contesto si può scrivere codice come ci si aspetterebbe in un qualsiasi altro script che non fa uso di PycoTCP, oppure utilizzare le API specifiche per l'IoTh per avere un controllo più approfondito sul comportamento dei singoli componenti.

D'altra parte, importando il modulo `socket` con un altro nome si ha anche la possibilità di usare sia il modulo per l'IoTh che quello standard contemporaneamente nello stesso script.

L'utilizzo di questa metodologia è comunque consigliato solo per nuovi progetti, quando si preferisce la flessibilità del Python all'efficienza del C.

Ecco uno script di esempio che utilizza il funzionamento esplicito del PycoTCP:

Listing 2.1: nuovo_script.py

```
from pycotcp.picotcpadapter import PicoTCPAdapter
from pycotcp.fdpicotcpadapter import FDPicoTCPAdapter
from pycotcp.lwipv6adapter import Lwipv6Adapter

from pycotcp.pycotcp import Pyco
from pycotcp.pycotcp import DeviceInfo
import pycotcp.socket as socket

# Soltanto un contesto per script
context = PicoTCPAdapter()
#context = FDPicoTCPAdapter()
#context = Lwipv6Adapter()

# ioth inizializza la libreria scelta come contesto
ioth = Pyco(context=context)
device = DeviceInfo(context=context).with_type(devtype) \
        .with_name(device) \
        .with_path(filename) \
```

```
        .with_address(address) \
        .with_netmask(netmask) \
        .create()

# Richiesto solo utilizzando PicoTCPAdapter (Vedi sviluppi futuri)
ioth.start_handler()

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM,
    context=context)
sock.bind(("10.40.0.55", "5500"))
# Codice usando l'API standard del Python o l'interfaccia
    specifica per IoTh

[...]

ioth.stop_handler() # Solo richiesto per PicoTCPAdapter
```

Inizializzazione del contesto

Per semplificare l'inizializzazione del contesto e renderla unica per qualsiasi libreria utilizzata sono state create le classi di comodo `Pyco` e `DeviceInfo`.

La classe `Pyco` inizializza la libreria sottostante al momento della creazione, compiendo passaggi speciali per determinate librerie se necessario. Il parametro fondamentale al momento della creazione è la libreria (o contesto) da utilizzare per le operazioni dei `Socket`. Gli altri parametri si applicano solo alla libreria `PicoTCPAdapter` e permettono di avere il controllo diretto sulle callback richiamate da PicoTCP.

La classe `DeviceInfo` si occupa di mantenere tutte le informazioni riguardo un dispositivo virtuale all'interno di un singolo oggetto, oltre a crearlo nella libreria sottostante. Creare un oggetto `DeviceInfo` richiede una serie di parametri passabili dal costruttore oppure con la sintassi tipica del Pat-

tern *Builder*. In base alla libreria utilizzata può essere necessario un numero maggiore o minore di questi parametri.

È importante però ricordare che il modulo non si occupa di inizializzare il contesto VDE sulla macchina, dato che questo richiede dei permessi in fatto di gestione delle reti. Per far questo, nel pacchetto è inclusa la utility `pyco-vde-setup`.

Le classi `Pyco` e `DeviceInfo` sono indipendenti dalle librerie sottostanti dato che tutte le chiamate alle librerie si svolgono mediante degli *Adapter* (Vedi la sezione 2.3.1).

2.1.2 Funzionamento implicito

Questa metodologia è la parte più importante del pacchetto. Il suo obiettivo è quello di eliminare la necessità di riscrivere i propri script per effettuare esperimenti con l'IoTh, e quindi abbattere qualsiasi barriera d'entrata.

Il risultato si ottiene mediante la sostituzione del modulo `socket` compreso tra i moduli standard del Python con quello nel pacchetto *PycoTCP*. Il modulo `socket` riscritto si basa sulla stessa interfaccia fornita dal modulo standard, sostituendo tutte le chiamate con richieste al contesto attuale. Il contesto specifica quale libreria utilizzare per la sessione IoTh corrente.

La scelta del contesto avviene nel momento in cui si decide di eseguire uno script attraverso `pyco-wrapper` con il parametro `-l` seguito da `'picotcp'`, `'lwipv6'` o `'fdpicotcp'`. Al momento, PicoTCP è sempre disponibile, mentre le altre due librerie sono opzionali e utilizzabili solo se presenti nel sistema al momento dell'installazione del pacchetto.

