Progetto di
Laboratorio di Sistemi Operativi (mod. B)

Applicazione client/server che simula il comportamento di un **Database Server**

Ferrara Francesco Saverio
566/811
fsterrar@studenti.unina.it

prof. Giovanni Verzino
AA. 2004/2005
Gruppo 2

INTRODUZIONE

Il programma in questione si divide in due moduli:

- Un server: conserva un database costituito da un'unica tabella fatta da contatti di una rubrica. Per ogni contatto conserviamo: nome, cognome, indirizzo, telefono, e-mail, e un campo speciale "marcatura" che indica se il record è nuovo o marcato per la cancellazione.
- Un client: per interrogare il database del server. Prevede le seguenti operazioni: "stampa tutti i contatti", "cerca un contatto", "modifica un contatto", "nuovo contatto" ,"cancella un contatto", e "compact tabella".

Prima di introdurre singolarmente il server e il client è bene dare una panoramica generale sul protocollo di comunicazione usato e le strutture dati che vengono scambiate tramite socket.
Tra i processi può avvenire uno scambio di dati contenuto nelle seguenti strutture dati:

- Record contenente i dati di un contatto nella rubrica:

```
struct contatto {
	char nome[MAX_NOME];
	char cognome[MAX_COGNOME];
	char indirizzo[MAX_INDIRIZZO];
	char telefono[MAX_TELEFONO];
	char email[MAX_EMAIL];
	char marcatura;
};
```

- Siccome nella tabella la chiave è [nome, cognome], bastano solo questi due valori per individuare univocamente un contatto:

```
struct ricerca_contatto {
	char nome[MAX_NOME];
	char cognome[MAX_COGNOME];
};
```

- Per lo scambio di comandi e altre informazioni sono state definite:

```
#define CMD_STAMPA 'a'
#define CMD_CERCA 'b'
#define CMD_MODIFICA 'c'
#define CMD_NUOVO 'd'
#define CMD_CANCELLA 'e'
#define CMD_COMPACT 'f'
#define CMD_EXIT 'g'
#define SIG_SUCCESS 'h'
#define SIG_FAILURE 'i'
#define SIG_TERM 'l'
#define SIG_NULL 'm'
```

Le funzioni utilizzate per scambiare i dati tra client e server sono quattro. Le prime due servono ad inviare oppure ricevere un record:

```
int ricevi(int id_connessione, void *buffer, int dimensione) {
	int count=0;
	int byte_letti=0;

	while(count < dimensione) { //finche' tutti i byte non vengono elaborati
		if((byte_letti=read(id_connessione, buffer, dimensione - count)) > 0) {
			count += byte_letti; //conto i byte processati
			buffer += byte_letti; //aggiorno la posizione del buffer
		}
		else
			return 1;
	}

	return 0;
}
```

```
int trasmetti(int id_connessione, void *buffer, int dimensione)
{
	int count=0;
	int byte_scritti=0;

	while(count < dimensione) {  //finche' tutti i byte non vengono elaborati
		if((byte_scritti=write(id_connessione, buffer, dimensione - count)) > 0) {
			count += byte_scritti; //conto i byte processati
			buffer += byte_scritti; //aggiorno la posizione del buffer
		}
		else
			return 1;
	}

	return 0;
}
```

