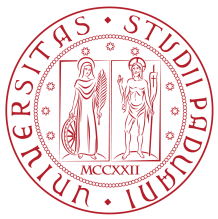
UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

WSUP: UN FRAMEWORK DISTRIBUITO PER WEB SERVICE E LA SUA INTEGRAZIONE NEL SERVIZIO UNIANTS

GIANLUCA PENGO

RELATORE:
PROF. SERGIO CONGIU

ANNO ACCADEMICO 2012/2013



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA

Indice

1. Introduzione

Negli ultimi anni si è assistito a una sempre maggior diffusione di Web service, che integrano al loro interno tecnologie eterogenee e sempre più spesso distribuite. Questo lavoro mira alla realizzazione di un framework flessibile su cui basare lo sviluppo e l'aggiornamento di Web service, in ambiente distribuito, senza imporre vincoli sul linguaggio utilizzato, sul pattern di elaborazione delle richieste e sulla topologia della rete che li ospita. La possibilità di distribuire l'intero Web service su più macchine, grazie alla definizione di specifiche per le dinamiche di comunicazione e un performante sistema che la supporta, permette di aumentare affidabilità e scalabilità, limitando la complessità degli applicativi e sollevando gli sviluppatori dalla responsabilità di gestire la comunicazione tra i componenti, riducendo al tempo stesso bug e tempi di sviluppo. La possibilità d'integrare applicativi sviluppati in differenti linguaggi di programmazione, permette invece di conciliare al meglio competenze del team di sviluppo e gli obiettivi di affidabilità e prestazioni che si vogliono raggiungere, nella realizzazione del Web service. La flessibilità è, infatti, una delle caratteristiche fondamentali che hanno guidato lo sviluppo di un framework snello e performante, senza tralasciare robustezza e affidabilità. Si è preso poi, in particolare considerazione l'utilizzo di linguaggi di programmazione a eventi, i quali permettono la gestione di alti volumi di richieste con un minor dispendio di risorse e maggiori prestazioni.

Lo sviluppo di tale lavoro nasce dall'esigenza di supportare l'evoluzione di Uniants, un servizio innovativo, che offre strumenti tecnologici a supporto della didattica e dello studio in ambito universitario, allo scopo di migliorare l'esperienza d'apprendimento. Uniants utilizza un approccio orizzontale, per favorire la condivisione e la collaborazione tra gli studenti e una maggior interazione con i docenti, il tutto accessibile tramite tecnologie mobile.

In questa relazione sarà esposta, dopo una breve introduzione all'argomento e descrizione del servizio Uniants, la progettazione e l'implementazione del framework, analizzando i punti affrontati durante questo lavoro di tesi. Saranno inoltre presentati i risultati dei test condotti, per la valutazione delle prestazioni.

2. Web Service

Secondo la definizione fornita dal W3C, un Web service è : "Un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete. Offre un'interfaccia software, descritta attraverso un linguaggio interpretabile da un elaboratore (ad esempio WSDL), grazie alla quale gli altri sistemi interagiscono con il Web service, attraverso le operazioni descritte nella sua interfaccia, tramite messaggi generalmente SOAP trasportati utilizzando il protocollo HTTP e formattati secondo lo standard XML, in congiunzione con altri standard Web"[1].

2.1. Cosa sono

Lo scopo di un Web service è di fornire funzionalità in nome del suo proprietario, persona o organizzazione, il quale fornisce un agente appropriato che implementa un particolare servizio. L'entità richiedente è anch'essa una persona o un'organizzazione che vuole far uso di tale servizio e utilizzerà un agente richiedente per scambiare messaggi con l'agente fornitore. Per scambiare messaggi, richiedente e fornitore devono accordarsi sulla semantica e sulla dinamica dello scambio dei messaggi.

La dinamica con cui avviene lo scambio dei messaggi tra le parti è documentata nella descrizione del Web service chiamata WSD (Web Service Description). La WSD è una specifica, interpretabile da un elaboratore, dell'interfaccia del Web service, descritta con il linguaggio WSDL (WSD Language)[2]. Definisce il formato dei messaggi, i tipi di dati, il protocollo di trasporto e il formato di serializzazione che deve essere utilizzato tra l'agente richiedente e l'agente fornitore. Specifica inoltre una o più locazioni di rete dove l'agente fornitore può essere invocato e può fornire informazioni sul pattern dei messaggi che questi può ricevere. In definitiva il WSD rappresenta un accordo che governa l'interazione tra le due parti.

La semantica di un Web service è l'aspettativa condivisa sul comportamento del servizio, in particolar modo in risposta ai messaggi ricevuti. Può essere considerato il contratto tra l'entità richiedente e l'entità fornitrice riguardo lo scopo e le conseguenze della loro interazione, contratto che può essere implicito o esplicito, legale o informale.

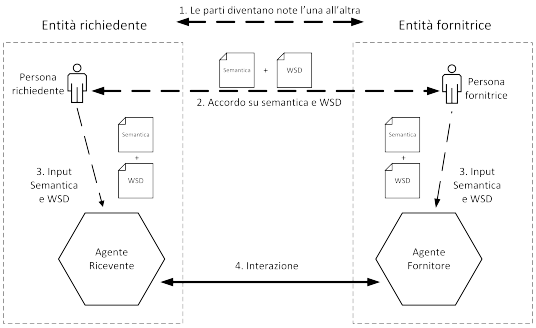


Figura 2.1.: Pattern generale d'interazione con Web Service, da [1]

Quindi la descrizione del servizio rappresenta un contratto che governa i meccanismi di interazione con un particolare servizio e la semantica rappresenta un contratto che descrive il significato e lo scopo dell'interazione. La distinzione tra le due descrizioni non è necessariamente rigida, infatti parte della semantica del Web service può essere descritta nella WSD, permettendo così di automatizzare gran parte del lavoro necessario a ottenere un'interazione efficace.

Ci sono diversi pattern in cui l'entità richiedente può interagire con un Web service. In generale, sono necessari i seguenti passi:

1. le entità, richiedente e fornitrice del servizio, vengono a conoscenza l'una dell'altra (o almeno una diventa nota all'altra);

2. le entità si accordano sulla descrizione del servizio e sulla semantica che governa l'interazione tra gli agenti delle due entità;

3. la descrizione del servizio e la semantica sono trasferite agli agenti;

4. il richiedente e il fornitore si scambiano messaggi, quindi eseguono alcune operazioni per conto delle entità richiedente e del fornitore.

Alcuni di questi passi possono essere automatizzati altri devono essere eseguiti manualmente.

Tramite un'architettura software basata sui Web service e grazie all'utilizzo di standard aperti, è possibile realizzare la comunicazione fra applicativi software implementati in linguaggi di programmazione differenti ed eseguiti su piattaforme hardware eterogenee. Permettendo inoltre di attuare un disaccoppiamento fra il sistema utente e il Web service stesso, grazie all'interfaccia uniforme esposta da quest'ultimo, cosicché modifiche in una delle due applicazioni possono essere

effettuate in modo trasparente all'interfaccia tra i due sistemi. Questa flessibilità consente la creazione di sistemi complessi, costituiti da componenti svincolati l'uno dall'altro e di raggiungere una forte riusabilità del codice e di applicazioni preesistenti.

L'utilizzo di Web service può comportare alcuni svantaggi tra cui: performance minori rispetto quelle riscontrabili utilizzando approcci alternativi di elaborazione distribuita quali Java RMI, CORBA o DCOM, e l'utilizzo del protocollo HTTP può permettere ai Web Service di evitare, in alcuni casi, misure di sicurezza imposte dai firewall.

Esiste poi l'UDDI (Universal Description Discovery and Integration), un archivio basato su XML che permette alle organizzazioni di pubblicare dati relativi ai servizi da loro offerti tramite Web service. Progettato per essere interrogato tramite messaggi SOAP[3], fornisce il collegamento ai documenti WSDL che descrivono i vincoli protocollari e i formati dei messaggi necessari per le interazioni con i Web service.

2.2. Web service e il WWW

Il Word Wide Web opera come un sistema informativo di rete che impone alcuni vincoli: gli agenti identificano gli oggetti del sistema, le risorse, con gli Uniform Resource Identifier (URI), gli agenti rappresentano, descrivono e comunicano lo stato delle risorse con una varietà di rappresentazioni come XML, HTML, PNG e si scambiano queste rappresentazioni attraverso protocolli che utilizzano gli URI per indentificare in modo diretto o indiretto gli indirizzi di risorse e agenti. Un stile architetturale ancora più vincolato per applicazioni Web affidabili conosciuto come Representational State Transfer (REST) è stato presentato da Roy Fielding nel 2000 [4], che ha poi ispirato la stesura del documento sull'architettura dei Web service del W3C[1]. Il Web REST è un sottoinsieme del WWW (basato principalmente su HTTP) dove gli agenti forniscono un'interfaccia semantica uniforme invece di una arbitraria o specifica dell'applicazione e manipola le risorse solo attraverso lo scambio delle loro rappresentazioni. Inoltre le interazioni REST sono stateless, quindi il significato del messaggio non dipende dallo stato della conversazione.

Si possono quindi identificare due classi principali di Web service:

- Web service REST-compliant, dove lo scopo principale del servizio è manipolare rappresentazioni di risorse Web usando un insieme uniforme di operazioni stateless;
- Web service arbitrari dove il servizio può esporre un insieme qualsiasi di operazioni, spesso basato su SOAP.

Entrambe le classi di Web service fanno uso di URI per identificare le risorse, protocolli Web (come HTTP e SOAP) e la serializzazione dei dati in XML per lo scambio dei messaggi.

2.3. SOAP

SOAP è un protocollo per lo scambio di informazioni strutturate in un sistema distribuito e decentralizzato [3]. Si basa sull'Extensible Markup Language (XML) per la formattazione dei messaggi e su una varietà di protocolli per la loro negoziazione e trasmissione, come HTTP o SMTP. I principali obiettivi di SOAP sono la neutralità (può essere usato su diversi protocolli come HTTP, SMTP, TCP e JMS), e l'estensibilità, raggiunti omettendo dal framework per lo scambio dei messaggi, caratteristiche come affidabilità, sicurezza, correlazione, routing, Message Exchange Pattern (MEP) e altre specifiche che sono definite come estensioni.

SOAP può formare il livello base della pila protocollare di un Web service. Le specifiche SOAP [5] definiscono un framework per lo scambio di messaggi che consiste di:

- **SOAP Processing Model**: definisce le regole per processare i messaggi;
- **SOAP Extensibility Model**: definisce i concetti di caratteristiche e moduli SOAP;
- **SOAP Protocol Binding Framework**: descrive le regole per definire binding a protocolli che possono essere impiegati nel trasporto dei messaggi SOAP;
- **SOAP Message Construct**: definisce la struttura di un messaggio SOAP.

I messaggi SOAP si basano sul metalinguaggio XML e sono composti da due segmenti: un Header opzionale e un Body obbligatorio. Il segmento opzionale Header contiene meta-informazioni sul contenuto del messaggio, su come processarlo, l'insieme di regole per la codifica per i tipi di dati definiti dall'applicazione, la convenzione per rappresentare le chiamate alle procedure e le risposte, e altre informazioni riguardanti routing, sicurezza e transazioni. Il segmento obbligatorio Body trasporta il contenuto informativo e deve essere serializzato in XML.

2.4. REST

REST, acronimo di Representational State Transfer, è un modello di architettura software per sistemi distribuiti, risultato essere quello predominante nello svi-

luppo di Web service, proposto da Roy Fielding (uno dei principali autori della specifica HTTP 1.0 e 1.1) nella sua tesi di dottorato nel 2000 [4].

Il modello lascia libertà nello sviluppo dei componenti, imponendo però i seguenti vincoli architetturali:

- **Client-server**: permette di separare aspetti dell'interfaccia utente da quelli della memorizzazione dei dati sfruttando un'interfaccia uniforme e permettendo lo sviluppo indipendente dei componenti, fattore molto importante in sistemi web based.

- **Stateless**: ogni richiesta del client contiene tutte le informazioni necessarie a servirla senza dover memorizzare alcuna informazione sullo stato del client nel server, migliorando così proprietà di visibilità, affidabilità e scalabilità. La visibilità è migliorata poiché un sistema di monitoraggio può basarsi sulla singola richiesta per determinarne la vera natura. L'affidabilità è migliorata perché sono più semplici le operazioni di ripristino in seguito a guasti parziali [6]. Infine la scalabilità è migliorata perché non è necessario gestire risorse (o informazioni relative allo stato del client) tra una richiesta e quelle successive, permettendo ai componenti di liberare velocemente risorse e semplificandone l'implementazione.

- **Cacheable**: i client possono utilizzare meccanismi di caching delle risposte. Questo permette di eliminare totalmente o in parte le interazioni client-server, consentendo un miglioramento di scalabilità e performance.

- **Interfaccia uniforme**: l'interfaccia uniforme tra client e server semplifica l'architettura e disaccoppia l'implementazione delle componenti, permettendone un'evoluzione indipendentemente.

- **Sistema a livelli**: il client non può sapere se è connesso direttamente al server o indirettamente attraverso sistemi intermedi. I sistemi intermedi possono migliorare la scalabilità del sistema, con meccanismi di load-balancing e shared caching, e rafforzare le politiche di sicurezza.

- **Code on demand** (opzionale): i server possono temporaneamente estendere o personalizzare le funzionalità di un client trasferendogli codice da eseguire.

Il rispetto di questi vincoli permette quindi di ottenere sistemi distribuiti che abbiano le proprietà di scalabilità, semplicità, manutenibilità, portabilità, visibilità e affidabilità.

Possiamo descrivere una schematizzazione semplificata di un'architettura REST come un insieme di client e server, dove questi ultimi possono essere: origin

server, gateway, proxy. Il client invia una richiesta al server e il server risponde con la risposta appropriata, la quale non tiene conto della storia passata delle altre richieste. Richieste e risposte contengono rappresentazioni di risorse, le quali rappresentano qualsiasi concetto coerente e significativo che può essere indirizzato. Le risorse sono identificate da URI e i vari componenti (client e server) comunicano attraverso un'interfaccia standard, come ad esempio HTTP. Un numero arbitrario di componenti (client, server, cache, tunnel ecc.) può mediare la richiesta, permettendo a un client di interagire con una risorsa conoscendone solamente l'identificatore e l'azione richiesta, senza sapere se ci sono proxy, gateway, firewall o tunnel tra esso e il server che fornirà la risposta. La rappresentazione della risorsa è generalmente un documento di cui il client conosce il formato, tipicamente HTML, XML o JSON.

A differenza di SOAP, REST non richiede necessariamente la serializzazione dei dati in XML, né di inserire un header nei messaggi, richiedendo quindi meno banda. Dalla versione 2.0 WSDL offre il supporto al binding di tutti i metodi di richieste HTTP, permettendo di descrivere facilmente anche servizi REST-full.

3. Uniants

Uniants è un servizio che mette a disposizione strumenti tecnologici a supporto dello studio e della didattica, semplificandone l'accesso e l'utilizzo. Sfrutta le possibilità offerte dalle tecnologie web e mobile per mettere realmente la tecnologia al servizio di studenti e docenti e migliorare l'esperienza di apprendimento.

3.1. Descrizione del progetto

Uniants [7] è progetto nato nell'ottobre del 2011 da due studenti di Ing. Informatica dell'Università degli Studi di Padova, Gianluca Pengo ed Enrico Battistella, con l'obiettivo di fornire un insieme di strumenti a supporto dello studio e dell'insegnamento di studenti e docenti universitari. Il servizio è stato aperto al pubblico il 25 settembre 2012, con una prima fase di test che ha coinvolto più di 600 studenti dell'Università degli Studi di Padova.

Le principali funzionalità offerte dal servizio sono la condivisione di documenti, quali appunti, slide, esercizi e dispense, la possibilità di postare domande e ricevere risposte e la condivisione di informazioni. Il tutto organizzato per corso e accessibile in modo semplice e veloce, con un'interfaccia estremamente intuitiva. Ciò che ha motivato la scelta di sviluppare tale progetto nasce dalla mancanza di un unico servizio che fornisca tali funzionalità e che sia utilizzabile attivamente sia da studenti sia da docenti.

Analizzando gli strumenti disponibili in rete è possibile trovare un gran numero di piattaforme di e-learning, dover per e-learning s'intende: *l'utilizzo delle tecnologie multimediali e di Internet per migliorare la qualità dell'apprendimento facilitando l'accesso alle risorse e ai servizi, così come anche agli scambi in remoto e alla collaborazione, per la creazione di comunità virtuali di apprendimento* [8]. Queste piattaforme necessitano di essere installate su server web, quindi di personale che le installi e le gestisca, cosa che nella maggior parte dei casi non può essere sostenuta dagli studenti ma è effettuata dalle singole facoltà o in alcuni casi dalle università, le quali devono sostenerne tutti i costi. Inoltre molto spesso queste piattaforme sono strumenti unidirezionali di comunicazione tra docenti e studenti, in cui solo i primi possono pubblicare documenti e aprire forum, dove è possibile l'interazione studente-studente e studente-docente. Questo approccio top-down, fa si che siano utilizzate dagli studenti principalmente come un

archivio da cui scaricare slide e altri documenti forniti dai docenti e reperire informazioni su corsi ed esami, non sfruttando adeguatamente i forum, poiché spesso non è possibile creare nuove discussioni e non è immediato vedere le discussioni aperte da i compagni o gli ultimi commenti inseriti, senza entrare all'interno dei forum. Questa piattaforme si rivelano spesso essere poco intuitive ed elastiche nella gestione dei corsi da parte dei professori. Alcune di queste piattaforme si pongono poi come strumento a supporto della didattica a distanza o della formazione aziendale, senza fornire strumenti efficaci necessari a soddisfare le effettive esigenze di studenti e docenti coinvolti nella didattica frontale, attualmente la più diffusa.

Nelle realtà in cui tali piattaforme non siano presenti, molti professori utilizzano siti web personali per fornire informazioni e documenti agli studenti che frequentano i corsi da loro tenuti. Tale soluzione, che richiede una minima conoscenza dello sviluppo di siti web (non sempre diffusa in ambiti d'insegnamento non scientifici), comporta una gestione del materiale e delle informazioni fornite non immediata e più impegnativa.

Il quadro della situazione appena fornito, fa si che ove siano offerti, questi servizi non sopperiscano al bisogno reale degli studenti di disporre di un mezzo per scambio di materiale e informazioni. Il ché ha portato alla nascita di una numerosa quantità di piccole e medie realtà come forum, wiki, gruppi Facebook e cartelle condivise su Dropbox, dove gli studenti condividono informazioni e documenti.