Ecco un esempio di script Python preso dalla documentazione:

Listing 2.2: socket_server.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.bind(("0.0.0.0", 8090))
s.listen(3)

while True:
    try:
        connection, address = s.accept()
        buf = connection.recv(64)
        if len(buf) > 0:
            print buf
            break
    except:
        print "exception"
        break

s.close()
```

Per eseguire lo script nell'IoTh senza effettuare modifiche basta eseguire il seguente comando:

```
$> pyco-wrapper <python-script> -a <indirizzo> -d
    <nome-dispositivo> -f <path-del-vde_switch> -l <nome-libreria>
    -c
# -c Per creare la VDE (richiede i permessi di amministratore)
# Nome libreria deve essere uno di: picotcp, fdpicotcp, lwipv6
# Default: picotcp

#Esempio:
$> pyco-wrapper socket_server.py -a 10.40.0.32 -d web0 -f
    /tmp/web0.ctl -l picotcp -c
```

Se si ha una VDE già impostata nel sistema o la si vuole impostare e gestire separatamente dall'esecuzione dello script basta omettere il flag *-c* e utilizzare la utility `pyco-vde-setup`.

Modifica dell'ambiente Python

Questo funzionamento è stato ottenuto modificando la variabile d'ambiente `PYTHONPATH` di Python stesso all'interno di `pyco-wrapper`:

Listing 2.3: pyco-wrapper.py

```
realpath = os.path.realpath(__file__)
scripts = realpath[:realpath.rfind('/')]
eggpath = realpath[:realpath.rfind('EGG-INFO')]
newpythonpath = eggpath + "pycotcp"

os.environ["PYTHONPATH"] = newpythonpath
```

Inoltre, se è stato specificato il flag *-c*, verrà creata una VDE in base ai parametri passati. Questo, senza l'aiuto di ulteriori librerie, richiede alcuni permessi per la creazione di TAP nel sistema.

Listing 2.4: pyco-wrapper.py

```
if create:
    subprocess.call([scripts + "/pyco-vde-setup", "start",
        cidrswitchaddress, filename, device])
```

La modifica all'ambiente di Python non si applica immediatamente allo script correntemente in esecuzione. Per permettere alla modifica di avere effetto si è preferito utilizzare il modulo *subprocess*:

Listing 2.5: pyco-wrapper.py

```
subprocess.call([scripts + "/pyco-runner.py", args.script] +
    runnerparams)
```

Come si vede dal codice, lo script messo in esecuzione non è direttamente quello bersaglio ma lo script `pyco-runner.py` (con il bersaglio passato come parametro).

Questo script, eseguito nell'ambiente personalizzato specificato da `pyco-wrapper`, provvede a creare il contesto IoTh in base ai parametri forniti. Per mantenere il contesto ed eseguire lo script bersaglio è stato necessario fare uso del modulo `runpy`, che esegue lo script dato come parametro come se fosse parte dello script corrente.

Listing 2.6: pyco-runner.py

```
[...]
os.environ["PYCOTCP_CONTEXT"] = args.library
os.environ["PYCOTCP_DEVTYPE"] = devtype
os.environ["PYCOTCP_DEVICE"] = device
os.environ["PYCOTCP_FILENAME"] = filename
os.environ["PYCOTCP_NETMASK"] = netmask
os.environ["PYCOTCP_ADDRESS"] = address
[...]
sys.argv = sys.argv[:1]
sys.argv = sys.argv + parameters

runpy.run_path(args.script, run_name = "__main__")
```

Così facendo si ottiene l'esecuzione dello script bersaglio con il nuovo ambiente specificato da `pyco-wrapper` e il contesto IoTh già inizializzato.

Ogni volta che l'interprete Python arriva a un'istruzione `import` deve effettuare la risoluzione del modulo specificato[1]. Per farlo ha una lista di path predefinite in cui cercare, cominciando dalla directory corrente dello script. Se lo stesso modulo si ripete in più di una di queste directory, viene preso il primo della lista. Dato che `PYTHONPATH` è modificato da `pyco-wrapper` in modo da puntare alla directory del modulo PycoTCP, ogni istruzione `import socket` importerà il modulo `socket.py` di PycoTCP stesso.

[1] `https://docs.python.org/2/tutorial/modules.html#the-module-search-path`

I parametri dello script bersaglio possono essere passati a `pyco-wrapper`. Per mantenerli all'esecuzione di `pyco-runner`, questi sono di nuovo passati come parametri. Al contrario, dato che l'esecuzione dello script bersaglio avviene in linea attraverso l'utilizzo di `runpy`, per effettuare il passaggio dei parametri è stata modificata direttamente la variabile `sys.argv`, solitamente usata in sola lettura.

`sys.argv[0]` è lasciato invariato all'interno del `pyco-runner`, affinché l'applicazione sottostante possa capire se è stata eseguita in modo normale o se è in corso un *wrapping* da parte di `pyco-wrapper`.

Alla creazione del primo socket sono effettuati passaggi di inizializzazione a partire dai parametri forniti dal wrapper all'interno della funzione `__init__`:

Listing 2.7: socket.py

