Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

UN APPROCCIO PER LO SVILUPPO DI APPLICAZIONI PORTABILI PER SISTEMI DI CLOUD COMPUTING

Relatore: Prof. Danilo ARDAGNA

Relatore: Prof.ssa Elisabetta DI NITTO

Tesi di Laurea di: Filippo GIOVE matr. 765720

Davide LONGONI matr. 765657

Anno Accademico 2011-2012

Questa Tesi è dedicata alla mia famiglia, che tanto mi ha amato e sostenuto in questi anni di Università. In particolare la dedico a mio padre, che anche se non potrà essere fisicamente al mio fianco nel giorno di questa tanto voluta e sudata Laurea, ha contribuito in modo determinante, grazie ai suoi sacrifici, al suo amore e fiducia in me, al raggiungimento di questo importante riconoscimento.

... Filippo ...

Dedico questa Tesi a tutte le persone che hanno contribuito a farmi diventare quello che sono oggi. In particolare alla mia famiglia, a Laura, a tutti i miei parenti e ai miei amici, che hanno sempre creduto in me e che mi hanno sempre incoraggiato per il raggiungimento di questo importate traguardo.

... Davide ...

Ringraziamenti

Davide ringrazia...

Ringrazio i miei genitori, per il sostegno economico e morale che mi hanno sempre dato in questi anni universitari, spronandomi a dare il meglio, incoraggiandomi a non mollare mai, senza mai però eccedere e ossessionarmi nel raggiungimento dei risultati.

Ringrazio mio fratello Mirco, che involontariamente, grazie alla sua passata esperienza universitaria, è stato per me un esempio e un modello da seguire.

Ringrazio Filippo, il collega, compagno e amico con cui ho condiviso la maggior parte degli anni universitari e che, grazie alla sua dedizione allo studio, mi ha saputo trascinare verso i risultati migliori.

Infine rigrazio Laura, che con il suo affetto e il suo amore, ha saputo sorreggermi nei momenti difficili trascorsi durante questi anni di studio. La ringrazio soprattutto per la tranquillità e la pazienza avuta nel sopportarmi durante le sessioni d'esame, in cui diventavo nervoso e poco socievole.

Milano, 20 Dicembre 2012

... Davide ...

Filippo ringrazia...

Sono molte le persone che voglio ringraziare per avermi supportato, aiutato o semplicemente voluto bene in questi anni di Università.

Ringrazio i miei genitori, che mi hanno sempre sostenuto e affiancato nelle mie scelte. Sono fortunato ad essere vostro figlio, oggigiorno non è facile trovare una coppia che si voglia così bene e che sia così dedita alla famiglia come voi.

Ringrazio ad Anna, mia sorella, con la quale da piccolo litigavo spesso, ma che ora è una figura importante, fondamentale della mia vita. Non ricordo la quantità immane di slide di corsi che ti ho fatto stampare al lavoro. Anche per queste piccole cose tu ci sei sempre stata e sempre ci sarai. Grazie.

Ringrazio i miei zii, Adriana e Piero, e i miei fantastici cuginetti, Simona e Marco, parti integranti della mia famiglia e della mia vita, che tanto amore e fiducia ripongono in me. Grazie per esserci sempre, nei momenti belli come oggi, ma soprattutto grazie per esserci stati in quei giorni, in cui la vita ha deciso di non essere molto clemente con me.

Ringrazio i miei compagni di avventure al PoliMi, elencarvi tutti sarebbe impossibile e avrei la paura di dimenticare qualche nome. Vi ringrazio tutti perché ne abbiamo passate veramente tante insieme.

Tra questi, un ringraziamento particolare va a Davide. Non è facile trovare un collega e amico come te. Grazie al sostegno e aiuto reciproco, abbiamo

raggiunto ottimi risultati accademici, che probabilmente da soli non saremmo riusciti ad ottenere. Segno evidente di come l'amicizia sia un valore importante da custodire.

Ringrazio tutti i miei amici, che grazie alla loro amicizia mi hanno regalato momenti unici e indimenticabili, che hanno fatto da splendida cornice a questi 5 anni di Politecnico.

Milano, 20 Dicembre 2012

. . . Filippo . . .

Filippo e Davide ringraziano...

Ringraziamo la professoressa Elisabetta Di Nitto e il professor Danilo Ardagna per averci concesso la possibilità di svolgere questo lavoro di tesi. Grazie per l'aiuto, la disponibilità, i suggerimenti e gli input che ci hanno stimolato ad arricchire ed approfondire gli argomenti trattati e per il tempo dedicatoci.

Milano, 20 Dicembre 2012

... Filippo & Davide ...

Sommario

Il Cloud Computing è un paradigma relativamente recente che promette di rivoluzionare la modalità attraverso cui i servizi IT vengono forniti.

Nonostante gli indubbi benefici in termini di scalabilità e di risparmio di costi, permangono ancora oggi delle problematiche che ne stanno limitando la massiccia diffusione. Uno dei più grossi impedimenti riguarda la percezione del cosiddetto effetto ¨lock-in˝ da parte del provider di servizi Cloud.

Relativamente a queste tematiche, in questo elaborato di tesi vengono introdotti i concetti di portabilità ed interoperabilità, focalizzandosi sul problema della portabilità di applicazioni tra i PaaS (Platform as a Service) provider.

In particolare viene proposto uno strato di astrazione che permette alle applicazioni sia di utilizzare i più comuni servizi Cloud, sia di essere portabili su più piattaforme. Questo approccio, denominato CPIM (Cloud Portable Independent Model), propone una libreria che espone delle API ¨Vendor-Independent˝ che forniscono un'intermediazione dei seguenti servizi Cloud: servizio Blob, servizio NoSQL, servizio SQL, servizio TaskQueue, servizio MessageQueue, servizio Memcache e di Mailing.

L'attuale versione della libreria CPIM supporta la portabilità di applicazioni Java tra le piattaforme di Google App Engine ed Windows Azure.

Infine è stata effettuata un'estesa attività sperimentale con l'obiettivo di valutare l'overhead della libreria rispetto all'uso diretto delle due piattaforme. I risultati hanno dimostrato come l'overhead nell'utilizzo delle API della libreria, sia marginale in termini di tempi di latenza e di utilizzo della CPU delle risorse computazionali.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici

Capitolo 1

Introduzione

Il Cloud Computing è un nuovo paradigma per l'utilizzo di risorse IT, che offre, sottoforma di servizi, risorse computazionali, software, storage, senza la necessità da parte dell'utente di conoscere la locazione fisica e la configurazione del sistema che li fornisce.

I servizi offerti, in ambito Cloud, possono essere classificati in tre macro categorie:

- Infrastructure as a Service (IaaS), mette a disposizione l'infrastruttura hardware utilizzata dagli utenti, che hanno la possibilità di creare e configurare macchine virtuali, dotandole dei sistemi operativi necessari per il loro utilizzo.

- Platform as a Service (PaaS), mette a disposizione l'infrastruttura hardware e le macchine virtuali in cui poter caricare ed eseguire le applicazioni sviluppate dagli utenti.

- Software as a Service (SaaS), consiste in un applicativo software che mette a disposizione dell'utente determinate funzionalità.

Il Cloud Computing, essendo ancora nelle prime fasi di sviluppo, soffre del classico problema della molteplicità di diversi approcci utilizzati per la sua progettazione ed implementazione. Praticamente, ogni singolo Cloud provider utilizza soluzioni diverse per offrire le proprie risorse e i propri servizi.

In questo elaborato di tesi ci si focalizzerà sulle problematiche causate da queste eterogeneità in ambito PaaS, cioè sulla difficoltà di trasferire un'applicazione da un provider ad altro. Idealmente, come in ogni forma di mercato

libero, le aziende dovrebbero essere in grado di cambiare il proprio fornitore PaaS, nei tempi e modi che desidera.

Le motivazioni che possono spingere un manager IT a spostare tutta, o parte, della propria infrastruttura software da un provider ad un altro possono essere:

- *SLA offerti non rispettati.*
- *Aumento dei costi.*
- *Servizi della concorrenza non presenti nell'attuale fornitore di servizi Cloud.*

In realtà esistono delle barriere a questa libertà di cambiamento di provider, principalmente causate da:

- Varietà dei servizi offerti dalle piattaforme esistenti.
  Ciò potrebbe in primo luogo sembrare un aspetto positivo, ma in realtà la combinazione e la varietà di questi servizi rendono ancora più complesso il problema della portabilità.
- Mancanza di API standard per l'accesso ai servizi PaaS.
  Ogni provider espone API proprietarie, non compatibili con quelle esposte da altri.

In definitiva, un manager IT che volesse trasferire la propria applicazione, da una piattaforma ad un altra, è chiamato a riscrivere e/o reingegnerizzare parti sostanziali del codice applicativo, operazione decisamente onerosa che lo farebbe probabilmente desistere al cambiamento.

1.1 Obiettivi

L'obiettivo principale del lavoro di tesi, è proprio quello di proporre un approccio di standardizzazione in grado di risolvere il problema della portabilità delle applicazioni tra differenti piattaforme Cloud, dando la possibilità agli utenti di scegliere, monitorare ed eventualmente cambiare il proprio provider, senza dover modificare parti sostanziali del codice applicativo, evitando così i costi, potenzialmente molto rilevanti, derivanti dal cambiamento.

Le difficoltà nell'attuazione di questi propositi, risiedono, come detto, nell'eterogeneità delle piattaforme Cloud, relativamente a:

- Servizi offerti
- Linguaggi di programmazione supportati
- API esposte

L'approccio proposto in questa tesi prevede l'introduzione di uno strato di intermedazione che permetta di disaccoppiare le applicazioni dalle API proprietarie esposte dai PaaS provider per l'accesso ai servizi offerti dalle loro piattaforme.

1.2 Risultati

Seguendo questo approccio si è implementata una libreria, chiamata CPIM (Cloud Platform Independet Model), che, esponendo API Vendor-Independent, introduce un'astrazione software tra le applicazioni e i servizi Cloud.

La scelta di quali piattaforme supportare nella prima versione della libreria è ricaduta su Google App Engine e su Windows Azure. Questa decisione è motivata dalla grande diffusione e importanza che hanno questi provider nel mercato PaaS.

Per quanto riguarda il linguaggio di programmazione, si è deciso per Java, linguaggio supportato dalla quasi totalità delle piattaforme PaaS, aspetto rilevante nell'ottica di un futuro ampliamento del set di piattaforme supportate.

La libreria CPIM espone API per l'accesso ai seguenti servizi Cloud, tra i più comuni nelle piattaforme PaaS:

- NoSQL Service
- SQL Service
- Blob Service
- Message Queue

- Task Queue
- Mail Service
- Memcache

La libreria, utilizzando solamente dei metadati contenuti in file di configurazioni, è in grado di rimappare, a runtime, le richieste a questi servizi Cloud ˝generici˝, in chiamate ˝Vendor-specific˝ relative al servizio del provider scelto.

Inoltre per agevolare l'utente durante la compilazione dei file di configurazione, è stato sviluppato un plugin per Eclipse 3.6 che, attraverso interfacce grafiche user-friendly, permette di velocizzare il processo di deployment di un'applicazione, soddisfando tutte le dipendenze connesse al Cloud scelto.

In sostanza, lo sviluppatore dovrà implementare l'applicazione come se fosse una semplice applicazione Java web standard, utilizzando però le interfacce esposte dalla libreria CPIM, e servirsi poi del plugin per migrare l'applicazione sulla piattaforma scelta. In questo modo, nel caso in cui l'utente volesse trasferire l'applicazione da un Cloud all'altro, basterà rieffettuare il passo di migrazione verso l'altra piattaforma, senza effettuare alcuna modifica al codice applicativo.

Con l'obiettivo di utilizzare e testare tutti i servizi standardizzati dalla libreria CPIM, si è sviluppata un'applicazione di esempio, denominata ˝Meeting In the Cloud˝ (MiC), un piccolo social network.

Per determinare l'overhead introdotto dalla libreria, oltre alla versione di MiC che fa uso della libreria, si sono sviluppate due versioni ˝native˝ per le due piattaforme, che utilizzano cioè le API proprietarie, senza l'intermediazione della libreria CPIM.

Utilizzando JMeter, un specifico tool di testing, si sono confrontate la versione di MiC ˝Vendor independent˝ con le versioni native, per entrambe le piattaforme. Le metriche utilizzate per l'analisi dell'overhead sono state i tempi di latenza e l'utilizzo medio della CPU delle Virtual Machine nelle quali sono state caricate le varie versioni dell'applicazione durante i test.

Dall'analisi dei risultati si è dimostrato che l'utilizzo dello strato software di intermediazione introduce un overhead trascurabile, per entrambe le metriche di confronto.

1.3 Struttura della tesi

Nel Capitolo 2 vengono presentante le nozioni generali relative al contesto tecnologico in cui è collocato il lavoro svolto, focalizzandosi sulle tematiche riguardanti i problemi della portabilità nei sistemi Cloud e i relativi progetti in tale ambito.

Il Capitolo 3 descrive in dettaglio i servizi offerti dalle piattaforme Google App Engine e Windows Azure, con particolare enfasi sui servizi standardizzati dalla libreria. In questo modo si cerca di fornire al lettore tutte le informazioni necessarie per poter comprendere l'approccio sviluppato.

Nel Capitolo 4 si illustra l'approccio utilizzato e la descrizione delle scelte progettuali utilizzate per l'implementazione dello strato di astrazione CPIM.

Nel Capitolo 5 vengono illustrate l'architettura e le funzionalità offerte dal plugin sviluppato per Eclipse.

Nel Capitolo 6 viene presentata l'applicazione di test MiC, descrivendo attraverso diagrammi UML il suo design, le sue funzionalità e quali servizi Cloud utilizza.

Nel Capitolo 7 vengono illustrati i risultati ottenuti dai test di performance effettuati con JMeter.

Nel Capitolo 8 sono presentate le considerazioni finali e le proposte di possibili ampliamenti futuri della libreria.

Infine nelle Appendici A e B vengono descritte, rispettivamente, le configurazioni e i requisiti necessari per l'utilizzo dello strato di astrazione CPIM senza l'ausilio del plugin.

Capitolo 2

Stato dell'arte

In questo capitolo iniziale, viene introdotto il contesto e il relativo stato dell'arte inerente a questo elaborato di tesi.

Nella Sezione 2.1, viene definito il concetto di Cloud Computing, definendone caratteristiche, requisiti, modelli standard di deployment e la classificazione delle varie offerte.

Nella Sezione 2.2, viene descritta con maggior attenzione e dettaglio la categoria di servizi Cloud comunemente definita PaaS (Platform as a Service), sui quali è orientato maggiormente tale elaborato.

Infine in Sezione 2.3 vengono introdotte le problematiche di portabilità ed interoperabilità in ambito PaaS, alle quali questo lavoro di tesi propone un approccio per la risoluzione.

2.1 Introduzione al Cloud Computing

NIST [1] definisce il Cloud Computing come un modello che permette di accedere tramite la rete, in qualunque posto e in qualunque momento, ad un pool condiviso di risorse informatiche configurabili (ad esempio, reti, server, storage, applicazioni e servizi) che possono essere rapidamente affittate e rilasciate con uno sforzo di gestione minimo e tramite l'interazione con il service provider. Le risorse informatiche del Cloud Computing vengono offerte dai provider, i quali le mantengono e le gestiscono, sotto forma di servizi Web.

Come stabilito in [1], per poter essere definito Cloud Computing un sistema deve possedere delle caratteristiche essenziali:

- *On-demand self-service.* L'utente può unilateralmente affittare risorse computazionali, ad esempio un dato numero di Virtual Machine a seconda della necessità, senza richiedere alcuna interazione con il personale del service provider.

- *Accesso alla rete essenziale.* I servizi sono disponibili tramite la rete e acceduti tramite meccanismi standard che promuovono l'uso di piattaforme client differenti (es. smartphone, tablet, computer, e workstation). Risulta quindi indispensabile la disponibilità di un accesso ad internet.

- *Risorse condivise.* Le risorse informatiche dei provider sono assemblate per servire utenti multipli usando un modello multi-tenant, con risorse fisiche e virtuali assegnate e riassegnate dinamicamente rispetto alla domanda dell'utente. L'utente generalmente non ha alcun controllo o conoscenza dell'esatto luogo in cui le risorse del provider vengono localizzate, ma potrebbe avere la possibilità di specificare la locazione delle risorse ad un alto livello di astrazione (es. paese, stato, o data center).

- *Elasticità rapida.* Le funzionalità possono essere riservate e rilasciate, in alcuni casi automaticamente, per scalare rapidamente aumentando o diminuendo le risorse a seconda dei requisiti computazionali.
  All'utente, le risorse disponibili appaiono spesso illimitate, quindi possono soddisfare ad ogni istante le quantità necessarie.

- *Monitoraggio.* I sistemi Cloud controllano e ottimizzano automaticamente le risorse usate sfruttando delle metriche basate su un appropriato livello di astrazione a seconda del tipo di servizio offerto.
  L'uso delle risorse può essere monitorato, controllato e segnalato, fornendo trasparenza per il provider e per l'utente del servizio.

Tradizionalmente, le aziende, per costruirsi la propria infrastruttura IT, sono costrette ad investire ingenti somme per l'acquisto di numerosi server, e a sostenerne i relativi costi di gestione (costi di manutenzione, di energia e di locazione). A volte, soprattutto per le aziende medio piccole, l'avere un data center di proprietà implicherebbe sforzi economici troppo elevati, spesso insostenibili.

Tendenzialmente l'infrastruttura IT viene sovradimensionata in termini di capacità di calcolo, ma in realtà la potenza di calcolo mediamente impiegata quotidianamente non supera mai il 10-20% della capacità totale.

A questo proposito il Cloud Computing viene visto come una rivoluzione tecnologica che fornisce, alle piccole e medie imprese, la possibilità di disporre di risorse informatiche in quantità pressoché illimitate, ad un prezzo molto più conveniente rispetto agli investimenti sostenuti per l'acquisto e per la gestione di nuovi server.

Infatti, l'affitto delle risorse informatiche di sistemi Cloud si basa sul paradigma pay-per-use, ossia il cliente paga solo per le risorse effettivamente utilizzate senza dover sostenere investimenti significativi, ad esempio l'acquisto di licenze software. Questo paradigma è molto ben visto dagli utenti, in gran parte aziende.

Ovviamente nel target di utenza del Cloud Computing rientrano anche le imprese che possiedono un proprio data center o una propria infrastruttura. Infatti il Cloud Computing è un modello che può essere applicato anche ad una infrastruttura proprietaria, ottimizzando l'uso delle risorse e semplificandone la gestione.

Ad esempio, se un applicativo software di un'azienda deve essere aggiornato, nello scenario tradizionale, l'aggiornamento deve essere propagato su ogni macchina in cui il software è installato. Con il Cloud Computing invece, l'aggiornamento verrà effettuato solo sulla macchina virtuale che ospita il servizio. A questo punto, quando gli utenti accederanno a tale servizio, otterranno, dalla macchina virtuale ospitante, un'istanza aggiornata dell'applicativo, riducendo drasticamente i tempi e i costi di deployment.

Un altro aspetto fondamentale del Cloud deriva dalla condivisione delle risorse informatiche, permettendo ad ogni utente di avere sempre a disposizione la potenza di calcolo, nonché i propri dati sempre aggiornati attraverso un qualsiasi dispositivo fornito di connettività internet, come ad esempio Smartphone, Tablet, etc.

2.1.1 Modelli di deployment

Il Cloud Computing, quindi, può essere applicato a diversi modelli di deployment [2] [3]:

- *Private Cloud*

  Come schematizzato in Figura 2.1, l'infrastruttura Cloud viene riservata per un uso esclusivo da parte di un'organizzazione comprendente molti utenti. Tale infrastruttura potrebbe appartenere ed essere gestita direttamente dall'organizzazione, da terze parti, oppure da una combinazione delle due: di proprietà dell'organizzazione, ma gestita da terze parti o viceversa. Inoltre può essere localizzata all'interno o all'esterno dell'organizzazione stessa. Di seguito vengono elencati i vantaggi e gli svantaggi di tale configurazione:

  - Vantaggi:
    - Ottimizzazione nell'utilizzo delle risorse (data center proprietari)
      Miglior organizzazione e distribuzione dei carichi di lavoro sulle varie macchine disponibili nel data center, con vantaggi in termini di risparmio energetico.
    - Infrastruttura totalmente dedicata
      In tale configurazione, le risorse informatiche non vengono condivise con altri utenti, ma vengono totalmente riservate all'utente.
    - Sicurezza
      Essendo totalmente dedicata o addirittura proprietaria, non si incorre nel rischio di violazione della privacy.
  - Svantaggi:
    - Costi elevati rispetto alle altre tipologie
      Tale configurazione ha costi inferiori rispetto ai sistemi tradizionali, ma data la riservatezza delle risorse, rispetto alle altre configurazioni, i costi di affitto sono più elevati.

Figura 2.1: Private Cloud

- *Public Cloud*

  L'infrastruttura Cloud viene messa a disposizione per un uso pubblico. Solitamente è di proprietà di un'azienda privata, detta Cloud provider, che mette a disposizione della clientela la propria infrastruttura. Di seguito in Figura 2.2 viene rappresentato lo schema logico di tale tipologia di deployment.

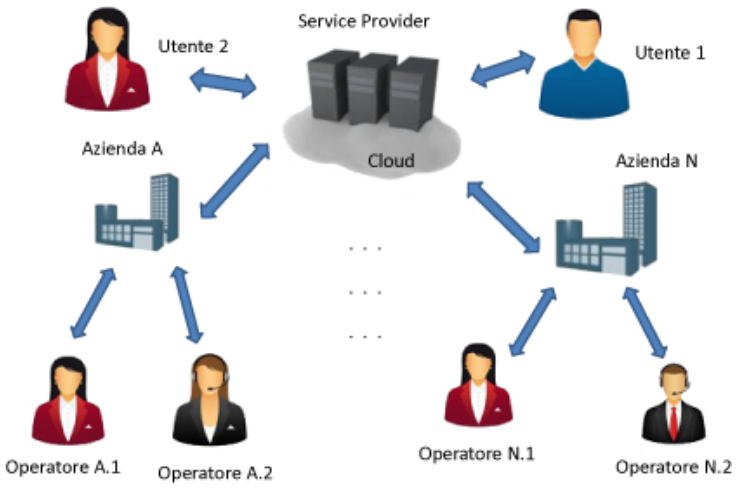


Figura 2.2: Public Cloud

I vantaggi e gli svantaggi introdotti da questa configurazione sono:

  - Vantaggi:
    - Disponibilità elevata

Essendo pubblico, il provider ha l'obbligo e l'interesse di offrire un servizio altamente affidabile e sempre disponibile.

* Risorse illimitate
  Data l'immensa dimensione dei data center dei provider che attualmente offrono questa tipologia di servizio, le risorse informatiche risultano pressoché illimitate, sia in termini di capacità di calcolo sia in termini di archiviazione.
* Economico rispetto alle altre configurazioni
  Tale configurazione, data l'elevata condivisioni di risorse tra più utenti offre un servizio ad elevate prestazione, ma ad un prezzo molto conveniente.

- Svantaggi:
  * Sicurezza
    Più che uno svantaggio, il discorso della sicurezza suscita timori e preoccupazioni negli utenti, riguardanti la privacy, soprattutto in materia di dati sensibili. Tali timori sono dati dal fatto che i dati sensibili sono normalmente tutelati e soggetti a normative nazionali e la loro migrazione dalle aziende verso i service provider potrebbe violarle.

Per chiarire meglio quest'ultimo aspetto si introduce un possibile esempio. Se un'azienda con sede in una determinata nazione decidesse di affidarsi ad un'infrastruttura Cloud di un service provider, potrebbe avere la necessità di migrare i dati sensibili in suo possesso, verso i dispositivi di archiviazione del service provider. Tale migrazione però potrebbe comportare lo sdoganamento dei dati al di fuori dei confini nazionali pertinenti all'azienda, violando così le normative statali vigenti e rischiando di non aver alcuna normativa che tuteli il trattamento dei dati sensibili.

Per la regolamentazione a livello europeo dei vari servizi del Cloud Computing e sulla manipolazione di dati sensibili da parte dei service provider, in ambito legislativo, non è ancora stata stilata una vera e propria normativa che definisca i termini e le condizioni a cui i fornitori e i consumatori devono attenersi, in modo da tutelare legalmente entrambe le parti anche in caso di migrazione extranazionale dei dati sensibili.

Per evitare questo rischio, si potrebbe optare per un Cloud di tipo privato, nel caso si disponesse di un'infrastruttura propria, oppure di un Cloud ibrido in modo da poter mantenere i dati sensibili nel Cloud privato e utilizzare i servizi e le risorse offerte da un'infrastruttura pubblica.

- *Hybrid Cloud*

  L'infrastruttura Cloud è una composizione di due o più infrastrutture Cloud (Private, Community, o Public) che rimangono entità distinte, ma che condividono una tecnologia standard o proprietaria che permette la portabilità delle applicazioni e dei dati all'interno di esse. Questo diventa molto utile, ad esempio per bilanciare i carichi di lavoro tra le varie Cloud. La Figura 2.3 rappresenta un possibile scenario di Hybrid Cloud.

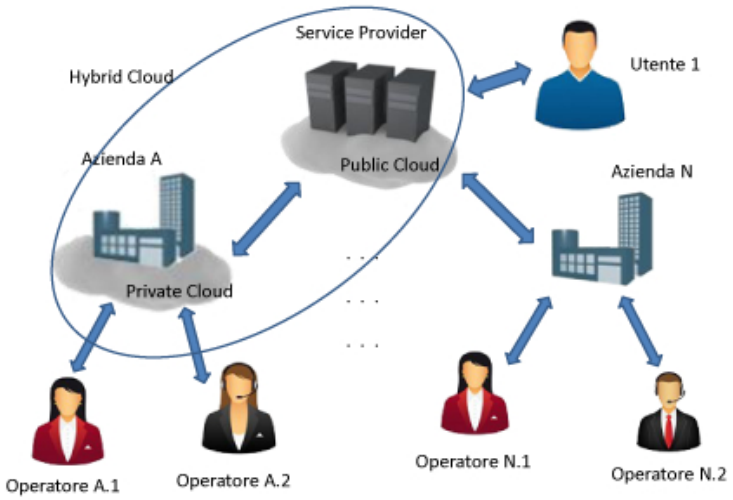


Figura 2.3: Hybrid Cloud

I vantaggi e gli svantaggi introdotti da questa configurazione sono:

  - Vantaggi:

    * Mantenimento dati sensibili in ambito privato
      Grazie alla possibilità di integrazione di vari configurazioni, i dati sensibili possono essere mantenuti nel Cloud privato, aumentando il livello di sicurezza su tali dati.
    * Aumento delle risorse informatiche

Una configurazione ibrida può essere utilizzata dagli utenti per soddisfare carichi di lavoro eccessivi, in modo da evitare il sovradimensionamento delle risorse private, ma facendo subentrare la parte pubblica solo per la copertura di fasi ad alto carico.

  - Svantaggi:
    * Maggiori costi rispetto alla configurazione pubblica
      Data la componente privata, tale configurazione risulta avere costi maggiori rispetto ad una configurazione totalmente pubblica.

- *Community Cloud*

  L'infrastruttura Cloud viene riservata per un uso esclusivo da parte di una community di utenti appartenenti a diverse organizzazioni, come rappresentato in Figura 2.4. L'infrastruttura potrebbe essere di proprietà di una o più organizzazioni nella community le quali la gestiscono, oppure potrebbero appartenere ed essere gestite da terze parti, oppure ancora potrebbe verificarsi una combinazione di entrambe.

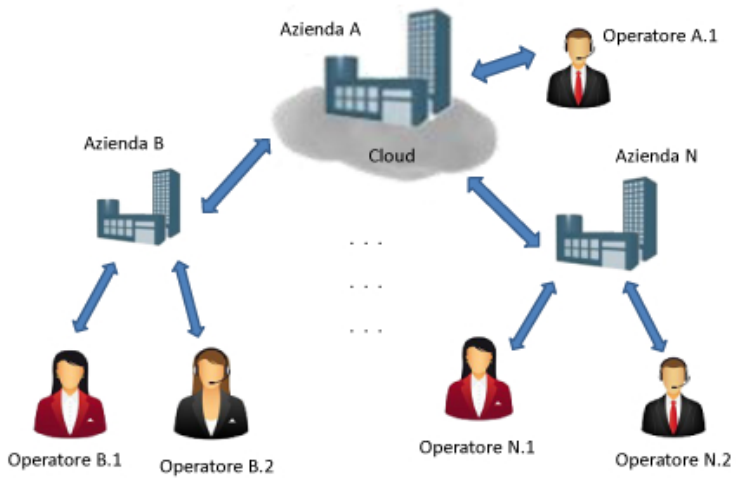


Figura 2.4: Community Cloud

I vantaggi e gli svantaggi introdotti da questa configurazione sono:

  - Vantaggi:
    * Livello di privacy commisurato rispetto alla community

Il livello di privacy e di sicurezza è commisurato rispetto a livello di sicurezza richiesto dalla tipologia di community, ad esempio nel settore finanziario, il ruolo della sicurezza dei dati sensibili ha un valore estremamente elevato.

* Risorse commisurate a seconda della tipologia di community
  Le tipologie di risorse comunemente richieste cambiano a seconda della tipologia di community. Infatti nel settore sanitario ad esempio, è necessario avere dei servizi con tempi di risposta molto rapidi. Nel settore finanziario è necessario avere un livello di replicazione elevato in modo da evitare perdite di dati, fondamentali per tale settore. Tale discorso vale per la tipologia di servizi offerti dalla community, quindi servizi designati per soddisfare le esigenze di uno specifico settore.

– Svantaggi:

* Difficoltà di attuazione
  La Community Cloud, è una tipologia molto difficile da implementare, in quanto non tutte le grandi aziende settoriali sono disposte a concedere la propria infrastruttura a delle loro concorrenti. Tale configurazione può essere utilizzata maggiormente nel settore pubblico.

2.1.2 La classificazione dei servizi Cloud

Il Cloud permette la condivisione di risorse informatiche sottoforma di servizi.

La classica nomenclatura dei servizi Cloud è XaaS (X as a Service), dove X assume uno specifico valore a seconda della tipologia di servizio offerto. Ci sono varie tipologie di servizi che possono essere offerte tramite il Cloud Computing, le quali vengono raggruppate in tre macro-classi, come spiegato in [3] [2], riportate in Figura 2.5:

- *IaaS* (Infrastracture as a Service)

  L'infrastruttura di elaborazione, di storage e di rete è fornita come servizio all'utente, lasciando sotto la sua responsabilità la gestione del sistema operativo, dell'eventuale middleware e della parte di runtime, oltre che

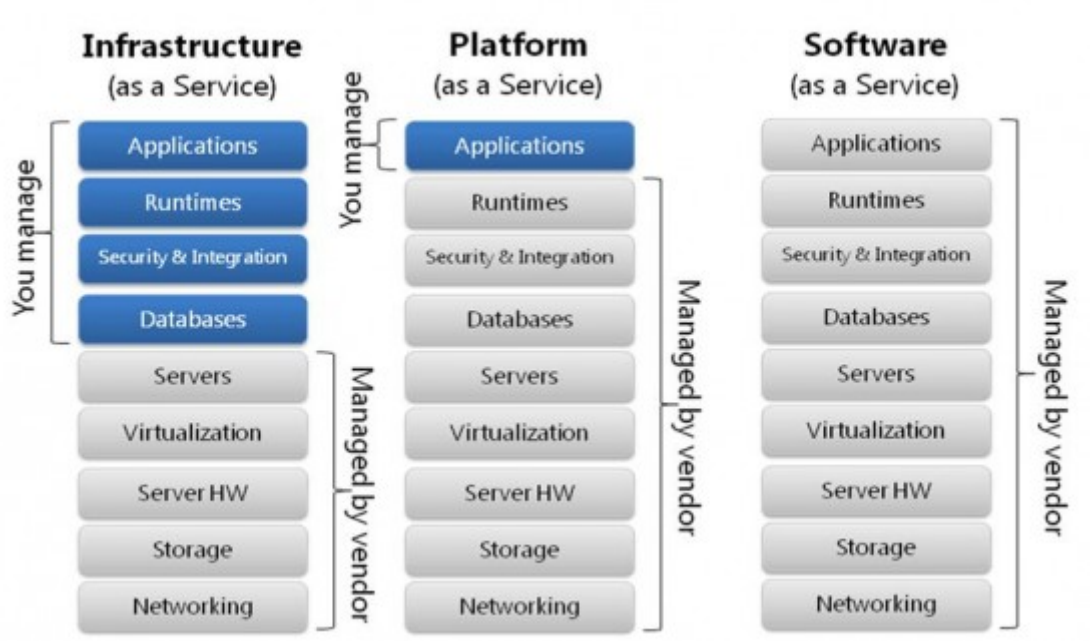


Figura 2.5: XaaS Stack

delle applicazioni da lui installate. L'esempio più famoso di IaaS è Amazon Elastic Compute Cloud (Amazon EC2) [4]. È un servizio web che offre una capacità di calcolo on demand nel Cloud, ed è stato progettato per rendere più facile agli sviluppatori la programmazione degli aspetti di basso livello per l'allocazione delle risorse. La semplice interfaccia web permette di ottenere e configurare le risorse con uno sforzo minimo. Fornisce un completo controllo delle risorse computazionali e permette l'esecuzione nell'ambiente computazionale di Amazon. Amazon EC2 riduce il tempo di richiesta per ottenere e avviare nuove istanze server in pochi minuti, permettendo di aumentare e diminuire rapidamente le risorse, a seconda del cambiamento computazionale richiesto.
Un altro esempio di IaaS è il servizio SmartCloud Enterprise [5] di IBM. È una soluzione Infrastructure as a Service (IaaS) progettata per le aziende che necessitano di accedere più rapidamente ad un ambiente di server virtuali, estremamente sicuro e utilizzabile per sviluppo, test e altri workload dinamici. Con IBM SmartCloud Enterprise è possibile accedere, senza alcun investimento iniziale, ad un'infrastruttura flessibile, comprensiva di hardware, software e connessione protetta, con capacità elaborative illimitate e scalabili, perfette anche in situazioni di picchi di

lavoro o di attività spot.

- *PaaS* (Platform as a Service)

  L'utente ha la possibilità di effettuare il deploy di un'applicazione sviluppata direttamente dall'utente o acquisita sull'infrastruttura Cloud, creata usando linguaggi di programmazione, librerie, servizi e strumenti supportati dal provider. Come per i SaaS, la gestione e il controllo dell'infrastruttura sottostante viene effettuata direttamente dal provider, ma in questo caso, l'utente ha il controllo sulle applicazioni caricate ed ha la possibilità di configurare l'ambiente di hosting delle proprie applicazioni. Esempi più importanti per questa tipologia di servizi sono Google App Engine [6] e Microsoft Windows Azure [7].

- *SaaS* (Software as a Service)

  La funzionalità riservata all'utente è l'utilizzo delle applicazioni del service provider che vengono eseguite direttamente sull'infrastruttura Cloud. Le applicazioni sono accessibili da diversi dispositivi client attraverso semplici interfacce, come ad esempio un web browser, oppure tramite l'interfaccia di un programma preinstallato sul dispositivo client. L'utente è sollevato dalla gestione e dal controllo dell'infrastruttura Cloud sottostante, ossia la rete, i server, il sistema operativo, lo storage e le funzionalità dell'applicazione stessa, fatta eccezione per le impostazioni e le configurazioni specifiche dell'utente.
  Tra questa tipologia di servizi si possono trovare Gmail [8], il servizio mail di Google, iCloud [9] che permette la sincronizzazione tra più dispositivi, Dropbox [10] che permette la sincronizzazione tra cartelle di archiviazione remote, permettendone inoltre la condivisione tra più utenti. Google Documents [11] e Microsoft SkyDrive [12] che permettono la creazione condivisa di documenti.

Dopo questa breve introduzione sui concetti generali del Cloud Computing, ci si sofferma maggiormente sui PaaS, sulle cui problematiche di portabiltà verte tale elaborato di tesi.

2.2 Introduzione ai PaaS

Si è precedentemente definito il concetto di Platform as a service (PaaS) come una categoria di servizi di Cloud Computing che forniscono una piattaforma computazionale in cui poter sviluppare e gestire le proprie applicazioni. La gestione delle reti, dei server e degli apparati di storage non è di competenza dell'utente, ma del provider.

Tali servizi facilitano lo sviluppo delle applicazioni ed evitano investimenti e costi di gestione, legati all'acquisto dell'infrastruttura hardware ed alle licenze software, necessari per poter fornire un adeguato ambiente di hosting su cui eseguire le applicazioni sviluppate.

2.2.1 Categorizzazione dei PaaS

Nel mercato del Cloud Computing e più precisamente in quello dei PaaS, vengono offerte diverse tipologie di piattaforme, in modo da poter coprire tutti i possibili target di clientela, dalle aziende più esigenti e con capacità e conoscenze di programmazione elevate, all'utente retail con capacità e conoscenze informatiche limitate, fino ad arrivare all'affollatissima clientela dell'open source. In tale ottica le piattaforme di Cloud Computing possono essere catalogate in due diverse categorie:

- *Piattaforme delivery-only.* Le piattaforme delivery-only, sono una categoria di PaaS che non prevede alcun ambiente di sviluppo on-line, ma si focalizza principalmente sulla sicurezza e sulla scalabilità. Tali piattaforme mettono a disposizione degli sviluppatori una serie di tool off-line (ad esempio i plugin di Eclipse). Sono un esempio le piattaforme Google App Engine e Windows Azure.

- *Open platform as a service.* Un Open PaaS è una piattaforma che permette di sviluppare, testare e caricare applicazioni implementate con qualsiasi linguaggio di programmazione. Tali piattaforme consentono di adattare lo stack tecnologico sottostante in base alle proprie necessità, scegliendo i componenti e le versioni che si adattano maggiormente agli standard dell'utente. Solitamente questa tipologia di piattaforme consente di scegliere la tipologia di distribuzione del cloud (pubblica o privata). Inoltre le piattaforme di tipo open permettono, senza alcun vincolo, la

migrazione delle proprie applicazioni e dei propri dati verso altri cloud. Tra questa tipologia di PaaS sono presenti la piattaforma OpenShift e la piattaforma Cloud Foundry.

2.2.2 Caratteristiche principali di un PaaS

Un buon PaaS, per essere considerato tale, deve possedere determinate caratteristiche, fondamentali per poter offrire un servizio comune, seppur personalizzabile, come definito in [13]. I requisiti più importanti sono:

- *Differenti servizi di persistenza*

  La persistenza dei dati è fondamentale in molte applicazioni. Semplificare la creazione, la configurazione e lo sviluppo di oggetti persistenti senza richiedere particolari esperienze di programmazione, è una delle caratteristiche importanti per creare un buon PaaS.

  Solitamente le piattaforme esistenti sul mercato del Cloud Computing, fruiscono due diverse tipologie di servizi di archiviazione:

  - *Database relazionali*: i classici e consolidati RDBMS come SQL Server o MySQL.
  - *Database schemaless*: servizio denominato NoSQL, molto performante e soprattutto con un elevatissimo grado di scalabilità, fondamentale in ambito Cloud, come MongoDB, DynamoDB, etc.

- *Integrazione di servizi esistenti*

  L'integrazione di servizi esistenti è un'altra fondamentale caratteristica che le piattaforme Cloud devono avere.
  L'utente deve avere la possibilità di integrare dei servizi web o dei SaaS già presenti, in modo da poter creare applicazioni che interagiscono direttamente tramite le rispettive API. In particolare, molte piattaforme presenti sul mercato, mettono a disposizione un servizio di code, in modo da permettere la comunicazione tra le parti interne di un'applicazione, o addirittura con applicazioni esterne. Tra questi servizi si possono trovare servizi di Caching, per velocizzare e limitare gli accessi ai servizi di persistenza, servizi di Mail, che permettono l'invio e la ricezione di messaggi di posta direttamente attraverso l'applicazione. Oltre a questi servizi che

tipicamente risiedono sulla piattaforma, un buon PaaS deve permettere alle applicazioni residenti in essa, di integrare, o almeno permetterne l'interazione, con servizi SaaS già presenti sul mercato e che mettono a disposizione delle API di interfacciamento.

- *Architettura condivisa multi-tenant o dedicabile*

  Data la grande condivisione di risorse presente nell'infrastruttura Cloud, la multi-tenancy ricopre un ruolo fondamentale per la fruizione di un buon servizio. Un'architettura software multi-tenant permette l'utilizzo condiviso di risorse hardware, senza la necessità di dover separare le diverse istanze di ogni singolo cliente, ma garantendo ugualmente un grado di personalizzazione elevato. L'utente utilizzatore deve poter ottenere prestazioni e riservatezza, allo stesso livello di un pool di risorse interamente dedicato all'utente.

  Per quanto riguarda le prestazioni, l'aumento o la diminuzione di risorse condivise simultaneamente, non deve aver alcun effetto sulle prestazioni dei singoli utenti. Mentre per quanto riguarda la riservatezza e la sicurezza, l'utente deve poter essere fiducioso del fatto che i dati da lui archiviati, non possano essere acceduti da utenti non autorizzati.

  In alcune realtà l'utente richiede esplicitamente la necessità di affittare in uso esclusivo un determinato pool di risorse, in quanto prospetta requisiti computazionali elevati. Si vengono a creare così delle Cloud private (remote) in cui la gestione di basso livello dell'infrastruttura è affidata al provider.

2.2.3 PaaS esistenti

Dato il grande consenso che il Cloud Computing sta ottenendo, è sempre più facile trovare nuove realtà emergenti che decidono di investire sulle infrastrutture per entrare nel mercato del Cloud Computing, offrendo soluzioni alternative rispetto alle realtà già presenti e in alcuni casi ben consolidate.

Di seguito verranno brevemente presentate alcune delle piattaforme Cloud disponibili oggi sul mercato. Dapprima le più famose e consolidate come quella di Microsoft con la piattaforma denominata Windows Azure e quella di Google

denominata App Engine, a quelle di tipo Open Platform come OpenShift di Red Hat e Cloud Foundry di VMware.

Microsoft Windows Azure

Windows Azure [7] è una piattaforma di Microsoft basata su un Cloud pubblico. Questa piattaforma può essere utilizzata in diversi modi. Per esempio, Windows Azure può essere utilizzata per creare un'applicazione web che esegua e memorizzi i propri dati nei data center di Microsoft. È inoltre possibile utilizzare la piattaforma solo per memorizzare i dati, i quali possono essere usati da applicazioni eseguite all'esterno del Cloud.

Windows Azure offre la possibilità di creare macchine virtuali per lo sviluppo e il testing di applicazioni ed implementare applicazioni fortemente scalabili. Con Windows Azure, è possibile creare e gestire applicazioni e servizi utilizzando linguaggi noti come .NET, Java, PHP e Ruby. Il Cloud Computing Microsoft ha un affidabilità di servizio pari a oltre il 99,9% e un supporto 24x7.

Tale piattaforma ha un ruolo importante nella trattazione di questa tesi, quindi verrà ripresa ed analizzata con maggior dettaglio in seguito.

Google App Engine

Google App Engine [6] consente l'esecuzione di applicazioni web sull'infrastruttura di Google. Le applicazioni di App Engine sono facili da creare, mantenere e scalare.

Con App Engine, si paga solo per quello che si usa. Nessun costo di set-up e nessun canone. L'utilizzo delle risorse viene misurato, in modo da controllare il consumo delle applicazioni, ed essere informati sui costi. Tutte le applicazioni possono utilizzare fino ad un 1 GB di memoria, CPU e banda sufficienti per supportare un'applicazione che riceve circa 5 milioni di richieste al mese, in modo gratuito. I costi vengono calcolati al netto di tali quote. Possono essere caricate applicazioni con accesso limitato (ad esempio applicazioni aziendali), oppure applicazioni senza alcuna restrizione di accesso.

Google App Engine supporta applicazioni scritte in diversi linguaggi di programmazione. Si possono sviluppare applicazioni usando le tecnologie standard di Java, tra cui la JVM, le servlet, e tutti i linguaggi di programmazione basati sull'interprete e sul compilatore della JVM (ad esempio JavaScript o Ruby).

App Engine permette inoltre lo sviluppo di applicazioni Python e fornisce un ambiente di runtime basato su Go[14], un progetto open source di Google.

Tutti questi ambienti di runtime sono progettati per assicurare l'esecuzione delle applicazioni in modo rapito, sicuro e senza alcuna interferenza da parte di altre applicazioni presenti sul sistema.

Tale piattaforma, come quella di Microsoft, ha un ruolo importante nella trattazione di questa tesi, quindi verrà ripresa ed analizzata con maggior dettaglio in seguito.

OpenShift Red Hat

OpenShift [15] è un Open PaaS, dove gli sviluppatori ed i team possono creare, testare, caricare ed eseguire le proprie applicazioni. OpenShift si preoccupa delle infrastrutture, del middleware e della loro gestione, e permette agli sviluppatori di focalizzarsi su cosa è meglio per loro: la progettazione e la codifica delle applicazioni.

OpenShift fornisce scalabilità automatica e manuale per le risorse che supportano le applicazioni. Con l'Auto-Scaling, la piattaforma può scalare aumentando le istanze dell'applicazione ed abilitando il clustering. In modalità manuale, l'utente decide quando scalare le risorse sulle quali è distribuita l'applicazione a seconda delle necessità.

La piattaforma è progettata per permettere agli sviluppatori di lavorare nella maniera preferita, fornendo linguaggi, framework e tool necessari per sviluppare applicazioni con velocità e facilità.

OpenShift supporta diversi linguaggi di programmazione come Node.js, Ruby, Python, PHP, Perl e Java. In aggiunta a questi linguaggi, viene data la possibilità di inserire un interprete e un compilatore per poter sviluppare ed eseguire applicazioni codificate in un qualsiasi linguaggio.

Cloud Foundry

Con il PaaS di Cloud Foundry [16], è possibile focalizzarsi sull'applicazione dimenticandosi la comunicazione tra macchine e middleware, sviluppando e scalando le applicazioni sempre più velocemente.

Cloud Foundry è un Open PaaS e dà la possibilità di scegliere i framework di sviluppo e i servizi da integrare. Inoltre si ha la flessibilità di caricare le

proprie applicazioni tra il Cloud pubblico e quello privato oppure optare per una configurazione ibrida.

Lo sviluppatore può liberamente implementare e testare la propria applicazione in locale e caricarla direttamente, senza alcuna modifica del codice sul cloud.

In Cloud Foundry si ha la possibilità di sviluppare le applicazioni con diversi linguaggi di programmazione, tra cui: Java, Grails, Ruby, Sinatra, Scala o Node.js. Inoltre le applicazioni possono essere integrate da servizi di persistenza, o di queue messaging come: Postgres, vFabric, RabbitMQ, MongoDB, MySQL e Redis.

Nella Tabella 2.1 vengono elencate le caratteristiche delle piattaforme presentate precedentemente.

Tipologia	PaaS	Linguaggi	Auto-Scaling	Persistenza	S.L.A.
Delivery-Only	Google App Engine	Java, Ruby, Python, Go	✓	MySQL, Datastore	99.95%
	Windows Azure	.NET, Java, PHP, Python, Node.js	✓	SQL Server Table Service	99.9%
Open PaaS	OpenShift	Ruby, Perl, Java, PHP, Python, Node.js	✓	MongoDB, MySQL, PostgreSQL	Non Garantiti
	CloudFoundry	Ruby, Scala, Sinatra, Java, PHP, Python, Node.js	✓	MongoDB, MySQL, PostgreSQL	Non Garantiti

Tabella 2.1: Riepilogo PaaS elencati

2.3 Problematiche nei sistemi PaaS

Il Cloud Computing è un paradigma relativamente nuovo di utilizzo e fruizione di servizi IT. Essendo quindi ancora nelle prime fasi di sviluppo, soffre del classico problema dei troppi diversi approcci utilizzati per la sua progettazione e implementazione. Praticamente, ogni singolo Cloud provider utilizza soluzioni diverse per interfacciarsi alle proprie risorse e servizi. Un esempio pratico è rappresentato dai servizi offerti dai vendor dominanti del settore, come Google, Amazon, Microsoft, che risultano fruibili tramite interfacce proprietarie, spesso molto differenti le une dalle altre.

Un ulteriore problematica nel contesto delle piattaforme Cloud è quella nota come ˝Data Portability˝. I dati presenti nei servizi di Storage ˝in the Cloud˝ vengono memorizzati dai vari provider utilizzando formati diversi, spesso incompatibili, e a volte perfino la metodologia di archiviazione dei dati (SQL vs NoSql) risulta diversa tra vendor e vendor. Tutto ciò comporta gravi problemi di interoperabilità e portabilità di applicazioni e dei dati risiedenti sulle piattaforme Cloud.

Idealmente le aziende dovrebbero essere in grado di spostare le proprie applicazioni e i relativi dati tra i vari provider di servizi Cloud, senza dover riscrivere le applicazioni o trasformare i dati. Spesso invece, non è calcolabile a priori quanto sia difficile e costoso spostare il proprio business da una soluzione Cloud ad un'altra, o perfino riportarne la gestione all'interno dell'azienda.

Per tutti questi motivi, e non solo, la portabilità delle applicazioni in ambienti Cloud è considerata come il Sacro Graal del Cloud Computing.

Il rischio effettivo quindi che si corre nell'effettuare il deploy di un'applicazione su piattaforme Cloud è quella di rimanere vittime del cosiddetto *lock-in* da parte del provider, cioè l'incapacità di spostare, a costi ragionevoli, il proprio business da tale vendor.

In questa trattazione ci si focalizzerà sulle tematiche relative alle problematiche di portabilità delle applicazioni nel contesto dei PaaS. Innanzitutto però si ritiene opportuno presentare una definizione più formale e completa dei concetti di interoperabilità e portabilità nel Cloud Computing.

2.3.1 Interoperabilità nel Cloud Computing

Nel glossario della IEEE [17], l'interoperabilità è definita come "*the ability of two or more system or components to exchange information and to use the information that has been exchanged*". Come definito in [18], tale concetto può essere applicato a tutti e tre i modelli di Cloud, ciascuno dei quali deve possedere diversi requisiti per essere definito interoperabile.

- *In ambito IaaS.* In ambito IaaS per interoperabilità ci si riferisce alla capacità di un cliente di utilizzare infrastrutture di diversi provider tramite una comune interfaccia di gestione delle stesse.

- *In ambito PaaS.* In ambito PaaS gli sviluppatori potrebbero aver bisogno di utilizzare tool, librerie o API diverse provenienti da PaaS diversi per creare le proprie applicazioni. Quindi si ha interoperabilità quando si ha la capacità di sviluppare applicazioni utilizzanti servizi provenienti da vendor diversi.

- *In ambito SaaS.* Infine in ambito SaaS la condizione riguarda la capacità di diverse applicazioni di scambiarsi dati.

2.3.2 La portabilità nel Cloud Computing

NIST definisce portabilità come "*the ability of prospective cloud computing customers to move their data or applications across multiple cloud environments at low cost and minimal disruption*" [19]. Anche questo concetto è riferibile a tutte le principali tipologie di Cloud esistenti, IaaS, PaaS e SaaS.

In accordo con la Cloud Security Alliance [20] esistono differenti requisiti di portabilità nei tre livelli di modelli di servizi Cloud:

- *In ambito IaaS.* In ambito IaaS si parla di portabilità quando si ha la capacità di spostare facilmente una macchina virtuale e i relativi dati, da un provider ad un altro.

- *In ambito PaaS.* In ambito PaaS per portabilità s'intende la capacità di effettuare il deployment della stessa applicazione su piattaforme differenti.

- *In ambito SaaS.* Infine in ambito SaaS la condizione riguarda la capacità, all'atto del passaggio da un software offerto in modalità SaaS ad un altro, di estrarre i dati dal primo e caricarli nel secondo. Per esempio se una compagnia che utilizza un'applicazione CRM offerta in modalità SaaS, decidesse in seguito di sostituirla con una diversa, è importante che sia possibile che tutti i dati dei clienti siano caricabili e utilizzabili dalla nuova piattaforma CRM.

Alcuni motivi che possono spingere, per esempio un manager IT, a spostare tutta, o parte, della propria infrastruttura software da un provider ad un altro sono per esempio:

- *SLA offerti non rispettati.*
- *Aumento dei costi.*
- *Servizi della concorrenza non presenti nell'attuale fornitore di servizi Cloud.*

Quindi, come in ogni forma di mercato, anche in quello del Cloud Computing, un cliente deve poter essere in grado di scegliere, monitorare ed eventualmente cambiare il proprio fornitore. Invece in questo settore è difficile che questo accada, poichè i costi per effettuare tale cambiamento risulterebbero troppo ingenti dal giustificarlo economicamente.

2.3.3 La portabilità nei PaaS

Nel mercato Paas sono presenti molti attori, alcuni molto popolari come Google con il suo Google App Engine [6], Microsoft con Windows Azure [7] e Force.com [21]. Tutti questi provider offrono diversi servizi, come il file e il data storage, servizio di code, di messaging, etc.

Questa varietà può inizialmente sembrare un aspetto positivo, ma in realtà la combinazione e la varietà di questi servizi rendono ancora più complicata la sfida della portabilità. Infatti alcuni servizi possono essere offerti da una piattaforma in modo assolutamente nativo, in altre possono risultare mancanti, in altre ancora per usufruirne si ha la necessità di utilizzare un fornitore esterno, in una sorta di sub-appalto. Il servizio Mail ne è un esempio: disponibile e

di facile utilizzo in App Engine, non offerto direttamente da Azure. Infatti se si volesse permettere ad una applicazione caricata in Azure di mandare o ricevere email, occorrerebbe utilizzare un server SMTP esterno o appoggiarsi a servizi di mailing esterni (es: SendGrid Email Service [22]).

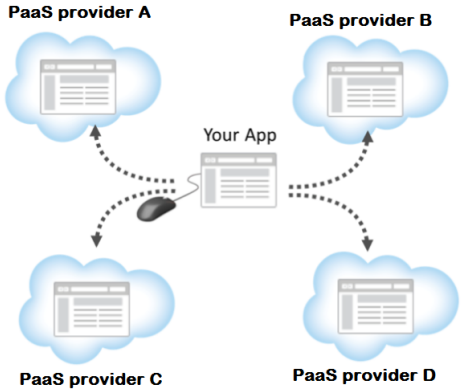


Figura 2.6: Applicazione portabile

2.3.4 Approcci generali per la portabilità nei Paas

Come definito in [18], esistono alcuni approcci generali che potrebbero essere adottati per facilitare la portabilità delle applicazioni nelle varie piattaforme.

- *Adozioni di Standard*

  L'approccio sicuramente migliore è quello che richiede l'adozione di standard globalmente riconosciuti e accettati. Ciò permetterebbe agli sviluppatori software di creare le loro applicazioni senza preoccuparsi su quale piattaforma verranno poi caricate. Ovviamente tali standard devono includere una serie di API standard di accesso ai vari servizi offerti dai provider. Questa risulta essere la soluzione più efficiente, ma anche quella attualmente di più difficile realizzazione, soprattutto per motivi non legati alla tecnologia.
  Infatti tutte le major di tale mercato, come precedentemente detto, utilizzano standard proprietari che di fatto inibiscono la portabilità delle applicazioni verso altre piattaforme, limitando così la perdita di clienti, poichè le difficoltà di riottenere tutti i dati memorizzati ed i costi risultanti dalla reingegnerizzazione del software risulterebbero eccessivi.

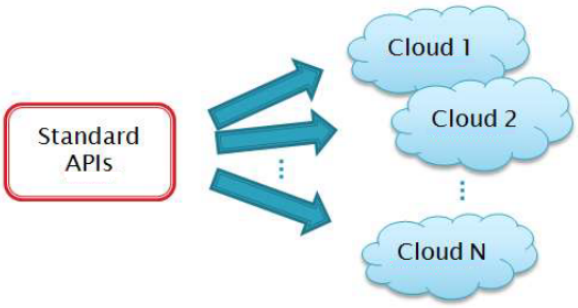


Figura 2.7: Adozioni di Standard

- *Utilizzo di Wrapper/Adapter*

  Un altro approccio è quello di utilizzare uno strato di intermediazione, che possa disaccoppiare le applicazioni dalle piattaforme Cloud e le relative API, formati dei file etc.
  In pratica tale livello ha la funzione di creare un wrapper tra il codice dell'applicazione e gli standard proprietari dei vari provider, che permetta solo a runtime di trasformare, o traslare, il codice platform-independent scritto dallo sviluppatore, in codice adatto alle specifiche del provider scelto. Questo approccio ha il vantaggio di non aver bisogno del consenso diffuso delle principali major del mercato Paas per poter funzionare.

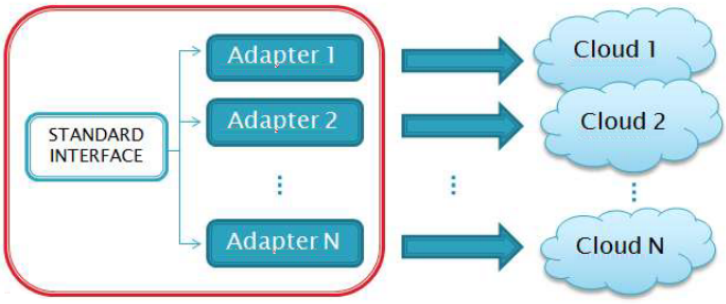


Figura 2.8: Intermediazione

2.3.5 Esempi di lavori esistenti

Esistono diversi lavori in letteratura che hanno l'obiettivo di agevolare la portabilità a livello PaaS.

IEEE P2301

Sicuramente quello più autorevole, ma anche quello con gli obiettivi di più difficile attuazione, almeno nel breve periodo, è quello relativo al gruppo di lavoro della IEEE, denominato P2301, Guide for Cloud Portability and Interoperability Profiles [23]

Tale working group ha lo scopo di tracciare delle direttive comuni in settori come le interfacce ai servizi (API), formati di file e convezioni computazionali che, se diffusamente adottati, agevolerebbero la portabilità e l'interoperabilità, la capacità di far interagire applicazioni su piattaforme diverse, in ambiente Cloud, permettendo una crescita ulteriore ed incisiva dell'utilizzo delle piattaforme Cloud.

VMware-Google Partnership

Un altro esempio è quello relativo alla partnership tra Google e VmWare [24].

Queste due aziende, da tempo attrici protagoniste nel mercato Cloud, tramite questa alleanza stanno cercando di rispondere alle richieste di portabilità negli ambienti Cloud facendo leva sulle tecnologie VMware vCloud, SpringSource, Google Web Toolkit e Google App Engine, per semplificare lo sviluppo e l'implementazione delle applicazioni aziendali.

Per esempio, utilizzando l'ambiente di sviluppo di Eclipse basato su Spring Source Tool Suite, gli sviluppatori possono creare le loro applicazioni, in modo familiare e produttivo, e avere in seguito la possibilità di scegliere di effettuarne il deployment su uno a scelta tra questi ambienti Cloud:

- *Nell'ambiante privato VMware vSphere*
- *In Vmware vCloud partner*
- *Direttamente in Google AppEngine*

SimpleCloud

Un altro progetto interessante è quello denominato SimpleCloud [25]. È un progetto open source ideato da ZendTechnologies con la collaborazione di IBM, Microsoft, Rackspace, Nirvanix e GoGrid, avente l'obiettivo di migliorare la portabilità di applicazioni PHP.

In tale progetto sono state sviluppate una serie di interfacce standard ai più comuni servizi offerti in ambienti PaaS, come riportato in Figura 2.9.

SERVIZIO	PIATTAFORME SUPPORTATE
File Storage	Amazon S3, Nirvanix, Azure Blob Storage, Rackspace Cloud Files
Document Storage Service	Amazon SimpleDB,Azure Table Storage
Queue Service	Amazon SQS, Azure TableStorage

Figura 2.9: Esempi di servizi offerti dalle SimpleCloud API

Il principio base di funzionamento di SimpleCloud comprende la presenza di un file di configurazione e di vari adapter, uno per ogni servizio di ciascuna piattaforma supportata. Lo sviluppatore è libero di decidere di utilizzare direttamente le API del provider scelto, limitando di fatto la portabilità del suo codice, oppure di utilizzare lo strato intermedio di astrazione che permette di scrivere codice portabile in uno qualsiasi dei vendor supportati.

mOSAIC

Il progetto mOSAIC [26] (Open-Source API and Platform for Multiple Clouds), finanziato dalla Commissione Europea, ha lo scopo di sviluppare una piattaforma open-source che verrà utilizzata come mezzo di comunicazione tra applicazioni e servizi Cloud. Le applicazioni saranno in grado di recuperare i requisiti di servizio richiesti dai loro utenti e inviarli alla piattaforma attraverso API. La piattaforma a questo punto, utilizzando un meccanismo multi-agente di intermediazione, eseguirà una ricerca dei servizi che corrispondono ai requisiti precedentemente espressi. Nel lungo periodo, ciò dovrebbe instaurare un contesto di maggiore concorrenza tra i fornitori di servizi Cloud.

mOSAIC, come raffigurato in Figura 2.10, è composto principalmente da due parti principali: il Resource Broker e l'Application Executor. Il Resource Broker è responsabile della negoziazione e prenotazione delle risorse ed è costituito da due sotto-sistemi, vale a dire l'interfaccia client e la Cloud agency. La prima descrive i fabbisogni e le richieste espresse dall'Application Executor, la seconda, invece, utilizza un insieme di strumenti tra cui: un monitor, un negoziatore, un mediatore, un registro di servizio, un motore semantico lato client e uno lato provider, un'ontologia Cloud e dei parametri QoS per

convalidare le specifiche dell'applicazione e generare gli SLA. L'Application Executor, cioè il responsabile della esecuzione delle applicazioni in base alle SLA definite dal contratto, è anch'esso composto da diversi sotto-sistemi: il motore di esecuzione delle API, i Providers Wrapper e il Resource Manager. Il motore di esecuzione delle API identifica le API utilizzate dall'utente per l'accesso alle risorse del provider. I Providers Wrapper sono speciali connettori che garantiscono un'interfaccia uniforme alle risorse disponibili in Cloud. Infine, il Resource Manager, utilizzando un Resource Scheduler e un Resource Monitor, assicura la disponibilità e la gestione delle risorse, ed è inoltre responsabile della gestione di eventuali richieste supplementari di risorse.

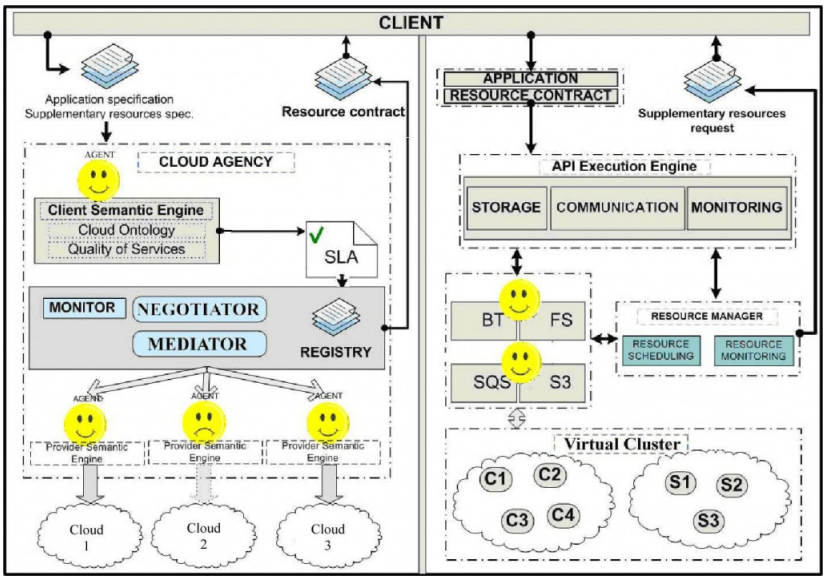


Figura 2.10: Architettura piattaforma mOSAIC

Utilizzando questo framework, chi sviluppa e mantiene le applicazioni Cloud è potenzialmente in grado di posticipare le decisioni riguardanti a chi deve offrire i servizi richiesti, al momento in cui questi verranno eseguiti.

Gli utenti finali, invece, saranno in grado di individuare i servizi Cloud che meglio corrispondono alle proprie esigenze.

Cloud4SOA

Il progetto Cloud4SOA [27] (A cloud interoperability framework and platform for user-centric, semantically enhanced service-oriented applications design, deployment and distributed execution) è un progetto finanziato nell'ambito

del Tema Tecnologie dell'informazione e della comunicazione (TIC) del Settimo programma quadro (7° PQ).

Coordinato dal fornitore di servizi IT Atos Origin [28], il consorzio Cloud4SOA mira a consolidare tre paradigmi informatici fondamentali e complementari, il Cloud Computing, le architetture orientate ai servizi (SOA) e la semantica leggera, con l'obiettivo di proporre un'architettura di riferimento che permetta l'interoperabilità tra diversi Cloud vendor, facilitando lo sviluppo, il caricamento e la migrazioni di applicazioni Cloud tra diversi PaaS provider.

Nello specifico, Cloud4SOA adotta un tecnologia semantica che permette agli sviluppatori e ai PaaS provider di esprimere i loro concetti con un vocabolario standardizzato e comuni relazioni. Questo permetterà un rilevante matchmaking tra le offerte e le richieste dei sviluppatori, provando inoltre ad indentificare gruppi di risorse complementari che potrebbero collaborare per offrire il servizio richiesto. Inoltre l'uso della semantica garantirà flessibililtà e facilità di adozione di future applicazioni e offerte che non esistevano al momento del caricamento del sistema Cloud4SOA. In Figura 2.11 viene rappresentata l'architettura ad alto livello componente il progetto Cloud4SOA.

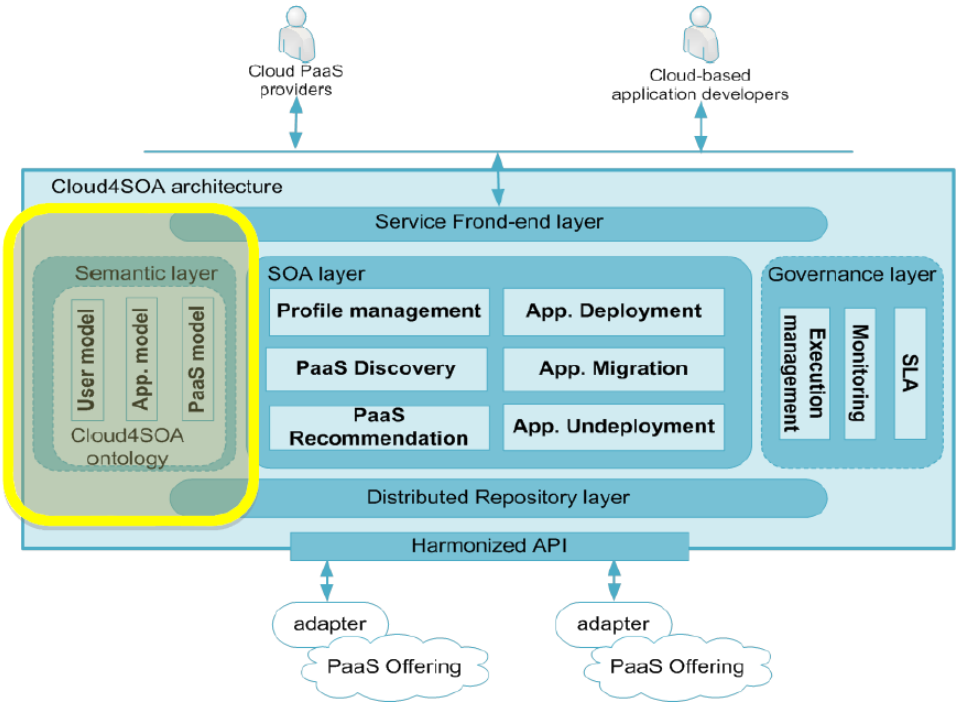


Figura 2.11: Architettura piattaforma Cloud4SOA

OCCI

OCCI, Open Cloud Computing Interface [29], è un protocollo e un set di API per tutti i tipi di attività di gestione.

