Programmazione Avanzata

High level networking

1

Il protocollo XML-RPC

- Nel protocollo XML-RPC, un client effettua una RPC (remote procedure call) inviando una richiesta HTTP ad un server che implementa XML-RPC.
- Un'invocazione può avere più parametri ed un risultato.
  - Il protocollo definisce alcuni tipi di dati per i parametri e per il risultato.

2

Applicazioni XML-RPC

- Comunicare su rete utilizzando protocolli a basso livello significa dover impacchettare i dati al momento dell'invio e spacchettarli alla destinazione e infine svolgere determinate operazioni in risposta ai dati inviati.
  - Questo processo può essere noioso e suscettibile di errori.
- Una soluzione consiste nell'usare una libreria RPC. Ciò consente di inviare semplicemente il nome di una funzione e gli argomenti e demandare alla libreria RPC il compito di impacchettare, spacchettare, e svolgere l'operazione.
- Le librerie che implementano il protocollo XML-RPC codificano i dati (cioè le funzioni e i loro argomenti) in formato XML e usano HTTP come meccanismo di trasporto.

3

Applicazioni XML-RPC

- La libreria standard di Python include i moduli xmlrpc.server e xmlrpc.client che forniscono supporto per il protocollo.
- Il protocollo in se stesso è neutro rispetto al linguaggio di programmazione cioè tale da permettere ad un server XML-RPC scritto in Python di essere accessibile ai client XML-RPC scritti in un qualsiasi linguaggio che supporta il protocollo. Allo stesso modo, è possibile scrivere in Python client XML-RPC che si connettono a server XML-RPC scritti in altri linguaggi.
- Il modulo xmlrpc permette di usare estensioni specifiche per Python, come ad esempio, passare oggetti Python. L'uso di queste estensioni impone però di usare client e server scritti in Pyhton.
  - Gli esempi che vedremo non fanno uso di queste estensioni.

4

Un wrapper

- I dati che i client e i server devono gestire sono incapsulati dal modulo Meter.py
- Questo modulo fornisce una classe Manager che immagazzina letture di contatore e fornisce metodi per permettere ai lettori di fare il login, acquisire compiti e sottomettere risultati.
- Questo modulo potrebbe facilmente essere sostituito da un altro concepito per gestire dati totalmente differenti.

5

Un wrapper

- SessionID è usata per assegnare a ciascuna login (avvenuta con successo) un'unica ID di sessione.
- La classe mantiene anche due dizionari statici: uno con coppie (chiave,valore) del tipo (session ID,username) e l'altro con con coppie (chiave,valore) del tipo (numero del contatore, lettura del contatore).
- In un contesto più realistico, i dati sono probabilmente memorizzati in un file DBM o in un database che potrebbero essere facilmente sostituiti dai dizionari usati in questo esempio.

```
class Manager:

    SessionId = 0
    UsernameForSessionId = {}
    ReadingForMeter = {}
```

6

Un wrapper

- I lettori devono fare login con la propria username e password prima di acquisire job e sottomettere risultati.
- Se username e password sono corrette, la funzione login() restituisce l'ID di sessione (unica per quell'utente) e il vero nome dell'utente
- A ciascuna login di successo viene assegnata un'unica ID che viene aggiunta al dizionario UsernameForSessionId
- Tutti gli altri metodi richiedono una ID di sessione valida

```
    def login(self, username, password):
        name = name_for_credentials(username, password)
        if name is None:
            raise Error("Invalid username or password")
        Manager.SessionId += 1
        sessionId = Manager.SessionId
        Manager.UsernameForSessionId[sessionId] = username
        return sessionId, name
```


Un wrapper

- name_for_credentials() computa il valore hash SHA-256 della password passata come argomento e se la coppia (username, valore hash) è nel dizionario privato del modulo _Users, restituisce il nome reale dell'utente altrimenti restituisce None.
- Le chiavi _User del dizionario _Users consistono di tuple ciascuna delle quali è formata dello username e del valore hash della password. I valori associati alle chiavi sono i nomi reali degli utenti.

```
_User = collections.namedtuple("User", "username sha256")
```

```
def name_for_credentials(username, password):
    sha = hashlib.sha256()
    sha.update(password.encode("utf-8"))
    user = _User(username, sha.hexdigest())
    return _Users.get(user)
```

str.encode(*encoding="utf-8", errors="strict"*) restituisce una versione della stringa sotto forma di un oggetto bytes (sequenza immutabile di singoli byte). La codifica di default è 'utf-8'. Il parametro *errors* può essere settato per utilizzare un diverso schema di gestione degli errori, Per default è settato a strict per cui un errore di codifica dà luogo ad un UnicodeError.

