Approfondimento JBoss Clustering

Sistemi Distribuiti M

Data

29/06/2012

Studente

Alessandro Montanari

Indice

1. Clustering in teoria

1.1 Clustering e Distribuzione

Una prima distinzione che vale la pena fare è quella tra clustering e distribuzione di componenti applicativi. La distribuzione è la tecnica secondo la quale l'applicazione viene suddivisa in diversi componenti che sono collocati su macchine distinte, quindi sono necessarie più macchine per servire una singola applicazione. Con il termine clustering invece si intende l'esecuzione simultanea di tutti i componenti costituenti una stessa applicazione su macchine diverse. Nel caso specifico di JBoss un cluster viene formato quando diverse istanze di application server servono la stessa applicazione. Un application server che è parte di un cluster viene detto nodo, tutti i nodi all'interno di un cluster comunicano fra di loro per realizzare funzionalità utili come la replicazione dello stato o il failover, descritti nelle successive sezioni.

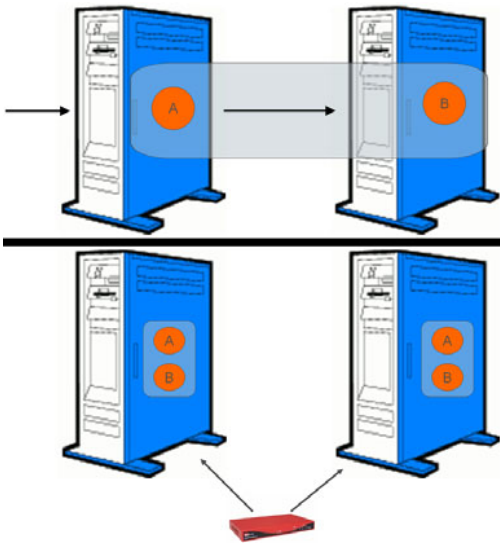


Figura 1.1 - Clustering e Distribuzione

Nella figura 1.1 viene mostrato come nel caso di un'applicazione distribuita ogni volta che è necessario utilizzare un componente che si trova su un'altra macchina è necessario effettuare un chiamata remota, mentre nel clustering dato che tutti i componenti sono presenti su ogni macchina, le chiamate avvengono direttamente. Nel clustering è di solito presente un load balancer che si occupa di distribuire il carico sulle diverse macchine (sezione 1.3).

In genere la distribuzione è più complicata da realizzare e meno performante del clustering in quanto le chiamate effettuate all'interno di uno stesso processo, come avviene nel clustering, sono estremamente più veloci di chiamate effettuate sulla rete.

1.2 Tipologie di Cluster

Dal punto di vista della topologia, un cluster può essere formato da nodi in esecuzione sulla stessa macchina o su macchine diverse. Quando i nodi sono in esecuzione sulla stessa macchina il cluster viene detto *verticale*, mentre quando i nodi sono in esecuzione su macchine diverse il cluster viene detto *orizzontale*. Eventualmente, è possibile formare cluster *misti* che sono cioè sia verticali che orizzontali, come mostrato nella figura successiva.

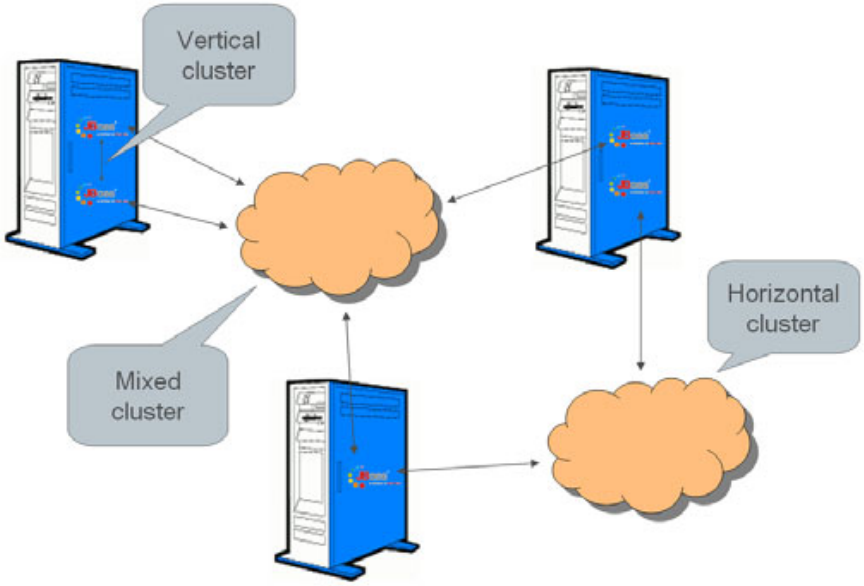


Figura 1.2 - Topologia di cluster

In generale i cluster orizzontali sono più facili da configurare in quanto non ci si deve preoccupare, ad esempio, di conflitti sulle porte.

Dal punto di vista delle prestazioni quale configurazione porti ai migliori risultati dipende da diversi fattori. Ad esempio se si dispone di un'unico server estremamente potente, l'esecuzione di più istanze di application server su quella macchina comporterebbe da un lato il miglior sfruttamento delle risorse hardware e dall'altro si eviterebbe l'overhead dovuto alle comunicazioni di rete con il risultato di ottenere un cluster più veloce. Una tale configurazione però soffrirebbe di guasti hardware, infatti se l'unico server disponibile si dovesse guastare l'applicazione smetterebbe di funzionare. Per sopportare tale evenienza un cluster orizzontale è più indicato. Un cluster orizzontale è preferibile anche in quelle situazioni in cui si vuole essere scalabili, ossia la capacità di sopportare più richieste semplicemente aggiungendo hardware alla configurazione, in cluster orizzontale infatti è sufficiente aggiungere altri server mentre in un cluster verticale di dovrebbe aggiornare l'unico server presente ma non è possibile aggiornare una singola macchina all'infinito. Per questi motivi spesso si utilizzano soluzioni miste.

Un cluster può inoltre essere *omogeneo* quando tutti i nodi hanno installati gli stessi servizi, mentre si dice *eterogeneo* quando ciò non accade. Anche in questo caso la scelta tra una configurazione e l'altra dipende da molti fattori e non esiste una soluzione migliore dell'altra in assoluto.

1.3 Load balancing

Con load balacing si intendono tutti i meccanismi che permettono di distribuire il carico di richieste su diverse istanze di application server per ottenere scalabilità e high availability. Il load balacing è uno dei principali metodi utilizzato per rendere la propria applicazione scalabile, infatti è possibile aggiungere più server oltre a quelli già disponibili e il load balancer sarà in grado di distribuire le richieste anche a tali server, potendo così incrementare il traffico sostenuto dall'applicazione, il tutto in modo trasparente per gli utenti.

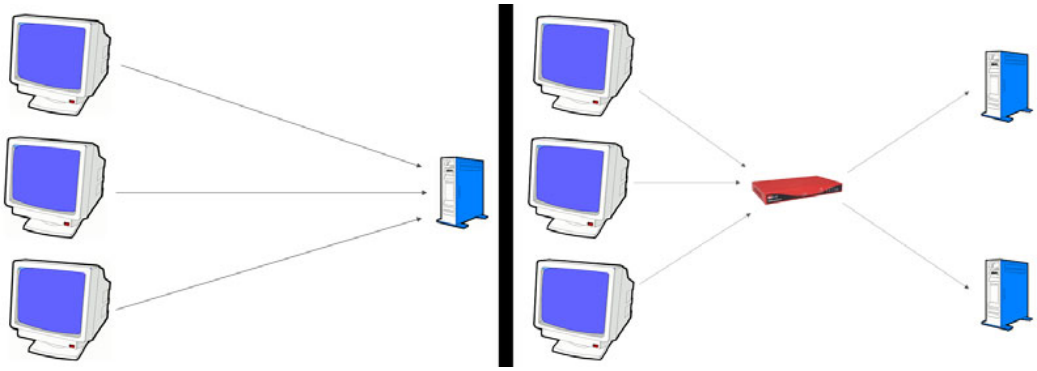


Figura 1.3 - Applicazione senza load balancing (sinistra) e con load balancing (destra)

Dalla Figura 1.3 si può notare come la scelta di load balancing, ossia a quale server inviare le richieste, viene presa prima ancora che la richiesta sia affettivamente gestita dall'application server, quindi di solito il load balancing non è una funzionalità sotto il controllo della nostra applicazione o dell'application server stesso e in generale non richiede un cluster. Un cluster infatti viene utilizzato ogni volta che i nodi devono comunicare tra di loro ma dal punto di vista del load balancing questo non è necessario: il carico di richieste può essere suddiviso su più server anche se ognuno di questi opera l'uno indipendentemente dagli altri.

Un load balancer può essere hardware o software. I load balancer hardware presentano di solito prestazioni migliori rispetto a quelli software ma sono molto più costosi e più difficili da configurare. I load balancer software invece sono meno costosi (in alcuni casi open-source) e possono essere implementati a livello di sistema operativo o come classiche applicazioni standalone. Nel caso di load balancer software però l'applicazione consuma risorse all'interno del sistema (CPU, memoria, ...) e le sue prestazioni dipendono fortemente dal carico del sistema stesso e dalle altre applicazioni in esecuzione.

Indipendentemente dal tipo di load balancer impiegato, esistono diverse strategie di load balancing che possono essere utilizzate per decidere verso quali server devono essere inviate le richieste. Tra le strategie più comuni ci sono Random, Round Robin e Sticky Session. Con la modalità Random ogni richiesta è inviata verso server scelti in modo casuale, con Round Robin invece le richieste sono inviate in modo sequenziale a tutti i server disponibili e infine con Sticky Session, la prima richiesta viene inviata in modo casuale o con Round Robin ma nel momento in cui il client ha stabilito una sessione con un particolare server, tutte le richieste successive saranno inviate allo stesso server. La modalità Sticky Session è molto utile in un cluster in quanto lo stato di interazione con l'utente non deve essere continuamente spostato da un server all'altro in base alle scelte

prese dal load balancer. Esistono in ogni caso diverse varianti che sono specifiche dei diversi prodotti hardware o software.

1.4 High Availability

Molte applicazioni web richiedono la cosiddetta high availability, ossia la capacità di accettare richieste con intervalli di non disponibilità minimi mantenendo comunque tempi di risposta accettabili.

Esistono principalmente due modi per ottenere high availability e dipendono dalla natura dell'applicazione che si deve rendere disponibile. Se l'applicazione è completamente senza stato, ossia non viene mantenuto alcuno stato di interazione con gli utenti, l'high availability può essere ottenuta semplicemente tramite il load balancing: infatti se un server smette di funzionare le richieste vengono reindirizzate verso altri server e gli utenti non percepiscono discontinuità nel servizio. Se invece l'applicazione mantiene stato di interazione con gli utenti è un errore inviare le successive richieste di un utente verso un server diverso da quello che contiene il suo stato, per questo di solito si utilizza la strategia di load balancing Sticky Session. Comunque quando l'utente deve poter continuare il proprio lavoro anche quando un server subisce un malfunzionamento deve essere messo in atto un meccanismo di *failover* che permette all'utente di proseguire anche quando il server con il quale aveva iniziato l'interazione non è più disponibile. Il meccanismo di failover è ancora una volta una funzionalità del load balancer (hardware o software) che è in grado di capire quando un server non è più disponibile e indirizzare le richieste verso altri server. Come accennato in precedenza, i massimi livelli di high availability si ottengono con cluster orizzontali, in quanto con cluster verticali il sistema non è in grado di sostenere fallimenti del sistema operativo, della rete, dell'alimentazione o dell'hardware.

1.5 Fault tolerance e Replicazione dello stato

Il concetto di fault tolerance è strettamente legato a quello di high availability, infatti un'applicazione si dice fault tolerant se è "altamente disponibile" e continua a servire le richieste senza problemi anche se un server fallisce. Agli utenti è garantito il corretto comportamento dell'applicazione anche se tra una richiesta e la successiva il server si guasta. Nel caso di applicazioni senza stato la fault tolerance si ottiene garantendo la high availability (solo load balancing). Nel caso invece di applicazioni con stato la fault tolerance implica che il meccanismo di failover sia abilitato e che lo stato di interazione sia disponibile sul nuovo server che accoglie le richieste dell'utente dopo il fallimento del primo.

In generale due tipi di dato sono associati allo stato di interazione tra utente e server: dati di sessione e dati di entity. I dati di sessione sono associati direttamente all'utente e sono mantenuti dal server (in memoria RAM o eventualmente passivati) con il quale l'utente sta interagendo. I principali componenti che mantengono tali dati sono la sessione HTTP e i bean Statefull. I dati di entity sono invece "di proprietà" del DB, nel senso che la copia principale è mantenuta nel DB ma le applicazioni utilizzano alcune cache in memoria per limitare il numero di accessi al DB stesso. Un esempio di componenti utilizzati per gestire tali dati sono gli Entity bean EJB3. Un'applicazione, per essere fault tolerant, deve garantire che lo stato di interazione tra utente e server sia ridondante e disponibile anche in caso di guasto. Questa ridondanza si ottiene replicando i dati di sessione su nodi di un cluster diversi da quello con cui l'utente sta correntemente interagendo. In sostanza

un'applicazione è fault tolerant se si ha failover e replicazione dello stato, come schematizzato nella figura sottostante. Quando il primo server si guasta le richieste vengono indirizzate verso il secondo e dato che lo stato degli utenti era stato replicato prima del guasto, gli utenti possono continuare il proprio lavoro sul secondo server senza discontinuità.

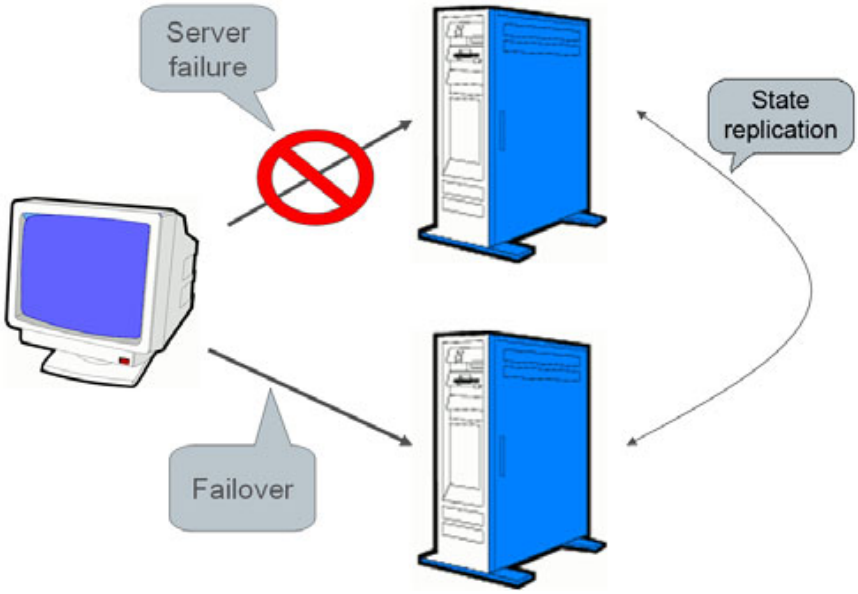


Figura 1.4 - Fault tolerance

Per rendere fault tolerant i dati di entity invece, di solito si cerca di rendere "altamente disponibile" il DB stesso tramite tecniche che sono proprie di ogni DBMS.

2. Applicazione di riferimento: myAlma

Per poter mettere in pratica i concetti riguardanti il clustering è ovviamente necessaria un'applicazione che possa essere ospitata da diversi application server per formare un cluster appunto, si è quindi deciso di realizzare un'applicazione web: myAlma. myAlma è intesa come un Content Management System (CMS) a supporto dei docenti e degli studenti universitari, infatti ha l'obbiettivo di migliorare, da un lato la gestione del materiale per un corso e dall'altro la fruizione dello stesso. Tramite un'unica applicazione i docenti potranno fornire materiali importanti per i corsi e gli studenti potranno accedere ad essi in modo uniforme.

myAlma offre ai docenti una piattaforma in cui è possibile inserire tutte le informazioni utili per un determinato corso (dispense, esercizi, ...) e organizzarle in modo appropriato. I docenti non si dovranno più occupare della realizzazione in prima persona di pagine web illustrative, myAlma offre infatti un editor integrato utilizzabile per realizzare documenti da caricare sulla specifica pagina di un corso (es. programma, obbiettivi, ...). I docenti non si occuperanno neanche dell'archiviazione dei file utili al corso, sarà sufficiente caricarli su myAlma e categorizzarli in modo opportuno. I docenti inoltre potranno comunicare tempestivamente agli studenti informazioni interessanti/utili mediante l'invio di opportuni avvisi.

Dall'altro lato gli studenti sapranno che tutte le informazioni e i materiali necessari per un corso possono essere trovati su myAlma. Gli studenti riceveranno una notifica (sull'applicazione, via mail o via sms) di ogni modifica/aggiunta di informazioni/materiali/ avvisi restando così costantemente aggiornati sull'evoluzione del corso. Inoltre, dato che la rappresentazione delle informazioni è in larga misura a carico di myAlma, gli studenti sapranno esattamente come navigare tra le informazioni.

2.1 Funzionalità

Entrando più nello specifico myAlma permette la pubblicazione di tre tipi di contenuti:

- **Materiale**: file di vario tipo (audio, video, testo) che è effettivamente caricato sul server.
- **Informazione**: contenuto informativo riguardante l'insegnamento in generale o altri argomenti in particolare.
- **Notizia/Avviso**: comunicazione breve e diretta con i fruitori dell'insegnamento.

Ogni contenuto viene classificato mediante una gerarchia di categorie costruendo così un albero in cui i nodi foglia rappresentano i contenuti veri e propri mentre i nodi intermedi rappresentano le categorie. Ogni contenuto deve avere un titolo e una descrizione e il sistema si occupa di tenere traccia di chi ha creato il contenuto e di chi lo ha modificato l'ultima volta, oltre alle relative date.

Ogni contenuto è associato ad un insegnamento che ha uno ed un solo docente che svolge il ruolo di titolare e può avere zero o più assistenti. Il titolare per un determinato

insegnamento è l'unico a poter aggiungere e rimuovere assistenti. Solo i docenti che fanno parte degli assistenti o il titolare stesso di un insegnamento possono aggiungere, modificare o eliminare contenuti per quell'insegnamento.

Ogni studente può "abbonarsi" ad un insegnamento nel momento in cui vuole ricevere informazioni (notifiche) riguardanti l'insegnamento stesso, e può "Archiviare un corso" quando non desidera più ricevere aggiornamenti (esame superato).

Ogni azione di aggiunta, modifica o rimozione di un contenuto viene notificata agli studenti abbonati all'insegnamento. Gli studenti possono disabilitare la ricezione delle notifiche per qualsiasi tipologia di contenuto tranne che per le Notizie. Nel momento in cui una notifica viene letta questa viene eliminata dal sistema e quindi non si è in grado di ricostruire la storia delle notifiche passate. Gli studenti possono decidere su quale mezzo ricevere le notifiche: al momento solo tramite mail.

myAlma può essere suddivisa in due macro-parti ben definite, la prima riguarda l'interazione tra l'applicazione e un docente e la seconda invece quella tra l'applicazione e uno studente. Per non complicare eccessivamente l'applicazione stessa è stato deciso di concentrarsi sulla parte relativa al docente, lasciando la parte relativa allo studente semi-implementata.

In particolare per quanto riguarda il docente sono state implementate sia mediante componenti lato server che lato client (tramite pagine web nella fattispecie) le seguenti funzionalità:

- Scelta tra uno degli insegnamenti di cui si è titolare o assistente;
- Esplorazione dei contenuti associati ad un determinato insegnamento;
- Possibilità di aggiunta/modifica/eliminazione di contenuti relativi ad un determinato insegnamento;
- Possibilità di salvare in modo non definitivo le modifiche apportate ad un contenuto fino a quel momento (salvataggio in sessione).

Di seguito sono riportati alcuni screenshot dell'applicazione.

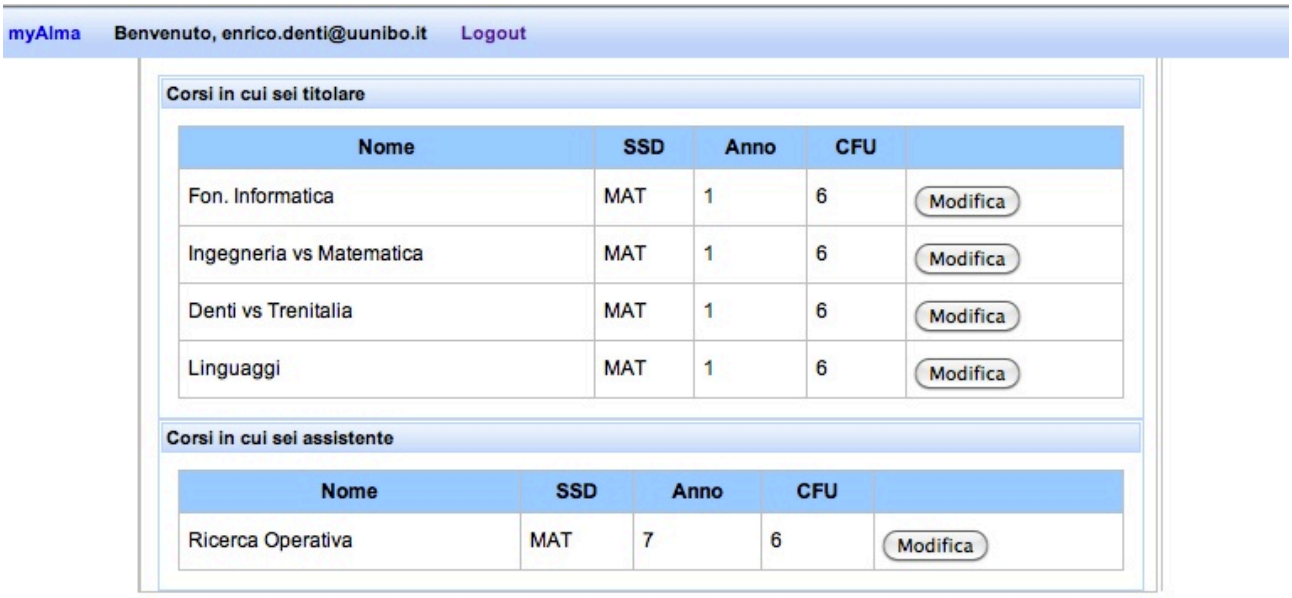


Figura 2.1 - Elenco dei corsi di cui si è titolare e assistente

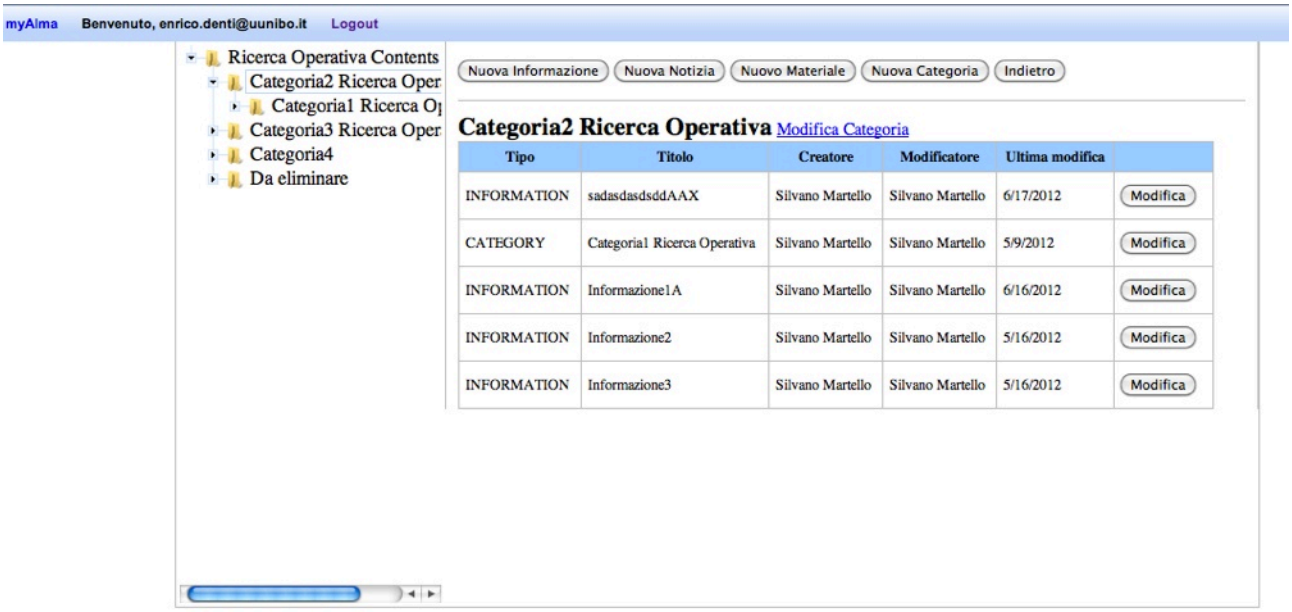


Figura 2.2 - Esplorazione di un corso

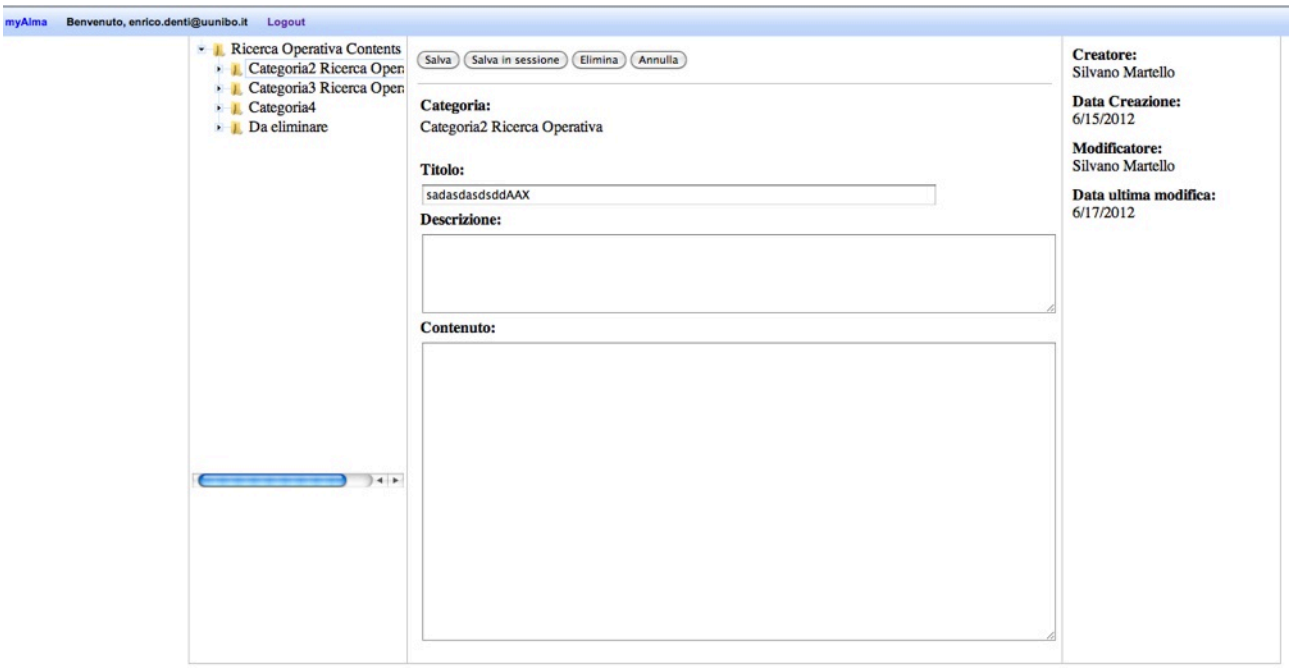


Figura 2.3 - Modifica di un contenuto

Per quanto riguarda invece la parte relativa allo studente sono state implementate lato server solo le funzionalità per la sottoscrizione/desottoscrizione ad un corso e il salvataggio in modo persistente delle notifiche e il loro invio tramite mail ma non sono state realizzate le interfacce web per l'utilizzo di tali funzionalità.