Lo scopo iniziale di OCCI era creare delle API di gestione remota per i servizi basati su modello IaaS, consentendo lo sviluppo di strumenti interoperabili per le attività più comuni, tra cui: il deployment, l'automatic scaling e il monitoraggio. Da allora si è evoluto e la versione corrente di OCCI si presta a molti altri modelli oltre a IaaS, come ad esempio i PaaS e i SaaS.

CSAL

CSAL [30], Cloud Storage Abstraction Layer, fornisce un'astrazione dei servizi tipici di storage offerti dalla maggior parte dei PaaS: blob, table e queue.

La sfida di tale progetto è relativa alla mappatura di requisiti astratti in chiamate concrete a tali servizi, presentando le applicazioni con un namespace integrato.

L'attuale implementazione supporta le piattaforme di storage di Microsoft Windows Azure e quella di Amazon (S3, SimpleDB, and SQS).

2.3.6 Discussioni

Una classificazione dei lavori in ambito ¨Cloud Portability and Interoperability˝ è stata fornita in [31]. I vari approcci possono essere classificati a seconda che siano costituiti da, o usino i seguenti componenti:

- Open API: offrono delle API per l'accesso a piattaforme di Cloud provider diversi.
- Open Protocol: definiscono protocolli per l'accesso a piattaforme di Cloud provider diversi.
- Standard: la soluzione migliore in termini di portabilità ed interoperabilità, ma l'utilizzo diffuso di standard è ancora una sorta di utopia nel mercato commerciale del Cloud.
- Layer of abstraction: livelli di astrazioni ai servizi Cloud.

- Semantic repository: semantica nel livello di interfaccia, dei componenti o dei dati, utilizzando un generico modello di risorse Cloud.

- Domain specific language: strategia per permettere la creazione di applicazioni "Cloud neutral". Sono disponibili poche soluzioni di questo tipo sul mercato.

Nella Tabella 2.2, viene mostrato come si suddividono i lavori descritti nella sezione precedente, relativamente a questa classificazione.

Lavoro	Classificazione
IEEE P2301	Standard
VMware-Google Partnership	Domain specific languages
SimpleCloud	Open API
Mosaic	Open API
Cloud4SOA	Semantic repository
OCCI	Open Protocol
CSAL	Layer of abstractions

Tabella 2.2: Classficazioni lavori presentati

Il nostro approccio è orientato alla creazione di un livello di astrazione che, fornendo delle API generiche di accesso a particolari servizi Cloud, permette di scrivere applicazioni Cloud-independent.

Ci si differenzia dai lavori presenti in letteratura, simili per classificazione e per scopo, soprattutto SimpleCloud e CSAL, perchè si offre l'astrazione e le relative API, non solo per servizi relativi allo storage, quali i servizi che gestiscono schemi NoSQL, SQL, Blob oppure Queue, ma anche per i servizi di Memcache e di invio Mail.

Inoltre un aspetto interessante riguarda la possibilità di utilizzare due diverse semantiche del servizio Queue: infatti si offre la possibilità di memorizzare non solo semplici messaggi, ma anche richieste strutturate di esecuzione di lavori, da eseguire in modo asincrono rispetto ai flussi web.

Ulteriore aspetto distintivo riguarda le piattaforme supportate, cioè Google App Engine e Windows Azure. Non risultano ad oggi, infatti, progetti simili che supportino la portabilità di applicazioni tra queste due protagoniste del mercato PaaS.

Capitolo 3

Piattaforme supportate

Prima di addentrarci nella presentazione dell'approccio CPIM, si ritiene propedeutico analizzare nel dettaglio le due piattaforme attualmente supportate.

Attraverso descrizioni ed esempi di codice verrano presentati i servizi offerti da Google App Engine e quelli relativi alla piattaforma Cloud di Microsoft. Ci si focalizzerà principalmente nell'analisi dei servizi che risultano supportati dall'attuale versione della libreria CPIM.

Il Capitolo è strutturato nel seguente modo: nella Sezione 3.1 si descrive la piattaforma di Google, mentre in quella successiva, 3.2, viene presentata Windows Azure.

3.1 Google App Engine

Google App Engine è la piattaforma di hosting e sviluppo di applicazioni di Google. Essa consente di creare applicazioni web senza doverne gestire e progettare l'infrastruttura per la loro gestione. Lo sviluppatore deve solo preoccuparsi di fornire il codice della sua applicazione.

Le applicazioni che vengono caricate su Google App Engine utilizzano la stessa tecnologia che fornisce velocità e affidabilità ai siti web di Google.

3.1.1 Linguaggi di programmazione supportati

Google App Engine supporta applicazioni scritte in diversi linguaggi di programmazione.

Con l'ambiente runtime per Java di App Engine, è possibile creare applicazioni utilizzando le tecnologie standard di Java, tra cui la JVM, Java servlet, o qualsiasi altro linguaggio con un interprete o compilatore JVM-based, come ad esempio JavaScript o Ruby.

App Engine offre anche due ambienti runtime dedicati per Python, ognuno dei quali include un interprete e le librerie standard di Python. Infine, App Engine fornisce un ambiente runtime per Go, il quale gira compilato nativamente.

Questi ambienti runtime sono costruiti per garantire che l'applicazione venga eseguita in modo rapido, sicuro e senza interferenze da parte di altre applicazioni del sistema.

Per questa breve descrizione di Google App Engine ci si focalizzerà solamente sull'ambiente Java, essendo il linguaggio scelto per l'implementazione della libreria CPIM.

3.1.2 Servizi disponibili sulla piattaforma

L'ambiente runtime di Java espone API per diverse tipologie di servizi, alcuni riguardanti il Data Storage, come ad esempio il servizio Datastore, che permette di memorizzare oggetti secondo un sistema ¨schema-less˝, servizi relativi alla computazione, quale ad esempio il Task Queue Service che permette di eseguire task in modo asincrono rispetto alle richieste degli utenti, servizi di mailing, per la gestione dell'autenticazioni, etc.

Nelle sezioni che seguono si presenteranno nel dettaglio i servizi supportati dall'attuale versione della libreria CPIM, e in modo piú sintetico alcuni tra quelli non attualmente gestiti.

Servizi supportati

I seguenti servizi sono supportati dalla versione attuale della libreria CPIM.

Blobstore

Le Blobstore API permettono ad una applicazione di servire tutti quegli oggetti che risultano essere troppo grossi per essere memorizzati nel Datastore, i cosiddetti Blob. Il servizio concede la possibilità di caricare oggetti con una

dimensione massima pari a 32MB. Inoltre i Blob salvati vengono organizzati secondo uno schema piatto, ossia tutti i Blob sono contenuti nella ¨root¨ dedicata all'applicazione.

La creazione di un Blob, i cui passi sono riportati in Figura 3.1, avviene attraverso il caricamento di un file tramite richieste HTTP.

Tipicamente, le applicazioni che permettono questa operazione, presentano un form contenente un campo di tipo ¨file¨ all'utente. Quando il form è inviato, questo servizio crea un blob dal contenuto del file e restituisce un riferimento opaco ad esso, chiamato blob key, tramite il quale è poi possibile referenziarlo. Nel Datastore vengono salvate alcuni metadati relativi al blob caricato, quali ad esempio la blobKey, la data di creazione, il nome del file, le dimensioni e il tipo, etc.

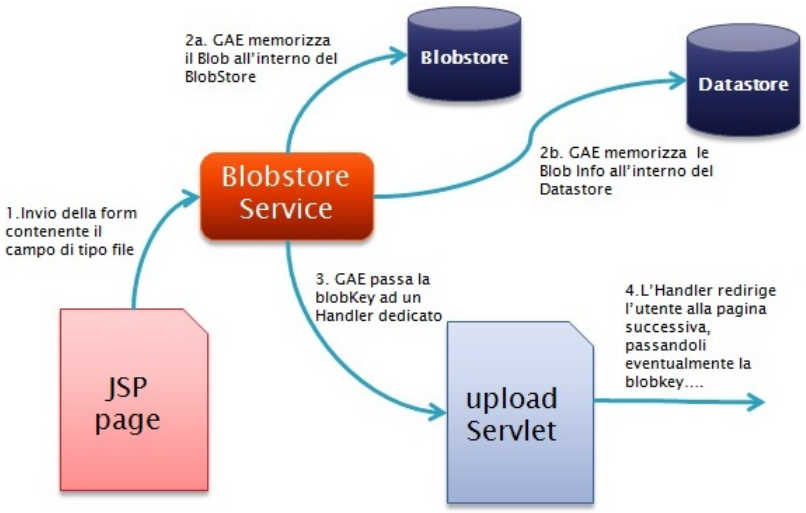


Figura 3.1: Flusso operazioni caricamento Blob

Oltre a questa modalità, è possibile creare blob direttamente via codice, utilizzando API simili a quelle relative alle operazioni su file. Il flusso logico delle operazioni di caricamento di un Blob è riportato in Figura 3.2.

Tale modalità è la sola supportata dalla libreria CPIM. La motivazione di questa scelta è dovuta alla sua affinità con la modalità di gestione dei Blob in Azure. Questo aspetto, verrà ripreso e dettagliato in seguito, nella Sezione 4.3.5 inerente la standardizzazione del servizio nella libreria sviluppata.

Le fasi necessarie per effettuare il caricamento di un Blob nel Blobstore di App Engine utilizzando le ¨File-like¨ API sono:

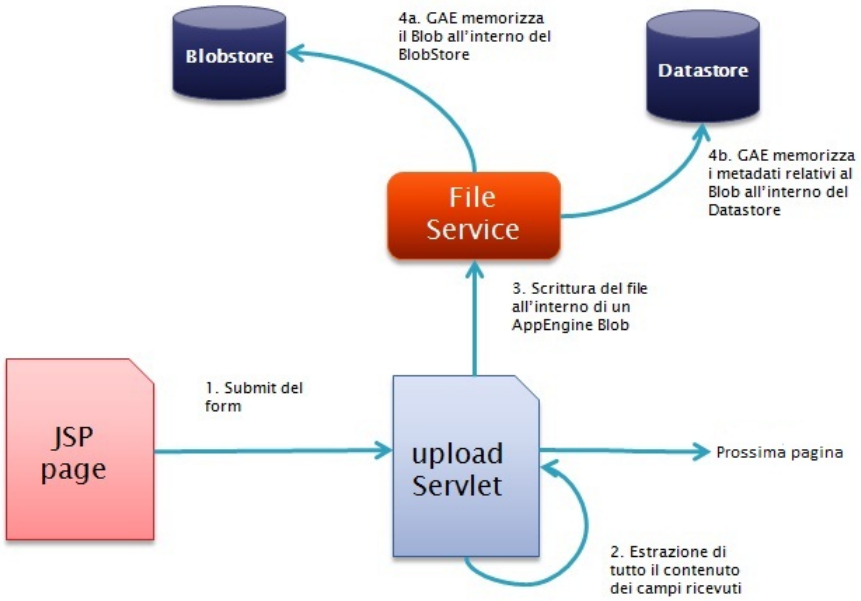


Figura 3.2: Flusso operazioni caricamento Blob con FileService

FASE 1: Form

La prima fase del caricamento consiste nella creazione di una form HTML (vedi esempio Listato 3.1) contenente un campo di tipo ˝file˝ con enctype di tipo ˝multipart/form-data˝.

Listato 3.1: Esempio form con un campo di tipo file

```
1 <form action="NextServlet" method="post" enctype="multipart/form-data">
2 <input type="file" name="myFile">
3 <input type="submit" value="Submit">
4 </form>
```

FASE 2: Ricezione e gestione dati ricevuti dalla form

Successivamente i dati inviati dalla form, compreso quello relativo al file, vengono gestiti da una Servlet. Attualmente, la versione supportata delle Servlet API di Google App Engine è la 2.5, la quale non permette l'accesso in modo efficiente ai campi di tipo ˝multipart/form-data˝ inviati tramite richieste HTTP.

Una soluzione consiste nell'utilizzo della libreria CommonFileUpload. Essa permette la distinzione tra campi di tipo ˝multipart/form-data˝ e non.

Nel Listato 3.2, viene riportato un semplice esempio di gestione dei dati ricevuti tramite chiamate HTTP POST, utilizzando la libreria sopra citata. Per

prima cosa occorre istanziare la classe *ServletFileUpload* adibita alla gestione delle richieste con encoding type di tipo multipart (riga 1), cioè quei form che contengono, fra gli altri, anche campi di tipo file. Attraverso essa, si ottiene un'iteratore sugli oggetti ricevuti (riga 2). La gestione dei campi di tipo file e non, è distinguibile attraverso la chiamata al metodo *item.isFormField()*, che ritorna *true* se non è un campo file, *false* viceversa (riga 7). Utilizzando poi, un classico *InputStream* è possibile salvare il contenuto del campo attualmente puntato dall'iteratore (riga 6). A seconda del suo tipo, tale contenuto verrà formattato in una Stringa per i campi di tipo *text* (riga 9) e in un vettore di byte per i campi di tipo *file* (riga 12).

Listato 3.2: Esempio gestione campi ricevuti con libreria CommonsFileUpload

```
1  ServletFileUpload upload = new ServletFileUpload();
2  FileItemIterator iterator = upload.getItemIterator(req);
3  while (iterator.hasNext()) {
4            FileItemStream item = iterator.next();
5          //attraverso uno stream si ottiene il contenuto del campo attualmente
6            InputStream stream = item.openStream();
7            if (item.isFormField()){//qui gestione dei campi non di tipo file
8             String field = item.getFieldName();
9                 String value = Streams.asString(stream);
10           }
11           else {   // gestione campi di tipo file ricevuti
12                   byte[] buffer = IOUtils.toByteArray(stream);
13                //metodo gestione file ricevuto
14                   uploadFile(buffer, fileName);
15                            }
16 }
```

FASE 3: Creazione di un AppEngine Blob dal file inviato tramite form
Il Listato 3.3 mostra un semplice esempio di creazione di un Blob. Essenzialmente, le fasi di questa operazione sono:

- Creazione di un AppEngine Blob (riga 3)
- Apertura di un canale remoto (riga 5)
- Scrittura, tramite il canale remoto, del contenuto del file inviato dalla form nel Blob appena creato (riga 7)

Terminata la scrittura, viene salvato il Blob nel Blobstore e i relativi metadati nel Datastore.

Listato 3.3: Esempio creazione di un App Engine Blob

```
1 //Scrittura del file da caricare all'interno di un Blob
2 String contentType = mftm.getContentType(fileName);
3 AppEngineFile blobToUpload = fileService.createNewBlobFile(contentType, fileName);
4 boolean lock = true;
5 FileWriteChannel wc = fileService.openWriteChannel(blobToUpload, lock);
6 blobToUpload = new AppEngineFile(blobToUpload.getFullPath());
7 wc.write(ByteBuffer.wrap(blobToUpload));
8 wc.closeFinally();
```

Datastore

Fino al Giugno 2012 la piattaforma App Engine non disponeva di un servizio che offrisse la persistenza di dati attraverso database relazionali, ma metteva però a disposizione dei propri utenti una base di dati distribuita che si differenziava nettamente dal modello relazionale, tanto da essere denominata datastore e non database come ci si sarebbe potuti aspettare.

Il Datastore di App Engine è infatti un tipico esempio di sistema di memorizzazione a oggetti di tipo ‶schema-less″.

L'unità fondamentale di memorizzazione è l'Entity, formate da una Key immutabile e da zero o più proprietà mutabili. Queste Entity possono essere create, aggiornate, cancellate, caricate fornendo la Key e ricercate per valore delle proprietà utilizzando un linguaggio SQL-like: chiamato Google Query Language (GQL).

GQL si differenzia dal più noto linguaggio SQL per alcune limitazioni, la più importante delle quali riguarda l'impossibilità di effettuare operazioni di join.

La ragione che inizialmente avevo spinto Google verso questa scelta era essenzialmente l'esistenza di una piattaforma di persistenza ad elevata scalabilità, basata sulla tecnologia BigTable sviluppata internamente e utilizzata a supporto dei propri servizi, come la web search, GMail, Google Maps e altri, e quindi ampiamente collaudata. Tale piattaforma richiedeva quindi un adattamento software minimale per la costruzione dell'interfaccia di App Engine.

Al contrario, la configurazione di un DBMS tradizionale al fine di garantire un ottimo compromesso tra scalabilità e affidabilità comportava una complicazione non banale, in quanto richiedeva l'introduzione di meccanismi di replicazione e load balancing che dovevano essere attentamente configurati.

La tecnologia BigTable può essere definita come una mappa:

- sparsa: memorizza solo valori non vuoti
- distribuita: situata nella Cloud di Google
- persistente: memorizzata sul Google File System
- multidimensionale: relativamente al numero di colonne
- ordinata: lessicograficamente per chiave

La SDK per App Engine di Java fornisce un set di API di basso livello per l'accesso diretto al Datastore, includendo semplici operazioni su Entity, quali la get, la put, la delete, e semplici query. Inoltre sono implementate ed esposte le interfacce Java Data Objects (JDO) e Java Persistence API (JPA) per la modelizzazione e la memorizzazione di oggetti. Queste interfacce standard includono meccanismi per la definizioni di classi di oggetti che verranno poi rimappate in Entity.

Ci si focalizzerà, attraverso semplici esempi, sull'uso della JPA per l'accesso al Datastore, poichè risulta essere il meccanismo utilizzato nella libreria CPIM. Le motivazione di questa scelta sono descritte nella sezione 4.3.1.

In [32] sono dettagliati tutti i passi da effettuare durante la fase iniziale di configurazione, quali la scrittura del file *persistence.xml*, la collocazione dei JAR file della JPA e del datastore nella cartella WEB-INF del war dell'applicazione, e l'enhancement delle classi, cioè quella particolare operazione eseguita post-compilazione che associa le classi Java all'implementazione JPA. Ciascun oggetto salvato tramite JPA diventa una Entity nel Datastore.

Per permettere che una classe Java sia rimappata in una Entity, occorre dare alla classe l'annotazione *@Entity*. Tutti gli attributi di tale classe diventeranno property della Entity. Uno dei quali, inoltre, rappresenterà la chiave per il riconoscimento univoco di tale Entity e sarà etichettato con l'annotazione *@Id*. Si veda a titolo di esempio il Listato 3.4.

Listato 3.4: Esempio di una classe java con le annotazioni JPA

```
1  @Entity
2  public class User {
3      // la chiave della Entity User
4      @Id
5      private String email;
6      //alcune properties
7      private String nome;
8      private String cognome;
9
10     //getter&setter
11 }
```

Per poter effettuare operazioni sul Datastore attraverso la JPA, occorre inanzitutto istanziare una classe che funga da gestore di queste operazioni: la classe *EntityManager*. L'istanziazione di quest'ultimo avviene tramite il relativo oggetto factory, rappresentato dalla classe *EntityManagerFactory*. La creazione di tale oggetto impiega diverso tempo, quindi è consigliato riutilizzare la stessa istanza diverse volte, attraverso ad esempio la creazione di una classe singleton, istanziabile tramite un metodo statico. Per un esempio, si veda l'implementazione della classe EM del Listato 3.5 (righe 4-13), che espone il metodo statico *get()* per tale scopo.

Sempre nel Listato 3.5 si mostra l'istanziazione di un *EntityManager* (riga 38) e la memorizzazione di un'entità, definita dalla classe *User*, tramite il metodo *persist()* (riga 41). Per un elenco completo delle operazioni possibili sul Datastore tramite *EntityManager* si veda [33].

Listato 3.5: API per ottenere un'istanza della classe EntityManager

```
1  //EMF.java
2  import javax.persistence.EntityManagerFactory;
3  import javax.persistence.Persistence;
4  public final class EMF {
5      //PersistentUnit si riferisce al nome dell'insieme delle configurazioni
6      //definite all'interno del file persistence.xml.
7      private static final EntityManagerFactory emfInstance = Persistence.
         createEntityManagerFactory(PersistentUnit);
8
9      private EMF() {}
10     public static EntityManagerFactory get() {
```

```
11          return emfInstance;
12      }
13  }
14
15  //User.java
16  @Entity(name="User")
17  public class User implements Serializable {
18      @Id
19      private String email;
20      private String firstName;
21      private String lastName;
22      private Integer age;
23
24      public User(String email, String firstName,String lastName, Integer age) {
25          this.email = email;
26          this.firstName = firstName;
27          this.lastName = lastName;
28          this.age = age;
29
30      }
31     //getters and setters
32  }
33
34  //Other File.java
35  import javax.persistence.EntityManager;
36  import javax.persistence.EntityManagerFactory;
37  //...
38      EntityManager em = EMF.get().createEntityManager();
39       try {
40           User user=new User(email,firstName,lastName,age)
41           em.persist(user);
42           ...
43              } finally {
44          em.close();
45      }
46  //...
```

Memcache Service

Questo servizio rende disponibile una memoria cache, distribuita e condivisa dalle istanze, per la memorizzazione o il recupero dei dati con prestazioni molto

più elevate rispetto al Datastore, utilizzando uno schema di tipo chiave-valore.

I dati risiedono in memoria principale, pertanto un fallimento dei server provocherebbe la cancellazione della relativa cache.

L'utilizzo tradizionale prevede che le applicazioni verifichino la possibilità di soddisfare le proprie query utilizzando i dati presenti all'interno della memcache, ricevendo così una risposta molto più veloce rispetto a quanto si potrebbe avere interrogando gli altri sistemi di storage, tipo il Datastore. Solo in caso di mancato hit nella memcache vengono richiesti i dati ad essi.

La SDK di App Engine fornisce API sia per una versione sincrona del servizio Memcache (MemcacheService), sia per una asincrona (AsyncMemcacheService). La differenza tra le due versioni è che nella modalità asincrona, le chiamate alle Memcache, come ad esempio *asyncCache.get(key)* ritornano immediatamente un oggetto della classe *Future<Object>*, descritta in [34], e l'esecuzione procede in parallelo rispetto a tale richiesta. Successivamente, tramite l'istanza ricevuta di tale classe, si richiede in modo sincrono il risultato della richiesta precedentemente inviata, attraverso la chiamata del metodo *get()*.

A titolo di esempio si presenta nel Listato 3.6 il codice per una *get* di un dato per entrambe le tipologie di Memcache.

Listato 3.6: API utilizzo servizio Memcache

```
1  String key = ..
2  byte[] value;
3  // Utilizzo cache sincrona, l'esecuzione attende il risultato
4  MemcacheService syncCache = MemcacheServiceFactory.getMemcacheService();
5  //lettura dalla cache del dato con chiave "key"
6  value = (byte[]) syncCache.get(key);
7  if (value == null) {
8    // valore non trovato nella Memcache, si cerca in altre sorgenti
9    // value=...;
10   //si salva il valore per eventuali altre richieste
11   syncCache.put(key, value);
12 }
13
14 // Esempio Utilizzo cache asincrona
15 AsyncMemcacheService asyncCache = MemcacheServiceFactory.
     getAsyncMemcacheService();
16 //Invio richiesta lettura dato alla cache
```

```
17    Future<Object> futureValue = asyncCache.get(key);
18    // Eseguo altro lavoro in parallelo
19    //....
20    //Ora verifico risultato richiesta alla Memcache
21    value = (byte[]) futureValue.get();
22    if (value == null) {
23      // valore non trovato nella Memcache, si cerca in altre sorgenti
24      // value=...;
25      //si salva il valore in modo asincrono per eventuali altre richieste
26      asyncCache.put(key, value);
27    }
```

Google Cloud SQL

Google Cloud SQL è un database MySQL collocato nella Cloud di Google. Esso offre tutte le risorse e le funzionalità di MySQL, con alcune limitazioni dovute a dalle funzionalità non supportate. Le limitazioni attualmente presenti riguardano:

- Limite di grandezza di 10GB per istanza
- Funzioni definite dall'utente non supportate
- Replicazione MySQL non supportata
- Alcune operazioni di basso livello non sono supportate, come ad esempio la *SELECT ... INTO OUTFILE/DUMPFILE*

Per utilizzare Google Cloud SQL, occorre effettuare le seguenti operazioni:

- Abilitazione billing per questo servizio
- Creazione di una nuova istanza, definendone il nome e le dimensioni
- Definizione dei diritti accesso. Si elencano gli identificativi delle applicazioni che hanno la facoltà di utilizzo dell'istanza creata
- Creazione Database e relative tabelle

A questo punto è possibile connettere le applicazioni ai DB seguendo i seguenti passi:

- Registrazione del driver JDBC (riga 4 del Listato 3.7)
- Connessione al DB (riga 6 del Listato 3.7)
- Predisposizione e invio richieste al DB (righe 9-11 del Listato 3.7)

Listato 3.7: API utilizzo servizio Google Cloud SQL

```
1  Connection c = null;
2     try {
3       //Registrazione driver JDBC
4       DriverManager.registerDriver(new AppEngineDriver());
5       //Creazione connessione al Db "mydb"
6       c = DriverManager.getConnection("jdbc:google:rdbms://instance_name/mydb");
7       //Preparazione comando SQL da eseguire sul Db "mydb"
8       String statement ="SQL_Statement";
9       PreparedStatement stmt = c.prepareStatement(statement);
10      //Esecuzione del comando
11      success = stmt.executeUpdate();
12      } catch (SQLException e) {
13        e.printStackTrace();
14    } finally {
15       if (c != null)
16         try {
17              //Chiusura connessione al Db
18            c.close();
19           } catch (SQLException ignore) {
20           }
21     }
```

Task Queue Service

Con le API relative al servizio Task Queue, Google offre la possibiltà di creare e gestire coda di attività, permettendo alle applicazioni di eseguire lavori richiesti da un utente in modo asincrono rispetto alle richieste web.

Utilizzando questo servizio, le applicazione con particolari necessità computazionali, possono organizzare i vari lavori da eseguire in piccole unità discrete, chiamate Task, e possono richiederne l'esecuzione inserendo in una TaskQueue un'istanza della classe *TaskOptions*, contenente tutte le informazioni utili per la loro esecuzione. I vari *TaskOptions*, residenti nelle code, saranno periodicamente estratti e verranno eseguiti i Task associati ad essi.

Le applicazioni Java definiscono le Task Queue utilizzate in un file di configurazione, chiamato *queue.xml*, collocato all'interno della cartella WEB-INF. Un esempio di tale file è riportato nel Listato 3.8.

Listato 3.8: Esempio di un file queue.xml

```
1 <queue-entries>
2   <queue>
3     <name>myqueue</name>
4     <rate>1/s</rate>
5   </queue>
6 </queue-entries>
```

Attualmente la piattaforma supporta due tipologie di Task Queue:

- Push Queues
- Pull Queues

Push Queues

Utilizzando questa tipologia di Task Queue, l'applicazione si deve solo preoccupare di inserire nella coda il relativo *TaskOptions*.

Elemento fondamentale che ogni *TaskOptions* inserito in questa tipologia di Task Queue deve contenere, è l'URL dell'unità lavorativa da eseguire, che in ambito Java è comunemente rappresentata da una Servlet. Infatti è App Engine che, in modo del tutto automatico e ad un tasso definito dal tag *<rate>* relativo alla coda configurata nel file *queue.xml*, invoca l'esecuzione delle unità lavorative presenti nella coda e, in caso di esecuzione terminata positivamente, cancella la richiesta. Inoltre non ci si deve neanche preoccupare della scalabilità dell'esecuzione di queste unità lavorative, poichè è anch'essa demandata alla piattaforma.

L'esecuzione di un Task deve però terminare con una risposta HTTP con codice 200-299 entro i 10 minuti dall'invio della richiesta. Se si desiderasse eseguire Task che richiedono maggior tempo per la loro esecuzione, si deve utilizzare il servizio di Backends di App Engine, descritto in [35].

Importante sottolineare che tutti le richieste inserite in una Push Queue devono referenziare Task con URL interno all'applicazione che ne richiede l'esecuzione. Per richiamare Task "remoti"occorre utilizzare Task Queue di tipo Pull.

In Figura 3.3 viene riportato uno schema qualitativo di esecuzione e gestione di un Task e nel Listato 3.9 viene presentato un esempio dell'utilizzo delle API Java per tale servizio, contenente tutte le operazioni standard eseguite, cioè la richiesta del riferimento di una particolare coda (riga 2), la successiva creazione di un'istanza della classe TaskOptions (riga 3), contenente l'URL della servlet che rappresenta l'unità lavorativa da eseguire, ed infine l'aggiunta del Task nella Coda (riga 4).

Listato 3.9: Esempio di utilizzo della Push Queues APIs

```
1 //...
2 Queue queue = QueueFactory.getQueue(queueName);
3 TaskOptions taskOptions = TaskOptions.Builder.withUrl("/WorkToDo")
4 queue.add(taskOptions);
5 //...
```

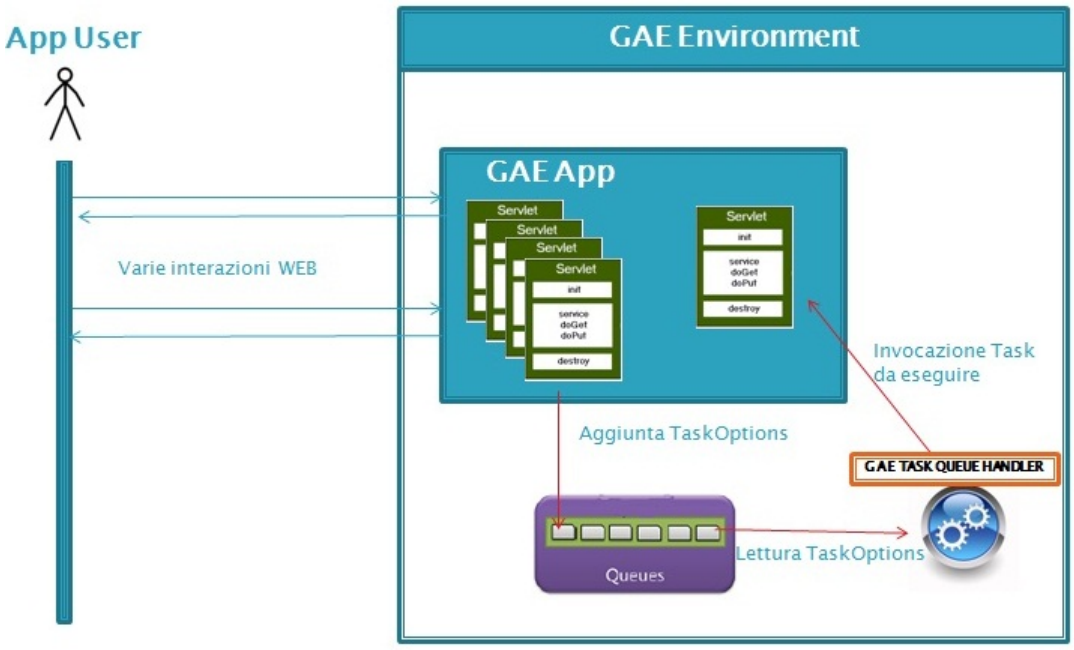


Figura 3.3: Esecuzione di un Task in una TaskQueue di tipo Push

Pull Queues

Le Pull Queues permettono di progettare il proprio sistema di gestione di Task. Tali gestori possono essere parte del sistema App Engine o anche di sistemi esterni ad esso.

Per definire una coda di tipo Pull occorre utilizzare il tag *<mode>pull </mode>* nella configurazione della coda contenuta nel file *queue.xml*. Un esempio è riportato nel Listato 3.10.

Listato 3.10: Definizione di una Pull Queue nel file queue.xml

```
1 <queue-entries>
2   <queue>
3     <name>mypullqueue</name>
4     <mode>pull</mode>
5   </queue>
6 </queue-entries>
```

Utilizzando questa tipologia di Task Queue sono richieste la gestione manuale di alcune operazioni che nelle Push Queues erano automatiche:

- La gestione della scalabilità dei gestori della coda in base ai volumi di elaborazioni
- La cancellazione dei Task al termine dell'esecuzione.

La differenza principale per queste tipologie di Task Queue, che si nota anche dal Listato 3.11, è che, dopo l'usuale richiesta del riferimento alla coda, si crea un TaskOptions settando campi differenti rispetto a quelli delle Push Queues. In particolare non è obbligatorio indicare alcun URL, ma spesso si inseriscono contenuti nel campo Payload (si veda riga 5), che rappresentano un messaggio che dovrà essere interpretato dal lettore ed esecutore del Task, progettato e implementato dal programmatore. Questo Consumer dovrà essenzialmente:

- Leggere i TaskOptions dalla Pull Queue, attraverso il metodo *leaseTask* (si veda esempio riga 5 Listato 3.12)
- Interpretarli ed eseguire il lavoro ad essi associato
- Cancellarli attraverso l'invocazione del metodo *deleteTask* (si veda esempio riga 8 Listato 3.12)

Listato 3.11: Esempio di utilizzo Pull Queues API

```
1 //PRODUCER (APPLICATIVO)
2 //Referenza coda di tipo Pull
```

```
3 Queue q = QueueFactory.getQueue("pull-queue");
4 //Aggiunta di un Task in una PullQueue
5 q.add(TaskOptions.Builder.payload("Messaggio_per_il_Consumer").taskName("foo"));
```

Listato 3.12: Esempio di utilizzo Pull Queues API

```
1 //CONSUMER
2 //Esempio di get di 100 messaggi da una Pull Queue
3 //Per 3600 secondi questi task non saranno visibili nella Pull Queue.
4 //Dovranno essere cancellati se l'esecuzione ha avuto esito positivo
5 List<TaskHandle> tasks = q.leaseTasks(3600, TimeUnit.SECONDS, 100);
6 //... processo dei messaggi letti
7 //Esempio cancellazione del Task chiamato "foo"
8 q.deleteTask("foo");
```

Mail Service

Le applicazioni che girano su App Engine sono in grado di inviare e ricevere messaggi di posta elettronica grazie all'utilizzo del servizio Mail presente sulla piattaforma. In particolare, un messaggio può essere inviato a nome degli amministratori delle applicazioni, e/o per conto di utenti con account Google.

Nel Listato 3.13 si mostra un esempio di invio di una mail tramite tale servizio.

Listato 3.13: API per invio di una email

```
1 // ...
2       Properties props = new Properties();
3       //tramite questo metodo si crea una sessione autenticata
4       Session session = Session.getDefaultInstance(props, null);
5       //testo del messaggio
6       String msgBody = "Hello World";
7
8       try {
9           //istanziazione oggetto Wrapper del messaggio da inviare
10          Message msg = new MimeMessage(session);
11          //set del mittente
12          msg.setFrom(new InternetAddress("admin@example.com", "Example.com
        Admin"));
13              //set del destinatario
14          msg.addRecipient(Message.RecipientType.TO,
15                      new InternetAddress("user@example.com", "Mr. User"));
```

```
16            //set del oggetto della mail
17            msg.setSubject("Your Example.com account has been activated");
18            //set del contenuto della mail
19            msg.setText(msgBody);
20            //invio della mail
21            Transport.send(msg);
22
23        } catch (AddressException e) {
24            // ...
25        } catch (MessagingException e) {
26            // ...
27        }
```

Per quanto riguarda la ricezione dei messaggi, le applicazione in App Engine ricevono le email in un indirizzo del tipo *string@appid.appspotmail.com*. I messaggi ricevuti sono inviati dai mittenti utilizzando chiamate HTTP di tipo POST all'URL */_ah/mail/address*. Un esempio di ricezione di una mail è riportato nel Listato 3.14.

Listato 3.14: API per ricezione di una email

```
1  public class MailHandlerServlet extends HttpServlet {
2      public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
         IOException {
3          Properties props = new Properties();
4          //creazione di una sessione autenticata
5          Session session = Session.getDefaultInstance(props, null);
6          //Ricezione del messaggio
7          MimeMessage message = new MimeMessage(session, req.getInputStream());
8          //... gestione attraverso tale oggetto del messaggio ricevuto
9      }
10 }
```

Per dare la possibilità alle applicazioni di ricevere email, occorre aggiungere nel file *appengine−web.xml*, situato all'interno della cartella WEB-INF del war, il codice riportato nel Listato 3.1.2.

```
1  <inbound−services>
2      <service>mail</service>
3  </inbound−services>
```

Servizi attualmente non supportati

Di seguito vengono descritti sinteticamente alcuni dei servizi offerti dalla piattaforma di Google App Engine che non sono supportati dall'attuale versione della libreria CPIM.

Capabilities

Utilizzando questo servizio, l'applicazione è in grado di rilevare interruzione e/o periodi di inattività di un specifico servizio o risorsa. Si può sfruttare tale servizio per ridurre i periodi di inattività di un'applicazione, bypassando le risorse non disponibili in un dato istante.

Channel

Questo servizio crea una connessione persistente tra un'applicazione e i server di Google. In questo modo è possibile inviare messaggi a client Javascript in tempo reale senza dover utilizzare il polling. Questo risulta essere particolarmente utile per quelle applicazioni che sono progettate per aggiornare l'utente in modo tempestivo su nuove informazioni disponibili e inviare rapidamente in modalità broadcasting agli altri utenti gli input immessi da esso. Esempio tipico che sfrutta questo servizio sono le Chat.

Images

Il servizio Images offre l'opportunità di manipolare immagini. È possibile ridimensionare, ruotare, capovolgere, ritagliare immagini e inoltre attraverso un algoritmo predefinito, si ha la possibilità di migliorare la qualità delle fotografie.

LogService

LogService è il servizio che permette l'accesso tramite API dei log relativi all'applicazione.

Multitenancy

Le Namespaces API di Google App Engine rendono facile la compartimentazione dei dati sulla piattaforma. Utilizzando un apposito manager è possibile settare un particolare namespace e renderlo utilizzabile globalmente. Il termine

inglese Multitenancy è utilizzato per identificare quella classe di architetture software nelle quali un'istanza di un'applicazione, che gira su un server remoto, è utilizzata da diverse classi di utenti, chiamati tenant. Per implementare la multitenancy, si utilizzano le Namespaces API, che possono facilmente partizionare i dati tra i vari tenant, semplicemente specificando un diverso namespace per ciascun tenant.

OAuth

Questo servizio è ancora in fase sperimentale. In particolare OAuth è un protocollo che permette ad un utente di concedere ad una terza parte dei permessi limitati di accesso ad un'applicazione a nome suo, senza condividere con esso le proprie credenziali.

Prospective Search

Questo servizio è ancora in fase sperimentale. Esso permette ad un'applicazione di registrare un gran numero di query che verrano poi simultaneamente eseguite su un flusso di documenti di input. Per ciascuno di questi documenti, Prospective Search ritorna un ID di tutte le query che hanno dato risultato positivo.

Prospective search è particolarmente utile per la costruzione di applicazioni che processano flussi di dati, come quelle che prevedono notifiche in caso di verifica di un particolare evento, monitoraggio o servizi di filtro.

Search

Questo servizio, ancora in fase sperimentale, offre alle applicazioni la possibilità di eseguire ricerche *Google-like* su dati strutturati all'interno dell'applicazione. È possibile eseguire ricerche full-text tra diversi tipi di testo (testo semplice, HTML, e altri). Questo servizio restituisce un elenco ordinato del testo corrispondente, ed è inoltre possibile personalizzare la classifica dei risultati.

URL Fetch

Le applicazioni che girano su App Engine sono potenzialmente in grado di comunicare con altre applicazioni o accedere ad altre risorse sul web attraverso i

relativi URL. Un'applicazione può utilizzare questo servizio per inviare richieste e/o ricevere risposte HTTP e HTTPS. URL Fetch utilizza l'infrastruttura di rete di Google.

Users

Le applicazioni App Engine possono autenticare gli utenti che dispongono di account Google o sul proprio dominio di Google Apps, oppure di identificatori di OpenID. Un'applicazione è quindi in grado di rilevare se l'utente corrente ha effettuato l'accesso, reindirizzandolo, in caso negativo, alla pagina di accesso per inserire le proprie credenziali o creare un nuovo account. L'applicazione può rilevare, inoltre, se l'utente autenticato è un amministratore, il che rende facile da implementare quelle aree particolari dell'applicazione accessibili solo se si dispone dei privilegi da amministratore.

XMPP

Le applicazione App Engine possono inviare e ricevere instant message da utenti che utilizzano servizi XMPP compatibili, compreso Google Talk.

3.2 Windows Azure

Windows Azure [7] è la soluzione PaaS pubblica di Microsoft. Questa piattaforma può essere utilizzata in diversi modi. Per esempio, Windows Azure può essere utilizzata per creare un'applicazione web che esegua un lavoro computazionale più, o meno, complesso e memorizzi i propri dati sui datacenter di Microsoft. È inoltre possibile utilizzare la piattaforma solo come storage server in cui memorizzare i dati, i quali possono essere usati da applicazioni eseguite all'esterno del Cloud, attraverso chiamate HTTP autenticate.

Windows Azure offre la possibilità di creare macchine virtuali chiamate *role* nelle quali caricare ed eseguire applicazioni fortemente scalabili. La differenziazione classica proposta dalla piattaforma è la suddivisione dell'applicazione in due diverse tipologie di role, chiamate Web e Worker role.

Nella Web role generalmente viene ospitato il front end e quindi la parte di presentazione in cui la computazione richiesta è limitata, mentre nella Wor-

ker role viene ospitata la parte di backend, in cui viene richiesto uno sforzo computazionale maggiore.

A seconda del ambiente di runtime richiesto, e quindi a seconda del linguaggio di sviluppo utilizzato, la differenziazione delle tipologie cambia.

Di seguito verranno presentati dapprima i linguaggi di programmazione supportati dalla piattaforma, per poi passare, in ambito Java, alla descrizione dei servizi offerti dalla piattaforma.

3.2.1 Linguaggi di programmazione supportati

Windows Azure supporta applicazioni scritte in molti linguaggi di programmazione.

L'ambiente di runtime maggiormente supportato dai servizi offerti dalla piattaforma è .NET, proprietario di Microsoft. Le applicazioni sviluppate in C# o in Visual Basic possono interagire direttamente con tutti i servizi offerti attraverso le API fornite dall'SDK, senza la necessità di integrare servizi di terze parti.

Tale ambiente di runtime viene supportato solamente dalla piattaforma di Microsoft, e quindi per l'obiettivo del lavoro discusso in questa tesi non risulta utilizzabile.

Oltre all'ambiente .NET, la piattaforma di Windows Azure, ha la possibilità di ospitare applicazioni scritte in molti altri linguaggi. Infatti, per permettere tale flessibilità, la piattaforma mette a disposizione una tipologia di role chiamata VM (Virtual Machine). In questo caso Windows Azure assume un ruolo di servizio IaaS, in cui poter configurare le proprie macchine virtuali, munendole dell'ambiente di runtime, necessario per l'esecuzione delle applicazioni sviluppate in un determinato linguaggio di programmazione. L'interazione con i servizi di Windows Azure è permessa grazie all'uso di chiamate HTTP/REST.

L'ambiente utilizzato nell'implementazione della libreria CPIM è quello di Java. In questo caso, ogni macchina virtuale di Azure, per poter eseguire l'ambiente Java, ha la necessità di eseguire una Java Virtual Machine ed un server (ad esempio Tomcat).

3.2.2 Servizi disponibili sulla piattaforma

Windows Azure offre, per l'ambiente di sviluppo Java, i seguenti servizi.

- Gestione dei dati:
  - Blob Service
  - Table Service
  - Caching Service
  - SQL Database
- Code:
  - Queue Service
  - Service Bus Queues
  - Service Bus Topics
- Servizi di comunicazione:
  - SendGrid Email Service
  - Twilio Voice and SMS Service
- Sicurezza:
  - Access Control

Nelle sezioni che seguono si presenteranno tutti i servizi sopra elencati, soffermandosi in modo particolare sulla trattazione dei servizi supportati dalla libreria.

Servizi supportati

I seguenti servizi, offerti da Windows Azure per l'ambiente di sviluppo Java, sono supportati dall'attuale versione della libreria CPIM.

Per quanto riguarda la persistenza, ossia Blob, Queue e Table Service, si utilizza una libreria denominata *jpa4azure*, la quale implementa e costruisce le chiamate HTTP necessarie per l'interazione con tali servizi.

Blob Service

Il Blob Service di Windows Azure è stato progettato per memorizzare dati binari non strutturati. Tale servizio fornisce uno storage economico in cui salvare blob, i quali possono arrivare fino ad un terabyte di dimensione ciascuno.

Per usare il servizio, occorre prima di tutto creare un storage account di Windows Azure. In tale storage account, ogni blob salvato deve risiedere in un container, concetto simile ad una cartella del file system di Windows. È un concetto simile, ma non uguale, dato dal fatto che non vi è la possibilità di creare delle gerarchie di container, in quanto è permesso un solo livello. Comunemente, per ovviare a questo problema, vengono utilizzati dei nomi di container formattati con uno specifico pattern, in modo da simulare il concetto di sotto container, come ad esempio *ContainerX* per il container di primo livello e *ContainerX_ContainerY* per quello di secondo livello.

Per quanto riguarda l'accesso ad un blob, è necessario effettuare una chiamata HTTP ad uno specifico URL, avente il formato seguente:

- *http://<StorageAccount>.blob.core.windows.net/<Container>/<BlobName>*

dove *<StorageAccount>* è un identificativo assegnato nel momento in cui viene sottoscritto l'account, *<Container>* e *<BlobName>* sono il nome del container e il nome del blob contenuto al suo interno.

Windows Azure fornisce due differenti tipi di blob.

- Block blob

  Un blob di questo tipo può contenere dati fino a 200 GB di grandezza. Come suggerisce il nome, un block blob è suddiviso in un certo numero di blocchi. Se durante il trasferimento di un block blob sorgessero problemi, la ritrasmissione può essere ripresa dal blocco successivo all'ultimo inviato correttamente, evitando la ritrasmissione dell'intero contenuto.

- Page blob

  La dimensione massima consentita per questa tipologia arriva fino ad un massimo di un terabyte di dimensione per ogni blob. Un page blob è progettato per accessi casuali, ed è diviso in un certo numero di pagine. In questo modo un'applicazione può leggere una pagina e scriverne o modificarne una seconda in parallelo.

Per evitare la perdita dei dati a fronte di un guasto hardware e per aumentarne la disponibilità, ogni blob viene replicato su tre computer in un data center di Windows Azure. Infatti, nel momento dell'aggiunta di un nuovo blob, l'operazione viene effettuata in parallelo sulle tre macchine. In questo modo le letture successive non ritornano risultati inconsistenti.

Inoltre, per aumentare ulteriormente la tolleranza ai guasti, Windows Azure mette a disposizione l'opzione di Geo-replicazione. Tale opzione permette di specificare un secondo data center in cui replicare i dati contenuti nello Storage Account. Anche in questo caso i dati vengono salvati su tre macchine differenti, in modo che il singolo data center possa recuperare dai guasti in modo indipendente. Il data center acceduto dall'utente è quello primario, le modifiche apportate a tale data center vengono propagate in pochi minuti verso il data center secondario.

Come precedentemente affermato, un blob deve risiedere in un container, quindi prima di poter caricare un blob, è necessaria la creazione del container che conterrà tale blob. Il container oltre ai blob, può contenere una serie di metadati assegnati al momento della creazione del container. Nel Listato 3.15, viene riportato un esempio di creazione di un container, senza metadati, con nome *Container1*.

Listato 3.15: Creazione di un container

```
1 AzureBlobManager abm = new AzureBlobManager(new DirectConnectToken(account,
      key));
2 abm.createContainer("Container1", null, ContainerAccess.BLOB);
```

Il *ContainerAccess* specifica i diritti di accesso a un container e può assumere tre possibili valori:

- PRIVATE - non è possibile accedere pubblicamente al contenuto del container, ma solo attraverso una chiamata autenticata
- BLOB - accesso pubblico ai blob, tramite richieste non autenticate. Altro contenuto del container accessibile solo con autenticazione
- CONTAINER - tutto il contenuto del container è accessibile tramite richieste non autenticate

Una volta creato il container, è possibile caricare al suo interno un blob. Nel Listato 3.16 viene riportato un esempio di caricamento di un blob, di tipo block, nel container creato nell'esempio precedente.

Listato 3.16: Caricamento di un blob

```
1  File file= new File("file1");
2  InputStream is = new FileInputStream(file);
3
4  byte[] buffer = IOUtils.toByteArray(is);
5
6  AzureBlobManager abm = new AzureBlobManager(new DirectConnectToken(account,
       key));
7  PutBlobCondition pbc = new PutBlobCondition();
8
9  // Blob's metadata
10 MetadataCollection metadata = new MetadataCollection();
11 String contentType = MimetypesFileTypeMap.getDefaultFileTypeMap().getContentType(
       file.getName());
12 metadata.add(Metadata.buildValid("fileName", file.getName()));
13 metadata.add(Metadata.buildValid("size", Integer.toString(buffer.length)));
14 metadata.add(Metadata.buildValid("contentType", contentType));
15
16 BlobData blobdata = new OctetBlobData(buffer);
17 abm.putBlockBlob("Container1", file.getName(), null, metadata, blobdata, pbc);
```

Il metodo, riportato nella riga 17 del Listato 3.16, riceve come parametro il nome del container in cui caricare il blob, il nome da assegnare al blob, i metadati di contorno, come ad esempio la dimensione del blob e il tipo di contenuto del blob, e il contenuto, sottoforma di array di byte, del file incapsulato in un oggetto chiamato OctetBlobData.

Per poter accedere e scaricare un blob precedentemente caricato, come esemplificato nel Listato 3.17, occorre invocare il metodo *getBlockBlob*, che riceve come parametro il nome del container e il nome del file da scaricare. Tale metodo ritorna un oggetto rappresentate il blob. Per poter ottenere il contenuto del blob, sottoforma di array di byte, occorre richiamare il metodo statico presente nella classe OctetBlobData e passare l'oggetto ritornato dalla chiamata precedente.

Listato 3.17: Download di un blob

```
1 AzureBlobManager abm = new AzureBlobManager(new DirectConnectToken(account,
      key));
2 AzureBlob blob = abm.getBlockBlob("Container1","File1");
3 byte[] blobContent = OctetBlobData.fromBlob(ab).getBytes()
```

Oltre a aggiungere e scaricare blob, la libreria mette a disposizione il metodo per poter cancellare un blob oppure cancellare l'intero container. Un esempio di cancellazione è riportato nel Listato 3.18.

Listato 3.18: Cancellazione di un blob/container

```
1 AzureBlobManager abm = new AzureBlobManager(new DirectConnectToken(account,
      key));
2
3 // Cancellazione di un blob
4 abm.deleteBlob("Container1", "File1", null, new DeleteBlobCondition());
5
6 // Cancellazione di un container
7 abm.deleteContainer("Container1", new DeleteContainerCondition())
```

L'AzureBlobManager espone altri metodi per poter gestire il servizio di Blob offerto da Windows Azure. La Figura 3.4 riporta un semplice riepilogo delle API disponibili.

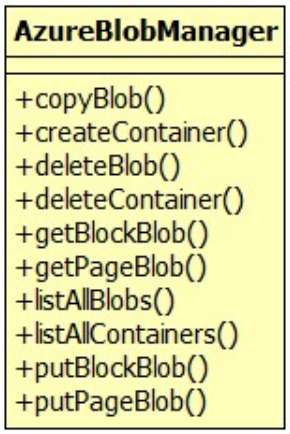


Figura 3.4: API AzureBlobManager contenuto nella libreria jpa4azure

Table Service

Windows Azure Table Storage è un servizio NoSQL, che permette alle applicazioni di accedere rapidamente e in modo semplice a grandi quantità di dati aventi strutture differenti tra loro. Il Table Storage fornisce una memorizzazione sottoforma di chiave-valore, associando un insieme di dati, rappresentanti una struttura, ad una chiave particolare, utilizzata per il recupero della struttura salvata.

Come per i blob, ogni tabella è associata ad uno storage account di Windows Azure. Per quanto riguarda l'accesso ad una tabella, è necessario effettuare una chiamata HTTP ad uno specifico URL, avente il formato seguente:

- *http://<StorageAccount>.table.core.windows.net/<TableName>*

Ogni tabella viene divisa in un numero arbitrario di partizioni, ognuna delle quali può essere memorizzata su una macchina separata (una forma di *sharding*, come avviene con SQL Federation). I dati sono accessibili, sia da applicazioni residenti sulla piattaforma di Windows Azure, sia da applicazioni situate all'esterno, utilizzando chiamate REST autenticate. Ogni partizione di una tabella contiene un numero arbitrario di entità, le quali a loro volta contengono un massimo di 255 proprietà (rappresentanti i campi dell'oggetto). Tutte le proprietà possiedono un proprio nome, un tipo e un valore. I tipi di proprietà supportati dal servizio sono: Binary, Bool, DateTime, Int e String.

Al contrario di uno storage relazionale, le tabelle non hanno alcuno schema fisso, ossia differenti entità della stessa tabella possono contenere proprietà aventi tipi diversi. Ad esempio, un'entità potrebbe avere una stringa come proprietà, mentre un'altra entità della stessa tabella potrebbe contenere due proprietà di tipo Int.

Questa struttura permette alle tabelle di diventare enormi (una singola tabella può contenere fino ad un massimo di 100 terabyte di dati) e permette un accesso veloce al loro contenuto.

Ovviamente data la particolarità della struttura, si vengono ad introdurre delle limitazioni rispetto alla struttura relazionale. Un'esempio di limitazione consiste nel non supportare gli aggiornamenti transazionali su più tabelle o addirittura su partizioni di un'unica tabella. Un insieme di aggiornamenti, riferiti ad un'unica tabella, possono essere raggruppati in una transazione atomica solo se tutte le entità coinvolte appartengono ad un'unica partizione.

Inoltre non c'è modo di effetture operazioni di join tra tabelle. A differenza dei database relazionali, le tabelle non hanno alcun supporto per le stored procedure.

Come precisato precedentemente, per implementare tale servizio nella libreria CPIM, si è utilizzata la libreria *jpa4azure*. Di seguito sono esemplificati i passaggi fondamentali per l'utilizzo del servizio tramite tale libreria.

1. Creazione del file *persistence.xml* contentente i dati necessari per l'accesso al servizio. Tale file deve essere situato all'interno della cartella *META −INF*.

Listato 3.19: Esempio di persistence.xml

```
<persistence>
   <persistence-unit name="myazure" transaction-type="RESOURCE_LOCAL"
    >
      <properties>
         <property name="storage.emulator" value="false" />
         <property name="account.name" value="your_account" />
         <property name="account.key" value="your_key" />
      </properties>
   </persistence-unit>
</persistence>
```

2. Creazione di una classe rappresentante l'entità da salvare. Tale oggetto deve possedere l'annotazione *@Entity*. Per quanto riguarda i campi di questa classe, deve essere presente un campo annotato con *@Id*. Inoltre se si necessita di inserire un campo, il cui tipo non figura tra quelli supportati, ma che comunque risulta serializzabile, allora tale campo può essere specificato ed annotato con *@Embedded*.

Listato 3.20: Esempio di entità

```
@Entity(name = "Topic")
public class Topic {
   @Id
   private String topicName;

   @Embedded
   private ArrayList<String> topicQuestions;

```

```
9     public Topic() {}
10
11    public Topic(String topicName, ArrayList<String> topicQuestions) {
12        super();
13        this.topicName = topicName;
14        this.topicQuestions = topicQuestions;
15    }
16
17    // getter & setter
18
19 }
```

3. Interfacciamento con il servizio e salvataggio di un'entità.

Listato 3.21: Esempio salvataggio di un'entità

```
1  try{
2     AzureEntityManagerFactory factory=
3        (AzureEntityManagerFactory)Persistence.createEntityManagerFactory("
       myazure", null);
4     AzureEntityManager em=(AzureEntityManager) factory.createEntityManager();
5
6     ArrayList<String> reading=new ArrayList<String>();
7     reading.add("How much do you love reading?");
8     reading.add("How much do you appreciate the use of ebooks as a means of
       reading?");
9     reading.add("How much do you love novels?");
10    reading.add("How much do you love yellow books?");
11    reading.add("How much do you love classics books?");
12    reading.add("How much do you love fantasy books?");
13    reading.add("How much do you love historical Books?");
14    reading.add("How much do you love poems?");
15
16    em.persist(new Topic("Reading", reading));
17 } catch (Exception e){
18 } finally{
19    em.close();
20 }
```

Caching Service

Windows Azure Caching è un servizio che offre la possibilità di allocare una certa quantità di dati direttamente nella memoria RAM riservata alla macchina virtuale in cui gira l'istanza. Il meccanismo di caching viene spesso impiegato per tenere dati frequentemente richiesti in memoria, in modo da ridurne la latenza.

Questo servizio permette inoltre di ridurre i costi e aumentare la scalabilità degli altri servizi di memorizzazione, come SQL Database e il servizio NoSQL.

Esistono due tipologie di distribuzione per il servizio di Caching: dedicato e co-locato. Nello scenario dedicato, si definisce una worker role dedicata a tale servizio. Ciò significa che tutta la memoria dell'istanza della worker role è usata per il Caching, esclusa la memoria utilizzata dal sistema operativo. In una tipologia co-locata invece, solo una percentuale della memoria disponibile sulle role viene dedicata al servizio di Caching. Per esempio, è possibile assegnare il 20% della memoria per il Caching su ogni istanza di una role su cui è caricata la parte di frontend di un'applicazione.

I dati contenuti nella cache sono organizzati secondo la coppia chiave-valore, costituita da oggetti serializzabili.

L'interazione con tale servizio, tramite API dirette, è supportata solo per l'ambiente .NET.

Per quanto riguarda l'ambiente di Java, l'unico modo possibile con cui le applicazioni possono interagire con il servizio è attraverso l'utilizzo di librerie di terze parti, che implementano il protocollo Memcache. Infatti tale protocollo viene supportato dal servizio di Caching, facilitando la migrazione di applicazioni memcache-based verso la piattaforma di Windows Azure.

Nel caso della libreria CPIM, si utilizza il client Memcache fornito dalla libreria open source *spymemcached-2.8.4*.

L'utilizzo del client Memcache è permesso importando il JAR contenente la libreria precedentemente citata nel classpath dell'applicazione.

Per utilizzare tale servizio, è necessario configurare la role ospitante. Per far ciò occorre modificare il file *ServiceDefinition.csdef*, specificando le seguenti impostazioni:

1. Importazione del modulo Caching, per l'abilitazione dell'intero servizio

```
<Imports>
   <Import moduleName="Caching"/>
   ...
</Imports>
```

2. Aggiungere uno storage locale, in modo da riservare memoria fisica utilizzata per memorizzare informazioni non inerenti alla cache, ma relativi al servizio, come ad esempio i log

```
<LocalResources>
   <LocalStorage cleanOnRoleRecycle="false" name="Microsoft.WindowsAzure.
    Plugins.Caching.FileStore" sizeInMB="1000"/>
</LocalResources>
```

3. Aggiungere un endpoint, utilizzato dal client per accedere alla cache attraverso il protocollo Memcache (specificato come InternalEndpoint in modo da non renderlo accessibile dall'esterno dell'applicazione)

```
<Endpoints>
   <InternalEndpoint name="memcache_default" port="11211" protocol="tcp"/>
   ...
</Endpoints>
```

Oltre al *ServiceDefinition.csdef* file, bisogna specificare nel file *ServiceConfiguration.cscfg* le configurazioni del servizio di Caching, come riportato nel Listato 3.22, rappresentanti:

- *NamedCaches* (riga 1), permette di creare cache multiple ognuna con le proprie configurazioni come: il tempo di vita, le politiche di cancellazione, etc
- *Loglevel* (riga 2), usato per specificare il livello di informazioni da registrare per scopi diagnostici
- *CacheSizePercentage* (riga 3), rappresenta la percentuale di memoria della role da allocare per la cache (in caso di role dedicata il valore non deve essere specificato)
- *ConfigStoreConnectionString* (riga 4), specifica lo storage account

Listato 3.22: Configurazioni Caching

```
1 <Setting name="Microsoft.WindowsAzure.Plugins.Caching.NamedCaches" value="" />
2 <Setting name="Microsoft.WindowsAzure.Plugins.Caching.Loglevel" value="" />
3 <Setting name="Microsoft.WindowsAzure.Plugins.Caching.CacheSizePercentage" value=
      "30" />
4 <Setting name="Microsoft.WindowsAzure.Plugins.Caching.ConfigStoreConnectionString"
       value="DefaultEndpointsProtocol=https;AccountName=
      YOURSTORAGEACCOUNT;AccountKey=YOURSTORAGEACCOUNTKEY" />
```

Con il client offerto dalla libreria, è possibile effettuare molte operazioni. Per semplicità verranno riportati e spiegati i metodi principali che implementano le funzionalità più importanti del servizio di Caching.

La connessione al servizio avviene nel momento dell'istanziazione della classe MemcachedClient, che riceve come parametro di ingresso l'InetSocketAddress dell'host. Il pattern dell'host generalmente utilizzato da Windows Azure per connettersi al servizio è così formato: *localhost_<roleName>:11211*.

Una volta connessi, è possibile utilizzare le funzionalità offerte dal servizio. Il Listato 3.23 esemplifica la connessione al servizio (riga 2) e le funzionalità di aggiunta (riga 7), recupero (riga 10) , modifica (riga 13-14) e cancellazione di un oggetto (riga 17) o di tutti gli oggetti presenti (riga 20) sulla Memcache.

Listato 3.23: Funzionalità principali del servizio Memcache

```
1  // Connessione alla memcache
2  MemcachedClient memcache = new MemcachedClient(AddrUtil.getAddresses("
       localhost_WorkerRole1:11211"));
3  String key= "Chiave1";
4  Integer value = new Integer(100);
5
6  // Aggiunge una nuova coppia e la mantiene per un'ora
7  memcache.add(key, 3600, value);
8
9  // Legge il valore riferito alla chiave passata
10 Object getvalue = memcache.get(key);
11
12 // Sovrascrive il valore
13 Integer newValue = new Integer(200);
14 memCache.replace(key,3600,newValue);
15
16 // Cancella la coppia riferita alla chiave passata
17 memcache.delete(key);
```

```
18
19 // Svuota l'intera memcache
20 memcache.flush();
```

Tra le API messe a disposizione dalla classe MemcachedClient, non vi è alcun metodo che implementi la funzionalità di verifica della presenza di una determinata chiave nella memcache. Tale funzionalità deve essere gestita tramite l'utilizzo del metodo *get(key)*. Infatti tale metodo ritorna l'oggetto associato alla chiave se quest'ultima esiste, altrimenti ritorna il valore *null*.

Una possibile implementazione del metodo *contains* è proposta nel Listato 3.24.

Listato 3.24: Metodo contains

```
1 public boolean contains(MemcachedClient memcache, String key){
2    return memcache.get(key)!=null;
3 }
```

SQL Database

Windows Azure SQL Database fornisce un DBMS relazionale per Windows Azure, ed è basato sulla tecnologia SQL Server. Con un'istanza di SQL Database è possibile fornire e caricare sul Cloud un database relazionale, ed usufruire dei vantaggi di un data center distribuito, tra cui disponibilità, scalabilità, e sicurezza.

Grazie al fatto che Windows Azure SQL Database è costruito su tecnologie SQL Server, l'accesso al servizio cloud attraverso Java è molto simile all'accesso da Java ad un SQL Server. In questo modo è inoltre possibile sviluppare un'applicazione localmente (usando SQL Server) e, nel momento del deploy sul cloud, cambiare solamente la stringa di connessione. Tale servizio utilizza, per interfacciarsi, un SQL Server JDBC driver.

Con SQL Database è possibile controllare i dati e assegnare i permessi di accesso, inoltre, dato che si tratta di un servizio Cloud, la parte amministrativa, come ad esempio la gestione dell'infrastruttura hardware, e il mantenimento del database e del software di sistema sempre aggiornato, viene automaticamente gestita dalla piattaforma.

SQL Database fornisce anche una *federation option* [36] che permette la distribuzione dei dati su server multipli. Questa funzionalità è molto utile

per applicazioni che lavorano con grandi quantità di dati oppure, hanno la necessità di diffondere le richieste di accesso ai dati su più server, in modo da migliorare le prestazioni.

Le federazioni costituiscono un modo per realizzare scalabilità e prestazioni maggiori a livello di database dell'applicazione tramite il partizionamento orizzontale. Una o più tabelle all'interno di un database vengono suddivise in base alla riga e divise tra più database (membri della federazione). Questo tipo di partizionamento orizzontale è definito *sharding*. Tale processo viene effettuato dinamicamente senza tempo di inattività. Le applicazioni client possono continuare ad accedere ai dati durante le operazioni di ridistribuzione senza interruzione del servizio. Tale servizio è disponibile anche per applicazioni residenti al di fuori della piattaforma.

Per poter integrare questo servizio in un'applicazione Java è necessario importare nel classpath il JAR relativo a Microsoft JDBC Driver 4.0 per SQL Server. Una volta importato il JAR, i passi usuali da compiere, per interagire con il servizio, sono i seguenti:

1. Connettersi al server SQL Database, riportato nel Listato 3.25

Listato 3.25: Connessione al servizio SQL Database

```
1  import java.sql.*;
2  import com.microsoft.sqlserver.jdbc.*;
3
4  public class HelloSQLAzure {
5
6      public static void main(String[] args)
7      {
8          String connectionString =
9                  "jdbc:sqlserver://your_server.database.windows.net:1433" + ";" +
10                 "database=master" + ";" +
11                 "user=your_user@your_server" + ";" +
12                 "password=your_password";
13         Connection connection=null;
14         try
15             {
16                 // Assicura che il driver sia raggiungibile
17                 Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
18
19                 // Stabilire la connessione
```

```
20              connection = DriverManager.getConnection(connectionString);
21
22              // Creazione Statement
23              ... Listato 2.22 ...
24
25              // Esecuzione Statement
26              ... Listato 2.23 ...
27
28          }
29          catch (Exception e)
30          {
31              System.out.println("Exception " + e.getMessage());
32              e.printStackTrace();
33          }
34          // Chiusura connessioni
35          ... Listato 2.24 ...
36      }
37  }
```

2. Definire un'istruzione SQL, per esempio per creare o cancellare una tabella, inserire/selezionare/eliminare righe, etc. Come riportato nel Listato 3.26

Listato 3.26: Esempi di istruzioni SQL

```
1               // SQL per la creazione di una tabella
2               String sqlCreateT =
3                   "CREATE TABLE Person (" +
4                       "[PersonID] [int] IDENTITY(1,1) NOT NULL," +
5                       "[LastName] [nvarchar](50) NOT NULL," +
6                       "[FirstName] [nvarchar](50) NOT NULL)";
7
8               // SQL per la selezione delle prime 10 righe
9               String sqlSelectT = "SELECT TOP 10 * FROM Person";
```

3. Eseguire l'istruzione SQL, attraverso la chiamata del metodo executeUpdate o executeQuery a seconda del fatto che l'istruzione prevedere il ritorno di dati oppure no. Come esemplificato nel Listato 3.27

Listato 3.27: Esecuzione dell'istruzione SQL

```
1  // Preparazione dello statement
2  Statement statement = connection.createStatement();
3
4  // Esecuzione query senza alcun risultato di ritorno
5  statement.executeUpdate(sqlCreateT);
6
7  // Esecuzione query con risultati di ritorno
8  ResultSet resultSet = statement.executeQuery(sqlSelectT);
```

4. Chiudere le connessioni, riportato nel Listato 3.28

Listato 3.28: Chiusura della connessione

```
1  finally
2  {
3     try
4     {
5        // Chiusurea delle risorse
6        if (null != connection) connection.close();
7        if (null != statement) statement.close();
8        if (null != resultSet) resultSet.close();
9     }
10    catch (SQLException sqlException)
11    {
12       // Se fallisce vuol dire che le risorse sono chiuse
13    }
14 }
```

Queue Service

Windows Azure Queue è un servizio per memorizzare grandi numeri di messaggi che possono essere acceduti da qualsiasi applicazione web attraverso chiamate HTTP/HTTPS autenticate.

