Università degli Studi di Urbino Carlo Bo

Informatica e Innovazione Digitale

Kademlia: una simulazione in Erlang

Progetto di
Applicazioni Distribuite e Cloud Computing
Sessione invernale 2025

Autori:

Nunzio D'amore
Matricola: 329163
Email: n.damore@campus.uniurb.it

Francesco Pio Rossi
Matricola: 331825
Email: f.rossi51@campus.uniurb.it

1 Il Protocollo Kademlia

Kademlia è un protocollo peer-to-peer progettato per implementare una *Distributed Hash Table* (DHT), una struttura dati che rappresenta un dizionario composto da un vasto numero di coppie chiave-valore. Il termine "distributed" indica che il dizionario è distribuito tra diversi nodi di una rete, evitando così il ricorso a un unico server centrale. La parola "hash", invece, suggerisce che le chiavi del dizionario sono generate tramite una funzione di hash, come ad esempio *SHA-256*, che garantisce una distribuzione uniforme e sicura dei dati all'interno della rete.

Un'architettura di questo tipo elimina i problemi legati a un server centrale, che funge da "single point of failure". Tuttavia, introduce anche nuove sfide: ogni nodo può agire sia come server, memorizzando i dati, sia come client, richiedendo informazioni ad altri nodi. La difficoltà principale sta nel fatto che, in un sistema distribuito, ogni nodo deve essere in grado di individuare a chi rivolgersi per ottenere un dato specifico. Kademlia affronta questo problema in modo intelligente ed estremamente efficiente.

1.1 Il Concetto di Distanza in Kademlia

In Kademlia lo spazio dei nodi coincide con quello delle chiavi, essendo entrambi descritti da un identificatore univoco di 160 bit. Questo consente di definire una misura di "distanza" tra un nodo e una certa chiave – o tra due nodi – chiamata *distanza* XOR. La distanza XOR è definita come il risultato dell'operazione XOR tra due identificatori, interpretata come un intero:

$$d(x, y) = x \oplus y$$

Questa nozione può essere considerata una vera e propria distanza, poiché soddisfa le seguenti proprietà fondamentali:

1. **Non negatività:** $d(x, y) \geq 0$ per ogni coppia di punti x e y. Inoltre, $d(x, y) = 0$ se e solo se $x = y$.
2. **Simmetria:** $d(x, y) = d(y, x)$ per ogni coppia di punti x e y.
3. **Disuguaglianza triangolare:** $d(x, z) \leq d(x, y) + d(y, z)$ per ogni tripla di punti x, y e z.

Possiamo dunque sfruttare lo spazio degli identificatori comune per calcolare la distanza tra un nodo e una chiave, e più tale distanza è piccola più è alta la probabilità che quel nodo memorizzi effettivamente il valore associato a quella chiave.

1.2 Routing

Poiché non c'è un'entità centrale che gestisce le operazioni di lookup, ogni nodo deve essere in grado di identificare chi contattare per ottenere un dato

specifico. Per farlo, ogni nodo dispone di una tabella di routing che gli consente di determinare facilmente a chi rivolgersi quando ha bisogno di recuperare un'informazione.

In Kademlia la tabella di routing di un nodo contiene riferimenti a nodi situati in diverse "regioni" della rete. Queste regioni sono definite dai prefissi degli identificatori, ed è bene che ogni nodo abbia almeno un riferimento a un altro nodo posizionato in una regione della rete diversa dalla propria.
La tabella di routing può essere immaginata come un albero binario in cui ogni foglia rappresenta un prefisso binario che contiene un insieme di puntatori con al massimo `BUCKET_SIZE` nodi caratterizzati da quel prefisso.

Supponiamo che un nodo, chiamato N_1, abbia un identificatore ID = 010. La sua tabella di routing conterrà riferimenti a nodi che si trovano in 3 diverse regioni, ognuna caratterizzata da un certo prefisso. In particolare, la tabella di N_1 dovrebbe includere:

- Almeno un nodo il cui identificatore inizia con il prefisso `1`.
- Almeno un nodo il cui identificatore inizia con il prefisso `00`.
- Un nodo il cui identificatore inizia con il prefisso `011`, cioè il nodo "adiacente" a N_1.

La Figura 1 mostra l'albero binario associato alla tabella di routing del nodo 010, dove ogni foglia rappresenta una entry della tabella.

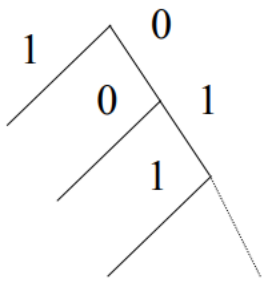

Figure 1: Albero binario che rappresenta la tabella di routing del nodo 010.
fonte: `https://pages.di.unipi.it/ricci/20-03-09-Kademlia.pdf`

La vicinanza tra un nodo ed N_1 viene determinata dalla somiglianza dei loro prefissi: più i prefissi sono uguali, più i due nodi sono vicini secondo la nozione di distanza XOR. Un esempio di tabella di routing per N_1 potrebbe essere la seguente, dove ogni nodo di contatto è una tripla contenente l'identificatore del nodo, il suo indirizzo IP e la porta UDP, che sono le informazioni necessarie per stabilire una comunicazione con esso.

Prefisso	Nodi di contatto
1	$N_2, \quad N_5$
00	N_3
011	N_4

Ciascuna riga della tabella di routing corrisponde a una foglia dell'albero binario e può contenere riferimenti a un massimo di `BUCKET_SIZE` nodi diversi. Questo limite aumenta le probabilità di disporre di almeno un riferimento a un nodo ancora attivo. Nel caso di Kademlia il valore di `BUCKET_SIZE` è fissato a 20 e l'insieme dei `BUCKET_SIZE` nodi associati a un certo prefisso si dice `K-bucket`. Il `K-bucket` è ordinato in base al tempo dell'ultimo contatto, con il nodo non contattato da più tempo posizionato in testa.

Un nodo tende a conoscere un numero maggiore di nodi vicini a sé rispetto a quelli lontani. Tuttavia, è comunque in grado di raggiungere qualsiasi nodo nella rete attraverso una serie di richieste inoltrate: la copertura dell'intera rete avviene con una complessità logaritmica rispetto al numero di nodi, rendendo le operazioni di ricerca particolarmente efficienti.

1.3 Aggiornamento della Routing Table

Quando un nodo riceve un messaggio da un altro nodo della rete tenta di aggiornare la propria tabella di routing nella speranza di riempirla con nodi attivi. Supponiamo che un nodo A riceva un messaggio da un nodo B. In questo caso, A calcola la distanza $d(A, B)$ e determina in quale `K-bucket` della propria tabella di routing inserire B.

Se non esiste il `K-bucket` per B, il nodo A lo crea; se invece esiste, A controlla se è pieno (verifica dunque se ci sono già `BUCKET_SIZE` nodi di contatto per quel `K-bucket`). Se il `K-bucket` non è pieno, il nodo B viene inserito nella coda; se invece è pieno, il nodo A verifica se il nodo in testa alla lista, ossia quello che non è stato contattato da più tempo, è ancora attivo inviandogli un messaggio di ping.
Se il nodo in testa non risponde, viene rimosso dal `K-bucket` da A che inserisce B in coda. Se invece il nodo in testa risponde, A lo sposta nella coda della lista e scarta B, senza aggiungerlo alla propria tabella di routing.

Questa procedura si basa sull'osservazione empirica secondo cui i nodi che restano attivi per un lungo periodo di tempo hanno una maggiore probabilità di rimanere attivi anche in futuro. Per questo motivo, il sistema privilegia il mantenimento di tali nodi. Inoltre la procedura assicura che le tabelle di routing siano costantemente aggiornate, purché i messaggi continuino a circolare nella rete.

1.4 Interfaccia di Comunicazione

In Kademlia i nodi comunicano attraverso le seguenti API:

- `PING()`: serve per verificare se il nodo destinatario del messaggio è ancora attivo. Viene utilizzato da un nodo principalmente per aggiornare la propria routing table.
- `FIND_NODE(ID)`: il nodo destinatario del messaggio risponde con i `BUCKET_SIZE` nodi nella propria routing table più vicini a `ID`, che può essere sia l'identificatore di un nodo che di una chiave del dizionario.
- `FIND_VALUE(Key)`: simile a `FIND_NODE`, ma se il destinatario del messaggio possiede la chiave richiesta `Key` restituisce direttamente il valore associato.
- `STORE(Key, Value)`: salva la coppia `Key`/`Value` nel nodo destinatario del messaggio.

Ogni messaggio include un *nonce* generato dal mittente. Questo permette di associare ogni risposta alla rispettiva richiesta.

