# Contents

# 1 Introduzione

Contact Tracing è un progetto di reti di calcolatori, basato sull'utilizzo delle socket in linguaggio C che ha lo scopo d'individuare il tracciamento dei contatti. Presenta un'architettura peer to peer in cui ogni peer è identificato attraverso un id alfanumerico di 64 byte.
Per entrare nella rete ogni peer deve registrarsi presso un server, che gestisce le richieste inoltrate dai peer. L'interazione tra il server e i peer avviene tramite socket TCP in modo da garantire affidabilità nella trasmissione.
I dati che i peer scambiano con il server o i restanti peer sono rappresentati da pacchetti di diverso tipo, che variano a seconda delle azioni da eseguire.
In tutti i casi però, i pacchetti sono formati da un header che definisce il tipo di messaggio, e da un body che può essere un indirizzo, un id o altro.
Il contatto tra due generici peer della rete è identificato dagli id alfanumerici precedentemente generati.
I contatti avvenuti sono salvati da ambo i peer coinvolti.
Il server, ha l'ulteriore compito di segnalare tramite un messaggio, i peer infetti. Quest'ultimi procederanno a informare i peer della rete riguardo la propria condizione di positività. Ciò è realizzato attraverso una trasmissione broadcast dove gli infetti inviano la lista dei peer con cui sono stati in contatto.
I peer che sono in ascolto sulla rete riceveranno quindi tale lista e controlleranno se vi figurano. In caso affermativo, sarà mostrato un messaggio all'utente.

# 2 Schemi del protocollo applicazione

Il protocollo implementato a livello applicazione prevede l'utilizzo di diversi tipi di pacchetti che saranno illustrati di seguito. Ogni pacchetto è contraddistinto da un particolare header, il cui valore sarà utilizzato per effettuare una verifica sulla tipologia di pacchetto ricevuto ed eventualmente scartarlo.

## 2.1 Pacchetti Server

In questa sezione vengono discussi i pacchetti gestiti dal server:

| msg_num_all_t | |
|---|---|
| Header | ACK_MSG_ALL: msg_type_t |
| Body | - noPeer: unsigned int |

| msg_type_t | |
|---|---|
| Header | MSG_POS: msg_type_t |
| | |

| msg_peer_t | |
|---|---|
| Header | ACK_MSG_UP: msg_type_t |
| Body | - peerAddr: struct sockaddr_in<br>- positivityAddr: struct sockaddr_in |

| msg_neigh_id_t | |
|---|---|
| Header | MSG_BRD: msg_type_t |
| Body | - id[ID_LEN]: Array di char |

- **msg_null_all_t**: utilizzato nella fase preliminare di un contatto, tale pacchetto consente al Peer di conoscere il numero dei Peer connessi alla rete in quel momento. L'informazione è nota al Server che provvederà a valorizzare tale campo e inviarlo al Peer.
  Successivamente, il Server invierà la lista di Peer connessi in quel momento. La lista è stampata a video all'utente che selezionerà il peer da contattare. Tale meccanismo è una conseguenza del fatto che, per questioni di sicurezza, solo il Server memorizza le informazioni relative ai Peer della rete.
  Pertanto, sotto tale punto di vista, i Peer possono considerarsi stateless.

- **msg_type_t**: tramite questo pacchetto il Server comunica l'eventuale positività al Peer.

- **msg_peer_t**: il pacchetto viene inviato al Peer come risposta della richiesta d' ingresso nella rete. Al suo interno troviamo due indirizzi,
  uno su cui fare bind e restare in attesa della segnalazione da parte del Server di una eventuale positività, e un altro che permette di comunicare all'interno della rete.
- **msg_neigh_id_t**: è un pacchetto utilizzato per il broadcast. Al suo interno troviamo la lista degli id dei vicini con cui è stato in contatto il Peer diventato positivo.