Una coda consiste in un'unità di archiviazione, simile ad un database, in cui le uniche operazioni consentite sono l'aggiunta, la lettura, e la cancellazione di un messaggio.

La lunghezza massima di un messaggio consentita è di 64KB. Nel caso si volesse mandare un messaggio più lungo, la tecnica comunemente utilizzata consiste nel memorizzare il contenuto del messaggio in un blob, attraverso il

Blob Service, e inserire nella coda un messaggio contenente la referenza a tale blob.

Una coda può contenere milioni di messaggi fino alla saturazione massima consentita dello storage account, che ammonta a 100 TB di grandezza.

Il Queue Service viene offerto per far comunicare le role tra loro, in modo da poterle rendere scalabili. In particolare le role non devono e non possono sapere l'indirizzo esatto della role con cui vogliono comunicare. Il meccanismo di code permette di richiamare asincronamente servizi offerti da altre role che, in ascolto su una determinata coda, leggono il messaggio e ne eseguono la richiesta. Inoltre, a causa della scalabilità dell'ambiente cloud, ogni role potrebbe eseguire più di un'istanza, ed in questo caso la coda risulterebbe utile come dispatcher dei messaggi per le diverse istanze della role.

Anche per il Queue Service, la libreria utilizzata all'interno della libreria CPIM per interagire con il servizio è contenuta nel JAR *jpa4azure*.

Tale libreria espone i metodi utilizzati per comunicare con il servizio ed usufruire delle sue funzionalità. Nel Listato 3.29 e nel Listato 3.30 sono esemplificati, rispettivamente, un consumer e un producer, che utilizzano la coda come mezzo di comunicazione. Il consumer crea la coda e cicla su di essa in attesa dell'arrivo di un messaggio, e nel caso arrivi un messaggio sulla coda, ne stampa il contenuto a video, poi lo cancella. Mentre il producer inserisce semplicemente i messaggi in coda.

Un caso particolare è la cancellazione di un messaggio da parte del consumer. Per poter effettuare tale operazione, bisogna essere in possesso del messaggio da cancellare, ed essere l'ultimo possessore, ossia l'ultimo processo che lo ha letto. Infatti il messaggio contiene tra le altre informazioni una stringa chiamata PopReceipt. La cancellazione va a buon fine se il processo chiamante è in possesso del messaggio con l'ultimo PopReceipt generato. Tale stringa infatti viene rigenerata ogni volta che il messaggio ritorna visibile nella coda, ossia quando scade il tempo di riservatezza di un messaggio.

Listato 3.29: Consumer

```
1 public class Consumer extends Thread{
2     public void run(){
3         AzureQueueManager aqm = new AzureQueueManager(new DirectConnectToken(
        account,key));
4
```

```
5         // Creazione coda "codauno"
6         aqm.createQueue("codauno");
7
8         while(true){
9             // Recupero di un messaggio dalla coda
10            // Il messaggio rimane riservato per 10 secondi, poi ritorna disponibile nella
      coda
11            AzureQueueMessageCollection collection = aqm.getMessages("codauno", 1, 10);
12            for(AzureQueueMessage m : collection){
13                System.out.println("Hello!! I read the message with text " + m);
14                // Cancellazione di un messaggio
15                aqm.deleteMessage("codauno", message.getMessageId(), message.
      getPopReceipt());
16            }
17        }
18    }
19 }
```

Listato 3.30: Producer

```
1  public class Producer{
2      public void addMessage(String text){
3          AzureQueueManager aqm = new AzureQueueManager(new DirectConnectToken(
       account,key));
4
5          // Inserimento di un messaggio nella coda con visibilità  di 60 secondi.
6          aqm.putMessage("codauno", "messaggio", 60);
7      }
8  }
```

SendGrid Email Service

Windows Azure non metta a disposizione un servizio Mail per Java interno, ma si affida ad un servizio esterno denominato SendGrid [22].

SendGrid è un servizio email cloud-based che fornisce consegne email affidabili, scalabili e analisi real-time, grazie ad API flessibili che permettono una semplice integrazione personalizzata.

Per poter utilizzare tale servizio in Java occorre semplicemente importare nella propria applicazione una qualsiasi libreria implementante javax.mail, per esempio da `http://www.oracle.com/technetwork/java/javamail`.

Ad alto livello, i passi necessari per utilizzare la libreria javax.mail per inviare mail usando un server SMTP sono i seguenti:

1. Specificare i parametri SMTP, tra cui il server, che per il servizio SendGrid è `smtp.sendgrid.net`, come mostrato nel Listato 3.31.

Listato 3.31: SendGrid Mail Service

```
1  public class MyEmailer {
2    private static final String SMTP_HOST_NAME = "smtp.sendgrid.net";
3    private static final String SMTP_AUTH_USER = "your_sendgrid_username";
4    private static final String SMTP_AUTH_PWD = "your_sendgrid_password";
5
6    public static void main(String[] args) throws Exception{
7      new MyEmailer().SendMail();
8    }
9
10   public void SendMail() throws Exception
11   {
12     Properties properties = new Properties();
13     properties.put("mail.transport.protocol", "smtp");
14     properties.put("mail.smtp.host", SMTP_HOST_NAME);
15     properties.put("mail.smtp.port", 587);
16     properties.put("mail.smtp.auth", "true");
```

2. Estendere la classe javax.mail.Authenticator, e nell'implementazione del metodo *getPasswordAuthentication*, ritornare lo username e la password dell'account di SendGrid, come nel Listato 3.32.

Listato 3.32: SendGrid Mail Service Authenticator

```
1  private class SMTPAuthenticator extends javax.mail.Authenticator {
2  public PasswordAuthentication getPasswordAuthentication() {
3    String username = SMTP_AUTH_USER;
4    String password = SMTP_AUTH_PWD;
5    return new PasswordAuthentication(username, password);
6  }
```

3. Creare una sessione email autenticata attraverso un oggetto javax.mail.Session, come riportato nel Listato 3.33.

Listato 3.33: SendGrid Mail Service Session

```
1 Authenticator auth = new SMTPAuthenticator();
2 Session mailSession = Session.getDefaultInstance(properties, auth);
```

4. Creare il messaggio e assegnare i valori riguardanti il mittente, il destinatario, il soggetto e il contenuto della mail, che può essere scritto in formato HTML oppure in formato testuale, come esemplificato nel Listato 3.34.

Listato 3.34: Creazione di una mail

```
1  MimeMessage message = new MimeMessage(mailSession);
2  Multipart multipart = new MimeMultipart("alternative");
3  BodyPart part1 = new MimeBodyPart();
4  part1.setText("Hello, Your Contoso order has shipped. Thank you, John");
5  BodyPart part2 = new MimeBodyPart();
6  part2.setContent(
7     "<p>Hello,</p>
8     <p>Your Contoso order has <b>shipped</b>.</p>
9     <p>Thank you,<br>John</br></p>",
10    "text/html");
11 multipart.addBodyPart(part1);
12 multipart.addBodyPart(part2);
13 message.setFrom(new InternetAddress("john@contoso.com"));
14 message.addRecipient(Message.RecipientType.TO,
15   new InternetAddress("someone@example.com"));
16 message.setSubject("Your recent order");
17 message.setContent(multipart);
```

5. Inviare il messaggio attraverso un oggetto javax.mail.Transport, riportato nel Listato 3.35.

Listato 3.35: Invio di una mail

```
1 Transport transport = mailSession.getTransport();
2
3 // Connettere l'oggetto Transport
4 transport.connect();
5
6 // Inviare il messaggio
7 transport.sendMessage(message, message.getRecipients(Message.RecipientType.TO)
      );
```

```
8
9  // Chiudere la connesione.
10 transport.close();
```

Servizi attualmente non supportati

I seguenti servizi, offerti dalla Windows Azure per l'ambiente di sviluppo Java, non sono supportati dall'attuale versione della libreria CPIM.

Access Control

Access Control è un servizio di Windows Azure che fornisce un modo semplice per autenticare gli utenti che accedono alle applicazioni ed ai servizi web, senza implementare alcuna logica di autenticazione attraverso il codice. Tale servizio fornisce un integrazione con la Windows Identity Foundation (WIF), utilizzata da molti siti web richiedenti preventiva autenticazione, come Windows Live ID, Google, Yahoo e Facebook.

Service Bus Queues

Service Bus Queues è un servizio di coda, che permette l'invio di messaggi secondo la logica FIFO (First In First Out) e supporta una serie di protocolli standard (REST,AMQP, WS*) ed API che permettono l'inserimento e la cattura dei messaggi da una coda.

Service Bus Topics

Service Bus Topics è un servizio che fornisce un modello publish/subscribe che supporta la comunicazione una a molti. A questo servizio è possibile assegnare delle regole di filtraggio per uno specifico topic a seconda della sottoscrizione, ossia permette di restringere l'invio dei messaggi ad un topic secondo una particolare sottoscrizione, creando così un meccanismo di sotto-sottoscrizioni.

Twilio Voice and SMS Service

Le applicazioni di Windows Azure possono usare Twilio per incorporare servizi VoIP e servizi SMS al proprio interno. Usando le API di Twilio è possibile

effettuare e ricevere chiamate telefoniche, inviare e ricevere messaggi SMS, ed abilitare la comunicazione vocale tramite connessioni internet esistenti.

3.3 Riepilogo comparativo

Nella Tabella 3.1 vengono riepilogate le varie caratteristiche e i limiti dei servizi offerti dalle due piattaforme e supportati dalla libreria CPIM.

Servizio	Google App Engine	Windows Azure
NoSQL	API dirette Interfaccia JPA Interfaccia JDO	API dirette Interfaccia JPA (terze parti) Limite tipo di dati supportati
SQL	MySQL-like Massimo 10GB per istanza JDBC Driver	SQL Server-like sharding JDBC Driver
Memcache	Sincrona/Asincrona	Dedicata/Co-locata Memcache supportata dal servizio Caching (terze parti)
Blob	Due tipologie di gestione Organizzazione piatta Massimo 32MB grandezza blob	Due tipologie di blob Organizzazione gerarchica Massimo 200GB/1TB per Block/Page blob
Queue	Due tipologie di gestione code Code di task	Code di messaggi Limite 64KB per messaggio
Mail	SMTP interno API dirette supporta java.mail	SMTP esterno supporta java.mail

Tabella 3.1: Riepilogo servizi standardizzati

Capitolo 4

Cloud Platform Independent Model

Come descritto nel capitolo precedente, uno dei maggiori impedimenti della diffusione massiccia del paradigma Cloud Computing, è il cosiddetto effetto ¨Lock-in˝, cioè la difficoltà di cambiare il proprio Cloud provider a costi e tempi ragionevoli.

In questo elaborato di tesi ci si è focalizzati sul problema della portabilità di applicazioni deployate su Cloud, presentando uno strato di astrazione a servizi Cloud, che permette da una parte di utilizzare i servizi offerti dai Cloud provider e dall'altra di rendere la propria applicazione portabile, senza drammatiche modifiche, da una piattaforma ad un'altra.

Questo layer di astrazione, chiamato CPIM (Cloud Platform Independent Model), permette di astrarre l'accesso ai principali servizi offerti dalle piattaforme di due dei PaaS attualmente più utilizzati e popolari: App Engine di Google e Azure di Microsoft.

In questo capitolo viene descritto nella Sezione 4.1 l'approccio scelto per dare un contributo al risolvimento del problema della portabilità in ambiente PaaS. Nella Sezione 4.2 vengono descritte le fasi di sviluppo di un'applicazione che utilizza la libreria CPIM. Successivamente, nella Sezione 4.3 sono descritte dettagliatamente le scelte di design e implementative degli strati di astrazione relativi ai servizi Cloud supportati dall'attuale versione della libreria.

4.1 Approccio utilizzato

L'approccio utilizzato è quello della creazione di una libreria, che permetta di esporre lato client delle API ˜Cloud-vendor independent˝ per l'accesso a servizi Cloud di un provider generico.
A runtime la libreria utilizza le informazioni contenute in file di configurazione per rimappare queste API nelle API proprietarie della piattaforma Cloud sul quale si è deciso di deployare l'applicazione.

Relativamente a lavori simili presenti in letteratura, la libreria CPIM si differenzia per i seguenti motivi:

- Permette la portabilità di applicazioni web scritte in Java tra le piattaforme di App Engine e Azure
- Supporta non solo servizi relativi allo storage (database relazionali, NoSQL) e code di messaggi (tipiche di Azure), ma anche code di Task (tipiche di Google App Engine) e i servizi di Memcache e di Mailing

Il linguaggio scelto per la sua implementazione è Java.
Una rappresentazione qualitativa dell'approccio è presentata in Figura 4.1.

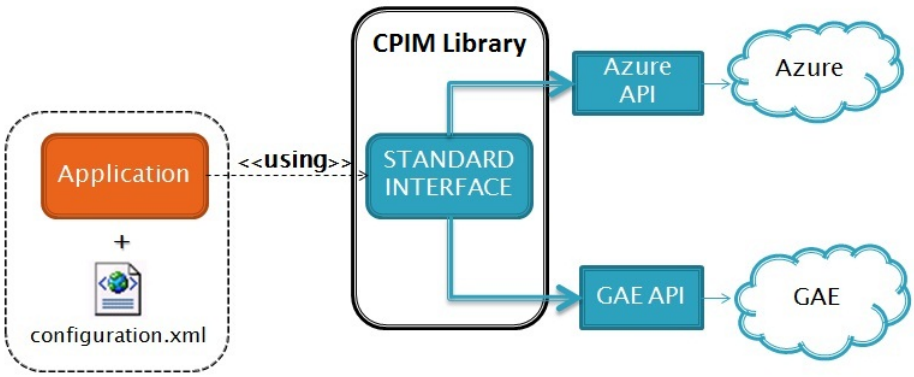


Figura 4.1: Approccio astrazione utilizzato nella libreria CPIM

Tutte le richieste ai servizi supportati dalla libreria CPIM, sono intermediate da una particolare classe, denominata MF (ManagerFactory).

Questa entità ha i compiti di leggere i metadati contenuti nel file di configurazione, necessari a decidere, a runtime, su quali API dovranno essere rimappate le chiamate ai servizi, e di esporre i metodi necessari all'istanziazione di tutti gli oggetti factory dei servizi messi a disposizione dalla libreria.

Oltre ai metodi sopra citati, espone il metodo statico *getFactory()*, attraverso il quale viene implementato il pattern singleton per questa classe.

Come rappresentato in Figura 4.2, durante la prima invocazione di questo metodo, nel quale si istanzia effettivamente il singleton della classe, avviene la creazione di un'altra classe fondamentale e trasversalmente utilizzata da tutti i servizi: la classe *CloudMetadata* (Class Diagram in Figura 4.3).

Tale classe effettua il parsing dei file di configurazione, presentati in dettaglio nell'Appendice A, memorizzando al suo interno tutti i valori letti, che saranno utilizzati dalla libreria nelle varie invocazioni ai servizi supportati. Anche la classe *CloudMetadata* utilizza il pattern singleton, così da non ripetere questa onerosa operazione di parsing ad ogni sua istanziazione.

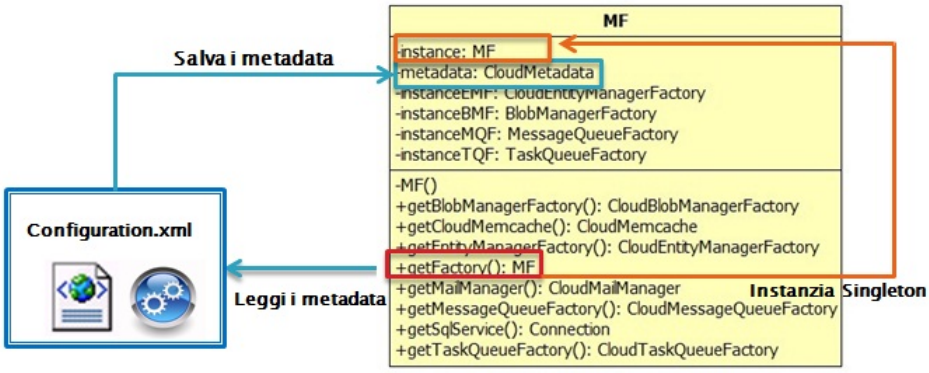


Figura 4.2: Operazioni effettuate alla prima invocazione del metodo getFactory

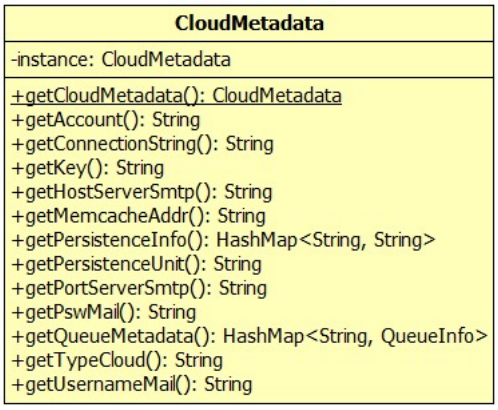


Figura 4.3: Classe CloudMetadata

Sicuramente il dato più importante, indispensabile, contenuto in questa classe, è reperibile tramite il metodo *getTypeCloud()*, che ritorna l'identificativo del Cloud provider su cui si è deciso di deployare l'applicazione. Questo dato è utilizzato dalla libreria per decidere su quale servizio di quale vendor, rimappare le chiamate "Vendor-independent", effettuate tramite le API esposte dalla libreria.

La tecnica di design utilizzata per l'implementazione di queste operazioni fa uso del pattern AbstractFactory [37]. Un esempio è raffigurato in Figura 4.4, nel quale è richiesta l'istanziazione di una *CloudEntityManagerFactory*, interfaccia relativa all'AbstarctFactory del servizio NoSQL (descritto in 4.3.1). Come si evince dalla figura, grazie ai metadati letti dal file di configurazione, la libreria ha la capacità di istanziare la factory concreta del provider prescelto.

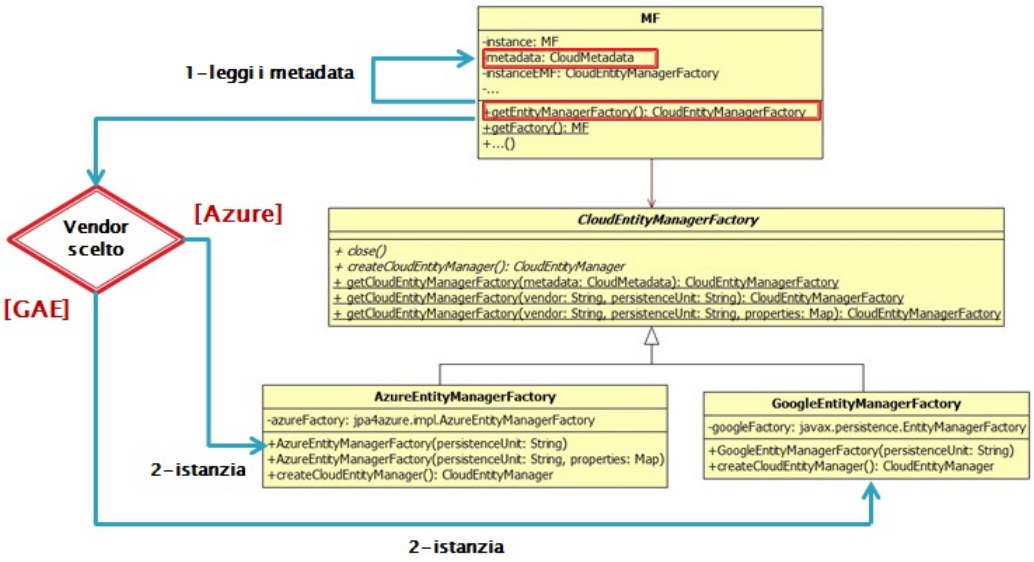


Figura 4.4: Esempio funzione MF

4.2 Ciclo di sviluppo di un'applicazione con CPIM

Uno sviluppatore che desidera scrivere un'applicazione utilizzando la libreria CPIM deve seguire i seguenti passi:

1. Scrittura del codice relativo alla propria applicazione, utilizzando per l'accesso ai servizi Cloud, le API esposte dalla libreria.

2. Compilazione dei file di configurazione, come descritto nell'Appendice A.

3. Creazione di un progetto ad-hoc (contenente le API proprietarie) relativo al vendor scelto, nel quale va inserito il war contenente il codice applicativo e il layer di astrazione CPIM.

Per automatizzare la fase di configurazione e la successiva di importazione del war contenente il codice "Vendor-Independent" in un progetto "Vendor-Specific", si è implementato un Plugin per l'ambiente di sviluppo Eclipse, che permette, attraverso una GUI apposita, di guidare l'utente durante questi passi. Al Plugin è stato dedicato il Capitolo 5.

Utilizzando questa libreria risulta quindi semplificata la portabilità di applicazioni tra le due piattaforme. Infatti non occorre nessuna modifica o reingegnerizzazione del codice, ma solo semplici modifiche a file di configurazioni.

4.3 Design di CPIM

In questa sezione vengono descritti nel dettaglio, le scelte di design e implementative degli strati di astrazione di tutti i servizi supportati dalla libreria CPIM. L'attuale versione supporta i seguenti servizi Cloud:

- NoSQL Service
- SQL Service
- Queue Service, con due diverse semantiche:
  - Task Queue Service
  - Message Queue Service
- Blob Service
- Mail Service
- Memcache Service

4.3.1 Servizio NoSQL

Questo servizio viene offerto nativamente sia da App Engine, con il suo Datastore service, sia da Azure, con il servizio Table Service.

Per poter standardizzare l'accesso a questi due servizi, si è deciso di utilizzare l'approccio di accesso attraverso l'interfaccia JPA. L'utilizzo di questa interfaccia permette una più agevole astrazione del servizio essenzialmente per i seguenti motivi:

- La JPA è un'interfaccia standard
- Crea uno strato di astrazione sulle chiamate di basso livello del servizio di memorizzazione

Per quanto riguarda App Engine, l'interfaccia e la relativa implementazione, sono fornite direttamente dalla SDK per Java di Google App Engine. Invece, per quanto riguarda Azure, non viene offerta da Microsoft nessuna implementazione ufficiale per l'accesso al servizio Table Service tramite questa modalità.

Per compensare questa limitazione, è stata utilizzata una libreria open source che, tra le altre cose, implementa l'interfaccia JPA per il Table Service di Azure. L'implementazione dell'interfaccia in questione è contenuta nel JAR denominato *jpa4azure.jar* (versione 0.7).

Questa libreria è stata estesa, implementando alcuni metodi dell'interfaccia JPA che non erano precedentemente supportati. Nello specifico si è fornita l'implementazione dei seguenti metodi della classe EntityManager:

- ***public*** *Query createNamedQuery(String query_name);*
- ***public*** *Query createQuery(String query);*
- ***public*** *<T> TypedQuery<T> createQuery(**final** String query, **final** Class<T> c);*

La decisione di estendere la libreria, implementando questi metodi, è stata presa per uniformare la possibilità di invocare Query scritte in un linguaggio SQL-like, presente nell'implementazione dell'interfaccia di Google.

Per effettuare il parsing della stringa contenente la query, si è utilizzata la libreria open source *zql* [38].

Inoltre si è introdotta la possibilità, attraverso ulteriori modifiche alla libreria, di testare localmente l'utilizzo della JPA sull'emulatore del Table Service di Azure. Infatti ora la libreria è in grado di riconoscere se l'applicativo in cui è stata importata, stia girando su Azure o localmente sull'emulatore, adattando di conseguenza le chiamate al servizio.

Factory del gestore del servizio NoSQL

Dopo la fase di configurazione del servizio, descritta nell'Appendice A.2 oppure nel Capitolo 5 relativo al plugin sviluppato, per poter cominciare ad invocare operazioni sul servizio, occorre creare un'istanza della classe *CloudEntityManagerFactory*, factory dell'oggetto *CloudEntityManager* responsabile dell'interazione con il servizio NoSQL.

Come tutte le factory presenti nella libreria, la *CloudEntityManagerFactory* è istanziabile in due diversi modi: attraverso l'oggetto MF, oppure tramite l'invocazione di un metodo statico presente nella factory, come riportato nel Listato 4.1. Tali chiamate sono del tutto equivalenti.

Listato 4.1: Istanziazione CloudEnityManager

```
1 //tramite MF
2 CloudEntityManagerFactory cemf=MF.getEntityManagerFactory();
3 //tramite metodo statico
4 CloudEntityManagerFactory cemf=CloudEntityManagerFactory.
      getCloudEntityManagerFactory(CloudMetadata.getCloudMetadata());
```

La factory esposta dalla libreria, è un oggetto Wrapper delle implementazioni degli oggetti factory delle due piattaforme.

A seconda della scelta effettuata in fase di configurazione del servizio (file *configuration.xml*), creando un'istanza della classe *CloudEntity ManagerFactory* sarà istanziata all'interno di essa un'istanza della factory nativa del provider scelto. Questa tecnica di ¨wrapping¨ è utilizzata in diverse situazioni nella libreria.

Lo standard JPA prevede la possibilità di istanziare una diversa istanza della classe *EntityManagerFactory* per ogni persistence-unit definita nel file *persistence.xml*.

Una limitazione presente nella libreria CPIM, è la possibilità di definire una sola persistence-unit all'interno dello stesso progetto applicativo. Tale

mancanza è dovuta a delle problematiche presenti in Azure, dove i servizi di Table Service, di Queue e di Blob sono referenziabili tramite un account di storage, avente delle proprie credenziali di accesso. Queste credenziali sono utilizzate all'atto di istanziazione delle factory di questi servizi.

In pratica in Azure è associato un account di storage ad ogni persistence-unit definita nel file *persistence.xml*. Quindi anche per le invocazione dei servizi Queue e Blob si dovrebbe passare tali credenziali o la referenza a quale persistent-unit (storage account) utilizzare.

In App Engine, invece, non esiste nessuna correlazione tra persistence-unit e account per l'accesso a questi servizi di storage, poichè l'autenticazione è unica, per tutti i servizi della piattaforma, ed è quella del profilo Google utilizzata per il caricamento dell'applicativo sulla piattaforma.

Per ovviare a questa problematica, si è deciso di permette la definizione e l'utilizzo di un'unica persistence-unit (storage account in azure), permettendo così di mascherare a livello applicativo l'estrazione e l'utilizzo delle credenziali.

Il parsing del file *persistence.xml* viene effettuato nella prima invocazione di un servizio qualsiasi della piattaforma, e i dati contenuti, comprese le credenziali per Azure, sono salvati nell'oggetto *CloudMetadata* e saranno reperibili e utilizzabili nelle successive richieste.

È per questo motivo che il programmatore non deve passare la persistence-unit quando invoca l'istanziazione di una *CloudEntityManagerFactory*, poichè tale informazione è letta e memorizzata in modo automatico durante tale parsing.

Come riportato in Figura 4.5, la classe *CloudEntityManagerFactory* espone il metodo *createCloudEntityManager()*, utilizzato per istanziare l'oggetto che implementa i metodi esposti dall'interfaccia *CloudEntityManager*, oltre ai metodi statici per richiedere un'istanza della factory.

Gestore del servizio NoSQL

La classe *CloudEntityManager* rappresenta il mezzo di comunicazione con il servizio NoSQL, poichè, tutte le operazioni possibili sulle entità memorizzabili in esso, sono fruibili attraverso tale interfaccia.

La richiesta di istanziazione di un oggetto *CloudEntityManager* avviene tramite il metodo *createCloudEntityManager()* del Wrapper relativo alla factory

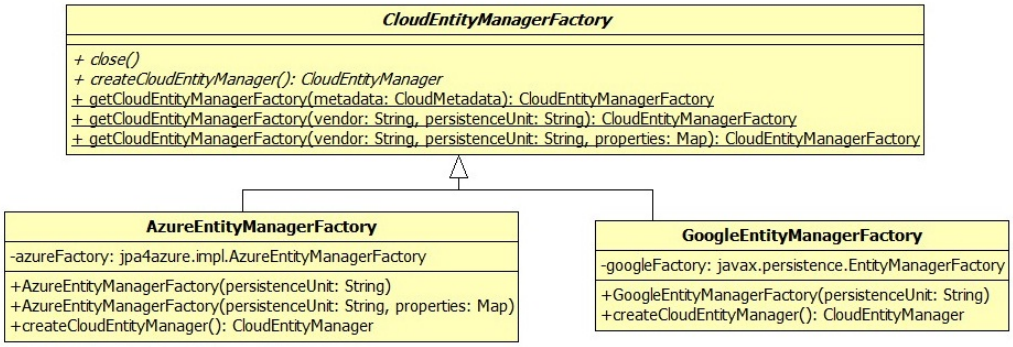


Figura 4.5: Class Diagram classe CloudEntityManagerFactory

dello specifico provider, precedentemente creato. Questa invocazione ritornerà il relativo manager nativo, che dovrà anch'esso essere incapsulato in una classe Wrapper: la classe *CloudEntityManager*.

Tramite questo nuovo intermediario, tutte le chiamate astratte, platform-independent, saranno rimappate, con i relativi aggiustamenti, in chiamate ai metodi del EntityManager del provider. In pratica si è creato uno strato di intermediazione che rimappa chiamate astratte a servizi di una piattaforma generica, in corrispondenti chiamate (attraverso le API proprietarie) ai servizi del provider scelto.

La struttura della classe *CloudEntityManager* è riportata in Figura 4.6.

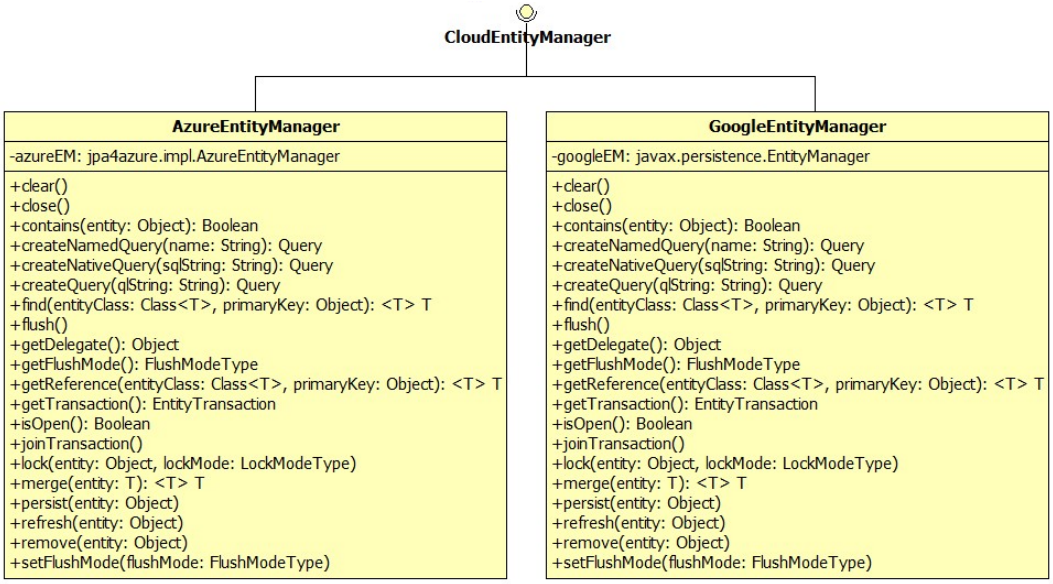


Figura 4.6: Class Diagram classe CloudEntityManager

Ottenuto il mezzo per interagire con il servizio, è possibile iniziare ad effettuarvi operazioni, la più importante delle quali è la memorizzazione di un'entità. Quest'ultima consiste in una classe Java corredata dalle annotazioni dello standard JPA.

Per entrambe le soluzioni Cloud, si utilizzano le annotazioni del package *javax.persistence*.

L'annotazione più importante è *@Entity*. Tale annotazione permette di definire una classe Java come oggetto persistente. Il nome della classe sarà utilizzato come tipologia di oggetto presente nello storage NoSQL.

Un'entità è caratterizzata da una o più proprietà, che non sono altro che gli attributi della classe Java.

Un oggetto persistente per poter essere identificato univocamente, tra gli oggetti della stessa tipologia, deve possedere un attributo annotato con @Id. A causa di incompatibilità tra i due sistemi Cloud, non è possibile utilizzare l'annotazione *@GeneratedValue* per la generazione automatica delle chiavi (Azure non supporta il tipo *Long*, ossia il tipo della chiave creata con tale annotazione).

Le tipologie di attributi rimappabili in proprietà, supportate da entrambe le piattaforme e di conseguenza dalla standardizzazione CPIM sono:

- Byte[]
- Date
- Boolean
- Integer
- String
- Double

Se dovesse rendersi necessario l'inserimento di un attributo di una tipologia differente da quelle supportate, purchè serializzabile, occorrerebbe annotare tale attributo con *@Embedded*. Tale annotazione permette alla JPA di convertire l'attributo in un vettore di byte nel momento della memorizzazione. Viceversa, nel momento in cui viene restituito il valore di un attributo annotato con *@Embedded*, la JPA converte tale vettore nel tipo di oggetto dichiarato nella classe Java relativa all'entità.

Esempio utilizzo API servizio NoSQL

In questo paragrafo viene presentato un esempio completo di utilizzo delle API per questo servizio. Nel Listato 4.2 è mostrato un esempio di definizione di una classe Java, utilizzando le annotazioni dello standard JPA.

Listato 4.2: Esempio di entità JPA

```
1  @Entity
2  public class UserRatings implements Serializable
3  {
4      @Id
5      private String id;
6      private String email;
7      private String topicName;
8      @Embedded
9      private ArrayList<Integer> ratings;
10
11     public UserRatings(String email, String topicName, ArrayList<Integer> ratings)
12     {...}
13
14     public UserRatings() {..}
15     //getter e setter...
16 }
```

Una volta definita l'entità è possibile, attraverso i metodi esposti dalla classe *CloudEntityManager*, memorizzarla, effettuare ricerche per chiave o per attributo, cancellarla, etc.

Nel Listato 4.3 è mostrato un esempio di memorizzazione di un'istanza dell'entità definita nel Listato 4.2.

Listato 4.3: Esempio utilizzo API per memorizzare entità

```
1  CloudEntityManagerFactory cemf=MF.getEntityManagerFactory();
2  CloudEntityManager em=cemf.createCloudEntityManager();
3  ...
4  UserRatings ur = new UserRatings(usermail,actualTopic,ratings);
5  em.persist(ur);
6  ..
7  em.close();
```

4.3.2 Servizio SQL

Entrambe le piattaforme utilizzano le interfacce standard contenute nel pacchetto *java.sql* per interagire con i rispettivi servizi SQL, Google Cloud SQL e SQL Azure.

Il servizio Google Cloud SQL utilizza l'istanza di un server MySQL, mentre SQL Azure utilizza il classico SQL Server di Microsoft.

Il problema della standardizzazione di questi due servizi riguarda il differente DDL (Data Definition Language) utilizzato.

In questo caso la standardizzazione del linguaggio DDL viene lasciata allo sviluppatore, che dovrà definire le tabelle utilizzando una sintassi compatibile per entrambi i linguaggi. Per esempio il Data Type *Enum*, è supportato da MySQL e non da SQL Server di Microsoft, quindi è da evitare per rendere il codice completamente portabile tra le due piattaforme. In [39] vengono elencate le diverse tipologie di campi utilizzabili per i DDL in questione.

Per la configurazione del servizio SQL si veda l'Appendice A.3, oppure se si utilizza il plugin per Eclipse, il Capitolo 5.

Le interazioni con questo servizio vengono gestite nella libreria attraverso la classe astratta *CloudSqlService*. Come riportato nel Listato 4.4, l'istanziazione di questo manager è permessa tramite due modalità: la prima tramite l'oggetto MF, mentre la seconda tramite il metodo statico interno alla classe stessa.

Listato 4.4: Istanziazione del servizio SQL

```
1 //tramite MF
2 CloudSqlService css = MF.getFactory().getSqlService();
3 //tramite metodo statico
4 CloudSqlService css= CloudSqlService.getCloudSqlService(CloudMetadata.
      getCloudMetadata());
5 //tramite il CloudSqlService si ottiene quindi l'oggetto Connection
6 Connection c = sql.getConnection();
```

La classe *CloudSqlService* espone il metodo *getConnection()* per ottenere un'istanza della classe *Connection*, responsabile della gestione della connessione (sessione) con uno specifico database. Tale metodo non richiede come parametro una connection string per connettersi ed autenticarsi alla particolare istanza di un database, poichè è automaticamente letta dal file

configurations.xml. La versione attuale della libreria supporta l'utilizzo di un solo database.

La classe *CloudSqlService* verrà poi adeguatamente estesa, ed il metodo sopra citato implementato, dalle versioni customizzate per i due provider (Class Diagram mostrato in Figura 4.7).

All'interno del metodo *getConnection()*, prima della richiesta di creazione della connessione, avviene la registrazione del driver JDBC utilizzato dal provider scelto.

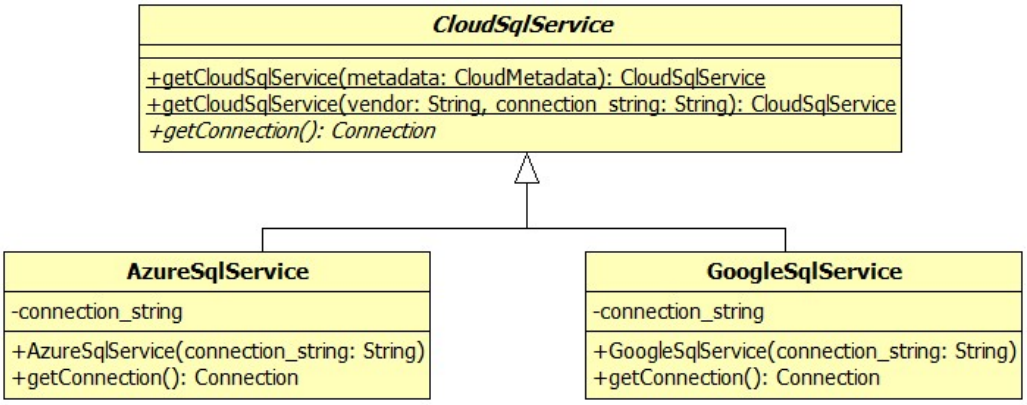


Figura 4.7: CloudSqlService API

Ottenuto l'oggetto *Connection* è possibile creare ed eseguire qualsiasi SQL Statement offerto dalle API standard di Java per l'accesso a questa tipologia di sistemi di storage.

Esempio utilizzo API servizio SQL

Nel Listato 4.5 viene mostrato un semplice esempio, nel quale si richiede la creazione di una tabella.

Listato 4.5: Esempio di utilizzo del servizio SQL

```
1 ...
2 Connection c=mf.getSQLService().getConnection();
3 String stm = "CREATE TABLE UserProfile (Email VARCHAR(255) NOT NULL,
      Password VARCHAR(255) NOT NULL, FirstName VARCHAR(255), LastName
      VARCHAR(255), Date_of_birth DATE, Gender CHAR(1) NOT NULL, Picture
      VARCHAR(255), PRIMARY KEY(Email))";
4 Statement statement;
```

```
5  try {
6          statement = c.createStatement();
7          statement.executeUpdate(stm);
8          } catch (SQLException e) {
9              e.printStackTrace();
10         }
11 ...
```

4.3.3 Servizio Task Queue

Un servizio molto usato nelle applicazioni web è il servizio che mette a disposizione code per effettuare operazioni in background, oppure come mezzo di comunicazione tra parti distinte dell'applicativo.

Come descritto nelle Sezioni 3.1 e 3.2, nei paragrafi riguardanti questo servizio, le piattaforme di Google e di Azure offrono un servizio di code differente:

- Il servizio Task Queue di App Engine utilizza le code per l'esecuzione di Task in background
- Lo scopo del Queue Service di Azure è permettere la comunicazione tra Role, mediante messaggi scambiati attraverso la coda

Quindi con il servizio di code di Azure, si perde il concetto di Task e si utilizza quello di messaggio. Si può definire quindi il servizio come un servizio di Message Queue.

Si è deciso di sfruttare queste differenze semantiche, per offrire allo sviluppatore la possibilità di utilizzare entrambe le tipologie di code, adattando i due servizi in modo da rendere fruibile entrambe le modalità.

Nella prossima Sezione viene descritto come si è adattato il servizio di Message Queue di Azure per ricreare il servizio di Task Queue, nativo di App Engine.

Task Queue service con la libreria CPIM

Con il servizio Task Queue, la coda viene considerata un contenitore di Task da eseguire in modo asincrono rispetto alle richieste web. In ambiente Java,

tale entità è essenzialmente rappresentata da una servlet contenente il codice applicativo da eseguire in background.

Tale servizio viene offerto nativamente da App Engine attraverso il servizio Task Queue di tipo Push, mentre per emulare tale servizio in Azure, viene riadattato il servizio di Message Queue.

La metodologia utilizzata per l'emulazione di un Task, è la creazione di un oggetto ¨Cloud-independent¨, contenente tutte le informazioni necessarie per l'esecuzione di un Task in App Engine. Nella libreria CPIM, tale oggetto è rappresentato dalla classe *CloudTask*, che, come mostrato nel relativo Class Diagram di Figura 4.8, incapsula le seguenti informazioni:

- Metodo HTTP utilizzato per l'invocazione del Task.
  Deve essere settato a POST.
- Parametri passati al Task. Rappresentati da una tipica coppia chiave-valore.
- Path del task (es:/ComputeSomethingServlet)
- Identificativo univoco all'interno della coda del Task.

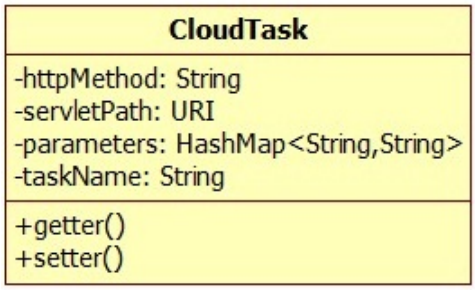


Figura 4.8: Class Diagram CloudTask

Nel caso della piattaforma Google, queste informazioni vengono estratte per popolare l'oggetto *TaskOption*, che verrà poi inserito nella Push Queue di App Engine. Invece per Azure, queste informazioni vengono estratte, adeguatamente formattate in un messaggio testuale, e inserite nella relativa Message Queue.

Un esempio di creazione di un *CloudTask* è mostrato nel Listato 4.7, righe 3-7.

Nella piattaforma di Google, inserito l'oggetto TaskOptions in coda, terminano le operazioni da svolgere lato client, infatti è lo stesso servizio Push Queue a mettere a disposizione un consumer automatico che, a un tasso definito dall'utente, legge la richiesta di esecuzione di un Task dalla coda, lo invoca tramite il path e, in caso di ricezione di una risposta HTTP positiva (entro 10 minuti), cancella definitivamente la richiesta dalla coda.

Per Azure, invece, è stato implementato un Consumer ad-hoc. Questo componente è fornito in un war separato rispetto al package contenente la libreria CPIM. Questa scelta è stata presa per permettere di caricare il war in una Role dedicata, disaccoppiando quindi lo strato di presentazione dell'applicativo da quello di backend. Nello specifico, il componente di backend sarà costituito da:

- war del Consumer
- war incapsulante le Servlet referenziate nei *ClouTask*, cioè le unità lavorative da eseguire

Il war del Consumer, fornito insieme alla libreria CPIM, include essenzialmente:

- Una Servlet: invocata in fase di startup del Java Application Server. Tale servlet istanzia tanti Consumer, quante sono le code di tipo Task definite nel file *queue.xml*
- Classe InternalWorker: Thread che rappresenta il Consumer relativo ad una specifica coda

Questi Thread eseguono, al tasso definito dall'utente, le seguenti operazioni:

- Leggono i messaggi dalla coda
- Creano un "sub-consumer" dedicato per ogni messaggio letto. Questi "sub-consumer" sono i responsabili dell'effettivo espletamento della richiesta di esecuzione del task, svolgendo le seguenti attività:
  - Estrazione delle informazioni contenute nel messaggio
  - Invocazione del task attraverso il path estratto dal messaggio

– Cancellazione del messaggio dalla coda, solo se viene ricevuto esito positivo dall'esecuzione

Il Thread effettua l'esecuzione periodica di queste operazioni, con periodo uguale al rate definito dall'utente, utilizzando il metodo *Sleep*. Nel caso di errata esecuzione del task, le due piattaforme si comportano in modo differente.

In App Engine è disponibile uno schema di riesecuzione automatica di un task in caso di fallimento, oltre alla possibilità di definire un proprio modello attraverso la configurazione di alcuni campi nel file *queue.xml* (si veda per maggiori dettagli `https://developers.google.com/appengine/docs/java/config/queue#Configuring_Retry_Attempts_for_Failed_Tasks`).

In Azure, invece, si è utilizzato il seguente schema all'interno del Consumer: si definisce, per tutti i messaggi inseriti nelle Task Queue (fittizie), il campo Visibility Timeout (che indica il tempo nel quale un messaggio non è più visibile nella coda dopo la sua lettura) a 10 minuti, per uniformare il tempo massimo di esecuzione di un task di App Engine. In caso di fallimento, all'interno di questo arco temporale di 10 minuti, il Consumer dedicato del messaggio, riprova autonomamente l'esecuzione del Task. Per non inondare l'applicativo di richieste di riescuzione, si è deciso di introdurre tra ogni richiesta di esecuzione, un intervallo base di 10 secondi, che aumenta linearmente rispetto al numero di tentativi effettuati (10s, 20s, 30s, 40s,...).

In modo analogo a come prescritto da App Engine, è compito del programmatore rendere ciascuna richiesta di esecuzione di un Task idempotente, o attuare delle contromisure per annullare le operazioni effettuate prima del fallimento dell'esecuzione.

Factory del servizio Task Queue

Lato libreria, l'istanziazione di una coda richiede innanzitutto la configurazione di alcuni parametri, utilizzando per tal fine il file *queue.xml*. Per i dettagli riguardanti la configurazione di questo servizio si veda la Sezione A.5, oppure il capitolo relativo al plugin.

Come per gli altri servizi, ci sono due modi differenti per istanziare la factory relativa ad una Task Queue: utilizzando l'MF, oppure tramite il metodo statico presente nella classe *CloudTaskQueueFactory*. Per entrambi i casi si utilizza il pattern singleton. Viene presentato un esempio di

istanziazione nel Listato 4.6 e in Figura 4.9 è mostrato il relativo Class Diagram.

Listato 4.6: Istanziazione CloudTaskQueueFactory

```
1 //tramite MF
2 CloudTaskQueueFactory ctqf=MF.getFactory().getTaskQueueFactory()
3
4 //tramite metodo statico
5 CloudTaskQueueFactory ctqf=CloudTaskQueueFactory.getCloudTaskQueueFactory(
      CloudMetadata.getCloudMetadata())
```

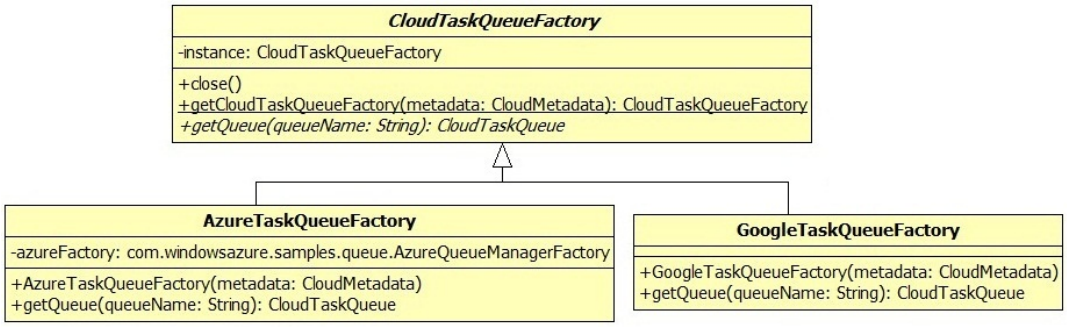


Figura 4.9: CloudTakQueueFactory API

Classe CloudTaskQueue

Dopo aver istanziato la factory, si può procedere con l'istanziazione dell'oggetto *CloudTaskQueue*, il wrapper ¨Cloud-independet˝ utilizzato per comunicare con le code dei due provider. A tal fine si utilizza il metodo *getQueue(" queueName")*, esposto dal factory.

Ottenuto tale oggetto, è possibile invocare i seguenti metodi per l'interazione con la Task Queue:

- *add(CloudTask)*: permette di aggiungere un CloudTask alla coda
- *delete(CloudTask)*: permette la cancellazione di un CloudTask dalla coda
- *purge()*: elimina tutti i CloudTask
- *getQueueName()*: ritorna il nome della coda

Il Class Diagram della classe *CloudTaskQueue* è mostrato in Figura 4.10.

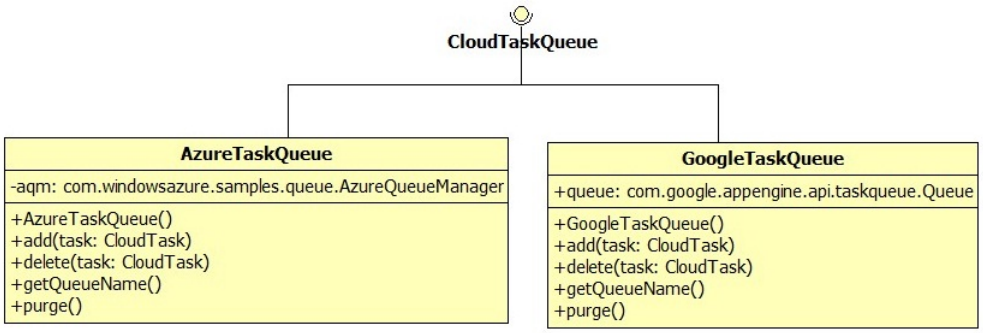


Figura 4.10: CloudTaskQueue API

Esempio utilizzo API servizio Task Queue

Nel Listato 4.7 è mostrato un esempio completo di utilizzo di questo servizio:

- righe 1-2: istanziazione dell'oggetto *CloudTaskQueue*
- righe 3-7: creazione di un *CloudTask*
- riga 9: inserimento, attraverso l'invocazione del metodo *add()* della *CloudTaskQueue*, del *CloudTask* precedentemente creato.

Listato 4.7: Esempio di utilizzo del servizio Task Queue

```
1  CloudTaskQueueFactor ctqf=MF.getFactory().getTaskQueueFactory();
2  CloudTaskQueue myqueue=ctqf.getQueue("mytaskqueue");
3  CloudTask t = new CloudTask();
4  t.setMethod(CloudTask.POST); //imposta il metodo della richiesta http
5  t.setParameters("user", "abcd@def.it"); //i parametri della richiesta
6  t.setServletUri("/servletName"); //la servlet responsabile dell'esecuzione della richiesta
7  t.setTaskName("taskname");
8  try {
9      myqueue.add(t);
10 } catch (CloudTaskQueueException e) {...}
11 ...
```

4.3.4 Servizio Message Queue

Con il servizio Message Queue, la coda viene considerata come contenitore di messaggi. Con questa tipologia di coda, viene anche data la possibilità di comunicare con l'esterno dell'applicazione, permettendo ad altre applicazioni di accedere alla coda e gestire le richieste contenute. Ad esempio si potrebbero creare due code, una in ingresso e l'altra in uscita dall'applicazione per comunicare con il mondo esterno.

Message Queue service con la libreria CPIM

Questo servizio è nativo di Azure, ma non è del tutto implementato in Google.

Infatti in App Engine viene offerta una variante della Task Queue precedentemente introdotta, denominata Pull Queue, nella quale i task non vengono invocati automaticamente, ma devono essere catturati e processati. Il programmatore dovrà quindi provvedere autonomamente a fornire un proprio Consumer per queste code. Anche per questa tipologia, l'elemento inserito nella coda è un'istanza della classe *TaskOptions*. In questo caso, nell'oggetto non verranno memorizzate le informazioni relative al Task da eseguire, quali il path della servlet e i vari parametri, ma solamente il contenuto del messaggio da passare, sfruttando il metodo *payload(String s)*, che permette di memorizzare nel *TaskOptions* il contenuto del payload relativo alla richiesta.

Nella libreria, i limiti dimensionali di un messaggio sono fissati ad una grandezza di 64KB. Tale limite è il limite superiore emerso dall'integrazione dei servizi delle due piattaforme, imposto dalla piattaforma di Microsoft. Nel caso si desiderasse inviare un messaggio di dimensione maggiore, è consigliabile memorizzare l'oggetto tramite il servizio NoSQL, SQL o Blob e caricare la referenza a tale oggetto come messaggio della coda.

Factory del servizio Message Queue

Come per il servizio di Task Queue, il servizio Message Queue prevede una fase di configurazione, in cui viene specificata la tipologia di coda.

Per i dettagli della configurazione di una Message Queue, si veda la Sezione A.5, oppure il capitolo del plugin.

Come per gli altri servizi, ci sono due modi differenti per istanziare la factory del servizio di Message Queue: utilizzando l'MF (riga 2), oppure tramite il

metodo statico presente nella classe *CloudMessageQueue* (riga 5); come riportato nel Listato 4.8. Le API relative alla classe *CloudMessageQueueFactory* sono riportate in Figura 4.11

Listato 4.8: Istanziazione CloudMessageQueueFactory

```
1 //tramite MF
2 CloudMessageQueueFactory cmqf=MF.getFactory().getMessageQueueFactory()
3
4 //tramite metodo statico
5 CloudMessageQueueFactory cmqf=CloudMessageQueueFactory.
      getCloudMessageQueueFactory(CloudMetadata.getCloudMetadata())
```

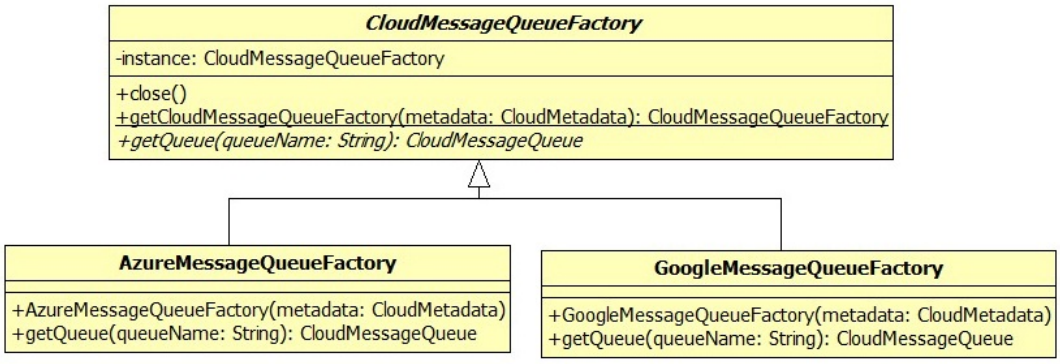


Figura 4.11: CloudMessageQueueFactory API

Classe CloudMessageQueue

Dopo aver istanziato la factory, si può procedere con l'istanziazione della coda vera e propria, chiamando il metodo *getQueue("queueName")* presente nella factory. Tale metodo crea la coda, se non esiste, e ritorna un valore che consiste in un'istanza di una classe che espone i metodi di un'interfaccia chiamata *CloudMessageQueue*.

L'interfaccia *CloudMessageQueue*, riportata in Figura 4.12, espone i metodi necessari per l'interazione con la coda, tra cui quelli per aggiungere un messaggio, leggerlo e cancellarlo dalla coda.

Esempio utilizzo API servizio Message Queue

Di seguito è riportato un esempio riguardante l'utilizzo della coda con il servizio Message Queue.

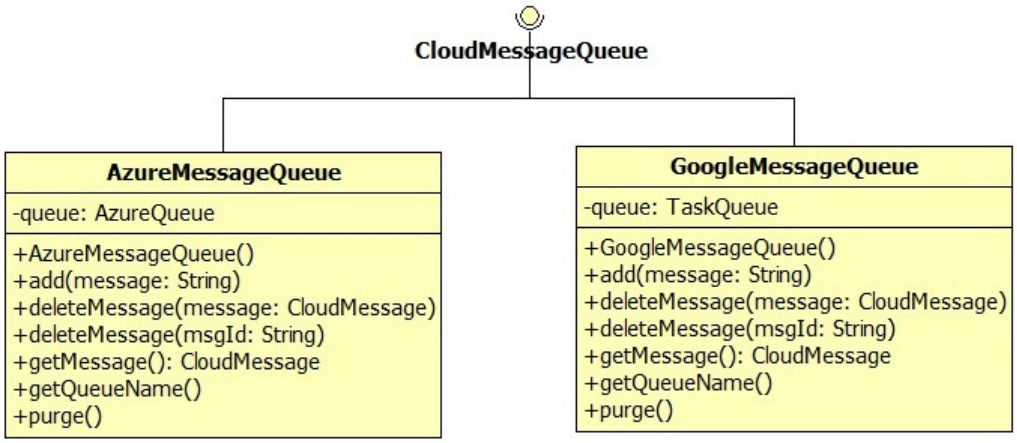


Figura 4.12: CloudMessageQueue API

Tale esempio, prevede l'implementazione di un producer, Listato 4.9, e di un consumer, Listato 4.10, in cui il producer aggiunge messaggi sulla coda e il consumer li cattura, li legge, li processa e li cancella. In particolare si noti nel consumer, alla riga 31 del Listato 4.10, la particolare modalità di istanziazione di un Thread.

Sulla piattaforma di Google, non è consentito istanziare Thread tramite la classe standard presente in *java.lang*, ma solo attraverso l'API *ThreadManager .createBackgroundThread(Runnable r)*.

Per uniformare la modalità di creazione di un Thread, e rendere così il codice completamente portabile, si è implementata una classe ad hoc, chiamata *CloudThread*, che espone un metodo statico *getThread()* che richiede in ingresso un oggetto della classe *Runnable*, contenente l'implementazione del metodo *run()* (esempio: righe 5-24), eseguito al momento dell'invocazione del metodo *start()* del *CloudThread* (esempio: riga 33).

Listato 4.9: Message Queue - Producer

```
1 public void producer(String message){
2     CloudMessageQueue q = MF.getFactory().getMessageQueueFactory().getQueue("queue
        ");
3     try {
4         q.add(message);
5     } catch (CloudMessageQueueException e) {...}
6 }
```

Listato 4.10: Message Queue - Consumer

```
1  public void consumer(){
2      final CloudMessageQueue queue = MF.getFactory().getMessageQueueFactory().
       getQueue("queue");
3    // Creazione del Runnable
4    Runnable r=new Runnable(){
5      public void run()
6      {
7        CloudMessage msg=null;
8        while(true)
9        {
10         msg=queue.getMessage();
11         if(msg!=null)
12         {
13           // esecuzione operazioni relative al messaggio letto
14           {
15             ...
16           }
17           try {
18             queue.deleteMessage(msg);
19           } catch (CloudMessageQueueException e) {
20             e.printStackTrace();
21           }
22         }
23       }
24     }
25   };
26   // Creazione Thread
27   Thread t=null;
28   try {
29     t = CloudThread.getThread(r);
30   } catch (ParserConfigurationFileException e) {
31     e.printStackTrace();
32   }
33   t.start();
34 }
```

4.3.5 Servizio Blob

Oltre ai servizi di persistenza di oggetti strutturati in schemi relazionali (SQL) e non (NoSQL), le piattaforme di Google e di Azure offrono un servizio di persistenza per oggetti caratterizzati da dimensioni rilevanti, denominati Blob (Binary Large Object). Tale servizio è denominato Blobstore per App Engine e Blob Service per Azure.

Blob service con la libreria CPIM

La standardizzazione di questo servizio si è rilevata complessa. Il servizio Blobstore di Google, propone delle modalità standard di caricamento e scaricamento di blob, del tutto incompatibili da quelle offerte dal Blob service di Azure.

Come accennato nel paragrafo della Sezione 3.1 riguardante questo servizio, App Engine, offre anche una modalità alternativa per la gestione dei blob, ossia utilizzando il servizio FileService. Questo servizio, esponendo delle API ˜file-like˝, permette di scrivere e leggere nei blob caricati sul Cloud. Questa modalità risulta essere più compatibile con la tecnica utilizzata in Azure per la gestione di questo servizio.

Il servizio offerto dalla libreria CPIM, ossia quello standardizzato, ha delle limitazioni rispetto ai servizi nativi, dovute all'integrazione dei servizi delle due piattaforme. In particolare, si è eliminata la possibilità, per quanto riguarda Azure, di organizzare i blob in container, in quanto tale funzionalità non è offerta in App Engine.

A tal proposito, per Windows Azure, i blob vengono inseriti in un unico container chiamato ˜default-container˝, settando il diritto di accesso al container al livello BLOB (si veda Sezione 3.2, paragrafo relativo al servizio blob).

Un'altra limitazione introdotta per il servizio di Azure, consiste nell'utilizzare un unico tipo di blob. La standardizzazione prevede infatti l'utilizzo dei soli Block Blob per quanto riguarda Azure. La dimensione massima dei blob deve essere inferiore a 32 Mb, limite imposto dalla piattaforma di Google, che prevede tale valore come limite superiore ammesso.

Factory del servizio Blob

Come per ogni servizio offerto dalla libreria, anche per i Blob è obbligatoria la configurazione iniziale di alcuni parametri. Per i dettagli riguardanti la configurazione di questo servizio si veda la Sezione A.4, oppure il Capitolo 5 relativo al plugin.

Come per gli altri servizi, ci sono due modi differenti per istanziare la factory relativa al servizio, ossia utilizzando l'MF, oppure tramite il metodo statico presente nella classe medesima, la *CloudBlobManagerFactory*.
Si veda a tal proposito il Listato 4.11. Il Class Diagram della factory è mostrato invece in Figura 4.13.

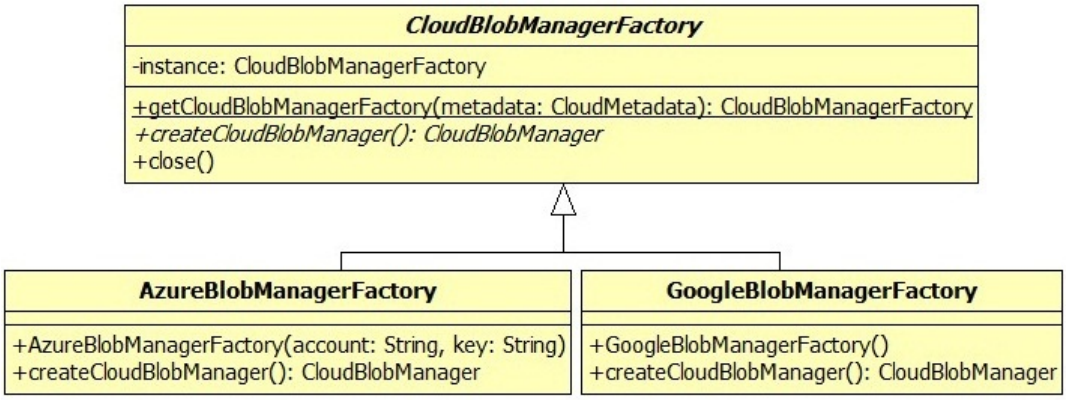


Figura 4.13: CloudBlobManagerFactory API

Listato 4.11: Istanziazione del servizio Blob

```
1 //tramite MF
2 CloudBlobManagerFactory cbmf=MF.getFactory().getBlobManagerFactory()
3
4 //tramite metodo statico
5 CloudBlobManagerFactory cbmf=CloudBlobManagerFactory.
      getCloudBlobManagerFactory(CloudMetadata.getCloudMetadata())
```

Gestore del servizio Blob

Dopo aver ottenuto un'istanza della factory del servizio Blob, occorre invocare il metodo *createCloudBlobManager()* per la creazione di un'istanza della classe *CloudBlobManager*. Questa classe, come mostrato in Figura 4.15, espone i seguenti metodi per interagire con il servizio Blob delle piattaforme:

- *UploadBlob*:

  Questo metodo permette il caricamento di un file tramite un array di byte che ne rappresenta il contenuto. Il nome del file rappresenta la chiave che univoca il file nello storage. Questo metodo permette di caricare fino a 32MB di contenuto in un'unica chiamata. Nel caso in cui il file superi tale limite, lo sviluppatore deve adottare la tecnica del partizionamento del file, ossia caricare cluster da 32MB contenenti una parte del file completo, ad esempio caricando file_1, file_2, etc. Nel momento del download, il file deve essere ricostruito ricomponendo i diversi cluster caricati precedentemente e relativi al file originale, richiamando il metodo *download("file_n")*, tante volte quanti sono i cluster caricati.

- *DownloadFile*

  Metodo che permette di recuperare il contenuto di un file tramite il nome, ossia la chiave, fornito al momento del caricamento del file. Tale metodo permette di recuperare file con una grandezza massima di 32MB, data dal fatto che tale dimensione corrisponde alla dimensione massima di caricamento. L'oggetto ritornato dall'invocazione del metodo *download ("fileName")*, è rappresentato dalla classe *CloudDownloadBlob*. Questo oggetto contiene tutte le informazioni necessarie per effettuare il download di un file da una servlet. La sua struttura è riportata di seguito in Figura 4.14.

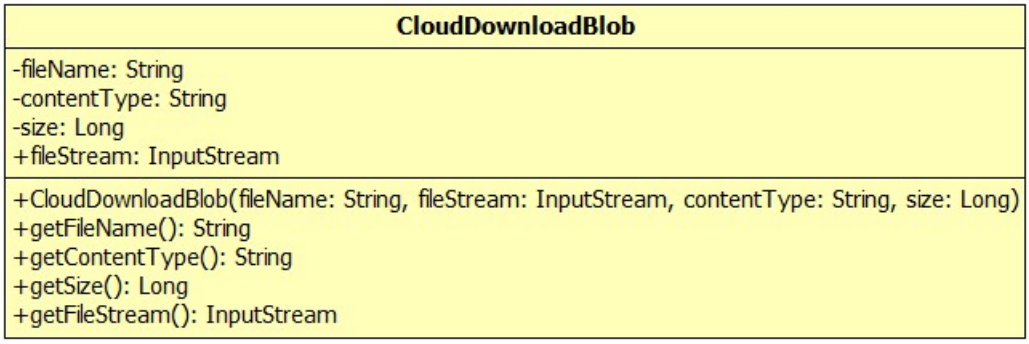


Figura 4.14: Struttura CloudDownloadBlob

- *DeleteBlob*

  Permette la cancellazione del blob, utilizzando il suo nome.

