Implementazione in Erlang del protocollo Kademlia

Arlind Pecmarkaj[1] and Leonardo Bigelli[2]

[1]a.pecmarkaj@campus.uniurb.it
[2]l.bigelli2@campus.uniurb.it

Riassunto

In questa relazione verrà spiegata l'implementazione in Earlang del protocollo Kademlia, illustrando le principali scelte tecniche attuate.

1 Introduzione

La crescente diffusione di sistemi distribuiti ha reso necessario lo sviluppo di protocolli in grado di garantire un'efficiente gestione e ricerca dei dati in reti dinamiche. Tra questi, Kademlia rappresenta una soluzione particolarmente efficace grazie alla sua architettura basata su una Distributed Hash Table (DHT). Le sue caratteristiche di scalabilità, robustezza e semplicità lo rendono ideale per applicazioni come reti peer-to-peer, condivisione file e sistemi di comunicazione distribuita.

Il presente progetto si propone di implementare il protocollo Kademlia utilizzando Erlang, un linguaggio noto per il supporto nativo alla concorrenza, la tolleranza ai guasti e il modello ad attori. Lo sviluppo mira a esplorare in modo pratico il funzionamento del protocollo, analizzandone l'efficacia e le prestazioni in un ambiente distribuito simulato. Questa implementazione costituisce non solo un'opportunità per approfondire la comprensione teorica del protocollo, ma anche per indagare le implicazioni pratiche legate alla sua applicazione in contesti reali.

2 Kademlia

Kademlia è un protocollo di Distributed Hash Table (DHT) progettato per fornire un meccanismo efficiente e scalabile per la memorizzazione e la ricerca di dati in un sistema distribuito. Il protocollo si basa su identificatori univoci a 160 bit assegnati sia ai nodi della rete sia ai dati [1]. La distanza tra due identificatori è calcolata utilizzando l'operazione XOR, che permette di definire una metrica logica utile per il routing.

Il protocollo prevede l'implementazione di quattro messaggi:

1. `PING`: verifica la raggiungibilità di un nodo. Quando un nodo invia un messaggio PING, il destinatario deve rispondere con un messaggio di conferma, garantendo così la sua presenza nella rete;
2. `STORE`: permette di salvare un valore associato a una chiave specifica in un nodo della rete. Il protocollo prevede che la chiave sia generata in modo deterministico, tipicamente attraverso una funzione di hash applicata al contenuto del dato, garantendo una distribuzione uniforme delle chiavi tra i nodi;
3. `FIND_NODE`: consente di individuare i nodi della rete che si trovano più vicini, in termini di distanza XOR, a un dato identificatore e contatta quest'ultimo. L'operazione è alla base del meccanismo di routing iterativo di Kademlia e si basa su richieste successive ai nodi più vicini noti al richiedente, fino a individuare i destinatari desiderati.
4. `FIND_VALUE`: utilizzato per recuperare un valore specifico associato a una chiave. Se il nodo interrogato possiede il valore richiesto, lo restituisce direttamente, altrimenti, fornisce un elenco di nodi più vicini alla chiave in questione, proseguendo così il processo di ricerca. É simile a `FIND_NODE`;

Le operazioni sopra descritte sono supportate da una tabella di routing strutturata in k-buckets. Ogni bucket raccoglie informazioni su nodi che si trovano a una determinata distanza logica dal nodo locale, consentendo un aggiornamento incrementale ed efficiente

della rete. Questo approccio ottimizza il bilanciamento del carico e la robustezza, garantendo resilienza anche in presenza di variazioni frequenti nella topologia della rete.

3 Implementazione

3.1 Rappresentazione dei Nodi, K-Buckets e Storage

Si è deciso di rappresentare un nodo come una tupla formata da:

- `{ID, Storage, K_Buckets}`, dove:
  - **ID**: Un identificatore univoco del nodo da 160 bit per poter implementare la distanza tramite XOR. Implementato attraverso l'hash con SHA di una stringa di 5 caratteri. I caratteri sono generati casualmente seguendo una distribuzione uniforme nello spazio delle 26 lettere dell'alfabeto. Ogni nodo alla creazione presenta un identificatore temporaneo (`idTMP`). L'assegnazione dell'identificatore effettivo avviene durante l'entrata nella rete da parte del nodo.
  - **Storage**: Una lista di coppie chiave-valore (la coppia è rappresentata da una lista) `[Key, Value]`, utilizzata per memorizzare i dati localmente. Inizializzato con una lista scelta casualmente di cinque caratteri.
  - **K_Buckets**: Una lista strutturata che rappresenta la vista del nodo sui nodi vicini. Ogni K-bucket è rappresentato come una tupla `{Distance, Id, Pid}` dove:
    - **Distance**: La distanza XOR tra l'ID del nodo e il nodo vicino a quello in questione.
    - **Id**: L'id del nodo vicino.
    - **Pid**: Il pid del nodo vicino.