Infine bisogna segnalare che in molte realtà non vi è nessuno strumento tecnologico a supporto della didattica e dello studio e materiale e informazioni sono diffusi di persona o tramite email. In questo panorama lo studente si trova a ricercare il materiale di supporto allo studio in siti differenti, perdendo inutilmente tempo, senza avere a disposizione un servizio unico che fornisca ciò di cui ha bisogno.

Uniants punta a fornire un servizio unico che integri le funzionalità offerte dai diversi servizi fin qui analizzati, semplificandone l'accesso e l'utilizzo, sfruttando le possibilità offerte dalle tecnologie web e mobile, per mettere realmente la tecnologia al servizio di studenti e docenti e migliorare l'esperienza di apprendimento.

Non si vuole porre quindi come uno dei tanti strumenti di e-learning o di insegnamento a distanza, né tantomeno come un semplice archivio di documenti dove si depositano slide e appunti per permetterne la condivisione. Ciò che si vuole favorire con tale servizio è una maggiore e semplificata condivisione della conoscenza tra studenti, offrendo la possibilità di sviluppare una più attiva interazione sia da parte dei docenti, ma in particolar modo da parte degli studenti. Utilizzando un approccio orizzontale, collaborativo e meno formale, fornendo

uno spazio in cui poter chiarire i propri dubbi, approfondire tematiche collegate ai corsi e commentare appunti e slide, permettendo di avere un unico strumento tecnologico dove gestire il proprio materiale didattico e trovare le informazioni necessarie. Il tutto accessibile da dispositivi mobili come smartphone e tablet, sempre più diffusi tra studenti e docenti.

Molto lavoro è stato e sarà svolto per lo sviluppo e la realizzazione un'interfaccia semplice e intuitiva, utilizzabile da chiunque, così da porre l'attenzione dell'utente sui contenuti e non sulle dinamiche di utilizzo della piattaforma.

Uniants pone molta attenzione sul rispetto del copyright dei contenuti fornendo la possibilità di segnalare contenuti che lo violano, i quali vengono preventivamente resi inaccessibili e successivamente verificati.

La piattaforma consente inoltre di ricercare informazioni inerenti corsi erogati in altre università e facoltà e creare contatti con studenti e docenti con interessi accademici simili.

Il servizio comporta poi vantaggi anche per le strutture universitarie poiché il servizio è completamente gratuito. È offerta la possibilità a quest'ultime di creare i corsi verificati all'interno della piattaforma, che si differenzieranno dai corsi creati da studenti.

3.1.1. Funzionali attualmente offerte

Le principali funzionalità attualmente offerte da Uniants sono:

- Condivisione di documenti come slide, appunti, dispense ed esercizi che possono essere condivisi sia da studenti sia da docenti con un semplice drag-n-drop sulla pagina del corso o del gruppo di studio. Tutti i documenti sono suddivisi per corso per facilitare l'organizzazione e la ricerca del materiale. È possibile commentare ciascun documento.

- Creazione di discussioni dove poter chiedere chiarimenti su argomenti trattatati nel corso, richiedere informazioni e approfondire tematiche collegate al corso.

- Possibilità di creare gruppi di studio in cui condividere documenti e discussioni.

3.1.2. Funzionalità rilasciate nei prossimi mesi

Nei prossimi mesi saranno rilasciate ulteriori funzionalità tra cui:

- Scrittura collaborativa di documenti, con una modalità appositamente sviluppata per prendere appunti durante le lezioni;

- Possibilità di prendere appunti sui documenti condivisi, operazione che potrà essere svolta anch'essa in modalità collaborativa;
- Servizi di messaggistica;
- Applicazione mobile.

Sarà poi migliorato il controllo e la personalizzazione dell'accesso ai contenuti condivisi e la loro gestione. Sarà migliorata l'interfaccia utente rendendola più intuitiva e veloce. Saranno poi fornite funzionalità come la gestione dei contenuti del corso in lezioni, la possibilità da parte del professore di creare sezioni protette da password, in modo simile a quanto avviene nelle attuali piattaforme di e-learning.

3.1.3. Utilizzo del servizio

Forniremo ora una breve descrizione sull'utilizzo del servizio.

Uno studente che intende utilizzare la piattaforma può registrarsi, fornendo alcuni dati personali, università e corso di studi d'appartenenza. Una volta completata la registrazione è possibile ricercare i corsi d'interesse della propria università. Sono poi suggeriti alcuni corsi del proprio corso di studio tra i più frequentati e nel caso in cui i corsi d'interesse non siano presenti all'interno della piattaforma è offerta la possibilità di crearli. Una volta individuati i corsi d'interesse ed eseguita l'iscrizione ad essi all'interno della piattaforma è possibile creare discussioni, inserire commenti e documenti all'interno dei singoli corsi. È poi possibile ricercare e aggiungere i propri compagni di corso all'insieme dei propri contatti e creare con questi, gruppi di studio. I gruppi di studio sono dei contenitori simili ai corsi, ai quali possono essere collegati, all'interno dei quali è possibile inserire contenuti accessibili solo ai membri del gruppo di studio se non specificato diversamente. È possibile condividere sui principali social network la creazione di discussioni e l'inserimento di commenti e documenti, i quali saranno però accessibili previa registrazione alla piattaforma e iscrizione al corso. La visualizzazione dei contenuti relativi ad un corso è possibile solo a chi è iscritto a tale corso. Vi è poi un sistema di notifiche, personalizzabile, che permette di visualizzare le ultime azioni eseguite da altri utenti nei corsi seguiti.

L'iscrizione da parte dei docenti è permessa attraverso l'utilizzo dell'indirizzo email accademico per verificare l'effettiva figura professionale e fornire una garanzia agli utenti su chi ricopre tale ruolo all'interno della piattaforma. Nel caso in cui l'iscrizione con tale indirizzo email non sia possibile, l'iscrizione è gestita contattando il servizio di supporto. I docenti possono gestire i corsi da loro tenuti, già presenti all'interno della piattaforma o creandoli in caso non siano presenti, e usufruire delle stesse funzionalità offerte agli studenti. Tale figura avrà,

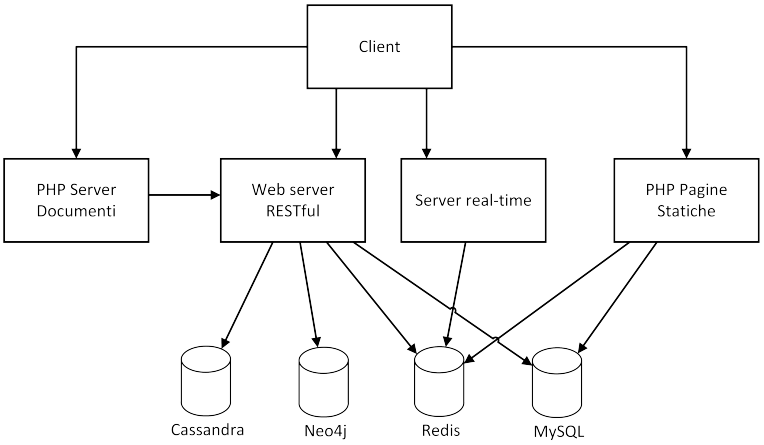


Figura 3.1.: Architettura applicativi Uniants.

nei prossimi aggiornamenti della piattaforma, funzionalità aggiuntive basate sulle necessità di tale ruolo.

3.2. Server attuale

Uniants si basa su applicativi lato server che integrano differenti tecnologie necessarie a soddisfare le esigenze specifiche delle diverse aree di applicazione. Vengono infatti utilizzate tecnologie lato server consolidate come Apache, PHP e MySQL e altre più recenti come Node.js e database NOSQL.

Attualmente il software lato server è suddiviso in quattro moduli principali:

- Due applicativi PHP, uno basato su Zend Framework 1 [9], che gestisce i contenuti statici delle pagine web, e un secondo che gestisce il download e l'upload dei documenti.

- Due applicativi Node.js: un server web RESTful per la gestione dei contenuti dinamici delle pagine web e un processo per il realt-time web, che gestisce la comunicazione tra server e client utilizzata per aggiornare dinamicamente i contenuti delle pagine web, inviando aggiornamenti dal server al client.

Saranno ora illustrate le principali tecnologie utilizzate e come sono state integrate negli applicativi.

3.2.1. Database

Tre dei quattro applicativi interagiscono con quattro differenti database: Cassandra, MySQL, Neo4j e Redis. Ognuno dei quali è utilizzato per memorizzare informazioni adatte alle proprie caratteristiche. Per esempio Neo4j, un database a grafo, memorizza le informazioni relative alle interazioni sociali, Cassandra, un database distribuito non relazionale, è utilizzato nella gestione delle notifiche e memorizzazione dello storico delle operazioni eseguite su corsi e gruppi di studio.

Sarà ora data una breve descrizione dei database utilizzati.

3.2.1.1. Apache Cassandra

Apache Cassandra [10] è un DBMS distribuito, NOSQL e open source. Interamente sviluppato in Java, da Facebook, è stato reso disponibile nel 2008, poi nel 2009 è entrato a far parte del progetto Incubator di Apache Software Fundation. Oggi è utilizzato da molte importanti compagnie come Netflix, eBay, Twitter, Reddit, Cisco e Digg. Apache Cassandra è decentralizzato in quanto i nodi del cluster sono identici, il throughput in lettura e scrittura scala linearmente con il numero di nodi aggiunti, la sostituzione di un nodo può essere effettuata senza downtime e offre un livello di consistenza regolabile. È un DBMS NOSQL, infatti non utilizza la sintassi SQL e non è relazionale, fornendo un struttura di memorizzazione chiave-valore con eventual consistency. Eventual consistency significa che: *dato un periodo di tempo sufficientemente lungo, nel quale non vengono effettuati aggiornamenti, si può assumere che tutti gli aggiornamenti saranno, eventualmente, propagati nel sistema e ogni sua replica sarà consistente.* Ciò significa che per brevi periodi di tempo, duranti i quali vengono propagati gli aggiornamenti, il database può risultare inconsistente. Questo concetto è conosciuto anche come BASE (Basically Available, Soft state, Eventual consistency), in opposizione a quello di ACID (Atomicity, Consistency, Isolation, Durability) dei database tradizionali. Cassandra utilizza un modello di dati column-oriented, in cui una tabella è mappa multidimensionale distribuita e indicizzata da una chiave. Le colonne (Column) sono le più piccole entità di memorizzazione composte dalla tripletta: nome, valore e timestamp. Le colonne sono raggruppate in liste, chiamate famiglie (ColumnFamily), e anch'esse associate a un nome e un timestamp. Esistono due tipologie di famiglie: semplici e super. Quelle super possono essere viste come delle famiglie di ColumnFamily. Infine è definito il Keyspace, un namespace di cui generalmente ne è definito uno per applicazione. Ogni operazione è atomica, per ogni singola tupla e per replica, ed è possibile specificare il tipo di ordinamento delle colonne all'interno di ColumFamily, semplici o super, il quale può essere alfabetico o cronologico.

Cassandra è impiegato nella memorizzazione delle informazioni relative alle notifiche e alle bacheche di utenti e corsi. I dati sono memorizzati in ordine cronologico inverso così da velocizzarne l'accesso agli ultimi inseriti, che rappresentano le ultime operazioni eseguite. Si utilizza tale database per memorizzare questi dati per la velocità di accesso a quelli più recenti e alla possibilità di gestire grandi moli di dati, distribuendo il database su più nodi, poiché c'è una forte ridondanza dei dati memorizzati per ottenere prestazioni più elevate. Parte di questi dati, principalmente notifiche, sono poi periodicamente trasferiti su altri sistemi di memorizzazione, per liberare risorse, quando risultano essere obsoleti.

3.2.1.2. MySQL

MySQL [11] è un DBMS relazionale open source, tra i più diffusi in ambito web e rilasciato sotto licenza GNU GPL.

MySQL è utilizzato per memorizzare dati prevalentemente statici come informazioni relative agli account degli utenti, alle attività didattiche e ai corsi di studio. Viene poi utilizzato per memorizzare log e statistiche dopo essere state elaborate.

3.2.1.3. Neo4j

Neo4j [12] è un database a grafo open source sviluppato interamente in Java. È transazionale e può essere utilizzato sia in modalità server, dove viene interrogato attraverso richieste REST, che embedded, permettendo di integrarsi nelle applicazioni sviluppate in linguaggi di programmazione, come Java e Scala, il cui codice viene eseguito su JVM, permettendo così di manipolare direttamente i nodi. I quali sono identificati da un nome e possono contenere proprietà, ossia dei valori corrispondenti ai tipi di dato elementari di Java, stringhe e array. È inoltre possibile unire nodi tramite archi, orientati o meno, i cui tipi sono definiti dallo sviluppatore e possono avere proprietà come i nodi. Il grafo è schema-less, il che da un lato permette di definire dati molto eterogenei con il minimo sforzo e dall'altro delega la responsabilità della consistenza dei dati all'applicazione. Neo4j integra poi un servizio di indicizzazione basato su Lucene.

Neo4j è utilizzato per memorizzare la maggior parte delle informazioni legate alle interazioni sociali del utente con le attività didattiche e con gli altri utenti. Per esempio informazioni sulle iscrizione alle attività didattiche, ai suoi contatti, ai post e documenti caricati. I singoli oggetti sono rappresentati come nodi e le azioni o relazioni tra gli oggetti sono rappresentate da archi. Si utilizza tale database nella gestione di questi dati per la velocità nel reperire informazioni collegate tra loro grazie all'attraversamento del grafo.

3.2.1.4. Redis

Redis [13] è un database chiave-valore in-memory e open source. Il suo modello dati può essere paragonato a un dizionario dove le chiavi sono mappate in valori, i quali non sono limitati alle sole stringhe ma possono essere liste, insiemi (ordinati e non) e hash. Ne consegue che le azioni eseguibili sulle chiavi sono dipendenti dal tipo di dato dei valori su cui sono mappate. Grazie al fatto che offre una buona configurazione della persistenza, può essere replicato su di un numero arbitrario di nodi e risiede quasi interamente in memoria, viene spesso utilizzato per implementare sistemi di virtual memory e caching per le sue elevate performance in lettura e scrittura. È infatti utilizzato in importanti realtà come GitHub, Guardiang, Flickr, Disqus e Stackoverflow. Redis implementa inoltre il modello per lo scambio di messaggi publish-subscribe, nel quale il mittente del messaggio, chiamato publisher, invia il messaggio senza specificarne il destinatario, chiamato subscriber. I messaggi del publisher sono classificati in canali e i subscriber esprimono interesse rispetto a uno o più canali, azione detta di sottoscrizione, e ricevono solo i messaggi delle classi di loro interesse.

Su Uniants, Redis è utilizzato per la memorizzazione delle sessioni degli utenti e dei dati grezzi relativi a log e statistiche, quest'ultime sono poi giornalmente elaborate e trasferite su un apposito database MySQL. Viene inoltre impiegato nella gestione delle notifiche e degli aggiornamenti con il client grazie al meccanismo di publish-subscribe.

3.2.2. Applicativi lato server

Analizzeremo ora i quattro applicativi lato server, due dei quali implementati in PHP e altri due in Node.js.

3.2.2.1. Applicativi PHP

Attualmente il servizio utilizza due differenti server Apache, uno per l'applicazione principale e uno per la gestione dei documenti, che risiedono in due macchine di differenti data center. L'applicazione principale risiede nella stessa macchina che ospita il server Node.js. I due applicativi sono utilizzati per servire i contenuti statici delle pagine web e per il download e upload dei documenti.

L'applicativo dedito a servire le pagine web è sviluppato in PHP e sfrutta lo *Zend Framework 1.* Zend Framework è un framework open source per lo sviluppo di applicazioni web, realizzato in PHP e rilasciato sotto licenza BSD, progettato con lo scopo di semplificare l'attività di sviluppo web e agevolare la produttività mettendo a disposizione degli sviluppatori una serie di librerie e componenti.

L'applicativo s'interfaccia con MySQL per reperire informazioni su utenti e corsi e gestire le operazioni di login. Interagisce inoltre con Redis per la gestione delle sessioni e operazioni di log e statistiche.

Il secondo applicativo, gestisce le operazioni di upload e download dei documenti cooperando con il server RESTful in Node.js, senza interfacciarsi con alcun database. Si occupa esclusivamente di gestire il trasferimento dei file caricati dagli utenti e di garantirne l'accesso ai soli utenti che ne possiedono l'autorizzazione, la quale viene verificata tramite il servizio RESTful offerto dal server sviluppato in Node.js.

3.2.2.2. Applicativi Node.js

Node.js [14]è un linguaggio di programmazione lato server sviluppato per realizzare applicazioni web altamente scalabili. Utilizza un modello a eventi e una gestione del I/O non bloccante che permette di ottenere applicazioni web snelle ed efficienti, adatte a utilizzi real-time e data-intensive che possono essere distribuite su differenti macchine. Node.js utilizza il linguaggio interpretato *Javascript*, il *V8 JavaScript engine* di Google per essere eseguito, la *libUV* come livello di astrazione e una core library realizzata principalmente in JavaScript. Node.js, inizialmente sviluppato da Ryan Dahl nel 2009 e poi sponsorizzato da Joyent, ha avuto una grande diffusione nello sviluppo di applicazioni di real-time web e a bassa latenza grazie alle sue ottime performance, dovute principalmente alla gestione asincrona del I/O, il suo modello a eventi, alla rapida curva di apprendimento (dato che la maggior parte degli sviluppatori di applicazioni web conosce JavaScript), alla vasta disponibilità di librerie e a una community molto attiva. Oggi è utilizzato da importanti compagnie come LinkedIn, Microsoft, Yahoo! e Walmart. Node.js, a differenza dei linguaggi e dei framework tradizionali, alloca le risorse necessarie al server web solo quando è necessario e senza pre-allocare grandi quantità di risorse. Per esempio per rispondere a una richiesta HTTP generalmente Apache pre-alloca 8MB di memoria mentre Node.js solo 8KB. Generalmente, nelle applicazioni web, la parte più time-intensive nell'elaborazione della risposta a una richiesta, è l'interazione con il database. Node.js avendo una gestione non bloccante dell'I/O può effettuare altre elaborazioni mentre è in attesa della risposta a una query al database, il tutto in modo trasparente al programmatore in quanto utilizza un modello a eventi. Questo fa si che il web server sia molto più efficiente nella gestione delle risorse computazionali e ne faccia un ottimo candidato per l'implementazione di applicazioni web based con necessità di elaborare un gran numero di richieste con un basso tempo di risposta.

