UNIVERSITÀ DEGLI STUDI DI UDINE

Corso di Sistemi Distribuiti

Professor Marino Miculan

Dottor Marco Peressotti

DISTRIBUTED COMPUTING SYSTEM

Cinzia Piccolo
188471

Francesca Palù
100908

ANNO ACCADEMICO 2014-2015

Introduzione

I sistemi peer-to-peer permettono di accedere a dati e risorse collocate su diverse macchine all'interno della rete. Tutti gli utenti contribuiscono ai servizi del sistema con le stesse funzioni e responsabilità, senza la presenza di server centrali.
Un algoritmo molto usato per il collocamento distribuito delle risorse è descritto come *distributed hash table* (DHT), che utilizza funzioni hash per memorizzare oggetti in modo univoco all'interno della rete. Esso mette inoltre a disposizione operazioni di inserimento, lettura e cancellazione degli oggetti.
Queste architetture permettono la realizzazione di sistemi di calcolo distribuito molto efficienti e complessi, che sfruttano la rete per la parallelizzazione dei calcoli.
Il nostro sistema, in una versione più semplice, permette a una rete di nodi di condividere job e farli eseguire anche contemporaneamente da altri nodi della rete, in una situazione di trasparenza di accesso e posizione.
Per la sua realizzazione è stato utilizzato Erlang, un linguaggio funzionale che permette la comunicazione distribuita, e la DHT Riak, progettata per sistemi peer-to-peer.
Questa relazione illustra il funzionamento del sistema da noi implementato, ed è strutturata come segue:

- il primo capitolo analizza il sistema da implementare, illustrandone le specifiche, i requisiti, e le prime scelte effettuate;
- il secondo spiega come installare le varie architetture necessarie al funzionamento del sistema;
- il terzo illustra come avviene la connessione alla DHT;
- il quarto spiega l'implementazione della struttura dati su cui si basa il sistema;
- il quinto specifica le operazioni possibili sulla DHT;
- il sesto analizza le operazioni di richiesta ed esecuzione di un job;

- il settimo riporta i test effettuati.

Indice

Elenco delle figure

Capitolo 1

Analisi del sistema

Questo capitolo illustra le specifiche del sistema, i requisiti da soddisfare e le prime scelte progettuali effettuate.

1.1 Specifiche del sistema

Le specifiche di progetto richiedevano di implementare un sistema peer-to-peer in Erlang per condividere job, con la possibilità di avvalersi di una DHT distribuita open source già implementata.
In questo sistema, i nodi possono:

- inserire nella DHT i job che desiderano siano eseguiti da altri nodi;
- eseguire job, prelevati dalla DHT, per conto di altri nodi.

A partire da questa consegna abbiamo definito meglio le specifiche del progetto da realizzare.

1.2 Requisiti

I requisiti che il sistema deve soddisfare sono:

- la scalabilità, per non perdere efficienza con l'aumentare dei nodi;
- la *fairness*, per evitare che un nodo faccia eseguire solo i suoi job, o che uno stesso nodo li esegua tutti;
- tolleranza ai guasti, per cui la caduta di un nodo non deve compromettere il funzionamento del sistema;

- la trasparenza, per mascherare la collocazione fisica delle risorse.

Non è stata richiesta alcuna forma di sicurezza, quindi non è stato implementato alcun meccanismo di protezione o accesso controllato.

1.3 Scelta della DHT

La scelta di una DHT che ci permettesse di soddisfare tali requisiti, dopo una valutazione delle DHT più conosciute, è ricaduta su Riak.
Il sistema Riak prevede l'esistenza di un cluster di nodi in costante comunicazione tra loro. Le informazioni sono distribuite in maniera uniforme. Il meccanismo di repliche permette la tolleranza ai guasti, salvaguardando dalla perdita di dati in caso di caduta di un nodo Riak.
L'uguaglianza tra i nodi permette un'alta scalabilità. La quantità di dati è ridistribuita automaticamente ogni volta che un nodo si unisce o esce dal cluster. Quindi più aumentano i nodi, più il sistema diventa efficiente.
Inoltre Riak mette a disposizione una libreria per la creazione di un client Erlang per comunicare con i nodi Riak.
Lasciati questi aspetti al funzionamento della DHT, ci siamo potute concentrare sulle funzionalità da implementare.

1.4 Struttura del sistema

Abbiamo progettato il nostro sistema come una rete di nodi Riak, a cui i client Erlang possono collegarsi per sottoporre e/o eseguire job. Una macchina fisica potrebbe ospitare più nodi Riak, ma ciò è stato sconsigliato dagli sviluppatori, quindi per ogni macchina fisica ci sarà al più un nodo Riak, ed un numero arbitrario di client che vogliono condividere job. Questi client si connettono come nodi Erlang al nodo Riak conosciuto. Li chiamiamo client perché usufruiscono dei servizi del nodo Riak.

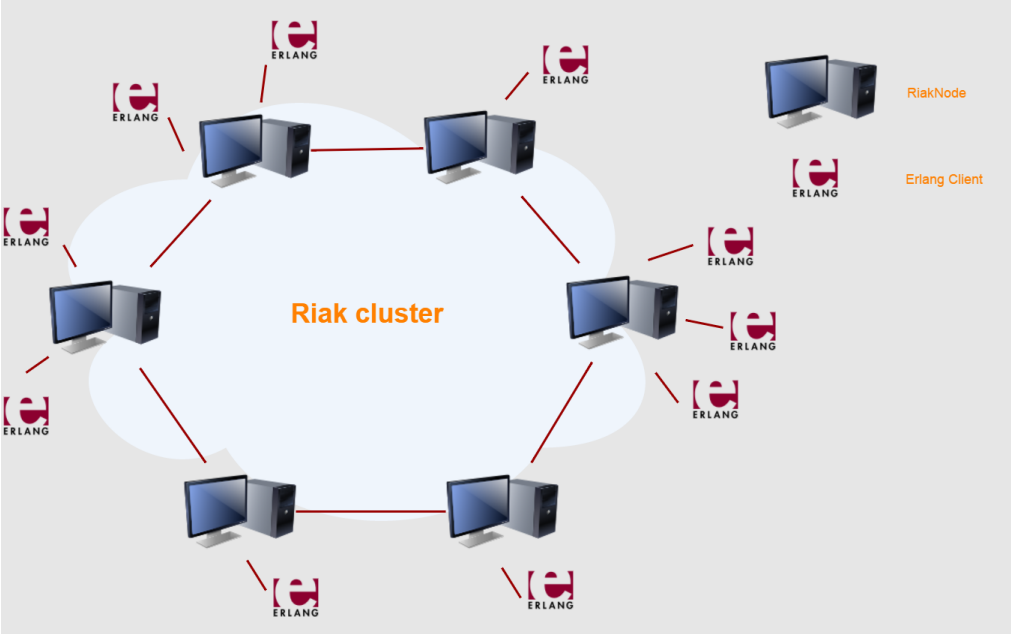

Figura 1.1: Diagramma di rete raffigurante la struttura del sistema: un cluster Riak con un nodo per ogni computer e uno o più nodi Erlang collegati ad ogni nodo Riak.

1.5 Stati del job

Per identificare lo stato corrente del job, sono stati seguiti gli stati dei processi, aggiungendo uno stato ulteriore per indicare i job da eseguire con priorità alta. Quindi un job appena inserito nella DHT di norma ha stato *ready*, se viene richiesto invece viene impostato a *important*. Durante l'esecuzione passa a *running*, e quando termina il suo stato passa a *terminated*. Se l'esecuzione non termina per motivi considerati nel capitolo 6, il processo torna automaticamente a *ready* o *important*, a seconda di com'era inizialmente. Inoltre, se il nodo che ha inserito il job cade prima che il job diventi *running*, lo stato diventa *nodedown*. Nella sezione 6.3 saranno spiegate le dinamiche di questo passaggio.

1.6 Modello adottato

L'implementazione del sistema inoltre prevedeva la scelta tra:

- il modello *push*, in cui vengono assegnati automaticamente i job da eseguire ai nodi che partecipano alla comunicazione;
- il modello *pull*, in cui i nodi sono liberi di scegliere se candidarsi o meno per eseguire i job.