Ogni nodo viene avviato come un processo indipendente tramite la funzione `spawn`, che ne consente l'esecuzione concorrente e la comunicazione asincrona. Non appena un nodo entra nella rete Kademlia (spiegato nella sezione successiva), viene assegnato ad esso sia l'identificativo che la sua K_buckets (tramite la ricezione del messaggio **refresh**.

I K-buckets vengono aggiornati dinamicamente in base alla distanza XOR. Ogni bucket mantiene una lista ordinata di nodi vicini, in modo da ottimizzare il routing nella rete.

Per i K-Buckets si è deciso di implementare un **aggiornamento dinamico:**

- Ogni nodo aggiorna la propria lista K-buckets in base all'attività di rete.
- La lista iniziale di k_buckets viene fornita quando un nodo entra con successo nella rete (`get_4_buckets/1`). Quando si verifica ciò, vengono ricalcolate le K_buckets per tutti i nodi vicini rispetto a quello nuovo.
- Ogni nodo invia periodicamente (ogni 30s) un messaggio **alive** (tramite il pid) a tutti i suoi nodi vicini. Se essi non rispondono vengono eliminati dalla k_buckets del nodo in questione. Questo comportamento è dato tramite l'utilizzo di una funzione già presente in Erlang, la **timer:send_after(T, Pid, MSG)**.
  Per effettuare dei test adeguati, un nodo può ricevere un messaggio di **crash** per poter effettuare una 'exit'.

Si è deciso di tenere 4 Buckets per nodo, di cui 2 sono per i nodi a distanza minima, uno a quelli a distanza intermedia e una per quelli alla distanza più lontana possibile. Il nodo può memorizzare un set ridotto di nodi, ma sufficiente a garantire una copertura ampia, consentendo una navigazione efficiente nella rete senza sovraccaricare la memoria o introdurre eccessiva latenza. Oltretutto vengono semplificate le operazioni di aggiornamento e refresh delle informazioni sui nodi in quanto la dimensione limitata dei k-buckets riduce la complessità computazionale date dalle operazioni di aggiunta e rimozione dei nodi.
Considerando che, ogni volta che un nodo nuovo entra nella rete è presente un ricalcolo parziale della lista dei nodi raggiungibili per molti nodi della rete, scegliere un numero più elevato di K porterebbe ad una crescita eccessiva dei tempi. Questa scelta ha come conseguenza che alcuni nodi potrebbero non essere raggiunti da altri, in quanto la rete non è totalmente connessa.

3.2 Comportamento dei Nodi

Un nodo può essere creato utilizzando la funzione **new_kademlia_node/1**. Quest'ultima consente la creazione di un nodo che, inizialmente, non farà parte della rete di Kademlia. Questa funzione invoca un'altra funzione che definisce il comportamento del nodo stesso tramite uno scambio di messaggi. La funzione inizializza il nodo, generando casualmente un valore da associare allo storage del nodo, fornendo

un identificativo temporaneo (`idTMP`) e impostando come sua `K_buckets` una lista vuota.

Il comportamento di un nodo è definito dalla funzione `node_behavior/1`, che implementa un ciclo di ricezione (`receive`) per gestire i messaggi. I principali messaggi gestiti includono:

- `ping - tramite Id`: Il messaggio generico per elaborare il ping è {ping, TargetId, From, T}, con il seguente comportamento:
  - Se il nodo target è presente nei `K-buckets`, viene contattato direttamente. Una volta contattato, si verifica che il destinatario del messaggio sia effettivamente chi lo sta elaborando tramite il messaggio: {ping, Id, From, StartTime}.
  - Se il target non è presente nei `K-buckets`, il ping viene inoltrato al nodo più vicino utilizzando la funzione `find_closest/2`.
  - Se il nodo cercato viene trovato, si utilizza il seguente messaggio: {ping_result, trovato, StartTime, Pid}; altrimenti viene inviato il messaggio: {ping_result, not_found}.
  - Per evitare loop, viene adottata una politica di ignoramento verso i nodi già visitati.
- `store`:
  - Inserisce una coppia chiave-valore nello storage locale del nodo tramite il seguente messaggio: {store, Value}.
  - Se il valore è già presente, non viene duplicato.
- `find_node`: Il messaggio generico scambiato per cercare un nodo nella rete è {find_node, TargetId, From, T}.
  - Cerca un nodo specifico nei `K-buckets` locali utilizzando il suo Id.
  - Se il nodo non viene trovato, la richiesta viene inoltrata al nodo più vicino. In caso di riscontro negativo, il messaggio scambiato è: {find_result, not_found}; altrimenti: {find_result, FoundId, FoundPid, StartTime}.
  - Come nel caso del ping, i nodi già visitati vengono esclusi per evitare loop.
- `find_value`: Il messaggio scambiato tra i nodi è: {find_value, Key, From, T}, con il seguente comportamento:
  - Prima cerca il valore nello storage locale.
  - Se il valore non viene trovato, inoltra la richiesta ai nodi più vicini nei `K-buckets`.
  - In caso di riscontro positivo, si scambia il seguente messaggio: {value_found, Key, Value, StartTime}; altrimenti: {value_not_found, Key}.
- `send_periodic`:
  - Implementa la sincronizzazione periodica.
  - Ogni 30 secondi, i dati dello storage locale vengono inviati ai nodi presenti nei `K-buckets`. Il messaggio utilizzato per inviare l'insieme di chiavi e valori è: {addToStore, MoreStore}.

Questa implementazione è altamente modulare e consente di estendere ulteriormente le funzionalità senza modificare il comportamento esistente.
Il messaggio del nodo `send_periodic` è fondamentale. Esso consente, ogni 30 secondi, di inoltrare il contenuto dello storage ai nodi presenti nei `K-buckets`. Prima di ciò, viene verificata la raggiungibilità dei nodi vicini: i nodi non raggiungibili vengono rimossi dai `K-buckets`. Segue il codice del messaggio appena descritto:

```
{isAlive, Pid, OtherId} ->
    case length(K_buckets) < 4 of
        true -> NewKB_tmp = {binary_xor
            (Id, OtherId), OtherId, Pid,
             0},
                case lists:any(fun({_,
                    _, X, _}) -> X ==
                    Pid end, K_buckets)
                    of
                 false -> NewKB = [
                     NewKB_tmp] ++
                     K_buckets;
                 true -> NewKB =
                     K_buckets
                end;
        false -> NewKB = K_buckets
    end,
    Pid ! {alive, self()},
    node_behavior({Id, Storage, NewKB})
        ;
{updateB, Pid} ->
    NewKB = lists:foldr(
        fun({A, B, C}, Acc) ->
```

```
15              case C of
16                  Pid -> NewAcc = Acc;
17                  _ -> NewAcc = [{A, B, C
                        }] ++ Acc
18              end,
19              NewAcc
20          end,
21          [],
22          K_buckets
23      ),
24      node_behavior({Id, Storage, NewKB})
            ;
25  {send_periodic} ->
26      Self = self(),
27      lists:foreach(
28          fun({_, OtherId, Pid}) ->
29              spawn(
30                  fun() ->
31                      Pid ! {isAlive,
                            self(), OtherId
                            },
32                      receive
33                          {alive, Pid} ->
                                ok
34                      after 5000 -> Self
                            ! {updateB, Pid}
35                      end
36                  end
37              )
38          end,
39          K_buckets
40      ),
41      lists:foreach(
42          fun({_, _, Pid, _}) ->
43              Pid ! {addToStore, Storage}
44          end,
45          K_buckets
46      ),
47      timer:send_after(30000, self(), {
            send_periodic}),
48      node_behavior({Id, Storage,
            K_buckets});
```

Il nodo invia un messaggio a tutti i suoi nodi presenti nella K_buckets e aspetta un massimo di 5 secondi che riceva una risposta. Se così non fosse, tramite l'invio di un messaggio, viene rimosso dalla sua lista di nodi raggiungibili. Ogni qual volta che un nodo riceve un ping, e ha la propria K_buckets incompleta, viene aggiunto il Pid che ha effettuato il controllo se fosse o meno ancora raggiungibile il nodo.

Per poter visualizzare tutto lo stato del nodo corrente, è possibile inviare a lui un messaggio **getInfo**.

