Alma Mater Studiorum · Università di Bologna

**facoltà di scienze matematiche, fisiche e naturali**
**Corso di Laurea Magistrale in Informatica**

# Un'architettura dinamica service-oriented per l'esecuzione distribuita di task

**Tesi di Laurea in Sistemi Middleware**

**Relatore:**
**Chiar.mo Prof.**
**Fabio Panzieri**

**Correlatore:**
**Dott.**
**Nicola Mezzetti**

**Presentata da:**
**Andrea Piemontese**

**Sessione II**
**Anno Accademico 2010/2011**

*It always seems impossible until it's done.*
*Nelson Mandela*

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Elenco dei codici

# Introduzione

La diffusione delle architetture orientate ai servizi affiancata alla recente introduzione del cloud computing, ha reso possibile la realizzazione di servizi che si adattano dinamicamente e proattivamente al carico di lavoro. Dal punto di vista dell'utente, un servizio o applicazione in cloud fornisce potenza di calcolo e memoria illimitate; tale illusione è resa possibile grazie a un'infrastruttura composta da un elevato numero di elaboratori organizzati in data center nei quali si utilizza la virtualizzazione e sofisticate tecniche per una distribuzione ottimale delle risorse.

Una tipologia di cloud realizzato interamente all'interno di un contesto aziendale ad uso privato è detto *private cloud*; esso è utilizzato per trattare dati particolarmente sensibili o quando è più conveniente gestire l'elaborazione dei dati internamente.

Scopo di questo lavoro è la progettazione di un'architettura service oriented fault-tolerant per lo scheduling e l'esecuzione di un workflow di job in un ambiente distribuito dinamico. Il sistema realizzato si propone come architettura di supporto alla realizzazione di un servizio di scheduling schierabile in private cloud. Il sistema permette l'esecuzione affidabile di un workflow di job distribuendo il carico di lavoro su un numero arbitrario di nodi che possono essere dinamicamente spostati, aggiunti o rimossi grazie al concetto di servizio e al discovery automatico dei nodi.

Si è scelto di realizzare la gestione di un workflow di job (o task per brevità) come funzionalità del sistema poiché è una necessità sentita e sempre attuale nelle realtà aziendali: c'è spesso bisogno di eseguire reiteratamente

una serie di operazioni interdipendenti, ad esempio, legate alla creazione di reportistica o alla manutenzione delle basi di dati.

Al momento della scrittura di questo elaborato, la maggioranza delle soluzioni disponibili per la gestione di task in cloud non sono sviluppate per essere fruite tramite private cloud o comunque non disponibili open source. Con la realizzazione di questo lavoro ci si propone di sviluppare e validare, seppur allo stadio prototipale, un sistema open source che possa essere utilizzato sia in private cloud che su un cluster fisico di macchine.

Il lavoro di tesi si articola in due fasi: la prima comprende la definizione, il progetto e l'implementazione del sistema ed è trattata nei capitoli 2 e 3; la seconda fase, discussa nel capitolo 4, è incentrata sul test e sulla validazione del prototipo con particolare riferimento all'affidabilità della soluzione in termini di *safety* e *liveness*.

Nella fase di progetto si discutono le problematiche affrontate e le relative soluzioni; sono state applicate alcune tecniche tipiche dei sistemi distribuiti quali coordinazione, eventi distribuiti, object leasing e gestione di partial failure. Inoltre sono state esaminate le tematiche relative alle funzionalità fornite dal sistema, ossia lo scheduling distribuito e la gestione dell'esecuzione di un workflow di job su una rete di nodi esecutori.

Il risultato dalla prima fase, è un'architettura prototipale basata sul pattern *master-workers* realizzata mediante l'integrazione di due componenti principali: un framework per la costruzione di sistemi middleware, *Apache River* con cui è stata definita l'architettura, ed un noto scheduler di job, chiamato *Quartz*, che è servito alla realizzazione delle funzionalità.

Si è ottenuto un sistema finale che presenta le seguenti caratteristiche principali:

- *affidabilità e tolleranza ai guasti;*
- *adattività al cambiamento;*
- *scalabilità e adattamento al carico;*
- *assenza di configurazione.*

Nella fase di test e validazione si verifica empiricamente la correttezza in uno scenario reale come un cluster di elaboratori sulla stessa rete locale. Per le varie parti del sistema sono state simulati partial failure e possibili condizioni di errore legate all'esecuzione dei task; si è poi valutata l'efficacia delle soluzioni implementate e schematizzato il comportamento del sistema ai malfunzionamenti. Le soluzioni applicate derivano dall'analisi di generici *pattern* per la gestione delle eccezioni, già presenti in letteratura, dai quali sono state elaborate delle *azioni di recovery* specifiche per il sistema sviluppato.

# Capitolo 1

# Stato dell'arte

In questo capitolo si passa in rassegna lo stato dell'arte sui paradigmi e le tecnologie utilizzate nella trattazione della tesi, in particolare se ne discutono lo stato attuale, i limiti ed i punti di forza.

## 1.1 Service Oriented Architectures (SOA)

### 1.1.1 Introduzione

Il *Service Oriented Computing (SOC)* è un paradigma di programmazione che si propone di facilitare la creazione di applicazioni fortemente distribuite interoperabili utilizzando il servizio come elemento principale per lo sviluppo delle soluzioni software. Il paradigma, promuove l'idea di assemblare componenti software attraverso una rete di servizi al fine di creare applicazioni e processi di business dinamici che si estendono tra diverse organizzazioni e piattaforme [11]. Il SOC si basa su un tipo di un'architettura orientata ai servizi nota come *Service Oriented Architecture (SOA)*, la quale riorganizza l'infrastruttura e il software in modo che sia fruibile come un insieme di servizi. I servizi permettono di integrare tra loro applicazioni distribuite non concepite per essere facilmente integrabili, essi sono sia gli strumenti per costruire nuove funzionalità che un mezzo per integrare funzionalità di applicazioni preesistenti. La definizione di "architettura" per la SOA è in qualche

modo limitativa, infatti, oltre all'aspetto tecnologico si devono includere le politiche, le tecniche e i framework necessari per fornire gestire e consumare i servizi.

Dagli anni '80 ad oggi, il livello di astrazione con cui sono definite, rese disponibili e fruite le funzionalità è diventato via via più alto. Si è passato dai moduli, agli oggetti, alle componenti fino ad arrivare agli attuali servizi come evoluzione naturale dei modelli precedenti.

### 1.1.2 I servizi

Il concetto di *servizio* è alla base delle architetture SOA, in generale, un servizio è un'entità autonoma che esegue una funzione. le funzioni possono spaziare dalla più semplice richiesta di calcolo all'esecuzione di un complicato processo aziendale, in generale le funzioni sono un raggruppamento logico di operazioni. I servizi permettono di esportare sulla rete le funzionalità in maniera sistematica attraverso l'uso di linguaggi e protocolli standardizzati come XML [1]. Teoricamente, ogni applicazione o componente software potrebbe essere trasformata in servizio e resa disponibile in rete come tale.

I servizi sono offerti dai fornitori di servizi o *service provider*, organizzazioni che, tramite la rete, ne mettono a disposizione le implementazioni e le descrizioni. I service provider formano una infrastruttura distribuita che permette l'integrazione e l'interoperabilità dei servizi, i fruitori dei servizi possono essere sia clienti che altri software o servizi (genericamente detti client). Le caratteristiche intrinseche dei servizi come appena definiti delineano i requisiti che un servizio deve soddisfare:

- *Neutralità tecnologica:* i servizi devono poter essere invocati attraverso una tecnologia standardizzata, semplice e comunemente accettata nella comunità IT. Ciò significa che i meccanismi di invocazione del servizio (protocolli, descrizione e ricerca) devono obbedire ad uno standard comune e tecnologicamente neutrale.

[1] XML: eXtensible Markup Language

- *Accoppiamento lasco:* un sistema composto da servizi deve poter essere facilmente diviso in parti e la gestione delle parti deve essere semplice. Inoltre, la fruizione di un servizio non deve richiedere nessuna conoscenza del funzionamento interno del servizio.

- *Location transparency:* ossia, la posizione delle risorse deve essere gestita in maniera trasparente. L'invocazione dei servizi deve prescindere dalla posizione fisica del servizio o del client. I servizi e le informazioni sulla posizione devono essere immagazzinate in appositi "registry", strutture di indicizzazione che permettono la ricerca dei servizi, come ad esempio quello fornito da UDDI [2].

Si può fare una distinzione tra due tipologie di servizi in base alla granularità delle funzionalità fornite: *servizi semplici* e *servizi composti*. Un servizio semplice svolge una funzione di base che non necessita di altri servizi per essere portata a termine, al contrario, i servizi composti accedono ad altri servizi e ne combinano le funzionalità per fornire un servizio più complesso. Si consideri ad esempio, una collezione di servizi semplici che realizzano alcune generiche compiti aziendali come il tracciamento degli ordini, la fatturazione e la gestione del magazzino; una azienda potrebbe utilizzare tali servizi per creare un'applicazione distribuita che fornisce servizi composti specializzati per un determinato settore (es. medico, trasporti o alimentare). Quindi, un servizio composto è contemporaneamente fruitore e fornitore di servizi.

### Struttura del servizio

Il servizio è logicamente composto da due parti fondamentali: l'*interfaccia* e l'*implementazione*. L'interfaccia è il meccanismo che permette al servizio di comunicare con le altre applicazioni o servizi, è la descrizione formale delle operazioni che il client può invocare sul servizio. Più in generale, si può includere l'interfaccia in quella che si definisce *descrizione del servizo*. L'implementazione è invece la realizzazione fisica del servizio, da un punto di vista

[2]UDDI: Universal Description Discovery and Integration

puramente logico, l'implementazione è di secondaria importanza poiché vediamo il servizio come una scatola nera utilizzabile tramite la sua interfaccia. Di seguito si illustrano la descrizione e l'implementazione del servizio.

*Descrizione del servizio:* è l'insieme di informazioni necessarie per descrivere il servizio in termini di interfaccia, funzionalità e qualità. La pubblicazione di tali informazioni su un service registry [3] fornisce i mezzi necessari per la ricerca (discovery), la connessione e l'uso (binding) dei servizi. L'interfaccia definisce la firma del servizio, mentre la descrizione delle funzionalità specifica lo scopo concettuale ed i risultati attesi quando si invoca tale servizio. Infine, può essere presente anche una descrizione delle qualità del servizio (QoS) che informa i client delle caratteristiche non funzionali quali costo, prestazioni e sicurezza. *Implementazione del servizio:* la realizzazione del servizio può essere molto complessa. In rari casi la realizzazione è costituita da una singolo programma, più spesso le funzionalità di un servizio vengono fornite da diversi servizi e applicazioni interagenti. Come già detto precedentemente in questa sezione, non è necessario approfondire i dettagli implementativi (se non in fase di sviluppo) poiché l'astrazione fornita dalla SOA ci libera da questo compito in fase di utilizzo dei servizi.

### 1.1.3 L'architettura orientata ai servizi

Il modello orientato ai servizi è messo in pratica utilizzando una SOA. Si può vedere la SOA come una metodologia per progettare sistemi software per fornire servizi a utenti finali o altri servizi distribuiti sulla rete, attraverso interfacce pubblicate e ricercabili. Più in particolare, SOA definisce l'infrastruttura che supporta la pubblicazione e la fruizione dei servizi in modo che siano utilizzabili tramite semplice scambio di messaggi con protocolli e interfacce standard. Questo approccio è particolarmente indicato quando numerose applicazioni devono comunicare tra loro, infatti, quando tutte le parti delle applicazioni sono schierate, le applicazioni esistenti e future pos-

[3] Il service registry è anche detto *Service Directory* o *Service Broker*

sono comunicare attraverso i servizi senza necessità di soluzioni ad hoc o imperscrutabili protocolli proprietari.

### Il modello SOA base

Il modello base di un'architettura SOA definisce le interazioni tra agenti software che usano lo scambio di messaggi come mezzo per l'utilizzo delle funzionalità fornite dai service provider come mostrato in figura 1.1. I ruoli delle parti nel modello sono intercambiabili, è molto comune che un agente software sia client e provider allo stesso tempo. I service provider hanno la responsabilità di pubblicare le descrizioni dei servizi che forniscono, la pubblicazione avviene su un *service regitry* che è un'entità nota a tutte le parti in gioco. I client devono poter trovare (find) le descrizioni dei servizi per poter eventualmente utilizzarli tramite l'operazione di associazione (bind).

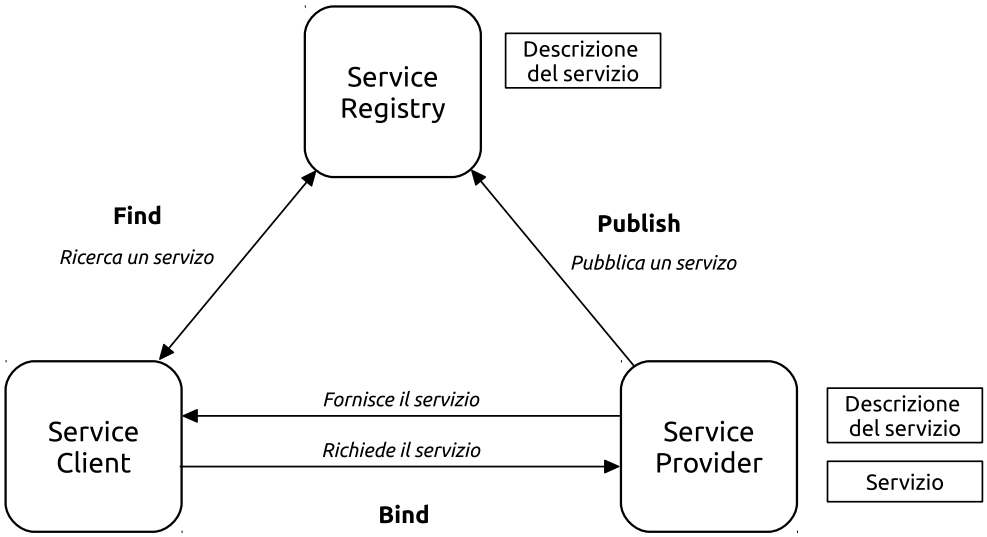


Figura 1.1: Modello base di un'architettura SOA

Nel modello base quindi, sono in relazione tre tipologie di partecipanti: il service provider, il service registry ed i client; l'artefatto su cui i partecipanti agiscono è il servizio. In un semplice scenario esemplificativo, il service provider ospita un modulo software utilizzabile tramite la rete (l'implementazione di un servizio). Il provider definisce la descrizione del servizio compresa di

interfaccia e la pubblica su un service registry, in questo modo il servizio diventa ricercabile. I client interessati al servizio possono così recuperare la descrizione con un'operazione di find, associarsi con un'operazione di bind e interagire con il servizio.

### I web service

Una tecnologia diffusa come standard aperto che ben supporta lo sviluppo di architetture orientate ai servizi è il *web service* (servizio web), il W3C [18] e l'OASIS [10] sono i due maggiori consorzi che curano l'avanzamento degli standard relativi ai web service. I web service permettono di implementare un'infrastruttura di servizi che garantisca le caratteristiche di neutralità tecnologica, interoperabilità e accoppiamento lasco richieste dalle SOA. Di seguito si elencano le caratteristiche fondamentali e gli standard correlati ai web service:

- È un servizio identificato da una URI (sopporto alla location transparency).
- Espone le sue funzionalità in internet usando protocolli e linguaggi standard (XML).
- L'interfaccia del servizio è descritta in un linguaggio automaticamente elaborabile, uno standard utilizzato è WSDL [4].
- L'interfaccia è pubblicata su un registry server, lo standard usato è UDDI che permette di localizzare, elencare ed ottenere la descrizione del servizio.
- L'interazione tra web service avviene attraverso lo scambio di messaggi utilizzando il protocollo standard SOAP [5].

[4] WSDL: Web Services Description Language
[5] SOAP: Simple Object Access Protocol

È importante sottolineare che i web service sono solo una parte nel quadro più generale quali sono le SOA, infatti, l'uso dei web service non è obbligatorio per la realizzazione di una SOA sebbene essi siano sempre più diffusi nella comunità IT.

## 1.2 Cloud computing

### 1.2.1 Introduzione

Il rapido sviluppo delle tecnologie di elaborazione e di memorizzazione affiancato alla sempre maggiore diffusione di internet hanno reso le risorse computazionali più economiche ed universalmente disponibili. Questo trend ha permesso la realizzazione di un nuovo modello computazionale chiamato cloud computing. Il cloud computing è recentemente emerso come paradigma per la realizzazione e la fornitura di servizi attraverso la rete, esso vede le risorse di calcolo (e.g., CPU e memoria) come generici strumenti o *utility* che possono essere presi in prestito e rilasciati tramite la rete in base alla necessità del momento. Questo modello permette di realizzare servizi internet con un alto livello di scalabilità riducendone i costi ed i rischi in tutte le fasi del loro ciclo di vita. Gli sviluppatori di servizi possono non preoccuparsi di dimensionare preventivamente le risorse hardware, possono invece incrementare o ridurre le risorse in base alla domanda reale degli utenti del servizio.

Il termine cloud computing indica sia le applicazioni fruibili come servizi, che l'infrastruttura necessaria per la loro fornitura. In un ambiente cloud computing, il ruolo del fornitore dei servizi è diviso in due: il *fornitore di infrastruttura* (o *fornitore della cloud*), che rende disponibili le risorse hardware e software con un modello di pagamento basato sull'utilizzo, ed il *fornitore di servizi*, che prende in affitto le risorse da uno o più fornitori di infrastruttura per fornire i servizi agli utenti finali.

L'introduzione del cloud computing ha avuto negli ultimi anni un forte impatto sull'industria dell'Information Technology (IT) e molte aziende di

grandi dimensioni tra cui Google, Amazon e Microsoft sono tuttora impegnate nella progettazione di piattaforme cloud sempre più efficienti. Sebbene questo paradigma presenti numerose opportunità per l'industria IT, esso introduce nuove sfide peculiari che devono essere affrontate debitamente.

### 1.2.2 Visione generale

L'idea alla base del cloud computing non è nuova, già negli anni '60 si immaginava la possibilità di fornire potenza di calcolo come utilità [13]. I vantaggi dell'utilizzo del cloud computing sono molteplici sia per gli utenti dei servizi che per i fornitori. I fornitori di applicazioni possono godere di un'infrastruttura scalabile che permette di semplificare il processi di schieramento, controllo di versione e manutenzione dei loro prodotti senza la necessità possedere alcun hardware privato; gli utenti finali possono memorizzare in maniera sicura i loro dati nella cloud ed accederci ovunque ed in ogni momento favorendo la condivisione degli stessi. Di seguito sono elencate le caratteristiche più attraenti del paradigma.

*Nessun investimento iniziale:* il fornitore di servizi non necessita nessun investimento iniziale poiché prende in affitto le risorse in base alle proprie necessità con un modello di pagamento basato sull'effettivo tempo di utilizzo.

*Alta scalabilità:* i fornitori di infrastruttura rendono accessibili a breve termine (e.g. processori per ore, memoria per ore) grandi quantità di potenza calcolo attraverso i propri data center, in questo modo i fornitori di servizi hanno l'illusione di una disponibilità infinita di risorse, ed il servizio può rispondere rapidamente ad un aumento o ad una diminuzione della domanda.

*Facilità di fruizione:* i servizi ospitati nella cloud, essendo on-line, sono accessibili più facilmente e da un numero maggiore di dispositivi rispetto ad una applicazione tradizionale.

*Riduzione dei rischi e dei costi:* l'esternalizzazione per mezzo della cloud sposta i rischi d'impresa (come i guasti hardware) sul fornitore dell'infrastruttura che ha le competenze adatta a gestirli, inoltre si riducono le spese per la manutenzione dell'hardware e per la formazione del personale tecnico.

### 1.2.3 Architettura

In generale, il modello architetturale in un ambiente cloud computing può essere diviso in 4 livelli: il livello hardware, il livello infrastruttura, il livello piattaforma ed il livello applicazioni come mostrato in figura 1.2. Di seguito vengono descritti più dettagliatamente:

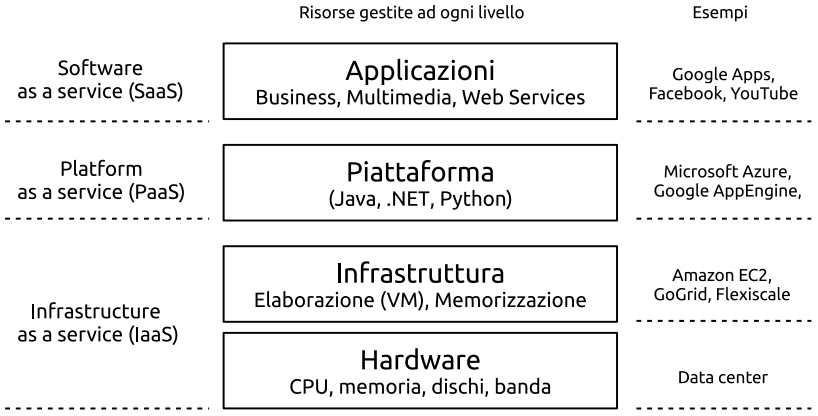


Figura 1.2: Architettura di un ambiente cloud computing

*Livello hardware:* questo livello è responsabile della gestione delle risorse fisiche come i server, router, sistemi di alimentazione e raffreddamento; in pratica, questo livello è implementato nei *data center*, i quali contengono migliaia di server. I problemi tipici includono la configurazione, la tolleranza ai guasti e la gestione del traffico dati.