2.2 Architettura

L'applicazione è stata strutturata su tre livelli: il web tier, l'EJB tier e il data tier. Per quanto riguarda il web tier e quindi il livello di presentazione sono state scelte le tecnologie JSF (Mojarra 1.2) e RichFaces 3.3.3.Final per implementare l'interfaccia web che sarà

mostrata sul browser ed inoltre è stato impiegato il framework JBoss Seam 2.2.2.Final che ha il vantaggio (tra gli altri) di permettere l'utilizzo diretto di componenti EJB 3.x come backing bean per le pagine JSF, risparmiando così la necessità di implementare componenti che hanno il solo scopo di fare da tramite tra le pagine e i componenti EJB.

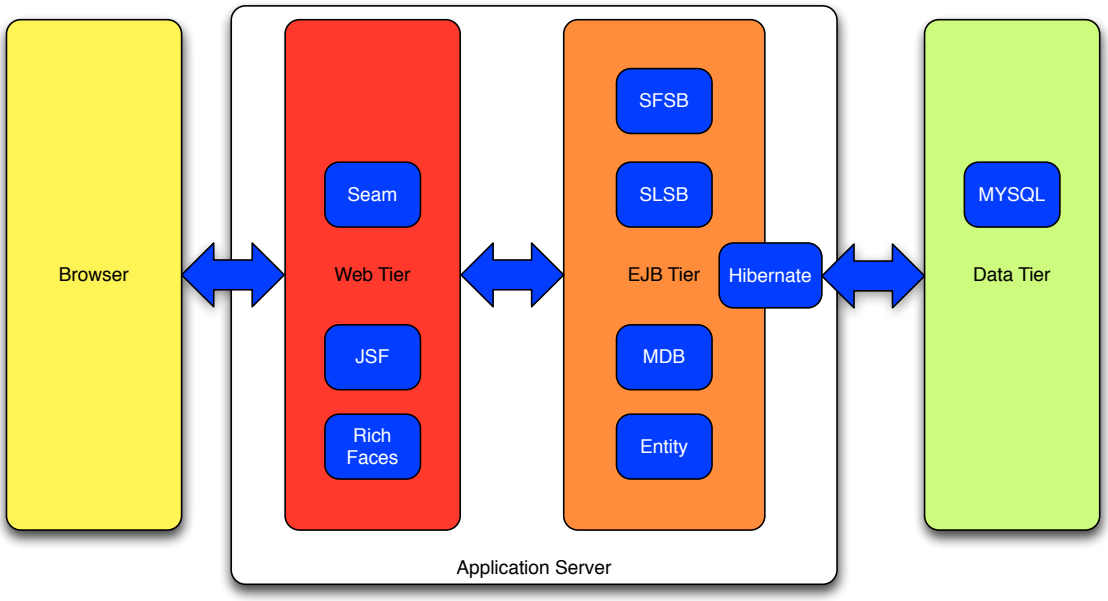


Figura 2.4 - Architettura dell'applicazione

Per quanto riguarda il livello applicativo sono state utilizzate le tecnologie e i componenti tipici dell'ambiente JEE come Stateless e Statefull session beans e message driven beans. Per l'interazione con il livello dei dati è stata utilizzata la tecnologia JPA 2.0 che permette di effettuare un mapping object-relational molto semplice mediante l'utilizzo di annotazioni Java. Dato che JPA è solo una specifica, come vendor è stato utilizzato Hibernate 4.0.1.Final che è già integrato e disponibile nell'application server JBoss 7.1.1.Final che ospita sia il web che l'EJB tier. Infine come DBMS per il salvataggio dello stato persistente è stato utilizzato l'open source MYSQL 5.1.31.

Lo stesso approccio a livelli è stato adottato anche in fase di progettazione, come è possibile infatti vedere dalla Figura 2.5 l'applicazione è stata suddivisa in 3 package: uno per il modello dei dati (it.unibo.myalma.model), uno per la parte applicativa (it.unibo.myalma.business) a sua volta suddiviso in ulteriori package e uno per i componenti utilizzati nella pagine web (it.unibo.myalma.ui). A parte si trovano poi le vere e proprie pagine JSF dell'interfaccia web e tutti i contenuti relativi ad essa, come icone, pagine di errore e descrittori.

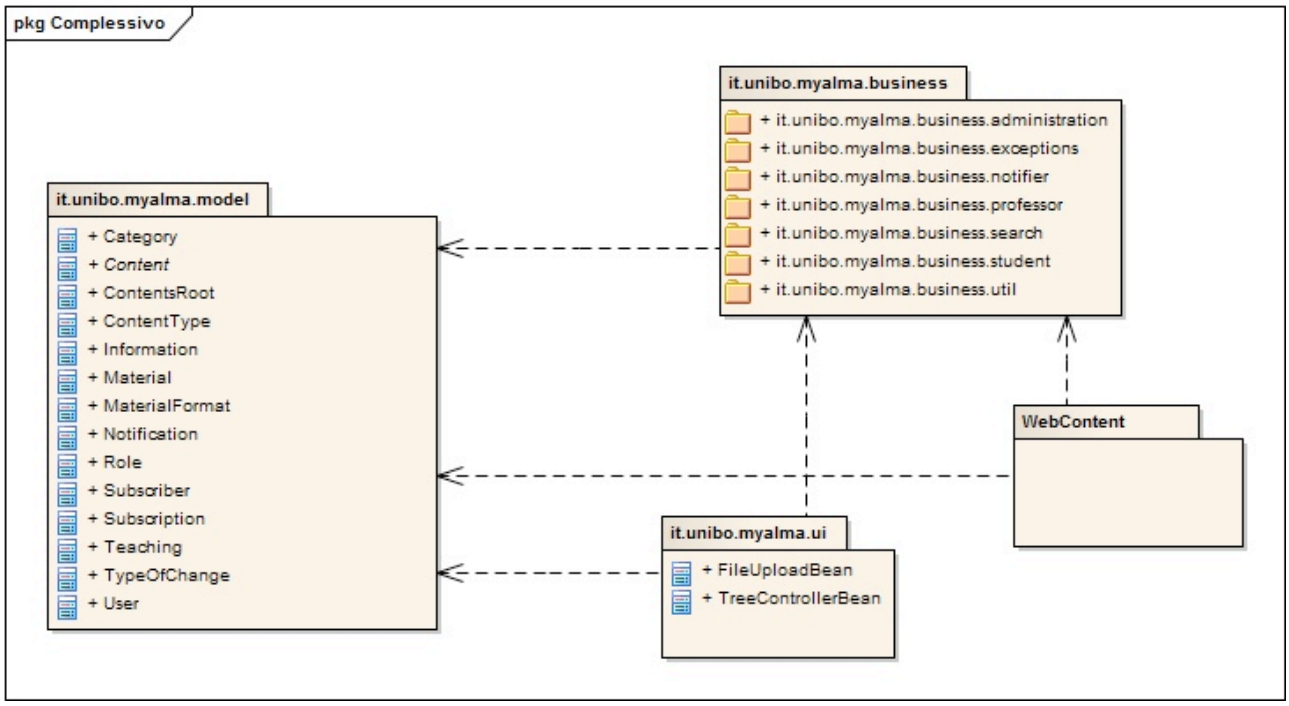


Figura 2.5 - Organizzazione packages

La divisione dell'applicazione in tre livelli distinti si è riflessa anche nella strutturazione dei progetti Eclipse, infatti è presente un progetto per ogni livello (web, EJB e JPA) oltre ad un progetto EAR che raccoglie i diversi moduli e i relativi descrittori di deployment in un unico archivio. E' inoltre presente un progetto (myalma-test) che racchiude tutti i test di unità (JUnit 4) dell'applicazione.

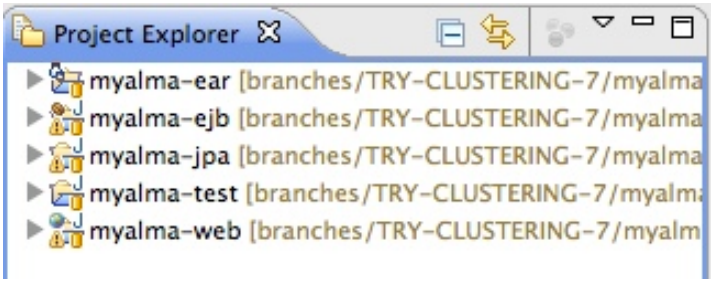


Figura 2.6 - Progetti in Eclipse

Nel seguito saranno analizzate più nel dettaglio le varie parti costituenti l'applicazione.

2.3 Modello dei dati

Come è possibile notare dalla Figura 2.5 tutti i livelli dell'applicazione dipendono dal livello dei dati, infatti le pagine web sono responsabili della visualizzazione e presentazione dei dati mentre il livello applicativo è responsabile della loro manipolazione, quindi ci si è concentrati inizialmente su questa parte del progetto.

Durante la progettazione l'obiettivo che si è cercato di raggiungere è il riutilizzo di quanti più componenti possibile, infatti nel diagramma delle classi di Figura 2.7 possono essere individuate tre sezioni che potrebbero essere riutilizzate singolarmente in altri progetti. La prima è la tassonomia che rappresenta i diversi tipi di contenuti, in cui è stato utilizzato il

pattern composite che ben si presta alla rappresentazione di strutture ad albero. La classe fondamentale è la superclasse astratta Content che incapsula tutte le informazioni relative ad un qualsiasi contenuto come, il titolo, la descrizione, l'utente creatore e via dicendo. Tale classe presenta anche le operazioni base che possono essere eseguite su un contenuto, come l'aggiunta o la rimozione di un contenuto, in particolare però tali metodi non sono implementati direttamente in *Content* ma sono stati aggiunti in questa classe per permettere una visione uniforme dei contenuti tramite l'interfaccia di *Content* appunto. Le dirette sottoclassi di *Content* sono *Category*, *Material* e *Information*: la prima rappresenta una categoria ed implementa le operazioni di aggiunta e rimozione di contenuti citate precedentemente, le altre due classi invece rappresentano i rispettivi tipi di contenuti. Per quanto riguarda il tipo Notizia, non è stata predisposta una nuova classe ma verrà utilizzata la classe *Information*: le due istanze saranno riconosciute in base al valore di *ContentType* con le quali sono costruite (NOTICE o INFORMATION). La classe *Category* viene ulteriormente subclassata dalla classe *ContentsRoot* che rappresenta la radice dell'albero dei contenuti e che aggiunge i concetti di editore (docente titolare nella nomenclatura myAlma) e autori (assistenti in myAlma). Rimanendo indipendenti da classi tipiche del dominio in esame, come ad esempio docente o studente, ed utilizzano soltanto il concetto di *User* è possibile riutilizzare la gerarchia di classi appena illustrata anche in altre applicazioni che necessitino di una organizzazione ad albero di certi contenuti.

La seconda parte che potrebbe essere riutilizzata è quella relativa alla modellazione degli utenti e dei loro ruoli, infatti tali classi sono del tutto generali e non hanno legami con classi specifiche del dominio.

Infine l'ultima sezione che presenta un buon grado di riutilizzo è quella relativa alle sottoscrizioni e alle notifiche, infatti dopo aver modellato una notifica come un'entità che comprende un messaggio, una data di creazione e una tipologia di cambiamento avvenuto su un contenuto (inserimento, rimozione e modifica) si è rappresentato un "abbonamento" mediante la classe *Subscription* che contiene un riferimento all'insegnamento interessato dall'abbonamento e una lista di notifiche non ancora lette. Per associare all'utente la possibilità di abbonarsi ad un corso si è scelto di realizzare una sottoclasse della classe *User* che includa una lista di sottoscrizioni (classe *Subsciber*): in questo modo non si è sporcata la rappresentazione generale dell'utente. Come detto in precedenza anche questa sezione presenta un discreto livello di riutilizzo con l'unico svantaggio però che è legata alla classe specifica del dominio *Teaching* e quindi probabilmente dovrà subire una modifica per poter essere integrata in altri progetti.

Come scelta progettuale si è deciso di specificare la modalità LAZY per il caricamento dal DB delle collezioni referenziate da classi del modello. Con tale modalità le collezioni non sono caricate nello stesso momento in cui la classe interessata è caricata (EAGER) ma solo nel momento in cui sono richieste. Tale modalità è pressoché indispensabile in caso di struttura dati ad albero per evitare che il caricamento di un nodo dell'albero non provochi il caricamento in cascata di tutto il suo sottoalbero anche se questo non viene mai utilizzato dall'utente. Tale modalità inoltre è applicabile grazie al concetto di contesto di persistenza esteso che consiste nel mantenere gli entity beans sotto il controllo dell'entity manager anche quando la transazione corrente termina, in questo modo quando una collezione è richiesta, sarà caricata automaticamente, senza esplicitamente programmare tale comportamento, se ne occuperà l'entity manager.

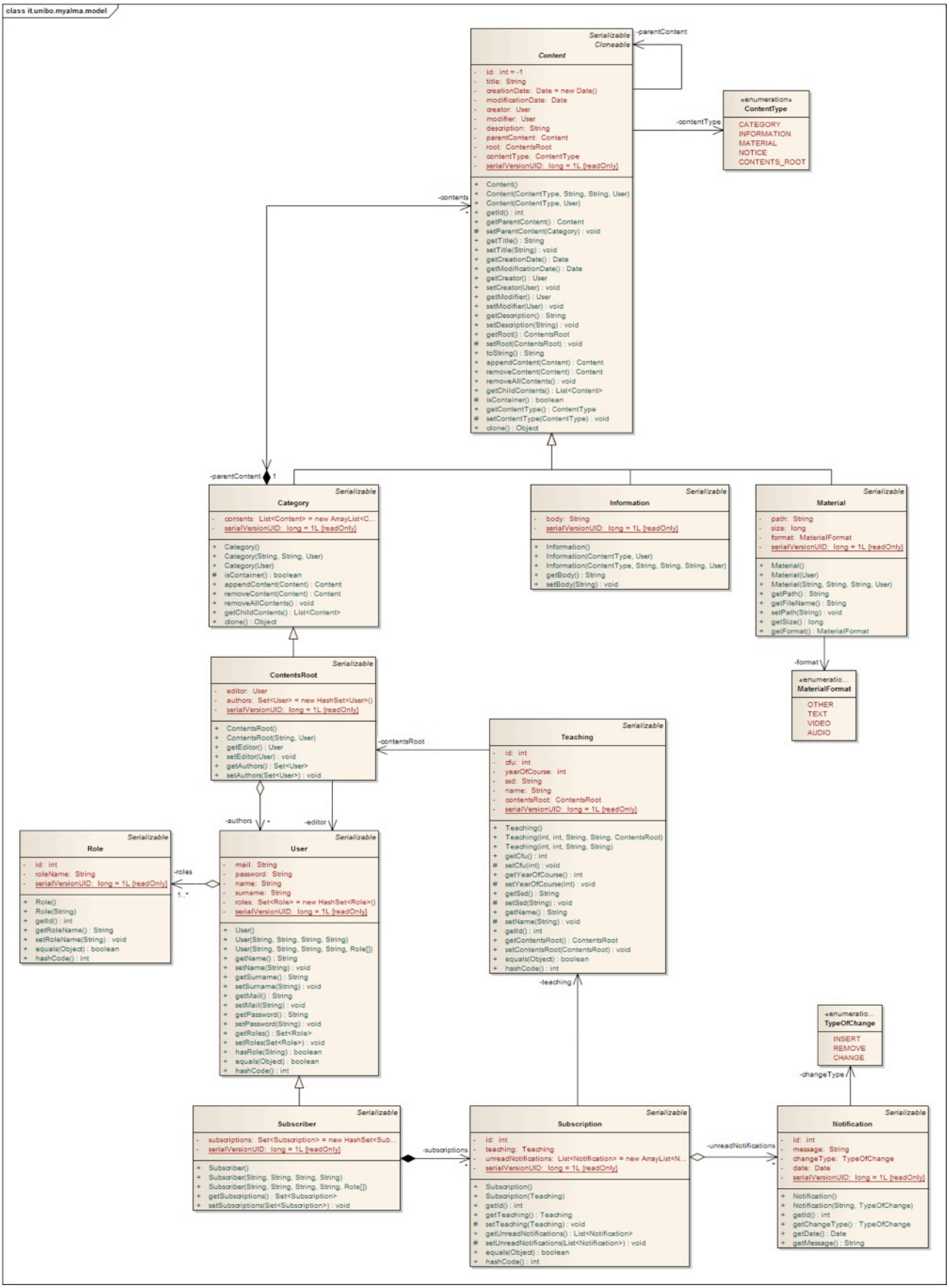


Figura 2.7 - Modello dei dati

Per semplicità è stato deciso di lasciar generare ad Hibernate il data base partendo dalle informazioni specificate in fase di codifica delle classi del modello mediante annotazioni. Tale tecnica è sicuramente adeguata in fase di prototipazione in quanto permette di ottenere in un tempo praticamente nullo un DB perfettamente definito ma non è assolutamente adeguata in fase di produzione in cui è necessario porre massima cura nella definizione del DB.

2.3.1 Configurazione

Per poter utilizzare un DB da un application server come JBoss è prima di tutto necessario configurare il relativo datasource. JBoss 7 presenta un unico file per le configurazioni di tutti i servizi del server, tale file è localizzato in JBOSS_HOME/standalone/configuration/standalone.xml o in JBOSS_HOME/domain/configuration/domain.xml a seconda della modalità utilizzata. In particolare per poter aggiungere un nuovo datasource è necessario aggiungere la sua definizione al sottosistema *urn:jboss:domain:datasources:1.0* come indicato nella Figura 2.8.

```
<datasource jndi-name="java:/myalma-ds" pool-name="myalma-ds"
            enabled="true" use-java-context="true">
    <connection-url>jdbc:mysql://localhost:3306/sd10db</connection-url>
    <driver>mysql</driver>
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
    <pool>
        <min-pool-size>20</min-pool-size>
        <max-pool-size>35</max-pool-size>
        <prefill>true</prefill>
        <use-strict-min>false</use-strict-min>
    </pool>
    <security>
        <user-name>sd10user</user-name>
        <password>sd10pwd</password>
    </security>
    <validation>
        <validate-on-match>false</validate-on-match>
        <background-validation>false</background-validation>
    </validation>
</datasource>
```

Figura 2.8 - Configurazione Datasource

In questa configurazione è necessario specificare l'url di connessione con il DB, le credenziali di accesso al data base stesso e il riferimento al driver che dovrà essere utilizzato per le comunicazioni tra application server e DB. Un'altra configurazione importante riguarda il pool di connessioni verso il DB. E' possibile scegliere un numero minimo di connessioni che saranno create all'avvio dell'appication server, risparmiando così l'overhead in cui si incorre nella creazione di altre connessioni quando il server è carico e necessita di altre connessioni, è inoltre possibile specificare un numero massimo di connessioni gestibili dall'application server per evitare ad esempio di sovraccaricare il DB con troppe connessioni. L'ultimo parametro è molto importante soprattutto in un cluster perché se non si imposta correttamente per ogni server si rischia che il numero di connessioni gestite dai server superi il numero massimo di connessioni offerte dal DB incorrendo così in errori a runtime.

Il secondo passo per poter utilizzare il DB è quello di definire l'unità di persistenza che rappresenta l'insieme di classi mappate tramite ORM su uno specifico DB. Tale definizione viene effettuata tramite il file persistence.xml che deve essere posto nella cartella META-INF dell'archivio che contiene gli entity beans. Come è possibile notare dalla Figura 2.9 le opzioni più importanti sono quelle che riguardano il nome dell'unità di persistenza (obbligatorio), il provider, ossia l'effettiva implementazione di JPA (in questo caso Hibernate) e il riferimento al datasource precedentemente configurato (è necessario specificare il nome jndi impostato in standalone.xml o domain.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="myalma-jpa">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>java:/myalma-ds</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.archive.autodetection" value="class"/>