## 2.2 Pacchetti Peer:

Di seguito i pacchetti gestiti dai Peer:

| msg_type_t | |
|---|---|
| Header | MSG_UP: msg_type_t |
| | |

| msg_num_all_t | |
|---|---|
| Header | MSG_ALL: msg_type_t |
| | |

| msg_type_t | |
|---|---|
| Header | MSG_POS: msg_type_t |
| | |

| msg_neigh_id_t | |
|---|---|
| Header | MSG_CONN: msg_type_t |
| Body | - id[ID_LEN]: Array di char |

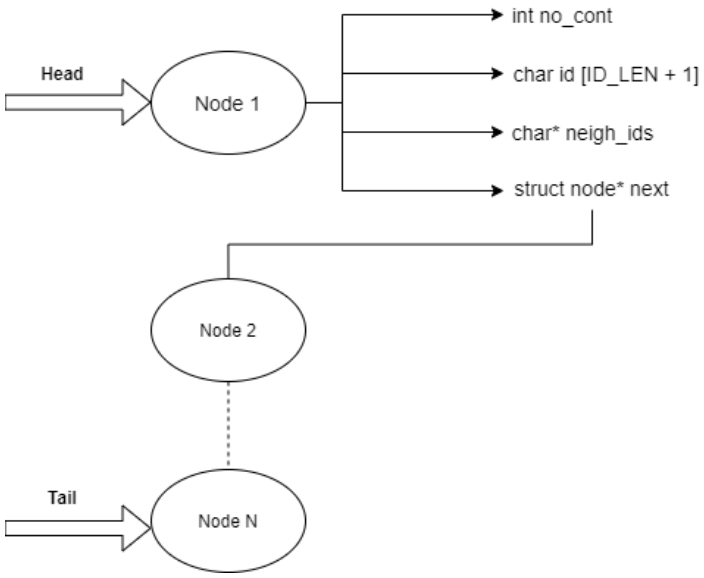
| msg_neigh_id_t | |
|---|---|
| Header | ACK_CONN: msg_type_t |
| Body | - id[ID_LEN]: Array di char |

- **msg_type_t**: questo pacchetto viene utilizzato per richiedere l'ingresso alla rete.

- **msg_num_all_t**: nel momento in cui è stato deciso di contattare qualcuno all'interno della rete, tramite questo pacchetto il Peer richiede lista dei Peer connessi in quel momento.
- **msg_type_t**: un Peer riceve questo pacchetto quando il Server segnala la positività.
- **msg_neigh_id_t**: quando due Peer comunicano, si scambiano i propri id in modo da tracciare il contatto. All'interno del pacchetto troviamo l'id che dobbiamo inviare/ricevere.

# 3 Struttura del peer

La struttura dati utilizzata dal peer per la memorizzazione degli id e delle informazioni connesse è una lista lineare gestita da una testa che punta al nodo iniziale ed una coda che punta al nodo finale.
Ogni nodo sarà una struct di tipo *node_t* contenente i seguenti campi:
1) char id[ID_LEN + 1];
2) char *neigh_ids;
3) int no_cont;
4) struct node *next;




Il primo campo contiene una stringa di 64 caratteri, generata periodicamente in maniera random, che rappresenta l'id del peer. Il secondo campo è un puntatore a char, allocato dinamicamente, il quale memorizzerà gli id relativi ai contatti avvenuti, effettuando una concatenazione di stringhe. L'allocazione dinamica è regolata sulla base del valore del terzo parametro che verrà incrementato ad ogni nuovo contatto.

Per effettuare operazioni di manipolazione e ricerca, ogni id è separato da un ulteriore carattere, "_", utilizzato, quindi, come delimitatore.
L'aggiunta di nuovi nodi alla lista avviene alla generazione di un nuovo id ed è scandita da un'apposita funzione che valorizzerà il primo campo.
La struttura proposta offre flessibilità, poichè permette di conoscere i contatti effettuati fissato un id ed effettuare le principali operazioni in un tempo lineare.