*Livello infrastruttura:* conosciuto anche come livello di virtualizzazione, questo livello partiziona e distribuisce le risorse fisiche attraverso l'uso di tecnologie di virtualizzazione come KVM [9] e VMware [19]. Questo livello è essenziale per il modello poiché proprio grazie alle tecnologie di virtualizzazione è possibile implementare l'allocazione dinamica delle risorse.

*Livello piattaforma:* è formato dai sistemi operativi e dai framework applicativi ed è costruito sul livello infrastruttura, il suo compito è facilitare il lavoro di spiegamento (o deployment) delle applicazioni nelle macchine virtuali. Un esempio di piattaforma è Google App Engine [5] che mette a

disposizione dello sviluppatore le API per supportare l'implementazione di applicazioni cloud.

*Livello applicazione:* in cima ai livelli architetturali, ospita effettivamente le applicazioni cloud che grazie ai livelli sottostanti possono beneficiare delle caratteristiche del paradigma. Le differenze principali con un tradizionale servizio di hosting risiedono nella modularità e nel lasco accoppiamento tra i livelli, ciò permette di far evolvere ciascun livello separatamente.

### 1.2.4 Il modello di business

Il cloud computing impiega un modello di business orientato ai servizi; concettualmente ogni livello dell'architettura visto nella sezione 1.2.3 può essere implementato come un servizio, ed erogato su richiesta ai livelli superiori o sottostanti. In pratica, i servizi offerti possono essere raggruppati in tre categorie: Software as Service (SaaS), Platform as a Service (PaaS) e Infrastructure as a Service (IaaS). In accordo con l'architettura, è possibile che un fornitore PaaS si appoggi ad un fornitore IaaS esterno alla sua azienda; più spesso però i fornitori di PaaS e IaaS fanno parte della stessa organizzazione. Per questo motivo ci si riferisce ai fornitori PaaS e IaaS come ai fornitori di infrastruttura o *fornitori di cloud* [3], il servizio da essi fornito è detto utility computing; di seguito nel testo, si fa spesso riferimento ai fornitori di servizi come agli *utenti cloud*. Il modello di business è raffigurato in figura 1.3.

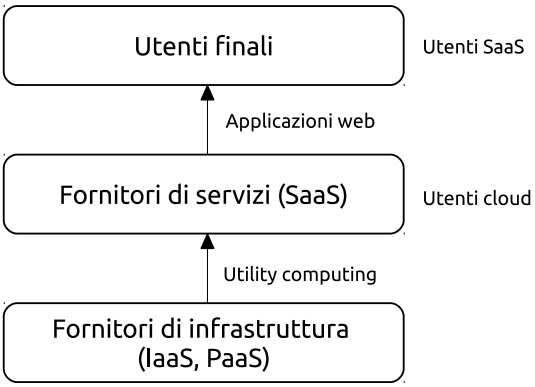


Figura 1.3: Modello di business del cloud computing

### 1.2.5 Caratteristiche

Vediamo in questa sezione le caratteristiche principali del cloud computing che lo distinguono dal modello tradizionale.

*Multi-tenancy:* letteralmente, multi proprietà, indica la convivenza di più servizi ospitati nello stesso data center le cui istanze sono possedute da diversi proprietari (tenants); la gestione delle problematiche e delle prestazioni di tali servizi è ripartita tra i fornitori di infrastruttura e di servizi. Sebbene la divisione architetturale in livelli fornisca una naturale divisione di responsabilità, è necessario comprendere e gestire le interazioni tra tutti i portatori di interesse.

*Raggruppamento delle risorse:* le tecnologie di virtualizzazione permettono di gestire in maniera più flessibile le risorse fisiche nei data center. I fornitori di infrastruttura, possono sfruttare ad esempio la migrazione dei processi per ottenere un alto livello di consolidamento dei server, riducendo di conseguenza i consumi di alimentazione e raffreddamento [17].

*Orientamento al servizio:* come già detto nella sezione 1.2.4, il modello del cloud computing è fortemente orientato al servizio, questo impone una gestione efficace dei servizi. L'erogazione dei servizi è vincolata da un accordo sul livello del servizio negoziato tra il fornitore e l'utilizzatore, detto *Service Level Agreement (SLA)*. Lo SLA definisce formalmente ed esplicitamente le caratteristiche come costo, tempo, responsabilità delle parti e performance del servizio; Il rispetto e dello SLA è un obiettivo critico per ogni fornitore.

*Fornitura dinamica delle risorse:* una caratteristica chiave del modello è la capacità di aggiungere e rimuovere risorse rapidamente, il modello tradizionale prevede invece un'attenta analisi preventiva per far fronte ai picchi di domanda da parte degli utenti. La gestione automatizzata delle risorse permette inoltre di fronteggiare rapidi aumenti della domanda come il *flash crowd effect*.

### 1.2.6 Prodotti commerciali

#### Amazon EC2

Amazon EC2 è la piattaforma cloud proposta da Amazon, è una piattaforma di basso livello, mette a disposizione risorse di computazione, non fornisce un servizio built-in per la gestione di task.

#### Microsoft Windows Azure

Anche Microsoft propone una piattaforma proprietari chiamata *Azure*, nemmeno questa piattaforma fornisce un servizio di schedulazione e gestione di task al momento della scrittura di questo lavoro.

#### Google App Engine

La piattaforma cloud di Google include tra le sue funzionalità un meccanismo per eseguire task chiamato *Task Queue* che per alcune caratteristiche visibili dal lato utente è simile al sistema progettato in questa tesi. Task Queue permette di schedulare lavori batch sulla piattaforma cloud ed integrare tale funzionalità con altre applicazioni, le applicazioni riceveranno poi al completamento del job delle notifiche di completamento.

Sebbene la funzionalità fornita ed alcuni degli obiettivi di Task Queue siano simili, il sistema di Google non prende in considerazione la gestione di un workflow e non è possibile utilizzarlo in personal cloud.

#### Flux Enterprise Scheduler

Flux è uno scheduler molto potente che fornisce una serie di funzionalità tra cui l'uso in private cloud, la gestione dei workflow e un'interfaccia grafica. Il principale svantaggio è che Flux è un prodotto proprietario ed il costo delle licenze è molto elevato.

### 1.2.7 Problematiche aperte

Sebbene il cloud computing sia ad oggi largamente utilizzato nell'industria IT, il suo sviluppo è ancora allo stadio iniziale e molte problematiche non sono state ancora adeguatamente risolte. Vediamo in questa sezione una panoramica delle principali questioni da affrontare.

#### Allocazione efficiente delle risorse

L'obiettivo principale dei fornitori di cloud è allocare e deallocare le risorse rispettando lo SLA e minimizzando i costi operativi; in particolare, è necessario garantire la qualità delle prestazionali del servizio detta *Quality of Service* (QoS) come ad esempio disponibilità, tempo di risposta e throughput. Non è facile mappare i requisiti del servizio alle risorse fisiche come CPU, banda e memoria soprattutto perché è necessario scalare rapidamente.

Il dimensionamento automatizzato dei servizi non è un problema recente ed è stato ampiamente studiato in passato. L'approccio tipico, impiega la costruzione di un "modello delle prestazioni" che preveda il numero di istanze di applicazione necessarie per garantire la QoS; l'esecuzione periodica di tale algoritmo e l'allocazione delle risorse in base alle previsioni ottenute. Il modello delle prestazioni può essere costruito usando varie tecniche, tra cui la teoria delle code e la teoria dei controlli.

#### Virtualizzazione e migrazione

La virtualizzazione fornisce importanti vantaggi al cloud computing tra cui la *migrazione live* che permette di spostare fisicamente dei processi sospendendo la loro esecuzione solo per un breve lasso di tempo (pochi millisecondi). È possibile inoltre spostare intere macchine virtuali, la migrazione di un intero sistema operativo e di tutte le sue applicazioni come singola unità riduce i problemi riscontrati nella migrazione a livello di processo [4]. Uno dei principale vantaggi della migrazione è il bilanciamento del carico di lavoro sui server dei data center per ridurre il surriscaldamento ed il consu-

mo di energia; nonostante i progressi fatti negli ultimi anni, si riscontra una mancanza di agilità nel rispondere a variazioni di carico improvvise.

### Sicurezza dei dati

La sicurezza è una delle principali obiezioni al cloud computing. Poiché i fornitori di servizi non hanno accesso ai sistemi di sicurezza implementati nei data center, devono fare affidamento interamente sui fornitori di infrastruttura per tutto ciò che riguarda la sicurezza dei dati. I due requisiti principali sono la *confidenzialità* dei dati e l'*auditability*, quest'ultimo comporta una valutazione della sicurezza e la produzione di documenti che ne attestino lo stato effettivo.

Gli utenti cloud (o fornitori dei servizi) devono fronteggiare minacce alla sicurezza sia interne che esterne. Molte minacce esterne sono simili a quelle già riscontrate anche nei data center non necessariamente facenti parte di una infrastruttura cloud; la differenza sta nella divisione delle responsabilità tra le parti. L'utente cloud è responsabile per la sicurezza a livello applicazione, deve quindi assicurarsi che sia sicura prima di schierarla. Il fornitore di cloud è responsabile per la sicurezza fisica e le minacce esterne, mentre per i livelli software intermedi come ad esempio le piattaforme, le responsabilità sono condivise tra l'utente ed il fornitore della piattaforma.

Sebbene il cloud computing renda più semplice la gestione di minacce esterne, introduce una serie di minacce interne da considerare. I fornitori di cloud devono proteggersi da eventuali furti di dati o attacchi degli utenti stessi, gli utenti devono essere protetti l'un l'altro. Il meccanismo attualmente utilizzato è la virtualizzazione e sebbene sia efficace, non tutte le risorse sono virtualizzate e non tutti gli strumenti di virtualizzazione sono esenti da falle.

Un'altra importante questione è la protezione degli utenti di cloud verso del fornitore, questo problema è comune in molti altri contesti. Il fornitore di cloud è situato al livello più basso dell'architettura e potrebbe volontariamente o involontariamente eludere i sistemi di protezione presenti ai livelli

superiori. In generale, la crittografia a livello utente e i concetti del *trusted computing* [14] possono migliorare la sicurezza, anche se l'applicazione di tali soluzioni risulta complessa in un ambiente cloud; inoltre, agli strumenti tecnologici possono sempre essere affiancati strumenti legali come i contratti.

#### Debugging nei sistemi distribuiti

Il debugging, o individuazione degli errori (bug) nel software, è un'attività non facile quando si considerano sistemi distribuiti su larga scala. Una caratteristica di questi sistemi è che i bug spesso non sono riproducibili se si analizza il sistema "in scala ridotta"; l'infrastruttura che compone la cloud è sostanzialmente un sistema distribuito su larga scala, come tale necessita di debugging quando il sistema è completamente schierato. Ancora una volta, la virtualizzazione potrebbe facilitare questo compito e rendere possibile l'acquisizione di informazioni che altrimenti non sarebbe possibile ottenere.

## 1.3 Java RMI e Codebase

Java Remote Method Invocation (RMI) fornisce un modello per la creazione e l'utilizzo remoto di oggetti in un ambiente distribuito. RMI è un'estensione del tradizionale meccanismo di invocazione remota delle procedure (Java RPC), grazie al meccanismo della serializzazione, il modello permette lo scambio di oggetti che includono sia il codice che lo stato. Gli oggetti che vengono trasferiti tra i processi comunicanti sono detti "mobile code", l'uso del mobile code e si contrappone alle tecniche in cui i processi scambiano tra loro solo i dati.

Lo scambio di oggetti tra JVM [6] differenti impone che i due host mettano a disposizione le definizioni degli oggetti che vogliono scambiarsi attraverso l'uso di una terza entità detta *codebase*. La codebase è un web server accessibile da tutti i gli host che utilizzano il mobile code e contiene le classi necessarie alla definizione degli oggetti.

[6] JVM: Java Virtual Machine

RMI fornisce una solida piattaforma ampiamente adottata per realizzare sistemi distribuiti object-oriented con tutti i vantaggi della programmazione in Java tra cui e sicurezza, portabilità e riuso.

I due meccanismi sopra descritti sono alla base di molti framework tra cui *Apache River* che sarà introdotto nella sezione 1.4

## 1.4 Apache River

### 1.4.1 Introduzione

Apache River [7] è un framework open source supportato dalla *Apache Software Foundation* [6] che definisce un modello di programmazione per la costruzione di sistemi distribuiti con architettura orientata ai servizi. River estende Java facilitando la costruzione di sistemi distribuiti flessibili, scalabili e sicuri in contesti in cui il cambiamento e la dinamicità della rete sono significativi.

### 1.4.2 Obiettivi di Apache River

River è basato sul concetto di aggregazione in gruppi, o federazioni, formate dagli utenti che ne sono membri e dalle risorse utili a tali membri. Lo scopo principale del sistema è permettere l'uso della rete come strumento facilmente configurabile per l'individuazione e l'accesso alle risorse da parte di utenti sia persone che client software. Sono elencati di seguito i principali obiettivi del sistema.

- Permettere agli utenti la condivisione dei servizi e delle risorse sulla rete.
- Fornire un accesso semplice alle risorse permettendo all'utente di cambiare posizione nella rete.
- Semplificare il compito di costruzione, gestione e modifica di una rete di dispositivi, servizi e utenti.

Le risorse possono essere software, come un servizio di stampa, hardware come una periferica fisica, o una combinazione delle due tipologie precedenti. Una caratteristica fondamentale è l'abilità di aggiungere o rimuovere risorse in maniera dinamica garantendo un alto livello di flessibilità ai servizi forniti da un sistema di questo genere. Di seguito sono elencate le parti che compongono un sistema realizzato con River.

- L'*infrastruttura*, le cui componenti permettono il funzionamento dei servizi nel sistema distribuito.
- Un *modello di programmazione* che supporta e promuove la costruzione di servizi distribuiti affidabili.
- I *servizi* che fanno parte del sistema e che forniscono le loro funzionalità a tutti i membri del gruppo.

Sebbene le parti sopra elencate siano separabili e distinte, esse sono interrelate rendendo la distinzione più sfumata nella realtà. Ad esempio, le componenti dell'infrastruttura e i servizi fanno uso del modello di programmazione, ma il modello di programmazione è sopratutto supportato dall'architettura stessa.

### 1.4.3 Concetti chiave

River permette di raggruppare componenti software e dispositivi in quello che appare all'utente come un singolo sistema distribuito. Vediamo in questa sezione quali sono i concetti fondamentali che permettono al framework di realizzare questo obiettivo.

#### Servizi

Il concetto più importante dell'intera architettura è il sevizio. Come già visto nella sezione 1.1.2, un servizio è un'entità che può essere usata da una persona, un software o un altro servizio. In River, i servizi sono definiti

attraverso interfacce Java e possono fornire ad esempio la memorizzazione di dati, il calcolo di una operazione, un canale di comunicazione o l'accesso ad una periferica. Gli utenti di un sistema River si affacciano al sistema per condividere appunto l'accesso ai servizi. Alla luce di questo concetto, il sistema non è un insieme di client e server o utenti e programmi, bensì un insieme di servizi che possono essere combinati per eseguire un particolare compito.

River mette a disposizione gli strumenti per la costruzione e la ricerca di servizi mediante l'infrastruttura. La scelta del protocollo per la comunicazione tra i servizi è libera, gli sviluppatori possono scegliere il protocollo più adatto alle esigenze del caso, una scelta classica è Java RMI. I servizi River più importanti sono elencati e descritti brevemente di seguito.

- *Lookup Service*: permette la pubblicazione e la ricerca dei servizi sulla rete.
- *Transaction Manager Service*: permette di utilizzare un meccanismo di transazioni con protocollo two-phase commit.
- *Lease Renewal Service*: è il servizio che implementa il meccanismo di leasing per i servizi registrati.
- *Lookup Discovery Service*: permette la ricerca dei servizi di lookup presenti sulla rete.
- *JavaSpaces Service*: implementa uno spazio distribuito in cui i processi possono scrivere e leggere dati. É uno strumento di coordinazione che prende spunto dal linguaggio Linda.[7]

### Il servizio di lookup

La ricerca di servizi (lookup) avviene attraverso l'omonimo *servizio di lookup*, esso fornisce il punto di contatto tra il sistema e gli utenti. Più preci-

[7]Linda è stato sviluppato da David Gelernter presso la Yale University.

samente, un servizio di lookup collega le interfacce che descrivono le funzionalità del servizio all'oggetto che implementa realmente il servizio, ossia è un registry. Ad ogni servizio registrato, possono essere associate informazioni o proprietà attraverso dei campi descrittivi che sono più significativi per gli utenti umani.

Un servizio di lookup può contenere riferimenti ad altri servizi di lookup così da ottenere una struttura gerarchica. Inoltre, un servizio di lookup può contenere oggetti che incapsulano riferimenti ad altri naming server (servizi di nomi), in questo modo client di sistemi diversi possono accedere ai servizi forniti da un sistema River.

### Il lesaing

In un sistema distribuito, la possibilità di un guasto ad una parte del sistema è una possibilità reale e da gestire. Non si può sapere se e quando un servizio fornito da un altro nodo smetterà di funzionare; a tale scopo River introduce un meccanismo di leasing per capire se un servizio è disponibile (o "vivo") in un dato istante.

Il leasing si basa su una registrazione periodica al servizio di lookup, il servizio che vuole registrarsi è tenuto a rinnovare la registrazione ad intervalli regolari prima della scadenza della stessa.

Il tempo di rinnovo dipende dal tipo di servizio e dai requisiti del sistema, in un sistema che effettua lavori batch non strettamente real-time, come quello oggetto di questo lavoro, un tempo di leasing di qualche minuto è più che sufficiente per avere un'idea dei servizi disponibili in ogni momento.

### Gli eventi distribuiti

Le componenti di un sistema software possono cambiare il loro stato ed aver bisogno di informare le altre componenti del cambiamento avvenuto. Il concetto di evento è presente da tempo nei linguaggi di programmazione, il modello tipico prevede che ci sia una sorgente generatrice di eventi in risposta ai cambiamenti di stato, ed uno o più parti interessate che si registrano per

ricevere tali eventi. Questo modello è alla base di un design pattern object oriented chiamato *observer*.

In un sistema distribuito è necessario notificare il cambiamento di una parte del sistema ad altre parti in maniere remota. River supporta il meccanismo degli eventi distribuiti per permettere alle parti del sistema di interagire in maniera asincrona. Un servizio River può registrarsi come "listener" ad un altro servizio e ricevere gli eventi da esso generati; anche in questo tipo di registrazione viene impiegato il meccanismo del leasing per tollerare i guasti.

## 1.5 Quartz Enterprise Job Scheduler

### 1.5.1 Lo scheduling

Lo scheduler (letteralmente, pianificatore) è un componente fondamentale dei sistemi operativi multitasking in grado di far eseguire al processore, attraverso l'omonima operazione di scheduling, più processi (task) concorrentemente attraverso varie politiche di scheduling. Esso è responsabile dell'assegnazione delle risorse di elaborazione ai vari processi in modo che siano distribuite efficientemente secondo determinati criteri implementati tramite algoritmi di scheduling.

Nel dettaglio, un algoritmo di scheduling si occupa di far avanzare uno dei processi interrompendo quello in esecuzione, realizzando così quello che si chiama cambio di contesto o *context switch* all'interno del ciclo di esecuzione del processore. Principalmente, un algoritmo di scheduling si pone due obiettivi: equità (Fairness), processi dello stesso tipo devono avere trattamenti simili, e bilanciamento (Balance), tutte le parti del sistema devono essere sfruttate in maniera equilibrata.

Il concetto di scheduling si può estendere a tutte le applicazioni che hanno bisogno di eseguire task in un preciso istante o ripetutamente nel tempo; molte applicazioni odierne utilizzano uno scheduler per attività quali la generazione di report o l'esecuzione di procedure di manutenzione. Esistono di-

verse implementazioni di scheduler che possono essere integrati direttamente nelle applicazioni, in questo lavoro di tesi si è scelto di utilizzare Quartz.

### 1.5.2 Quartz scheduler

Quartz [15] è uno scheduler open source che può essere integrato con qualsiasi applicazione scritta in Java; è uno strumento semplice e potente per pianificare l'esecuzione di task anch'essi definibili come semplici classi Java. Nonostante la sua semplicità, Quartz si adatta bene allo sviluppo di applicazioni che spaziano dalla semplice applicazione indipendente, che ha bisogno di pianificare poche decine di task, ad applicazioni molto complesse come quelle di commercio elettronico che gestiscono di decine di migliaia di task.

