Università di Pisa

Informatica Applicata

Progetto di Reti e Calcolatori

# Reduced Dynamo

*Autore:*
Luca Lovagnini

A.A. 2012/2013

# Contents

# Chapter 1

# Architettura e Sturtture Dati

**Introduzione e Versione del Progetto**

In questo capitolo verranno trattate le entità principali del Progetto, descrivendone il ruolo all'interno del sistema e le loro caratteristiche.

Si faccia notare che alcune di queste entità, rispetto alle specifiche inizialmente fornite per il progetto, sono state introdotte dall'autore, per renderne più chiaro lo scopo in questo e nei capitoli successivi.

Le entità sono distinguibili in due categorie:

- Entità Principali: contraddistinte da una sezione dedicata, dalla loro complessità e dall'importanza dei compiti da essi svolte.
- Subentità: entità create/gestite da Entità Principali, svolgono un ruolo di monitoraggio e/o di controllo, oppure rappresentano un sottoinsieme di Entità Principali. Esse verranno trattate nel corso del capitolo all'interno di altre Entità Principali (generalmente quelle che le gestiscono).

Infine, si ricorda che questo Progetto riguarda l'implementazione di *Reduced Dynamo* nella versione dove *non* è contemplata l'operazione di rimozione di nodi dalla *DHT*.

## 1.1 Principali Entità

### 1.1.1 Bootstrap Server

È l'entità che rappresenta il Server di Sistema. Si occupa essenzialmente di 3 compiti:

1. Ricevere in ingresso la richiesta di un nuovo Storage Node che desidera essere inserito all'interno della *DHT*. Il Bootstrap Server, in seguito a tale richiesta, decide quale sarà il Nodo di Bootstrap a cui il nuovo Storage Node dovrà rivolgersi (tramite la storageNodeList).

2. Ricevere in ingresso la richiesta di un Client che vuole effettuare la ricerca di un particolare dato all'interno della *DHT* (o più precisamente, del suo identificatore). Come nel caso del nuovo Storage Node, il Boostrap Server si occupa di decidere quale Bootstrap Node dovrà occuparsi della richiesta del Cient in questione.

3. Decreta quali Storage Node all'interno della *DHT* sono dei Bootstrap Node, ed aggiorna la storageNodeList di conseguenza.

#### Log Manager

Il Log Manager è una subentità creata e gestita dal Bootstrap Server. Ha il compito di registrare tutti gli Storage Node che hanno completato la loro operazione di inerimento all'interno della *DHT* (e le relative informazioni che li caratterizzano).

Secondo l'implementazione svolta, la registrazione di tali eventi viene effettuata su un file di testo.

#### Keep Alive Controller

La subentità in questione (da ora chiamata Controller) è stata introdotta dallo sviluppatore del progetto in seguito alla necessità di un processo che si dedicasse esclusivamente al relativo scopo.

Infatti, a causa di possibili malfunzionamenti all'interno della rete dove risiede il Sistema, oppure a problemi derivata dall'uso di certi protocolli (ad esempio l'UDP), un Nodo di Bootstrap potrebbe non essere più raggiungibile (sia dal Bootstrap Server che da qualsiasi altra entità nel Sistema). Il Controller si occupa quindi di gestire i messaggi di Keep Alive inviati dai Bootstrap Node e, nel caso essi non vengano ricevuti, comuniarlo al Bootstrap Server (che di conseguenza ne sceglierà di nuovi).

### 1.1.2 Storage Node

Si tratta dell'entità che detiene i dati presenti nel Sistema. Inizialmente segue una procedura per il proprio inserimento all'interno della *DHT*, il quale coinvolge il Bootstrap Server, uno Bootstrap Node ed un altro Storage Node. Dopo aver

effettuato con successo il proprio inserimento nella struttura ad anello, si occupa di svolgere i seguenti compiti:

1. In seguito ad un nuovo nodo che diviene predecessore dello Storage Node nell'anello del Sistema, lo Storage Node in questione ha il compito di fornire al nuovo nodo i propri dati che divengono di sua competenza. Ovviamente, è possibile che lo Storage Node in questione non detenga alcun dato (e di conseguenza neppure il suo nuovo predecessore), come è anche possibile che il nuovo predecessore non riceva alcun dato (perchè non di sua competenza).

2. La ricerca da parte di un cliente viene gestita dallo Storage Node che detiene il dato cercato all'interno della propria dataTable. Tuttavia, lo Storage Node detentore del dato non riceve in ingresso una richiesta direttamente da parte del Client, bensì dal Bootstrap Node a cui il Client si è rivolto. Invece, la risposta da parte dello Storage Node detentore del dato cercato, avviene direttamente verso il Client in questione.

#### Bootstrap Node

Malgrado sia stata classificata come subentità, l'insieme dei Bootstrap Node ricorprono un ruolo fondamentale in *Reduced Dynamo*: gestiscono in maniera diretta le richieste che vengono fatte al Bootstrap Server , permettendo così di distribuire il carico di lavoro tra i vari Bootstrap Node. In particolare, essi si occupano di:

1. Ricevere le comunicazioni da parte di un nuovo Storage Node, decretare quale nodo nell'anello ricoprirà il ruolo di suo successore ed infine fornigli le informazioni necessarie per poterlo contattare.

2. In seguito ad una richiesta da parte di un Client, determinare quale Storage Node è il detentore del dato cercato e contattarlo personalmente inoltrando la richiesta del Client.

#### Keep Alive Maker

Questa subenetità è creata e gestita dai nodi nell'anello che assumono il ruolo di Bootstrap Node. Per le ragioni spiegate durante la descrizione del Keep Alive Controller, il Keep Alive Maker è un processo (o meglio, un Thread) che esegue il Task di inviare periodicamente segnali di presenza al Controller.

