Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Eat@
Gestione ristoranti e catering con sistema d'asta tramite un servizio REST e client Android

Relatore:
Chiar.mo Prof.
Fabio Panzieri

Presentata da:
Nicolò Grasso

II Sessione
Anno Accademico 2015-2016

A me stesso,
per ricordarmi di tenere sempre duro

Indice

Capitolo 1

Introduzione

Questa tesi si occupa di discutere tutto ciò che concerne lo sviluppo e la messa in opera di Eat@, un sistema di gestione ristoranti e catering con aste tramite un servizio web di tipo REST [1] (Representational State Transfer) e con client eseguiti su applicazioni per il sistema operativo mobile Android. Si occupa inoltre di mostrare lo stato dell'arte nell'ambito delle aste, citando 3 articoli reperibili su Google Scholar [2].

Eat@ è un sistema che consente ad un utente di consultare vari ristoranti che mettono a disposizione pasti (singoli, composti, per una o più persone) che sono sottoposti ad un'asta al ribasso nel corso del loro periodo di prenotazione fino a quando una singola unità del pasto viene acquistata, congelando il prezzo. Questo consente agli utenti convenienza ed ottimizzazione temporale (grazie alla prenotazione). Anche i ristoranti ne traggono vantaggio, essendo quello di Eat@ un business model emergente e molto sotto i riflettori. Eat@ è in realtà una delle applicazioni che fanno parte della più vasta piattaforma AuCTION, un (per ora ipotetico) portale web d'aste che fornisce servizi ad aziende e imprenditori di varia natura e dimensione. Il portale AuCTION fornisce all'utente diverse istanze di sistemi d'asta, divisi per tipo (inglese, olandese, in busta chiusa e in busta chiusa al secondo prezzo), e Eat@ sarebbe uno dei servizi consultabili.

Eat@ è formato da 3 componenti: due App per il sistema operativo Android, Eat@ e Eat@ Restaurants, che consentono rispettivamente agli utenti di consultare i ristoranti

per vedere ed eventualmente acquistare le loro offerte (con la possibilità di dare una valutazione da una a cinque stelle e scrivere una recensione), e ai ristoranti di creare queste offerte per gli utenti. Entrambe le App possiedono un sistema di gestione account multipli. Il terzo componente, nonché nucleo di tutti servizi offerti alle App, è il server, eseguito in modo replicato (per consentire un ottimale bilanciamento del carico) su 4 macchine nei laboratori Ercolani e Ranzani (due a testa) dell'Università di Bologna. Come detto nelle righe iniziali, il server è di tipo REST, o per meglio dire è conforme alle linee guida REST, riassumibili in 5 principi:

- Identificazione delle risorse
- Utilizzo esplicito dei metodi HTTP (GET, POST, PUT, DELETE) per mappare le azioni CRUD (Create, Read, Update, Delete) sulle risorse. In breve: Create → POST, Read → GET, Update → PUT, Delete → DELETE
- Risorse autodescrittive (nomi, non verbi, possibilmente)
- Collegamenti tra risorse
- Comunicazione *stateless*, cioè ogni richiesta è indipendente e non influenzata dalle precedenti

Inoltre, utilizza un database MySQL [3] ed è stato scritto in linguaggio Javascript, utilizzando il runtime Node.js [4]. Come ulteriore funzione, ogni utente che visualizza i pasti di un ristorante, è fino all'uscita dall'applicazione o ad un cambio di schermata, sincronizzato con un *topic* dedicato a quel ristorante su Firebase [5], in modo che modifiche rilevanti ai pasti (inserimento, eliminazione, modifica od esaurimento) vengano immediatamente comunicate agli utenti con un refresh della schermata. Firebase è una piattaforma online di proprietà di Google dedicata al mondo mobile, che fornisce strumenti ed infrastrutture per aiutare gli sviluppatori ad integrare servizi nelle loro App. Tra i servizi offerti si cita Firebase Cloud Messagging, utilizzato da Eat@ per la sincronizzazione.

Il motivo che ha portato alla creazione di questo progetto è principalmente la scarsità (o

comunque la minor diffusione rispetto ad altri ambiti) di servizi ed applicazioni (specialmente mobili) riguardanti aste, in particolare aste olandesi, che è la tipologia utilizzata da Eat@. L'asta olandese è nient'altro che l'asta al ribasso, dove ad un bene viene assegnato un prezzo iniziale (in genere eccessivamente alto) che in assenza di compratori viene progressivamente abbassato fino ad un prezzo minimo. Il bene viene eliminato dall'asta se allo scadere di un limite inferiore temporale non vi è nessun compratore.

Data la sopracitata non grandissima diffusione di sistemi simili o riconducibili, lo sviluppo di Eat@ è venuto naturale, e si spera che i risultati ottenuti, sia in termini prestazionali che di mera innovazione, possano portare alla nascita di progetti simili e all'espansione di questo argomento.

Il resto del documento è così strutturato: nel prossimo capitolo vi è un'analisi dello stato dell'arte, con analisi e commenti sui sopracitati 3 articoli di Google Scholar, e un'esposizione sullo stato dell'arte delle tecnologie adottate. Nel capitolo 3 si descrive il progetto architetturale, la sua forma ideale, la sua forma effettiva (che ne è un sottoinsieme, ma sufficiente a fini accademici) ed i meccanismi adottati per reagire ad eventuali malfunzionamenti di una delle macchine su cui il server di Eat@ è in esecuzione. Si espongono inoltre anche i punti deboli rimanenti. Nel capitolo 4 sono descritti l'implementazione dell'architettura client-server, con spiegazioni dettagliate delle funzionalità, la valutazione prestazionale di Eat@ con test dei tempi di risposta (anche su accessi concorrenti), e la conseguente validazione dei dati raccolti, comparando con servizi similari. Infine vi è il capitolo delle conclusioni, dove si tirano le somme dell'intero progetto e dove sono esposti possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

In questo capitolo è descritto lo stato dell'arte. Nel paragrafo 2.1 si descrive lo stato dell'arte dei sistemi e delle applicazioni riguardanti aste (in particolare olandesi) utilizzando come riferimento 3 articoli reperiti su Google Scholar, mentre nel paragrafo 2.2 si descrive lo stato dell'arte degli strumenti utilizzati.

2.1 Analisi articoli

Dei tre articoli che verranno esposti, due trattano il ramo delle aste dal punto di vista più teorico e analitico (non mancano però le sperimentazioni), mentre il terzo è più pratico e quindi simile ad Eat@. Per quanto riguarda il primo articolo analizzato nello svolgimento di questa tesi, *Information Effects in Uniform Price Multi-Unit Dutch Auctions* [36], di Joy Buchanan, Steven Gjerstad e David Porter, esso tratta una particolare tipologia di asta olandese, la cosiddetta Asta Olandese a Unità Multiple con Prezzo Uniforme. Questa variazione consiste ovviamente in un'asta olandese al ribasso, dove un gruppo di n utenti competono per un determinato insieme di m elementi, con $m < n$. La particolarità è che il prezzo a cui gli elementi saranno venduti, allo scadere dell'asta, sarà pari all'offerta più bassa (da qui Prezzo Uniforme), anche per chi aveva ottenuto il suo oggetto precedentemente ad un prezzo più alto (in contrapposizione alle aste di tipo *discriminatorio*). Questo tipo di asta viene studiato nei suoi risultati in relazione

alla variazione delle combinazioni di due parametri: dimensione del gruppo e numero di unità rimanenti, confrontando i risultati sperimentali con i valori teorici. Sono state condotte 16 sessioni sperimentali con 128 soggetti. In ognuna, 8 soggetti (divisi casualmente con eguale probabilità in gruppi da 4 o 8) hanno partecipato a 16 aste olandesi, ciascuna con 3 elementi disponibili e valore iniziale di 100 *token*, con decremento di 2 token ogni 2 secondi. Ad ogni soggetto è stato fornito fornito un numero di token iniziale compreso tra 1 e 100, ed è stata data la possibilità di fare un'offerta immediata per il prezzo corrente (*Instant Bid*) o una volta che il prezzo fosse sceso sotto una soglia decisa dall'utente (*Proxy Bid*). Dagli esperimenti si traggono varie conclusioni:

- Mostrare la dimensione del gruppo diminuisce la rendita, abbassando i prezzi con una media del 7%. Questo risultato è l'esatto opposto di quello previsto dalle previsioni basate sui modelli teorici
- Il ricavo maggiore si ha quando né la dimensione del gruppo né le unità rimanenti sono mostrate
- Generalmente, mostrare informazioni riduce la rendita, con il caso più influente in "Dimensione gruppo conosciuta, unità rimanenti sconosciute". Questo è probabilmente dovuto al fatto che conoscendo più informazioni sull'asta a cui si partecipa, si può adottare una strategia più ragionata. Non conoscendo alcun dettaglio, gli utenti tendono a puntare subito, per paura di perdere
- Mostrare le unità diminuisce i prezzi quando la dimensione del gruppo è sconosciuta
- Quando la dimensione del gruppo è nota, mostrare le unità rimanenti non sortisce alcun effetto
- La *Instant Bid* viene utilizzata maggiormente quando le unità rimanenti sono mostrate
- I soggetti partecipanti all'esperimento hanno fatto offerte maggiori a quelle teoricamente previste
- Circa il 20% delle offerte erano equivalenti al numero di token forniti al soggetto

Nel secondo articolo consultato, *Bid pooling in reverse multi-unit Dutch auctions: an experimental investigation* [37], di Philippe Gillen, Alexander Rasch, Achim Wambach e Peter Werner, viene analizzata sperimentalmente l'Asta Olandese Inversa ad Unità Multiple, in cui vari offerenti competono per vendere la loro singola unità ad un compratore che intende acquistare più oggetti. L'asta si conclude quando l'acquirente ha ottenuto il numero desiderato di oggetti. Viene detta "Inversa" perché è il compratore a decidere il prezzo di partenza, che subirà poi un rialzo ad intervalli costanti, fino a quando uno dei venditori non accetterà di vendere a quel prezzo il suo oggetto. Questo costituisce l'esatto opposto della classica asta olandese, dov'è il venditore a decidere il prezzo, sottoposto a ribasso regolare.

La sperimentazione è consistita in 3 sessioni con un totale di 88 soggetti. In ciascuna, sono state condotte 20 aste per ogni gruppo di 4 soggetti con il ruolo di venditori di un singolo bene, riferendosi ad un compratore virtuale con necessità di 3 oggetti. Il prezzo di partenza è stato stabilito in 20 Experimental Currency Units (ECU), incrementato di 5 unità ogni 5 secondi fino alla vendita di 3 beni (e quindi al termine dell'asta) oppure fino al raggiungimento di 100 ECU, a cui l'asta si arresta. Ad ogni incremento di prezzo, i partecipanti sono stati informati delle unità rimamenti da vendere. Le 20 aste di una sessione hanno sempre gli stessi 4 partecipanti, in quanto si è voluto innanzitutto rappresentare il mondo reale, in cui spesso le aste consistono in un ristretto numero di persone interagenti, ma soprattutto perché si è voluto studiare come i partecipanti acquisiscano esperienza ed eventualmente modifichino comportamenti e strategie nel corso di round ripetuti con avversari che imparano a "conoscere". Come per il precedente articolo, si traggono alcune conclusioni dai risultati:

- I prezzi medi di vendita sono sostanzialmente più alti del prezzo di partenza ma vi si avvicinano nel corso del tempo
- Una volta venduto il primo oggetto, la maggioranza delle offerte successive avvengono esattamente a distanza di un solo price step (*bid pooling*)
- L'offerta vincente più alta di un'asta è significativamente correlata alle offerte e alla strategia nell'asta successiva. In più del 40% dei casi, il prezzo più alto di

un'offerta è identico a quello del round precedente

Il secondo punto delle conclusioni tratte dai risultati, fenomeno che prende il nome di *bid pooling*, è probabilmente causato dalla fretta che può affliggere i partecipanti quando vedono che si è iniziato a vendere, ed è quindi finito il periodo di attesa dove si cerca di massimizzare il guadagno. Anche il primo punto si può ricondurre alla fretta, o comunque ad una strategia atta a precedere gli altri venditori, anche sacrificando un possibile profitto maggiore.