Per l'implementazione di questo lavoro di tesi si è utilizzato Quartz come punto di partenza per le necessità di scheduling. È opportuno sottolineare, che alla versione attuale di Quartz (2.0.1) non è possibile gestire la concatenazione dei job per formare un workflow più complesso; tale funzionalità è stata definita nel corso di questo lavoro.

#### Job e Trigger

I due concetti fondamentali che permettono la schedulazione in Quartz sono il *job* ed il *trigger*.

Il job (dall'inglese lavoro), è essenzialmente l'operazione che si vuole pianificare, teoricamente più essere qualsiasi operazione eseguibile da un elaboratore, nel caso di Quartz è una classe Java che contiene un metodo che viene al richiamato quando necessario.

Il trigger (innesco) definisce invece quando e quante volte eseguire un job. Sebbene siano spesso usate insieme, la separazione delle due entità è intenzionale e benefica poiché permette di modificare i dettagli sull'esecuzione senza modificare il job; inoltre, un job può essere associato a più trigger.

# Capitolo 2

# Progettazione

## 2.1 Concetti fondamentali

Nella sezione vengono esposti concetti fondamentali per una migliore comprensione della parte seguente di questo capitolo. Vale la pena ricordare che il progetto si pone l'obiettivo di definire un'architettura per lo scheduling e l'esecuzione distribuita di task.

### 2.1.1 Job

Per job si intende una qualsiasi azione semplice o composta che viene eseguita al fine di ottenere un risultato. Ogni job ha un identificativo unico nel sistema che è composto da due parti: una chiave detta *job key* ed un gruppo detto *job group*, combinate indicano univocamente un job nel sistema. Un job è eseguito in un *contesto* che comprende una struttura dati in cui salvare i risultati di esecuzione detta *Job Data Map*, ed altre informazioni di esecuzione come ad esempio le eccezioni.

Nel corso dell'esecuzione, un job è caratterizzato da uno *stato di esecuzione* che rappresenta il suo ciclo di vita; di seguito sono elencati gli stati corredati della relativa descrizione.

- INCEPTION: rappresenta lo stato iniziale, il job è stato appena creato e non è stato mai eseguito.
- STARTED: l'esecuzione del job è iniziata ma non ancora terminata.
- EXECUTED: il job è stato eseguito correttamente senza eccezioni e l'esecuzione è terminata.
- NOT EXECUTED: il job non è stato eseguito per via di una eccezione a cui è possibile ovviare ritentando l'esecuzione.
- EXECUTED WITH EXCEPTION: il job è stato eseguito ma durante l'esecuzione si è verificata una eccezione che non può essere risolta se non con una "compensazione" del job.

In figura 2.1 si mostra il modello delle transizioni tra gli stati di esecuzione di un job. Come è possibile notare nel modello, un job correttamente terminato permette, se presente, l'esecuzione del job successivo come definito nel workflow. Qualora il job sia stato eseguito con eccezioni, vengono intraprese azioni correttive.

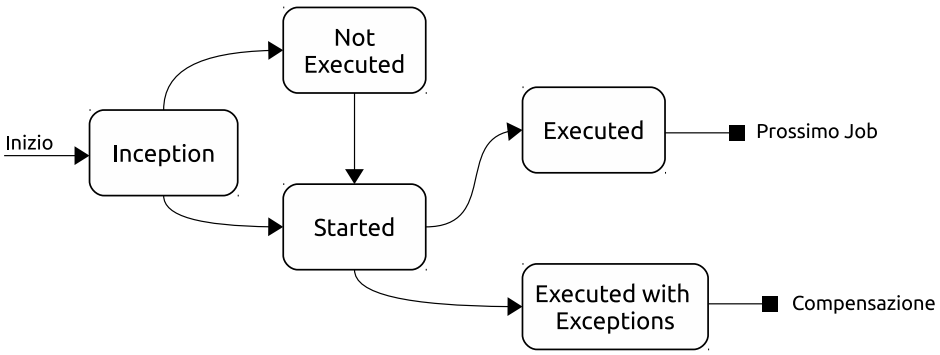


Figura 2.1: Modello delle transizioni tra gli stati di esecuzione di un job

### 2.1.2 Workflow di Job

Si può immaginare il workflow come una sequenza di job connessi che vanno eseguiti in un ordine prestabilito rispettando dei vincoli definiti. Soli-

tamente i job appartenenti allo stesso workflow concorrono alla realizzazione di un task comune composto logicamente da tanti passi quanti sono i job che lo compongono.

Il workflow più semplice è rappresentato da una sequenza di job in serie in cui l'unico vincolo è la dipendenza dal job precedente. Il vincolo di dipendenza è dettato dal fatto che il risultato dell'elaborazione di un job sarà l'input per l'esecuzione del job successivo.

Non sempre il workflow viene correttamente completato nella sua interezza, può capitare che un job fallisca lasciando il task comune incompleto o peggio in uno stato inconsistente; l'annullamento o l'interruzione del workflow e quindi del task spesso non rappresentano una soluzione praticabile nei contesti reali. A tal proposito è stato introdotto in questo lavoro il concetto di *compensazione*. La compensazione, già trattata in letteratura in [2], è un'azione correttiva, volendo un altro job, che si esegue in caso di errore e che permette di terminare il workflow "invertendo" il senso di esecuzione. Il meccanismo della compensazione permette al job di effettuare un'azione correttiva a posteriori avendo a disposizione dati "futuri".

In figura 2.2 è mostrato un workflow in serie con compensazione per ogni job, si noti il senso delle frecce e la possibilità di completare il workflow in due punti.

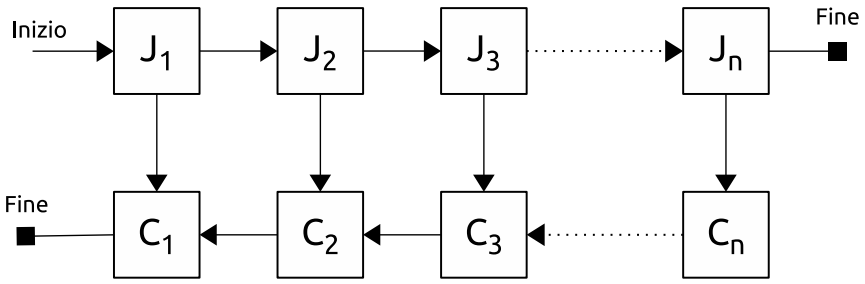


Figura 2.2: Rappresentazione di un workflow di job sequenziale

Per lo sviluppo e la validazione del sistema oggetto di questo lavoro ci concentreremo su un workflow come visto in figura 2.2; questa scelta fornisce un'ottima base per testare le potenzialità del sistema senza essere al

contempo troppo limitativa. Nell'attuale implementazione è già presente un workflow manager estendibile, ma si auspica di introdurre negli sviluppi futuri modelli più complessi di workflow.

### 2.1.3 Schedulazione

La schedulazione è la modalità con cui l'utente finale o i client interagiscono con il sistema e definiscono i task da eseguire. Il sistema deve gestire diversi task di diversi utenti simultaneamente, ogni schedulazione sottoposta al sistema riceve, come per i job, un identificativo unico nel sistema. La specifica di una schedulazione deve includere tre elementi non opzionali descritti di seguito.

- *Task descriptor:* un file in formato XML nel quale è definita la sequenza di job da eseguire insieme agli eventuali parametri di esecuzione; un esempio di task descriptor è mostrato nel listato 2.1.

- *Cron expression:* è un'espressione che indica gli istanti di tempo in cui far partire l'esecuzione del workflow; ha una sintassi ispirata ad un noto scheduler di job originariamente sviluppato per sistemi operativi Unix-like chiamato Cron. Una schedulazione è attivata ad intervalli regolari secondo quanto specificato dalla Cron expression, ogni esecuzione è detta "run".

- *Jar file:* uno o più file contenenti fisicamente le classi Java referenziate nel task descriptor. La sintassi dei job obbedisce alla sintassi adottata da Quartz, come già accennato nella sezione 1.5.2.