Un wrapper

- Nel modulo hashlib.py c'è un costruttore per ciascun tipo di algoritmo hash. Ciascun costruttore restituisce un oggetto per un particolare tipo di hash.
- Il metodo hash.update(*data*) aggiorna l'oggetto con l'oggetto bytes-like passato come argomento. Chiamate ripetute a questo metodo hanno lo stesso effetto di un'unica chiamata con input uguale alla concatenazione di tutti gli argomenti: m.update(a); m.update(b) è equivalente a m.update(a+b).
- hash.hexdigest() restituisce il digest delle stringhe passate fino a quel momento ad update(). Il digest è restituito come una stringa di digit esadecimali.

```
def name_for_credentials(username, password):
    sha = hashlib.sha256()
    sha.update(password.encode("utf-8"))
    user = _User(username, sha.hexdigest())
    return _Users.get(user)
```


9

Un wrapper

- Una volta che un lettore ha effettuato il login, può invocare questo metodo per ottenere il numero di un contatore da leggere.
- Il metodo comincia con il controllare che l'ID di sessione sia valida e se non lo è il metodo _username_for_sessionid() lancia l'eccezione Meter.Error .
- Non disponendo di un database reale di contatori da leggere, ogni volta che un lettore richiede un contatore, di fatto viene creato un finto contatore (ad esempio, "E350718" o "G72168") .
  - Se il contatore non è già presente nel dizionario ReadingForMeter allora viene inserito nel dizionario con valore uguale a None.

random.choice(*seq*) restituisce un elemento scelto in modo casuale nella sequenza seq. Se seq è vuota viene lanciato IndexError.

```
    def get_job(self, sessionId):
        self._username_for_sessionid(sessionId)
        while True: # Create fake meter
            kind = random.choice("GE")
            meter = "{}{}".format(kind, random.randint(40000,
                    99999 if kind == "G" else 999999))
            if meter not in Manager.ReadingForMeter:
                Manager.ReadingForMeter[meter] = None
                return meter
```


10

Un wrapper

- Questo metodo o restituisce la username per l'ID di sessione data o converte una generica eccezione KeyError (dovuta ad un'ID non valida) in un'eccezione Meter.Error.
- è spesso più appropriato usare un'eccezione specifica per l'applicazione in uso piuttosto che un'eccezione built-in.
  - In questo modo è possibile fare il catch di eccezioni specifiche e non si rischia così di catturare eccezioni generiche che avrebbero rivelato errori nella logica del codice.

```
def _username_for_sessionid(self, sessionId):
    try:
        return Manager.UsernameForSessionId[sessionId]
    except KeyError:
        raise Error("Invalid session ID")
```


11

Un wrapper

- Questo metodo accetta un'ID di sessione, un numero di contatore, la data e l'ora della lettura, il valore della lettura (un intero positivo o -1 nel caso in cui non sia stata ottenuta alcuna lettura) e la motivazione per la quale una lettura non è stata presa (che è una stringa non vuota per le letture non andate a buon fine)

```
def submit_reading(self, sessionId, meter, when, reading, reason=""):
    if isinstance(when, xmlrpc.client.DateTime):
        when = datetime.datetime.strptime(when.value,
                "%Y%m%dT%H:%M:%S")
    if (not isinstance(reading, int) or reading < 0) and not reason:
        raise Error("Invalid reading")
    if meter not in Manager.ReadingForMeter:
        raise Error("Invalid meter ID")
    username = self._username_for_sessionid(sessionId)
    reading = Reading(when, reading, reason, username)
    Manager.ReadingForMeter[meter] = reading
    return True
```


12

Un wrapper

- è possibile impostare il server XML-RPC in modo che usi tipi Python built-in ma questo non avviene per default (in questo esempio, non vengono usati i tipi built-in di Python)
  - Ciò significa che il server XML-RPC può servire client scritti in qualsiasi linguaggio che supporta XML-RPC.
  - Lo svantaggio di non usare tipi Python è che gli oggetti date/time sono passati come xmlrpc.client.DateTimes invece che come datetime.datetimes per cui è necessario convertirli in datetime.datetimes.