Nel terzo ed ultimo articolo, *An Android Application for Online Agri-Auction* [38], di Nirali A. Kansagara, Trupti M. Khurape, Jyoti S. Kamble e Manasi M. Kulkarni, viene discussa e presentata un'applicazione Android per la creazione e gestione di aste di vario genere (inglese, olandese, in busta chiusa e in busta chiusa al secondo prezzo), con target di riferimento le aste relative a prodotti agricoli. L'applicazione è stata creata per facilitare lo svolgimento di aste nell'ambito dell'agricoltura, in quanto consente di vendere e comprare prodotti da qualunque parte del mondo e a qualunque orario, senza dover sottostare a limitazioni storiche come la burocrazia cartacea e il presentarsi al luogo dell'asta (impossibile per molti agricoltori in quanto probabilmente impegnati o residenti in aree rurali poco praticabili). L'App è composta da un classico sistema di account, con schermate di creazione utente e login. Una volta entrati, è presente una home page da cui cercare aste attive, ed è anche presente un menu laterale da cui è possibile accedere a varie funzionalità: creare un'asta su un oggetto, impostando orario di partenza e durata, consultare le proprie aste, create o a cui si sta partecipando o si è partecipato, vedere le aste concluse con successo, e fare logout. Nel consultare la letteratura su Google Scholar dal 2015 in poi per la scrittura di questo capitolo, si è visto come questo articolo sia l'unico rappresentante di articoli relativi ad App Android trattanti aste. È proprio per la scarsezza di testi relativi a questo particolare ambito, che lo sviluppo di Eat@ è avvenuto.

In conclusione, lo sviluppo di Eat@ può sicuramente far tesoro in particolare dei risultati ottenuti ed esposti nel terzo articolo, trattante l'App per l'asta di prodotti agricoli, in

quanto espone che il settore delle aste è ampio, aperto e da approfondire ulteriormente, essendo ancora molto poco diffuso nel settore mobile. Sono molteplici i campi in cui si potrebbero applicare le nozioni esposte da questi tre articoli, ma anche dagli altri disponibili su Google Scholar. Le conclusioni sperimentali dei primi due testi qui analizzati danno un grande aiuto per comprendere il comportamento degli eventuali utilizzatori di Eat@, in modo da poter adattare la piattaforma per rispondere a schemi tipici di offerta e domanda.

2.2 Strumenti utilizzati

Lo sviluppo di Eat@ in tutti i suoi aspetti ha richiesto l'utilizzo di molteplici strumenti. Per quanto riguarda la parte client, le due App sono state sviluppate utilizzando Android Studio [6], IDE (Integrated Development Environment) sviluppato da Google per creare applicazioni per il proprio sistema operativo. Si tratta di uno strumento a dir poco indispensabile, in quanto fornisce controllo errori in tempo reale, un debugger eccellente, anteprime grafiche dell'interfaccia dell'applicazione in sviluppo, e se il PC su cui si sta scrivendo il codice lo supporta, un'intera macchina virtuale in cui testare l'App, se non si vuole eseguire subito il deploy su un vero dispositivo. Per aiutarsi nello sviluppo, si è ovviamente consultata la sezione sviluppatori del sito ufficiale Android [7], dov'è presente una completissima descrizione di classi, metodi ed interfacce delle API Android. Sia Eat@ che Eat@ Restaurants richiedono come requisito minimo le API level 19 (Android 4.4 KitKat). È stato indispensabile anche Stack Overflow [8] (sia per la parte client che server), dove a praticamente ogni problema si è trovata risposta con spiegazioni o interi frammenti di codice.

Lo sviluppo del server (ma anche la scrittura di questo documento) è stato svolto utilizzando l'editor di testo Sublime Text [9], uno dei più completi e attivi disponibili. Come già detto nell'Introduzione, le informazioni elaborate dai server HTTP sono memorizzate in un database MySQL, e si è utilizzato come storage engine il predefinito InnoDB [10]. Tutto il codice del server è basato sul runtime Javascript Node.js, scelto per la sua

semplicità ed efficacia nel poter creare abbastanza velocemente web server scalabili e prestazionalmente ottimi. Per un'approfondita spiegazione su Node.js e un elenco delle sue espansioni utilizzate, si rimanda alla seguente sottosezione.

2.2.1 Node.js

Node.js è un runtime che permette la creazione di web server e strumenti di rete utilizzando il linguaggio di programmazione Javascript, con la possibilità di avvalersi dei cosiddetti *moduli*, file Javascript o JSON, che aggiungono varie funzionalità, come I/O sul file system, networking (HTTP, DNS, TCP/IP, SSL) o funzioni crittografiche. Per importare un modulo, si utilizza l'istruzione *require()*. La gestione dei moduli è affidata al package manager npm [11], che consente da terminale di installare molto facilmente moduli aggiuntivi.

Node.js lavora su un singolo thread, ma è estremamente scalabile ed efficiente nella gestione della concorrenza grazie al cosiddetto *event loop*. Tramite l'utilizzo del design pattern Observer [12] infatti, il main thread di Node.js rimane in attesa del verificarsi di un evento, al seguito del quale viene eseguita (sempre nel main thread) la sua funzione di callback. Gli eventi si trovano nella cosiddetta Event Queue, e vi sono posti ad esempio alla fine di un'operazione di I/O, eseguita parallelamente in un thread separato preso dal thread pool riservato di Node.js, oppure alla fine di un'operazione di networking, eseguita parallelamente in un socket a livello di Sistema Operativo. Il tutto è perciò completamente asincrono, consentendo a Node.js di essere semplice, leggero e veloce. Quanto scritto è chiarito ulteriormente dalla figura 2.1 alla pagina seguente, presa da [13].

Questo approccio event-driven non è esente da difetti ovviamente, ad esempio in caso di un numero molto elevato di operazioni di rete, queste potrebbero incorrere in timeout perché la Event Queue è troppo lunga e l'event loop non riesce ad eseguire la funzione di callback in tempo. Inoltre, essendo le funzioni di callback eseguite nel main thread di Node.js, computazioni di lunga durata e qualsiasi altro task CPU-bound (cioè utilizzanti maggiormente il processore rispetto all'I/O), bloccheranno l'event loop fino al loro

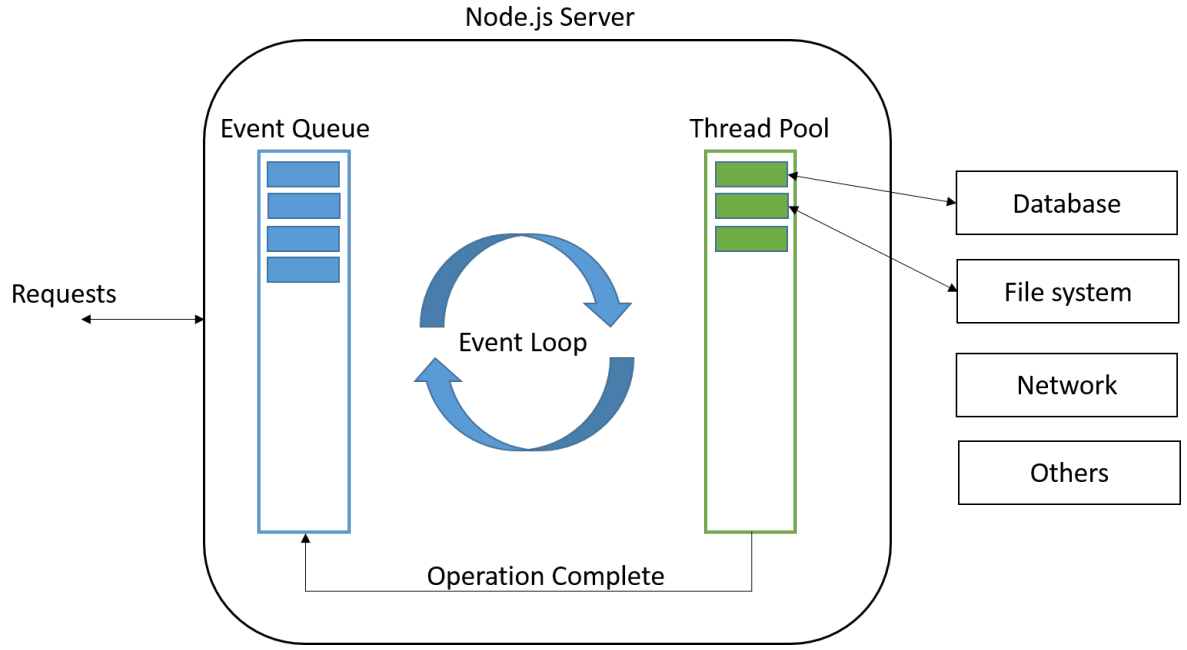

Figura 2.1: Descrizione grafica dell'event loop di Node.js

completamento.

Oltre alle funzionalità core di Node.js, si è utilizzato come modulo principale Express.js [14], lo standard *de facto* dei server framework per Node.js, e strumento indispensabile per definire una API REST correttamente ed ordinatamente. Qui di seguito sono elencati gli altri moduli utilizzati:

- body-parser [15]: gestione dei body delle richieste HTTP
- cron [16]: per impostare i task periodici che il server svolge (reset dati dei pasti a mezzanotte, asta olandese ogni minuto e check status *chief* di un database e *timezoneschief* di un fuso orario 10 secondi prima dell'asta)
- express-fileupload [17]: semplifica l'upload di file. Utilizzato per l'upload dell'immagine profilo di un ristorante al server
- http-proxy [18]: mette a disposizione tutto il necessario per creare un proxy HTTP

- moment [19]: libreria Javascript per parsing e manipolazione di date
- moment-timezone [20]: estensione di Moment.js per il supporto alle Timezone IANA [21] (considera anche l'alternanza ora solare/legale dove necessario, per evitare errori)
- mysql [22]: per interfacciarsi con database di tipo MySQL
- ping [23]: semplice modulo che implementa una funzione di ping. Utilizzato nei controlli per verificare se un server o un proxy è attivo o meno
- request [24]: modulo per semplificare la scrittura di richieste HTTP

Capitolo 3

Progetto architetturale

In questo capitolo si descrive il progetto architetturale di Eat@, sia nella più estesa versione ideale, sia nella ridotta versione effettiva in esecuzione nelle macchine dei laboratori dell'Università di Bologna (paragrafo 3.1). Sono inoltre esposte nel paragrafo 3.2 le tecniche adottate per rimediare ad eventuali malfunzionamenti di una macchina di una sottorete in più casi, ed i punti deboli rimanenti.

3.1 Struttura della rete

Di seguito sono mostrate immagini e spiegazioni relative alla struttura della rete ideale di Eat@, e l'implementazione reale, più ristretta ma equivalente in termini di funzionalità e quasi equivalente in termini di sicurezza e scalabilità.