Quando un nodo desidera unirsi alla rete Kademlia utilizza la funzione `Join()`. Per avviare questo processo il nodo entrante deve conoscere almeno un altro nodo già presente nella rete. Il nodo entrante aggiunge il nodo che conosce nella propria routing table e, tramite una serie di richieste di tipo `FIND_NODE`, raccoglie informazioni sui nodi vicini a sé, aggiornando gradualmente la propria routing table.

2 Simulazione in Erlang: le Principali Scelte di Progetto

In questa sezione viene introdotta la nostra simulazione di Kademlia realizzata in Erlang, con una descrizione delle principali scelte progettuali effettuate, delle motivazioni alla base di tali scelte e delle semplificazioni introdotte. Nella simulazione ogni nodo di Kademlia è rappresentato come un processo Erlang. Pertanto, quando in seguito faremo riferimento al termine "nodo", ci riferiremo a un processo Erlang che simula il comportamento di un nodo in Kademlia, e non ai nodi nel contesto di Erlang.

2.1 Utilizzo di gen_server per la gestione dei nodi

Nella simulazione si è scelto di utilizzare il behaviour `gen_server` di Erlang per gestire ogni nodo della rete Kademlia. Questa scelta è stata motivata dalla natura stessa dei nodi nel protocollo – che devono gestire richieste provenienti da altri nodi durante il loro ciclo di vita – e dai vantaggi offerti da `gen_server`. In particolare questo behaviour automatizza la gestione dello stato di ogni nodo e consente di gestire richieste sia sincrone che asincrone in maniera molto semplice.
Oltre ai vantaggi appena citati, `gen_server` associa automaticamente ogni risposta alla richiesta originale, consentendo di distinguere chiaramente tra richieste diverse. In particolare, le richieste sincrone vengono gestite dalla funzione

`handle_call/3`, che utilizza il valore `Tag` presente nella tupla `From = {Pid, Tag}` come identificatore univoco per ogni richiesta, svolgendo un ruolo analogo a quello del nonce descritto nella Sezione 1.4.

Nella nostra simulazione, un nodo Kademlia è un processo Erlang che viene istanziato tramite la funzione `start_server/4`. Questa funzione, a sua volta, utilizza internamente la funzione `start/3` di `gen_server` per gestire l'avvio del processo. L'istanzianzione del processo porta all'esecuzione della funzione `init/1` di `gen_server`, che si occupa invece di inizializzare lo stato del nodo. Lo stato di un nodo Kademlia nella simulazione è descritto dalla tupla `{Routing Table, ValuesTable, K, T, Bucket_Size, MyValuesTable}`, dove:

- `RoutingTable` rappresenta la tabella di routing locale del nodo.
- `ValuesTable` contiene le coppie chiave/valore memorizzate localmente dal nodo.
- `K` indica il numero di bit utilizzati per rappresentare gli identificatori dei nodi e delle chiavi.
- `T` specifica ogni quanti millisecondi i dati vengono ripubblicati.
- `Bucket_Size` è la dimensione del `k_bucket`, fissata a 20 nella specifica di Kademlia.
- `MyValuesTable` è la tabella dei valori pubblicati dal nodo stesso.

Il Listing 1 mostra il codice della funzione `init/1`, che inizializza lo stato del nodo. `RoutingTable` e `ValuesTable` sono inizializzate come tabelle di tipo *set* vuote, ed hanno un nome univoco per ogni nodo.
All'interno di un Listing, il simbolo [...] viene utilizzato come segnaposto per indicare che alcune righe di codice sono state omesse. In questo caso, sono momentaneamente tralasciati alcuni passaggi di configurazione, la cui funzionalità verrà spiegata nelle sezioni successive. Si trascuri per ora il significato dei parametri `InitAsBootstrap` e `Verbose`.

Listing 1: Codice della funzione `init/1`, che inizializza lo stato di un nodo

```
% Initializes the state of the node when it is created.
init([K, T, InitAsBootstrap, Verbose]) ->
    [...]
    % Generate a unique integer to create distinct ETS
        table names for each node.
    UniqueInteger = integer_to_list(erlang:unique_integer([
        positive])),
    % Create unique table name for routing by appending the
         unique integer.
    RoutingTableName = list_to_atom("routing_table" ++
        UniqueInteger),
    ValuesTableName = list_to_atom("values_table" ++
        UniqueInteger),
```

```
9       MyValuesTableName = list_to_atom("my_values_table" ++
            UniqueInteger),
10      % Initialize ETS table with the specified properties.
11      RoutingTable = ets:new(RoutingTableName, [set, public])
            ,
12      % Initialize ValuesTable as an empty map.
13      ValuesTable = ets:new(ValuesTableName, [set, public]),
14      % Initialize MyValuesTable as an empty map.
15      MyValuesTable = ets:new(MyValuesTableName, [set, public
            ]),
16      % Define the bucket size for routing table entries.
17      Bucket_Size = 20,
18      [...]
19      % Return the initialized state, containing ETS tables
            and configuration parameters.
20      {ok, {RoutingTable, ValuesTable, K, T, Bucket_Size,
            MyValuesTable}}
21  .
```

2.2 Gestione degli identificatori

Gli identificatori dei nodi e delle chiavi sono rappresentati come `bitstring`, un tipo di dato in Erlang progettato per rappresentare sequenze di bit a basso livello. Questo approccio garantisce un'elevata efficienza sia in termini prestazionali che di occupazione della memoria.

La funzione illustrata nel Listing 2 calcola l'identificatore di un nodo o di una chiave utilizzando l'algoritmo di hashing `SHA256`. Poiché la simulazione è parametrizzata rispetto `K` – ossia il numero di bit che compone ogni identificatore – l'hash generato viene troncato ai primi `K` bit, adattandolo così alla configurazione scelta.

Listing 2: Funzione che calcola l'identificatore di un nodo o di una chiave

```
1  % This function is used to create a K-bit hash from a given
       data.
2  k_hash(Data, K) when is_pid(Data) ->
3      k_hash(pid_to_list(Data), K);
4  k_hash(Data, K) when is_integer(K), K > 0 ->
5      Hash = crypto:hash(sha256, Data),
6      % Takes the first K bits of the hash
7      <<KBits:K/bits, _/bits>> = Hash,
8      KBits.
```

L'identificatore di un nodo viene calcolato utilizzando il PID del processo Erlang, convertito in una stringa e passato come argomento `Data`. Per calcolare l'identificatore di una chiave, invece, si utilizza direttamente la chiave del dizionario, considerata come una stringa.

2.3 Gestione della `RoutingTable` e della `ValuesTable`

Nella simulazione la routing table è implementata come una tabella `ETS`, scelta che garantisce operazioni di lookup estremamente rapide. Ogni record della tabella corrisponde a una foglia dell'albero binario descritto nella Sezione 1.2 ed è costituito da due attributi:

- Il primo attributo, utilizzato come chiave, identifica una specifica regione dello spazio degli identificatori, definita da un determinato prefisso.
- Il secondo attributo rappresenta il `k-bucket` associato a quella regione, contenente informazioni sui nodi di contatto appartenenti a quell'intervallo.

Poiché le chiavi della tabella sono univoche, la struttura più adatta per implementare la tabella è il `set`. Si osservi che la routing table può contenere al massimo `K` entry, dove `K` rappresenta il numero di bit utilizzati per gli identificatori.

Entrando più nel dettaglio, il `K-bucket` è rappresentato come una lista Erlang composta da al più `BUCKET_SIZE` tuple, dove ciascuna tupla rappresenta un nodo di contatto e include l'identificatore del nodo e il suo `PID` (Process ID in Erlang). Il `PID` è l'informazione necessaria per contattare il nodo, e sostituisce l'indirizzo IP e la porta UDP utilizzati nei sistemi Kademlia tradizionali.

Rappresentare il `k-bucket` come una lista è una scelta naturale e razionale. Il `k-bucket`, infatti, è un elenco ordinato: i nodi meno recentemente contattati si trovano in testa, mentre quelli visti più di recente si trovano in coda. Questa struttura consente di gestire facilmente l'aggiornamento del `k-bucket` descritto nella Sezione 1.3. In particolare, quando si contatta il nodo in testa, è facile spostarlo in coda se risponde al ping, dimostrando così di essere ancora attivo.

