Implementazione di un sistema di Aste

Gabriele Gemmi
gabriele.gemmi@stud.unifi.it

Febbraio 2017

Introduzione

Per il corso di Reti di Calcolatori dell'anno 2016-2017 è stato richiesto di definire un protocollo per un applicazione client server di un sistema di aste e di implementarlo in un linguaggio a scelta tra Python o Java. Il protocollo doveva rispettare una serie di specifiche lasciando libertà allo studente sulle scelte implementative.
Per garantire la scalabilità del sistema è stato definito un protocollo stateless, i cui messaggi sono basati sul formato JSON. Le specifiche del protocollo sono visibili nella sezione 1.6.4.
Dato che in un sistema di aste le connessioni simultanee e in tempo reale di più client sono un fattore importante, è stato implementato un server multithread. Per evitare problemi di accessi concorrenti, tutte le operazioni sulla memoria sono state bloccate con un "lock". Tra le specifiche del progetto era presente una funzionalità di notifiche. Questa è stata I particolari sull'implementazione sono disponibili nella sezione 2.3

Contents

1 Protocollo 3
1.1 Messaggio di risposta . . . 4
1.2 Autenticazione . . . 4
1.2.1 Sessioni . . . 4
1.2.2 Formato . . . 5
1.3 Gestione delle Notifiche . . . 5
1.3.1 Registrazione . . . 6
1.3.2 Messaggi di notifica . . . 6
1.4 Categorie . . . 6
1.4.1 Registrazione categoria . . . 7
1.4.2 Lista delle categorie . . . 7

1.4.3 Ricerca di una categoria 7
1.5 Prodotti 8
1.5.1 Registra un nuovo prodotto 8
1.5.2 Ricerca di un prodotto 8
1.5.3 Lista tutti i prodotti 9
1.5.4 Messaggio di risposta 9
1.6 Asta 9
1.6.1 Offerta 9
1.6.2 Ritiro di un offerta 10
1.6.3 Chiusura di un'asta 10
1.6.4 Messaggio di risposta 11

2 Implementazione 11
2.1 Server 11
2.1.1 IbaiServer 11
2.1.2 ClientManager 11
2.1.3 model 12
2.2 Client 13
2.2.1 IbaiClient 13
2.3 Librerie utilizzate 13

3 Tests 14
3.1 Test sulle funzionalità 14
3.2 Test sulle notifiche 16

1 Protocollo

E' stato utlizzato JSON come formato per lo scambio di dati in quanto erano disponibili nella libreria standard python i metodi per serializzare e deserializzare gli oggetti.

Il formato di un generico messaggio è:

```
{
  "msg_id": <id>
}
```

La prima cifra del campo msg_id indica il tipo di richiesta, la seconda indica il metodo

- -1: Messaggio di risposta a un comando
- 1: Autenticazione
  - 11: Log in
  - 12: Log out
- 2: Notifiche
  - 21: Registra l'endpoint per le notifiche
  - 22: Messaggio di notifica
- 3: Categorie
  - 31: Registra nuova categoria
  - 32: Ricerca di una categoria
  - 33: Lista tutte le categorie
- 4: Prodotti
  - 41: Registra un nuovo prodotto
  - 42: Ricerca un prodotto
  - 43: Lista tutti i prodotti
- 5: Asta
  - 51: Fai un offerta
  - 52: Cancella la tua ultima offerta
  - 53: Chiudi l'asta

1.1 Messaggio di risposta

Il formato di un generico messaggio di risposta è il seguente.
Se non è esplicitamente indicato un'altro formato per la risposta, allora la risposta rispetterà quest'ultimo.

```
{
  "msg_id": -1
  "response" : <code>
}
```

Campo	Tipo	Descrizione
response	int	'1' in caso di successo '-1' comando non sia valido '0' errore '2' sessione scaduta '3' sessione non valida

Questo messaggio può essere esteso dalle varie funzioni aggiungendo campi e codici di risposta.