### 1.1.3 Client

È l'unica entità esterna alla struttura ad anello che caratterizza la *DHT*. Infatti ricopre il ruolo di utilizzatore dei servizi offerti dal Sistema, che consistono nella ricerca di dati presenti in esso. Il Client fornirà come richiesta l'identificatore del dato desiderato, e come risposta riceverà (se presente) il contenuto del dato in questione presente nell'anello.

## 1.2 Strutture Dati

### 1.2.1 DHT (*Distributed Hash Table*)

È la struttura "madre" di *Reduced Dynamo*, nonchè unica Stuttura Dati qui descritta di natura astratta.

Infatti, la complicata struttura ad anello che caratterizza il Sistema, non è una struttura dati concreta manipolabile in maniera diretta, ma la combinazione delle dataTable e nodeTable presenti (e diverse) su ciascun Storage Node facenti parti del sistema.

Comunque, si ricorda che il criterio per la generazione di identificatori per Storage Node e dati presenti nella *DHT* si basa sull'utilizzo della funzione *SHA-1*.

### 1.2.2 storageNodeList

La storageNodeList, utilizzata esclusivamente dal BootstrapServer, è un ArrayList di stringhe contente le informazioni che identificano l'insieme attuali di Nodi di Bootstrap nel Sistema. Come vedremo in seguito, tale tabella viene modificata da questa entità, ma viene copiata ed utilizzata anche dal Keep Alive Controller.

### 1.2.3 dataTable

La dataTable è una delle due strutture dati che formano e rappresentano la *DHT*. Ironia della sorte, non è altro che una Hash Table che funge da Database dei dati presenti nel Sistema, con gli unici due campi del dato stesso e del suo identificatore (che funge da chiave ed è ottenuto tramite la funzione *SHA-1*).

Ogni Storage Node ha la propria dataTable, ed ogni elemento è presente solo in quel particolare nodo in tutto l'anello. Come verrà spiegato dettagliatamente in seugito, essa viene gestita dall'oggetto che si occupa delle connessioni TCP dello Storage Node, e viene utilizzata ogni volta che viene aggiornato il predecessore del relativo Storage Node, oppure per effettuare la ricerca di un dato voluto da un Client.

### 1.2.4 nodeTable

Come per la dataTable, anche la nodeTable è una struttura dati presente in ogni Storage Node del Sistema. A differenza dell'applicazione originale, dove ciascun nodo del Sistema conosce solo un sottoinsieme dei nodi presenti nell'anello, in *Reduced Dynamo* ciascun Storage Node conosce le informazioni necessarie a contattare qualsiasi altro anello attualmente presente nella *DHT*.

Anche in questo caso, la nodeTable è stata implementata come una Hash Table dove la chiave è rappresentata da un intero, ovvero l'identificatore del nodo di cui sono riportate le informazioni per poterlo contattare. Tali informazioni sono:

- Indirizzo IP
- Porta su cui è aperta una Socket UDP in ascolto
- Porta su cui è aperta una Socket TCP in ascolto

Queste informazioni sono contenute in un'unica stringa che rappresenta il dato della Hash Table. Come vedremo in seguito, la nodeTable è gestita dall'oggetto che gestisce le operazioni di Multicast dello Storage Node.

### 1.2.5 socketList

Si tratta di un Array List di Socket, utilizzata dagli oggetti che formano il meccanismo di "Server Polling" che caratterizzano gli oggetti che gestiscono le connessioni TCP ed UDP per ciascun Storage Node del Sistema.

**Nota:** nella sezione di codice in cui si tratta tale struttura dati, essa viene indicata semplicemente come "list"; in questa relazione è stata rinominata per renderne più chiaro il suo scopo ed utilizzo.

### 1.2.6 dataFile

Questa semplice struttura dati non è altro che un file di testo che contiene tutti i dati necessari che verranno caricati nel Sistema. Il delimitatore di un dato nel file è l'interruzione della relativa linea di testo.

# Chapter 2

# Descrizione delle Classi e Scelte Implementative

In questo capitolo ci concenteremo sulla descrzione di ciascuna delle classi Java che compongono il Progetto.

Verranno riportati anche una serie di diagrammi UML (diagrammi di Sequenza) per rendere ancora più chiaro la struttura delle Classi e come si relazionano tra loro i vari oggetti che le implementano.

Infine, per non rendere troppo pesante la descrizione delle classi, non verranno trattate le differenze tra la versione in Ambiente Locale del Progetto e la versione di Rete durante la descrizione delle Classi, dato che le differenze sono essenzialmente due:

1. Un utilizzo diverso dei parametri con cui si eseguono i vari Main del progetto. Tale aspetto verrà dettagliatamente trattato nel Capitolo 3.

2. Una definizione diversa delle variabili che permettono le comunicazioni sopradescritte: in particolare, verranno assegnati valori diverse alle InetAddress a seconda della versione utilizzata.

## 2.1 BootstrapServer

Di seguito verranno riportate le classi che implementano l'entità Bootstrap Server. Si anticipa comunque che il numero di Thread che rappresentano il Bootstrap Server sono 2: un Thread che esegue il Task della Classe LogManager_Implementation ed un altro quello della Classe BootstrapServer_KeepAlive. Dato che entrambe le classi sono l'implementazione di subentità, si è deciso che i relativi Thread siano classificati come demoni.

### 2.1.1 BootstrapServer_Monitor

Come è stato spiegato durante la descrizione delle principali Strutture Dati presenti in *Reduced Dynamo*, ciascun nodo ha una propria nodeTable, la quale contiene le informazioni necessarie a contattare qualsiasi nodo presente sull'anello. Inoltre, ciascun nodo ha una propria dataTable, la quale contiene i dati che detiene il relativo nodo.

Come si può bene immaginare, è necessario un meccanismo che garantisca una mutua esclusione tra vari oggetti che vogliano modificare i contenuti di queste tabelle.