`K-bucket` = `[{NodeId1, Pid1}, {NodeId2, Pid2}, ...,{NodeIdN, PidN}]`

Per quanto riguarda la chiave del set che identifica una specifica regione dell'albero binario, non si utilizza un prefisso, ma un numero intero n che va da 1 a `K`. Questo numero rappresenta la posizione del primo bit che differisce rispetto al bit corrispondente nell'identificatore del nodo proprietario della routing table. Ad esempio, nel caso di una routing table di un nodo con identificatore 0100, la tabella conterrà 4 chiavi. La prima chiave, 1, corrisponde ai nodi che hanno un prefisso che inizia con 1, quindi rappresenta i nodi più lontani dal nodo proprietario della routing table. La seconda chiave, 2, identifica i nodi il cui prefisso inizia con 00, e così via per le altre chiavi. La scelta di una chiave così fatta di consente di eseguire la ricerca del `K-bucket` adatto in maniera più agevole. Un esempio di una tabella di routing strutturata in questo modo è riportato nella Tabella 1.

Anche per la gestione della tabella dei valori viene utilizzata una tabella `ets` di tipo `set`, dove ogni record ha come chiave primaria l'identificatore della chiave del dizionario distribuito e come secondo attributo una mappa Erlang. Tale scelta viene motivata nella sezione che segue.

Chiave	K-Bucket (Lista di Nodi)
1	[{1000, <0.130.0>}, {1010, <0.150.0>}]
2	[{0010, <0.102.0>}, {0001, <0.110.0>}]
3	[{0110, <0.130.0>}]
4	[{0101, <0.170.0>}]

Table 1: Esempio di Routing Table in Erlang per il nodo con identificatore 0100

2.4 Gestione delle collisioni con `K` piccolo

La simulazione è parametrizzata rispetto a `K`, cioè il numero di bit utilizzati per gli identificatori di nodi e chiavi. Questo comporta la necessità di gestire le collisioni, ossia cosa fare quando due nodi o due chiavi del dizionario condividono lo stesso identificatore, specialmente quando `K` è molto piccolo.

L'identificatore di un nodo viene ottenuto troncando i primi `K` bit dell'output di `SHA256` applicato al PID del processo. Di conseguenza, se `K` è piccolo, è più probabile che due nodi abbiano lo stesso identificatore. In questo caso, i due nodi vengono trattati come distinti venendo salvati nello stesso `K-bucket` di una routing table, differenziandoli tramite il PID.

Nella simulazione si potrebbe immaginare che la chiave del dizionario dei dati corrisponda al nome di un file, mentre il valore sia il suo contenuto. Dunque l'identificatore si ottiene calcolando l'hash del nome del file, troncato ai primi `K` bit. Tuttavia, se `K` è piccolo, è possibile che due file con nomi diversi abbiano lo stesso identificatore.
`ValuesTable` utilizza l'identificatore del file come chiave primaria, mentre il valore associato è una mappa Erlang che collega il nome di un file al suo contenuto. Quando viene invocata la funzione `FIND_VALUE(FileName)`, il nome del file viene passato come input e, internamente, viene calcolato il suo identificatore. Se il nodo destinatario della richiesta contiene l'identificatore nella sua `ValuesTable`, controlla la mappa associata, che potrebbe contenere più file. Verrà quindi selezionato solo il file che corrisponde al nome richiesto, se presente.
Questo meccanismo presuppone che i nomi dei file nella rete siano univoci, altrimenti un file potrebbe essere sovrascritto da un altro con lo stesso nome.

2.5 I thread

Nel contesto della simulazione, un ruolo fondamentale è svolto dai cosiddetti "thread", particolari processi creati da un nodo padre tramite la funzione `thread:start`. Una caratteristica distintiva dei thread è la capacità di inviare messaggi utilizzando il PID del padre, motivo per cui possono essere considerati a tutti gli effetti dei sotto-processi.

Questo comportamento si basa sull'uso del *dizionario di processo*, una struttura dati in Erlang che consente di memorizzare privatamente coppie chiave/-

valore in modo semplice grazie alle primitive `put` e `get`. Al momento della creazione, ogni thread memorizza il PID del padre nel proprio dizionario di processo tramite la funzione `save_address/0` (Listing 3), associandolo alla chiave `my_address`.

Listing 3: Salvataggio del PID del padre nel dizionario di processo

```
save_address(Address) ->
    put(my_address, Address).
```

Di conseguenza, all'interno della simulazione la funzione `self()` viene sostituita da `my_address` (Listing 4), che permette ai thread di recuperare il PID del processo padre, utilizzandolo come se fosse il proprio.

Listing 4: Recupero del PID del padre dal dizionario di processo

```
my_address() ->
    case get(my_address) of
        undefined -> self();
        Address -> Address
    end.
```

Quando il processo principale di un nodo richiama `my_address/0`, questa restituirà automaticamente il contenuto di `self/0` non essendo inizializzato il valore `my_address` nel dizionario.

Il Listing 5 mostra la funzione `thread:start/1`, utilizzata da un nodo per avviare un nuovo thread. Durante l'istanziazione, l'indirizzo del processo padre viene memorizzato nel dizionario dei processi del thread. Successivamente, il thread inizia ad eseguire una funzione, che potrebbe essere un ciclo infinito. La funzione `spawn_link/1` crea un link tra un nodo e i suoi thread, così che se il nodo muore, anche i suoi thread vengono terminati. Se invece è un thread a morire, l'errore che ne deriva viene gestito dal nodo padre tramite `trap_exit`, che dunque non termina.

Listing 5: Istanziazione del thread tramite la funzione `thread:start/1`

```
start(Function) ->
    ParentAddress = com:my_address(),
    Verbose = utils:verbose(),
    Pid = spawn_link(
        fun()->
            % Saving parent address and verbose in the new
                process so it can behave like the parent
            utils:set_verbose(Verbose),
            com:save_address(ParentAddress),
            Function()
        end
    ),
    ?MODULE:save_thread(Pid),
    Pid.
```

I thread consentono di eseguire operazioni in parallelo per conto del nodo padre, che può così continuare a gestire le richieste in arrivo senza interruzioni. Questa scelta progettuale migliora l'efficienza della simulazione, distribuendo i compiti tra diversi sotto-processi. Nella simulazione, ogni nodo è padre di tre thread:

1. `republisher`: è il thread responsabile della ripubblicazione periodica dei dati di cui il nodo padre è proprietario ogni `T` millisecondi;
2. `spare_manager`: è il thread incaricato di gestire il salvataggio di un nodo nella tabella routing del padre quando il `K_bucket` di destinazione è pieno;
3. `routing_table_filler`: è il thread che si occupa di avviare la fase di join di un nodo e di mantenere sempre piena la sua routing table, controllando quindi che ogni `k_bucket` contenga almeno un elemento.

Essi vengono tutti istanziati al momento della creazione del nodo, all'interno della funzione `init/1` (Listing 6). Se uno dei tre thread muore, viene ristanziato dal nodo padre. Le loro funzioni verranno spiegate più nel dettaglio nella sezione successiva.

Listing 6: Istanziazione dei thread di un nodo

```
1 init([K, T, InitAsBootstrap, Verbose]) ->
2     [...]
3     spare_node_manager:start(RoutingTable, K),
4     republisher:start(RoutingTable, K, T, Bucket_Size,
          MyValuesTable),
5     join_thread:start(K,RoutingTable,Bucket_Size),
6     [...]
7 .
```

3 Ruoli e funzioni dei Thread

In questa sezione vengono descritti in dettaglio i compiti svolti dai tre thread associati a ciascun nodo Kademlia nella simulazione. I thread sono utilizzati da ogni nodo per delegare attività specifiche o eseguire controlli periodici, migliorando così le prestazioni complessive. Questo approccio consente al nodo padre di continuare a gestire le richieste in arrivo senza interruzioni.

3.1 Gestione della ripubblicazione dei dati ogni `T` millisecondi tramite il thread `republisher`

Un nodo delega a un thread dedicato, chiamato `republisher`, il compito di ripubblicare periodicamente i dati di cui è responsabile ogni `T` millisecondi. Per svolgere questa funzione, il thread consulta la MyValuesTable contenente le coppie chiave/valore gestite dal nodo padre. Quando viene creata una nuova

coppia di dati, il nodo notifica il `republisher` tramite la funzione `add_pair/2`, che provvede ad aggiornare la ets con la nuova coppia.

Il thread, una volta istanziato tramite `thread:start`, esegue la funzione `republish_behaviour/6`. Quest'ultima implementa un ciclo infinito durante il quale il `republisher` verifica, come prima operazione, se il nodo padre ha generato nuove coppie di dati, utilizzando la funzione `check_for_new_pairs/2` (Listing 7). Per garantire un'esecuzione non bloccante viene impiegata una clausola `after`.