- *GetAllBlobFileName*

  Permette di richiedere la lista dei nomi dei file attualmente caricati nel servizio

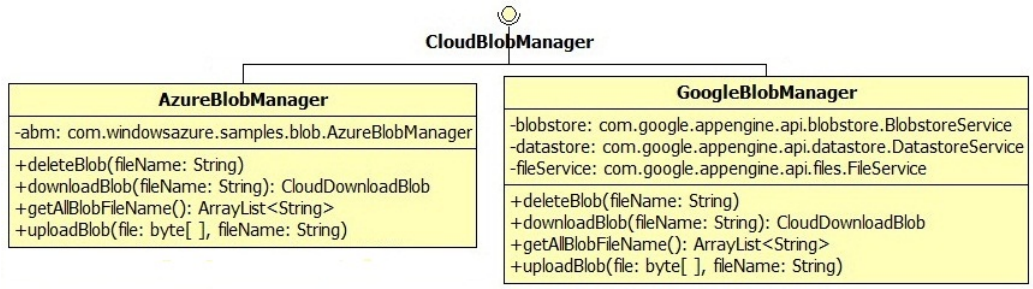


Figura 4.15: CloudBlobManager API

Esempio utilizzo API servizio Blob

Per spiegare meglio l'uso di questo servizio, si faccia riferimento alla creazione di una form contenente un campo file, riportata nel Listato 4.12.

Listato 4.12: Esempio di form HTML con campo file

```
1 <form action="Register" method="post" enctype="multipart/form-data">
2     E-mail: <input type="text"  name="mail" id="mail"/> <br />
3     First Name: <input type="text" name="firstName" id="firstName" /> <br />
4     Last Name: <input type="text" name="lastName" id="lastName" /> <br />
5     Picture: <input type="file" name="file" accept="image/*" />
6     <input name="submit" type="image" value="SUBMIT" />
7 </form>
```

Nel Listato 4.12 si può notare che nel caso di form contenente un campo file, occorre inserire *enctype="multipart/form-data"* nel tag form, come riportato nella riga 1. Questo valore permette alla servlet incaricata dell'esecuzione della richiesta, nel nostro caso *Register*, di capire la tipologia del contenuto della form. Nel Listato 4.13 viene riportata la servlet in questione.

Listato 4.13: Metodo doPost della servlet Register

```
1 protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
2     try {
3         request.setCharacterEncoding("UTF-8");
```

```
4       ServletFileUpload upload = new ServletFileUpload();
5       FileItemIterator iterator = upload.getItemIterator(request);
6       HashMap<String, String> fieldsMap = new HashMap<String, String>();
7       while (iterator.hasNext()) {
8           FileItemStream item = iterator.next();
9           InputStream stream = item.openStream();
10          if (item.isFormField()) {
11              String field = item.getFieldName();
12              String value = Streams.asString(stream);
13              fieldsMap.put(field, value);
14              stream.close();
15          } else {
16              String filename = item.getName();
17              fieldsMap.put("filename", filename);
18              byte[] buffer = IOUtils.toByteArray(stream);
19              MF.getFactory.getBlobManagerFactory().createCloudBlobManager()
20                  .uploadBlob(buffer, filename);
21              stream.close();
22          }
23      }
24  } catch (UnsupportedEncodingException e) {...}
25    catch (FileUploadException e) {...}
26 }
```

Come è intuibile dal Listato 4.13, si utilizza una modalità differente per la gestione dei dati ottenuti tramite form, rispetto a quella classica tramite l'oggetto *HttpServletRequest*. Questa modalità prevede l'utilizzo della libreria *CommonFileUpload*.

Come descritto nella Sezione 3.1, la motivazione del suo utilizzo è il mancato supporto da parte di App Engine della nuova versione delle Servlet, che permettono la gestione dei campi file ricevuti direttamente tramite l'oggetto *HttpServletRequest*.

Come riportato nella riga 5 del Listato 4.13, la servlet ottiene un iteratore, in modo da poter processare, item per item, la richiesta. Ottenuto un particolare oggetto (si veda la riga 8), la servlet controlla se rappresenta un campo di una form (riga 10). In caso affermativo, il valore di tale campo viene scaricato tramite l'apertura di un InputStream e salvato nella mappa. In caso negativo, l'item letto rappresenta il contenuto del campo file.

Per poter salvare il contenuto dello stream in un array di byte occorre utilizzare il metodo *IOUtils.toByteArray()*, passando come argomento lo stream dell'item (riga 18).

Una volta ottenuto il contenuto del file, non resta altro che richiamare il metodo del servizio Blob designato al caricamento del blob (riga 19).

Dopo aver mostrato un esempio di come caricare un file in un blob, nel Listato 4.14 viene fornito un'esemplificazione dello scaricamento di un blob. In tale esempio, si presuppone che la servlet riceva come parametro il nome del file (riga 2). In questo caso le operazioni da effettuare sono:

- Invocazione metodo *downloadBlob(fileName)*: si ottiene un'istanza della classe *CloudDownloadBlob*, che, come descritto precedentemente, contiene tutto ciò che serve per scaricare localmente il contenuto di un blob (riga 4)
- Apertura *InputStream*: apertura dello stream contenuto nel *CloudDownloadBlob* appena scaricato (riga 5)
- Impostazione di alcune informazioni necessarie al download di un file da parte di una servlet (righe 6-8)
- Download del file: aprire un *OutputStream*, che permette di restituire nella risposta il contenuto del blob richiesto (righe 9-15)

Listato 4.14: Esempio di download Blob tramite servlet

```
1  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
       ServletException, IOException {
2      String fileName=req.getParameter("fileName");
3      CloudBlobManager bm = MF.getFactory().getBlobManagerFactory().
         createCloudBlobManager();
4      CloudDownloadBlob blob = bm.downloadBlob(fileName);
5      InputStream streamDown = blob.getFileStream();
6      resp.setContentLength((int)blob.getSize());
7      resp.setContentType(blob.getContentType());
8      resp.setHeader("Content-Disposition", "attachment; filename=\"" + fileName + "\""
         );
9      ServletOutputStream out = resp.getOutputStream();
10     BufferedOutputStream bufferOut = new BufferedOutputStream(out);
11     int b = 0;
```

```
12      byte[] bufferData = new byte[8192];
13      while ((b = streamDown.read(bufferData)) != -1) {
14          bufferOut.write(bufferData, 0, b);
15      }
16      bufferOut.flush();
17      bufferOut.close();
18      out.close();
19      streamDown.close();
20  }
```

4.3.6 Servizio Mail

Il servizio Mail consente alle applicazioni di inviare mail, attraverso un server SMTP.

Per App Engine il server SMTP messo a disposizione dell'SDK Java è quello di Google. Mentre per quanto riguarda la piattaforma di Windows Azure, non vi è alcun SMTP all'interno della piattaforme, ma viene data la possibilità di integrare un servizio chiamato SendGrid [22]. SendGrid è un SaaS che offre le capacità di un server SMTP.

L'istanziazione di questo servizio avviene, come tutti i servizi disponibili nella libreria, attraverso due modalità: tramite l'MF (riga 2), oppure tramite il metodo statico presente nella classe *CloudMailManager* (riga 5). Si veda come esempio, il Listato 4.15.

Listato 4.15: Istanziazione del servizio Mail

```
1  //tramite MF
2  MF.getFactory().getMailManager()
3
4  //tramite metodo statico
5  CloudMailManager.getCloudMailManager(CloudMetadata.getCloudMetadata())
```

In Figura 4.16 è mostrato il Class Diagram della classe *CloudMailManager*. La standardizzazione di questo servizio permette momentaneamente la sola funzionalità di invio di una mail senza allegati e in formato testuale, creata tramite l'oggetto *CloudMail* (Figura 4.17).

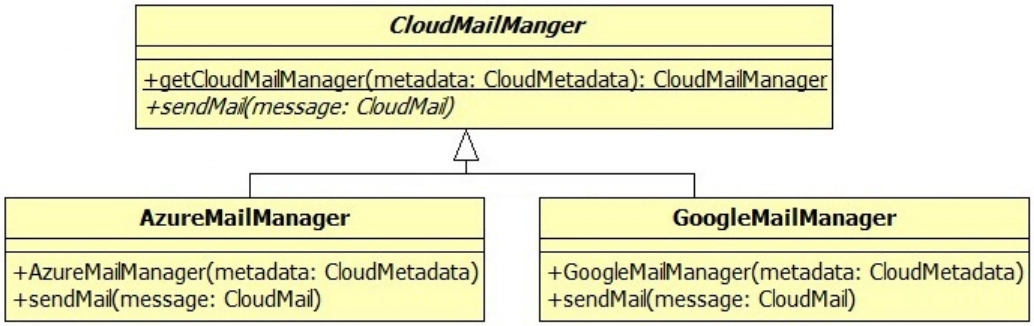


Figura 4.16: CloudMailManager API

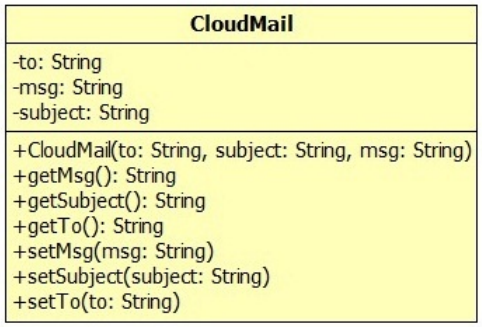


Figura 4.17: CloudMail

Esempio utilizzo API servizio Mail

Nel Listato 4.16 vi è un semplice esempio di invio di una mail tramite il servizio standardizzato, in cui viene creato un oggetto *CloudMail* (riga 2), specificando l'indirizzo mail del destinatario, l'oggetto e il testo del messaggio. Tale oggetto viene poi passato come parametro al metodo *sendMail()*, responsabile dell'invio (riga 3).

Listato 4.16: Esempio di invio mail

```
1 String messageBody =”testo della mail” ;
2 CloudMail msgToSend = new CloudMail(”foo@foo.com”,”Welcome to mic.com!”,
      messageBody);
3 MF.getFactory().getMailManager().sendMail(msgToSend);
```

4.3.7 Servizio Memcache

Il servizio di Memcache viene utilizzato per ridurre gli accessi ai vari servizi di storage, utilizzando parte della memoria RAM come memoria di archiviazione. Viene usato per velocizzare il processo di recupero di quei dati che vengono spesso acceduti e raramente modificati.

Google App Engine offre nativamente il servizio, fornendo le relative API direttamente nella SDK per Java. Per quanto riguarda Windows Azure, il servizio Windows Azure Caching supporta il protocollo Memcache, ma occorre configurare ed implementare manualmente la componente client di questo protocollo, come descritto nella Sezione 3.2, nel paragrafo relativo al servizio di Caching.

Memcache service con la libreria CPIM

Nella standardizzazione si è deciso di implementare un servizio Memcache con accesso sincrono. Si è deciso inoltre di includere nelle API solo i metodi che implementano le operazioni fondamentali che si svolgono tipicamente su una cache, riportate in seguito:

- Verifica presenza di un particolare oggetto
- Cancellazione di un'oggetto
- Inserimento di un nuovo oggetto
- Richiesta di un oggetto
- Sostituzione di un oggetto

Per interagire con il servizio occorre richiedere l'istanziazione di un oggetto della classe *CloudMemcache*. Ciò è possibile attraverso due modalità: tramite l'MF, oppure tramite il metodo statico presente nella classe medesima. Un esempio è riportato nel Listato 4.17.

Listato 4.17: Istanziazione del servizio di Memcache

```
1 //tramite MF
2 CloudMemcache cm=MF.getFactory().getCloudMemcache()
3
4 //tramite metodo statico
```

```
5 CloudMemcache cm=CloudMemcache.getCloudMemcache(CloudMetadata.
      getCloudMetadata())
```

Tramite l'istanza ottenuta, il programmatore ha la possibilità di salvare sulla RAM una coppia chiave-valore di qualsiasi oggetto, purché serializzabile, e richiederlo successivamente tramite chiave.

Il Class Diagram della classe *CloudMemcache* è mostrato in Figura 4.18. Si noti che molte funzionalità hanno una duplice forma: oltre a quella standard che esegue l'operazione su una singola coppia chiave-valore, è presente il metodo che permette, in un'unica chiamata, di eseguire l'operazione su un set di coppie.

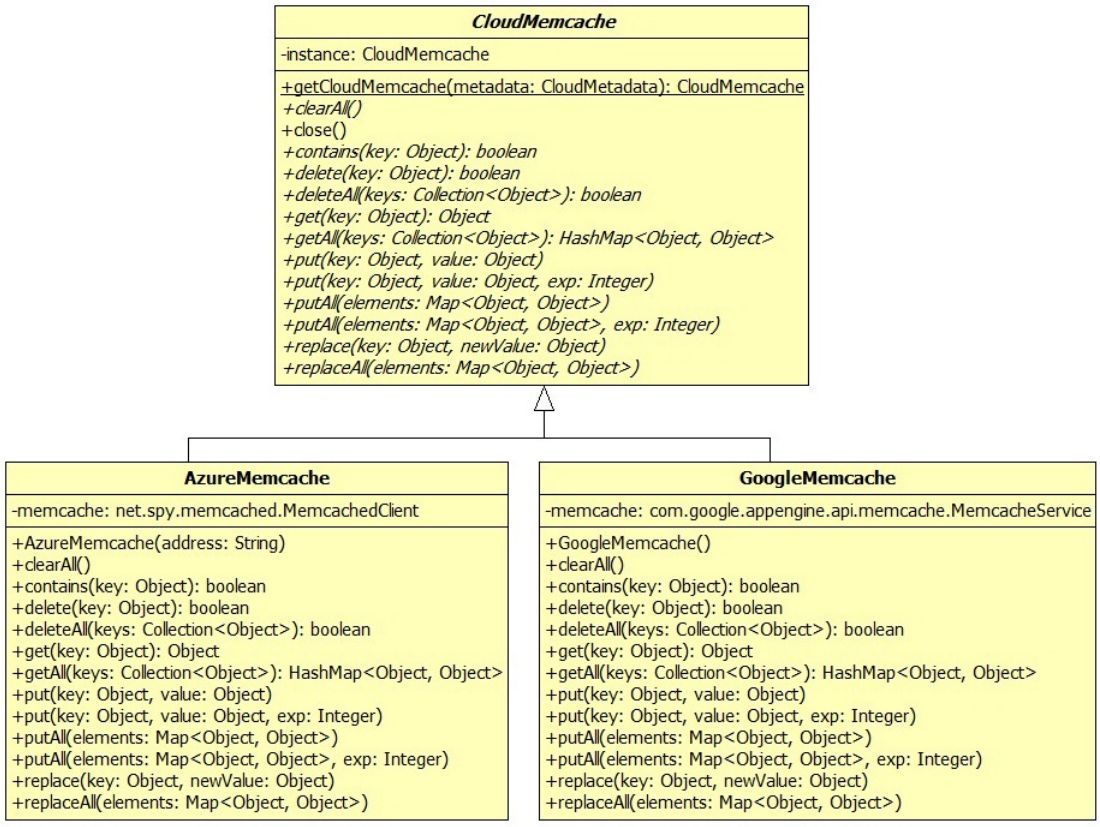


Figura 4.18: CloudMemcache API

Esempio utilizzo API servizio Memcache

Di seguito viene riportato, nel Listato 4.18, un esempio di utilizzo del servizio Memcache, nel quale:

- Si richiede l'istanza di un *CloudMemcache* (riga 1)
- Si controlla se una particolare coppia chiave-valore è presente nella memcache (riga 4)
- In caso positivo, si richiedere il valore associato alla chiave (riga 5)
- In caso negativo, si richiede il valore ad un altro sistema di storage (righe 7-12). Ottenuto il valore, si salva una coppia chiave-valore relativa a questo dato in memcache, per una possibile richiesta futura (riga 13).

Listato 4.18: Esempio di utilizzo del servizio di Memcache

```
1  CloudMemcache cache = MF.getFactory.getCloudMemcache();
2  String usermail= "abcd@def.it";
3  Object value= null;
4  if(cache.contains(usermail))
5          value = cache.get(usermail);
6  else {
7          CloudEntityManager em = MF.getFactory.getEntityManagerFactory().
       createCloudEntityManager();
8          List<UserRatings> ratings = em.createQuery("SELECT ur FROM UserRatings ur
       WHERE email=:email").setParameter("email", usermail).getResultList();
9          ArrayList<String> topics = new ArrauList<String>();
10         for (UserRatings ur : ratings) {
11             topics.add(ur.getTopicName());
12         }
13         cache.put(usermail,topics);
14 }
```

4.4 Riepilogo

Nella Tabella 4.1 vengono riepilogate le caratteristiche e il limiti delle due piattaforme e della libreria CPIM, relativamente ai servizi supportati da quest'ultima.

Servizio	Google App Engine	Windows Azure	Libreria CPIM
NoSQL	API dirette Interfaccia JPA Interfaccia JDO	API dirette Interfaccia JPA (terze parti)	Interfaccia JPA Supporto oggetti serializzabili (@Embedded)
SQL	MySQL-like Massimo 10GB per istanza JDBC Driver	SQL Server-like sharding JDBC Driver	java.sql
Memcache	Sincrona/Asincrona	Dedicata/Co-locata Memcache supportato dal servizio Caching (terze parti)	Sincrona
Blob	Due tipologie di gestione Organizzazione piatta Massimo 32MB grandezza blob	Due tipologie di blob Organizzazione gerarchica Massimo 200GB/1TB per Block/Page blob	Organizzazione piatta Massimo 32MB grandezza blob
Queue	Due tipologie di gestione code Code di task	Code di messaggi Limite 64KB per messaggio	Code di messaggi Code di task Limite 64KB per messaggio
Mail	SMTP interno API dirette supporta java.mail	SMTP esterno supporta java.mail	supporta java.mail

Tabella 4.1: Riepilogo servizi standardizzati

Capitolo 5

Cloud Platform Independent Model: Plugin

Il plugin sviluppato per Eclipse Helios 3.6, permette la configurazione e la preparazione dell'ambiente di sviluppo per il deployment di progetti Web, codificati in Java, che fanno utilizzo della libreria CPIM per l'interfacciamento con i vari servizi offerti dalle piattaforme Cloud supportate, quali App Engine di Google e Azure di Windows.

Tale plugin ha lo scopo di guidare il programmatore, attraverso finestre grafiche, alla compilazione delle varie configurazioni necessarie per l'utilizzo dei servizi. Le configurazioni vengono utilizzate dalla libreria CPIM, grazie alle quali riesce a capire quale Cloud è stato scelto in modo da configurare l'interfacciamento ai vari servizi nativi.

Il capitolo inizialmente descrive a livello architetturale la composizione del plugin, gli oggetti e i meccanismi utilizzati per effettuare le varie operazioni (Sezione 5.1). Successivamente vengono illustrate le interfacce grafiche componenti il plugin, il loro contenuto, i requisiti e le predisposizioni necessarie per il corretto utilizzo del plugin (Sezione 5.2).

5.1 Modello architetturale

Il plugin è stato sviluppato utilizzando il Plugin Development Environment (PDE) di Eclipse. L'aggiunta di tale plugin ad Eclipse non è invasiva, in quanto la sua installazione prevede l'inserimento di un solo pulsante sulla barra

degli strumenti. L'azione assegnata a quest'ultimo prevede l'istanziazione di un oggetto chiamato CloudWizard.

5.1.1 CloudWizard

Il CloudWizard è la classe che implementa l'intero plugin. Essa contiene tutte le pagine componenti la procedura guidata, nonchè il CloudWizardModel: classe contenente tutte le informazioni derivanti dalle interazioni con le pagine, necessarie per l'attuazione del deployment.

Oltre a questi oggetti, il CloudWizard si occupa dell'abilitazione della procedura di terminazione del plugin, basata sulla corretta e completa compilazione delle pagine contenute al suo interno.

Infine viene gestita la logica applicativa necessaria per attuare il processo di deployment invocato alla terminazione della procedura guidata descritta in seguito.

Al termine della procedura di deployment, viene salvato un file XML nel progetto sorgente contenente le informazioni inserite durante la procedura. Il file così ottenuto potrà essere utilizzato per il completamento automatico della procedura guidata, in modo da velocizzarne i successivi deployment.

5.1.2 Pagine

Il plugin ha un numero di pagine abbastanza considerevole a causa del gran numero di informazioni richieste per l'esecuzione della procedura di deployment. Ad ogni cambio pagina, le informazioni contenute in essa vengono salvate nel CloudWizardModel e prelevate al termine della procedura per l'esecuzione delle operazioni di deployment.

Di seguito vengo elencate le pagine presenti e la loro funzionalità, suddividendole nelle tre macrofasi del processo:

- Definizione informazioni relative al progetto sorgente

  Questa prima fase ha l'obiettivo di istruire la procedura guidata in merito alla locazione del progetto sorgente. Tale fase è composta da due pagine:

  - ProjectPage: richiede la specifica del percorso in cui prelevare il progetto sorgente, inoltre, in caso di presenza di configurazioni pas-

sate, permette il caricamento delle informazioni e il conseguente completamento automatico della procedura

  - SourceCodePage: richiede la specifica di due cartelle costituenti il progetto sorgente, la prima contenente le classi Java, la seconda contenente le pagine JSP e la cartella WEB-INF

- Configurazione dei servizi

  In questa fase la compilazione del plugin richiede informazioni relative alle configurazioni dei vari servizi. Si compone di tre pagine:

  - ConfigurationPage: viene richiesto all'utente di specificare la piattaforma su cui effettuare il deployment, oltre alle impostazioni per la configurazione dei servizi di Mail, Memcache e SQL
  - PersistencePage: in tale pagina vengono richieste le informazioni necessarie per la configurazione della JPA utilizzata per i servizi di persistenza NoSQL, in Google, e per lo storage account in Azure (il layout di tale pagina cambia a seconda del Cloud scelto)
  - QueueConfigurationPage: permette di aggiungere e configurare le code utilizzate dall'applicazione, la procedura di aggiunta di una coda prevede la visualizzazione a video di una schermata di dialogo in cui specificare le informazioni necessarie

- Definizione informazioni relative al deployment

  In quest'ultima fase il plugin richiede le informazioni necessarie per istruire la procedura di deployment per quanto riguarda il progetto da configurare e in cui migrare il contenuto. Tale fase è composta da un numero variabile di pagine dipendente dal Cloud scelto e del numero di livelli su cui deployare l'applicazione:

  - Deployment in Azure
    * AzureDeploymentPage: l'utente deve specificare il percorso del progetto di destinazione, una Worker Role all'interno di esso in cui fare il deployment e il nome del WAR da attribuire all'applicazione. In caso di deployment su due livelli, all'utente viene richiesto di specificare una seconda Worker Role, differente dalla prima, in cui deployare il backend dell'applicazione

- Deployment in Google
  - GoogleDeploymentPage: tale pagina richiede di specificare le stesse informazioni richieste nella fase di definizione del progetto sorgente per impostare le informazioni del progetto di destinazione, con l'aggiunta della specifica dell'application id e della versione da deployare sul Cloud
  - GoogleBackendPage (solo su due livelli): identica alla precedente, ma riferita al progetto dedicato alla parte di backend

Modello UX e sequence diagram

Di seguito vengono riportati il modello UX (Figura 5.1) del plugin e i sequence diagram di due scenari di deployment: il primo, riportato in Figura 5.2 e in Figura 5.3, raffigura il deployment di un'applicazione su Windows Azure, mentre il secondo, riportato in Figura 5.4 e in Figura 5.5, raffigurante il processo di deployment di un'applicazione su Google App Engine.

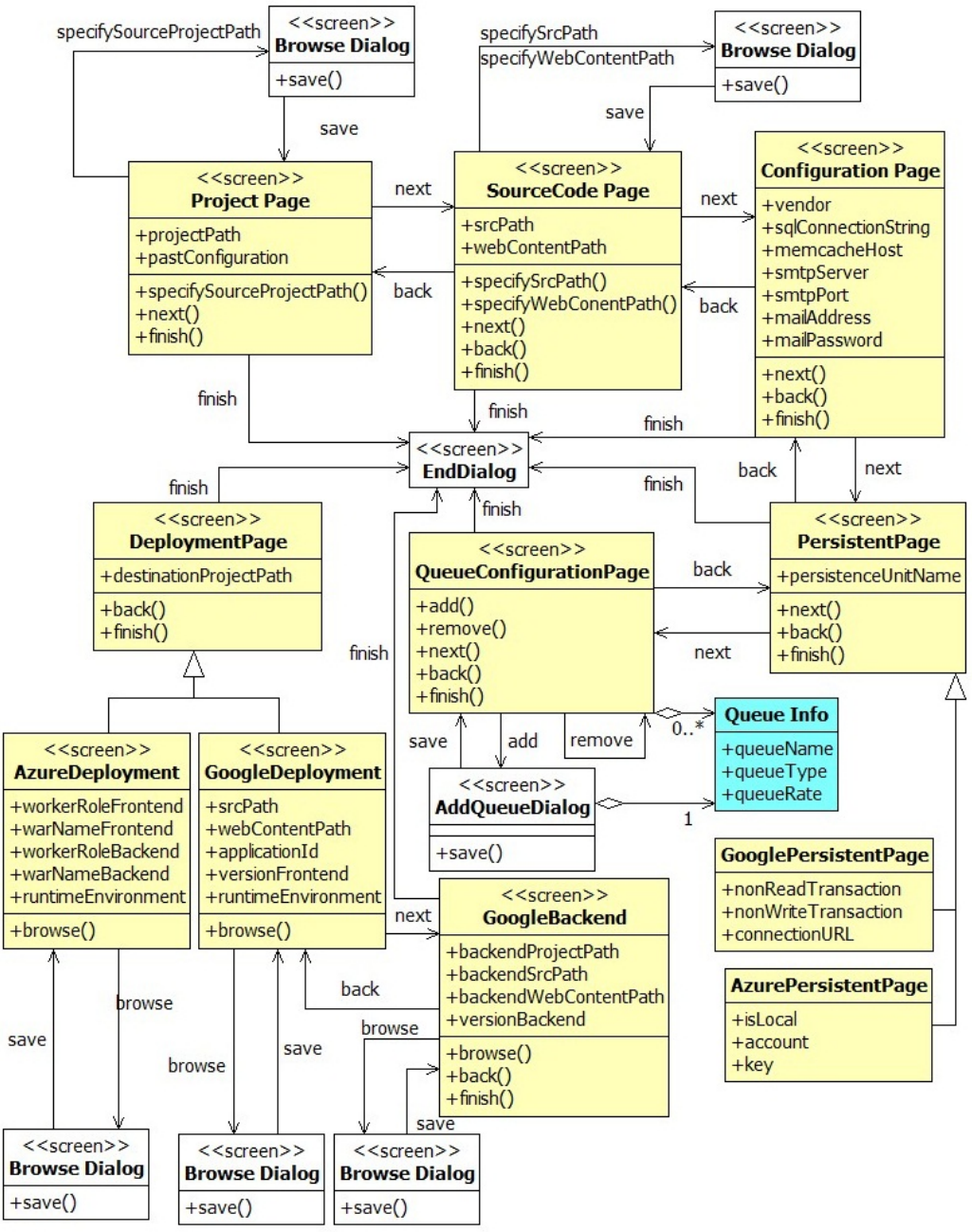


Figura 5.1: UX model

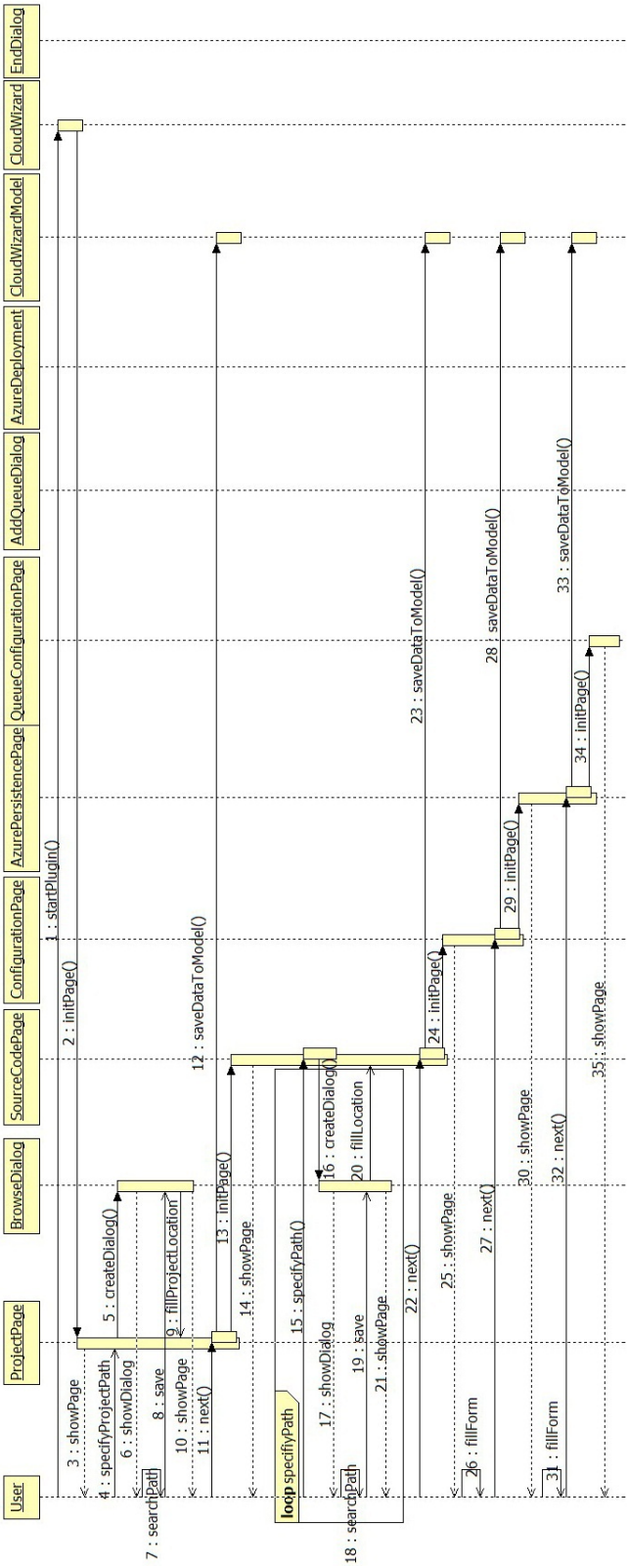


Figura 5.2: Sequence Diagram: deployment in Azure (1 di 2)

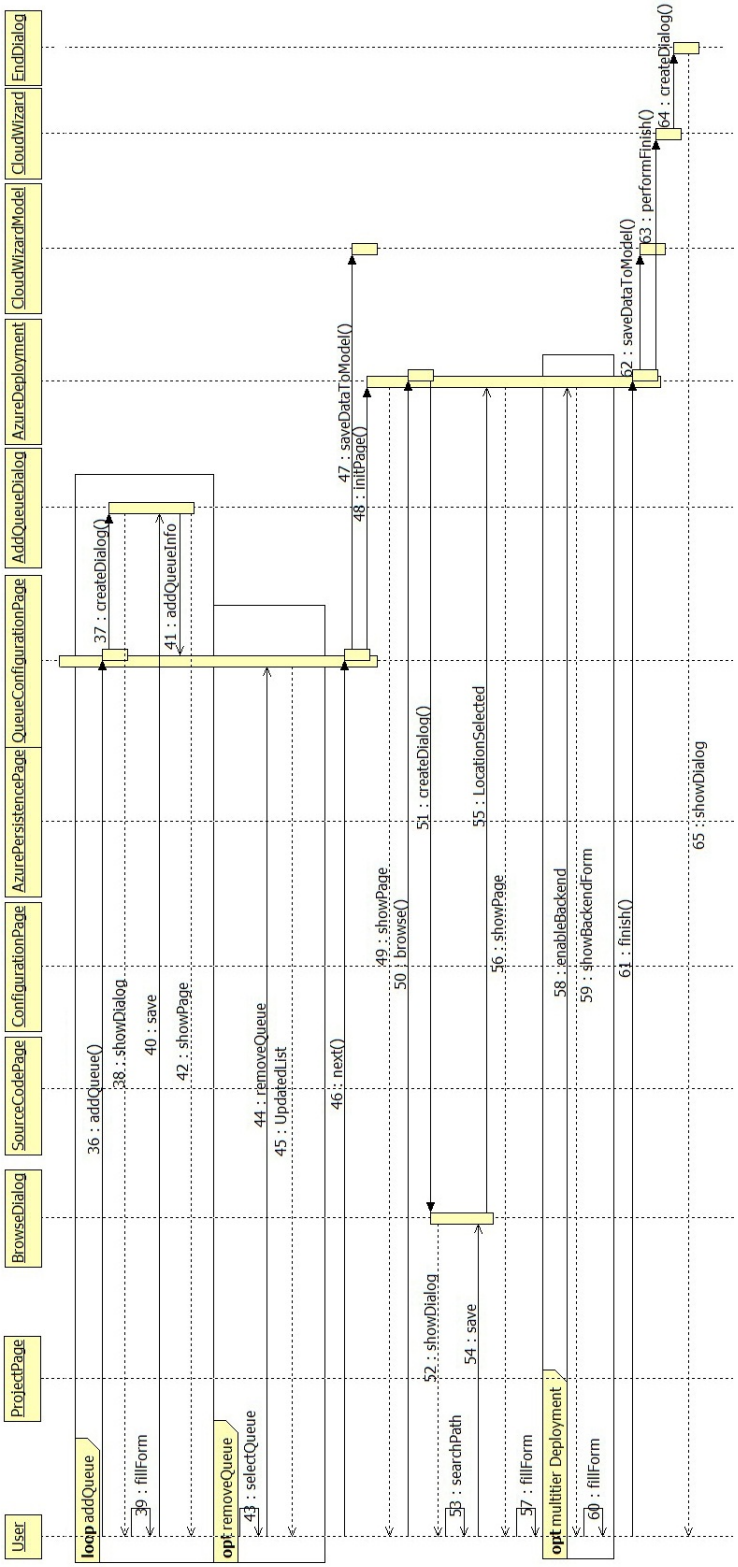


Figura 5.3: Sequence Diagram: deployment in Azure (2 di 2)

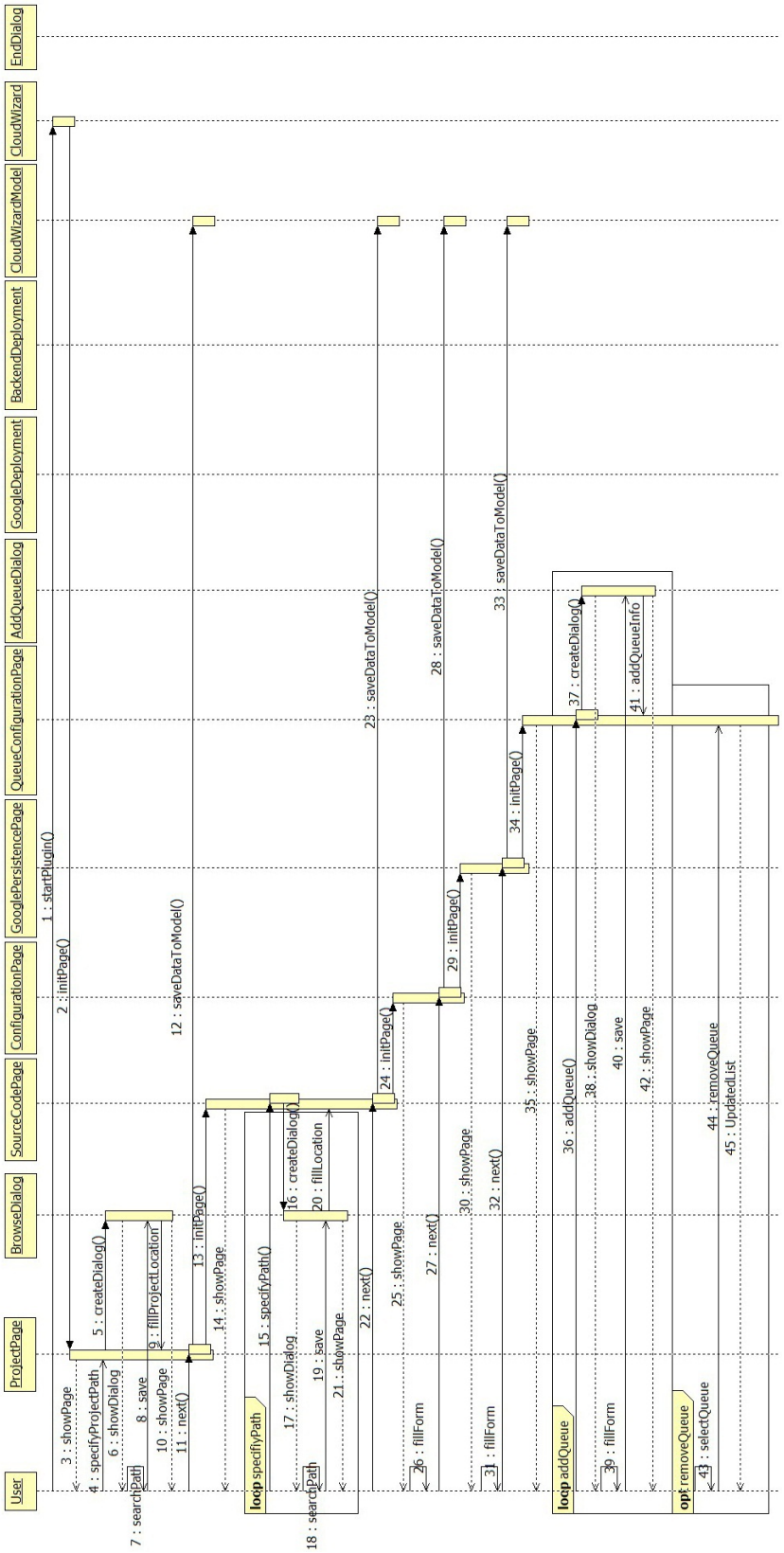


Figura 5.4: Sequence Diagram: deployment in App Engine (1 di 2)

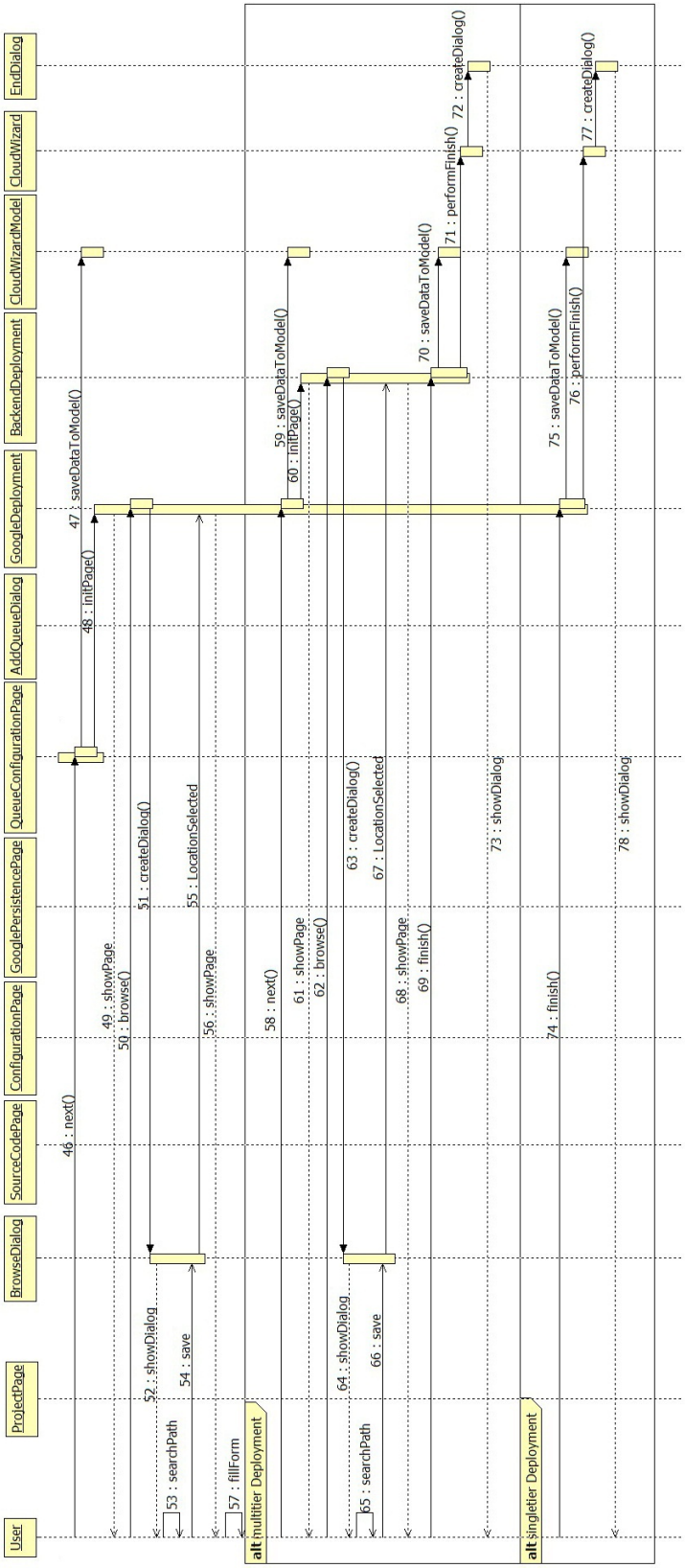


Figura 5.5: Sequence Diagram: deployment in App Engine (2 di 2)

5.1.3 Procedura di deployment

Di seguito vengono illustrate le operazioni eseguite dal plugin al termine della procedura guidata, per entrambe le piattaforme supportate.

Deployment in Google

Nel caso in cui il Cloud provider scelto fosse quello di Google App Engine, i passi effettuati dal processo di deployment sono i seguenti:

1. Spostamento del seguente contenuto dal progetto sorgente al progetto di destinazione:
   (a) cartella contenente le classi Java
   (b) cartella contenente le pagine JSP e la cartella WEB-INF
2. Aggiunta delle referenze del classpath del progetto sorgente nel progetto di destinazione
3. Creazione dei seguenti file XML di configurazione nel progetto di destinazione, nella cartella META-INF, contenuta all'interno della cartella contenente i sorgenti
   (a) *queue.xml*: relativo alla configurazione delle code (copiato anche nella cartella WEB-INF del progetto di destinazione)
   (b) *persistence.xml*: relativo alla configurazione della JPA
   (c) *configuration.xml*: relativo alle configurazioni dei servizi restanti
4. Spostamento delle librerie relative alle dipendenze della libreria CPIM per Google App Engine, qui elencate:
   - appengine-api-1.0-sdk-1.6.4.jar
   - appengine-api-labs-1.6.4.jar
   - appengine.jsr107cache-1.6.4.jar
   - datanucleus-appengine-1.0.10.final.jar
   - datanucleus-core-1.1.5.jar
   - datanucleus-jpa-1.1.5.jar

- geronimo-jpa 3.0 spec-1.1.1.jar
- geronimo-jta 1.1 spec-1.1.1.jar
- jdo2-api-2.3-eb.jar
- jsr107cache-1.1.jar
- mysql-connector-java-5.1.21-bin.jar

5. Modifica del file *appengine−web.xml*

Nel caso in cui il deployment venga fatto su due livelli, quindi differenziando frontend e backend, i passi precedentemente elencati vengono ripetuti per ogni livello.

Deployment in Azure

Nel caso in cui il Cloud provider scelto fosse quello di Windows Azure, i passi effettuati dal processo di deployment sono i seguenti:

1. Copia del contenuto del progetto sorgente in una cartella temporanea
2. Creazione dei seguenti file XML di configurazione
   (a) *queue.xml*: relativo alla configurazione delle code
   (b) *persistence.xml*: relativo alla configurazione della JPA
   (c) *configuration.xml*: relativo alle configurazioni dei servizi restanti
3. Spostamento delle librerie relative alle dipendenze della libreria CPIM per Windows Azure, qui elencate:
   - commons-lang-2.6.jar
   - hibernate-jpa-2.0-api-1.0.1.Finale.jar
   - javassist-3.12.1.GA.jar
   - jpa4azure-1.0.jar (versione estesa)
   - microsoft-windowsazure-api-0.2.2.jar
   - spymemcached-2.8.4.jar
   - sqljdbc4.jar
   - zql.jar

4. Creazione WAR file nel progetto di destinazione
   (a) In caso di deployment su un unico livello
      - Creazione WAR relativo all'applicazione
      - Configurazione e creazione WAR relativo al consumer della coda (solo se presente almeno una Task Queue)
   (b) In caso di deployment su due livelli (solo se presente almeno una Task Queue)
      i. Livello frontend
         - Creazione WAR relativo all'applicazione
      ii. Livello backend
         - Creazione WAR relativo all'applicazione
         - Configurazione e creazione WAR relativo al consumer della coda
5. Cancellazione della cartella temporanea e del suo contenuto

5.2 Guida all'utilizzo

In questa sezione vengono fornite le informazioni necessarie per il corretto impiego del plugin. Vengono dapprima indicati i requisiti necessari all'utilizzo del plugin, dall'installazione in Eclipse, alla predisposizione dell'ambiente di sviluppo necessario per un corretto deployment dell'applicazione (Sezione 5.2.1). Infine vengono dettagliate pagina per pagina le informazioni richieste durante la procedura guidata (Sezione 5.2.2).

5.2.1 Precondizioni e predisposizioni per il deployment

La precondizione necessaria per poter utilizzare il plugin consiste nell'utilizzare le interfacce esposte dalla libreria CPIM, come metodo di interfacciamento con i servizi offerti dalle piattaforme Cloud, per l'implementazione dell'applicazione web. Occorre quindi importare nel classpath dell'applicazione web il JAR relativo alla libreria CPIM.

Per quanto riguarda lo sviluppo dell'applicazione, è necessaria la referenziazione di una libreria esponente l'interfaccia JPA versione 1.0, qualsiasi essa sia,

in modo da permettere una corretta compilazione. La libreria JPA referenziata, non deve però essere importata nel classpath dell'applicazione web, in quanto tale servizio verrà coperto interamente a runtime dalle librerie specifiche delle piattaforme Cloud.

Per effettuare il deploy, occorre utilizzare i plugin per Eclipse, messi a disposizione dalle singole piattaforme. In particolare sono necessari:

- Per Google App Engine

  Il plugin scaricabile all'indirizzo `https://dl.google.com/eclipse/plugin/3.6`

- Per Windows Azure

  Il plugin scaricabile all'indirizzo `http://dl.msopentech.com/eclipse`

Il plugin relativo alla libreria CPIM, non effettua direttamente il deploy sulla piattaforma scelta, ma configura e trasferisce il contenuto del progetto Web nel progetto relativo al plugin specifico della piattaforma, che ne permetterà il deployment sul Cloud.

La predisposizione necessaria, consiste nella creazione di un progetto (o più) tramite i plugin delle piattaforme. In particolare per il deployment sulla piattaforma di Google, occorre creare un ¨New Web Application Project˝, e deselezionare l'utilizzo di GWT, in quanto tale tool non è standardizzato dalla libreria. Nel caso si decidesse di caricare l'applicativo su più livelli, ad esempio frontend e backend, occorrerebbe creare un progetto per ogni livello.

Per il deployment sulla piattaforma di Windows, occorre invece creare un unico progetto, attraverso il plugin sopracitato. Il progetto è denominato ¨Windows Azure Deployment Project˝. La procedura guidata per la creazione di questo tipo di progetto, richiede di specificare il percorso in cui poter trovare la JDK e il server su cui far girare l'applicazione web. In particolare il server a cui si farà riferimento in seguito è Tomcat versione 7.

Al termine della procedura di creazione, sarà possibile notare la presenza di una cartella particolare, denominata ¨WorkerRole1˝. Tale cartella rappresenta appunto la Worker Role che si verrà a creare nel momento del deployment sul Cloud. Per caricare un applicazione su più livelli, occorrerà creare una Worker Role per ogni livello.

Dopo questa breve spiegazione dei requisiti necessari per il corretto utilizzo del plugin relativo alla libreria CPIM, vengono analizzate passo per passo le finestre grafiche proposte dalla procedura guidata, nonchè le istruzioni per una corretta compilazione delle informazioni richieste.

5.2.2 Interfacce grafiche

Lo starting del plugin avviene tramite un pulsante situato sulla barra degli strumenti, rappresentato dall'icona .

L'azione catturata al click di tale pulsante inizializza l'interfaccia grafica e visualizza a video la prima pagina della procedura guidata.

Caricamento progetto sorgente

La prima pagina visualizzata a video, nel momento dell'attivazione del plugin, è composta dall'interfaccia grafica riportata in Figura 5.6.

Tale pagina richiede di specificare il percorso in cui si trova il progetto web sorgente, ossia quello da caricare. Una volta specificato il percorso, il plugin controlla la presenza di configurazioni passate usate per il deployment dell'applicazione. Questa opzione è stata introdotta per permettere la velocizzazione dei successivi deployment, dovuti ad esempio ad aggiornamenti dell'applicazione e per la correzione di errori.

Il plugin mantiene in memoria le ultime configurazioni utilizzate per tutte e quattro i deployment possibili:

- Azure locale
- Azure su Cloud
- Google locale
- Google su cloud

Nel caso in cui venisse selezionata una configurazione passata, l'azione di cambio pagina effettua la compilazione automatica di tutte le rimanenti pagine, e la conseguente abilitazione del tasto "Finish". È comunque consigliabile ricontrollare la corretta compilazione.

Al contrario, nel caso in cui non venisse selezionata alcuna configurazione passata, l'utente deve fornire tutte le informazioni richieste per il corretto completamento della procedura guidata.

In entrambi i casi, una volta specificato il percorso del progetto sorgente, occorre cliccare sul tasto "Next", visualizzando la seconda pagina.

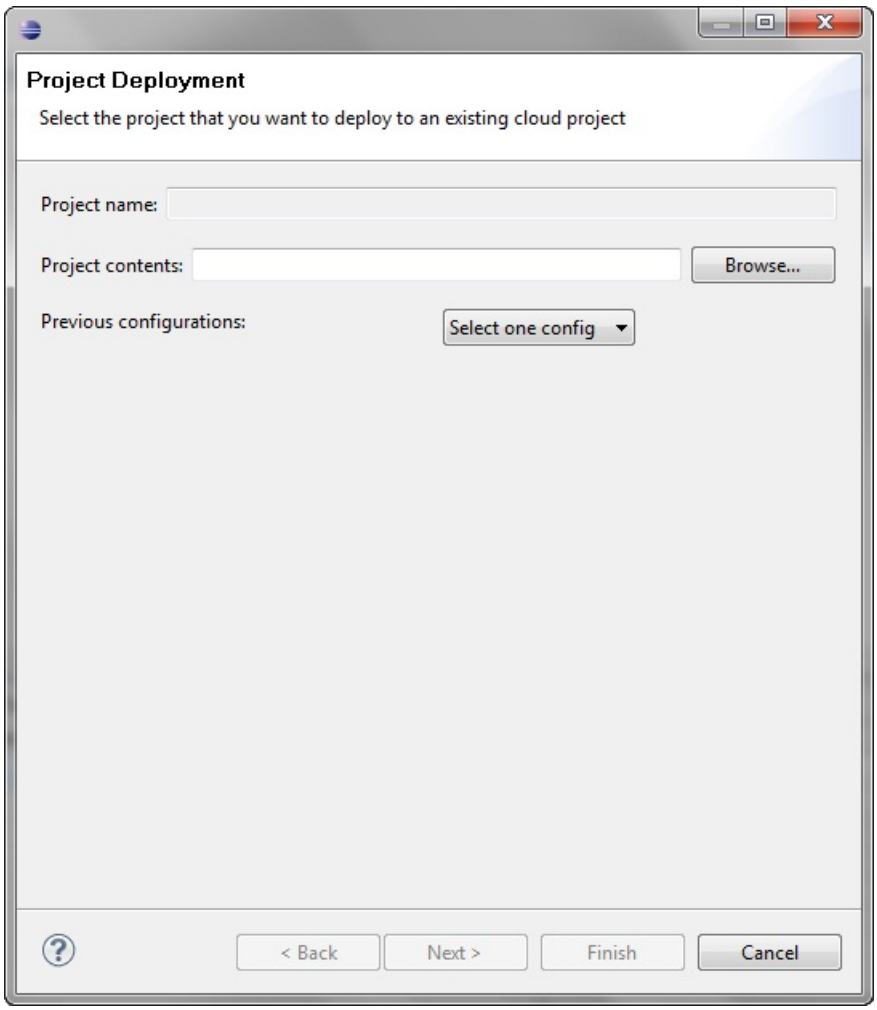


Figura 5.6: Project page

Nella seconda pagina, riportata in Figura 5.7, viene richiesto all'utente di specificare il percorso delle cartelle in cui si trovano le sorgenti (Src) e il contenuto web (WebContent). Le cartelle richieste, devono essere contenute nel percorso, precedentemente specificato, relativo al progetto web. Tali percorsi potrebbero essere automaticamente specificati nel caso in cui venissero utilizzate le cartelle impostate di default al momento della creazione del progetto web, ossia quelle denominate "src"e "WebContent".

In questa fase, il plugin ha la capacità di rilevare, nella cartella relativa al contenuto web, la presenza o meno del JAR relativo alla libreria CPIM. In caso di assenza di tale JAR, viene restituito un errore, in quanto la presenza della libreria è richiesta dal plugin.

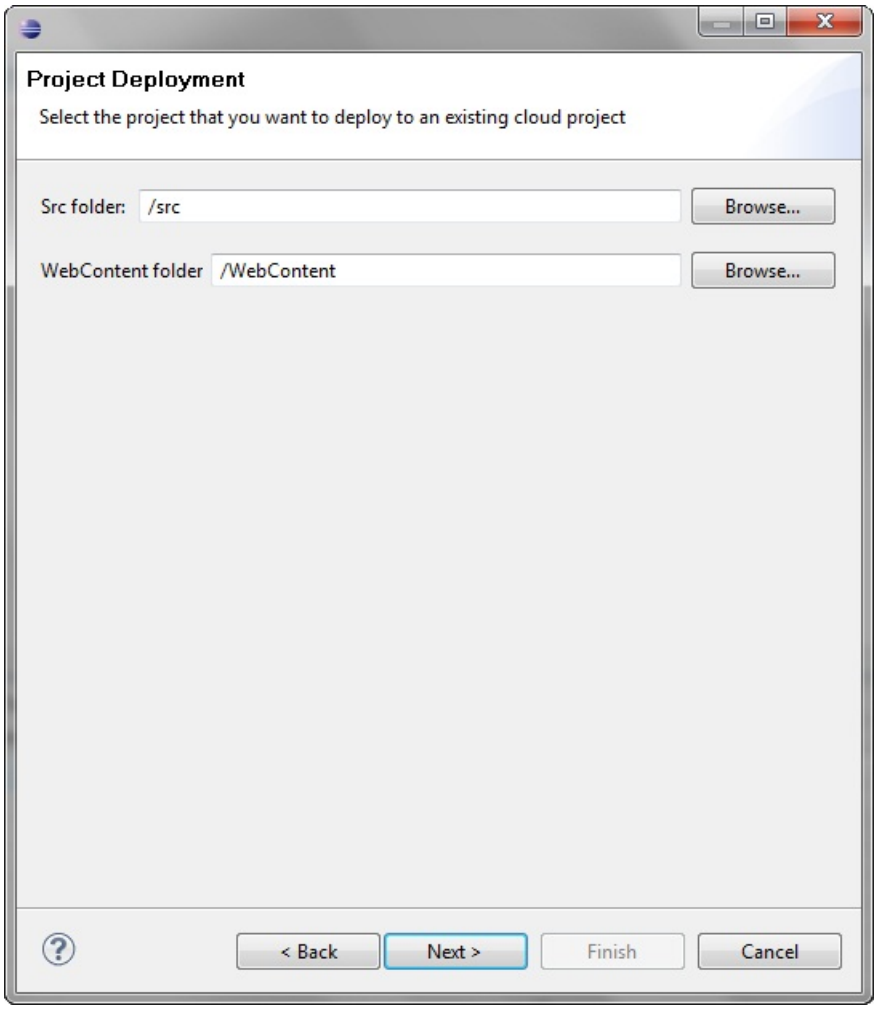


Figura 5.7: SourceCode page

Dopo questa prima fase, in cui si richiedono i percorsi relativi al progetto sorgente, la procedura guidata passa alla fase di scelta della piattaforma e configurazione dei servizi.

Configurazione dei servizi

La procedura di configurazione dei servizi inizia richiedendo su quale piattaforma Cloud si voglia effettuare il deployment.

A seconda del Cloud scelto, vengono richieste configurazioni diverse per ogni servizio. Ogni servizio ha la possibilità di essere attivato e disattivato attraverso la spunta o meno della relativa Checkbox.

Configurazioni Azure

In Figura 5.8 è riportata l'interfaccia grafica relativa alla selezione della piattaforma di Windows Azure, ed all'abilitazione di tutti i servizi.

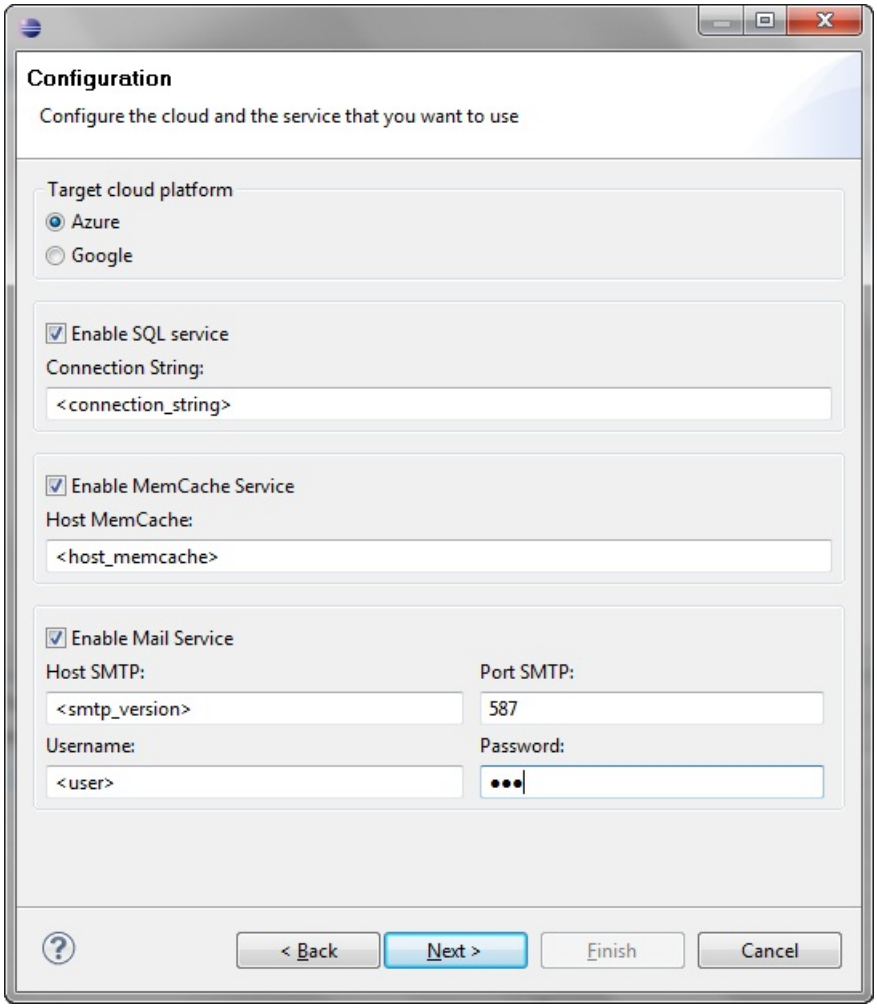


Figura 5.8: Configurazioni Azure

Nello specifico, per la piattaforma Azure, vengono richieste le seguenti informazioni:

- Servizio SQL

  Viene richiesto di specificare la connection_string utilizzata per connettersi al servizio SQL Database, attraverso JDBC. Tale stringa è reperibile direttamente dal portale di Windows Azure nella sezione relativa al servizio SQL.

- Servizio Memcache

  L'informazione necessaria consiste nell'indirizzo dell'host su cui è configurato il servizio di Caching, riportato nella Sezione 3.2 dedicata a Azure. In particolare si distinguono due pattern:

  - Testing in locale: *127.0.0.1:11211*
  - Deployment su Cloud: *localhost_<WorkerRoleName>:11211*

- Servizio Mail

  Prevede la specifica dell'indirizzo de della porta del server SMTP, oltre alle credenziali di accesso.

- Servizio NoSQL

  Configurabile attraverso la pagina successiva, riportata in Figura 5.9. La configurazione di tale servizio è obbligatoria, in quanto prevede la specifica delle credenziali d'accesso al servizio di storage, utilizzate anche dai servizi Queue e dal servizio Blob. Le informazioni specificate saranno utilizzate per creare il file *persistence.xml* utilizzato dalla JPA per l'interfacciamento con il servizio.

  Questa pagina prevede la specifica del nome della persistence-unit. Inoltre l'utente deve specificare dove vuole eseguire l'applicazione: in locale o sul Cloud. Nel caso venisse selezionata l'opzione ˜Cloud˝, l'utente deve specificare anche le credenziali relative all'account storage che intende utilizzare. Le credenziali sono reperibili direttamente dalla piattaforma di Azure, nella sezione relativa agli storage account.

Configurazioni Google

In Figura 5.10 è riportata l'interfaccia grafica relativa alla selezione della piattaforma di Google App Engine e dei relativi servizi.

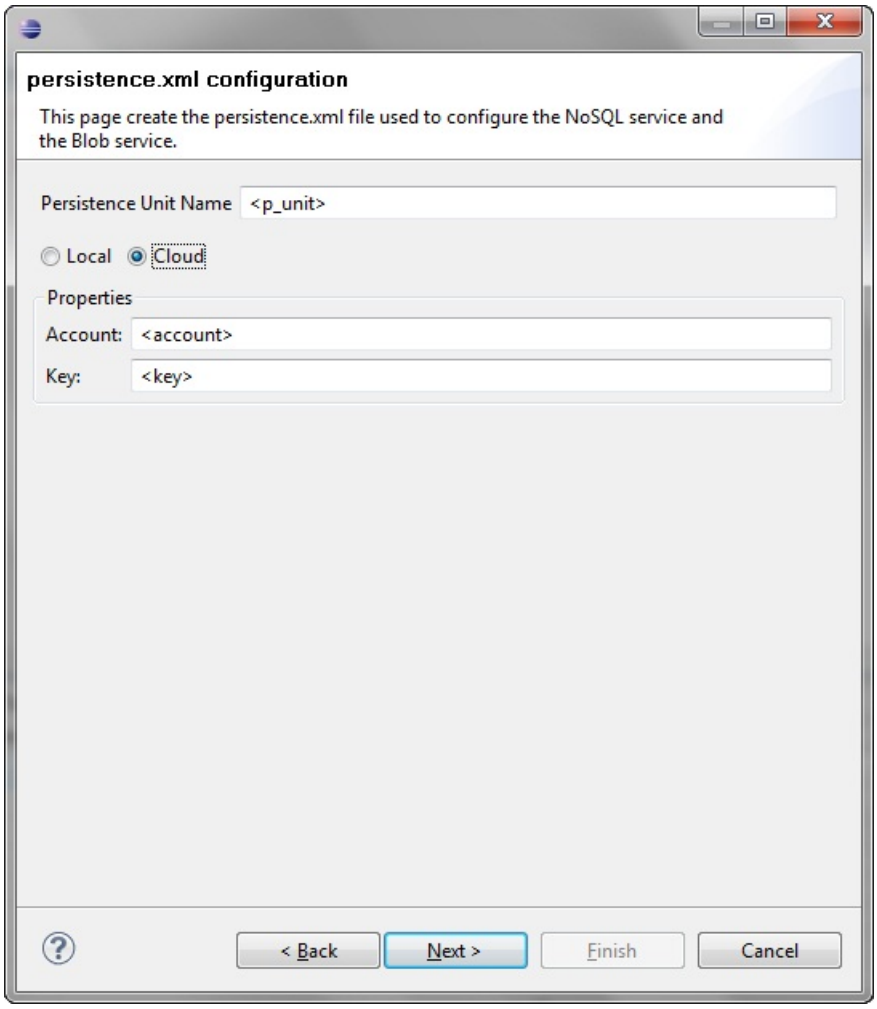


Figura 5.9: Configurazioni JPA per Azure

Nello specifico, per la piattaforma Google App Engine, le configurazioni richieste sono:

- Servizio SQL

  Viene richiesto di specificare la connection_string utilizzata per connettersi a Google Cloud SQL, attraverso JDBC, avente il seguente pattern:

  - Testing in locale: *jdbc:mysql://localhost:3306/yourdatabase?user=username&password=password*
  - Deployment sul Cloud: *jdbc:google:rdbms://instance_name/database*

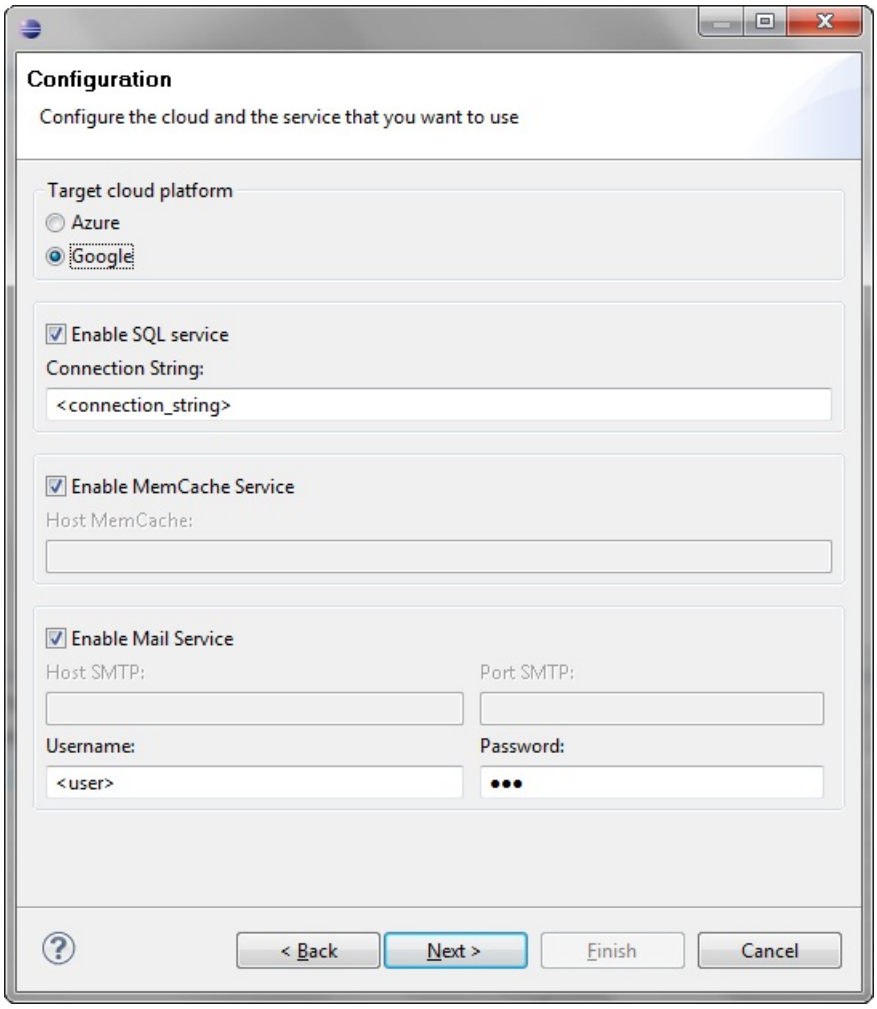


Figura 5.10: Configurazioni Google

- Servizio Memcache

  Non è richiesta alcuna configurazione aggiuntiva oltre all'abilitazione del servizio

- Servizio Mail

  Prevede la specifica delle credenziali relative all'indirizzo con cui si vuole mandare mail. Il server SMTP non è richiesto.