Saranno ora esposti i due applicativi sviluppati in Node.js

Server web RESTful Il server RESTful è composto da un web server e un insieme di controller e model con un pattern simile al *Model-View-Controller*, dove la view risiede sul client. Ricevuta una richiesta, il server web, utilizzando un meccanismo di routing, individua il controller e l'azione associata a tale richiesta e la invoca in modo asincrono. Una volta ricevuti i risultati dell'elaborazione, questi sono serializzati in JSON e inviati al client. Sono presenti numerosi controller ognuno dei quali espone un insieme di azioni su di un particolare tipo di oggetto come ad esempio l'utente, un documento o una discussione. I controller utilizzano poi specifici model per interfacciarsi con i differenti database. È utilizzato un sistema di sessioni, memorizzate su Redis e inizializzate dall'applicativo PHP, per ottenere velocemente informazioni sull'utente che ha effettuato la richiesta e a cui è possibile garantire l'accesso alle sole risorse di cui possiede l'autorizzazione.

Server per real-time web Il *real-time web* è un insieme di tecnologie e pratiche che consentono all'utente di ricevere le informazioni non appena sono pubblicate, invece di richiedere a lui o a software di controllare periodicamente la presenza di aggiornamenti. Questo server gestisce le connessioni, tra client e server, necessarie a inviare messaggi dal server al client per aggiornare contenuti dinamici delle pagine web. Per realizzare la connessione client-server si utilizza la libreria Socket.IO [15] che seleziona a runtime, in base alle caratteristiche del browser e della connessione, la tecnologia di trasporto più conveniente tra le seguenti: WebSocket, Adobe® Flash® Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe, JSONP Polling. Inoltre mette a disposizione funzionalità come heartbeating, timeout e riconnessione automatica. Una volta istanziata la connessione, viene attivata la sottoscrizione al canale Redis relativo all'utente e non appena viene effettuato un publish su tale canale, il messaggio ricevuto viene inviato al client. L'azione di publish viene effettuata da specifici controller del server web REST-full, richiamati in seguito all'esecuzione di specifiche azioni da parte di un utente, i quali verificano gli utenti "interessati" a tale azione ed effettuano l'operazione di publish sui canali di questi utenti.

3.3. Obiettivi nuovo lato server

Da quanto descritto precedentemente si evince che la gran parte della logica del servizio risiede in un unico processo, il ché limita molto l'affidabilità e la scalabilità del servizio. Si è quindi deciso di riprogettare buona parte dell'applicazione lato server così da poter garantire una maggiore affidabilità del servizio e raggiungere un buona scalabilità.

Per ottenere questi due principali obiettivi è necessario sviluppare un framework su cui basare l'aggiornamento dell'applicazione. Tale framework dovrà

essere distribuito, garantire la comunicazione tra i componenti che integrerà, i quali potranno essere sviluppati in differenti linguaggi di programmazione ed essere distribuiti e replicabili. Dovrà poi offrire un buon load balancing e la possibilità di aggiornare i componenti senza downtime. Il framework dovrà essere utilizzato principalmente con componenti sviluppati in Node.js, riadattando quelli precedentemente sviluppati e in uso nel lato server di Uniants, cercando di limitare il lavoro necessario per la loro integrazione con tale framework.

È stata sottolineata la necessità di dover utilizzare il framework con componenti sviluppati in Java, poiché in programma lo sviluppo di un applicativo Java che gestisca tutte le interazioni con il database a grafo Neo4j, per poter utilizzare il database in modalità embedded, permettendo così di gestire tutte le operazioni in modo più efficiente e ottenendo prestazioni molto più elevate.

Il framework non dovrà limitarsi a creare una soluzione ad-hoc per il servizio Uniants, ma fornire una soluzione generale per lo sviluppo di applicazioni lato server distribuite, per la realizzazione di Web service. Sarà inoltre necessario fornire alcuni componenti essenziali come un web server che gestisca le richieste e un'interfaccia per gestire e monitorare l'esecuzione del framework.

4. Progettazione framework

In questo capitolo saranno esposte le diverse soluzioni analizzate per la realizzazione del framework e in particolare del middleware message-oriented.

Il framework che prenderà il nome di WSUP dovrà fornire una base per sviluppare Web service distribuiti, garantire una buona scalabilità del servizio e cercare di minimizzare overhead introdotto nel tempo di risposta alle richieste. Dovrà semplificare gestione, configurazione e distribuzione dei componenti e richiedere il minimo sforzo necessario a integrare applicativi precedentemente sviluppati. Definire un interfaccia uniforme per permettere la comunicazione di componenti sviluppati in differenti linguaggi di programmazione ed eseguiti su piattaforme hardware eterogenee e differenti sistemi operativi. Offrire poi la possibilità di replicare i componenti per aumentare affidabilità e scalabilità. Permettere l'aggiornamento e l'aggiunta di servizi e nuovi nodi senza downtime per il Web service.

Il framework sarà quindi composto da un sistema per lo scambio dei messaggi tra i componenti (middleware message-oriented) e una serie di componenti per il load balancing, la gestione del framework , e due server web uno per richieste REST e uno per il real-time web.

4.1. Middleware message-oriented

Lo sviluppo di WSUP inizia con la progettazione del middleware message-oriented (MOM) che permetterà lo scambio di messaggi tra i vari componenti, i quali possono risiedere nei differenti nodi in cui sarà distribuito il Web service.

Un Middleware Message-Oriented (MOM), è un'infrastruttura client-server che attraverso lo scambio asincrono di messaggi permette di distribuire un'applicazione tra più piattaforme eterogenee, incrementandone l'interoperabilità, la portabilità e la flessibilità. Tale infrastruttura semplifica lo sviluppo di applicazioni che utilizzano sistemi operativi e protocolli di rete differenti, consentendo al programmatore di ignorare i dettagli degli stessi, grazie a API che coprono diverse piattaforme e tipologie di rete. I messaggi inviati verso applicazioni non disponibili sono memorizzati in apposite code e vengono consegnati quando l'applicazione destinataria torna disponibile.

4.1.1. Scelta dell'architettura

Saranno ora presentati i due principali approcci architetturali che sono stati valutati ed esposta l'analisi di mercato effettuata per la scelta della tecnologia da utilizzarsi nello sviluppo del middleware.

4.1.1.1. Broker vs Brokerless

La maggior parte dei sistemi di messaging utilizza un'architettura a stella o a hub con al centro un messaging server, il *broker*, utilizzato dai vari componenti o applicazioni per comunicare tra di loro. Questo modello porta alcuni vantaggi. Le applicazioni per comunicare tra di loro, devono conoscere solo l'indirizzo del broker, il quale poi instrada i messaggi alla giusta applicazione in base a determinati criteri (che si basano generalmente sul nome della coda, meccanismi di routing o proprietà del messaggio), invece di utilizzare indirizzi fisici delle applicazioni. Il tempo di vita del mittente e del destinatario non devono necessariamente sovrapporsi. L'applicazione mittente può inviare il messaggio e chiudersi, perché il messaggio sarà disponibile in qualsiasi momento successivo per il destinatario. Il modello è robusto contro i guasti dell'applicazione, in quanto, quando se ne verifica uno, i messaggi precedentemente inviati e non ancora ricevuti non vengo persi. Gli svantaggi sono principalmente due: richiede un'eccessiva quantità di comunicazione di rete e il broker può rappresentare un collo di bottiglia per l'intero sistema, dato che tutti i messaggi devo passare attraverso di lui.

Questo modello può essere utilizzato sia con approcci di tipo SOA sia con più efficienti approcci a pipeline.

Nelle architetture *brokerless* la comunicazione tra le applicazioni è diretta. In questo modo si elimina il collo di bottiglia introdotto dal broker, ideale per applicazioni che necessitano di bassa latenza e un alto throughput. S'introduce però la necessità di conoscere gli indirizzi fisici delle applicazioni per instaurare la comunicazione, condizione accettabile per casi semplici ma inaccettabile nel caso in cui si debbano supportare centinaia di applicazioni.

Si può quindi introdurre il concetto di *broker come servizio di directory* dove il broker distribuisce informazioni sugli indirizzi delle applicazioni. In quest'approccio l'applicazione si registra al broker, rendendo noto il suo indirizzo fisico, e per inviare messaggi a un'altra applicazione richiede l'indirizzo di quest'ultima al broker, il quale mantiene aggiornate le informazioni sulle applicazioni attive e i loro indirizzi. In questo modo si possono ottenere buone performance e una più semplice gestione del sistema.

Un altro modello architetturale è quello a *broker distribuito*, che permette di sfruttare alcuni vantaggi sia del modello a broker sia di quello brokerless. Permette al tempo di vita delle applicazioni di non doversi necessariamente sovrapporre,

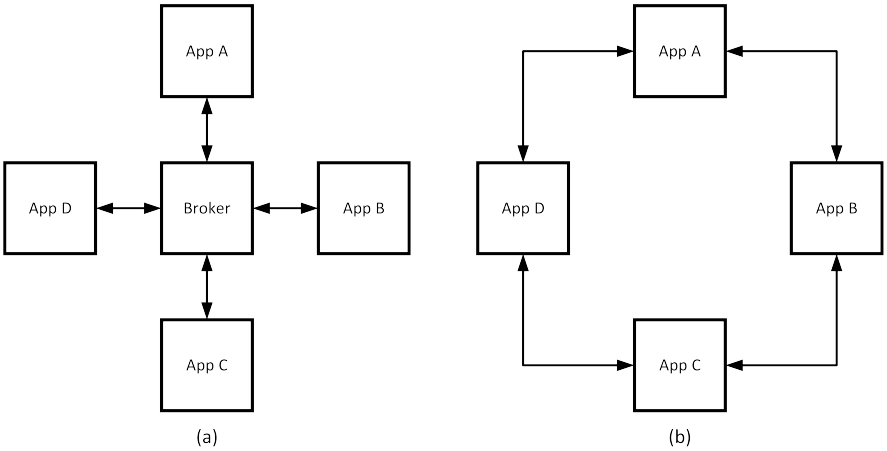


Figura 4.1.: (a) Architettura a broker. (b) Architettura brokerless.

perché i messaggi sono memorizzati nel broker fino a che il destinatario non diventa disponibile, garantendo inoltre che i messaggi non vengano persi a seguito di un crash di un'applicazione. Per ottenere ciò è necessario introdurre un broker tra ogni coppia di applicazioni che vogliono comunicare, il quale si occupa anche di ottenere gli indirizzi delle applicazioni attraverso il servizio di directory. Si mantiene in questo modo la manutenibilità senza avere il collo di bottiglia di un singolo broker, i vari broker posso anche risedere in macchine dedicate, differenti da quelle in cui risiedono le applicazioni.

Per aumentare l'affidabilità si può utilizzare un *servizio di directory distribuito*. Nei precedenti modelli se il servizio di directory non è disponibile, in seguito a un guasto, non è possibile istanziare nuove comunicazioni, poiché non è possibile ottenere gli indirizzi delle applicazioni. In questo modello il servizio di directory è distribuito eliminando il single point of failure nella gestione della configurazione.

4.1.1.2. Analisi di mercato

Sarà ora esposta l'analisi di mercato condotta per l'individuazione di una tecnologia da utilizzarsi per la realizzazione del middleware message-oriented. Tale scelta viene trattata ora in quanto condiziona la successiva progettazione del middleware stesso.

Sono stati valutati differenti prodotti open source, disponibili sul mercato, da impiegarsi per realizzare il middleware, saranno però esposti, per brevità, solo i due che sono stati reputati i più indicati per lo scopo. I candidati sono *ØMQ* e *RabbitMQ*. Si è scelto di appoggiarsi a componenti di terze parti in quanto implementare l'architettura ex-novo avrebbe richiesto un impegno eccessivo e i prodotti selezionati sono consolidati e supportati da una community attiva e numerosa. I

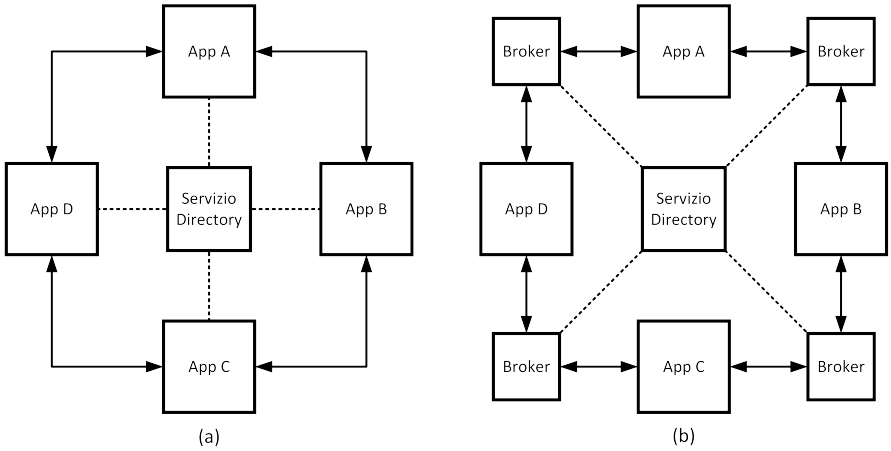


Figura 4.2.: (a) Architettura brokerless con servizio di directory. (b) Architettura a broker distribuito.

due prodotti esaminati differiscono principalmente per la diversa natura: RabbitMQ, è un middleware message-oriented, ØMQ, una libreria di messagging molto performante su cui poter sviluppare un middleware messag-oriented. Diamo una panoramica delle caratteristiche dei due prodotti individuati:

- **RabbitMQ** [16] è un middleware message-oriented open source, che implementa lo standard *Advanced Message Queuing Protocol*[1] (AMQP). Il server di RabbitMq è implementato in Erlang e sviluppato sull'*Open Telecom Platform* framework per il clustering e il failover, e implementa il modello a broker. RabbitMq è supportato da *SpringSource* una divisione *VMWare* e rilasciato sotto Mozilla Public License. È compatibile con la maggior parte dei sistemi operativi, offre binding per i principali linguaggi di programmazione e gateway per i protocolli HTTP, STOMP e MQTT. Offre poi una piattaforma per plug-in addizionali personalizzabili, con una collezione predefinita di plug-in che include: uno "Shovel" plugin che si occupa di replicare i messaggi da un broker all'altro, un "Federation" plug-in che abilita uno scambio efficiente di messaggi da un broker all'altro e un "Management" plug-in che abilita il monitoraggio e il controllo di cluster di broker.

[1] AMQP è un standard aperto che definisce un protocollo di livello applicazione per middleware message-oriented. Le caratteristiche definite dallo standard sono message orientation, queuing, routing (incluso point-to-point e publish-subscribe), affidabilità e sicurezza [17]. AMPQ garantisce l'interoperabilità di sistemi che rispettino tale standard definendo il formato dei dati che vengono inviati nella rete come flusso di byte.

- **ØMQ** [18] è una libreria asincrona ad alte prestazioni di messaging utilizzata per creare applicazioni concorrenti, scalabili e distribuite. Fornisce una coda di messaggi, ma a differenza dei middleware message-oriented, può essere eseguita senza un broker dedicato. È sviluppato da *iMatrix Corporation* (che ha collaborato attivamente nella definizione dello standard AMQP), rilasciato sotto licenza LGPLv3, compatibile con la maggior parte dei sistemi operativi e sono disponibili binding, sviluppati da terzi, per i più popolari linguaggi di programmazione. Fornisce socket che trasportano messaggi su diversi protocolli di comunicazione come TCP, PGM, IPC e ITC. È possibile collegare socket secondo pattern molti a molti come fan-out/fan-in, pubblish-subscribe, task distribution e request-reply. Offre prestazioni molto elevate in throughput e una bassa latenza. La persistenza dei messaggi non è offerta ma può essere implementata a livelli più alti.

Nel valutare le due tecnologie presentate ci si è basati sulle prestazioni offerte, in particolare sui tempi di latenza, e sulla flessibilità offerta per realizzare il framework, in quanto entrambe sono soluzioni abbastanza consolidate che mettono a disposizione binding per un buon numero di linguaggi di programmazione e una community attiva e numerosa. La scelta è ricaduta su ØMQ in quanto offre una maggiore flessibilità nel realizzare un middleware message-oriented su cui basare il framework, prestazioni più elevate in particolare una latenza più bassa e permette di realizzare un MOM a broker distribuito. Di contro non fornisce un middleware pronto all'uso ma solo una libreria, il ché significa che sarà necessario sviluppare un middleware per supportare la comunicazione dei componenti all'interno del framework che si andrà a realizzare.

4.1.1.3. ØMQ

Sarà ora analizzato in maggior dettaglio ØMQ, per poter comprendere le successive scelte implementative. La libreria mette a disposizione una serie di socket la cui vita può essere suddivisa in quattro parti, come nel caso dei socket BSD:

- Creazione e distruzione dei socket;
- Settaggio e controllo delle configurazioni;
- Inserimento del socket all'interno di una topologia creando connessioni entranti o uscenti;
- Utilizzo dei socket per trasportare dati inviando e ricevendo messaggi su di essi.

Come già accennato in precedenza ØMQ offre differenti tipologie di socket per implementare differenti pattern di comunicazione:

- Socket per pattern *request-reply*: in questo pattern una delle parti effettua una richiesta e l'altra invia una risposta a tale richiesta. Per questo pattern sono disponibili quattro differenti socket:
  - *REQ*: utilizzato per inviare richieste e ricevere risposte, permettendo il funzionamenti con la sola sequenza ciclica di invio e ricezione. Ogni richiesta è inviata selezionando un destinatario con modalità round-robin e ogni risposta corrisponde all'ultima richiesta effettuata.
  - *REP*: utilizzato per rispondere alle richieste, permettendo il funzionamento con la sola sequenza ciclica di ricezione e invio. Ogni risposta è inviata al rispettivo mittente che ha effettuato la richiesta.
  - *DEALER*: utilizzato per estendere il pattern request-reply. Ogni messaggio è inviato scegliendo un peer connesso con un algoritmo round-robin.
  - *ROUTER*: utilizzato per estendere il pattern request-replay. Permette l'invio di un messaggio a uno specifico socket di destinazione, il quale deve sempre essere specificato nell'invio del messaggio.
- Socket per pattern *publish-subscribe*: con questo pattern si può effettuare una distribuzione dei dati. Vi è un publisher che pubblica dei dati categorizzati in canali e dei subscriber che effettuano sottoscrizioni a canali specifici, ricevendo solo i messaggi appartenenti a canali cui è stata effettuata la sottoscrizione. I messaggi vengono ricevuti dal subscriber solo se questo è attivo nel momento in cui è effettuato l'invio. Sono disponibili due socket per questo pattern:
  - *PUB*: utilizzato dai publisher per distribuire dati. I messaggi sono distribuiti a tutti i peer connessi.
  - *SUB*: utilizzato dai subscriber per sottoscriversi ai canali su cui i dati sono distribuiti dai publisher e riceve tutti i messaggi distribuiti da quest'ultimi, corrispondenti alle sottoscrizioni.
- Socket per pattern a *pipeline*: questo pattern permette di distribuire e collezionare dati. Sono disponibili due socket:
  - *PUSH*: utilizzato dai nodi della pipeline per inviare messaggi ai nodi successivi. I messaggi sono inviati secondo lo schema round-robin ai peer connessi.
  - *PULL*: utilizzato dai nodi della pipeline per ricevere messaggi dai nodi precedenti.