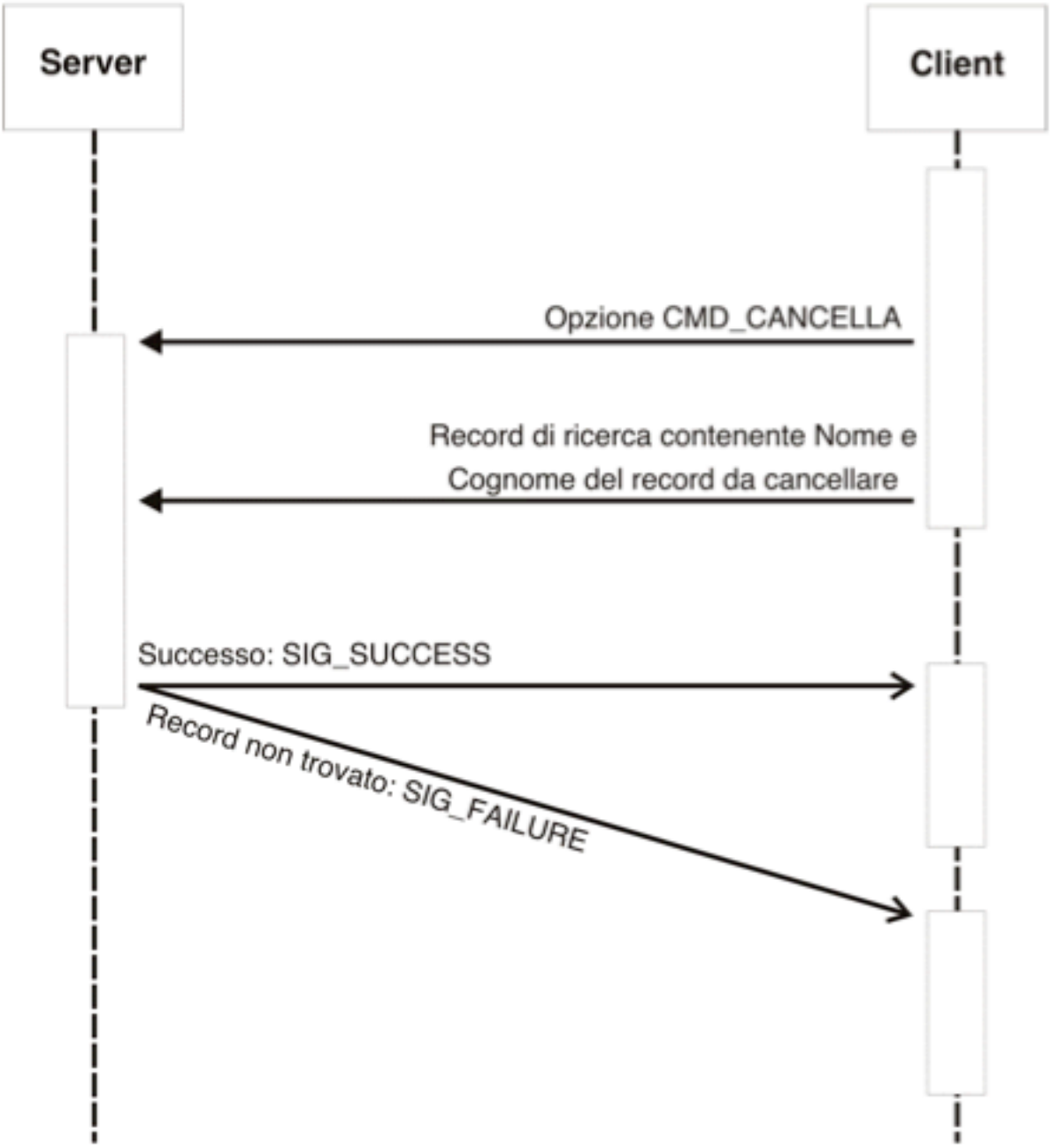
Le altre due sono usate per scambiarsi le “opzioni” ossia dei caratteri speciali per lo scambio di comandi e altre informazioni:

```
int ricevi_opzione(int id_connessione, char *buffer_r)
{
	int count=0;
	int byte_letti=0;
	void *buffer = buffer_r;
	int dimensione = sizeof(char);

	while(count < dimensione) { //finche' tutti i byte non vengono elaborati
		if((byte_letti=read(id_connessione, buffer, dimensione - count)) > 0) {
			count += byte_letti; //conto i byte processati
			buffer += byte_letti; //aggiorno la posizione del buffer
		}
		else
			return 1;
	}

	return 0;
}
```