# 4 Threads

Tutti i thread sono stati creati con l'attributo *PTHREAD_CREATE_DETACHED*, dato che svolgono funzioni non interattive e non necessitano di meccanismi di sincronizzazione tra thread. Tutte le strutture dati condivise sono state utilizzate in mutua esclusione, evitando problemi di incosistenza di dati. Inoltre, i thread non effettuano busy wait, poichè effettuano chiamate bloccanti o restano in attesa su un timer.
*Link to diagram*

## 4.1 Thread del Server:

- **signalPositive**: tramite questo thread il Server individua in modo casuale quale Peer all'interno della rete è infetto tra quelli attualmente connessi. Individuato il positivo, invia la notifica per segnalare l'emergenza.

## 4.2 Thread dei Peer:

- **addNode**: il thread è impiegato per la generazione degli id temporali. Ogni 60 secondi il thread alloca memoria per contenere un nuovo id e una nuova lista degli id dei vicini, successivamente si occupa di generare l' identificativo di 66 bytes (64 per il reale identificativo, 1 per il token utile per separare gli id nella lista, 1 per il terminatore di stringa).

- **recvId**: per permettere la comunicazione tra Peer è necessario che un thread si occupi di restare in attesa di un eventuale messaggio. In seguito alla ricezione del messaggio, il Peer memorizza l'identificativo del nuovo vicino all'interno della sua lista e invia una risposta con all' interno il suo attuale id. Come parametro al thread viene passato l'indirizzo e la porta che il peer ha ricevuto dal Server per comunicare nella rete. Terminata la comunicazione, il peer ritorna in attesa di una successivo contatto.

- **listBroadcast**: con questo thread il Peer si mette in ascolto di un messaggio in broadcast. Nel caso in cui arrivasse un messaggio, il peer controlla innanzitutto l'header del pacchetto e successivamente se uno degli id generati fino a quel momento è contenuto nella lista ricevuta. Terminata la ricerca, il thread torna in attesa di un successivo messaggio.

- **sendBroadcast**: il peer è incaricato di restare in ascolto di una notifica da parte del Server che comunica l'eventuale positività.

In seguito alla notifica, il peer deve necessariamente inviare in broadcast la lista dei suoi contatti. Al thread viene consegnato l'indirizzo dedicato alla comunicazione con il Server per il messaggio di positività. Infine torna in attesa di un'ulteriore notifica.

# 5 Scelte implementative lato Client

Il lato client di Contact Tracing è rappresentato dai peer e dall'interazione tra essi e con il server. In fase di inizializzazione il peer per entrare nella rete deve registrarsi presso il server; ciò avviene nella funzione:

```
struct sockaddr_in enterIntoNetwork(int *fd_server,
    struct sockaddr_in *check_pos_addr);
```

All'interno della function viene creato il socket che consente al peer di poter comunicare con il server. Il client contatterà il server inviando, tramite la fullWrite, un pacchetto di tipo *msg_type_t* valorizzato *MSG_UP*. Successivamente il server risponderà inviando un pacchetto di tipo *msg_peer_t* valorizzato *ACK_MSG_UP* contenente due indirizzi: uno utilizzato per la segnalazione di una eventuale positività, e un altro che permette la comunicazione tra i peer all'interno della rete.
Il peer, tramite la fullRead, riceve il pacchetto controllando se è di tipo *msg_peer_t* valorizzato *ACK_MSG_UP* e in tal caso procede alla memorizzazione dei due indirizzi. La funzione utilizza il protocollo TCP e garantisce la scrittura degli esatti bytes da dover inviare.