Abbiamo scelto il secondo modello perché ci è sembrato più adatto far sì che i nodi fossero consapevoli di mettere a disposizione le loro risorse computazionali. Questo modello si adatta bene ad un eventuale sviluppo futuro, in cui i nodi potrebbero ricevere una ricompensa per il lavoro eseguito.
Più nodi possono candidarsi per eseguire uno stesso job, ma solo uno di essi lo eseguirà. La scelta del nodo a cui assegnare l'esecuzione del job è affidata al nodo che lo sottoscrive, in base alle prestazioni dei candidati. In questo modo un nodo con elevate prestazioni ha più possibilità di vedersi assegnato un job. Allo stesso tempo non sarà sempre lui ad eseguirli, perché una volta impegnato, a lungo andare, non sarà più lui il nodo più performante.
Quindi non ci sono meccanismi di elezione né supervisori esterni. Questo è possibile se i nodi possono comunicare tra loro, aspetto che rende necessaria un'attenta gestione delle cadute dei nodi. Questo aspetto sarà approfondito nel capitolo 6.

Capitolo 2

Installazione

In questo capitolo verranno illustrati i passaggi da seguire per installare Erlang, Riak, e il client Erlang per Riak. Verrà inoltre spiegato come configurare Riak, illustrati alcuni dei suoi comandi più utili, e cosa fare per poter utilizzare il nostro sistema.

2.1 Installazione Erlang

La versione di Erlang utilizzata è la 17.5.3, scaricata dal sito:
`https://www.erlang-solutions.com/downloads/download-erlang-otp`
Riak non è configurato per funzionare con versioni superiori a questa.

2.2 Installazione Riak

La versione di Riak utilizzata è la 2.1.1, per Ubuntu 14.04 Trusty Tahr. Da terminale digitare le seguenti istruzioni:

```
sudo apt-get install build-essential libc6-dev-i386 git

wget http://s3.amazonaws.com/downloads.basho.com/riak/2.1/2.1.1/riak-2.1.1.tar.gz

tar zxvf riak-2.1.1.tar.gz
cd riak-2.1.1
make rel
```

Se manca la libreria pam, per installarla:

```
sudo apt-get install libpam0g-dev
```

Se manca il file rebar, per installarlo:

```
sudo apt-get install rebar
```

Da terminale installare anche:

```
curl -s https://packagecloud.io/install/repositories/basho/riak/script.deb.sh | sudo bash
```

```
sudo apt-get install riak=2.1.1-1
```

2.3 Configurazione Riak

A questo punto, verificare di avere nella cartella `etc/riak` il file `riak.conf`, e modificarlo nel seguente modo:

- per indicare il nome del nodo riak:

  ```
  nodename = riak@indirizzo_IP del proprio PC
  ```

- per utilizzare l'interfaccia `Protocol Buffer`:

  ```
  listener.http.internal = 0.0.0.0:8098
  listener.protobuf.internal = 0.0.0.0:8087
  ```

- per abilitare l'interfaccia grafica Riak Control, visualizzabile alla pagina `http://127.0.0.1:8098/admin#`:

  ```
  riak_control = on
  ```

- per abilitare la ricerca tramite indici secondari:

  ```
  storage_backend = leveldb
  ```

Riavviare il PC per rendere effettive le modifiche.

2.4 Comandi Riak

Aprire un terminale e digitare:

- per avviare Riak:

  ```
  sudo riak start
  ```

- per chiudere Riak:

  ```
  sudo riak stop
  ```

- per effettuare il *join* di un nodo Riak ad un cluster di cui si conosce il nome di un nodo che ne fa parte:

  ```
  sudo riak-admin cluster join <nome_nodo_riak>
  sudo riak-admin cluster plan
  sudo riak-admin cluster commit
  ```

Nel caso fosse necessario eliminare il ring del cluster Riak:

```
sudo -rm -rl /var/lib/riak/ring/
```

2.5 Riak Control

Riak Control è un'interfaccia web, disponibile all'indirizzo `http://127.0.0.1:8098/admin#/cluster` utilizzabile per visualizzare e manipolare il cluster di Riak. La pagina di spiegazione del contenuto è `http://docs.basho.com/riak/latest/ops/advanced/riak-control/`.

Il nostro sistema notifica l'utente quando può essere utile controllare lo stato del cluster in caso di mancata disponibilità delle risorse. La situazione specifica è spiegata nella sezione 6.1.

2.6 Installazione client Erlang

Il client Erlang utilizzato è stato sviluppato dagli autori di Riak. Per installarlo:

```
git clone git://github.com/basho/riak-erlang-client.git
cd riak-erlang-client
make
```

2.7 Utilizzo del sistema

Dalla cartella del nostro sistema, eseguire il `Makefile` da terminale con il comando `make`. Verranno creati i codici oggetto dei moduli presenti. Per comunicare con Riak attraverso un nodo Erlang, bisogna far partire la console dalla cartella `riak-2.1.1/rel/riak/` includendo i path necessari al codice del client Riak:

```
erts-<vsn>/bin/erl -name nomenodolocale -setcookie riak
-pa $PATH_TO_RIAKC/ebin $PATH_TO_RIAKC/deps/*/ebin
$PATH_TO_RIAK/deps/*/ebin $PATH_TO_CODE
```

`$PATH_TO_RIAK` = cartella di installazione di Riak, che dovrebbe corrispondere a `$HOME/riak-2.1.1`
`$PATH_TO_RIAKC` = cartella di installazione del client Erlang di Riak, che dovrebbe corrispondere a `$HOME/riak-2.1.1/riak-erlang-client/`
`$PATH_TO_CODE` = cartella contenente i codici oggetto del nostro sistema

ll nome del nodo Erlang deve essere `nome@indirizzo_IP` del proprio PC. È necessario scrivere il nome in questo modo per il corretto funzionamento di Riak.
Quindi, se l'indirizzo della vostra macchina nella rete locale è per esempio 192.168.192.104, il nome del nodo Riak sarà `riak@192.168.192.104` mentre il nome del nodo Erlang sarà `nome@192.168.192.104`.
Per testare il nostro sistema vedere il capitolo 7.

Capitolo 3

Connessione a Riak

In questo capitolo verrà illustrato come avviene la connessione di un nodo Erlang ad un nodo Riak e come un nodo Erlang si mette in ascolto di eventuali messaggi da parte degli altri nodi Erlang.

3.1 Connessione di un nodo Erlang a un nodo Riak

Un nodo Erlang che vuole sottoporre un job o eseguirne uno deve mettersi in comunicazione con la DHT, ricevere messaggi da parte degli altri nodi Erlang e agire di conseguenza. Queste azioni sono permesse dalla funzione `Pid = connect:start(RiakNode)`, in cui `RiakNode` è il nome di un nodo Riak conosciuto. Vediamo di seguito i dettagli di questa funzione:

- la funzione `Pid = pid_link(RiakNode)`, necessaria per connettersi alla DHT, richiama l'istruzione `riakc_pb_socket:start_link(Address, Port)`, messa a disposizione dalla libreria di Riak per il client Erlang. L'address è ricavato dal RiakNode, mentre la porta è sempre 8087;
- `gen_server:start_link(local, ?SERVER, ?MODULE, Pid, [])` permette al nodo Erlang di mettersi in ascolto di eventuali messaggi da parte di altri nodi Erlang. Come si può vedere è stato utilizzato il behaviour gen_server, nel capitolo X vedremo la gestione delle chiamate;
- `connect:start_os_mon()` inizializza il monitor del sistema operativo, necessario per calcolare successivamente la performance del nodo. Per ulteriori informazioni vedere la sezione 6.2;
- `net_adm:ping(RiakNode)` invia un ping al nodo Riak. In questo modo si viene a conoscenza di tutti i nodi presenti nella rete;

- `dht:check_my_jobs(Pid)` cerca tra i job del nodo quelli che hanno come stato nodedown, e li aggiorna allo stato ready. Per la spiegazione dettagliata, vedere la sezione 6.3;
- vengono stampate alcune istruzioni di utilizzo;
- infine viene restituito il Pid, che costituirà lo stato interno del gen_server.