1.2 Autenticazione

Il primo scambio di messaggi tra client e server è necessariamente di autenticazione. Viene effettuato utilizzando un username ed una password. Il server mantiene una lista di utenti con relative password, quando il client effettua l'autenticazione vengono confrontati e nel caso corrispondano l'autenticazione è avvenuta. Per evitare di memorizzare le password in chiaro, è stato scelto di memorizzare e trasmettere l'hash md5 della password. Il Client invia al server l'hash md5 della password inserita dall'utente, che lo confronta con quello memorizzato.

1.2.1 Sessioni

Al fine di gestire le sessioni mantenendo un procollo stateless è stato scelto di generare un token di autenticazione ed inviarlo al client. Una volta effettuato il login il server invia il token al client, questo lo utilizza per autenticare tutti i successivi comandi.
Al termine della scadenza il token perde di validità ed è necessario autenticarsi nuovamente. Il token inviato al client è della forma:

```
{
  "username" : <user>
  "expiration": <expiration>
  "signature": <signed proof of autenticity>
```

```
}
```

Campo	Tipo	Descrizione
username	stringa	Username dell'utente
expiration	int	Data e ora di scadenza della sessione (in secondi)
signature	stringa	Codice di verifica del messaggio

Il campo signature è un hash dei precedenti due campi (username e expiration) più una chiave segreta generata dal server. In questa maniera il server può verificare che il token sia autentico senza doverne mantenere una copia in memoria.

1.2.2 Formato

Il messaggio di autenticazione è del formato:

```
{
  "msg_id": 11,
  "user": <username>,
  "pass": <hash of pwd>
}
```

Campo	Tipo	Descrizione
user	stringa	Username dell'utente
pass	stringa	Hash MD5 della password

La risposta è del formato:

```
{
  "msg_id": -1,
  "response": <code>
  "token": <token>
}
```

Campo	Tipo	Descrizione
response	int	codice di risposta
token	json	token per la sessione

1.3 Gestione delle Notifiche

Il server di aste può inviare delle notifiche asincrone ai client per comunicare aggiornamenti di stato. Per esempio il client riceve una notifica quando un'asta viene terminata ed ogni volta che sua offerta viene superata.

1.3.1 Registrazione

Per ricevere delle notifiche asincrone il client rimane in ascolto su una determinata porta, questa viene comunicata insieme al proprio indirizzo al server. Per fare questo è disponibile un comando. Di seguito il formato:

```
{
  "token": <token>,
  "msg_id": 21,
  "host": <hostname>,
  "port": <port>
}
```

Campo	Tipo	Descrizione
token	json	Token per la sessione
host	string	Hostname del client
port	int	Porta alla quale il client riceve le notifiche

1.3.2 Messaggi di notifica

Un messaggio di notiica, inviato dal server al client, rispetta il formato:

```
{
  "msg_id": 22,
  "code": <notification code>,
  "text": <human readable meaning>
}
```

Campo	Tipo	Descrizione
code	int	codice di notifica: -1 Notifiche OK 1 Asta chiusa 2 Hai vinto 3 Offerta superata
text	string	Messaggio di notifica

1.4 Categorie

I prodotti in vendita nel sistema di aste sono suddivisi per categorie merceologiche. E' possibile effettuare varie operazioni sulle categorie: Ogni categoria è identificata in maniera univoca dal suo nome.

1.4.1 Registrazione categoria

Attraverso il seguente comando è possibile aggiungere una nuova categoria
Il nome della categoria è utilizzato per identificarla.

```
{
  "token": <token>,
  "msg_id": 31,
  "category": <category name>
}
```

Campo	Tipo	Descrizione
category	string	Nome della categoria, univoco.

1.4.2 Lista delle categorie

Attraverso il seguente comando è possibile ottenere una lista delle categorie

```
{
  "token": <token>,
  "msg_id": 33
}
```

Il formato della risposta è:

```
{
  "msg_id": -1,
  "response": <code>
  "categories":
  [
    <category1>,
    <category2>
  ]
}
```

1.4.3 Ricerca di una categoria

Attraverso il seguente comando è possibile ricercare tra le categorie per nome

```
{
  "token": <token>,
  "msg_id": 32,
  "category": <category name>
}
```

Campo	Tipo	Descrizione
category	string	Espressione da ricercare.