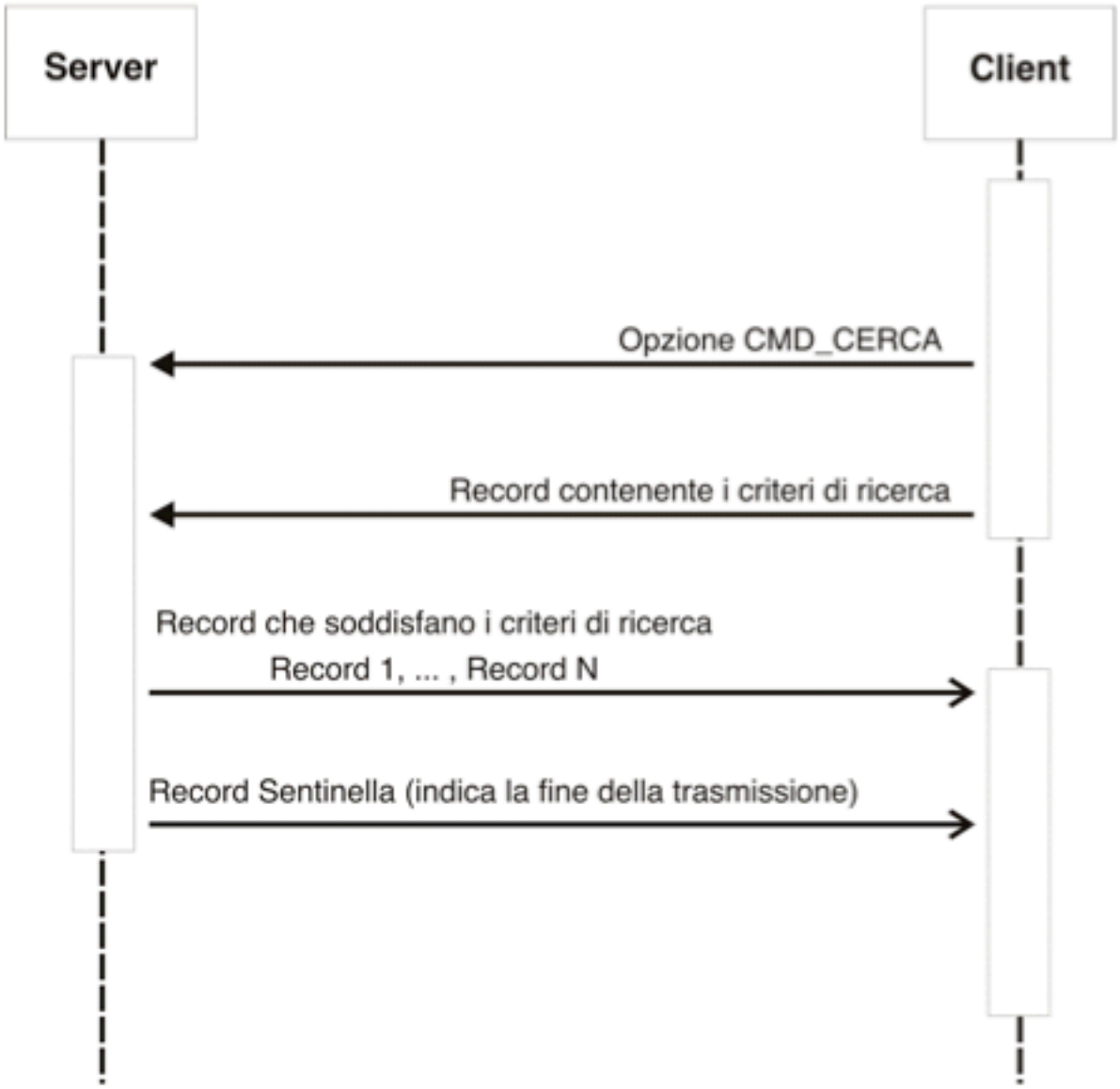
```
int trasmetti_opzione(int id_connessione, char buffer_r)
{
	int count=0;
	int byte_scritti=0;
	void *buffer = &buffer_r;
	int dimensione = sizeof(char);

	while(count < dimensione) {  //finche' tutti i byte non vengono elaborati
		if((byte_scritti=write(id_connessione, buffer, dimensione - count)) > 0) {
			count += byte_scritti; //conto i byte processati
			buffer += byte_scritti; //aggiorno la posizione del buffer
		}
		else
			return 1;
	}

	return 0;
}
```

Il protocollo di comunicazione si sintetizza negli schemi riportati nelle prossime pagine.