            <!-- Per poter referenziare la factory in Seam (vedi components.xml) -->
            <property name="jboss.entity.manager.factory.jndi.name" value="myalma-jpaManagerFactory"/>
        </properties>
    </persistence-unit>
</persistence>
```

Figura 2.9 - Configurazione Unità di Persistenza

All'interno del file persistence.xml è anche possibile impostare proprietà che sono specifiche del particolare provider in uso, nel file mostrato in Figura 2.9 viene impostato ad esempio il dialetto di MYSQL 5 permettendo ad Hibernate di utilizzare quel particolare DBMS senza problemi, si specifica tramite la proprietà *hibernate.hbm2ddl.auto* che ad ogni deploy dell'applicazione lo schema del DB deve essere aggiornato con le modifiche apportate alle classi entity (è possibile anche impostarlo a *create*, in quel caso il DB viene eliminato e ricreato completamente, perdendo quindi tutti i dati che vi erano presenti, con la modalità *update* invece i dati non vengono persi). Infine viene configurato HIbernate per effettuare una scansione dell'archivio JAR che contiene il file di configurazione per individuare autonomamente le classi che rappresentano entity beans le quali devono essere annotate con *@Entity*.

2.4 Business Tier

Per quanto riguarda il livello applicativo si è cercato di suddividere i componenti in base al loro ruolo all'interno dell'applicazione (Figura 2.10). Come accennato in precedenza i package su cui si è posta maggiore attenzione sono quelli relativi la gestione dei contenuti lato docente, quindi in particolare, *it.unibo.myalma.business.professor* che contiene i bean utilizzati per apportare modifiche all'albero dei contenuti, *it.unibo.myalma.business.search* che contiene il bean utilizzato per interrogare il DB ed esplorare quindi l'albero dei contenuti e infine *it.unibo.myalma.business.notifier* che contiene invece i componenti utilizzati per generare e inviare le notifiche agli studenti abbonati agli insegnamenti.

Per quanto riguarda i package *it.unibo.myalma.business.administration* e *it.unibo.myalma.business.student* sono state implementate solo le funzionalità base per

l'amministrazione (creazione di insegnamenti e aggiunta di utenti e ruoli) e per la gestione lato studente (abbonamento e archiviazione di un insegnamento).

Infine nel package *it.unibo.myalma.business.util* sono state implementate alcune classi di utilità, che si occupano della gestione dei file caricati sul server e della individuazione dell'utente correntemente loggato all'interno dell'applicazione.

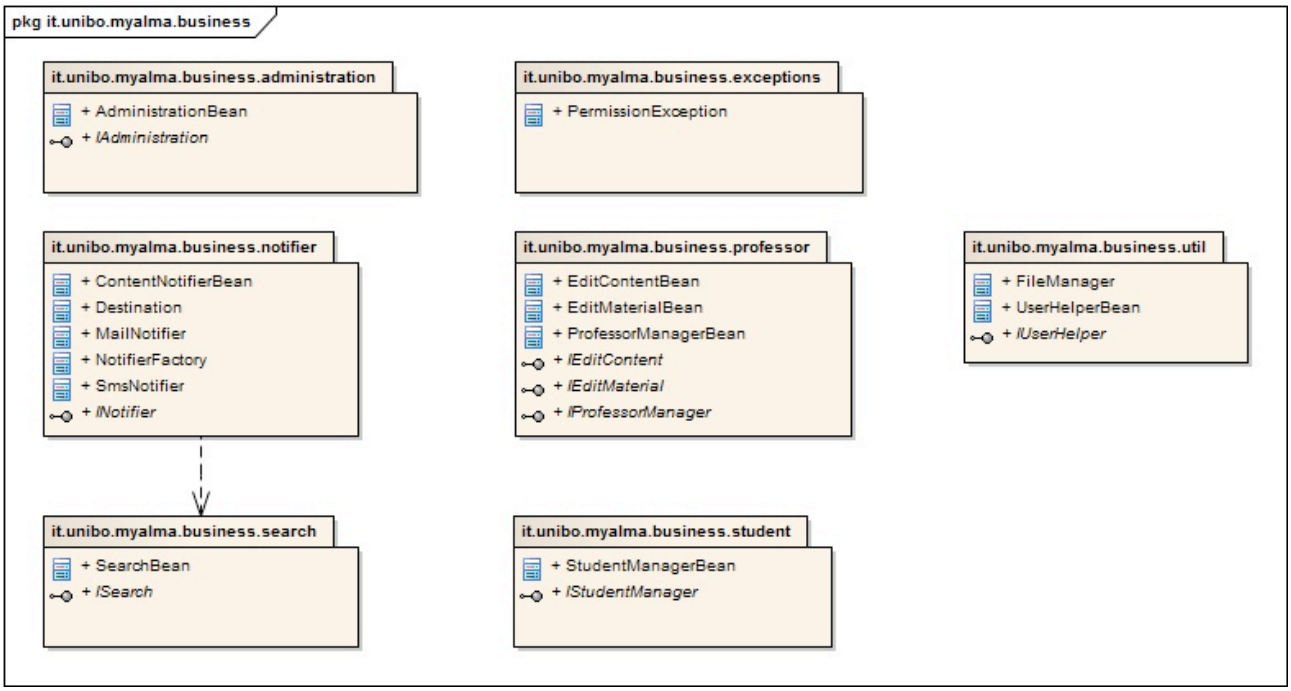


Figura 2.10 - Organizzazione Package Business Tier

Dato che sia il web tier che l'ejb tier sono integrati all'interno dello stesso application server la comunicazione tra i componenti e le pagine web avverrà tramite interfacce locali (*@Local*) evitando così l'overhead di invocazioni remote.

Nel seguito saranno analizzati più in dettaglio i componenti relativi la parte di gestione dei contenuti e delle notifiche.

2.4.1 Stateless Beans

Il componente principale per la gestione dei contenuti è lo Stateless bean *it.unibo.myalma.business.professor.ProfessorManagerBean*, infatti tale componente presenta le operazioni di aggiunta, rimozione e modifica di un qualsiasi contenuto implementando anche le regole di business indicate precedentemente secondo le quali solo il docente titolare di un'insegnamento o i suoi assistenti possono modificare l'albero dei contenuti ed eventualmente solo il titolare può modificare l'insieme degli assistenti (la sezione sulla sicurezza 2.4.5 descrive più in dettaglio la soluzione a tale problema). In tale componente sono state anche implementate le operazioni necessarie per inviare un messaggio JMS ad un topic che si occupa di raccogliere i messaggi di notifica da inviare successivamente agli studenti (sezione 2.4.4 per maggiori dettagli), ogni operazione di modifica dell'albero scatena la creazione e il successivo invio di un messaggio verso il topic *java:jboss/exported/jms/topic/contentEvents*. Il fatto di concentrare in tale componente la logica di invio di messaggi JMS è favorevole in quanto chiunque voglia

modificare l'albero dei contenuti dovrà utilizzare tale componente e quindi non si dovrà occupare dell'invio dei messaggi ma potrà fare affidamento sul componente stesso per tale operazione.

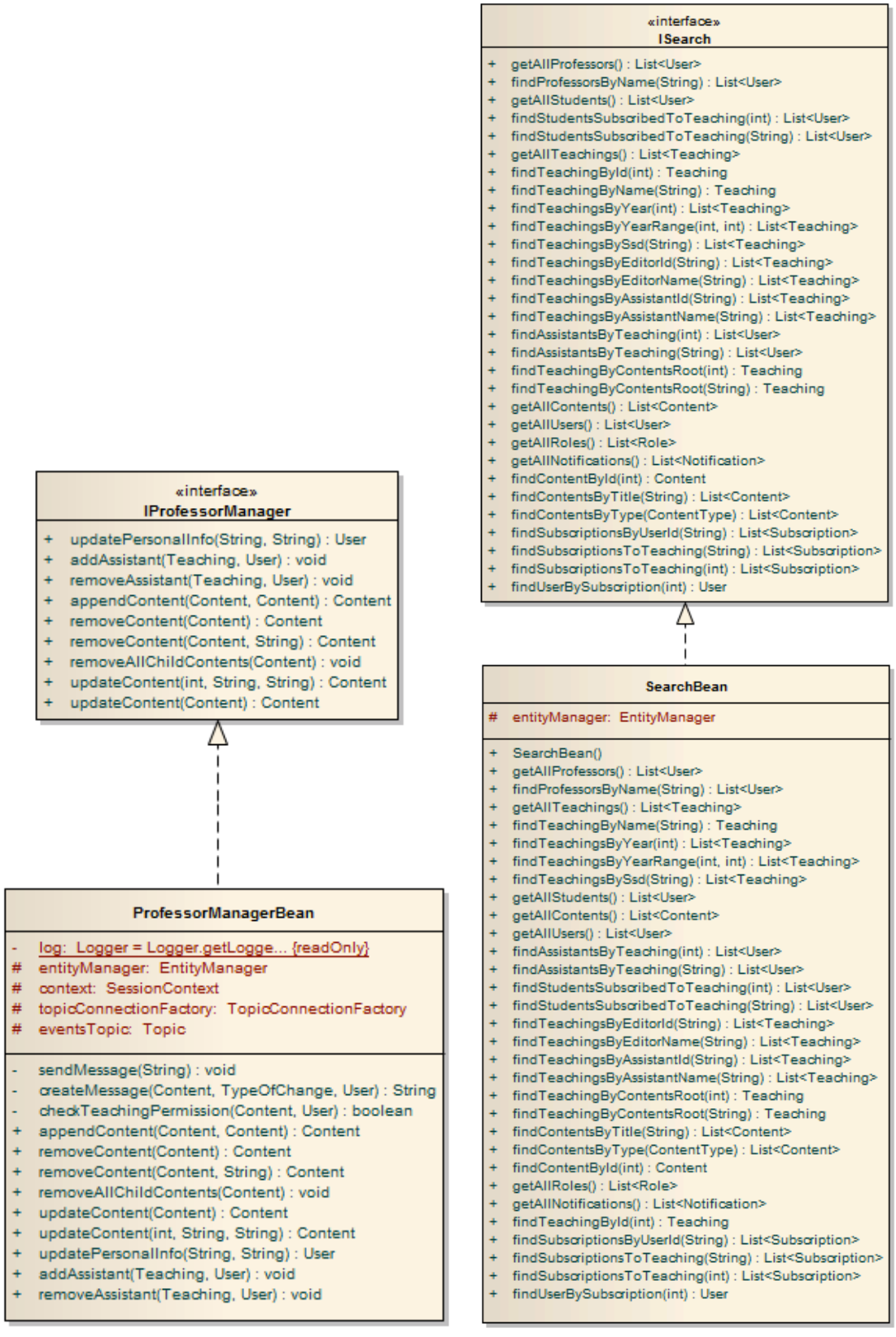


Figura 2.11 - a) Stateless Bean Gestione Contenuti - b) Stateless Bean Interrogazione DB

Un altro componente Stateless importante è *it.unibo.myalma.business.search.SearchBean* (Figura 2.11b), il quale si occupa di racchiudere in un'unico componente tutte le operazioni fondamentali di interrogazione del DB. Alcune operazioni importanti offerte da tale bean sono quelle che permettono di ottenere tutti gli abbonamenti per un determinato insegnamento oppure tutti gli studenti che sono abbonati ad un insegnamento, inoltre è possibile cercare tra i contenuti e gli insegnamenti utilizzando diverse chiavi di ricerca come il titolo, il tipo, l'anno e via dicendo. Tale bean può essere utilizzato sia lato docente che lato studente in quanto sfruttando i meccanismi di sicurezza messi a disposizione da JEE (sezione 2.4.5) gli studenti non potranno accedere alle informazioni riservate ai docenti e agli amministratori, come ad esempio quelle relative tutti gli utenti del sistema.

2.4.2 Statefull Beans

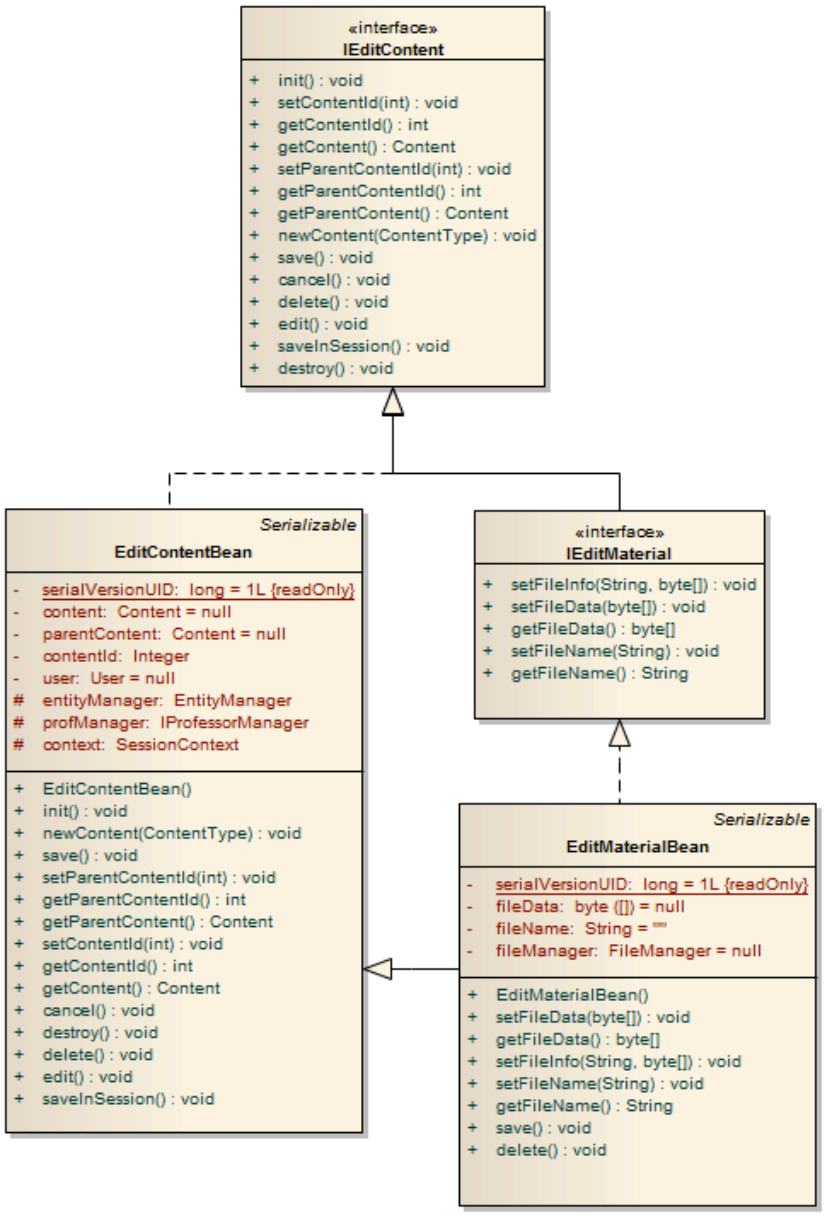


Figura 2.12 - Statefull beans Modifica di Contenuti

Per poter mantenere le modifiche apportate su un contenuto senza però renderle persistenti sono stati impiegati due Statefull bean, uno per la gestione di un contenuto

generico che può essere un'informazione, una notizia o una categoria e l'altro invece per la gestione di un materiale che richiede delle operazioni specifiche. Il componente principale è *it.unibo.myalma.business.professor.EditContentBean* che mantiene un riferimento al contenuto che si sta modificando (campo *content*) e al suo padre all'interno dell'albero (campo *parentContent*): solo nel momento in cui viene invocato il metodo *save ()* il contenuto è effettivamente reso persistente tramite l'utilizzo del bean Stateless *ProfessorManagerBean* illustrato precedentemente, gestendo correttamente anche il caso in cui il contenuto sia stato spostato in un'altra posizione dell'albero. Il componente *it.unibo.myalma.business.professor.EditMaterialBean* invece deriva da *EditContentBean* e aggiunge le operazioni specifiche per il salvataggio di un file, come il suo nome e la sua rappresentazione sotto forma di array di byte.

2.4.3 Message Driven Beans

Come citato in precedenza le notifiche agli studenti sono create dal componente *it.unibo.myalma.business.professor.ProfessorManagerBean* sotto forma di messagi JMS e successivamente inviati ad un opportuno topic all'interno del sistema. Innanzi tutto è stato scelto un topic e non una queue perché si prevede che in futuro potrebbero essere aggiunti altri listener di messaggi JMS e quindi il topic che è per sua natura un mezzo di comunicazione molti-a-molti è più adeguato. Per garantire una certa affidabilità nella ricezione dei messaggi, si è scelto di adottare una Durable Subscription, in questo modo se il ricevente dei messaggi (un Message Driven Bean in questo caso) non è attivo quando un messaggio viene prodotto, il messaggio non viene perso ma sarà conservato dal provider JMS fino a che il destinatario non ritorna attivo. Nel caso di JBoss 7.1.1.Final il provider JMS integrato all'interno dell'application server è HornetQ 2.2.13.Final.

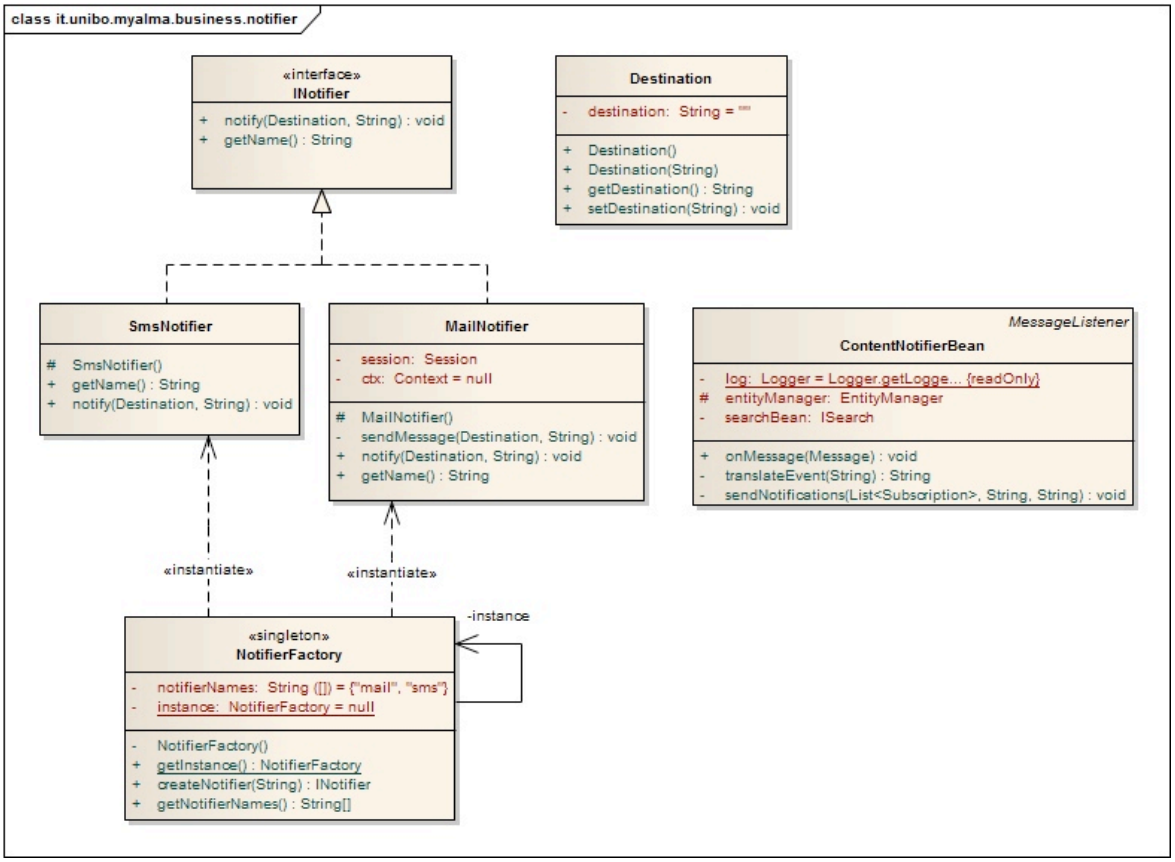


Figura 2.13 - Package it.unibo.myalma.business.notifier

I messaggi che raggiungono il topic sono poi gestiti dal Message Driven Bean *it.unibo.myalma.business.notifier.ContentNotifierBean* (Figura 2.13), il suo compito è quello di ricevere i messaggi JMS, creare le relative notifiche che dovranno poi essere rese persistenti sul DB associandole alle giuste sottoscrizioni e inviarle direttamente agli studenti tramite il mezzo che hanno scelto (ad esempio mail o sms). Per poter inviare le notifiche il MDB si appoggia su alcune classi che cercano di astrarre il concetto di notificatore e di destinazione della notifica. Infatti un notificatore è un'entità rappresentato dall'interfaccia (*INotifier*) ed è in grado di inviare messaggi sotto forma di stringhe ad una generica destinazione (classe *Destination*). L'interfaccia *INotifier* viene poi implementata da classi concrete che sono in grado di inviare messaggi tramite un ben preciso mezzo di comunicazione, ad esempio via mail (*MailNotifier*). Per mascherare agli utilizzatori dei notificatori le reali classi concrete è stata predisposta una Factory (*NotifierFactory*) che partendo dal nome del notificatore è in grado di crearne un'istanza concreta. La destinazione del messaggio è rappresentata da una stringa: in questo modo si è sufficientemente generali e si è in grado si rappresentare destinazioni come un indirizzo di posta elettronica o un numero di telefono senza troppi problemi.

Configurazione
Prima di poter utilizzare un topic per la ricezione di messaggi JMS è necessario configurarlo all'interno del sotto-sistema *urn:jboss:domain:messaging:1.1* inserendo la sua definizione nella sezione *<jms-destinations>* come mostrato nella successiva figura.

```
<jms-destinations>
    <jms-topic name="contentEvents">
        <entry name="java:jboss/exported/jms/topic/contentEvents"/>
    </jms-topic>
</jms-destinations>
```

Figura 2.14 - Definizione di un Topic

2.4.4 Sicurezza

Per evitare che utenti non autorizzati apportino modifiche non lecite ai contenuti è stato configurato il supporto alla sicurezza messo a disposizione da JBoss.

La specifica JEE definisce un semplice modello di sicurezza basato su ruoli sia per componenti EJB che per componenti web. L'implementazione di JBoss è delegata al framework PicketBox che fornisce servizi (tra gli altri) di autenticazione e autorizzazione. Un container JEE fornisce due tipi di sicurezza, quella dichiarativa e quella programmatica (dall'inglese programmatic), il primo prevede la definizione delle informazioni di sicurezza ("chi può fare cosa") tramite descrittori di deployment o annotazioni Java (*@RolesAllowed* e *@RunAs*), il secondo invece viene utilizzato quando la sicurezza dichiarativa non è sufficientemente espressiva per definire il modello di sicurezza dell'applicazione e consiste nell'utilizzo di opportune API per ottenere l'identità dell'utente corrente e la sua appartenenza ai ruoli richiesti. In myAlma sono stati utilizzati entrambi i tipi: quello dichiarativo è stato impiegato per evitare che ad esempio studenti (ruolo *student*) modifichino l'albero dei contenuti (per tali operazioni è necessario almeno il ruolo di *professor*), quello programmatico invece è stato utilizzato per esprimere la politica di sicurezza secondo la quale solo il titolare di un insegnamento e i suoi assistenti possono modificare i contenuti relativi a quel insegnamento, in quest'ultimo caso quindi per ogni

operazione eseguita sul sistema si determina l'identità dell'utente chiamante e si verifica se è il titolare o un assistente per l'insegnamento in oggetto (vedere metodo *checkTeachingPermission* in *ProfessorManagerBean*).

Per configurare un cosiddetto dominio di sicurezza è necessario agire sul sotto-sistema *urn:jboss:domain:security:1.1* (Figura 2.15). In un dominio di sicurezza sono configurati i moduli di autenticazione (*login module*) che sono utilizzati nel momento in cui una richiesta raggiunge l'application server per decidere se il chiamante può procedere oppure no.

```
<security-domain name="myalma-security-domain" cache-type="default">
    <authentication>
        <login-module code="Database" flag="required">
            <module-option name="dsJndiName" value="java:/myalma-ds"/>
            <module-option name="principalsQuery" value="SELECT password FROM users WHERE mail=?"/>
            <module-option name="rolesQuery" value="SELECT R.roleName, 'Roles'
                                                    FROM users_roles UR, roles R
                                                    WHERE UR.roles_id=R.id and UR.users_mail=?"/>
            <module-option name="hashAlgorithm" value="SHA"/>
            <module-option name="hashEncoding" value="rfc2617"/>
            <module-option name="hashCharset" value="UTF-8"/>
            <module-option name="hashUserPassword" value="true"/>
            <module-option name="hashStorePassword" value="false"/>
            <module-option name="passwordIsA1Hash" value="true"/>
            <module-option name="storeDigestCallback" value="org.jboss.security.auth.spi.RFC2617Digest"/>
            <module-option name="unauthenticatedIdentity" value="guest"/>
        </login-module>
    </authentication>
</security-domain>
```

Figura 2.15 - Configurazione Sicurezza

In particolare, come è possibile osservare in Figura 2.15, è stato impiegato un solo login-module chiamato *Database*, il quale permette di memorizzare e utilizzare le credenziali di accesso degli utenti e i loro ruoli su un data base relazionale. Tale modulo deve essere configurato indicando due query SQL (*principalsQuery* e *rolesQuery*) necessarie per ottenere la password e i ruoli per un determinato utente, specificando il proprio userid. In questo caso il data base utilizzato è lo stesso che contiene i contenuti degli insegnamenti e quindi viene inserito il riferimento al datasource *java:/myalma-ds* ma è comunque permesso, dove necessario, l'utilizzo di un database diverso. Per evitare di memorizzare le password degli utenti in chiaro sono state aggiunte alcune opzioni extra che indicano l'algoritmo utilizzato per calcolare l'hash della password (SHA in questo caso).

Per poter rendere sicuro anche l'accesso al topic per i messaggi JMS è necessario configurare opportunamente anche il sotto-sistema di messaging (*urn:jboss:domain:messaging:1.1*) indicando quali ruoli possono eseguire quali operazioni. Nella Figura sottostante, ad esempio, viene mostrato come l'operazione di *send* sul topic è permessa per i ruoli *jmssender*, *student*, *admin* e *professor*, mentre l'operazione *consume* è permessa solo agli amministratori.