Nella struttura ideale, mostrata in figura 3.1 nella pagina seguente, gli utenti utilizzando tramite i loro device le App Eat@ o Eat@ Restaurants, si connettono ad un URL comune, e sarà poi compito del DNS del provider internet reindirizzare la connessione ad uno dei proxy possibili. Ogni proxy a sua volta esegue un bilanciamento del carico reindirizzando le richieste HTTP ad uno dei server da esso gestiti, tramite un semplice round-robin (previo controllo che il server a cui bisogna inoltrare la richiesta sia funzionante). Il database non è singolo ma distribuito, real-time, con sincronizzazione immediata e con-

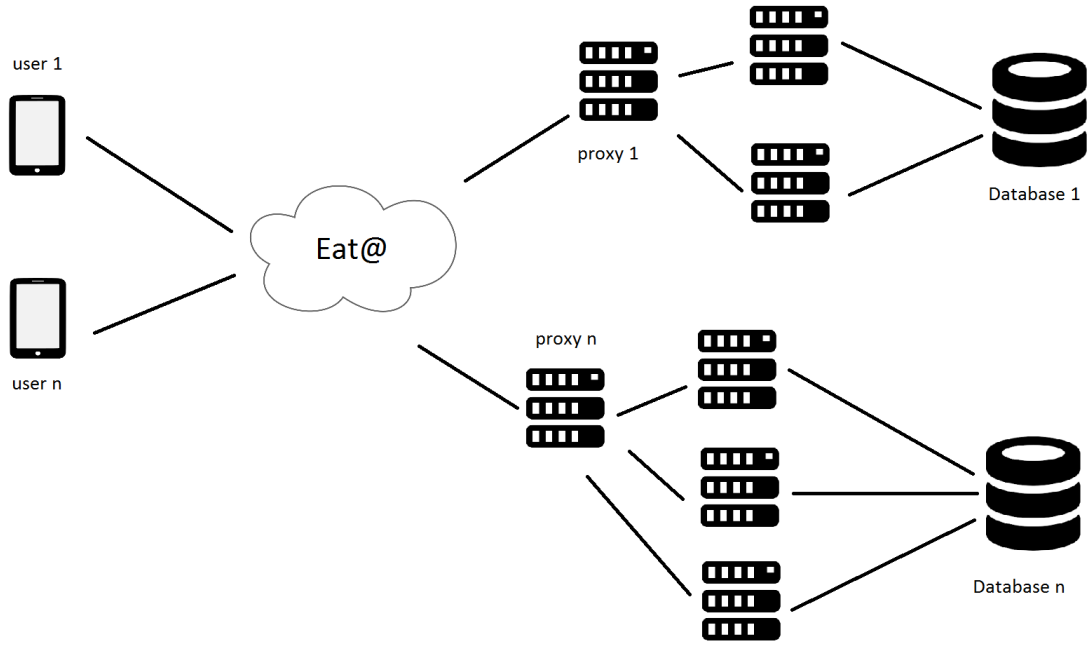

Figura 3.1: Struttura ideale della rete

trollo concorrenza (approfondimento in 5.2.1). Come ulteriore bilanciamento del carico, le connessioni contemporanee dei database non sono illimitate, ma un numero finito a disposizione tramite un *pool* (100 nell'implementazione reale, molti di più in quella ideale dove i mezzi sarebbero più performanti). Ulteriori connessioni eccedenti la capacità del *pool*, si mettono in coda attendendo una connessione venga rilasciata e si liberi un posto.

Per ogni database, deve esserci sempre un solo server adibito alle funzioni di reset pasti e asta olandese (in modo da non eseguire lo stesso task due volte portando a valori errati). Questo server con ulteriori responsabilità viene detto *chief*. In caso di malfunzionamento, viene eletto come *chief* il primo disponibile da un'apposita lista. Inoltre, per i vari database presenti in un certo fuso orario (costituito da più Timezone IANA), deve esserci un solo *timezoneschief*, che deve anche essere *chief* del suo database. Il *timezoneschief* ha come compito aggiuntivo il mandare a Firebase le richieste HTTP relative alla sincronizzazione di tutti i ristoranti (delle sue Timezone gestite) che hanno avuto almeno un pasto decrementato di prezzo in un determinato minuto. Sarà poi Firebase ad occuparsi

di notificare tutti i device iscritti ai *topic* dei vari ristoranti coinvolti.

Il disegno è comunque una semplificazione, occorre precisare che più proxy potrebbero gestire la stessa sottorete, che un singolo proxy può reindirizzare a macchine di diverse sottoreti, e che un database può essere utilizzato da più sottoreti. È però importante, per il funzionamento degli algoritmi di gestione guasti, che i server di una sottorete siano interfacciati con lo stesso database, in modo da avere lo stesso *chief*.

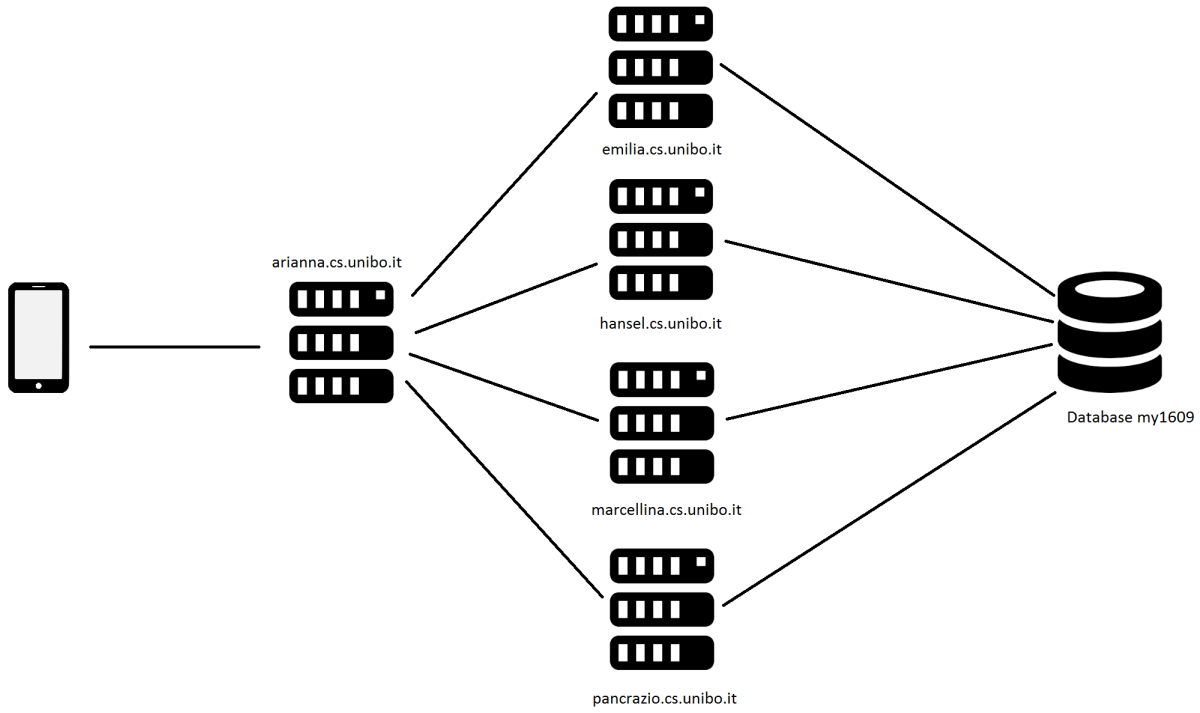

Figura 3.2: Struttura effettiva della rete

La struttura reale, in figura 3.2, funziona allo stesso modo e segue gli stessi principi di quella ideale (valgono perciò le stesse regole e definizioni già esposte). Il proxy è unico e costituito dalla macchina con hostname 'arianna' del laboratorio Ercolani della facoltà di Informatica dell'Università di Bologna. Il database è unico e ha come host 'golem', il web server dell'Università. I server connessi a questo database e che sono "visti" dal

proxy arianna sono 'emilia' e 'marcellina' del Laboratorio Ranzani (emilia inoltre è *chief* e *timezoneschief* di default), 'hansel' e 'pancrazio' del Laboratorio Ercolani. Eat@ utilzza due porte [25] dedicate per le connessioni in ingresso: 9090 per le richieste HTTP al proxy, e 9091 per le richieste HTTP ai server.

3.2 Tecniche per la robustezza della rete

Sono 5 i casi possibili che possono portare alla compromissione delle funzionalità di Eat@:

1. Un proxy smette di funzionare, rendendo una sottorete di macchine e un database potenzialmente inaccessibili se vi era un solo proxy di accesso

2. Uno dei server HTTP gestiti da un proxy smette di funzionare

3. Il *chief* di un database smette di funzionare, rendendo ineffettive le richieste HTTP ad esso dirette ed impedendo che su quel database siano eseguiti il reset dei dati dei cibi a mezzanotte ed il sistema di asta olandese ogni minuto

4. Il *timezoneschief* di una certa Timezone smette di funzionare, rendendo ineffettive le richieste HTTP, bloccando le operazioni sul database di cui è *chief* e non rendendo possibile la sincronizzazione degli utenti con Firebase in caso di asta

5. Un database smette di funzionare, quindi ogni richiesta HTTP inviata alle macchine della sottorete di quel database, non potrà portare ad un effettivo risultato

Dato che l'effettiva implementazione di Eat@ prevede un solo proxy ed un solo database, sono stati risolti solo i casi 2-3-4.

3.2.1 Gestione malfunzionamento server HTTP

Il codice è il seguente:

```
const proxies = [];
var currentServer = 0;
var currentProxy = 0;
var t=1;

function loadBalanceProxy(req, res, flag, target){
    var cur;

    if(!flag){
        if(t<=servers.length){
            cur = currentServer%servers.length;
            currentServer++;
            target = servers[cur];
            t++;
        }
        else{
            cur = currentProxy%proxies.length;
            currentProxy++;
            target = proxies[cur];
        }

        ping.sys.probe((target.split('//')[1]).split(':')[0], function(isAlive){
            loadBalanceProxy(req, res, isAlive, target);
        });
    }
    else{
        t=1;
        proxy.web(req, res, { target: target }, function(err){
            if(err) loadBalanceProxy(req, res, false, '');
        });
    }
}
```

Listing 3.1: Codice Javascript per la gestione dei server HTTP

La gestione è molto semplice: il proxy seleziona uno dei server della sottorete da un apposito array, e verifica se è attivo. Se sì, inoltra la richiesta HTTP, ed in caso di errore riprova selezionando il server successivo. Se il server non è attivo, seleziona il successivo e verifica anch'esso. Se nessun server della sottorete è attivo, inoltra la richiesta ad un

altro proxy (il primo attivo) selezionato dall'apposito array (vuoto nel caso dell'effettiva implementazione di Eat@). L'array proxies, per evitare tentativi inutili, non deve contenere gli indirizzi di altri proxy che gestiscono la stessa sottorete. In questo modo si riesce a gestire sia il malfunzionamento di una macchina della sottorete, sia l'eventuale malfunzionamento di tutte le macchine gestite dal proxy.

N.B. Si noti l'utilizzo della ricorsione, invece che una semplice iterazione (un do ... while sarebbe il più adatto in questo caso), a causa dell'asincronicità di Node.js. L'utilizzo delle funzioni di callback impedisce infatti di ricorrere ai cicli, in quanto questi continuano nel main thread mentre la callback sarà eseguita parallelamente non appena completato il ping.

3.2.2 Gestione malfunzionamento *chief*

Il codice è il seguente:

```
chkCf: function(flag, key, i){
    if(flag){
        CHIEF=key;
        request.get('http://'+servers[Object.keys(servers)[i]]+':'+port+'/chief', {timeout: 1500},
             function(error, response, body){
            if(!error){
                if(response==CHIEF){
                    if(i!=(servers.length-1)) module.exports.chkCf(true, key, i+1);
                }
                else{
                    chiefIndex=0;
                    module.exports.chkCf(false, Object.keys(servers)[0], 0);
                }
            }
            else if(Object.keys(servers)[i]!=CHIEF) module.exports.chkCf(true, key, i+1);
            else{
                chiefIndex++;
                module.exports.chkCf(false, Object.keys(servers)[chiefIndex%servers.length], 0);
            }
        });
    }
    else{
```

```
        ping.sys.probe(servers[key], function(isAlive){
            if(isAlive) module.exports.chkCf(isAlive, key, 0);
            else{
                chiefIndex++;
                module.exports.chkCf(isAlive, Object.keys(servers)[chiefIndex%servers.length], 0);
            }
        });
    }
}
```

Listing 3.2: Codice Javascript per la gestione del *chief*

Se il *chief* attuale non è attivo, tutti i server selezionano il primo attivo che trovano nell'array servers, partendo dal successivo al *chief* attuale (in modo che ogni server nell'array servers possa potenzialmente diventare un *chief*). La variabile *chiefIndex* serve per utilizzare l'array servers come un array circolare. Una volta fatto, e impostata l'apposita variabile CHIEF, ogni server chiede a tutti gli altri se hanno selezionato lo stesso *chief*, se sì allora si ha un nuovo *chief*, altrimenti il procedimento ricomincia (viene resettato anche *chiefIndex*, in quanto si avranno valori discordanti tra i server). Questo previene il caso in cui un server diventi inattivo negli istanti tra una risposta ad uno dei server e la richiesta di un altro, che causerebbe la selezione di due (o più) *chief* differenti.