Listing 7: Funzione per verificare se ci sono nuove coppie chiave/valore di cui occuparsi

```
1 check_for_new_pairs(ValuesTable, RoutingTable, K,
      BucketSize) ->
2     receive
3         {new_pair, Key, Value} ->
4             ets:insert(ValuesTable, {Key,Value}),
5             node:distribute_value(Key, Value, RoutingTable,
                  K, BucketSize)
6     after 10 -> % This has to be 10 ms to avoid blocking
          the network
7         ok
8     end.
```

Successivamente, il thread verifica se sono trascorsi `T` millisecondi dall'ultima ripubblicazione dei dati. In caso affermativo, per ciascuna coppia chiave/valore presente nella mappa esegue la funzione `distribute_value/5`. Questa funzione individua all'interno della rete una lista di `BUCKET_SIZE` nodi più vicini all'identificatore della chiave tramite una serie di richieste `FIND_NODE`; a ciascuno nodo della lista viene quindi inviato un messaggio `store`, con l'istruzione di salvare la coppia all'interno della propria `valuesTable`.

3.2 Descrizione di `spare_node_manager`

Quando un nodo riceve una qualsiasi richiesta da un altro membro della rete, tenta di salvarlo nella propria routing table seguendo l'algoritmo descritto nella Sezione 1.3. Se però il `K_bucket` destinato al nuovo nodo è pieno, la sua gestione viene affidata al thread `spare_node_manager` attraverso la funzione `delegate/1`, in modo tale da migliorare l'efficienza della simulazione.

Il thread `spare_node_manager` utilizza il behaviour `gen_server`, che semplifica la gestione dello stato e delle richieste provenienti dal nodo padre. La funzione `delegate/1` consente al nodo padre di inviare al thread il messaggio con formato {`check, NodePid`}, che viene gestito in modo asincrono tramite il metodo `handle_cast` fornito da `gen_server`.

Una volta ricevuta la richiesta, lo `spare_node_manager` sa che il `K_BUCKET` di destinazione di `NodePid` è pieno, e provvede dunque a gestire la situazione. Come prima cosa invia un messaggio di `ping` al nodo in testa al `K_bucket` per verificarne l'attività. Se risponde, viene spostato in coda con una probabilità

dell'80%, scartando di conseguenza `NodePid`; nel restante 20% dei casi, invece, viene scartato lo stesso per fare spazio a `NodePid`. Questo comportamento si discosta leggermente da quanto descritto nella Sezione 1.3 e può sembrare controintuitivo: lo scopo è favorire l'inserimento dei nuovi nodi nella rete, aumentando la probabilità che vengano memorizzati nelle routing table degli altri membri. Se il nodo in testa alla lista non risponde viene seguita la procedura originale: esso viene rimosso e sostituito con `NodePid`. L'intera procedura è illustrata nel Listing 8.

Listing 8: Gestione del salvataggio di un nodo se il `K_bucket` di destinazione è pieno

```
handle_cast({check, NodePid}, State) ->
    [...]
    case node:ping(LeastRecentNodePid) of
        % If the least recent node is responsive, discard
            the new node with
        % a probability of 4/5 and add the new node with a
            probability
        % of 1/5.
        % This is used to increase the probability that a
            new node
        % is known by some node in the network.
        {pong, ok} ->
            RandomNumber = rand:uniform(5),
            if RandomNumber == 5 ->
                ?MODULE:append_node(RoutingTable, Tail,
                    NodeHashId, NodePid, BranchID);
            true ->
                ?MODULE:append_node(RoutingTable, Tail,
                    LeastRecentNodeHashId,
                    LeastRecentNodePid, BranchID)
            end;
        % If the least recent node is not responsive,
            discard it and add the new node.
        {pang, _} ->
            ?MODULE:append_node(RoutingTable, Tail,
                NodeHashId, NodePid, BranchID)
    end;
    [...]
.
```

Lo stato di `spare_node_manager` è dato da una tupla che include la routing table del nodo padre, il parametro `K` (che specifica il numero di bit che compongono ogni identificatore), e l'ultimo `K_bucket` aggiornato. Tenere traccia di quest'ultima informazione serve a prevenire cicli infiniti di richieste `ping`, e dunque `spare_node_manager` esegue la procedura solo se il nodo da salvare appartiene a un `K_bucket` diverso dall'ultimo aggiornato.

{`RoutingTable, K, LastUpdatedBranch`}

3.3 Descrizione di routing_table_filler e della fase di join

Il `routing_table_filler` è il thread responsabile di verificare che ogni `k_bucket` della routing table del nodo padre contenga almeno un elemento e, se necessario, avviare una procedura di ricerca per completare le informazioni mancanti. Inoltre gestisce la fase di join, che consente a un nodo di entrare in possesso di quanti più contatti possibile da salvare nella propria routing table. Il thread viene istanziato tramite la funzione `thread:start` e avvia la procedura di join, che ha inizio contattando un nodo di bootstrap noto. Per evitare problemi di sovraccarico causati dall'utilizzo di un unico nodo di bootstrap, esiste un insieme di nodi di questo tipo disponibili. Il primo nodo da contattare è quello più vicino al nodo entrante, determinato sulla base della distanza XOR; una volta individuato, viene aggiunto alla routing table del nuovo nodo.

La procedura di join consiste nell'invio di una serie di richieste di tipo `fill_my_routing_table` ai nodi di cui si viene a conoscenza progressivamente, a partire dal nodo di bootstrap. Questo processo va avanti fino a quando non si sono contattati tutti i nodi scoperti. Terminata la fase di join, il thread continua a cercare nuovi nodi, inviando ulteriori richieste `fill_my_routing_table` a partire da un altro nodo di bootstrap. La ricerca si conclude solo quando ogni `K_bucket` della routing table contiene almeno un elemento.

Quando la routing table contiene almeno un nodo di contatto per ogni "zona" della rete, il `routing_table_filler` si mette in attesa di eventuali messaggi dal formato `deleted_node` inviati dal nodo padre. Questi messaggi servono a notificare al thread che un nodo è stato rimosso dalla routing table, probabilmente perché non ha risposto a una richiesta, e che quindi un `K_bucket` della tabella potrebbe essere rimasto senza nodi di contatto. In risposta, il `routing_table_filler` controlla la presenza di `K_bucket` vuoti; se ne individua, avvia nuovamente la procedura di ricerca per colmare le informazioni mancanti, partendo dall'interrogare nuovamente un nodo di bootstrap randomico. Tale procedimento è illustrato nel Listing 9.

Listing 9: Riavvio della procedura di ricerca di nodi in caso di informazioni mancanti

```
1  % This function checks if a node has been deleted from the
       network and if so it starts the
       check_for_empty_branches function to check if there are
        empty branches in the routing table.
2  check_deletion_message(RoutingTable, K, BucketSize) ->
3      receive
4          {deleted_node} ->
5              ?MODULE:check_for_empty_branches(RoutingTable,
                   K, BucketSize)
6
7      after 1000 ->
8          ok
9      end,
10
```

```
11      ?MODULE:check_deletion_message(RoutingTable, K,
            BucketSize).
12
13  % This function checks if there are empty branches in the
        routing table and if so it starts the research of new
        nodes again.
14  check_for_empty_branches(RoutingTable, K, BucketSize) ->
15      EmptyBranches = utils:empty_branches(RoutingTable, K),
16      if EmptyBranches ->
17          ?MODULE:fill_routing_table(RoutingTable, K,
                BucketSize);
18      true -> ok
19      end.
```

Quando un nodo riceve una richiesta del tipo `fill_my_routing_table`, risponde fornendo una selezione di nodi dalla propria routing table. Questi nodi possono essere salvati dal `routing_table_filler` nella routing table del nodo padre e successivamente contattati, qualora vi siano ancora `K_bucket` da riempire.
La selezione dei nodi da restituire segue una logica precisa. Il nodo che riceve la richiesta condivide un certo numero di `K_bucket` con il nodo che la invia. Ad esempio, supponiamo che il primo abbia l'identificatore `1001` e il secondo `1010`. In questo caso, condivideranno due `K_bucket`: quello dei nodi con identificatori che iniziano con `0` e quello con identificatori che iniziano con `11`. Vengono quindi restituiti tutti i nodi presenti in questi `K_bucket`, consentendo così ad un nodo in fase di join di popolare rapidamente la propria routing table. Inoltre, quanto più i due condividono un prefisso lungo nei loro identificatori, tanto più veloce risulta tale operazione. Per questo motivo, inizialmente si sceglie di contattare il bootstrap più vicino.
Oltre ai nodi presenti nei `K_bucket` comuni, vengono inviati anche alcuni che non appartengono a questi ultimi. Questo approccio permette al nodo entrante di ampliare la conoscenza della rete, ottenendo informazioni su altri membri e facilitando ulteriormente il riempimento di tutti i `K_bucket`.