3.3 Implementazione delle funzioni principali

Alla luce del protocollo e dell'implementazione, consideriamo i messaggi principali del programma i seguenti:

- `find_closest`
- `binary_xor`
- `find_node`
- `find_value`

Segue una spiegazione in dettaglio del loro funzionamento.

3.3.1 find_closest

find_closest/2

```
1 find_closest(TargetId, K_bucketsTMP) ->
2     case K_bucketsTMP of
3         [] -> undefined;
4         _ ->
5             Sorted = lists:sort(
6                 fun({Distance1,_,_}, {
                        Distance2,_,_}) ->
7                     Distance1<Distance2
                          end,
8                 lists:map(
9                     fun({_,Id,Pid}) ->
10                        {binary_xor(
                              TargetId,Id)
                              , Id,Pid}
11                    end, K_bucketsTMP)
12            ), hd(Sorted) end.
```

La funzione in questione permette di restituire il nodo più vicino nella propria lista K_buckets rispetto a un dato Id. Ricordiamo che gli Id sono ottenuti utilizzando la funzione **sha** che produce un hash a 160 bit. La funzione calcola la distanza tramite l'utilizzo della funzione **binari_xor/2** tra tutti i nodi nella k_bucket e li ordina in ordine crescente (dal più vicino al nodo più distante). Tramite **hd/1**, restituisce la testa della lista, quinid il nodo più vicino.
Questa è una delle funzioni principali di tutta la rete Kademlia da noi implementata.

3.3.2 binary_xor

binary_xor/2

```
1 binary_xor(Bin1, Bin2) when is_binary(
      Bin1), is_binary(Bin2) ->
2     case byte_size(Bin1) == byte_size(
          Bin2) of
```

```
3         true ->
4             lists:foldl(
5                 fun({Byte1, Byte2}, Acc
                      ) ->
6                     Acc bsl 8 bor (
                          Byte1 bxor Byte2
                          )
7                 end,
8                 0,
9                 lists:zip(binary:
                      bin_to_list(Bin1),
                      binary:bin_to_list(
                      Bin2))
10            );
11        false ->
12            erlang:error(badarith)
13    end.
```

La funzione in questione permette di calcolare la metrica della distanza tramite due parametri dati alla funzione che devono essere espressi in binario. Una volta controllata l'ugualianza tra le due lunghezze, calocla e restituisce il risultato dato. Altrimenti restituisce un errore tipico del linguaggio Erlang.

3.3.3 find_node e find_value

I messaggi **find_node**, **ping** e **find_value** condividono una logica molto simile. Segue il procedimento per l'implementazione di tale logica:

1. Controllare se l'Id da trovare è presente nella propria `K_buckets`. In caso affermativo, non è necessario procedere ulteriormente. Altrimenti, passare al punto successivo;
2. Trovare il nodo più vicino all'Id da cercare all'interno della propria `K_buckets`. Se la `K_buckets` è vuota, restituire che il nodo in questione non è stato trovato;
3. Inoltrare lo stesso messaggio al nodo più vicino individuato.

Il punto 3 è bloccante: il nodo aspetta per un massimo di 5 secondi la risposta dal nodo successivo. Tutti i messaggi bloccanti sono stati gestiti con un meccanismo di **Timeout**, per garantire che il nodo non rimanga bloccato indefinitamente.

Sia `find_node`, `ping` che `find_value` aspettano al massimo 5 secondi.

Questi messaggi sono stati progettati intenzionalmente come bloccanti poiché, in questa implementazione personalizzata di Kademlia, avvengono numerosi scambi di messaggi. Rendere, ad esempio, il messaggio `find_node` bloccante garantisce che il nodo non vada in conflitto con altri messaggi, dando priorità alla gestione di quelli inerenti al `find_node`. Una volta completata la gestione dei messaggi bloccanti, il nodo riprenderà la gestione degli altri messaggi presenti nella sua coda.

Durante lo scambio di messaggi, è possibile incorrere in un ciclo, poiché le connessioni tra i nodi della rete non prevedono un controllo diretto per evitare la creazione di cicli. Per prevenire questa situazione durante la ricerca, è stato implementato un sistema che evita di inoltrare un messaggio a un nodo già attraversato. Ogni volta che un nodo viene visitato, esso viene inserito in una coda. Il prossimo nodo da visitare dovrà essere diverso da qualsiasi nodo presente nella coda dei già visitati.