```
<security-domain>myalma-security-domain</security-domain>
<security-settings>
    <security-setting match="jms.topic.contentEvents">
        <permission type="send" roles="jmssender student admin professor"/>
        <permission type="consume" roles="admin"/>
        <permission type="createDurableQueue" roles="admin"/>
        <permission type="deleteDurableQueue" roles="admin"/>
        <permission type="createNonDurableQueue" roles="student admin professor"/>
        <permission type="deleteNonDurableQueue" roles="student admin professor"/>
    </security-setting>
</security-settings>
```

Figura 2.16 - Configurazione Sicurezza JMS

2.5 Web Tier

Come descritto durante la presentazione dell'architettura dell'applicazione per agevolare l'implementazione del web tier è stato utilizzato il framework Seam, di seguito saranno brevemente elencate le quattro caratteristiche fondamentali di tale framework.

Innanzi tutto Seam mostra la sua validità permettendo di utilizzare componenti EJB 3.x direttamente all'interno di pagine JSF. Infatti utilizzando l'annotazione *@org.jboss.seam.annotations.Name* è possibile definire un qualsiasi componente EJB (o eventualmente un POJO) come un componente Seam permettendo così il suo utilizzo tramite l'Expression Language all'interno delle pagine JSF.

Altra caratteristica importante di Seam è quella di estendere i classici contesti JSF (Application, Session, Request e Page) aggiungendone due nuovi (Figura 2.17): *Coversation* e *Business Process*. Il secondo rappresenta un contesto ancora più ampio di quello di applicazione e può sopravvivere alla applicazione stessa. Il contesto Conversation invece permette di suddividere la sessione di un utente in contesti indipendenti l'uno dall'altro, di solito utilizzati quando l'utente deve portare a termine un determinato compito sull'applicazione. Nel caso di myAlma una nuova conversazione viene iniziata quando l'utente seleziona il corso da esplorare e una nuova conversazione, innestata nella prima, viene creata quando si seleziona un contenuto da modificare, in questo modo la gestione della sessione nel caso in cui un docente voglia esplorare un altro insegnamento o modificare un altro contenuto in parallelo al primo è molto semplice in quanto sarà sufficiente iniziare una nuova conversazione, le due saranno indipendenti e conterranno i rispettivi componenti. Senza il supporto delle conversazioni, uno scenario come il precedente si tradurrebbe nel salvataggio nella sessione dell'utente di un gran numero di oggetti che rappresentano i diversi stati dell'applicazione, con conseguente difficoltà nella loro gestione nel momento in cui l'applicazione cresce di complessità.

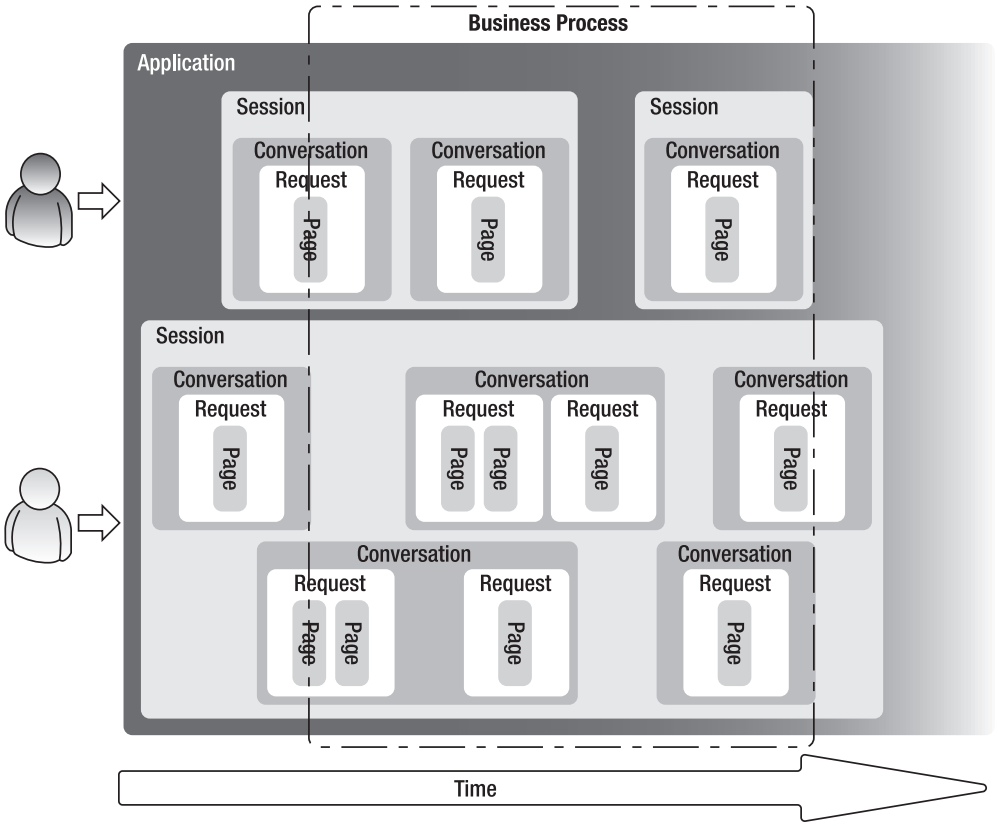


Figura 2.17 - Contesti in Seam

In Seam i contesti possono anche essere utilizzati agevolmente come mezzo di comunicazione tra componenti che vivono in contesti diversi, tramite il concetto di Biinjection. Utilizzando opportune annotazioni è possibile infatti associare un oggetto ad un particolare contesto (*@Out - Outjection*) oppure iniettare all'interno di un componente Seam l'oggetto presente in quel contesto (*@In - Injection*). Agli oggetti "esportati" in un contesto può essere associato un nome permettendo così il loro utilizzo anche in pagine JSF, in questo modo l'utente può modificare l'oggetto tramite l'interfaccia web e nel momento in cui invia una richiesta all'applicazione (ad esempio *submit* di un form) l'oggetto modificato è iniettato di nuovo dentro al componente Seam. Un esempio di tale utilizzo è rappresentato dai campi *content* e *parentContent* dello SFSB *EditContentBean*, in questo caso infatti i due contenuti sono esportati nel contesto di conversazione e utilizzati nella pagine JSF per mostrare i loro valori ma quando l'utente clicca il bottone *Salva* i due contenuti sono iniettati nuovamente in EditContentBean per poter essere salvati.

Ultima funzionalità significativa di Seam è quella che permette di associare ad una conversazione un contesto di persistenza esteso, garantendo così che gli entity beans utilizzati all'interno di quella conversazione non diventino detached sfruttando quindi appieno le potenzialità di JPA, tra le quali, soprattutto il lazy loading. Questo è possibile perché é Seam stesso a gestire l'Entity Manager (e non più il container JEE) e ad iniettarlo all'interno dei componenti. In myAlma questa potenzialità viene sfruttata condividendo l'Entity Manager tra i componenti che si occupano della visualizzazione dell'albero dei contenuti (Figura 2.2) e del salvataggio delle modifiche (TreeControllerBean, EditContentBean e ProfessorManagerBean). Con tale soluzione le modifiche apportate agli entity beans sono subito disponibili per tutti i componenti che condividono lo stesso

Entity Manager (e quindi anche quelli che mostrano i contenuti sulla interfaccia grafica) senza dover effettuare operazioni di refresh del proprio contesto di persistenza (di solito molto pesanti in termini di accesi al DB). Quando il contesto di persistenza è gestito da Seam è possibile inoltre impostare il *flush-mode* al valore *MANUAL* (vedere *WEB-INF/pages.xml*) per avere il controllo sul momento in cui le modifiche al contesto di persistenza sono riportate sul DB (EntityManager.flush()).

Grazie a Seam è stato quindi possibile utilizzare nelle pagine JSF i bean SearchBean, EditContentBean ed EditMaterialBean, comunque è stato necessario implementare due ulteriori componenti (POJO con annotazione Seam *@Name* - *Figura 2.18*) per gestire due controlli dell'interfaccia grafica: uno per l'albero dei contenuti (TreeControllerBean) e l'altro per il caricamento di file sul server (*FileUploadBean*), anch'essi associati al contesto di conversazione.

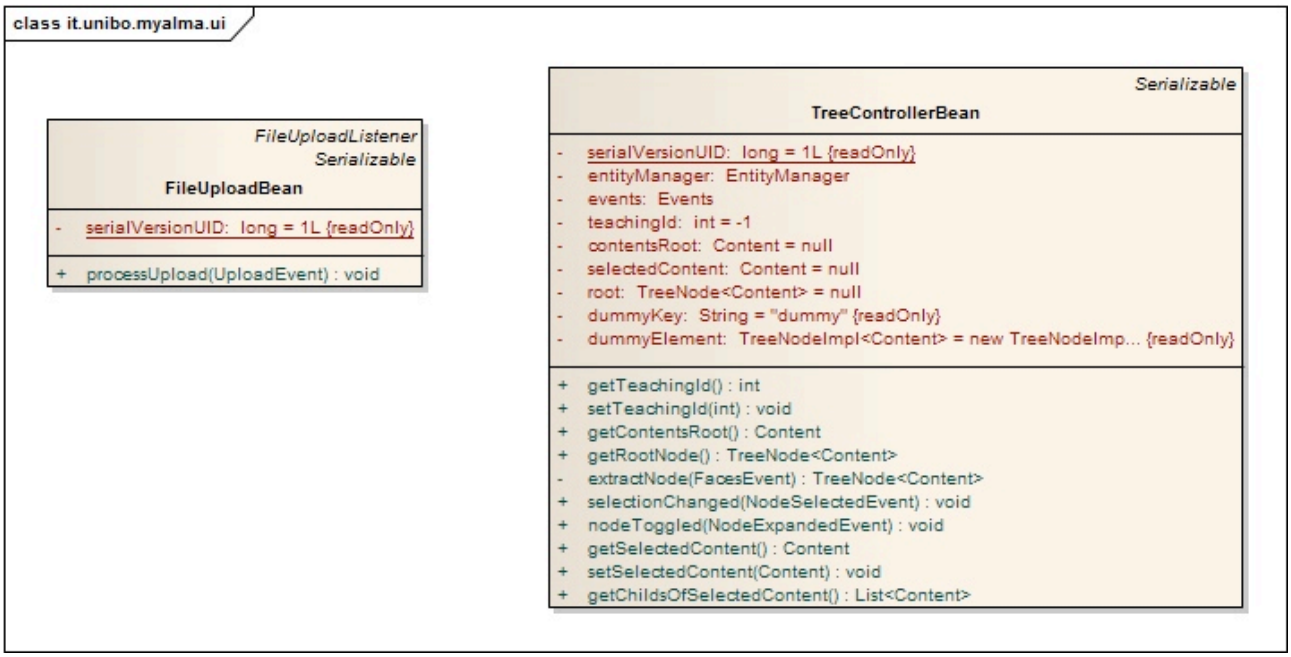


Figura 2.18 - Componenti Seam Interfaccia Grafica

2.5.1 Sicurezza

Per rendere sicuro l'accesso alle pagine JSF sono state necessarie due configurazioni. La prima è specifica di JBoss ed è necessaria per indicare al container web quale dominio di sicurezza deve essere utilizzato per autenticare e autorizzare gli utenti e si concretizza nell'aggiunta della riga mostrata in Figura 2.19 all'interno del file *WEB-INF/jboss-web.xml*.

```
<security-domain>java:/jaas/myalma-security-domain</security-domain>
```

Figura 2.19 - Dichiarazione Dominio di Sicurezza Container Web

La seconda configurazione, presente invece nel file *WEB-INF/web.xml*, consiste nel definire tre diversi aspetti della sicurezza web. Prima di tutto tramite i tag *<security-role>* è necessario definire quali sono i ruoli che saranno utilizzati dall'applicazione. Il secondo aspetto riguarda la modalità di inserimento delle credenziali di acceso, nel caso di myAlma è stato scelto di utilizzare un classico form HTML che richiede nome utente e password,

quindi è stata indicata la pagina che contiene tale form, un'alternativa è quella di utilizzare la modalità BASIC che mostra una finestra di dialogo del sistema operativo per richiedere i dati. Infine l'ultima parte da configurare è quella relativa alla definizione di quali ruoli possono accedere a quali pagine: in figura 2.19 è mostrato come a tutte le pagine che si trovano nella cartella */restricted/professor/* è ammesso l'accesso (sia in GET che in POST) ai soli ruoli di amministratore e professore (è presente una configurazione analoga per le pagine degli studenti).

```
<security-role>
 <role-name>admin</role-name>
</security-role>
<security-role>
 <role-name>professor</role-name>
</security-role>
<security-role>
 <role-name>student</role-name>
</security-role>

<login-config>
 <auth-method>FORM</auth-method>
 <form-login-config>
  <form-login-page>/PreLogin.html</form-login-page>
  <form-error-page>/errors/error.xhtml</form-error-page>
 </form-login-config>
</login-config>

<security-constraint>
 <display-name>Professor</display-name>
 <web-resource-collection>
  <web-resource-name>XHTML</web-resource-name>
  <url-pattern>/restricted/professor/*</url-pattern>
  <http-method>GET</http-method>
  <http-method>POST</http-method>
 </web-resource-collection>
 <auth-constraint>
  <role-name>admin</role-name>
  <role-name>professor</role-name>
 </auth-constraint>
</security-constraint>
```

Figura 2.19 - Configurazione Sicurezza web.xml

2.6 Collaudo

La verifica del corretto funzionamento dei componenti implementati è stata realizzata mediante la stesura di diversi test case utilizzando il framework JUnit 4. I componenti EJB sono stati testati come componenti remoti, dato che soluzioni di testing all'interno del container (ad esempio Arquillian) presentano ancora alcuni problemi per quanto riguarda la gestione della sicurezza. Tale scelta ha complicato in alcuni casi la stesura dei test perché, come descritto in precedenza, tutte le collezioni sono gestite in modalità LAZY LOADING, quindi ad esempio dopo un'inserimento di un nuovo contenuto nell'albero non è possibile verificare la presenza del nuovo contenuto partendo dal suo padre nell'albero perché la collezione dei figli non è inizializzata; il problema è aggirato facendo delle interrogazioni al DB tramite il SearchBean.

Il testing dei componenti POJO Seam è invece stato effettuato manualmente tramite l'interfaccia grafica, dato che tali componenti agiscono proprio su tale sezione dell'applicazione.

3. Clustering in pratica: JBoss AS 7

In questa sezione verrà analizzato il supporto al clustering offerto dall'application server JBoss 7 facendo riferimento ai concetti descritti nella sezione 1.

3.1 Introduzione a JBoss

JBoss è un application server open-source scritto in Java e conforme alla specifica JEE sviluppato da JBoss, una divisione di Red Hat. L'obbiettivo della release numero 7 è quello di offrire un ambiente potente e ricco di funzionalità mantenendo al contempo la configurazione leggera e flessibile.

Il nuovo kernel JBoss è basato principalmente su due progetti:

- **JBoss Modules**: si occupa del caricamento delle classi necessarie per le risorse all'interno del container.
- **Modular Service Container (MSC)**: fornisce la possibilità di installazione, disinstallazione e gestione dei servizi utilizzati dal container. MSC inoltre abilita l'injection di risorse all'interno dei servizi e la gestione delle dipendenze tra i servizi stessi.

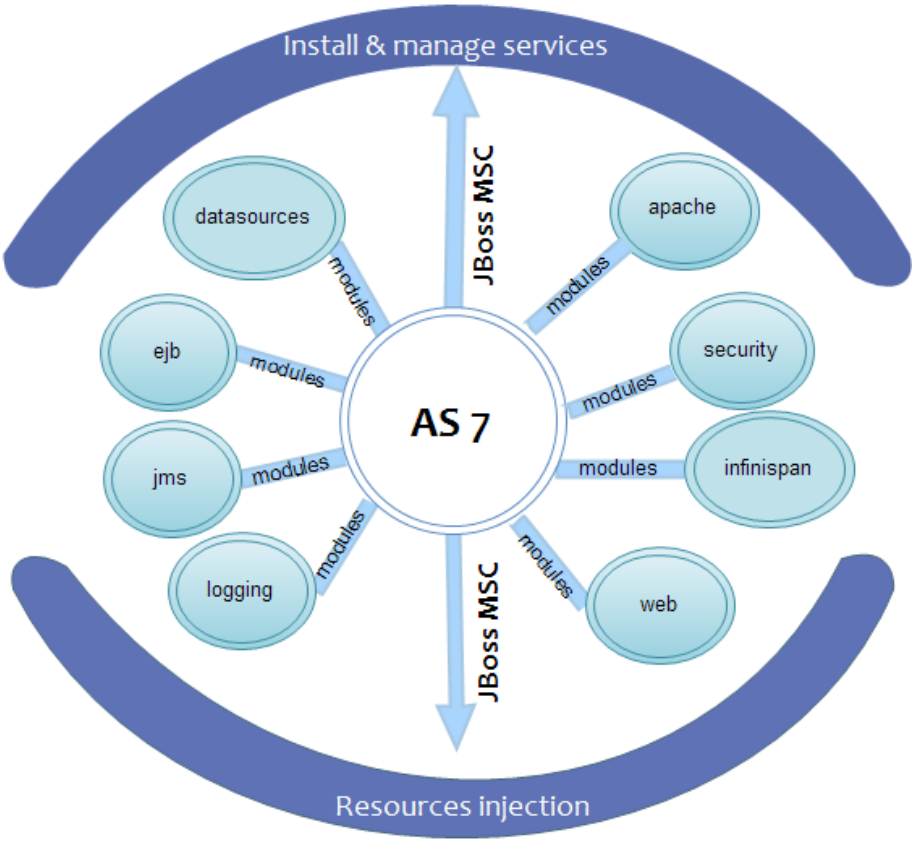


Figura 3.1 - Architettura Kernel JBoss 7

La volontà di mantenere semplice la configurazione del server si manifesta nel fatto che a differenza delle precedenti versioni, in cui era presente un file di configurazione per ogni sotto-sistema, in questa versione esiste praticamente un unico file (*standalone.xml* o *domain.xml*) di configurazione che racchiude le impostazioni per i diversi sotto-sistemi (alcuni file di configurazione esterni possono ancora essere impiegati soprattutto per le impostazioni specifiche per ogni applicazione).

Un'altra grande differenza rispetto alle precedenti versioni riguarda come l'application server carica in memoria i servizi e le librerie richieste. Il nuovo server prevede un approccio modulare: le librerie stesse sono dei moduli che dichiarano solo ed esclusivamente i moduli dalle quali dipendono e possono utilizzare solo le classi presenti nei moduli dichiarati (isolamento dei moduli). All'avvio del server il kernel si occupa di caricare solo i moduli che sono richiesti come dipendenza da quelli di base, lo stesso avviene quando si fa il deploy della propria applicazione, infatti un'applicazione è vista a sua volta come un modulo. Questa soluzione garantisce che i moduli non siano caricati fino a che non sono effettivamente utilizzati, avendo così un forte impatto positivo sulle performance del server (soprattutto in termini di start-up time e consumo di memoria). Alcune librerie sono qualificate come dipendenze implicite, quindi sono aggiunte automaticamente all'applicazione di cui si sta facendo il deploy quando il sistema di deployment ne individua l'utilizzo (ad esempio, javax.ejb, javax.persistence, org.hibernate, ...) tutte le altre dipendenze devono essere dichiarate esplicitamente all'interno del descrittore di deployment *jboss-deployment-structure.xml* da posizionare nella cartella META-INF o WEB-INF oppure nel file *MANIFEST.MF*.

La figura sottostante mostra l'architettura di base per il clustering in JBoss AS 7, come è possibile notare non esiste un'unica libreria che si occupa del clustering ma piuttosto un insieme di librerie che coprono diversi aspetti. I due blocchi principali, su cui si appoggiano tutti i servizi applicativi (EJB, HTTP Session, JPA, Messaging), sono JGroups e Infinispan. Il primo fornisce un mezzo di comunicazione affidabile tra i membri del cluster, il secondo invece si occupa di gestire la replicazione dello stato dell'applicazione su tutto il cluster per garantire così fault tollerance. Tali componenti saranno analizzati nel dettaglio nelle successive sezioni.

Per poter realizzare un cluster di application server JBoss 7 è necessario utilizzare i file di configurazione *standalone-ha.xml* o *standalone-full-ha.xml* che contengono le configurazioni di tutti i sottosistemi necessari al cluster.

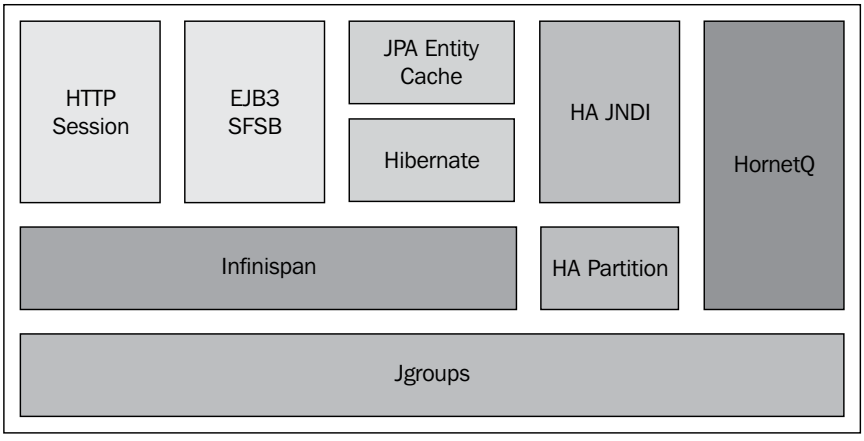


Figura 3.2 - Architettura Clustering JBoss AS 7

3.1.1 JGroups

JGroups è un framework Java per le comunicazioni multicast affidabili. Utilizza le infrastrutture di rete e i protocolli esistenti per trasmettere messaggi multicast affidabili nel senso che i destinatari dei messaggi possono richiedere il reinvio dei messaggi persi. JGroups supporta diversi protocolli di trasporto (UDP e TCP), garantisce la failure detection e il discovery automatico dei membri del cluster.

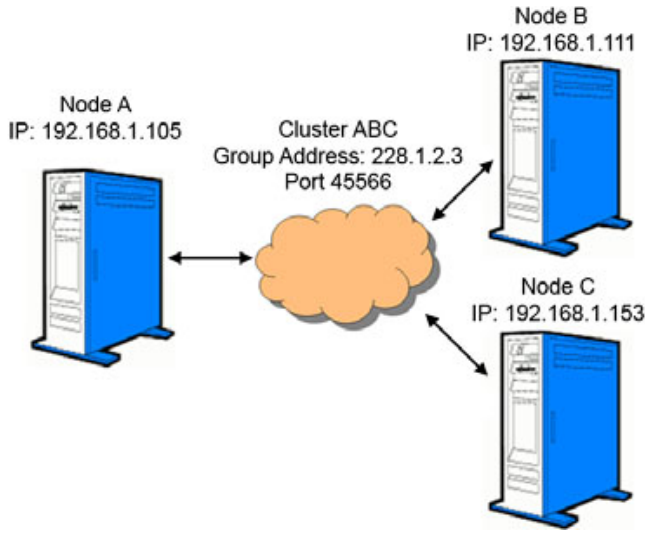


Figura 3.3 - Comunicazione Multicast

L'infrastruttura offerta da JGroups si divide principalmente in tre parti (Figura 3.4): il Canale, i Building Blocks e lo Stack di protocollo. Il Canale rappresenta un'interfaccia di comunicazione (molto simile alle socket) utilizzata dalle applicazioni per scambiare messaggi affidabili all'interno di un gruppo. I Building Blocks sono interfacce per l'utilizzo dei canali ad un più alto livello di astrazione rispetto ai canali stessi, quindi le applicazioni possono utilizzare i Building Blocks invece del Canale direttamente quando sono richiesti servizi aggiuntivi. Lo Stack di protcollo è costituito da una serie di protocolli, impilati in un certo ordine, che devono essere attraversati quando il messaggio è inviato e ricevuto (in un senso o nell'altro). Tali protocolli non corrispondono necessariamente ad un protocollo di trasporto, infatti esisto protocolli con finalità diverse, come ad esempio la frammentazione dei messaggi, la failure detection e altri.

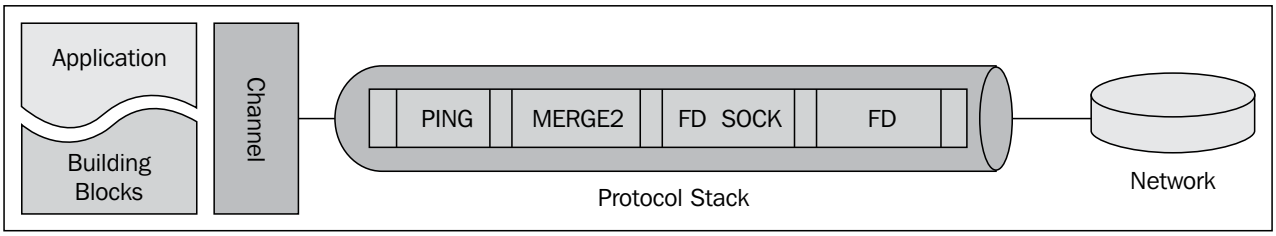


Figura 3.4 - Arichitettura JGroups

JGroups è direttamente integrato all'interno dell'application server JBoss 7 (versione 3.0.6.Final) e come configurazione predefinita prevede l'utilizzo del protocollo di trasporto UDP (Figura 3.5). Con questa configurazione, se l'invio in multicast è abilitato, è possibile

inviare un singolo messaggio all'indirizzo e alla porta di multicast confidando che il messaggio sarà ricevuto da tutti i nodi che sono in ascolto su quell'indirizzo e su quella determinata porta, se il multicast è disabilitato JGroups invia singoli messaggi unicast. Nonostante il protocollo UDP è per sua natura non affidabile, la comunicazione è resa affidabile anche tramite tale protocollo da JGorups stesso. Oltre allo stack UDP predefinito, all'interno del file *standalone-ha.xml* è presente anche uno stack che utilizza il protocollo TCP. Stack basati su TCP sono di solito utilizzati per due motivi: quando il multicast è disabilitato oppure quando si vogliono realizzare gruppi di nodi distribuiti geograficamente, collegati quindi a reti di tipo WAN (possibilità consentita ma probabilmente non ottimale dal punto di vista delle performance). Dato che quando si utilizza il protocollo TCP JGroups utilizza messaggi multipli unicast le performance ne risentono sensibilmente, quindi conviene (come regola generale) utilizzare UDP quando possibile e specialmente nei casi in cui le performance hanno un peso maggiore sull'affidabilità.