Una volta che un nodo entrante riceve una risposta alla richiesta `fill_my_routing_table`, tenta di aggiungere i nodi ottenuti nella propria routing table. Prima di procedere, però, verifica la loro attività inviando un messaggio di Ping. Questo approccio ha due vantaggi: da un lato, evita di inserire nodi inattivi nella routing table, dall'altro consente al nodo entrante di farsi conoscere dagli altri membri della rete.

La procedura di riempimento della routing table è progettata per essere altamente efficiente. Per evitare inutili duplicazioni, i nodi già contattati vengono esclusi dalle richieste successive. Inoltre, quando si invia una richiesta `fill_my_routing_table`, al destinatario vengono comunicati i `K_bucket` già riempiti, così da escluderli dalla ricerca e ottimizzare ulteriormente il processo.

4 Gestione dell'Interfaccia di Comunicazione di un Nodo

Ogni nodo è progettato per comunicare con altri nodi, sia inviando richieste che ricevendole. Lato server, le richieste ricevute vengono gestite attraverso le funzioni `handle_call/3` e `handle_cast/2` fornite da `gen_server`. La funzione `handle_call/3` si occupa delle richieste sincrone, mentre `handle_cast/2` gestisce quelle asincrone; entrambe le funzioni accettano come parametro `Request`, che specifica il tipo di richiesta da elaborare.

4.1 Salvataggio di un nodo nella propria routing table

Come accennato in precedenza, ogni volta che un nodo riceve una richiesta, sia essa sincrona o asincrona, tenta di salvare il nodo mittente nella propria routing table utilizzando la funzione `save_node/4`. Questa funzione viene chiamata all'interno delle funzioni `handle_call/3` (Listing 10) e `handle_cast/2` (Listing 11) di `gen_server`. In questo modo la routing table viene mantenuta costantemente aggiornata, includendo nodi attivi.
Dopodiché, ogni richiesta viene gestita nella clausola appropriata della funzione `request_handler/3` per le richieste sincrone o `async_request_handler/2` per quelle asincrone. La gestione separata in queste funzioni consente di evitare la ripetizione della chiamata alla funzione `save_node/1` per ogni tipo di messaggio, mantenendo il codice più pulito e facilmente mantenibile.
È importante notare che il `SenderPid` deve essere esplicitamente incluso nel messaggio invece di essere preso automaticamente dal secondo parametro delle funzioni di handling. Questo è necessario perché la richiesta potrebbe provenire da un thread, che comunica per conto del processo padre e utilizza il suo PID invece del proprio.

Listing 10: Gestione di richieste sincrone

```
handle_call({Request, SenderPid}, _, State) ->
    % Save the sender node in the routing table.
    utils:debug_print("Handling ~p ~p~n", [Request,
        SenderPid]),
    {RoutingTable, _, K, _, BucketSize,_} = State,
    ?MODULE:save_node(SenderPid,RoutingTable,K,BucketSize),
    ?MODULE:request_handler(Request, SenderPid, State).
```

Listing 11: Gestione di richieste asincrone

```
handle_cast({Request, SenderPid}, State) when is_tuple(
    Request) ->
    utils:debug_print("Handling ~p", [Request]),
    {RoutingTable, _, K, _, BucketSize,_} = State,
    ?MODULE:save_node(SenderPid,RoutingTable,K,BucketSize),
    ?MODULE:async_request_handler(Request, State).
```

L'algoritmo per salvare un nodo, implementato nella funzione `save_node/4`, segue la stessa logica descritta nella Sezione 1.3, con alcune leggere modifiche già descritte nella Sezione 3.2. Per prima cosa viene calcolato l'identificatore del nodo da salvare, così da individuare il `K_bucket` di destinazione all'interno della routing table. Se il `K_bucket` esiste ma è già pieno, la gestione del nodo viene affidata al thread `spare_node_manager` tramite la funzione `delegate/1`. In caso contrario, il nodo viene salvato direttamente nel `K_bucket`, essendoci ancora spazio disponibile. Il Listing 12 mostra un piccolo estratto di questa procedura.

Listing 12: Controllo per verificare se il `K-bucket` di destinazione è pieno

```
save_node(NodePid, RoutingTable, K, K_Bucket_Size) when
    is_pid(NodePid) ->
    [...]
     % Check if the list is full or not.
    case length(NodeList) < K_Bucket_Size of
        % If the list is not full, add the new node to the
            tail.
        true ->
            NewNodeList = NodeList ++ [{NodeHashId, NodePid
                }],
            ets:insert(RoutingTable, {BranchID, NewNodeList
                });
        % If the list is full, check the last node in the
            list.
        false ->
            % Delegating spare node to the
                spare_node_manager
            spare_node_manager:delegate(NodePid)
    end
    [...]
.
```

4.2 Richieste Sincrone

Le richieste sincrone gestite da un nodo sono principalmente tre: `FIND_NODE`, `FIND_VALUE` e `PING`. Quando un nodo invia una di queste richieste, deve necessariamente attendere una risposta. Se il nodo destinatario della richiesta non risponde, il mittente lo rimuove dalla propria routing table attraverso la funzione `delete_node`(Listing 13). Si osservi che il `routing_table_filler` viene avvisato dell'eliminazione del nodo, permettendogli di riavviare, se necessario, la procedura di ricerca di nuovi nodi (per maggiori dettagli, si rimanda alla Sezione 3.3).

Listing 13: Rimozione di un nodo dalla propria routing table

```
% This function delete a node from the routing table
```

```
delete_node(NodePid, RoutingTable, K) when is_pid(NodePid)
    ->
    NodeHashId = utils:k_hash(NodePid, K),
    BranchID = utils:get_subtree_index(NodeHashId, com:
        my_hash_id(K)),
    case ets:lookup(RoutingTable, BranchID) of
        [{BranchID, NodeList}] ->
            NewNodeList = lists:filter(fun({_, Pid}) -> Pid
                =/= NodePid end, NodeList),

            LenBefore = length(NodeList),
            LenAfter = length(NewNodeList),
            if LenBefore /= LenAfter ->
                ets:insert(RoutingTable, {BranchID,
                    NewNodeList}),
                join_thread:deleted_node();
            true -> ok
            end;
        [] -> ok
    end.
```

Di seguito è presentata una breve descrizione del modo in cui un nodo gestisce le tre richieste sincrone appena menzionate.

4.2.1 FIND_NODE

Come descritto nella Sezione 4.1, le richieste sincrone vengono elaborate lato server dalla funzione `request_handler`; essa contiene diverse clausole, una per ciascun formato di messaggio. Una richiesta `FIND_NODE` ha il formato {`find_node`, `HashID`}, dove `HashID` rappresenta l'identificatore del nodo o della chiave che si desidera trovare.
La ricerca effettiva dei nodi viene effettuata dalla funzione `find_node/4`, che esamina la tabella di routing del nodo che ha ricevuto la richiesta e restituisce la lista dei `BUCKET_SIZE` nodi più vicini a `HashID` secondo il concetto di distanza XOR. Tale lista è ordinata in base alla distanza da `HashID`.
La ricerca inizia dal `K_bucket` a cui appartiene `HashID`, che contiene i nodi a lui più vicini. Se non vengono trovati abbastanza nodi in quel `K_bucket`, vengono esplorati i `K_bucket` più lontani, e la ricerca continua fino a quando non viene esaminata tutta la tabella di routing o non viene trovato un numero sufficiente di nodi (cioè `BUCKET_SIZE`). Il Listing 14 riporta il codice che gestisce tale richiesta, dove l'effettiva ricerca viene effettuata dalla funzione `find_node/4`.

Listing 14: Gestione di `FIND_NODE`

```
request_handler({find_node, HashID}, _From, State) ->
    {RoutingTable, _, K, _, Bucket_Size,_} = State,
    % Look up the closest K nodes within the routing table.
    NodeList = ?MODULE:find_node(RoutingTable, Bucket_Size,
        K, HashID),
```

```
5       % Reply to the caller with the list of closest nodes
            and the current state.
6       {reply, {ok, NodeList}, State};
```

Lato client, quando un nodo necessita di ottenere una lista di `BUCKET_SIZE` nodi vicini a un determinato `HashID`, svolge una ricerca nell'intera rete tramite la funzione `find_k_nearest_node(RoutingTable, HashID, BucketSize, K)`. Come prima cosa, il client esamina la propria tabella di routing e raccoglie `BUCKET_SIZE` nodi di cui è gia a conoscenza, sfruttando la funzione `find_node/4`. Dopodiché, procede a contattarli ricorsivamente attraverso una richiesta `FIND_NODE`, aggiornando e affinando la lista finale sulla base delle nuove informazioni ricevute, fino a ottenere i nodi più pertinenti.
Ogni nodo contattato fornisce una lista dei nodi più vicini a `HashID`, basandosi sulle informazioni presenti nella propria routing table. Questa lista viene combinata con quella già raccolta, escludendo però i nodi già contattati per evitare duplicazioni. Quando non ci sono più nodi da interrogare, la lista finale viene ordinata in base alla distanza da `HashID`, e vengono selezionati i `BUCKET_SIZE` nodi più vicini all'identificatore target.