3.4 Bootstrap e gestione della persistenza

La rete viene inizializzata tramite la funzione `start_system/1`. Si assume che la rete venga creata esclusivamente dopo l'invocazione di questa funzione. Il nodo principale da creare è il nodo **Bootstrap**. Esso funge da punto di ingresso alla rete (caratterizzato da un Id e un Ruolo). Un nuovo nodo, per accedere alla rete, dovrà contattare il nodo `Bootstrap`.
start_system/1

```
1 start_system(P) ->
2     mnesia:create_schema([node()]),
3     mnesia:start(),
4     mnesia:create_table(bootstrap_table
          , [
5         {attributes, record_info(fields
              , bootstrap_table)},
6         {type, set},
7         {ram_copies, [node()]}
8     ]),
9     NodeId = rand:uniform(1 bsl 160 -
          1),
10    BackupNodeId = rand:uniform(1 bsl
          160 - 1),
11    Pid = spawn(node(), fun() ->
          init_bootstrap(NodeId, primary)
          end),
12    global:register_name(bootstrap, Pid
          ),
13    Pid ! {createBackup, BackupNodeId},
14    global:sync(),
15    P ! {ok}.
```

La funzione sopra citata crea, di default, un nodo bootstrap (che avrà il compito principale di fungere da punto di accesso) e il suo backup, con le seguenti caratteristiche:

- **Bootstrap principale** (`primary`):
  - Memorizza i dettagli di tutti i nodi partecipanti nella tabella distribuita **Mnesia**.
  - Gestisce l'ingresso di nuovi nodi tramite un processo transazionale sicuro.
  - Sincronizza i `K-buckets` dei nodi esistenti ogni volta che un nuovo nodo entra nella rete.
- **Bootstrap di backup** (`backup`):
  - Funziona come failover per il nodo principale.
  - È creato dinamicamente per garantire una gestione resiliente in caso di crash del nodo principale.

Entrambi i nodi bootstrap vengono registrati utilizzando il costrutto `global:register_name(name)`. La scelta di registrare i due nodi (rispettivamente 'bootstrap' e 'backup_bootstrap') è stata adottata per eliminare la dipendenza dal PID, rendendo più semplice contattarli direttamente.

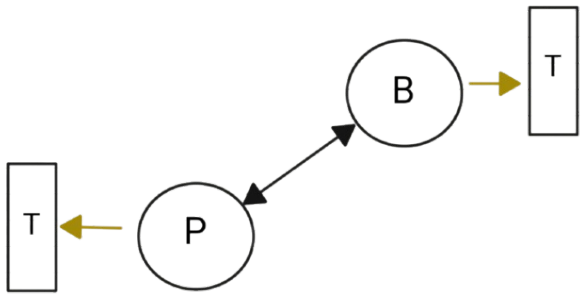

Figura 1: Astrazione del collegamento tra il bootstrap principale e il backup. La freccia nera rappresenta il collegamento bidirezionale tra i due nodi, utilizzato per monitorarsi a vicenda. Il nodo A rappresenta il Bootstrap principale, mentre il nodo B rappresenta il Backup. Le componenti T rappresentano la tabella distribuita gestita dal Bootstrap.

L'algoritmo utilizza il meccanismo di failover di Erlang, basato su messaggi di `exit`, per trasferire il controllo al nodo di backup in caso di guasto:

- Il nodo principale e il nodo di backup si monitorano a vicenda tramite **link**. Il nodo di backup viene creato inviando al nodo principale (bootstrap) un messaggio di **createBackup**, all'interno della funzione che avvia l'intero sistema.
- In caso di crash del nodo di bootstrap principale:
  - Il nodo di backup assume il ruolo di nodo principale, registrandosi come tale.
  - Un nuovo nodo di backup viene creato dinamicamente e registrato.
- In caso di crash del nodo di backup:
  - Il nodo principale ricrea una nuova istanza di backup, registrandola opportunamente.