Codice 2.1: Esempio di Task Descriptor in formato XML.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <task name="Submission Test">
3    <job jobClass="jobs1.Job1" compensationClass="jobs1.Comp1" order="1">
4       <jar>job1.jar</jar>
5    </job>
6    <job jobClass="jobs1.Job2" compensationClass="jobs1.Comp2" order="2">
7       <jar>job1.jar</jar>
8    </job>
9    <job jobClass="jobs1.Job3" compensationClass="jobs1.Comp3" order="3">
10      <jar>job3.jar</jar>
11   </job>
12   <param name="Param" value="Default" />
13   <param name="OtherParam" value="OtherDefault" />
14 </task>
```

### 2.1.4 Coda di lavoro

Una coda di lavoro, come intesa in questa trattazione, è un cluster nodi in grado di eseguire job. I nodi che compongono la medesima coda sono aggregati in base ad una caratteristica comune, essa può essere una caratteristica prestazionale dei nodi o un raggruppamento logico qualsiasi come ad esempio la trattazione della stessa tipologia di task.

Si può decidere di creare, per esempio, una coda per i lavori di reportistica o una coda per i task cpu-bound che comprenda un gran numero di nodi ad alte prestazioni. La creazione di una nuova coda è dinamica, così come la modifica del numero di nodi. Due o più servizi appartengono alla stessa coda di lavoro se posseggono lo stesso attributo che contiene il nome della la coda di lavoro.

Nello specifico, i nodi sono delle istanze di *Queue Server* e l'attributo del servizio che indica la coda di lavoro è "QueueName". Questo concetto sarà chiarito nel corso della trattazione del della sezione 2.2.2 in cui verrà descritta l'architettura del livello middleware.

### 2.1.5 Orchestrazione

L'orchestrazione riguarda l'organizzazione e la coordinazione dei servizi e dei flussi di informazione in sistemi complessi come le architetture service oriented e più recentemente quelli di cloud computing. La componente che svolge il compito di orchestrazione è chiamata *orchestratore* e fornisce un punto di controllo sulla moltitudine di servizi che compongono un sistema. Caratteristica fondamentale è l'accoppiamento lasco tra orchestratore e servizi, quindi tra la logica del processo e i servizi utilizzati.

Si può vedere l'orchestrazione come un livello aggiuntivo e completo che svolge una funzione di integrazione dei servizi che possa permettere l'adattamento al cambiamento; ad esempio, non è necessario che tutti i servizi siano in esecuzione e disponibili nello stesso momento per permettere l'orchestrazione.

Nel sistema progettato si realizzato un orchestratore per la coordinazione dei servizi che compongono le code di lavoro, sarà descritto in dettaglio nella sezione 2.2.1.

## 2.2 L'architettura

In questa sezione si presenta l'architettura del sistema e le componenti che lo compongono. Si parte da una visione generale per chiarire le dinamiche di interazione delle tre macro-componenti che saranno meglio descritte nelle sotto sezioni immediatamente successive.

In figura 2.3, delimitate dai rettangoli tratteggiati, si notano tre componenti principali: l'orchestratore, i servizi di River e le code composte dai Queue Server. Ogni componente è ospitata su uno o più nodi della rete, l'orchestratore è unico nella rete mentre le altre componenti possono essere distribuite su un numero qualsiasi di nodi.

L'orchestratore usa i servizi di River per il discovery dei servizi da orchestrare e per creare una cache dei servizi presenti sulla rete. La cache fornisce all'orchestratore una panoramica aggiornata dei servizi; l'aggiunta, la rimozione o il cambiamento delle caratteristiche di un servizio viene comunicato

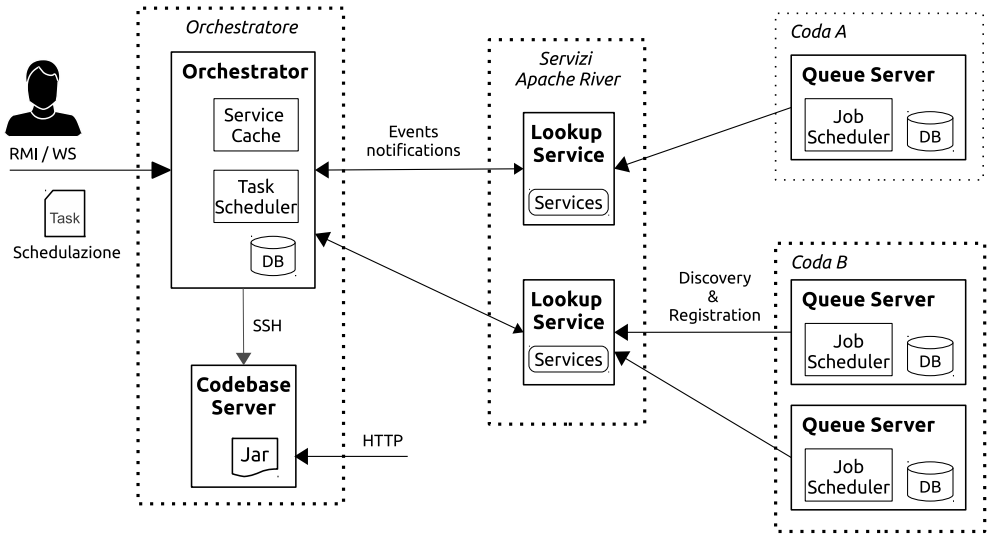


Figura 2.3: Visione generale dell'architettura del sistema

all'orchestratore attraverso delle notifiche generate dai servizi River. Inoltre, l'orchestratore potrebbe anch'esso registrarsi come servizio River e rendersi disponibile per essere usato a sua volta da altri servizi nella rete.

I servizi River permettono alle componenti la pubblicazione, la gestione e la fruizione di altri servizi. Parte dell'architettura del progetto è stata realizzati con l'ausilio di servizi River; sebbene l'implementazione di River sia matura, una fase di configurazione e specializzazione dei servizi è stata comunque necessaria.

Il più importante servizio fornito da River è quello di lookup, come già discusso nella sezione 1.4.3, esso permette il discovery dei servizi. Uno o più nodi che ospitano il servizio di lookup devono essere schierati sella rete.

Sempre in figura 2.3 si possono vedere due semplici code di lavoro a titolo esplicativo; le code di lavoro sono un raggruppamento logico di istanze di Queue Server. Ogni Queue Server esporta il proprio servizio di esecuzione job su tutti lookup service che scopre, da quel momento sarà disponibile per essere utilizzato dall'orchestratore. I Queue Server interagiscono anche con il codebase server qualora abbiano bisogno dei JAR necessari per eseguire un job.

### 2.2.1 L' orchestratore

In figura 2.4 è mostrata l'architettura dell'orchestratore in dettaglio, si passano in rassegna le componenti descrivendole nell'ordine in cui vengono utilizzate alla sottomissione di una schedulazione.

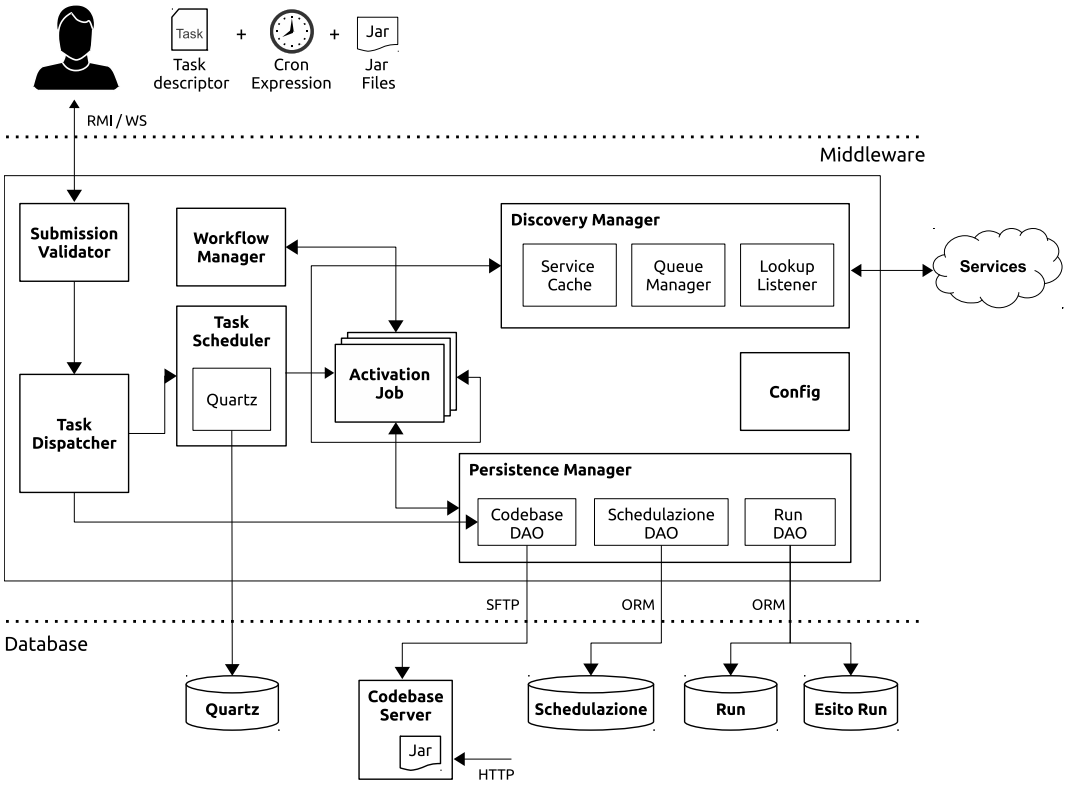


Figura 2.4: Architettura dettagliata del progetto Orchestratore

Come si può osservare in figura, l'interazione con l'utente avviene tramite RMI, il sistema è progettato per essere inoltre fruibile, con minime modifiche, tramite un Web Service o come servizio River. Il client sottopone la schedulazione all'orchestratore che viene validata tramite il *Submission Validator* per accertarne la correttezza, in questa fase iniziale vengono effettuate due operazioni importanti:

- i JAR contenenti le classi dei job vengono copiate tramite protocollo

SFTP [1] nel *Codebase Server* per renderle accessibili mediante HTTP;

- alla schedulazione viene assegnato un identificativo usando UUID [2] che viene restituito ala client come conferma della presa in carico del task e come handle per le operazioni future.

Eseguite le operazioni iniziali, la schedulazione viene passata al *Task Dispatcher*. Il Task Dispatcher elabora le richieste dei client secondo l'ordine di arrivo usando una coda: per ogni schedulazione delega la creazione le strutture dati necessarie a garantire la persistenza avvalendosi del *Persistence Manager*. Assicurata la persistenza, vengono eseguite le operazioni che permettono fisicamente l'esecuzione del task come specificato dall'utente:

- viene create il job che porterà avanti la schedulazione nelle sue successive esecuzioni; il job incaricato di "attivare" l'esecuzione del task è chiamato *Activation Job*, ogni schedulazione avrà un proprio Activation Job associato;
- viene creato un trigger in base all'espressione Cron fornita dal client;
- l'Activation Job con relativo trigger viene affidato al *Task Scheduler* che, attraverso l'istanza Quartz, ne assicura l'esecuzione a tempo debito.

L'activation Job è una componente fondamentale del sistema, gestisce il workflow di una singola schedulazione per tutte le esecuzioni della stessa. Le funzioni principali dell'Activation Job sono:

- controllo dell'esecuzione del task nella sua interezza;
- controllo e dell'esecuzione dei singoli job che compongono il task;
- assegnazione degli stati di esecuzione ai job ed al task;

[1] SFTP: Secure File Transfer Protocol
[2] UUID: Universally unique identifier

- uso dei servizi disponibili per l'orchestrazione.

In forte accoppiamento con l'Activation Job lavora il *Workflow Manager* che è la componente che si occupa di definire qual'è il successivo job del workflow da eseguire. Il Workflow Manager funziona in maniera simile ad un'automa a stati finiti: dato in input il job attuale, la struttura del workflow e lo stato di esecuzione del task, restituisce il prossimo job da eseguire.

Un'altra componente fondamentale è il *Discovery Manager* che ha il ruolo di interfaccia con i servizi della rete rappresentati in figura dalla nuvoletta. Il Discovery Manager è composto da tre sotto componenti che sono descritte di seguito.

La *Service Cache* è responsabile del monitoraggio i servizi disponibili e rappresenta il punto d'accesso per tali servizi. La cache è configurata per monitorare uno o più tipi di servizio, nel sistema progettato la Service Cache monitora i solo i servizi di tipo Queue Server permettendo due operazioni fondamentali: l'interrogazione in base all'identificativo o a filtri sulle caratteristiche del servizio e la registrazione al servizio per riceverne le notifiche.

Il *Lookup Listener* effettua il discovery dei servizi di lookup di River, la scoperta o la rimozione di un servizio di lookup genera una notifica per l'orchestratore, il quale può attuare le operazioni necessarie a gestire il cambiamento.

Terza ed ultima componente del Discovery Manager è il *Queue Manager*; tale componente provvede alla ricezione e allo smistamento delle notifiche ricevute dai servizi Queue Server. Le notifiche degli eventi informano il Queue Manager dello stato dei job in esecuzione sui Queue Server, questo meccanismo consente la sincronizzazione dei diversi workflow.

Altra funzione essenziale fornita dal Queue Manager è la richiesta di notifica o "notification booking": in condizioni particolari, una notifica può essere smarrita o non ricevuta, a tale scopo il Queue Manager permette di "prenotare" una notifica per un servizio non attualmente disponibile, la notifica sarà così ricevuta appena tale servizio tornerà disponibile.

L'accesso alle basi di dati dell'orchestratore è gestita dal *Persistence Manager*, che implementa il pattern *Data Access Object (DAO)* per le entità che hanno bisogno della persistenza usando una tecnica ORM [3].

Astraendo, la base di dati può essere divisa in tre sezioni ognuna con un DAO dedicato: la sezione che contiene i dati delle schedulazioni, la sezione dei dati delle run, ed in ultimo, la sezione dei risultati delle run. I dettagli delle tabelle saranno discussi nella sezione 2.2.3 relativa alle basi di dati.

Si può inoltre considerare appartenete alla persistenza l'accesso SFTP al server Codebase per la copia dei file JAR delle schedulazioni. Un'ulteriore parte della base di dati è utilizzata dall'istanza di Quartz, questa parte è rappresentata in figura connessa direttamente a Quartz poiché è gestita direttamente da Quartz e non dal gestore Persistence Manager.

La componente *Config*, apparentemente disconnessa solo per chiarezza di rappresentazione, è utilizzata dalle altre parti del sistema per la configurazione e la personalizzazione dell'orchestratore. Config utilizza un file di proprietà in cui sono definiti i parametri per l'accesso alla codebase ed altre impostazioni relative alle preferenze sui servizi. Un esempio di file di proprietà è riportato in appendice B.

### 2.2.2 Il Queue Server

Come già accennato, l'architettura progettata si compone di più istanze di Queue Server logicamente organizzati in code di lavoro. L'architettura dettagliata di un Queue Server è mostrata in figura 2.5. Di seguito se ne descrivono le componenti elencando le funzionalità fornite da ognuna.

La prima operazione che un Queue Server esegue è l'esportazione di se stesso come servizio River usando il *Service Exporter*. Una volta esporto, il servizio può ricevere una richiesta di registrazione dall'orchestratore che ne vuole usufruire. Le funzioni fornite dal Service Exporter sono:

- generazione di un identificativo persistente del servizio Queue Server;

[3] ORM: Object Relational Mapping

- ricerca e monitoraggio dei servizi di lookup presenti;
- esportazione del servizio Queue Server presso tutti i servizi di lookup trovati;
- rinnovo periodico del leasing per il servizio esportato.

Dopo l'esportazione il Queue Server avvia lo *Scheduler Engine*, ossia lo scheduler locale istanza di Quartz, che eseguirà solo job con esecuzione immediata poiché lo scheduling vero e proprio è eseguito dall'orchestratore. Lo Scheduler Engine include una componente chiamata *Job Listener* che è responsabile del monitoraggio dello stato di esecuzione dei job e dell'invio delle notifiche all'orchestratore.

L'esecuzione dei job implica il caricamento delle giuste classi dai JAR presenti sul codebase server, il Queue Server però non conosce la posizione del (o dei) server ed utilizza la componente *Custom Class Loader* per permettere allo Scheduler Engine di eseguire job che utilizzano classi remote.

Il *Crash Handler* è la componente che viene invocata in caso di guasti per intraprendere azioni correttive. Tale componente è utilizzata in due situazioni: al riavvio del Queue Server, per il ripristino di un guasto crash tramite la reinizializzazione dei class loader dei job interrotti; ed al fallimento di un'operazione di notifica dell'orchestratore, in modo da memorizzare la notifica per l'invio futuro.

Similmente all'orchestratore, il Queue Server gestisce la persistenza grazie al *Persistence Manager* che utilizza due Data Access Object per le operazioni con le notifiche e con i class loader. Anche in questo caso, la persistenza di Quartz è gestita dallo stesso Quartz.

Come nell'orchestratore, anche il Queue Server ha un meccanismo di configurazione che utilizza una file di proprietà per impostare alcuni parametri del servizio come ad esempio la coda di lavoro a cui aggregarsi. Un esempio di file di proprietà è riportato in appendice B.

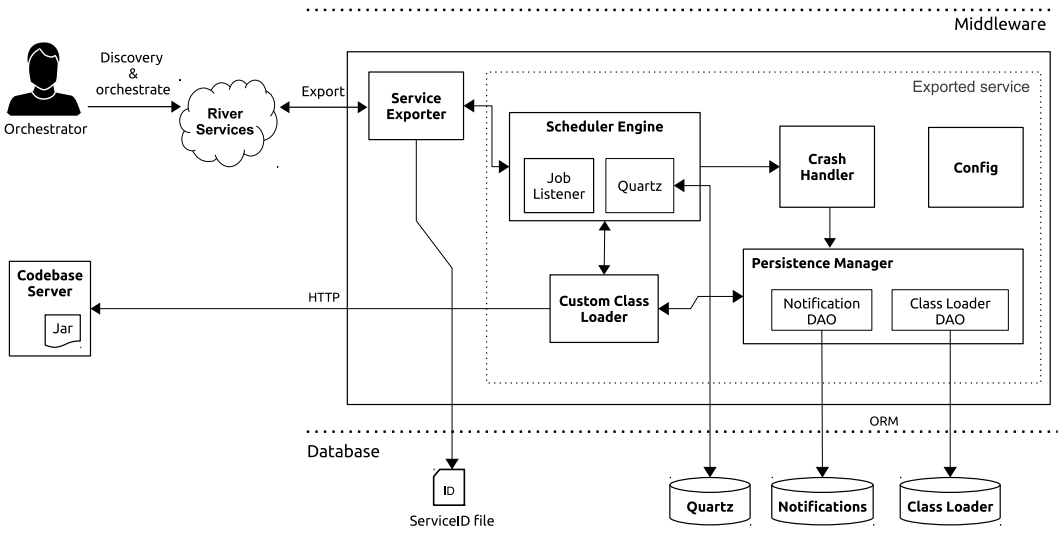


Figura 2.5: Architettura dettagliata del progetto Queue Server

### 2.2.3 Le basi di dati

Le basi di dati sono naturalmente distinte per progetto: si descrivono in ordine prima la base di dati del progetto Orchestratore e successivamente quella relativa al progetto Queue Server. Entrambi i progetti includono delle tabelle per il funzionamento di Quartz, tali tabelle saranno presentate solo brevemente in quanto esulano dallo scopo di questo elaborato.

#### Orchestratore DB

In figura 2.6 è mostrato lo schema Entity-Relationship del progetto orchestratore; segue una descrizione delle entità più significative.

**SCHEDULAZIONE:** ogni entry della tabella rappresenta una schedulazione sottomessa dall'utente all'orchestratore. Oltre al nome e alla data di inserimento, sono presenti il nome della coda di lavoro su cui eseguire la schedulazione e l'espressione Cron. Il campo SCHEDULAZIONE_ID è la chiave che permette di costruire le relazioni con i job, i parametri e i JAR della schedulazione.

**JOB_SCHEDULAZIONE:** in questa tabella sono inseriti tutti i job che compongono le schedulazioni, ogni entry è associata ad una ed una sola

schedulazione tramite l'attributo SCHEDULAZIONE_ID. Ciascun job ha un ordine di esecuzione relativo alla sua schedulazione ed indica le classi da eseguire nel caso predefinito e in caso di compensazione.

**SCHEDULAZIONE_PARAM:** ogni entry di questa tabella rappresenta un parametro associato ad una schedulazione nella forma nome-valore.

**CODEBASE:** questa tabella contiene i nomi dei JAR e il relativo contenuto per ogni job presente in JOB_SCHEDULAZIONE, la relazione è mantenuta tramite la tabella CODEBASE_X_JOBS.

**RUN_SCHEDULAZIONE:** ogni schedulazione viene eseguita periodicamente con conseguente esecuzione di tutto il workflow di job in essa contenuti; questa tabella contiene una riga per ogni esecuzione di ogni schedulazione. Ogni esecuzione della schedulazione è indicata da un numero sequenziale indicato nel campo RUN_NUMBER, mentre lo stato di esecuzione è rappresentato dalla costante nel campo STATO_ESECUZIONE.

**JOB_RUN:** in questa tabella sono memorizzate le esecuzioni parziali dei job relative ad ogni run delle schedulazione. Rappresenta una parte fondamentale dell'esecuzione dei workflow, a tal proposito si descrivono dettagliatamente i campi presenti.

- RUN_CLASS: la classe eseguita nella run del job; può essere la classe predefinita o la classe di compensazione in base ai job precedenti del workflow.
- STATO_ESECUZIONE e STATO_TERMINAZIONE: aggiornati durante l'esecuzione del job, indicano la presenza di errori o la corretta esecuzione del job.
- JOB_RUN_DETAIL: contiene l'oggetto serializzato contenente il risultato e i dati parziali dell'esecuzione del job. Tale oggetto viene usato dal job successivo per realizzare una concatenazione del risultato fino alla fine del workflow.
- RUN_QUEUE_SERVICE_ID: viene avvalorato con l'identificativo del servizio Queue Server appena il job inizia l'esecuzione. Questo campo

è necessario per tenere traccia del servizio che prende effettivamente in carico il job.

**ESITO_RUN_SCHEDULAZIONE:** memorizza i risultati delle esecuzione di ogni schedulazione. i due campi RISULTATO e TIPO_RISULTATO contengono rispettivamente il valore e il tipo della variabile risultato. Nell'implementazione realizzata ci si limita a risultati di tipo stringa ma la generalizzazione ai tipi serializzabili è immediata.

**UNAVAILABLE_SERVICE_WORKING:** nonostante il meccanismo di leasing alcuni servizi possono risultare disponibili quando in realtà non lo sono. In questa tabella sono inseriti i servizi nel momento in cui vengano interrogati sul completamento di un job e non risultino disponibili: ad esempio, dopo un crash, l'orchestratore interroga i servizi impegnati nell'esecuzione dei job, e richiede le notifiche che potrebbero essere andate perdute. Questa tabella è alla base del meccanismo di notification booking che permette di ricevere le notifiche non appena il servizio torni disponibile.

### Queue Server DB

In figura 2.7 è mostrato lo schema Entity-Relationship del progetto Queue Server; segue una descrizione delle entità.

JOB_CLASSLOADER: contiene la mappatura tra le classi dei job e gli URL che puntano ai JAR delle classi. La tabella contiene una riga per ogni job eseguito dal Queue Server. La tabella è utilizzata dal Custom Class Loader per gestire il caricamento delle classi remote.