- Servizio NoSQL

  Configurabile attraverso la pagina successiva, riportata in Figura 5.11.

  Questa pagina prevede la specifica del nome della pesistence-unit ed alcune proprietà. Le informazioni specificate saranno utilizzate per creare

il file *persistence.xml* utilizzato dalla JPA per l'interfacciamento con il servizio.

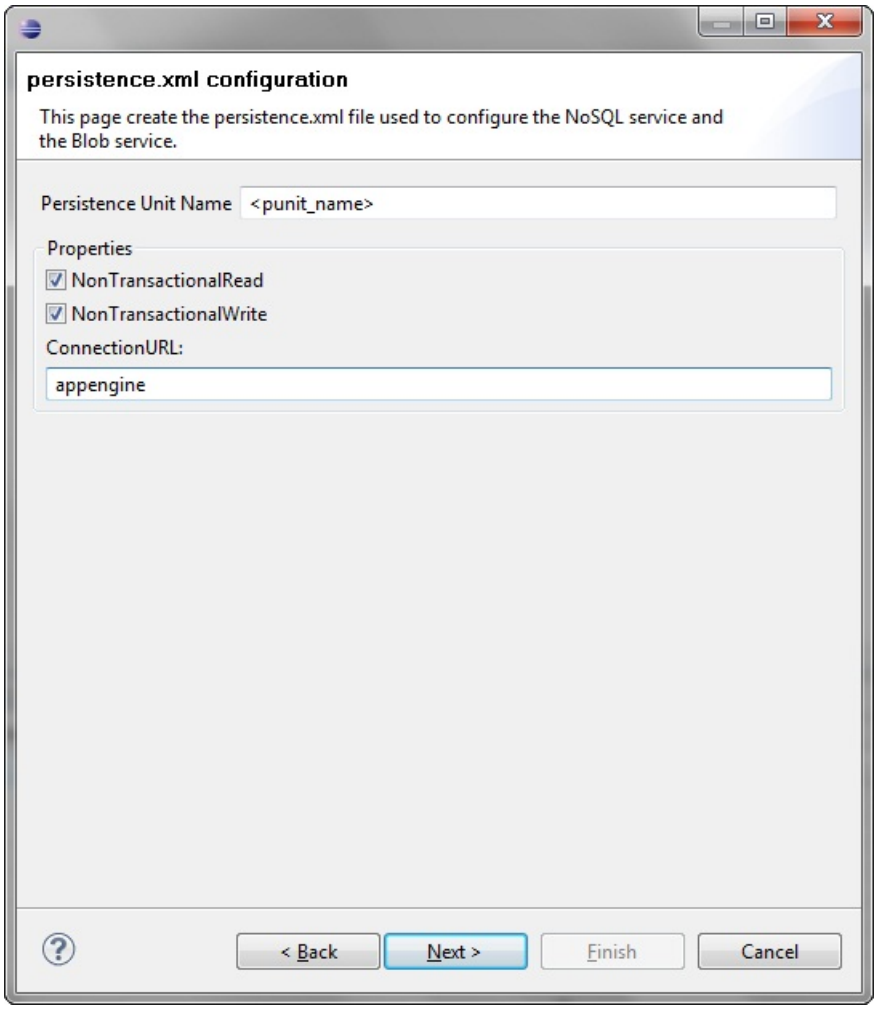


Figura 5.11: Configurazioni JPA per Google

Configurazione delle code

La configurazione delle code è indipendente dalla piattaforma scelta. L'interfaccia grafica relativa a tale funzionalità è riportata in Figura 5.12.

Nella pagina è presente il tasto di spunta per abilitare il servizio. Una volta abilitato il servizio, viene data la possibilità di aggiungere le code utilizzate nell'applicazione da deployare.

L'aggiunta di una coda comporta l'aggiunta di una riga nella tabella riepilogativa, in cui vengono elencate tutte le code impostate, visualizzando: il

nome, il tipo ed eventualmente, solo per code di tipo Task, il tasso di consumo dei task. Oltre ad aggiungere una coda, la pagina permette di eliminarne una esistente, selezionandola e premendo il tasto ¨Remove˝.

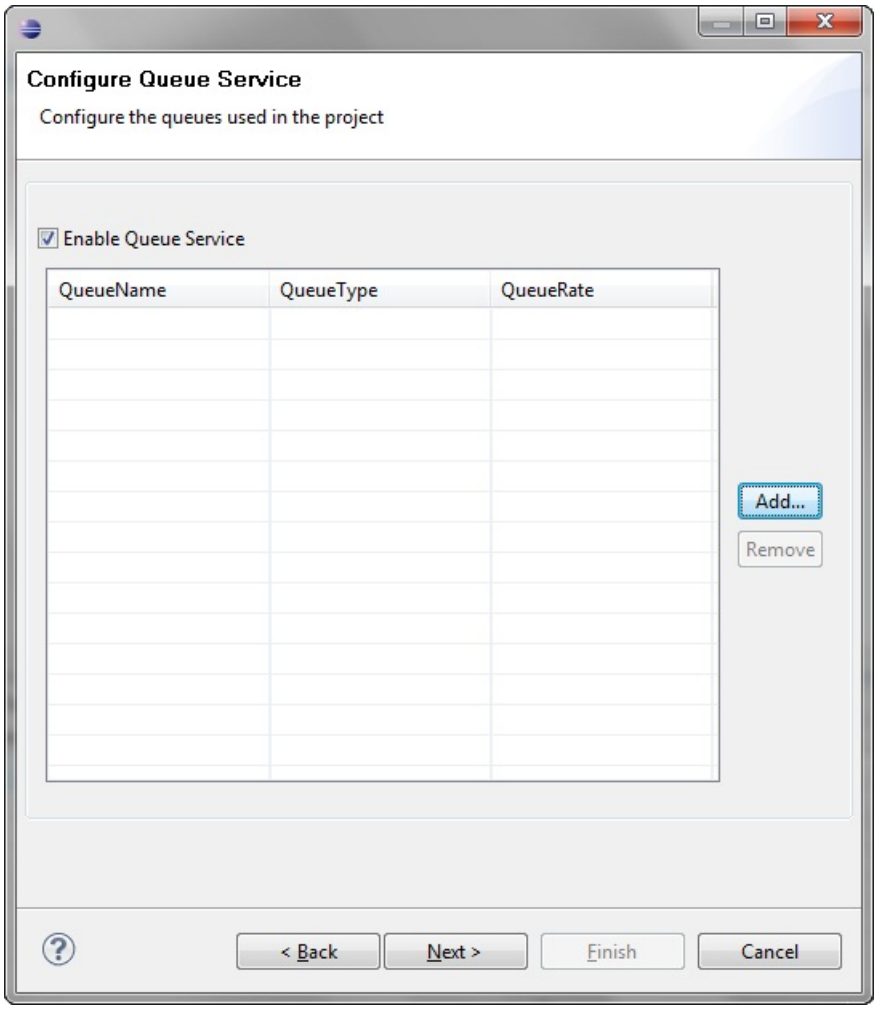


Figura 5.12: Configurazione code

La pressione del tasto ¨Add˝visualizza a video un schermata di dialogo, in cui è possibile specificare le informazioni relative alla coda. Tale finestra riportata in Figura 5.13 richiede l'immissione del nome della coda da creare, il tipo di coda e, nel caso di Task Queue, l'immissione del rate di consumo e l'unità di misura da attribuire ad esso. In particolare, il nome della coda deve essere costituito, come riportato dall'intestazione del campo, da sole lettere minuscole, inoltre, deve essere univoco rispetto alle code già create.

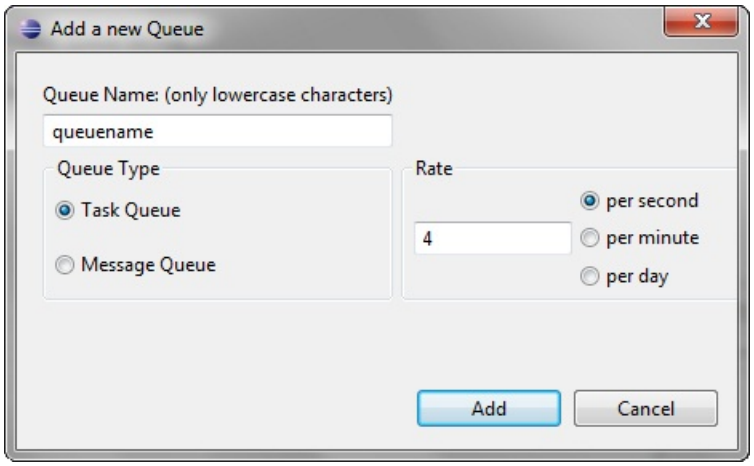


Figura 5.13: Aggiunta di una coda

Configurazione del deployment

Dopo aver configurato tutti i servizi, non resta che specificare il percorso del progetto in cui deployare l'applicazione e fornire le informazioni necessarie per la configurazione del deployment. Tale progetto permetterà all'utente di effettuare il deploy direttamente sulla piattaforma, oppure testare l'applicazione in locale attraverso le modalità previste dai plugin delle rispettive piattaforme.

Deployment su Azure

L'interfaccia grafica della pagina di deployment per Windows Azure è riportata in Figura 5.14.

In questa fase, l'utente deve effettuare i seguenti passaggi:

1. Specificare il percorso del progetto di tipo Windows Azure Deployment Project in cui deployare l'applicazione. Il plugin controlla se il percorso specificato è relativo ad un progetto del tipo precedentemente indicato e, in caso contrario, il percorso non viene accettato. Mentre in caso positivo, il plugin controlla la presenza di Worker Role e ne inserisce la referenza nelle Combobox successive.

2. Selezionare la Worker Role in cui deployare l'applicazione. Scegliendola tra quelle presenti nella lista della Combobox.

3. Selezionare l'ambiente di runtime su cui si intende eseguire l'applicazione. Il runtime Cloud viene abilitato solo se le credenziali utilizzate nel

servizio NoSQL sono di tipo "Cloud".

Se l'ambiente di runtime scelto fosse il Cloud e, tra le code configurate, ce ne fosse almeno una di tipo Task Queue, l'utente avrebbe la possibilità di scegliere di deployare l'applicativo su due livelli distinti, suddividendo il backend dal frontend. In tal caso, l'applicazione viene deployata su due Worker Role, con la differenza che la Worker Role dedicata al frontend è accessibile pubblicamente tramite chiamate HTTP, mentre la Worker Role dedicata al backend non è accessibile dall'esterno, ma viene riservata per l'esecuzione asincrona dei task presenti nelle code, evitando il rallentamento del servizio dovuto dal sovraccaricamento del frontend.

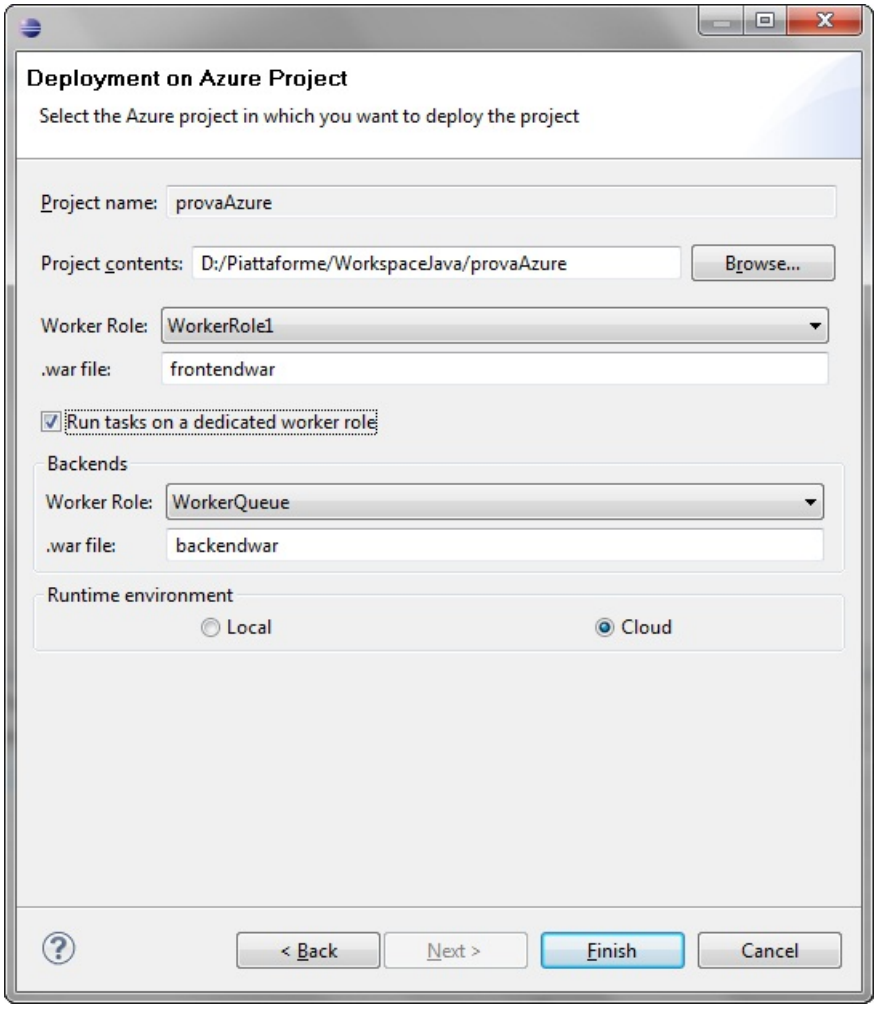


Figura 5.14: Deployment Azure

Per il deployment su due livelli, occorre specificare oltre alla prima Worker Role, dedicata al frontend, la Worker Role in cui deployare il backend. Tale

Role deve essere differente rispetto a quella specificata per il frontend.

Deployment su Google

L'interfaccia grafica della pagina di deployment per Google App Engine è riportata in Figura 5.15. Tale pagina prevede un processo simile alla specifica del progetto sorgente richiesto nella fase iniziale del plugin.

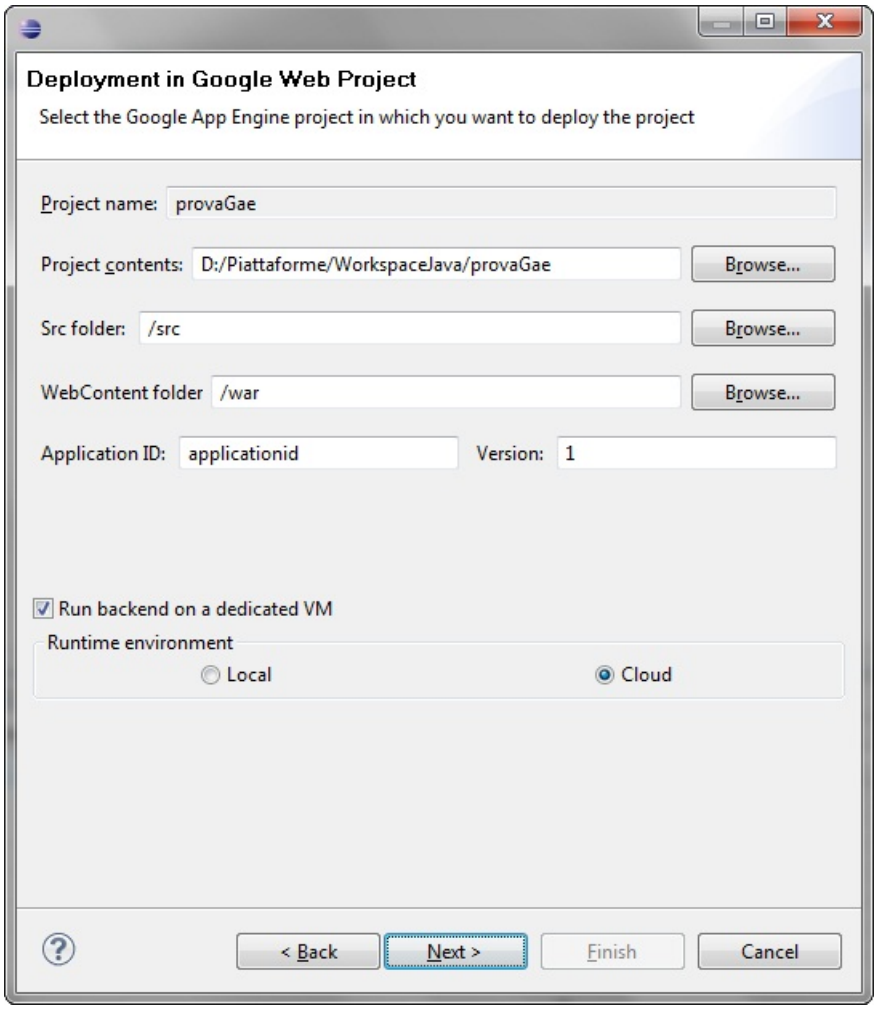


Figura 5.15: Deployment Google

In questa fase, l'utente deve effettuare i seguenti passaggi:

1. Specificare il percorso del progetto di tipo Google Web Application in cui deployare l'applicazione. Il plugin controlla se il percorso specificato è relativo ad un progetto del tipo precedentemente indicato e, in caso contrario, il percorso non viene accettato.

2. Specificare il percorso della cartella in cui copiare le sorgenti Java del progetto di partenza e quello relativo alla cartella in cui copiare il contenuto web. Il plugin controlla che i percorsi indicati siano relativi a cartelle contenute nel progetto precedentemente specificato. In caso negativo i percorsi non vengono accettati.

3. Specificare l'application id e la versione con cui deployare l'applicativo sul Cloud.

4. Selezionare l'ambiente di runtime su cui si intende eseguire l'applicazione.

Se tra le code configurate nella fase precedente, ce ne fosse una di tipo Task Queue e, se l'ambiente di runtime selezionato fosse il Cloud, l'utente avrebbe la possibilità di deployare l'applicativo su due livelli, specificandolo tramite la Checkbox presente nella pagina e spostandosi nella pagina successiva.

La pagina relativa alle configurazioni del livello di backend è riportata in Figura 5.16. Tale pagina è pressochè identica alla precedente.

Nella pagina dedicata al backend, l'utente deve specificare le stesse tipologie di informazioni richieste nella pagina precedente differenziando il progetto di destinazione, che deve essere diverso rispetto a quello dedicato al frontend. L'id dell'applicazione non viene richiesto perchè identico a quello di frontend, mentre viene richiesta la versione che deve essere obbligatoriamente differente a quella assegnata al frontend.

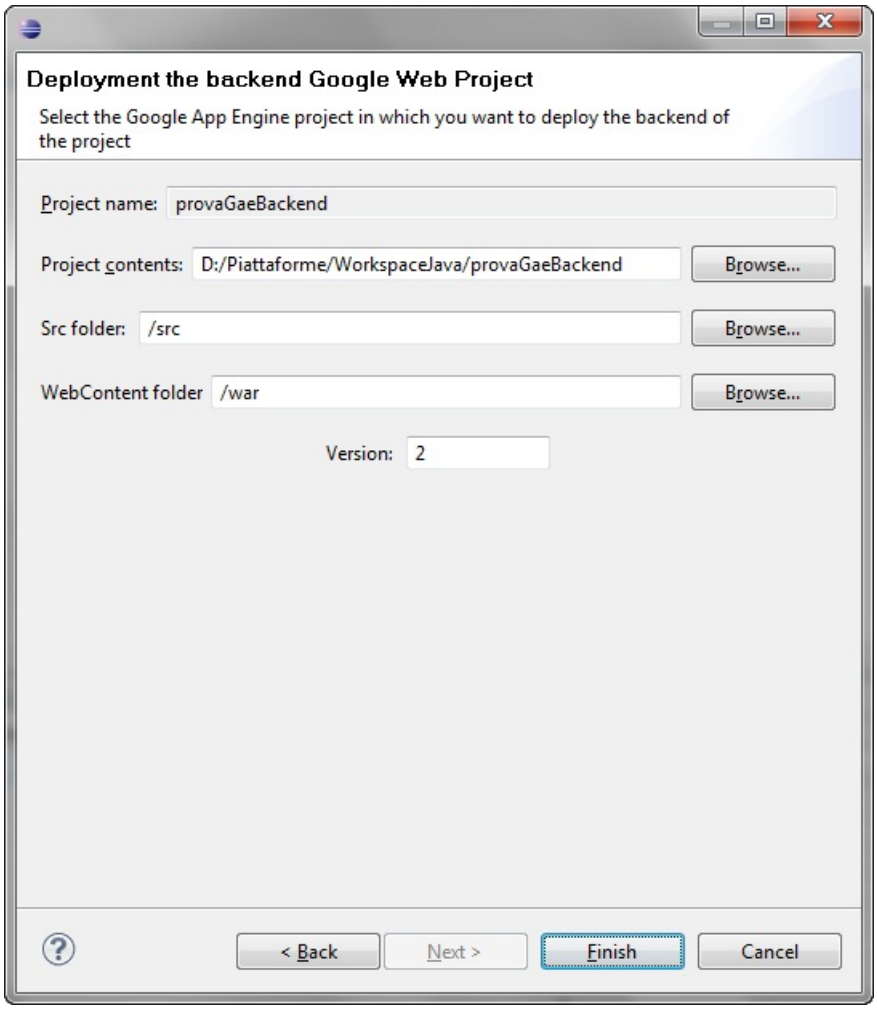


Figura 5.16: Deployment Backend Google

Capitolo 6

Applicazione MiC: Meeting In the Cloud

Per validare il nostro approccio e valutare anche dal punto di vista delle prestazioni le funzionalità della libreria CPIM, si è deciso di progettare ed implementare un'applicazione di esempio che utilizzi tutti i servizi supportati e che sarà considerata nel capitolo come caso di studio.

Il capitolo che segue è costituito da una parte introduttiva (Sezione 6.1) che descrive, ad alto livello, le funzionalità offerte da questa applicazione, seguita da sezioni che ne descrivono le tecnologie usate per la sua implementazione (Sezione 6.2), il design funzionale attraverso diagrammi UML (Sezione 6.3) e la sua architettura, dal punto di vista di componenti software, di deploy e di livello dati (Sezione 6.4).

6.1 Descrizione funzionalità

L'applicazione MiC è un'applicazione di Social Networking. MiC permette ad un utente, durante la fase di registrazione al sito, di profilare i suoi interessi, scegliendo degli argomenti a cui risulta particolarmente interessato. Per ciascuna delle tematiche scelte viene richiesto di rispondere a delle domande, utilizzando una scala di valutazione 1-5. A questo punto MiC calcola in modo asincrono, quali utenti, tra quelli registrati, risultano essere i più affini all'utente per le preferenze espresse.

Terminata questa fase di registrazione e profilazione, l'utente può accedere al portale. Nella sua Homepage, può interagire con i suoi ˝Best Contacts˝ scrivendo e leggendo messaggi relativi ai topic selezionati.

6.2 Tecnologie usate per l'implementazione

L'applicazione è stata scritta in linguaggio Java, utilizzando pagine JSP insieme ad AJAX per lo strato di presentazione e Servlet per il relativo livello applicativo.

Sia le pagine JSP che le Servlet utilizzano la libreria CPIM per l'accesso ai seguenti servizi Cloud:

- Blob Service
- NoSQL Service (utilizzando l'intermediazione delle JPA)
- SQL Service
- Task Queue Service (attraverso alcune modifiche è comunque possibile utilizzare il Message Queue Service)
- Memcache Service

6.3 Design funzionale di MiC

In questa sezione viene descritta l'architettura funzionale di MiC, attraverso l'uso di diagrammi UML.

6.3.1 Attori

Gli attori che interagiscono con l'applicativo sono essenzialmente due:

- Guest
- MiC User, cioè un utente correttamente autenticato.

Il primo attore, può solamente richiedere la registrazione, compilando i relativi form e fornendo la propria profilazione, cioè le proprie preferenze sugli argomenti disponibili e rispondendo ai relativi quesiti.

L'utente autenticato, invece, ha la possibilità di scrivere dei messaggi relativi ad uno o più argomenti, che sono leggibili da tutti coloro che hanno tale utente nella propria lista dei "Best Contacts" per quel dato argomento. A sua volta l'utente può leggere tutti i messaggi scritti dagli utenti con un profilo simile al suo. Inoltre, l'utente ha la possibilità di richiedere l'aggiornamento delle proprie liste di contatti preferiti o modificare la propria profilazione, ricompiendo le fasi di selezione degli argomenti e di risposta alle domande relative alle preferenze.

6.3.2 Use Case

Gli Use Case relativi ad entrambe le tipologie di attori sono mostrati in Figura 6.1.

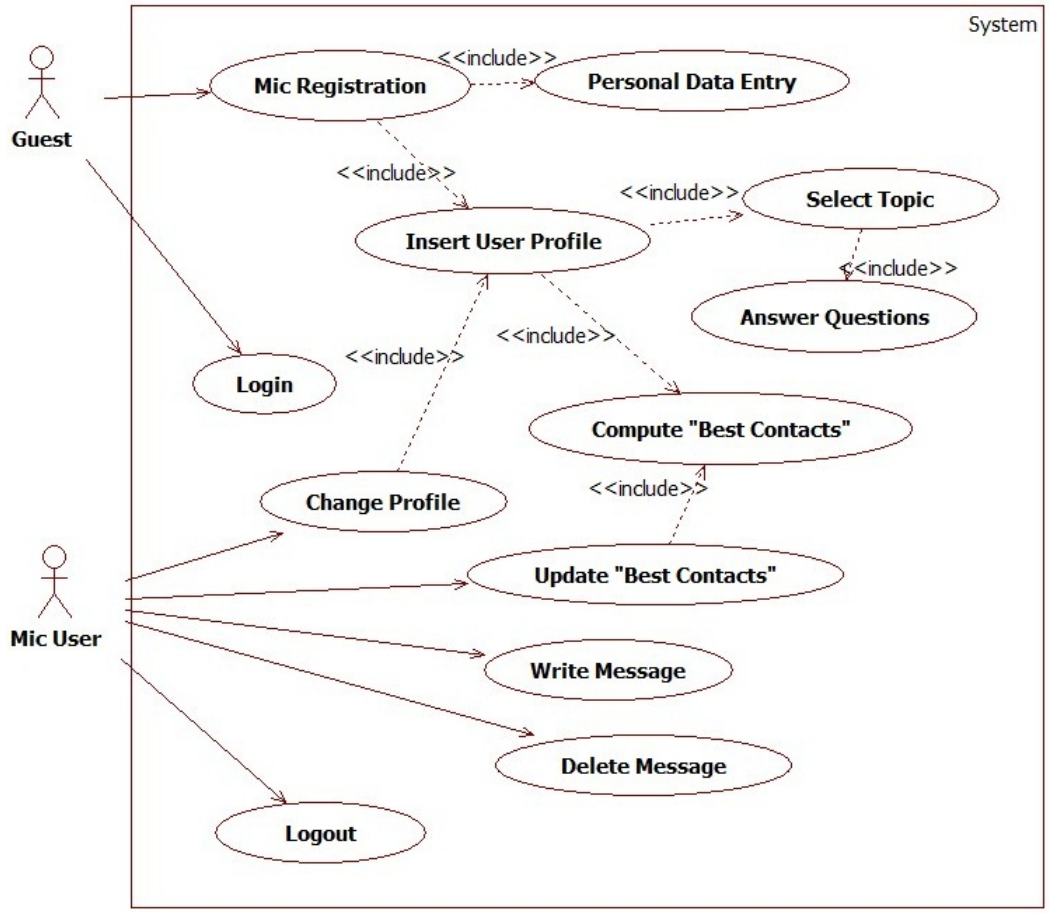


Figura 6.1: Use Case Diagram

Relativamente all'attore Guest, si sono identificati i seguenti Use Case:

- Registrazione a MiC: questo Use Case include due ulteriori Use Case, che rappresentano le due fasi che compongono la registrazione all'applicativo:
  - Inserimento Dati Personali: tramite apposite form l'utente fornisce alcuni suoi dati anagrafici e una sua foto profilo.
  - Inserimento Profilazione: all'utente è proposta una lista di sette argomenti che saranno oggetto di discussioni, scambio di idee all'interno di MiC. L'attuale versione propone le seguenti tematiche:
    - Amore
    - Cinema
    - Lettura
    - Musica
    - Politica
    - Sport
    - Tecnologia

    L'utente deve scegliere a quali di questi argomenti è interessato.
    Per ogni topic selezionato, sono proposte alcune domande a cui l'utente deve rispondere tramite una scala di valori 1-5.
    Il sistema calcola, per ciascun argomento, una lista dei 3 contatti potenzialmente più simili all'utente.
    Per produrre questo risultato, MiC effettua il calcolo della similarità (utilizzando il coefficiente di Pearson [40]) tra le risposte date alle domande relative ad un topic dall'utente e quelle di tutti gli altri utenti.

- Login: questo caso d'uso fa riferimento all'invio delle credenziali per la richiesta di accesso al sistema, da parte di un utente registrato.

Relativamente all'attore MiC User, cioè ad un utente autenticato, gli Use Case possibili sono:

- Modifica profilazione: l'utente può essenzialmente rifare la propria profilazione, modificando la lista dei propri argomenti preferiti e le relative risposte.

- Aggiornamento "Best Contacts": l'utente può richiedere di aggiornare la propria lista di contatti, in questo modo saranno inclusi nella computazione tutti gli utenti che si sono registrati dopo la registrazione dell'utente o gli aggiornamenti effettuati in precedenza.
- Scrittura di un messaggio: l'utente può scrivere un messaggio, che sarà leggibile da tutti coloro che lo hanno nella propria lista di simili.
- Cancellazione di un messaggio: l'utente può cancellare un messaggio da lui precedentemente scritto.

6.3.3 Class Diagram

In Figura 6.2 è mostrato il Class Diagram delle macro componenti costituenti MiC. Si possono dividere le classi presenti in tre gruppi:

- MiC: strato di presentazione dell'applicazione. Gestisce le interazioni con gli utenti.
- Servizi Cloud: insieme delle componenti che rappresentano i servizi Cloud utilizzati dall'applicativo. MiC utilizza i seguenti servizi:
  - Blob Service: per il salvataggio della foto profilo degli utenti.
  - NoSQL Service: sono memorizzati con questa soluzione di storage i topic, le domande associate e tutte le risposte date dagli utenti ad esse.
  - SQL Service: nel database relazionale vengono memorizzate le anagrafiche degli utenti, nonché tutte le tuple necessarie a mantenere traccia dei "Best Contacts" di ogni utente.
  - Memcache Service: il servizio di Memcache, permette all'applicazione di salvare alcuni dati durante le normali fasi di navigazione di un utente. Nello specifico vengono salvati i dati relativi alle associazioni tra un utente loggato e i suoi "Best Contacts", per evitare di doverli sempre richiedere ai classici sistemi di storage. Utilizzando tale tipologia di memoria, la velocità di accesso a queste informazioni risulta notevolmente migliorata.

  - Task Queue Service: utilizzato per registrare le richieste di computazione delle liste dei simili. Il Consumer relativo a questo servizio è rappresentato dalla classe TaskQueue Handler, che ha il compito di leggere le richieste, invocare il task di computazione associato e cancellarlo dalla coda.

- Livello Dati: componenti che costituiscono la base informativa dell'applicativo, rappresentata dalle classi:

  - UserRatings: contiene le risposte di un utente alle domande relative ad un particolare topic.
  - Topic: rappresenta un argomento di possibile discussione in MiC e contiene le domande ad esso associate.
  - Message: messaggio scritto da un utente, inerentemente ad un dato topic.
  - UserProfile: contiene i dati anagrafici di un utente.
  - UserSimilarity: contiene la lista dei tre contatti più simili per un dato utente relativamente ad un certo argomento.

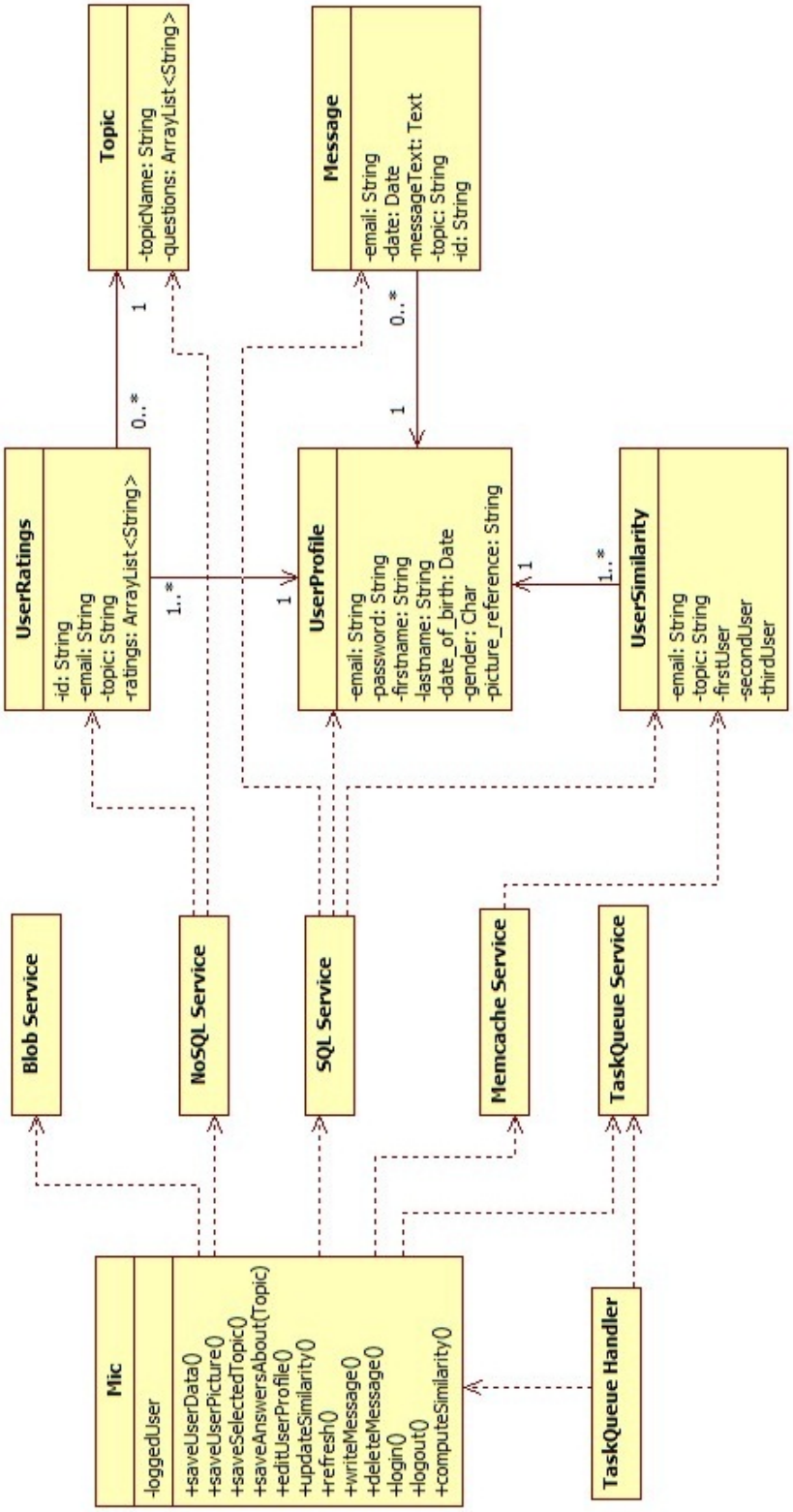


Figura 6.2: Class Diagram

6.3.4 Sequence Diagram

Per esprimere come gli Use Case identificati per MiC interagiscono tra di loro, con le richieste esterne di un utente e quali servizi Cloud utilizzano, si forniscono i relativi Sequence Diagram. In particolare, per ognuno di questi, si descrive quali servizi Cloud utilizza.

Registrazione a MiC

Il Sequence Diagram relativo alla Registrazione a MiC è mostrato in Figura 6.3. Questo Use Case utilizza i seguenti servizi Cloud:

- Servizio SQL: vengono memorizzati nella tabella UserProfile, i dati relativi all'utente inseriti nella prima fase della registrazione, mentre nella tabella UserSimilarity, le associazioni tra tale utente e i suoi simili, calcolati dall'applicazione. A tale scopo si inserisce una tupla per ogni argomento di interesse selezionato, contenente la liste dei tre utenti più simili.

- Servizio NoSQL: si utilizza tale servizio per memorizzare le risposte relative a ciascun argomento selezionato, all'interno dell'entità UserRatings. Queste risposte sono poi utilizzate dal sistema per il calcolo della similarità dei profili.

- Servizio Blob: si utilizza tale servizio per memorizzare all'interno di un apposito Blob, il file contenente la foto profilo dell'utente.

- Servizio Task Queue: si utilizza questo servizio per richiedere l'esecuzione del calcolo delle varie liste dei ˜Best Contacts˝. Nello specifico si inserisce nella coda offerta dal provider, una richiesta contenente le seguenti informazioni, utili per l'esecuzione del calcolo della similarità:

  – Path della Servlet adibita al calcolo della similarità

  – E-mail dell'utente richiedente

  Il Task Handler della coda, periodicamente, legge le richieste dalla coda ed effettua una richiesta HTTP di tipo POST alla Servlet contenente la logica di calcolo, passandogli come parametro l'e-mail dell'utente richiedente.

Modifica profilazione

Il Sequence Diagram è mostrato in Figura 6.4. Come si evince dal diagramma, i servizi Cloud utilizzati da questo Use Case sono:

- Servizio SQL: si utilizza questo servizio per aggiornare le tuple della tabella UserSimilarity relative ai dati delle associazioni tra tale utente e i suoi simili, dopo la richiesta di modifica.
- Servizio NoSQL: si utilizza tale servizio per aggiornare le entità UserRatings dell'utente, relative a ciascun argomento selezionato, a seguito della richiesta di modifica. Inoltre il servizio viene utilizzato per richiedere le risposte dell'utente e degli altri user per il nuovo calcolo della similarità.
- Servizio Task Queue: si utilizza questo servizio per richiedere l'esecuzione del calcolo delle varie liste dei ¨Best Contacts˝, a seguito della richiesta di modifica.
- Servizio Memcache: si richiama questo servizio per cancellare le informazioni relative all'utente memorizzate nella cache precedentemente alla richiesta di modifica del profilo.
  Tali informazioni, se non cancellate, creerebbero inconsistenza tra la basi dati e la cache.

Aggiornamento ¨Best Contacts˝

Il diagramma è illustrato in Figura 6.5 e mostra come questo Use Case utilizza i seguenti servizi Cloud:

- Servizio SQL: si utilizza questo servizio per aggiornare i dati relativi all'associazione tra tale utente e i suoi simili, dopo la richiesta di aggiornamento.
- Servizio NoSQL: si utilizza tale servizio per richiedere le risposte dell'utente e degli altri user per il nuovo calcolo della similarità.
- Servizio Task Queue: si utilizza questo servizio per richiedere l'aggiornamento delle varie liste dei ¨Best Contacs˝.

- Servizio Memcache: si richiama questo servizio per aggiornare le informazioni in cache relative all'utente, a seguito della richiesta di aggiornamento. Tali informazioni, se non aggiornate correttamente, creerebbero inconsistenza tra la basi dati e la cache.

Login-Scrittura Messaggio-Cancellazione Messaggio-Logout

Vista la semplicità di questi Use Case, si fornisce un'unico Sequence Diagram (Figura 6.6) che descrivere lo scenario in cui un utente effettua il login, scrive un messaggio, cancella un messaggio ed effettua il logout dal sistema. I servizi Cloud utilizzati in questo scenario sono:

- Servizio SQL: per richiedere i dati relativi all'utente e all'associazione tra tale utente e i suoi simili, nonché tutti i messaggi scritti da essi, per visualizzarli nella Homepage dell'utente.
- Servizio Blob: per richiedere i Blob contenenti le foto profilo dell'utente e dei suoi simili, per visualizzarli nella sua Homepage.
- Servizio Memcache: per verificare se sono presenti nella cache le informazioni relative all'utente.

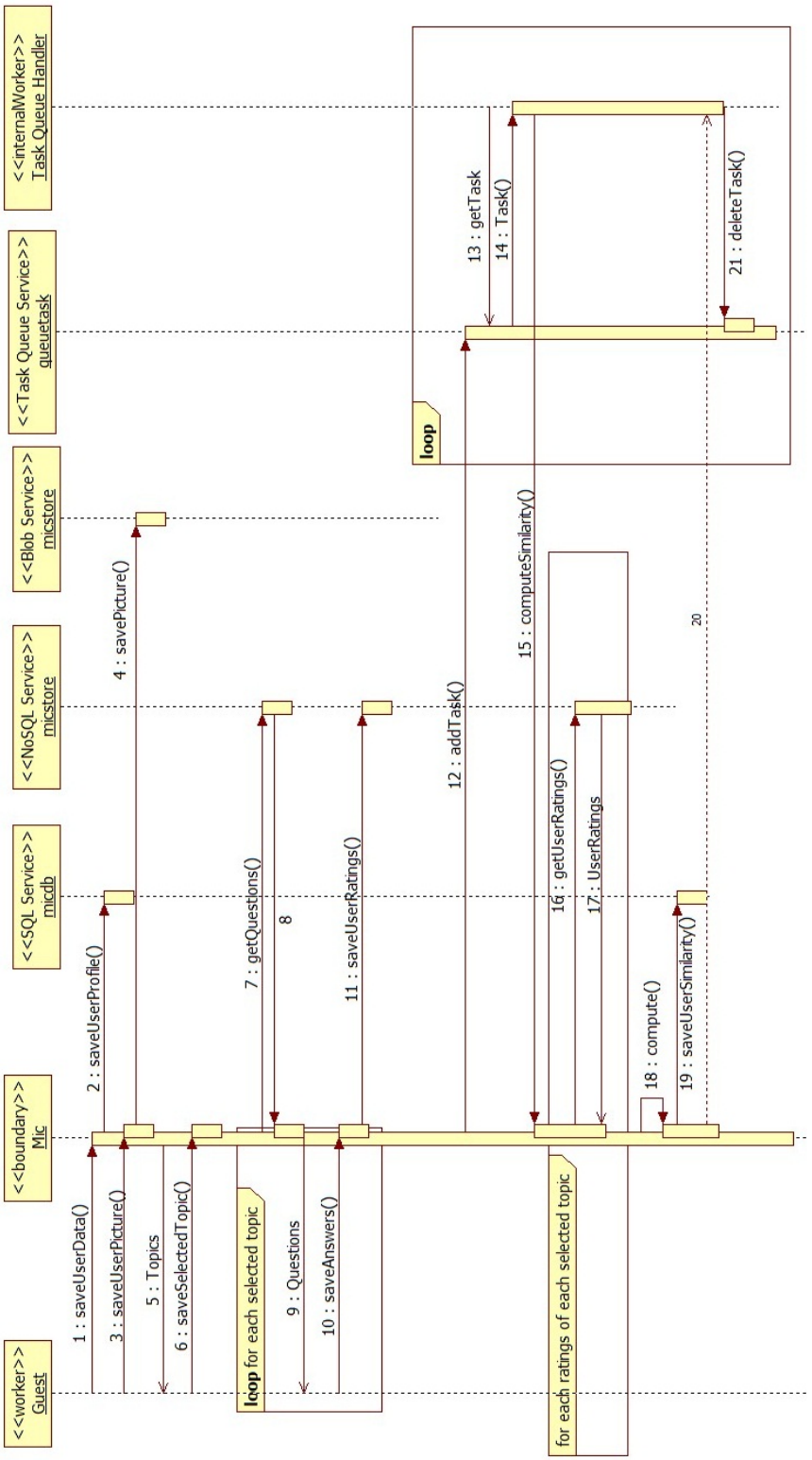


Figura 6.3: Sequence Diagram: Registrazione a MiC

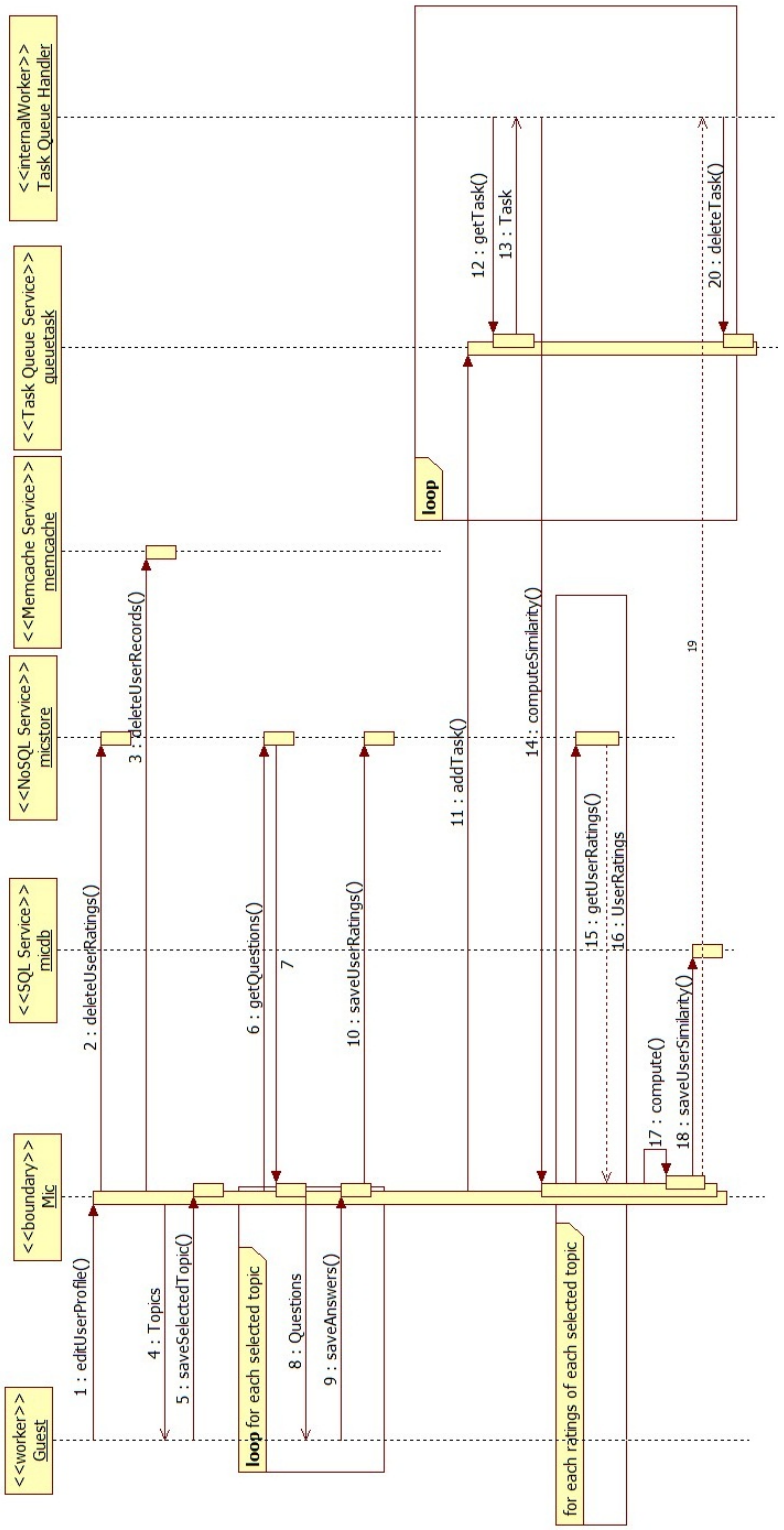


Figura 6.4: Sequence Diagram: Modifica Profilazione

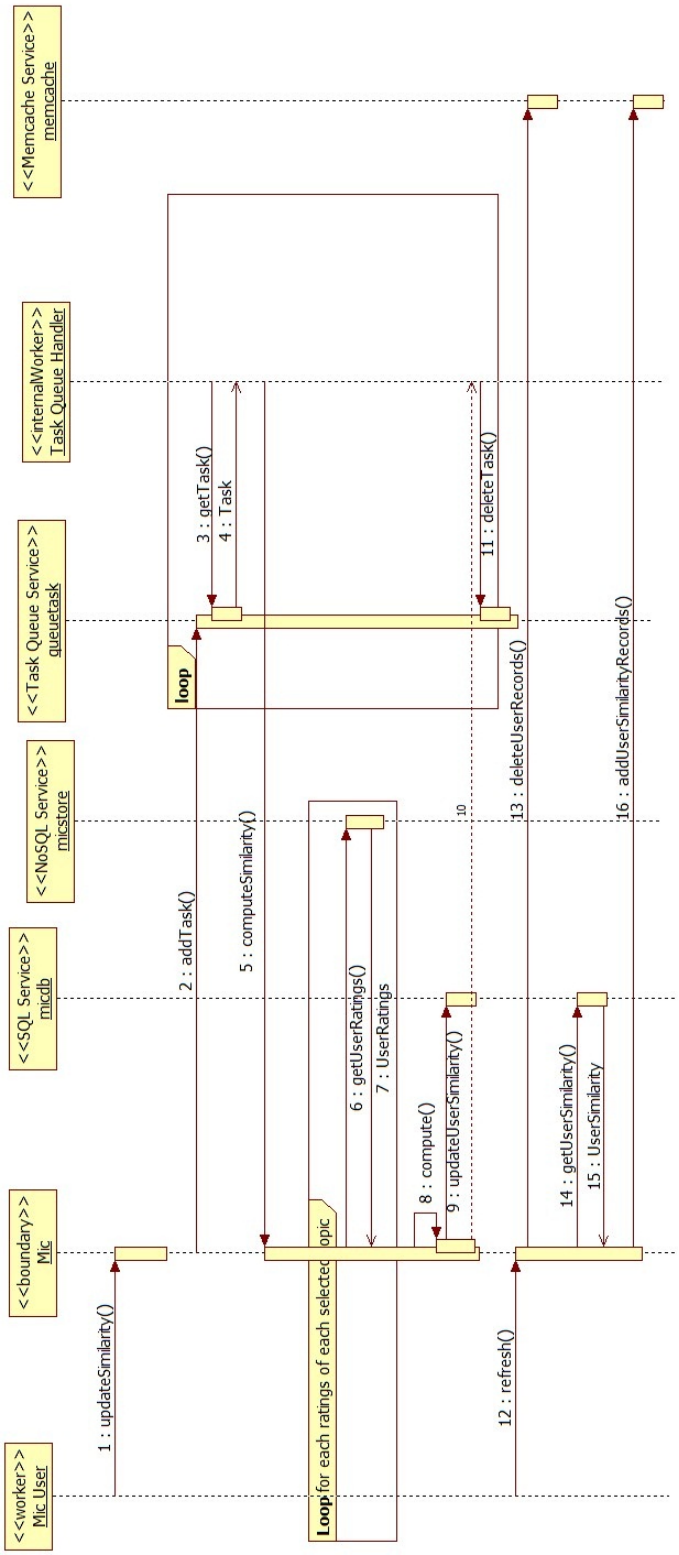


Figura 6.5: Sequence Diagram: Aggiornamento "Best Contacts"

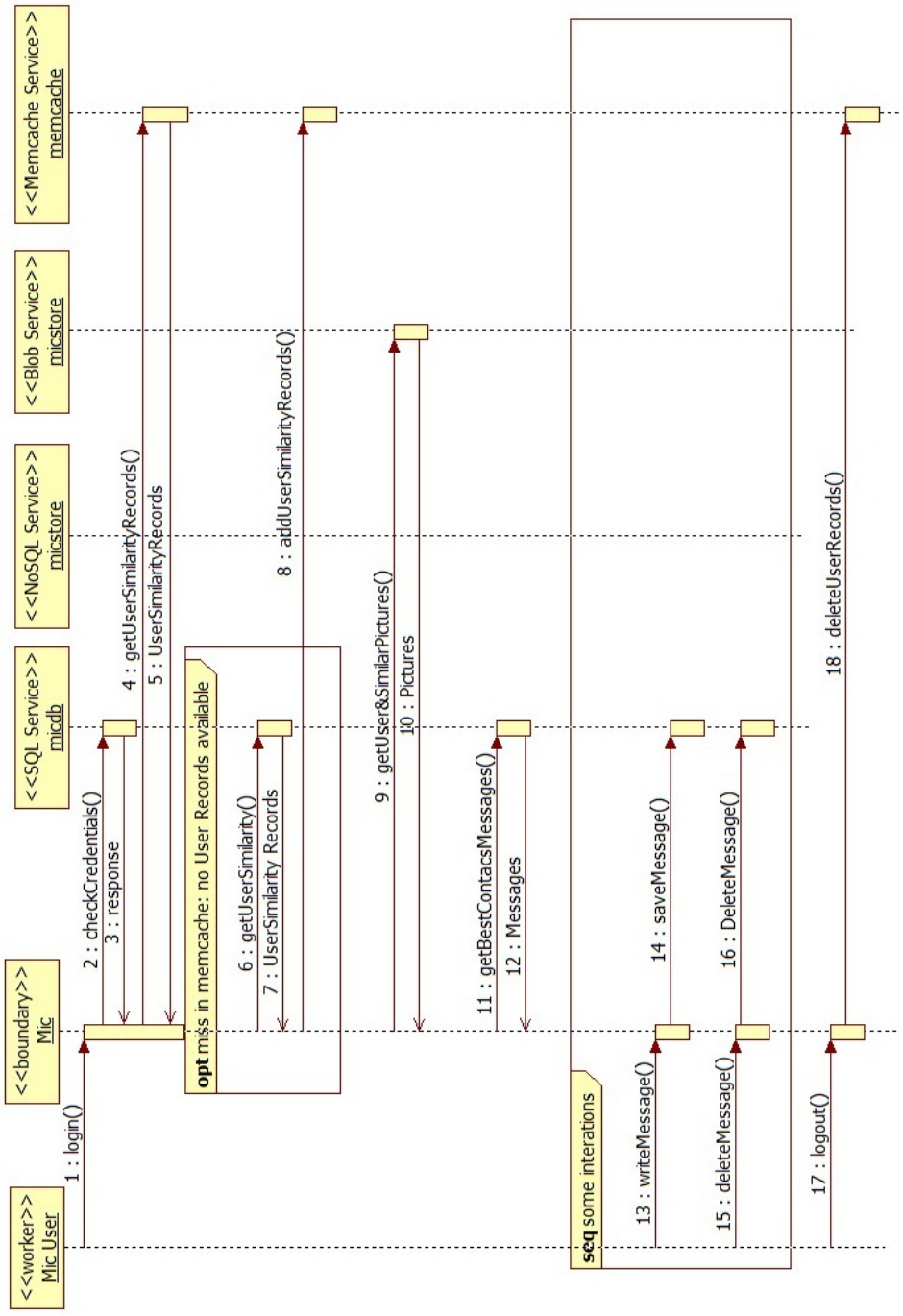


Figura 6.6: Sequence Diagram: scenario Login-ScritturaMessaggio-CancellazioneMessaggio-Logout

6.4 Architettura

L'architettura logica dell'applicazione è quella tipica delle web application implementate servendosi del paradigma JSP-SERVLET. Consiste quindi di un tier di presentazione, che interagisce con le richieste degli utenti dell'applicazione, di uno strato applicativo che le elabora e ne fornisce i contenuti per le relative risposte e di uno strato contenente i dati prodotti da queste interazioni.

Nello specifico di questa applicazione, l'unico lavoro computazionalmente gravoso compiuto, è quello relativo al calcolo della similarità, durante la fase di registrazione e/o modifica del profilo. Questa elaborazione viene svolta in modo asincrono rispetto al flusso di interazioni tra l'utente e MiC, utilizzando il servizio Cloud delle Task Queue.

Gli strati di presentazione e di applicazione possono risiedere sia nella stessa macchina virtuale (vedi Figura 6.7), sia in due separate, soluzione che rende l'applicazione più scalabile, permettendo inoltre di spostare il peso computazionale del calcolo dei "Best Contacts" in una VM dedicata (vedi Figura 6.8).

Infine, utilizzando servizi Cloud di storage, il controllo del tier, relativo alla gestione e scalabilità dei dati applicativi, è delegato al provider scelto.

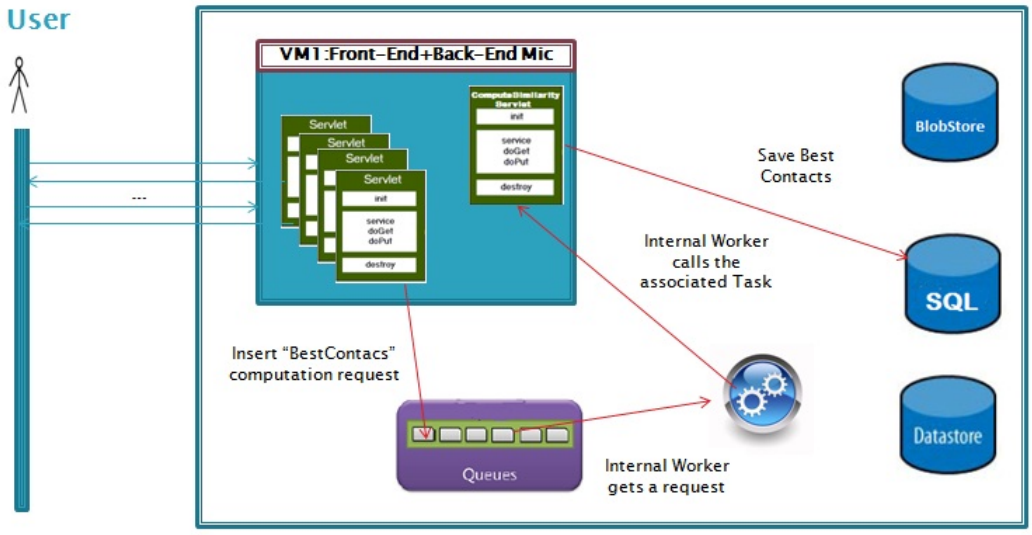


Figura 6.7: Architettura con singola VM

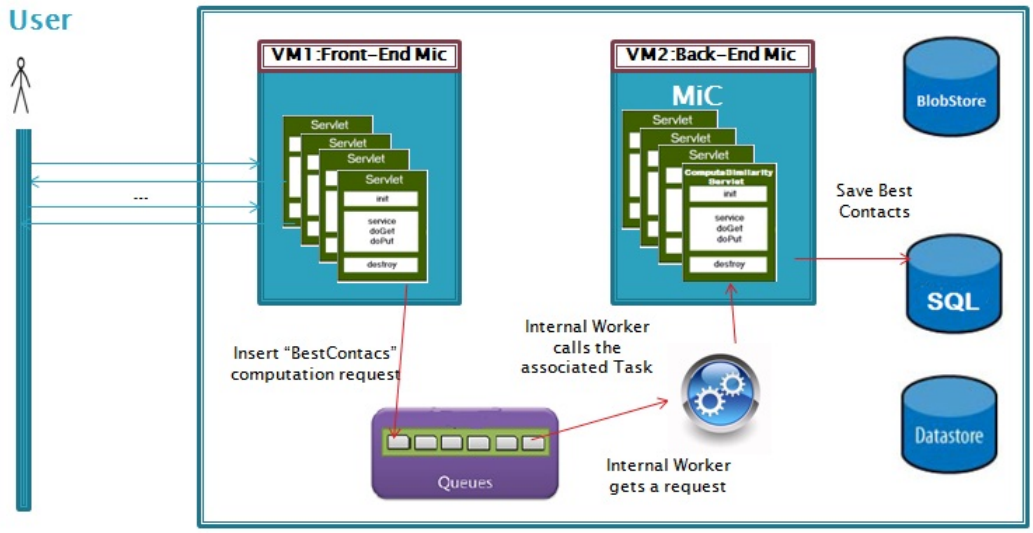


Figura 6.8: Architettura con VM dedicata per calcolo "Best Contacts"

6.4.1 Component Diagram

La struttura delle macro-componenti software che costituiscono MiC dipende dalla scelta del Cloud provider che si desidera utilizzare per il caricamento dell'applicativo. Nel seguito vengono presentati i Component Diagram per i seguenti casi:

- Deploy su Google App Engine (Figura 6.9)
- Deploy su Windows Azure (Figura 6.10)

Deploy su App Engine

Se la scelta del provider ricadesse su Google App Engine, i componenti essenziali del sistema software di MiC sarebbero:

- Web Application Project: questo progetto web contiene:
  - MiC war: war costituito da:
    - Classi Servlet e pagine JSP dell'applicativo, che utilizzano le API della libreria CPIM
    - Configuration.xml: file di configurazione della libreria
    - Libreria CPIM

  - Google App Engine SDK per Java: SDK per lo sviluppo di applicazioni Java per GAE. Include le API Java per l'accesso ai servizi offerti dalla piattaforma
  - Java SQL API: APIs per l'accesso e l'utilizzo di DB relazionali

- GAE Services: servizi offerti dalla piattaforma di Google

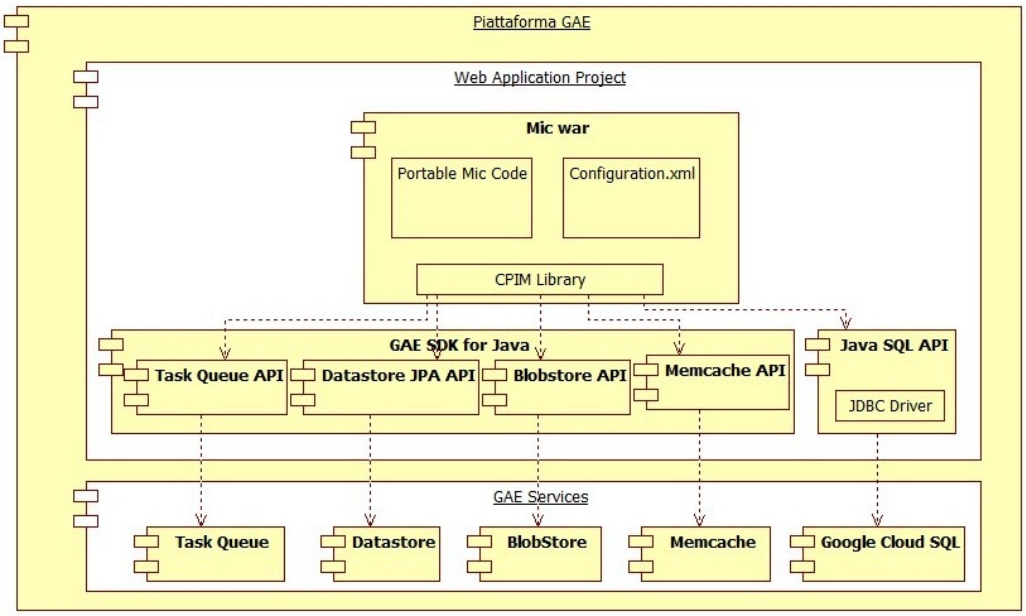


Figura 6.9: Component Diagram: deploy su Google App Engine

Deploy su Azure

Se invece si decidesse di utilizzare la piattaforma di Microsoft, il Component Diagram relativo risulterebbe composto dalle seguenti componenti:

- MiC war: war contenente:
  - Classi Servlet e pagine JSP dell'applicativo, che utilizzano le API della libreria CPIM
  - Configuration.xml: file di configurazione della libreria
  - Libreria CPIM
- Internal Worker war: war contenente il codice relativo al gestore delle code per Azure. Questa componente è fornito insieme alla libreria in caso di deploy in Azure, poichè non nativamente presente nella piattaforma

- jpa4Azure API: API per l'accesso ai servizi Queue, Blob e Table (attraverso l'interfaccia JPA) di Azure
- Java SQL API: API per l'accesso e l'utilizzo di database relazionali
- SpyMemcache client: componente client del servizio Memcache
- Azure Services: servizi della piattaforma Azure

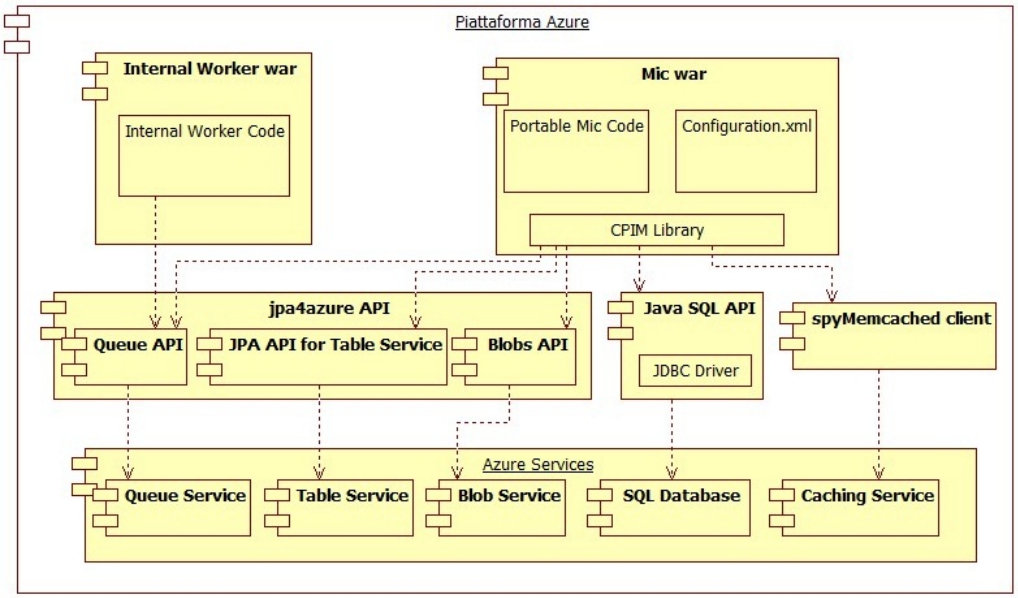


Figura 6.10: Component Diagram: deploy su Azure

6.4.2 Deployment Diagram

Come precedentemente accennato, l'applicazione può essere deployata su un'unica Virtual Machine, oppure essere suddivisa su due tier, separando la componente di frontend da quella di backend, costituita dalla Servlet che esegue il calcolo della similarità. Queste due parti verranno caricate su due istanze diverse, permettendo di spostare il carico computazionale del calcolo dei ¨Best Contacts˝ su una VM dedicata. Quindi, le configurazioni di caricamento possibili per MiC, per entrambe le piattaforme, prevedono l'uso di una singola istanza o di due istanze.

