

PROGETTO LABORATORIO DI SISTEMI OPERATIVI

Simple Public Ledger

Applicazione client - server che offre un servizio di memorizzazione coppie (chiave, valore). I processi client inviano comandi ai server disponibili, che li elaborano per restituire ai client il risultato desiderato.

Ernesto De Crecchio N86001596

Università degli Studi di Napoli Federico II, A.A. 2018/19



Sommario

Introduzione	2
Compilazione.....	3
Server.....	3
Client	3
File di Configurazione.....	4
Cos'è e a cosa serve.....	4
Come è fatto	4
Come crearlo	4
Esempio.....	5
Risultato finale	5
Comandi	6
Lista comandi disponibili	6
Store	6
Corrupt.....	6
Search.....	6
List.....	6
Esecuzione.....	7
Server.....	7
Esempio esecuzione Server	7
Note sull'esecuzione del Server	9
Client	10
Esempio esecuzione Client	11
Esempio esecuzione Client con comando Store	11
Esempio esecuzione Client con comando Corrupt.....	12
Esempio esecuzione Client con comando Search.....	12
Esempio esecuzione Client con comando List.....	13
Note sull'esecuzione del Client.....	13
Protocollo livello applicazione	14
Socket.....	14
<i>Socket in fase di startup</i>	14
<i>Thread Master</i>	15
<i>Thread Slave</i>	15
<i>Gestione Comandi</i>	16
Schema Protocollo Applicativo.....	17
Codice Integrale	18
<i>Client</i>	18
<i>Server</i>	20

INTRODUZIONE

Il progetto a cui fa riferimento questa documentazione consiste in un'applicazione basata su un'architettura client – server.

Più server saranno eseguiti concorrentemente su una o più macchine e saranno connessi tra loro per lo scambio di messaggi.

Più client possono accedere ad uno o più server nello stesso momento, effettuando operazioni di inserimento, modifica o richiesta valori da questi ultimi.

I server si occuperanno di mantenere la coerenza tra i dati memorizzati nelle proprie liste locali e forniranno adeguati output ai client connessi.

COMPILAZIONE

Server

Per poter utilizzare l'applicativo è necessario effettuare la compilazione dei file di cui esso è composto; cominciamo con la compilazione del server: avviare un Terminale su un sistema Linux/UNIX e spostarsi nella directory contenente il file sorgente; a questo punto bisogna seguire la seguente sintassi:

```
gcc server.c -o server -lpthread
```

Il primo parametro indica che si sta invocando GCC, un compilatore comunemente utilizzato nei sistemi Linux/UNIX, che effettua operazioni di preprocessing, compiling, assembly e linking.

Il secondo parametro indica il sorgente sul quale effettuare le operazioni sopra descritte.

Il terzo parametro, in combinazione con il quarto, indica che si sta redirigendo l'output del comando nel file "server" (che in questo caso sarà un eseguibile).

Il quinto ed ultimo parametro è necessario per includere la libreria di gestione dei thread presenti nel file sorgente, pena errori di compilazione.

In caso di errori o warning, il terminale restituirà adeguati output; in caso di successo non sarà generato output.

Client

Come per il server, anche il client necessita di essere compilato; per fare ciò, bisogna avviare un Terminale (o utilizzare quello utilizzato in precedenza) su un sistema Linux/UNIX e spostarsi nella directory contenente il file sorgente; a questo punto bisogna seguire la seguente sintassi:

```
gcc client.c -o client
```

Come è possibile notare, il comando è pressoché identico al precedente, a meno di un parametro mancante, "-lpthread", poiché nel sorgente client.c non vi sono thread da gestire. In questo caso l'output del comando sarà rediretto nel file "client" (che anche in questo caso sarà un eseguibile).

Così come accadeva per il server, anche per il client in caso di errori o warning saranno restituiti adeguati output; in caso di successo invece non sarà generato output.

FILE DI CONFIGURAZIONE

Cos'è e a cosa serve

Un file di configurazione è un file di testo contenente l'elenco degli indirizzi IP e numero di porta che identificano i server a cui il server che ne usufruisce deve collegarsi.

Viene passato al server in fase di esecuzione per indicargli quali altri server compongono la struttura.

Come è fatto

Un file di configurazione possiede tante righe quanti sono i server a cui bisogna collegarsi.

Ognuna di queste righe identifica un unico server e riporta l'indirizzo IP e il numero di porta nel seguente formato:

`xxx.yyy.zzz.www:pppp`

Come crearlo

Per creare un file di configurazione è necessario seguire poche e semplici indicazioni:

- Su una riga non può esserci più di un riferimento ad un server;
- Ogni riga deve contenere esattamente 20 caratteri (carattere di new line non incluso) e:
 - I primi 15 caratteri devono essere un indirizzo IPv4 rappresentato in notazione decimale puntata estesa;
 - Il 16-esimo carattere deve essere il segno di punteggiatura ":";
 - I restanti 4 caratteri devono essere un numero di porta (compreso tra 1024 e 9999, riservati dai sistemi Linux/UNIX ai processi utente).
- Il numero di righe deve essere equivalente al numero di server a cui collegarsi;
- L'ultima riga non deve essere seguita dal carattere di new line;
- Non deve essere presente il riferimento al server che usufruirà del file di configurazione.

È necessario seguire questa serie di operazioni per ogni server, al fine di ottenere una connessione omogenea tra i vari server ed il corretto funzionamento dell'applicativo; da ciò si può dedurre che, affinché avvenga il corretto collegamento tra i server, è necessario che il numero di file di configurazione sia pari al numero di server che comporranno la rete.

È necessario, inoltre, che i file di configurazione siano coerenti tra loro al fine di preservare il corretto funzionamento dell'applicativo.

Esempio

Supponiamo di dover effettuare il collegamento di tre server tra loro, avremo quindi necessità di tre file di configurazione di questo tipo:

	<i>config0</i>	<i>config1</i>	<i>config2</i>
<i>Riga #1</i>	127.000.000.001:4441	127.000.000.001:4440	127.000.000.001:4440
<i>Riga #2</i>	127.000.000.001:4442	127.000.000.001:4442	127.000.000.001:4441

Il file di configurazione *config0* verrà passato su riga di comando al server *127.0.0.1:4440*, il file *config1* verrà passato al server *127.0.0.1:4441* e il file *config2* verrà passato al server *127.0.0.1:4442*.

Risultato finale

La corretta creazione di file di configurazione coerenti tra loro permette l'instaurazione di una solida connessione tra tutti i server del sistema.

COMANDI

Lista comandi disponibili

L'applicativo prevede l'utilizzo di quattro comandi:

- `store (x,y)`
- `corrupt (x,z)`
- `search (x)`
- `list`

Store

Il comando `store` riceve in input i parametri chiave e valore. Questo comando memorizza in una lista dinamica una coppia chiave-valore solo se la chiave non è già presente nella lista; il comando viene inoltrato a tutti i server attivi.

```
store chiave valore
```

Corrupt

Il comando `corrupt` riceve in input i parametri chiave e nuovoValore. `corrupt` sostituisce una coppia chiave-valore esistente con la coppia chiave-nuovoValore. Se la lista non contiene una coppia con la medesima chiave, il client riceve un messaggio d'errore. A differenza di `store`, questo comando non viene inoltrato.

```
corrupt chiave nuovoValore
```

Search

Il comando `search` ricerca nella lista locale la coppia chiave-valore relativa alla chiave cercata. Riceve in ingresso il valore di una chiave. Se la lista non contiene una coppia con la medesima chiave, il client riceve il messaggio "Chiave assente". Se la coppia è invece presente nel server, la richiesta viene inoltrata a tutti gli altri server e, se tutte le coppie chiave-valore coincidono, il server ritorna la coppia; in caso di discrepanze tra queste, il client riceve il messaggio "Ledger corrotto".

```
search chiave
```

List

Il comando `list` invia al client tutte le coppie chiave-valore presenti nella lista locale. Non riceve parametri in ingresso.

```
list
```

ESECUZIONE

Server

Una volta terminata la fase di compilazione dei sorgenti e di creazione dei file di configurazione, si può passare all'esecuzione dell'applicativo.

Per prima cosa è necessario avviare i server poiché, prima di poter interagire con i client, essi hanno bisogno di stabilire una connessione tra di loro per poter comunicare.

Per eseguire un server bisogna: avviare un Terminale su un sistema Linux/UNIX e spostarsi nella directory contenente l'eseguibile; a questo punto bisogna seguire la seguente sintassi:

```
./server fileConfig portaServer
```

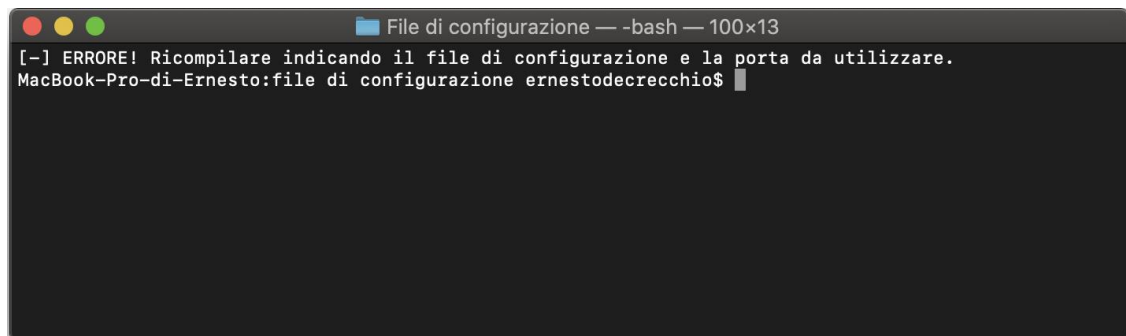
Il primo parametro indica il path del file eseguibile.