```
<subsystem xmlns="urn:jboss:domain:jgroups:1.1" default-stack="udp">
    <stack name="udp">
        <transport type="UDP" socket-binding="jgroups-udp"
                              diagnostics-socket-binding="jgroups-diagnostics"/>
        <protocol type="PING"/>
        <protocol type="MERGE2"/>
        <protocol type="FD_SOCK" socket-binding="jgroups-udp-fd"/>
        <protocol type="FD"/>
        <protocol type="VERIFY_SUSPECT"/>
        <protocol type="BARRIER"/>
        <protocol type="pbcast.NAKACK"/>
        <protocol type="UNICAST2"/>
        <protocol type="pbcast.STABLE"/>
        <protocol type="pbcast.GMS"/>
        <protocol type="UFC"/>
        <protocol type="MFC"/>
        <protocol type="FRAG2"/>
    </stack>
</subsystem>
```

Figura 3.5 - Stack di protocollo predefinito in JBoss AS 7

Differenti servizi all'interno del server possono utilizzare differenti canali JGroups, questo comporta però l'utilizzo di più thread e quindi un maggiore utilizzo della risorsa CPU, per questo motivo di solito si utilizzano pochi canali per diversi servizi: JGroups si occupa a quel punto di smistare il traffico sui canali verso i corretti servizi destinatari (funzionalità conosciuta come *Multiplexing del canale*).

Come descritto in precedenza i protocolli che possono essere utilizzati all'interno dello Stack di protocollo non sono relativi soltanto al trasporto dei messaggi ma ne esistono diversi che si occupano di aspetti anche molto diversi della comunicazione. Questo rende JGroups molto flessibile e permette agli sviluppatori di definire lo stack più appropriato per i bisogni dell'applicazione: è possibile configurare stack che spaziano dal migliore livello di performance a discapito però dell'affidabilità fino a stack che soffrono in termini di velocità ma che garantiscono alto grado di affidabilità. Nell'esempio di Figura 3.5 ad esempio vengono impostati i protocolli *PING* e *MERGE2* che si occupano del discovery automatico dei membri del gruppo, i protocolli *FD_SOCK* e *FD* si occupano di failure detection, il protocollo *pbcast.NAKACK* che si occupa dell'affidabilità nella trasmissione e il protocollo *FRAG2* che si occupa della frammentazione e deframmentazione dei messaggi troppo grandi.

Per quanto riguarda myAlma, è stato deciso di utilizzare la configurazione predefinita basata su UDP senza ulteriori modifiche in quanto non si è ritenuto necessario spingere la configurazione di questo aspetto del clustering verso una granularità troppo fine, ci si è invece concentrati maggiormente sugli aspetti di caching.

3.1.2 Infinispan

Come descritto nella sezione 1, applicazioni che mantengono lo stato di interazione con l'utente, devono abilitare la replicazione di tale stato per essere tolleranti ai guasti. JBoss permette la replicazione dello stato di interazione appoggiandosi sul framework open-source Infinispan. Infinispan permette il caching locale e soprattutto distribuito di oggetti Java. Infinispan viene infatti utilizzato in JBoss 7 (sostituendo a partire dalla versione 6 del server la libreria JBoss Cache) come infrastruttura di caching e replicazione per la sessione HTTP, lo stato di Statefull beans, nomi inseriti in JNDI e come cache di secondo livello per Hibernate. Infinispan eventualmente può essere utilizzato al di fuori di JBoss come una qualsiasi libreria.

Strategie
Infinispan permette l'utilizzo di quattro strategie di caching:

- **Locale**: in questa modalità gli oggetti sono inseriti solo in una cache in memoria locale e non sono disponibili localmente su altri nodi, anche se è stato formato un cluster. Tale modalità è di solito preferita per la cache di secondo livello di Hibernate come descritto successivamente.
- **Replicazione**: quando gli oggetti inseriti in cache sono replicati su tutti i nodi che formano il cluster, quindi tutti gli oggetti sono disponibili localmente su tutti i nodi. Tale modalità è di solito utilizzata per cluster di piccole dimensioni, in quanto i messaggi di replicazione crescono notevolmente al crescere del dimensione del cluster.

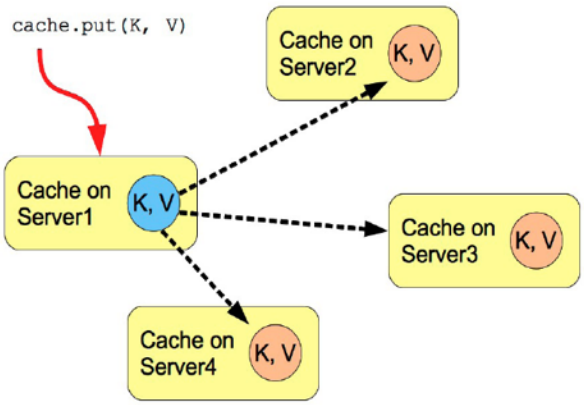


Figura 3.6 - Modalità Replicazione

- **Distribuzione**: con la modalità distribuzione gli oggetti in cache sono replicati solo su un sottoinsieme fisso di nodi (configurabile), tali oggetti quindi non sono disponibili localmente su tutti i nodi del cluster ma le performance della replicazione crescono sensibilmente, specialmente al crescere della dimensione del cluster.

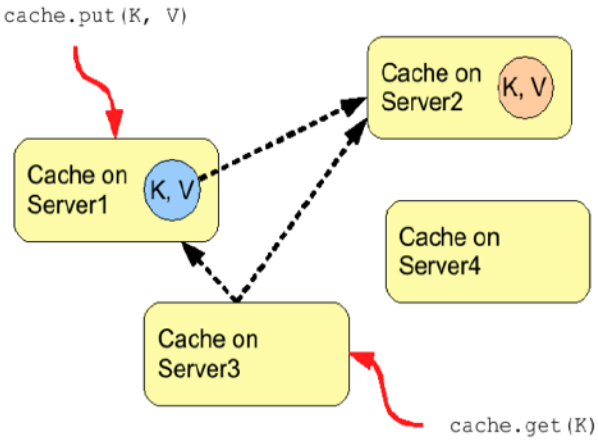


Figura 3.7 - Modalità Distribuzione

- **Invalidazione**: in questa modalità non è prevista alcuna replicazione di oggetti ma l'unico obbiettivo è quello di rimuovere dalle cache remote i dati che non sono più validi. Questa modalità ha senso solo se i dati sono mantenuti in una locazione persistente (un DB ad esempio) e se si vuole utilizzare Infinispan per evitare di leggere continuamente i dati da tale locazione. Durante l'utilizzo di tale modalità ogni volta che un dato è modificato viene inviato un messaggio agli altri nodi del cluster informandoli di rimuovere quel dato dalla cache perché non più aggiornato, una successiva richiesta dello stesso dato si trasformerà in una lettura dalla locazione persistente.

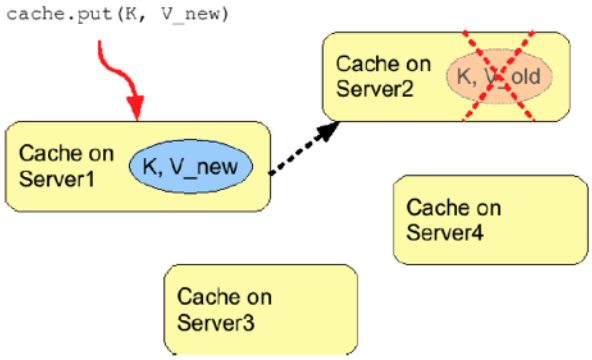


Figura 3.8 - Modalità Invalidazione

Per quanto riguarda le ultime tre modalità, la replicazione o l'invalidazione dei dati può avvenire in modo sincrono o asincrono. La modalità sincrona blocca il chiamante, ossia chi sta modificando la propria cache, fino a che la sincronizzazione tra tutti i nodi non ha avuto successo (ovviamente in caso di distribuzione sono coinvolti solo i nodi configurati per la distribuzione e non tutti quelli appartenenti al cluster). In modalità asincrona invece la replicazione (o la invalidazione) avviene in background e il chiamante può procedere immediatamente, senza attese. La modalità asincrona prevede anche l'abilitazione di una coda di sincronizzazione per accumulare modifiche da apportare alle cache remote ed eseguirle tutte in un'unica operazione, tale opzione offre di solito dei vantaggi in termini di performance in quanto si limita il numero di messaggi che circolano sulla rete. La modalità asincrona garantisce performance migliori di quella sincrona in quanto quest'ultima deve

attendere la risposta di esito positivo da parte di tutti i nodi. Nel caso di operazioni sincrone però, quando l'operazione termina con successo si ha la certezza che tutte le modifiche siano state apportate correttamente, cosa che invece non accadde con le operazioni asincrone. Nella modalità asincrona infatti gli errori sono solo riportati su un file di log e anche se sono utilizzate le transazioni, una transazione può terminare con successo ma la replicazione potrebbe fallire su alcuni nodi. La scelta quindi di quale strategia di caching e di quale modalità di sincronizzazione utilizzare si traduce in un compromesso tra performance e garanzia sulla consistenza dei dati a livello di cluster.

Per migliorare le performance quando si accede ripetutamente a cache remote, Infinispan permette di abilitare una cache locale, chiamata L1, in cui sono contenuti gli oggetti ricevuti da remoto, in questo modo letture ripetute non si traducono in chiamate remote ma solo alla cache L1 locale. Comunque, abilitare la cache L1 ha un costo dovuto al fatto che ogni volta che un'entry nella cache remota viene aggiornata deve essere inviato un messaggio a tutti i partecipanti del cluster per invalidare la loro cache L1. In generale quindi conviene effettuare delle prove con la L1 abilitata e disabilitata per determinare la scelta ottimale.

Eviction
Infinispan supporta la rimozione (eviction) di oggetti dalla cache per limitare l'uso di memoria. Di solito l'eviction è utilizzata insieme ad un cosiddetto cache store in cui vengono memorizzati gli oggetti in modo persistente per evitare che vengano persi definitivamente, infatti l'eviction è solo una rimozione di entry dalla memoria centrale ma non è una rimozione definitiva. L'eviction avviene solo a livello locale e non a livello di cluster, quindi anche in caso di replicazione può capitare che le entry in memoria principale siano diverse da nodo a nodo perché su ciascun nodo le entry che hanno subito eviction possono essere diverse. Il thread che si occupa dell'eviction delle entry non viene avviato quando l'ammontare di memoria libera raggiunge una certa soglia ma è necessario impostare un parametro (*maxEntries*) che indica il numero massimo di elementi che possono rimanere in memoria centrale, quando il limite viene raggiunto l'operazione di eviction viene avviata. Il valore *maxEntries* deve essere scelto in modo appropriato, tenendo conto sia del numero di entry che si desidera inserire in cache che della loro dimensione, per non rischiare di saturare la memoria. Possono essere utilizzate diverse strategie per scegliere le entry da rimuovere dalla memoria: *UNORDERED* quando la scelta avviene casualmente, *FIFO* quando sono rimosse le entry che sono state aggiunte per prime alla cache e *LRU* quando invece sono rimosse le entry utilizzate meno recentemente. In generale la strategia migliore per la maggior parte degli scenari è la *LRU*.

Cache Loaders
Un'altra caratteristica supportata da Infinispan è quella che va sotto il nome di Cache Loaders. Un Cache Loader rappresenta una connessione tra Infinispan e un data store utilizzato per memorizzare in modo persistente le entry presenti in cache, in questo modo quando un oggetto non è trovato in memoria centrale lo si cerca nel data store. Infinispan supporta diverse tipologie di data store come il file system locale, un DB, un cloud store (servizio Amazon S3 ad esempio), un cluster Infinispan remoto e altri. L'opzione fondamentale quando si usa un Cache Loader è quella relativa alla passivazione. Se la passivazione è disabilitata ogni modifica, aggiunta o rimozione di entry dalla cache in memoria si riflette anche sulla cache presente nel data store persistente, attraverso il cache loader (ciò che c'è nel data store è la copia di ciò che c'è in memoria). Nel caso in

cui invece la passivazione è abilitata la scrittura di un'entry nel data store avviene solo come parte del processo di eviction, quindi solo quando un'entry è rimossa dalla memoria centrale essa viene scritta sul data store. Quando l'entry è nuovamente richiesta essa viene letta ed eliminata dal data store poiché viene inserita nuovamente in memoria centrale. Con la passivazione abilitata quindi non c'è intersezione tra le entry mantenute in memoria centrale e quelle presenti nel data store. Disabilitando la passivazione si ha un grado di affidabilità maggiore dato che le entry in cache sono rese persistenti ma le performance ne risentono sensibilmente (anche se le scritture sul data store possono essere asincrone).

Livelli di isolamento e locking

Dato che una cache rappresenta un'entità condivisa e accessibile a diversi thread all'interno di un application server aspetti riguardanti i lock sui dati e le caratteristiche di transazionalità assumono una discreta importanza. Infinispan supporta solo due livelli di isolamento, del tutto equivalenti dal punto di vista semantico a quelli presenti nei DBMS: *REPEATABLE_READ* e *READ_COMMITTED*. Con entrambi, una lettura su una porzione della cache non blocca altre letture sulla stessa porzione mentre una scrittura blocca altre letture. Nel primo livello (default) una lettura blocca una scrittura (non si presentano fenomeni di *non-repeatable reads* ma potrebbero presentarsi delle *phantom reads*) mentre nel secondo una lettura non blocca una scrittura (potrebbero presentarsi fenomeni di *repeatable-reads*). La scelta tra i due livelli è dettata da un compromesso tra performance che si vogliono ottenere e tolleranza alle possibili inconsistenze sui dati letti: il livello *REPEATABLE_READ* garantisce una certa sicurezza dal punto di vista della consistenza ma peggiora il grado parallelismo, il livello *READ_COMMITTED* invece presenta caratteristiche duali. Infinispan, per migliorare le performance, offre inoltre la possibilità di utilizzare il locking ottimistico (OPTMISTIC LOCKING) in cui non sono mantenuti lock dalle transazioni però prima di rendere definitive le modifiche (COMMIT) viene verificato se altre transazioni hanno modificato lo stesso dato, in tal caso la la transazione non viene completata con successo (ROLLBACK).

Infinispan e JBoss

Come accennato in precedenza Infinispan può essere utilizzato come libreria standalone o come integrato all'interno dell'application server JBoss, in particolare nella versione 7.1.1.Final del server è integrata la versione 5.1.2.Final di Infinispan. Come tutti gli altri servizi attivi sul server le configurazioni di Infinispan sono contenute nel file *standalone.xml* o *standalone-ha-full.xml* nel caso in cui si voglia configurare un cluster. Tutte le impostazioni sono contenute all'interno del sotto-sistema *urn:jboss:domain:infinispan:1.2* e come configurazione predefinita sono previste quattro cache per i servizi principali utilizzati in un cluster: ha-partition, sessione HTTP, stato Statefull beans, cache di secondo livello per Hibernate. Come è possibile notare dalla Figura 3.9, per ogni cache (*<cache-container>*) sono presenti diverse configurazioni (*<replicated-cache>*, *<distributed-cache>*, *<local-cahe>* o *<invalidation-cache>*) che possono essere selezionate tramite l'attributo *default-cache* oppure tramite opportuni descrittori di deployment o annotazioni.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-container="cluster">
    <cache-container name="cluster" aliases="ha-partition" default-cache="default">
        <transport lock-timeout="60000"/>
        <replicated-cache name="default" mode="SYNC" batching="true">
            <locking isolation="REPEATABLE_READ"/>
        </replicated-cache>
    </cache-container>
    <cache-container name="web" aliases="standard-session-cache" default-cache="repl">
        <transport lock-timeout="60000"/>
        <replicated-cache name="repl" mode="ASYNC" batching="true">
            <file-store/>
        </replicated-cache>
        <replicated-cache name="sso" mode="SYNC" batching="true"/>
        <distributed-cache name="dist" mode="ASYNC" batching="true">
            <file-store/>
        </distributed-cache>
    </cache-container>
    <cache-container name="ejb" aliases="sfsb sfsb-cache" default-cache="repl">
        <transport lock-timeout="60000"/>
        <replicated-cache name="repl" mode="ASYNC" batching="true">
            <eviction strategy="LRU"/>
            <file-store/>
        </replicated-cache>
        <replicated-cache name="remote-connector-client-mappings" mode="SYNC"
                          batching="true"/>
        <distributed-cache name="dist" mode="ASYNC" batching="true">
            <eviction strategy="LRU"/>
            <file-store/>
        </distributed-cache>
    </cache-container>
    <cache-container name="hibernate" default-cache="local-query">
        <transport lock-timeout="60000"/>
        <local-cache name="local-query">
            <locking isolation="READ_COMMITTED"/>
            <transaction mode="NONE" locking="OPTIMISTIC"/>
            <eviction strategy="LRU" max-entries="10000"/>
            <expiration max-idle="100000"/>
        </local-cache>
        <invalidation-cache name="entity" mode="SYNC">
            <locking isolation="READ_COMMITTED"/>
            <transaction mode="NON_XA" locking="OPTIMISTIC"/>
            <eviction strategy="LRU" max-entries="10000"/>
            <expiration max-idle="100000"/>
        </invalidation-cache>
        <replicated-cache name="timestamps" mode="ASYNC">
            <locking isolation="READ_COMMITTED"/>
            <transaction mode="NONE" locking="OPTIMISTIC"/>
            <eviction strategy="NONE"/>
        </replicated-cache>
    </cache-container>
</subsystem>
```

Figura 3.9 - Configurazione Predefinita in JBoss AS

Infinispan e JGroups

Infinispan si appoggia su JGroups per tutte le comunicazioni all'interno del cluster quindi è possibile, come descritto in precedenza, abilitare il discovery automatico dei membri del cluster e meccanismi di fault detection. Come configurazione predefinita JBoss 7 istanzia un canale JGroups per ogni tipo di cache che deve essere gestita e utilizza lo stack predefinito basato su UDP (Figura 3.5). Tale impostazione può essere modificata tramite l'attributo *stack* del tag *<transport>* da inserire all'interno dell'elemento *<cache-container>*.

3.2 Clustering dei servizi JBoss

Nel seguito saranno analizzate le modalità di clustering per i vari servizi JBoss e le configurazioni di caching più appropriate da utilizzare nelle varie occasioni.

3.2.1 HTTP Load Balancing: mod_cluster

La nuova versione dell'application server utilizza il progetto *mod_cluster (1.2.0.Final)* come load balancer predefinito. mod_cluster è un load balancer software basato su HTTP e presenta tre caratteristiche principali che lo contraddistinguono da prodotti più datati come mod_jk e mod_proxy:

- **Configurazione dinamica del cluster**: è possibile aggiungere o rimuovere nodi al cluster dinamicamente in quanto viene utilizzato un meccanismo di advertising: in pratica le librerie mod_cluster sul load balancer inviano messaggi UDP ad un un gruppo di multicast, presso il quale si registrano gli application server, in questo modo i server fanno il discovery automatico del load balancer (o eventualmente dei load balancer se ce ne sono più di uno).

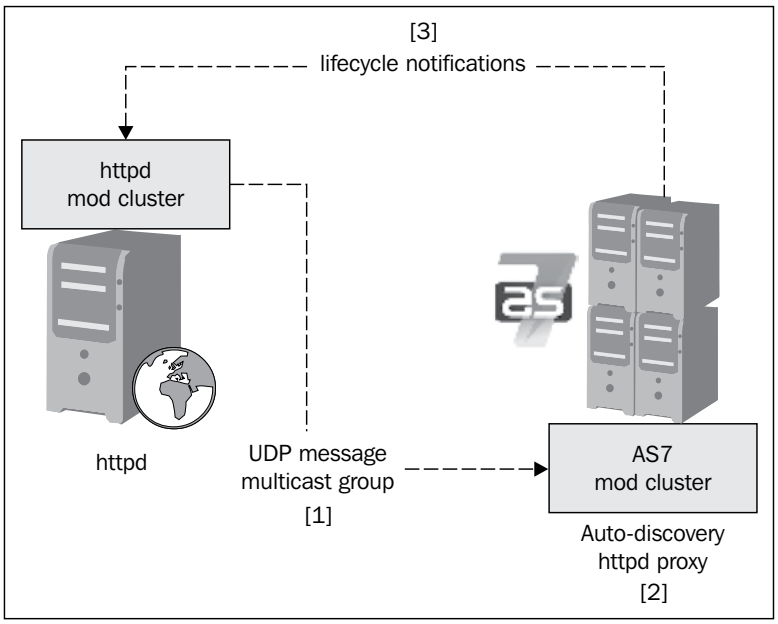


Figura 3.10 - Auto-Discovery in mod_cluster

- **Supporto a diverse metriche per la misurazione del carico lato server**: il carico dei server viene misurato direttamente sul server in cui è possibile scegliere tra diversi fattori (cpu, memoria, connessioni, ...). Le misure sono inviate al load balancer il quale le utilizza per distribuire il carico sui vari server. Questo approccio è decisamente nuovo rispetto alle vecchie soluzioni in cui le metriche per il carico erano staticamente configurate sul load balancer.
- **Notifiche dello stato delle applicazioni presenti sull'application server**: mod_cluster è in grado di intercettare eventi riguardanti le applicazioni presenti sugli application server (re-deploy, un-deploy, ...) e dirigere le richieste in ingresso verso i server che sono in grado di soddisfarle.

Installazione
L'installazione di mod_cluster lato application server JBoss 7 non richiede alcuna operazione dato che il load balancer è già presente sotto forma di modulo. Per quanto riguarda invece l'installazione lato web server che farà da front-end per i clienti del cluster di solito si installano i moduli mod_cluster all'interno un server web Apache. In alternativa è possibile scaricare dal sito web del progetto mod_cluster un pacchetto che contiene il server web Apache con le librerie mod_cluster pre-installate, quindi è subito pronto per l'uso. Per semplicità è stata scelta la seconda via che permette di avere un load balancer funzionante in un tempo piuttosto breve.

Configurazione lato load balancer
La configurazione principale lato load balancer è quella mostrata in Figura 3.11. Con la direttiva *Listen* è possibile configurare indirizzo IP e porta sul quale il load balancer sarà in attesa di richieste da parte dei clienti. L'indirizzo IP e la porta presenti invece all'interno della definizione del *VirtualHost* rappresentano il punto di comunicazione tra il load balancer e gli application server. Di solito l'indirizzo IP è lo stesso per entrambe le configurazioni e corrisponde con l'indirizzo associato al load balancer, le porte invece possono essere diverse.

```
Listen 192.168.10.1:8888
<VirtualHost 192.168.10.1:8888>

   <Location />
      Order deny,allow
      Deny from all
      Allow from 192.168.10.
   </Location>

 KeepAliveTimeout 60
 MaxKeepAliveRequests 0

 ManagerBalancerName mycluster
 ServerAdvertise On

</VirtualHost>
```

Figura 3.11 - Configurazione Principale mod_cluster

Oltre alle direttive per il riuso delle connessioni entro un certo tempo (*KeepAliveTimeout*) e quelle per il controllo degli accessi (*<Location>*) la direttiva fondamentale è *ServerAdvertise On* che abilita il meccanismo di advertising per il discovery automatico dei load balancer da parte degli application server. Come impostazione predefinita l'indirizzo IP e la porta di multicast utilizzati per l'advertising sono 224.0.1.105:23364, tale configurazione può essere sovrascritta tramite la direttiva *AdvertiseGroup xxx.xxx.xxx.xxx:xxxx*. La modifica di tali parametri nella configurazione del load balancer si deve riflettere nella modifica degli stessi parametri nella configurazione lato application server, altrimenti i server non saranno in grado di individuare i load balancer.

Configurazione lato application server

La configurazione di mod_cluster lato application server è presente all'interno del file *standalone-ha.xml* o *domain.xml* e in particolare nel sotto-sistema *urn:jboss:domain:modcluster:1.0* (Figura 3.12).

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
    <mod-cluster-config advertise-socket="modcluster">
        <dynamic-load-provider>
            <load-metric type="cpu" weight="2" capacity="1"/>
            <load-metric type="sessions" weight="1" capacity="512"/>
        </dynamic-load-provider>
    </mod-cluster-config>
</subsystem>
```

Figura 3.12 - Configurazione mod_cluster lato application server

Come è possibile notare la configurazione è particolarmente semplice e sintetica poiché si basa principalmente su valori di default che si rivelano adeguati nella maggior parte delle situazioni. Ad esempio, in modo predefinito sono abilitate sia la sticky session che l'advertise, in ogni caso se si vogliono modificare tali valori possono essere utilizzati opportuni attributi del tag *<mod-cluster-config>*. Per quanto riguarda l'advertising, l'indirizzo e la porta del gruppo sono configurati in un'altra posizione nel file, in particolare nella sezione *<socket-binding-group>* e poi collegati con la configurazione di mod_cluster tramite l'attributo *advertise-socket*; tale schema è tipico della configurazione di JBoss AS 7. Nella figura successiva è possibile notare come sono specificati l'indirizzo e la porta per l'advertise e come essi coincidano con i valori di default utilizzati da mod_cluster lato load balancer.