Il formato della risposta è lo stesso del precedente comando.

1.5 Prodotti

All'interno delle categorie merceologiche sono presenti i prodotti. E' possibile aggiungere nuovi prodotti, ricercare i prodotti per nome o visualizzarli tutti.

1.5.1 Registra un nuovo prodotto

Il seguente comando aggiunge una nuova asta per il determinato prodotto.

```
{
  "token": <token>,
  "msg_id": 41,
  "category": <cat_name>,
  "product": <prod_name>,
  "price": <price>
}
```

Campo	Tipo	Descrizione
category	string	categoria alla quale appartiene il prodotto.
product	string	nome del prodotto (univoco per categoria).
price	float	prezzo di partenza dell'asta.

1.5.2 Ricerca di un prodotto

Attraverso questo comando è possibile ricercare un prodotto all'interno di una categoria.

```
{
  "token": <token>,
  "msg_id": 42,
  "category": <category name>
  "product": <product name>
}
```

Campo	Tipo	Descrizione
category	string	Categoria alla quale appartiene il prodotto.
product	strin	Espressione da ricercare come nome del prodotto.

1.5.3 Lista tutti i prodotti

```
{
  "token": <token>,
  "msg_id": 43,
  "category": <category name>
}
```

Campo	Tipo	Descrizione
category	string	categoria della quale si vuole la lista di prodotti.

1.5.4 Messaggio di risposta

La risposta del server eredita il formato base, introducendo un nuovo codice di risposta:

Campo	Tipo	Descrizione
response	int	'4' categoria non esistente

1.6 Asta

Una volta che i prodotti sono stati inseriti i vari utenti posso partecipare ad un asta offrendo un prezzo per l'oggeto.
L'asta termina quando si raggiunge il tempo limite, o quando il venditore decide di chiduerla. In entrambi i casi i partecipanti verranno notificati E' anche possibile ritirare un offerta, ma solo nel caso sia l'offerta più alta.

1.6.1 Offerta

L'utente può effettuare un offerta per un prodotto già inserito.
Al fine di evitare che un utente possa far salire arbitrariamente il prezzo di un prodotto non è possibile inviare offerte successive da parte dello stesso utente..
Nel caso l'offerta sia inferiore all'ultima offerta registrata allora sarà restituito un errore.

```
{
  'token': <token>,
  'msg_id': 51,
  'category': <category name>,
  'product': <product name>
  'price': <price>
}
```

Campo	Tipo	Descrizione
category	string	nome della categoria
product	string	nome del prodotto
price	float	valore dell'offerta

1.6.2 Ritiro di un offerta

E' possibile ritirare un offerta fatta ad un'asta, ma solo nel caso sia ancora l'offerta più alta.

```
{
  'token': <token>,
  'msg_id': 52,
  'category': <category name>,
  'product': <product name>
}
```

Campo	Tipo	Descrizione
category	string	nome della categoria
product	string	nome del prodotto

1.6.3 Chiusura di un'asta

```
{
  'token': <token>,
  'msg_id': 53,
  'category': <category name>,
  'product': <product name>
}
```

Campo	Tipo	Descrizione
category	string	nome della categoria
product	string	nome del prodotto

1.6.4 Messaggio di risposta

La risposta del server eredita il formato base, introducendo nuovi codici di risposta:

Campo	Tipo	Descrizione
response	int	’4’ categoria non valida
response	int	’5’ asta non valida
response	int	’6’ offerta non valida
response	int	’7’ utente non autorizzato
response	int	’8’ asta chiusa senza vincitori

2 Implementazione

Questo sisistema è stato implementato in Python 2.7, si divide in Client e Server, i quali sono composti da varie classi:

2.1 Server

Il server è stato implementato utilizzando più Thread. Il thread principale si occupa solamente di accettare le connessioni dagli utenti e smistarle. Per ogni connessione viene fatto partire un thread ”ClientManager” che si occupa della comunicazione con il client.
E’ stata scelta la porta 7652 per la comunicazione tra client e server. Per le notifiche la porta viene scelta casualmente dal sistema operativo.