Il secondo parametro indica il file di configurazione da passare al server.

Il terzo ed ultimo parametro indica il numero di porta del server.

In caso di errori il terminale restituirà adeguati output; in caso di successo sarà avviato l'applicativo.

Nel caso in cui venissero passati parametri sbagliati o più parametri del richiesto, il programma restituirebbe un messaggio d'errore, come visibile in figura.



Esempio esecuzione Server

Supponiamo di voler avviare tre istanze di server, ognuno dei quali riceverà su riga di comando, rispettivamente, i file di configurazione *config0*, *config1*, *config2* e le porte *4440*, *4441* e *4442*.

La sintassi sarà quindi:

```
./server config0 4440
./server config1 4441
./server config2 4442
```

Nota.: ognuno di questi comandi deve essere lanciato su terminali diversi.

Una volta eseguito il primo comando, l'output somiglierà al seguente:

```
File di configurazione — server config0 4440 — 100x13
[+] Server socket creato correttamente!
[+] Connesso correttamente alla porta 4440

Server pronto...

In attesa di collegamento con gli altri server...
█
```

Se in fase di collegamento un server dovesse ricevere una richiesta di connessione non autorizzata, questa verrebbe rifiutata e verrebbe mostrato in output un messaggio che avvisa che la connessione in entrata è stata rifiutata.

```
File di configurazione — server config0 4440 — 100x13
[+] Server socket creato correttamente!
[+] Connesso correttamente alla porta 4440

Server pronto...

In attesa di collegamento con gli altri server...

[-] Connessione in entrata rifiutata.
█
```

Una volta avviati tutti i server, l'output somiglierà invece al seguente:

```
File di configurazione — server config0 4440 — 100x20
Server pronto...

In attesa di collegamento con gli altri server...

[+] Connessione in entrata accettata (1/10)
[+] Connessione in entrata accettata (2/10)
[+] Connessione in entrata accettata (3/10)
[+] Connessione in entrata accettata (4/10)
[+] Connessione in entrata accettata (5/10)
[+] Connessione in entrata accettata (6/10)
[+] Connessione in entrata accettata (7/10)
[+] Connessione in entrata accettata (8/10)
[+] Connessione in entrata accettata (9/10)
[+] Connessione in entrata accettata (10/10)
[+] Collegamento con tutti i server effettuato con successo.

-----
In attesa di un client...
█
```

L'output ci informa:

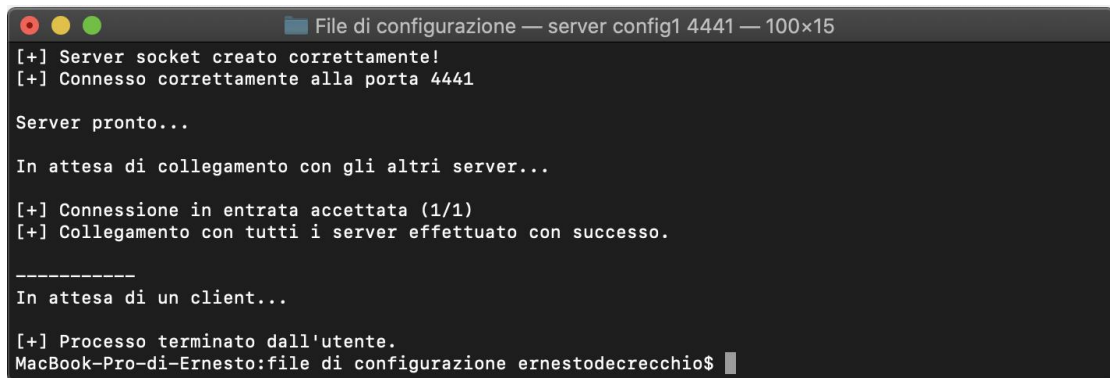
- Dello stato della creazione del socket;
- Dello stato della connessione del server sulla porta passata da riga di comando;
 - Informazione utile per discriminare diversi server in esecuzione sulla stessa macchina.
- Dello stato della connessione con i server presenti nel file di configurazione;
- Delle interazioni col client (discuteremo di questa funzionalità nelle prossime sezioni).

Note sull'esecuzione del Server

È importante sapere che, una volta avviata la fase di startup, non è possibile che nella stessa istanza di esecuzione vengano eseguiti altri server non autorizzati, poiché la connessione viene effettuata soltanto tra i server presenti nei file di configurazione.

Alla chiusura di un server vengono automaticamente chiusi tutti i server ad esso collegati per tutelare la coerenza del ledger.

Quando questo avviene, vuol dire che il server ha intercettato un segnale di SIGINT o SIGTERM (quindi una terminazione voluta del server stesso) e verrà visualizzata la seguente schermata:



```
File di configurazione — server config1 4441 — 100x15
[+] Server socket creato correttamente!
[+] Connesso correttamente alla porta 4441

Server pronto...

In attesa di collegamento con gli altri server...

[+] Connessione in entrata accettata (1/1)
[+] Collegamento con tutti i server effettuato con successo.

-----
In attesa di un client...

[+] Processo terminato dall'utente.
MacBook-Pro-di-Ernesto:file di configurazione ernestodecreschio$
```

Client

Terminata la fase di esecuzione dei server, si può passare all'esecuzione dei client, coloro che detteranno ai server le operazioni da effettuare.

Per eseguire un client bisogna: avviare un Terminale su un sistema Linux/UNIX e spostarsi nella directory contenente l'eseguibile; a questo punto bisogna seguire la seguente sintassi:

```
./client IPServer portaServer comando [par1] [par2]
```

Il primo parametro indica il path del file eseguibile.

Il secondo parametro indica l'indirizzo IP (in notazione decimale puntata) del server a cui connettersi.

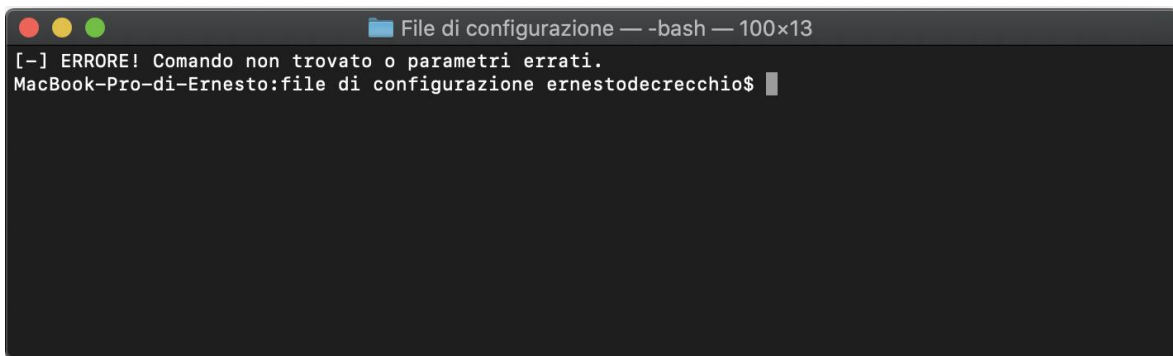
Il terzo parametro indica il numero di porta del server a cui connettersi.

Il quarto parametro indica il comando da inviare al server.

I parametri finali sono opzionali e dipendono dal comando richiesto.

In caso di errori il terminale restituirà adeguati output; in caso di successo sarà avviato l'applicativo.

Nel caso in cui venissero passati parametri sbagliati o più parametri del richiesto, il programma restituirebbe un messaggio d'errore, come visibile in figura.



Esempio esecuzione Client

Come indicato precedentemente, il client è la parte dell'applicativo che detta le operazioni da effettuare.

Il client è colui che utilizza maggiormente i comandi accennati in precedenza e, dunque, in questa sezione ci soffermeremo sulle varie possibilità offerte da questo.

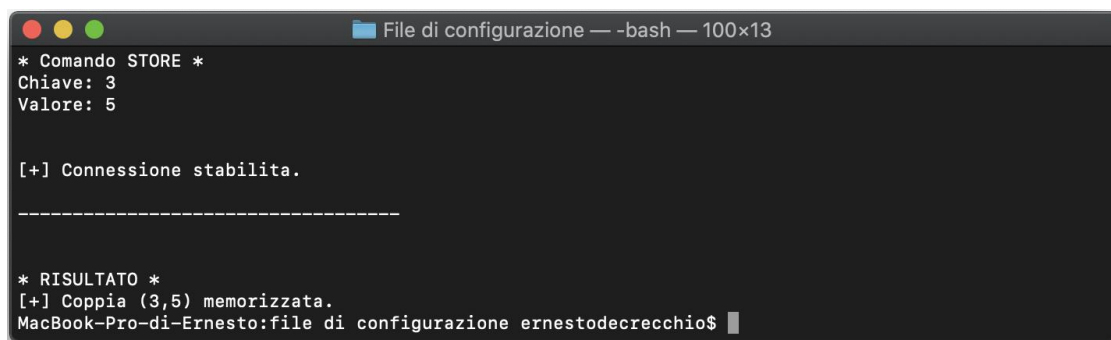
Esempio esecuzione Client con comando Store

Supponiamo di voler avviare un client e di voler memorizzare nel server *127.0.0.1:4440* la coppia (3, 5).