NOTIFICATION_FAILED: questa tabella registra le notifiche remote il cui invio è fallito. Ogni record contiene l'identificativo e il gruppo del job per cui la notifica è fallita nei campi JOB_KEY_NAME e JOB_KEY_GROUP. L'elemento fondamentale della notifica, salvato nel campo REMOTE_EVENT, è l'oggetto Java che viene inviato al destinatario ed incapsula l'informazione.

Le tabelle Quartz formano una parte a se stante della base di dati, si invita il lettore che voglia approfondire a consultare la documentazione ufficiale di Quartz [15].

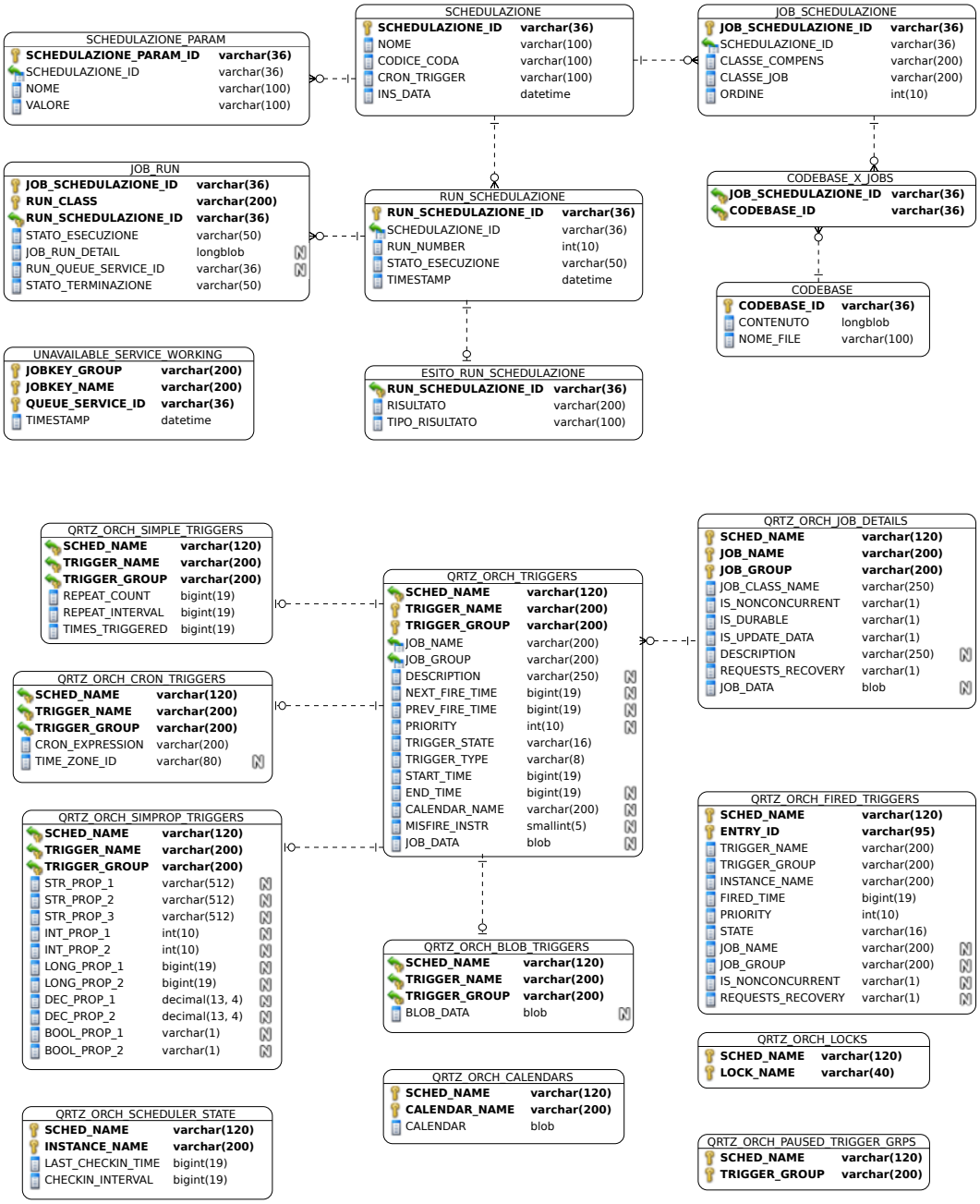


Figura 2.6: Modello Entità-Relazioni del progetto Orchestratore

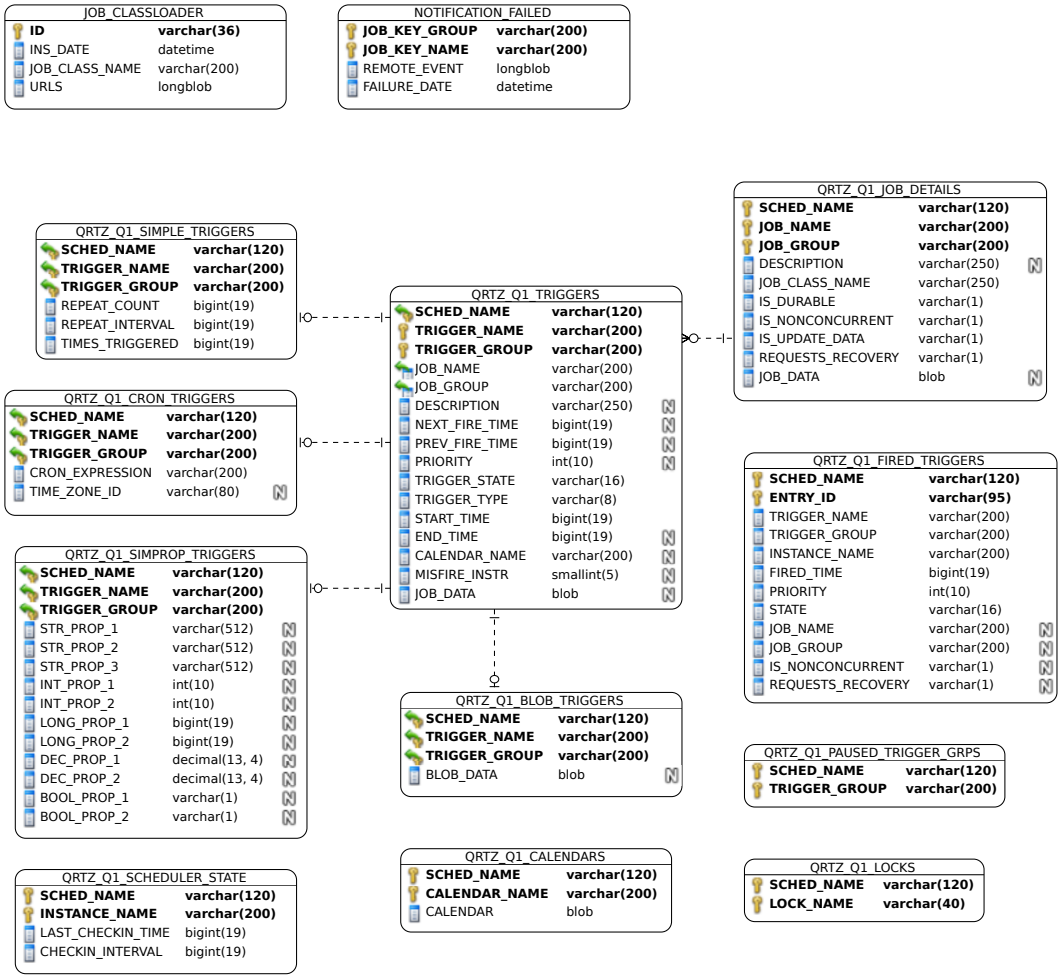


Figura 2.7: Modello Entità-Relazioni del progetto Queue Server

## 2.3 Casi d'uso

Per meglio comprendere il comportamento del sistema, si illustrano i principali casi d'uso tramite diagrammi di sequenza UML: la sottomissione di un task all'orchestratore, l'attivazione di un task, il recovery dell'orchestratore e del Queue Server.

### 2.3.1 Sottomissione di un task

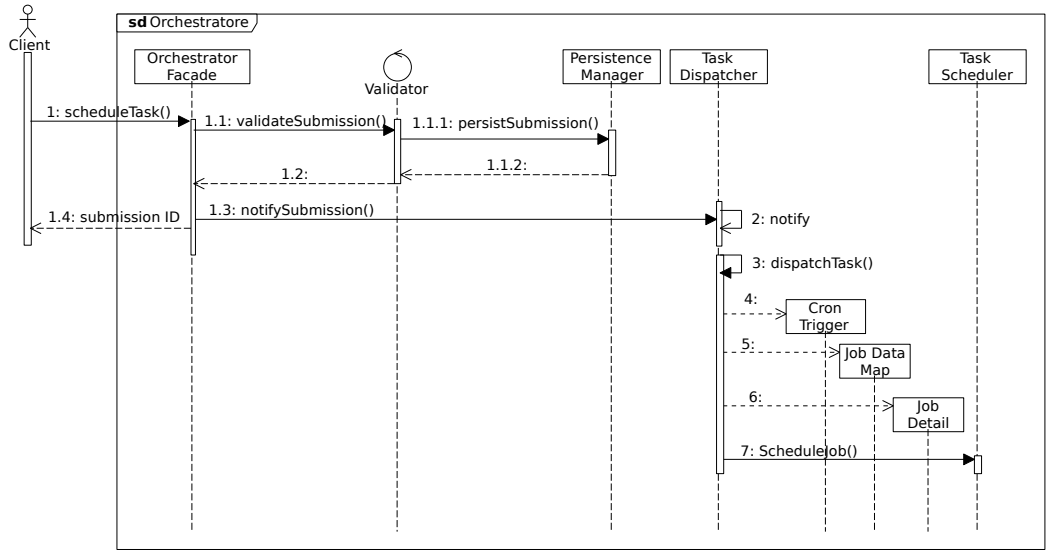


Figura 2.8: Caso d'uso: sottomissione di un task all'orchestratore.

In figura 2.8 si mostra il caso d'uso per la sottomissione di un task da parte di un client; la sottomissione di un task è iniziata dal client tramite la richiesta di schedulazione al façate dell'orchestratore. L'orchestratore controlla la correttezza e la completezza della schedulazione tramite l'operazione `validateSubmission()`, successivamente, i dati vengono memorizzati mediante l'operazione `persistSubmission()`. La schedulazione viene notificata al Task Dispatcher ed il client riceve l'identificativo della nuova schedulazione come conferma della presa in carico.

Ricevuta la notifica, il thread del Task Dispatcher preleva dalla sua coda la sottomissione gli oggetti che permettono effettivamente l'esecuzione del task: il Cron Trigger, la Job Data Map ed il Job Detail. Il Job Detail in questione è un'istanza di Activation Job che viene schedulato nel Task Scheduler tramite l'operazione `ScheduleJob()`. Il caso d'uso termina con l'Activation Job pronto per essere eseguito al momento indicato dal trigger, l'attivazione del task al fine di eseguire i job del workflow è il passo successivo che verrà illustrato nella sezione 2.3.2.

### 2.3.2 Attivazione di un task

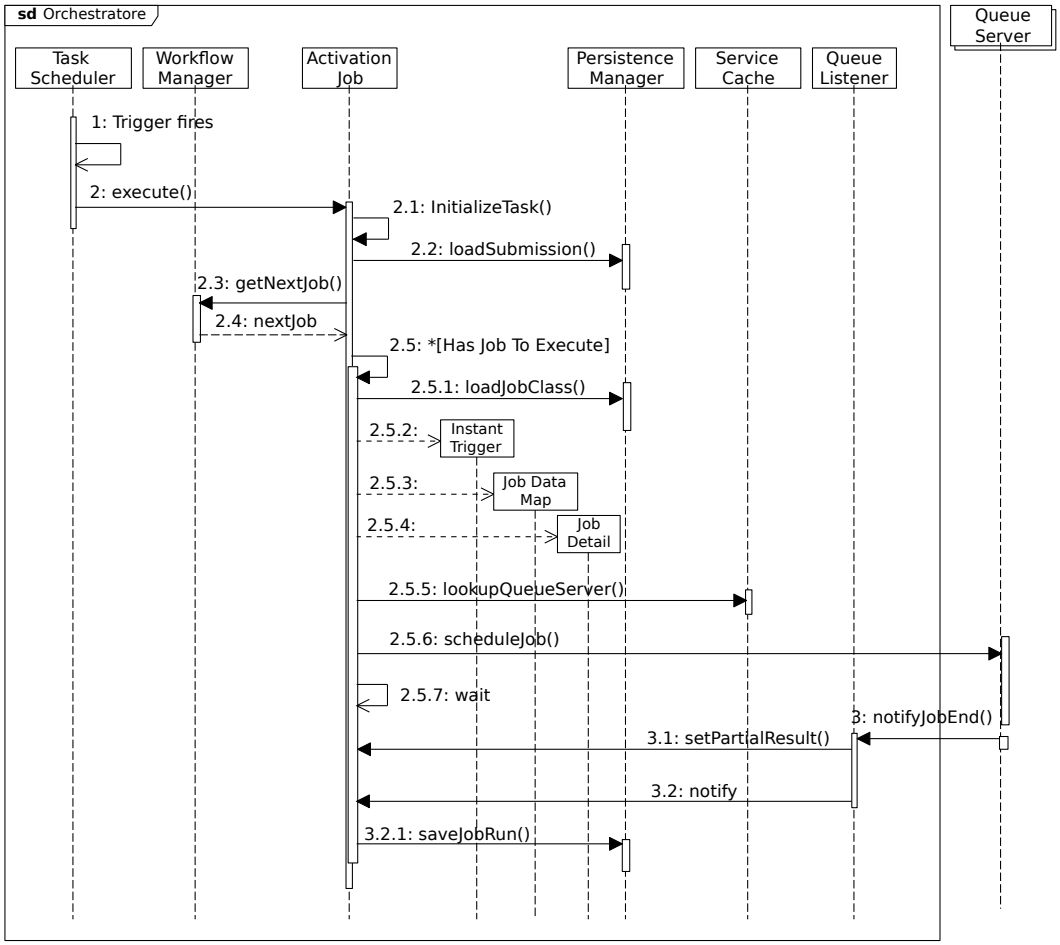


Figura 2.9: Caso d'uso: l'attivazione di un task.

L'attivazione di un task (figura 2.9) comincia allo scattare del Cron Trigger; il trigger innesca l'esecuzione del metodo `execute()` dell'Activation Job, il quale controllerà tutto il flusso di esecuzione del task.

L'activation Job richiama ad ogni esecuzione il metodo d'inizializzazione `initializeTask()` con cui prepara le variabili per la run corrente e carica i dati della schedulazione dal database. Dopo l'inizializzazione, l'esecuzione dei job incomincia: per ottenere il prossimo job del workflow da eseguire, l'Activation Job richiama il metodo `getNextJob()` del Workflow Manager. Fino a che ci sono job da eseguire, per ogni job del task, saranno eseguite le seguenti operazioni:

- caricamento delle classi del job da eseguire tramite l'operazione `loadJobClass()` e creazione di un class loader personalizzato per che utilizzi il codebase server via HTTP;
- creazione di un'istanza di trigger istantaneo, creazione del Job Detail ed aggiornamento della Job Data Map che contiene lo stato complessivo del task fino al job corrente;
- ricerca dei servizi Queue Server appartenenti alla coda di lavoro indicata nella schedulazione usando il metodo `lookupQueueServer()` della Service Cache;
- schedulazione del job appena creato in uno dei Queue Server trovati, il trigger istantaneo farà partire immediatamente l'esecuzione del job sul servizio di coda.

Dopo la schedulazione, l'Activation Job si pone in attesa della terminazione del job; il Queue Server che ha terminato l'esecuzione del job invierà una notifica di completamento al Queue Listener. La notifica contiene l'oggetto *JobRunDetail* che include la Job Data Map modificata dall'esecuzione dell'ultimo job, essa serve a costruire il risultato parziale del task. il Queue Listener consegna il il JobRunDetail all'Activation Job richiamando il metodo `setPartialResult()`, subito dopo notifica l'Activation Job che può

così salvare il risultato del job sul database e passare all'esecuzione del job successivo.

### 2.3.3 Recovery dell'orchestratore

Per le specifiche del progetto e per la sua unicità, il servizio di orchestrazione deve tollerare i guasti; inoltre, poiché si vuole un'architettura dinamica, deve essere possibile riavviare, sostituire o spostare l'orchestratore senza causare danni ai task in esecuzione: il recovery dell'orchestratore è quindi una funzionalità essenziale. In questa sezione si illustra il caso d'uso del recovery inteso come ripristino da un crash o dallo spegnimento dell'orchestratore.

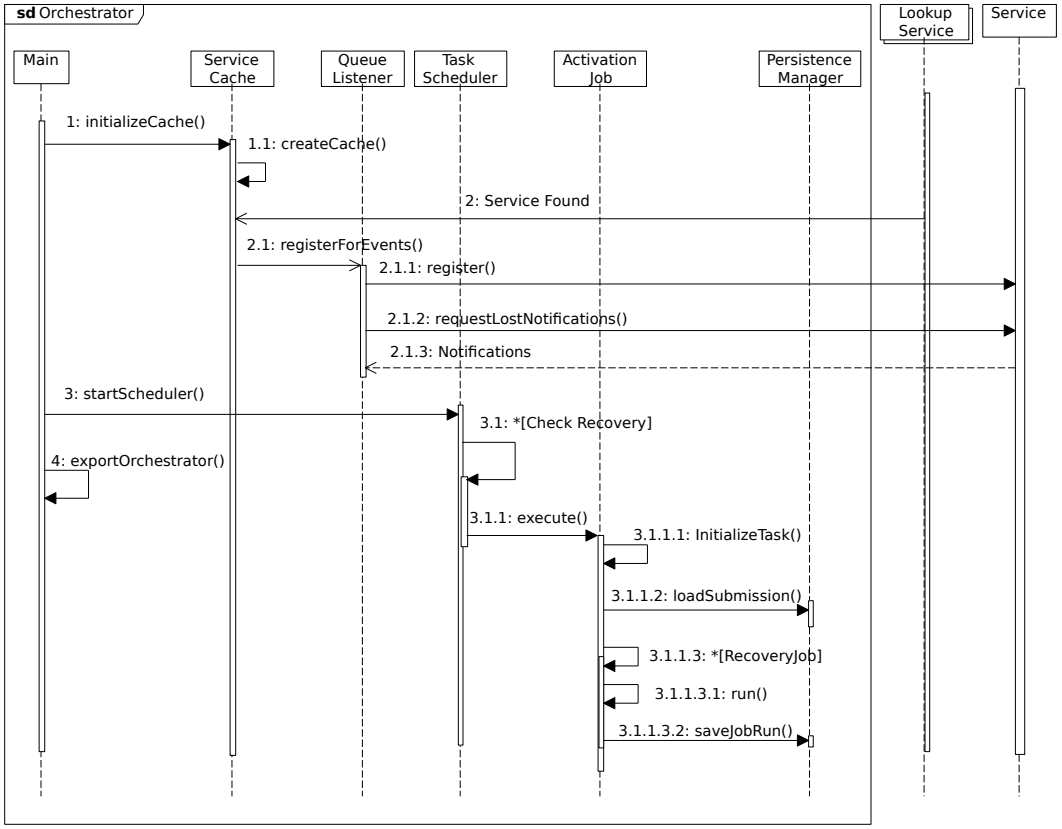


Figura 2.10: Caso d'uso: recovery dell'orchestratore.

In figura 2.10 si mostra il recovery dell'orchestratore, di seguito si descrivono le interazioni seguendone l'ordine. Al riavvio dell'orchestratore viene

eseguita la classe Main che si occupa dell'avvio e dell'esportazione del servizio di esportazione tramite RMI o Servizio River. La prima operazione effettuata è l'avvio della cache dei servizi tramite il metodo `initializeCache()` della classe Service Cache, in questo modo l'orchestratore ottiene una visione aggiornata dei servizi disponibili sulla rete.

Per ogni servizio individuato (tramite multicast e Lookup Service), l'orchestratore effettua una registrazione per ricevere gli eventi usando il metodo `registerForEvents()` del service object. Se il servizio appena scoperto risulta avere in carico dei job non terminati, l'orchestratore invia una richiesta per ricevere le eventuali notifiche perse richiamando il metodo `requestLostNotifications()` del service object.

Il successivo passo per il recovery è il riavvio del Task Scheduler che contiene l'istanza di Quartz. Dopo essere stato avviato con l'operazione `startScheduler()`, lo il Task Scheduler avvia il meccanismo di recovery di tutti gli Activation Job che erano in esecuzione prima del guasto. La logica del recovery è però implementata in gran parte negli Activation Job, tale logica permette di riprendere l'esecuzione del job dal punto in cui è stato interrotto conservando i la Job Data Map calcolata fino a quel punto.

Come ultimo passo, dopo l'avvio del Task Scheduler, l'orchestratore esporta il suo servizio chiamando la funzione `exportOrchestrator()` per rendersi nuovamente disponibile ai client.

### 2.3.4 Recovery del Queue Server

I Queue Server possono guastarsi o essere spenti quando ve ne sia bisogno, ad esempio, per ridurre le code di lavoro. Questa sezione illustra il caso d'uso del recovery di un Queue Server dopo un guasto si tipo crash, la figura 2.11 mostra il diagramma di sequenza per il caso d'uso citato.

Come per l'orchestratore, anche in questo caso il riavvio è gestito dalla classe Main che è l'entry point del progetto. Per l'ambiente in cui saranno schierati, i Queue Server possono essere spenti o spostati, per questo motivo l'esportazione del servizio tramite il metodo `exportQueueServer()`

effettuata di default il recovery di eventuali job non terminati. La prima operazione è il controllo sull'identificativo del servizio: è necessario che un Queue Server conservi l'id per poter essere riconosciuto nuovamente dal servizio di orchestrazione; al contrario, se è avviato per la prima volta, il metodo `generateID()` crea un file contenente l'id persistente.

Il passo successivo del Service Exporter è il riavvio dello Scheduler Engine: all'avvio dello scheduler vengono ricaricati dalla base di dati i class loader per i job eseguiti in precedenza attraverso il metodo `restoreClassLoader()` della classe Custom Class Loader. Avviato lo scheduler, gli eventuali job interrotti dal guasto possono riprendere; il meccanismo del recovery prevede in questa implementazione l'esecuzione dal principio del job utilizzando la Job Data Map iniziale: sebbene si assicuri che il job utilizzerà sempre dati non corrotti, è compito dello sviluppatore assicurare l'idempotenza dell'esecuzione secondo le necessità.

L'ultima operazione effettuata dal Service Exporter è richiamare il proprio metodo `exportService()` che permette l'esportazione del servizio in tutti i Lookup Service presenti sulla rete ed in quelli che si annunceranno in futuro.

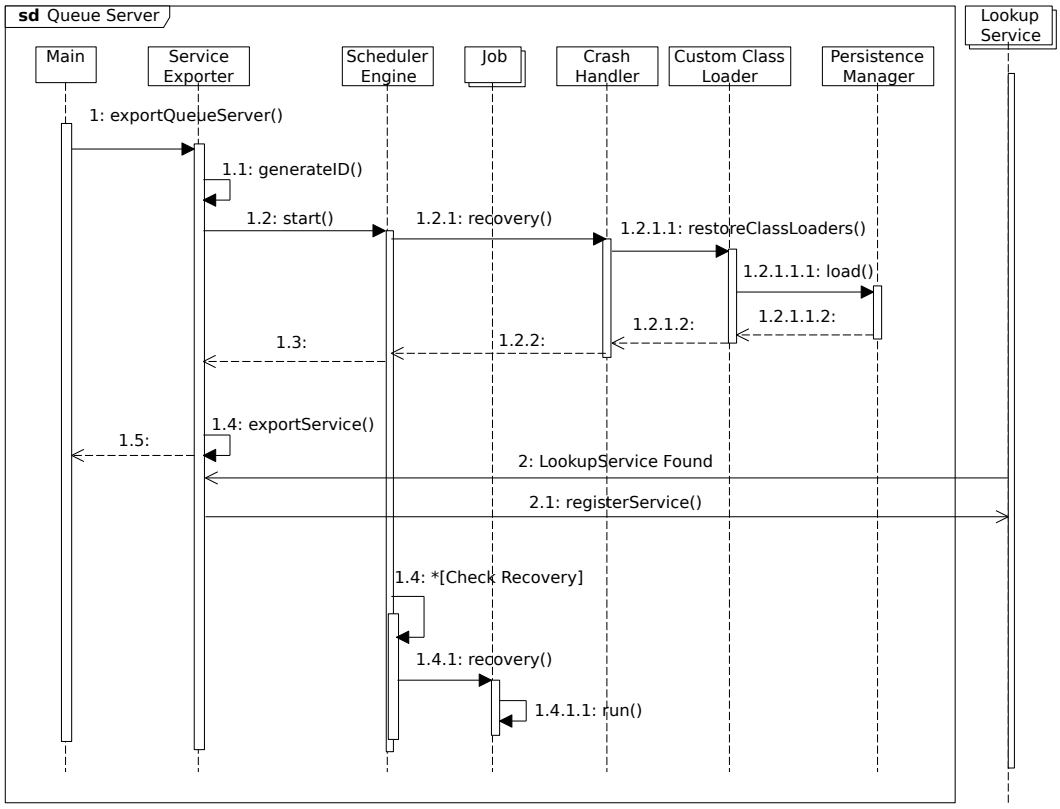


Figura 2.11: Caso d'uso: recovery di un Queue Server.

# Capitolo 3

# Implementazione

In questo capitolo si descrive la realizzazione del sistema progettato. Il sistema è stato realizzato a partire da due progetti distinti e interamente scritti Java: **il progetto orchestratore** ed il **progetto Queue Server** che realizzano le due componenti del sistema sulle specifiche definite durante la fase di progettazione.

Il resto del capitolo presenta prima gli strumenti e i framework utilizzati per l'implementazione, segue poi la descrizione della struttura dei due progetti attraverso la descrizione dei package, e si conclude con una guida alla configurazione e allo schieramento (deployment).

La scelta del linguaggio Java è stata dettata principalmente dalla necessità di avere un progetto platform-indipendent e che potesse essere usato semplicemente in ambienti virtualizzati; l'uso della JVM [1] fornisce nativamente la proprietà di indipendenza dalla piattaforma necessaria.

[1] JVM: Java Virtual Machine

## 3.1 Gli strumenti utilizzati

### L'ambiente di sviluppo Eclipse

La scrittura e la gestione dei file sorgenti è stata gestita usando Eclipse Indigo in versione 3.7. Eclipse è uno dei più completi IDE [2] disponibili gratuitamente, inoltre, si integra perfettamente con *Apache Ant*, la libreria utilizzata per compilare, distribuire ed eseguire i progetti realizzati.

### Apache River Framework

Già visto ampiamente nella sezione 1.4, si è utilizzato il framework middleware River nella versione 2.1 per costruire l'architettura orientata ai servizi del progetto. River fornisce inoltre un servizio di Web Server minimale che è stato utilizzato in ambiente di test per realizzare la codebase dei JAR.

### Quartz Enterpise Scheduler

Come già accennato precedentemente, per le funzionalità di scheduling ci si è affidati a Quartz Enterprise Scheduler (ver. 2.0.1) poiché ampiamente affermato per affidabilità e prestazioni in organizzazioni come Apache, Sun, Adobe e Cisco Systems.

### Hibernate e MySQL

Per migliorare le caratteristiche di manutenibilità, e portabilità si è adottate una soluzione ORM utilizzando la piattaforma middleware Hibernate (ver. 3.6) che al momento è il più adottato tra gli strumenti ORM per il linguaggio Java. Il database è stato realizzato utilizzando il DBMS MySQL nella versione 5.1 che è la versione presente sul cluster dei laboratori del dipartimento di informatica.

[2] IDE: Integrated Development Environment

#### Java Secure Channel

Il trasferimento dei file sul codebase server tramite protocollo SFTP è stato realizzato usando una libreria chiamata Java Secure Channel (JSch) in versione 0.1;

## 3.2 L'interazione tra servizi

I servizi sono alla base dell'architettura implementata in questo lavoro, i progetti realizzati fanno un massiccio uso di servizi, una descrizione tecnica dei protocolli usati nelle interazioni tra servizi è d'obbligo. Il concetto di servizio è stato trattato in generale nella sezione 1.4.3, in questa sezione si descrivono i servizi dal punto di vista delle loro interazioni utilizzando il framewrok implementativo River.

River utilizza tre protocolli per la gestione dei servizi: i protocolli di discovery, join e lookup che sono illustrati di seguito mediante tre diagrammi corredati di descrizione.

La coppia di protocolli *discovery-join* è utilizzata quando un nuovo servizio sulla rete vuole registrarsi e rendersi disponibile. Il discovery (in figura 3.1) è usato dal service provider per individuare i servizi di lookup presenti sulla rete; a tale scopo il protocollo usa una tecnica multicast o unicast per permettere ai servizi di lookup di identificarsi.

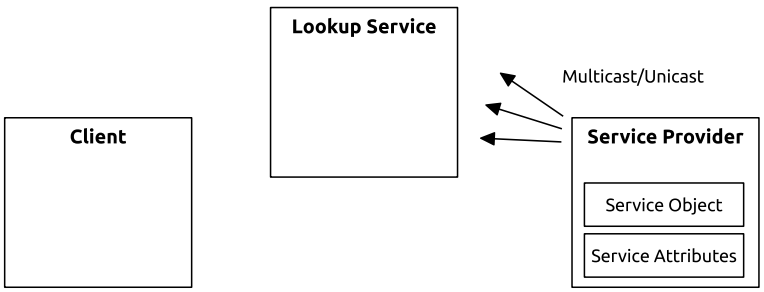


Figura 3.1: L'interazione tra servizi: il protocollo di discovery.

Identificati i servizi di lookup, il service provider utilizza il protcollo di join, in figura 3.2, per registrare il proprio servizio nei servizi di lookup che ritiene opportuni. La registrazione avviene usando un *service object* che contiene l'interfaccia Java per il servizio assieme ad altri attributi descrittivi definibili tramite i *service attributes*. É importante precisare che anche i servizi di lookup sono normali servizi, sono già disponibili nella distribuzione di River e possono essere usati per registrare altri servizi di lookup per creare una struttura gerarchica in stile DNS [3].

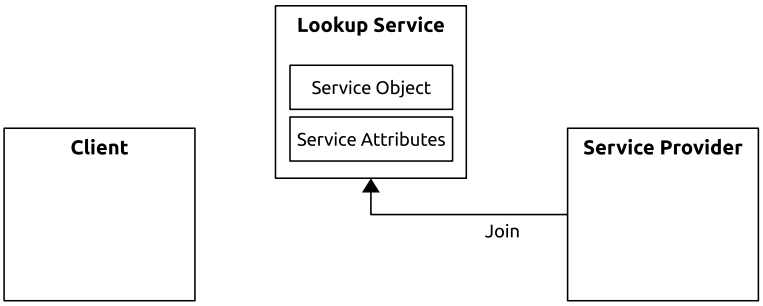


Figura 3.2: L'interazione tra servizi: il protocollo di join.

Il protocollo di lookup schematizzato in figura 3.3, è usato dal client per ricercare un servizio "per tipo", ossia tramite la sua interfaccia Java o per "attributi" tramite i service attributes. Il service object detto anche *proxy*, viene caricato nel client e permette di interagire con il servizio ospitato in un altro punto della rete. L'unica informazione posseduta dal client è che sta utilizzando un'implementazione di un'interfaccia scritta in Java, permettendo al codice che implementa tale interfaccia d'essere ovunque.

La comunicazione tra il service object ed il servizio può avvenire tramite protocolli diversi: il protocollo classicamente usato da RMI è JRMP (Java Remote Method Protocol) che è costruito direttamente sul TCP, successivamente altri protocolli sono stati usati allo stesso scopo tra cui HTTP, IIOP [4] o SSL.

[3]DNS: Domain Name System

[4]IIOP: Internet Inter-ORB Protocol, è il protocolli di trasporto usato da CORBA.

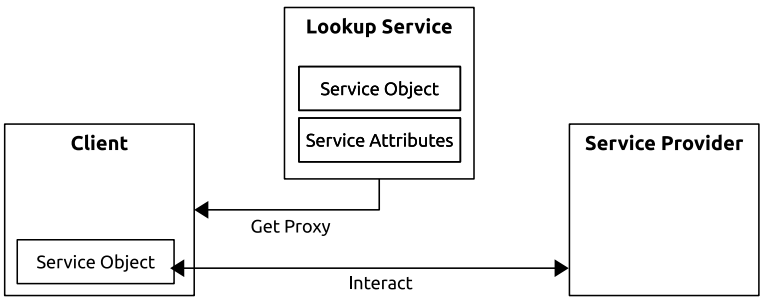


Figura 3.3: L'interazione tra servizi: il protocollo di lookup.

Nell'implementazione di questo lavoro per la comunicazione tra i servizi si è utilizzato il protocollo *Jini Extensible Remote Invocation (JERI)*, che è un'evoluzione del protocollo JRMP e include numerosi miglioramenti.

## 3.3 Le API dei progetti

Ogni progetto genera a tempo di compilazione un insieme di API [5] che sono fondamentali per l'uso del progetto stesso da parte di altre entità quali i client o altri progetti. Il possesso delle API è un requisito fondamentale per l'interazione col progetto che le ha generate. Si presenta brevemente la struttura dei progetti in termini di JAR prodotti in compilazione per comprendere quali API devono essere possedute da ogni progetto.

La compilazione del progetto orchestratore produce tre file JAR:

- *Orchestrator.jar* è il file di distribuzione che contiene il progetto;
- *Orchestrator-api.jar è l'archivio contenente le API per l'uso dell'orchestratore tramite RMI;*
- *Orchestrator-jobs-api.jar* è l'archivio che fornisce le API agli sviluppatori per la creazione di task che rispettino le specifiche dell'orchestratore.

La compilazione del progetto Queue Server produce due file JAR:

[5] API: Application Programming Interface

- *QueueServer.jar* è il file di distribuzione dell'omonimo progetto;
- *QueueServer-api.jar* è l'archivio contenente le API che permettono all'orchestratore l'uso dei servizi di tipo Queue Server.

## 3.4 Il progetto orchestratore

La realizzazione dell'orchestratore segue fedelmente la struttura dell'architettura delineata in fase di progettazione nella sezione 2.2.1. É utile ricordare le funzioni principali per cui è stato sviluppato l'orchestratore:

- fornire un *sigle point of contact* per la sottomissione di schedulazioni;
- coordinare un insieme di servizi nel sistema, in particolare le code di lavoro;
- monitorare l'esecuzione dei workflow e raccoglierne i risultati.

### 3.4.1 I package

In figura 3.4 è rappresentato il diagramma dei package in cui compaiono per chiarezza illustrativa solo le relazioni di inclusione. Si descrivono di seguito i package che differiscono dalle componenti trattate ampiamente nella definizione architetturale in sezione 2.2.1.

#### orchestrator

Il package orchestrator contiene le classi che implementano il pattern façate per le funzioni dell'orchestratore oltre ad occuparsi dell'esportazione del servizio via RMI. Il package orchestrator è di fatto il punto di contatto tra il sistema e i client che vogliono schedulare un task. Il package in questione contiene inoltre l'interfaccia remota esportata durante la creazione del servizio di orchestrazione: tale oggetto remoto è detto nel linguaggio di River *service object o proxy*.

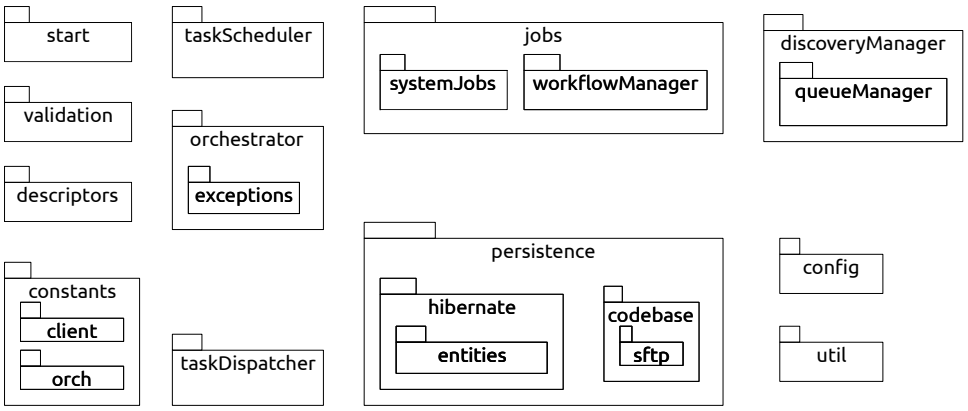


Figura 3.4: Diagramma dei package del progetto orchestratore.

### start

Il package start contiene l'entry point del progetto ed ha il compito di inizializzare tutte le componenti per permettere l'avvio dell'orchestratore. Un'importante funzione del package start è l'avvio del *Security Manager* di RMI, il gestore della sicurezza incaricato di far rispettare le politiche riguardo all'esecuzione del mobile code; il Security Manager è configurabile tramite un opportuno file di policy che sarà meglio descritto nella sezione 3.6.1.

### descriptors

Nel package descriptors sono definiti le sue strutture dati che permettono di rappresentare il task e i job come istanze di oggetti Java tali da poter essere elaborati al livello middleware del sistema. Un task è inizialmente definito in XML, un formalismo che ben si presta alla descrizione, ma risulta meno efficiente se utilizzato durante l'elaborazione del task. L'uso di oggetti rende più agevole l'attuazione della persistenza tramite la soluzione ORM utilizzata.

Le due semplici strutture che rappresentano rispettivamente il job e il task sono il *JobDescriptor* ed il *TaskDescriptor*; nei listati 3.1 e 3.2 si mostrano gli estratti del codice per la definizione dei descrittori.

Codice 3.1: Definizione del JobDescriptor in formato Java.