- Socket *exclusive pair*: questo pattern permette la comunicazione esclusiva tra due socket. È disponibile un solo socket:
  - *PAIR*: può connettersi con un solo peer per volta.

Messaggi ØMQ utilizza messaggi che possono essere multipart, dove i frame variano a seconda del diverso pattern di comunicazione utilizzato. I messaggi multipart sono utilizzati da diversi socket, per inserire frame contenenti identità di socket necessari all'indirizzamento dei messaggi. Possono anche essere utilizzati per inserire informazioni arbitrarie gestite dall'applicazione.

Quando un REQ o un ROUTER invia un messaggio inserisce un frame in testa a questo, contenente la propria identità. In questo modo la risposta al messaggio iniziale può tornare al mittente anche attraversando più socket. Quando un REP riceve un messaggio rimuove i frame contenenti le identità e ritorna il messaggio originale, quando poi invia la risposta inserisce all'inizio del messaggio le identità precedentemente rimosse. Infine quando un ROUTER invia un messaggio a un DEALER non inserisce la propria identità.

Pattern di comunicazione Sono già stati descritti i pattern principali ottenibili con le diverse tipologie di socket, saranno ora presentate alcune loro implementazioni:

- *ROUTER-REQ*: questo pattern permette la comunicazione tra un ROUTER e più REQ. L'utilizzo di REQ è utile perché permette di segnalare quando il socket è "pronto" a ricevere un messaggio permettendo di gestire una richiesta per volta. Generalmente viene utilizzato per realizzare broker, accoppiando questo pattern con altri (ad esempio DEALER-REP).

- *DEALER-REP*: questo pattern permette la comunicazione tra un DEALER e più REP. Viene utilizzato per il fatto che il REP risponde a una sola risposta per volta e accetta messaggi multipart di qualsiasi lunghezza. Utilizzato generalmente quando c'è necessità di indirizzare il messaggio a un REP specifico.

- *ROUTER-DEALER*: questo pattern permette la comunicazione asincrona tra un ROUTER e più DEALER, distribuendo i messaggi dal ROUTER al DEALER secondo un algoritmo definito dallo sviluppatore.

- *ROUTER-ROUTER*: questo pattern permette la comunicazione molti a molti tra ROUTER. Ogni comunicazione deve avvenire con indirizzamento, specificando il destinatario.

Sono poi possibili le implementazioni più semplici come *REQ-REP*, *PUB-SUB* e *PUSH-PULL*. È possibile combinare più pattern di comunicazione per realizzare quelli che in ØMQ vengono definiti *device* e spesso svolgono il compito di broker e possono contenere al loro interno code persistenti o meno. I device possono essere utilizzati anche per realizzare brige per i protocolli di trasporto, per esempio per passare traffico IPC su TCP, o esporre servizi di una rete locale all'esterno della rete.

4.1.2. Progettazione

Sarà ora esposta la progettazione del middleware message-oriented basato su un broker distribuito con un servizio di directory distribuito, basato sulla libreria di messaging ØMQ.

4.1.2.1. Vincoli, aspetti particolari o critici

Lo sviluppo deve avere come obiettivo ottenere componenti snelli e robusti, utilizzando la metodologia crash only ove possibile, evitando di introdurre single point of failure.

Nel caso in cui si verifichi un guasto su di un componente è accettabile, anche se preferibilmente evitabile, la perdita dei messaggi in transito su di esso in quell'istante, demandando ai servizi l'implementazione di meccanismi per garantire l'affidabilità della risposta alle richieste o la segnalazione del guasto. Deve essere inoltre fornita la possibilità di implementare sia approcci di tipo SOA e che a pipeline.

Quando ci riferiremo a un socket faremo riferimento a un socket ØMQ, specificheremo altrimenti nel caso in cui ci volessimo riferire ad altre tipologie di socket. Il middleware sarà installato su di una rete di macchine che identificheremo come nodi (i peer).

4.1.2.2. Topologia

Sarà ora presentata la topologia del framework e successivamente analizzati i vari componenti.

Le funzionalità offerte dall'applicazione saranno raggruppate in servizi, come ad esempio il servizio per interfacciarsi con il database a grafo Neo4j, il servizio per le notifiche o quello per le statistiche. I singoli processi che erogano tali servizi sono definiti worker. Per rispettare gli obiettivi di affidabilità e scalabilità è necessario che tali worker possono essere replicati e distribuiti su più nodi. Il worker di un servizio può effettuare richieste ad altri servizi. Per gestire tali richieste si utilizza una variante del modello a broker distribuito, dove vi è un broker per

ciascun servizio il quale smista le richieste da evadere tra le repliche del worker di tale servizio. Per la gestione degli indirizzi dei broker si utilizza un approccio a directory service distribuita in cui vi è una replica per ogni nodo. Queste repliche prendono il nome di Master Node Broker e svolgono anche funzioni di gestione del nodo. I MNB sono connessi tra di loro per scambiarsi informazioni relative ai servizi presenti sul nodo da loro gestito. I broker possono poi essere connessi ad altri peer broker dello stesso servizio che risiedono su altri nodi, in questo modo è possibile delegare richieste effettuate su di un nodo ad altri nodi per effettuare load balancing e per evadere richieste a servizi che non dispongono di worker attivi sul nodo in cui è effettuata la richiesta. La topologia che ne risulta è riportata in figura 4.3.

Analizziamo ora più in dettaglio i tre componenti della topologia del middleware.

Worker I servizi saranno erogati da worker, i quali possono essere distribuiti e replicati per raggiungere i requisiti di scalabilità e affidabilità. I worker potranno poi far uso di uno o più controller, i quali esporranno una più azioni invocabili su di essi. Un servizio può quindi essere considerato un aggregatore di controller. La suddivisione in controller permette di sviluppare servizi più snelli e semplici da mantenere.

Un worker viene detto *disponibile* o *pronto* quando è correttamente connesso a un broker e non sta eseguendo alcuna elaborazione associata a una richiesta.

Un servizio viene definito *attivo* su di un nodo quando è disponibile almeno un worker correttamente connesso al broker del servizio. Un worker può effettuare richieste a servizi differenti da quello di appartenenza. Per tale motivo ogni singolo worker avrà un modulo di comunicazione (CM) che permetterà di effettuare richieste ad altri servizi e di ricevere le richieste per il servizio di appartenenza.

Broker Il Broker si occupa di ricevere le richieste per un dato servizio, le quali possono essere evase sia in locale, dai worker a lui connessi, sia da worker remoti connessi tramite peer broker. Quando un broker riceve una richiesta questa viene affidata ad un worker locale disponibile o accodata all'interno del broker nel caso non vi siano worker disponibili. Nel caso il servizio sia attivo su altri nodi l'elaborazione della richiesta può essere delegata ad un altro nodo in base alla strategia di load balancing adottata, attraverso un peer broker di un nodo in qui tale servizio è attivo.

Un servizio viene definito *disponibile* se è attivo o se è connesso a almeno un peer broker di un nodo in cui tale servizio è attivo. Quindi su di un nodo possono essere evase solo richieste di servizi disponibili.

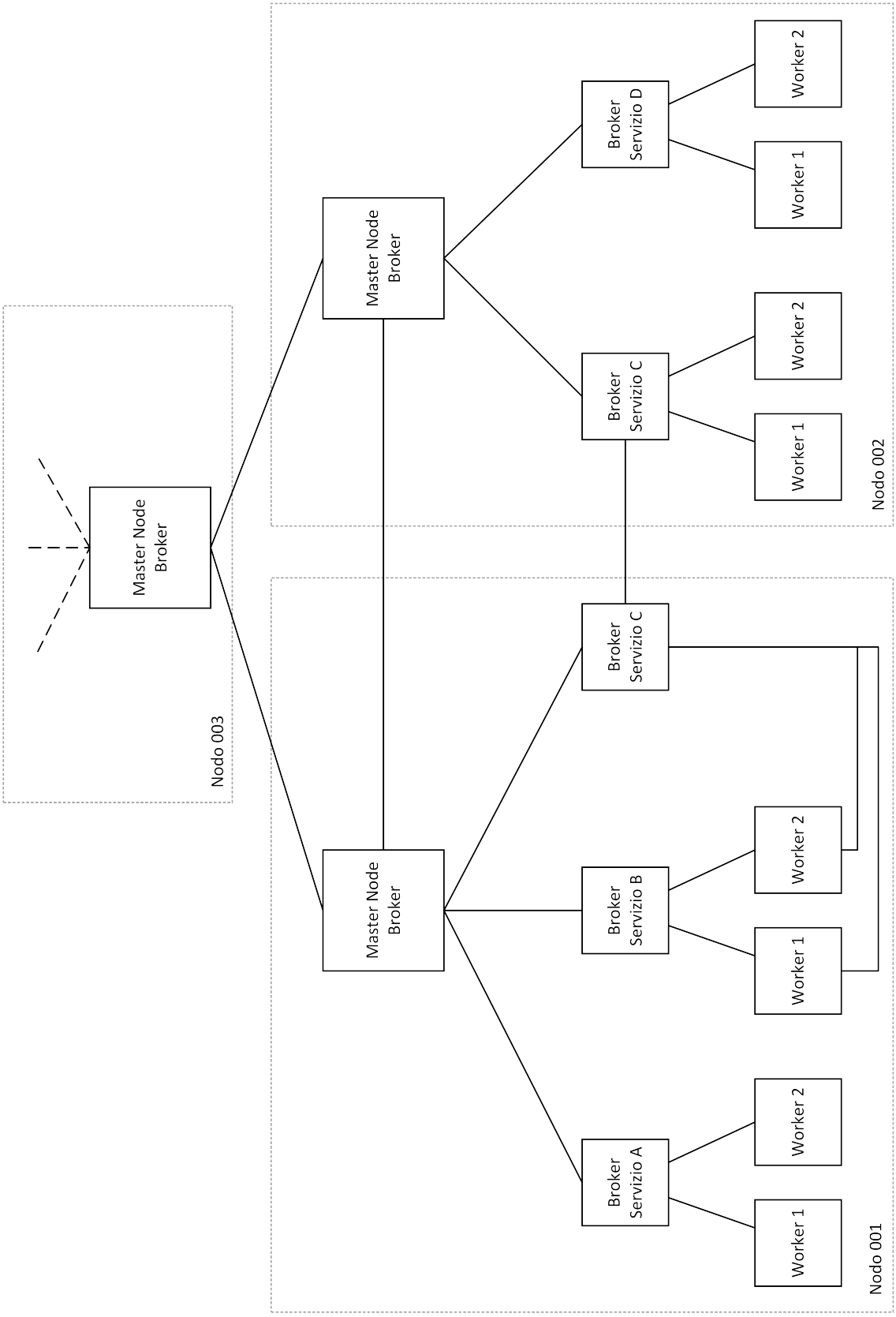


Figura 4.3.: Topologia componenti middleware

Master Node Broker Il Master Node Broker (MNB) rappresenta un "componente replica" della servizio di directory distribuito presente nel nodo. È presente un MNB per ogni nodo della rete. La sua funzione principale consiste nella gestione delle informazioni necessarie ai componenti, che interagiscono all'interno del framework, per instaurare connessioni tra di loro. Il MNB broker genera tali informazioni per i singoli componenti i quali poi li utilizzano per istanziare i socket e li rende disponili ai componenti che necessitano di istaurare una comunicazione. Integra poi al suo interno funzionalità per la gestione del nodo, come ad esempio l'attivazione di broker e worker per i singoli servizi del nodo. Si occupa poi di fornire le informazioni ai broker sui relativi peer broker di altri nodi.

Master Node Broker appartenenti a nodi differenti vengono identificati come *peer MNB*, i quali si scambiano messaggi sullo stato dei servizi attivi sui nodi.

Specifiche ØMQ della topologia Presentiamo ora i socket e i pattern identificati per supportare le comunicazioni tra i componenti. Possiamo dividere i socket utilizzati in tre categorie in base al tipo di comunicazioni che supportano: quelli per la comunicazione tra i broker e i worker locali, quelli per la comunicazione con l'MNB locale e quelli per la comunicazione con i peer remoti.

La comunicazione tra broker e worker locali avviene attraverso un pattern ROUTER-ROUTER + ROUTER-DEALER. Il worker che effettua la richiesta, svolgendo il ruolo del client, invia un messaggio utilizzando il ROUTER *backend* del suo modulo di comunicazione al ROUTER del broker del servizio a cui viene effettuata la richiesta. Questo socket, che prende il nome di *local_frontend*, riceve tutte le richieste per il dato servizio che provengono da worker locali. Il broker poi invia il messaggio, attraverso il ROUTER *local_backed*, al DEALER del modulo di comunicazione di uno dei worker disponibili. Nel caso in cui non vi sia alcun worker disponibile il messaggio viene inserito in una coda FIFO, la *local_queue*. Non appena un worker torna disponibile viene inviato il primo messaggio della coda. Una volta elaborata la risposta alla richiesta il worker invia il messaggio di risposta al ROUTER, *local_backend* del broker, tramite il DEALER che torna poi il messaggio al client, attraverso il local_frontend. Il *local_backend* ROUTER invia i messaggi contenenti le richieste ai DEALER dei worker connessi e riceve poi da quest'ultimi i messaggi di risposta.

È stato scelto di utilizzare un ROUTER per il socket *backend* sul CM del client poiché in questo modo è possibile inviare richieste, e ricevere le relative risposte, a un numero arbitrario di servizi tramite il ROUTER *front_end* dei loro broker. È invece stato scelto un DEALER per il frontend poiché è necessario collegarsi al solo ROUTER *local_backend* del broker del servizio di appartenenza e da la possibilità di gestire più richieste per volta, inviando appositi messaggi al broker per richiedere la successiva richiesta, permettendo di aumentare il parallelismo e

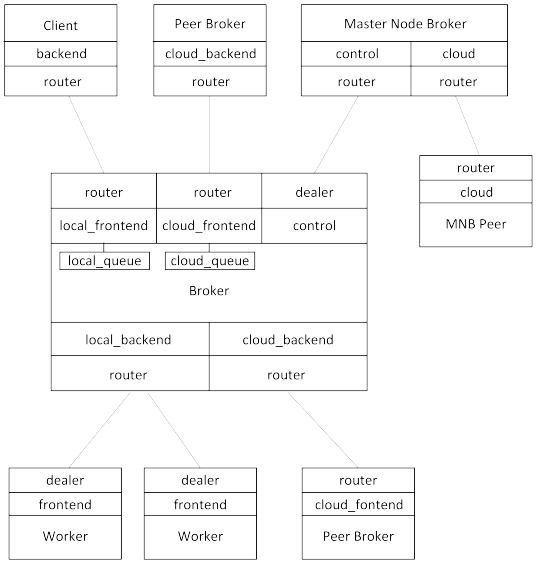


Figura 4.4.: Schema socket della topologia del middleware.

di sfruttare al meglio la gestione asincrona dell'I/O in quei linguaggi come Node.js dove è presente. Sul broker vengono impiegati ROUTER per poter indirizzare correttamente le richieste.

La comunicazione tra MNB e broker avviene attraverso un socket dedicato, *control*, di tipo ROUTER sul MNB e di tipo DEALER sul broker. Su questo canale vengono inviati messaggi di controllo per gestire i broker e per segnalare condizioni particolari al MNB.

Per delegare l'elaborazione di richieste a servizi attivi su altri peer vengono utilizzati i ROUTER *cloud_frontend* e *cloud_backend*. Il primo serve ad accettare le richieste provenienti da altri peer broker e inviare le relative risposte, il secondo, connesso al *cloud_frontend* degli altri peer broker della rete, è utilizzato per delegare le richieste ai peer remoti e ricevere le risposte associate.

Infine sono istanziate delle comunicazioni di tipo pubblish-subscribe per lo scambio dei messaggi tra MNB di nodi differenti. Nell'attivazione del SUB, dopo essersi connessi ai PUB degli altri MNB, è effettuata un sottoscrizione a un canale associato al nodo su cui risiede il MNB e a uno globale. In questo modo è possibile inviare messaggi a uno specifico MNB o a tutti, gestendo l'invio in modo più semplice rispetto all'utilizzo di un singolo socket di tipo ROUTER.

Forniremo ora un esempio del meccanismo di request-replay offerto dall'architettura presentata. Supponiamo che il servizio A effettui una richiesta al servizio

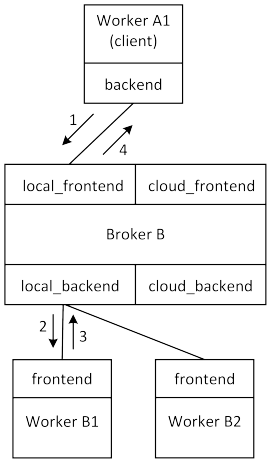


Figura 4.5.: Esempio request-repaly con approccio SOA.

B e che per entrambi i servizi siano attivi due worker sullo stesso nodo. Attraverso il worker A1 il servizio A effettua una richiesta al servizio B, inviando un messaggio tramite il ROUTER backend di A1 al ROUTER local_frontend del broker di B, il quale è connesso ai worker B1 e B2 entrambi disponibili. Il broker estrae il primo worker dalla coda e gli inoltra il messaggio di A1 attraverso il ROUTER local_backend. B1 riceve il messaggio sul DEALER frontend, elabora la richiesta e ritorna la risposta inviando un messaggio, sempre tramite il DEALER frontend, al local_backend del broker di B, il quale poi inoltra a sua volta il messaggio, tramite il local_frontend, al backend di A1. Questo esempio utilizza un approccio di tipo SOA.

Riprendendo l'esempio precedente, supponiamo ora che il servizio B sia disponibile ma non attivo sul nodo 001 che ospita il servizio A e che sia invece attivo sul nodo 002. In questo caso il broker di B presente su 001 sarà connesso al suo peer broker del nodo 002. Quindi una volta ricevuto il messaggio di A1 il broker di B (su 001) inoltra il messaggio tramite il suo ROUTER cloud_backend al ROUTER cloud_frontend del broker di B attivo sul nodo 002. Una volta elaborata la risposta alla richiesta tramite i worker a lui connessi il broker invia il messaggio di risposta al suo peer broker tramite il cloud_frontend. Infine ricevuto il messaggio sul cloud_frontend il broker di B su 001 lo inoltra ad A1 sempre tramite il local_frontend.