Nel caso in cui il RiakNode cadesse, con il comando `nodes()` è possibile vedere tutti i nodi, tra cui altri nodi Riak a cui collegarsi. In questo caso dovrà anche essere ricreato il processo gen_server, quindi bisogna eseguire la funzione `Pid2 = connect:start(RiakNode2)`. Suggeriamo perciò di dare ai nodi Riak nomi che inizino per "riak@...", in modo che siano riconoscibili in tutta la rete.

Il Pid (Process Identifier) del processo comunicante con il server Riak è utilizzato sempre come argomento in tutte le funzioni che interagiscono con la DHT. è quindi necessario salvarlo inizialmente in una variabile con
`Pid = connect:start(RiakNode)`. Se si dimentica questo passaggio, è possibile salvarla in un secondo momento con l'istruzione
`Pid = connect:pid_link(RiakNode)`. Nel caso in cui si presentasse un'eccezione, è necessario rinizializzare tutto e memorizzare il Pid in una nuova variabile con `Pid2 = connect:start(RiakNode)`. Queste istruzioni vengono visualizzate nel momento in cui avviene la prima connessione.

Capitolo 4

Implementazione della struttura basata su Riak

In questo capitolo verranno introdotti i concetti di oggetto, bucket e secondary index in Riak. Verranno elencati i tipi di oggetti salvati nei bucket, i bucket creati e spiegato il loro scopo. Infine verrà illustrato come sono stati utilizzati i secondary index.

4.1 Oggetti di Riak

Gli oggetti utilizzati dalla DHT sono strutture dati del tipo riakc_obj, della forma bucket/chiave/valore. Quindi per ogni elemento da inserire nei vari bucket bisogna definire la chiave che, utilizzando `riak_obj:new(Bucket, Key)`, deve essere binaria, identificare univocamente l'oggetto, ed il valore. Le operazioni di Riak permettono, conoscendo il bucket e la chiave, di ricavare il valore dell'oggetto.

4.2 Bucket: definizione

Il bucket è una struttura che permette di organizzare gli oggetti Riak in base al loro tipo, in modo simile alle tabelle dei database relazionali.
è utile associare ad ogni bucket un diverso tipo di oggetto, in modo da semplificare la ricerca degli oggetti.
Due oggetti in bucket diversi possono avere la stessa chiave e diverso valore.

4.3 Bucket creati

Nel nostro sistema abbiamo creato quattro bucket:

- Jobs, per memorizzare i job da eseguire;
- Modules, per memorizzare i codici oggetto delle funzioni richieste dai job;
- Candidates, per gestire la lista dei candidati per l'esecuzione dei job;
- MyJobs, per memorizzare i job inseriti da ogni singolo nodo.

Nei bucket sono stati inseriti i dati sotto forma di riak_obj, di cui abbiamo definito la chiave e il valore a partire da record da noi definiti. La chiave corrisponde al primo valore del record, il valore all'intero record (convertito in binario da `riakc_obj:new(Bucket, Key, Value)`).

4.4 Jobs

Il bucket Jobs contiene tutti i job inseriti da tutti i nodi.

- La chiave degli oggetti è il timestamp. è stata scelta questa chiave per poter poi ricercare più facilmente i job in base al momento dell'inserimento.
- Il valore degli oggetti contiene il record job, che presenta i seguenti campi:
  - il timestamp calcolato al momento dell'inserimento del job;
  - il nome del nodo che ha inserito il job;
  - il nome del modulo in cui si trova la funzione;
  - il nome della funzione;
  - la lista degli argomenti della funzione;
  - il risultato della funzione.

4.5 Modules

Il bucket Modules contiene tutti i file binari relativi ai job da eseguire ed eseguiti.

- La chiave degli oggetti è il nome del modulo. è stata scelta questa chiave per poter avere nomi di moduli univoci, in modo da poter verificare se il modulo che si vuole inserire sia già presente nella DHT, ed evitare scritture inutili.

- Il valore degli oggetti contiene il record code, che presenta i seguenti campi:
  - nome del modulo;
  - codice oggetto del modulo.

4.6 Candidates

Il bucket Candidates contiene le liste dei candidati di ogni job.

- la chiave degli oggetti è il timestamp del job. In questo modo è possibile recuperare facilmente i candidati relativi ad ogni job salvato nel bucket Jobs.
- Il valore degli oggetti contiene il record candidates, che presenta i seguenti campi:
  - timestamp del job
  - lista dei candidati con la relativa performance, strutturata nel seguente modo: [[nodo_1, performance_1],...,[nodo_n, performance_n]]

4.7 MyJobs

Il bucket MyJobs contiene le liste dei job inseriti da ogni nodo.

- la chiave degli oggetti è il nome del nodo che inserisce i job. In questo modo ogni nodo può risalire ai job che ha inserito.
- il valore degli oggetti contiene il record myJobs, che presenta i seguenti campi:
  - nome del nodo che sottopone i job,
  - lista dei timestamp relativi ai job salvati nel bucket Jobs da quel nodo.

4.8 Secondary index

I secondary index sono metadati nella forma chiave/valore di tipo binario, stringa o intero, che permettono di taggare oggetti appartenenti ad un bucket. Gli indici vengono inseriti al momento della scrittura dell'oggetto e l'operazione è atomica. Quando un oggetto è eliminato dal bucket, anche gli indici a lui associati vengono eliminati. Un oggetto in un bucket può avere associati uno più secondary index.
Con una secondary index query è possibile recuperare tutti gli oggetti che hanno

lo stesso indice secondario all'interno di un bucket specifico. Il risultato di una secondary index query è una lista di chiavi corrispondenti agli oggetti del bucket, il cui indice secondario corrisponde a quello cercato.

I secondary index sono utilizzati per salvare gli stati dei job:

- la chiave è "status"
- il valore, salvato in binario, corrisponde ad ognuno dei possibili stati, ossia <<"ready">>, <<"important">>, <<"running">>, <<"terminated">>, <<"nodedown">>.

Capitolo 5

Operazioni sulla DHT

In questo capitolo verranno illustrate le operazioni possibili sulla DHT: scrittura, lettura, ricerca, aggiornamento ed eliminazione. Esse sono raccolte nella pagina dht.html fornita in allegato. Per implementare queste operazioni abbiamo utilizzato le funzioni disponibili per il client Erlang di Riak, consultabili all'indirizzo `http://basho.github.io/riak-erlang-client/`.

5.1 Scrittura

Quando un nodo vuole sottoporre un job, viene inserito un elemento nel bucket Jobs con le informazioni necessarie per l'esecuzione, aggiunto il codice binario della funzione da eseguire nel bucket Modules, inizializzata a vuota la lista dei candidati nel bucket Candidates, e aggiunto il job alla lista dei job del nodo nel bucket MyJobs. L'inserimento di un job può essere fatto senza restrizioni di tempo o frequenza. Vediamo ora nel dettaglio come avvengono queste operazioni.

La scrittura sulla DHT si presenta quando un nodo vuole inserire un job da eseguire. Questa operazione può avvenire attraverso le funzioni:

- `dht:add_job(Pid, Module, Function, Args)` se si vuole inserire un job con priorità normale. Verrà poi chiamata la funzione `dht:add_job(Pid, Module, Function, Args, <<"ready">>)`;
- `dht:add_important_job(Pid, Module, Function, Args)` se si vuole inserire un job con priorità alta. Verrà poi chiamata la funzione `dht:add_job(Pid, Module, Function, Args, <<"important">>)`.

In entrambi i casi, gli argomenti sono i seguenti:

- Pid: Process Identifier del processo comunicante con il nodo Riak, definito al momento della connessione, come visto nel capitolo 3;

- Module: il nome del modulo in cui si trova la funzione del job da eseguire. Il modulo non deve importare altri moduli definiti dall'utente, altrimenti potrebbe insorgere un'eccezione se questi non sono presenti sul nodo che eseguirà il job;
- Function: il nome della funzione da eseguire;
- Args: la lista degli argomenti da passare alla funzione del job. Devono essere del numero e del tipo richiesto dalla funzione, o causeranno l'insorgere di un'eccezione durante l'esecuzione del job.