```
1  public class JobDescriptor implements Serializable {
2
3     private static final long serialVersionUID = 1L;
4     //Full job Class Name
5     private String jobClassName;
6     //Compensation Class Name
7     private String compensationClassName;
8     //Execution order of this job
9     private int order;
10    //Contains the full names of classes needed for jobExecution
11    private Set <String> neededJars = new TreeSet <String>();
12
13    [getter and setter methods omitted]
14 }
```

Codice 3.2: Definizione del TaskDescriptor in formato Java.

```
1  public class TaskDescriptor implements Serializable {
2
3     private static final long serialVersionUID = 1L;
4     //submission name
5     private String submissionName;
6     //list of jobs for this task
7     private List <JobDescriptor> jobsDesc = new ArrayList <JobDescriptor>();
8     //Contains the parameters needed to execute the jobs
9     private HashMap <String,String> params = new HashMap <String,String>();
10
11    [getter and setter methods omitted]
12 }
```

### validation

La correttezza e la completezza del task espresso tramite il TaskDescriptor (che include diversi JobDescriptor) è controllata dalla classe *validator* del package validation. Il package validation in particolare analizza i dati inviati nella richiesta del client per verificare:

- la presenza di tutti i campi necessari;
- la correttezza della Cron expression;

- la presenza dei JAR da utilizzare nell'esecuzione dei job;
- l'unicità dei nomi delle classi nel sistema.

### taskDispatcher

Il package taskDispatcher contiene le classi che implementano il meccanismo per la sottomissione dei task all'orchestratore. Tecnicamente è realizzato tramite un thread dedicato che rimane in attesa su una struttura dati di tipo coda bloccante. La coda bloccante, più precisamente la *BlockingQueue* del package *java.util.concurrent*, permette al thread di restare in attesa se la coda e vuota e di elaborare le richieste accodate in ordine di sottomissione fino a quando la coda non è nuovamente vuota.

Nel listato 3.3 si mostra il codice eseguito dal metodo **run()** del *taskDispatcherThread* che elabora richieste subito dopo la validazione e memorizzazione. Alla riga 7 è eseguita l'operazione di **take()** sulla coda bloccante per prelevare il prossimo identificativo della schedulazione da elaborare. Nelle righe dalla 15 alla 18 sono recuperati dal DB i dati relativi alla schedulazione per poter inizializzare l'Activation Job che la porterà avanti.

Nell'ultima parte del codice a partire dalla riga 20, vengono creati l'Activation Job ed il Cron trigger che sono aggiunti poi allo scheduler per l'effettiva esecuzione.

Codice 3.3: Codice del taskDispatcherThread: sottomissione.

```
1  public void dispatchTasks() {
2
3     while (true) {
4        Uuid schedulazioneUuid = null;
5        try {
6           _log.debug("Task dispatcher thread ready to dispatch..");
7           schedulazioneUuid = submissionsQueue.take();
8        } catch (InterruptedException ex) {
9           _log.warn("Task dispatcher interrupted.");
10       }
11
12       //Process the submission by ID..
13       String submisId = schedulazioneUuid.toString();
14
```

```
15        //Get data from DB
16        Schedulazione entry = SchedDAO.getInstance().find(submisId);
17        String cronTrigger = entry.getCronTrigger();
18        String submisName = entry.getNome();
19
20        //Create a new ActivationJob
21        JobDetail actJob = buildNewActivationJob(submisId, submisName);
22
23        //Define the Cron Trigger
24        CronTrigger trigger = null;
25        try {
26           trigger = newTrigger()
27              .withIdentity(submisId,
                    JobAndTriggerConst.ACTIVATION_TRIGGER_KEY_GROUP)
28              .withSchedule(cronSchedule(cronTrigger))
29              .withDescription("TriggerForSubmission: " + submisName +" Id: "
                    + submisId)
30              .build();
31        } catch (ParseException e) {
32           //will be never thrown thanks to the submission validation
33           _log.error("Cron trigger parse exception.");
34        }
35
36        //Dispatch to LocalScheduler singleton
37        LocalScheduler.getInstance().scheduleJob(actJob, trigger);
38        _log.info("Submission accepted: " + submisName + " ["+ submisId +"]");
39     }
40  }
```

Nel listato 3.4 è riportato il codice della funzione *buildNewActivationJob* nella quale viene inizializzato e creato l'Activation Job.

Codice 3.4: Codice del taskDispatcherThread: creazione ActivationJob.

```
1  private JobDetail buildNewActivationJob(String submId, String submName) {
2
3     //Create a new ActivationJob
4      JobKey key = new JobKey(submId,
           JobAndTriggerConst.ACTIVATION_JOB_KEY_GROUP);
5      JobDetail actJob = newJob(ActivationJob.class)
6        .withIdentity(key)
7          .usingJobData(ActivationJob.SUBMISSION_ID, submId)
8          .usingJobData(ActivationJob.SUBMISSION_NAME, submName)
9          .usingJobData(ActivationJob.JOB_GROUP,
               JobAndTriggerConst.ACTIVATION_JOB_KEY_GROUP)
10         .usingJobData(ActivationJob.EXECUTION_COUNT, 0)
```

```
11          .usingJobData(ActivationJob.EXECUTION_STATE,
                ExecutionStateConst.INCEPTION_STATE)
12          .usingJobData(ActivationJob.RUN_SCHEDULAZIONE_ID,
                JDMConst.JDM_NEVER_RUN_BEFORE_RUN_ID)
13          .withDescription("ActSubmission: " + submName +" Id: " + submId)
14          .requestRecovery(true)
15          .storeDurably(true)
16          .build();
17
18      return actJob;
19  }
```

### constants

Infine il package constants contiene, come il nome suggerisce, le costanti usate per indicare gli stati dei job e dei task. Il sotto package *client* contiene invece solo le costanti utili per la sottomissione dei task e destinate ad essere esportate come API dell'orchestratore.

### linee di codice

Nella tabella 3.1 si mostra il numero di linee di codice prodotte per il progetto orchestratore con aggregazione per package.

| **Package** | **Numero di classi** | **Righe di codice** |
|---|---|---|
| jobs.systemJobs | 1 | 695 |
| jobs.workflowManager | 2 | 136 |
| discoveryManager.queueManager | 1 | 316 |
| discoveryManager | 3 | 310 |
| persistence.hibernate.entities | 10 | 1285 |
| persistence.hibernate.dao | 5 | 482 |
| persistence.hibernate | 2 | 135 |
| persistence.codebase.sftp | 3 | 283 |
| taskScheduler | 1 | 131 |
| taskDispatcher | 2 | 138 |
| descriptors | 2 | 159 |
| config | 1 | 69 |
| start | 1 | 56 |
| orchestrator | 2 | 149 |
| constants | 5 | 62 |
| util | 1 | 22 |
| validation | 1 | 105 |
| Totale | 43 | 4533 |

Tabella 3.1: Il progetto orchestratore: contenuto dei package.

## 3.5 Il progetto Queue Server

Il progetto Queue Server realizza quella parte del sistema che esegue fisicamente i job e il cui servizio è oggetto di orchestrazione.

### 3.5.1 I package

In figura 3.5 è mostrato il diagramma dei package in cui sono rappresentate solo le relazioni di inclusione. La divisione in package fornisce un'idea del raggruppamento per funzione delle classi sviluppate secondo quanto definito nella fase di progettazione. Si descrivono di seguito i package che non sono stati già trattati nella sezione 2.2.1 come componenti.

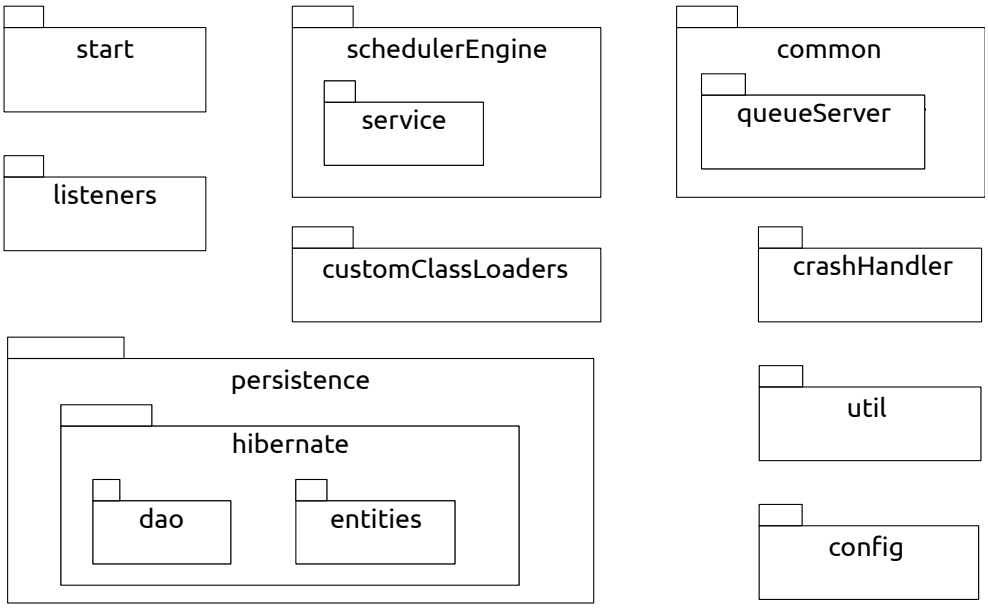


Figura 3.5: Diagramma dei package del progetto Queue Server.

#### start

Il package start fornisce l'entry point per il progetto, le due funzioni fondamentali fornite dal package sono la creazione del Security Manager di

RMI e l'esportazione del servizio Queue Server attraverso l'invocazione delle classi del package service.

#### service

La pubblicazione del servizio di coda è iniziata e gestita dal package service, a tale scopo sono state implementate due interfacce fornite da River:

- *net.jini.discovery.DiscoveryListener* che permette di ricevere una notifica qualora un servizio di lookup sia aggiunto o rimosso dalla rete; grazie alle notifiche, il Queue Server può pubblicare il suo servizio e renderlo disponibile disponibile;
- *net.jini.lease.LeaseManager* che facilita il rinnovo del leasing per il servizio esportato.

Le funzioni svolte dal package service sono svolte durante tutta la durata dell'esecuzione del servizio esportato e possono essere schematizzate come segue:

- generazione dell'identificavo persistente per il servizio;
- costruzione del servizio in base al file di configurazione che specifica il nome del servizio e la coda di lavoro a cui aggregarsi;
- pubblicazione del servizio presso tutti i servizi di lookup della rete;
- rinnovo periodico del leasing per tenere vivo il servizio;
- ricezione delle notifiche per l'aggiunta di servizi di lookup.

#### common

Nel package common sono contenute le costanti che il Queue Server usa come contenuto degli eventi distribuiti per notificare all'orchestratore lo stato di terminazione dei job; chiaramente, una notifica contiene sia la costante indicante il tipo di evento che la chiave del job che ha generato la notifica. Ci sono tre tipologia di eventi a cui corrispondono le seguenti costanti:

- **JOB_TO_BE_EXECUTED_EVENT** indica che il job sta per essere eseguito;

- **JOB_EXECUTED_EVENT** indica il completamento dell'esecuzione del job;

- **JOB_VETOED_EVENT** indica che l'esecuzione del job è stata impedita.

Il package common contiene anche l'interfaccia remota esportata come proxy durante la creazione del servizio. Il contenuto del package common è esportato nelle API del Queue Server per poi essere distribuito all'orchestratore o a chi voglia utilizzare il servizio di coda.

Un'importante classe del package è il *JobRunDetail* che è l'oggetto serializzabile che incapsula le informazioni raccolte dalla run di un job. Il JobRunDetail è incluso nelle notifiche che i Queue Server inviano all'orchestratore; il codice del listato 3.5 mostra la definizione della classe.

Codice 3.5: Definizione del JobRunDetail in Java.