Una volta entrato nella rete, il peer può procedere a contattare un altro peer attraverso la funzione:

```
void contactNeighbour(int fd_server, const struct
    sockaddr_in *myAddr);
```

Inizialmente vengono creati due pacchetti; il primo di tipo *msg_num_all_t* che serve al peer per conoscere il numero di nodi connessi alla rete in quel momento, e il secondo di tipo *msg_type_t* valorizzato *MSG_ALL* utilizzato per contattare il server tramite la fullWrite. Quest'ultimo invia al peer un messaggio di risposta di tipo *msg_num_all_t* valorizzato *ACK_MSG_ALL* contenente il numero dei peer connessi alla rete e successivamente con un'altra fullWrite invia la lista dei peer connessi alla rete. Il peer legge la risposta ricevuta dal server e con lo stesso controllo effettuato nella funzione *enterIntoNetwork* verifica il tipo del pacchetto ricevuto.
Se l'header di tale pacchetto corrisponde a *ACK_MSG_ALL*, viene creato un array di struct sockaddr_in (*msg_peer_to_recv*) di dimensione pari al numero di peer connessi nella rete. Con una *memcmp* è confrontato l'indirizzo del peer da contattare con l'indirizzo del peer corrente in modo da permettere all'utente di selezionare correttamente un peer valido a cui connettersi.

A questo punto l'utente sceglie il peer da contattare che viene passato come parametro d'input alla funzione *sendId*:

```
void sendId(&msg_peer_to_recv[input]);
```

La funzione è stata sviluppata per consentire lo scambio d'id tra il peer chiamante con il peer passato in input. È stato scelto un socket *UDP* in quanto l'interazione tra le due parti è di tipo richiesta/risposta ma anche per alleggerire il carico a livello di trasmissione. Il pacchetto utilizzato per questa function è di tipo *msg_neigh_id_t* valorizzato *MSG_CONN* dove al suo interno viene copiato solo l'ultimo id generato per il peer chiamante. Attraverso la *sendto* il peer chiamante invia il proprio id al peer che l'utente ha scelto di contattare.

Successivamente si mette in attesa di ricevere l'id del peer contattato tramite l'utilizzo della funzione recvfrom. Se l'header del pacchetto ricevuto è *ACK_CONN*, il peer chiamante procede con la memorizzazione dell'id del peer contattato attraverso la funzione *writeNeighbourID* che ha il compito di concatenare tutti gli id dei peer con cui è avvenuto un contatto.

Siccome la memorizzazione dell'id viene svolta da ambo i peer coinvolti, è stata implementata la funzione recvId:

```
void *recvId(void *args);
```

La funzione viene invocata da un thread[1] ed è utilizzata dal peer che è stato scelto per il contatto. L'obiettivo è quello di mettere il peer in ascolto sul socket in attesa di un contatto da un altro peer che arriverà dalla sendId. Con l'utilizzo della recvfrom, il peer riceve l'id dal peer chiamante, controllando se l'header del pacchetto ricevuto è di tipo *MSG_CONN* e in caso affermativo procede con la memorizzazione dell'id attraverso la funzione *writeNeighbourID*.
Successivamente, il peer contattato invia, tramite la sendto, il suo id al peer chiamante che lo riceverà attraverso la recvfrom situata nella function sendId.

Con la funzione *sendBroadcast*, eseguita da un thread, il peer si mette in attesa di un messaggio di positività da parte del Server:

```
void *sendBroadcast(void *args);
```

Inizialmente vengono create tre strutture di tipo sockaddr_in:
1) La prima (*check_pos_addr*) contentente l'indirizzo che viene utilizzato per consentire la comunicazione con il server che invierà un messaggio di positività.
2) La seconda (*myAddr*) oggetto di bind sul canale di comunicazione per il broadcast .
3) La terza (*broadAddr*) contenente l'indirizzo utilizzato dai peer per mettersi in ascolto sul broadcast.

Successivamente viene invocata due volte la function *buildSocket* per consentire la costruzione di due socket, uno utilizzato per ricevere il messaggio dal

[1] vedi sezione 4

Server, e l'altro per inviare il pacchetto in broadcast contenente la lista degli id relativi ai contatti avvenuti. Il peer controlla se tale messaggio corrisponde ad un pacchetto di tipo *msgToRecv* valorizzato *MSG_POS*, e in caso affermativo informerà i peer nella rete riguardo la sua positività. In particolare invierà un numero di pacchetti, contenenti i rispettivi id, pari al numero di contatti avvenuti.