```
[...]
environ_context = os.environ.get("PYCOTCP_CONTEXT")
new_context = None

if Socket.default_context is None and environ_context is
    not None:
    if environ_context == "picotcp":
        from picotcpadapter import PicoTCPAdapter
        new_context = PicoTCPAdapter()

    elif environ_context == "lwipv6":
        from lwipv6adapter import Lwipv6Adapter
        new_context = Lwipv6Adapter()

    elif environ_context == "fdpicotcp":
        from fdpicotcpadapter import FDPicoTCPAdapter
        new_context = FDPicoTCPAdapter()
```

```
elif Socket.default_context is not None:
    print "Taking default context %s..." %
        Socket.default_context
    context = Socket.default_context
[...]
elif context is None and new_context is not None:
    print "Taking context %s from environment..." %
        environ_context
    self.context = new_context

    devtype = os.environ.get("PYCOTCP_DEVTYPE")
    device = os.environ.get("PYCOTCP_DEVICE")
    filename = os.environ.get("PYCOTCP_FILENAME")
    address = os.environ.get("PYCOTCP_ADDRESS")
    netmask = os.environ.get("PYCOTCP_NETMASK")

    ioth = Pyco(context=new_context)
    device =
        DeviceInfo(context=new_context).with_type(devtype) \
            .with_name(device) \
            .with_path(filename) \
            .with_address(address) \
            .with_netmask(netmask) \
            .create()

    if environ_context == "picotcp":
        ioth.start_handler()

    Socket.default_context = new_context
[...]
```

Funzioni del modulo socket.py originale

La sostituzione del modulo 'socket.py' implica la perdita di tutte le definizioni al suo interno. Alcune di queste funzioni sono applicabili al contesto dell'IoTh senza ulteriori modifiche, perciò una reimplementazione sarebbe ridondante.

Per ovviare al problema si è ricorso all'importazione manuale del modulo socket.py dalle cartelle di ricerca delle librerie del Python:

Listing 2.8: socket.py

```
oldsocket = imp.load_source('socket', os.path.dirname(os.__file__)
    + "/socket.py")
#inet_pton = oldsocket.inet_pton
#print "inet_pton: %s" % str(inet_pton)