Un caso piuttosto ovvio per cui si rende necessario quanto sopra detto è quando (ad esempio) un Client effettua la ricerca su un dato che, in contemporanea, viene trasferito su un altro nodo, a seguito del simultaneo ingresso di un nuovo Storage Node nell'anello. Il risultato è la risposta dell'ex proprietario al Client dicendo che il dato non è presenta nel sistema.

Il modello quindi risultante da questa serie di osservazioni è il modello noto in letteratura come "Readers & Writers". In particolare, i Writers (ovvero i nuovi Storage Node), sono autorizzati a procedere con le loro operazioni di inserimento nell'anello se e solo se non sono presenti altri Writers o Readers (cioè i Client) che stanno svolgendo le proprie operazioni. Viceversa, i Readers sono autorizzati a procedere se e solo se nessun Writers sta eseguendo le proprie operazioni di inserimento (ma possono procedere in presenza di altri Readers). Per evitari situazioni di Starvation, si è deciso di alternare le autorizzazioni concesse ai Writers ed ai Readers.

Rispetto al classico modello sopra descritto, c'è un fattore aggiuntivo: se il Keep Alive Controller desidera aggiornare la storageNodeList del Bootstrap Server, allora ha la precedenza su tutti i Writers e/o Readers che hanno intenzione di procedere. In questo modo si evita che nuovi Storage Node o Client si rivolgano a Bootstrap Node potenzialmente non più raggiungibili.

I metodi di `start_join_Node` e `start_search_Client` indicano l'introduzione nel modello utilizzato rispettivamente di un nuovo Writer e di un nuovo Reader, iniziando così i tentativi di acquisizione della lock. Analogamente, i metodi `end_join_Node` e `end_search_Client` indicano il rilascio della lock acquisita. Infine, i metodi di `start_Keep_Alive` e `end_Keep_Alive` hanno operazioni analoghe a quelle già descritte, ma con condizioni sull'acquisizioni della lock "meno severe" (avendo una maggiore priorità).

### 2.1.2 BootstrapServer_Interface

Dato che le interrogazini da parte di un Client oppure di un nuovo Storage Node verso il Server deve avvenire tramite RMI, è necessaria un'interfaccia che indichi quali sono i metodi del Bootstrap Server possano essere chiamati in remoto.

Questi metodi riguardano l'operazione di ingresso da parte di un nuovo Storage Node e di un Client.

### 2.1.3 BootstrapServer_Implementation

È la parte più importante nell'implementazione dell'entità Bootstrap Server, in quanto (come suggerisce il nome della Classe) vengono implementati i metodi remoti descritti in BootstrapServer_Interface.

In particolare, il metodo `start_join_node` (chiamato da un nuovo Storage Node) permette per prima cosa di ottenere la mutua esclusione sull'accesso alle strutture condivise (dato che si richiama immediatamente il relativo metodo del BootstrapServer_Monitor).

Successivamente, nel caso in cui il nodo in questione è il primo ad essere inserito nel sistema, si crea e lancia il Thread che esegue il Task della Classe BootstrapServer_KeepAlive, ovvero si crea la subentità Keep Alive Controller.

Altrimenti si sceglie casualmente un elemento presente nella storageNodeList, si ricavano i dati per poterlo contattare e li si forniscono come parametro di ritorno al metodo. Prima della fine del metodo, si verifica se sono presenti già $k$ elementi nella storageNodeList: se così non fosse, si aggiunge il nodo che ha chiamato il metodo.

Il metodo `start_client_search` risulta più semplice del precedente, in quanto non sono necessarie tutte le operazioni e controlli che riguardano l'inserimento di un nodo nell'anello.

Infine, i metodi di `setStorageNodeList` e `getStorageNodeList` vengono richiamati dall'oggetto che implementa la Classe BootstrapServer_KeepAlive per permettere la manipolazione della storageNodeList; mentre il `setLogManager` è per permettere al Bootstrap Server di avere un riferimento al Log Manager creato (e comunicargli l'avvenuto ingresso di nuovi nodi).

### 2.1.4 BootstrapServer_KeepAlive

Il Thread demone che esegue il `run()` di questa classe scrive priodicamente su un file *KeepAlive.txt* i pacchetti di notifica di presenza da parte di tutti i Bootstrap Node, la tabella dei Bootstrap Node attuale e la tabella dei Bootstrap Node (eventualmente) aggiornata.

Per fare ciò, crea una DatagramSocket su una porta nota ed esclusiva. Dopodichè inizia la propria fase di ascolto per un periodo di 5 secondi, durante i quali aggiunge i datagrammi ricevuti in un'apposita lista messList.

Dopodichè ottiene la storageNodeList attuale dal Bootstrap Server tramite il metodo `getStorageNodeList()` della classe BootstrapServer_Implementation, e verifica che tutti gli elementi in essa presenti compaiano anche in messList. Se così non fosse, elimina gli elementi della storageNodeList presente sul Bootstap Server di cui non si è ricevuto il Keep Alive tramite il metodo `setStorageNodeList()`.

Infine elimina i possibili elementi ancora presenti all'interno di messList (è infatti possibile che un pacchetto "perso nella rete" arrivi a destinazione quando il relativo Storage Node ormai non è più un Bootstrap Node, e quindi viene scartato).

### 2.1.5 LogManager_Interface

Dato che le notifiche da parte del Bootstrap Server di avvenuto inserimento di un nuovo Storage Node avvengono tramite un meccanismo di callback attraverso RMI, qui viene dichiarato il metodo chiamato in remoto da parte del Bootstrap Server, ovvero `notifyMe` (chiamato durante l'esecuzione del metodo `end_join_node`).

### 2.1.6 LogManager_Implementation

Al momento della creazione del Log Manager, il costrutture crea il file di testo *LogManager.txt* nella directory di esecuzione del Sistema. Il file in questione verrà aggiornato ogni volta che il metodo `notifyMe` verrà eseguito, indicando la data ed ora in cui l'inserimento è stato ultimato e le informazioni che contraddistinguono il nuovo nodo.