Nella sezione 6.4 verrà spiegato come sono state gestite queste eccezioni.
La funzione `dht:add_job(Pid, Module, Function, Args, Priority)` esegue le seguenti operazioni:

- crea la chiave dell'oggetto da inserire nel bucket Jobs a partire dal timestamp della chiamata di funzione;
- crea il record che costituirà il valore dell'oggetto da inserire nel bucket Jobs, con timestamp, nome del nodo, modulo, funzione, argomenti, e risultato inizializzato con l'atomo noresult;
- prova ad aggiungere il modulo nel bucket Modules con la funzione `dht:add_module(Pid, Module)`. Se quest'operazione va a buon fine, si prosegue nell'inserimento del job, altrimenti non viene inserito nulla nella DHT, avvisando l'utente del tipo di errore in base al risultato di `dht:add_module(Pid, Module)`;
- viene chiamata la funzione `dht:save_job(Pid, Job, Priority)` che crea un riak_obj a partire dalla chiave e dal valore del job, lo inserisce nel bucket Jobs, e imposta, tramite la funzione `dht:set_secondary_index(Pid, KeyJob, Status)`, l'indice secondario in base alla priorità indicata;
- viene chiamata la funzione `dht:empty_candidates(Pid, KeyJob)`, che crea un riak_obj, da inserire nel bucket Candidates, la cui chiave è la chiave del job appena inserito (timestamp), ed il valore una lista vuota;
- viene chiamata la funzione `dht:add_my_job(Pid, KeyJob)` che aggiorna l'oggetto presente nel bucket MyJobs, avente come chiave il nome del nodo che sta inserendo il job, e ne modifica il valore aggiungendo alla lista di job la chiave del job appena inserito. Se questo oggetto non era già presente, viene inizializzato con la lista vuota (vedi funzione `dht:get_my_jobs(Pid)` nella sezione5.2);

- Infine viene stampato un messaggio a video per notificare l'utente della chiave assegnata al job inserito. L'operazione di inserimento è reversibile in qualsiasi momento, eccetto durante l'esecuzione del job, richiamando la funzione `dht:delete_my_job(Pid, KeyJob)`.

Vediamo ora il funzionamento di `dht:add_module(Pid, Module)`, quali errori può causare e come essi vengono gestiti da `dht:add_job/5`:

- per prima cosa si cerca di prelevare il codice oggetto relativo al modulo con `code:get_object_code(Module)`. Se questa operazione dà errore, `dht:add_job/5` avvisa l'utente che il job non è stato inserito perché il codice oggetto del modulo non è stato trovato nel path;

- se trova il codice oggetto, chiama la funzione `dht:verify_module(Pid, Module, Code)`, che verifica se il modulo sia già stato inserito nel bucket Modules;

- se nel bucket Modules non esiste già un modulo con lo stesso nome, `dht:verify_module/3` crea il record che costituirà l'oggetto da inserire, con nome del modulo e binario del codice oggetto, quindi lo inserisce nel bucket Modules, chiamando `dht:save_module(Pid, Module)`;

- se invece nel bucket Modules esiste già un oggetto con la stessa chiave (il nome del modulo), allora viene confrontato il binario del codice oggetto del modulo con quello presente nella DHT. Nel caso siano uguali, il modulo non viene reinserito, e `dht:add_job/5` può proseguire nell'inserimento del job. Nel caso in cui siano diversi, `dht:add_job/5` non inserirà nulla, e avviserà l'utente di cambiare il nome del modulo, perché è già presente un codice diverso con lo stesso nome.

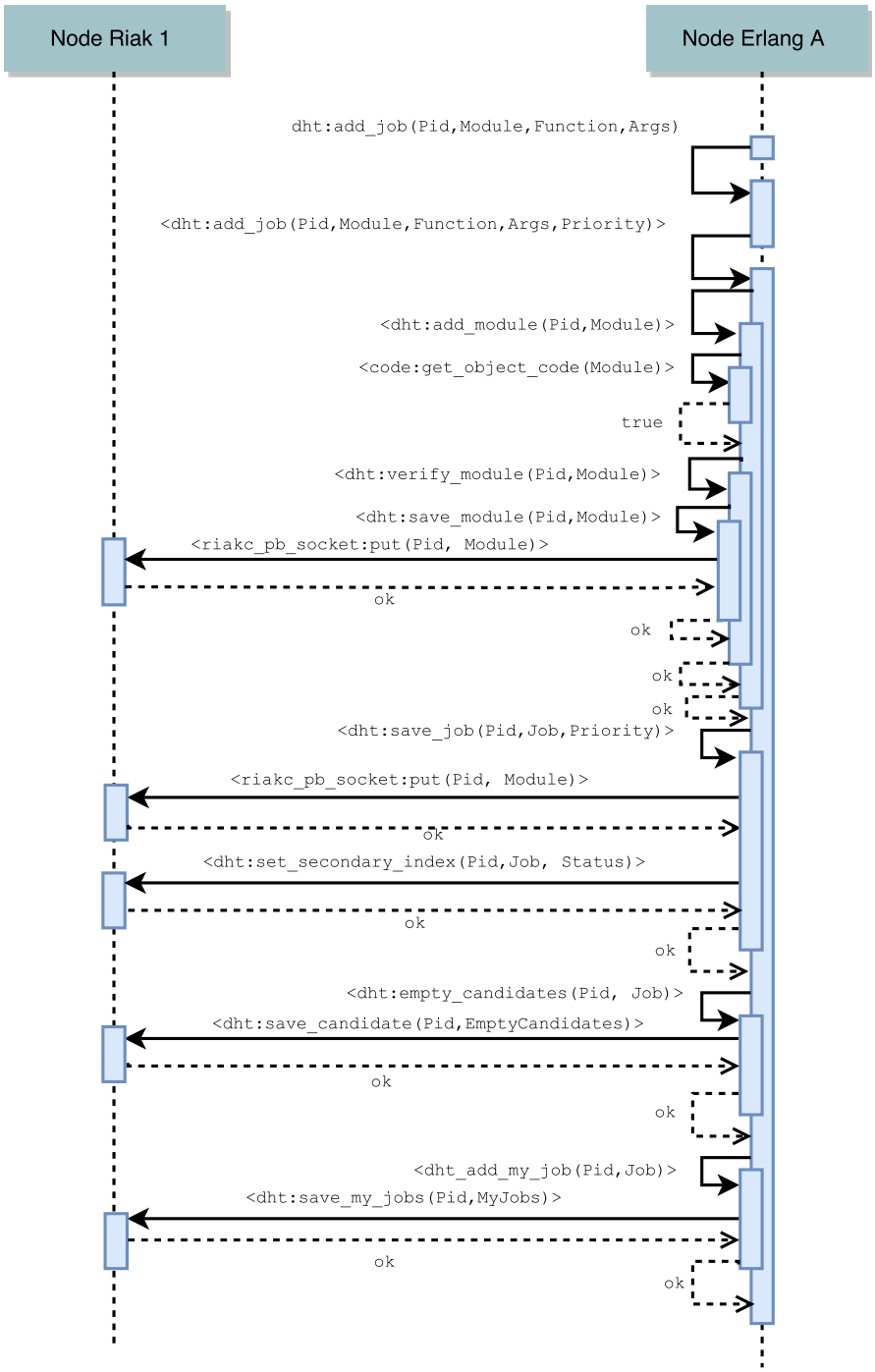

Figura 5.1: Diagramma di sequenza raffigurante l'operazione `dht:add_job(Pid,Module,Function,Args)`. Non sono raffigurate in tutti i loro dettagli le operazioni `set_secondary_index/3`, `save_candidate/2`, e `save_my_jobs/2`.