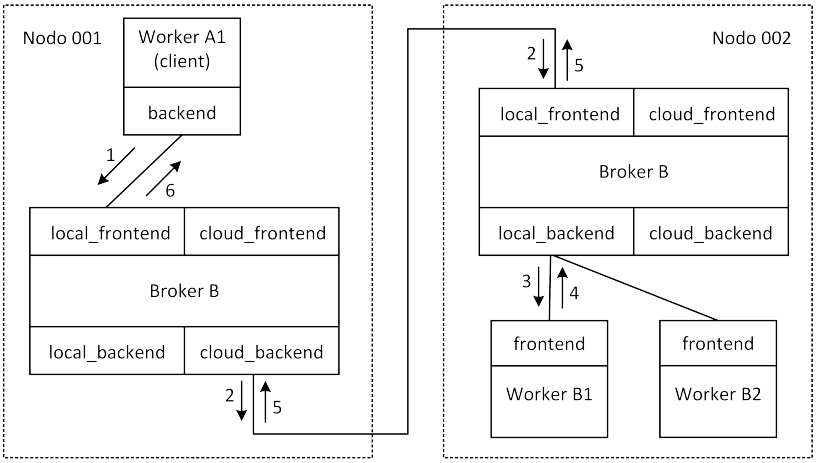


Figura 4.6.: Esempio request-reply con approccio a pipeline.

Osservazioni Grazie alla topologia presentata è possibile gestire i servizi in modo gerarchico, dove più servizi sono raggruppati sotto un macro-servizio, dove uno o più worker del macro-servizio i quali in base alla richiesta ricevuta smistano le richieste verso il broker dello specifico servizio.

Con un approccio simile si può anche limitare il numero di connessioni uscenti da un nodo realizzando un servizio che smista tutte le richieste entranti in quel nodo. Utile nel caso in cui si desideri limitare il numero di connessioni attive tra nodi distribuiti tra differenti data center, anche se questo può rappresentare un collo di bottiglia analogo a quello che si verifica in sistemi a singolo broker.

La topologia presentata permette di estendere il concetto di nodo a più macchine, le quali fanno riferimento allo stesso Master Node Broker. In questo modo è possibile allocare broker e/o worker in macchine differenti, per aumentare la scalabilità della soluzione, senza aumentare la complessità della topologia.

La topologia permette un iterazione tra i servizi sia di tipo SOA sia a pipeline, più efficiente in termini di carico sui broker e di tempo di risposta. Nell'interazione di tipo SOA se una richiesta per essere evasa deve far uso di più servizi, dove il primo servizio che elabora la richieste esegue una richiesta a un altro servizio e questo si ripete ricorsivamente, si ha che le risposte devono tornare al primo servizio che ha iniziato l'elaborazione per inviare poi la risposta al client. Se non vengono eseguite elaborazioni sulle risposte fornite dai servizi nei vari passi della ricorsione si ha un inutile spreco di banda e tempo per far tornare la risposta dall'ultimo worker utilizzato fino al client. In un approccio a pipeline invece è possibile inviare la risposta al client dal servizio che elabora l'ultima richiesta,

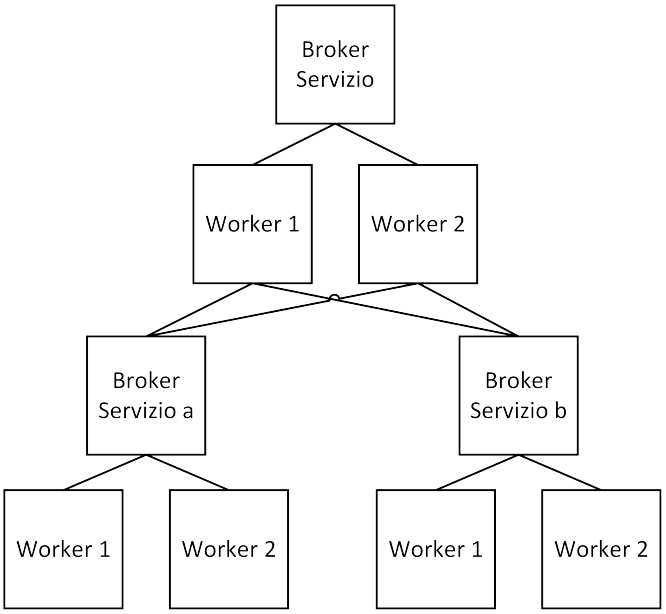


Figura 4.7.: Esempio topologia con servizi gerarchici.

diminuendo il traffico e i tempi di risposta al client. Entrambi gli approcci sono implementabili sull'architettura fin qui esposta, la scelta implementativa è lasciata allo sviluppatore per offrire un approccio flessibile allo sviluppo.

Messaggi Analizzeremo ora il formato dei messaggi utilizzato dal middleware. Come esposto in 4.1.1.3, ØMQ utilizza messaggi multipart in cui generalmente l'ultimo frame contiene il messaggio iniziale e i frame precedenti contengono l'identità dei socket attraversati durante l'indirizzamento del messaggio. Il frame relativo al messaggio originale è trattato come blob di dati binari che può quindi contenere qualsiasi tipo di informazioni. Vi sono due differenti tipi di messaggi che vengono trasmessi dal middleware: messaggi dati e messaggi di controllo. I messaggi dati trasportano le richieste verso i servizi, quelli di controllo sono utilizzati da broker, worker e MNB per comunicare tra loro. I messaggi si differenziano poi in richieste e risposte. I messaggi dati di richiesta possono contenere un campo relativo all'utente che ha effettuato la richiesta, che può contenere dati necessari a elaborare la risposta.

I messaggi dati sono serializzati per mantenere la compatibilità tra differenti piattaforme hardware, sistemi operativi e linguaggi di programmazione. Il formato di serializzazione predefinito è il JSON, ma possono essere utilizzati altri formati sia in forma testuale che binaria. Le richieste contengono l'identificativo

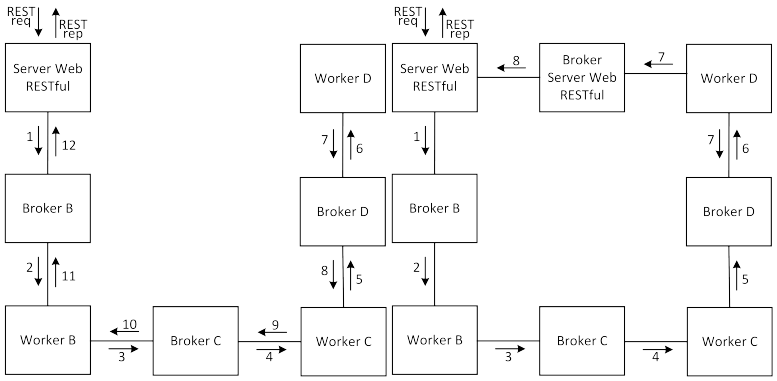


Figura 4.8.: (a) Interazione di tipo SOA. (b) Interazione di a pipeline.

della richiesta, il *reqId*, del servizio, dell'azione e opzionalmente del controller. Infine contengono un numero arbitrario di parametri specifici dell'azione invocata. La risposta invece contiene tre campi: l'identificativo della richiesta a cui è associata, un campo contenente gli eventuali errori verificatisi nell'elaborazione della risposta e uno contenente i risultati della risposta. L'identificativo della richiesta è un intero positivo progressivo che identifica in modo univoco la richiesta inviata da un componente, sia esso un worker, un MNB o qualsiasi altro componente che necessiti di comunicare all'interno del framework.

Per quanto riguarda i messaggi di controllo, la richiesta è composta da tre campi: l'identificativo progressivo della richiesta (che in questo caso è opzionale), il tipo di richiesta e un campo opzionale che contiene i parametri della richiesta. Anche in questo i caso i messaggi di risposta contengono l'identificativo associato alla richiesta e due campi, uno contente gli eventuali errori verificatisi nell'elaborazione della risposta e uno contenente i risultati della risposta, i quali però sono opzionali. I codici identificativi delle richieste sono stati definiti nella specifica riportata in A.

Azioni Saranno ora riportate le azioni invocate sul Master Node Broker relative ai servizi:

- startService permette l'attivazione del broker relativo a un servizio e a seconda della configurazione rende il servizio attivo o disponibile.
- activeService compie le operazioni necessarie all'attivazione del servizio.
- makeAvailableService compie le operazioni necessarie a rendere disponibile il servizio connettendo il relativo broker ai peer broker disponibili.

- stopService disattiva il servizio , disattivando anche i relativi broker e worker. Se il servizio era attivo l'evento viene comunicato agli altri peer.
- addWorker compie le operazioni necessarie per attivare uno o più worker di un servizio.
- removeWorker compie le operazioni necessarie per disattivare uno o più worker di un servizio, specificando il numero di worker o gli identificativi dei singoli worker che si desidera disattivare.
- addPeer compie le operazioni necessarie per aggiungere un peer remoto al broker.
- removePeer compie le operazioni necessarie per rimuovere un peer remoto dal broker.

Saranno ora riportate le azioni invocate sul Master Node Broker relative al MNB stesso:

- init effettua le operazioni necessarie a inizializzare il Master Node Broker, caricare la configurazione del nodo, attivare i servizi ed effettuare la peer discovery attraverso le altre azioni.
- close questa azione disattiva tutti servizi attivi e disponibili sul nodo (chiudendo anche i relativi broker) e chiude le connessioni con gli altri peer, informandoli prima della disattivazione dei servizi e del MNB stesso.
- addPeer compie le operazioni necessarie per aggiungere un peer al MNB.
- removePeer compie le operazioni necessarie per rimuover un peer dal MNB.
- publishNodeInfo distribuisce informazioni relative al nodo e ai servizi attivi su di esso agli altri peer.
- updatePeersInfo aggiorna le informazioni, memorizzate sul nodo, relative ai peer e ai servizi attivi su di essi.

Strategie di fault tollerance Una delle principali strategie adottate per aumentare l'affidabilità del middleware è l'introduzione di meccanismi di heartbeating tra i componenti. Infatti tra broker e worker avviene un continuo heratbeating che permette al broker di conoscere lo stato del worker e se questo non risponde per un numero prefissato di volte non è più considerato come worker attivo e l'anomalia viene segnalata al Master Node Broker. Viene eseguita la stessa procedura anche tra peer broker e tra MNB e broker. Quando è segnalato un worker o broker in guasto, il MNB provvederà a riavviarlo.

L'assunzione che se un componente non risponde all'heartbeating significa che è in fault è ammissibile in quanto, come vedremo successivamente, viene utilizzata la metodologia crash only. Assunzione rafforzata poi da una progettazione di componenti snelli e robusti, caratteristiche che limitano il verificarsi di condizioni di guasto e facilitano l'eventuale individuazione degli errori. È per questo motivo che all'interno dei componenti dedicati al trasporto dei messaggi sono state implementate solo le funzioni essenziali. Altre funzionalità possono essere aggiunte con l'implementazione di servizi specifici, come nel caso del load balancing.

Interfaccia controllo e monitoraggio Per la gestione del middleware sarà necessario esporre un'interfaccia attraverso cui poter inviare comandi al MNB. I comandi devo poter essere inviati sia da un operatore umano, attraverso strumenti di vario genere (CLI o grafici), sia da servizi attivi sul nodo e da altri MNB. La possibilità di inviare comandi al MNB da parte dei servizi, rende possibile l'implementazione di funzionalità, come load balancing o gestione ed esecuzione di operazioni programmate, in servizi dedicati così da non aggiungere complessità ai componenti del middleware, ottenendo un sistema più flessibile, personalizzabile e robusto. La possibilità di inviare comandi da MNB ai suoi peer permette di gestire l'intero framework distribuito, collegandosi a un singolo nodo, garantendo così maggiore manutenibilità del sistema e una sua più semplice gestione.

In questo lavoro non sono stati approfonditi aspetti legati alla sicurezza nell'invio dei comandi, aspetto molto importante che verrà affrontato nei lavori successivi, in quanto in una prima fase il framework sarà distribuito su macchine che comunicheranno su reti private.

4.1.2.3. Interfaccia controllo e monitoraggio

Un obiettivo molto importante che si vuole perseguire con lo sviluppo di questo framework è l'affidabilità. Per ottenerla è necessario attuare delle strategie per fa si che in caso di guasto di un componente, questi torni il prima possibile a svolgere le sue funzionalità. A tale scopo si è sviluppato un sistema di gestione dei processi del framework, il Long Running Process Manager (LRPM), il quale tiene traccia dei processi attivi, li monitora e in caso di crash li riattiva nel minor tempo possibile.

Il sistema è composto da due componenti: un server, per la gestione dei processi e loro il monitoraggio, e un client che può richiedere l'attivazione o la disattivazione dei processi. La comunicazione avviene tramite socket (ROUTER lato server e DEALER lato client), per permettere la gestione di processi del framework anche su macchine remote nel caso in cui un nodo sia composto da più macchine, in questo caso sarà necessario attivare un server per ogni macchina, mentre il client potrà essere uno unico.

Le azioni che il server dovrà offrire sono le seguenti:

- startProcess: attiva un processo e memorizza le informazioni fornite associandole al processo.
- stopProcess: disattiva un processo.
- processInfo: restituisce le informazioni relative al processo precedentemente fornite dal client.
- checkProcess: verifica se il processo specificato è attivo.

Queste azioni saranno invocabili attraverso una libreria accessibile lato client, utilizzata dal Master Node Broker per attivare i processi associati ai servizi, come broker e worker.

Anche in questo sistema viene utilizzato un meccanismo di heartbeating dal client al server per verificare costantemente che il server sia attivo e in caso contrario riattivarlo o segnalare la condizione di guasto.

La separazione del sistema in client e server è necessaria per far si che nel caso in cui una delle parti vada in guasto l'altra continui il suo funzionamento. Nel verificarsi di un crash del MNB, quando questi viene riattivato e richiede al server LRPM i processi attivi da lui gestiti, memorizzando poi queste informazioni. Nel caso opposto, quando va in crash il server LRPM, la libreria lato client utilizzata sul MNB, non ricevendo più risposta al suo heartbeating riattiva il server LRPM. Questi poi richiede al MNB quali sono i processi attivi da lui memorizzarti e verifica che siano effettivamente in esecuzione, in caso contrario lo segnala al MNB.

Come già accennato precedentemente, il monitoraggio dei processi e la loro immediata riattivazione in caso di crash è possibile perché è impiegata la metodologia crash only nello sviluppo delle componenti del framework. I software sviluppati con metodologia crash only, sono software che gestiscono i fallimenti semplicemente riavviandosi, senza alcuna sofisticata procedura di recovery [19]. Tali software possono eseguire microreboot a stati definiti senza l'intervento dell'utente, in quanto gestiscono la procedura di ripristino in seguito a un guasto allo stesso modo in cui avviene normalmente l'avvio. Questo facilita anche l'individuazione di errori all'interno del codice per la gestione dei guasti, in quanto è lo stesso codice che gestisce l'avvio.

Nel caso specifico del framework sottolineiamo che i worker gestiscono richieste stateless quindi posso essere riavviati senza alcun problema, al più la richiesta che ha portato lo stato di guasto sarà persa. Lo stato dei broker è rappresentato dalla coda di richieste non ancora distribuite ai worker ed è stata fatta l'assunzione che sia accettabile perdere alcune richieste in caso di guasti data la loro natura,

è quindi possibile riavviare semplicemente entrambe le tipologie di componenti in caso di guasto. Il Master Node Broker invece ha uno stato rappresentato dalla configurazione del nodo, in cui sono memorizzate anche le informazioni relative ai servizi. Tale configurazione sarà salvata in modo persistente e validata all'avvio. In questo modo è possibile riavviare il MNB broker in ogni momento senza comportare rilevanti ripercussioni sul sistema.

4.1.2.4. Aggiornamento servizi senza downtime

Da quanto finora esposto è possibile definire una procedura che permetta l'aggiornamento dei servizi senza downtime. Questo è molto importante in sistemi web based perché, questi sistemi, sono aggiornati frequentemente sia per la correzione di bug sia per il rilascio di nuove funzionalità o migliorie. Per aggiornare servizi implementati in Java, Node.js o altri linguaggi in cui vi è un processo in esecuzione che elabora le richieste è necessario riavviare il processo, operazione che può quindi comportare un disservizio. Grazie al gestore dei processi è possibile effettuare l'aggiornamento, senza dowtime, attivando un numero di processi (aggiornati) pari a quelli già attivi e successivamente disattivare i processi precedentemente attivi, che utilizzano versioni precedenti. Nel caso in cui il numero di worker attivi sia elevato si può gestire un aggiornamento progressivo aggiornando un piccolo numero di worker per volta.

4.1.2.5. Aggiornamento servizi senza downtime

Da quanto finora esposto è possibile definire una procedura che permetta l'aggiornamento dei servizi senza downtime. Questo è molto importante in sistemi web based perché, questi sistemi, sono aggiornati frequentemente sia per la correzione di bug sia per il rilascio di nuove funzionalità o migliorie. Per aggiornare servizi implementati in Java, Node.js o altri linguaggi in cui vi è un processo in esecuzione che elabora le richieste, è necessario riavviare il processo, operazione che può quindi comportare un disservizio. Grazie al gestore dei processi è possibile effettuare l'aggiornamento, senza dowtime, attivando un numero di processi (aggiornati) pari a quelli già attivi e successivamente disattivare i processi precedentemente attivi, che utilizzano versioni precedenti. Nel caso in cui il numero di worker attivi sia elevato si può gestire un aggiornamento progressivo aggiornando un piccolo numero di worker per volta.

4.1.2.6. Load balancing

Il load balancing sarà gestito in due componenti separate. Una componente, integrata nel broker, gestisce il load balancing delle richieste a uno stesso servizio

tra i nodi in cui tale servizio è attivo. Inizialmente sarà implementata una semplice strategia in cui l'invio di richieste a peer broker, da parte di un broker, è regolato da un rapporto tra le richieste elaborate in locale e quelle elaborate in remoto. In particolare viene delegato un blocco di richieste a un peer remoto dopo aver aggiunto alla coda un numero prefissato di richieste, scegliendo il peer remoto utilizzando l'algoritmo round-robin. Se vi è almeno un worker disponibile la richiesta viene sempre elaborata in locale, in quanto la possibilità di delegare richieste a peer avviene solo per le richieste accodate, questo per diminuire il tempo di risposta. Successivamente sarà implementata un strategia più complessa ed efficace basata su valutazioni della velocità di elaborazione delle richieste sui peer remoti, sui tempi di latenza tra peer, sulla dimensione della coda sui peer broker e altre informazioni valutate dinamicamente, che permetteranno di individuare il peer migliore a cui delegare le richieste. Tale strategia necessità di disporre di informazioni ottenute attraverso un sistema di monitoraggio del framework che in questa prima fase di sviluppo non sarà trattato, per questo motivo si utilizzerà la prima strategia. Ricordiamo inoltre che questa prima versione del framework, pur offrendo la possibilità di interconnettere nodi di data center differenti, non approfondisce tale tematica, supponendo di comunicare su reti gigabit a bassa latenza e con macchine prestazionalmente simili all'interno dello stesso data center.