```
1  public class JobRunDetail implements Serializable {
2
3      static final long serialVersionUID = 1L;
4
5      //Job key for the run job
6      private JobKey jobKey = null;
7      //Data map containing job partial result
8      private JobDataMap dataMap = null;
9      private Date fireTime = null;
10     private long runTime = 0;
11     //flag fro exceptions in job run
12     private boolean isException = false;
13     private String exceptionMessage = null;
14
15     [getter and setter methods omitted]
16 }
```

### listeners

Il package listeners contiene la classe *JobsListener* che implementa le funzioni richiamate quando il Queue Server inizia, termina o vieta l'esecuzione di un job. Grazie a tali funzioni, l'orchestratore che si è precedentemente registrato, viene notificato degli eventi relativi ai job. Il codice nel listato 3.6 mostra la creazione del JobRunDetail e l'invio della notifica remota per l'evento di completamento di un job.

Codice 3.6: JobListener: invio della notifica per un job completato

```
1   public void jobWasExecuted(JobExecutionContext jec,
        JobExecutionException jee) {
2
3       //Check execution Exceptions
4     boolean exceptions = false;
5     String exceptionMsg = "";
6       if (jee != null) {
7        _log.warn("Job Executed with exceptions: " + jee.getMessage());
8        exceptions = true;
9        exceptionMsg = jee.getMessage();
10      }
11
12      //Create a JobRunDetail using job context and exceptions
13      JobRunDetail runDetails = new JobRunDetail(jec, exceptions,
            exceptionMsg);
14      JobKey jobKey = jec.getJobDetail().getKey();
15      _log.info("Job Executed: " + jobKey);
16
17      try {
18          //Fire notify on the remote listener (the orchestrator)
19          remoteScheduler.fireNotify(jobKey, Constants.JOB_EXECUTED_EVENT,
20                  new MarshalledObject(runDetails));
21      } catch (IOException ex) {
22          _log.error("Failed to fire Notify for job executed.", ex);
23      }
24  }
```

L'esecuzione dei job richiede il caricamento delle classi contenute nei JAR presenti nel codebase server; sebbene il framewrok River ed RMI forniscano questa funzionalità in maniera quasi trasparente, è necessario interfacciare tale meccanismo con lo scheduler Quartz attraverso l'implementazione del-

l'interfaccia *org.quartz.spi.ClassLoadHelper*. Il package **customClassLoaders** del progetto realizzato permette, tramite la classe *URLClassLoaderHelper*, il caricamento delle classi remote associando ogni job ad un preciso URLClassloader (del package java.net).

#### linee di codice

Nella tabella 3.2 si mostra il numero di linee di codice del progetto Queue Server con aggregazione per package.

| Package | Numero di classi | Righe di codice |
|---|---|---|
| schedulerEngine.service | 1 | 223 |
| schedulerEngine | 1 | 338 |
| customClassLoaders | 2 | 312 |
| listeners | 1 | 84 |
| crashHanler | 1 | 77 |
| persistence.hibernate.entities | 3 | 331 |
| persistence.hibernate.dao | 3 | 303 |
| persistence.hibernate | 1 | 28 |
| common.queueServer | 4 | 126 |
| config | 1 | 45 |
| crashHandler | 1 | 27 |
| start | 1 | 36 |
| util | 1 | 53 |
| Totale | 21 | 1983 |

Tabella 3.2: Il progetto Queue Server: contenuto dei package.

## 3.6 Guida alla configurazione

### 3.6.1 I file di configurazione

Benché il sistema sia progettato per ridurre al minimo l'attività di configurazione, sono presenti alcune componenti che necessitano di una configurazione preventiva. Per chiarezza, in questa sezione si fa riferimento ai file di

configurazione senza riportarne il contenuto; tutti i file citati sono comunque presenti in versione completa nelle appendici A, B e C.

### Configurazione e deployment dei servizi River

Nello sviluppo dei progetti si è utilizzata l'implementazione di un servizio di lookup chiamato *Reggie* messo a disposizione da River che ben si adatta alle esigenze di questo lavoro; Reggie necessita di un servizio addizionale per distribuire le librerie della piattaforma chiamato *httpd* che implementa un web server HTTP minimale.

La configurazione dei due servizi utilizzati avviene tramite il file *transient-jeri-services.config*, visibile in appendice A, nel quale sono definiti i "service descriptor" per i due servizi come spiegato di seguito.

Per il servizo httpd si devono impostare:

- il JAR che implementa il servizio;
- la porta su cui rendere disponibile il servizio;
- il percorso del file di policy per le politiche di sicurezza (*jsk-all.policy*).

Per il servizio Reggie si impostano invece:

- il JAR che implementa il servizio;
- gli URL sul quale httpd rende disponibili i JAR delle librerie di River
- il percorso del file di configurazione per Reggie (*transient-reggie.config*);
- il percorso del file di policy per le politiche di sicurezza (*jsk-all.policy*).

I servizi sono avviati utilizzando un JAR di utilità di River detto *Service Starter* tramite uno script Bash come nel codice che segue:

```
#! /bin/sh
java -Djava.security.policy=jsk-all.policy \
     -jar $RIVER_HOME/lib/start.jar \
     transient-jeri-services.config
```

Dove la variabile di sistema $RIVER_HOME rappresenta il percorso locale della distribuzione River.

Anche il Codebase Server che ospita i JAR dei job si può considerare parte dell'infrastruttura; qualsiasi web server HTTP può ospitare la codebase. Nell'implementazione realizzata si è utilizzato lo stesso web server alla base dal servizio httpd. Il Codebase Server è avviato tramite uno script Bash con il comando seguente:

```
#! /bin/sh
java -jar $RIVER_HOME/lib/classserver.jar \
     -port 60080 -dir $CODEBASE_SERVER_HOME/jars/ -verbose
```

Dove si la variabile di sistema $CODEBASE_SERVER_HOME indica il percorso della directory da pubblicare tramite web server.

Lo schieramento sulla rete dei servizi River è immediato: si lancia un'istanza del Service Starter su uno o più nodi della rete dopo aver personalizzato il file di configurazione. Per il funzionamento del sistema progettato è necessario che sia attivo sulla rete almeno un servizio di lookup. Un ipotetico crash di tutti i servizi di lookup non compromette le operazioni in corso ma interrompe temporaneamente l'esecuzione di nuovi job fino alla disponibilità di un servizio di lookup.

### Configurazione e deployment dell'orchestratore

L'orchestratore è configurato mediante i file di proprietà (disponibili in appendice C)che vengono descritti di seguito in questa sezione.

La configurazione "principale" viene eseguita tramite il file *orchestrator.properties* modificando i parametri in esso contenuti. Nel file si devono impostare quattro gruppi di parametri:

- il gruppo di servizi del quale l'orchestratore farà parte;
- l'accesso HTTP al codebase server;
- l'accesso SFTP alla directory del codebase server;

- l'indirizzo e la porta su cui creare l'RMI registry.

Il file *hibernate.cfg.xml* definisce il mapping ORM tra le classi Java e le entità della base di dati, nonché i parametri di accesso tramite JDBC [6].

L'orchestratore incorpora Quartz, il quale va configurato usando il file quartz.properties; le modifiche strettamente necessarie riguardano solamente l'accesso tramite JDBC alla base di dati definita da Qaurtz 'jobStore".

Un'importante impostazione legata al jobStore è la soglia di *misfire*, cioè il ritardo che viene tollerato nell'esecuzione di un trigger.

L'impostazione *org.quartz.jobStore.misfireThreshold* controlla il tempo di misfire; un valore di 60000 millisecondi, ad esempio, permette ad un trigger di essere eseguito fino ad un minuto dopo rispetto al momento di esecuzione stabilito; superato il minuto di ritardo l'esecuzione verrà rimandata al successivo verificarsi delle condizioni del trigger.

Il sistema progettato prevede il deployment di una sola istanza dell'orchestratore; un'operazione preliminare allo schieramento, da eseguire una sola volta, è la creazione del database attraverso lo script SQL incluso nella distribuzione con il nome *orchestrator-required.sql*. Lo script genera le tabelle per l'orchestratore includendo anche quelle necessarie per l'esecuzione di Quartz.

Creata la struttura del DB e configurati i file sopra descritti, si può fare partire l'orchestratore tramite script Ant *build.xml* incluso nella distribuzione richiamando il target "run".

### Configurazione e deployment dei Queue Server

La configurazione delle istanze dei Queue Server è simile alla configurazione dell'orchestratore; i file descritti di seguito possono essere visionati consultando l'appendice C.

Il file *queueServer.properties* contiene due parametri fondamentali: il gruppo a cui il servizio si unirà e, più importante, il nome della coda di lavoro a cui il Queue Server sarà aggregato.

[6] JDBC: Java DataBase Connectivity

La configurazione della base di dati è configurata tramite il file *hibernate.cfg.xml*, le uniche modifiche necessarie riguardano solamente l'accesso al DB tramite JDBC.

Ogni istanza Queue Server incorpora una istanza di Quartz; la configurazione delle preferenze e del jobStore di Quartz è contenuta nel file *quartz.properties*. É essenziale che ogni istanza di Quartz (dunque di Queue Server) abbia a disposizione delle tabelle riservate e che quindi altre istanze non modifichino tali tabelle.

Il sistema realizzato prevede la presenza di numerose istanze di Queue Server nella rete, ogni istanza dovrà avere una serie di tabelle riservate che devono essere create prima dello schieramento dell'istanza eseguendo due script:

- **queueServer-required.sql** genera le tabelle che non riguardano Quartz e possono essere anche condivise tra le istanze;
- **queueServer-NomeIstanza.sql** genera le tabelle per la singola istanza "NomeIstanza".

Configurati i file descritti e create le tabelle del DB, le istanze possono essere lanciate tramite il target "run" dello script Ant *build.xml* e sono pronte per partecipare all'orchestrazione dei servizi.

# Capitolo 4

# Test e validazione

Questo capitolo è incentrato sulla validazione del prototipo realizzato, con particolare attenzione alla tolleranza ai guasti. Con la validazione si intende verificare che tutti i requisiti delineati siano stati soddisfatti e che il sistema realizzato si comporti come voluto: nello specifico, che il sistema continui a funzionare in presenza di determinati guasti mantenendo un comportamento "fail safe" e rispettando le proprietà di *safety* e *liveness*.

La validazione è stata eseguita empiricamente tramite una serie di **test** sia sulle componenti architetturali che sulla struttura dei task: sulle componenti sono stati provocati dei guasti per simulare i partial failure, nei task invece sono stati inseriti appositi job per generare eccezioni e per valutare le azioni di recovery.

Nel corso del capitolo saranno introdotti i pattern che permettono al sistema di reagire ai guasti e alle eccezioni derivanti dall'esecuzione dei task; saranno poi presentate le categorie di guasti e si descriveranno gli effetti che hanno sull'orchestrazione dei servizi. Infine, si riportano i risultati dei test per ogni componente.

É necessario fare una precisazione sulla terminologia adottata in questo capitolo:

Si indicherà con *failure* o guasto l'evento che interrompe temporaneamente o permanentemente tutte le funzionalità di una componente: un esempio

di failure è il crash o lo spegnimento di una macchina.

Si indicherà con *fault* una situazione anomala che, se non correttamente gestita, può causare un guasto: un esempio di fault può essere un'eccezione imprevista lanciata da un job in esecuzione.

## 4.1 Azioni di recovery

In letteratura, come in [2] e [1], sono stati definiti dei pattern specificamente pensati per la gestione dei guasti e delle eccezioni nei sistemi che trattano workflow di job. Partendo dai pattern, si sono definite le azioni correttive o di recovery che il sistema realizzato utilizza.

In presenza di un fault il sistema attua una precisa azione di recovery, lo stato del sistema al momento del fault influenza la scelta dell'azione da eseguire. Sono descritte di seguito le azioni di recovery.

- **Retry:** ritenta l'operazione che ha provocato il fault. Questa azione è utilizzata quando la causa del fault è temporanea, ad esempio l'assenza di una coda di lavoro al momento della schedulazione di un job.

- **Redirect:** redirigi l'operazione su un altro esecutore. Un esempi di redirect è la scelta di un Queue Server diverso per l'esecuzione di un job.

- **Compensation:** esegui un'azione di compensazione per un'operazione fallita. Questa operazione si applica esclusivamente in caso di fault di un job.

- **Polling:** controlla lo stato d'esecuzione dell'operazione per ottenerne lo stato. Questa operazione è eseguita dall'orchestrare per testare il completamento di un job quando si presume la perdita di una notifica.

- **Misfire:** salta l'esecuzione di un'operazione se il ritardo accumulato è maggiore di una soglia prestabilita. La soglia di misfire permette di

inibire l'esecuzione dei trigger per i job che altrimenti sarebbero eseguiti in ritardo.

- **Pause o wait:** completa le operazioni in corso e sospendi l'esecuzione di nuove operazioni fino a quando non si verifica un evento che permetta di continuare. Quest'azione è eseguita quando una componente essenziale del sistema non è disponibile, ad esempio, l'infrastruttura River.

- **Force fail:** se nessuna azione di recovery è possibile, forza la terminazione dell'operazione e dichiarala fallita. Quando non è possibile completare né il job né la compensazione, si dichiara fallita l'esecuzione del job e dunque anche del task a cui appartiene.

## 4.2 Macro categorie di guasti

Un sistema fault tolerant deve in teoria attuare azioni correttive e continuare a funzionare correttamente in presenza di qualunque guasto; nella realtà, come in questo lavoro, si analizza e si considera solo un sottoinsieme dei guasti possibili.

Il sistema prototipo è progettato per gestire i guasti di tipo crash ed i guasti di omissione relativi al workflow; la gestione dei guasti bizantini non è stata affrontata in questa trattazione.

Possiamo classificare i guasti in due categorie: quelli architetturali e quelli relativi al workflow. Le due categorie non sono però disconnesse, un guasto architetturale potrebbe provocare un fault a livello di workflow. Di seguito si descrivono i tipi di guasti secondo la classificazione appena definita.

### 4.2.1 Failure architetturali

L'architettura del prototipo è composta da diverse componenti, ognuna delle quali è soggetta a possibili guasti. Dal punto di vista architetturale, possiamo classificare i failure come segue.

- **Component failure:** questo tipo di failure è causato dal malfunzionamento di una componente del sistema. Nel sistema implementato possiamo avere 4 tipi di failure elencati di seguito.
  - Failure dell'orchestratore;
  - failure di un Queue Server;
  - failure della base di dati;
  - failure dell'infrastruttura (servizi River e codebase server).

  In caso di failure ad una componente del sistema, si ipotizza che la macchina su cui essa è in esecuzione sia riavviata e che il servizio prima o poi riprenda il suo regolare funzionamento, in alternativa il servizio dovrà essere schierato su un'altra macchina. Benché questa ipotesi sia ragionevole, in alcuni contesti potrebbe essere limitativa; si è comunque lasciata la trattazione di questo aspetto per un eventuale sviluppo futuro.

- **Resource unavailability:** indica la mancanza di una risorsa, come ad esempio la coda di lavoro. Il sistema usa le code di lavoro per eseguire i task; il malfunzionamento di tutti i Queue Server che costituiscono una coda produce una situazione di resource unavailability.

### 4.2.2 Failure relativi al workflow

Cambiando ottica, i guasti possono essere classificati a livello di workflow. Sono state individuate 4 categorie di failure alcune delle quali hanno come causa un gusto architetturale mentre altre hanno origine dalla struttura del job. In questa sezione si fa riferimento alle *azioni di recovery* che verranno esposte dettagliatamente nella sezione 4.1.

- **Job/Task Submission failure:** la sottomissione del job o del task non è andata a buon fine, il job o il task non è stato né iniziato né eseguito. La causa di questo guasto può essere il failure dell'orchestratore

o di un Queue Server oppure la non disponibilità di una coda di lavoro; deve essere attuata una azione di recovery che solitamente è il *retry* o *redirect*.

- **Data staging failure:** i dati (JAR, DB) necessari all'esecuzione di un job non possono essere recuperati. La causa è il guasto del codebase server o del DB. A seconda dei casi, il job potrebbe non essere eseguito o fallire nel caso pessimo; l'azione di recovery può essere un retry o *force fail*.

- **Job execution failure:** il job ha generato un'eccezione non gestibile durante la sua esecuzione, questo guasto dipende principalmente dalle operazioni effettuate dal job e da com'è stato progettato; l'azione di recovery è la compensazione.

- **Job notification failure:** la comunicazione tra i Queue Server avviene tramite le notifiche, questo tipo di guasto si presenta se una notifica fallisce o viene perduta dall'orchestratore. In presenza di questo guasto si esegue il *polling*.

## 4.3 Test

Vediamo in questa sezione i test effettuati prima per validare l'architettura e poi per verificare il funzionamento a livello di workflow di job.

Il sistema è stato schierato sul cluster dei laboratori del dipartimento di informatica, le macchine utilizzate sono tutte dotate di sistema operativo Linux e Virtual Machine Java di Sun.

É opportuno ricordare che il compito principale del sistema è eseguire task che richiedono spesso un tempo di elaborazione molto lungo, data questa caratteristica, il tempo necessario per orchestrare i servizi è sicuramente trascurabile in un contesto reale. In base alla considerazione precedente, non si riporteranno le configurazioni dettagliate delle macchine e si trascurerà la dimensione temporale delle operazioni. I test effettuati non vogliono essere un benchmark delle prestazioni del software ma piuttosto una prova empirica del funzionamento del sistema sotto determinate condizioni di errore.

### 4.3.1 Test sull'architettura del sistema

#### Test Queue Server

Per il Queue Server sono stati individuati quattro scenari da testare:

- il guasto crash di un servizio inattivo, ossia, che non sta eseguendo job al momento del crash;
- lo scheduling di un job su un servizio guasto prima che il meccanismo di leasing provveda a rimuoverlo dalla cache;
- il crash di un servizio attivo nell'esecuzione di uno o più job.
- lo spostamento di un servizio inattivo.

Tabella 4.1: Crash di un Queue Server inattivo.

| | |
|---|---|
| **Descrizione** | Si simula il crash di un Queue Server interrompendo il processo, si fa poi ripartire il Queue Server prima dello scadere del tempo di leasing; si ripete il processo facendo scadere il leasing. |
| **Contesto** | Nessun job in esecuzione sul Queue Server oggetto del test. |
| **Effetti** | Il crash e la riattivazione del servizio prima dello scadere del leasing non producono nessun effetto poiché la cache dei servizi non si accorge della rimozione del servizio; se si lascia scadere il leasing, il servizio viene rimosso dalla cache. Il Queue Server conserva il suo identificativo persistente anche in caso di crash. Il crash genera una finestra di tempo in cui il servizio risulta disponibile pur non essendolo, nel caso in cui il servizio venga richiesto in tale finestra viene lanciata un'eccezione; questa eventualità è trattata nei successivi test. |
| **Recovery** | Il meccanismo di leasing fornisce già una soluzione al problema, nello scenario in cui si tenti di schedulare un job sul servizio non disponibile viene lanciata un'eccezione che è opportunamente gestita. |

Tabella 4.2: Schedulazione di un job su un Queue Server non disponibile.

| | |
|---|---|
| **Descrizione** | Si schedula un job sul Queue Server subito dopo il crash e del prima dello scadere del leasing. In questo test il servizio è ancora disponibile per la cache. |
| **Contesto** | Nessun job in esecuzione sul Queue Server oggetto del test. |
| **Effetti** | L'orchestratore non percepisce il crash del servizio poiché il leasing non è scaduto, tenta la schedulazione del job sul Queue Server non disponibile e riceve un'eccezione che lo informa dell'impossibilità di comunicare con il servizio. |
| **Recovery** | Al tentativo fallito di schedulare il job si applica il *redirect* del job su un altro Queue Server della stessa coda di lavoro. Se la coda di lavoro non dispone di ulteriori server allora l'orchestratore applicherà il *retry* ritentando l'esecuzione al successivo scattare del trigger. |

Tabella 4.3: Crash di un Queue Server attivo.

| | |
|---|---|
| **Descrizione** | Si simula il crash di un Queue Server che sta eseguendo uno o più job interrompendo il processo, si fa ripartire il Queue Server prima dello scadere del tempo di leasing. |
| **Contesto** | Uno o più job sono in esecuzione sul Queue Server oggetto del test. |
| **Effetti** | L'orchestratore non percepisce il crash del servizio poiché il leasing non è scaduto. Il riavvio del Queue Server fa partire automaticamente il recovery dei job interrotti rieseguendoli con la Job Data Map di partenza. |
| **Recovery** | L'azione di recovery intrapresa dal Queue Server è il *retry* del job. Il recovery da la sicurezza che se un job viene interrotto durante l'esecuzione esso può ricominciare partendo dai dati iniziali e scartando i dati parzialmente elaborati; questa caratteristica può essere utilizzata per implementare una semantica "at most once". |

Tabella 4.4: Spostamento di un Queue Server inattivo.

| | |
|---|---|
| **Descrizione** | Si sposta di un Queue Server che non sta eseguendo job, si fa ripartire il Queue Server dopo lo scadere del tempo di leasing. |
| **Contesto** | Nessun job è in esecuzione sul Queue Server di test, il Queue Server non è momentaneamente utilizzato per l'orchestrazione. |
| **Effetti** | L'orchestratore percepisce la rimozione del servizio poiché il leasing è scaduto. Il riavvio del Queue Server su un'altra macchina non richiede il recovery perché nessun job era in esecuzione. |
| **Recovery** | Il Queue Server utilizza il suo identificativo persistente per registrarsi nuovamente. |

### Test Orchestratore

L'orchestratore è fondamentale per il funzionamento del sistema data la sua unicità; è possibile schierare più orchestratori, ma essi devono orchestrare servizi appartenenti a **gruppi disgiunti**. Per l'orchestratore è necessario verificare il comportamento in tre scenari:

- riprendere correttamente i task dal job interrotto in caso di guasto;
- gestire il misfire dei job in caso di ritradi;
- recuperare le notifiche perse dai Queue Server.

Tabella 4.5: Crash dell'orchestratore: test misfire.

| | |
|---|---|
| **Descrizione** | Si simula il crash dell'orchestratore terminandone il processo, si riavvia il processo dopo un tempo variabile. Si vuole verificare se il meccanismo di misfire inibisce correttamente l'esecuzione dei job che hanno accumulato ritardo per via del guasto all'orchestratore. |
| **Contesto** | L'orchestratore è impegnato nell'orchestrazione di due task di test, di conseguenza uno o più Queue Server presenti sono impegnati nell'esecuzione dei job assegnati. In questa simulazione nessun job termina la sua esecuzione nella finestra di tempo in cui l'orchestratore è guasto. |
| **Effetti** | Al suo riavvio l'orchestratore provvede a registrarsi nuovamente presso tutti i servizi di coda che rileva e richiede le eventuali notifiche perdute; il test corrente prevede che nessun job abbia completato l'esecuzione nel tempo di guasto, quindi nessuna notifica è stata perduta. Al riavvio dell'orchestratore, vengono ripresi i task che erano già in esecuzione ripartendo dallo specifico job interrotto ed avviandone il recovery. I task che dovevano essere eseguiti durante il tempo del guasto per i quali è stata superata la soglia di misfire non vengono avviati, gli altri job vengono avviati con il ritardo ritenuto accettabile poiché sotto la soglia. |
| **Recovery** | L'azione di recovery attuata per riprendere i task in esecuzione è il *retry*; per la richiesta delle notifiche perse viene invece usato il *polling*. |

Tabella 4.6: Crash dell'orchestratore: test notifiche perse.

| | |
|---|---|
| **Descrizione** | Si simula il crash dell'orchestratore terminandone il processo, si riavvia il processo dopo un tempo sufficiente al completamento dei job in esecuzione sulle code di lavoro. Si vuole verificare se le notifiche di completamento non inviate per via del guasto siano effettivamente ricevute al riavvio dell'orchestratore. |
| **Contesto** | L'orchestratore è impegnato nell'orchestrazione di due task di test, di conseguenza uno o più Queue Server presenti sono impegnati nell'esecuzione dei job assegnati. Le code di lavoro terminano l'esecuzione dei job, tentano l'invio della notifica remota all'orchestratore che non è disponibile. |
| **Effetti** | Al suo riavvio l'orchestratore provvede a registrarsi nuovamente presso tutti i servizi di coda e richiede le eventuali notifiche perdute; i servizi che hanno terminato i job assegnati inviano la notifica di completamento o di errore. I servizi che non hanno ancora terminato il loro job rispondono con una notifica concordata che indica che il job è ancora in esecuzione; i servizi che non rispondono (che potrebbero essersi guastati a loro volta) sono "marcati" non funzionanti e al loro ripristino saranno richieste nuovamente le notifiche. |
| **Recovery** | L'azione di recovery attuata per la richiesta delle notifiche perse è il *polling*; se il polling fallisce per via di un gusto al servizio di coda che ha in carico il job, si attende il ripristino per poi applicare il *retry*. |

### Test Infrastruttura

In questa sezione si verifica il comportamento del sistema qualora si presentino guasti all'infrastruttura River e al codebase server.

Tabella 4.7: Crash dei servizi di lookup.

| | |
|---|---|
| **Descrizione** | Si verifica il comportamento del sistema al crash e all'aggiunta dei servizi di lookup; successivamente si simula il guasto di tutti i servizi di lookup. |
| **Contesto** | Il sistema è impegnato nell'esecuzione di due task di test. |
| **Effetti** | La rimozione di uno o più servizi di lookup è rilevata dall'orchestratore e dai Queue Server. Fino a quando sulla rete rimane attivo almeno un servizio di lookup l'orchestrazione continua ininterrotta; se tutti i servizi di lookup si guastano il sistema è impossibilitato a continuare l'orchestrazione sebbene possa terminare le operazioni già avviate. All'annuncio di un nuovo servizio di lookup sulla rete, l'orchestratore aggiorna la cache dei servizi disponibili poiché i Queue Server registrano il loro servizio di coda e l'orchestrazione può continuare. |
| **Recovery** | In caso di crash di tutti i servizi di lookup il sistema deve interrompere l'orchestrazione dei servizi poiché la cache dipende dall'infrastruttura. Si applica quindi un'azione di *pause*. |

Tabella 4.8: Guasto del codebase server.

| | |
|---|---|
| **Descrizione** | Si schedula un task, si attende che sia stato eseguito almeno una volta e si disattiva il codebase server; successivamente si schedula un altro task mentre la codebase è guasta. |
| **Contesto** | Il sistema è impegnato nell'esecuzione di un task di test. |
| **Effetti** | Il caricamento delle classi necessarie per l'esecuzione dei job del task non può essere portato a termine. I Queue Server effettuano il caching dei file remoti caricati precedentemente, il task sottomesso prima del guasto può essere eseguito solo se assegnato ad un Queue Server che ha già eseguito una run ed ha effettuato il caching dei JAR. Il task che viene sottomesso dopo il guasto viene preso in carico dall'orchestratore ma non viene eseguito fino al ripristino della codebase. |
| **Recovery** | Il caching dei file permette spesso di continuare l'esecuzione dei task per cui è stata già effettuata una run, quando ciò non è possibile l'esecuzione è rimandata al ripristino del codebase server. |

#### Test sulla base di dati

Tabella 4.9: Guasto della base di dati.

| | |
|---|---|
| **Descrizione** | Si simula un guasto alla base di dati dell'orchestratore, poi del Queue Server. |
| **Contesto** | Il sistema è impegnato nell'esecuzione di un task di test. |
| **Effetti** | Il guasto alle basi di dati non è supportato in questa versione prototipale, questo tipo di guasto sarà integrato negli sviluppi futuri. |
| **Recovery** | Non supportato |

### 4.3.2 Test sul workflow

I guasti architetturali possono provocare direttamente o indirettamente fault a livello di workflow. In questa sezione si verifica il comportamento del sistema alle eccezioni lanciate durante l'esecuzione dei workflow. Si elencano le eccezioni esplicitandone la causa, gli effetti e le azioni di recovery.

Tabella 4.10: Eccezione: nessun servizio soddisfa i requisiti.

| | |
|---|---|
| **Eccezione** | *NoQueueServerAvailable* |
| **Descrizione** | Un task richiede l'esecuzione su una coda di lavoro non disponibile. |
| **Causa** | I Queue Server della coda di lavoro su cui eseguire il task non sono disponibili o sono guasti. |
| **Effetti** | Il job del task non viene eseguito, il task non può essere completato nella run corrente. Il job passa nello stato di esecuzione *NOT_EXECUTED_FOR_EXCEPTION*. |
| **Recovery** | Retry: il task viene continuato se possibile alla successiva esecuzione del trigger. |

Tabella 4.11: Eccezione: codebase server non disponibile.

| | |
|---|---|
| **Eccezione** | *CodebaseLoadingException* |
| **Descrizione** | Non è possibile caricare i JAR per il job in esecuzione né dalla cache né dal codebase server. |
| **Causa** | La codebase non è raggiungibile o non disponibile a causa di un guasto. |
| **Effetti** | Il job del task non viene eseguito, il task non può essere completato nella run corrente. Il job passa nello stato di esecuzione *NOT_EXECUTED_FOR_EXCEPTION*. |
| **Recovery** | Il task viene ritentato alla successiva esecuzione del trigger. |

Tabella 4.12: Eccezione: errore durante l'esecuzione del job.

| | |
|---|---|
| **Eccezione** | *JobExecutionException* |
| **Descrizione** | Il job del task in esecuzione ha lanciato un eccezione non preventivata. |
| **Causa** | Una delle operazioni eseguite dal job non è andata a buon fine e non era previsto in fase di progettazione del job. |
| **Effetti** | Il job del task non viene eseguito, il task non può essere completato è necessario avviare la compensazione. Il job passa nello stato di esecuzione *EXECUTED_WITH_EXCEPTIONS*. |
| **Recovery** | Vengono eseguiti a ritroso e in sequenza i job di compensazione per ogni job del task eseguito prima. |

Tabella 4.13: Eccezione: errore durante l'esecuzione della compensazione.

| | |
|---|---|
| **Eccezione** | *CompensationException* |
| **Descrizione** | L'esecuzione di un job di compensazione non è andata a buon fine. |
| **Causa** | Una delle operazioni eseguite dal job di compensazione non è andata a buon fine e non era previsto in fase di progettazione del job. |
| **Effetti** | Il job del task non viene eseguito, il task non può essere compensato. Il job passa nello stato *NOT_COMPENSABLE*. |
| **Recovery** | In questo caso è richiesto l'intervento di un operatore, non è possibile completare il task. |

# Conclusioni

Con questa tesi si è progettata un'architettura service oriented per lo scheduling e l'esecuzione di un workflow di job in un ambiente distribuito dinamico. Dalla progettazione si è realizzato un prototipo funzionante che è stato validato per la fault tolerance tramite una serie di test sull'architettura e sulle funzionalità fornite.

Il sistema realizzato in questo lavoro permette di gestire in maniera affidabile un workflow di job mediante l'orchestrazione di servizi distribuiti su una rete di "nodi" elaboratori che possono essere fisici o virtuali come accade nel private cloud. Il sistema possiede la caratteristica innovativa di funzionare in ambienti di rete in cui i cambiamenti sono rilevanti: la gestione dell'aggiunta, la rimozione e lo spostamento dei servizi in base alle necessità è una nota distintiva dell'architettura implementata.

Si è inoltre definito il concetto originale di *coda di lavoro* come cluster dinamico di servizi con caratteristiche comuni e capaci di elaborare job secondo le direttive di un orchestratore.

I test effettuati sul prototipo in uno scenario reale hanno confermato l'aderenza del prototipo ai requisiti definiti in fase di progettazione, inoltre, le prove evidenziato che le caratteristiche di affidabilità, tolleranza ai guasti e adattività inizialmente richieste sono state ampiamente soddisfatte.

Sotto le ipotesi definite in progettazione, le proprietà di *safety* e *liveness* fondamentali perché l'esecuzione di un workflow termini sicuramente e senza danni, sono state verificate e possono essere ritrovate nei comportamenti del sistema:

- se un client sottomette un task al sistema, il task viene sicuramente eseguito e l'esito memorizzato;
- l'esecuzione di un task è sempre conclusa o dichiarata fallita nel caso pessimo anche in presenza di guasti;
- se una coda di lavoro prende in carico un job, restituisce sempre un risultato;
- i dati di un job contenuti nella sua JobDataMap non sono mai corrotti anche in caso di crash grazie al ripristino.

Riassumendo, si è stato realizzato un sistema distribuito che presenta le caratteristiche di seguito elencate.

- **affidabilità e tolleranza ai guasti:** per mezzo delle proprietà di safety e liveness e la tolleranza ai guasti della maggior parte delle componenti.
- **adattività al cambiamento:** la proprietà principale del sistema è la dinamicità grazie alla possibilità di aggiungere o rimuovere servizi e code di lavoro.
- **scalabilità e adattamento al carico:** il sistema è per costruzione scalabile, si può pensare che l'orchestratore sia un punto debole, ma esso effettua solo un'attività di controllo che produce un carico di lavoro estremamente basso.
- **minima configurazione:** l'architettura realizzata si basa sul discovery dei servizi, è necessaria solo una minima configurazione iniziale delle componenti.

Il lavoro svolto per questa tesi sarà portato avanti al fine di pubblicare la soluzione realizzata come progetto SourceForge sotto licenza open source; al momento, non sono disponibili soluzioni che coniughino la maturità delle funzionalità ad una licenza open source.

Sebbene le funzionalità fornite dal sistema sono valide e di provata utilità, in futuro, la gestione dei workflow può essere estesa: i workflow gestiti attualmente dal sistema sono lineari, un'interessante estensione prevede l'introduzione di un meccanismo per la definizione di task che comprendano parallelizzazione, iterazioni e costrutti decisionali. Un altro possibile sviluppo futuro riguarda le capacità dell'orchestratore: la possibilità di poter orchestrare, oltre ai servizi, anche l'aggiunta e la rimozione degli stessi in maniera automatica in base al carico di lavoro complessivo; questo tipo di comportamento renderebbe il sistema più adattivo.

# Appendice A

# File di configurazione dei servizi River

Codice A.1: Esempio di file *transient-jeri-services.config*.