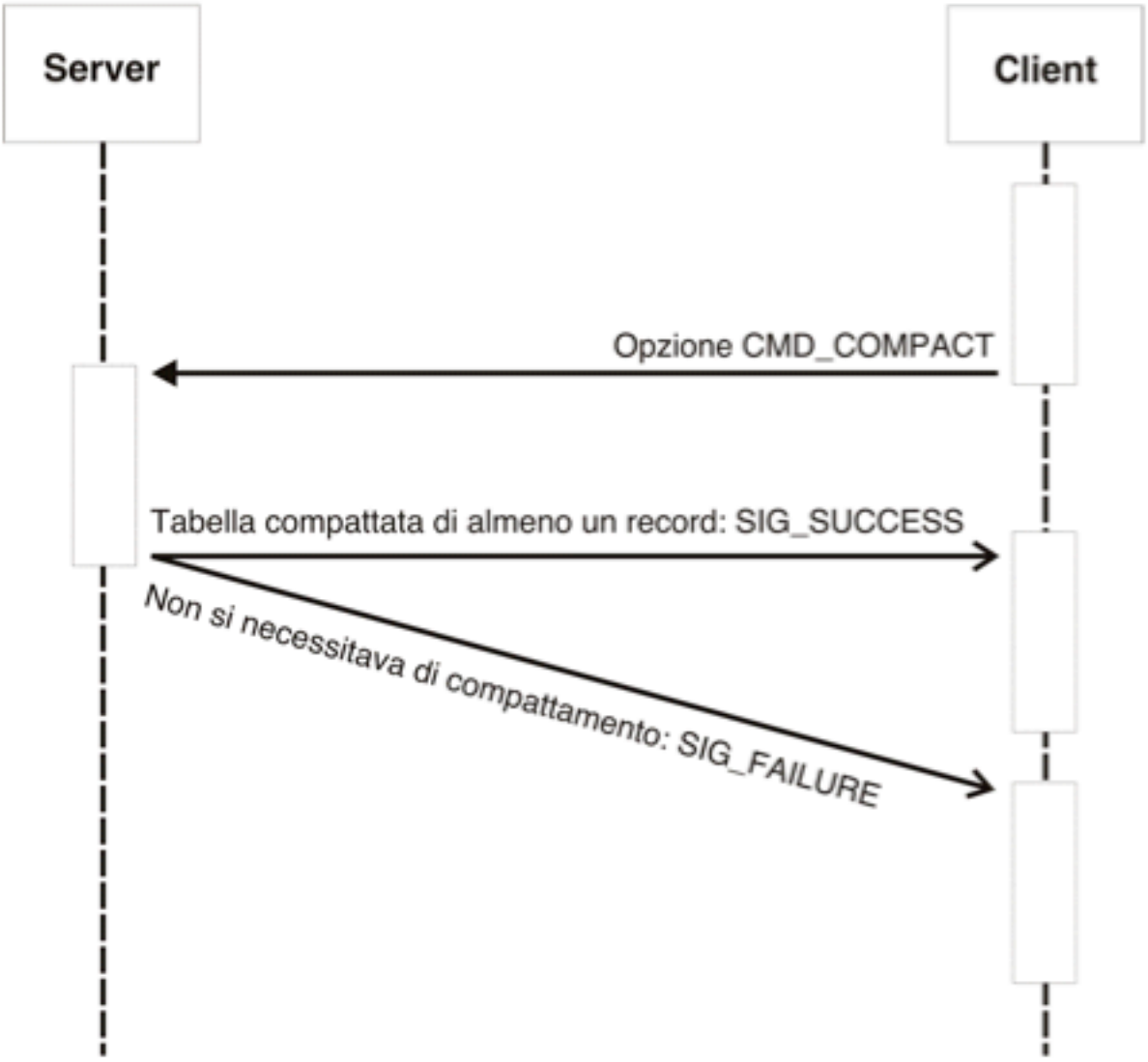
Operazione "Cancella Contatto"