Un altro thread gestisce la ricezione degli id potenzialmente infetti (id dei peer che sono stati in contatto con un peer infetto), la cui firma è:

```
void *listBroadcast();
```

Tramite la recvfrom, il peer riceve un pacchetto dalla funzione *sendBroadcast* invocata da un peer positivo e se l'header del pacchetto corrisponde a *MSG_BRD*, procede con la ricerca del proprio id all'interno della lista.
Tale ricerca avviene invocando la funzione *searchNeighbour*:

```
int searchNeighbour(char *target_id, struct
    sockaddr_in *previousSender, struct sockaddr_in *
    positiveSender, int *toSkip)
```

Il primo argomento della funzione contiene l'id ricevuto, il secondo paramentro contiene l'indirizzo del peer relativo alla precedente ricezione, il terzo parametro identifica l'indirizzo del peer positivo e, infine, l'ultimo paramentro di tipo I/O è una variabile che consente di evitare inutili confronti superflui.
La function attraverso un costrutto iterativo scorre l'intera lista d'id generati confrontandoli con l'id ricevuto in input. Una volta che viene individuato un match segnala al peer di essere stato in contatto un peer infetto settando la variabile toSkip.

Di seguito sono riportate le funzioni secondarie relative alla gestione delle strutture dati utilizzate dai peer:
1) void *addNodeToList(): creazione di un nodo[2] contenente l'id generato.
2) void deleteNodeList(): eliminazione dell'intera lista.
3) void writeNeighbourID(char *neigh_id): aggiunta dell'id del contatto.
4) int printGeneratedId(): stampa degli id generati.
5) void printPeersContacted(): stampa della lista dei contatti per quel nodo.
6) void generate_id(size_t size, char *str): generazione pseudocasuale dell'id.
7) int generateMenu(): menù di scelta presentato all'utente.

[2]vedi sezione 3

# 6 Manuale utente

Il progetto è stato realizzato utilizzando un IDE moderno, Clion. A differenza di altri IDE, esso fornisce una suite di debugging molto interessante e tool di code analysis e smart refactoring che ci hanno permesso di aumentare la qualità e la leggibilità del codice.

## 6.1 Istruzioni per la compilazione

E' stato utilizzato il build tool CMake, che ricordiamo utilizza scripts chiamati *CMakeLists* per generare gli eseguibili per uno specifico environment (makefiles su macchine Unix ad esempio). Nella project root è presente un file *CMakeLists.txt* che contiene il nome del progetto, librerie da linkare (-pthread.h) e la lista dei sorgenti da buildare.
Il progetto è stato buildato spostandosi nel path *./cmake-build-debug* e lanciando da terminale il comando ***make***.
Tuttavia, generalmente, il Makefile generato su una macchina non è portabile. Per ovviare a ciò, nella project root è stato creato uno script (*buildWithGcc.sh*) che utilizza ***gcc***.

## 6.2 Istruzioni per l'esecuzione

Dopo la fase di compilazione, si procede, innanzitutto, ad avviare il server e successivamente uno o più peer. E' possibile utilizzare lo script precedente, indicando, per l'esecuzione del peer, il numero di istanze desiderate.[3]
Il server mostrerà un messaggio, restando in attesa di richieste da parte dei peer. Al lancio dei peer, come precedentemente illustrato, essi invieranno un messaggio al server per entrare a far parte della rete e il server mostrerà a video l'arrivo dei peer.
Quest'ultimi resteranno in attesa di un input da parte dell'utente che dovrà inserire una delle opzioni mostrate dal menù di scelta.

[3] è consigliato un numero basso, siccome lo script aprirà n volte la shell

# 7 Test cases

Di seguito sono elencati casi di test, ordinati per difficoltà crescente, utili per illustrare in concreto il comportamento. Tali test hanno consentito di potenziare la robustezza del codice ed individuare soluzioni al verificarsi di eventi che non erano emersi durante le fasi iniziali di progettazione.