## 2.2 Storage Node

È forse l'entità la cui implementazione si è rivelata più difficile, dato l'alto numero di compiti assegnatogli e la loro difficoltà.

In questa versione di *Reduced Dynamo* sono 4 i Thread che gestiscono le operazioni di ciascuno Storage Node (rappresentato ciascuno da una Classe diversa):

- Un Thread per svolgere nell'ordine corretto le operazioni per l'inserimento del nuovo Storage Node all'interno dell'anello. In particolare, questo Thread genererà a sua volta gli altri Thread descritti in seguito; inoltre è l'unico Thread che termina la propria esecuzione con l'esecuzione del proprio Task (a differenza degli altri Thread che non terminano mai i propri Task).

- Un Thread che gestisca le comunicazioni che avvengono tramite il Multicast, il quale si occuperà anche dell'aggiornamento della nodeTable

- Un ulteriore Thread per la gestione della connessione UDP per il nodo in questione (sia in ingresso in uscita).

- Ed infine un ultimo Thread per permettere la comunicazione tramite TCP.

  Le operazioni svolte da ciascuno di questi Thread non si limitano a quanto sopra descritto (i dettagli saranno illustrati nelle relative classi), inoltre, come è facile immaginare, dato che l'unione di questi Thread formano l'entità di Storage Node, è chiaro come questi collaborino e comunichino tra loro.

### 2.2.1 StorageNode

Lo scopo principale di questa Classe consiste nella costruzione di uno Storage Node: infatti, dopo aver contattato il Bootstrap Server e creato gli altri Thread che si occupano dei vari tipi di comunicazioni che riguardano lo Storage Node in questione, esso termina normalmente (a differenza degli altri Thread da lui generati, come si è già detto nel precedente paragrafo).

La comunicazione inziale con il Bootstrap Server ha lo scopo di ottenere l'indirizzo del Bootstrap Node a cui rivolgersi (tranne che non si tratti del primo Storage Node!) ed ottenere la Lock da parte dell BootstrapServer_Monitor.

Fatto ciò , contattiamo (tramite UDP) il Bootstrap Node che ci è stato assegnato, fornendogli il nostro identificatore generato tramite *SHA-1*. Il Bootstrap Node risponderà fornendoci gli indirizzi necessari per comunicare con il nostro Successore tramite TCP, insieme ad una copia della sua nodeTable. Durante questa trasmissione con il Bootstrap Node vengono effettuati anche controlli sulla presenza nell'anello di un altro nodo con il nostro stesso identificatore.

Infine creiamo (nell'ordine seguente) gli oggetti delle Classi di StorageNode_Multicast, di StorageNode_UDP e di StorageNode_TCP e ne lanciamo i Thread.

Se il nodo aggiunto è il primo dell'anello, le operazioni che riguardano altri nodi che sono state descritte in questo paragrafo (ovviamente) non avvengono, ma si leggono i dati contenuti all'interno del file *textfile.txt* (che rappresenta la struttura dati dataFile) tramite il metodo `reader()` della Classe NodeFileReader, inzializzando così la dataTable.

Tutte le Socket utilizzate durante le comunicazioni che riguardano questa Classe sono ottenute tramite l'utilizzo di un oggetto della Classe ScannerPorte (di cui parleremo più avanti).

### 2.2.2 StorageNode_Multicast

I compiti svolti da questa classe sono:

- Creazione del gruppo di Multicast (nel caso in cui l'oggetto di questa classe sia stato creato dal primo Storage Node dell'anello).
- Invio delle proprie informazioni al gruppo di Multicast.
- Operazione di join al grupo di Multicast.
- Inizio di un loop di ascolto, dove si "sniffano" tutti i messaggi spediti al gruppo di Multicast, aggiornando di conseguenza la propria nodeTable.
- Tramite il metodo di `getTable()` si fornisce la nodeTable aggiornata.

Si faccia notare che la nodeTable iniziale è fornita al momento della creazione dell'oggetto dell Classe in questione, garantendo così una nodeTable aggiornata per tutti gli Storage Node.

### 2.2.3 StorageNode_UDP&TCPMonitor

Prima di parlare degli oggetti che implementano le funzionalità di TCP e UDP all'interno degli Storage Node, è bene descrivere le due classi di StorageNode_UDPMonitor e StorageNode_TCPMonitor, le quali sono uguali ma operano in contesti diversi.

Questi due Monitor sono necessari dal momento che più Storage Node che vogliono essere inseriti nella *DHT* possono contattare lo stesso Bootstrap Node in contemporanea (e quindi è possibile che accadano eventi non corretti nella gestione delle funzioni UDP del nodo), oppure può capitare che più Bootstrap Node contattino insieme lo stesso Storage Node detentore di un dato cercato da un Client (e quindi un comportamento non corretto dei compiti svolti dal TCP).

Per risolvere questo problema è stato implementato un meccanismo di "Server Polling": il Writer (ovvero la connessione in ingresso UDP e TCP) acquisisce la lock per poter accedere in maniera esclusiva sulla propria struttura dati socketList e rimane in ascolto per un certo tempo (circa 1 secondo), aggiungendo alla socketList (ovvero un Array List di DatagramPacket nel caso di UDP, mentre di Socket generate dal metodo di `accept()` nel caso del TCP) l'elemento ricevuto; allo scadere del tempo, il Writer rilascia la lock e viene acquisita dal Reader. Il Reader si occupa di leggere, soddisfare ed infine rimuovere tutte le richieste presenti nella socketList.

I termini di Writer e Reader non sono casuali: l'implementazione dei Monitor sopra descritti è stata realizzata seguendo il modello dei Readers & Writers di cui abbiamo già parlato durante la descrizione del BootstrapServer_Monitor. Tuttavia, si faccia notare come nel modello qui descritto siano presenti un solo Reader ed un solo Writer, e quindi il meccanismo utilizzato risulta forse "troppo potente" o comunque "non utilizzato al massimo delle sue capacità".