La seconda componente di load balancing è implementata attraverso un servizio dedicato che raccoglie informazioni su tempi di elaborazione delle richieste ai vari servizi e richiede l'attivazione o la disattivazione di worker, al MNB, così da diminuire il tempo di risposta nel caso di carichi elevati o di liberare risorse (worker non necessari) nel caso il carico diminuisca. Per implementare tale strategia è necessario aggiungere un piccolo componente al CM dei worker che raccoglie dati grezzi sui tempi di elaborazione alle richieste effettuate ad altri servizi e poi invii periodicamente tali informazioni al servizio di load balancing.

Il servizio di load balancing dovrà comunicare con il MNB attraverso l'interfaccia di comando, per richiedere l'attivazione o la disattivazione di worker per un dato servizio.

Il servizio monitorerà la media dei tempi di risposta per ciascun servizio e periodicamente valuterà se questa supererà la soglia superiore o inferiore di controllo. Sarà fissato un numero massimo di volte consecutive in cui la media può superare una soglia, oltre il quale sarà richiesta l'attivazione o la disattivazione di worker.

Il servizio utilizzerà la seguente formula per calcolare periodicamente la media μ_i del tempo di risposta alle richieste di un dato servizio, basandosi sui tempi di risposta ottenuti dai moduli di comunicazione:

$$\mu_i = \mu_{i-1} * a + \frac{(1-a)}{n} \sum_{t_t \in T}$$

$$0 \leq a \leq 1$$

dove μ_{i-1} è la media precedente, a il peso della media passata, T l'insieme dei tempi di risposta ottenuti dai worker successivamente al precedente calcolo della media e n la sua cardinalità. Il peso a della media passata è calcolato con la seguente formula:

$$a = \frac{\mu_{Th_{i-1}}}{\mu_{Th_{i-1}+Th_i}}$$

dove $\mu_{Th_{i-1}}$ è il throughput medio passato, T_i è il throughput medio attuale. Il valore di a dipende quindi dalla variazione del throughput attuale rispetto quello medio precedentemente calcolato.

Inizialmente la soglia inferiore è impostata a 0 e pari al doppio della prima media calcolata quella superiore. In seguito alla richiesta di attivazione di un nuovo worker le soglie sono aggiornate in base alla seguenti formule:

$$s_{sup_t} = 2 * s_{sup_{t-1}}$$

$$s_{inf_t} = s_{inf_{t-1}} + \frac{s_{sup_{t-1}} - s_{inf_{t-1}}}{2} = \frac{s_{sup_{t-1}} - 3s_{inf_{t-1}}}{2}$$

Mentre in seguito alla richiesta di disattivazione di un worker le soglie sono aggiornate in base alla seguenti formule:

$$s_{sup_t} = s_{sup_{t-1}} - \frac{s_{sup_{t-1}} - s_{inf_{t-1}}}{2} = \frac{s_{sup_{t-1}} + s_{inf_{t-1}}}{2}$$

$$s_{inf_t} = \frac{s_{inf_{t-1}}}{2}$$

Dove s_{sup_t}e s_{inf_t} sono i nuovi valori di soglia, mentre $s_{sup_{t-1}}$e $s_{inf_{t-1}}$ sono quelli precedenti.

4.1.2.7. Sicurezza

In questa prima versione del framework saranno tralasciate questioni relative alla sicurezza del framework, come ad esempio la cifratura asimmetrica della comunicazione tra MNB e peer broker, le quali saranno affrontate nei successi lavori.

4.2. Altri componenti del framework

Il framework dovrà fornire altri componenti come ad esempio web server. Questi saranno due, uno che riceverà richieste REST e un secondo che utilizzerà una connessione client-server per implementare meccanismi di notifica e diminuire il

tempo di risposta delle richieste (rispetto al server REST). I due server utilizzeranno un meccanismo di routing che permetterà data una richiesta, di individuare servizio, controller e action a cui inoltrarla.

I due server potranno far uso di un meccanismo per la gestione delle sessioni che potrà essere esteso dallo sviluppatore.

Saranno poi fornite le specifiche per la realizzazione dei moduli di comunicazione e per le interfacce dei metodi esposti dai controller. Sarà così possibile definire controller e worker in differenti linguaggi di comunicazione. Sarà inizialmente sviluppato il modulo di comunicazione in Node.js e Java.

5. Implementazione e integrazione del framework

Verranno ora illustrati dettagli implementativi e successivamente l'integrazione del framework nel software lato server utilizzato dal servizio Uniants.

5.1. Scelte implementative

Si è scelto di sviluppare, dove possibile, le componenti del framework in Node.js, per sfruttare le possibilità offerte da tale linguaggio nella gestione asincrona dell'I/O e nello sviluppo di applicazione web con necessità di gestire un elevato numero di richieste. Come esposto precedentemente si è scelto di basarsi sulla libreria di messagging ØMQ per sviluppare il middleware message-oriented.

5.2. Middleware

5.2.1. Socket

Dopo aver eseguito alcuni test preliminari su semplici prototipi, si è evidenziata una carenza prestazionale nell'invio dei messaggi della libreria ØMQ per Node.js, dovuta a operazioni di cast dei dati inviati nel binding alla libreria C. In particolare si è evidenziato che un notevole sforzo computazionale è svolto nella conversione dei dati in *Buffer*, operazione necessaria per poter inviare tipi generici di dati. Per ottenere prestazioni migliori è possibile modificare il binding della libreria e limitare i dati inviati a array di stringhe. Questo è ammissibile se si serializzano i dati inviati in forma testuale, ma limita la possibilità di utilizzare strategie serializzazione dei dati in forma binaria. Ulteriori studi sul binding dalla libreria C di ØMQ associati alla limitazione proposta possono migliorare notevolmente le prestazioni in invio dei messaggi nel middleware da parte di componenti Node.js. Le prestazioni in ricezione sono paragonabili a quelle ottenibili da applicazioni C.

Nelle connessioni tra i vari componenti della rete si è scelto di utilizzare socket IPC (Inter-Process Comunication) per le comunicazioni interne alla stessa macchina e il TCP per quelle tra macchine differenti. La scelta di utilizzare socket IPC, invece che TCP utilizzando l'indirizzo di loopback, non è dettata

da motivazioni prestazionali, in quanto entrambe le soluzioni offrono prestazioni comparabili, ma piuttosto perché grazie a questa scelta non si utilizzano porte TCP. Questo permette una più semplice gestione del sistema, impedisce l'accesso dall'esterno a tali socket ed è possibile definire permessi specifici sull'utilizzo dei socket da parte degli utenti del sistema, allo stesso modo di come sono definiti i permessi sui file in ambienti UNIX.

È stata definita una specifica per l'identità dei socket, che impone una lunghezza di 12 caratteri e la seguente forma:

- 3 caratteri identificativi del nodo, l'id numerico del nodo con eventuale padding;
- 3 caratteri identificativi del servizio (mnb nel caso del MNB);
- 3 caratteri identificativi del componente del servizio (es. W01,W02 worker o B01 broker);
- 3 caratteri identificativi del socket (es. local frontend LF1).

È stata imposta una lunghezza massima di 35 caratteri (comprensivi di prefisso "ipc://") per gli indirizzi IPC che corrisponde alla massima lunghezza di un indirizzo TCP (con IPv6) per la libreria ØMQ.

Tutti gli indirizzi e le identità assegnate ai socket sono gestite a runtime dal MNB il quale li assegna ai componenti al loro avvio, dopo averli generati.

5.2.2. Broker

Dall'osservazione effettuata sulla libreria Node.js per ØMQ si è deciso di implementare i broker in C. Data l'importanza e la criticità del componente all'interno del middleware si è preferito eliminare qualsiasi perdita prestazionale legata al binding della libreria C di ØMQ ad altri linguaggi.

Per rendere flessibile l'impiego del broker la configurazione dei socket viene definita a runtime. All'avvio del broker sono passati come parametri l'indirizzo e l'identità del socket del MNB, successivamente quando è attivato il canale di controllo con il MNB viene richiesta la configurazione dei socket . Questo permette la distribuzione di un nodo su più macchine e una gestione più flessibile degli indirizzi assegnati ai componenti.

I worker attivi vengono inseriti in una coda FIFO e ad ogni richiesta viene assegnato quello in testa. Se la coda è vuota la richiesta viene anch'essa inserita in una coda FIFO e non appena si rende disponibile un worker la richiesta in testa alla coda viene inviata a tale worker. Richieste locali e remote sono gestite in due code separate.

Viene effettuato un heartbeating bidirezionale con worker, peer broker e con il MNB. Nel caso dei worker, l'heartbeating serve anche per aumentare l'affidabilità della connessione, in quanto permette a un worker di connettersi ad un broker che viene avviato dopo di lui senza l'intervento del MNB, azione necessaria nel caso di un riavvio del broker e che permette di velocizzare l'avvio dei servizi, poiché è possibile avviare contemporaneamente sia il broker che i worker, che altrimenti devono essere attivati sequenzialmente. Sul broker sono stati impostati i seguenti valori di heartbeating per i worker: frequenza 1s e massimo numero di mancate risposte che portano ad un riavvio pari a 5. Nel caso in cui un worker non mandi l'heartbeat per più di 5 volte viene segnalato al MNB e viene eliminato dalla coda dei worker disponibili se presente.

Viene utilizzato inoltre un monitoraggio sul tempo di risposta dei worker, per individuare situazioni di stallo in worker sviluppati in linguaggi di programmazione event driven, dove è possibile che l'heartbeating funzioni correttamente, ma che il worker sia in stallo in seguito all'elaborazione di una richiesta, che attende il risultato da un operazione di I/O. Se il tempo di risposta supera i 180s viene richiesto il suo riavvio. Nel caso di servizi che richiedano lunghi tempi di elaborazione delle richieste è possibile definire tempi differenti compilando un nuovo broker. Se fosse presente un solo worker e questi fosse in stallo, la situazione verrebbe segnalata anche attraverso il servizio di load balancing, che notando la media dei tempi di risposta salire, richiederebbe l'attivazione di uno o più worker.

5.2.3. Master Node Broker

Come è stato esposto precedentemente il Master Node Broker è il componente che ha il compito di gestire i servizi presenti sul nodo e scambiare informazioni con gli altri peer MNB sullo stato dei servizi. Espone inoltre un'interfaccia per eseguire azioni di monitoraggio e controllo sui servizi.

Per gestire la configurazione dei servizi si fa uso di un file di configurazione in JSON (vedi B) e di un database distribuito per memorizzare la configurazione attiva sul nodo e ottenere informazioni sugli altri nodi attivi. Il database utilizzato per tale proposito è *Zookeeper* [20], il quale è impiegato per la memorizzazione di configurazioni in sistemi distribuiti. Tale database offre una struttura dati ad albero, in cui i dati sono memorizzati sia nelle foglie che nei nodi intermedi, l'identificazione dei nodi avviene con una sintassi simile a quella utilizzata nei filesystem UNIX e offre la possibilità di definire un'ACL(Access Control List) sui nodi. Sarà ora esposto brevemente il modello adottato per rappresentare le configurazioni dei nodi. .

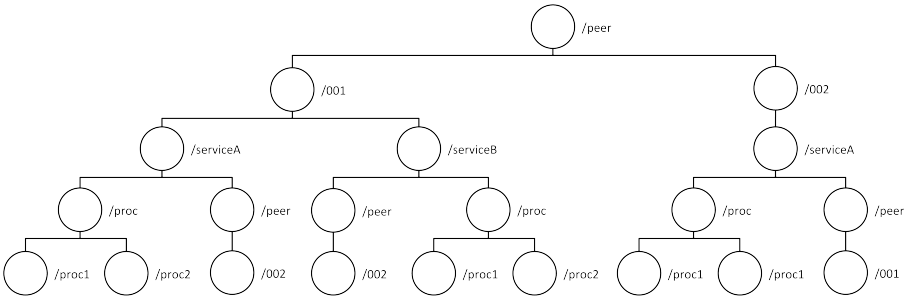


Figura 5.1.: Struttara dati configurazione nodi su Zookeeper

È presente un nodo peer che contiene tutte le configurazioni dei nodi della rete. Ogni nodo della rete è rappresentato da un nodo identificato dall'identificativo del nodo della rete. I suoi figli sono i servizi attivi e disponibili sul nodo, i quali memorizzano all'interno del nodo dati su socket e altre informazioni di stato, e hanno due figli proc e peer, relativamente informazioni sui processi attivi e peer broker.

Viene poi sfruttato un meccanismo di notifica dell'inserimento di nuovi figli sul nodo /peer che permette di rilevare l'attivazione di peer MNB e attivare il canale di subscribe a tale peer.

Una volta avviato, il MNB legge entrambe le configurazioni e valida quella presente sul database. Se la validazione non va a buon fine ne viene creata una nuova basandosi su quella presente sul file JSON.

L'heartbeating con i broker è impostato con una frequenza di 1 s e massimo numero di mancate risposte che portano ad un riavvio pari a 5. Nell'heartbeating con i peer invece si utilizza una frequenza di 2 s.

L'interfaccia di comando esposta è rappresentata da un insieme di richieste che è possibile effettuare le cui codifiche sono riportate in A.

5.2.4. Comunication Module

Il modulo di comunicazione permette ai worker di ricevere e effettuare richieste ed è sviluppato per facilitare l'integrazione del framework in sistemi preesistenti. Il modulo è stato implementato in Node.js e saranno fornite le linee guida per sviluppare un modulo analogo anche per altri linguaggi di programmazione.

Tale modulo, attraverso il DEALER, riceve le richieste dal broker. Queste richieste devono poi essere associate ad un'azione di un controller e per far ciò viene lanciato l'evento *request* catturato poi dal worker che attiverà l'azione associata al controller.

Il modulo dovrà fornire un metodo per poter inviare richieste, il quale necessità di conoscere il servizio e l'azione richiesta oltre i parametri. Il modulo di comu-

nicazione dovrà poi occuparsi di connettersi con il broker del servizio, inviare la richiesta e una volta ricevuta la risposta restituire il risultato e gli eventuali errori. Per le specifiche sulla codifica dei messaggi vedere A.

Il modulo si occupa inoltre delle operazioni di heartbeating e di raccolta delle informazioni necessarie al servizio di load balancing, le quali vengono poi periodicamente inviate a quest'ultimo.

5.3. Servizi predefiniti

Saranno ora presentati i servizi predefiniti offerti dal framework.

5.3.1. Web server

Il framework fornisce un web server per richieste REST. Il server riceve le richieste e grazie ad un meccanismo di routing ottiene la tripletta servizio, controller e azione a cui inoltrare le richieste attraverso il modulo di comunicazione. Una volta ricevuta la risposta alla richiesta dal servizio, viene inviata la risposta alla richiesta REST. È possibile utilizzare un timer per il tempo di risposta alle richieste, il quale fissa un tempo massimo per la ricezione della risposta, superato il quale viene inviato al client un messaggio di errore. È possibile usufruire inoltre di un sistema di sessioni basato su Redis, che può essere integrato con il meccanismo di sessioni utilizzato da PHP. Se attivato, tale meccanismo fornisce dati relativi alla sessione dell'utente che ha effettuato la richiesta ai controller dei vari servizi.

5.3.2. Server per real-time web

Il server per il real-time web riceve richieste attraverso un canale di comunicazione bidirezionale instaurato con il client, grazie alla libreria Socket.IO per Node.js. Come esposto precedentemente la comunicazione avviene su WebSocket o altre tecnologia dove questa non sia supportata. L'utilizzo di questo server permette la diminuzione del tempo latenza nelle richieste e l'invio da parte dei servizi di inviare messaggi a client specifici. Questo server è generalmente utilizzato per l'aggiornamento di contenuti dinamici sul client e meccanismi di notifica.

5.3.3. Load Balancer

Questo servizio si occupa di monitorare i tempi di risposta alle richieste dei vari servizi e al superamento di un data soglia, inferiore o superiore, richiede al MNB la rimozione o l'aggiunta di worker al servizio. Risponde poi a richieste sui tempi

di risposta dei servizi monitorati. Le informazioni necessarie al monitoraggio vengono fornite periodicamente dai moduli di comunicazione abilitati.

5.3.4. Web Interface

Questo servizio implementa un interfaccia web che permette il monitoraggio e la gestione del framework. Utilizza l'interfaccia di controllo e monitoraggio del MNB per eseguire le operazioni di gestione e ottenere informazioni sullo stato e sulla configurazione dei nodi. Permette la gestione degli altri nodi del framework e fornisce informazioni su tempi di risposta medi alle richieste dei servizi, ottenuti dal servizio di load balancing.

5.4. Integrazione WSUP negli applicativi lato server di Uniants

Completato lo sviluppo del middleware e dei servizi predefiniti offerti dal framework, lo si è integrato negli applicativi Node.js di Uniants. Per prima cosa si sono individuati i servizi in cui aggregare i controller, si è poi passati alla realizzazione dei worker per tali servizi, all'adattamento dei controller per l'integrazione ove necessario e alla configurazione dei web server e dei servizi.

I due server web precedentemente utilizzati sono stati sostituiti con i due servizi predefiniti del framework che offrono funzionalità analoghe, mantenendo la stessa gestione delle sessioni sviluppata in precedenza. È stato rimosso il sistema di notifiche basato sul meccanismo di publish-subscribe offerto da Redis, poiché grazie al framework è possibile inviare notifiche ai client inviando una richiesta al server per il real-time web. Sono stati quindi modificati i controller che gestivano l'invio delle notifiche in tempo reale al client per sfruttare tale servizio, permettendo di inviare notifiche da macchine differenti da quella in cui risiede il servizio per il real-time web, senza attivare una replica del database di Redis, utilizzato precedentemente per le notifiche, sulla macchina che invia la notifica semplificando l'aggiunta di nuovi nodi.

Per semplificare l'integrazione del precedente applicativo, sono stati sviluppati dei *wrapper* che permettono di esporre i servizi offerti precedentemente l'introduzione del framework senza modificare i controller. Questo è stato possibile sui controller che interagivano solo con controller appartenenti allo stesso servizio. Mentre nel caso di controller che interagiscono con controller di servizi differenti è stato necessario modificare il controller e integrare il CM per effettuare tali richieste.

Il lavoro di progettazione e implementazione svolto ha permesso di raggiungere gli obiettivi prefissati per il framework e ottenere un integrazione efficacie con il

minimo sforzo, dato che la maggior parte dei controller interagivano con controller dello stesso servizio.