```
1  import com.sun.jini.start.NonActivatableServiceDescriptor;
2  import com.sun.jini.start.ServiceDescriptor;
3
4  com.sun.jini.start {
5
6      // HTTPD Service
7      private static httpd_codebase = "";
8      private static httpd_policy = "jsk-all.policy";
9      private static httpd_classpath =
           "/home/apache-river-2.1.2-bin/lib/classserver.jar";
10     private static httpd_impl = "com.sun.jini.tool.ClassServer";
11     private static httpd_service =
12         new NonActivatableServiceDescriptor(
13             httpd_codebase, httpd_policy, httpd_classpath, httpd_impl,
14             new String[]{"-port", "40080", "-dir",
                   "/home/apache-river-2.1.2-bin/lib-dl", "-verbose"});
15
16
17     // Reggie (Lookup Service)
18     private static reggie_codebase = "http://127.0.0.1:40080/reggie-dl.jar"
           + " http://127.0.0.1:40080/jsk-dl.jar";
19     private static reggie_policy = "jsk-all.policy";
20     private static reggie_classpath =
           "/home/apache-river-2.1.2-bin/lib/reggie.jar";
21     private static reggie_config = "transient-reggie.config";
```

```
22      private static reggie_impl =
            "com.sun.jini.reggie.TransientRegistrarImpl";
23      private static reggie_service =
24          new NonActivatableServiceDescriptor(
25              reggie_codebase, reggie_policy, reggie_classpath,
26              reggie_impl, new String[] { reggie_config });
27
28      //Descriptors list
29      static serviceDescriptors = new ServiceDescriptor[] {
30          httpd_service,
31          reggie_service
32      };
33  }
```

Codice A.2: Esempio di file *transient-reggie.config*.

```
1 com.sun.jini.reggie {
2     initialMemberGroups = new String[] {"thesis.test.group"};
3 }
```

Codice A.3: Impostazione delle policy di sicurezza per RMI: file *jsk-all.policy*

```
1 /* Policy file for the service starter */
2 /* Grant the local JAR files all permissions */
3
4 /* Add directory you want to allow security permissions */
5 grant codebase
      "file:${user.home}${/}workspace${/}apache-river-2.1.2-bin${/}lib${/}*" {
6     permission java.security.AllPermission;
7 };
```

# Appendice B

# File di configurazione dell'orchestratore

Codice B.1: Esempio di file *orchestrator.properties*.

```
# Project: Orchestrator

#Service Group name for orchestrator
ORCHESTRATOR_GROUP = thesis.test.group

# HTTP protocol (Needed for download)
JOB_CODEBASE_PROTOCOL = http
JOB_CODEBASE_HOST = 127.0.0.1
JOB_CODEBASE_PORT = 60080

# SFTP ACCESS TO JOB CODEBASE (Needed for upload)
JOB_CODEBASE_SFTP_PORT = 22
JOB_CODEBASE_SFTP_USERNAME = apiemont
JOB_CODEBASE_SFTP_PASSWORD = test_pass
#Relative directory of codebase server
JOB_CODEBASE_SFTP_DIR = codebase/jobs/

# Max matches when searching for services
MAX_SERV_MATCHES = 100

# Setting to create RMI registry
RMI_REGISTRY_HOST = 127.0.0.1
RMI_REGISTRY_PORT = 51099
```

Codice B.2: Esempio di file *hibernate.cfg.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
    Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property
        name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
        name="connection.url">jdbc:mysql://localhost/orchestrator</property>
    <property name="connection.username">thesis_user</property>
    <property name="connection.password">password</property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">5</property>
    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!-- Disable the second-level cache  -->
    <property
        name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">false</property>
    <!-- Drop and re-create the database schema on startup -->
    <property name="hbm2ddl.auto">validate</property>

    <!-- CLASS Mapping -->
    <mapping class="persistence.hibernate.entities.Schedulazione"/>
    <mapping class="persistence.hibernate.entities.JobSchedulazione"/>
    <mapping class="persistence.hibernate.entities.Codebase"/>
    <mapping class="persistence.hibernate.entities.SchedulazioneParam"/>
    <mapping class="persistence.hibernate.entities.RunSchedulazione"/>
    <mapping class="persistence.hibernate.entities.JobRun"/>
  <mapping class="persistence.hibernate.entities.EsitoRunSchedulazione"/>
  <mapping
      class="persistence.hibernate.entities.UnavailableServicesWorking"/>


  </session-factory>
</hibernate-configuration>
```

Codice B.3: Esempio di file *quartz.properties*.

```
# Configure Main Scheduler Properties (Orchestrator)
org.quartz.scheduler.skipUpdateCheck = true
org.quartz.scheduler.instanceName = OrchestratorScheduler
```

```
4
5 # Configure ThreadPool
6 org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
7 org.quartz.threadPool.threadCount = 5
8 org.quartz.threadPool.threadPriority = 5
9
10 # Configure JobStore (DataBase access for persistence)
11 org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
12 org.quartz.jobStore.driverDelegateClass =
       org.quartz.impl.jdbcjobstore.StdJDBCDelegate
13 org.quartz.jobStore.misfireThreshold = 60000
14 org.quartz.jobStore.tablePrefix = QRTZ_ORCH_
15 org.quartz.jobStore.dataSource = myDS
16 org.quartz.dataSource.myDS.driver = com.mysql.jdbc.Driver
17 org.quartz.dataSource.myDS.URL = jdbc:mysql://localhost/orchestrator
18 org.quartz.dataSource.myDS.user = thesis_user
19 org.quartz.dataSource.myDS.password = password
20 org.quartz.dataSource.myDS.maxConnections = 6
```

# Appendice C

# File di configurazione del Queue Server

Codice C.1: Esempio di file *queueServer.properties*.

```
1 # Project: QueueServer
2
3 #Group name for this QueueServer
4 QUEUE_NAME = LONG_WORKS_QUEUE
5 QUEUE_GROUP = thesis.test.group
```

Codice C.2: Esempio di file *quartz.properties*.

```
1 # Configure Main Scheduler Properties (SERVER)
2 org.quartz.scheduler.skipUpdateCheck = true
3 org.quartz.scheduler.instanceName = QueueServerScheduler
4 org.quartz.scheduler.classLoadHelper.class =
      customClassLoaders.CascadingClassLoadHelper
5
6 # Configure ThreadPool
7 org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
8 org.quartz.threadPool.threadCount = 5
9 org.quartz.threadPool.threadPriority = 5
10
11 # Configure JobStore
12 org.quartz.jobStore.misfireThreshold = 60000
13 org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
14 org.quartz.jobStore.driverDelegateClass =
      org.quartz.impl.jdbcjobstore.StdJDBCDelegate
15 org.quartz.jobStore.tablePrefix = QRTZ_Q1_
```

```
16 org.quartz.jobStore.dataSource = myDS
17
18 org.quartz.dataSource.myDS.driver = com.mysql.jdbc.Driver
19 org.quartz.dataSource.myDS.URL = jdbc:mysql://localhost/queueServer
20 org.quartz.dataSource.myDS.user = thesis_user
21 org.quartz.dataSource.myDS.password = password
22 org.quartz.dataSource.myDS.maxConnections = 6
```

# Bibliografia


[1] M. Adams et al. “Dynamic and extensible exception handling for workflows: A service-oriented implementation”. In: *BPM Center Report BPM-07-03, BPMcenter.org* (2007).

[2] M. Adams et al. “Dynamic, extensible and context-aware exception handling for workflows”. In: *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS* (2007), pp. 95–112.

[3] M. Armbrust et al. “Above the clouds: a Berkeley view of cloud computing”. In: *UC Berkeley Technical Report* UCB/EECS-2009-28 (2009), pp. 3247–3259.

[4] C. Clark et al. “Live migration of virtual machines”. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Vol. 2. USENIX Association. 2005, pp. 273–286.

[5] Google App Engine. URL: `http://code.google.com/appengine/` (visitato il 05/09/2011).

[6] Apache Software Foundation. URL: `http://www.apache.org/` (visitato il 31/10/2011).

[7] Apache River Framework. URL: `http://river.apache.org/` (visitato il 31/09/2011).

[8] S. Kalayci et al. “Design and implementation of a fault tolerant job flow manager using job flow patterns and recovery policies”. In: *Service-Oriented Computing–ICSOC 2008* (2008), pp. 54–69.

[9] KVM. *Kernel Based Virtual Machine*. URL: `http://www.linux-kvm.org/` (visitato il 25/09/2011).

[10] OASIS. *Organization for the Advancement of Structured Information Standards*. URL: `http://www.oasis-open.org/` (visitato il 14/09/2011).

[11] M.P. Papazoglou. "Service-oriented computing: Concepts, characteristics and directions". In: (2003).

[12] M.P. Papazoglou et al. "Service-oriented computing: State of the art and research challenges". In: *Computer* 40.11 (2007), pp. 38–45. ISSN: 0018-9162.

[13] D. F. Parkhill. *The Challenge of the Computer Utility*. Addison-Wesley Educational Publishers Inc., 1966.

[14] N. Santos, K.P. Gummadi e R. Rodrigues. "Towards trusted cloud computing". In: *Proceedings of the 2009 conference on Hot topics in cloud computing*. USENIX Association. 2009, pp. 3–3.

[15] Quartz Enterprise Scheduler. URL: `http://www.quartz-scheduler.org/` (visitato il 20/09/2011).

[16] Khawaja S. Shams et al. "Polyphony: A Workflow Orchestration Framework for Cloud Computing". In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 606–611. ISBN: 978-0-7695-4039-9.

[17] M. Uddin e A. A. Rahman. "Server Consolidation: An Approach to Make Data Centers Energy Efficient and Green". In: *International Journal of Scientific & Engineering Research* 1.1 (ott. 2010), pp. 1–7.

[18] W3C. *World Wide Web Consortium*. URL: `http://www.w3.org/` (visitato il 13/09/2011).

[19] WMWare. *ESX Server*. URL: `http://www.vmware.com/products/server/` (visitato il 25/09/2011).