il metodo datetime.strptime(*date_string*, *format*) crea un oggetto datetime.datetime a partire dalla stringa date_stringa, che rappresenta una data e un'ora, e dalla stringa di formato format.

```
def submit_reading(self, sessionId, meter, when, reading, reason=""):
    if isinstance(when, xmlrpc.client.DateTime):
        when = datetime.datetime.strptime(when.value,
                "%Y%m%dT%H:%M:%S")
    if (not isinstance(reading, int) or reading < 0) and not reason:
        raise Error("Invalid reading")
    if meter not in Manager.ReadingForMeter:
        raise Error("Invalid meter ID")
    username = self._username_for_sessionid(sessionId)
    reading = Reading(when, reading, reason, username)
    Manager.ReadingForMeter[meter] = reading
    return True
```


13

Un wrapper

- Una volta che sono stati controllati i dati, la funzione recupera la username per il lettore la cui ID di sessione è stata passata come argomento.
- La username è quindi usata per creare l'oggetto Meter.Reading che consiste di una named tuple.

```
Reading = collections.namedtuple("Reading", "when reading reason username")
```

- Alla fine viene settata la lettura e viene restituito True invece del valore di default None che, per default, non è supportato dal modulo xmlrpc.server.

```
def submit_reading(self, sessionId, meter, when, reading, reason=""):
    if isinstance(when, xmlrpc.client.DateTime):
        when = datetime.datetime.strptime(when.value,
                "%Y%m%dT%H:%M:%S")
    if (not isinstance(reading, int) or reading < 0) and not reason:
        raise Error("Invalid reading")
    if meter not in Manager.ReadingForMeter:
        raise Error("Invalid meter ID")
    username = self._username_for_sessionid(sessionId)
    reading = Reading(when, reading, reason, username)
    Manager.ReadingForMeter[meter] = reading
    return True
```

14

Un wrapper

- Dopo che un lettore ha sottomesso una lettura, esso potrebbe voler conoscere il proprio stato, cioè quante letture ha fatto e il numero totale di letture gestite fino a quel momento dal server.
- Questo metodo calcola questi numeri e li restituisce in output.

```
    def get_status(self, sessionId):
        username = self._username_for_sessionid(sessionId)
        count = total = 0
        for reading in Manager.ReadingForMeter.values():
            if reading is not None:
                total += 1
                if reading.username == username:
                    count += 1
        return count, total
```


15

Un wrapper

- Questo metodo serve solo per il debugging e rende possibile controllare che tutte le letture effettuate siano state memorizzate correttamente.

```
    def _dump(file=sys.stdout):
        for meter, reading in sorted(Manager.ReadingForMeter.items()):
            if reading is not None:
                print("{}={}@{}[{}]{}".format(meter, reading.reading,
                        reading.when.isoformat()[:16], reading.reason,
                        reading.username), file=file)
```

Le funzionalità fornite dal Meter.Manager, un metodo per il login e i metodi per prelevare e settare i dati, sono tipiche di una classe per il data-wrapping utilizzabile da un server.

16

Il server XML-RPC

- Questa funzione ottiene un hostname ed un numero di porta e crea un Meter.Manager e un xmlrpc.server.SimpleXMLRPCServer invocando setup.

```
def main():
    host, port, notify = handle_commandline()
    manager, server = setup(host, port)
    print("Meter server startup at  {} on {}:{}{}".format(
            datetime.datetime.now().isoformat()[:19], host, port, PATH))
    try:
        if notify:
```


17

Il server XML-RPC

- Se la variabile notify contiene il nome di un file, il server crea un file e scrive un singolo newline in esso.
- Se il server è attivato da un client GUI, il client passa al server un filename.
  - Il client GUI attende fino a quando il file è creato e a quel punto il client sa che il server è attivo. Quindi il client cancella il file e comincia a comunicare con il server.
- Il server comincia a servire e può essere fermato con Ctrl+C o inviando un segnale INT (ad esempio, kill -2 pid su Linux). Ciò dà luogo ad un Keyboard-Interrupt in seguito al quale viene invocata la funzione dump di Manager

```
            with open(notify, "wb") as file:
                file.write(b"\n")
        server.serve_forever()
    except KeyboardInterrupt:
        print("\rMeter server shutdown at {}".format(
                datetime.datetime.now().isoformat()[:19]))
        manager._dump()
```

18

Il server XML-RPC

- xmlrpc.server.SimpleXMLRPCServer(*addr*,*requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

crea una nuova istanza di server.