Questa routine di gestione errori e la routine equivalente per i *timezoneschief*, sono eseguite al cinquantesimo secondo di ogni minuto, quindi 10 secondi prima che scatti il minuto successivo e inizi l'asta olandese, che dipende dalla presenza di *chief* e *timezoneschief* attivi. 10 secondi è stato considerato un tempo sufficiente per lo svolgimento di entrambe le routine.

3.2.3 Gestione malfunzionamento *timezoneschief*

Il codice è il seguente:

```
chkTmzCf: function(flag, url, name, i){
    if(flag){
        TIMEZONESCHIEFURL=url;
```

```
        if(name==hostname) TIMEZONESCHIEF=true;
        request.get('http://'+all[Object.keys(all)[i]]+':'+port+'/timezoneschiefurl', {timeout: 1500},
             function(error, response, body){
            if(!error){
                if(response==TIMEZONESCHIEFURL){
                    if(i!=(all.length-1)) module.exports.chkTmzCf(true, all[Object.keys(all)[i+1]], name, i
                         +1);
                }
                else{
                    TIMEZONESCHIEF=false;
                    timezoneschiefIndex=0;
                    module.exports.chkTmzCf(false, Object.keys(all)[0], '', 0);
                }
            }
            else if(Object.keys(all)[i]!=CHIEF) module.exports.chkTmzCf(true, all[Object.keys(all)[i+1]],
                 name, i+1);
            else{
                TIMEZONESCHIEF=false;
                timezoneschiefIndex++;
                module.exports.chkTmzCf(false, all[Object.keys(all)[timezoneschiefIndex%all.length]], '',
                     0);
            }
        });
    }
    else{
        request.get('http://'+url+':'+port+'/name', {timeout: 1500}, function(error, response, body){
            if(!error) module.exports.chkTmzCf(true, url, response, 0);
            else{
                timezoneschiefIndex++;
                module.exports.chkTmzCf(false, all[Object.keys(all)[timezoneschiefIndex%all.length]], '',
                     0);
            }
        });
    }
}
```

Listing 3.3: Codice Javascript per la gestione del *timezoneschief*

Il funzionamento è assolutamente equivalente alla gestione del *chief*. Si controlla che il *timezoneschief* sia attivo, se non lo è si seleziona il primo server attivo successivo nella lista (array all), utilizzando per la circolarità la variabile *timezoneschiefIndex*. Se l'hostname del nuovo *timezoneschief* selezionato è uguale all'hostname della macchina che sta eseguendo i controlli, viene impostata a *true* la variabile TIMEZONESCHIEF, in modo da poter eseguire la sincronizzazione con Firebase alla fine dell'asta olandese.

È anche qui presente il controllo tra i vari server per verificare se è stato selezionato lo stesso *timezoneschief*, con ripetizione del procedimento in caso di valori discordanti.

Capitolo 4

Implementazione, valutazione e validazione

In questo capitolo si descrivono l'implementazione dell'architettura di Eat@ e le funzionalità sia delle App che del server (paragrafo 4.1), la valutazione prestazionale su tempi di risposta e accessi concorrenti (paragrafo 4.2), e la validazione dei dati ottenuti dai test (paragrafo 4.3).

4.1 Implementazione

Lo svolgimento del progetto si è articolato in più fasi intrecciate: innanzitutto, durante il tirocinio obbligatorio previsto dal Corso di Laurea in Informatica, si è sviluppata una versione preliminare della App Eat@, ancora incompleta in grafica e funzionalità, ma che soprattutto simulava il non ancora esistente server utilizzando il database locale SQLite [26] fornito da Android. Terminato il tirocinio, e dopo aver svolto altri esami, è iniziato lo sviluppo di questa tesi, spiegato, per una maggiore comodità, nelle seguenti sottosezioni separate.

4.1.1 Server

Dopo essersi adeguatamente documentati sul funzionamento di Node.js ed Express.js, è iniziato lo sviluppo del server, inizialmente in Localhost, utilizzando come web server per il PC locale XAMPP [27]. Come prima cosa è stata definita la struttura della directory contenente l'intero server, per evitare di dover scrivere codice e poi doverlo spostare successivamente. La struttura è mostrata nella seguente tabella:

Struttura directory EatAt	Struttura directory proxy	Struttura directory di uno dei server HTTP
|-EatAt | |-images | |-package.json | |-proxyArianna | |-serverEmilia | |-serverHansel | |-serverMarcellina | |-serverPancrazio	|-proxyArianna | |-log |	|-stdout.txt |	|-stderr.txt | |-node_modules |	|-body-parser |	|-cron |	|-express |	|-express-fileupload |	|-http-proxy |	|-moment |	|-moment-timezone |	|-mysql |	|-ping |	|-request | |-proxy.js	|-serverEmilia | |-accounts.js | |-foods.js | |-hub.js | |-log |	|-stdout.txt |	|-stderr.txt | |-node_modules |	|-body-parser |	|-cron |	|-express |	|-express-fileupload |	|-http-proxy |	|-moment |	|-moment-timezone |	|-mysql |	|-ping |	|-request | |-restaurants.js | |-serversync.js

La directory *images* è comune a tutti i 4 server di Eat@, e le directory dei 4 server si trovano nella stessa directory *EatAt*, in quanto su ogni macchina, tramite il login con l'account universitario, si accede alla home directory del proprio utente. Ovviamente, in una vera implementazione, le directory dei server si troverebbero in dischi seperati, con

magari solo images come cartella condivisa in rete. La directory *log* contiene i file dove sono redirezionati stdout e stderr, utili per la manutenzione ed il controllo errori di ogni server. *node_modules* contiene i moduli esterni di Node.js di cui il server ha bisogno per le sue funzionalità. Per quanto riguarda i file contenenti il codice Javascript del server, *proxy.js* contiene la gestione del proxy, quindi il redirezionamento delle richieste HTTP in ingresso ad uno dei 4 server, *accounts.js*, *restaurants.js* e *foods.js* contengono rispettivamente le gestioni delle richieste REST relative ad utenti, ristoranti e pasti. *hub.js* è il file dove viene avviato il server, che include tutti i moduli e avvia i vari handler, infine *serversync.js* contiene il reset dei pasti, l'asta olandese e la gestione dei malfunzionamenti di *chief* e *timezoneschief*. Occorre precisare che essendo l'architettura di Eat@ divisa per Timezone, ed essendo reset ed asta olandese eseguiti solo sui pasti di determinate Timezone, è necessario innanzitutto che le varie Timezone gestite abbiano lo stesso offset da GMT, e che questo offset sia lo stesso dell'orario di sistema, in quanto viene utilizzato l'oggetto Date(). In caso contrario verrebbero resettati pasti disponibili in luoghi in cui non è effettivamente mezzanotte, o decrementati di prezzo pasti in orari incoerenti.

Dopo aver impostato la divisione dei file, è stato deciso lo schema di ogni tabella presente nel database (alcuni campi sono stati aggiunti successivamente quando si è capito che sarebbero stati necessari), schemi di seguito mostrati:

Tabella 4.1: Struttura della tabella accounts

Colonna	Tipo	Null	Predefinito	Collegamenti a	Commenti
id	int(11)	No			
name	text	No			
surname	text	No			
email	text	No			
password	text	No			
salt	text	No			

Tabella 4.2: Struttura della tabella foods

Colonna	Tipo	Null	Predefinito	Collegamenti a	Commenti
_id	int(11)	No			
restID	int(11)	No		restaurants (_id)	Foreign Key
name	text	No			
ingredients	text	No			Opzionale
price	double	No			
priceMin	double	No			
amount	int(11)	No			
beginRes	text	No			Inizio prenotazione
endRes	text	No			Fine prenotazione
begin	text	No			Inizio disponibilità
end	text	No			Fine disponibilità
pricebkcp	double	No			Backup del prezzo massimo
amountbkcp	int(11)	No			Come per pricebkcp
stepPrice	double	No			Decremento di prezzo
stepTime	int(11)	No			Intervallo decrementi
nA	int(11)	No			"never acquired"
timeChange	text	No			Orario a cui decrementare

Tabella 4.3: Struttura della tabella restaurants

Colonna	Tipo	Null	Predefinito	Collegamenti a	Commenti
_id	int(11)	No			
name	text	No			
address	text	No			
city	text	No			
rating	double	No			
nrates	int(11)	No			Numero valutazioni
imgsrc	text	No			Nome del file immagine profilo

Tabella 4.3: Struttura della tabella restaurants (continua)

Colonna	Tipo	Null	Predefinito	Collegamenti a	Commenti
email	text	No			
password	text	No			
salt	text	No			
timeZone	text	No			Esempio: 'Europe/Rome'

Tabella 4.4: Struttura della tabella reviews

Colonna	Tipo	Null	Predefinito	Collegamenti a	Commenti
_id	int(11)	No			
restID	int(11)	No		restaurants (_id)	Foreign Key
name	text	No			
surname	text	No			
date	text	No			Formato: YYYY-MM-DD
rating	int(11)	No			
review	text	No			Testo recensione

Il campo *nA* nella tabella *foods*, se settato a 1, indica che il pasto non è mai stato acquistato, ed è infatti usato nella query della funzione *dutchAuction* (in serversync.js) per individuare i cibi mai acquistati che hanno *timeChange* equivalente al minuto corrente, e perciò devono essere decrementati di prezzo. In caso di acquisto, il campo nA di un pasto viene settato a 0, in modo che non figuri nei risultati della query della funzione *dutchAuction*. I campi *amountbkcp* e *pricebkcp* sono utilizzati per ripristinare i valori originali di quantità e prezzo dei pasti nella funzione *resetFood*, anch'essa in serversync.js.

Dopo aver definito l'organizzazione in file del codice sorgente, e dopo aver definito lo schema delle tabelle del database, sono stati scritti un proxy.js primordiale (ancora senza

controllo malfunzionamenti) e hub.js, che come già detto si occupa di includere i moduli necessari e chiamare le funzioni principali. Per accounts.js, restaurants.js e foods.js in realtà la funzione è una sola: *setListeners*, al cui interno sono definiti gli handler delle varie route della API REST. Di seguito un esempio, che mostra la cancellazione di un utente, quindi una richiesta con metodo DELETE alla risorsa '/users/:id':

```
app.delete('/users/:id', function (req, res){
    pool.getConnection(function(err, connection){
        if(!err){
            connection.beginTransaction(function(err){
                if(!err){
                    connection.query('SELECT id FROM '+ACCOUNTS_TABLE+' WHERE id='+req.params.id+' LIMIT 1
                         FOR UPDATE', function(err, rows, fields){
                        if(!err){
                            connection.query('DELETE FROM '+ACCOUNTS_TABLE+' WHERE id = ? LIMIT 1', [req.
                                 params.id], function (err, rows, fields){
                                if(!err){
                                    connection.query('SELECT id FROM '+ACCOUNTS_TABLE+' LIMIT 1', function (
                                         err, rows, fields){
                                        if(!err){
                                            connection.commit(function(err){
                                                if(!err){
                                                    connection.release();
                                                    res.status(200).send(rows);
                                                }
                                                else{
                                                    return connection.rollback(function(){
                                                        connection.release();
                                                        console.log(err+'\n');
                                                        res.sendStatus(500);
                                                    });
                                                }
                                            });
                                        }
                                        else{
                                            return connection.rollback(function(){
                                                connection.release();
                                                console.log(err+'\n');
                                                res.sendStatus(500);
                                            });
                                        }
                                    });
```

```
//[...]il resto sono solo delle altre connection.rollback
```

Listing 4.1: Codice di gestione eliminazione di un certo account

Convenzionalmente, *app* è l'oggetto che denota l'applicazione Express.js. Viene creato chiamando la funzione express() esportata dal modulo express. Perciò, la variabile app contiene tutti i metodi forniti da Express.js per il routing di richieste HTTP e altre funzionalità non necessarie ad Eat@ (ad esempio, il rendering HTML tramite app.render()). app.delete denota quindi la gestione di una richiesta HTTP con metodo DELETE, e come parametri vengono passati la route da gestire e una funzione di callback, eseguita all'arrivo della richiesta.