5.2 Lettura

Tutte le funzioni di lettura prevedono il ritorno del valore dell'oggetto, o di un determinato campo, dato il Pid e la chiave dell'elemento. Vediamole nel dettaglio:

- `dht:get_status(Pid, KeyJob)` ritorna l'indice secondario del job, corrispondente al suo stato corrente;
- `dht:get_job(Pid, KeyJob)` ritorna il valore del job che ha come chiave KeyJob, memorizzato nel bucket Jobs;
- `dht:get_job_result(Pid, KeyJob)` ritorna il risultato del job richiesto, ossia il campo result dell'oggetto che ha come chiave KeyJob nel bucket Jobs;
- `dht:get_job_node(Pid, KeyJob)` ritorna il nome del nodo che ha inserito il job, ossia il campo node dell'oggetto che ha come chiave KeyJob nel bucket Jobs;
- `dht:get_beam(Pid, ModuleName)` ritorna il modulo richiesto dall'atomo ModuleName, ossia il valore dell'oggetto che ha come chiave `atom_to_binary(ModuleName,latin1)` nel bucket Modules;
- `dht:get_candidates(Pid, KeyJob)` ritorna i candidati relativi al job indicato, ossia il valore dell'oggetto che ha come chiave KeyJob nel bucket Candidates;
- `dht:get_my_jobs(Pid)` ritorna il valore dell'oggetto presente nel bucket MyJobs, che ha come chiave il nodo che chiama questa funzione (ricavato con la funzione `node()`). Se quest'oggetto non esiste, viene creato automaticamente con la funzione `dht:empty_my_jobs(Pid)`, che crea la chiave a partire dal nodo che chiama la funzione, e il valore a partire da un record che ha come campi il nodo e una lista vuota, la quale rappresenta i job inseriti (nessuno). A questo punto `dht:empty_my_jobs(Pid)` chiama la funzione `dht:save_my_jobs(Pid, JobList)` che crea il riak_obj e lo inserisce nel bucket MyJobs;
- `dht:list_my_jobs(Pid)` ritorna la lista dei job inseriti dal nodo chiamante, ossia il campo jobs dell'oggetto nel bucket MyJobs, che ha come chiave il nome del nodo che chiama la funzione.

5.3 Ricerca

La ricerca fornisce ai nodi che vogliono eseguire un job quello con stato important o, in mancanza, ready presente da più tempo nella DHT.
La funzione che permette la ricerca di questo job è `dht:search_oldest_job(Pid)`. Vediamone l'implementazione nel dettaglio:

- per prima cosa viene verificata la presenza di job con stato important, chiamando la funzione `dht:search_jobs(Pid, <<"important">>)`. Questa funzione torna la lista delle chiavi dei job che hanno <<"important">> come valore del secondary index. Se questa lista non è vuota, ne viene scelto l'elemento più vecchio, ossia con la chiave, che è il timestamp, minima, e quindi viene restituita una lista contenente la chiave del job e il valore del suo secondary index;

- se `dht:search_jobs(Pid, <<"important">>)` ritorna una lista vuota, viene chiamato `dht:search_jobs(Pid, <<"ready">>)`, che quindi ritorna la lista delle chiavi dei job con secondary index uguale a <<"ready">>. Se questa lista non è vuota, viene scelto il job più vecchio, e restituita la lista contenente la chiave del job trovato e il valore del suo secondary index;

- se non esistono nemmeno job con stato ready viene ritornato come output [notfound, notfound].

5.4 Aggiornamento

Vediamo ora che tipo di aggiornamenti sono possibili e come vengano effettuati, senza specificare quando essi avvengano. Questo argomento sarà invece trattato nel capitolo successivo.

- `dht:add_candidate(Pid, KeyJob, Performance)` aggiunge un candidato alla lista dei candidati per l'esecuzione del job indicato da KeyJob. La lista già presente viene recuperata da `dht:get_candidates(Pid, KeyJob)`, e modificata aggiungendo in coda un valore nel seguente modo: $[[\text{node}_1, \text{performance}_1] \ldots [\text{node}_n, \text{performance}_n]]$, in cui node_n è il nome del nodo che chiama la funzione (ottenuto con `node()`), mentre performance_n è il valore della sua performance;

- `dht:add_my_job(Pid, KeyJob)` aggiunge KeyJob, che indica il job appena inserito, al campo jobs dell'oggetto nel bucket MyJobs avente come chiave il nodo (ottenuto con `node()`) che chiama la funzione. In questo modo viene aggiornata la lista dei job inseriti in DHT da quel nodo.

L'indice secondario può essere modificato a seconda dei casi trattati nel capitolo X, tramite le seguenti funzioni:

- `dht:return_ready(Pid, KeyJob, Priority)` riporta l'indice secondario allo stato indicato da Priority, <<"ready">> o <<"important">>. Inoltre rinizializza la lista dei candidati chiamando la funzione `dht:empty_candidates/2`;
- `dht:running_job(Pid,KeyJob)` aggiorna l'indice secondario del job a <<"running">>;
- `dht:nodedown_job(Pid,KeyJob)` aggiorna l'indice secondario del job a <<"nodedown">>;
- `dht:terminated_job(Pid, KeyJob, Result)` aggiorna l'indice secondario del job a <<"terminated">> , scrive il risultato Result nel campo result dell'oggetto KeyJob presente nel bucket Jobs e svuota la lista dei candidati di quel job, presente come valore nel bucket Candidates, chiamando la funzione `dht:empty_candidates(Pid, KeyJob)`;
- `dht:check_my_jobs(Pid)` recupera la lista dei job inseriti dal nodo chiamante (con `node()`), seleziona tra questi i job che hanno il secondary index <<"nodedown">>, e per ognuno aggiorna il secondary index a <<"ready">> (da notare che il timestamp rimane quello iniziale) e rinizializza la lista dei candidati.

5.5 Cancellazione

Abbiamo permesso ai nodi di poter cancellare liberamente solo i job da loro inseriti, per garantire una forma di protezione delle informazioni inserite nella DHT.
L'eliminazione di un job comporta la cancellazione di un elemento dal bucket Jobs, dal bucket Candidates, e l'aggiornamento di un elemento del bucket MyJobs.
Vediamole ora come funzionano nel dettaglio:

- `dht:delete_my_job(Pid, KeyJob)` dato il Pid e la chiave del job da cancellare,
  - controlla se il job è presente nel bucket Jobs, se non lo è avvisa l'utente che il job non esiste più;
  - controlla se chi lo ha inserito è lo stesso nodo che sta chiamando una funzione. Se non è così, stampa un messaggio per avvisare l'utente

che non ha il permesso di cancellare quel job, perché appartiene ad un altro nodo;

- chiama la funzione `dht:delete_my_safe_job(Pid, KeyJob)`, che controlla lo stato del job, se non ha stato running prosegue, altrimenti stampa il messaggio di errore per avvisare che il job è in esecuzione da un altro nodo;
- chiama la funzione `dht:delete_job(Pid, KeyJob)` che cancella il job dal bucket Jobs,
- chiama la funzione `dht:delete_job_candidates(Pid, KeyJob)` per cancellare l'elemento contenente la lista dei candidati di quel job,
- aggiorna l'elemento nel bucket MyJobs avente come chiave il nome del nodo, cancellando dalla lista di job il job corrispondente a KeyJob,
- quindi salva l'elemento con la nuova lista di job nel bucket MyJobs.

- `dht:delete_my_jobs(Pid)` cancella tutti i job non running del nodo, chiamando `dht:delete_my_safe_job/2` per ogni job presente nell'elemento del bucket MyJobs, corrispondente alla lista dei job inseriti da quel nodo;
- `dht:delete_my_terminated_jobs(Pid)` cancella tutti i job con stato terminated inseriti da quel nodo. Infatti chiama `dht:delete_job/2` solo sui job che hanno come secondary index <<"terminated">>, prendendo le chiavi dall'elemento del bucket MyJobs che ha come chiave il nodo che chiama la funzione.
- `dht:reset_all(Pid)` resetta tutti i bucket presente nella DHT, riportandola allo stato iniziale. è stata pensata solo per permettere il test del sistema.

Capitolo 6

Richiesta ed esecuzione del job

Questo capitolo spiega nel dettaglio come avviene la richiesta di un job da eseguire, la sua esecuzione, come viene gestito lo scambio di messaggi tra i nodi coinvolti, l'occorrenza di eccezioni ed errori ed eventuali cadute dei nodi.