# Esempio di riutilizzazione di variabili della libreria socket
    standard
error = oldsocket.error
gaierror = oldsocket.gaierror
herror = oldsocket.herror
timeout = oldsocket.timeout
```

Così facendo si possono ripristinare le funzioni originali mediante un semplice assegnamento, a mano a mano che si individuano incompatibilità durante l'esecuzione del codice.

2.2 Integrazione delle librerie C

La struttura del codice è stata per la maggior parte influenzata dall'integrazione delle varie librerie C che, sebbene svolgessero la stessa funzionalità, sono eterogenee rispetto all'interfaccia esposta verso l'esterno.

Per integrarle in modo omogeneo si è voluta usare una struttura basata su delle classi 'Adattatore', ognuna delle quali gestisce il funzionamento di una libreria.

Una parte del codice integra le librerie scritte in C nel codice Python, aggiungendo funzioni che ne semplificano la gestione.

2.2.1 PicoTCP

PicoTCP è stata integrata in maniera diversa dalle altre due librerie poichè era già presente nel vecchio prototipo sviluppato per il Progetto di Sistemi Virtuali dell'A.A. 2015/2016 da parte di un altro studente.

Questa vecchia implementazione si basa sulla costruzione di un modulo C esterno, compilato a parte e poi integrato con il modulo Python finale. Questo include la libreria PicoTCP e ne espone un'interfaccia completamente modificata per rendere l'utilizzo più diretto.

Il primo passo è stato scrivere l'`Adapter` `PicoTCPAdapter` di questa implementazione. L'interfaccia fornita da questa classe doveva diventare la stessa fornita da eventuali futuri `Adapter`, ed è stata creata sulla base del modulo C esterno già esistente.

Avendo già le classi che forniscono il funzionamento principale `Pyco` e `DeviceInfo`, una volta scritto l'Adattatore è stato necessario risolvere ogni problema riscontrato con l'utilizzo della libreria.

Difficoltà con PicoTCP

La diversa API fornita da PicoTCP rispetto ai Socket standard ha causato problemi d'integrazione. PicoTCP si aspetta che l'applicazione fornisca varie

Callback in cui ottenere/inviare dati attraverso i Socket e si ponga in un ciclo continuo di chiamate alla funzione `stackTick()`. Questa funzione si occupa di avanzare la simulazione di un passo.

Essendo l'obiettivo principale del progetto quello di rendere l'interfaccia simile se non identica a quella del Python, andava trovato un metodo per la linearizzazione di questo processo, per evitare le chiamate alla funzione `stackTick()` e fornire le tipiche funzioni `recv`, `send` e analoghe.

A questo scopo, la classe `Pyco` (se inizializzata con il contesto `PicoTCPAdapter`) mette in esecuzione un Thread che non fa altro che chiamare lo `stackTick()` fino alla chiusura dell'interità dell'applicazione. Attraverso un sistema di semafori è stato possibile rendere sincrone le richieste ai Socket, trasformando questo sistema basato su *Callback* in uno più conforme ai Berkeley Socket. [7]

2.2.2 LWIPv6 e FDPicoTCP

Le altre librerie, inizialmente non incluse nel progetto, sono state integrate utilizzando CFFI, un modulo Python che permette di chiamare librerie C dal codice Python con pochi passi intermedi.

È stato necessario creare un nuovo file Python per ognuna delle librerie, di nome `nome-libreria-build.py` contenente le dichiarazioni CFFI e la definizione di ulteriori funzioni di comodo in C.

CFFI crea automaticamente il modulo C esterno da integrare al modulo Python a partire dal codice e le librerie fornite all'interno del file `build.py`.

A questo punto, nei rispettivi Adattatori è stato sufficiente chiamare le funzioni esposte dal modulo generato da CFFI, utilizzando funzioni di comodo per la creazione di `struct` le quali non potevano essere fissate nel codice.

2.3 Unificazione delle librerie

2.3.1 Struttura ad Adattatori

È stata creata una classe `Adapter` base per tutte le librerie. Fornisce l'interfaccia base che sarà comune a tutte, lanciando un'eccezione se la funzione richiamata non è disponibile nella sottoclasse utilizzata.

Listing 2.9: adapter.py

```
class Adapter:

    NOT_IMPLEMENTED = "Not yet implemented"

    def __init__(self):
        print "initing adapter"
        pass

    def deleteLink4(self):
        raise NotImplementedError(NOT_IMPLEMENTED)

    def createDevice(self):
        raise NotImplementedError(NOT_IMPLEMENTED)

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

Per ogni libreria è stato creato un `Adapter` specifico che si occupa di svolgere l'azione richiesta attraverso una sequenza di chiamate alla libreria alla quale si riferisce.

Listing 2.10: picotcpadapter.py

```
class PicoTCPAdapter(Adapter):
```

```
    initialized = False

    LIBRARY_NAME = "picotcp"

  . . .

    def __init__(self):
        if not PicoTCPAdapter.initialized:
            pycoclib.init()
            PicoTCPAdapter.initialized = True

    . . .

    def createDevice(self, device_type, device_name = None,
        device_path = None):
        result = None
        if device_name is not None:
            if device_path is not None:
                result = pycoclib.createDevice(device_type,
                    device_name, device_path)
            else:
                result = pycoclib.createDevice(device_type,
                    device_name)
        else:
            result = pycoclib.createDevice(device_type)

        return self.check_error(result)

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

Listing 2.11: lwipv6adapter.py

```
class Lwipv6Adapter(Adapter):

   ...

   def __init__(self):
        print "initing lwipv6adapter"

        stack_pointer = lwipv6.lwip_stack_new()
        self.stack_pointer_list.append(stack_pointer)

        self.current_stack = len(self.stack_pointer_list) - 1

   ...