L'integrazione del framework ha inoltre permesso la progettazione e l'avvio dello sviluppo di una nuova interfaccia al database a grafo Neo4j, in Java, che consentirà in primo luogo il raggiungimento di prestazioni più elevate di quasi un ordine di grandezza rispetto a quelle attuali, grazie all'utilizzo dell'API core di Neo4j che permette di manipolare direttamente i nodi senza l'ausilio del linguaggio Cypher. In questo lavoro non è stato possibile condurre test comparativi delle prestazioni ottenute con la nuova libreria rispetto alla precedente in quanto al momento della stesura dell'elaborato questa è ancora in fase di sviluppo.

6. Test e analisi delle prestazioni

In questo capitolo saranno esposti i risultati dei test condotti sugli applicativi lato server del servizio Uniants integrati nel framework WSUP. Sono stati condotti una serie di test per valutare le prestazioni e la scalabilità del framework e per effettuare la comparazione delle prestazioni dell'applicativo lato server nella versione precedente e successiva all'introduzione del framework.

6.1. Configurazione del sistema

I test sono stati condotti con una configurazione a quattro macchine virtuali collegate tra loro in rete. Ciascuna di queste macchine, virtualizzate con *QEMU 1.3*, disponeva di una configurazione con 4GB di RAM, tre processori e *Ubuntu 12.10*. La macchina utilizzata per la simulazione è invece una workstation *Dell R5500* che dispone di due processori *Intel Xeon X5675* (3.07 GHz, 6 core e 12 MB di cache), 128GB di RAM DDR3 1333 MHz e le immagini delle macchine virtuali memorizzate su un server NFS *Netapp 3240* collegato su rete gigabit.

Per effettuare i test sono stati scelti i due servizi che gestiscono l'interazione con i database, rispettivamente Cassandra e Neo4j, e il servizio del server web RESTful. Sono state poi scelte due differenti configurazioni del framework: una prima in cui tutto il framework viene eseguito su una singola macchina e una seconda in cui il framework viene distribuito su tre macchine, allocando ciascun servizio su di una macchina differente. In ciascuna delle due configurazioni sono stati condotti test al variare del numero di worker per i servizi ad esclusione del servizio del server web che ha sempre utilizzato un singolo worker.

È stata utilizzata una macchina per eseguire i tool di benchmarking e per condurre test di funzionalità ospitando il server Apache. Per eseguire i test sono stati utilizzati due differenti software di benchmarking: *Apache Bench v2.2* [21] e *Apache JMeter v2.9* [22]. Il primo consente di effettuare richieste HTTP, definendo il numero totale di richieste e il numero di richieste concorrenti e fornisce poi risultati sia riassuntivi che delle singole richieste. Il secondo permette di eseguire test più complessi ed è stato infatti impiegato nei test di simulazione di traffico reale, dove è stato inviato contemporaneamente un insieme di richieste per un numero fissato di volte e con differenti valori di concorrenza. Infine è stato

utilizzato *nmon v14g* [23] per monitorare l'utilizzo di risorse del sistema e *nmon analyzer* per l'estrapolazione dei risultati.

Per eseguire alcuni test sono state individuate due richieste, che permetto di effettuare test sui due servizi di interfaccia ai database scelti, che riportiamo qui di seguito:

- */class/112293/history*: fornisce le ultime cinquanta azioni eseguite sul corso identificate con il codice 112293, utilizza il servizio di interfaccia a Cassandra;
- */career/115311/class*: fornisce la lista dei corsi a cui è iscritto l'utente identificato dal codice 115311, utilizza il servizio di interfaccia a Neo4j.

A ogni variazione della configurazione sono stati riavviati sia i servizi del framework sia i database, i risultati dei test sono stati ottenuti eseguendo due volte ciascun test e prendendo il valor medio tra i valori ottenuti. Nei test indicheremo con *SM* la configurazione a singola macchina e con *D* la configurazione distribuita.

Per la distribuzione della versione del framework e dei componenti di Uniants è stata utilizzata una repository privata di *GitHub*.

Le versioni dei software utilizzati durante i test sono le seguenti: *Node.js v0.8.20*, *Zookeeper 3.4.5*, *ØMQ 2.2*.

6.2. Test di scalabilità

Questi test permetto il confronto delle prestazioni dei servizi di interfaccia ai database scelti, al variare del numero di worker utilizzati. I test sono stati condotti variando il numero di richieste eseguite e il numero di richieste concorrenti, nelle due configurazioni scelte. Per condurre i test è stato utilizzato Apache Bench variando numero di richieste totali e concorrenti, utilizzando le due richieste definite precedentemente.

6.2.1. Neo4j

Sono ora riportati i dati riassuntivi dei test, relativi a throughput e tempo di risposta alle richieste al servizio di interfaccia al database Neo4j, nelle due differenti configurazioni del framework. Sono inoltre riportati i risultati ottenuti dall'applicazione precedente all'introduzione del framework, come misura di confronto delle prestazioni.

configurazione	semplice	singola macchina				distribuita			
worker		1	2	4	6	1	2	4	6
1-1	76,055	52,25	59,4	79,165	13,84	52.25	59.4	79.17	13.84
10-1	125,2	196,0	188,43	191,5	115,8	196,0	188,4	191,165	115,8
10-10	168,3	358,8	550,9	793,4	647,6	358,8	550,9	793,4	476,8
100-1	144,9	190,2	207,4	213,3	204,9	190,2	207,3	213,3	168,2
100-10	177,2	278,2	585,4	803,2	1095,4	278,2	585,4	803,2	714,9
100-100	130,4	311,4	522,1	1251,1	1120,4	311,4	522,1	1251,1	622,7
1000-1	143,8	184,3	202,2	243,5	213,5	184,3	202,2	243,5	172,9
1000-10	175,5	188,5	370,6	1012,4	1475,5	188,5	370,6	1012,4	932,8
1000-100	170,0	192,5	395,7	867,1	1563,3	192,5	395,7	687,1	905,4
1000-1000	166,1	179,2	299,6	1264,8	1016,7	179,2	299,6	1264,8	1016,7

Tabella 6.1.: Risultati test di scalabilità, throughput (richieste/s) Neo4j.

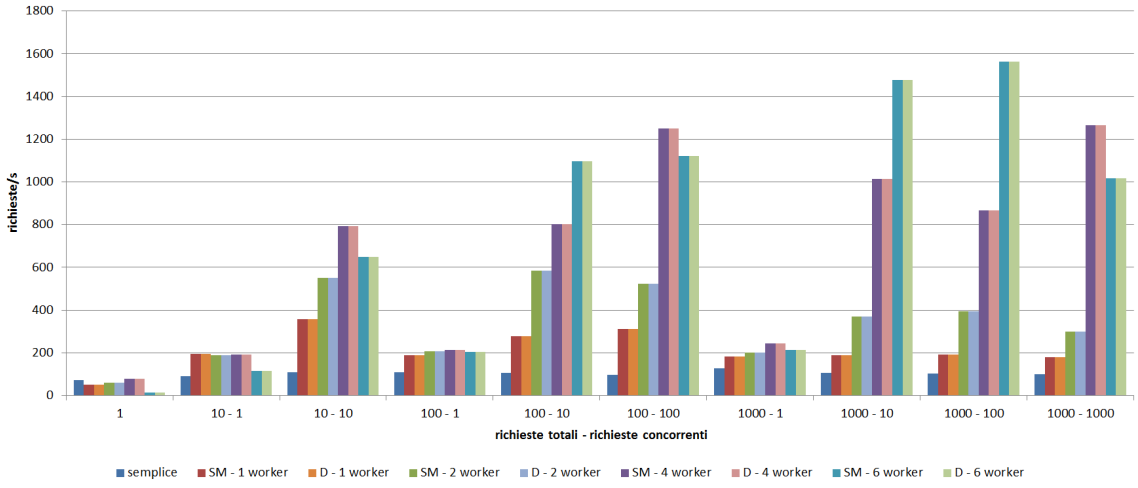


Figura 6.1.: Grafico test scalabilità, throughput Neo4j**??**.

configurazione	semplice	singola macchina				distribuita			
worker		1	2	4	6	1	2	4	6
1-1	13,2	63,9	40,0	37,0	35,5	63,9	40,0	36,4	35,4
10-1	8,0	5,1	5,3	5,3	12,2	5,1	5,3	5,2	12,2
10-10	6,0	2,8	1,8	1,8	1,5	2,8	1,8	1,3	1,5
100-1	6,9	5,3	4,8	4,8	4,9	5,3	4,8	4,7	4,9
100-10	5,7	4,6	2,6	2,6	0,9	4,6	2,6	1,6	0,9
100-100	8,8	5,0	4,7	4,7	0,9	5,0	4,7	0,8	0,9
1000-1	7,0	5,4	5,0	5,0	4,7	5,4	5,0	4,6	4,7
1000-10	5,5	5,3	2,7	2,7	0,7	5,3	2,7	1,0	0,7
1000-100	5,9	5,2	2,5	2,5	0,6	5,2	2,5	1,2	0,6
1000-1000	6,0	5,6	3,4	3,4	1,2	5,6	3,4	0,8	1,2

Tabella 6.2.: Risultati test di scalabilità, tempo di risposta medio (ms) Neo4j.

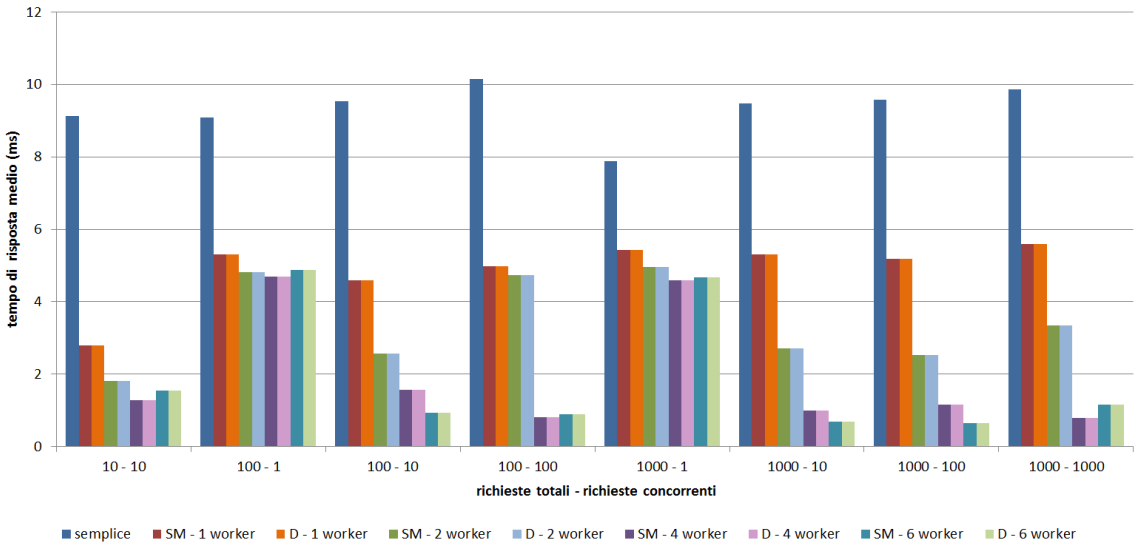


Figura 6.2.: Grafico test scalabilità, tempo di risposta medio Neo4j.

Come si evince dai grafici 6.1 e 6.2, l'applicativo ottiene un guadagno prestazionale, legato al numero di worker attivi, maggiore quando aumenta il grado di concorrenza delle richieste che permette di sfruttare al meglio la loro elaborazione parallela. Dai risultati si deduce anche che nell'ultimo test con 1000 richieste concorrenti, la configurazione a sei worker ha presentato prestazioni inferiori rispetta a quella a quattro worker. Tale fatto può essere attribuito all'eccessivo numero di richieste effettuate verso il server REST che permette l'interazione con Neo4j, il quale le elabora sequenzialmente. Nei primi due test (con una e dieci richieste) i tempi di risposta sono più elevati a causa della maggiore incidenza sulla media, dei tempi necessari all'inizializzazione della comunicazione tra il server web RESTful e i servizi, che avviene nella prima richiesta al servizio.

Segnaliamo poi, che prestazioni così elevate in throughput di Neo4j (con Cypher e in modalità non embedded) sono dovute all'utilizzo della cache, che è stata mantenuta attiva per non limitare le possibili richieste che il servizio può gestire, che sono comunque legate alle prestazioni del server REST di Neo4j.

Da quanto è possibile veder dai grafici non vi è alcuna sostanziale differenza tra le due configurazioni, ne in throughput ne nei tempi di risposta.

6.2.2. Apache Cassandra

Sono ora riportati i dati riassuntivi dei test, relativi a throughput e tempo di risposta alle richieste al servizio di interfaccia al database Apache Cassandra, nelle due differenti configurazioni del framework. Sono inoltre riportati i risultati ottenuti dall'applicazione precedente all'introduzione del framework come misura di confronto delle prestazioni.

Durante i test con 1000 richieste concorrenti si sono verificati crash di Apache Cassandra a causa della scarsa quantità di RAM a disposizione.

Come è possibile notare dai grafici il guadagno prestazionale dovuto a un maggior numero di worker è evidente anche se non proporzionale al numero di worker utilizzati. Inoltre non si ha alcun guadagno ma anzi una leggera perdita prestazionale quando il grado di concorrenza è minimo, quindi si hanno richieste sequenziali. Ciò è dovuto al fatto che l'overhead introdotto dal framework, non è compensato dalla maggior capacità computazionale, che è resa comparabile alla situazione a singolo worker, dalla mancanza di concorrenza nelle richieste e da un

configurazione	semplice	singola macchina				distribuita			
worker		1	2	4	6	1	2	4	6
1-1	76,1	61,2	41,0	24,9	9,4	45,2	21.7	10,0	14,5
10-1	125,2	83,4	57,5	55,2	31,8	50,9	49,8	41,1	37,5
10-10	168,3	124,0	158,8	248,8	178,0	110,4	161,9	205,9	184,2
100-1	144,9	83,5	71,8	67,4	59,6	75,3	67,4	59,8	58,4
100-10	177,3	141,5	241,9	279,4	302,5	135,6	251,0	254,4	297,0
100-100	130,4	147,0	249,7	283,4	278,0	134,9	256,9	290,1	307,1
1000-1	143,8	86,4	76,8	71,6	68,5	73,6	72,6	68,0	63,7
1000-10	175,6	133,2	259,7	296,0	324,9	138,5	265,3	303,4	227,4
1000-100	170,0	144,3	253,6	288,5	314,4	133,7	258,1	289,8	211,1
1000-1000	166,1	145,0	244,2	284,5	308,2	136,5	247,6	284,8	310,1

Tabella 6.3.: Risultati test di scalabilità, throughput (richieste/s) Apache Cassandra.

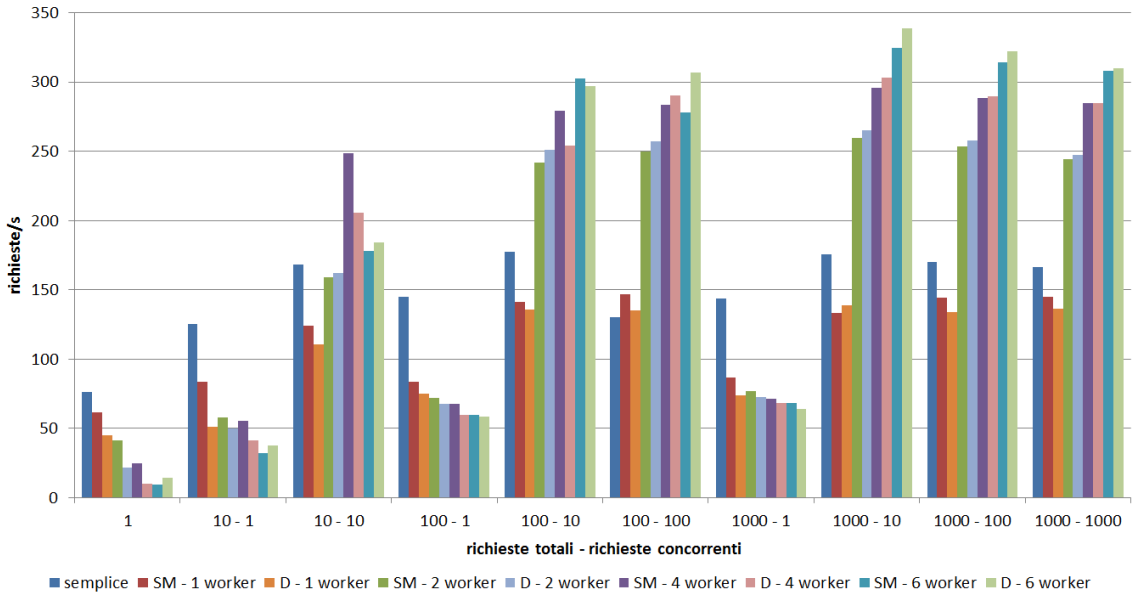


Figura 6.3.: Grafico test scalabilità, throughput Apache Cassandra.

configurazione	semplice	singola macchina				distribuita			
worker		1	2	4	6	1	2	4	6
1-1	13,71	16,4	24,5	61,1	107,2	22,3	66,2	99,9	76,1
10-1	8,0	12,0	17,4	18,2	37,3	19,9	20,2	24,5	26,7
10-10	6,0	8,2	6,4	4,0	5,6	9,4	6,3	4,5	5,6
100-1	6,9	12,1	13,9	14,8	16,8	13,3	14,8	16,8	17,1
100-10	5,7	7,1	4,1	3,6	3,3	7,4	4,0	3,9	3,7
100-100	8,8	6,8	4,0	3,5	3,6	7,4	3,9	3,5	3,3
1000-1	7,0	11,6	13,0	14,0	14,6	13,6	13,8	14,7	15,2
1000-10	5,6	7,5	3,9	3,1	3,1	7,2	3,8	3,3	3,0
1000-100	5,9	6,9	3,9	3,2	3,2	7,5	3,9	3,5	3,1
1000-1000	6,0	6,9	4,1	3,5	3,3	7,3	4,2	3,6	3,2

Tabella 6.4.: Risultati test di scalabilità, tempo di risposta medio (ms) Apache Cassandra.