Tuttavia, avendo l'autore già implementato il codice del BootstrapServer_Monitor ("riciclabile" in questo contesto), e non riducendo l'efficienza del Sistema nel suo utilizzo, si è comunque preferito utilizzare il modello sopra descritto piuttosto che uno ad hoc per un solo Reader ed un solo Writer.

Il motivo per cui sono state create due classi uguali tra loro, al posto dell'utilizzo di due diversi oggetti per la stessa Classe, è stata una scelta dell'autore per rendere più chiaro il contesto in cui tali Classi vengono utilizzate.

La differenza rispetto alla classe di BootstrapServer_Monitor è la non presenza della entità di massima precedenza, ovvero il Keep Alive.

### 2.2.4 StorageNode_UDP&TCPWriter

Queste due Classi (utilizzate a seconda del tipo di connessione a cui fanno riferimento), svolgono i compiti di Writer del Server Polling descritto nel precedente paragrafo. Differenziano sul tipo di Monitor a cui fanno riferimento, il tipo di Socket utilizzate per ricevere dati in ingresso ed il tipo di dato che aggiungono alla propria socketList.

### 2.2.5 StorageNode_UDP

Di segutio sono elencate le principali funzioni di questa Classe:

- Si inizia il loop di lettura della socketList. Dopo aver acquisito la lock sulla socketList a cui si fa riferimento, si leggono tutti gli elementi presenti in essa. Di fatto quindi, si svolgono i compiti del Reader nel modello del Server Polling.
- Nel caso in cui si tratti di un nuovo Storage Node:
  - Viene richiamato il metodo `getTable()` dell'oggetto di StorageNode_Multicast per ottenere la nodeTable aggiornata.
  - Vengono calcolati il successore e predecessore nell'anello del nuovo nodo (tramite i loro identificatori) e se ne salvano gli indirizzi
  - Si verifica che non esistano altri nodi nell'anello con lo stesso identificatore del nodo che ci ha contatti.
  - Si risponde tramite UDP al nodo con il risultato dell'operazione.
- Nel caso in cui si tratti invece di un Client:
  - Si decreta quale sia il Nodo che detiene il dato cercato. Anche in questo caso, utilizziamo la nodeTable aggiornata tramite il metodo `getTable()`.
  - Si crea una Socket TCP connessa allo Storage Node detentore del dato e si invia uno stream di dati contente l'identificatore del dato cercato e le informazioni con cui contattare il Client in questione.

Anche in questa Classe si utilizza l'oggetto di ScannerPorte, in modo da poter creare una Socket TCP in sicurezza durante la comunicazione con il nodo detentore di un dato cercato da un Client.

### 2.2.6 StorageNode_TCP

Questa Classe, oltre ad amministrare la connessione TCP dello Storage Node, implementa una serie di altre funzionalità. In particolare esegue le seguenti operazioni:

- Si stabilisce una connessione TCP con il proprio Sucessore comunicandogli la nostra natura di nuovo Storage Node e si ottengono da esso i dati di nostra competenza (sempre che ve ne siano), definendo di fatto la nostra dataTable.
- Si stampa la propria dataTable
- Si inizia il loop di lettura della socketList. Dopo aver acquisito la lock sulla socketList a cui si fa riferimento, si leggono tutti gli elementi presenti in essa. Di fatto quindi, si svolgono i compiti del Reader nel modello del Server Polling.

- Nel caso in cui si tratti di un nuovo Storage Node:
  - Si ottiene la nodeTable dall'oggetto StorageNode_Multicast tramite il metodo `getTable()`.
  - Si calcolano quali dati sono di sua competenza leggendo la propria dataTable (la nodeTable è necessaria per casi particolari descritti nel codice).
  - Si inviano tali elementi in risposta tramite una Socket TCP.
  - Si stampa la propria dataTable aggiornata.
- Nel caso contrario, ovvero si tratta di un Client:
  - Si effettua la ricerca dell'identificatore ricevuto (che rappresenta il dato cercato) nella propria dataTable
  - Si crea una Socket UDP e si invia il risultato della ricerca al Client.

Anche in questa Classe si utilizza l'oggetto di ScannerPorte, in modo da poter creare una Socket TCP in sicurezza durante la risposta al nuovo Storage Node che ci ha contattati ed inoltre la creazione di una Socket UDP per inviare il risultato sulla ricerca richiesta dal Client.

**Nota sull'esecuzione:** la stampa dei contenuti della propria dataTable avviene la prima volta che essa viene definita ed ogni volta che essa viene aggiornata (ovvero quando si aggiorna il nostro predecessore). Di conseguenza, ogni volta vengono stampate due dataTable in contemporanea (cioè quella del nuovo Storage Node e quella del suo sucessore a cui la dataTable è stata aggiornata): ciò potrebbe causare dei problemi di leggibilità delle due tabelle se il Progetto viene utilizzato nella sua versione locale, dato che le stampe degli elementi delle due tabelle potrebbero sovrapporsi l'una all'altra.

È stato quindi deciso di far dormire ciascun Thread per un tempo proporzionale all'ordine cronologico con cui è stato creato il relativo Storage Node nella *DHT*, garantendo così le stampe corrette delle due Tabelle. Tuttavia, tale ritardo potrebbe comportare la stampa delle Tabelle dopo che sono state svolte (e stampate) altre operazioni (come l'inizio di inserimento di un nuovo nodo nel Sistema, o l'inizio di ricerca da parte di un Client).

### 2.2.7 StorageNode_KeepAlive

Tramite questa classe, un Nodo di Bootstrap invia i propri messaggi di Keep Alive all'oggetto della Classe BootstrapServer_KeepAlive.