4.2.2 `FIND_VALUE`

Questa richiesta viene gestita dalla funzione `request_handler` quando un nodo riceve un messaggio nel formato {`find_value, Key`}, dove `Key` rappresenta una chiave nella hash table distribuita in formato stringa. Ad esempio, `Key` potrebbe essere il nome di un file, come spiegato nella Sezione 2.4.

Il comportamento di `FIND_VALUE` è simile a quello di `FIND_NODE`, con l'unica differenza che, prima di eseguire la ricerca, il nodo verifica se il valore cercato è presente nella sua `ValuesTable`. Se il valore lo possiede, lo restituisce direttamente a chi ha fatto la richiesta; se invece il valore non è presente, cerca nella propria routing table i `BUCKET_SIZE` nodi più vicini al valore richiesto tramite la funzione `find_node/4`. In questo caso, la ricerca viene effettuata calcolando l'identificatore della chiave.

Listing 15: Gestione di `FIND_VALUE`

```
1  request_handler({find_value, Key}, _From, State) ->
2      {RoutingTable, ValuesTable, K, _, Bucket_Size,_} =
           State,
3
4      case ?MODULE:get_value(Key, K, ValuesTable) of
5          {ok, Key, Value} ->
6              % If the key is found in the local values table
                   , return the value.
7              {reply, {ok, Key, Value}, State};
8          {no_value, empty} ->
9              % Compute the hash ID of the key to search for.
10             KeyHashId = utils:k_hash(Key, K),
11             % If the key is not found, look up the closest
                   K nodes to the key's hash ID.
```

```
12              NodeList = ?MODULE:find_node(RoutingTable,
                    Bucket_Size, K, KeyHashId),
13              % Reply with the list of closest nodes to the
                    key.
14              {reply, {nodes_list, NodeList}, State}
15      end;
```

Lato client, un nodo invia una richiesta `FIND_VALUE` quando desidera ottenere il valore associato a una specifica chiave. Per questo scopo, la funzione `lookup_for_value/5` consente al nodo client di interrogare la rete, inviando una serie di richieste `FIND_VALUE` ai nodi di cui viene progressivamente a conoscenza fino a quando non trova qualcuno che possiede quel valore.

Ad ogni chiamata ricorsiva di `lookup_for_value/5`, il nodo contattato restituisce una lista di nodi vicini al valore cercato se non lo possiede. Tra questi, viene scelto e contattato il nodo più vicino. Questo processo si ripete, consentendo di avvicinarsi progressivamente all'identificatore della chiave target ad ogni iterazione. Il procedimento si conclude quando un nodo contattato possiede il valore richiesto oppure quando tutti i nodi di cui si è venuti a conoscenza durante la procedura sono stati contattati senza successo.

4.2.3 `PING`

Questo messaggio viene utilizzato per verificare se un nodo è ancora attivo. Un nodo conferma la sua attività rispondendo con un semplice `pong`, come illustrato nel Listing 16.

Listing 16: Gestione di `PING` lato server

```
1  % Handles the ping message sent to a node.
2  request_handler(ping, _, State) ->
3      % Reply with pong to indicate that the node is alive
           and reachable.
4      {reply, {pong, ok}, State};
```

Lato client, il nodo può inviare un messaggio di ping tramite la funzione `ping/1`, e determina lo stato del nodo destinatario in base alla risposta ricevuta: se riceve un `pong`, il nodo destinatario è attivo; se invece riceve un `pang`, il nodo si considerà non più attivo (Listing 17).

Listing 17: Gestione di `PING` lato client

```
1  % Pings a specific node (NodePid) to check its availability
2  ping(NodePid) when is_pid(NodePid) ->
3      case com:send_request(NodePid, ping) of
4          {pong,ok} ->
5              {pong,ok};
6          % If the node is unreachable, the function returns
               pang.
7          {error, Reason} ->
8              {pang, Reason}
```

```
    end.
```

4.3 Richieste Asincrone : STORE

L'unica richiesta asincrona rilevante è STORE, che consente a un nodo di "ordinare" a un altro nodo di salvare una coppia chiave/valore nella propria ValuesTable. Poiché si tratta di una richiesta asincrona non è prevista alcuna risposta. Il Listing 18 mostra il codice eseguito da un nodo quando riceve un messaggio STORE.

Listing 18: Gestione di STORE lato server

```
% A node store a key/value pair in its own values table.
% The node also saves the sender node in its routing table.
async_request_handler({store, Key, Value}, State) ->
    {_, ValuesTable, K, _, _,_} = State,
    KeyHash = utils:k_hash(Key, K),
    case ets:lookup(ValuesTable, KeyHash) of
        [] ->
            ets:insert(ValuesTable, {KeyHash, #{Key =>
                Value}});
        [{_,ValuesMap}] ->
            KeyIsKey = maps:is_key(Key, ValuesMap),
            if not KeyIsKey ->
                NewValuesMap = maps:put(Key, Value,
                    ValuesMap),
                ets:insert(ValuesTable, {KeyHash,
                    NewValuesMap});
            true ->
                ok
            end
    end,
    analytics_collector:stored_value(Key),
    {noreply, State};
```

Lato client, quando un nodo desidera distribuire una coppia chiave/valore, utilizza la funzione distribuite_value/5. Questa funzione consente innanzitutto di individuare nella rete una lista di BUCKET_SIZE nodi vicini all'identificatore della chiave. Successivamente, a ciascuno di questi nodi viene inviata una richiesta di STORE. Questa funzione è la stessa utilizzata dal thread republisher per ripubblicare i dati ogni T millisecondi (si veda la Sezione 3.1). Il codice è illustrato nel Listing 19.

Listing 19: Gestione di STORE lato client

```
% Client-side function to store a key/value pair in the K
    nodes closest to the hash_id of the pair.
distribute_value(Key, Value, RoutingTable, K, Bucket_Size)
    ->
```

```
    KeyHashId = utils:k_hash(Key, K),
    NodeList = ?MODULE:find_k_nearest_node(RoutingTable,
        KeyHashId, Bucket_Size, K),
    lists:foreach(
        fun({_NodeHashId, NodePid}) ->
            com:send_async_request(NodePid, {store, Key,
                Value})
        end,
        NodeList
    ).
```

5 Calcolo delle Statistiche e Risultati Ottenuti

In questa sezione finale viene illustrato l'`analytics_collector`, un sistema basato su eventi progettato per calcolare statistiche sulle attività che avvengono all'interno della rete. Inoltre, vengono presentati i dettagli dell'esecuzione della simulazione e i risultati ottenuti.

5.1 Descrizione di `analytics_collector`

L'`analytics_collector` è un sistema basato su eventi progettato per raccogliere informazioni sulle attività svolte dai nodi e calcolare statistiche su tali operazioni, come ad esempio il tempo medio di join nella rete. Il sistema opera come un server implementato utilizzando il comportamento `gen_server` e gestisce una tabella `ETS` di tipo `set` denominata `analytics`. Questa tabella viene utilizzata per memorizzare informazioni sui vari eventi che si verificano all'interno della rete. La chiave di `analytics` è un atomo `EventType` che rappresenta il tipo di evento, mentre il valore associato è una lista di tuple, ognuna delle quali descrive un'istanza di quel tipo di evento. Ogni tupla segue il formato {`Pid, EventValue, Millis`}, dove `Pid` è il processo (o nodo) associato all'evento, `EventValue` rappresenta un valore che identifica o descrive l'evento specifico, e `Millis` indica il momento, espresso in millisecondi, in cui l'evento è stato gestito.
Il valore `EventValue` può assumere diverse forme. Può essere un identificatore univoco (`EventId`) utilizzato per associare un evento di inizio a un evento di fine di una procedura (ad esempio, l'inizio e la fine della procedura di riempimento della routing table). In questo caso, entrambi gli eventi condivideranno lo stesso `EventValue`, che sarà un numero intero univoco. Può anche essere una tupla, ad esempio {`ValueId, Key`}, per descrivere eventi relativi al salvataggio di una coppia chiave/valore nella routing table di un nodo (attraverso la procedura di `store`).