Ovviamente, se un nodo riceve un segnale che l'altro ha effettuato un `exit`, il nodo in questione gestirà il messaggio senza effettuare anch'esso un `exit`. Questa scelta è necessaria poiché, essendo il bootstrap un nodo principale, non è possibile lasciare la rete priva di un nodo bootstrap. Segue il codice di implementazione che descrive come il nodo bootstrap gestisce il messaggio di `exit`.

exit

```
{'EXIT', _, _Reason} ->
    case Role of
        primary ->
            NewBackupPid = spawn_link(
                node(), fun() ->
                bootstrap_node_loop(
                    rand:uniform(1 bsl
                    160 - 1), backup)
                    end),
                global:register_name(
                    backup_bootstrap,
                    NewBackupPid),
                bootstrap_node_loop(Id,
                     primary);
        backup ->
            global:unregister_name(
                backup_bootstrap),
            global:register_name(
                bootstrap, self()),
            NewBackupPid = spawn_link(
                node(), fun() ->
                bootstrap_node_loop(
                    rand:uniform(1 bsl
                    160 - 1), backup)
                end),
            global:register_name(
                backup_bootstrap,
                NewBackupPid),
            bootstrap_node_loop(Id,
                primary)
    end;
```

Utilizzando la funzione **bootstrap:print_all()**, è possibile visualizzare l'intero contenuto della tabella

di Bootstrap. Quest'ultima consente solo operazioni di inserimento; quindi, se un nodo muore, non verrà rimosso dalla tabella. Anche se a un nuovo nodo venisse fornito il PID di uno non più esistente, ciò non causerebbe problemi, poiché il nodo aggiornerà autonomamente la propria `K_buckets` con il passare del tempo.

I nodi Bootstrap utilizzano **Mnesia**, un database distribuito nativo di Erlang, per memorizzare informazioni su tutti i nodi della rete. In particolare:

- La tabella `bootstrap_table` archivia `ID` e `PID`.
- Le operazioni su questa tabella sono transazionali, garantendo consistenza anche in presenza di errori.

La scelta di utilizzare `Mnesia` deriva dalla necessità di distribuirne il contenuto tra i vari nodi Bootstrap. Di base, `Mnesia` garantisce persistenza locale, ma per un corretto funzionamento della rete è necessario eliminare la cartella locale generata da `Mnesia` per creare nuovamente la rete da zero. In caso contrario, la rete partirebbe considerando nodi creati in precedenza ma non più esistenti, causando conflitti.

Il comportamento generale del Bootstrap è definito dalla funzione **bootstrap_node_loop**, che consente lo scambio dei seguenti messaggi:

- `ping`: stampa a video un messaggio che segnala la ricezione del ping. Il messaggio utilizzato è: {ping, From}.
- `createBackup`: consente la creazione del nodo di backup, ovvero una nuova istanza del nodo principale che ha il compito di monitorarlo. Il messaggio utilizzato è `{createBackup, backupId}`.
- `enter`: permette a un nuovo nodo, non ancora parte della rete di Kademlia, di effettuare la richiesta di ingresso. Questo passaggio è considerato obbligatorio. Il messaggio utilizzato è `{enter, From}`.
- `crash`: forza il nodo in questione a effettuare una `exit`. Il messaggio utilizzato è `{crash}`.
- `EXIT`: consente la gestione dell'uscita di un nodo Bootstrap, qualunque sia la causa.

3.4.1 Entrata di un nodo nella rete

Come affermato precedentemente, qualsiasi nod deve contattare il nodo Bootstrap per fare richiesta di ingresso nella rete Kademlia. Questo viene effettuato inviando il seguente messaggio:

```
global:whereis_name(bootstrap) ! {enter, Nodo}.
```

Dove, `Nodo` non è altro che il nodo creato in precedenza. Una volta ricevuto il messaggio, il bootstrap inserirà il nodo nella sua bootstrap_table e fornirà al nodo appena entrato la lista di k_buckets. La funzione che si occupa della creazione della k_buckets è chiamata **get_4_buckets/1**:

```
get_4_buckets(NodeId) ->
    Tran = fun() ->
        mnesia:foldr(
            fun(#bootstrap_table{id = Id, pid = Pid}, Acc) ->
            case Id == NodeId of
                true ->
                    Acc;
                false ->
                  Distance=binary_xor(NodeId, Id),
                  Acc ++ [{Distance, Id, Pid}]
            end,
        [],
        bootstrap_table
    )
    end,
    {atomic, AllRecords} = mnesia:transaction(Tran),
    SortedRecords = lists:sort(fun({D1,_,_},{D2,_,_})->D1<D2 end, AllRecords),
    case SortedRecords of
        [] ->
            [];
        [_] ->
            SortedRecords;
        _ ->
            ClosestTwo = lists:sublist(SortedRecords, 2),
            TotalNodes = length(SortedRecords),
            MiddleNodeIndex =
                case TotalNodes > 2 of
                    true -> round(TotalNodes / 2);
                    false -> 1
                end,
            MiddleNode = lists:nth(MiddleNodeIndex, SortedRecords),
            FarthestNode = lists:last(SortedRecords),
```

```
34              UniqueNodes = lists:usort(
                    ClosestTwo ++ [
                    MiddleNode, FarthestNode
                    ]),
35              lists:sublist(UniqueNodes,
                    4)
36      end.
```