In particolare si crea una DatagramSocket e si imposta un timeout di ascolto di 15 secondi (nel caso in cui il Keep Alive Controller non rispondesse). Dopodichè si invia il proprio messaggio di Keep Alive (contente le nostre informazioni sugli indirizzi che ci identificano) e si rimane in attesa di una risposta da parte del Keep Alive Controller.

Se il timeout scade, si considera il pacchetto perduto, altrimenti si invia un nuovo messggio di Keep Alive.

Si faccia notare che la porta su cui è aperta DatagramSocekt del BootstrapServer_KeepAlive è nota (come l'indirizzo IP del BootstrapServer).

Per l'ennesima volta viene utilizzato il metodo `DatagramSocket()` della Classe ScannerPorte per la DatgramSocket qui utilizzata.

## 2.3 Client

L'implementazione di questa entità è piuttosto semplice: basta infatti una Classe (il cui Task è esegutio da un Thread dedicato) per implementare tutte le sue funzionalità.

### 2.3.1 Client

Il primo compito che il `run()` di questa classe deve svolgere è generare in maniera pseudo-casuale il dato che vuole cercare. Per fare ciò sfrutta il metodo `getRandomIdData()` della classe NodeFileReader (di cui parleremo più avanti).

Dopodichè, deve contattare il Bootstrap Server tramite RMI ed ottenere gli indirizzi necessari a contattare tramite UDP il Bootstrap Node che gli verrà assegnato.

Fatto ciò, viene creata una Socket UDP per poter inviare al BootstrapNode che ci è stato assegnato il messaggio contente i nostri indirizzi e l'identificatore del dato che vogliamo cercare. Dopo aver inviato tali informazioni, il Client rimane in ascolto sulla DatagramSocket creata in attesa del risultato della ricerca (che giungerà dal nodo detentore del dato, e non dal Bootstrap Node, in maniera del tutto trasparente al Client), la quale verrà stampata su schermo.

Anche in questo caso viene utilizzato il metodo `DatagramSocket()` della Classe ScannerPorte per la DatgramSocket qui utilizzata.

## 2.4 Altre Classi

### 2.4.1 ScannerPorte

Questa classe (così frequentemente richiamata) permette la creazione di Socket TCP o UDP in maniera "sicura", cioè senza rischiare che vengano create (ed utilizzate) più Socket sulla stessa porta.

Per fare ciò semplicemente si tenta la creazione della Socket in questione in maniera sistematica su un insieme di porte deciso dall'autore, ignorando l'eventuale BindingException sollevata.

Si fa notare come i metodi implementati siano di tipo synchronized: questa necessità deriva dal fatto che più oggetti possano richiamare nello stesso momenti questi metodi, e quindi utilizzare diverse Socket sulla stessa porta. Garantendo la mutua esclusione all'accesso di questi metodi, si garantisce una corretta creazione delle Socket.

### 2.4.2 NodeFileReader

Il metodo `read()` (chiamato solo dal primo Storage Node inserito) si occupa di leggere i dati contenuti nella struttura dati dataFile, ovvero il file di testo *"textfile.txt"*, dove ciascun dato è rappresentato da ciascuna linea nel file. I tipi di dati presenti quindi in *Reduced Dynamo* sono stati implementati come Stringhe.

Il metodo `getRandomIdData()` (eseguito da ciascun Client) seleziona casualmente un dato presente all'interno del file e ne calcola l'identificatore tramite *SHA-1*. In questo modo si garantisce che ciascun Client effettui la ricerca su un dato effettivamente presente nel Sistema.

## 2.5 Programmi Eseguibili

I programmi esegubili decritti in seguito permettono la creazione vera e propria del Sistema. Per ottimizzare l'uso di molti Thread all'interno dello stesso esegubile, si è deciso di utilizzare una serie di Thread Pool.

### 2.5.1 BootstrapServer_Main

Tramite questo esegubile si crea l'entità Bootstrap Server, in particolare si effettuano le operazioni necessarie per la creazione di un oggetto RMI (creazione del registro, creazione dello stub, binding del nome ecc.) che rappresentano il Bootstrap Server stesso.

È necessario che questo sia il primo esegubile ad essere lanciato, se non ci si vuole imbattere in un errore da parte dei Client e/o degli Storage Node che provano a contattare un Bootstrap Server inesistente!

### 2.5.2 StorageNode_Client_Main

L'esecuzione del Main di questa Classe consiste nel richiedere all'Utente di inserire il numero di Storage Node e Client che si intendono creare. Dopodichè, tramite un Thread Pool, verrano creati (e lanciati) una serie di Thread, il cui task è la creazione di un numero di Storage Node pari a quello indicato dall'Utente.

Dopodichè, dopo un secondo di pausa, vengono creati i Client con lo stesso procedimento con cui sono appena stati creati gli Storage Node.

Sebbene questo eseguibile fosse stato creato durante la fase di debugging, se ne consiglia l'uso nel caso in cui non si è presa ancora una certa familiarità con le stampe prodotte durante l'esecuzione del Sistema.

### 2.5.3 Simulatore

Il Simulatore è stato creato per (appunto) simulare in maniera casuale la creazione di Storage Node e Client: vengono creati una serie casuale di elementi (per un massimo di 5) di Client e Storage Node, i cui Thread vengono lanciati con un ritardo anch'esso casuale.

Dopodichè il Simulatore dorme per un tempo casuale e riprende da capo.

Si sconsiglia fortemente l'uso di questo eseguibile se non si è familiari con le stampe prodotte dal Sistema.