- La classe xmlrpc.server.SimpleXMLRPCServer fornisce metodi per la registrazione di funzioni che possono essere invocate dal protocollo XML-RPC
- Il parametro *requestHandler* è una factory di istanze di gestori di richieste. Il valore di default è xmlrpc.server.SimpleXMLRPCRequestHandler
- I parametri addr e requestHandler sono passati al costruttore socketserver.TCPServer.
  - La classe SocketServer.TCPServer usa il protocollo Internet TCP che permette di realizzare un flusso continuo di dati tra client e server
  - *class* SocketServer.TCPServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
    - Se *bind_and_activate* è true, il costruttore automaticamente tenta di invocare server_bind() (assegna al socket l'indirizzo desiderato) e server_activate() (attiva il server)
    - Gli altri parametri vengono passati al costruttore della classe SocketServer.BaseServer che è la superclasse di tutti gli oggetti server nel modulo SocketServer.

Il server XML-RPC

- Se *logRequests* è true (il valore di default), le richieste sono loggate;
- I parametri *allow_none* ed *encoding* servono a controllare le risposte XML-RPC che sono restituite dal server.
- Il parametro *use_builtin_types* è passato alla funzione loads()(loads() converte richieste o risposte XML-RPC in oggetti Python) e serve per stabilire quali tipi sono processati quando vengono ricevuti valori data/ora o dati binari.

Il server XML-RPC

- Questa funzione è usata per creare il meter manager e il server.
- Il metodo register_introspection_functions() rende disponibili ai client tre funzioni di introspezione : system.listMethods(), system.methodHelp() e system.methodSignature() (prendono in input il nome di un metodo supportato dal server XML-RPC e restituiscono informazioni su di esse)

```
def setup(host, port):
    manager = Meter.Manager()
    server = xmlrpc.server.SimpleXMLRPCServer((host, port),
            requestHandler=RequestHandler, logRequests=False)
    server.register_introspection_functions()
    for method in (manager.login, manager.get_job, manager.submit_reading,
            manager.get_status):
        server.register_function(method)
    return manager, server
```


Il server XML-RPC

- Il server non ha bisogno di gestire alcuna richiesta speciale e quindi il gestore di richieste creato è il più semplice possibile:
  - discende da xmlrpc.server.SimpleXMLRPCRequestHandler e ha un unico path per identificare le richieste.

```
PATH = "/meter"

class RequestHandler(xmlrpc.server.SimpleXMLRPCRequestHandler):
    rpc_paths = (PATH,)
```

Il client XML-RPC

- Esempio di una tipica sessione interattiva:
- il server meterserver-rpc.py (descritto nelle slide precedenti) deve essere stato attivato prima che cominci questa interazione.

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn't find the meter
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447:
Reason for meter G40447:
$
```


Il client XML-RPC

- L'utente Carol User Carol avvia un client per la misurazione
  - Le viene richiesto di immettere la sua username
  - Dopo che Carol ha premuto Enter, le viene chiesto di immettere la password, cosa che lei fa.
  - Il server la riconosce e le dà il benvenuto mostrando il nome per intero

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn't find the meter
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447:
Reason for meter G40447:
$
```

Il client XML-RPC

- Il client chiede al server un contatore da leggere e invita Carol a immettere la lettura.
- Carol digita la lettura: se immette un numero questo è normalmente accettato; altrimenti se Carol fa un errore (come accade per la seconda lettura) o se la lettura non è valida per qualche ragione, le viene richiesto di inserire una nuova lettura.

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn't find the meter
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447:
Reason for meter G40447:
$
```


25

Il client XML-RPC

- Ogni volta che una lettura (o una motivazione) è accettata, le viene comunicato quante letture ha fatto e quante letture sono state fatte in totale durante la sessione, cioè vengono conteggiate anche le letture fatte da altri utenti che stanno usando il server.
- Se Carol preme Enter senza immettere un lattura, le viene chiesto di digitare la motivazione per la quale non può inserire una lettura. Se lei non inserisce né una lettura né una motivazione, il client termina.

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn't find the meter
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447:
Reason for meter G40447:
$
```

26

Il client XML-RPC

- Questa funzione ottiene prima il nome dell'host e il numero della porta e poi la username dell'utente e la password.
- Quindi la funzione crea un proxy per l'istanza di Meter.Manager usata dal server.
- Una volta creato il proxy manager, la funzione usa il proxy per l'operazione di login e comincia ad interagire con il server.
- Se nessun server è attivo, si ha un'eccezione ConnectionError .

```
def main():
    host, port = handle_commandline()
    username, password = login()
    if username is not None:
        try:
            manager = xmlrpc.client.ServerProxy("http://{}:{}{}".format(
                    host, port, PATH))
            sessionId, name = manager.login(username, password)
            print("Welcome, {}, to Meter RPC".format(name))
            interact(manager, sessionId)
        except xmlrpc.client.Fault as err:
            print(err)
        except ConnectionError as err:
            print("Error: Is the meter server running? {}".format(err))
```