   def createDevice(self, device_type, device_name = None,
       device_path = None):
        print "Creating LWIPv6 device %s %s %s" % (device_type,
            device_name, device_path)

        if device_name is None:
            device_name = "vde"

        if device_path is None:
            device_path = "/tmp/" + device_name + ".ctl"

        interface = None
        if device_type == "vde":
            print "building vde interface"
            interface =
                lwipv6.lwip_vdeif_add(self.get_current_stack_pointer(),
                device_path)
        elif device_type == "tun":
```

```
            interface =
                lwipv6.lwip_tunif_add(self.get_current_stack_pointer(),
                device_path)
        elif device_type == "tap":
            interface =
                lwipv6.lwip_tapif_add(self.get_current_stack_pointer(),
                device_path)
        print "interface done building %s" % str(interface)

        lwipv6.lwip_ifup(interface)
        self.interfaces[device_name] = {
                    'pointer': interface
                }
        return interface

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

Listing 2.12: fdpicotcpadapter.py

```
class FDPicoTCPAdapter(Adapter):

    initialized = False

    LIBRARY_NAME = "fdpicotcp"

    ...

    def __init__(self):
        print "initing fdpicotcpadapter"
        if not FDPicoTCPAdapter.initialized:
            #pycoclib.pico_stack_init()
            FDPicoTCPAdapter.initialized = True
```

```
...

# In FDPicoTCP createDevice is not needed, ignore all calls to it
def createDevice(self, device_type, device_name = None,
    device_path = None):
     result = None
     return result

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

2.3.2 La select del Python

La funzione `select` del Python svolge lo stesso compito di quella in qualsiasi altro linguaggio, ma si aspetta di ottenere dei File Descriptor validi da controllare. Nel caso di PycoTCP, tutte le librerie non forniscono file descriptor validi o ne fanno un uso che confonde la funzione select originale. Questo è un problema quando si cerca di fare il *wrapping* di qualsiasi script che utilizza la `select` o la `poll`, che comprendono qualsiasi libreria WSGI per Python.

Il problema poteva essere risolto in due modi:

Sostituzione del modulo `select.py` in modo analogo alla sostituzione di `socket.py`. Questo avrebbe richiesto una riscrittura ad-hoc del modulo che avrebbe potuto rompere qualsiasi utilizzo differente della stessa libreria, perciò non si è scelta questa soluzione.

Creazione di File Descriptor fittizi da restituire nel momento in cui si richiede il file descriptor di un Socket. Questi sono stati creati attraverso l'uso di `pipe`. Le `pipe` sono state riempite e svuotate in concordan-

za con gli eventi dei `socket` in modo da riprodurre il comportamento voluto dalla funzione `select`.

Si è deciso di utilizzare il secondo metodo per evitare di implementare di nuovo tutte le parti critiche del modulo `select.py`.

2.4 Futuri sviluppi

Sebbene gli obiettivi siano stati raggiunti, il pacchetto non è ancora completo. Sarà disponibile liberamente su Github[2] in modo che chiunque possa contribuire. Tra le parti che richiedono ulteriore lavoro vi sono:

Testing su più piattaforme e Script Python per verificare che il modulo socket.py sostitutivo si adatti a più situazioni possibili;

Pulizia del codice sopratutto all'interno degli Adapter. Alcune parti sono state scritte durante la ricerca di una soluzione a un problema, lasciando anche sezioni necessarie solo al debugging;

Pubblicazione nell'indice Pip permettendo a chiunque di installare il pacchetto PycoTCP con un semplice "pip install pycotcp";

Semplificare le funzioni avanzate di PicoTCP rendendole utilizzabili con una sola chiamata in funzionamento esplicito;

Migliorare la conformità al PEP 8, le linee guida per lo stile del codice in Python [8];

Rimuovere la dipendenza a PicoTCP integrando la libreria usando CFFI invece di un modulo esterno scritto in C.

[2] `https://github.com/LuigiPower/pycotcp`

Capitolo 3

Casi d'uso

3.1 Libreria WSGI Python

Il primo tipo di utilizzo, abbastanza naturale, è la creazione di semplici Web Server che vivono all'interno della rete Internet of Threads (e quindi nel Virtual Distributed Ethernet). Per far questo sarebbe stato necessario creare una libreria che gestisse le comunicazioni HTTP, con tutte le complessità necessarie a renderla robusta e funzionante nel tempo.

Inoltre, dato che le librerie per l'IoTh sono tre ed eterogenee in natura, sarebbe stato necessario creare tre diverse parti della libreria, ognuna dedicata a una delle tre librerie IoTh.

Con PycoTCP basta prendere una libreria WSGI esistente e richiamare lo script di esecuzione del server attraverso `pyco-wrapper`, specificando quale delle librerie utilizzare:

Listing 3.1: Esecuzione dello script con pyco-wrapper