```
<socket-binding name="modcluster"
                port="0"
                multicast-address="224.0.1.105"
                multicast-port="23364"/>
```

Figura 3.13 - Configurazione socket mod_cluster lato application server

Un'aspetto della configurazione che non prevede valori predefiniti è quello riguardante le metriche per la determinazione del carico dei server. Tali metriche possono essere indicate tramite i tag *<load-metric>*. E' possibile specificare più metriche e attribuire a ciascuna un peso che determina l'impatto di quella metrica sulle altre. Più una metrica ha peso maggiore più essa sarà presa in considerazione nella scelta del nodo verso cui inviare le richieste. Un'altro parametro che può essere utilizzato nella configurazione delle metriche è la capacità, specificando la quale è possibile favorire un nodo rispetto agli altri. Un nodo con capacità maggiore per una metrica sarà favorito rispetto ai nodi con capacità minore per la stessa metrica.

Le principali metriche messe a disposizione da mod_cluster prevedono la misurazione del carico sulla CPU (*cpu*), la quantità di memoria utilizzata (*mem*), la percentuale di memoria heap utilizzata sul totale (*heap*), il numero di sessioni (*sessions*) attive e il numero di richieste al secondo gestite (*requests*). E' comunque possibile realizzare la propria metrica personalizzata implementando l'interfaccia *org.jboss.load.metric.LoadMetric*.

Collaudo

Una volta avviato il web server con mod_cluster installato è possibile avviare il cluster di application server e notare che i nodi si riconoscono in automatico e formano il cluster (grazie a JGroups) e inoltre riconoscono il load balancer. Nella Figura 3.14 è riportata la pagina web visualizzata da mod_cluster se si naviga verso l'indirizzo *http:///<IP load balancer>:<port>/mod_cluster_manager*, da questa pagina è possibile controllare che tutti i nodi del cluster abbiano riconosciuto il load balancer e che il load balancer abbia acquisito correttamente le informazioni sulle applicazioni presenti sui server (sezione *Contexts* nella pagina).

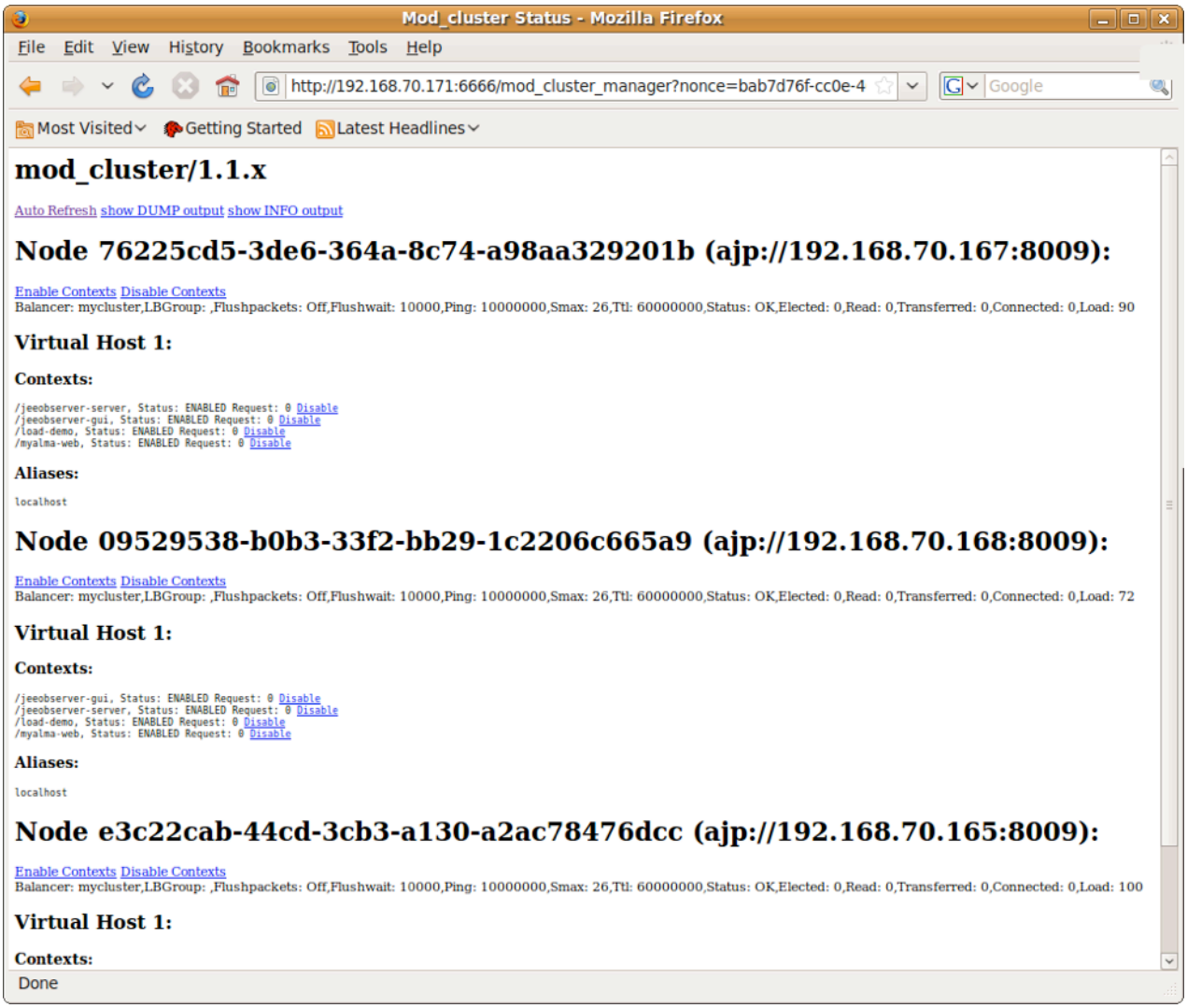


Figura 3.14 - Manager Web mod_cluster

Dal sito web del progetto mod_cluster è possibile scaricare una piccola applicazione che permette il testing della propria configurazione. L'applicazione consiste di una parte lato server da installare sui nodi del cluster e di una parte lato client che si occupa di inviare le richieste verso il load balancer e registrare da quali server le richieste sono soddisfatte. Nella Figura 3.15 si nota come, in condizioni di carico uniforme tra i nodi, il numero di sessioni gestite dai nodi è circa lo stesso per tutti. Quando si aumenta il carico su un nodo (è possibile farlo tramite l'applicazione di testing) si nota (Figura 3.16) come il load balancer favorisca gli altri due nodi che sono più scarichi, indirizzando verso tali nodi più richieste rispetto a quelle inviate al nodo più carico (linea verde in questo caso).

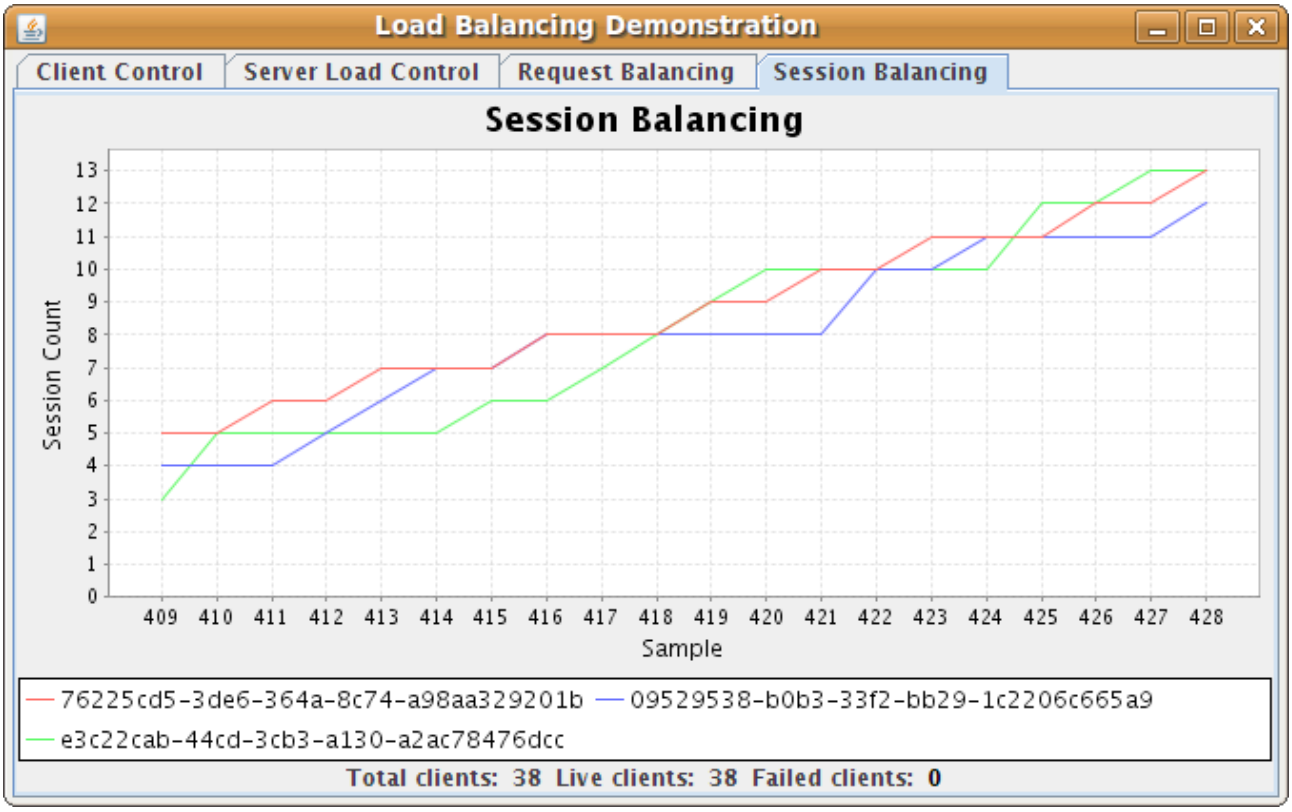


Figura 3.15 - Applicazione testing mod_cluster: carico uniforme tra i nodi

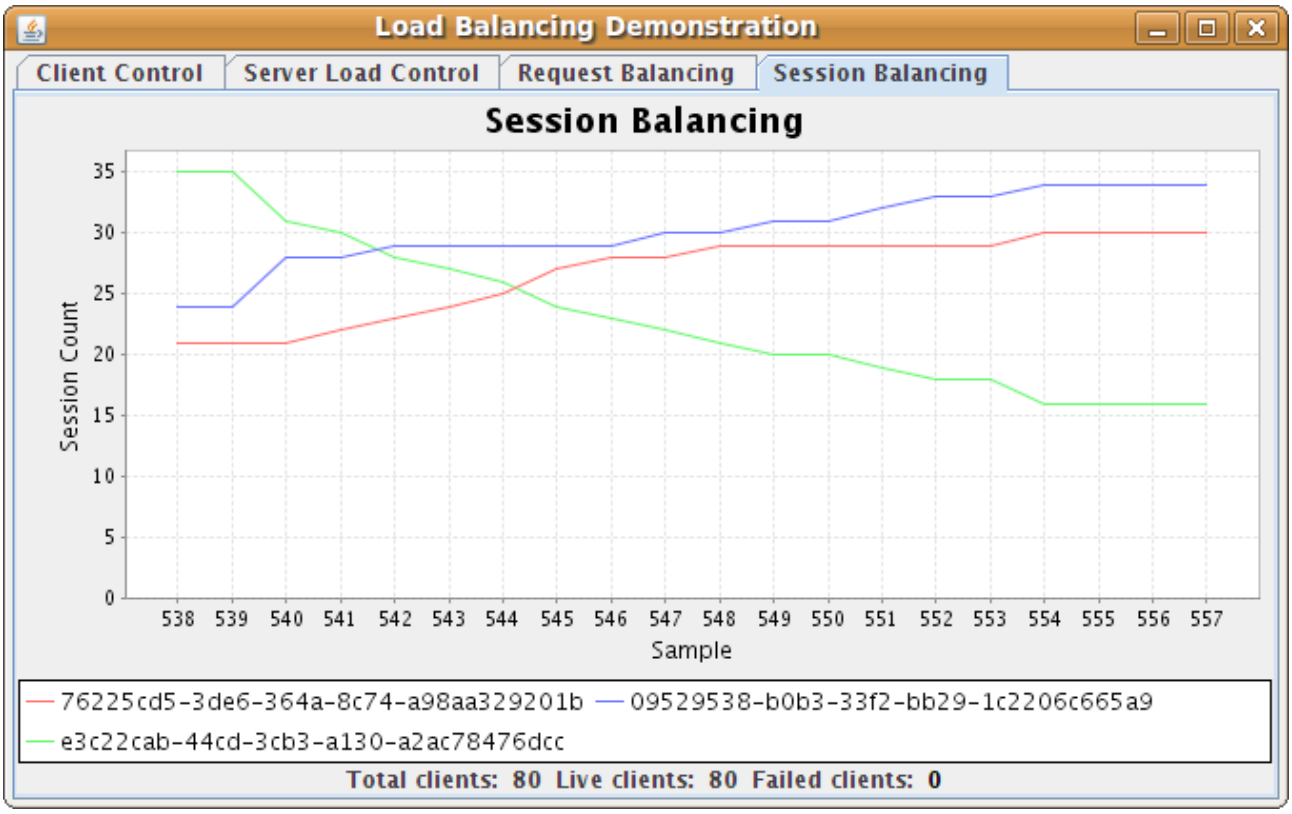


Figura 3.16 - Applicazione testing mod_cluster: un nodo più carico degli altri

3.2.2 Sessione HTTP

Nella sessione HTTP sono di solito mantenuti i dati relativi la parte web dell'applicazione. Dato che per myAlma è stato utilizzato Seam, tutti i componenti gestiti dal framework sono mantenuti nella sessione HTTP, quindi volendo rendere myAlma completamente fault tollerant (a tutti i livelli) è necessario assicurarsi che anche la sessione HTTP sia replicata.

Per poter utilizzare la replicazione di qualsiasi tipo di stato in JBoss 7 è necessario avviare il server specificando come file di configurazione *standalone-ha.xml* o *standalone-full-ha.xml*, in questo modo vengono avvitati tutti i servizi necessari per la replicazione, in particolare per la sessione HTTP un canale JGroups e una cache Infinispan. In ogni caso la scelta se replicare la sessione HTTP è una scelta specifica per ogni applicazione, infatti per abilitare tale replicazione è necessario aggiungere l'elemento *<distributable/>* al file di configurazione *WEB-INF/web.xml*. Inoltre anche Seam deve essere informato che la sessione HTTP deve essere replicata impostando a *true* l'attributo *distributable* dell'elemento *<core:init>* presente nel file *WEB-INF/components.xml* e specificando gli intercettori che devono essere utilizzati (Figura 3.17).

```
<core:init distributable="true" debug="false" jndi-pattern="java:app/myalma-ejb/#{ejbName}">
  <core:interceptors>
          <value>org.jboss.seam.core.SynchronizationInterceptor</value>
          <value>org.jboss.seam.async.AsynchronousInterceptor</value>
          <value>org.jboss.seam.ejb.RemoveInterceptor</value>
          <value>org.jboss.seam.persistence.HibernateSessionProxyInterceptor</value>
          <value>org.jboss.seam.persistence.EntityManagerProxyInterceptor</value>
          <value>org.jboss.seam.core.MethodContextInterceptor</value>
          <value>org.jboss.seam.core.EventInterceptor</value>
          <value>org.jboss.seam.core.ConversationalInterceptor</value>
          <value>org.jboss.seam.bpm.BusinessProcessInterceptor</value>
          <value>org.jboss.seam.core.ConversationInterceptor</value>
          <value>org.jboss.seam.core.BijectionInterceptor</value>
          <value>org.jboss.seam.transaction.RollbackInterceptor</value>
          <value>org.jboss.seam.transaction.TransactionInterceptor</value>
          <value>org.jboss.seam.webservice.WSSecurityInterceptor</value>
          <value>org.jboss.seam.security.SecurityInterceptor</value>
          <value>org.jboss.seam.persistence.ManagedEntityInterceptor</value>
      </core:interceptors>
</core:init>
```

Figura 3.17 - Impostazione Clustering Seam

Se tali configurazioni non sono specificate la sessione HTTP della propria applicazione non verrà replicata anche se la relativa cache Infinispan è attiva e altre applicazioni nel server la stanno utilizzando.

Oltre alle impostazioni tipiche della cache Infinispan che saranno analizzate più nel dettaglio nella sezione 3.3.1, è possibile configurare anche altri aspetti riguardanti la replicazione della sessione HTTP. Il primo punto interessante è rappresentato da quali operazioni comportano la replicazione della sessione. L'interazione con la sessione HTTP avviene tramite le Servlet API che consentono di inserire/leggere degli attributi (oggetti) in/da una mappa presente in sessione, come nell'esempio sottostante:

```
HttpSession session = request.getSession();
Employee emp = (Employee) session.getAttribute("employee");
String firstName = emp.getFirstName();
```

In questo caso dato che il valore dell'attributo è soltanto letto non è necessario effettuare alcuna replicazione. Ma lo scenario potrebbe essere diverso:

```
HttpSession session = request.getSession();
Employee emp = (Employee) session.getAttribute("employee");
emp.setFirstName("Javid");
```

In questo caso infatti l'interazione con la sessione è la stessa dell'esempio precedente però adesso l'attributo letto è stato anche modificato e quindi è necessario che tali modifiche siano replicate su tutto il cluster. Per gestire tali situazioni sono disponibili quattro modalità (triggering):

- **SET**: con questa configurazione la replicazione avviene solo quando un attributo è inserito all'interno della sessione (*setAttribute()*), se si legge un attributo e si modifica uno dei suoi valori, il nuovo valore non è replicato. Tale modalità impone allo sviluppatore di adottare una forte disciplina di programmazione secondo la quale dopo ogni modifica di un attributo ci si assicuri di re-inserirlo nella sessione. Nonostante ciò con tale impostazione si hanno le performance migliori in quanto si replica la sessione solo quando ce ne è effettivamente bisogno.
- **SET_AND_GET**: la replicazione viene effettuata sia per le operazioni di inserimento che di lettura per qualsiasi tipo di attributo. E' evidente come tale modalità sia molto più pesante dal punto di vista del numero di replicazioni che devono essere eseguite.
- **SET_AND_NON_PRIMITIVE_GET** (default): il comportamento è lo stesso della modalità SET_AND_GET con l'unica differenza che la replicazione in caso di lettura viene effettuata solo se il dato letto non è un wrapper per i tipi primitivi Java (Long, Integer, ...). Tale modalità presenta un miglioramento rispetto a quella precedente e si basa sul fatto che i wrapper per i tipi primitivi sono immutabili, quindi non è possibile modificare una loro istanza direttamente ma è necessario re-inserire l'oggetto all'interno della sessione per assicurarsi che venga replicato.
- **ACCESS**: con questa modalità anche il semplice accesso alla sessione fa si che sia marcata come "dirty" e che quindi sia replicata. Questa è la modalità con le performance peggiori però garantisce replicazione certa.

Di solito la modalità SET_AND_NON_PRIMITIVE_GET offre un buon compromesso tra performance e sicurezza sulla replicazione delle modifiche. Nel caso specifico di myAlma è stata mantenuta l'impostazione di default in quanto le operazioni sulla sessione HTTP sono effettuate da Seam e quindi non si può avere la certezza che sia stata utilizzata la disciplina di cui si parlava per l'opzione SET, che sarebbe stata la scelta più performante.

Oltre a decidere le azioni che scatenano la replicazione della sessione è possibile decidere anche cosa replicare, in tal caso le configurazioni possibili sono tre:

- **SESSION** (default): con questa modalità viene replicata l'intera sessione. Tale opzione è di solito da preferire quando la sessione è di piccole dimensioni.
- **ATTRIBUTE**: sono replicati soltanto gli attributi che sono stati modificati in aggiunta ad alcune informazioni legate alla sessione. Tale modalità offre un sensibile vantaggio di performance rispetto a quella precedente, in particolare se la sessione è di grandi dimensioni.

- **FIELD**: in questo caso sono replicati solo i singoli campi che sono stati modificati. Questa è la granularità più fine possibile però non è supportata da Infinispan e quindi non potrà essere utilizzata.

Tali opzioni possono essere specificate all'interno del file *WEB-INF/jboss-web.xml* tramite gli elementi *<replication-trigger>* e *<replication-granularity>*, come mostrato nella Figura sottostante.

```
<jboss-web>
    <replication-config>
        <replication-trigger>SET</replication-trigger>
        <replication-granularity>ATTRIBUTE</replication-granularity>
    </replication-config>
</jboss-web>
```

Figura 3.18 - Configurazione Triggering e Granualarity Sessione HTTP

3.2.3 Single Sign On

In generale con il termine Single Sign-On (SSO) ci si riferisce a quel meccanismo che consente a chi è autorizzato all'uso di una certa risorsa di poter utilizzare altre risorse, collegate alla prima, senza dover inserire nuovamente le credenziali di accesso. In particolare parlando di clustering si è interessati alla possibilità di autenticare un utente su un particolare nodo del cluster e di non richiedere le credenziali nel caso in cui l'utente dovesse interagire con un altro nodo. Tale possibilità risulta molto utile in caso di failover, in quanto l'utente potrebbe proseguire la propria sessione su un nodo differente da quello in cui l'ha iniziata in modo del tutto trasparente, senza accorgersi che il primo server ha subito un blocco. In ogni caso se il SSO non è abilitato, non si pregiudica la caratteristica di failover però quando la richiesta dell'utente sarà dirottata verso un nuovo server sarà necessario inserire le credenziali di accesso per proseguire.

Per poter funzionare il meccanismo di SSO si appoggia su una cache distribuita e sulla sua replicazione: dopo che l'autorizzazione ha avuto successo vengono memorizzate e replicate le informazioni per quel determinato utente, se il nodo con il quale sta interagendo l'utente si blocca le informazioni vengono lette dalla cache locale del nuovo nodo che riceve le richieste di quell'utente.

Per quanto riguarda la configurazione del SSO in JBoss AS 7, come è possibile notare dalla Figura 3.9 è già disponibile la definizione della cache necessaria (*<replicated-cache name="sso" mode="SYNC" batching="true"/>*), la quale trovandosi all'interno del cache container "web" utilizzerà lo stesso canale JGroups utilizzato per la replicazione della sessione HTTP. La cache per il SSO è configurata in *replication mode* sincrono, anche se questa sembra la scelta non ottimale dal punto di vista delle performance in realtà non lo è in quanto i dati che devono essere replicati sono di dimensione ridotta, comunque nel caso in cui si abbia un cluster molto grande è ragionevole adottare una cache in *distribution mode*. Per abilitare l'utilizzo del SSO è necessario configurare opportunamente sia il sotto-sistema web che la propria applicazione. La prima configurazione prevede l'inserimento della linea evidenziata nella figura sottostante, in cui si indica qual'è la cache da utilizzare e il fatto che ogni richiesta non deve essere autenticata nuovamente ma possono essere utilizzati opportuni cookies SSO.

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-server="default-host" native="false">
    <connector name="http" protocol="HTTP/1.1" scheme="http" socket-binding="http"/>
    <connector name="ajp" protocol="AJP/1.3" scheme="http" socket-binding="ajp"/>
    <virtual-server name="default-host" enable-welcome-root="true">
        <alias name="localhost"/>
        <alias name="example.com"/>
        <sso cache-container="web" cache-name="sso" reauthenticate="false"/>
    </virtual-server>
</subsystem>
```

Figura 3.19 - Abilitazione SSO sotto-sistema web

Per ogni applicazione è necessario invece aggiungere al file *WEB-INF/jboss-web.xml* gli elementi presenti nella seguente figura in cui si indica che si vuole utilizzare la “valve” per il SSO su cluster.

```
<valve>
    <class-name>org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn</class-name>
</valve>
```

Figura 3.20 - Abilitazione SSO applicazione

3.2.4 Session Beans

Due motivi principali spingono al clustering dei Session beans. Il primo è per distribuire il carico di richieste su diversi server applicativi per migliorare le performance. Il secondo invece è per abilitare la replicazione dello stato per gli Statefull session beans per ottenere fault tollerance. Il load balancing delle richieste è più indicato per i bean Stateless, in quanto essendo privi di stato ciascuna richiesta può essere indirizzata verso un server diverso senza incorrere in problemi. Per i bean Statefull invece di solito si utilizza la sticky session perché tutte le richieste per uno stesso utente devono essere indirizzate verso lo stesso server che mantiene lo stato di interazione per quell’utente. La replicazione dello stato invece per i bean Stateless non ha senso dato che per definizione tali bean non mantengono alcun stato di interazione.

Stateless Session Beans
Per gli Stateless beans la logica di load balancing è inserita all’interno dello stub cliente (detto anche smart proxy) il quale contiene anche la lista dei nodi disponibile nel cluster. Ogni invocazione di metodo viene indirizzata dallo stub verso un determinato nodo, se avviene una modifica nella composizione del cluster, la nuova lista di nodi viene restituita con la successiva invocazione di metodo. Nel caso in cui un nodo fallisse lo stub è in grado di attuare meccanismi di failover tentando l’invocazione su un altro nodo.

Per abilitare il clustering degli Stateless beans è necessario annotare il bean con *@org.jboss.ejb3.annotation.Clustered*. Come impostazione predefinita la strategia di load balancing utilizzata è la Round Robin, comunque è possibile modificarla tramite l’attributo *loadBalancePolicy* della stessa annotazione. Le politiche di load balancing disponibili sono:

- **FirstAvailable** (sticky session): ogni stub seleziona in modo casuale un nodo del cluster e indirizza tutte le invocazioni verso quel nodo. Se il nodo scelto fallisce ne viene scelto un altro sempre casualmente.
- **FirstAvailableIdenticalAllProxies**: tutti gli stub per un determinato bean indirizzano le invocazioni verso lo stesso nodo, scelto in modo casuale. Se il nodo fallisce ne viene scelto in modo casuale un altro.

- **RandomRobin**: ogni richiesta è inviata ad un nodo selezionato in modo casuale.
- **RoundRobin**: le richieste sono inviate ciclicamente verso tutti i nodi del cluster.

Nel caso specifico di myAlma, dato che EJB tier e web tier sono integrati all'interno dello stesso application server gli Stateless beans non sono stati abilitati per il clustering e utilizzano interfacce locali. Le richieste sono quindi inviate dal web tier verso i bean presenti sullo stesso server e non su server remoti. Il load balancing avviene perciò solo a livello di richieste HTTP (sezione 3.2.1), come schematizzato dalla figura sottostante.

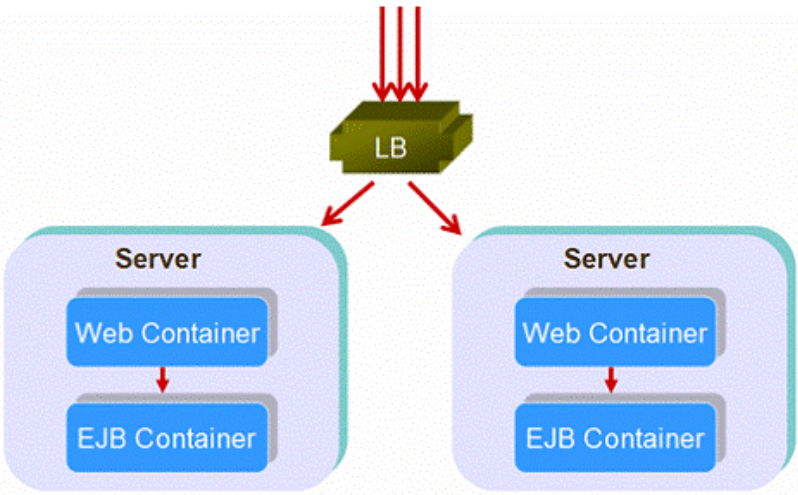


Figura 3.21 - Schema di load balancing

Nel caso in cui i clienti dell'applicazione non siano solo clienti web ma anche applicazioni standalone, non è possibile utilizzare interfacce locali e il clustering deve essere abilitato anche per i bean Stateless.

Statefull Session Beans

L'abilitazione del clustering per gli Statefull beans ha un significato diverso rispetto agli Stateless, infatti in questo caso abilitare il clustering significa abilitare la replicazione dello stato del bean. Per abilitare il clustering si utilizza la stessa annotazione utilizzata per i bean Stateless con l'unica differenza che la sola politica di load balancing che può essere utilizzata è la *FirstAvailable*. Come impostazione predefinita viene utilizzata la cache di default definita all'interno del cache container *ejb* (Figura 3.9), per scegliere un altro cache container è possibile utilizzare l'annotazione *@org.jboss.ejb3.annotation.CacheConfig.*

3.2.5 Entity Beans

Gli Entity bean descritti nella specifica 3.x non sono accessibili da remoto quindi non ha nessun senso (e non è possibile) abilitare il clustering di tali componenti con lo scopo di distribuire il carico di richieste. E' possibile comunque utilizzare una cache per limitare il più possibile le interazioni con il DB, di solito uno dei colli di bottiglia delle applicazioni enterprise. La specifica EJB3 non indica come debba essere gestita la cache per gli Entity beans, però dato che JBoss utilizza Hibernate come provider JPA predefinito è possibile utilizzare i suoi meccanismi. HIbernate prevede il caching di informazioni implementato su due livelli. Il primo è quello classico di JPA in cui i dati letti dal DB sono memorizzati per il tempo di vita della transazione corrente (*EntityManager* in JPA e *Session* in Hibernate). Il secondo livello di caching (Second Level Cache - 2LC) invece, memorizza dati che hanno un tempo di vita maggiore della singola transazione e possono essere condivisi tra più

transazioni evitando così interrogazioni sul DB. Hibernate offre la possibilità di scegliere quale implementazione di cache utilizzare per la 2LC però dato che in JBoss è già integrato il supporto ad Infinispan, quest'ultimo sarà utilizzato come cache provider per Hibernate.

I concetti illustrati in questo paragrafo non sono strettamente legati al clustering, nel senso che è possibile abilitare la 2LC anche in applicazioni che operano su un singolo nodo. Nel caso in cui però si operi all'interno di un cluster la configurazione della cache dovrà essere opportunamente adeguata.

Nella cache di secondo livello possono essere memorizzati quattro tipi di dati:

- **Entity beans**: oggetti Entity memorizzati secondo il proprio identificatore di riga. Infinispan crea una cache per ogni classe di Entity bean.
- **Collezioni**: nel caso in cui un Entity abbia un campo che è una collezione di altre Entity, Hibernate pone nella cache di secondo livello gli identificatori degli Entity referenziati all'interno di tale collezione.
- **Risultato delle query**: vengono memorizzate nella cache le query eseguite, i relativi parametri e gli identificatori degli Entity che sono stati selezionati con la query. Quando la query è successivamente eseguita, gli Entity corrispondenti sono letti dalla cache degli Entity o dal DB.
- **Timestamps**: tale cache è utilizzata in associazione con la cache per le query in quanto memorizza i timestamp di esecuzione delle query e di modifica degli Entity, per evitare che i risultati delle query non siano allineati con il reale stato del DB. Quando viene letto un dato dalla cache delle query il suo timestamp viene confrontato con il timestamp di tutti gli Entity coinvolti nella query, se anche un solo Entity ha un timestamp più vecchio, il risultato della query presente in cache viene invalidato e la query viene eseguita sul DB.

In generale per ogni tipologia di dato memorizzato viene istanziata una cache differente perché ogni dato richiede modalità di gestione differenti, soprattutto se l'applicazione è configurata per il clustering. Ad esempio facendo riferimento alle strategie di caching offerte da Infinispan (sezione 3.1.2) la cache relativa gli Entity (o le collezioni) viene di solito gestita con modalità *Invalidation*, la distribuzione (o la replicazione) infatti oltre ad essere più pesante dal punto di vista del numero di messaggi che devono essere scambiati comporterebbe la copia di Entity su server che non li hanno richiesti, quindi l'intera operazione di copia sarebbe del tutto inutile. Con la modalità locale invece si rischierebbe di utilizzare Entity non aggiornati, in quanto modificati da altri nodi. Per quanto riguarda i timestamps invece l'unica modalità ammessa (altrimenti verrà lanciata un'eccezione) è proprio la replicazione in quanto ci si deve assicurare che tutti i server abbiano in cache tutti i timestamps aggiornati altrimenti si corre il rischio di utilizzare il risultato di una query che non è più valido. La modalità locale è di solito da preferire per la cache dei risultati delle query (a patto di avere abilitata la replicazione dei timestamps) perché come per gli Entity e le collezioni non si ha la certezza che le stesse query siano eseguite su tutti i nodi quindi anche in questo caso la replicazione sarebbe inutile. Anche la modalità invalidazione è inutile per la cache delle query in quanto di fatto le query non sono modificate (nel senso inteso per gli Entity) e la loro invalidazione avviene mediante il meccanismo dei timestamp descritto in precedenza. Riassumendo la modalità di caching locale è indicata per le query, la modalità invalidazione è adatta per Entity e collezioni e la modalità replicazione asincrona è indicata per i timestamps.

Di solito, nelle impostazioni per la cache di secondo livello, non vengono abilitati i cache loaders di Infinispan in quanto non avrebbe senso (porterebbe anzi ad overhead inutile) salvare su un data store persistente dei dati che sono già persistenti e presenti all'interno di un DB.