27

Il client XML-RPC

- *class* xmlrpc.client.ServerProxy(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)
- Crea un'istanza di ServerProxy, cioè un oggetto che gestisce la comunicazione con un server remoto XML-RPC.
- Il primo argomento è un URI (Uniform Resource Indicator) ed è normalmente la URL del server.
- Il secondo argomento una factory di transport: per default è un'istanza di SafeTransport per https ed è un'istanza di Tranport altrimenti.
- Il terzo argomento è un encoding. Per default è UTF-8.
- Il quarto argomento è un flag per il debugging.
- I restanti argomenti stabiliscono l'uso del proxy restituito:
- Se *allow_none* è true, la costante Python None è tradotta in XML; per default l'uso di None provoca un TypeError. Questa estensione non è supportata da tutti i client e server,
- Il flag *use_builtin_types* può essere usato per permettere l'impiego di oggetti datetime.datetime per rappresentare date e orari e di oggetti bytes per rappresentare dati binari. Per default, il flag e false.
- Il parametro *headers* è una sequenza di header HTTP da inviare ad ogni richiesta, espressa come sequenza di coppie ciascuna formata dal nome dello header e dal valore.
- Il flag *use_datetime* è obsoleto ed è simile a *use_builtin_types* anche se si applica solo a date/orari.
- context serve a configurare i settaggi SSL della connessione HTTPS sottostante.

28

Il client XML-RPC

- La funzione getuser() del modulo getpass (modulo che contiene le utilità per ottenere username e password) restituisce la username dell'utente loggato in quel momento e questo nome viene usato come username di default.
- La funzione getpass richiede la password e non produce alcun echo in risposta.
- Sia input() che getpass() restituiscono stringhe senza newline alla fine.

```
def login():
    loginName = getpass.getuser()
    username = input("Username [{}]: ".format(loginName))
    if not username:
        username = loginName
    password = getpass.getpass()
    if not password:
        return None, None
    return username, password
```


29

Il client XML-RPC

- Se il tentativo di login va a buon fine, questa funzione viene invocata per gestire l'interazione client-server
- Ciò avviene nel seguente modo:
  - viene ripetutamente acquisito un job dal server (cioè un contatore da leggere)
    - se non viene ottenuto un contatore, il ciclo si interrompe
  - viene poi ottenuta una lettura o una motivazione dall'utente
  - i dati vengono quindi sottomessi al al server
  - si va avanti in questo modo fino a quando accade che l'utente non inserisce né una misura valida né una motivazione.

```
def interact(manager, sessionId):
    accepted = True
    while True:
        if accepted:
            meter = manager.get_job(sessionId)
            if not meter:
                print("All jobs done")
                break
        accepted, reading, reason = get_reading(meter)
        if not accepted:
            continue
        if (not reading or reading == -1) and not reason:
            break
        accepted = submit(manager, sessionId, meter, reading, reason)
```

30

Il client XML-RPC

- Questa funzione gestisce tre casi:
  - l'utente inserisce una lettura valida (un intero)
  - l'utente inserisce una lettura non valida
  - l'utente non inserisce nessuna lettura.
    - Se non viene inserita alcuna lettura, l'utente o inserisce la motivazione o non inserisce alcuna motivazione, nel qual caso l'utente ha finito

```
def get_reading(meter):
    reading = input("Reading for meter {}: ".format(meter))
    if reading:
        try:
            return True, int(reading), ""
        except ValueError:
            print("Invalid reading")
            return False, 0, ""
    else:
        return True, -1, input("Reason for meter {}: ".format(meter))
```


Il client XML-RPC

- Ogni volta che l'utente immette una lettura o una motivazione, questa funzione viene usata per sottometterla al server attaverso il proxy.
- Una volta sottomessa, la lettura o la motivazione, la funzione chiede lo stato (numero di letture sottomesse dall'utente e numero di letture totale)

Una RPC termina restituendo un valore che può essere costituito o da alcuni dati o da un oggetto di tipo Fault o di tipo ProtocolError (descrive un errore nel livello di trasporto sottostante, come ad esempio se il server a cui fa riferimento la URI non esiste)

```
def submit(manager, sessionId, meter, reading, reason):
    try:
        now = datetime.datetime.now()
        manager.submit_reading(sessionId, meter, now, reading, reason)
        count, total = manager.get_status(sessionId)
        print("Accepted: you have read {} out of {} readings".format(
                count, total))
        return True
    except (xmlrpc.client.Fault, ConnectionError) as err:
        print(err)
        return False
```