Innanzitutto si richiede una connessione dal pool di 100 connessioni contemporanee che il database concede (come spiegato in 3.1, questo contribuisce al bilanciamento del carico), di seguito inizia una transazione. Il concetto di transazione è molto importante nelle basi di dati, ed indica una sequenza di operazioni che in caso di successo deve avere risultato permamente e persistente, mentre in caso di insuccesso causa un ripristino del database alle condizioni precedenti all'inizio della transazione (il cosiddetto *rollback*). Una transazione deve rispettare le proprietà ACID:

A: Atomicity - Una transazione è "o tutto o niente", se un'operazione fallisce allora fallisce l'intera transazione e il database rimane invariato

C: Consistency - Dopo la transazione, il database si troverà in uno stato valido, cioè rispetterà tutte le regole di vincoli definite

I: Isolation - L'esecuzione concorrente di transazioni è equivalente alla loro esecuzione seriale. L'eventuale fallimento di una transazione non deve interferire con le altre

D: Durability - Terminata la transazione eseguendo un *commit*, i cambiamenti apportati non devono essere persi

L'utilizzo di una transazione è necessario, oltre che per questioni di maggiore protezione

dei dati da errori, perché tramite esso la variabile di sistema di MySQL *autocommit* viene settata a 0, e solo in questo caso può essere usato il meccanismo di row-level locking dell'engine InnoDB. Il row-level locking (in contrasto con il lock di un'intera tabella tramite la query LOCK TABLES) si ottiene con la query SELECT ... FOR UPDATE, che ottiene un lock esclusivo sulle righe restituite dalla query. Bloccare solo alcune righe, e non l'intera tabella, consente alle query su righe differenti di non rimanere in attesa, evitando congestioni. Dopo la SELECT ... FOR UPDATE viene eseguita la query SQL DELETE sulla riga con id uguale a quello nella route della richiesta (esempio: DELETE http://arianna.cs.unibo.it:9090/users/3). Infine si esegue una SELECT sulla tabella *accounts* per sapere se dopo la DELETE vi è ancora un utente. Questo serve solo all'App Eat@ per decidere se dopo la cancellazione dell'utente deve mostrare la schermata di creazione nuovo account (in quanto non ve ne sono più) o quella di login, in quanto ve ne sono ancora.

Le altre route hanno gestioni tutte molto simili, brevi operazioni su una o poche righe, query rapide e mirate.

N.B. Il server utilizza prevalentemente il row-level locking, eccetto che nelle richieste POST per la creazione di un nuovo utente, ristorante o pasto. In quei casi si utilizza una query LOCK TABLES di tipo WRITE (quindi lock esclusivo) su intera tabella, in quanto col metodo POST si crea una nuova risorsa, perciò non c'è nessuna riga da bloccare.

Per concludere questa sottosezione relativa al server, si espone il codice della funzione *dutchAuction*, cuore delle funzionalità di Eat@:

```
dutchAuction: function(pool){
    if(hostname == CHIEF){
        var job = new CronJob('00 * * * * *', function(){ //asta olandese ogni minuto
            var d = new Date();
            var h = d.getHours();
            var m = d.getMinutes();
            if(String(h).length==1) h = '0'+String(h);
            if(String(m).length==1) m = '0'+String(m);
```

```
var time = h+":"+m;
var rests = [];
var iii = 0;
pool.getConnection(function(err, connection){
    if(!err){
        connection.beginTransaction(function(err){
            if(!err){
                function a(i){
                    if(i<timeZones.length){
                        connection.query('SELECT * FROM((SELECT * FROM '+FOODS_TABLE+' WHERE
                             amount > 0 AND nA = 1 AND timeChange = \"'+time+'\") AS f INNER JOIN
                              (SELECT _id AS rID, timeZone FROM '+RESTAURANTS_TABLE+' WHERE
                             timeZone = \"'+timeZones[i]+'\") AS r ON f.restID = r.rID) FOR
                             UPDATE', function(err, rows, fields){
                            if(!err){
                               if(rows.length>0){
                                   function b(i){
                                       if(!(rests.indexOf(rows[i].restID)>=0)) rests.push(rows[i].
                                            restID);

                                       var amount = rows[i].amount;
                                       var price = rows[i].price;
                                       var timeChange = rows[i].timeChange;
                                       var stepPrice = rows[i].stepPrice;
                                       var stepTime = rows[i].stepTime;
                                       var endRes = rows[i].endRes;

                                       price = price*1 - stepPrice*1;
                                       if(price<0) price=0;

                                       var d = new Date();
                                       d = new Date(d.getYear(), d.getMonth(), d.getDay(),
                                            timeChange.split(':')[0], timeChange.split(':')[1]);
                                       d.setMinutes(d.getMinutes()+stepTime*1);
                                       var h = d.getHours();
                                       var m = d.getMinutes();
                                       if(String(h).length==1) h = '0'+String(h);
                                       if(String(m).length==1) m = '0'+String(m);
                                       timeChange = h+":"+m;

                                       if(timeChange > endRes) amount=0;
                                       connection.query('UPDATE '+FOODS_TABLE+' SET amount='+
                                            amount+', price='+price+', timeChange=\"'+timeChange
                                            +'\" WHERE _id = '+rows[i]._id, function(err, rows2,
                                            fields){
                                           if(err){
                                              return connection.rollback(function(){
```

```
                                connection.release();
                                console.log(err+'\n');
                            });
                        }
                        else{
                            if(i==(rows.length-1)){
                                iii+=1;
                                a(iii);
                            }
                            else b(i+1);
                        }
                    });
                }

                b(0);
            }
        }
        else{
            return connection.rollback(function(){
                connection.release();
                console.log(err+'\n');
            });
        }
    });
}
else{
    connection.commit(function(err){
        if(!err){
            connection.release();
            if(rests.length>0) console.log('Dutch Auction '+time+' DONE\n');
            if(TIMEZONESCHIEF){
                for(var i=0; i<rests.length; i++){
                    module.exports.sendFCMUpdate(rests[i], "DutchAuction");
                }
            }
        }
        else{
            return connection.rollback(function(){
                connection.release();
                console.log(err+'\n');
            });
        }
    });
}
}

a(iii);
```

```
                    }
                    else{
                        connection.release();
                        console.log(err+'\n');
                    }
                });
            }
            else console.log(err+'\n');
        });
    },
    function(){
    /* This function is executed when the job stops */
    },
    true, /* Start the job right now */
    timeZones[0] /* Time zone of this job. */
    );
  }
}
```

Listing 4.2: Codice della funzione *dutchAuction* per l'esecuzione dell'asta

Innanzitutto, vi è un controllo se l'hostname è equivalente alla variabile CHIEF, se sì, viene impostato il *CronJob* ogni minuto (la sintassi è equivalente a quella dell'utility *Cron* [28] dei sistemi Unix), dopodiché si ottiene l'ora di sistema e si inizia una transazione, che conterrà due funzioni ricorsive annidate: la prima si occupa di trovare quali pasti devono essere decrementati di prezzo, controllando quindi che abbiano nA a 1, $amount > 0$, *timeChange* equivalente al minuto corrente, e che appartengano ad un ristorante la cui Timezone sia una di quelle gestite dal *chief*. La seconda funzione si occupa, per ogni pasto trovato, di decrementare il prezzo, calcolare il successivo *timeChange* ed aggiornare i dati del pasto nel database. Se *timeChange* supera *endRes*, significa che l'asta del pasto è terminata. Il campo *amount* viene perciò settato a 0, in modo che il pasto risulti come non più disponibile.

Per quanto riguarda la funzione *resetFood*, essa esegue una query quasi equivalente per trovare i pasti (con l'unica differenza che si sostituisce alla prima subquery l'intera tabella *foods*), e per ogni pasto utilizza i campi *pricebkcp* ed *amountbkcp* per ripristinare i valori iniziali. Inoltre ricalcola il primo *timeChange*. Come lo faccia è spiegato nella sottosezione 4.1.3.

4.1.2 Eat@

Per prima cosa è stato rimosso ogni riferimento a SQLite, sostituendo ogni chiamata al database locale con chiamate HTTP asincrone al proxy arianna.cs.unibo.it con i vari URL presenti nella API REST definita per questa tesi (all'inizio solo '/users', in quanto il sistema di account è stato il primo servizio ad essere reimplementato sul server). Per le chiamate HTTP asincrone, è stata utilizzata la libreria AsyncHttpClient [29]. Il sistema di account consente le usuali funzioni di creazione, modifica ed eliminazione (metodi POST, PUT e DELETE sulla stessa route '/users') tramite apposite Activity (il nome con cui nell'ecosistema Android si definiscono le schermate). Il bottone di eliminazione si trova nell'Activity di modifica. Vi è inoltre la chiara possibilità (sempre in un'Activity dedicata) di fare login (richiesta GET su '/users' con username e password passati come parametri dell'URL). Le password sono inviate in chiaro nelle richieste HTTP, perciò questo costituisce un problema di sicurezza (facilmente aggirabile usando HTTPS), ma sono salvate nel server come hash tramite l'algoritmo SHA512 [30] con salt di 16 caratteri. Il salt è però salvato nella stessa tabella dove si trova la password, e questa è un'ulteriore piccola falla, anch'essa risolvibile con relativamente poco lavoro.

Si è poi proceduto alla funzione di ricerca ristoranti, per nome o per città, distinte tramite due barre di ricerca separate nell'Activity principale di Eat@. La distinzione avviene tramite il valore del parametro *filter* (name o city) nella richiesta GET della route '/restaurants', ma la query svolta dal server è la stessa. I ristoranti restituiti ad Eat@ sono ovviamente quelli che rispettano il filtraggio iniziale tramite nome o città, ma sono ulteriormente filtrati per restituire solo i ristoranti che hanno cibi prenotabili in quel momento nella loro Timezone. È per questo importante dettaglio che è servita la libreria Javascript moment-timezone, perché consente di create un oggetto Date() con parametro una determinata Timezone IANA (e come si è visto in 4.1.1, ogni ristorante ha la propria Timezone salvata), in modo che un'istanza del server di Eat@ in qualsiasi parte del mondo possa calcolare l'ora di un qualsiasi ristorante in qualunque parte del mondo. Basta quindi che un ristorante abbia un solo pasto prenotabile (quindi l'oggetto Date() creato con moment-timezone è compreso tra i campi *beginRes* ed *endRes* del

pasto) perché questo venga incluso nel JSONArray restituito.