6.1 Richiesta di un job da eseguire

Quando un nodo richiede di eseguire un job, viene effettuata la ricerca del job più vecchio, una volta trovato, inizia la comunicazione tra questo nodo ed il nodo che aveva inserito il job. Vediamo nel dettaglio cosa succede.

- un nodo chiede di poter eseguire un job con la funzione `trade:ask_for_job(Pid)`. Se non tutte le risorse sono disponibili a causa di troppi nodi Riak non raggiungibili, può insorgere un'eccezione, che viene segnalata all'utente, avvisandolo di controllare lo stato del cluster Riak alla pagina Riak Control (`http://127.0.0.1:8098/admin#/cluster`).
  Questa è l'unica funzione pubblica del modulo trade, tutte le altre sono chiamate automaticamente, e quindi non sono esportate.
  Questa funzione chiama `dht:search_oldest_job(Pid)`, vista nella sezione 5.3, che ritorna la chiave di un job da eseguire con il relativo stato, important o ready. Se ne trova uno viene fatta una spawn sulla funzione `trade:submit_for_job(Pid, KeyJob, Priority)`. Se non ci sono job disponibili per l'esecuzione, l'utente ne viene notificato.

- `trade:submit_for_job(Pid, KeyJob, Priority)` verifica che il job sia ancora presente nella DHT e, in caso affermativo, calcola la performance del nodo, aggiunge il nodo con la sua performance alla lista di candidati per l'esecuzione del job con `dht:add_candidate(Pid, KeyJob,`

`Performance)`, e infine invoca `trade:verify_chosen(Pid, To, KeyJob, Priority)`.

Prima di spiegare le successive operazioni, illustriamo come avviene il calcolo della performance.

6.2 Calcolo della performance

Erlang mette a disposizione diverse operazioni per misurare lo stato e le prestazioni del calcolatore. Per utilizzarle è necessario avviare le applicazioni SASL(System Architecture Support Libraries) e l'applicazione OS_Mon. La funzione `aux:start_os_mon()` si occupa di avviare le due applicazioni ed è chiamata appena il nodo si connette, nella funzione
`connect:start(RiakNode)`. Per ulteriori informazioni su queste applicazioni, consultare le relative pagine web (`http://www.erlang.org/doc/man/sasl_app.html`, `http://www.erlang.org/doc/man/os_mon_app.html`).
Per misurare le prestazioni del nodo abbiamo utilizzato la funzione Erlang `memsup:get_system_memory_data()`. La funzione invoca un memory check e ne ritorna il risultato dipendente dal sistema operativo sotto forma di una lista di tuple o una lista vuota se memsup non è disponibile o se il memory check va in timeout. Le grandezze di memoria riportate sono espresse in bytes. Nel sistema operativo linux, la memoria disponibile per l'emulatore è data dalla somma di:

- free_memory, memoria disponibile per l'allocazione per l'emulatore Erlang;
- cached_memory, memoria cache che il sistema usa per i file letti dal disco;
- buffered_memory, memoria utilizzata dal sistema per dati temporanei.

Quindi la funzione `aux:performance()`, chiamata dalla funzione
`trade:submit_for_job/3`, ritorna un intero corrispondente alla memoria disponibile per l'emulatore Erlang.

6.3 Scelta del candidato migliore

Per semplificare la spiegazione, d'ora in poi chiameremo nodo A il nodo che ha inserito nella DHT il job da eseguire, e nodo B il nodo che vuole eseguirlo.
La funzione `trade:verify_chosen/4` notifica il nodo A che ci sono dei candidati.

- Se nella lista dei candidati per il job c'è solo un elemento, vuol dire che il nodo B è il primo a candidarsi, e quindi notifica il nodo A con `trade:first_submit(To, KeyJob)`.
  Questa funzione esegue una rpc di `connect:first_apply(KeyJob)` sul nodo A, che, poiché è una chiamata al gen_server, viene gestita dalla `handle_call(first_apply, KeyJob, _From, Pid)`. Questa handle_call notifica il nodo A che c'è un primo candidato per eseguire il job. Da questo momento gli altri nodi hanno 10 secondi per candidarsi, al termine dei quali il nodo A sceglierà il nodo migliore a cui assegnare il job, e notificherà il nodo B del nodo scelto.

- Se invece c'è più di un candidato, il nodo A è notificato tramite `trade:submit(To, KeyJob)`, che esegue una rpc sul nodo A della funzione `connect:apply(KeyJob)`.
  Essa viene gestita dalla handle_call(apply, KeyJob, _From, Pid), che notifica il nodo A della presenza di un altro candidato e notifica il nodo B del nodo scelto.

- Per scegliere il nodo migliore, all'interno dell'handle_call viene chiamata la funzione `aux:choose_worker(Pid, KeyJob)`, che recupera i candidati del job e sceglie il worker attivo con la performance più alta, controllando che:

  - il job sia ancora presente nella DHT. Se non lo è non sceglie nessuno, e sia il nodo A che il nodo B sono avvisati del fatto che il job non è più presente.
  - il worker scelto sia ancora attivo. Se non lo è, controlla il secondo worker migliore, fino ad esaurire la lista. Se nessun worker è più attivo, il nodo A riceve la notifica che tutti i worker sono caduti.

- Se il nodo A cade prima di aver scelto il candidato migliore, il nodo B ne riceve la notifica, e cambia lo stato del job a nodedown. L'aggiornamento dello stato del job lo attuano tutti i candidati, perché se fosse incaricato di farlo solo il primo ma questo cadesse, il job rimarrebbe al suo stato iniziale. Quando il nodo A si riconnetterà al sistema con Pid = `connect:start(RiakNode)`, eseguirà `dht:check_my_jobs(Pid)` per aggiornare tutti i job con stato nodedown allo stato ready.

6.4 Esecuzione del job

Se la scelta del worker va a buon fine, il nodo B esegue la funzione `trade:request_job(Pid, To, KeyJob, Chosen, Priority)`, in cui To è il nodo A, KeyJob è la chiave del job, Chosen è il nodo scelto, Priority è lo stato iniziale.

- `trade:request_job/5` verifica che il nodo B sia il worker scelto dal nodo A.
- Se il nodo B è il worker scelto, verifica che il job sia ancora nel bucket, se c'è registra con un nome globale il pid corrente (`self()`), quindi prova ad eseguire il job con la funzione `trade:execute_job(Pid, KeyJob, RJob, Priority)` all'interno del costrutto try-catch.
- `trade:execute_job(Pid, KeyJob,RJob,Priority)`, in cui RJob è il valore del job con chiave KeyJob, esegue le seguenti operazioni:
  - per prima cosa aggiorna lo stato del job a running, in modo che non sia più cancellabile;
  - notifica con `trade:notify_running(Node, KeyJob, Priority, PidWorker)` il nodo A che lo sta eseguendo. Essa verrà spiegata nel dettaglio nella sezione sucessiva;
  - carica il codice oggetto con la funzione `aux:load_beam(Pid, Module)`;
  - esegue il job;
  - notifica il nodo A che il job è terminato, comunicandogliene il risultato con `trade:notify_terminated(To, KeyJob, Result)`, che sarà spiegata nella sezione successiva;
  - infine carica il risultato nel campo result del job nel bucket Jobs , ne aggiornalo stato a terminated e svuota la lista dei candidati, con la funzione `dht:terminated_job(Pid, KeyJob, Result)`.
- Se l'esecuzione del job causa un'eccezione, questa viene catturata dal catch, che notifica il nodo B che il job non è stato terminato, ed il nodo A che è avvenuta un'eccezione con la funzione `trade:deliver_exception(To, KeyJob, Priority)`, che sarà spiegata nella sezione successiva.
- In ogni caso, sia che l'esecuzione vada a buon fine o non avvenga nemmeno per eccezioni, cadute, interruzioni, o perché il nodo B non è il candidato

scelto, all'utente viene chiesto se desidera ricercare o no un altro job da eseguire, attraverso la funzione `aux:ask_again_for_job(Pid)`.