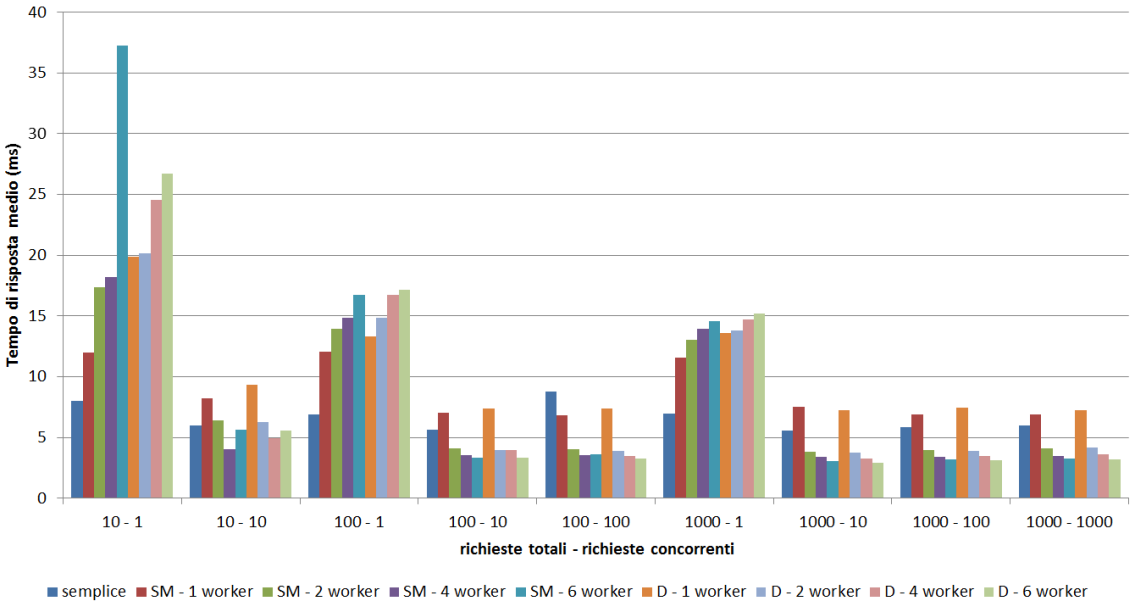


Figura 6.4.: Grafico test scalabilità, tempo di risposta medio Apache Cassandra.

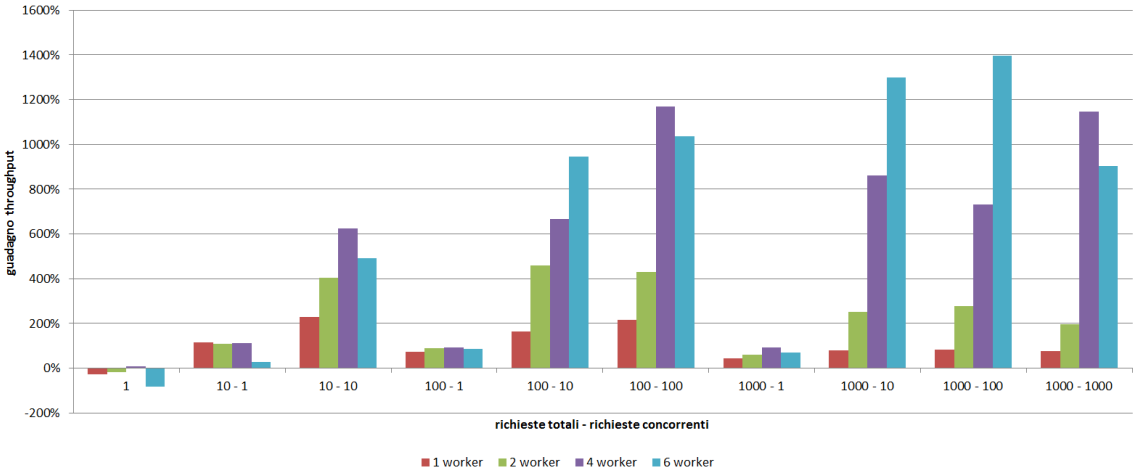


Figura 6.5.: Grafico riassuntivo risultati test di scalabiltà.

tempo di elaborazione della richiesta, da parte del servizio, più elevata rispetto al caso precedente, dove queste problematiche non sono presenti.

Fatta qualche eccezione, anche in questo caso il throughput e i tempi di risposta sono comparabili nelle due differenti configurazioni.

6.2.3. Risultati riassuntivi

Il grafico in figura 6.5 riporta in sintesi il guadagno prestazionale ottenuto dalla soluzione che integra WSUP. È possibile vedere come l'integrazione del framework ha portato guadagni notevoli soprattutto con l'utilizzo di quattro worker. Utilizzando più di quattro worker si ottengono prestazioni inferiori dovute probabilmente al maggior numero di richieste concorrenti effettuate sul server REST per l'interrogazione del database Neo4j, limite che sarà risolto con la futura introduzione della nuova interfaccia al database che lo utilizzerà in modalità embedded.

6.3. Test sull'utilizzo delle risorse

La finalità di questo test è mettere a confronto le risorse utilizzate dall'applicativo lato server di Uniants.com nella versione precedente e successiva all'introduzione del framework, con una configurazione a singola macchina, effettuando richieste ai due servizi di interfaccia ai database scelti, per i quali sono stati attivati due

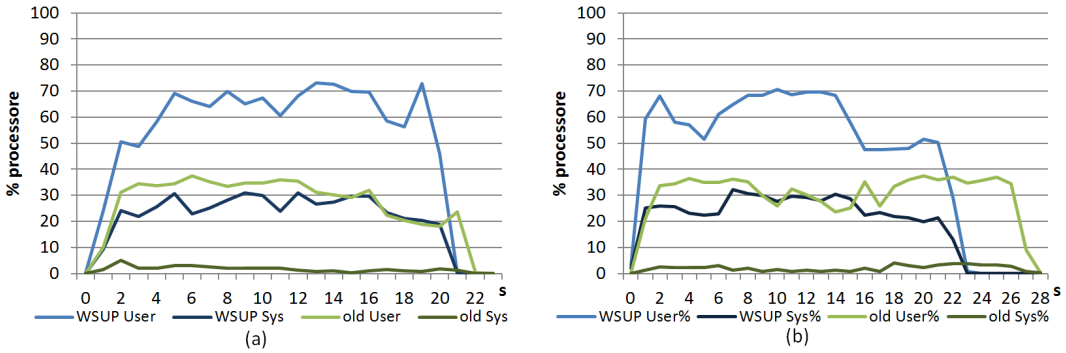


Figura 6.6.: (a) Confronto utilizzo processore Neo4j. (b) Confronto utilizzo processore Apache Cassandra.

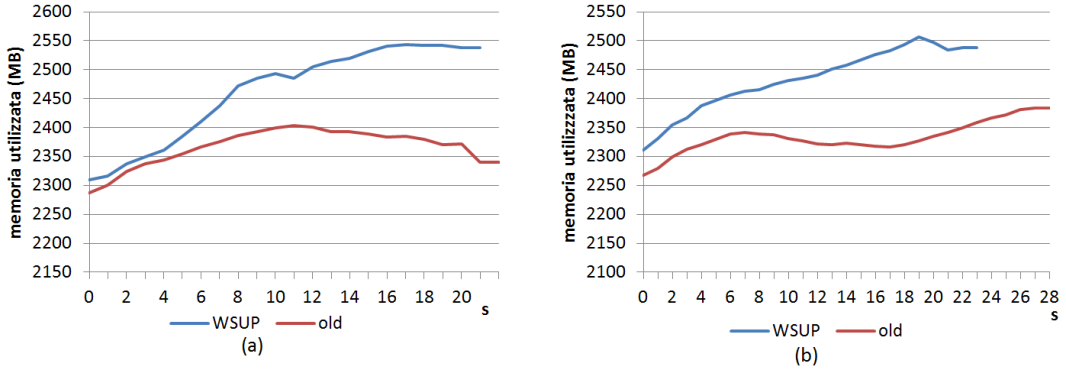


Figura 6.7.: (a) Confronto utilizzo memoria Neo4j. (b) Confronto utilizzo memoria Apache Cassandra.

worker. Il test è stato effettuato con 500 richieste totali e concorrenti, effettuate con Apache Bench.

È possibile notare in entrambi i casi un notevole incremento della percentuale di processore dedicata al sistema, dovuta allo scambio di messaggi attraverso socket IPC, la quale passa da una media di circa il 3% al 25%. Anche la percentuale di processore dedicata all'utente ha un notevole incremento in quanto passa da una media del 30% ad una superiore al 60%. A fronte di questo aumento di risorse sia ha però un aumento delle prestazioni con il throughput medio, che passa da circa 100 richieste al secondo a più di 370 (+270%), per il servizio di interfaccia a Neo4j, e dalle circa 160 richieste al secondo alle circa 250 (+56%),

per il servizio di interfaccia a Cassandra. Per quanto riguarda la memoria invece si ha un utilizzo di 200MB (+8,5%) in più nel caso di Neo4j e 180MB (+7,7%) nel caso di Cassandra.

6.4. Conclusioni

Grazie ai risultati ottenuti e in relazione alle specifiche delle macchine utilizzate, possiamo concludere che il framework riesce a scalare piuttosto bene fino al raggiungimento dei limiti prestazionali imposti dai database con cui si interfacciano i servizi con i quali è stato effettuato il test. Da notare che l'incremento ottenuto nel numero di richieste servite, varia tra il 30% a quasi il 1400%. È poi possibile ipotizzare che le ottimizzazioni discusse in 5.2.1, relative al binding della libreria di ØMQ a Node.js, non abbiano influito sui guadagni prestazionali, in quanto le prestazioni dei database hanno limitato le prestazioni del framework a tal punto da non raggiungere i limiti imposti dai singoli moduli di comunicazione nell'invio di richieste e risposte.

Conclusioni

In questo lavoro è stata trattata la progettazione e l'implementazione di un framework distribuito per Web service. La necessità dello sviluppo di tale framework, nasce dall'esigenza di supportare l'evoluzione degli applicativi lato server del servizio Uniants, offrendo caratteristiche di scalabilità e affidabilità e la possibilità d'interazione tra componenti sviluppati in differenti linguaggi di programmazione in ambiente distribuito.

Sono stati soddisfatti tutti gli obiettivi inizialmente definiti, ottenendo una prima versione del framework, composto di un middleware message-oriented, un servizio di load balancing, un'interfaccia web per la sua gestione, un server web RESTful e uno per il real-time web. Il middleware è basato su una variante del modello architetturale a broker distribuito con servizio di directory distribuito, che permette la comunicazione tra i componenti del framework sviluppati in differenti linguaggi di programmazione ed eseguiti su piattaforme hardware e software eterogenee. I due server web integrano poi, un sistema di routing e uno per la gestione delle sessioni basato su Redis. Per implementare tale sistema si è scelto di utilizzare la libreria di messaging ØMQ e Zookeeper come database distribuito per la memorizzazione delle configurazioni dei nodi, implementando le componenti in Node.js.

Il lavoro svolto ha permesso una facile integrazione del framework, nell'applicativo lato server di Uniants precedentemente sviluppato, grazie ad una progettazione rivolta alla realizzazione di un framework flessibile e facilmente integrabile in applicazioni preesistenti. Sono state definite delle specifiche sullo scambio dei messaggi e sulla loro formattazione, per integrare applicativi sviluppati in differenti linguaggi di programmazione.

Analizzando i risultati dei test condotti, l'implementazione del framework ha permesso di ottenere un buon guadagno prestazionale all'applicativo lato server del servizio Uniants, dove il numero medio di richieste servite al secondo ha raggiunto incrementi fino al 1400%, mettendo in evidenza un buon comportamento del framework relativamente alla scalabilità e al ridotto overhead, introdotto nei tempi di risposta alle richieste.

I risultati ottenuti incoraggiano successivi lavori e sviluppi del framework, che proseguiranno per affrontare tematiche riguardanti la sicurezza, un più evoluto sistema di monitoraggio, gestione e configurazione del sistema e strategie di load balancing più evolute. Saranno poi implementati moduli di comunicazione per

altri linguaggi di programmazione e ottimizzate ulteriormente le performance dei componenti del framework, anche grazie al suo utilizzo nel servizio Uniants.

Elenco delle figure

Elenco delle tabelle

A. Specifiche messaggi

Codifica messaggi Campi contenuti nei messaggi dati di richiesta:

- *reqId*: intero progressivo che identifica univocamente la richiesta del componente;
- *fn*: funzione da richiamare sul servizio;
- *ctr*: controller su cui invocare la funzione, opzionale;
- *args*: parametri da passare alla funzione.

Campi contenuti nei messaggi dati di risposta:

- *reqId*: codice identificativo della richiesta cui è associata la risposta;
- *err*: errori verificatesi nell'elaborazione della risposta;
- *res*: risultati della risposta.

Campi contenuti nei messaggi di controllo di richiesta:

- *reqId*: intero progressivo che identifica univocamente la richiesta del componente, opzionale;
- *req*: codice identificativo della richiesta;
- *args*: parametri della richiesta.

Campi contenuti nei messaggi dati di risposta:

- *reqId*: codice identificativo della richiesta cui è associata la risposta;
- *args*: valori di ritorno della risposta.

Codifica Richieste Sono qui riportati i codici, utilizzati dai componenti nello scambio di messaggi, per codificare le richieste:

- *REQ_BOOT ("bt")*: richiesta informazioni per l'inizializzazione di un broker;
- *REQ_ADD_PEER ("ap")*: richiesta aggiunta peer su di un broker;
- *REQ_REM_PEER ("rp")*: richiesta rimozione peer su di un broker;
- *REQ_UPD_PEER ("up")*: richiesta aggiornamento informazioni relative a un peer broker;
- *REQ_INFO_PEER ("ip")*: identifica un messaggio contente informazioni sul peer, es. stato e servizi;
- *REQ_RESTART_WORKER ("rw")*: richiesta riavvio di un worker;
- *REQ_CONFIG ("cf")*: identifica messaggi per la richiesta o l'invio di configurazioni;
- *REQ_HEARTBEAT ("hb")*: identifica un messaggio di heartbeat.

Sono ora riportate le codifiche specifiche delle azioni che è possibile richiedere al LRP Manager:

- *REQ_START_PROCESS ("sp")*: richiesta per l'avvio di un processo;
- *REQ_RESTART_PROCESS ("rp")*: richiesta per il riavvio di un processo;
- *REQ_STOP_PROCESS ("kp")*: richiesta per la terminazione di un processo;
- *REQ_RECOVERY ("re")*: richiesta informazioni necessarie alla procedura di rispristino in seguito a un guasto;
- *REQ_PROC_INFO ("pi")*: richiesta d'informazioni sui processi attivi;
- *REQ_UPDATE_PROC_INFO ("pu")*: richiesta per l'aggiornamento d'informazioni relative ai processi.

B. Configurazione servizi

Configgurazione JSON È ora riportato un esempio di configurazione JSON relativa a un servizio da attivare su di un nodo. Il file JSON contenente la configurazione, ha al suo interno un insieme di oggetti che rappresentano le configurazioni dei servizi da attivare su di un nodo.

```
"id_servizio": {
        "name": "nome_servizio",
        "worker": {
                "config":{
                        "file": "path/to/service/file",
                        "procType": tipo_processo,
                        "startProcNum": numero_worker
                }
        }
}
```

Dove:

- *id_servizio*: identificativo del servizio composto di tre caratteri, utilizzato nelle identità dei socket associati al servizio;
- *nome_servizio*: nome del servizio;
- *path/to/service/file*: percorso su disco del worker del servizio;
- *tipo_processo*: tipo di worker tra i seguenti:
  - *0*: file Node.js;
  - *1*: file eseguibile;
  - *2*: file Java;
  - *10*: file Node.js che utilizza il wrapper per i servizi sviluppati in Node.js.
- *numero_worker*: numero di worker da istanziare all'avvio del servizio.

Configurazione Zookeeper Sarà esposta la configurazione dei nodi memorizzata su Zookeeper, di cui si può trovare una schematizzazione d'esempio in 5.1. All'interno dei nodi, le informazioni sono serializzate in JSON. La struttura è ad albero e si possono individuare i seguenti nodi:

- */peer*: rappresenta il nodo padre contenente la configurazione dei nodi;
- */node_id*: contiene informazioni di stato sul nodo. node_id è l'identificativo numerico a tre cifre del nodo (con eventuale padding), contiene indirizzo e identità del PUB socket.
- */service_id*: contiene informazioni di stato del servizio. Ha due figli contenti i processi attivi associati al servizio e i peer broker cui è connesso il broker. Su di esso sono memorizzate le seguenti informazioni: o name: nome del servizio o worker: oggetto contente una configurazione analoga a quella definita nella configurazione JSON (B) del worker per il dato servizio. o broker: contiene l'oggetto config che al suo interno ha l'oggetto socket contenente identità e indirizzo (id e address) del socket cloud_frontend del broker del servizio.
- */proc*: rappresenta il nodo padre dei processi del servizio;
- */proc_id*: contiene informazioni di stato sul processo. Su di esso sono memorizzate le seguenti informazioni:
  - *lrpId*: identificativo del processo per LRP Manager; o args: argomenti con cui è stato avviato;
  - *procType*: tipo di processo (analogo a quanto definito nella specifica JSON)
  - *type*: 0 per broker, 1 per worker.
- */peer*: rappresenta il nodo padre dei peer broker del servizio;
- */peer_id*: contiene informazioni sul peer broker. Su di esso è memorizzato l'oggetto socket contenente identità e indirizzo (id e address) del cloud_forntend del broker.

Bibliografia

[1] F. M. E. N. M. C. C. F. D. O. David Booth, Hugo Haas, "Web services architecture," tech. rep., W3C, 2004.

[2] A. A. L. J.-J. M. D. O. S. W. Roberto Chinnici, Hugo Haas, "Web Services Description Language (WSDL) version 2.0 part 2: Adjuncts," tech. rep., W3C, 2007.

[3] Y. L. Nilo Mitra, "Soap version 1.2 part 0: Primer (second edition)," tech. rep., W3C, 2007.

[4] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[5] N. M. J.-J. M. H. F. N. A. K. Y. L. Martin Gudgin, Marc Hadley, "Soap version 1.2 part 1: Messaging framework (second edition)," tech. rep., W3C, 2007.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - a technique for cheap recovery," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2004.

[7] "Uniants." `http://www.uniants.com`.

[8] C. delle comunità europee, "Piano d'azione eLearning." `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2001:0172:FIN:IT:PDF`, March 2001.

[9] "Zend Framework 1." `http://framework.zend.com/`. Last visit Feb 2013.

[10] "Apache Cassandra." `http://cassandra.apache.org/`, 2009.

[11] "MySQL." `http://www.mysql.it/`.

[12] "Neo4j." `http://www.neo4j.org/`.

[13] "Redis." `http://redis.io/`.

[14] "Node.js." `http://nodejs.org/`.

[15] "Socket.IO." `http://socket.io/`.

[16] "RabbitMQ." `http://www.rabbitmq.com/`.

[17] J. O'Hara, "Toward a commodity enterprise middleware," *Queue*, vol. 5, pp. 48–55, May 2007.

[18] "ØMQ." `http://www.zeromq.org/`.

[19] G. Candea and A. Fox, "Crash-only software," in *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2003.

[20] "Zookeeper." `http://zookeeper.apache.org/`.

[21] "Apache Bench." `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[22] "Apache JMeter." `http://jmeter.apache.org/`.

[23] "nmon." `http://www.ibm.com/developerworks/wikis/display/WikiPtype/nmon`.