La sintassi sarà:

```
./client 127.0.0.1 4440 store 3 5
```

Una volta eseguito il comando, l'output somiglierà al seguente:



```
File di configurazione — -bash — 100x13
* Comando STORE *
Chiave: 3
Valore: 5

[+] Connessione stabilita.
-----

* RISULTATO *
[+] Coppia (3,5) memorizzata.
MacBook-Pro-di-Ernesto:file di configurazione ernestodecrecchio$
```

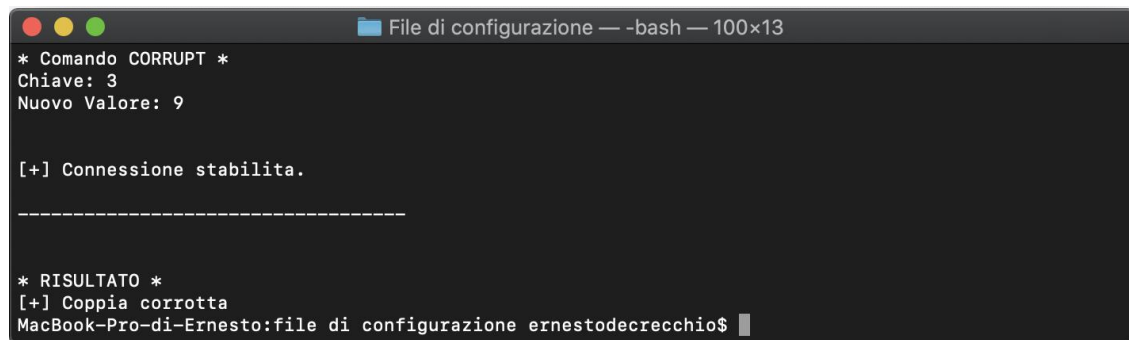
Esempio esecuzione Client con comando Corrupt

Supponiamo di voler avviare un client e di voler corrompere la coppia (3,5), precedentemente memorizzata nel server *127.0.0.1:4440*, con la coppia (3,9).

La sintassi sarà:

```
./client 127.0.0.1 4440 corrupt 3 9
```

Una volta eseguito il comando, l'output somiglierà al seguente:



```
File di configurazione — -bash — 100x13
* Comando CORRUPT *
Chiave: 3
Nuovo Valore: 9

[+] Connessione stabilita.
-----

* RISULTATO *
[+] Coppia corrotta
MacBook-Pro-di-Ernesto:file di configurazione ernestodecreschio$
```

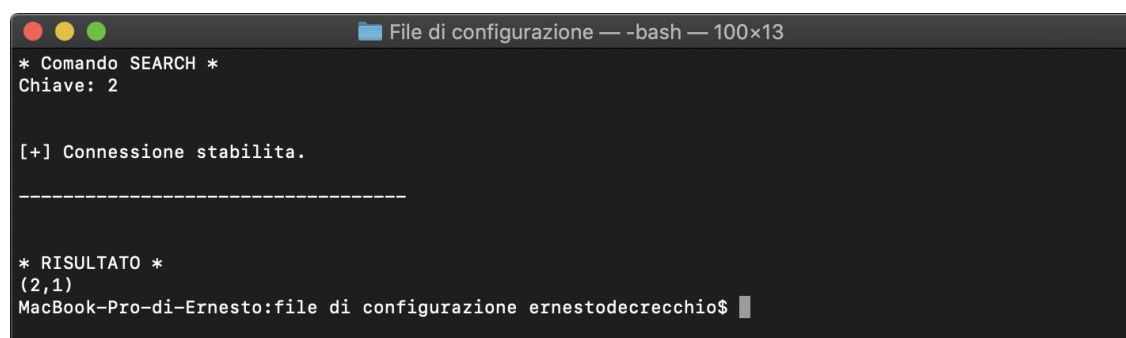
Esempio esecuzione Client con comando Search

Supponiamo di voler avviare un client e di voler ricercare nel server *127.0.0.1:4440* la coppia con chiave 3.

La sintassi sarà:

```
./client 127.0.0.1 4440 search 2
```

Una volta eseguito il comando, l'output somiglierà al seguente:



```
File di configurazione — -bash — 100x13
* Comando SEARCH *
Chiave: 2

[+] Connessione stabilita.
-----

* RISULTATO *
(2,1)
MacBook-Pro-di-Ernesto:file di configurazione ernestodecreschio$
```

Esempio esecuzione Client con comando List

Supponiamo di voler avviare un client e di voler conoscere tutte le coppie chiave-valore presenti nel server *127.0.0.1:4440*.

La sintassi sarà:

```
./client 127.0.0.1 4440 list
```

Una volta eseguito il comando, l'output somiglierà al seguente:



```
MacBook-Pro-di-Ernesto:file di configurazione ernestodecreschio$ ./client 127.0.0.1 4440 list

* Comando LIST *

[+] Connessione stabilita.

-----

[
  * RISULTATO *
  Key: 1 - Value: 2
  Key: 3 - Value: 4
  Key: 2 - Value: 4
  Key: 5 - Value: 6
  Key: 7 - Value: 8
  Key: 9 - Value: 10
  Key: stringa - Value: stringa
  Key: 10 - Value: stringa
  Key: 4 - Value: 6
]
MacBook-Pro-di-Ernesto:file di configurazione ernestodecreschio$
```

Note sull'esecuzione del Client

È importante sapere che i comandi *store*, *corrupt* e *search* accettano in input soltanto stringhe che al loro interno non contengono parentesi tonde o virgole.

PROTOCOLLO LIVELLO APPLICAZIONE

Socket

La connessione client-server e server-server avviene utilizzando i socket TCP.

Un socket è un canale di comunicazione asimmetrico tra due processi in esecuzione sullo stesso calcolatore o su due computer diversi interconnessi da una rete di comunicazione; l'asimmetria si riflette anche nella diversa sequenza di operazioni che client e server devono eseguire per stabilire la connessione.

Il seguente applicativo utilizza i socket per permettere la comunicazione tra client e server; dove per server si intende un processo che fornisce un servizio, mentre invece per client si intende un processo che usufruisce di tale servizio.

Per la connessione di un server ad un client bisogna seguire i seguenti passaggi:

- Creazione del socket (`socket`)
- Assegnamento di un indirizzo (`bind`)
- Attivazione della connessione (`listen`)
- Accettazione di nuove connessioni (`accept`)
- Interazione con il client
- Chiusura del socket (`close`)

Per la connessione di un client ad un server invece, bisogna seguire i seguenti passaggi:

- Creazione del socket (`socket`)
- Connessione al server (`connect`)
- Interazione con il server
- Chiusura del socket (`close`)

I socket vengono identificati attraverso un file descriptor che viene ritornato e/o processato da opportune system call; ciò consente di utilizzare le system call classiche per l'I/O di basso livello (es.: `read`, `write`, `close`, etc) per interagire con essi.

Socket in fase di startup

Una volta avviato il server, viene creato un nuovo socket TCP con indirizzo e porta rappresentativi del server stesso.

Prima di procedere all'analisi delle connessioni in entrata da parte di eventuali client, tramite la funzione `startupConfig` vengono inizializzate le connessioni con gli altri server.

Durante la fase di startup vi è l'analisi del file di configurazione; ogni riga del file viene analizzata e il suo contenuto (IP e porta di un server) viene salvato in una struttura apposita che verrà utilizzata successivamente.

Una volta analizzato l'intero file, per ogni server salvato nella struttura viene creato un nuovo thread che si occuperà della sua gestione.

A questo punto dell'esecuzione dell'applicativo vengono creati diversi thread per gestire le operazioni richieste.

Thread Master

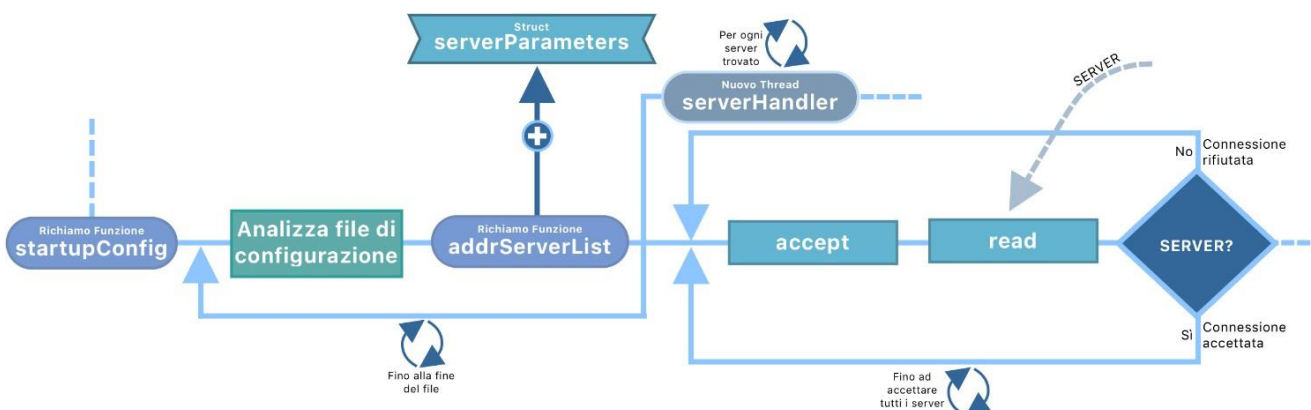
Il thread “master” si occuperà di accettare nuove connessioni in entrata da parte di altri server utilizzando i socket come canale di comunicazione, per poi salvarne i file descriptor che verranno utilizzati per lo scambio dei messaggi.