Selezionando un ristorante, ne viene visualizzato (Activity denominata *Foodlist*) l'elenco di pasti disponibili, oltre che informazioni come nome, indirizzo ed immagine profilo. La gestione delle immagini, sia in Eat@ che Eat@ Restaurants, è affidata alla libreria Picasso [31]. Selezionando un pasto, si visualizzano le sue informazioni, come prezzo, quantità, orario di inizio e fine prenotazione, ed un bottone che permette di acquistarlo, potendo scegliere la quantità. L'acquisto è l'unica richiesta HTTP con un URL non REST-compliant al 100% (POST su '/restaurants/:rid/foods/:fid/buy'), in quanto "buy" è un verbo e non il nome di una risorsa. Dopo attente ricerche si è però visto che con REST avvengono spesso questi casi limite, l'importante è documentarlo. Proprio nell'Activity *Foodlist* avviene la sincronizzazione con Firebase (implementato nell'App come banale servizio, pochissime righe di codice) sul *topic* del ristorante (che è semplicemente il suo id univoco nella tabella *restaurants* del database), e in caso di notifica ricevuta la lista dei cibi viene aggiornata (quando il FirebaseMessaggingService riceve un messaggio, questi informa un Broadcast Receiver attivo in *Foodlist*, che aggiorna i pasti richiamando un apposito metodo). La sincronizzazione con Firebase avviene ad ogni round dell'asta eseguito su uno o più pasti del ristorante visualizzato, oppure se un pasto si esaurisce.

Tramite il menu sulla ActionBar di *Foodlist*, è possibile valutare il ristorante (POST su '/restaurants/:id/reviews') con un voto da 1 a 5 stelle e scriverne una recensione. Inoltre è possibile visualizzare le recensioni, in un'altra Activity, dove è presente anche un istogramma orizzontale col numero di valutazioni per ogni stella. I grafici sono realizzati con la libreria MPAndroidChart [32].

Dal punto di vista meramente grafico, Eat@ è costituito dalle Activity di login, creazione e modifica account, ricerca ristoranti, visualizzazione ristoranti cercati, lista pasti ed elenco recensioni. Nelle Activity di ricerca, visualizzazione ristoranti e lista pasti è disponibile un menu laterale tramite Navigation Drawer, che consente di accedere alle

Activity di ricerca e di modifica account, e di eseguire il logout.

Eat@ non richiede alcun permesso aggiuntivo. Nelle figure 4.1, 4.2 e 4.3, degli screenshot.

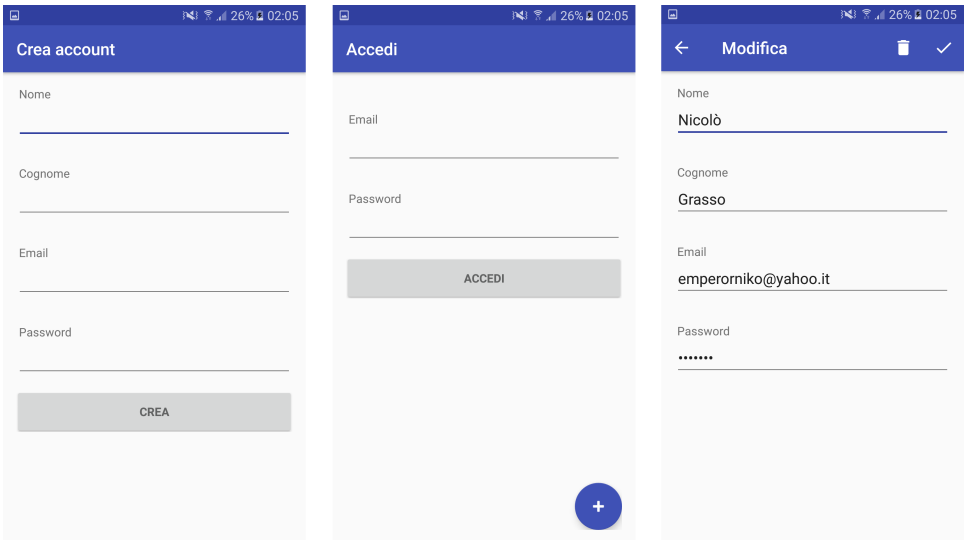

Figura 4.1: Creazione account, login, modifica account

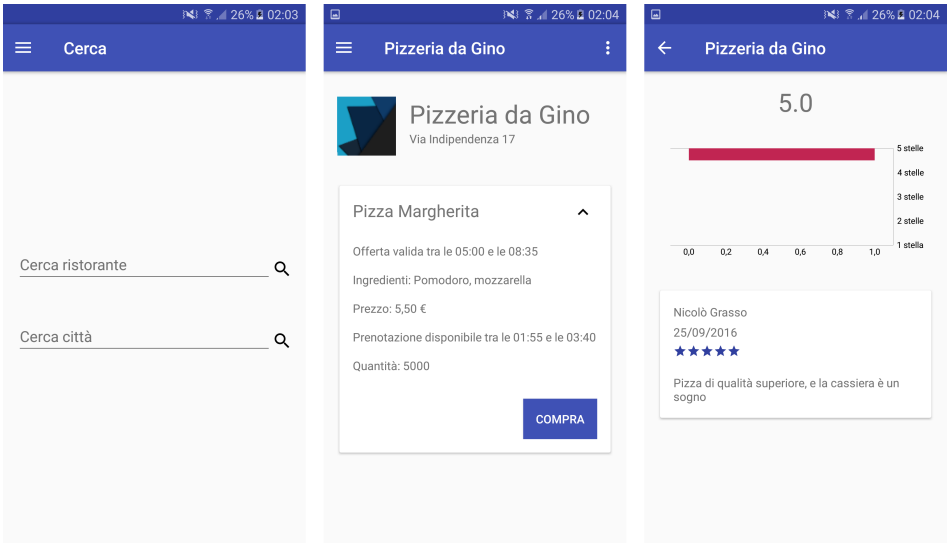

Figura 4.2: Activity di ricerca, lista pasti prenotabili, elenco recensioni

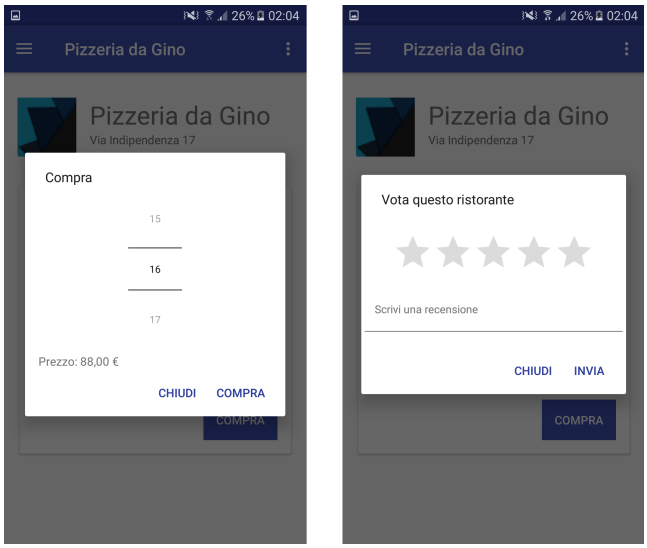

Figura 4.3: Dialog acquisto, dialog valutazione

N.B. Eat@, così come Eat@ Restaurants, è utilizzabile solo verticalmente (modalità *portrait*), in quanto orizzontalmente (modalità *landscape*) non si è ritenuto vi fosse una user experience adeguata.

4.1.3 Eat@ Restaurants

Eat@ Restaurants è un'App di dimensioni ridotte e sviluppata nell'arco di 3 giorni, dove sostanzialmente sono presenti solo un sistema di account totalmente identico a quello di Eat@, con le uniche differenze nell'URL utilizzato (metodi POST, PUT e DELETE su '/restaurants' invece che '/users') e nelle informazioni immesse, una visualizzazione delle recensioni anch'essa totalmente identica a quella di Eat@, e la funzionalità primaria di visualizzazione, inserimento, modifica e cancellazione di pasti (rispettivamente metodi GET, POST, PUT e DELETE sull'URL '/restaurants/:id/foods'). In caso di inserimento, modifica o cancellazione di un pasto all'interno del suo intervallo di prenotazione, avviene una sincronizzazione con Firebase. L'Activity principale, denominata *Overview* e dove sono visualizzati i pasti, è quella dove avviene la sincronizzazione con Firebase, e

anche questo aspetto è gestito in modo equivalente ad Eat@, con lo stesso FirebaseMessaggingService e lo stesso Broadcast Receiver.

Tramite l'apposito FloatingActionButton presente in *Overview* si accede all'Activity di creazione di un nuovo pasto (la stessa Activity, lievemente cambiata, viene anche utilizzata per la modifica). In questa Activity sono presenti varie EditText dove inserire i dati del nuovo pasto. Il campo relativo alla diminuzione di prezzo ad ogni esecuzione del ribasso dell'asta, cioè il campo *stepPrice* nella tabella *foods*, è costituito da un NumberPicker elencante, per prevenire utilizzi impropri dell'utente, solo i divisori della differenza di prezzo tra prezzo iniziale e prezzo minimo. Sempre per prevenire utilizzi impropri, i campi relativi ad inizio e fine prenotazione ed inizio e fine disponibilità, sono inseribili solo in sequenza, con scarto minimo di 30 minuti, e con la disponibilità chiaramente successiva alla prenotazione.

Fondamentale, nel momento in cui si preme il pulsante di inserimento, è il calcolo del primo *timeChange*, che si ricorda essere un campo presente nella tabella *foods*, ed è il campo che viene utilizzato per il confronto con il minuto corrente dal server quando controlla se vi sono pasti da diminuire di prezzo. Il *timeChange* viene così calcolato, ed in modo analogo viene ricalcolato quando a mezzanotte il *chief* resetta le informazioni dei pasti:

1. Viene definito il numero di *splits*, cioè il numero di intervalli in cui la prenotazione sarà divisa, con un semplice calcolo: ((prezzo - prezzo minimo) / stepPrice) + 1

2. Viene calcolato lo *stepTime*, cioè ogni quanti minuti avviene il decremento di prezzo. Determinato da: durata prenotazione / splits

3. Si imposta come *timeChange* iniziale: (*beginRes* + stepTime + (durata prenotazione - (stepTime * splits)))

Dal calcolo del *timeChange* iniziale si evince come il primo intervallo, quello a prezzo pieno, può durare qualche minuto in più se nel calcolo dello *stepTime* non si è avuto

resto 0. Questo accorgimento serve per fare in modo che l'ultimo intervallo termini esattamente all'orario indicato da *endRes*.

Eat@ Restaurants richiede come permessi aggiuntivi la possibilità di utilizzare la fotocamera e accedere ai contenuti della Galleria. Questi permessi sono necessari per impostare un'immagine profilo, ma in caso non vengano concessi rendono inutilizzabile solo questa funzionalità. Il resto dell'App è utilizzabile senza problemi. Nella figura 4.4, degli screenshot.