2.1.1 IbaiServer

Questa è la classe principale del Server. Contiene le funzioni di inizializzazione del server e del database. Contiene il loop principale all’interno del quale accetta le connessioni dai client e le passa alla classe ClientManager. 1.4.2 1.4.3 1.5.2 1.5.3

2.1.2 ClientManager

Questa classe esegue la maggior parte delle operazioni relative ai client. Eredita la classe thread e definisce un metodo ”run” che viene eseguito all’avvio del thread.
Questo metodo è un loop di iterazione che riceve i comandi dal client. Una volta ricevuto un comando viene passato alla funzione “read_command” che lo interpreta ed esegue la funzione appropriata. Le altre funzioni sono funzioni specifiche per ogni comando definito nel protocollo: sell, buy, bid, etc.
Sono stati implementati tutti i comandi definiti nel protocollo eccetto i comandi definiti nelle sezioni:

Sincronizzazione I vari threads del server accedono contemporaneamente alla memoria condivisa del thread principale. Qui sono memorizzati gli utenti, le aste e le categorie. Per evitare problemi di accessi concorrenti alla memoria, tutte le funzioni che vi accedono sono decorate con una funzione detta “Synchronizer” che impedisce a due thread di eseguire un accesso contemporaneamente. Il blocco è effettuato per mezzo di un lock.

2.1.3 model

Modelli delle strutture dati utilizzate:

Auction Questa classe rappresenta un asta. Gli attributi di questo oggetto sono:

- name: Nome dell’oggetto in vendita (Case sensitive)
- price: Prezzo di base dell’asta
- owner: Utente che ha messo in vendita l’oggetto
- users: Lista di utenti che hanno partecipato all’asta
- bids: Lista di offerte con i relativi utenti

I metodi della classe sono:

- bid(self, price, user): Inserisce un offerta dell’utente "user" e con prezzo "price"
- unbid(self, user): Rimuove l’ultima offerta dell’utente "user"
- winner(self): Ritorna l’utente che ha fatto l’ultima offerta più alta
- close(self, user): Se user corrisponde all’attributo "owner", chiude l’asta e notifica il vincitore.
- __notify_all(self, code, msg): Notifica tutti gli utenti iscritti all’asta con un messaggio

Category Questa classe rappresenta una categoria. I suoi attributi sono:

- name: Nome della categoria merceologica
- auctions: Lista di aste oggetti facenti parte della categoria

I suoi metodi sono:

- add_auction(self, auction): Aggiunge un oggetto alla categoria
- del_auction(self, auction): Rimuove un oggetto dalla categoria
- search_auction(self, name): Cerca un oggetto per nome nella categoria

User Questa classe rappresenta un utente, i suoi attributi sono:

- name: Nome utente
- password: Hash MD5 della password dell'utente
- date: Data di nascita dell'utente
- remote_port: Porta remota per l'invio delle notifiche
- remote_host: Host remoto per l'invio delle notifiche

I suoi metodi sono:

- update_notif_socket(self, host, port): Aggiorna l'hostname e la porta dell'endpoint remoto per l'invio delle notifiche
- notify(self, code, msg): Notifica l'utente con un messaggio e un codice di notifica

Exceptions Qua sono definite tutte le eccezioni personalizzate usate nel programma.

2.2 Client

2.2.1 IbaiClient

Questa classe definisce un interfaccia client per collegarsi al server ed eseguire i comandi. E' stata implementata per semplificare lo svolgimento dei test. Si divide in due parti:
La parte principale contiene tutti i metodi per interfacciarsi al server (register, sell, buy, unbuy, etc.) e i relativi metodi per interpretare la risposta del server.
La seconda parte invece è un thread che sta in ascolto e riceve le notifiche. Quando viene chiamata la funzione "listen_notify" dal thread principale, viene creato un secondo thread (questo), che sta in ascolto fino a che non termina il programma.