Ogni connessione in entrata viene analizzata in attesa della ricezione della stringa “SERVER” che assicura che la connessione arrivata proviene da un server.

In mancanza di tale messaggio, il master thread dedurrà che la connessione proviene da un client (o da qualche altro applicativo in esecuzione) e la richiesta sarà rifiutata.

Per ogni connessione viene salvato il socket file descriptor nell'apposito array `socketOthServer`.

Una volta che il master thread avrà accettato un numero di connessioni in entrata pari al numero dei server totali meno uno (sé stesso), passerà all'analisi delle connessioni in entrata da parte dei client.



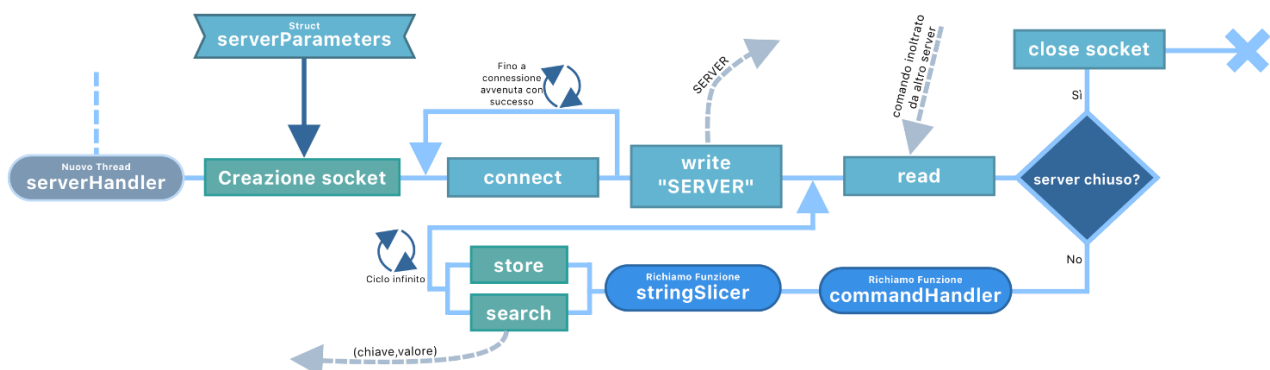
Thread Slave

Il thread “slave” invece, creeranno un nuovo socket per gestire le comunicazioni con ogni altro server presente nella lista salvata precedentemente.

Il tentativo di connessione verrà ripetuto fin quando non avrà successo.

Una volta avvenuta la connessione, verrà inviato tramite il socket la stringa “SERVER” per avvisare il server a cui ci si è collegati che la connessione richiesta è una connessione valida per la fase di startup.

Una volta che la connessione è stata accettata, il thread transita in fase di ricezione dei messaggi da parte del server a cui si è collegato.



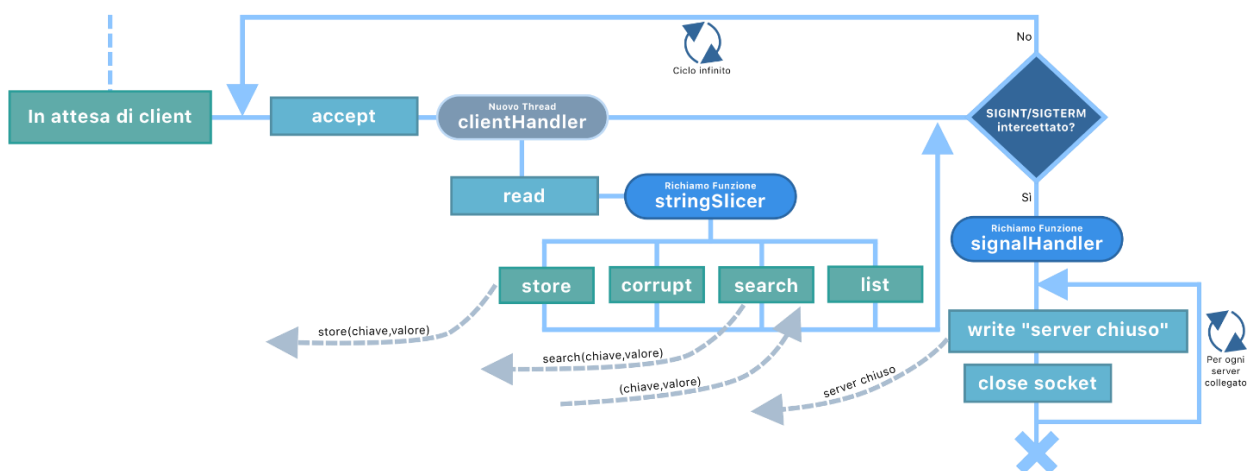
Gestione Comandi

Quando la fase di startup è completa e quindi tutti i server sono collegati tra loro, ognuno di questi si mette in ascolto per la ricezione di connessioni da parte di client.

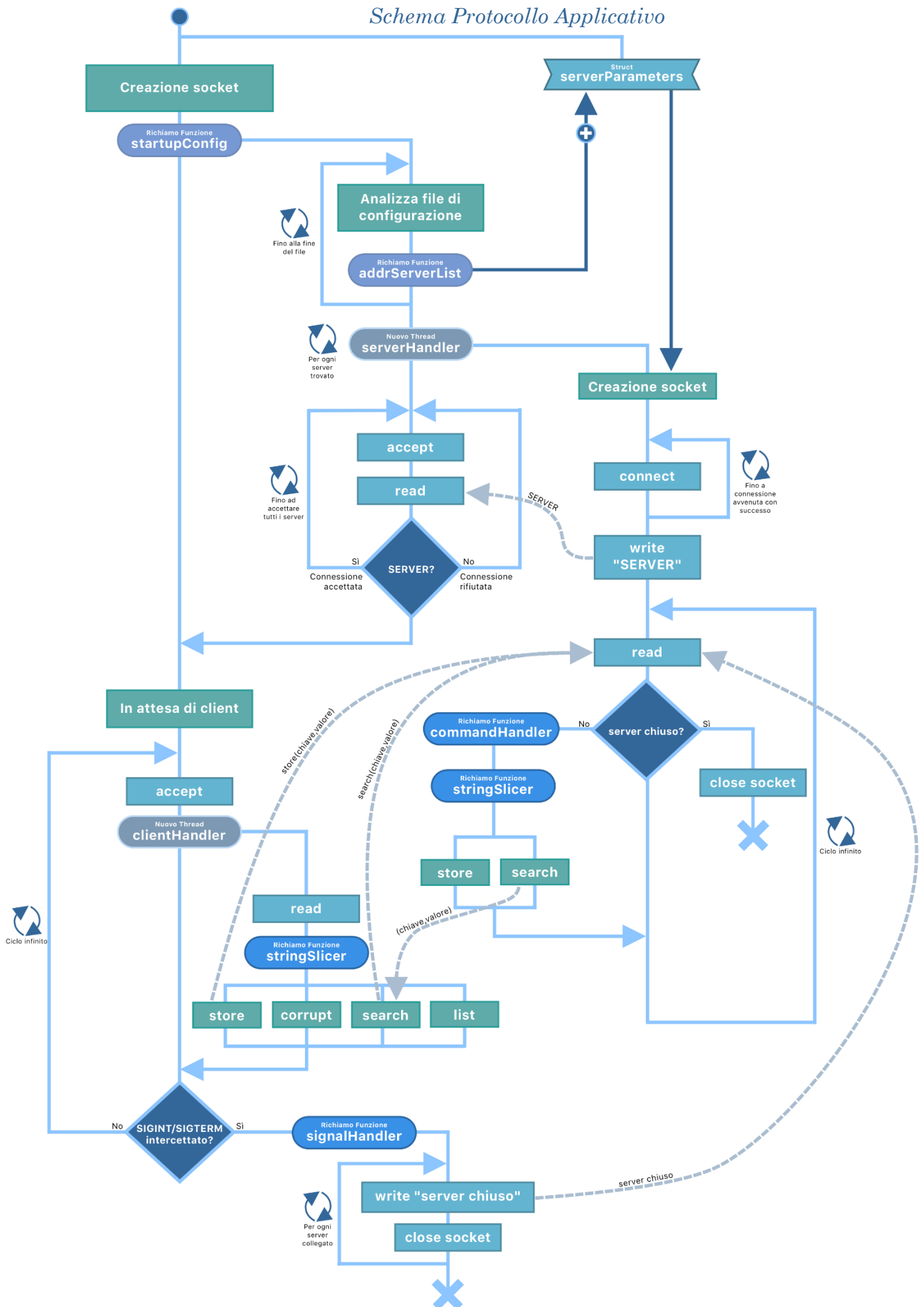
Ogni connessione in entrata da parte di un client viene gestita in thread appositi. Dal socket che gestisce le connessioni in entrata viene letto il comando, analizzato e suddiviso in opportune sottostringhe per facilitarne l'esecuzione.