Lo stato del server è dato da `ListenersMap`, una mappa Erlang in cui la chiave è nuovamente un tipo di evento `EventType`, mentre il valore associato è la lista di PID dei nodi che devono essere notificati quando si verifica un'istanza di quell'evento. Il Listing 20 mostra l'inizializzazione di `analytics_collector`.

Listing 20: Inizializzazione di `analytics_collector`

```
% This function is called to initialize the gen_server.
% It registers the analytics_collector Pid and creates the
    analytics ets to
% collect data.
init([K, T, ListenerPid]) ->
  register(analytics_collector, self()),
  ets:new(analytics, [set, public, named_table]),
  ListenersMap = #{},
  ?MODULE:notify_server_is_running(ListenerPid),
  {ok, {K, T, ListenersMap}}.
```

Nella simulazione vengono monitorati diversi tipi di eventi `EventType`. Tra questi, vi sono gli eventi temporali, nei quali ogni evento di inizio di una procedura è sempre associato a un corrispondente evento di fine. A tale scopo, si utilizzano le funzioni `started_time_based_event/1` e `finished_time_based_event/2`, che prendono in input il tipo di evento (`EventType`). Ad esempio, gli atomi `started_filling_routing_table` e `finished_filling_routing_table` rappresentano rispettivamente l'inizio e la conclusione della procedura di riempimento della routing table di un nodo da parte del thread `routing_table_filler`. Questi eventi vengono registrati mediante le funzioni apposite, come mostrato nel Listing 21.

Listing 21: Gestione degli eventi relativi alla procedura di riempimento della routing table

```
started_filling_routing_table() ->
  ?MODULE:started_time_based_event(
      started_filling_routing_table).

% the filling_routing_table procedure
finished_filling_routing_table(EventId) ->
  ?MODULE:finished_time_based_event(
      finished_filling_routing_table, EventId).
```

Tali funzioni vengono collocate nel codice rispettivamente prima e dopo l'esecuzione di una procedura, come mostrato nel Listing 22, che illustra il caso della procedura di riempimento della routing table. In questo esempio, l'evento di inizio viene registrato poco prima dell'avvio della procedura, mentre l'evento di fine viene notificato subito dopo la sua conclusione.

Listing 22: Notifica di inizio e fine della procedura di riempimento della routing table

```
[...]
EventId = analytics_collector:started_filling_routing_table
    (),
?MODULE:fill_routing_table(RoutingTable, K, BucketSize),
analytics_collector:finished_filling_routing_table(EventId)
    ,
```

```
[...]
```

Tra gli altri eventi temporali monitorati vi sono quelli relativi all'inizio e alla fine della procedura di join, eseguita sempe dal thread `routing_table_filler` (vedi Sezione 3.3). Vengono inoltre tracciati gli eventi di inizio e fine della procedura di lookup per la ricerca di un valore, così come quelli relativi alla procedura di ripubblicazione dei dati (vedi Sezione 3.1).

Quando un evento temporale viene salvato nella tabella `analytics`, viene registrato anche il timestamp in millisecondi, consentendo il calcolo di statistiche temporali. L'aggiunta di un evento nella tabella `analytics` avviene tramite la funzione `add/2`. Inoltre, la funzione `started_time_based_event/1` genera un identificativo univoco (`EventId`), che serve per collegare l'evento di inizio alla corrispondente istanza di fine (vedi Listing 23).

Listing 23: Funzioni di notifica di un nuovo evento

```
started_time_based_event(EventType) ->
  EventId = erlang:unique_integer([positive]),
  CurrentTime = erlang:monotonic_time(millisecond),
  ?MODULE:add(EventType, {EventId,CurrentTime}),
  EventId.

finished_time_based_event(EventType, EventId) ->
  CurrentTime = erlang:monotonic_time(millisecond),
  ?MODULE:add(EventType, {EventId,CurrentTime}).
```

Quando un nuovo evento e le relative informazioni vengono aggiunti in `analytics`, tutti i processi interessati a quel tipo di evento vengono notificati tramite la funzione `notify_listeners/3`. Questa funzione invia a ciascun processo un messaggio nel formato {`event_notification, EventType, EventValue`}.

L'`analytic_collector` mette a disposizione diverse funzioni utili per calcolare statistiche o ottenere altre informazioni utili. Ad esempio, la funzione `join_procedure_mean_time/0` consente di calcolare il tempo medio di join dei nodi. Allo stesso modo, funzioni come `distribute_mean_time/0`, `lookup_mean_time/0` e `filling_routing_table_mean_time/0` permettono di ottenere il tempo medio rispettivamente per le procedure di ripubblicazione, di lookup e di riempimento della routing table.
Oltre alle statistiche temporali, è possibile recuperare ulteriori informazioni utili. Ad esempio, la funzione `get_node_that_stored(Key)` restituisce la lista dei nodi che hanno memorizzato la chiave `Key`.

Infine, `analytics_collector` monitora anche gli eventi legati alla registrazione dei nodi, distinguendo tra nodi normali e nodi bootstrap, all'interno della tabella `analytics`. La registrazione di un nodo come bootstrap o come nodo normale avviene tramite le funzioni dedicate mostrate nel Listing 24.

Listing 24: Registrazione di nodi normali e di bootstrap

```
enroll_bootstrap() ->
  ?MODULE:add(bootstrap, 1).
```

```
3
4  enroll_node()->
5    ?MODULE:add(node, 1).
```

Questo meccanismo consente di ottenere la lista dei nodi bootstrap tramite la funzione `get_bootstrap_list/0`. Tale funzione risulta particolarmente utile, ad esempio, durante la fase di join o quando il thread `routing_table_filler` deve contattare nodi per riempire i `K_bucket` vuoti (vedi Sezione 3.3).

5.2 Testing ed esecuzione della rete Kademlia

Per testare una rete Kademlia e valutarne le prestazioni bisogna utilizzare la funzione `starter:start()`. Questa operazione apre una schermata, mostrata nella Figura 2, in cui l'utente può scegliere tra quattro possibili test. Per selezionare un test, basta digitare il numero corrispondente a quello desiderato. Nei test numero (2), (3) e (4), l'utente ha la possibilità di simulare la rete con

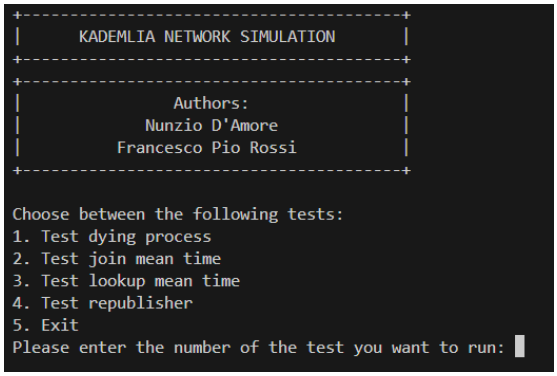

Figure 2: Scelta del test da eseguire

diverse configurazioni di nodi normali e nodi di bootstrap. In particolare, può scegliere tra reti di dimensione piccola, media o grande, come mostrato nella Figura 3. Ogni test riporta una serie di statistiche raccolte grazie all'ausilio di `analytic_collector`. Ecco cosa fanno i quattro test:

1. Il primo test mostra come, quando un nodo muore e di conseguenza non risponde a una richiesta, viene automaticamente rimosso dalla routing table del nodo client.

2. Il secondo test calcola il tempo medio necessario affinché i nuovi nodi si uniscano alla rete, fornendo poi le relative statistiche.

3. Il terzo test, invece, consente di calcolare il tempo medio di lookup per la ricerca di un valore all'interno della rete prima e dopo la chiusura di n nodi.

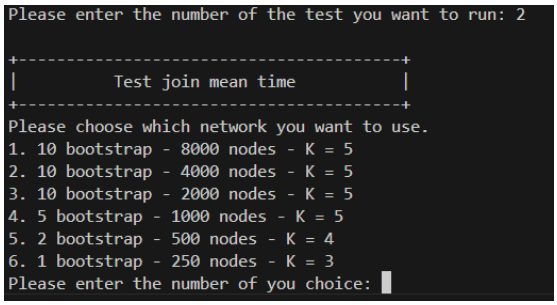

Figure 3: Scelta della tipologia di rete

4. Il quarto e ultimo test verifica il comportamento della rete quando un nodo ripubblica una propria coppia chiave/valore.

Ad esempio, il test di join inizia avviando la rete con un determinato numero di nodi. Successivamente, vengono aggiunti 250 nuovi nodi, e si misura il tempo medio necessario affinché questi si uniscano alla rete già esistente. È importante notare che la rete si considera convergente quando tutti i nodi iniziali hanno popolato tutti i propri `k_bucket` con almeno un elemento. Le statistiche relative a questo test sono riportate nella Figura 4.