3.5 Considerazioni

L'implementazione così descritta, riesce a presentare queste proprietà:

- **Concorrenza e scalabililità**:
  - Ogni nodo è un processo indipendente, sfruttando il modello ad attori per una gestione efficiente della concorrenza.
  - L'approccio a bucket consente la scalabilità per reti di grandi dimensioni.
- **Robustezza**:
  - Il sistema tollera il fallimento dei nodi, garantendo che la rete rimanga operativa anche in condizioni avverse.
- **Estensibilità**:
  - La modularità del codice consente di aggiungere nuove funzionalità senza interrompere il comportamento esistente.

4 Creazione e avvio della rete

Per creare da zero e avviare la rete Kademlia descritta, vanno seguiti i seguenti passi:

1. Controllare che non sia presente la cartella riferita ad Mnesia;
2. Avviare Erlang su due terminali distinti. Ciascuna istanza di Elang deve avere il parametro `sname` diverso;
3. Effettuare dal nodo 1 una `spawn('nodo2@...', fun() -> ok end.)` per far communicare le due istanze;
4. `bootstrap:start_system(self()).`
5. Creiamo un nuovo nodo che però ancora non farà parte della rete di Kademlia:
   `{_, Nodo1} = nodo:new_kademlia_node(self()).`
6. `global:whereis_name(boostrapt) ! {enter, Nodo1}.`

5 Esperimenti e Risultati

In questa sezione vengono descritti gli esperimenti svolti e i relativi risultati. Le simulazioni sono state effettuate con una rete inizializzata a 0 nodi, 100 nodi e 500 nodi. Le operazioni princi-pali testate includono l'inserimento di nodi, la ricerca (`find_node` e `find_value`), e il ping. I risultati, espressi in microsecondi, sono riassunti nelle tabelle seguenti.

5.1 Metriche di inserimento e ricerca

La Tabella 1 riporta i tempi medi di esecuzione per l'inserimento di nodi e le operazioni di ricerca (`find_node` e `ping`) nelle diverse configurazioni della rete. Per quanto riguarda il tempo dello **store** (inserimento di un valore all'interno di un nodo), il processo risulta molto rapido, con una media di 3175 microsecondi. Questo tempo non dipende dal numero di nodi presenti nella rete, grazie all'implementazione ottimizzata del sistema.

	Numero di nodi		
Operazione	0	100	500
Inserimento	2560	119,808	3,302,607
`find_node`	-	10,957	5734
`ping`	-	8500	7373

Tabella 1: Tempi medi per inserimento e ricerca (`find_node`, `ping`), in microsecondi.

5.2 Efficienza di `find_value`

La nostra implementazione di Kademlia dimostra un'elevata efficienza nell'operazione `find_value`. I nodi inviano periodicamente (ogni 30 secondi) il loro contenuto (chiavi e valori) ai nodi raggiungibili. Questo meccanismo consente di mantenere un tempo di ricerca costante, indipendentemente dal numero di nodi nella rete.

	Numero di nodi		
Operazione	0	100	500
`find_value`	-	102	102

Tabella 2: Tempo medio di esecuzione di `find_value`, in microsecondi.

5.3 Tempi di Ripristino del Bootstrap

In caso di fallimento di un nodo **Bootstrap**, il sistema è progettato per garantire un ripristino rapido ed efficiente come riportato nella Tabella 3. Questi valo-

Nodo Bootstrap	Tempo Medio [μs]
Principale	1228
Backup	1024

Tabella 3: Tempi medi di ripristino in caso di fallimento del Bootstrap.

ri sono stati ottenuti simulando volutamente un fallimento del nodo di Bootstrap mediante il comando `crash`. Quest'ultimo genera un messaggio di `exit`, attivando il meccanismo di ripristino implementato nella logica del sistema.

Riferimenti bibliografici

[1] Petar Maymounkov e David Mazières. «Kademlia: A Peer-to-Peer Information System Based on the XOR Metric». In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 53–65. ISBN: 3540441794.