In particolare:

- Il comando `store`, dopo aver effettuato opportune verifiche, memorizza la coppia nella lista locale e inoltra il comando ad ogni altro server il cui socket descriptor è stato salvato precedentemente nell'array `socketOthServer`. Gli altri server riceveranno il messaggio e salveranno anch'essi la coppia. Nel caso in cui la coppia sia già presente nel server, il comando non verrebbe inoltrato agli altri server.
- Il comando `corrupt` lavora solo sul server che riceve il comando, quindi non interagisce con gli altri server.
- Il comando `search` controlla innanzitutto se la chiave è presente nel server. In caso di risposta negativa, il server invia al client un messaggio di errore. In caso di risposta affermativa, invece, viene inoltrato il comando a tutti gli altri server che cercheranno a loro volta la coppia e la restituiranno al server che invia la richiesta. Il server che invia il comando controlla che ogni coppia ricevuta dagli altri server corrisponda a quella memorizzata su di esso: se tutte le coppie sono uguali, allora viene inviata al client la coppia cercata, altrimenti viene inviato al client il messaggio "Ledger Corrotto".
- Il comando `list` infine, al pari del comando `corrupt`, non interagisce con altri server oltre a quello ricevente.



Schema Protocollo Applicativo



CODICE INTEGRALE

Client

```

1. /*****
2. *
3. *          SIMPLE PUBLIC LEDGER
4. *          (2018/2019)
5. *
6. *          Ernesto De Crecchio (N86001596)
7. *
8. *****/
9.
10. #include <stdio.h>
11. #include <stdlib.h>
12. #include <unistd.h>
13. #include <string.h>
14. #include <sys/uio.h>
15. #include <strings.h>
16. #include <arpa/inet.h>
17.
18. #define MAXBUFFER 200
19.
20. int main (int args, char *argv[]) {
21.     system("clear");
22.
23.     char buffer[MAXBUFFER];
24.
25.     char *ipServer = argv[1];
26.     char *port = argv[2];
27.     char *command = argv[3];
28.
29.     //Controllo parametri
30.     if ( args == 6 && strcmp(command, "store") == 0) { //Controllo validità comando 'store' (richiede 2 parametri)
31.         write(STDOUT_FILENO, "* Comando STORE *\n", 18);
32.
33.         sprintf(buffer, "Chiave: %s\nValore: %s\n\n", argv[4], argv[5]);
34.         write(STDOUT_FILENO, buffer, strlen(buffer));
35.
36.         bzero(buffer, MAXBUFFER);
37.
38.         sprintf(buffer, "store(%s,%s)", argv[4], argv[5]);
39.     } else if ( args == 6 && strcmp(command, "corrupt") == 0) { //Controllo validità comando 'corrupt' (richiede 2 parametri)
40.         write(STDOUT_FILENO, "* Comando CORRUPT *\n", 20);
41.
42.         sprintf(buffer, "Chiave: %s\nNuovo Valore: %s\n\n", argv[4], argv[5]);
43.         write(STDOUT_FILENO, buffer, strlen(buffer));
44.
45.         bzero(buffer, MAXBUFFER);
46.
47.         sprintf(buffer, "corrupt(%s,%s)", argv[4], argv[5]);
48.     } else if (args == 5 && strcmp(command, "search") == 0) { //Controllo validità comando 'search' (richiede 1 parametro)
49.         write(STDOUT_FILENO, "* Comando SEARCH *\n", 19);
50.
51.         sprintf(buffer, "Chiave: %s\n\n", argv[4]);
52.         write(STDOUT_FILENO, buffer, strlen(buffer));
53.
54.         bzero(buffer, MAXBUFFER);
55.
56.         sprintf(buffer, "search(%s)", argv[4]);
57.     } else if (args == 4 && strcmp(command, "list") == 0) { //Controllo validità comando 'list' (richiede 0 parametri)

```

```

58.     write(STDOUT_FILENO, "* Comando LIST *\n", 17);
59.
60.     sprintf(buffer, "list()");
61. } else {
62.     write(STDOUT_FILENO, "[-] ERRORE! Comando non trovato o parametri errati.\n", 52);
63.     exit(-1);
64. }
65.
66. ////// INIZIALIZZAZIONE SOCKET //////
67. int sockFd; // Socket file descriptor
68. struct sockaddr_in serverAddr;
69.
70. //Apertura del socket client
71. if((sockFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
72.     perror("[-] ERRORE Socket!"), exit(-1);
73.
74. //Preparazione dell'indirizzo del socket
75. memset(&serverAddr, '\0', sizeof(serverAddr)); //Inizializza la struttura di tipo sockaddr_in a valori "n
    ulli"
76. serverAddr.sin_family = AF_INET;
77. serverAddr.sin_port = htons(atoi(port));
78. serverAddr.sin_addr.s_addr = inet_addr(ipServer);
79.
80. //Connessione del socket al server
81. if(connect(sockFd, (struct sockaddr*) &serverAddr, sizeof(serverAddr)) < 0) {
82.     perror("[-] ERRORE Connect!"), exit(-1);
83. } else {
84.     write(STDOUT_FILENO, "\n[+] Connessione stabilita.\n", 28);
85. }
86.
87. write(sockFd, buffer, strlen(buffer));
88.
89. read(sockFd, buffer, MAXBUFFER - 1);
90. if(strcmp("RIFIUTATO", buffer) == 0) {
91.     write(STDOUT_FILENO, "[-] ATTENZIONE! Il server è ancora in fase di startup.\n", 56);
92.     exit(-1);
93. }
94.
95. write(STDOUT_FILENO, "\n-----\n\n\n* RISULTATO *\n", 53);
96. bzero(buffer, MAXBUFFER);
97. while(read(sockFd, buffer, MAXBUFFER - 1) > 0) {
98.     write(STDOUT_FILENO, buffer, strlen(buffer));
99.     bzero(buffer, MAXBUFFER);
100. }
101.
102. close(sockFd);
103. }

```

Server

```

1. /*****
2. *
3. *          SIMPLE PUBLIC LEDGER
4. *          (2018/2019)
5. *
6. *          Ernesto De Crecchio (N86001596)
7. *
8. *****/
9.
10. #include <stdio.h>
11. #include <stdlib.h>
12. #include <unistd.h>
13. #include <sys/types.h>
14. #include <sys/socket.h>
15. #include <arpa/inet.h>
16. #include <string.h>
17. #include <sys/stat.h>
18. #include <fcntl.h>
19. #include <sys/uio.h>
20. #include <strings.h>
21. #include <pthread.h>
22. #include <signal.h>
23.
24. #define MAXBUFFER 200
25.
26. // - STRUCTS -
27. //Struct contenente i nodi del ledger salvati sul server attuale
28. struct nodeList {
29.     char *key;
30.     char *value;
31.     struct nodeList *nextNode;
32. };
33.
34. //Struct contenente le informazioni degli altri server a cui collegarsi per l'inoltro dei messaggi
35. struct serverParameters {
36.     char* ip;
37.     char* port;
38.
39.     struct serverParameters *nextServer;
40. };
41.
42. // - PROTOTIPI FUNZIONI PRIMARIE -
43. void startupConfig(char *fileName, int sockServer);
44. void *clientHandler(void *newSockFd);
45. void *serverHandler(void *parameters);
46. void commandHandler(char* command, int socket);
47. void signalHandler(int segnale);
48.
49. // - PROTOTIPI FUNZIONI AUSILIARIE -
50. void stringSlicer(char* buffer, char *command, char *key, char *value);
51. void addNodeList(struct nodeList **head, char* newkey, char* newValue);
52. void addServerList(struct serverParameters **head, char* newIp, char* newPort);
53. void printList(struct nodeList *head, int where);
54. void corruptNode(struct nodeList *head, char *key, char *newValue);
55. char* findValue(struct nodeList *head, char *key);
56. int isStored(struct nodeList *head, char *key);
57. void destroyServerList(struct serverParameters *head);
58. void destroyNodeList(struct nodeList *head);
59.
60. // - VARIABILI GLOBALI -
61. int *socket0thServer;
62. int serverNumber;
63. struct nodeList *head = NULL;

```

```

64. struct serverParameters *serverSockets = NULL;
65.
66. // - MUTEX -
67. pthread_mutex_t commandHandlerMutex = PTHREAD_MUTEX_INITIALIZER;
68. pthread_mutex_t clientHandlerMutex = PTHREAD_MUTEX_INITIALIZER;
69.
70. // - MAIN -
71. int main (int args, char *argv[]) {
72.     signal(SIGINT, signalHandler); // Listener segnale SIGINT
73.     signal(SIGTERM, signalHandler); // Listener segnale SIGTERM
74.
75.     system("clear"); // Pulisce il terminale dai vecchi comandi
76.
77.     // Dichiarazione variabili locali
78.     int sockFd, newsockFd, checkValue;
79.     struct sockaddr_in serverAddress, clientAddress;
80.     socklen_t addrSize = sizeof(clientAddress);
81.     char *buf = malloc(sizeof(char)*MAXBUFFER);
82.
83.     // Verifica corretto inserimento parametri di input
84.     if(args < 3 || args > 3 ) {
85.         write(STDOUT_FILENO, "[-] ERRORE! Numero di parametri errato.\n", 40), exit(-1);
86.     } else if (argv[1] == NULL || argv[2] == NULL || (strlen(argv[2]) > 4)) write(STDOUT_FILENO, "[-]
ERRORE! Ricompilare indicando il file di configurazione e la porta da utilizzare.\n", 86), exit(-1);
87.
88.     // Creazione socket per la connessione server-client
89.     if ((sockFd = socket(AF_INET, SOCK_STREAM, 0)) == -1) perror("[-] ERRORE Socket!"), exit(-1);
90.     write(STDOUT_FILENO, "[+] Server socket creato correttamente!\n", 40);
91.
92.     // Settaggio parametri struttura sockaddr_in
93.     serverAddress.sin_family = AF_INET;
94.     serverAddress.sin_port = htons(atoi(argv[2]));
95.     serverAddress.sin_addr.s_addr = inet_addr("127.000.000.001");
96.
97.     // Assegnamento nome al socket
98.     if ((checkValue = bind(sockFd, (struct sockaddr *) &serverAddress, sizeof(serverAddress))) == -
1) perror("[-] ERRORE Bind!"), exit(-1);
99.     sprintf(buf, "[+] Connesso correttamente alla porta %d\n", atoi(argv[2]));
100.    write(STDOUT_FILENO, buf, strlen(buf));
101.    bzero(buf, MAXBUFFER);
102.
103.    // Socket in ascolto
104.    if (listen(sockFd, 10) == -1) perror("[-] ERRORE Listen!"), exit(-1);
105.    write(STDOUT_FILENO, "\nServer pronto...\n\nIn attesa di collegamento con gli altri server...\n\n", 70);
106.
107.    startupConfig(argv[1], sockFd); // Connessione (in fase di avvio) tra i vari server
108.
109.    write(STDOUT_FILENO, "\n-----\nIn attesa di un client...\n\n", 40);
110.    while (1) {
111.        // Accettazione connessioni in ingresso dai client
112.        if ((newsockFd = accept(sockFd, (struct sockaddr *) &clientAddress, &addrSize)) == -1) perror("[-]
ERRORE Accept!"), exit(-1);
113.        write(newsockFd, "ACCETTATO\n", 10);
114.
115.        pthread_t tid;
116.        pthread_create(&tid, NULL, clientHandler, &newsockFd);
117.        pthread_join(tid, NULL);
118.
119.        if ((checkValue = close(newsockFd)) == -1) perror("[-] ERRORE Close!"), exit(-1);
120.    }
121.
122.    // Chiusura socket
123.    if ((checkValue = close(sockFd)) == -1) perror("[-] ERRORE Close!"), exit(-1);
124.
125.    // Deallocazione memoria
126.    free(buf);