Deploy su Google App Engine utilizzando una singola istanza

Nel caso si decidesse di caricare MiC su un unico tier, le componenti costituenti il Deployement Diagram (Figura 6.11) sarebbero le seguenti:

- MiC Application
- Push Queue Handler: componente nativamente presente nella piattaforma. Esso in modo del tutto autonomo e automatico, legge le richieste di computazioni presenti nella Push Queue, invoca il Task associato e al termine della sua esecuzione, cancella la richiesta dalla coda.
- Push Queue
- Datastore
- Blobstore
- Google Cloud SQL
- Memcache

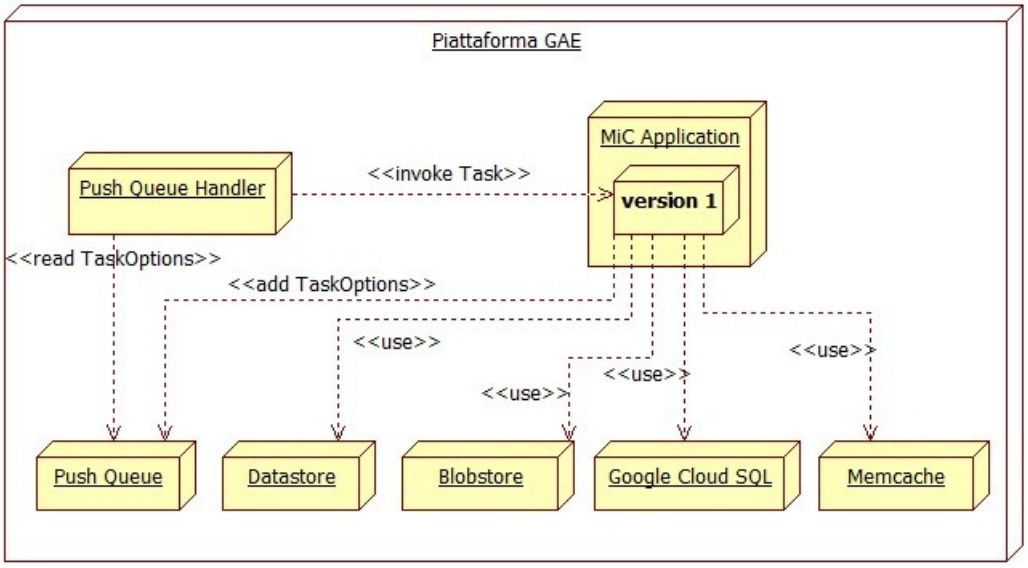


Figura 6.11: Deployment Diagram: deploy su Google App Engine usando una singola istanza

Deploy su Google App Engine utilizzando due istanze

Nel caso si decidesse di deployare MiC separando la componente di frontend da quella di backend, si deve fare riferimento all'uso del versioning, l'unica possibilità per implementare un sistema multi-tier su Google App Engine: in pratica si caricano due copie identiche dell'applicativo, ma con version ID differenti. Una versione, quella di default, verrà adibita alla gestione delle interazioni con gli utenti e rappresenterà quindi il frontend di MiC. La seconda invece si occuperà solamente di eseguire le richieste computazionali inerenti al calcolo dei "Best Contacts". Riassumendo:

- Versione 1: rappresenta il frontend, e gestisce l'interazione web con gli utenti di MiC
- Versione 2: rappresenta il backend, che elabora le richieste di calcolo delle similarità

Le componenti del Deployement Diagram (Figura 6.11) sono le seguenti:

- MiC Application: costituita da due istanze, una per il frontend, l'altra per il backend.
- Push Queue Handler: componente nativamente presente nella piattaforma. Esso in modo del tutto autonomo e automatico, legge le richieste di computazioni presenti nella Push Queue, invoca il Task associato e al termine della sua esecuzione, cancella la richiesta dalla coda.
- Push Queue
- Datastore
- Blobstore
- Google Cloud SQL
- Memcache

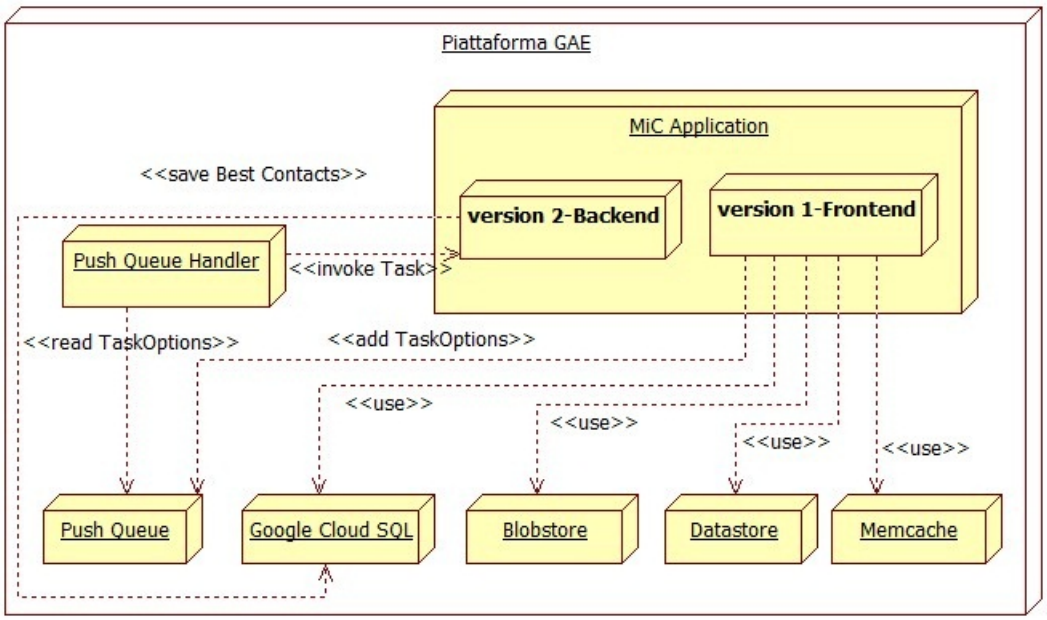


Figura 6.12: Deployment Diagram: deploy su Google App Engine usando due istanze

Deploy su Azure utilizzando una singola istanza

Nel caso del deploy congiunto delle componenti di frontend e backend, il Deployement Diagram (Figura 6.13) include:

- Worker Role: contenente le seguenti componenti:
  - Java JDK: necessaria per poter utilizzare il Java Application Server.
  - Java Application Server: all'interno del quale sono caricati:
    - Internal Worker war: war contenente il codice relativo al gestore delle code per Azure
    - MiC war: applicativo MiC
- Queue Service
- Table Service
- Blob Service
- SQL Database
- Caching Service (per il supporto del protocollo memcache)

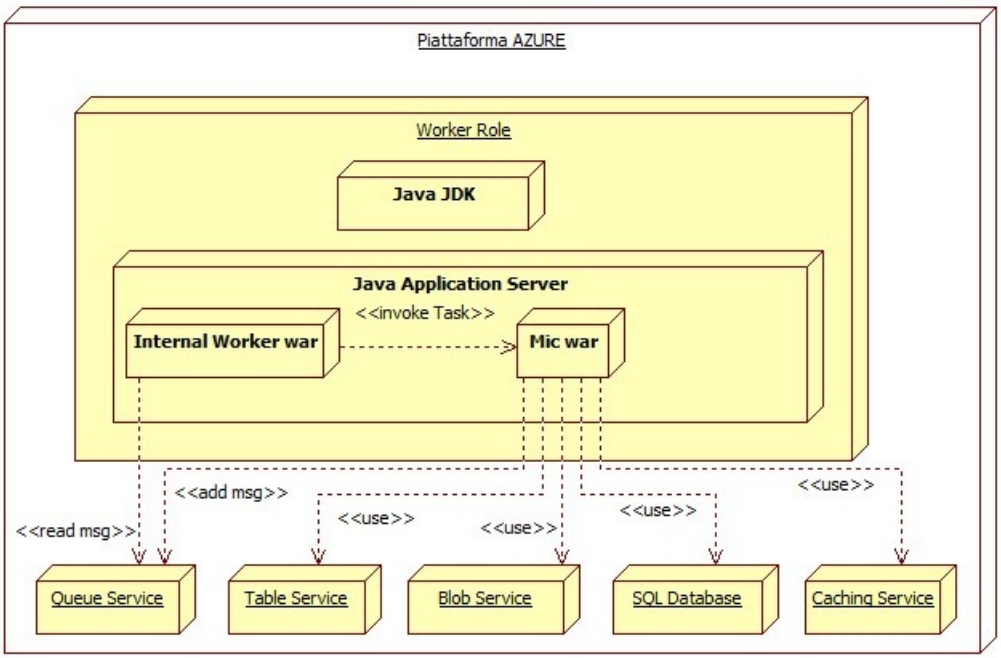


Figura 6.13: Deployment Diagram: deploy su Azure usando una singola istanza

Deploy su Azure utilizzando due istanze

Nel caso si decidesse di deployare MiC su Azure separando la componente di frontend da quella di backend, occorre utilizzare due Worker Role distinte, ognuna con il proprio Java Application Server correttamente configurato. Anche per Azure si esportano due copie dell'applicativo, una per il frontend, l'altra per il backend.

La prima verrà installata all'interno dell'Application Server della Worker Role che chiameremo di Frontend (si ricorda che non è possibile utilizzare Role di tipo Web per applicazioni Java), adibita alla gestione delle interazioni con gli utenti.

La seconda invece sarà installata all'interno del Java Server dell'altra Role, insieme al war relativo al gestore delle code.

L'interazione tra le due Role avverrà proprio attraverso il servizio Queue di Azure: la Worker Role di Frontend inserirà le richieste di computazione nella coda, che saranno poi gestite dalla Role di Backend, attraverso il gestore delle code, che le leggerà ed invocherà la servlet adibita al calcolo, quella all'interno della versione backend di MiC. Le componenti del Deployment Diagram (Figura 6.14) sono le seguenti:

- Worker Role-Frontend: che include le seguenti componenti:
  - Java JDK: necessaria per poter utilizzare il Java Application Server.
  - Java Application Server: all'interno del quale è caricato:
    - MiC-Frontend war: applicativo MiC, versione adibita alle interazioni web
- Worker Role-Backend: contenente le seguenti componenti:
  - Java JDK: necessaria per poter utilizzare il Java Application Server.
  - Java Application Server: all'interno del quale sono caricati:
    - MiC-Backend war: applicativo MiC, versione adibita al calcolo delle similarità
    - Internal Worker war: war contenente il codice relativo al gestore delle code per Azure
- Queue Service
- Table Service
- Blob Service
- SQL Database
- Caching Service (per il supporto del protocollo memcache)

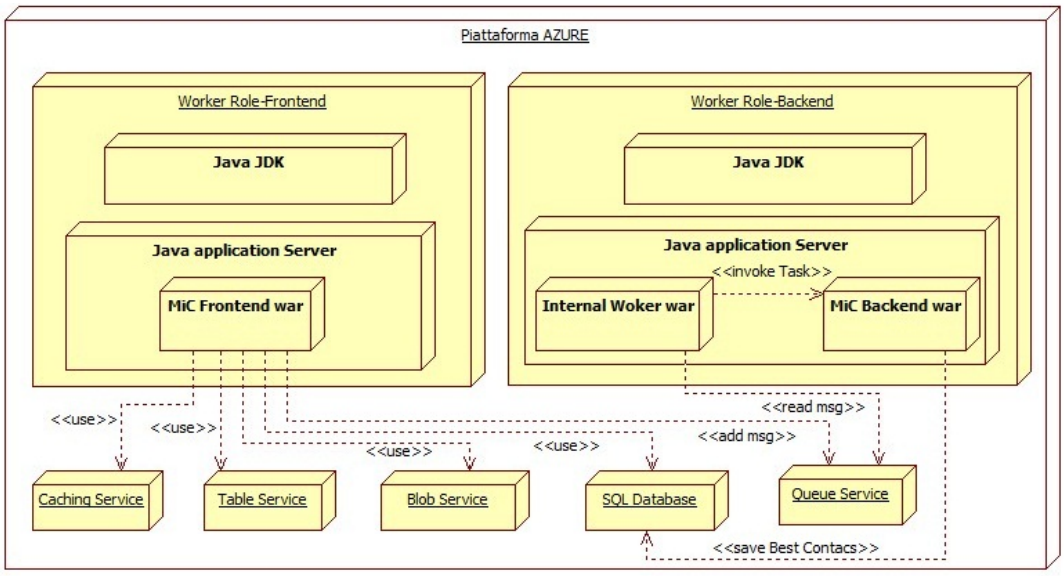


Figura 6.14: Deployment Diagram: deploy su Azure usando due istanze

6.4.3 Livello Dati

MiC gestisce le seguenti informazioni, che dovranno essere memorizzate in appositi sistemi di storage:

- Dati anagrafici utenti registrati: memorizzati utilizzando il servizio SQL
- Foto profilo degli utenti: memorizzate utilizzando il servizio Blob
- Dati che associano ad ogni utente, per ogni argomento selezionato durante la fase di profilazione, i tre utenti le cui risposte riguardanti tale topic risultano essere le più simili, secondo il coefficiente di similarità di Pearson, a quelle dell'utente. Si utilizza anche per queste informazioni il servizio SQL
- Risposte alle domande relative agli argomenti: utilizzando il servizio NoSQL. Questo servizio è generalmente più performante e scalabile del servizio SQL. Ha però alcune limitazioni, una delle quali riguarda la possibilità di effettuare JOIN, oppure query complesse contenenti operatori logici diversi.
  Questi dati sono utilizzati solo dall'entità, interna all'applicazione, che calcola, in modo asincrono rispetto ai flussi di interazioni web, le varie

liste dei "Best Contacts". Non avendo la necessità di effettuare query complesse per poter accedere a tali dati, non avendo particolari vincoli inter-relazionali con altri dati e per motivi prestazionali si è presa la decisione di utilizzare questo servizio per la loro archiviazione.

- I messaggi inseriti dai vari utenti: utilizzando il servizio SQL. La motivazione di questa decisione riguarda la complessità della query per poter accedere ai soli messaggi riguardanti un particolare argomento, scritto da uno dei tre "Best Contacts"per un dato utente e un suo particolare argomento.

- Gli argomenti di discussione disponibili e le relative domande di profilazione: utilizzando il servizio NoSQL. Questi dati sono richiesti solo in fase di profilazione. Non richiedono query complesse, quindi per motivi prestazionali e di scalabiltà si è deciso per il servizio NoSQL.

Diagramma ER tabelle Servizio SQL

La Figura 6.15 mostra il diagramma Entità-Relazioni relativo alle tabelle memorizzate all'interno del database relazionale offerto dal provider Cloud. È composto da tre tabelle:

- UserProfile
  Contiene i dati anagrafici e il link al Blob contenente la foto profilo dell'utente.
  È composta dai seguenti attributi:
  - Email: e-mail dell'utente, chiave primaria
  - Password: password associata all'account dell'utente
  - FirstName: nome dell'utente
  - LastName: cognome dell'utente
  - Date_of_birth: data di nascita dell'utente
  - Gender: sesso dell'utente
  - Picture: URL del Blob contenente la foto profilo dell'utente

- UserSimilarity
  Contiene per ogni utente ed ogni suo argomento di interesse, i tre utenti più simili. È composta dai seguenti attributi:
  - Email: e-mail dell'utente
  - Topic: nome dell'argomento a cui si riferisce la lista di ¨Best Contacs˝
  - First User: e-mail dell'utente con il più alto tasso di similarità
  - Second User: e-mail dell'utente con il secondo più alto tasso di similarità
  - Third User: e-mail dell'utente con il terzo più alto tasso di similarità

  La chiave primaria di questa tabella è la coppia (Email, Topic).

- Message
  - Id: codice che identifica univocamente il messaggio, chiave primaria
  - UserId: e-mail dell'utente autore del messaggio
  - Date: data inserimento del messaggio
  - MessageTxt: contenuto del messaggio
  - Topic: argomento del messaggio

Entità servizio NoSQL

Nella Figura 6.16 viene mostrato il Class Diagram delle entità memorizzate nel servizio NoSQL, utilizzando l'interfaccia JPA. Le due entità utilizzate da MiC sono:

- Topic
  Utilizzata per memorizzare per ogni argomento trattabile in MiC, le relative domande a cui devono rispondere gli utenti interessati ad esso.
  - topicName: nome dell'argomento. Attributo chiave dell'entitità
  - topicQuestions: lista delle domande (a cui dovranno rispondere gli utenti) associate all'argomento.

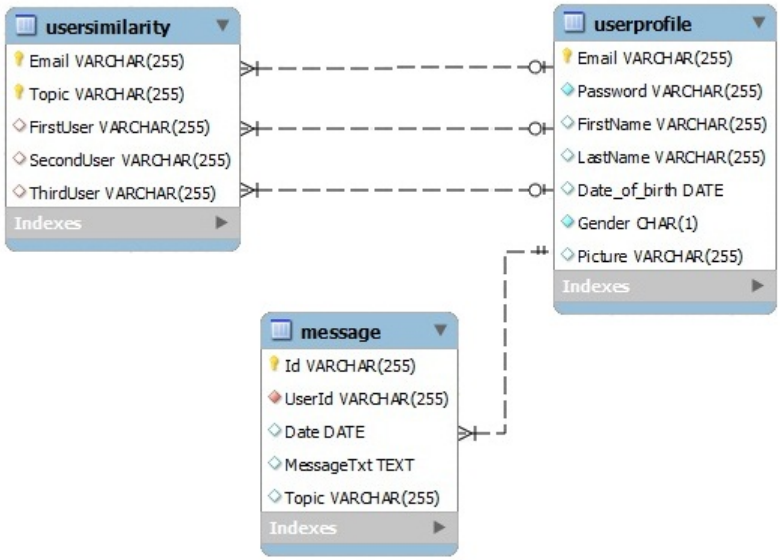


Figura 6.15: Diagramma ER tabelle memorizzate utilizzando servizio SQL

- UserRatigs
  Entità contenente le risposte alle domande degli utenti.
  - id: codice univoco che identifica le risposte. È composto dalla concatenzione del campo "email" e di quello "topicName".
  - email: e-mail dell'utente che ha dato tali risposte
  - topicName: nome dell'argomento a cui si riferiscono le risposte
  - ratings: risposte alle domande relative al "topicName" date dall'utente "email".

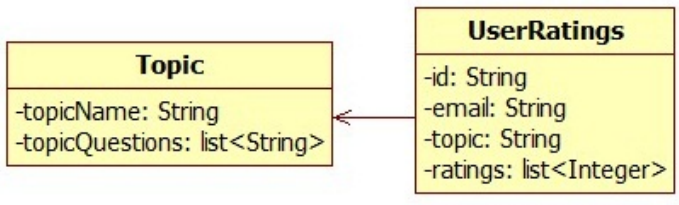


Figura 6.16: Class Diagram Entitità memorizzate utilizzando servizio NoSQL

Capitolo 7

Test di valutazione

Per valutare l'overhead introdotto dalla libreria, si sono svolti dei test di valutazione, atti ha verificare le differenze di performance delle chiamate ai servizi utilizzando le API della libreria rispetto alle chiamate native.

Le metriche utilizzate per il confronto sono essenzialmente due:

- La latenza
- L'overehead computazionale, in termini di utilizzo della risorsa CPU

La Sezione 7.1 descrive nel dettaglio come sono stati progettati e implementati i test di performance, nella Sezione 7.2 vengono descritte le tecniche utilizzate per l'acquisizione dei dati inerenti l'utilizzo della CPU e infine nelle Sezioni 7.3 e 7.4 sono riportati e analizzati i risultati dei test.

7.1 Test Setup

Per l'esecuzione dei test si è utilizzato il tool di testing JMeter [41], un progetto open source della fondazione Apache sviluppato per automatizzare le operazioni di test di carico su macchine remote (web server, application server, database).

Si è utilizzata l'applicazione MiC come Test Case. Nello specifico, oltre alla versione ˝vendor-independent˝, sono state scritte le versioni native per le due piattaforme, che utilizzano le API proprietarie per l'accesso ai servizi Cloud.

La modalità di deploy utilizzata è quella a due tier, separando quindi su due VM differenti il frontend e il backend dell'applicazione, come descritto in 6.4.2.

Tramite JMeter sono stati scritti due diverse tipologie di Test Plan, che simulano due possibili scenari di utilizzo dell'applicazione:

- Test Plan 1: sessione relativa alla registrazione, si veda come riferimento la descrizione del relativo Use Case nella Sezione 6.3.2.
- Test Plan 2: Simulazione scenario nel quale un utente registrato compie le seguenti azioni:
  - Login
  - Modifica del suo profilo
  - Scrittura di un messaggio
  - Logout

7.1.1 Test Plan 1

Questo Test Plan simula lo scenario di registrazione di un utente in MiC, il cui Sequence Diagram è mostrato in Figura 6.3 del Capitolo relativo a MiC. In JMeter, per simulare questo scenario, è stato creato il seguente flusso di HTTP REQUEST:

- HTTP REQUEST REGISTER: questa richiesta HTTP invoca la Servlet responsabile della memorizzazione dei dati anagrafici e della foto profilo dell'utente che sta effettuando la registrazione. Per la creazione casuale dei dati si è utilizzato un apposito BSFPreProcessor, un Javascript che permette la creazione casuale dei dati necessari alla registrazione.
- HTTP REQUEST SELECT TOPICS: questa richiesta HTTP invoca la Servlet responsabile della selezione degli argomenti di interesse per l'utente. Per la creazione casuale dei topic selezionati si è utilizzato un apposito BSFPreProcessor.
- HTTP REQUEST SAVE ANSWERS: questa richiesta HTTP invoca la Servlet responsabile della memorizzazione delle risposte fornite dall'utente alle domande inerenti ad uno dei topic selezionati. Per la creazione casuale delle risposte si è utilizzato anche in questo caso un BSFPreProcessor. Questa richiesta è effettuata per ogni topic selezionato nella richiesta HTTP REQUEST SELECT TOPICS.

Nella Tabella 7.1 sono elencati, per ogni richiesta, i servizi Cloud che vengono utilizzati, il tier interessato e i think time introdotti per rendere realistico il comportamento di un utente nella sessione. A tal fine si sono introdotti degli intervalli tra le richieste utilizzando un timer di tipo Gaussian Random, nel quale si può definire la media (parametro K nella tabella) e la deviazione standard (parametro D) della distribuzione Gaussiana dei ritardi.

HTTP REQUEST	Tier	Servizi Cloud utilizzati	Think Time (sec)
REGISTER	Frontend	Blob, SQL	(K=10;D=2)
SELECT TOPICS	Frontend	-	(K=5;D=1)
SAVE ANSWER	Frontend	NoSQL, TaskQueue	(K=20;D=5)

Tabella 7.1: Tabella riassuntiva Test Plan 1

Relativamente alla richiesta SAVE ANSWER, il servizio Task Queue è utilizzato solo nell'ultima richiesta di ogni flusso. Questo perchè il servizio è impiegato per inserire la richiesta di calcolo della similarità nella coda, operazione effettuata una volta sola e solo al termine della memorizzazione di tutti le risposte di un utente. Il gestore della coda leggerà questa richiesta ed invocherà il task adibito al calcolo della lista dei simili, risiedente nel backend.

7.1.2 Test Plan 2

Questo Test Plan emula un possibile scenario di utilizzo dell'applicazione MiC di un utente registrato. In questo scenario un utente effettua il login, richiede la modifica del proprio profilo, ricompiendo quindi le fasi di selezione degli argomenti e la conseguente di risposta alle domande, richiede un refresh della propria pagina, per poter visualizzare i risultati della nuova computazione della lista dei simili, scrive un messaggio sulla propria bacheca ed infine effettua il logout dal sistema. I Sequence Diagram che descrivono questo scenario sono riportati nella Figure 7.1 7.2 e 7.3.

In JMeter, per simulare questo scenario, è stato creato il seguente flusso di HTTP REQUEST:

- HTTP REQUEST LOGIN: questa richiesta HTTP invoca la Servlet responsabile della verifica delle credenziali inviate dall'utente per richiedere l'accesso al sistema. Per l'invio dinamico di credenziali valide alla Servlet,

si utilizza una base dati contenente gli utenti registrati, rappresentata da un file CSV. JMeter leggerà sequenzialmente le righe di questo file, estraendo i dati relativi allo username e password necessari per l'accesso.

- HTTP REQUEST EDIT PROFILE: questa HTTP REQUEST permette l'invio della richiesta di modifica del profilo, inviando semplicemente il parametro "editprofile" con valore "true".

- HTTP REQUEST SELECT TOPICS: nello scenario in cui un utente desidera modificare la propria profilazione in MiC, questa richiesta non è solamente dedita alla fase di selezione dei nuovi argomenti di interesse per l'utente, ma anche della precedente di cancellazione dei dati relativi la vecchia profilazione. Come descritto per il Test Plan 1, per la creazione casuale dei nuovi topic selezionati, si è utilizzato un apposito script di tipo BSFPreProcessor.

- HTTP REQUEST SAVE ANSWERS: questa richiesta HTTP invoca la Servlet responsabile della memorizzazione delle risposte fornite dall'utente alle domande inerenti ad uno dei topic selezionati. Per la creazione casuale delle risposte si è utilizzato anche in questo caso un apposito BSFPreProcessor. Questa richiesta è effettuata per ogni topic selezionato nella richiesta HTTP REQUEST SELECT TOPICS.

- HTTP REQUEST REFRESH: permette l'invio della richiesta di refresh della pagina profilo dell'utente. La richiesta invia semplicemente il parametro "refresh" con valore "true", che causa la cancellazione dei dati dell'utente presenti nella memcache e la successiva richiesta dei medesimi agli altri sistemi di storage, contenenti i dati aggiornati.

- HTTP REQUEST SEND A POST: questa richiesta HTTP invoca la Servlet responsabile dell'invio di un messaggio per conto dell'utente. Per la creazione del messaggio si è utilizzato anche in questo caso un apposito BSFPreProcessor.

- HTTP REQUEST LOGOUT: questa richiesta effettua il logout dal sistema. La Servlet adibita a tale compito cancella anche tutte le informazioni relative all'utente presenti in memcache.

Nella Tabella 7.2 sono elencati per ogni richiesta i servizi Cloud che utilizza, il tier interessato e i think time introdotti. A tal fine, anche per questo Test Plan si sono utilizzati timer di tipo Gaussian Random.

HTTP REQUEST	Tier	Servizi Cloud utilizzati	Think Time (sec)
LOGIN	Frontend	SQL,Memcache,NoSQL,Blob	(K=5;D=1)
EDIT PROFILE	Frontend	-	(K=2;D=0.1)
SELECT TOPICS	Frontend	NoSQL,SQL,Memcache	(K=5;D=1)
SAVE ANSWER	Frontend	NoSQL, TaskQueue	(K=20;D=5)
REFRESH	Frontend	NoSQL,SQL,Memcache	(K=2;D=0.5)
WRITE POST	Frontend	SQL	(K=10;D=3)
LOGOUT	Frontend	Memcache	-

Tabella 7.2: Tabella riassuntiva Test Plan 2

Per le medesime motivazione descritte per il Test Plan 1, il servizio Task Queue nella SAVE ANSWERS è utilizzato solo nell'ultima richiesta di ogni flusso.

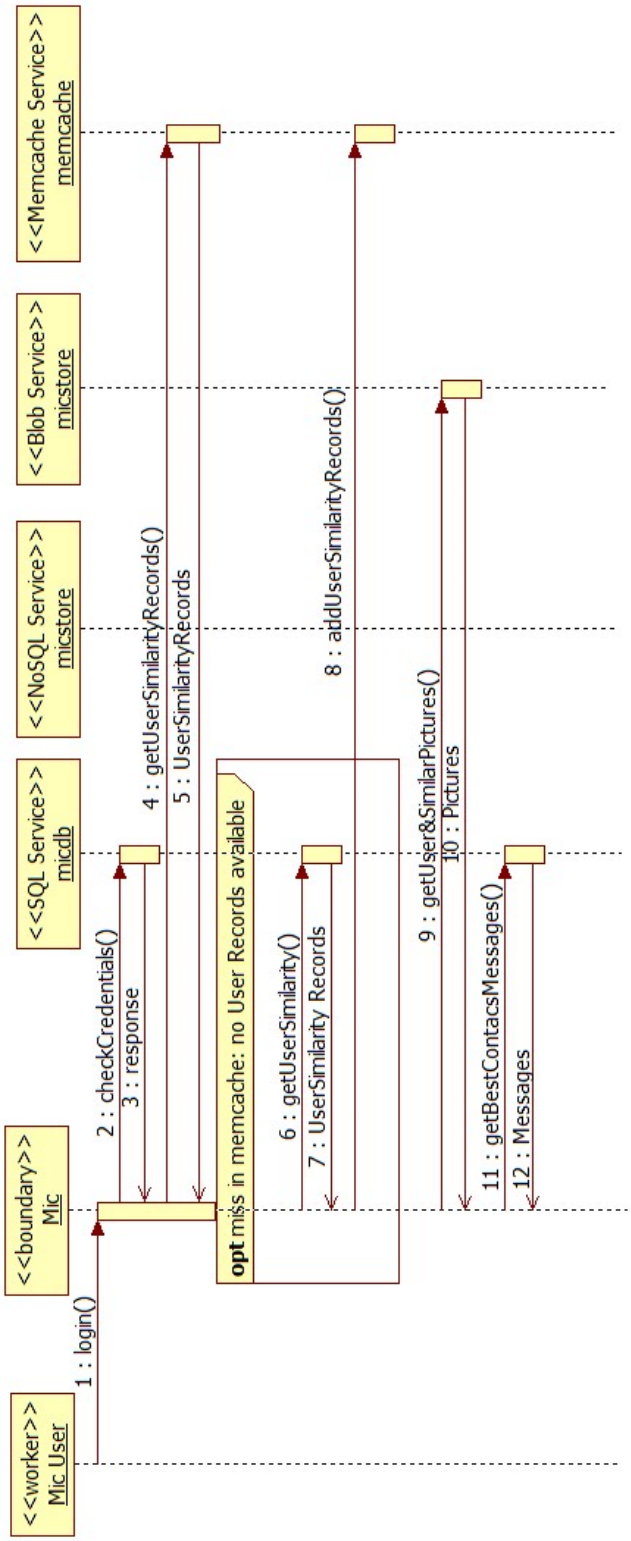


Figura 7.1: Sequence Diagram: Test Plan 2(a)

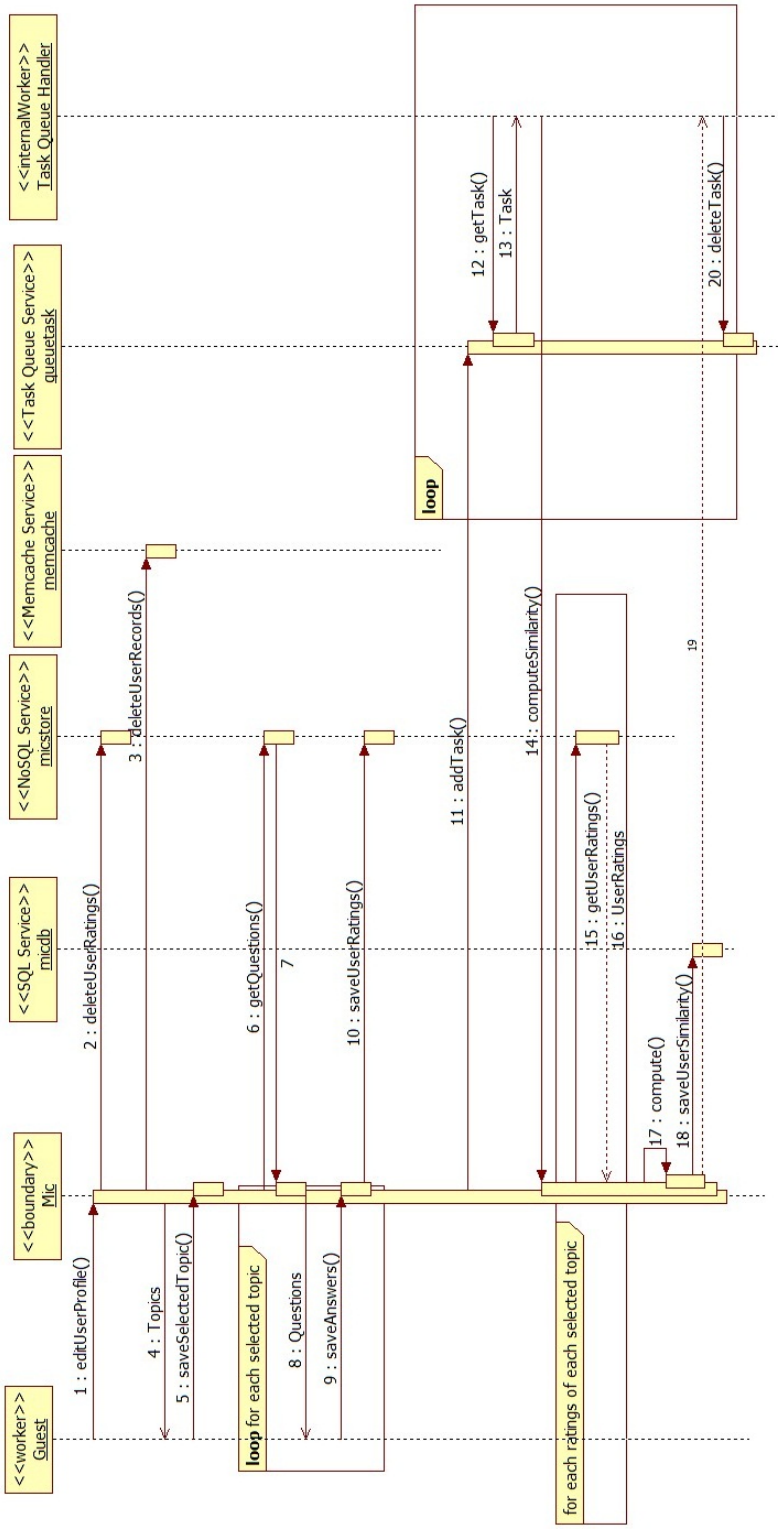


Figura 7.2: Sequence Diagram: Test Plan 2(b)

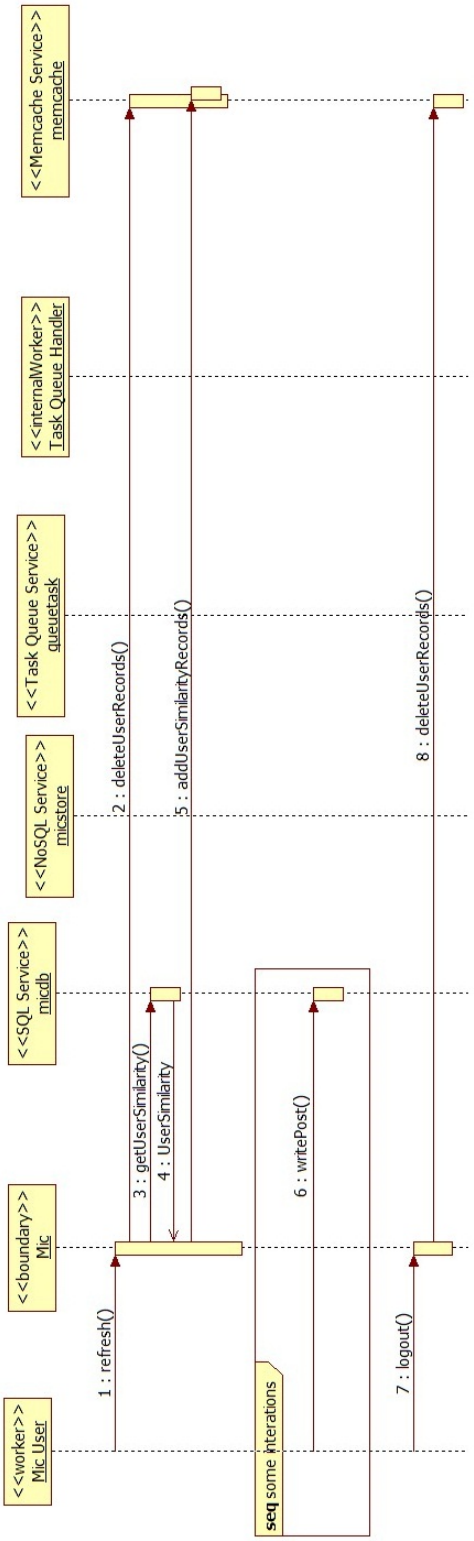


Figura 7.3: Sequence Diagram: Test Plan 2(c)

7.2 Tecnica acquisizione utilizzo CPU

JMeter permette di acquisire informazioni riguardanti i tempi di latenza delle varie richieste, ma non fornisce alcun dato riguardante l'utilizzo della CPU della Virtual Machine durante i test. Per acquisire queste informazioni si sono utilizzati strumenti diversi per le due piattaforme.

Per quanto riguarda Azure, grazie alla possibilità di accesso remoto alla VM ospitante il nostro deployment (tramite Remote Desktop Protocol (RDP)), si sono raccolti i dati di utilizzo della CPU direttamente dal Tool di Monitoring della stessa. L'unità di misura delle rilevazione è la percentuale di utilizzo della CPU.

Per App Engine invece, non avendo a disposizione questi strumenti, si è utilizzato il servizio *Quota*, le cui API sono fornite direttamente nella SDK Java, che permettono di acquisire, per un particolare blocco di codice, la misura relativa al tempo speso dalla richiesta all'interno della sandbox di App Engine. L'unità di misura utilizzata sono i megacicli macchina. Se tutte le istruzioni fossero eseguite sequenzialmente su una macchina standard 1.2 GHz 64-bit x86 CPU, 1200 megacicli equivarrebbero ad un secondo. Un esempio della tecnica di acquisizione dell'utilizzo della CPU in App Engine è mostrato nel Listato 7.1.

Listato 7.1: Esempio utilizzo Quota API

```
1  ...
2  QuotaService quota=QuotaServiceFactory.getQuotaService();
3  long start=quota.getCpuTimeInMegaCycles();
4  istruction1();
5  istruction2();
6  ...
7  istructionN();
8  long end=quota.getCpuTimeInMegaCycles();
9  System.out.println("Durata esecuzione blocco di codice:"+(end-start)+"megacicli");
10 ...
```

Relativamente ai test effettuati, si utilizzerà questa tecnica per registrare i megacicli macchina consumati per l'esecuzione delle Servlet/JSP richiamate dalle HTTP REQUEST presenti nel Test Plan.

Tramite i dati ricavati, si andrà a calcolare la percentuale di utilizzo della CPU, attraverso la seguente formula:

$$\frac{\sum_{k=1}^{N}(M_k)}{1200*T}$$

M_k= Megacicli consumati dalla k-esima richiesta
N= numero totale di richieste effettuate dal Test Plan
T= durata del Test Plan, espressa in secondi

7.3 Risultati Test Plan 1

Per il Test Plan 1 sono stati effettuati diversi test, sia sulle versioni di MiC utilizzanti le API native sia sulla versione ¨vendor-independent˝, variando ogni volta gli schemi di carico, come descritto in seguito:

- Test 1 utente
  Lo schema di carico relativo a questo test, mostrato in Figura 7.4, mantiene costante per 60 minuti il carico relativo ad un utente.
- Test 10 utenti
- Test 20 utenti
- Test 30 utenti
- Test 50 utenti

Per questi test si è definito lo schema di carico che prevede, dopo un periodo di 15 minuti di rump-up, di mantenere costante per 60 minuti il carico, rispettivamente, di 10, 20, 30 oppure 50 utenti.
Le raffigurazioni qualitative degli schemi di carico sono mostrate nelle Figure 7.5, 7.6, 7.7 e 7.8.

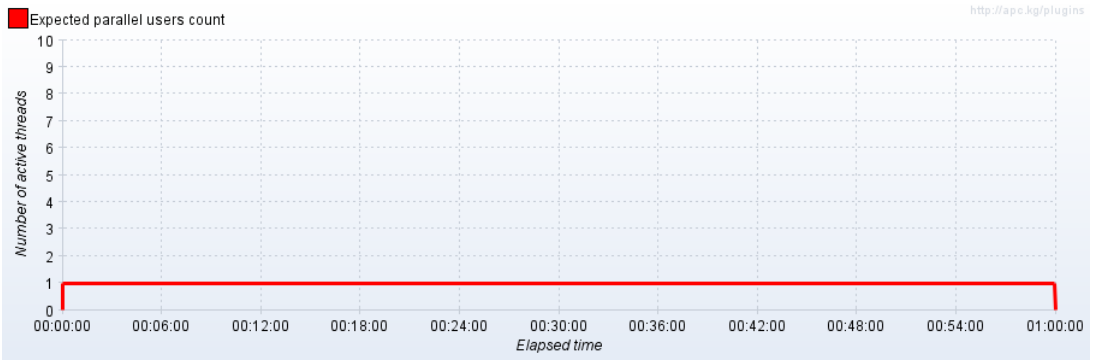


Figura 7.4: Test Plan 1: carico di 1 utente

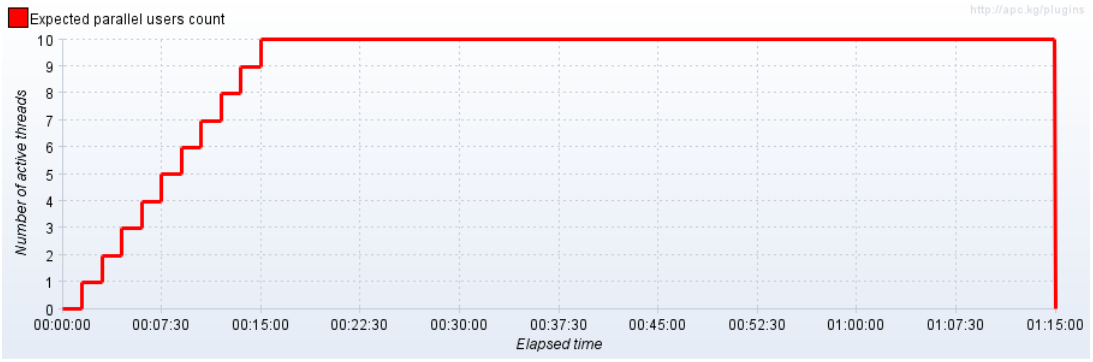


Figura 7.5: Test Plan 1: carico di 10 utenti

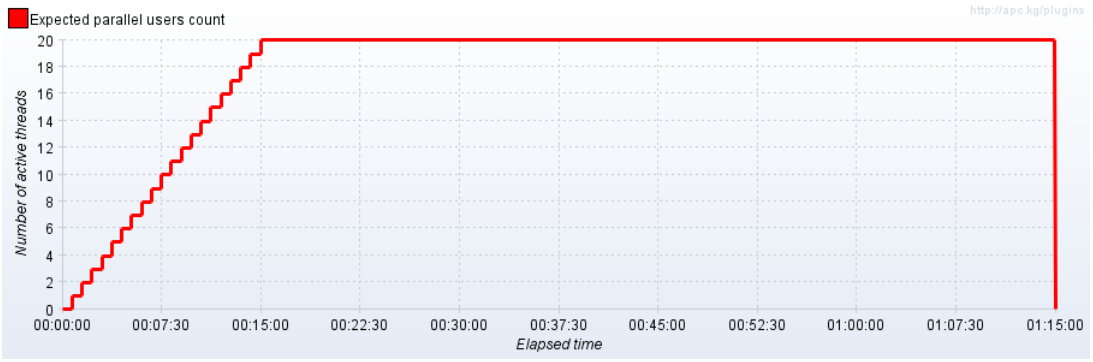


Figura 7.6: Test Plan 1: carico di 20 utenti

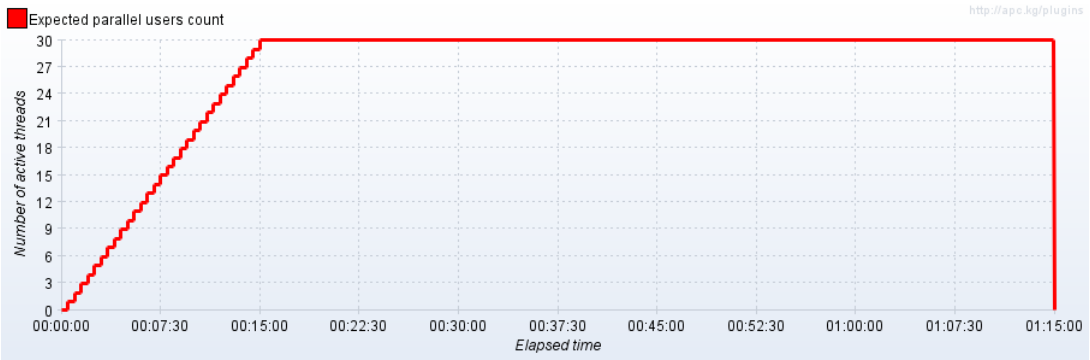


Figura 7.7: Test Plan 1: carico di 30 utenti

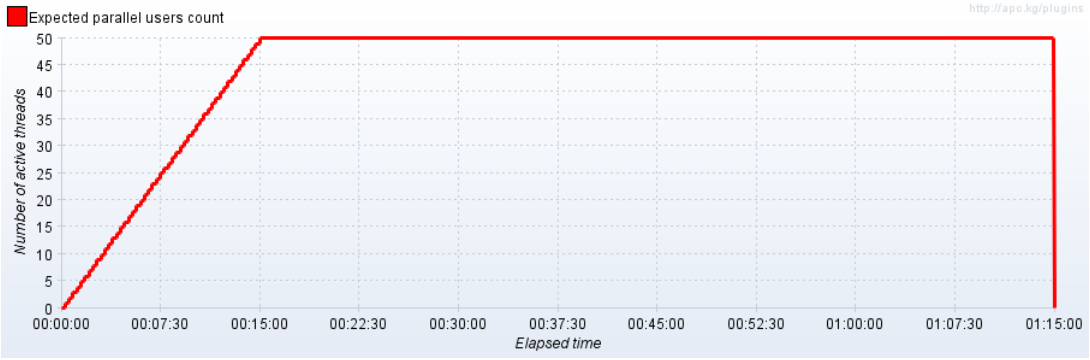


Figura 7.8: Test Plan 1: carico di 50 utenti

7.3.1 Risultati per Azure

I test sono stati effettuati deployando MiC in due Worker Role (una per il frontend e l'altra per il backend) di dimensioni Small, aventi le seguenti caratteristiche:

- CPU: 1,6 GHz
- Memoria RAM: 1,75 GB

Il contenuto dei sistemi di storage, mantenuto equivalente per tutti i test di entrambe le versioni di MiC, è stato di:

- Relativamente al servizio SQL Database:
  - Numero Tuple tabella Message: 3000 circa
  - Numero Tuple tabella UserProfile: 4300 circa
  - Numero Tuble tabella UserSimilarity: 14800 circa
- Relativamente al Table Service:
  - Numero entità UserRatings: 14800 circa
  - Numero entità Topic: 7

Confronto tempi di Latenza

Come è chiaramente visibile nei grafici contenuti nelle Figure 7.9, 7.10, 7.11, dal confronto dei dati ricavati da JMeter, non si osservano differenze sostanziali tra le richieste che utilizzano lo strato di astrazione CPIM e quelle che utilizzano le API native di Azure.

Unica nota è relativa al test a 50 utenti, dove nella richiesta REGISTER la versione nativa ha tempi medi di latenza maggiori rispetto alla versione che utilizza CPIM. In realtà questa differenza è causata da poche richieste che, subendo elevati tempi di latenza dovuti probabilmente all'azione del Garbage Collector oppure da altre attività interne del Cloud, hanno falsato il risultato della media. Come si può notare infatti, i valori delle mediane per questo test sono praticamente identici.

In Figura 7.12 si mostra il grafico JMeter del Response Time di tutte le richieste inviate durante il test di carico con 50 utenti, dal quale si notano

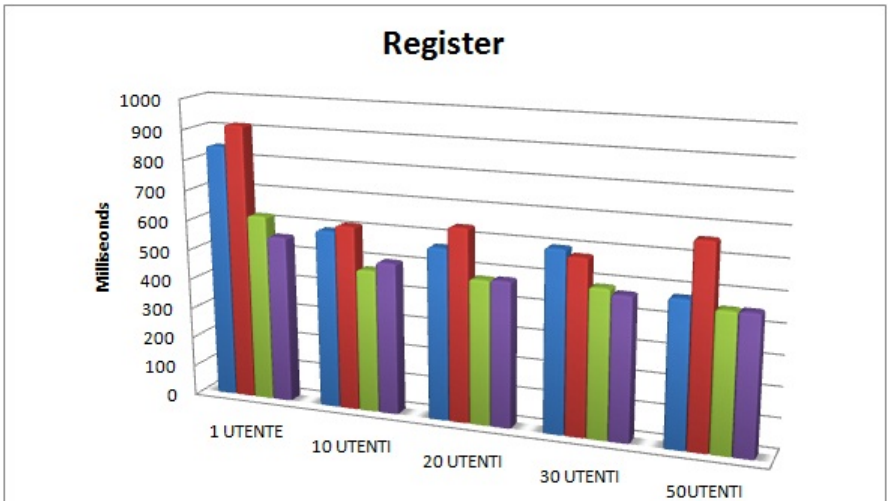
chiaramente le richieste REGISTER con tempi di latenza elevati, non registrati invece nel test relativo alla libreria, come confermato dal grafico di Figura 7.13. Inoltre in Figura 7.14 viene riportato il grafico relativo alla deviazione standard, inerente alla richiesta di REGISTER, nel quale si evidenzia il picco del valore relativo al test in questione.

Confronto utilizzo medio CPU

Un ulteriore importante riscontro, è quello ottenuto dall'analisi dell'overhead introdotto dalla libreria dal punto di vista dell'utilizzo della CPU.

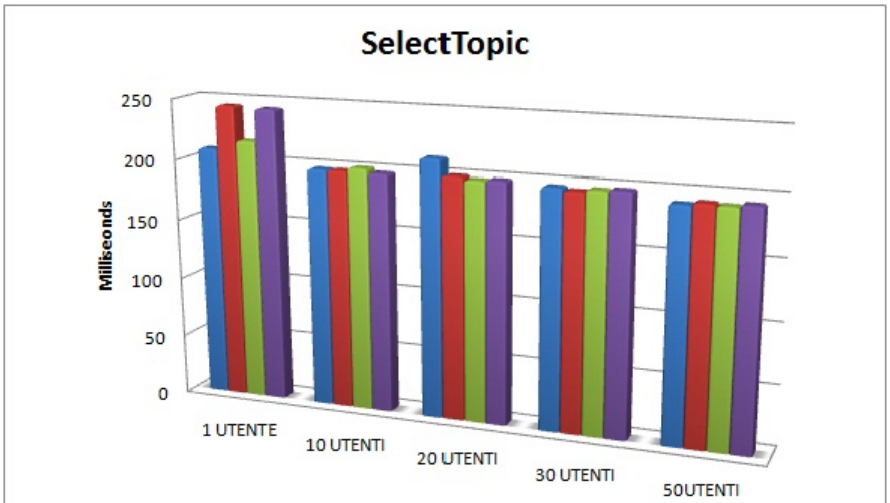
Dal grafico di Figura 7.15, si nota come l'utilizzo medio della CPU da parte del frontend delle due versioni è praticamente il medesimo, indice che in condizioni di basso carico l'overhead computazionale introdotto dalla CPIM è pressoché trascurabile.

Buoni risultati sono stati ottenuti anche per il backend (grafico in Figura 7.16), dove l'overhead introdotto è risultato sempre inferiore al 5%, anche in condizioni di alto carico. Quest'ultimo risultato conferma che l'emulazione del servizio Task Queue, utilizzato per la gestione del calcolo della similarità nella versione CPIM e risiedente nel backend, ha buone performance.



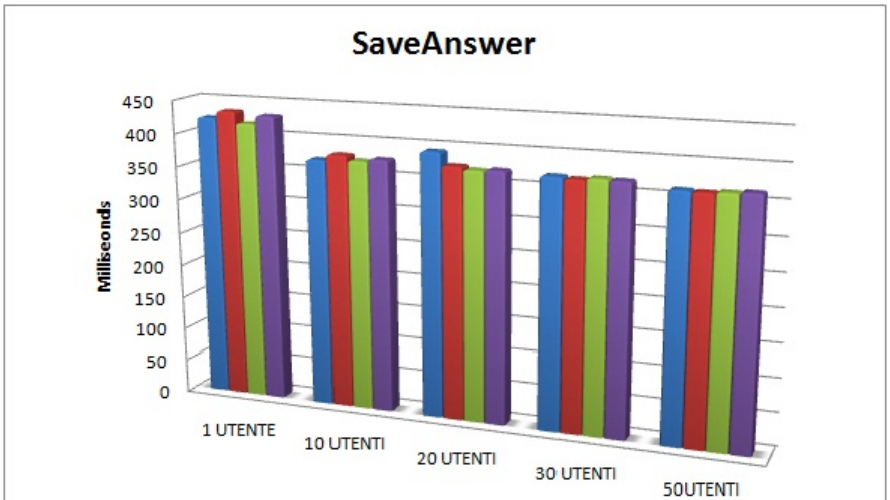
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	845	593	570	600	479
Nativa - AVG	918	614	640	579	663
CPIM - MEDIAN	623	475	479	489	456
Nativa - MEDIAN	558	505	482	473	457

Figura 7.9: Confronto tempi medi di latenza per la richiesta REGISTER



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	209	198	213	197	191
Nativa - AVG	245	198	200	195	193
CPIM - MEDIAN	217	201	198	197	192
Nativa - MEDIAN	244	198	199	198	194

Figura 7.10: Confronto tempi medi di latenza per la richiesta SELECT TOPIC



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	424	371	394	370	363
Nativa - AVG	435	380	375	368	362
CPIM - MEDIAN	418	373	371	371	364
Nativa - MEDIAN	431	377	373	370	366

Figura 7.11: Confronto tempi medi di latenza per la richiesta SAVEANSWER

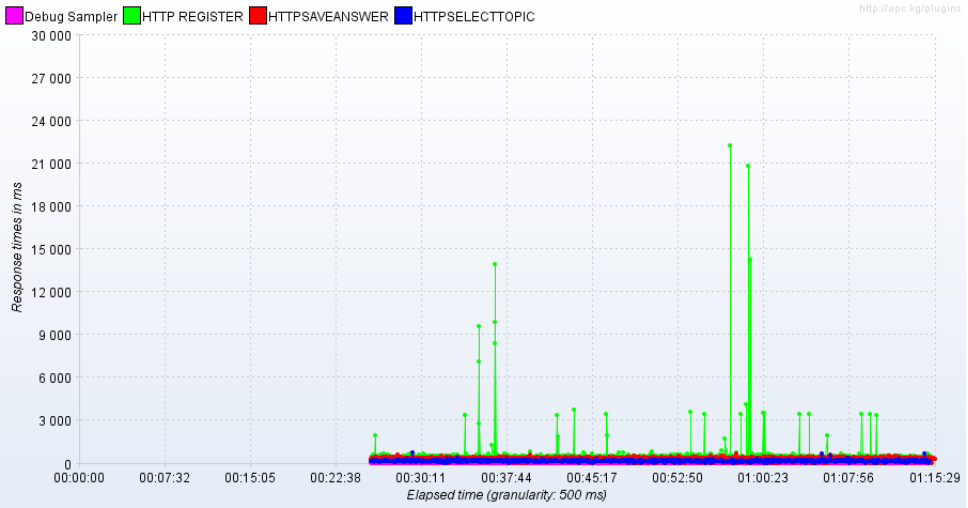


Figura 7.12: Response Time: test 50 utenti - versione MiC Nativa

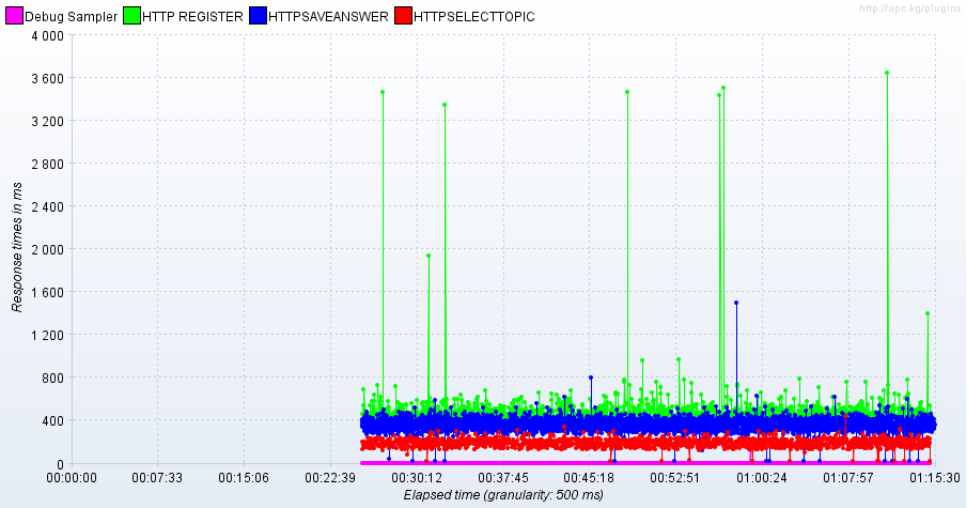
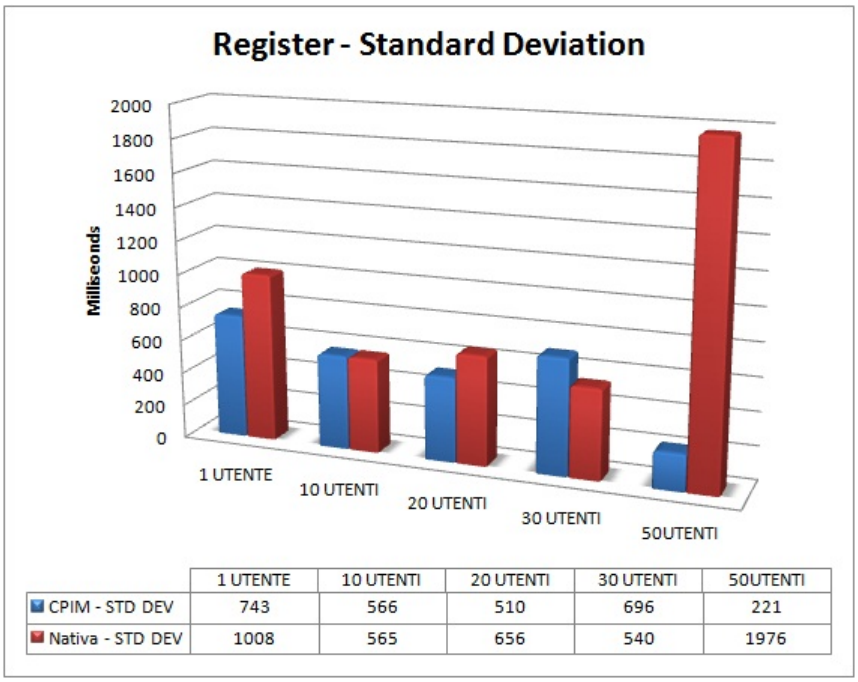


Figura 7.13: Response Time: test 50 utenti - versione MiC CPMI



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - STD DEV	743	566	510	696	221
Nativa - STD DEV	1008	565	656	540	1976

Figura 7.14: Deviazione Standard per la richiesta REGISTER

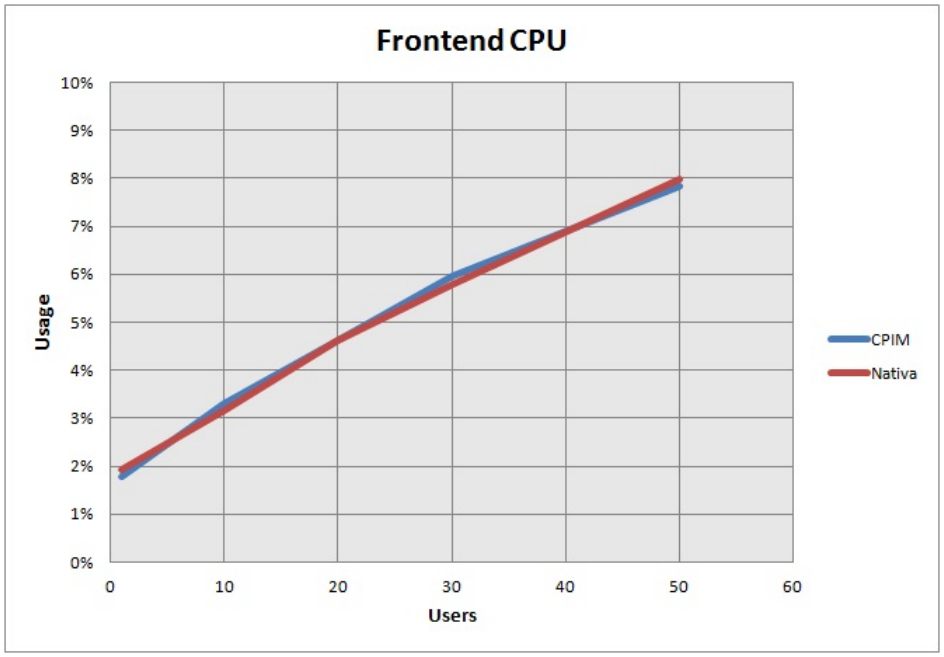


Figura 7.15: Confronto utilizzo medio CPU Frontend

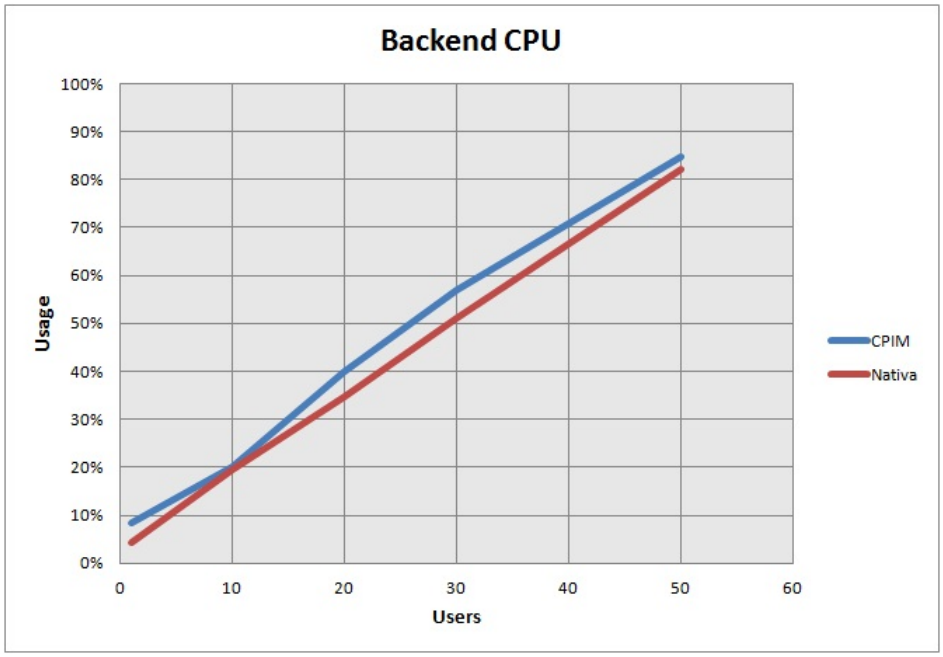


Figura 7.16: Confronto utilizzo medio CPU Backend

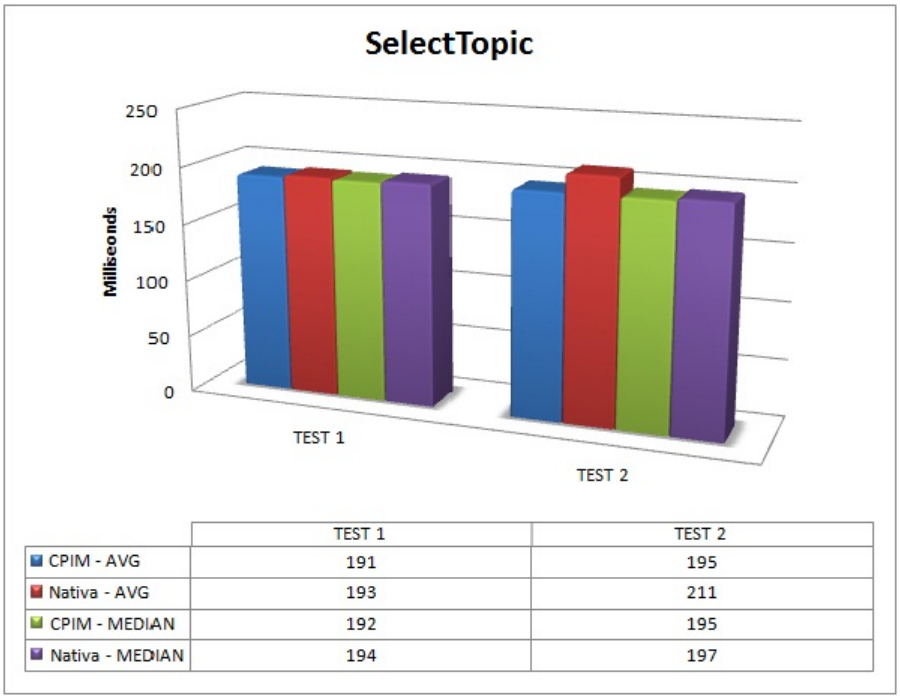
Replica Test 50 utenti

Per verificare la variabilità dei test ad alto carico, si è deciso di replicare il test con 50 utenti.

Dal confronto dei due test non si evincono sostanziali differenze, sia dal confronto delle latenze sia da quello relativo l'utilizzo della CPU. I grafici dei confronti sono mostrati nelle Figure 7.17, 7.18 e 7.19, per quanto riguarda le latenze e nelle Figure 7.20 e 7.21 per quanto riguarda l'utilizzo della CPU.