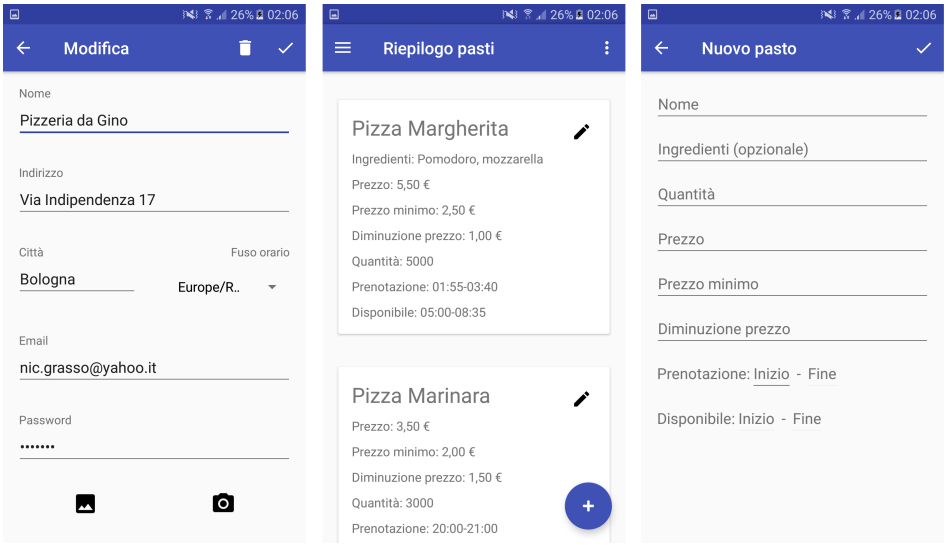

Figura 4.4: Modifica account ristorante, lista pasti, inserimento nuovo pasto

4.2 Valutazione

Per eseguire i test di carico sul server di Eat@, ed i test di paragone su siti conosciuti o similari, si è utilizzato ApacheBench [33], un semplice tool command-line per misurare le performance di server HTTP, eseguendo un elevato numero di richieste in breve tempo, anche concorrenti. Di seguito sono mostrati i report per più test, sia richieste

singole (GET su '/users' e POST su '/restaurants/1/foods/2/buy') che con concorrenza. Si precisa che i livelli di concorrenza utilizzati non sono troppo elevati, innanzitutto a causa delle limitate capacità del PC su cui i test sono stati eseguiti (in una rete esterna al server), e in quanto il server non è in esecuzione su macchine dedicate al ruolo, ma su normali PC, che possono costituire dei bottleneck. Nonostante questo però, anche con concorrenza (troppo) elevata, il tempo medio di esecuzione delle richieste rimane costante e coerente.

Di seguito i report per le richieste singole:

```
Server Hostname: arianna.cs.unibo.it
Server Port: 9090

Document Path: /users
Document Length: 10 bytes

Concurrency Level: 1
Time taken for tests: 0.109 seconds
Complete requests: 1
Failed requests: 0
Total transferred: 211 bytes
HTML transferred: 10 bytes
Requests per second: 9.14 [#/sec] (mean)
Time per request: 109.372 [ms] (mean)
Time per request: 109.372 [ms] (mean, across all concurrent requests)
Transfer rate: 1.88 [Kbytes/sec] received

Connection Times (ms)
              min mean[+/-sd] median max
Connect: 47 47 0.0 47 47
Processing: 62 62 0.0 62 62
Waiting: 62 62 0.0 62 62
Total: 109 109 0.0 109 109
```

Listing 4.3: Test di singola richiesta HTTP GET su '/users'

Considerato che un semplice ping (che comunque utilizza il protocollo ICMP e non HTTP) ad arianna.cs.unibo.it impiega come tempo medio 44ms, 109ms significa indicativamente 65ms per reindirizzare la richiesta dal proxy ad uno dei server, eseguire la query, elaborarne i dati e rispedirli, un ottimo risultato. Come verifica, si è eseguita una

richiesta equivalente direttamente ad uno dei server, senza passare per il proxy, ottenendo 64ms come tempo di risposta.

```
Server Hostname: arianna.cs.unibo.it
Server Port: 9090

Document Path: /restaurants/1/foods/2/buy
Document Length: 16 bytes

Concurrency Level: 1
Time taken for tests: 0.125 seconds
Complete requests: 1
Failed requests: 0
Total transferred: 218 bytes
Total body sent: 191
HTML transferred: 16 bytes
Requests per second: 8.00 [#/sec] (mean)
Time per request: 124.997 [ms] (mean)
Time per request: 124.997 [ms] (mean, across all concurrent requests)
Transfer rate: 1.70 [Kbytes/sec] received
                     1.49 kb/s sent
                     3.20 kb/s total

Connection Times (ms)
             min mean[+/-sd] median max
Connect: 62 62 0.0 62 62
Processing: 62 62 0.0 62 62
Waiting: 62 62 0.0 62 62
Total: 125 125 0.0 125 125
```

Listing 4.4: Test di singola richiesta HTTP POST su '/restaurants/1/foods/2/buy'

Un singolo acquisto (di 10 elementi, un valore solo di prova usato nel JSON mandato in POST) impiega 125ms, solo lievemente superiore al GET su '/users', comprensibile in quando esegue 3 query invece che una sola. È comunque un valore ottimale.

Si mostra ora il report di un test più impegnativo, 2000 richieste con livello di concorrenza 10:

```
Server Hostname: arianna.cs.unibo.it
Server Port: 9090
```

```
Document Path: /users
Document Length: 10 bytes

Concurrency Level: 10
Time taken for tests: 98.795 seconds
Complete requests: 2000
Failed requests: 0
Total transferred: 422000 bytes
HTML transferred: 20000 bytes
Requests per second: 20.24 [#/sec] (mean)
Time per request: 493.975 [ms] (mean)
Time per request: 49.397 [ms] (mean, across all concurrent requests)
Transfer rate: 4.17 [Kbytes/sec] received

Connection Times (ms)
              min mean[+/-sd] median max
Connect: 31 49 6.0 47 94
Processing: 47 443 21.7 437 591
Waiting: 47 265 117.6 250 578
Total: 94 493 21.9 500 641

Percentage of the requests served within a certain time (ms)
  50% 500
  66% 500
  75% 500
  80% 500
  90% 500
  95% 500
  98% 516
  99% 516
 100% 641 (longest request)
```

Listing 4.5: Test concorrente richiesta HTTP GET su '/users'

Data la concorrenza, sono presenti dei ritardi, a causa di cui la richiesta con tempo di completamento maggiore impiega 641ms, un tempo alto ma ancora accettabile. Si noti però che il 95% delle richieste è stato servito entro 500ms, ed il 99% entro 516. Utilizzando un livello di concorrenza 50, il delay massimo si attesta sui 2536ms, di poco superiore ai 2469ms necessari per servire il 50% delle richieste. Un valore elevato ma causato da fattori tecnici.

È stato eseguito un test equivalente, vale a dire 2000 richieste con livello di concorrenza

10, anche per l'acquisto:

```
Server Hostname: arianna.cs.unibo.it
Server Port: 9090

Document Path: /restaurants/1/foods/2/buy
Document Length: 16 bytes

Concurrency Level: 10
Time taken for tests: 100.341 seconds
Complete requests: 2000
Failed requests: 0
Total transferred: 436000 bytes
Total body sent: 380000
HTML transferred: 32000 bytes
Requests per second: 19.93 [#/sec] (mean)
Time per request: 501.707 [ms] (mean)
Time per request: 50.171 [ms] (mean, across all concurrent requests)
Transfer rate: 4.24 [Kbytes/sec] received
                    3.70 kb/s sent
                    7.94 kb/s total

Connection Times (ms)
             min mean[+/-sd] median max
Connect: 31 50 6.5 47 94
Processing: 47 450 22.3 453 609
Waiting: 47 276 115.6 281 594
Total: 94 500 22.6 500 656

Percentage of the requests served within a certain time (ms)
  50% 500
  66% 500
  75% 500
  80% 500
  90% 516
  95% 516
  98% 516
  99% 518
 100% 656 (longest request)
```

Listing 4.6: Test concorrente richiesta HTTP POST su '/restaurants/1/foods/2/buy'

I risultati sono molto simili all'altro test di concorrenza su '/users', e si possono trarre le stesse conclusioni. Si noti che la differenza tra le richieste singole è 16ms, mentre la

differenza tra le richieste dalla durata massima nei test concorrenti è 15ms, indice della costanza delle prestazioni del server sia con compiti semplici, che sotto carico, con una sola query o più query. Con livello di concorrenza 50, i valori sono altrettanto similari, con richieste soddisfatte mediamente in 2 secondi e mezzo, decisamente troppi per funzionalità così immediate che restituiscono pochi byte, ma risolvibile utilizzando come server computer appositi.

4.3 Validazione

Sempre utilizzando ApacheBench, sono stati condotti test anche su siti web similari che trattano di aste. Sono stati presi in considerazione ebay.it, webidz.com e ebid.net. Per tutti e tre, i test di carico con 2000 richieste con livello di concorrenza 10 sono falliti, in quanto ApacheBench riportava come la maggioranza delle richieste fossero terminate con uno status code diverso da 200. Questo è probabilmente causato da contromisure atte ad evitare eccessive richieste HTTP dallo stesso host in un tempo ridotto, quindi per evitare un cosiddetto attacco DoS [34] (Denial of Service). Sono stati condotti perciò solo test su richieste singole, mostrati di seguito.

```
Server Software: ebay
Server Hostname: www.ebay.it
Server Port: 80

Document Path: /sch/Notebook-e-portatili/175672/i.html/
Document Length: 226570 bytes

Concurrency Level: 1
Time taken for tests: 1.253 seconds
Complete requests: 1
Failed requests: 0
Total transferred: 227934 bytes
HTML transferred: 226570 bytes
Requests per second: 0.80 [#/sec] (mean)
Time per request: 1252.907 [ms] (mean)
Time per request: 1252.907 [ms] (mean, across all concurrent requests)
Transfer rate: 177.66 [Kbytes/sec] received

Connection Times (ms)
```

```
            min mean[+/-sd] median max
Connect: 27 27 0.0 27 27
Processing: 1226 1226 0.0 1226 1226
Waiting: 1032 1032 0.0 1032 1032
Total: 1253 1253 0.0 1253 1253
```

Listing 4.7: Test di singola richiesta HTTP GET su ebay

Poco più di 1 secondo per circa 221KB, risultati simili sono stati ottenuti eseguendo più volte il test, e anche via browser il tempo di caricamento era compatibile coi dati ottenuti.

```
Server Software: nginx
Server Hostname: webidz.com
Server Port: 80

Document Path: /categories.php?category=Electronics&parent_id=2247
Document Length: 59767 bytes

Concurrency Level: 1
Time taken for tests: 1.125 seconds
Complete requests: 1
Failed requests: 0
Total transferred: 60135 bytes
HTML transferred: 59767 bytes
Requests per second: 0.89 [#/sec] (mean)
Time per request: 1124.981 [ms] (mean)
Time per request: 1124.981 [ms] (mean, across all concurrent requests)
Transfer rate: 52.20 [Kbytes/sec] received

Connection Times (ms)
            min mean[+/-sd] median max
Connect: 172 172 0.0 172 172
Processing: 953 953 0.0 953 953
Waiting: 594 594 0.0 594 594
Total: 1125 1125 0.0 1125 1125
```

Listing 4.8: Test di singola richiesta HTTP GET su webidz

Poco più di un secondo per una pagina però molto articolata.

```
Server Software: nginx/1.6.2
Server Hostname: www.ebid.net
```

```
Server Port: 80

Document Path: /it/perl/main.cgi?type1=&type2=&words=notebook&category2=13325&adult=&pcode=&categoryid
    =13325&categoryonly=on&mo=search&type=keyword
Document Length: 80763 bytes

Concurrency Level: 1
Time taken for tests: 2.121 seconds
Complete requests: 1
Failed requests: 0
Total transferred: 81048 bytes
HTML transferred: 80763 bytes
Requests per second: 0.47 [#/sec] (mean)
Time per request: 2121.057 [ms] (mean)
Time per request: 2121.057 [ms] (mean, across all concurrent requests)
Transfer rate: 37.32 [Kbytes/sec] received

Connection Times (ms)
            min mean[+/-sd] median max
Connect: 169 169 0.0 169 169
Processing: 1952 1952 0.0 1952 1952
Waiting: 1088 1088 0.0 1088 1088
Total: 2121 2121 0.0 2121 2121
```

Listing 4.9: Test di singola richiesta HTTP GET su ebid

Poco più di 2 secondi, ma anche una risposta più grande di quelle restituite da Eat@.

Dei tre siti in esame, il più rapido risulta ebay, un risultato tutt'altro che inaspettato.

N.B. I test sono influenzati dalla velocità di connessione del PC su cui i test sono stati eseguiti, dalle sue capacità di calcolo e da fattori aleatori relativi al sistema operativo.

Capitolo 5

Conclusioni e sviluppi futuri

In questo capitolo si trattano le conclusioni. Nel paragrafo 5.1 si tirano le somme del progetto e dei risultati ottenuti. Nel paragrafo 5.2 si espongono possibili sviluppi futuri di Eat@.