Vediamo ora nel dettaglio alcune funzioni utilizzate:

- `trade:notify_running(Node, KeyJob, Priority, PidWorker)` esegue una rpc sul nodo A della funzione `connect:running(KeyJob, From, Priority, PidWorker)`, in cui From è il nodo B, e PidWorker corrisponde al pid relativo alla `trade:submit_for_job/3`. Questa rpc viene gestita dalla corrispondente handle_call, che avvisa il nodo A che il job è in esecuzione, e gli attiva il processo di monitoraggio sul nodo B con la funzione `aux:monitor_worker(PidWorker, Pid, KeyJob, Priority)`;

- `aux:monitor_worker(PidWorker, Pid, KeyJob, Priority)` fa sì che se il nodo B cade prima di aver completato l'esecuzione del suo job, quindi lasciandone lo stato a running, il nodo A ne venga notificato, ed esegua la funzione `dht:return_ready(Pid, KeyJob, Priority)`;

- `trade:notify_terminated(To, KeyJob, Result)` esegue una rpc sul nodo A della funzione `connect:terminated(KeyJob, Result, From)`, gestita dalla relativa handle_call, che notifica il nodo A del risultato del job. Se il nodo A è caduto e non può ricevere questo messaggio, può recuperare il risultato in un secondo momento con `dht:get_job_result/2`;

- `trade:deliver_exception(To, KeyJob, Priority)` esegue una rpc sul nodo A della funzione `connect:exception(KeyJob, From, Priority)`, gestita dalla relativa handle_call, che notifica il nodo A che è avvenuta un'eccezione. Viene quindi interpellato l'utente del nodo A, facendogli scegliere se preferisce cancellare il job dalla DHT, o riportarlo allo stato iniziale (ready o important). Abbiamo voluto lasciare al nodo A la decisione di cancellare o meno il job dalla DHT, perché non si può sapere a priori il motivo dell'eccezione.

Abbiamo inoltre dato la possibilità al nodo A di interrompere in modo sicuro il processo sul nodo B che sta eseguendo il job. Questa operazione deve essere attuata esplicitamente dall'utente del nodo A, con la funzione `aux:interrupt_runner(Pid, KeyJob, Priority, Worker)`. Questa funzione gli viene suggerita con gli argomenti corretti dalla handle_call di `connect:running/4`. Il processo da interrompere era stato registrato con un nome globale prima dell'inizio dell'esecuzione del job. La funzione, se il processo è ancora registrato, e quindi il job è ancora con stato running, interrompe il processo forzatamente e riporta alla priorità iniziale il job, resettando anche la lista dei candidati, con la funzione `dht:return_ready/3`. La possibilità di interrompere il processo può essere utile per il nodo A qualora

decidesse che il nodo B ci sta mettendo troppo tempo ad eseguire il job, o che il job non può essere portato a termine e quindi è meglio eliminarlo, o qualsiasi altro motivo.
In seguito a questa interruzione forzata, al nodo B non sarà richiesto se desidera eseguire un altro job, dal momento che il processo di `submit_for_job/3` è stato interrotto. Potrà però eseguire senza problemi qualsiasi nuova operazione.
Mentre esegue il job, il nodo B è libero di fare qualsiasi altra operazione, compresa quella di interrompere il processo del job che sta eseguendo, dal momento che l'interruzione viene gestita dall'handle_call di
`connect:running/4`.

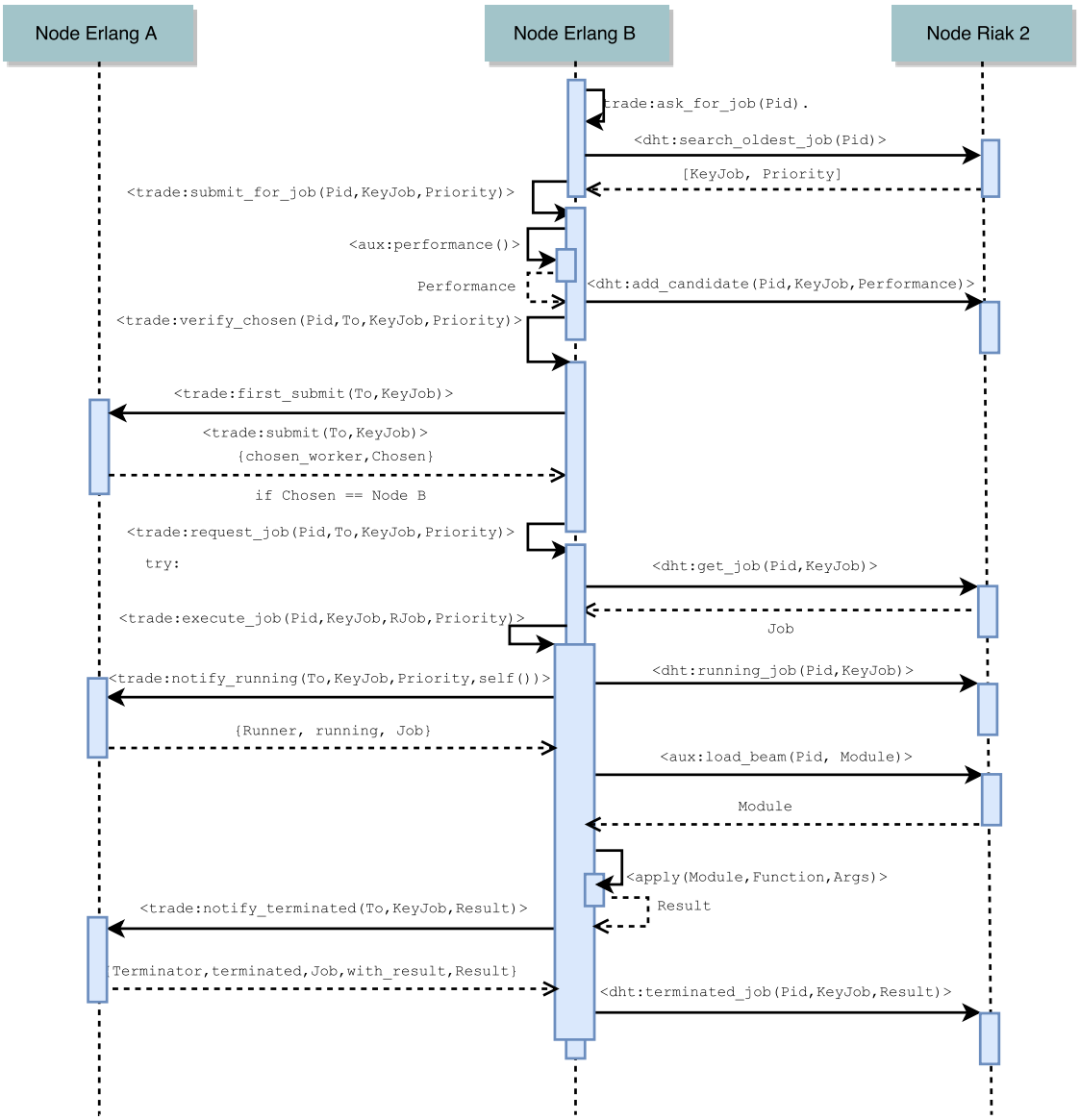

Figura 6.1: Diagramma di sequenza raffigurante l'operazione `dht:ask_for_job(Pid)`. Sono rappresentati gli scambi di messaggio tra i vari nodi nel caso di una richiesta di esecuzione del job conclusa senza problemi. Le operazioni non sono raffigurate nel dettaglio.

Capitolo 7

Test del sistema

In questo capitolo vengono spiegate le condizioni di sviluppo e test del sistema, e illustrate le funzioni messe a disposizione dell'utente per testarne alcune funzionalità.