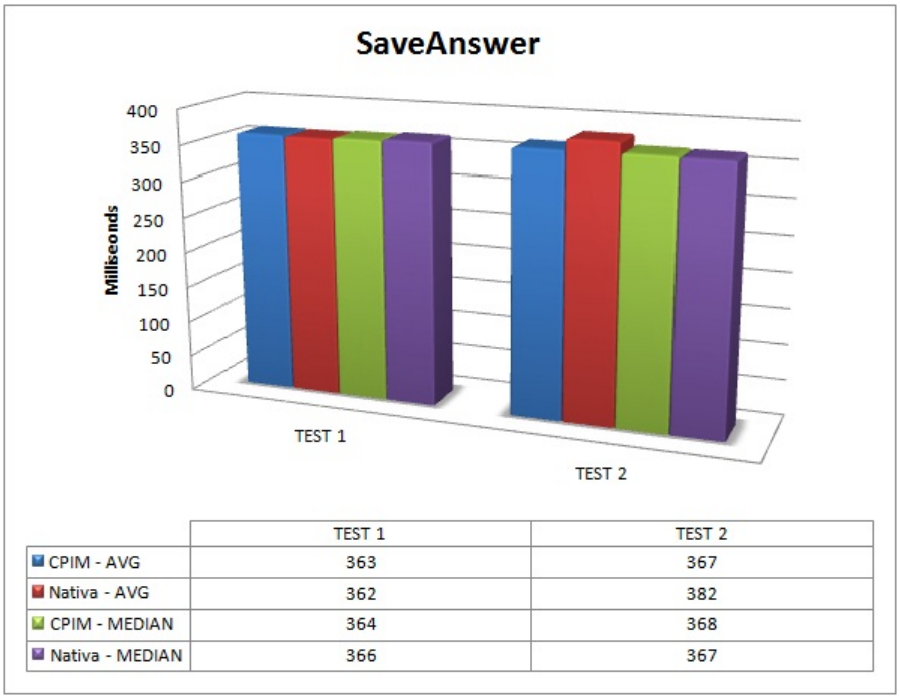
	TEST 1	TEST 2
CPIM - AVG	479	502
Nativa - AVG	663	538
CPIM - MEDIAN	456	468
Nativa - MEDIAN	457	471

Figura 7.17: Confronto test 50 utenti: latenza richieste REGISTER



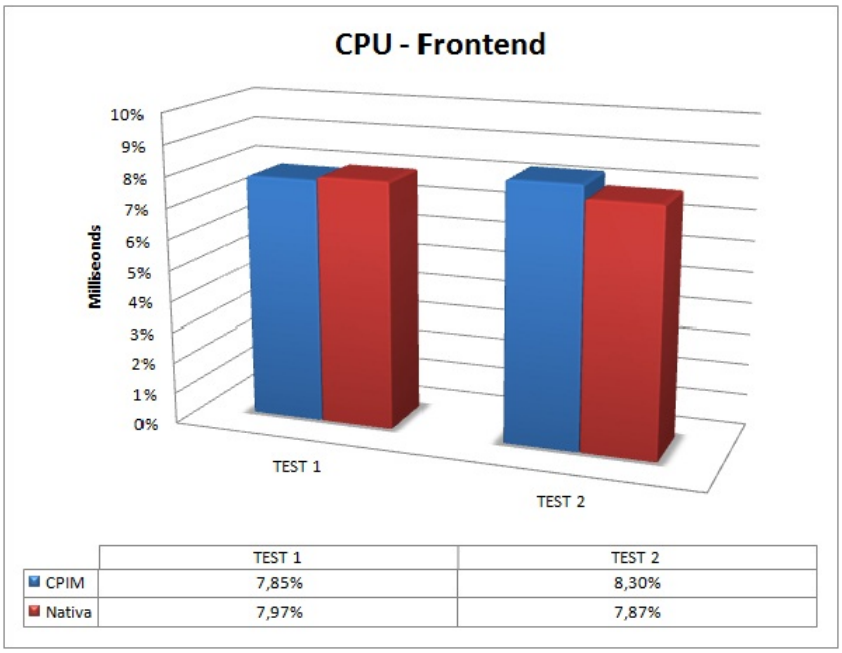
	TEST 1	TEST 2
CPIM - AVG	191	195
Nativa - AVG	193	211
CPIM - MEDIAN	192	195
Nativa - MEDIAN	194	197

Figura 7.18: Confronto test 50 utenti: latenza richieste SELECT TOPIC



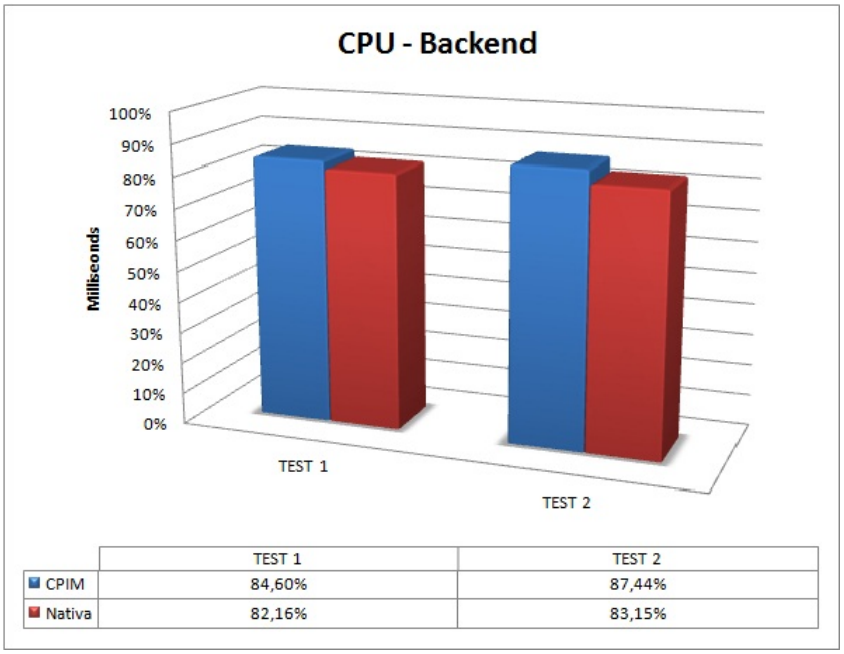
	TEST 1	TEST 2
CPIM - AVG	363	367
Nativa - AVG	362	382
CPIM - MEDIAN	364	368
Nativa - MEDIAN	366	367

Figura 7.19: Confronto test 50 utenti: latenza richieste SAVEANSWER



	TEST 1	TEST 2
CPIM	7,85%	8,30%
Nativa	7,97%	7,87%

Figura 7.20: Confronto test 50 utenti: utilizzo medio CPU Frontend



	TEST 1	TEST 2
CPIM	84,60%	87,44%
Nativa	82,16%	83,15%

Figura 7.21: Confronto test 50 utenti: utilizzo medio CPU Backend

7.3.2 Risultati per App Engine

I test sono stati effettuati deployando MiC in due istanze (una per il frontend e l'altra per il backend) di dimensioni F1, aventi le seguenti caratteristiche:

- CPU: 600 MHz
- Memoria RAM: 128 MB

Per quanto riguarda la piattaforma di Google occorre anche impostare le configurazioni relative alle performance minime richieste dall'applicazione, configurando:

- Il numero delle Idle Instance, cioè delle istanze sempre disponibili per la versione di Default (nel nostro caso quella contenente il frontend). È richiesto di impostare il valore minimo e massimo di Idle Instance. Il range utilizzato per i test è (Minimo=Automatic, Massimo=1), che richiede in pratica di utilizzare meno istanze idle possibili.
- Parametro Pending Latency, che rappresenta il limite di quanto una richiesta pendente può attendere in coda prima di essere servita da un'istanza dell'applicazione. In caso di superamento di questo limite viene messa a disposizione una nuova istanza. Anche per questo parametro è richiesta l'impostazione del valore minimo e massimo di attesa in coda. Il range utilizzato per i test è (Minimo=15 sec, Massimo=Automatic), che richiede in pratica di scalare il meno possibile.

Il contenuto dei sistemi di storage, mantenuto equivalente per tutti i test di entrambe le versioni di MiC, è stato di:

- Relativamente al servizio Google Cloud SQL:
  - Numero Tuple tabella Message: 3700 circa
  - Numero Tuple tabella UserProfile: 2400 circa
  - Numero Tuble tabella UserSimilarity: 8000 circa
- Relativamente al Datastore:
  - Numero entità UserRatings: 8000 circa
  - Numero entità Topic: 7

Confronto tempi di Latenza

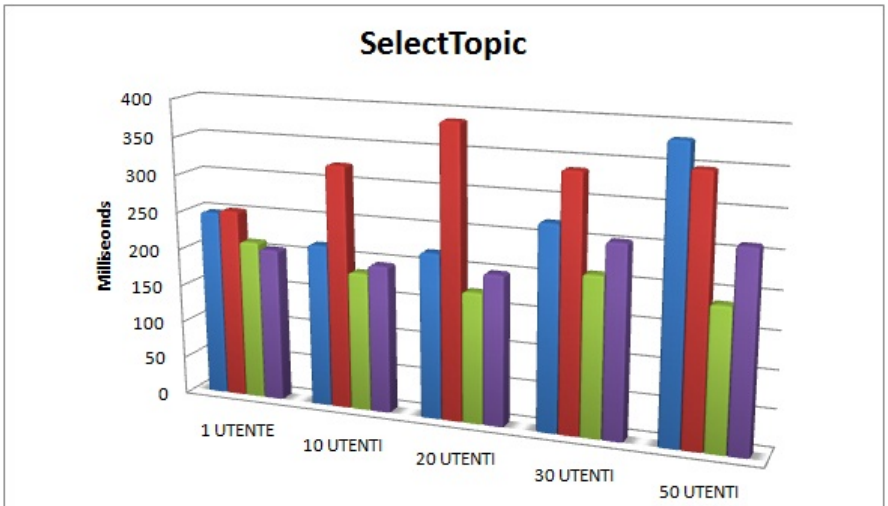
Come è chiaramente visibile nei grafici contenuti nelle Figure 7.22, 7.23, 7.24, dal confronto dei dati ricavati da JMeter, non si osservano differenze sostanziali tra le richieste che utilizzano lo strato di astrazione CPIM e quelle che utilizzano le API native di App Engine. Tutte le richieste hanno differenze inferiori ai 100 millisecondi, ordine di grandezza del tutto trascurabile. Osservando anche i valori delle mediane si riscontrano valori molto vicini, con differenze in media nell'ordine dei 50 millisecondi.

Confronto utilizzo medio CPU

Per quanto riguarda la piattaforma di Google, si sono rilevati dati riguardanti l'utilizzo della CPU solo per il Test Plan 2.

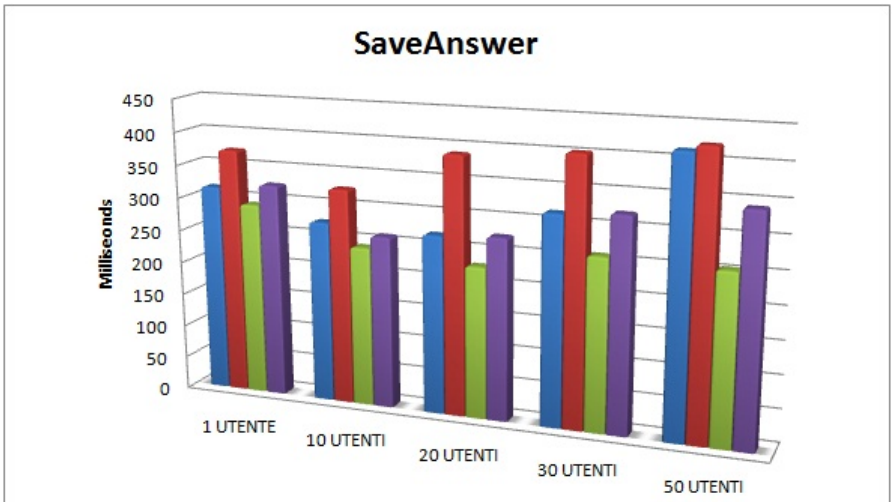
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	640	485	481	565	562
Nativa - AVG	693	563	588	626	663
CPIM - MEDIAN	581	469	447	508	485
Nativa - MEDIAN	640	506	521	573	580

Figura 7.22: Confronto tempi medi di latenza per la richiesta REGISTER



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	249	217	220	271	382
Nativa - AVG	253	324	389	338	349
CPIM - MEDIAN	213	185	174	211	188
Nativa - MEDIAN	205	197	200	254	262

Figura 7.23: Confronto tempi medi di latenza per la richiesta SELECTTOPIC



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	316	275	270	316	415
Nativa - AVG	374	327	390	403	424
CPIM - MEDIAN	293	242	229	260	256
Nativa - MEDIAN	325	261	275	322	343

Figura 7.24: Confronto tempi medi di latenza per la richiesta SAVEANSWERS

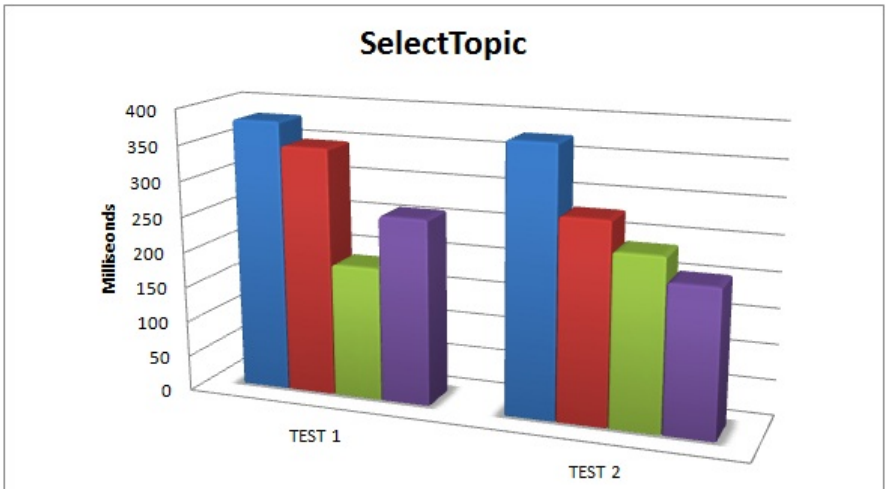
Replica Test 50 utenti

Per verificare la variabilità dei test ad alto carico, si è deciso di replicare il test con 50 utenti.

Dal confronto dei due test non si evincono sostanziali differenze riguardo le latenze registrate: il gap tra la versione con CPMI e quella nativa rimane sempre sotto la soglia, trascurabile, dei 100ms. I grafici relativi ai confronti sono mostrati nelle Figure 7.25, 7.26, 7.27.

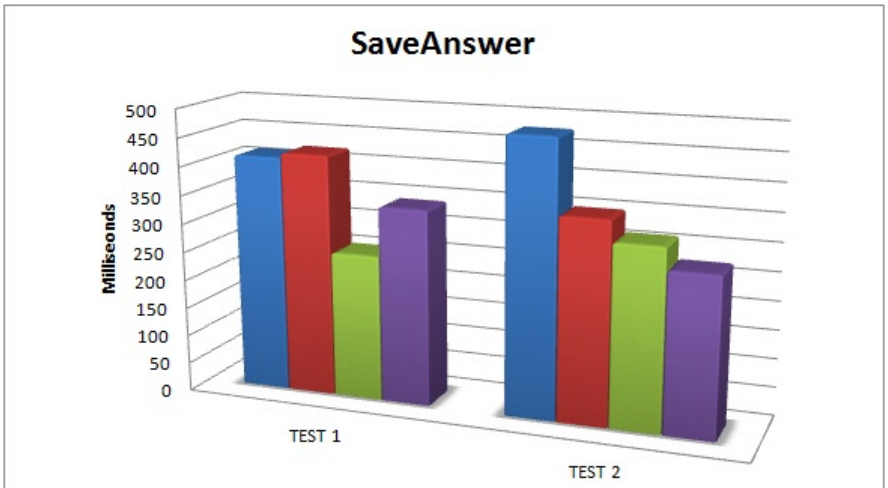
	TEST 1	TEST 2
CPIM - AVG	562	713
Nativa - AVG	663	617
CPIM - MEDIAN	485	552
Nativa - MEDIAN	580	550

Figura 7.25: Confronto tempi medi di latenza per la richiesta REGISTER



	TEST 1	TEST 2
CPIM - AVG	382	376
Nativa - AVG	349	281
CPIM - MEDIAN	188	240
Nativa - MEDIAN	262	206

Figura 7.26: Confronto tempi medi di latenza per la richiesta SELECTTOPIC



	TEST 1	TEST 2
CPIM - AVG	415	480
Nativa - AVG	424	352
CPIM - MEDIAN	256	315
Nativa - MEDIAN	343	277

Figura 7.27: Confronto tempi medi di latenza per la richiesta SAVEANSWERS

7.4 Risultati Test Plan 2

Per il Test Plan 2 sono stati effettuati i diversi test, sia sulle versioni di MiC che utilizzano le API native sia sulla versione ˝vendor-independent˝, con i medesimi schemi di carico utilizzati per il Test Plan 1.

7.4.1 Risultati per Azure

I test sono stati effettuati deployando MiC in due Worker Role (una per il frontend e l'altra per il backend) di dimensioni Small, aventi le seguenti caratteristiche:

- CPU: 1,6 GHz
- Memoria RAM: 1,75 GB

Il contenuto dei sistemi di storage, mantenuto equivalente per tutti i test di entrambe le versioni di MiC, è stato di:

- Relativamente al servizio SQL Database:
  - Numero Tuple tabella Message: 3000 circa
  - Numero Tuple tabella UserProfile: 4300 circa
  - Numero Tuble tabella UserSimilarity: 14800 circa
- Relativamente al Table Service:
  - Numero entità UserRatings: 14800 circa
  - Numero entità Topic: 7

Confronto tempi di Latenza

Dai grafici contenuti nelle Figure 7.28, 7.29, 7.30, 7.31, 7.32, 7.33, 7.34, che misurano i tempi di latenza rilevati da JMeter, non si osservano differenze sostanziali tra le richieste che utilizzano lo strato di astrazione CPIM e quelle che utilizzano le API native di Azure.

Un'attenta analisi è stata fatta per il test relativo a 10 utenti, nel quale si osservano dei tempi medi di latenza superiori per la versione nativa. Nello specifico, nei risultati delle richieste LOGIN, SELECT TOPICS e SAVE

ANSWERS si sono osservate delle differenze maggiori di 300 millisecondi. Osservando i grafici relativi alle deviazione standard per queste richieste (rispettivamente nelle Figure 7.35, 7.37 e 7.36) è lampante come gli alti valori delle deviazioni standard per questi casi, indichino che qualche richiesta ha avuto tempi di latenza molto alti. La conferma si ha grazie al grafico relativo ai tempi di risposta relativi a questo test, mostrato in Figura 7.38, dove è visibilmente chiaro che si sono verificate delle richieste con tempi di latenza nell'ordine dei 30 secondi in tre punti concentrati del test, che hanno poi pesato in modo consistente nel computo delle medie. I valori delle mediane per questi casi, infatti, sono praticamente identici per entrambe le versioni.

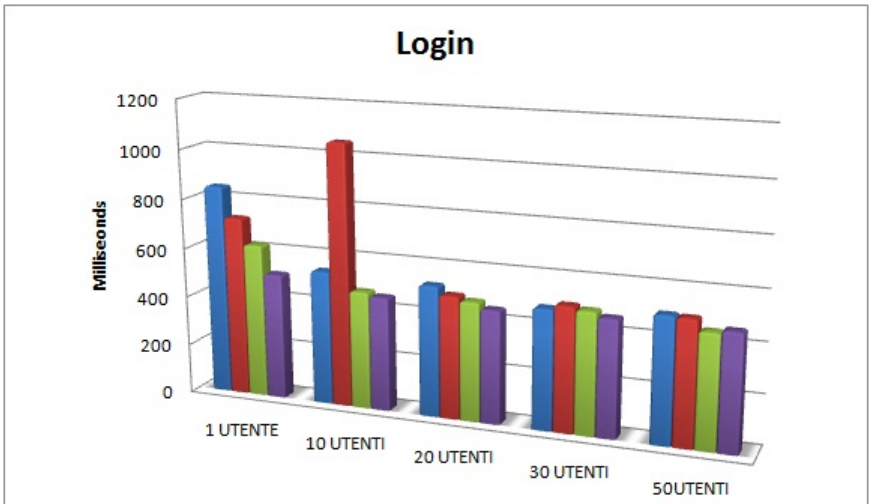
Una possibile motivazione di questi ritardi potrebbe essere la migrazione della VM ospitante l'applicazione su una macchina differente, che ha causato un blocco momentaneo nell'esecuzione delle richieste. Questo problema, come mostrato nel grafico di Figura 7.39, non si è verificato nel caso del test della versione con CPIM.

Confronto utilizzo medio CPU

Anche per quanto riguarda questo Test Plan si sono avuti dei buoni riscontri riguardo l'overhead computazionale introdotto dalla libreria.

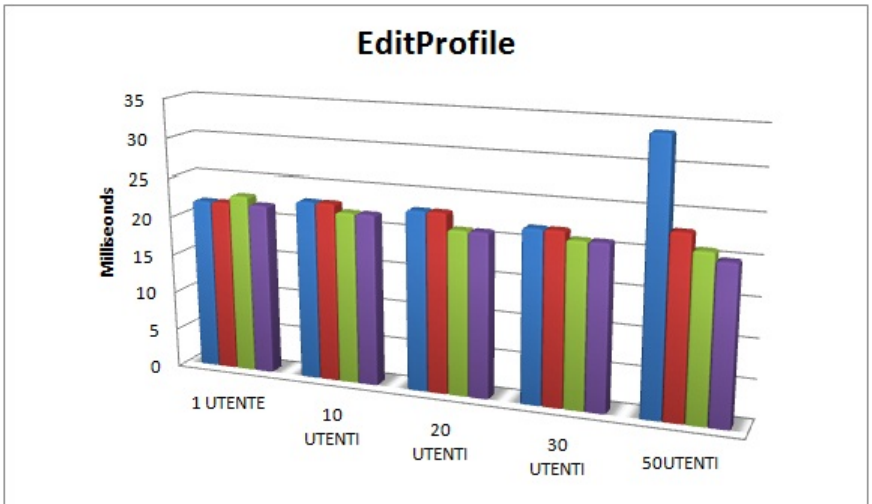
Dal grafico di Figura 7.40, si nota come l'utilizzo medio della CPU da parte del frontend delle due versioni è praticamente il medesimo. A conferma che in condizioni di basso carico l'overhead computazionale introdotto dalla libreria CPIM è pressoché trascurabile.

Buoni risultati sono stati ottenuti anche per il backend (grafico in Figura 7.41), dove l'overhead introdotto è ancora risultato essere sempre inferiore al 5%, anche in condizioni di alto carico. Quest'ultimo risultato è un'ulteriore conferma del fatto che l'emulazione del servizio Task Queue, utilizzato per la gestione del calcolo della similarità nella versione CPIM e risiedente nel backend, ha delle buone performance.



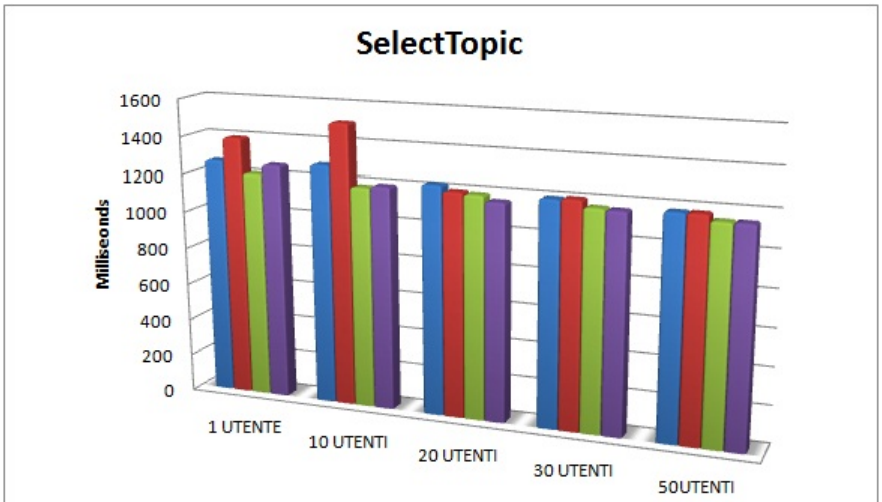
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	850	542	527	482	503
Nativa - AVG	727	1059	495	501	499
CPIM - MEDIAN	623	475	479	489	456
Nativa - MEDIAN	510	460	455	469	467

Figura 7.28: Confronto tempi medi di latenza per la richiesta LOGIN



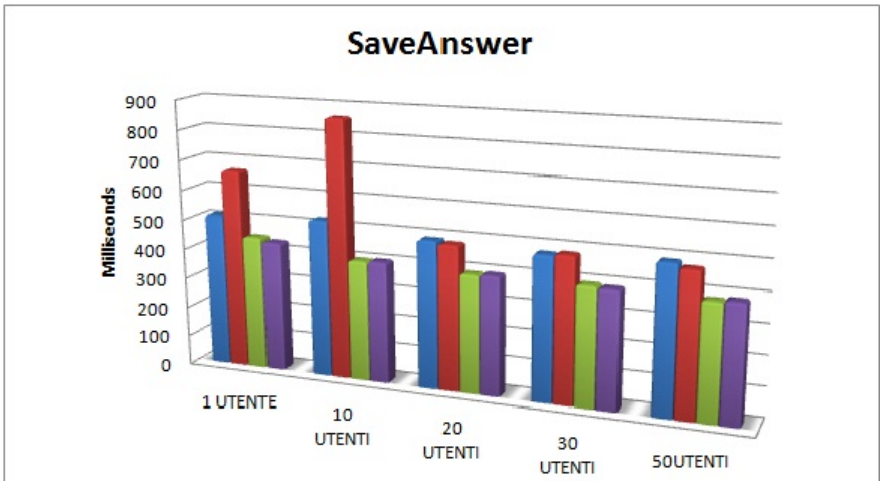
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	22	23	23	22	34
Nativa - AVG	22	23	23	22	23
CPIM - MEDIAN	23	22	21	21	21
Nativa - MEDIAN	22	22	21	21	20

Figura 7.29: Confronto tempi medi di latenza per la richiesta EDITPROFILE



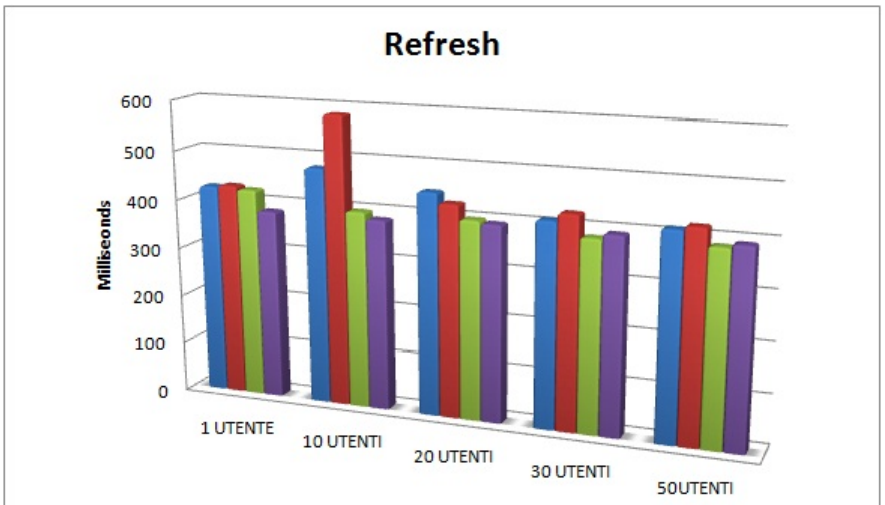
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	1272	1286	1229	1201	1182
Nativa - AVG	1398	1510	1200	1207	1183
CPIM - MEDIAN	1213	1187	1193	1172	1152
Nativa - MEDIAN	1266	1199	1164	1167	1153

Figura 7.30: Confronto tempi medi di latenza per la richiesta SELECTTOPIC



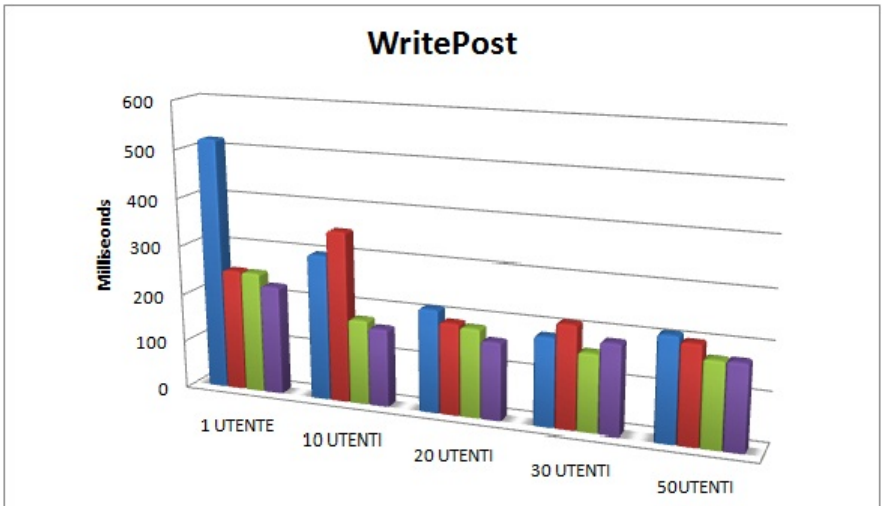
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	515	526	492	480	491
Nativa - AVG	667	863	484	485	476
CPIM - MEDIAN	448	402	394	396	386
Nativa - MEDIAN	436	405	397	391	391

Figura 7.31: Confronto tempi medi di latenza per la richiesta SAVEANSWERS



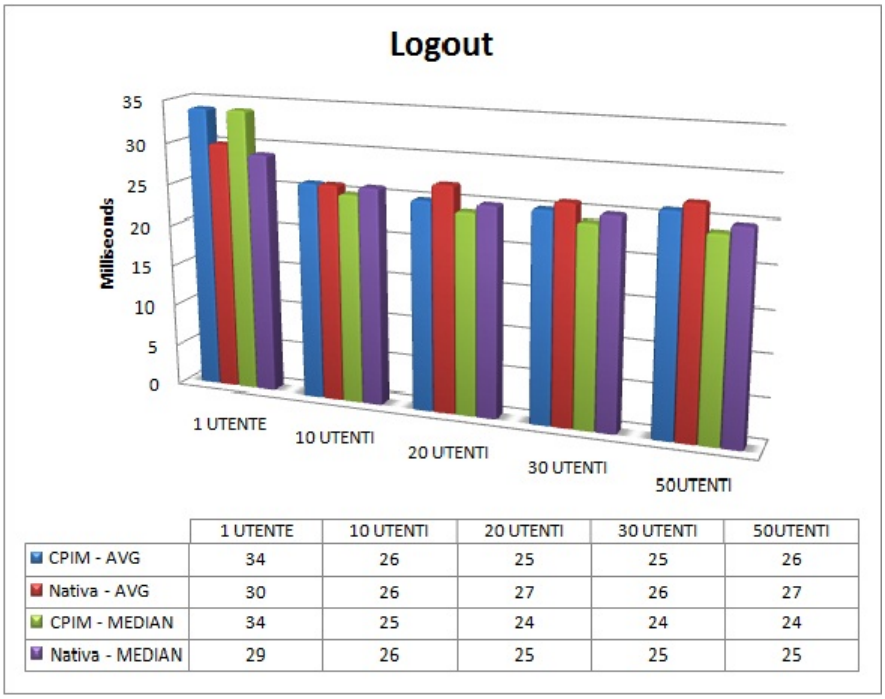
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	427	479	448	411	413
Nativa - AVG	431	585	428	427	421
CPIM - MEDIAN	425	398	400	384	387
Nativa - MEDIAN	384	385	395	394	394

Figura 7.32: Confronto tempi medi di latenza per la richiesta REFRESH



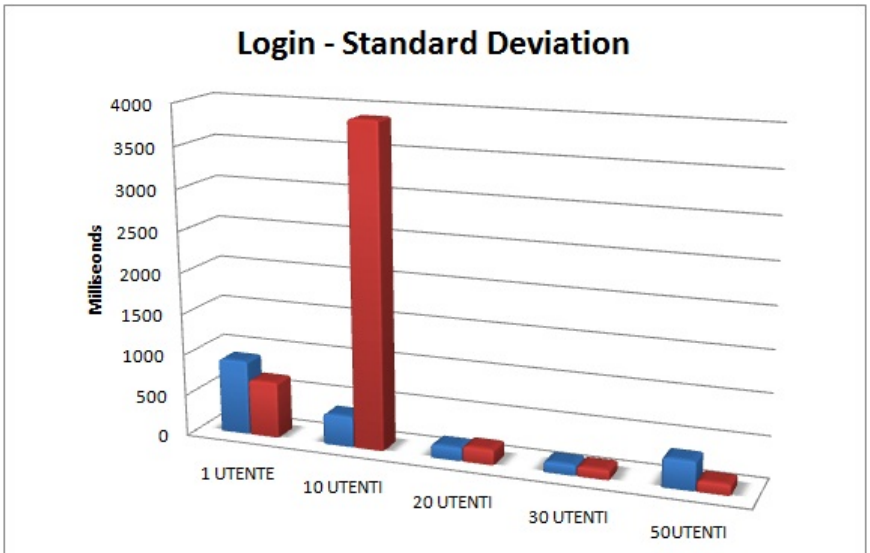
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	521	300	214	183	212
Nativa - AVG	253	352	190	213	201
CPIM - MEDIAN	252	176	184	160	173
Nativa - MEDIAN	227	162	161	185	174

Figura 7.33: Confronto tempi medi di latenza per la richiesta WRITEPOST



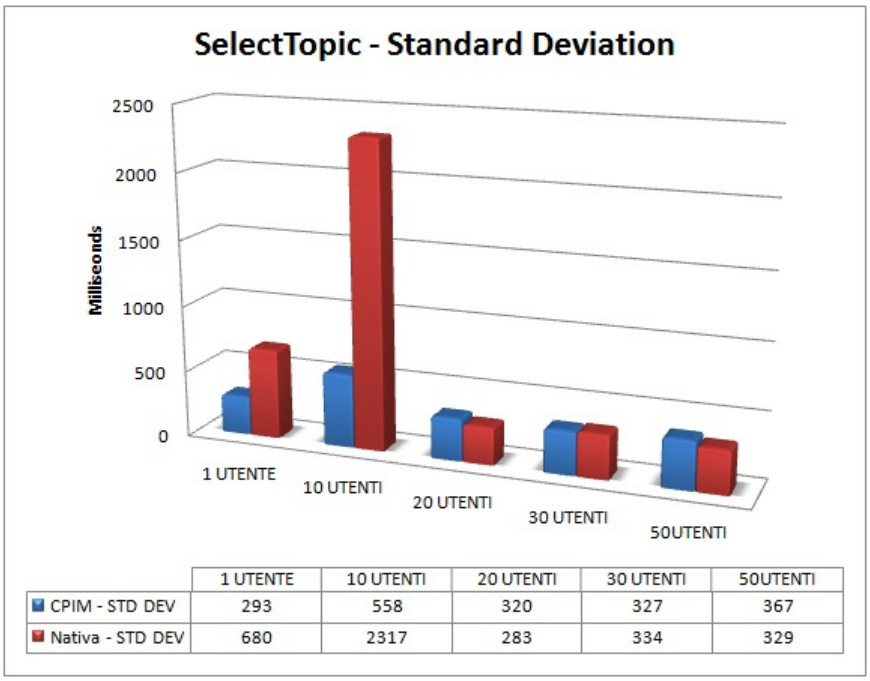
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - AVG	34	26	25	25	26
Nativa - AVG	30	26	27	26	27
CPIM - MEDIAN	34	25	24	24	24
Nativa - MEDIAN	29	26	25	25	25

Figura 7.34: Confronto tempi medi di latenza per la richiesta LOGOUT



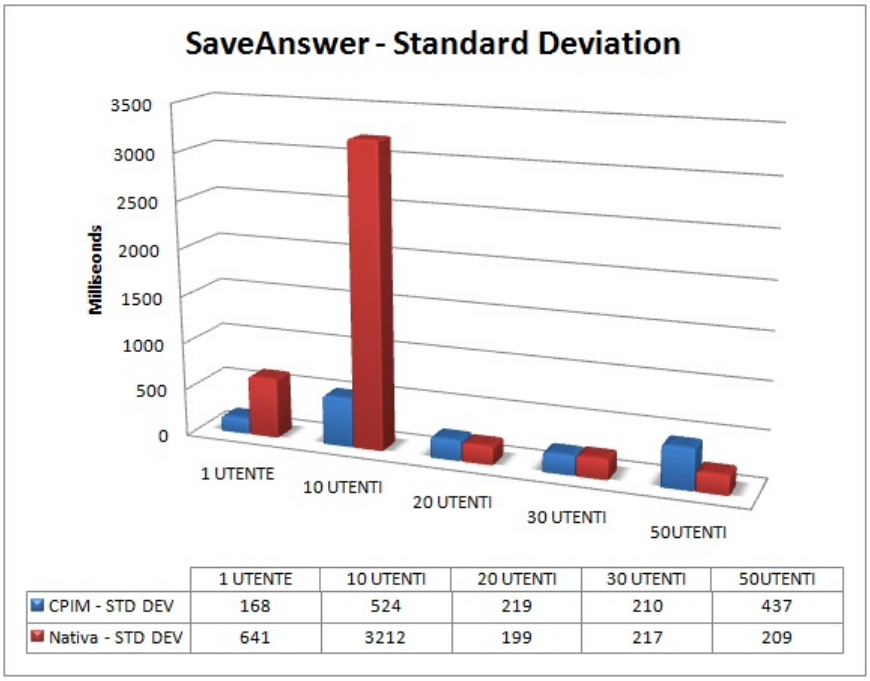
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - STD DEV	908	374	161	130	350
Nativa - STD DEV	684	3895	188	120	132

Figura 7.35: Confronto Deviazione Standard tempi di latenza per la LOGIN



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - STD DEV	293	558	320	327	367
Nativa - STD DEV	680	2317	283	334	329

Figura 7.36: Confronto Deviazione Standard tempi di latenza per la SELECTTOPIC



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50UTENTI
CPIM - STD DEV	168	524	219	210	437
Nativa - STD DEV	641	3212	199	217	209

Figura 7.37: Confronto Deviazione Standard tempi di latenza per la SAVEANSWERS

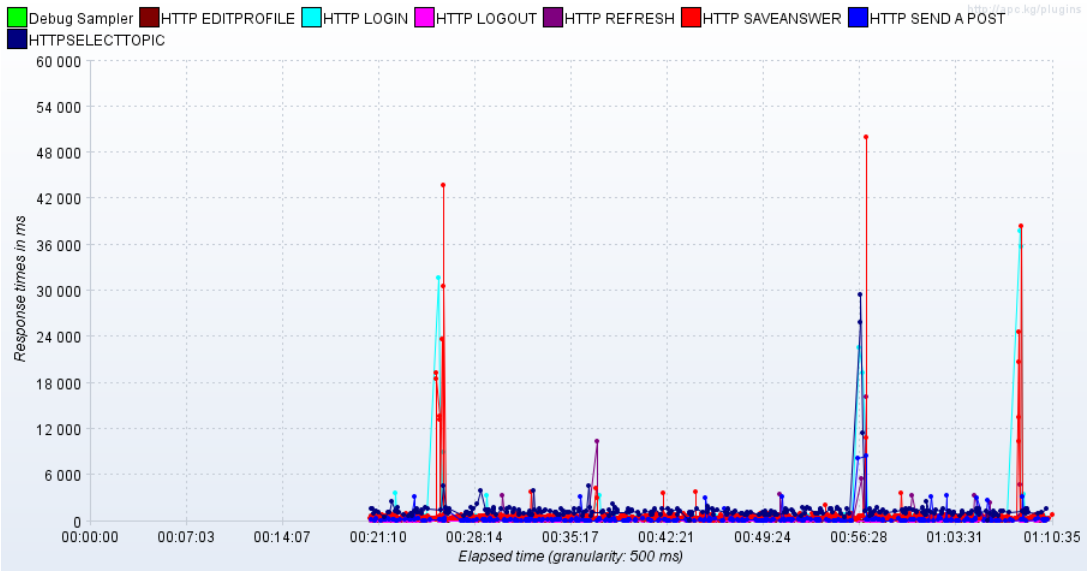


Figura 7.38: Grafico dei Response Time per caso 10 utenti versione Nativa

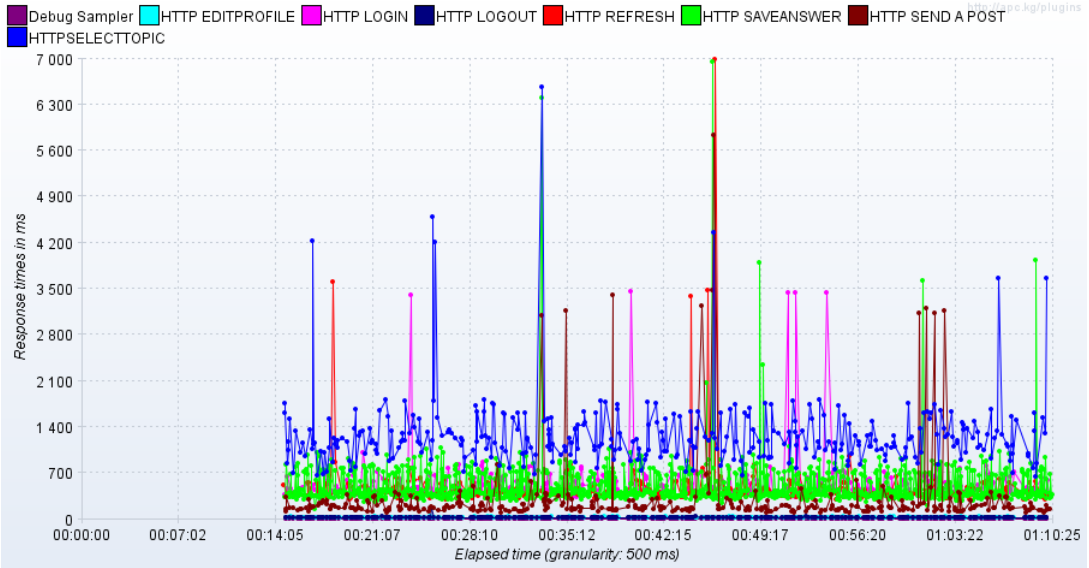


Figura 7.39: Grafico dei Response Time per caso 10 utenti versione con CPIM

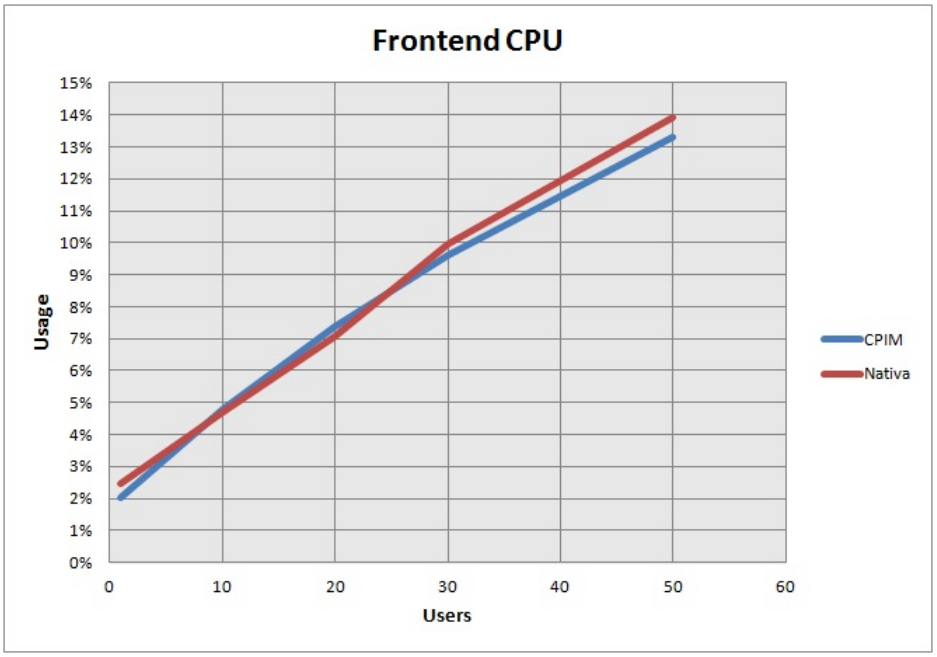


Figura 7.40: Confronto utilizzo medio CPU - Frontend

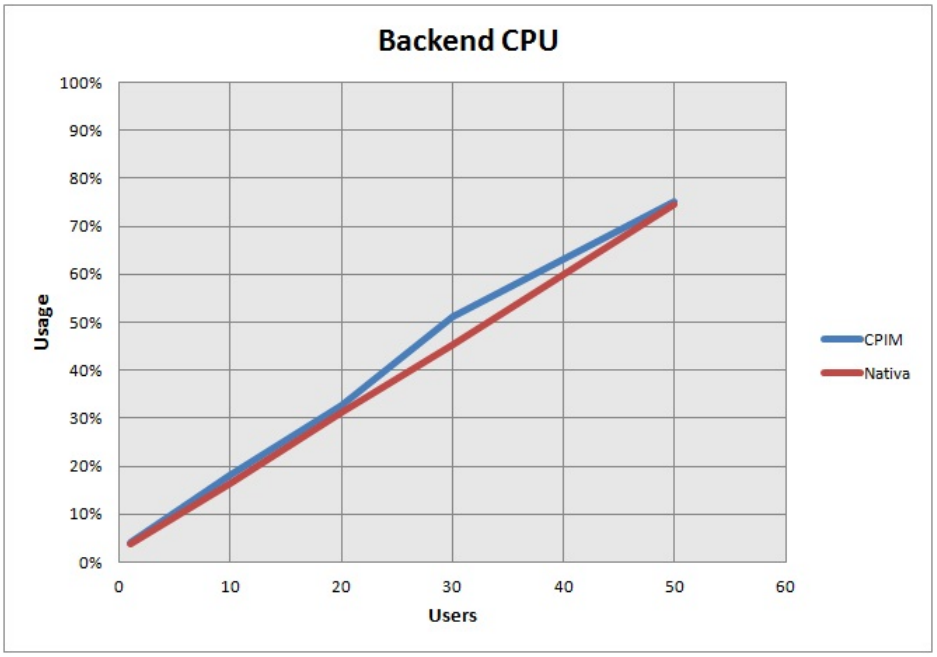


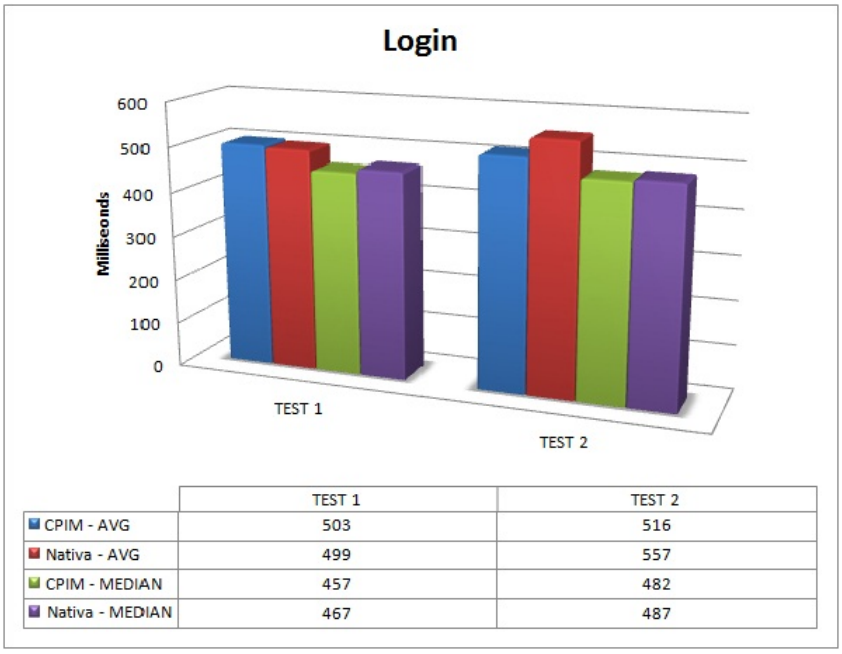
Figura 7.41: Confronto utilizzo medio CPU - Backend

Replica Test 50 utenti

Anche per questo Test Plan, per verificare la variabilità dei test ad alto carico, si è deciso di replicare il test con 50 utenti.

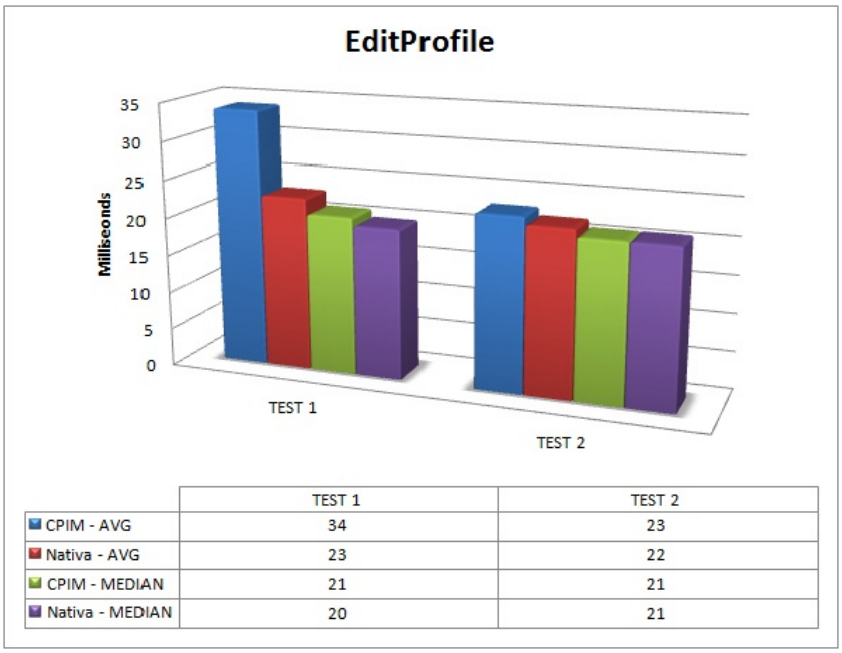
Dal confronto dei test non emergono particolari differenze riguardo le latenze registrate tra la versione con CPMI e quella nativa.

I grafici di confronto delle latenze sono mostrati nelle Figure 7.42, 7.43, 7.44, 7.45, 7.46 e 7.47, mentre quelli relativi all'utilizzo della CPU nelle Figure 7.49 e 7.50.



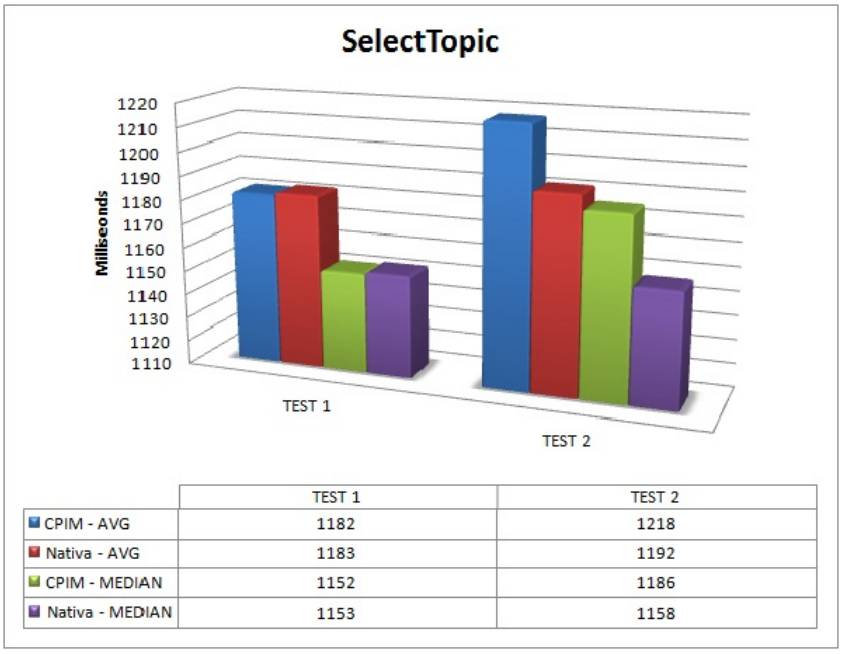
	TEST 1	TEST 2
CPIM - AVG	503	516
Nativa - AVG	499	557
CPIM - MEDIAN	457	482
Nativa - MEDIAN	467	487

Figura 7.42: Confronto test 50 utenti: latenza richieste LOGIN



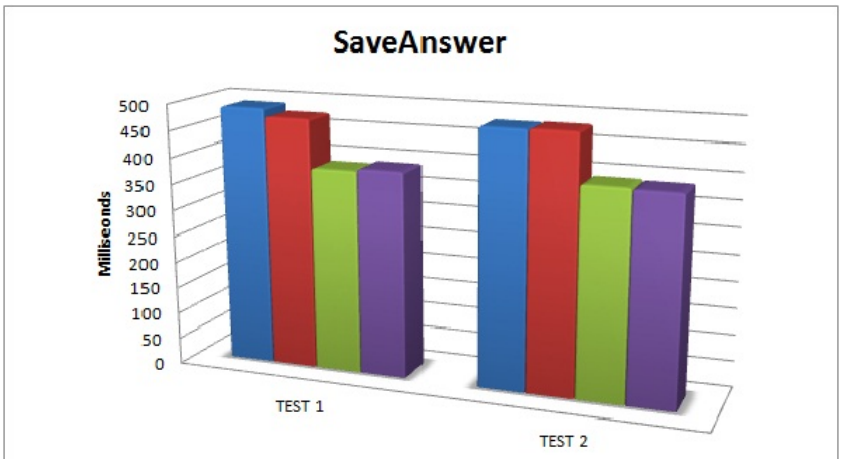
	TEST 1	TEST 2
CPIM - AVG	34	23
Nativa - AVG	23	22
CPIM - MEDIAN	21	21
Nativa - MEDIAN	20	21

Figura 7.43: Confronto test 50 utenti: latenza richieste EDITPROFILE



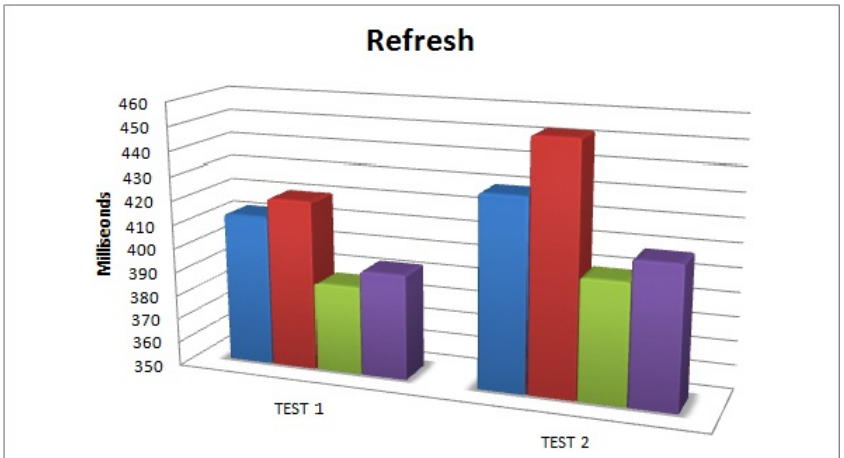
	TEST 1	TEST 2
CPIM - AVG	1182	1218
Nativa - AVG	1183	1192
CPIM - MEDIAN	1152	1186
Nativa - MEDIAN	1153	1158

Figura 7.44: Confronto test 50 utenti: latenza richieste SELECTTOPIC



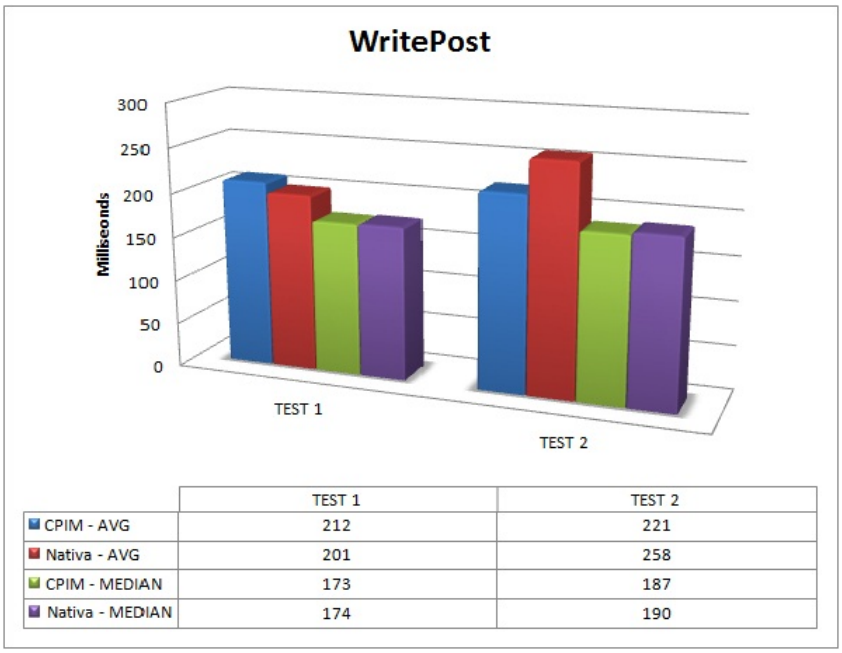
	TEST 1	TEST 2
CPIM - AVG	491	482
Nativa - AVG	476	483
CPIM - MEDIAN	386	392
Nativa - MEDIAN	391	390

Figura 7.45: Confronto test 50 utenti: latenza richieste SAVEANSWERS



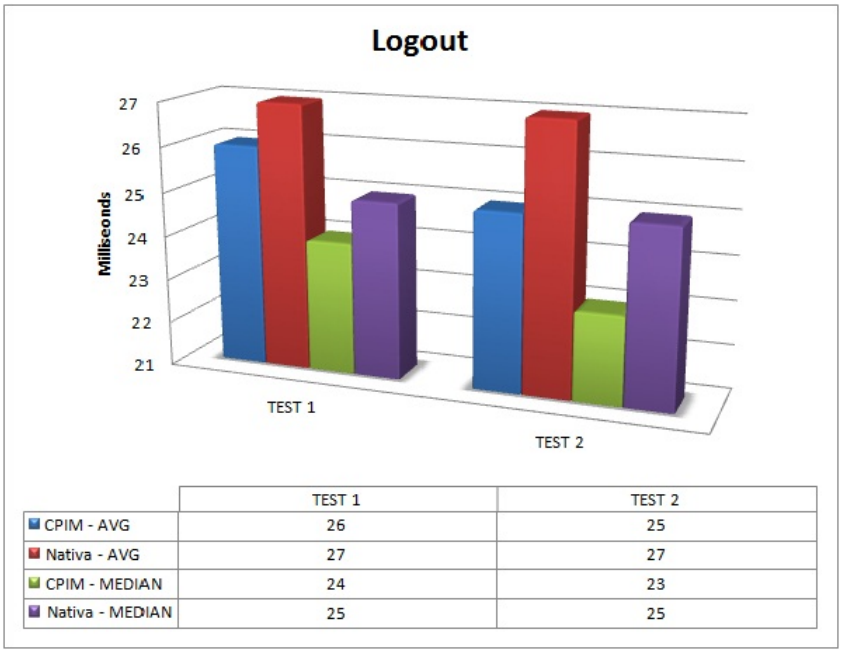
	TEST 1	TEST 2
CPIM - AVG	413	430
Nativa - AVG	421	453
CPIM - MEDIAN	387	400
Nativa - MEDIAN	394	409

Figura 7.46: Confronto test 50 utenti: latenza richieste REFRESH



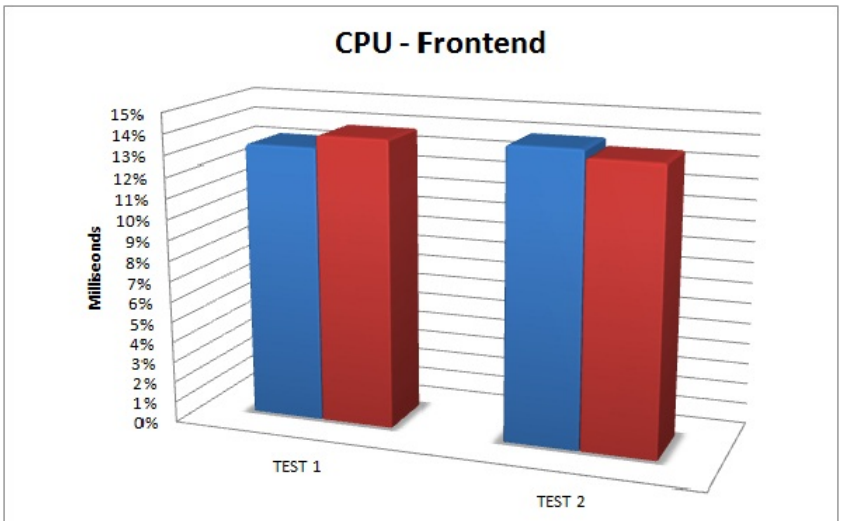
	TEST 1	TEST 2
CPIM - AVG	212	221
Nativa - AVG	201	258
CPIM - MEDIAN	173	187
Nativa - MEDIAN	174	190

Figura 7.47: Confronto test 50 utenti: latenza richieste WRITEPOST



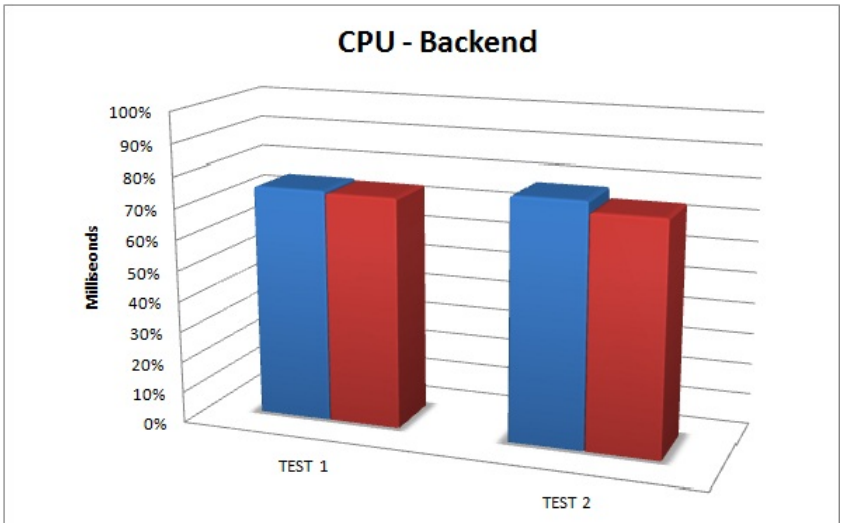
	TEST 1	TEST 2
CPIM - AVG	26	25
Nativa - AVG	27	27
CPIM - MEDIAN	24	23
Nativa - MEDIAN	25	25

Figura 7.48: Confronto test 50 utenti: latenza richieste LOGOUT



	TEST 1	TEST 2
CPIM	13,29%	14,00%
Nativa	13,90%	13,62%

Figura 7.49: Confronto test 50 utenti: utilizzo medio CPU Frontend



	TEST 1	TEST 2
CPIM	75,19%	78,11%
Nativa	74,58%	74,70%

Figura 7.50: Confronto test 50 utenti: utilizzo medio CPU Backend

7.4.2 Risultati per App Engine

I test sono stati effettuati deployando MiC in due istanze (una per il frontend e l'altra per il backend) di dimensioni F4, aventi le seguenti caratteristiche:

- CPU: 2400 MHz
- Memoria RAM: 512 MB

Per quanto riguarda le configurazioni relative alle performance minime richieste dall'applicazione:

- Numero delle Idle Instance: il range utilizzato per i test è (Minimo=Automatic, Massimo=1).
- Parametro Pending Latency: il range utilizzato per i test è (Minimo=15 sec, Massimo=Automatic).

Il contenuto dei sistemi di storage, mantenuto equivalente per tutti i test di entrambe le versioni di MiC, è stato di:

- Relativamente al servizio Google Cloud SQL:
  - Numero Tuple tabella Message: 3700 circa
  - Numero Tuple tabella UserProfile: 2400 circa
  - Numero Tuble tabella UserSimilarity: 8000 circa
- Relativamente al Datastore:
  - Numero entità UserRatings: 8000 circa
  - Numero entità Topic: 7

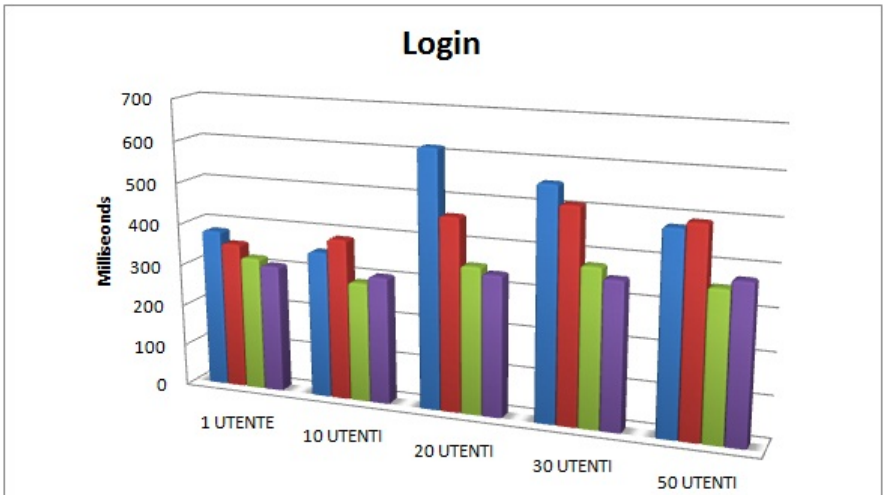
Confronto tempi di Latenza

Dai grafici riportati nei nelle Figure 7.51, 7.52, 7.53, 7.54, 7.55, 7.56, 7.57, che riportano le misure effettuate da JMeter, non si osservano differenze sostanziali tra le richieste che utilizzano lo strato di astrazione CPIM e quelle che utilizzano le API native di App Engine.

Confronto utilizzo medio CPU

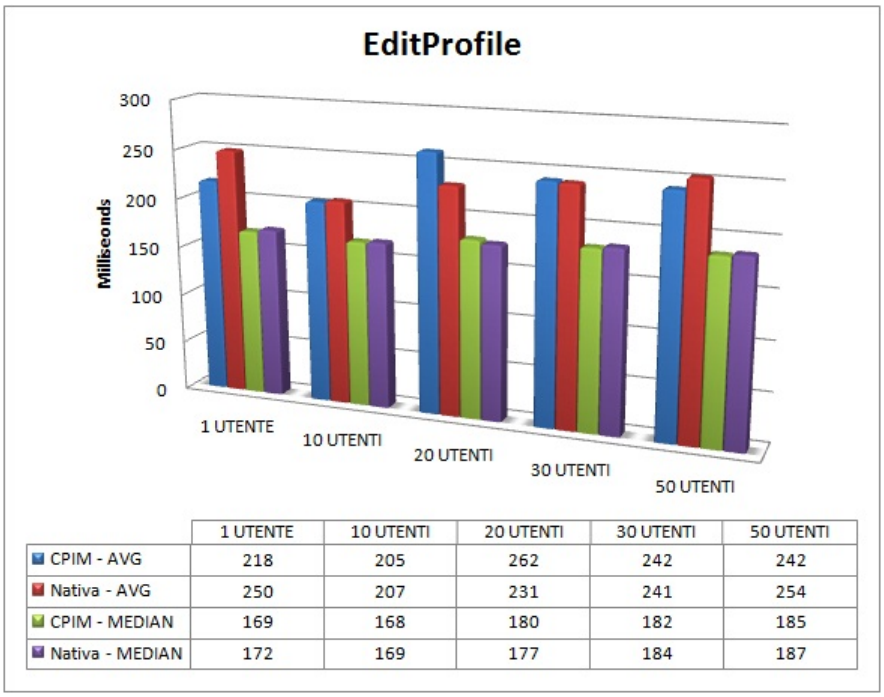
Anche per quanto riguarda App Engine si sono avuti dei buoni riscontri rispetto l'overhead computazionale introdotto dalla libreria.

Dal grafico di Figura 7.58, si nota come l'overhead computazionale introdotto dalla libreria sia al massimo dell'8% per quanto riguarda il frontend e quasi trascurabile per quanto riguarda il tier di backend.