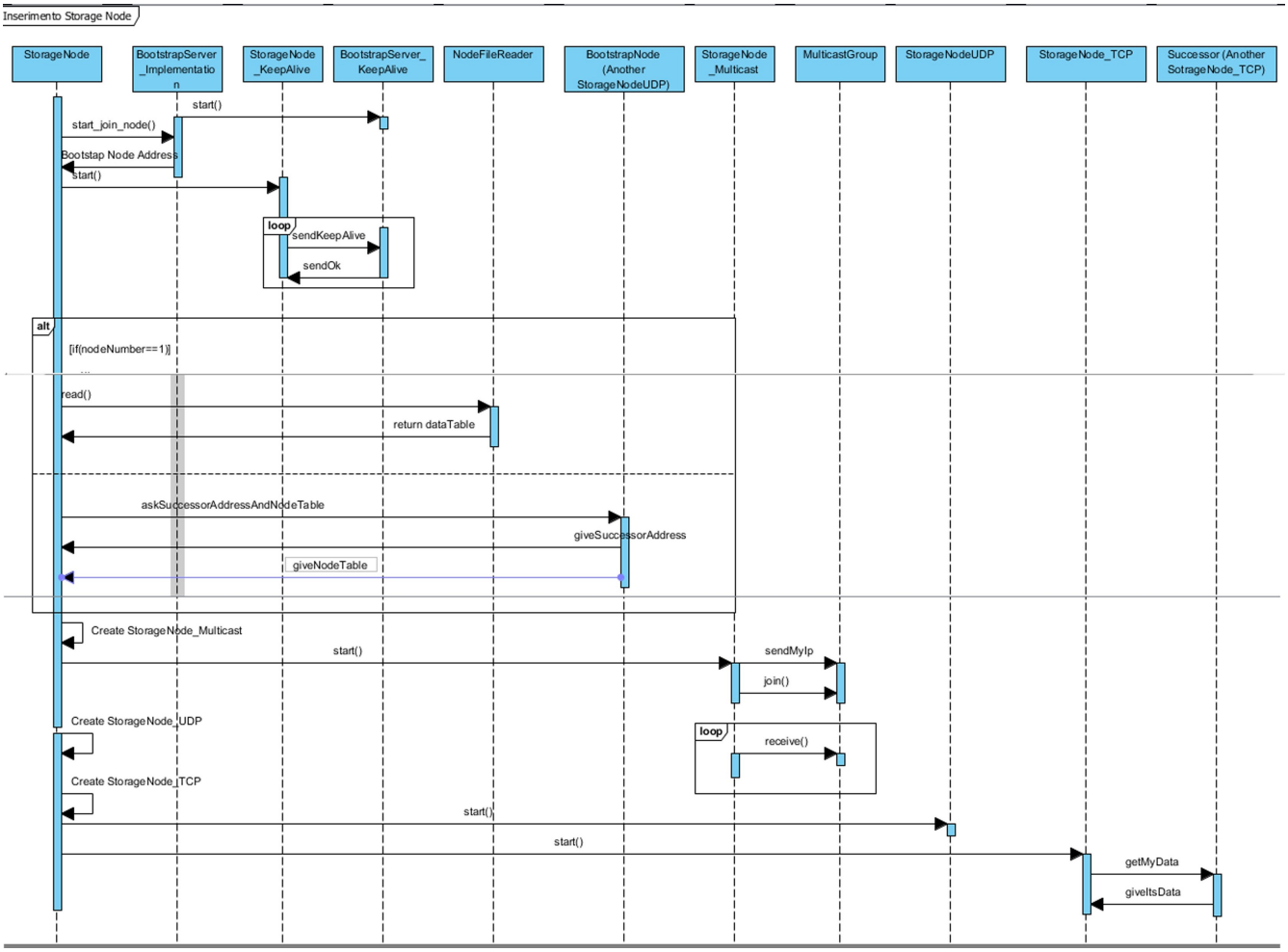
## 2.6 Diagramma di Sequenza

In questa sezione si riporteranno i diagrammi di sequenza che riguardano la creazione di uno Storage Node e la ricerca da parte di un Client, nel tentativo di rendere più chiari i procedimenti che li caratterizzano.

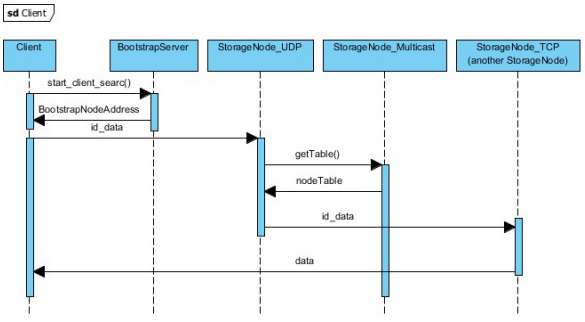
Per problemi di spazio, i Diagrammi di Sequenza sono stati "sintetizzati" in un unico messaggio tra due LifeLine. Ad esempio, l'insieme di istruzioni che comportano la domanda di un nuovo Storage Node della nodeTable e dell'indirizzo del successore al Bootstrap Node, vengono riassunte in un unico messaggio chiamato "askSuccessorAddressAndNodeTable" (che di fatto è un metodo inesistente).

Inoltre, per non rendere i Diagrammi ulteriormente complicati, si è deciso di non riportare nel dettaglio le operazioni che riguardano le classi StorageNode_UDP e StorageNode_TCP, ma solo i passi fondamentali che consistono nella creazione dello Storage Node e nella ricerca effettuata da un Client.

### 2.6.1 StorageNode




### 2.6.2 Client



# Chapter 3

# Manuale Utente

## 3.1 Manuale Utente

In questa sezione verranno illustrate le istruzioni necessarie per la corretta compilazione ed utilizzo di *Reduced Dynamo*, sia in un ambiente Locale che in Rete.

**Nota su Software IDE:**

nel caso in cui si voglia utilizzare *Reduced Dynamo* in ambienti di sviluppo integrati (come ad esempio Eclipse), è necessario semplicemente importare i file *.java* contenuti nella directory *src*; inoltre è necessario creare il file *filtext.text* all'interno dell'ambiente/directory di riferimento utilizzato dall'IDE desiderato (nel caso di Eclipse, la directory di default a cui si fa riferimento è *workspace*). È necessario inoltre impostare i corretti parametri per l'utilizzo della versione in locale oppure in rete, come descritto in seguito.