5.1 Considerazioni finali, risultati

Lo svolgimento di questa tesi progettuale è stato di estremo aiuto dal punto di vista dell'arricchimento delle competenze, in quanto ha consentito di lavorare ulteriormente con l'ecosistema Android, di rispolverare le conoscenze sulle basi di dati, con importanti approfondimenti sulla parte relativa alla concorrenza, ma soprattutto di conoscere Node.js, che si è rivelato (insieme ai tantissimi framework ed estensioni) uno strumento eccezionale per sviluppare applicazioni lato server con una curva di difficoltà assolutamente ben graduata, una documentazione completa e una community molto attiva.

Per quanto riguarda i risultati ottenuti, si sono rivelati positivi in termini di gestione dei guasti e responsività del sistema sia client che server, in particolare nei test di carico, con tempi di risposta alle query regolari e ottimali. Sono certamente possibili ulteriori miglioramenti in questi ambiti, ma già questi dati si possono ritenere più che soddisfacenti. Confrontandoli con servizi simili, si nota come Eat@ sia più rapido, ma è da

considerarsi che è un sistema molto più semplice, che esegue query più semplici su un database estremamente più ridotto (e singolo), e restituisce pochi byte in risposta, al contrario dei vari KB dei servizi presi come confronto.

5.2 Sviluppi futuri

In merito a sviluppi ed estensioni future di Eat@, sono 4 quelle a cui si è pensato, due triviali e facilmente implementabili con poco codice e in relativamente poco tempo, una lievemente più lunga e complessa, e una, la principale in termini di durata, dimensione e soprattutto complessità (molto probabilmente più consona ad una tesi di Laurea Magistrale):

1. Implementare il protocollo HTTPS nelle comunicazioni con il server di Eat@

2. Aggiungere la possibilità di eliminare le recensioni da parte dei ristoranti, e di votarle positivamente o negativamente da parte degli utenti (contando sempre nella buona fede che questi non diventino mezzi di censura o spam)

3. Creare un'interfaccia web per Eat@ e Eat@ Restaurants, in modo da poter usufruire dei servizi offerti non solo da dispositivi mobili, ma anche da PC via browser

4. Eseguire un refactoring dell'intero codice del server per rendere anche il database distribuito, in quanto ora Eat@ è distribuito, scalabile e resistente ai guasti solo per quanto riguarda le macchine che ricevono richieste HTTP, ma sono tutte connesse allo stesso database presente sul web server golem dell'Università di Bologna. Oltre a distribuire il database, sarebbe poi necessario aggiungere del codice da eseguire in caso di malfunzionamento di una delle basi di dati, non essendo stato fatto, come già fatto notare in 3.2.

Un approfondimento in merito alla distribuzione del database di Eat@ è presente nella seguente sottosezione.

5.2.1 Eat@ Distributed

La trasformazione del singolo database di Eat@ in un database distribuito real-time con sincronizzazione e gestione della concorrenza tramite Strict Two Phase Locking [35], è una naturale evoluzione del progetto, attualmente solo un'idea, seppur esista in realtà una versione di Eat@ distribuita ma non funzionante (e infatti vi è anche un secondo database attualmente vuoto ed inutilizzato, richiesto quando ancora si voleva provare a svolgere questo compito). Il prototipo non funzionante di Eat@ Distributed è stato creato eseguendo il refactoring di un codice vecchio e assolutamente non perfezionato, mancavano infatti oltre ai vari bugfix una corretta gestione delle Timezone e la concorrenza era gestita interamente tramite query LOCK TABLES, mentre si è mostrato come l'engine InnoDB metta a disposizione tramite la query SELECT ... FOR UPDATE il row-level locking, decisamente più adatto alle caratteristiche di Eat@. Il refactoring di ogni gestione di richieste HTTP (eccetto quelle con metodo GET, in quanto in esse sono presenti solo query SELECT, che non necessitano chiaramente di sincronizzazione) consisteva principalmente nel wrapping del codice già presente nell'algoritmo (la cui totale correttezza in merito anche ai casi particolari è tutta da dimostrare) di seguito mostrato:

```
function wrapper(){
    //[...] altre istruzioni
    connection.query('LOCK TABLES '+TABLENAME+' WRITE', function(){
        timeLock[TABLENAME]=getMilliseconds();
        //[...] altre istruzioni, magari delle SELECT
        function querySync(i){
            request.post(externals[i]+'/lock', {'timeLock' : timeLock[TABLENAME], 'tableName' : TABLENAME},
                  function(res){
                if(i<externals.length){
                    if(res.statusCode===200) querySync(i+1);
                    else if(res=='less'){
                        connection.query('UNLOCK TABLES', function(){
                            connectionGlobal[TABLENAME].query('LOCK TABLES '+TABLENAME, function(){
                                wrapper();
                            });
                        });
                    }
                    else{
                        connection.query('UNLOCK TABLES', function(){
                            for(var j=0; j<i; j++){
                                request.post(externals[j]+'/unlock', {'tableName' : TABLENAME}, function(){
```

```
                            if(j===(i-1)) wrapper();
                        });
                    }
                });
            }
        }
        else{
            connection.query(query, function(){
                connection.query('UNLOCK TABLES', function(){
                    connection.release();
                    for(var j=0; j<externals.length; j++){
                        request.post(externals[j]+'/query', {'tableName' : TABLENAME, 'query' : query
                            });
                    }
                });
            });
        }
    });
}

    querySync(0);
});
}
```

Listing 5.1: Pseudocodice Javascript-like per il tentativo di distribuzione di Eat@

Sostanzialmente, l'algoritmo ottiene il blocco sul database a cui il server è connesso, dopodiché chiede ai *chief* degli altri database di ottenere un analogo blocco (array externals). Se tutte le risposte hanno status code 200, si può eseguire la query, rilasciare il blocco e mandare la query agli altri *chief*, che la eseguiranno e rilasceranno il blocco. Se anche una sola response ha status code 400 con messaggio "less", significa che un altro *chief* ha ottenuto il blocco in un istante precedente e quindi ha la precedenza (e questo spiega perché la richiesta '/lock' ha nel suo body il timeLock), perciò il blocco viene scambiato tra la connessione corrente (che si metterà in attesa) e una connessione globale sempre attiva sul database (una connessione globale per ogni tabella), che eseguirà la query in arrivo dal *chief* che ha vinto il confronto. L'utilizzo del timeLock è indispensabile per evitare deadlock. Per completezza, si mostra nella pagina seguente anche il codice (sempre preliminare) delle nuove route '/lock', '/unlock' e '/query'.

```
app.post('/lock', function(req, res){
    connectionGlobal[req.body.tableName].query('SHOW OPEN TABLES FROM '+DBNAME+' WHERE `Table` LIKE '+req.
        body.tableName+' AND In_use > 0', function(rows){
        if(rows.length===0){ //tabella non bloccata
            connectionGlobal[req.body.tableName].query('LOCK TABLES '+req.body.tableName+' WRITE', function
                (){
                timeLock[req.body.tableName]=getMilliseconds();
                res.sendStatus(200);
            });
        }
        else if(timeLock[req.body.tableName] < req.body.timeLock) res.status(400).send('less');
        else if(timeLock[req.body.tableName] === req.body.timeLock) res.status(400).send('equal');
        else res.sendStatus(200);
    });
});



app.post('/unlock', function(req, res){
    connectionGlobal[req.body.tableName].query('SHOW OPEN TABLES FROM '+DBNAME+' WHERE `Table` LIKE '+req.
        body.tableName+' AND In_use > 0', function(rows){
        if(rows.length>0){
            connectionGlobal[req.body.tableName].query('UNLOCK TABLES', function(){
                res.sendStatus(200);
            });
        }
    });
});



app.post('/query', function(req, res){
    connectionGlobal[req.body.tableName].query(req.body.query, function(){
        if(req.body.tableName!=='reviews') connectionGlobal[req.body.tableName].query('UNLOCK TABLES');
        res.sendStatus(200);
    });
});
```

Listing 5.2: Codice per le nuove route necessarie ad Eat@ Distributed

La route '/lock' controlla innanzitutto se la tabella è già bloccata, se no la blocca e imposta il timeLock, altrimenti esegue il confronto dei timeLock. Il caso (davvero raro) che due timeLock siano uguali al millisecondo, viene trattato inviando un messaggio di-

verso nella risposta (status code 400 con "equal"), che sarà appositamente gestito nella funzione *querySync* (mostrata nel listing 5.1) semplicemente riprovando il procedimento, sperando i timeLock non siano identici anche al secondo tentativo (ancor più improbabile). La route '/unlock' controlla che la tabella sia bloccata e se sì, la sblocca. La route '/query' esegue la query ricevuta e sblocca la tabella (a parte la tabella *reviews*, che non viene mai bloccata dal server di Eat@ non presentando problemi di concorrenza potenziali). Nessuna di queste nuove route è perfettamente REST-compliant, ma come per il caso "buy", è una forzatura accettabile.

Gli algoritmi mostrati sono embrionali e necessitano di lunghi ulteriori studi, ma, insieme a molte altre parti che si occupino della sincronizzazione tra i server e i proxy, sarebbe sicuramente indispensabile implementarli se si volesse rendere Eat@ qualcosa di più che un esercizio accademico.

Elenco dei listing

Elenco delle figure

Elenco delle tabelle

Riferimenti bibliografici

[36] Joy Buchanan, Steven Gjerstad e David Porter. “Information Effects in Uniform Price Multi-Unit Dutch Auctions”. In: *Southern Economic Journal* (2016).

[37] Philippe Gillen et al. “Bid pooling in reverse multi-unit Dutch auctions: an experimental investigation”. In: *Theory and Decision* (2016), pp. 1–24.

[38] Ms Nirali A Kansagara et al. “An Android Application for Online Agri-Auction”. In: (2016).

Sitografia

[1] URL: https://en.wikipedia.org/wiki/Representational_state_transfer.

[2] URL: https://scholar.google.com.

[3] URL: http://www.mysql.com.

[4] URL: https://nodejs.org/en/.

[5] URL: https://firebase.google.com.

[6] URL: https://developer.android.com/studio/index.html.

[7] URL: https://developer.android.com/index.html.

[8] URL: http://stackoverflow.com.

[9] URL: https://www.sublimetext.com.

[10] URL: http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[11] URL: https://www.npmjs.com.

[12] URL: https://en.wikipedia.org/wiki/Observer_pattern.

[13] URL: http://abdelraoof.com/blog/2015/10/28/understanding-nodejs-event-loop/.

[14] URL: http://expressjs.com.

[15] URL: https://www.npmjs.com/package/body-parser.

[16] URL: https://www.npmjs.com/package/cron.

[17] URL: https://www.npmjs.com/package/express-fileupload.

[18] URL: https://www.npmjs.com/package/http-proxy.

[19] URL: http://momentjs.com.

[20] URL: http://momentjs.com/timezone/.

[21] URL: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones.

[22] URL: https://www.npmjs.com/package/mysql.

[23] URL: https://www.npmjs.com/package/ping.

[24] URL: https://www.npmjs.com/package/request.

[25] URL: https://en.wikipedia.org/wiki/Port_(computer_networking).

[26] URL: https://sqlite.org.

[27] URL: https://www.apachefriends.org/index.html.

[28] URL: https://en.wikipedia.org/wiki/Cron.

[29] URL: http://loopj.com/android-async-http/.

[30] URL: https://en.wikipedia.org/wiki/Secure_Hash_Algorithm.

[31] URL: http://square.github.io/picasso/.

[32] URL: https://github.com/PhilJay/MPAndroidChart.

[33] URL: http://httpd.apache.org/docs/2.4/programs/ab.html.

[34] URL: https://en.wikipedia.org/wiki/Denial-of-service_attack.

[35] URL: https://en.wikipedia.org/wiki/Two-phase_locking.