7.1 Rete e nodi utilizzati

Il sistema è stato testato su una stessa rete locale, utilizzando fino a tre macchine fisiche, ognuna ospitante un nodo Riak e un numero variabile di nodi Erlang. Il sistema operativo utilizzato è Ubuntu, installato su partizione primaria e non con macchina virtuale.
Il numero di repliche è stato lasciato al valore di default (n_val=3). In queste condizioni si è notato che con un nodo down su tre, tutti i dati erano accessibili, mentre con due nodi down su tre alcune repliche non risultavano disponibili. In queste condizioni abbiamo riscontrato che la ricerca del job da eseguire portava all'insorgere di un eccezione. Abbiamo gestito questa situazione avvisando l'utente che i dati si trovano su nodi Riak non raggiungibili e di verificare la situazione del cluster alla pagina `http://127.0.0.1:8098/admin#`.
Con un cluster di due nodi Riak, i dati sono tutti disponibili.
Nella situazione in cui alcuni nodi sono down, ma i dati sono comunque disponibili, il sistema funziona correttamente, anche se le cancellazioni non avvengono sempre in modo completo. Nonostante questo, l'unico effetto non voluto riscontrato è che rimangono nella DHT dati che dovrebbero essere cancellati, ma questo non ha causato errori nell'utilizzo delle funzioni.

7.2 Funzioni di test

Per testare il sistema, abbiamo predisposto delle funzioni di test che eseguono una di seguito all'altra operazioni di scrittura e richiesta di eseguire un job.
Si possono trovare nei moduli testA e testB.
Per eseguirli bisogna installare Erlang, Riak, ed il client Erlang di Riak secondo le modalità spiegate nel capitolo 2.
Una volta verificata la corretta installazione, è possibile utilizzare il sistema seguendo le istruzioni della sezione 2.7.
Una volta aperte correttamente una o più shell di Erlang, sulla stessa o su macchine diverse, digitare l'istruzione `testA:main(RiakNode).` oppure `testB:main(RiakNode).`
RiakNode è l'atomo contenente il nome completo del nodo Riak a cui collegarsi (esempio: testA:main('riak@192.168.192.104').
Eseguendole in diverso ordine, sullo stesso o su diversi terminali, è possibile visualizzare diversi output, che rappresentano le diverse interazioni tra i vari nodi.

testA:main(RiakNode)
La funzione di test è la seguente:

```
main(RiakNode) ->
    Pid = connect:start(RiakNode),
    sleep(3000),
    io:format("Now it will be executed: ~n
    dht:add_job(Pid, test, make_prime, [30]).~n"),
    dht:add_job(Pid, test, make_prime, [30]),
    sleep(3000),
    io:format("Now it will be executed:~nask_for_job(Pid).~n"),
    ask_for_job(Pid).
```

Eseguendo solo questa funzione su un unico nodo, prima compaiono i progress report dell'applicazione SASL, poi il seguente output:

```
Welcome!
To add a job, type:
dht:add_job(Pid, Module, Function, Args).
or, if your job is important, type:
dht:add_important_job(Pid, Module, Function, Args).
To see if there are executable jobs, type:
trade:ask_for_job(Pid).
If you forgot to store your Pid, store it with:
Pid=connect:pid_link('riak@192.168.192.104').
If an exception occurs you'll have to restore your Pid in another variable with:
Pid2=connect:pid_link('riak@192.168.192.104').
If the known RiakNode crashes, with nodes() you can see other nodes.
Connect to a new RiakNode with:
Pid2=connect:start(NewRiakNode).
Now it will be executed:
dht:add_job(Pid, test, make_prime, [300]).
Job inserted in bucket <<"Jobs">> with key: <<"2015-10-30T22:58:34.423Z">>
```

```
Now it will be executed:
dht:ask_for_job(Pid).
Submit for job <<"2015-10-30T22:58:34.423Z">>
{<0.134.0>}
My performance is 4930740224
Nodes start applying for job <<"2015-10-30T22:58:34.423Z">>
Node chosen for job <<"2015-10-30T22:58:34.423Z">> is 'fra3@192.168.192.104'
The chosen node for job <<"2015-10-30T22:58:34.423Z">> is 'fra3@192.168.192.104'
Node 'fra3@192.168.192.104', running job: <<"2015-10-30T22:58:34.423Z">>
If you want to interrupt the job, type:
aux:interrupt_runner(Pid, <<"2015-10-30T22:58:34.423Z">>, <<"ready">>,
'fra3@192.168.192.104').
Loaded module test
If you want to interrupt the job, type:
exit(global:whereis_name('2015-10-30T22:58:34.423Z'),kill).
Generating a 30 digit prime
Terminated job test:make_prime[30] with result 776345026128328198297525968017
Job <<"2015-10-30T22:58:34.423Z">> terminated by 'fra3@192.168.192.104',
the result is: 776345026128328198297525968017
To delete it, type:
dht:delete_my_job(Pid,<<"2015-10-30T22:58:34.423Z">>).
To delete all your terminated jobs, type:
dht:delete_my_terminated_jobs(Pid).
Result sent
Do you want to look for other ready jobs? y/n>
```

Rispondendo "y" la risposta sarà:

```
There aren't any ready jobs!
```

Rispondendo "n" la risposta sarà:

```
To get other jobs later:  trade:ask_for_job(Pid).
```

Come si può notare, è stato effettuato un inserimento e la richiesta di un job da parte dello stesso nodo, che ha quindi eseguito il job da lui inserito.
Un ulteriore test potrebbe essere eseguire le funzioni proposte durante l'esecuzione.

testB:main(RiakNode)
La funzione di test è la seguente:

```
main(RiakNode) ->
    Pid = connect:start(RiakNode),
    sleep(3000),
    dht:add_important_job(Pid, test, remove_prefix, ["ciao", "ciao, come va?"]),
    sleep(3000),
    io:format("Now it will be executed:~nask_for_job(Pid).~n"),
    ask_for_job(Pid).
```

Eseguendo questa funzione su un unico nodo, l'output si presenta simile al precedente.

Per ulteriori prove, è possibile eseguire prima le istruzioni del capitolo 3 per collegarsi a Riak, poi le operazioni sulla DHT spiegate nel capitolo 5, e l'operazione di richiesta di un job spiegata nel capitolo 6.
Il modulo test riporta alcune funzioni di esempio, tra cui `test:make_prime/1`, che con un numero elevato in input raggiunge tempi lunghi di esecuzione, prestandosi bene per i test di tutte le funzioni.
Per chiarimenti su quali sono le funzioni utilizzabili, alleghiamo un file html di riepilogo per ogni modulo.

Conclusione

Caratteristiche del sistema

Il nostro sistema permette a qualsiasi nodo di inserire in una DHT job da far eseguire ad altri nodi nella rete. I nodi che eseguono i job possono scegliere se richiedere un job da eseguire, ma non possono scegliere quale job.
Il sistema gestisce autonomamente:

- le cadute dei nodi:
  - caduta del nodo che richiede di eseguire un job;
  - caduta del nodo che inserisce il job;
  - possibilità di riconnettersi senza effetti collaterali se cade un nodo Riak
- cattura diversi errori ed eccezioni:
  - operazioni di inserimento errate nella DHT;
  - operazioni di lettura errate della DHT;
  - cancellazione impropria nella DHT;
  - possibilità di interrompere l'esecuzione del job senza errori;
  - esecuzione di job che portano ad eccezione;
  - eccezione che si presenta, se ci sono problemi nel cluster Riak, durante la ricerca del job da eseguire.

Inoltre notifica l'utente sulle operazioni in corso, sulle possibili operazioni da attuare, e interagisce con lui ogni volta che un job viene eseguito.

Sviluppi futuri

Per il nostro progetto abbiamo immaginato i seguenti sviluppi futuri:

- implementare un meccanismo di pagamento, da parte dei nodi che hanno inserito i job, per i nodi che li eseguono, per esempio sotto forma di bonus per inserire job nella DHT;
- predisporre un sistema di cancellazione sicura dei job con stato terminated o nodedown;
- offrire la possibilità a un nodo che inserisce un job di candidarsi per la sua esecuzione;
- aggiungere limitazioni sull'inserimento di job, soprattutto quelli con priorità alta;
- incrementare i livelli di priorità;
- offrire la possibilità ai worker di scegliere che job eseguire;
- dare la possibilità ai nodi che inseriscono un job di fissare un limite di tempo entro cui l'esecuzione del job deve essere terminata;
- impedire che siano sempre gli stessi nodi ad eseguire i job, anche se risultano quelli con la performance migliore.