```
pyco-wrapper pynative_sample.py -a 10.40.0.86 -d web0 -f
    /tmp/vde.ctl -l picotcp -c
# Il flag -c richiede i permessi per la modifica delle reti
```

O creare il contesto VDE e in un secondo momento utilizzare PycoTCP:

Listing 3.2: Esecuzione dello script con pyco-wrapper e pyco-vde-setup

```
pyco-vde-setup start 10.40.0.1/24 /tmp/vde.ctl # Richiede i
    permessi

pyco-wrapper pynative_sample.py -a 10.40.0.86 -d web0 -f
    /tmp/vde.ctl -l picotcp
```

A questo punto basta accedere all'URL specificato nell'esecuzione di `pyco-wrapper` per visualizzare il sito web.

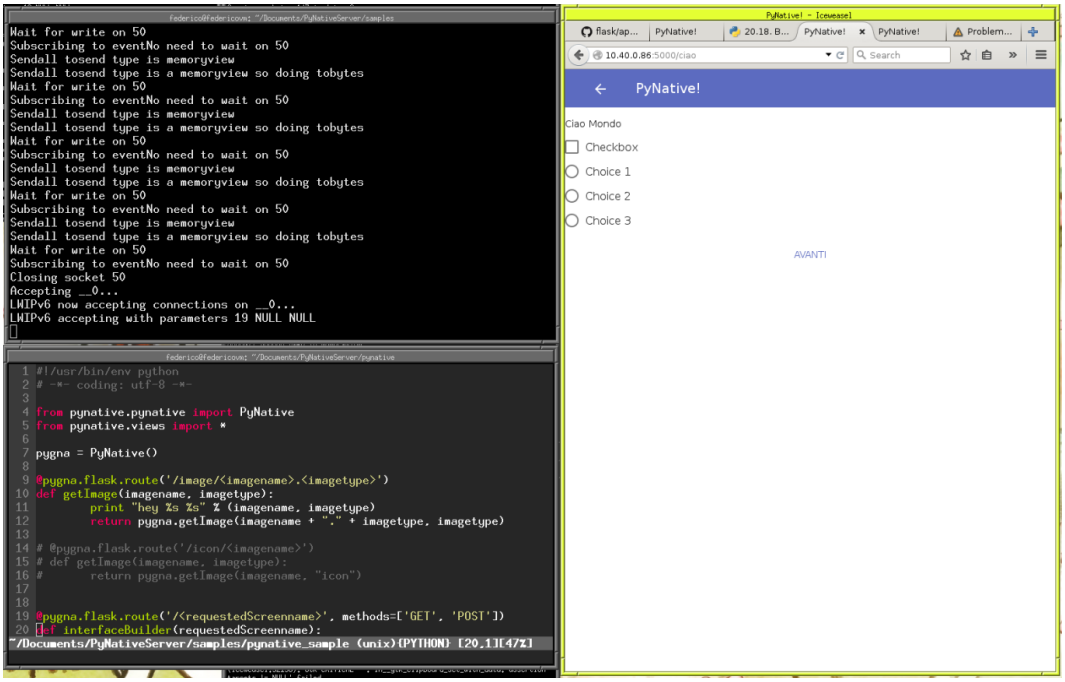

Figura 3.1: Esempio di esecuzione di un Web Server basato su Flask con `pyco-wrapper`

Conclusioni

Il modulo Python ottenuto alla fine del processo di sviluppo raggiunge tutti gli obiettivi prefissati.

Ora chiunque può sperimentare con l'idea dell'Internet of Threads e utilizzarlo per le proprie idee e ricerche.

Utilizzandolo con libpamnet è possibile saltare la necessità di avere permessi di amministratore fornendo all'utente una 'sandbox' di rete dove lavorare senza far danni.

Una volta sperimentato l'utilizzo dell'IoTh con PycoTCP, è consigliato scrivere le proprie applicazioni in C, usando le librerie adattate da PycoTCP.

Appendice A

CFFI e ctypes

CFFI e ctypes sono entrambi moduli per l'integrazione di una libreria C in uno script Python. Nonostante la funzionalità sia la stessa, gli obiettivi sono differenti.

ctypes è un modulo già incluso tra quelli standard del Python. Questo lavora soltanto al livello *ABI* (Application Binary Interface) e permette di caricare una libreria dinamica a runtime ed effettuare chiamate su di essa.

CFFI richiede l'installazione attraverso PIP, dato che si tratta di un modulo di terze parti. Permette di lavorare sia a livello *ABI*, sia a livello *API* (Application Programming Interface) grazie all'utilizzo del compilatore C per validare e collegare i costrutti scritti in linguaggio C. Permette ugualmente di caricare librerie dinamiche a runtime, ma anche di generare un modulo C esterno compilato a parte per il funzionamento a livello API.

Entrambi potrebbero sembrare adatti allo sviluppo di PycoTCP, ma ctypes ha una grave mancanza nella gestione delle `struct`, che rende il suo utilizzo molto scomodo per un progetto di questo tipo.

Per utilizzare una `struct` definita nella libreria utilizzata è necessario definirla, campo per campo, all'interno del Python stesso:

Listing A.1: Esempio di struct in ctypes

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
```