```

```

127.
128.     return 0;
129. }
130.
131. /*****
132. *
133. *             FUNZIONI PRIMARIE
134. *
135. *****/
136.
137. /*
138.  - FUNZIONE STARTUPCONFIG -
139.  Si occupa della fase di startup dei server, analizzando il file di configurazione contenente le informazioni
    dei server a cui collegarsi,
140.  creando nuovi thread per la connessione agli altri server e gestendo le richieste di connessione in entrata.
141.
142.  -- Parametri:
143.  - fileName: Nome del file di configurazione contenente le informazioni degli altri server.
144.  - sockServer: File descriptor del socket tramite il quale accettare le nuove connessioni.
145.  -- Valore di ritorno: Nessuno.
146.  */
147. void startupConfig(char *fileName, int sockServer) {
148.     int configfd; //File descriptor del file di configurazione
149.     char *buf = malloc(sizeof(char)*MAXBUFFER);
150.
151.     if ((configfd = open(fileName, O_RDONLY)) == -1) perror("[-] ERRORE Open!"), exit(-1);
152.     off_t endPosition = lseek(configfd, 0, SEEK_END);
153.     serverNumber = endPosition/21; //Il numero dei server è dato dividendo i byte totali contenuti nel file
    per 21 in quanto ogni riga, rappresentante un server, contiene esattamente 21 caratteri
154.     off_t currentPosition = lseek(configfd, 0, SEEK_SET);
155.
156.     char ip[16];
157.     char port[5];
158.
159.     //Ciclo che analizza il file di configurazione riga per riga gestendo ogni server opportunamente
160.     while(currentPosition < endPosition) {
161.         read(configfd, buf, 20);
162.
163.         //Salva nella stringa 'ip' l'ip del server alla riga del file corrente
164.         for(int i=0; i<15; i++) {
165.             ip[i] = buf[i];
166.         }
167.         ip[15] = '\0';
168.
169.         //Salva nella stringa 'port' la porta del server alla riga del file corrente
170.         for(int j=16; j<20; j++) {
171.             port[j-16] = buf[j];
172.         }
173.         port[4] = '\0';
174.
175.         currentPosition = lseek(configfd, 1, SEEK_CUR);
176.
177.         addServerList(&serverSockets, ip, port); //Aggiunge alla lista dei server a cui il server chiamante
    dovrà collegarsi, il server appena "estrapolato" dal file di configurazione
178.
179.         bzero(buf, MAXBUFFER);
180.     }
181.
182.     //Per ogni server salvato, viene aperto il thread apposito
183.     struct serverParameters *refServer = serverSockets;
184.     while(refServer != NULL) {
185.         pthread_t serverTid;
186.
187.         pthread_create(&serverTid, NULL, serverHandler, (void *)refServer);
188.
189.         refServer = refServer->nextServer;

```

```

189.     }
190.
191.     socket0thServer = malloc(sizeof(int)*serverNumber);
192.     struct sockaddr_in serverClientAddress;
193.     socklen_t addrSize = sizeof(serverClientAddress);
194.
195.     int newsockFd;
196.     int serverConnected = 0;
197.
198.     bzero(buf, MAXBUFFER);
199.
200.     //Ciclo che si mette in attesa di richiesta di connessione da parte di tutti i server
201.     while(serverConnected <= serverNumber) {
202.         if ((newsockFd = accept(sockServer, (struct sockaddr *) &serverClientAddress, &addrSize)) == -1) {
203.             perror("[-] ERRORE Accept!");
204.             exit(-1);
205.         } else {
206.             //Ogni connessione in entrata viene analizzata ed effettivamente accettata solamente se viene ri-
207.             //cevuta la stringa 'SERVER' che evita che un client riesca ad inviare un comando prima del docuto
208.             read(newsockFd, buf, MAXBUFFER);
209.             if(strcmp("SERVER", buf) != 0) {
210.                 write(STDOUT_FILENO, "[-] Connessione in entrata rifiutata.\n", 38);
211.                 write(newsockFd, "RIFIUTATO\0", 10);
212.                 close(newsockFd);
213.             } else {
214.                 //Se la connessione viene accettata, il socket corrispondente viene salvato nell'array socke-
215.                 //t0thServer per utilizzi futuri
216.                 socket0thServer[serverConnected] = newsockFd;
217.                 serverConnected++;
218.                 sprintf(buf, "[+] Connessione in entrata accettata (%d/%d)\n", serverConnected, serverNumber
219. +1);
220.                 write(STDOUT_FILENO, buf, strlen(buf));
221.             }
222.         }
223.         bzero(buf, MAXBUFFER);
224.     }
225.     write(STDOUT_FILENO, "[+] Collegamento con tutti i server effettuato con successo.\n", 61);
226.     free(buf);
227.     close(configfd);
228. }
229.
230. /*
231.  - FUNZIONE CLIENTHANDLER -
232.  Gestisce l'intera sessione tra client e server eseguendo i vari comandi e inoltrandoli agli altri server qua-
233.  ndo necessario.
234.  -- Parametri:
235.  - newSockFd: Il socket file descriptor col quale vengono effettuati gli scambi di messaggio tra il client e
236.  il server al quale si è connesso.
237.  -- Valore di ritorno: Nessuno.
238.  */
239. void *clientHandler(void *newSockFd) {
240.     pthread_mutex_lock(&clientHandlerMutex);
241.
242.     int sockFd = *(int *)newSockFd;
243.
244.     char *buf = malloc(sizeof(char)*MAXBUFFER); //Buffer di servizio per la comunicazione dei messaggi
245.     char *command = malloc(sizeof(char)*MAXBUFFER); //Buffer contenente il comando arrivato dal client
246.     bzero(buf, MAXBUFFER);
247.
248.     read(sockFd, command, MAXBUFFER);
249.
250.     //Gestisco il comando per il server attuale
251.     char *commandType = malloc(sizeof(char)*MAXBUFFER);

```



```

250.     char *key = malloc(sizeof(char)*MAXBUFFER);
251.     char *value = malloc(sizeof(char)*MAXBUFFER);
252.
253.     stringSlicer(command, commandType, key, value); //Il comando viene suddiviso nelle opportune sottostringhe
254.
255.     if(strcmp(commandType, "store") == 0) { //Comando 'store' arrivato
256.         if(isStored(head, key) == 0) { //Controlla se esiste già nel ledger una coppia con chiave uguale a quella che il client ha chiesto di memorizzare
257.             addNodeList(&head, key, value);
258.
259.             sprintf(buf, "[+] Coppia (%s,%s) memorizzata.\n", key, value);
260.             write(STDOUT_FILENO, buf, strlen(buf));
261.             write(sockFd, buf, strlen(buf));
262.
263.             //Inoltra il comando a tutti gli altri server
264.             for(int i=0; i<=serverNumber; i++) {
265.                 write(socketOthServer[i], command, strlen(command));
266.             }
267.         } else {
268.             sprintf(buf, "[-] Chiave '%s' già presente nel ledger.\n", key);
269.             write(STDOUT_FILENO, buf, strlen(buf));
270.             write(sockFd, buf, strlen(buf));
271.         }
272.         bzero(buf, MAXBUFFER);
273.     } else if(strcmp(commandType, "corrupt") == 0) { //Comando 'corrupt' arrivato
274.         if(isStored(head, key) == 0) { //Controlla se esiste nel ledger una coppia con chiave uguale a quella che il client ha chiesto di corrompere
275.             sprintf(buf, "[-] Chiave '%s' non presente nel ledger.\n", key);
276.             write(STDOUT_FILENO, buf, strlen(buf));
277.             write(sockFd, buf, strlen(buf));
278.         } else {
279.             corruptNode(head, key, value);
280.
281.             write(STDOUT_FILENO, "[+] Coppia corrotta\n", 20);
282.             write(sockFd, "[+] Coppia corrotta\n", 20);
283.         }
284.     } else if (strcmp(commandType, "search") == 0) { //Comando 'search' arrivato
285.         if(isStored(head, key) == 0) { //Controlla se esiste nel ledger una coppia con chiave uguale a quella che il client ha chiesto di cercare
286.             sprintf(buf, "[-] Chiave %s assente.\n", key);
287.             write(STDOUT_FILENO, buf, strlen(buf));
288.             write(sockFd, buf, strlen(buf));
289.         } else {
290.             //Viene generata opportunamente una stringa contenente solo la chiave e il valore della coppia da cercare
291.             char* node = malloc(sizeof(char)*MAXBUFFER);
292.             strcat(node, "(");
293.             strcat(node, key);
294.             strcat(node, ",");
295.             strcat(node, findValue(head, key));
296.             strcat(node, ")");
297.
298.             int ledgerCorrupted = 1;
299.             //Inoltra il comando a tutti gli altri server
300.             for(int i=0; i<=serverNumber; i++) {
301.                 write(socketOthServer[i], command, strlen(command));
302.                 read(socketOthServer[i], buf, MAXBUFFER); //Il server avrà restituito una stringa del tipo '(chiave,valore)' con i dati contenuti in esso
303.
304.                 if(strcmp(node, buf) != 0) { //Per ogni coppia che gli altri server reinviano al server inoltrante, viene controllata la validità
305.                     ledgerCorrupted = 0;
306.                     break; //Se una coppia ritornata dal server è diversa da quella attuale, allora tale coppia è stata corrotta
307.                 }