2.3 Librerie utilizzate

Per scrivere il programma sono state utilizzate alcune librerie di terze parti. Tutte sono presenti nel set di librerie standard di python:

- **json**: Questa libreria viene utilizzata per serializzare e de-serializzare i messaggi in formato json inviati sulla socket.
- **hmac**: Questa liberia viene utilizzate per generare la firma presente nel token di autenticazione
- **socket**: Questa libreria gestisce la comunicazione di rete

- **threading**: Questa libreria gestisce i thread paralleli, sia lato server (gestione contemporanea di più client) che lato client(gestione delle notifiche asincrone)
- **time**: Questa liberia viene usata per controllare la scadenza dei token di autenticazione
- **Queue**: Questa libreria viene usata per la comunicazione tra i thread (lato client). In particolare è usata per memorizzare i messaggi di notifica non ancora processati.
- **haslib**: Questa libreria viene utilizzata per generare l'hash MD5 della password utente
- **random**: Questa libreria viene utilizzata per generare la chiave privata del server.
- **unittest**: Questa libreria viene utilizzata per il testing dell'applicazione

3 Tests

I test sono contenuti nel file tests.py Può essere eseguito con il comando "nosetests" oppure "python tests.py"

3.1 Test sulle funzionalità

Connessione :

def `test0_connection(self):`

Questo test verifica che il server sia raggiungibile e accetti le connessioni.

Login :

def `test10_empty_login(self):`

Questo test verifica che un login con user e password vuoti sia gestito correttamente dal server

def `test11_login(self):`

Questo test prova l'autenticazione con un utente e una password validi. Fallisce se il server ritorna un codice diverso da 1

Registrazione :

def test3_wrong_register(self):

Questo test esegue prova a registrare una categoria merceologica già presente(“libri”). Da esito positivo se il server ritorna il codice 0

def test4_wrong_register2(self):

Questo test esegue prova a registrare una categoria vuota (“”). Da esito positivo se il server ritorna il codice 0

def test5_register(self):

Questo test registra una categoria merceologica non presente nel sistema (“elettrodomestici”). Ha successo se il server ritorna il codice 1

def test92_register_unlogged(self):

Questo test prova a registrare una categoria merceologica senza autenticazione. Ha successo se il server ritorna 3.

Vendita :

def test6_sell(self):

Questo test inserisce un prodotto in vendita, si aspetta una risposta dal server uguale a 1.

def test7_wrong_sell(self):

Questo test inserisce un prodotto in vendita vuoto (“”). Si attete una risposta dal server uguale a 0.

Offerta :

def test8_sell_bid(self):

Questo test inserisce un nuovo prodotto in vendita e prova a fare un offerta. Il risultato atteso è un fallimento in quanto non è possibile, da specifica, fare un offerta per il proprio prodotto. Risultato atteso 5

def test9_sell_wrongbid(self):

Questo test prova a inserire un offerta per un prodotto non esisteente. Il risultato atteso è 5.

def test93_close_auction_nowinner(self):

Questo test inserisce un prodotto e chiude l’asta. Il risultato atteso è 8, perchè non c’è nessun vincitore.

def test94_fail_close_auction(self):

Questo test effettua un offerta su un oggetto inserito da un'altro utente, dopodichè prova a chiudere l'asta. Il risultato atteso è 7 in quanto l'utente non è autorizzato a chiudere l'asta

3.2 Test sulle notifiche

Quest verificano la funzionalità delle notifiche asincrone.

def `test2_notify(self):`

Quest test verfica che il server risponda correttamente quando si registra l'endpoint di notifica.

def `test991_notif_win(self):`

Questo test effettua la connessione e l'autenticazione con due utenti. Il primo utente crea un prodotto dopodichè il secondo utente effettua un offerta per il prodotto e il primo utente chiude l'asta.

Il test verifica che il secondo utente riceva la notifica con il codice corrispondente alla vincita dell'asta.

def `test992_notif_close_auct(self):`

Questo test esegue la stessa procedura del test precedente, ma verifica che il secondo client riceva il codice corrispondente alla chiusura dell'asta.

def `test993_notif_manybids(self):`

Questo test simula un'asta completa. 1 utente crea l'asta e altri 2 utenti effettuano varie offerte. Il test fallisce se ogni utente riceve i messaggi di notifica corrispondenti al superamento della propria offerta.