Nel caso di librerie moderatamente complicate si dovrebbero riscrivere tutte le definizioni delle strutture all'interno del modulo Python, rendendo il pacchetto effettivamente dipendente dalla versione delle librerie sottostanti. Era necessario un metodo che ignorasse le complessità interne delle strutture ma potesse comunque modificarne dei campi quando necessario. Per questo si è preferito utilizzare CFFI. Mediante la sua modalità di funzionamento API out-of-line è stato possibile evitare la definizione delle strutture (che vengono completate dal compilatore C stesso) e la loro modifica avviene tramite funzioni apposite definite nel file che genera il modulo C esterno.

Ad esempio, per la libreria lwipv6:

Listing A.2: lwipv6build.py

```
from cffi import FFI
ffi = FFI()

ffi.set_source("_pycolwipv6",
       """
           #include <stdio.h>
           #include <lwipv6.h>
           [...]
           static unsigned int ip4_addrx(struct ip_addr* addr,
               unsigned int a, unsigned int b, unsigned int c,
               unsigned int d)
           {
               return IP4_ADDRX(addr, a, b, c, d);
```

```
            }
            [...]
        """,
        libraries=['vdeplug', 'lwipv6'],
        extra_objects=['/usr/lib/liblwipv6.so']
        )

ffi.cdef("""
    extern "Python" void event_callback(void* arg);
    struct __sigset_t {
        ...;
    };
    typedef struct __sigset_t sigset_t;
    struct stack *lwip_stack_new(void);
    static struct sockaddr_in* create_sockaddr(int family, int
        port, char* addr);
    static lwip_connectw(int socket_fd, char* string_address,
        unsigned int port_number, int family);
    [...]
        """)

if __name__ == "__main__":
    ffi.compile()
```

I tre puntini all'interno delle definizioni delle `struct` dicono al compilatore di cercare i campi rimanenti all'interno della struttura nominata.

L'esecuzione di questo script crea il file _pycolwipv6.c, per poi richiamare il compilatore C creando il file _pycolwipv6.o, che poi sarà integrato nel modulo Python attraverso il setup.py:

Listing A.3: setup.py

```
...
    setup_requires=['cffi>=1.0.0'],
    cffi_modules=['pycotcp/fdpicobuild.py:ffi',
        'pycotcp/lwipv6build.py:ffi'],
    install_requires=['cffi>=1.0.0']
...
```

Appendice B

Creazione del modulo installabile Python

Un modulo python moderatamente complesso può essere installato in maniera semplice mediante la libreria `setuptools`.

Questa libreria può essere installata usando `pip` come per tutte le librerie del Python, ed è richiesta per il funzionamento dello script di installazione di PycoTCP.

Per trasformare un qualsiasi modulo Python in un pacchetto installabile basta creare varie sottocartelle in cui porre gli script del pacchetto, per poi scrivere uno script `setup.py` posto alla radice del pacchetto stesso.

