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 vrede l'implementazione di quattro messaggi:

1. `PING`: verifica la raggiungibilità di un nodo. È utilizzato per mantenere aggiornata la tabella di routing, eliminando i nodi non più attivi. 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. 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, con-

sentendo 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, Timer}`, 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.
  - **Storage**: Una lista di coppie chiave-valore (la coppia è rappresentata da una lista) `[Key, Value]`, utilizzata per memorizzare i dati localmente.
  - **K_Buckets**: Una lista strutturata che rappresenta la vista del nodo sui nodi vicini. Ogni K-bucket è rappresentato come una tupla `{Distance, Nodi}` dove:
    - **Distance**: La distanza XOR tra l'ID del nodo e gli altri nodi presenti nel bucket.
    - **Nodi**: Una lista di nodi (tuple `{ID, PID}`) che si trovano entro una certa distanza.
  - **Timer**: Un valore che rappresenta il timestamp dell'ultima attività del nodo.

Ogni nodo viene avviato come un processo indipendente tramite la funzione `spawn`, che ne consente l'esecuzione concorrente e la comunicazione asincrona.

I K-buckets vengono e 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:

- Aggiornamento Dinamico:
  - Ogni nodo aggiorna la propria lista K-buckets in base all'attività di rete (es. ricezione di ping).
  - I bucket vengono ricalcolati ogni volta che un nuovo nodo entra nella rete (`get_4_buckets/1`).
- Meccanismo di Timeout:
  - Ogni nodo memorizza un timer per ogni peer nella sua lista, eliminando i nodi inattivi dopo un certo periodo.

Si è deciso di tenere 4 K-Buckets per nodo, di cui 2 sono per i nodi a distanza vicina, uno a quelli a distanza intermedia e una per quelli alla distanza più lontana possibile. Inoltre 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 nodu in quanto la dimensione limitata dei k-buckets riduce la complessità computazionale date dalle operazioni di aggiunta e rimozione dei nodi.

3.2 Comportamento dei Nodi

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`:
  - Se il nodo target è presente nei K-buckets, viene contattato direttamente.
  - Se il target non è nei K-buckets, il ping viene inoltrato al nodo più vicino utilizzando la funzione `find_closest/2`.
  - Per evitare loop, viene attuata una politica di ignoramento verso i nodi già visitati.
- `store`:
  - Inserisce una coppia chiave-valore nello storage locale del nodo.
  - Se il valore è già presente, non viene duplicato.
- `find_node`:
  - Cerca un nodo specifico nei K-buckets locali.
  - Se il nodo non è trovato, la richiesta viene inoltrata al nodo più vicino.

  - Come per il ping, i nodi già visitati non vengono considerati per evitare loop.
- `find_value`:
  - Prima cerca il valore nello storage locale.
  - Se non trovato, inoltra la richiesta ai nodi più vicini nei K-buckets.
- `send_periodic`:
  - Implementa la sincronizzazione periodica.
  - Ogni 30 secondi, i dati dello storage locale vengono inviati ai nodi presenti nei K-buckets.

Questa implementazione è altamente modulare e consente l'estensione per ulteriori funzionalità senza modificare il comportamento esistente.

3.3 Inizializzazione della rete e gestione della persistenza

La rete è inizializzata tramite la funzione `start_system/1`, che crea un nodo bootstrap con le seguenti caratteristiche:

- **Nodo Principale** (`primary`):
  - Memorizza i dettagli di tutti i nodi partecipanti nella tabella distribuita **Mnesia**.
  - Gestisce l'ingresso di nuovi nodi con un processo transazionale sicuro.
  - Sincronizza i K-buckets dei nodi esistenti ogni volta che un nuovo nodo entra nella rete.
- **Nodo di Backup** (`backup`):
  - Funziona come failover per il nodo principale.
  - È creato dinamicamente e registrato tramite `register/2` per garantire una gestione resiliente in caso di crash del nodo principale.

L'algoritmo utilizza il meccanismo di failover di Erlang basato su `trap_exit`, per passare il controllo al backup in caso di guasto:

- Il nodo principale e il nodo di backup si monitorano a vicenda tramite `trap_exit`.
- In caso di crash di uno dei due nodi bootstrap:
  - Il nodo rimanente assume il ruolo principale.
  - Un nuovo nodo di backup viene creato dinamicamente per ripristinare il failover.

Il sistema utilizza **Mnesia**, un database distribuito nativo di Erlang, per memorizzare informazioni sui nodi nella rete. In particolare:

- La tabella `bootstrap_table` archivia ID, PID e l'ultimo timestamp di ping per ogni nodo.
- Le operazioni su questa tabella sono transazionali, garantendo consistenza in presenza di errori.

3.4 Considerazioni

- **Concorrente e Scalabile**:
  - 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 Metriche

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.