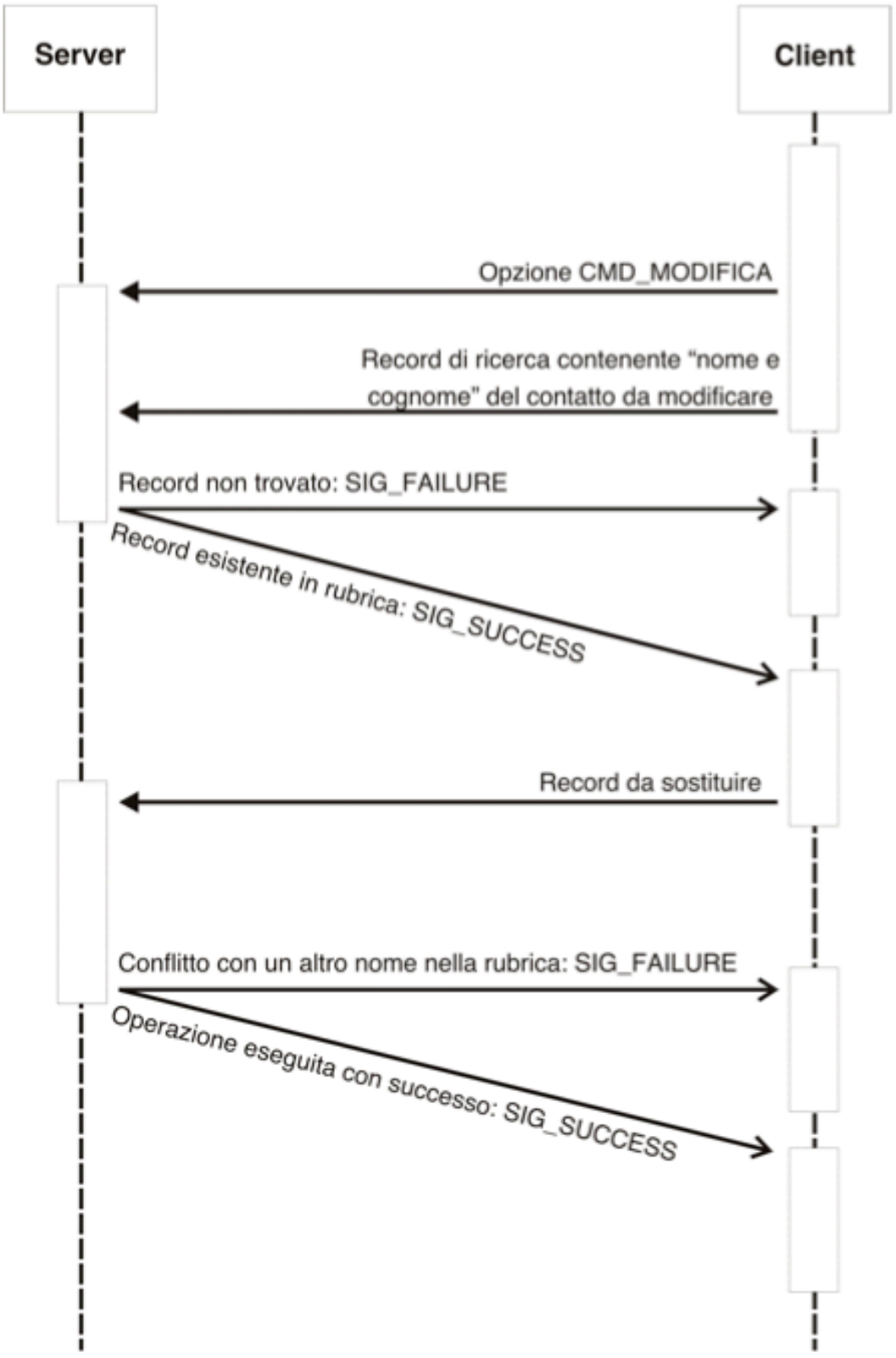
Operazione "Cerca Contatto"