Il file setup.py specifica quali script devono essere inclusi nel modulo, dove si trova la documentazione e anche eventuali moduli esterni C da utilizzare per l'installazione:

Listing B.1: setup.py

```
from setuptools import setup, find_packages, Extension
with open('README.md') as f:
    readme = f.read()
with open('LICENSE') as f:
    license = f.read()
```

```
m_pyco = Extension('pycoclib',
      include_dirs = ['./pycoclib/include','./pycoclib/headers'],
      libraries = ['vdeplug'],
      library_dirs = ['./pycoclib/lib'],
      extra_objects=["./pycoclib/lib/libpicotcp.a"],
      sources =
          ['./pycoclib/sources/pycoclib.c','./pycoclib/sources/pycoutils.c'])

setup(
    name='pycotcp',
    version='0.9.0',
    description='PycoTCP',
    long_description=readme,
    author='Federico Giuggioloni',
    author_email='federico.giuggioloni@gmail.com',
    url='fedegiugi.noip.me',
    license=license,
    scripts=['scripts/pyco-wrapper', 'scripts/pyco-runner.py',
        'scripts/pyco-vde-setup'],
    zip_safe=False,
    packages=find_packages(exclude=('tests', 'docs')),
    ext_modules=[m_pyco],
    setup_requires=['cffi>=1.0.0'],
    cffi_modules=['pycotcp/fdpicobuild.py:ffi',
        'pycotcp/lwipv6build.py:ffi'],
    install_requires=['cffi>=1.0.0']
)
```

Inoltre, dato che l'installazione di PycoTCP richiede comunque degli ulteriori passaggi per la compilazione delle parti in C, è stato creato un Makefile che si occupa proprio di questo:

Listing B.2: Makefile

```
cleanpicotcp:
   (cd pycoclib/picotcp && make clean)

clean: cleanpicotcp
   rm pycoclib/include || echo "No need to clean
      pycoclib/include..."
   rm pycoclib/lib || echo "No need to clean pycoclib/lib..."
   rm pycoclib/modules || echo "No need to clean
      pycoclib/modules..."

picotcp: clean
   (cd pycoclib/picotcp && make posix core ARCH=shared && make test
      ARCH=shared)
   (cd pycoclib && ln -s picotcp/build/include ./include)
   (cd pycoclib && ln -s picotcp/build/lib ./lib)
   (cd pycoclib && ln -s picotcp/build/modules ./modules)

cffi: picotcp
   (cd pycotcp && python fdpicobuild.py) || echo "FDPICOTCP NOT
      FOUND IN THE SYSTEM, -l fdpicotcp AND FDPicoTCPAdapter WILL
      NOT BE AVAILABLE"
   (cd pycotcp && python lwipv6build.py) || echo "LWIPV6 NOT FOUND
      IN THE SYSTEM, -l lwipv6 AND Lwipv6Adapter WILL NOT BE
      AVAILABLE"

install: cffi
   python setup.py install --user

globalinstall: cffi
   sudo python setup.py install
```

Bibliografia

[1] Renzo Davoli. *Internet of Threads*. www.cs.unibo.it/~renzo/papers/2013.iciw.pdf, 2013

[2] Virtual Square Team. *Virtual Square Wiki*. http://wiki.v2.cs.unibo.it/wiki/index.php?title=Main_Page

[3] Virtual Square Team. *LWIPV6*. http://wiki.v2.cs.unibo.it/wiki/index.php/LWIPV6

[4] Filippo Morselli. *FDPicoTCP*. https://github.com/exmorse/fd_picotcp, 2016

[5] Intelligent-Systems Altran. *picoTCP*. https://github.com/tass-belgium/picotcp

[6] Armin Ronacher. *Welcome to Flask*. http://flask.pocoo.org/docs/0.11/, 2015.

[7] Python Software Foundation. *socket – Low-level networking interface*. https://docs.python.org/2/library/socket.html

[8] Python Software Foundation. *PEP 8 – Style Guide for Python Code*. https://www.python.org/dev/peps/pep-0008/

[9] Armin Rigo, Maciej Fijalkowski. *CFFI documentation*. http://cffi.readthedocs.io/en/latest/

[10] Python Software Foundation. *ctypes – A foreign function library for Python*. https://docs.python.org/2/library/ctypes.html

[11] Python Software Foundation. *Should I use Python 2 or Python 3 for my development activity?*. https://wiki.python.org/moin/Python2orPython3, 2016

Ringraziamenti

Ringrazio amici e famiglia che continuano a sopportarmi nonostante io vada sempre più lontano per studiare. Sopratutto la famiglia dato che ancora non porto a casa un soldo.