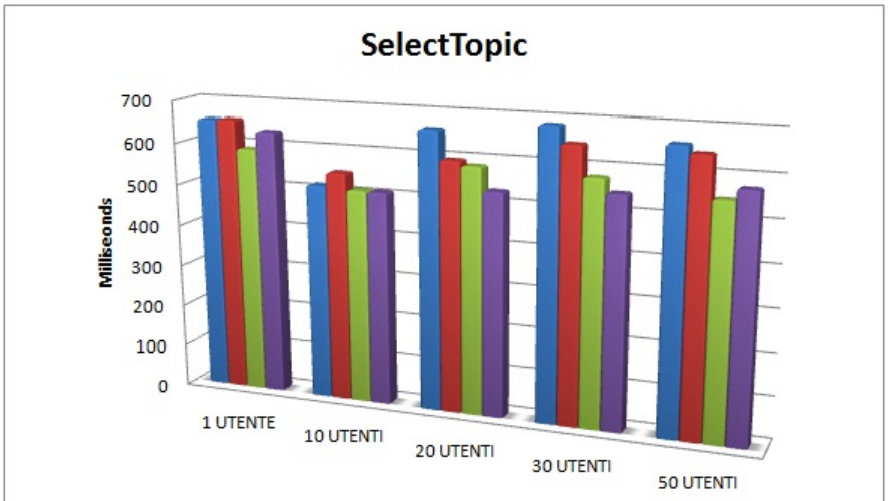
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	382	353	618	555	479
Nativa - AVG	354	388	464	511	494
CPIM - MEDIAN	323	288	353	378	355
Nativa - MEDIAN	308	306	338	353	376

Figura 7.51: Confronto tempi medi di latenza per la richiesta LOGIN



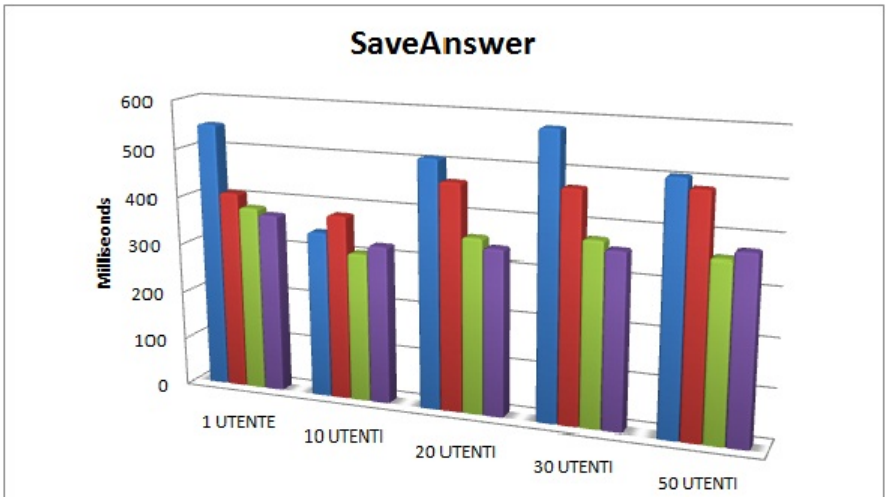
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	218	205	262	242	242
Nativa - AVG	250	207	231	241	254
CPIM - MEDIAN	169	168	180	182	185
Nativa - MEDIAN	172	169	177	184	187

Figura 7.52: Confronto tempi medi di latenza per la richiesta EDIT PROFILE



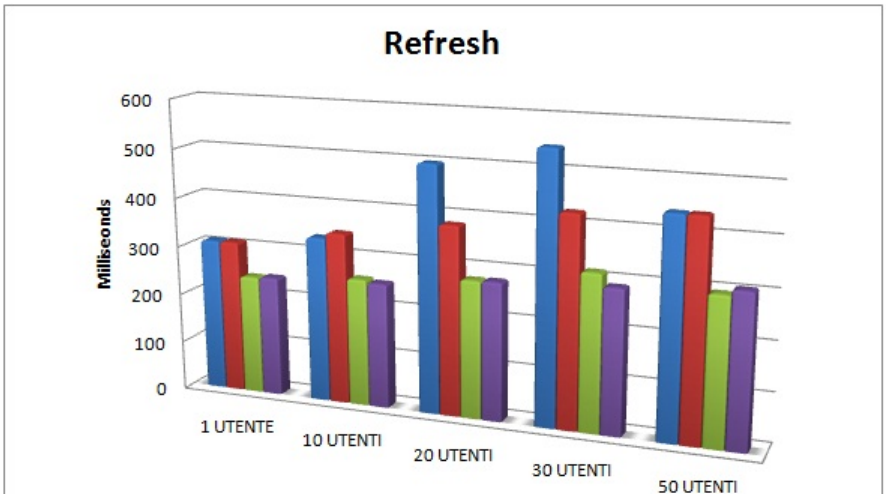
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	656	516	663	688	662
Nativa - AVG	658	549	596	649	645
CPIM - MEDIAN	591	511	586	579	550
Nativa - MEDIAN	634	510	532	547	576

Figura 7.53: Confronto tempi medi di latenza per la richiesta SELECT TOPICS



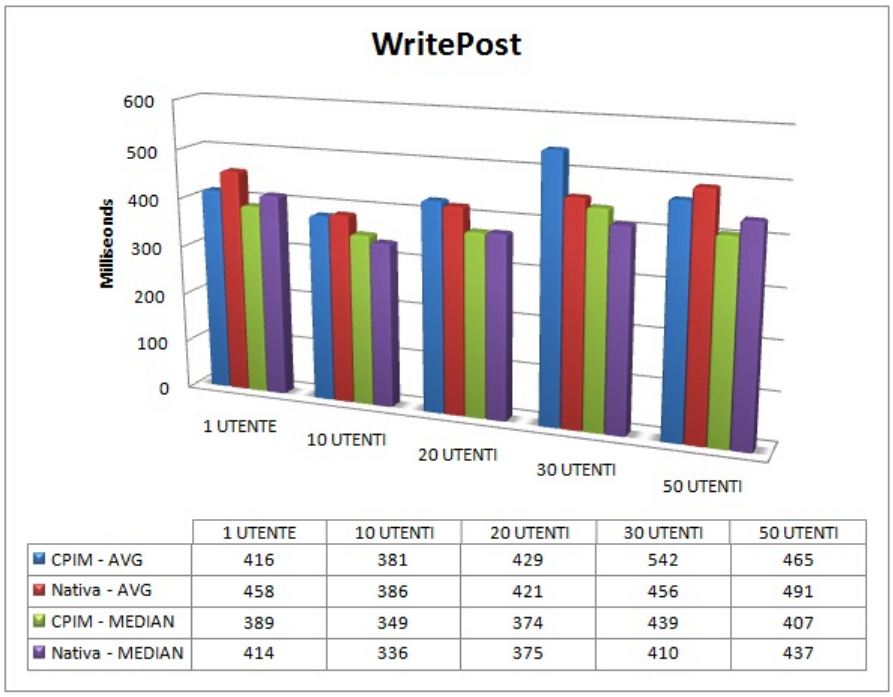
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	550	343	510	582	506
Nativa - AVG	410	381	466	471	486
CPIM - MEDIAN	382	306	359	375	360
Nativa - MEDIAN	370	325	341	358	376

Figura 7.54: Confronto tempi medi di latenza per la richiesta SAVE ANSWERS



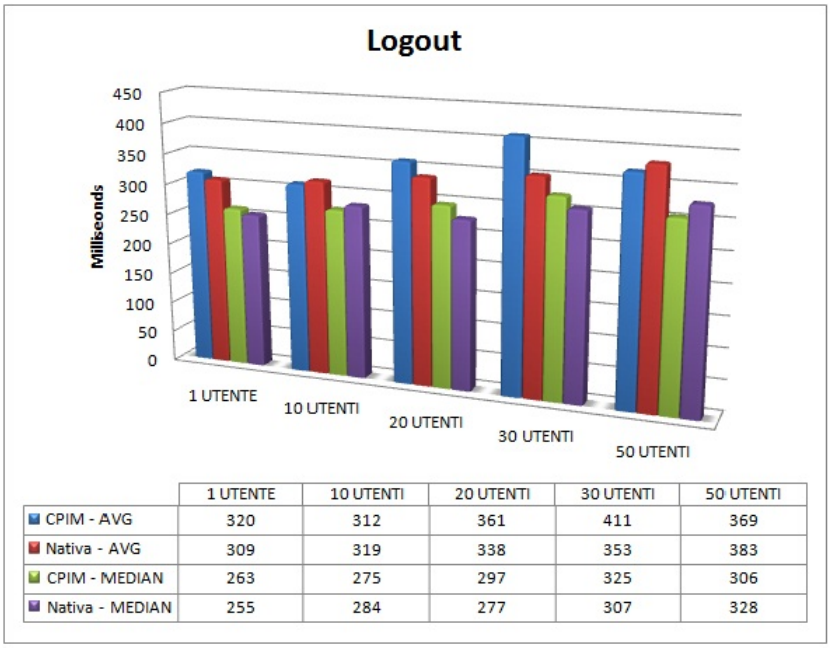
	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	311	336	500	545	440
Nativa - AVG	311	348	384	426	440
CPIM - MEDIAN	242	259	279	316	297
Nativa - MEDIAN	244	254	280	291	308

Figura 7.55: Confronto tempi medi di latenza per la richiesta REFRESH



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	416	381	429	542	465
Nativa - AVG	458	386	421	456	491
CPIM - MEDIAN	389	349	374	439	407
Nativa - MEDIAN	414	336	375	410	437

Figura 7.56: Confronto tempi medi di latenza per la richiesta WRITE POST



	1 UTENTE	10 UTENTI	20 UTENTI	30 UTENTI	50 UTENTI
CPIM - AVG	320	312	361	411	369
Nativa - AVG	309	319	338	353	383
CPIM - MEDIAN	263	275	297	325	306
Nativa - MEDIAN	255	284	277	307	328

Figura 7.57: Confronto tempi medi di latenza per la richiesta LOGOUT

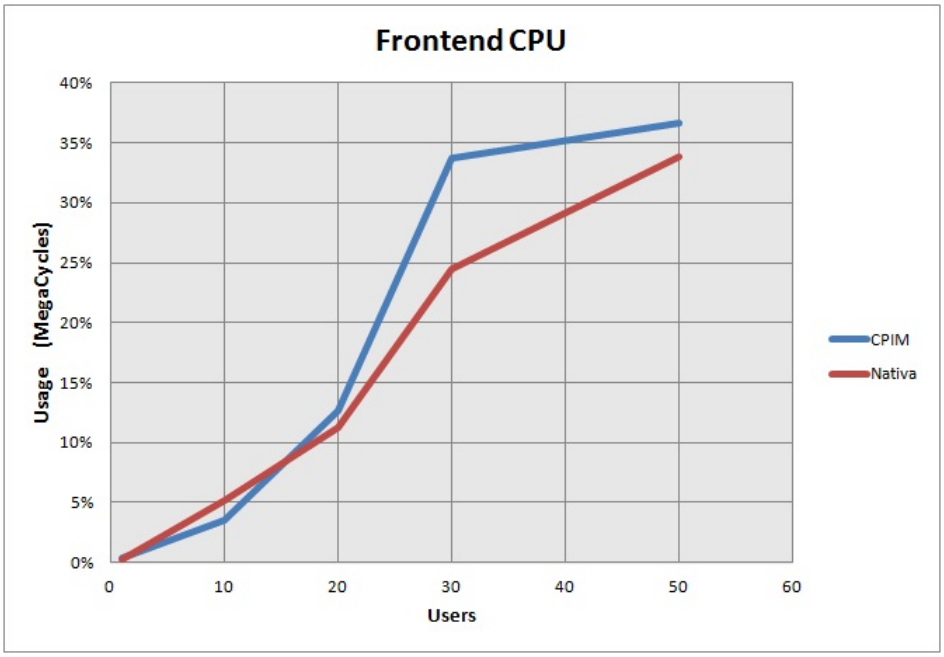


Figura 7.58: Confronto utilizzo medio CPU - Frontend

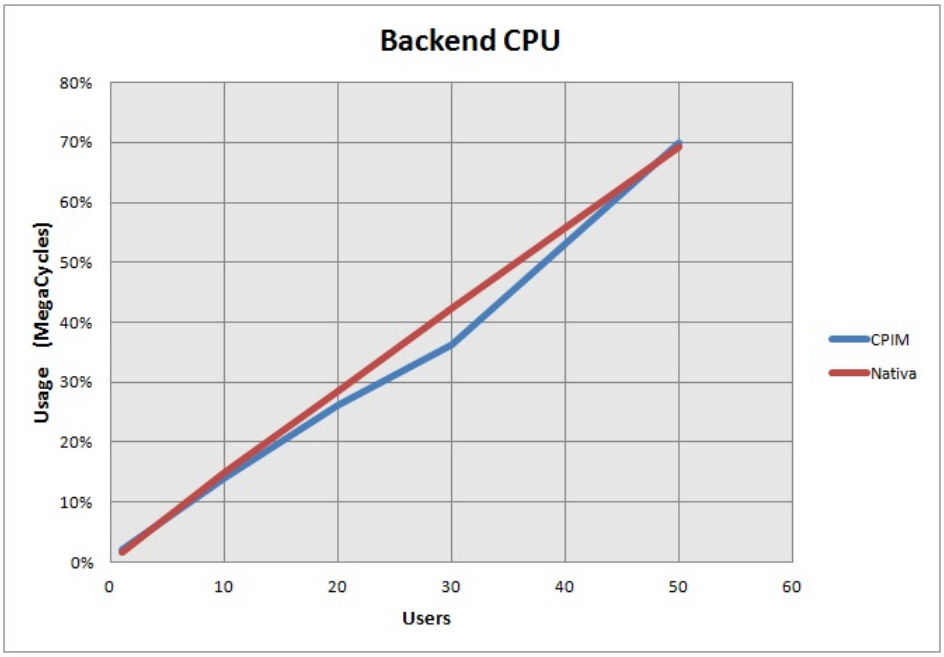


Figura 7.59: Confronto utilizzo medio CPU - Backend

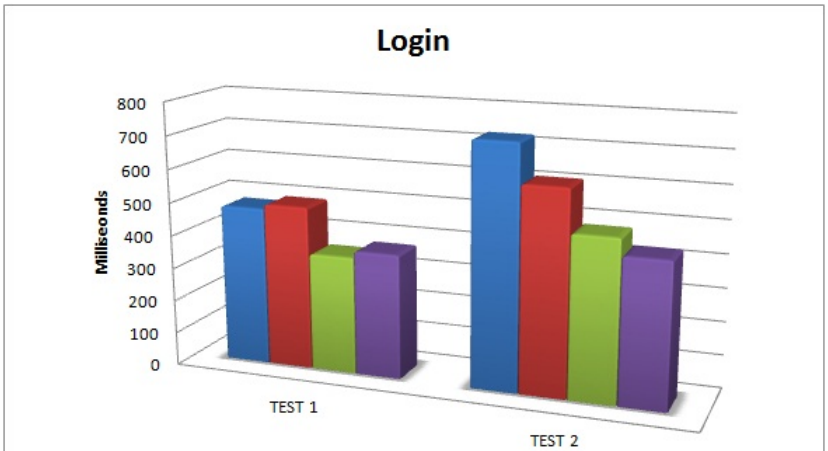
Replica Test 50 utenti

Anche per quanto riguarda la piattaforma di Google, per verificare la variabilità dei test ad alto carico, si è deciso di replicare il test con 50 utenti.

I grafici di confronto delle latenze sono mostrati nelle Figure 7.60, 7.61, 7.62, 7.63, 7.64 e 7.65, mentre quelli relativi all'utilizzo della CPU nelle Figure 7.67 e 7.68.

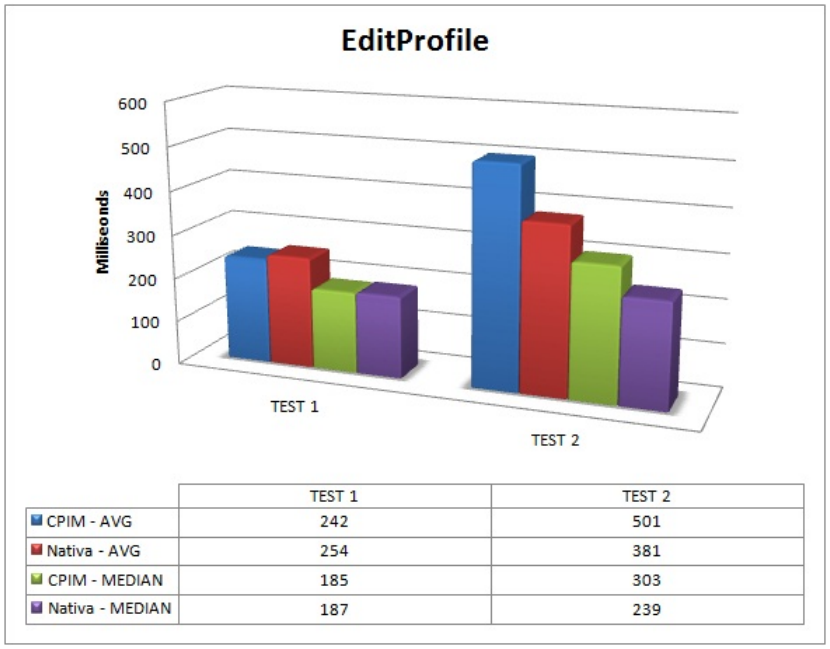
Per entrambe le versioni i tempi di latenza risultano più alti nel secondo test, ma il dato che interessa del confronto tra le due, conferma che l'overhead introdotto dalla libreria risulta essere in media di 100 millisecondi.

Relativamente all'utilizzo della CPU, si è osservato un sostanziale equilibrio tra le due versioni, la cui differenza non va mai al di sopra del 5%.



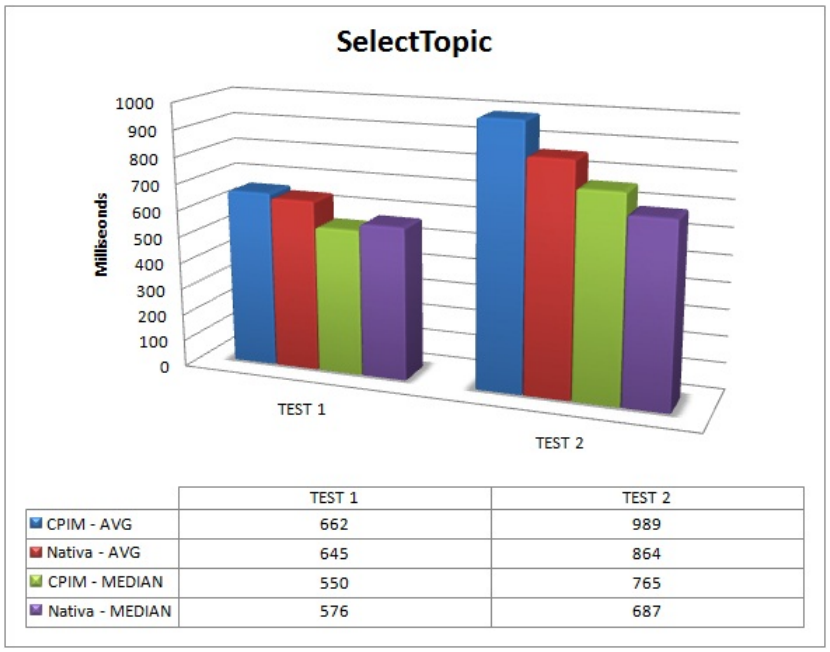
	TEST 1	TEST 2
CPIM - AVG	479	729
Nativa - AVG	494	612
CPIM - MEDIAN	355	484
Nativa - MEDIAN	376	434

Figura 7.60: Confronto test 50 utenti: latenza richieste LOGIN



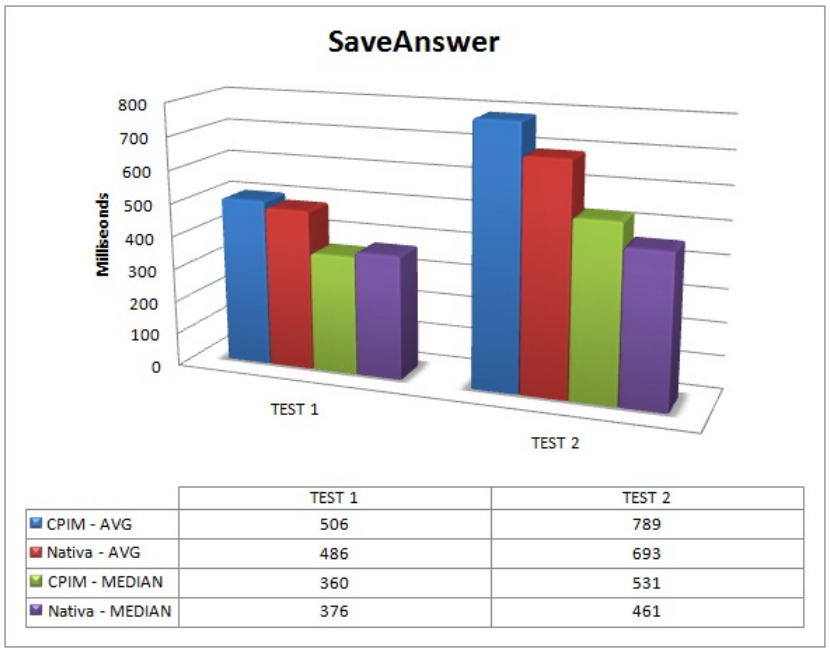
	TEST 1	TEST 2
CPIM - AVG	242	501
Nativa - AVG	254	381
CPIM - MEDIAN	185	303
Nativa - MEDIAN	187	239

Figura 7.61: Confronto test 50 utenti: latenza richieste EDITPROFILE



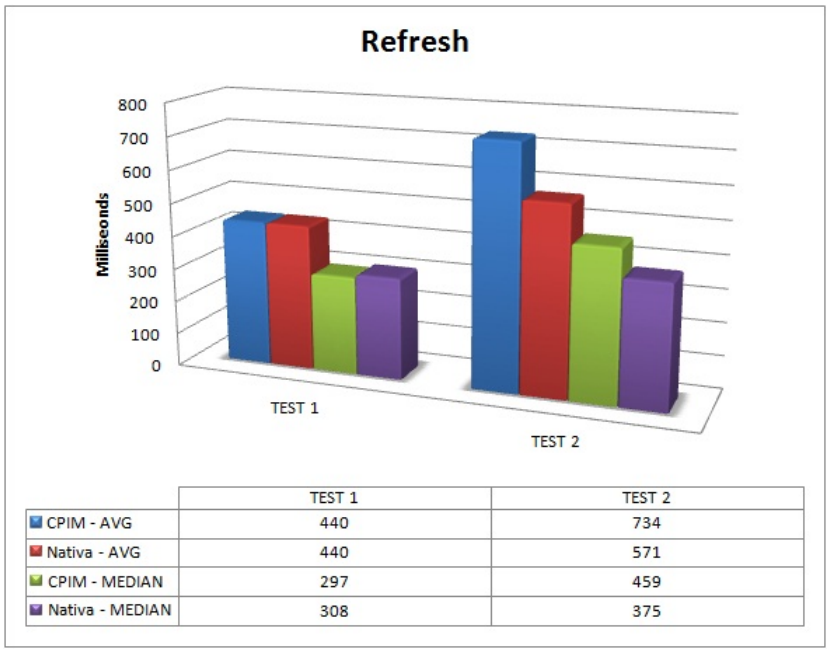
	TEST 1	TEST 2
CPIM - AVG	662	989
Nativa - AVG	645	864
CPIM - MEDIAN	550	765
Nativa - MEDIAN	576	687

Figura 7.62: Confronto test 50 utenti: latenza richieste SELECTTOPIC



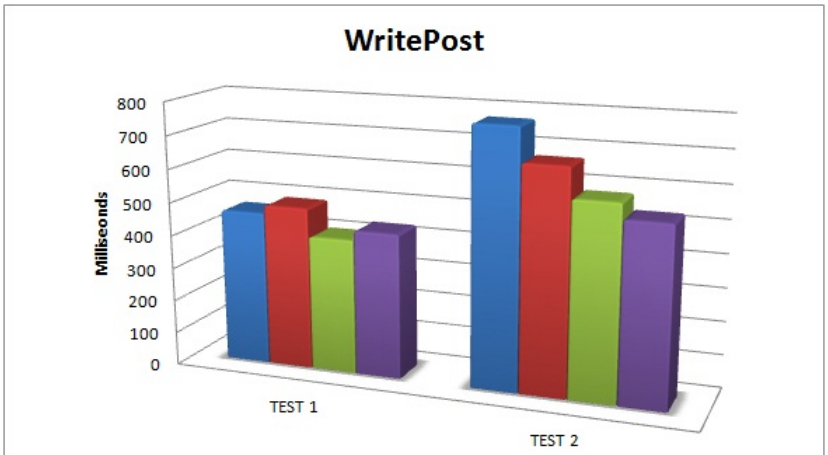
	TEST 1	TEST 2
CPIM - AVG	506	789
Nativa - AVG	486	693
CPIM - MEDIAN	360	531
Nativa - MEDIAN	376	461

Figura 7.63: Confronto test 50 utenti: latenza richieste SAVEANSWERS



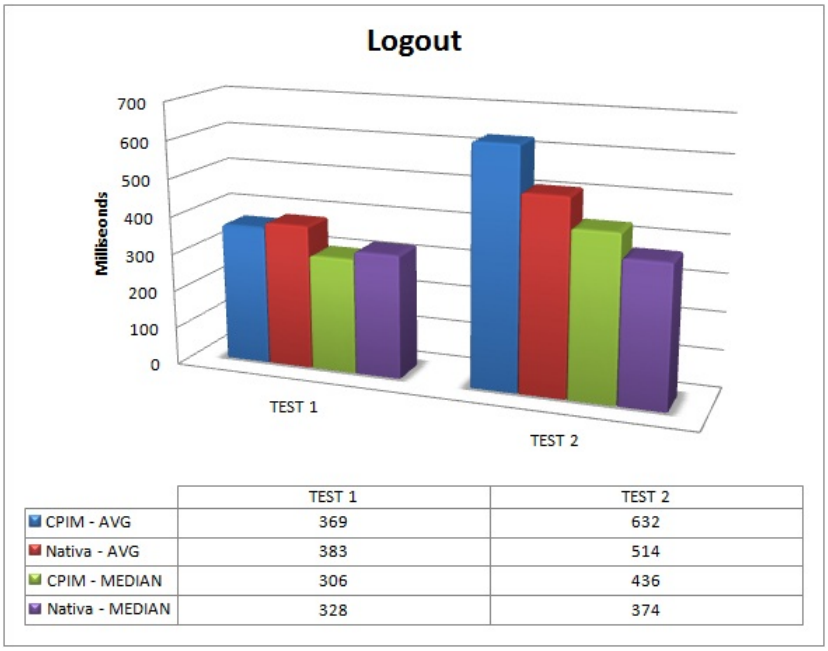
	TEST 1	TEST 2
CPIM - AVG	440	734
Nativa - AVG	440	571
CPIM - MEDIAN	297	459
Nativa - MEDIAN	308	375

Figura 7.64: Confronto test 50 utenti: latenza richieste REFRESH



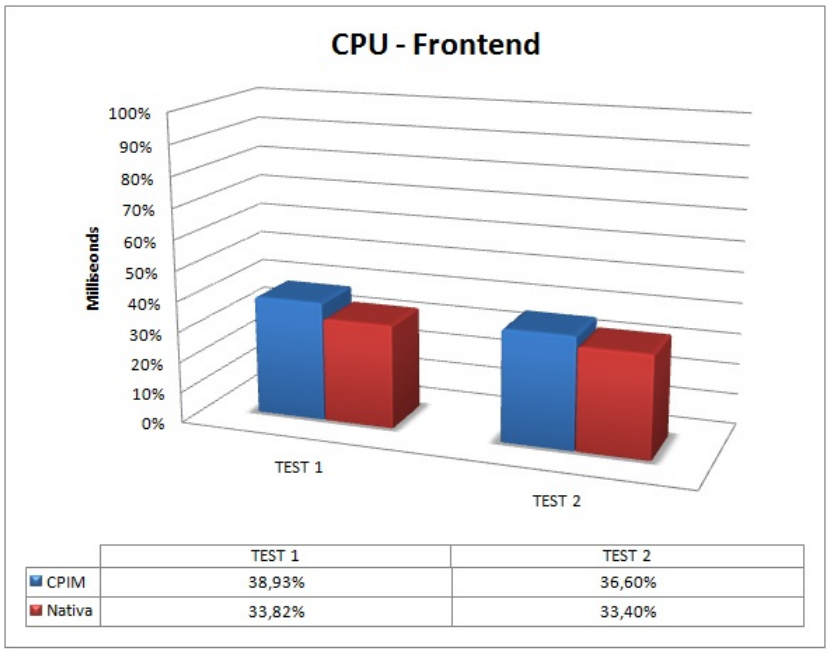
	TEST 1	TEST 2
CPIM - AVG	465	773
Nativa - AVG	491	671
CPIM - MEDIAN	407	581
Nativa - MEDIAN	437	535

Figura 7.65: Confronto test 50 utenti: latenza richieste WRITEPOST



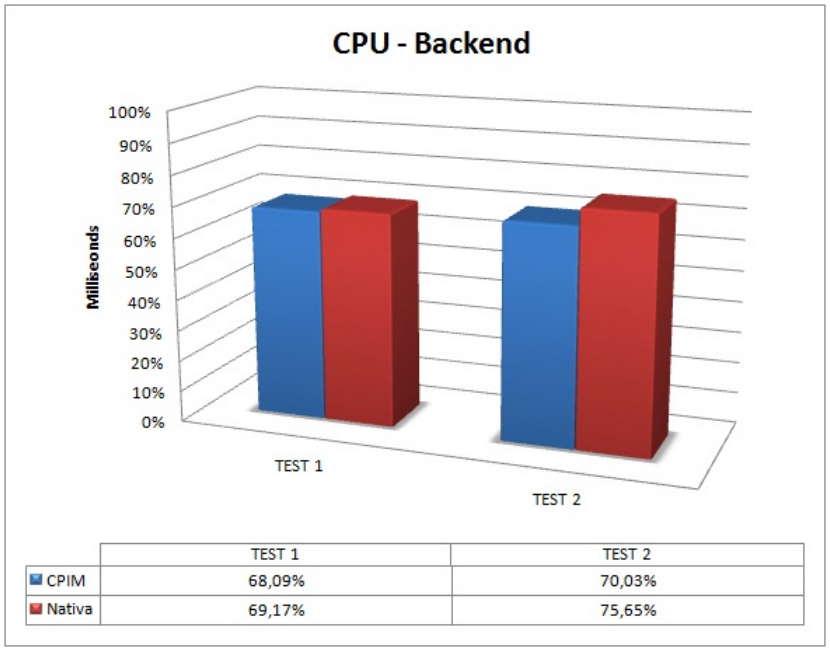
	TEST 1	TEST 2
CPIM - AVG	369	632
Nativa - AVG	383	514
CPIM - MEDIAN	306	436
Nativa - MEDIAN	328	374

Figura 7.66: Confronto test 50 utenti: latenza richieste LOGOUT



	TEST 1	TEST 2
CPIM	38,93%	36,60%
Nativa	33,82%	33,40%

Figura 7.67: Confronto test 50 utenti: utilizzo medio CPU Frontend



	TEST 1	TEST 2
CPIM	68,09%	70,03%
Nativa	69,17%	75,65%

Figura 7.68: Confronto test 50 utenti: utilizzo medio CPU Backend

Capitolo 8

Conclusioni e sviluppi futuri

In questo elaborato di tesi si è presentato un approccio per consentire la portabilità di applicazioni in sistemi Cloud Computing, con particolare attenzione per sistemi PaaS.

Come descritto nel Capitolo 2, l'analisi dell'attuale stato dell'arte riguardante il Cloud Computing, ha dimostrato come una delle maggiori problematiche che sta limitando la diffusione massiccia di questo paradigma, soprattutto in ambito PaaS, sia la difficoltà di trasferire le proprie applicazioni deployate sul Cloud da un provider ad un altro, come e quando si desidera, a costi ragionevoli. Questo è dovuto essenzialmente dalla totale mancanza di standard relativi alle metodologie di accesso ai servizi offerti dalle varie piattaforme, che invece espongono API proprietarie, tra loro incompatibili, come interfaccia di accesso ad essi.

Nell'attuale contesto, chi volesse trasferire la propria applicazione da una piattaforma ad un'altra, dovrà probabilmente sostenere degli alti costi, dovuti alla riscrittura e reingegnerizzazione del codice applicativo. Questo fenomeno è universalmente riconosciuto come effetto ¨Lock-in¨.

Come contributo alla risoluzione di queste problematiche, nel Capitolo 4 è stato proposto l'approccio CPIM (Cloud Platform Independent Model), che introduce uno strato di astrazione rappresentato da una libreria che, esponendo API ¨Vendor-Independent¨, permette di utilizzare i più importanti e comuni servizi offerti da sistemi PaaS, senza preoccuparsi a design time su quale piattaforma effettivamente verrà deployata l'applicazione.

L'attuale versione della libreria supporta l'utilizzo di svariati servizi, che consentono di sviluppare applicazioni web complete e scalabili.

Le applicazioni che utilizzeranno il nostro strato di astrazione, potranno essere trasferite da un sistema ad un altro, senza la necessità di riscrivere parti sostanziali di codice, ma solo modificando alcuni file di configurazione.

L'approccio proposto è orientato al linguaggio Java che, data la sua grande diffusione, è supportato dalla quasi totalità dei PaaS disponibili sul mercato Cloud. Le piattaforme attualmente supportate dalla libreria CPIM sono quelle di Google App Engine e Windows Azure, che ricoprono ampiamente il mercato dei PaaS.

Da studi sui lavori presenti in letteratura non sono emersi lavori atti a permettere, o almeno agevolare, la portabilità di applicazioni tra questi due provider. Inoltre ci si differenzia dai lavori presenti in letteratura, come SimpleCloud e CSAL, perchè si offre l'astrazione e le relative API, non solo per servizi relativi allo storage, quali i servizi che gestiscono schemi NoSQL, SQL, Blob oppure Queue, ma anche per servizi di Memcache e di invio Mail.

Un ulteriore aspetto distintivo riguarda il supporto di due diverse semantiche di code: infatti si offre la possibilità di memorizzare non solo semplici messaggi, ma anche richieste strutturate per l'esecuzione di lavori, da eseguire in modo asincrono rispetto ai flussi web.

Per agevolare e guidare lo sviluppatore nelle tediose fasi di configurazione, si è implementato un Plugin per l'ambiente Eclipse, la cui progettazione e modalità di utilizzo è discussa nel Capitolo 5.

Per testare le qualità funzionali della libreria CPIM, si è implementata un'applicazione di esempio, denominata MiC (Meeting In the Cloud), descritta nel Capitolo 6, un piccolo social network che utilizza la libreria per interfacciarsi ai servizi Cloud utilizzati.

Utilizzando MiC come Test Case, si sono effettuati dei test per valutare l'overhead introdotto dall'utilizzo dello strato di astrazione, rispetto all'utilizzo diretto delle API proprietarie dei due vendor. A tale scopo si sono prodotte le versioni native dell'applicativo per entrambe le piattaforme, che utilizzano cioè le rispettive API proprietarie.

Come descritto nel Capitolo 7, le analisi dei risultati dei test hanno dimostrato che gli overhead introdotti dalla libreria, sia dal punto di vista dei tempi di latenza che di utilizzo della risorsa CPU, sono trascurabili, sia in condizioni di basso che di alto carico computazionale.

Possibili estensioni future della libreria CPIM potrebbero riguardare l'ampliamento della libreria CPIM, integrando ulteriori servizi offerti dalle due piattaforme ed estendendo i linguaggi di sviluppo supportati. Inoltre lo strato di astrazione potrebbe essere esteso a più piattaforme Cloud (Amazon, Heroku, etc...), aumentando così la portabilità delle applicazioni su più PaaS.

Un'estensione interessante potrebbe riguardare la possibilità di permette l'utilizzo di servizi di piattaforme diverse nell'ambito di una stessa applicazione, facendo divenire CPIM un modello non solo per la portabilità, ma anche di interoperabilità tra sistemi Cloud appartenenti a piattaforme diverse.

Bibliografia

[1] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Special pubblication 800-145, National Institute of Standards and Technology, Information Technology Laboratory, September 2011.

[2] Y. Jadeja and K. Modi. Cloud computing - concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, pages 877 –880, march 2012.

[3] S. Rajan and A. Jairath. Cloud computing: The fifth generation of computing. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 665 –667, june 2011.

[4] Amazon. Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[5] IBM. IBM Cloud Computing: IaaS. www.ibm.com/it/cloud/iaas.html.

[6] Google. Google App Engine. https://developers.google.com/appengine/.

[7] Microsoft. Windows azure. https://www.windowsazure.com/.

[8] Google. Gmail: Email from Google. https://mail.google.com/.

[9] Apple. iCloud. https://www.icloud.com/.

[10] Dropbox. Dropbox. https://www.dropbox.com/.

[11] Google. Google Drive. https://drive.google.com/.

[12] Microsoft. Microsoft SkyDrive. https://skydrive.live.com/.

[13] Amit Manghani. Key Characteristics of PaaS. *Cloudbook Journal*, Volume 2 Issue 1, 2011.

[14] Google and open source community. The Go Programming Language. http://golang.org/.

[15] Red Hat. OpenShift by Red Hat. https://openshift.redhat.com/.

[16] CloudFoundry. CloudFoundy. http://www.cloudfoundry.com/about.

[17] IEEE. Standard Glossary. http://www.ieee.org/education_careers /education/standards/standards_glossary.html.

[18] F. Gonidis, I. Paraskakis, and D. Kourtesis. Addressing the challenge of application portability in cloud platforms. In *Proceedings of the 7th South East European Doctoral Student Conference (DSC 2012)*, pages 565 –576, 2012.

[19] Robert B. Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. NIST Cloud Computing Reference Architecture. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, pages 594–596, Washington, DC, USA, 2011. IEEE Computer Society.

[20] G. Brunette and R. Mogull. Security Guidance for critical areas of focus in Cloud Computing V2. 1. *CSA (Cloud Security Alliance), USA. Online: http://www.cloudsecurityalliance.org/guidance/csaguide.v2*, 1, 2009.

[21] Force.com. Application Development Platform. http://www.force.com/.

[22] SendGrid. SMTP Relay Server: App and Transactional Email Delivery. http://sendgrid.com/.

[23] S. Ortiz. The problem with cloud-computing standardization. *Computer*, 44(7):14 –15, july 2011.

[24] VMware and Google. http://www.vmware.com/cloudportability.

[25] Zend, IBM, Microsoft, Rackspace, Nirvanix, and GoGrid. SimpleCloud API. http://www.simplecloud.org/.

[26] mOSAIC. http://www.mosaic-cloud.eu/.

[27] Cloud4SOA. http://www.cloud4soa.eu/.

[28] Atos. http://www.atos.com/.

[29] Open Cloud Computing Interface. http://occi-wg.org/.

[30] Z. Hill and M. Humphrey. CSAL: A Cloud Storage Abstraction Layer to Enable Portable Cloud Applications. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 504 –511, 30 2010-dec. 3 2010.

[31] Dana Petcu. Portability and interoperability between clouds: Challenges and case study. In Witold Abramowicz, IgnacioM. Llorente, Mike Surridge, Andrea Zisman, and Julien Vayssière, editors, *Towards a Service-Based Internet*, volume 6994 of *Lecture Notes in Computer Science*, pages 62–74. Springer Berlin Heidelberg, 2011.

[32] Google. Using JPA with App Engine. https://developers.google.com /appengine/docs/java/datastore/jpa/overview.

[33] Oracle. EntityManager (Java EE 6). http://docs.oracle.com/javaee/6 /api/javax/persistence/EntityManager.html.

[34] Oracle. Future (Java 2 Platform SE 5.0). http://docs.oracle.com/javase/ 1.5.0/docs/api/java/util/concurrent/Future.html.

[35] Google App Engine. Backends (Java). https://developers.google.com /appengine/docs/java/backends.

[36] Microsoft. Federations in Windows Azure SQL Database. http://msdn.microsoft.com/en-us/library/windowsazure/hh597452.aspx.

[37] Steven John Metsker and William C. Wake. *Design Patterns in Java*, chapter 17. Addison-Wesley Professional, 2006.

[38] An Open Source Java SQL Parser. http://zql.sourceforge.net/.

[39] w3school.com. SQL Data Types for MS Access, MySQL and SQL Server. http://www.w3schools.com/sql/sql_datatypes.asp.

[40] Karl Pearson. Similarity Metrics: Pearson Correlation Coefficient. http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio _exports/sphilip/pear.html.

[41] The Apache Software Foundation. http://jmeter.apache.org/.

Appendici

Appendice A

Configurazioni libreria CPIM

Questa appendice contiene le informazioni necessarie per poter configurare correttamente un'applicazione che utilizza la libreria CPIM manualmente, senza l'ausilio del plugin descritto nel Capitolo 5.

Tali configurazioni sono le uniche modifiche richieste allo sviluppatore per trasferire un'applicazione da un provider all'altro.

Vengono descritte le configurazioni necessarie riguardanti i singoli servizi offerti dalla libreria, da inserire nei file *configuration.xml*, *queue.xml* e *persistence .xml*. Tali file devono essere collocati nella cartella src/META-INF del progetto. Infine sono presentate le configurazioni dei file *ServiceConfiguration.cscfg* e *ServiceDefinition.csdef* necessari per il deployment di un'applicazione su Windows Azure.

A.1 Scelta del Cloud Provider

Uno dei tag XML presenti nel file *configuration.xml*, è il tag *<vendor>*. Tale tag permette di settare la piattaforma Cloud scelta per il deployment dell'applicazione. Questa configurazione è obbligatoria.

Listato A.1: Configurazione scelta Cloud Provider

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2   <configurations>
3     <vendor>{Google|Azure}</vendor>
4   ...
5 </configurations>
```

Come intuibile dal Listato A.1, le scelte ammissibili sono:

- *<vendor>Google</vendor>*: se la scelta ricade sulla piattaforma di Google
- *<vendor>Azure</vendor>*: se si desidera utilizzare Windows Azure

Tale valore verrà utilizzato dalla libreria CPIM durante l'istanziazione di ogni servizio, per determinare la tipologia della "Factory" specifica per il provider scelto.

A.2 Configurazione servizio NoSQL

Il servizio NoSQL, utilizzando lo standard JPA, richiede la scrittura del file *persistence.xml*. Come precedentemente descritto, tale file deve essere collocato nella cartella src/META-INF del progetto e serve per definire la persistence-unit e alcune sue proprietà.

Per Google App Engine

Per quanto riguarda App Engine, un esempio di questo file è presentato nel Listato A.2. Per App Engine, che permette l'utilizzo nativo dell'interfaccia JPA come mezzo di comunicazione con il proprio Datastore, sono definibili diversi tag, per customizzare diverse proprietà della persistence-unit definita. Per un'ampia trattazione dei tag ammessi, si rimanda al portale di App Engine.

Listato A.2: Contenuto file persistence.xml per App Engine

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2    <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
3      <persistence-unit name="persistenceUnitName">
4        <provider>org.datanucleus.store.appengine.jpa.DatastorePersistenceProvider</
       provider>
5      <properties>
6        <property name="datanucleus.NontransactionalRead" value="true" />
7        <property name="datanucleus.NontransactionalWrite" value="true" />
8        <property name="datanucleus.ConnectionURL" value="appengine" />
9      </properties>
10   </persistence-unit>
11 </persistence>
```

Per Windows Azure

Nel Listato A.3, invece è proposto un esempio di *persistence.xml* per il servizio TableService di Azure.

Listato A.3: Struttura persistence.xml di Windows Azure

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
3     <persistence-unit name="persistenceUnitName">
4        <properties>
5           <property name="storage.emulator" value="booleanValue" />
6           <property name="account.name" value="account" />
7           <property name="account.key" value="key" />
8        </properties>
9     </persistence-unit>
10 </persistence>
```

Le configurazioni esprimibili tramite questo file per Azure sono:

- Persitence unit: si definisce il nome della persitence unit
- Altre configurazioni, utilizzanti pattern property:
  - property ˝storage_emulator˝: se impostata a ˝true˝ la JPA rimappa le chiamata HTTP nel formato compatibile per le chiamate locali verso l'emulatore
  - property ˝account_name˝: username della sottoscrizione del account storage di Azure. Se si desiderasse utilizzare lo storage emulator locale, settare il valore della property a ˝devstoreaccount1˝.
  - property ˝account_key˝: password relativa alla sottoscrizione dell'account storage di Azure.
    Se si desiderasse utilizzare lo storage emulator locale, settare il valore della property a:
    ˝Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==˝

A.3 Configurazione servizio SQL

Per poter configurare il servizio SQL bisogna specificare nel file *configuration.xml* la stringa di connessione per permettere all'applicazione di connettersi al database desiderato tramite la JDBC (Java Database Connectivity).

```
1  <?xml version="1.0" encoding="UTF..8" ?>
2  <configurations>
3      ...
4      <services>
5          ...
6          <sql>
7              <connection string="value"/>
8          </sql>
9          ...
10     </services>
11 </configurations>
```

Per Google App Engine

Per utilizzare un'istanza di un database del servizio Google Cloud SQL, occorre semplicemente specificare la ˜connection_ string˝, utilizzando lo schema indicato nella riga 7 del Listato A.4.

Se si sta testando la propria applicazione in locale e si sta utilizzano Eclipse e il relativo Plugin di App Engine, oltre a specificare la ˜connection_ string˝ (vedere schema nel Listato A.4, riga 4), occorre configurare alcune proprietà del progetto. Nello specifico, bisogna selezionare con il tasto destro il progetto, espandere il menù Google e selezionare AppEngine Settings. A questo punto apparirà una finestra in cui, tra le altre cose, è possibile abilitare Google Cloud SQL.

L'abilitazione del servizio per il testing locale, mette a disposizione due differenti possibilità:

- MySQL
- Google Cloud SQL

Normalmente si utilizza un'istanza locale di MySQL. Per configurare correttamente il progetto occorre cliccare su ˜Configure˝. A questo punto è richiesto

l'inserimento di una serie di parametri tra cui hostname, che deve essere settato a *localhost*, nome del database, username e password per l'autenticazione.

Sia per il testing locale, che per l'utilizzo del servizio Cloud, occorre configurare l'istanza di App Engine SQL, cliccando sul link "Configure" corrispondente. A seconda che l'esecuzione sia in locale oppure sulla piattaforma, sono necessarie diverse configurazioni:

- Configurazioni per utilizzo locale:
  - Instance name: localhost
  - Altre configurazioni: non occorre specificare nulla, già configurate precedentemente
- Configurazioni per utilizzo servizio Cloud remoto:
  - Instance name: nome dell'istanza creata
  - Database name: nome del database
  - Database username: eventuale username, se creato account d'accesso, altrimenti lasciare vuoto, in automatico verra utilizzato lo username dell'account di default.
  - Database password: eventuale password dell'account d'accesso, se creato, altrimenti lasciare vuoto, in automatico verra utilizzato la password dell'account di default.

Confermando, si è conclusa la configurazione del servizio.

Listato A.4: Configurazione servizio SQL per Google App Engine

```
1 ...
2 <sql>
3 //locale
4 <connection string="jdbc:google:rdbms://localhost/<database>?user=<username>
5 &amp;password=<password>" />
6 //remota
7 <connection string="jdbc:google:rdbms://<instance name>/<database>" />
8 </sql>
9 ...
```

Per Windows Azure

Lo schema richiesto per la *connection_string* necessaria per la configurazione del servizio SQL in Azure è descritto, sia per il caso locale, che per quello di effettivo utilizzo del servizio sulla piattaforma, nel Listato A.5.

La *connection_string* remota di Azure viene creata nel momento in cui viene attivata un'istanza SQL sulla piattaforma. Quindi è consigliabile copiarla direttamente dal portale Azure.

Listato A.5: Configurazione servizio SQL per Azure

```
1 //locale
2 <connection string="jdbc:sqlserver://127.0.0.1:1433;database=<db>;user=<username>;
      password=<psw>" />
3 //remota
4 <connection string="jdbc:sqlserver://<servername>.database.windows.net:1433;database
      =<name>; user=<username>;password=<userpsw>;"/>
```

A.4 Configurazione Blob Service

Per quanto riguarda questo servizio, per la piattaforma di Google non serve nessuna configurazione, mentre per Azure i dati richiesti sono semplicemente le credenziali di accesso all'account di storage, cioè quelle contenute nel già citato file *persistence.xml*. Quindi la configurazione per questo servizio, sono le medesime di quelle descritte per il servizio NoSQL.

A.5 Configurazione Queue Service

Per poter configurare la tipologia e le caratteristiche principale delle code, bisogna creare un file *queue.xml* da inserire nella cartella META-INF e, solo per Google App Engine, nella cartella war/WEB-INF/ del progetto.

Queste file permette di specificare le seguenti caratteristiche per entrambe le piattaforme:

- La tipologia della coda tramite il tag <*mode*> (PUSH = Task e PULL = Message)

- Il rate di lettura dei task presenti nella coda (solo per code di tipologia PUSH).
  Il rate deve essere espresso tramite lo schema:
  (1|2|...|9)(0|1|...|9)∗/(s|m|d).
  Dove s indica secondi, m minuti e d giorni.
  Esempio: 5/s indica rate di lettura pari a 5 al secondo.

Listato A.6: Esempio contenuto queue.xml

```
1  <queue-entries>
2     <queue>
3        <name>queue</name>
4        <mode>pull</mode>
5     </queue>
6     <queue>
7        <name>queuetask</name>
8        <mode>push</mode>
9        <rate>5/s</rate>
10    </queue>
11 </queue-entries>
```

La libreria CPIM permette inoltre di poter eseguire i task presenti in una coda, configurata con il tag *<mode>PUSH</mode>*, su una VM dedicata. Questo è permesso inserendo il tag *backend*, come riportato nel Listato A.7, nel file *configuration.xml*.

Listato A.7: Configurazione Backend

```
1  <configurations>
2     ...
3     <services>
4        ...
5        <backend name="value"/>
6     </services>
7  </configurations>
```

A seconda della piattaforma su cui viene eseguita l'applicazione, i valori da specificare nell'attributo *name* cambiano:

- Per Google App Engine: bisogna specificare l'indirizzo `version.your_app_id.appspot.com` specificando la versione in cui risiederà la parte di backend dell'applicazione.

- Per Windows Azure: bisogna specificare il nome del WAR file, senza l'estensione, contenente la parte di backend dell'applicazione.

A.6 Configurazione Memcache Service

Per poter configurare il servizio Memcache è necessario specificare nel file *configuration.xml* il tag <*memcache*>, in modo da permettere l'abilitazione di tale servizio.

Listato A.8: Configurazione Memcache

```
1 <configurations>
2    ...
3    <services>
4       <memcache>
5       ...
6       </memcache>
7       ...
8    </services>
9 </configurations>
```

Per Google App Engine

Per la piattaforma Google App Engine è necessario specificare solamente il tag senza alcuna ulteriore informazione.

Per Windows Azure

Per la piattaforma di Microsoft è necessario specificare l'indirizzo e la porta dell'host che fornisce la propria RAM per la fruizione del servizio Memcache, come riportato nel Listato A.9.

Oltre a questa configurazione, per poter abilitare il servizio su Cloud, sono necessarie ulteriori impostazioni da inserire nei file *ServiceConfiguration.cscfg* e *ServiceDefinition.csdef* discussi successivamente, nella sezione A.8 dedicata a questi file.

Listato A.9: Configurazione Memcache per Azure

```
1 //Configurazione Host remoto
2 <memcache>
```

```
3     <host address="localhost_WorkerRoleName" port="11211"/>
4  </memcache>
5
6  //Configurazione Host locale
7  <memcache>
8     <host address="127.0.0.1" port="11211"/>
9  </memcache>
```

Per quanto riguarda il funzionamento in locale del servizio, è richiesta l'installazione di un server Memcache, il quale resterà in ascolto sulla porta specificata (11211 di default).

Si noti dal Listato A.9 che la porta da utilizzare è sempre la 11211, porta generalmente utilizzata per il servizio Memcache.

Per quanto riguarda la configurazione dell'indirizzo dell'host, per la soluzione locale è sufficiente indicare *localhost* (oppure *127.0.0.1*), per quella remota occorre utilizzare lo schema *localhost_<WorkerRoleName>*.

A.7 Configurazione servizio Mail

Per poter configurare il servizio Mail è necessario specificare nel file *configuration.xml* il tag *<mail>*, in modo da permettere l'abilitazione del servizio.

```
1  <configurations>
2     ...
3     <services>
4        <mail>...</mail>
5        ...
6     </services>
7  </configurations>
```

Per Google App Engine

Per quanto riguarda un'applicazione destinata al Cloud di Google, non occorre aggiungere particolari configurazioni, poichè essendo il servizio offerto nativamente dalla piattaforma, si utilizzano automaticamente server SMTP interni alla rete di Google.

È però richiesto aggiungere nel file *appengine−web.xml*, presente nella cartella WEB-INF di ogni progetto per Google App Engine creato con il plugin di Eclipse, i tag mostrati nel Listato A.10.

Listato A.10: Configurazione servizio Mail per Google App Engine

```
1 <inbound−services>
2    <service>mail</service>
3 </inbound−services>
```

Per Windows Azure

Come descritto nella sezione di questo capitolo dedicata alla piattaforma di Microsoft, Azure non offre nativamente questo servizio. Per questo motivo occorre utilizzo un server SMTP esterno.

Per la sua configurazione, è indispensabile quindi aggiungere i tag indicanti l'indirizzo e la porta del server scelto, nonchè le credenziali per l'accesso, mostrati nel Listato A.11.

Listato A.11: Configurazione servizio Mail per Azure

```
1 <mail>
2    <server_smtp host="smtp.provider.com" port="587" />
3    <account_info username="mail_address" password="psw" />
4 </mail>
```

A.8 Altre configurazioni

La struttura di un progetto per il Deployment di Windows Azure, contiene due file di configurazione usati per configurare la piattaforma, rispetto ai servizi utilizzati. Tali file sono:

- *ServiceDefinition.csdef*: definisce le configurazioni runtime per l'applicazione, incluse quante Role servono, i vari Endpoint e la dimensione della virtual machine.
- *ServiceConfiguration.cscfg*: configura quante istanze di una Role sono eseguite a i valori delle impostazioni per ogni Role.

In questa sezione verrano presentate solo le configurazioni necessarie per un corretto utilizzo dei servizi supportati dalla libreria.

A.8.1 Service Configuration file

Nel Listato A.12 viene riportato un esempio di *ServiceConfiguration* file. Le configurazioni presenti sono:

- <*Role*> : da replicare per ogni Role utilizzata, specificando per ognuna il nome attraverso l'attributo "name". Contiene i seguenti tag:
  - <*Instances*> : permette tramite l'attributo "count"di specificare il numero di istanze da assegnare alla Role.
  - <*ConfigurationSettings*>: permette di inserire le configurazioni di alcuni servizi utilizzati dall'applicazione. In particolare nel esempio riportato nel Listato A.12, sono presenti le configurazioni relative al modulo Caching necessarie per l'abilitazione di tale servizio. La semantica di questi tag è descritta nella Sezione 3.2, nel paragrafo relativo al servizio di Caching di Azure.

Listato A.12: Esempio di un ServiceConfiguration file

```
1  <?xml version="1.0" encoding="utf..8" standalone="no"?>
2  <ServiceConfiguration xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/
       ServiceConfiguration" osFamily="2" osVersion="*" serviceName="
       WindowsAzureDeploymentProject">
3   <Role name="WorkerRole1">
4    <Instances count="1"/>
5    <ConfigurationSettings>
6     <Setting name="Microsoft.WindowsAzure.Plugins.Caching.NamedCaches" value=""/>
7     <Setting name="Microsoft.WindowsAzure.Plugins.Caching.Loglevel" value=""/>
8     <Setting name="Microsoft.WindowsAzure.Plugins.Caching.CacheSizePercentage" value
        ="30"/>
9     <Setting name="Microsoft.WindowsAzure.Plugins.Caching.
        ConfigStoreConnectionString" value="DefaultEndpointsProtocol=https;
        AccountName=<account>;AccountKey=<key>"/>
10   </ConfigurationSettings>
11  </Role>
12 </ServiceConfiguration>
```

A.8.2 Service Definition file

Nel Listato A.13 viene riportato un esempio di ServiceDefinition file. Le configurazioni che seguono devono essere replicate per ogni Role utilizzata dal progetto da caricare in Azure:

- <*WorkerRole*>: permette di specificare il nome della WorkerRole e la dimensione della relativa istanza. Contiene i seguenti tag:
  - <*Startup*>: identifica il comando da eseguire per effettuare il corretto avvio della WorkerRole, in particolare per l'ambiente Java, predispone la piattaforma con i file e le cartelle necessarie per l'avvio della WorkerRole. Solitamente questa impostazione è configurata di default al momento della creazione del progetto.
  - <*Runtime*>: permette di specificare il comando, e il relativo livello di autorizzazione, da eseguire al momento dell'avvio della WorkerRole tramite il tag:
    - <*EntryPoint*>: specifica il comando da eseguire.
  - <*Imports*>: permette di importare i moduli definiti nel file *ServiceConfiguration*. In particolare, il modulo necessario da importare se si desiderasse utilizzare il servizio di Memcache della libreria, è il modulo denominato "Caching".
  - <*LocalResources*>: permette di specificare le risorse utilizzate, come se fossero istanziate localmente all'interno della WorkerRole. Nel esempio riportato, viene configurato uno storage utilizzato dalla Memcache per il salvataggio delle informazioni di logging, relative a tale servizio, all'interno della WorkerRole.
  - <*Endpoints*>: permette di definire i punti d'accesso della WorkerRole. In particolare vengono riportate due tipologie d'accesso:
    - <*InputEndpoint*>: specifica il protocollo e la porta con cui accedere alla WorkerRole dall'ambiente esterno.
    - <*InternalEndpoint*>: specifica il protocollo, la porta e il nome del servizio con cui interagisce la WorkerRole localmente. Nel caso si utilizzasse il servizio memcache, occorre definire un

InternalEndpoint apposito, come mostrato in riga 20 del Listato A.13.

Listato A.13: Esempio di un ServiceDefinition file

```
1  <?xml version="1.0" encoding="utf..8" standalone="no"?>
2  <ServiceDefinition xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/
       ServiceDefinition" name="WindowsAzureDeploymentProject">
3  <WorkerRole name="WorkerRole1" vmsize="Small">
4      <Startup>
5          <Task commandLine="util/.start.cmd .startup.cmd" executionContext="elevated"
       taskType="simple"/>
6      </Startup>
7      <Runtime executionContext="elevated">
8          <EntryPoint>
9              <ProgramEntryPoint commandLine="run.cmd" setReadyOnProcessStart="true
       "/>
10         </EntryPoint>
11     </Runtime>
12     <Imports>
13         <Import moduleName="Caching"/>
14     </Imports>
15     <LocalResources>
16         <LocalStorage cleanOnRoleRecycle="false" name="Microsoft.WindowsAzure.
       Plugins.Caching.FileStore" sizeInMB="1000"/>
17         </LocalResources>
18     <Endpoints>
19         <InputEndpoint localPort="8080" name="http" port="80" protocol="tcp"/>
20         <InternalEndpoint name="memcache default" port="11211" protocol="tcp"/>
21     </Endpoints>
22 </WorkerRole>
23 </ServiceDefinition>
```

Appendice B

Predisposizione dell'ambiente di sviluppo

In questa appendice vengono riportate tutte le operazioni necessarie ad effettuare il deployment di un'applicazione sviluppata attraverso la libreria CPIM, senza l'ausilio del plugin descritto nel Capitolo 5.

L'ambiente di sviluppo utilizzato per lo sviluppo della libreria e dei relativi test funzionali è Eclipse, più precisamente Eclipse versione 3.6. A seconda della scelta del provider, cambiano i plugin utilizzati e le relative dipendenze.

B.1 Ambiente di sviluppo per Google App Engine

Di seguito sono riportate le informazioni relative a come predisporre correttamente l'ambiente di sviluppo in Eclipse per poter sviluppare applicazioni Java per la piattaforma di Google.

B.1.1 Librerie e plugin necessari

Per poter effettuare il deploy di una applicazione sulla piattaforma di Google tramite Eclipse, è necessario installare il relativo plugin. Quest'ultimo, può essere scaricato direttamente tramite Eclipse (dal menu Help→Install New Software) e digitando l'indirizzo `http://dl.google.com/eclipse/plugin/3.6`.

La versione dell'SDK di Google utilizzata per sviluppare la libreria è la 1.6.4.

B.1.2 Creazione di un progetto web

Dopo aver installato il plugin, si ha la possibilità di creare un nuovo progetto web per la piattaforma App Engine (tramite il percorso: File→New→Other→Google→Web Application Project).

Il progetto creato avrà la struttura classica di un progetto web in Java: la cartella src, dove saranno contenute le servlet e le varie classi Java e la cartella contenente il war, a sua volta contenente, tra le altre, le pagine jsp/html costituenti il tipico strato di presentazione di una applicazione web in Java.

B.1.3 Dipendenze della libreria CPIM

Per poter utilizzare la libreria CPIM, bisogna innanzitutto copiarne il relativo file JAR nella cartella war/WEB-INF/lib e, se non automaticamente configurato da Eclipse, occorre configurarne la referenza nel BuildPath del progetto.

Se si desidera utilizzare la piattaforma di Google, la libreria CPIM ha, a runtime, le seguenti dipendenze:

- appengine-api-1.0-sdk-1.6.4.jar
- appengine-api-labs-1.6.4.jar
- appengine.jsr107cache-1.6.4.jar
- datanucleus-appengine-1.0.10.final.jar
- datanucleus-core-1.1.5.jar
- datanucleus-jpa-1.1.5.jar
- geronimo-jpa 3.0 spec-1.1.1.jar
- geronimo-jta 1.1 spec-1.1.1.jar
- jdo2-api-2.3-eb.jar
- jsr107cache-1.1.jar

- mysql-connector-java-5.1.21-bin.jar

Tutti questi JAR devono essere copiati all'interno della cartella war/WEB-INF/lib e referenziati nel BuildPath del progetto. Normalmente tali librerie sono già presenti e referenziate, in quanto costituiscono l'SDK di Google App Engine, fatta eccezione per l'ultimo JAR rappresentante il driver di connessione per il servizio SQL.

B.1.4 Deployment di un'applicazione

Una volta terminata l'implementazione dell'applicazione, e dopo averla testata in locale, il processo di pubblicazione di un'applicazione su App Engine è molto semplice utilizzando il plugin per Eclipse.

Innanzitutto occorre connettersi ad un profilo di Google (operazione possibile utilizzando il tasto presente in basso a sinistra dell'interfaccia grafica di Eclipse). Una volta autenticati, si deve cliccare con il tasto destro sul progetto, aprire il menù Google e cliccare su "Deploy to App Engine".

Dopo aver configurato le impostazioni necessarie (tramite il link AppEngine project settings), premendo il tasto "Deploy" il processo di caricamento parte in automatico.

Nel caso in cui si volesse deployare l'applicazione su due tier differenti, in modo da permettere l'esecuzione dei task presenti nelle Task Queue separatamente rispetto al frontend, occorre effettuare il processo sopra descritto anche per il livello di backend. L'unica differenza risiede nella versione con cui deployare l'applicazione (modificabile tramite il link AppEngine project settings), che deve essere necessariamente differente rispetto alla versione del frontend e il cui valore deve essere concorde con la stringa di configurazione riportata in A.5.

B.2 Ambiente di sviluppo per Windows Azure

Seguono informazioni relative a come predisporre correttamente l'ambiente di sviluppo in Eclipse per poter sviluppare applicazioni Java per la piattaforma di Windows Azure.

B.2.1 Librerie e plugin necessari

Per quanto riguarda lo sviluppo di un'applicazione per la piattaforma Windows Azure, gli elementi necessari sono differenti.

Innanzitutto bisogna installare l'SDK di Windows Azure. Tale pacchetto installa sul computer l'emulatore utilizzato per simulare l'ambiente di Azure in locale.

Per configurare l'ambiente di Eclipse in modo da poter sviluppare applicazioni per Windows Azure, è necessario installare il plugin, scaricabile e installabile direttamente da Eclipse, inserendo il repository dell'indirizzo `http://dl.msopentech.com/eclipse`. Le funzionalità offerte da questo plugin sono descritte in `http://msdn.microsoft.com/en-us/library/windowsazure/hh694271.aspx`

B.2.2 Creazione di un progetto web

Per creare un progetto web per Windows Azure basta semplicemente creare un progetto web standard di Java (File→New→Project→Web →Dynamic Web Project).

B.2.3 Dipendenze della libreria CPIM

Per poter utilizzare la libreria CPIM, bisogna innazitutto copiarne il relativo file JAR nella cartella war/WEB-INF/lib e, se non automaticamente configurato da Eclipse, occorre configurarne la referenza nel BuildPath del progetto.

Se si desidera utilizzare la piattaforma di Windows Azure, la libreria CPIM ha, a runtime, le seguenti dipendenze:

- commons-lang-2.6.jar
- hibernate-jpa-2.0-api-1.0.1.Finale.jar
- javassist-3.12.1.GA.jar
- jpa4azure-1.0.jar (versione estesa)
- microsoft-windowsazure-api-0.2.2.jar
- spymemcached-2.8.4.jar

- sqljdbc4.jar
- zql.jar

Tutti questi JAR devono essere copiati all'interno della cartella war/WEB-INF/lib e referenziati nel BuildPath del progetto.

B.2.4 Deployment di un'applicazione

Una volta terminata l'implementazione dell'applicazione, per poterla testare in locale o per effettuare il deploy sulla piattaforma Azure, è necessario creare un progetto *Windows Azure Deployment Project* (da File→New Project→Windows Azure Deployment Project). A questo punto si apre una finestra che richiede la configurazione del progetto.

Per poter effettuare il deployment di un'applicazione Java in Azure, occorre caricare anche una JDK, un Application Java Server e uno script batch per avviarlo, nonché il WAR contenente il codice applicativo. Questi componenti sono indispensabili, poichè Azure non mette nativamente a disposizione l'ambiente di runtime per poter eseguire sulla piattaforma codice Java. Per questo motivo, utilizzando impropriamente come IaaS una piattaforma Paas, occorre installare nell'istanza di una Worker Role di Azure, la JDK e successivamente il Server Java, nel quale sarà poi effettivamente caricata l'applicazione web. A tale proposito, durante la configurazione del progetto, è necessario selezionarne l'inclusione della JDK e del server, specificando il percorso in cui poter trovare le relative cartelle.

Una volta configurati questi due elementi, bisogna caricare il WAR relativo all'applicazione e, nel caso in cui l'applicazione facesse uso del servizio Task Queue, il WAR relativo al consumer, distribuito insieme alla libreria e denominato Worker.war. Tale file deve essere preventivamente adattato all'applicazione.

In pratica, occorre estrarre il contenuto del file WAR, copiare i file di configurazione (*persistence.xml*, *queue.xml* e *configuration.xml*) dell'applicazione, e posizionarli nella cartella WEB-INF/classes/META-INF del Worker. Fatto questo, bisogna ricreare il file Worker.war, con il nuovo contenuto. La modifica di altri file potrebbe causare il blocco del servizio di Task Queue.

Nel caso in cui si volesse deployare l'applicazione su due tier differenti, in modo da permettere l'esecuzione dei task presenti nella Task Queue separatamente rispetto al frontend, occorre creare una seconda Worker Role, comprensiva di JDK e server, in cui collocare il WAR contenente il codice applicativo relativo al backend dell'applicazione e il WAR relativo al consumer delle code preventivamente adattato con le configurazioni precedentemente descritte. In questo caso la Worker Role contenente il frontend non deve contenere il WAR del consumer. Inoltre, il nome del WAR file contenente il backend applicativo, deve essere concorde alla configurazione riportata in A.5.

A questo punto, occorre modificare i file *ServiceDefinition* e *Service Configuration*, come descritto in A.8.

Attraverso il plugin di Azure è possibile, con pochi click, testare l'applicazione in locale, utilizzando appositi emulatori della piattaforma Cloud, oppure richiedere il caricamento in Azure.

I tempi di caricamento di un'applicazione Java sulla piattaforma Azure sono considerevoli, in quanto oltre all'applicazione, vengono trasferiti il server e la JDK, quindi si consiglia un testing esaustivo in locale, tramite l'emulatore.

B.3 Trasferimento di un'applicazione da un provider all'altro

Per trasferire un'applicazione che utilizza la libreria CPIM da un cloud all'altro, dopo aver predisposto l'ambiente di sviluppo per il Cloud di destinazione e creato un nuovo progetto secondo le modalità precedentemente discusse, basta copiare in esso le classi Java e le pagine JSP/HTML relative all'applicazione.

L'unica operazione di modifica richiesta è l'adattamento dei file di configurazioni, descritti nell'Appendice A, alla piattaforma di destinazione.