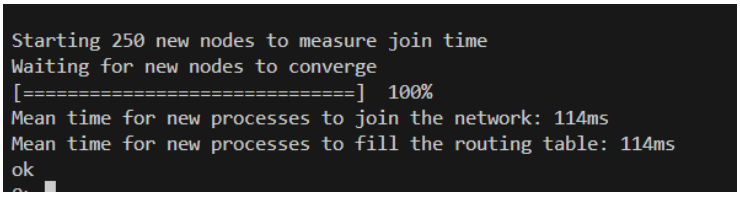

Figure 4: Statistiche relative al test di join

La simulazione è configurata tramite quattro parametri principali: il numero iniziale di nodi di bootstrap, il numero inziale di nodi standard per l'avvio della rete, la lunghezza (in bit) degli identificatori, e l'intervallo di tempo (in millisecondi) per la ripubblicazione dei dati. L'avvio della simulazione avviene attraverso la funzione `start_simulation/4`, che attende il raggiungimento della convergenza della rete, ovvero il momento in cui i nodi iniziali hanno completato il riempimento delle proprie tabelle di routing. Successivamente, è possibile aumentare la dimensione della rete utilizzando la funzione `start_new_nodes/4`, che consente di specificare il numero di nodi di bootstrap e standard da aggiungere.

Quando la rete è pronta e ha completato la convergenza, è possibile interagirvi tramite alcuni comandi da shell definiti nel modulo `node` che consentono di controllare le operazioni dei nodi. Di seguito sono elencati i principali comandi disponibili:

- **Visualizzazione della tabella di routing**: Utilizzando la funzione

`get_routing_table(NodePid)` è possibile ottenere la tabella di routing di `NodePid`.

- **Distribuzione di una coppia chiave/valore**: La funzione `distribute( NodePid, Key, Value)` ordina a `NodePid` di inviare un messaggio di store ai nodi più vicini a `Key`, richiedendo loro di memorizzare la coppia chiave/valore specificata. Si ricorda che `Key` è una stringa.
- **Ricerca di un valore**: Con `lookup(NodePid, Key)`, si avvia la procedura di ricerca per il valore associato a `Key` da parte di `NodePid`, che contatterà inizialmente i nodi nella sua routing table.
- **Invio di un messaggio di ping**:
  - La funzione `send_ping(NodePid, ToNodePid)` consente di ordinare a `NodePid` di inviare un messaggio di ping al nodo `ToNodePid`.
  - In alternativa, con `ping(NodePid)` è possibile inviare un messaggio di ping direttamente al nodo `NodePid` per verificarne lo stato di attività.
- **Terminazione di un nodo**: La funzione `kill(Pid)` permette di terminare il nodo identificato da `Pid`.
- **Ricerca dei nodi più vicini**: Utilizzando la funzione `shell_find_nearest_nodes(NodePid, ValueToFind)`, si ordina a `NodePid` di individuare i nodi più vicini a `ValueToFind` all'interno della rete, che potrebbe essere sia il pid di un processo che una chiave.

Volendo si può inizializzare la rete direttamente con la funzione `start_new_nodes/4`. In questo modo non si attende la convergenza (che potrebbe non arrivare) per iniziare a testare la rete. Per venire a conoscenza dei Pid di alcuni nodi con cui interagire è possibile utilizzare la funzione `analytics_collector:get_node_list()`.

Per concludere, vengono presentati tre grafici che illustrano il tempo di convergenza (Figura 5), il tempo medio di join (Figura 6) e il tempo medio di riempimento della routing table (Figura 7), in funzione del numero di nodi. Queste statistiche si riferiscono ai nodi iniziali che vengono utilizzati per avviare la rete.

Nella Figura 8 è illustrato un confronto tra il tempo medio di join e il tempo medio necessario per riempire la routing table di 250 nodi che si aggiungono a una rete già attiva. La rete considerata ha nodi iniziali (rappresentati sull'asse delle x) che hanno già raggiunto la convergenza. Si osserva che i due tempi sono molto simili, indicando che la procedura di join consente di acquisire rapidamente informazioni su molti nodi, grazie al contatto con il bootstrap più vicino.

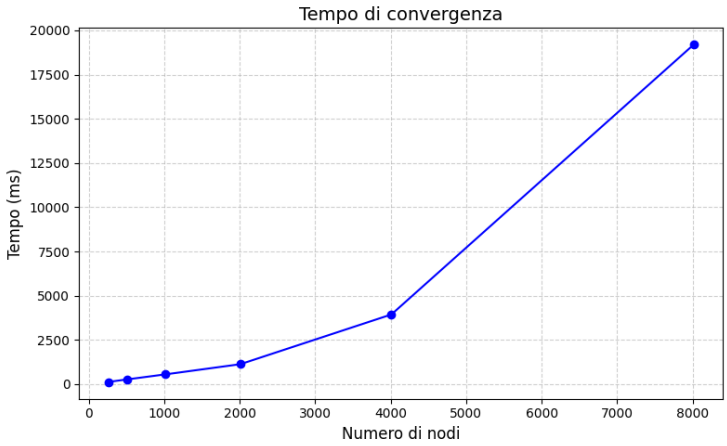

Figure 5: Tempo di convergenza della rete

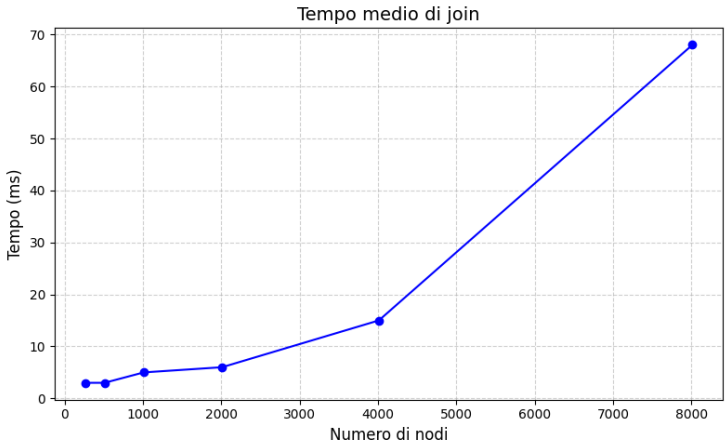

Figure 6: Tempo medio di join

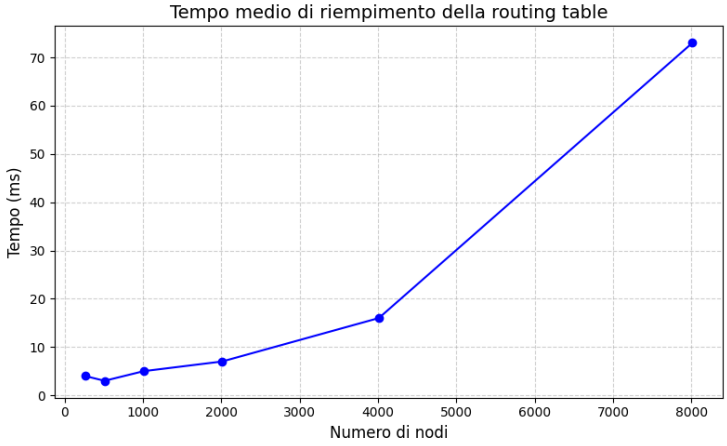

Figure 7: Tempo medio di riempimento della routing table

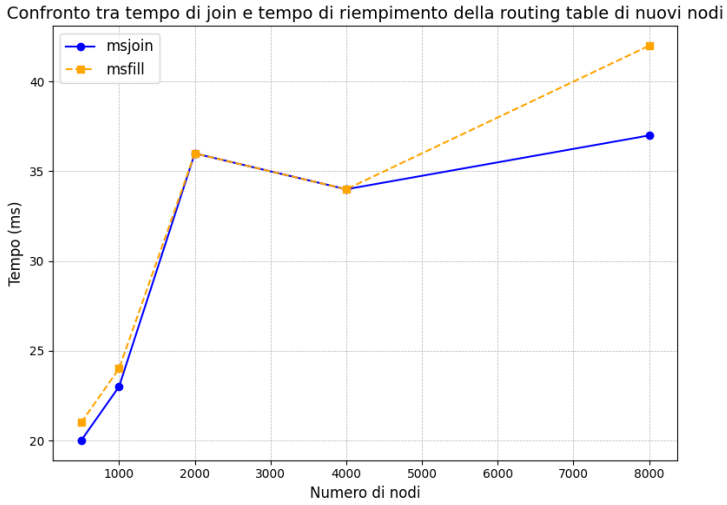

Figure 8: Confronto tra tempo di join e tempo di riempimento della routing table dei nuovi nodi nella rete