### 3.1.1 Operazioni Comuni

1. Assicurarsi che nella cartella *jar* sia presente il file *textfile.txt*
2. Assicurarsi che il file in questione non sia vuoto, e che tutte le stringhe (dove ciascuna è corrispondente ad un dato diverso) siano separate dall'inizio di una nuova riga.
3. Nel caso in cui si desideri utilizzare un file diverso rispetto a quello di default, effettuare i due passi precedenti e modificare tutte le occorrenze di "textfile.txt" nei file *NodeFileReader.java* e *StorageNode.java* con il nome del file di testo desiderato.
4. Compilare il progetto tramite *makefile.bat*

5. Se il progetto è stato compilato correttamente, all'interno della directory *classes* saranno presenti file *.class* per ogni file *.java* contenuto nella directory *src*; inoltre saranno presenti nella directory *jar* i file *BootstrapServer.jar*, *Simulatore.jar* ed infine *StorageNode_Client.jar* (oltre ai corrispondenti file *.bat* già presenti prima della compilazone.

### 3.1.2 Versione Locale

1. Aprire tramite editor di testo il file *BootstrapServer.bat* ed impostare il primo parametro a 0.

2. Salvare il file, chiuderlo ed infino eseguirlo. Se il Server funziona correttamente, nella nuova finestra creata dovrebbe essere stampato l'indirizzo IP della macchina su cui si sta lavorando (non di nostro interesse nella versione locale) e la creazione del file *LogaManager.txt* (attualmente vuoto dato che non è stato creato nessuno StorageNode) all'interno della Directory corrente.

3. A seconda che si voglia eseguire una simulazione controllata oppure casuale, aprire tramite editor di testo il file *StorageNodeClient.bat* oppure *Similuatore.bat* ed assicurarsi che non siano presenti parametri.

4. Salvare il file, chiuderlo ed infine esegurilo. Durante l'esecuzione del Sistema, è possibile aprire i file *LogManager.txt* e *KeepAlive.txt* per vedere (oltre le relative stampe prodotte dalle finestre create) l'evoluzione del Sistema.

### 3.1.3 Versione di Rete

1. Aprire tramite editor di testo il file *BootstrapServer.bat* ed impostare il primo parametro a 1.

2. Salvare il file, chiuderlo ed infino eseguirlo. Oltre ai risultati ottenuti dello stesso punto nella versione locale, è necessario prendere nota dell'indirizzo IP stampato dal Bootstrap Server.

3. A seconda che si voglia eseguire una simulazione controllata oppure casuale, aprire tramite editor di testo il file *StorageNodeClient.bat* oppure *Similuatore.bat* ed utilizzare come primo (ed unico) parametro l'indirizzo IP stampato dal Bootstrap Server.

4. Come punto 3. del locale.

## 3.2 Test

Sebbene un normale progetto prevedi la strutturazione di una serie di test rilevanti (ed eventualmente trattarne i risultati), per *Reduced Dynamo* la loro progettazionee risulta difficile da definire, dal momento che una parte dei fattori

fondamentali che li compongono sono basati sul caso (come la determinazione degli indirizzi IP nella versione locale, e quindi la posizione degli Storage Node all'interno dell'anello) o su risultati non facilmente previdibili (come gli identificatori generati dalla funzione *SHA-1*).

Detto questo, nella directory *testResult* sono presenti una serie di file di testo che riportano le stampe ottenute da un certo numero simulazioni casuali, in cui si sono verificati eventi esemplari o comunque di particolare interesse. Per facilitarne la comprensione, sono stati commentati dall'autore ed i dati utilizzati sono Stringhe di un carattere. Inoltre si consiglia di visualizzare i file di testo con Blocco Note di Wondiows occupando tutta la finestra.

Di seguito riportiamo un breve elenco, con descrizione, dei file in questione (si consiglia fortemente di leggere per primo il file del test *3StorageNode.txt*):

- *3StorageNode.txt*: lo scopo di questo test è far prendere confidenza all'utente con le stampe prodotte dal sistema nell'inserimento di nuovi Storage Node all'interno della *DHT*. Inoltre è presente il caso particolare in cui il nuovo Storage Node, essendo il nodo con identificatore più grande di tutto il Sistema, si rivolge al nodo con identificatore più piccolo.

- *StorageNodeClientMonitor.txt*: in questo test viene mostrato il comportamento del Monitor nell'implementazione del modello Readers&Writers con l'inserimento di 3 Storage Node e 3 Client. Infatti dopo l'inserimento dei primi due nodi, vengono creati i 3 Client, i quali hanno la precedenza rispetto allo Storage Node in quanto è il loro turno (si ricorda che il turno viene alternato per evitare meccanismi di starvation). Secondo il modello utilizzato, più Readers (cioè i Client) possono accedere alla risorsa condivisa in comporanea, e quindi la ricerca dei 3 Client avviene in maniera simultanea.

- *BootstrapServer.txt*: in questo semplice test vengono visualizzate le stampe generate dal Bootstrap Server in seguito all'inserimento di una serie di Storage Node. In particolare viene stampato l'indice occupato dal Bootstrap Node scelto in maniera casuale dal Bootstrap Server all'interno della storageNodeList ed i suoi indirizzi.

- *FailedConnection.txt*: qui invece viene visualizzato il comportamento (o meglio, la stampa) di nuovi Storage Node nel tentativo di contattare un Bootstrap Server inesistente.

- *NoStorageNode.txt*: in quest'ultimo test vengono fatte vedere le stampe generate dai Client nel caso in cui tentino una ricerca sulla *DHT* che non contiene Storage Node.