Configurazione
Per abilitare la cache di secondo livello è necessario configurare alcune proprietà all'interno della propria applicazione, in particolare nel file *META-INF/persistence.xml* se si sta utilizzando JPA oppure negli specifici file di configurazione di Hibernate se si sta utilizzando quest'ultimo direttamente. Tali proprietà sono riportate nella figura sottostante.

```
<shared-cache-mode>ALL</shared-cache-mode>

<properties>
    <property name="hibernate.cache.use_second_level_cache"
              value="true"/>
    <property name="hibernate.transaction.factory_class"
              value="org.hibernate.transaction.CMTTransactionFactory"/>
    <property name="hibernate.transaction.manager_lookup_class"
              value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
</properties>
```

Figura 3.22 - Abilitazione Hibernate 2LC via JPA

Con l'elemento *<shared-cache-mode>* si specifica quali Entity beans devono essere aggiunti alla cache, i valori possibili sono quattro:

- **ALL**: tutti gli Entity appartenenti all'unità di persistenza sono inseriti nella cache.
- **NONE**: praticamente disabilita la 2LC in quanto nessun Entity viene inserito nella cache.
- **ENABLE_SELECTIVE**: vengono inseriti in cache solo gli Entity annotati con *@javax.persistence.Cacheable*.
- **DISABLE_SELECTIVE**: sono inseriti in cache tutti gli Entity tranne quelli che sono annotati con *@javax.persistence.Cacheable(false)*.

La proprietà *hibernate.cache.use_second_level_cache* è invece una proprietà specifica di Hibernate e indica al provider di abilitare la 2LC. Con le proprietà *hibernate.transaction.factory_class* e *hibernate.transaction.manager_lookup_class* si configura Hibernate per utilizzare le transazioni JTA, in questo modo le operazioni sulla 2LC e sul DB sono racchiuse all'interno di un'unica transazione e quindi sono trattate come una sola unità. Se tale configurazione viene omessa ciò che può succedere è che siano utilizzate due transazioni diverse per il salvataggio dei dati sul DB e per l'aggiornamento della cache con il rischio di lasciare la cache con valori obsoleti mentre le modifiche sono state salvate correttamente sul DB. Il valore della seconda proprietà è specifico per ogni application server.

Con la configurazione mostrata in Figura 3.22 vengono abilitate solo le cache per gli Entity beans e per le collezioni, se si vuole abilitare anche la cache per le query (e di conseguenza per i timestamps) è necessario aggiungere la seguente riga alla configurazione:

```
<property name="hibernate.cache.use_query_cache" value="true" />
```

Inoltre le query devono essere “Named” nella nomenclatura di JPA, ossia devono essere dichiarate tramite l’annotazione *@javax.persistence.NamedQuery* come mostrato nella Figura 3.23 e devono impostare a *true* l’hint di Hibernate *org.hibernate.cacheable* per abilitare il cahing.

```
@NamedQuery(name= "findTeachingsByAssistantId",
            query="SELECT t FROM Teaching t, IN (t.contentsRoot.authors) auth WHERE auth.mail=:id",
            hints={ @QueryHint(name = "org.hibernate.cacheable", value = "true") }),
```

Figura 3.23 - Esempio di NamedQuery con abilitazione cache

Come impostazione predefinita Infinispan utilizza per la 2LC delle ben precise configurazioni di cache per ogni tipologia di dato (Figura 3.9). Nel caso in cui sia necessario modificare tale impostazione è possibile modificare direttamente le configurazioni delle cache nel file *standalone-ha.xml* oppure indicare, tramite opportune proprietà in *META-INF/persistence.xml*, quali configurazioni utilizzare per ogni tipo di dato (Figura 3.24). Se necessario è possibile specificare la configurazione per la cache per ogni singola classe Entity (granularità di configurazione molto fine).

```
<property name="hibernate.cache.infinispan.entity.cfg"
          value="custom-entity"/>
<property name="hibernate.cache.infinispan.collection.cfg"
          value="custom-collection"/>
<property name="hibernate.cache.infinispan.query.cfg"
          value="custom-collection"/>
<property name="hibernate.cache.infinispan.timestamp.cfg"
          value="custom-timestamp"/>
```

Figura 3.24 - Configurazione cache per ogni tipo di dato

Collaudo

Dopo aver configurato tutti gli aspetti della 2LC è possibile avviare il server e verificare tramite JConsole se le impostazioni sono state recepite dal server correttamente. Come si nota dalla Figura 3.25 viene avviato un canale JGroups chiamato *hibernate* e sono istanziate le diverse cache per gli Entity (una per ogni classe) e per le query e i timestamps. Nel nome di ogni cache, tra parentesi tonde, è riportata la strategia di caching utilizzata.

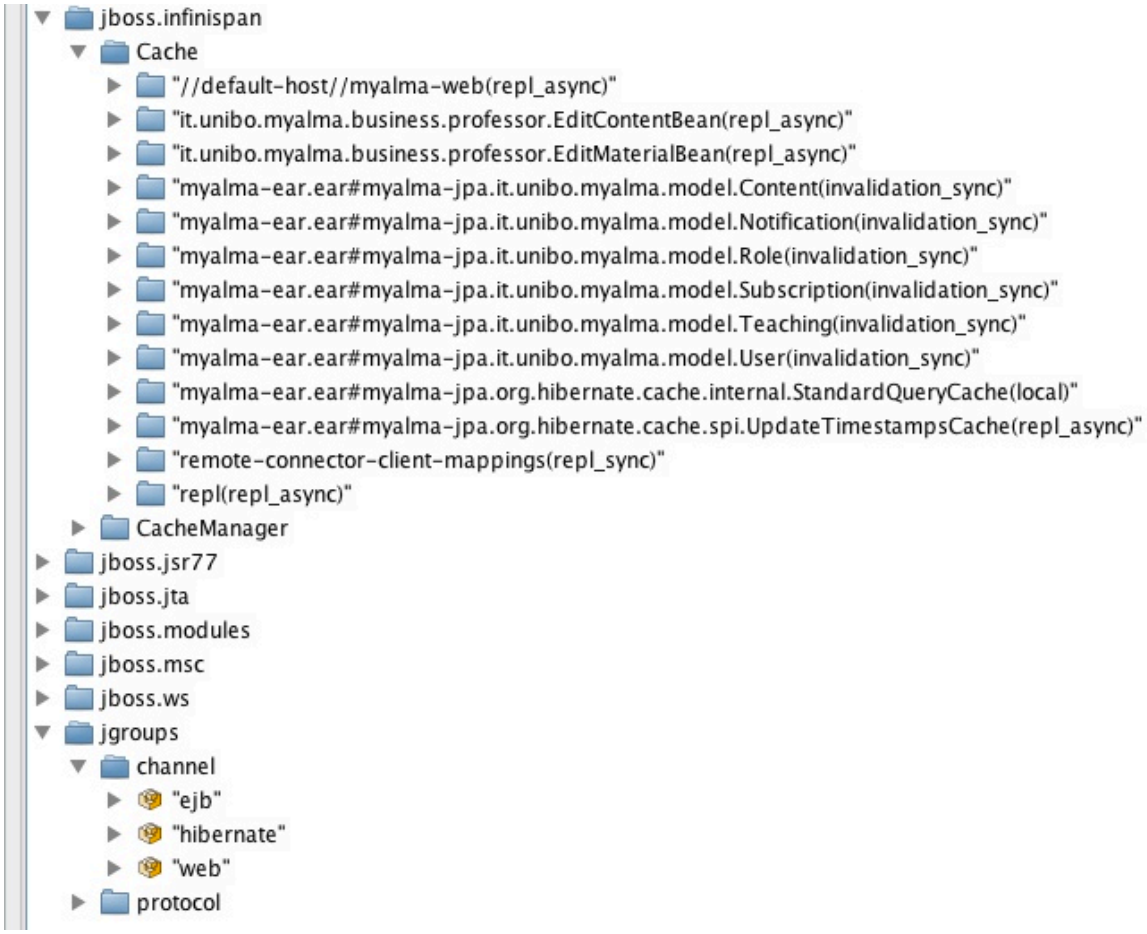


Figura 3.25 - Vista cache e canali JGroups da JConsole

3.2.6 Message Driven Beans

Come accennato nel capitolo 2, il provider per JMS utilizzato da JBoss è HornetQ. Tale provider consente la creazione di gruppi di server JMS permettendo così la distribuzione dei messaggi su tutti i server che compongono il gruppo.

Per abilitare questa funzionalità è necessario configurare opportunamente il sotto-sistema *urn:jboss:domain:messaging:1.1*, indicando innanzitutto che l'ambiente in cui si sta operando è un cluster di server, per fare ciò è sufficiente aggiungere la seguente riga alla configurazione dell'elemento *<hornetq-server>*:

```
<clustered>true</clustered>
```

Il secondo punto da configurare riguarda le connessioni all'interno del cluster, ossia come i diversi server JMS si riconoscono e si connettono per scambiarsi i messaggi. Nella Figura 3.26 è riportata una configurazione comune per le connessioni. L'elemento *<address>* indica quali messaggi subiranno il load balancing, in particolare i messaggi inviati a quale indirizzo. Nell'esempio sono distribuiti tutti i messaggi inviati ad un indirizzo che inizia con *jms*. L'elemento *<connector-ref>* indica il riferimento al connector configurato in un'altra sezione del sotto-sistema: nell'esempio si utilizza il connector *netty* (framework client-server). Con l'elemento *<use-duplicate-detection>* si riconoscono e ignorano i messaggi duplicati. L'elemento *<max-hops>* invece imposta il numero di "salti" su server JMS che un messaggio può fare. Impostando il valore 1 i messaggi saranno inviati solo a server che

sono direttamente collegati con il server corrente, se il valore è maggiore di 1 invece i messaggi possono essere inviati anche a server che sono collegati con quello corrente indirettamente, con altri server che svolgono il ruolo di intermediari. L'elemento *<discovery-group-ref>* rappresenta il riferimento alla configurazione necessaria per definire indirizzo e porta utilizzati dai server per il discovery automatico.

```
<cluster-connections>
    <cluster-connection name="my-cluster">
        <address>jms</address>
        <connector-ref>netty</connector-ref>
        <discovery-group-ref discovery-group-name="dg-group1"/>
        <use-duplicate-detection>true</use-duplicate-detection>
        <max-hops>1</max-hops>
    </cluster-connection>
</cluster-connections>

 <discovery-groups>
    <discovery-group name="dg-group1">
        <group-address>231.7.7.7</group-address>
        <group-port>9876</group-port>
        <refresh-timeout>10000</refresh-timeout>
    </discovery-group>
</discovery-groups>
```

Figura 3.26 - Tipica configurazione per le connessioni in un cluster JMS

La policy di load balancing può essere specificata all'interno dell'elemento *<connection-factory>*. Le policy disponibili sono Round-Robin (*org.hornetq.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy*) e Random (*org.hornetq.api.core.client.loadbalance.RandomConnectionLoadBalancingPolicy*). E' comunque possibile realizzare la propria policy implementando l'interfaccia *org.hornetq.api.core.client.loadbalance.ConnectionLoadBalancingPolicy*.

3.3 Prestazioni

In questa sezione viene descritto l'approccio utilizzato per la misura e l'analisi delle prestazioni dell'applicazione con diverse configurazioni.

3.3.1 Piano

Per la misura delle prestazioni è stato configurato un cluster di quattro nodi fisici utilizzando il Laboratorio 2 della Facoltà di Ingegneria di Bologna (Figura 3.27).

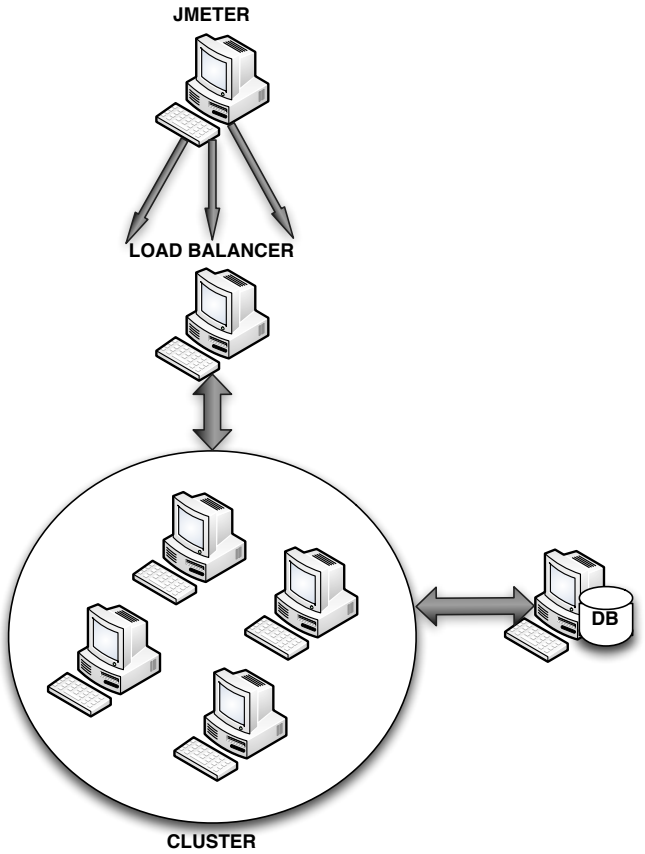


Figura 3.27 - Configurazione macchine per misura prestazioni

Oltre alle quattro macchine che ospitano gli application server sono state utilizzate anche tre macchine aggiuntive, la prima con lo scopo di fare da front-end per tutte le richieste provenenti dagli utenti e quindi vi è stato installato il load balancer mod_cluster, la seconda per ospitare il DB MYSQL e la terza per simulare gli utenti e inviare le richieste all'applicazione.

Tutte le macchine utilizzate sono dotate di Processore Intel Core 2 Duo E7400 2.80GHZ e 2048 MB di RAM DDR2.

Scenari applicativi

Per poter misurare le prestazioni di myAlma nel modo più realistico possibile sono stati realizzati 10 scenari applicativi in cui sono state simulate diverse tipologie di utenti. Le tipologie sono:

- **Login:** utenti che fanno login nell'applicazione e rimangono inattivi per un certo tempo prima di eseguire il logout.
- **Aggiunta Categoria/Informazione**: utenti che svolgono tutte le operazioni necessarie per aggiungere una nuova categoria o una nuova informazione.
- **Modifica Contenuti**: gli utenti di questa tipologia modificano due contenuti diversi.
- **Navigazione nei corsi e nei contenuti**: gli utenti entrano in diversi corsi e in diversi contenuti ma non apportano alcuna modifica a quest'ultimi.
- **Rimozione Informazioni/Categorie**: gli utenti di questa tipologia rimuovo una categoria o un'informazione.

Gli scenari contengono tutte le tipologie di utenti ma si distinguono l'uno dall'altro per il numero di utenti per ogni tipologia, come mostrato nella tabella sottostante. Il numero di utenti è via via crescente così da simulare carichi via via crescenti.

	Login	Agg. Cat.	Agg. Inf.	Modifica	Navigazione	Rim. Info.	Rim. Cat.	Totale
Scenario 1	0	1	0	0	0	0	0	1
Scenario 2	1	2	2	2	2	1	0	10
Scenario 3	10	20	20	20	20	5	5	100
Scenario 4	50	100	100	100	100	25	25	500
Scenario 5	100	200	200	200	200	50	50	1000
Scenario 6	100	250	200	200	200	50	50	1050
Scenario 7	100	250	250	200	200	50	50	1100
Scenario 8	100	250	250	250	250	50	50	1200
Scenario 9	150	300	300	300	300	75	75	1500
Scenario 10	150	350	350	350	350	75	75	1700

Tabella 3.1 - Composizione scenari applicativi

Lo scenario 1 presenta un solo utente, però lo scenario stesso viene ripetuto 10 volte per avere delle misure più realistiche, infatti se si eseguisse lo scenario solo una volta si avrebbero dei tempi di risposta decisamente lungi in quanto il server con le prime richieste prepara tutte le risorse necessarie, invece eseguendo lo scenario 10 volte questo effetto viene in media mitigato.
Altri due parametri che sono stati presi in considerazione per la progettazione degli scenari applicativi sono il *Ramp-up Period* e il *Think Time*. Il primo indica il tempo entro il quale tutti gli utenti simulati sono avviati e deve essere sufficientemente grande da non sovraccaricare eccessivamente il server all'inizio della misura. Per myAlma è stato scelto un *Ramp-Up Period* di 500 secondi (8 minuti e 20 secondi) per tutti gli scenari tranne che per il primo (1 secondo), in quanto si ha l'obbiettivo di raggiungere l'avvio di 2 utenti al secondo con lo Scenario 5 (1000/500). Il secondo parametro, *Think Time*, rappresenta il tempo di attesa tra una richiesta e la successiva da parte di ogni utente. Tale parametro è necessario per rendere gli scenari ancora più realistici in quanto le richieste non sono inviate dall'utente una immediatamente dopo l'altra ma intercorre un certo tempo tra le due in cui l'utente "pensa" appunto, altrimenti si rischierebbe di sovraccaricare il server inutilmente in quanto anche in uno scenario reale non si avrebbe mai una tale frequenza di richieste. I *Think Time* scelti per myAlma sono uniformemente distribuiti tra il valore massimo e il valore minimo riportati nella Tabella 3.2.

Login	Agg. Cat.	Agg. Info.	Modifica	Navigazione	Rimozione
1 - 5	3 - 10	3 - 20	3 - 20	3 - 5	3 - 5

Tabella 3.2 - Think Time myAlma (tempi in secondi, Min-Max)

Ogni utente della tipologia Login rimane registrato all'intero dell'applicazione per un tempo uniformemente distribuito tra 2 e 4 minuti. Il *Think Time* per l'aggiunta delle informazioni e per la modifica dei contenuti è maggiore rispetto agli altri in quanto le informazioni che l'utente deve inserire in questi due casi sono di più rispetto agli altri, quindi si simula la permanenza maggiore nella pagina di inserimento dei dati.

Configurazioni in esame
Si è cercato di analizzare configurazioni che potessero soddisfare diverse esigenze applicative che vanno dalle massime performance possibili ma con scarso supporto all'affidabilità fino alla massima affidabilità con degrado però delle prestazioni complessive. I parametri sui quali si è esclusivamente intervenuto sono quelli riguardanti le cache per la replicazione delle varie tipologie di stato, per quanto riguarda JGorups invece non sono state apportate modifiche alle impostazioni di default che utilizzano canali basati su UDP. Per il load balancer invece sono state utilizzate le metriche *cpu*, *heap* e *sessions* con sticky session abilitata.

In breve le configurazioni sono le seguenti:

- **conf0-SINGLE**: questa configurazione non prevede l'utilizzo di un cluster e tutte le richieste sono gestite da un solo server.
- **conf1-dist-ASYNC**: in questo caso è previsto l'utilizzo di cache in modalità *distribution* asincrona sia per la sessione HTTP che per lo stato dei bean Statefull. Il numero di repliche mantenute all'interno del cluster è 1, ossia lo stato viene replicato su un solo server. Questa è la prima configurazione che utilizza il cluster mostrato nella Figura 3.27.
- **conf2-dist-SYNC**: come la conf1 con l'unica differenza che le comunicazioni tra le cache sono sincrone.
- **conf3-repl-ASYNC**: sia la sessione HTTP che lo stato dei bean Statefull utilizzano cache in modalità replicazione asincrona. In questo caso specifico il numero di repliche mantenute all'interno del cluster è pari a 3.
- **conf4-dist-ASYNC-2LC**: tale configurazione è identica alla conf1 con in più l'abilitazione della 2LC di Hibernate. In particolare la 2LC è stata configurata con le impostazioni predefinite (e consigliate) ossia: query in modalità locale con timestamps in modalità replicazione asincrona e Entity e collezioni in modalità *invalidation* sincrona.
- **conf5-dist-ASYNC-2LC-noquery**: come la conf4 disabilitando però la cache per le query e per i timestamp.
- **conf6-dist-ASYNC-2LC-noquery-ATTRIBUTE**: come la conf5 con l'impostazione della granularità di replicazione della sessione HTTP al valore ATTRIBUTE.

La conf0 è quella da cui ci si deve aspettare le massime performance in quanto non c'è replicazione dello stato tra i server e quindi operano l'uno indipendentemente dall'altro, per questo motivo si è deciso di testare le prestazioni con un solo server in quanto il risultato è facilmente estendibile a più di uno. La conf0 è comunque la configurazione più rischiosa dal punto di vista dell'utente perché un malfunzionamento del server con il quale sta interagendo comporterebbe la perdita del suo lavoro fino a quel momento, senza alcuna possibilità di ripristino. Le configurazioni conf1 e conf2 invece garantisco una discreta affidabilità per quanto riguarda il mantenimento dello stato di interazione utente-server, con tali configurazioni infatti il cluster riesce a sopportare la caduta di un server in modo trasparente per l'utente. La conf2 inoltre utilizzando la distribuzione sincrona aumenta ulteriormente il livello di affidabilità assicurandosi, prima di inviare la risposta al cliente, che la replicazione sul server di supporto sia avvenuta con successo, ovviamente ciò va a

ledere leggermente le prestazioni. La configurazione conf3 è quella che presenta il maggior livello affidabilità in quanto la replicazione dello stato viene eseguita su tutti i server del cluster, dato che da tale configurazione ci si aspetta le performance peggiori si è deciso di utilizzare comunicazioni asincrone per cercare di limitare l'effetto negativo della replicazione su tutti i nodi. Nelle configurazioni conf4, conf5 e conf6 è stata utilizzata come base la conf1, dalla quale ci si aspetta le performance migliori (tra quelle riguardanti il cluster ovviamente), e si è abilitata la 2LC per verificare se e in che misura tale meccanismo permetta l'incremento delle prestazioni. Nella conf6 inoltre è stata impostata una granularità più fine per la replicazione per verificare se il fatto di dover replicare meno informazioni porta ad un miglioramento delle prestazioni.

```
<distributed-cache name="dist" owners="1" mode="ASYNC" batching="true">
    <locking striping="false" isolation="REPEATABLE_READ"/>
    <transaction locking="OPTIMISTIC"/>
    <eviction strategy="LRU"/>
    <file-store passivation="true" fetch-state="false" purge="true"/>
</distributed-cache>
```

Figura 3.28 - Configurazione cache in distribution/replication mode

Nella Figura 3.28 è riportata la configurazione utilizzata per le cache in modalità *distribution*. Gli stessi parametri sono impostati anche per la configurazione in modalità replicazione, l'unica differenza è che al posto dell'elemento *<distributed-cache>* deve essere utilizzato l'elemento *<replicated-cache>*. Tramite l'attributo *owners* si è impostato il numero di repliche da mantenere all'interno del cluster, con gli elementi *<locking>* e *<transaction>* sono stati impostati il livello di isolamento a *REPEATABLE_READ* e la gestione ottimistica delle transazioni. In particolare l'attributo *striping* indica se devono essere mantenuti dei lock condivisi sulle entry delle cache (valore *true*) oppure se deve essere creato un lock distinto per ogni entry che viene utilizzata (valore *false*), impostandolo a *false* si consuma più memoria ma il livello di concorrenza aumenta. Come strategia di eviction è stata scelta la Least Recently Used ed è stato abilitato un cache loader che utilizza un file come data store (*<file-store>*) abilitando però la passivazione, quindi le entry sono scritte nel data store solo quando subiscono eviction. Nella Figura 3.29 sono invece riportate le configurazioni per le diverse cache utilizzate da Hibernate: sostanzialmente sono quelle predefinite con l'unica differenza che è stato impostato il livello di isolamento a *READ_COMMITTED* perché il livello *REPEATABLE_READ* è inutile in una cache di secondo livello, infatti eventuali letture ripetute non raggiungeranno la cache di secondo livello ma saranno gestite dalla cache di primo livello (EntityManager in JPA o Session in Hibernate) e non si presenteranno quindi problemi di *non-repeatable-reads*.

```
<cache-container name="hibernate" default-cache="local-query">
   <transport lock-timeout="60000"/>
   <local-cache name="local-query">
       <locking isolation="READ_COMMITTED"/>
       <transaction mode="NONE" locking="OPTIMISTIC"/>
       <eviction strategy="LRU" max-entries="10000"/>
       <expiration max-idle="100000"/>
   </local-cache>
   <invalidation-cache name="entity" mode="SYNC">
       <locking isolation="READ_COMMITTED"/>
       <transaction mode="NON_XA" locking="OPTIMISTIC"/>
       <eviction strategy="LRU" max-entries="10000"/>
       <expiration max-idle="100000"/>
   </invalidation-cache>
   <replicated-cache name="timestamps" mode="ASYNC">
       <locking isolation="READ_COMMITTED"/>
       <transaction mode="NONE" locking="OPTIMISTIC"/>
       <eviction strategy="NONE"/>
   </replicated-cache>
</cache-container>
```

Figura 3.29 - Configurazione 2LC

Parametri misurati e Strumenti

Per poter confrontare le diverse configurazioni in termini di prestazioni si è deciso di misurare il tempo di risposta medio delle richieste inviate dagli utenti simulati: minore è il tempo di risposta maggiore sono le prestazioni. Inoltre si è deciso di misurare il carico dei server in termini di utilizzo di CPU e memoria heap per verificare che impatto hanno le configurazioni su queste parti del sistema hardware per capire eventualmente come dovrebbe essere assemblato un sistema reale.

Per simulare gli utenti è stato utilizzato JMeter. JMeter è uno strumento open-source, scritto in Java che permette il load testing di un gran numero di tipologie di server (web, DB, server JMS, ...), in particolare per il testing di myAlma è stata utilizzata la capacità di inviare richieste HTTP. JMeter si presta molto bene all'organizzazione di test in cui intervengono diverse tipologie di utenti, infatti è sufficiente creare un *Thread Group* per ogni tipologia e per quel gruppo è poi possibile impostare tutti i parametri utili, come, *Think Time* (*Uniform Random Timer*), *Ramp-Up Period* e numero di utenti. Le richieste possono inoltre essere parametrizzate e i parametri possono essere letti da un file csv, questo è ciò che avviene per i gruppi *Rimozione Informazioni* e *Rimozione Categorie* in cui gli identificatori dei contenuti da eliminare sono letti da due file csv. Nella figura sottostante viene mostrato uno degli scenari applicativi di myAlma in JMeter.