```

```

308.         bzero(buf, MAXBUFFER);
309.     }
310.
311.     //Se la coppia è stata corrotta, viene mostrato il messaggio 'Ledger Corrotto', altrimenti viene
mostrata la coppia trovata
312.     if(ledgerCorrupted == 0) {
313.         write(sockFd, "Ledger Corrotto.\n", 18);
314.     } else {
315.         strcat(node, "\n");
316.         write(sockFd, node, strlen(node));
317.     }
318.     free(node);
319. }
320. } else {
321.     write(STDOUT_FILENO, "* ELENCO COPPIE *\n", 18);
322.     printList(head, STDOUT_FILENO);
323.     printList(head, sockFd);
324. }
325.
326. free(buf);
327. free(commandType);
328. free(key);
329. free(value);
330.
331. pthread_mutex_unlock(&clientHandlerMutex);
332. pthread_exit(NULL);
333. }
334.
335. /*
336.  - FUNZIONE SERVERHANDLER -
337.  Gestisce le connessioni con gli altri server analizzando i comandi inoltrati da parte di altri server
338.  -- Parametri:
339.  - parameters: Nodo della linked list serverParameters contenente le informazioni del server a cui collegarsi
340.  -- Valore di ritorno: Nessuno.
341.  */
342. void *serverHandler(void *parameters) {
343.     struct serverParameters *server = (struct serverParameters *) parameters;
344.
345.     char *buf = malloc(sizeof(char)*MAXBUFFER);
346.
347.     int sockFd;
348.     struct sockaddr_in serverAddr;
349.
350.     //Preparazione dell'indirizzo del socket
351.     memset(&serverAddr, '\0', sizeof(serverAddr)); //Inizializza la struttura di tipo sockaddr_in a valori "
nulli"
352.     serverAddr.sin_family = AF_INET;
353.     serverAddr.sin_port = htons(atoi(server->port));
354.     serverAddr.sin_addr.s_addr = inet_addr(server->ip);
355.
356.     //Ciclo che si ripete fin quando la connessione col server destinatario non è andata a buon fine
357.     int checkValue;
358.     do {
359.         if ((sockFd = socket(AF_INET, SOCK_STREAM, 0)) == -1) perror("[-] ERRORE Socket!"), exit(-1);
360.
361.         //Connessione del socket al server
362.         if((checkValue = connect(sockFd, (struct sockaddr*) &serverAddr, sizeof(serverAddr))) != 0) {
363.             close(sockFd);
364.         }
365.     } while (checkValue != 0);
366.
367.     write(sockFd, "SERVER", 6); //Messaggio inviato per avvisare il server a cui ci si sta collegando che si
è uno dei server "autorizzati" alla connessione
368.
369.     int nBytes;

```

```

370.     while(1) {
371.         bzero(buf, MAXBUFFER);
372.         nBytes = read(sockFd, buf, MAXBUFFER); //Legge il comando arrivato da uno degli altri server
373.
374.         if(nBytes > 0) {
375.             if (strcmp(buf, "Server chiuso") == 0) { //Se il comando è 'Server chiuso' allora uno dei server
376.                 //è stato chiuso (volutamente o meno) ed è necessario chiudere anche quello attuale
377.                 write(STDOUT_FILENO, "[+] Processo terminato dall'utente.\n", 36);
378.                 close(sockFd);
379.                 exit(-1);
380.             }
381.             commandHandler(buf, sockFd); //Se non è stata richiesta la chiusura del server, viene gestito il
382.             //comando effettivo
383.         }
384.     }
385.     pthread_exit(NULL);
386. }
387.
388. /*
389.  - FUNZIONE COMMANDHANDLER -
390.  Gestisce il comando inoltrato da parte di un altro server.
391.  -- Parametri:
392.  - command: Stringa contenente l'intero comando inoltrato da un altro server. Della forma comando([valore][,]
393.  [valore]).
394.  - socket: Socket descriptor utilizzato per rispondere al server da cui è arrivato il comando (necessario per
395.  il comando 'search').
396.  -- Valore di ritorno: Nessuno.
397.  */
398. void commandHandler(char* command, int socket) {
399.     pthread_mutex_lock(&commandHandlerMutex);
400.
401.     char *buf = malloc(sizeof(char)*MAXBUFFER);
402.     char *commandType = malloc(sizeof(char)*MAXBUFFER);
403.     char *key = malloc(sizeof(char)*MAXBUFFER);
404.     char *value = malloc(sizeof(char)*MAXBUFFER);
405.
406.     stringSlicer(command, commandType, key, value); //Il comando viene suddiviso nelle opportune sottostring
407.     //he
408.     //Essendo inoltrato da un altro server, il comando potrà essere solo 'store' e 'search', inutile quindi
409.     //gestire anche 'list' e 'corrupt'
410.     if(strcmp(commandType, "store") == 0) {
411.         addNodeList(&head, key, value); //Aggiunge direttamente la nuova coppia alla lista visto che i contr
412.         //olli opportuni sono stati fatti a monte dal server inoltrante
413.     } else if (strcmp(commandType, "search") == 0) {
414.         //Viene reinviata al server inoltrante la coppia chiave-
415.         //valore presente sul server attuale (che non necessariamente corrisponderà a quella salvata sul server inoltr
416.         //ante)
417.         strcpy(value, findValue(head, key));
418.         sprintf(buf, "(%s,%s)", key, value);
419.
420.         write(socket, buf, strlen(buf));
421.     }
422.
423.     free(buf);
424.     free(commandType);
425.     free(key);
426.     free(value);
427.
428.     pthread_mutex_unlock(&commandHandlerMutex);
429. }
430.
431. /*
432.  - FUNZIONE SIGNALHANDLER -

```

```

427. Gestisce l'interruzione di uno dei server terminando automaticamente la sessione di tutti i server collegati
428. .
429. -- Parametri:
430. - segnale: Variabile intera rappresentante in segnale arrivato.
431. -- Valore di ritorno: Nessuno.
432. */
433. void signalHandler(int segnale) {
434.     if (segnale == SIGINT || segnale == SIGTERM) {
435.         write(STDOUT_FILENO, "[+] Processo terminato dall'utente.\n", 36);
436.         for(int i = 0; i <= serverNumber; i++) {
437.             write(socket0thServer[i], "Server chiuso", 15);
438.             close(socket0thServer[i]);
439.         }
440.         free(socket0thServer);
441.         destroyNodeList(head); //Dealloca i nodi contenuti nel ledger
442.         head = NULL;
443.         destroyServerList(serverSockets); //Dealloca i riferimenti agli altri server
444.         serverSockets = NULL;
445.         exit(-1);
446.     }
447. }
448. }
449. }
450.
451.
452. /*****
453. *
454. *          FUNZIONI AUSILIARIE
455. *
456. *****/
457.
458. /*
459. - FUNZIONE STRINGSLICER-
460. Data come parametro una stringa contenente il comando e gli eventuali parametri, stringSlicer suddivide tale
461. stringa in modo opportuno.
462. -- Parametri:
463. - buffer: Stringa contenente l'intero comando arrivato dal client. Della forma comando([valore][,][valore]).
464. - command: Sottostringa di buffer contenente, in output, il comando.
465. - key: Sottostringa di buffer contenente, in output, la chiave (può essere vuota).
466. - value: Sottostringa di buffer contenente, in output, il valore (può essere vuota).
467. -- Valore di ritorno: command, key e value possono essere considerati come valori di ritorno "multipli".
468. */
469. void stringSlicer(char *buffer, char *command, char *key, char *value) {
470.     int bufferIndex = 0; //Indice che tiene traccia del punto in cui si è arrivati ad analizzare il buffer c
471.     ontenente l'intero comando da suddividere in sottostringhe
472.     int fieldIndex = 0; //Indice di servizio che serve a copiare la sottostringa interessata carattere per c
473.     arattere
474.     //Ciclo che separa memorizza in 'command' la parte della stringa 'buffer' corrispondente al comando
475.     while(buffer[bufferIndex] != '(') {
476.         command[fieldIndex] = buffer[bufferIndex];
477.         bufferIndex = bufferIndex + 1;
478.         fieldIndex = fieldIndex + 1;
479.     }
480.     command[fieldIndex] = '\0';
481.     fieldIndex = 0;
482.     bufferIndex = bufferIndex + 1;
483.     //Analisi della presenza di parametri
484.     if((strcmp(command, "store") == 0) || (strcmp(command, "corrupt") == 0) || (strcmp(command, "search") ==
485.     0)) { //I comandi 'store', 'corrupt' e 'search' prevedono almeno un parametro (key)