## 7.1 Test 1: singolo peer e server offline

Entità presenti nella rete:

1. Peer A

**Contatti avvenuti**: nessuno.

**Risultato atteso**: è mostrato all'utente un messaggio d'errore poichè non riesce a contattare il Server per entrare nella rete P2P.

## 7.2 Test 2: singolo peer

Entità presenti nella rete:

1. Server
2. Peer A

**Contatti avvenuti**: A cerca di contattare un Peer.

**Risultato atteso**: è mostrato all'utente un messaggio d'errore poichè non è presente un Peer da poter contattare.

## 7.3 Test 3: peer inattivo risultato positivo

Entità presenti nella rete:

1. Server
2. Peer A: in stato di idle

**Contatti** avvenuti: nessuno.

**Risultato** atteso: dopo T secondi, il Server comunica al Peer A la positività; quest'ultimo mostrerà un messaggio all'utente. Esso, tuttavia, non invierà un messaggio di broadcast contenente i contatti avvenuti, perchè nella rete non è presente nessun altro Peer.
Tale comportamento è dovuto a questioni di performance.

### 7.4 Test 4: due peer, di cui uno positivo

Entità presenti nella rete:

1. Peer A
2. Peer B
3. Server: notifica la positività al peer B

**Contatti** avvenuti: A <—> B

**Risultato** atteso: B riceve la notifica della positività dal Server e mostra a video un messaggio, informando l'utente. Successivamente invierà in broadcast un messaggio contenente gli dei Peer con cui è venuto in contatto. Il Peer A riceverà tale messaggio, verificando la presenza del proprio id all'interno di esso e mostrerà a video un messaggio "——".

### 7.5 Test 5: due peer, di cui uno positivo con duplice contatto

Entità presenti nella rete:

1. Peer A
2. Peer B
3. Server: notifica la positività al peer B

**Contatti** avvenuti: A <—> B ; A <—>B ; ...

**Risultato** atteso: il risultato è duale al caso di Test 4. Sarà mostrato un solo messaggio a video, in quanto il Peer A memorizza l'indirizzo dell'ultimo Peer positivo con cui è entrato in contatto. Pertanto, alla seconda ricezione (o successiva), A scarterà il messaggio ricevuto, evitando una ricerca di costo lineare.

### 7.6 Test 6: tre peer, di cui uno positivo con triplice contatto su due peer

1. Peer A
2. Peer B
3. Peer C
4. Server: notifica la positività al peer A

**Contatti** avvenuti: A <—> B ; A <—>C ; A <—>B

**Risultato** atteso: il Peer A invierà in broadcast la lista di contatti. B riceve il messaggio e mostra un solo messaggio a video (vedi Test 5). Analogamente anche C riceverà il messaggio inviato da A e stamperà un messaggio a video poichè, come B, ha avuto un contatto con un Peer positivo.

# 8 Implementazioni future.

Sono state individuate le seguenti implementazioni future:

- Gestione della sincronizzazione dei thread: anche se con una buona gestione dei pacchetti, i thread sono stati sincronizzati in modo sufficiente, è stato pensato di migliorare questo aspetto per aumentare le performance ed evitare inconsistenze.
- Gestione degli id: durante lo sviluppo sono stati discussi diversi modi su come memorizzare la lista degli id, anche se l'attuale scelta permette un corretto funzionamento del programma, è stato considerato come futuro perfezionamento.
- Gestione invio positività: come ottimizzazione futura è stato deciso di evitare l'utilizzo di un secondo indirizzo da parte del Peer per restare in attesa della notifica del Server, ed renderlo univoco.
- Feature di quarantena: è una feature di contesto che impedirebbe al Peer positivo di effettuare nuovi contatti per X secondi, simulando quindi un periodo di quarantena.