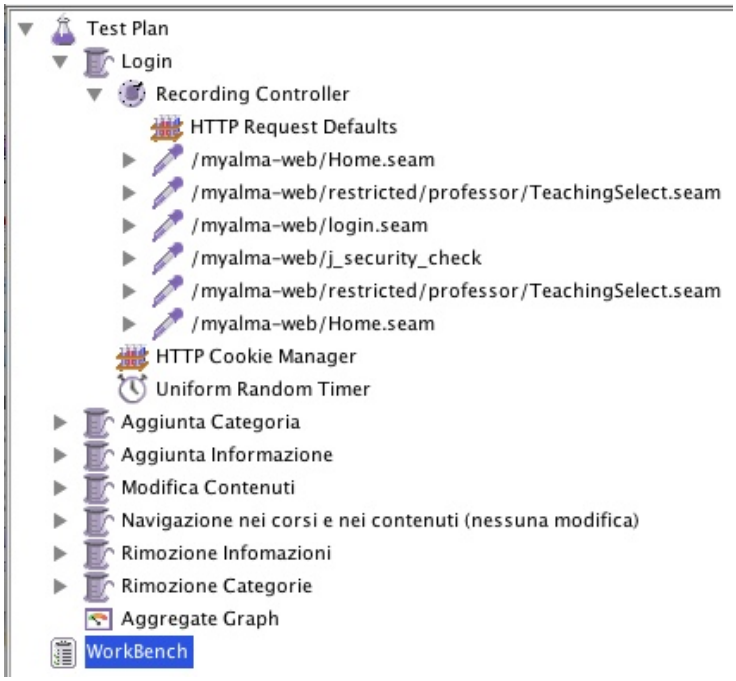
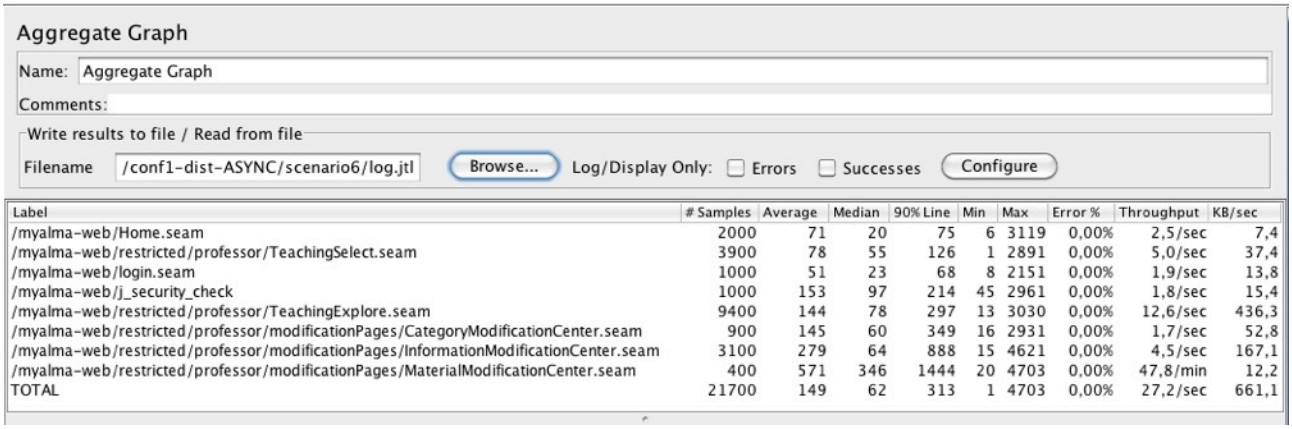


Figura 3.30 - Scenario Applicativo in JMeter

Per la misura dei parametri riguardanti le richieste, JMeter mette a disposizione diversi Listener, in particolare per il testing di myAlma è stato utilizzato il Listener chiamato *Aggregate Graph* (Figura 3.31). Tale componente permette innanzitutto la misurazione del tempo medio di riposta per ogni pagina e inoltre offre altre statistiche come il valore mediano, il valore minimo e massimo, la linea a 90% ossia il tempo di risposta all'interno del quale cadono il 90% delle richieste e la percentuale degli eventuali errori presentati all'utente. Con un solo Listener è quindi possibile avere tutte le informazioni utili a capire se il test è andato a buon fine (non ci sono stati errori) e quali livelli di prestazioni si sono raggiunti.


Aggregate Graph

Name: Aggregate Graph

Comments:

Write results to file / Read from file

Filename /conf1-dist-ASYNC/scenario6/log.jtl Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
/myalma-web/Home.seam	2000	71	20	75	6	3119	0,00%	2,5/sec	7,4
/myalma-web/restricted/professor/TeachingSelect.seam	3900	78	55	126	1	2891	0,00%	5,0/sec	37,4
/myalma-web/login.seam	1000	51	23	68	8	2151	0,00%	1,9/sec	13,8
/myalma-web/j_security_check	1000	153	97	214	45	2961	0,00%	1,8/sec	15,4
/myalma-web/restricted/professor/TeachingExplore.seam	9400	144	78	297	13	3030	0,00%	12,6/sec	436,3
/myalma-web/restricted/professor/modificationPages/CategoryModificationCenter.seam	900	145	60	349	16	2931	0,00%	1,7/sec	52,8
/myalma-web/restricted/professor/modificationPages/InformationModificationCenter.seam	3100	279	64	888	15	4621	0,00%	4,5/sec	167,1
/myalma-web/restricted/professor/modificationPages/MaterialModificationCenter.seam	400	571	346	1444	20	4703	0,00%	47,8/min	12,2
TOTAL	21700	149	62	313	1	4703	0,00%	27,2/sec	661,1

Figura 3.31 - Aggrgate Graph in JMeter

Per la misurazione dell'utilizzo di CPU e heap è stato preso in considerazione inizialmente JeeObserver che è uno strumento open-source e molto valido che permette la misurazione di tali parametri ed eventualmente altri (sessioni attive, thread, tempo di esecuzione dei metodi, ...) però ancora non funzionante su JBoss AS 7, quindi si è deciso di utilizzare VisualVM 1.3.4. Anche questo è uno strumento open-source integrato in

diversi JDK che permette la misura in tempo reale dei parametri quali la CPU e la memoria utilizzata oltre al profiling dei metodi. Per poter registrare l'andamento dell'utilizzo di CPU e memoria è stato necessario installare un plugin (*Tracer-Monitor Probes*) che aggiunge una sezione (*Tracer*) in cui è possibile scegliere cosa registrare e far partire la registrazione. Nella Figura 3.32 viene mostrata la registrazione di CPU, GC e heap.

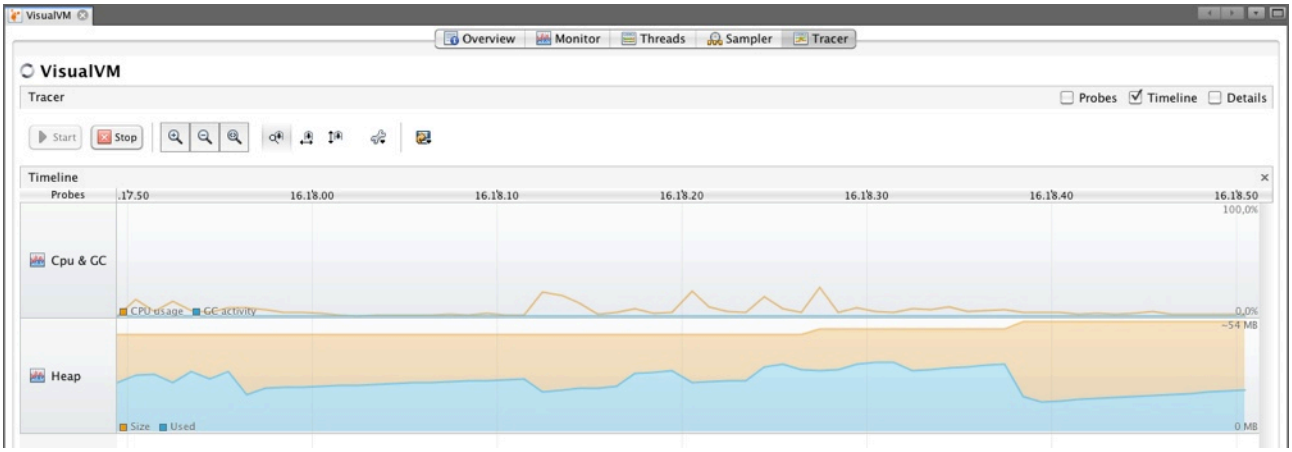


Figura 3.32 - Sezione Tracer in VisualVM

3.3.2 Analisi

Nella Tabella 3.3 sono riportati i tempi medi di risposta per le configurazioni 0, 1, 2, 3, 5 e 6. La configurazione conf4 con cache per le query abilitata non è stata riportata in quanto non è stato possibile andare oltre lo scenario 5 (1000 utenti), in particolare nello scenario 5 i server cominciano a lanciare eccezioni riguardanti le cache e si presentano errori nel riepilogo del listener Aggregate Graph. Nello scenario 6 invece il test non termina dopo più di 45 minuti di esecuzione e i server non fanno altro che lanciare eccezioni. La figura 3.33 mostra il profilo di CPU e heap di uno dei server durante l'esecuzione dello scenario 6 e si nota come sembri particolarmente scarico ma praticamente non svolge nessuna operazione utile. Per questi motivi la configurazione conf4 è stata scartata poiché non sono stati raggiunti neanche 1000 utenti. Tale risultato può essere dovuto al fatto che la cache relativa alle query viene invalidata ogni volta che c'è un cambiamento nel DB, questo comporta la gestione di molti messaggi di invalidazione e soprattutto uno scarso sfruttamento della cache stessa che presenta una percentuale di hit molto bassa: in sostanza è come se la cache non ci fosse però c'è il peso della sua gestione. Per questi motivi la cache per le query risulta molto più efficace in applicazioni che prevedono principalmente letture, oppure con DB relativamente ampi in cui è meno probabile che i dai letti siano modificati dopo un breve periodo.

Dalla tabella 3.3 si nota che la configurazione che presenta i tempi di risposta medi più bassi è la conf0 in cui non è stato costituito un cluster e le richieste sono gestite da un solo server. Tale risultato è del tutto ragionevole in quanto il fatto di non dover replicare lo stato riduce notevolmente il numero di operazioni che un server deve eseguire per ogni richiesta ricevuta, inoltre le operazioni che non vengono eseguite sono operazioni che riguardano comunicazioni di rete e quindi presentano in genere tempi decisamente lunghi rispetto a operazioni che coinvolgono solo la CPU.

Utenti	conf0	conf1	conf2	conf3	conf5	conf6
1	82	94	92	158	86	153
10	77	113	141	183	120	176
100	45	86	92	96	63	90
500	51	88	96	118	66	100
1000	101	149	168	703	144	367
1050	126	147	189	978	159	1124
1100	141	272	283	1336	229	1818
1200	226	386	451	1887	600	5766
1500	1128	1494	1957	5778	1465	9106
1700	4030	13616	4465	14091	3966	9149

Tabella 3.3 - Tempi Medi di Risposta

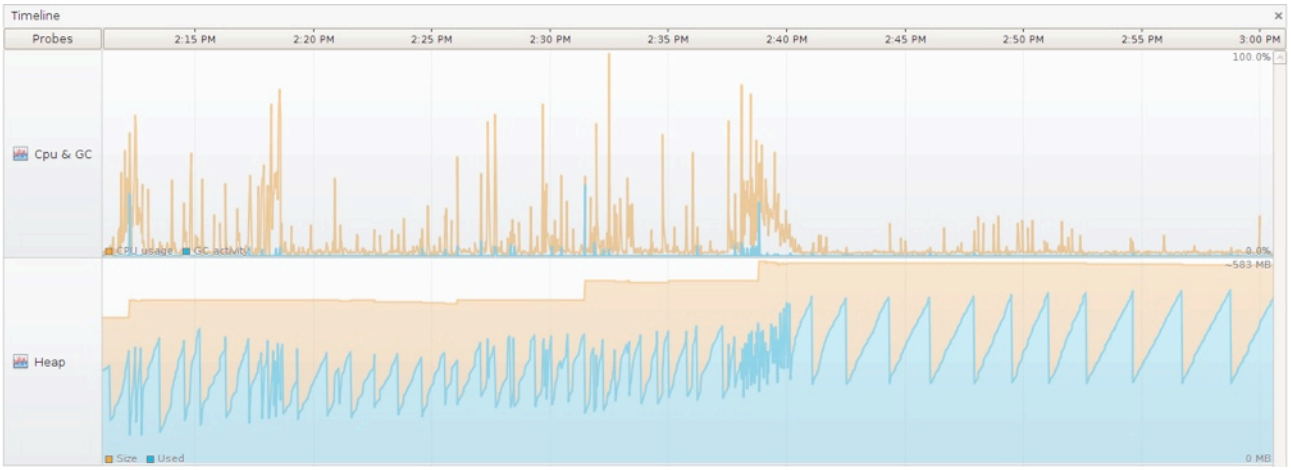
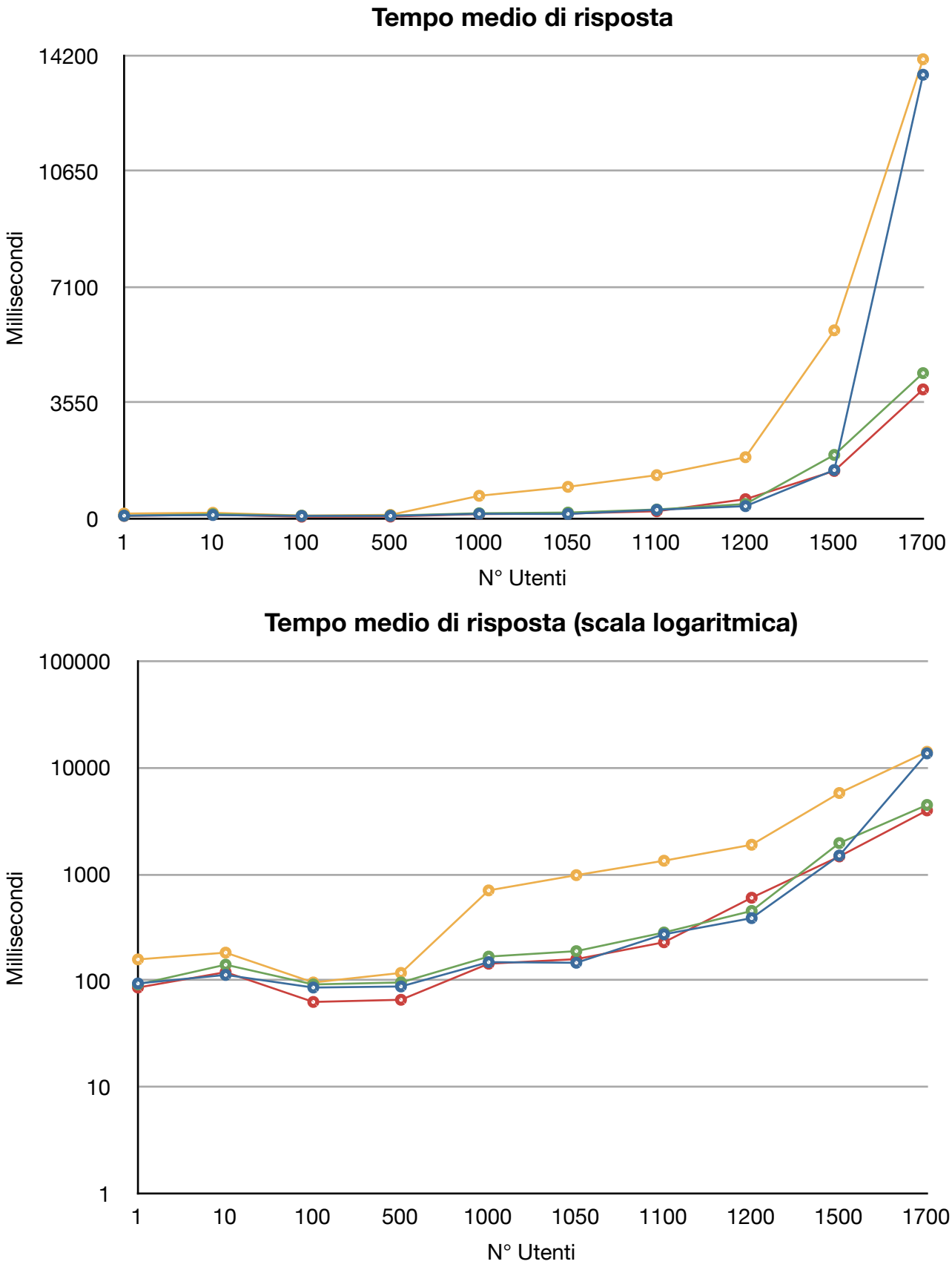


Figura 3.33 - Profilo Esecuzione Scenario 6 conf4-dist-ASYNC-2LC

Il colore di sfondo delle celle, nella tabella 3.3, indica com'è terminata l'esecuzione dei diversi scenari, in particolare: l'assenza di colore indica che il test è terminato senza alcun errore, il colore arancione indica che il test è terminato senza alcun errore per l'utente (colonna Error in JMeter) ma che alcuni messaggi JMS che rappresentano le notifiche per gli utenti non sono stati processati e quindi alcune notifiche non sono state scritte sul DB, il colore rosso invece indica che alcune notifiche non sono state scritte sul DB e in più sono stati riportati alcuni errori per l'utente. Il fatto che alcune notifiche non siano scritte correttamente sul DB potrebbe essere dovuto a diversi fattori ma quello che sembra più probabile analizzando i log è lo scadere di alcuni timeout riguardanti le transazioni che comunicano con il DB, il problema potrebbe essere risolto incrementando tali tempi di timeout oppure una soluzione migliore sarebbe quella di replicare anche il data tier costituendo un cluster di DB.


Tempo medio di risposta
14200
10650
7100
3550
0
Millisecondi
1
10
100
500
1000
1050
1100
1200
1500
1700
N° Utenti
Tempo medio di risposta (scala logaritmica)
100000
10000
1000
100
10
1
Millisecondi
1
10
100
500
1000
1050
1100
1200
1500
1700
N° Utenti
Distribution Mode ASYNC (conf1)
Distribution Mode SYNC (conf2)
Replication Mode ASYNC (conf3)
Distribution Mode ASYNC, 2LC noquery (conf5)

Utenti	Differenza conf0 - conf1	Differenza conf1 - conf2	Differenza conf1 - conf3	Differenza conf1 - conf5	Differenza conf5 - conf6
1	-12.77	2.17	-40.51	9.30	-43.79
10	-31.86	-19.86	-38.25	-5.83	-31.82
100	-47.67	-6.52	-10.42	36.51	-30.00
500	-42.05	-8.33	-25.42	33.33	-34.00
1000	-32.21	-11.31	-78.81	3.47	-60.76
1050	-14.29	-22.22	-84.97	-7.55	-85.85
1100	-48.16	-3.89	-79.64	18.78	-87.40
1200	-41.45	-14.41	-79.54	-35.67	-89.59
1500	-24.50	-23.66	-74.14	1.98	-83.91
1700	-70.40	204.95	-3.37	243.32	-56.65

Tabella 3.4 - Differenza percentuale tra i tempi di risposta medi delle diverse configurazioni

Dai grafici della pagina precedente e dalla Tabella 3.4, che riporta le differenze percentuali tra i tempi di risposta medi delle diverse configurazioni, si nota che la configurazione in cui è prevista la replicazione asincrona su tutti i nodi (conf3) presenta i tempi di risposta più alti, mentre la configurazione con modalità distribuzione asincrona (conf1) è decisamente migliore della conf3 e leggermente migliore di quella con distribuzione sincrona (conf2), escludendo l'ultimo scenario in quanto durante l'esecuzione con la conf1 tutti i server si sono bloccati e ne sono risultati tempi di risposta molto elevati, infatti la differenza tra conf1 e conf2 per 1700 utenti indica che la conf1 ha tempi più elevati del 200% rispetto alla conf2 mentre per meno utenti la conf1 presenta tempi più piccoli di quelli per la conf2 (la differenza è infatti sempre negativa). Dalla Tabella 3.4 si nota inoltre come abilitare la cache di secondo livello di Hibernate comporti in genere un vantaggio (la differenza tra conf1 e conf5 è in genere maggiore di zero indicando che i tempi per la conf1 sono maggiori di quelli per la conf5) ma che tale vantaggio non è così sostanziale e in alcuni casi è addirittura assente, infatti per certi scenari i tempi di risposta sono più piccoli per la conf1 che non prevede la 2LC abilitata. Tale risultato è probabilmente dovuto al fatto che il DB utilizzato ha delle dimensioni relativamente piccole, quindi gli Entity beans memorizzati in cache sono invalidati molto frequentemente poiché sono modificati da altri utenti, sicuramente in uno scenario reale in cui la modifica degli stessi contenuti da parte di diversi docenti è meno frequente, l'abilitazione della 2LC porterebbe a delle prestazioni ancora superiori rispetto a quelle emerse da questi test. Analizzando la tabella 3.4 si nota anche che la conf6, nonostante abiliti un livello di granularità della replicazione più fine rispetto alla replicazione di tutta la sessione, non permette di ottenere prestazioni migliori, in particolare al crescere del numero di utenti la differenza in termini di tempo di risposta medio tra questa configurazione e la conf5 aumenta. Tale risultato è sicuramente sostenuto dal fatto che la sessione HTTP nel caso di myAlma ha delle dimensioni decisamente piccole, quindi conviene replicarla per intero piuttosto che determinare quali

attributi replicare e replicare solo quelli. Probabilmente il fatto di dover individuare quali attributi hanno subito una modifica comporta un overhead maggiore rispetto alla replicazione completa della sessione.

Nelle tabelle 3.5 e 3.6 sono riportati i dati relativi al consumo di CPU, l'intervento del Garbage Collector, la memoria riservata per JBoss e l'effettiva memoria utilizzata per un nodo del cluster (o per l'unico server nella conf0). Sono riportati i dati solo per tre scenari (100, 1000 e 1200 utenti) in quanto se si fossero inseriti anche gli altri scenari le differenze tra uno scenario e l'altro sarebbero state troppo piccole e quindi il confronto tra le configurazioni sarebbe risultato meno efficace. Tutti i dati riportano valori medi per ogni parametro misurato.

Utenti	conf0	conf1	conf2	conf3	conf5	conf6
100	7.58 - 0.09	4.23 - 0.04	4.37 - 0.08	5.09 - 0.10	3.66 - 0.06	4.82 - 0.05
1000	45.03 - 0.88	24.27 - 0.72	18.63 - 0.60	26.25 - 1.13	19.83 - 0.76	26.17 - 0.80
1200	57.65 - 1.73	27.30 - 0.85	24.61 - 0.71	31.56 - 1.55	34.02 - 1.17	21.73 - 0.75

*Tabella 3.5 - Utilizzo medio CPU e intervento Garbage Collector in percentuale (*CPU - GC*)*

Utenti	conf0	conf1	conf2	conf3	conf5	conf6
100	311 - 157	401 - 193	399 - 194	387 - 190	461 - 218	458 - 227
1000	541 - 314	505 - 272	540 - 275	568 - 316	522 - 281	525 - 300
1200	585 - 339	563 - 303	589 - 328	658 - 368	573 - 298	615 - 360

Tabella 3.6 - Media dimensione memoria e media utilizzo memoria in MB (Dimensione - Utilizzata)

La prima cosa che si nota dai dati è che nella conf0 il carico dell'unico server presente è decisamente più alto rispetto al carico di un server all'interno del cluster: d'altra parte la distribuzione del carico è uno degli obbiettivi del clustering. Nella conf0 inoltre la memoria utilizzata è in genere minore rispetto alle altre configurazioni in quanto non deve essere mantenuto lo stato replicato dagli altri server, però la differenza non è molto alta, probabilmente perché il cluster è di piccole dimensioni e la dimensione degli stati che devono essere replicati è piccola. La differenza in termini di memoria utilizzata si nota invece tra la conf1 e la conf3 in cui la replicazione dello stato di tutti i server porta ad un aumento della memoria utilizzata di circa 65 MB per 1200 utenti. Un leggero aumento di memoria utilizzata è presente anche per le configurazioni conf5 e conf6 rispetto a conf1 in quanto nelle prime è abilitata la 2LC e quindi più dati devono essere mantenuti in memoria. Andamento analogo si manifesta anche per l'utilizzo di CPU, infatti la conf3 presenta in media un carico leggermente maggiore rispetto alla conf1, mentre la conf5 impegna maggiormente la CPU soprattutto per molti utenti (1200) probabilmente perché aumenta la frequenza di invalidazione degli Entity.

4. Conclusioni e Sviluppi Futuri

L'utilizzo di un cluster, soprattutto se di grandi dimensioni, è una scelta valida quando si vuole suddividere il carico di richieste su nodi fisici che dal punto di vista hardware sono poco costosi e quando si vuole garantire all'utente continuità di servizio anche a fronte di fallimenti o danni fisici. La chiave del successo nell'utilizzo di un cluster sta nella sua configurazione complessiva e in particolare nella configurazione della replicazione dello stato di interazione. La configurazione ottimale dipende fortemente dal tipo di applicazione o servizio che si deve fornire, comunque in generale, dai test effettuati in questa attività sono emerse alcune regole:

- a meno che l'affidabilità e l'alta disponibilità non sia un requisito centrale evitare di effettuare troppe repliche dello stato;
- utilizzare comunicazioni asincrone basate su UDP dove possibile in quanto permettono di ottenere buone performance, soprattutto se il numero di repliche da mantenere è molto elevato;
- l'abilitazione della 2LC deve essere sempre presa in considerazione per migliorare le performance mentre la scelta per la cache per le query deve essere ponderata con attenzione analizzando la propria applicazione e magari effettuando dei test per confermare la scelta;
- discorso analogo a quello per le query può essere fatto per la scelta della granularità di replicazione: conviene verificare l'effettivo guadagno in termini prestazionali prima di rendere disponibile l'applicazione agli utenti.

Nel corso dell'attività sono stati messi alla prova diversi scenari realistici, comunque ci si potrebbe spingere oltre e raggiungere un livello di fedeltà, rispetto a configurazioni effettivamente utilizzate in produzione, più elevato, andando innanzitutto ad aumentare sensibilmente le dimensioni del cluster (decine di nodi o più), sfruttando le possibilità di clustering del DBMS e impiegando più load balancer. Al momento infatti i due colli di bottiglia principali sono rappresentati dal load balancer e dal DB. Per spingersi ancora oltre si potrebbe misurare le prestazioni di un cluster all'interno di una rete ad alta velocità per minimizzare i tempi di replicazione tra i nodi.

Bibliografia

- Francesco Marchioni (2011), *JBoss AS 7 - Configuration, Deployment and Administration*, PACKT Publishing
- Dan Allen (2009), *Seam in Action - Covers Seam 2.0*, Manning
- Jim Farley (2007), *Practical JBoss Seam Projects*, Apress
- Max Katz, Ilya Shaikovsky (2011), *Practical RichFaces*, Apress
- Christian Baver, Gavin King (2007), *Java Persistence with Hibernate*, Manning
- Andrew Lee Rubinger, Bill Burke (2010), *Enterprise JavaBeans 3.1*, Sesta Edizione, O'Reilly
- Documentazione web