```

```

486. //Ciclo che memorizza in 'key' la parte della stringa 'buffer' corrispondente alla chiave della copp
   ia
487. while(buffer[bufferIndex] != ',' && buffer[bufferIndex] != ')') {
488.     key[fieldIndex] = buffer[bufferIndex];
489.
490.     bufferIndex = bufferIndex + 1;
491.     fieldIndex = fieldIndex + 1;
492. }
493. key[fieldIndex] = '\0';
494.
495. fieldIndex = 0;
496. bufferIndex = bufferIndex + 1;
497.
498. if((strcmp(command, "store") == 0) || (strcmp(command, "corrupt") == 0)) { //I comandi 'store' e 'co
   rrupt' prevedono un ulteriore parametro (value)
499.     //Ciclo che memorizza in 'value' la parte della stringa 'buffer' corrispondente al valore della
   coppia
500.     while(buffer[bufferIndex] != ')') {
501.         value[fieldIndex] = buffer[bufferIndex];
502.
503.         bufferIndex = bufferIndex + 1;
504.         fieldIndex = fieldIndex + 1;
505.     }
506.     value[fieldIndex] = '\0';
507. }
508. }
509. }
510.
511. /*
512.  - FUNZIONE ADDNODELIST -
513.  Aggiunge una nuova coppia chiave-
   valore alla linked list contenente i nodi del ledger attualmente memorizzati.
514.  -- Parametri:
515.  - head: Linked list contenente le coppie chiave-valore memorizzate nel ledger.
516.  - newKey: La chiave del nuovo nodo da aggiungere nel ledger.
517.  - newValue: Il valore del nuovo nodo da aggiungere nel ledger.
518.  -- Valore di ritorno: nessuno
519.  */
520. void addNodeList(struct nodeList **head, char* newKey, char* newValue) {
521.     struct nodeList *newNode = (struct nodeList *) malloc(sizeof(struct nodeList));
522.
523.     newNode->key = malloc(sizeof(char)*MAXBUFFER);
524.     newNode->value = malloc(sizeof(char)*MAXBUFFER);
525.     newNode->nextNode = NULL;
526.
527.     strcpy(newNode->key, newKey);
528.     strcpy(newNode->value, newValue);
529.
530.     struct nodeList *ref = *head;
531.
532.     if (*head == NULL) {
533.         *head = newNode;
534.         return;
535.     }
536.
537.     while (ref->nextNode != NULL)
538.         ref = ref->nextNode;
539.
540.     ref->nextNode = newNode;
541. }
542.
543. /*
544.  - FUNZIONE ADDSERVERLIST -
545.  Aggiunge alla linked list 'serverParameters' l'ip e la porta di un server alla quale l'esecutore dovrà colle
   garsi.
546.  -- Parametri:

```

```

547. - head: Linked list contenente le informazioni dei server.
548. - newIp: Ip del nuovo server da aggiungere alla lista.
549. - newPort: Porta del nuovo server da aggiungere alla lista.
550. -- Valore di ritorno: Nessuno
551. */
552. void addServerList(struct serverParameters **head, char* newIp, char* newPort) {
553.     struct serverParameters *newServer = (struct serverParameters *) malloc(sizeof(struct serverParameters))
554.     ;
555.     newServer->ip = malloc(sizeof(char)*MAXBUFFER);
556.     newServer->port = malloc(sizeof(char)*MAXBUFFER);
557.     newServer->nextServer = NULL;
558.
559.     strcpy(newServer->ip, newIp);
560.     strcpy(newServer->port, newPort);
561.
562.     struct serverParameters *ref = *head;
563.
564.     if (*head == NULL) {
565.         *head = newServer;
566.         return;
567.     }
568.
569.     while (ref->nextServer != NULL)
570.         ref = ref->nextServer;
571.
572.     ref->nextServer = newServer;
573. }
574.
575. /*
576. - FUNZIONE PRINTLIST -
577. Stampa tutte le coppie memorizzate nel ledger in ordine di memorizzazione sul file descriptor indicato come
578. parametro.
579. -- Parametri:
580. - head: Linked list contenente le coppie chiave-valore memorizzate nel ledger.
581. - where: File descriptor sul quale si vuole stampare (es STDOUT_FILENO, un socket).
582. -- Valore di ritorno: Nessuno
583. */
584. void printList(struct nodeList *head, int where) {
585.     char buffer[MAXBUFFER];
586.     struct nodeList *ref = head;
587.
588.     while (ref != NULL) {
589.         bzero(buffer, MAXBUFFER);
590.         sprintf(buffer, "Key: %s - Value: %s\n", ref->key, ref->value);
591.         write(where, buffer, strlen(buffer));
592.         ref = ref->nextNode;
593.     }
594. }
595.
596. /*
597. - FUNZIONE CORRUPTNODE -
598. Data una linked list, una chiave ed un valore, la funzione corruptNode cambia il valore della coppia che ha
599. la chiave corrispondente a quella passata come parametro (key) con il valore passato come parametro (newValue).
600. -- Parametri:
601. - head: Linked list contenente le coppie chiave-valore memorizzate nel ledger.
602. - key: La chiave della coppia che si vuole "corrompere".
603. - newValue: Il valore che la coppia assumerà dopo il comando corruptNode.
604. -- Valore di ritorno: Nessuno.
605. */
606. void corruptNode(struct nodeList *head, char *key, char *newValue) {
607.     struct nodeList *ref = head;
608.
609.     while (ref != NULL) {

```

```

609.         if(strcmp(ref->key, key) == 0) {
610.             strcpy(ref->value, newValue);
611.             break;
612.         }
613.         ref = ref->nextNode;
614.     }
615. }
616.
617. /*
618. - FUNZIONE FINDVALUE -
619. Data una linked list e una chiave, la funzione findNode cerca una coppia che ha la chiave corrispopndente
620. a quella passata come parametro (key) e ne restituisce il valore (se presente).
621. -- Parametri:
622. - head: Linked list contenente le coppie chiave-valore memorizzate nel ledger.
623. - key: La chiave della coppia di cui si vuole cercare il valore.
624. -
625. - Valore di ritorno: Una stringa contenente il valore della coppia se è presente una coppia con chiave corri
626. spondente a quella passata come parametro, NULL altrimenti.
627. */
628. char* findValue(struct nodeList *head, char *key) {
629.     struct nodeList *ref = head;
630.     while (ref != NULL) {
631.         if(strcmp(ref->key, key) == 0) {
632.             return ref->value;
633.         }
634.         ref = ref->nextNode;
635.     }
636.     return NULL;
637. }
638.
639. /*
640. - FUNZIONE ISSTORED -
641. Data come parametro una linked list e una chiave, la funzione isStored verifica se nella linked list è prese
642. nte una coppia che ha la chiave corrispondente a quella passata come parametro (key).
643. -- Parametri:
644. - head: Linked list contenente le coppie chiave-valore memorizzate nel ledger.
645. - key: La chiave della coppia di cui si vuole verificare la presenza nella linked list.
646. -
647. - Valore di ritorno: 1 se esiste una coppia con chiave corrispondente a quella passata alla funzione, 0 altr
648. imenti.
649. */
650. int isStored(struct nodeList *head, char *key) {
651.     struct nodeList *ref = head;
652.     int isFound = 0;
653.     while (ref != NULL) {
654.         if(strcmp(ref->key, key) == 0) {
655.             isFound = 1;
656.             break;
657.         }
658.         ref = ref->nextNode;
659.     }
660.     return isFound;
661. }
662.
663. /*
664. - FUNZIONE DESTROYNODELIST -
665. Dealloca tutti le coppie chiave-valore memorizzati nel server chiamante.
666. -- Parametri:
667. - head: Linked list contenente le coppie chiave-valore memorizzate nel ledger.
668. - Valore di ritorno: Nessuno.
669. */
670. void destroyNodeList(struct nodeList *head) {

```

```

670.     if (head == NULL) return;
671.     struct nodeList *temp;
672.
673.     temp = head->nextNode;
674.     free(head->key);
675.     free(head->value);
676.     free(head);
677.
678.     destroyNodeList(temp);
679. }
680.
681. /*
682.  - FUNZIONE DESTROYSERVERLIST -
683.  Dealloca tutte le informazioni degli altri server collegati al server chiamante.
684.  -- Parametri:
685.  - head: Linked list contenente le informazioni degli altri server collegati al server chiamante.
686.  -- Valore di ritorno: Nessuno.
687.  */
688. void destroyServerList(struct serverParameters *head) {
689.     if (head == NULL) return;
690.     struct serverParameters *temp;
691.
692.     temp = head->nextServer;
693.     free(head->ip);
694.     free(head->port);
695.     free(head);
696.
697.     destroyServerList(temp);
698. }

```