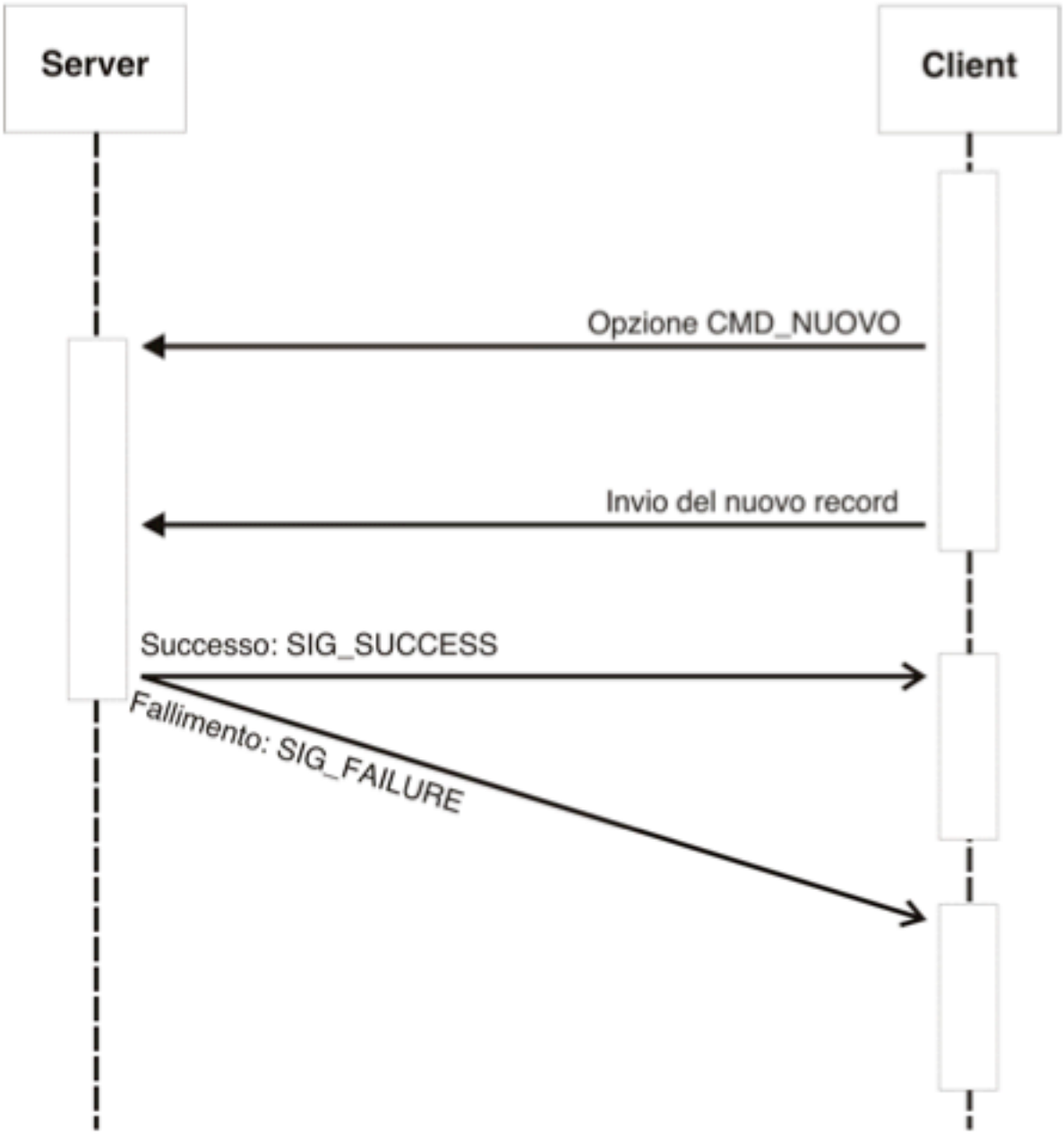
Operazione "Compact Tabella"



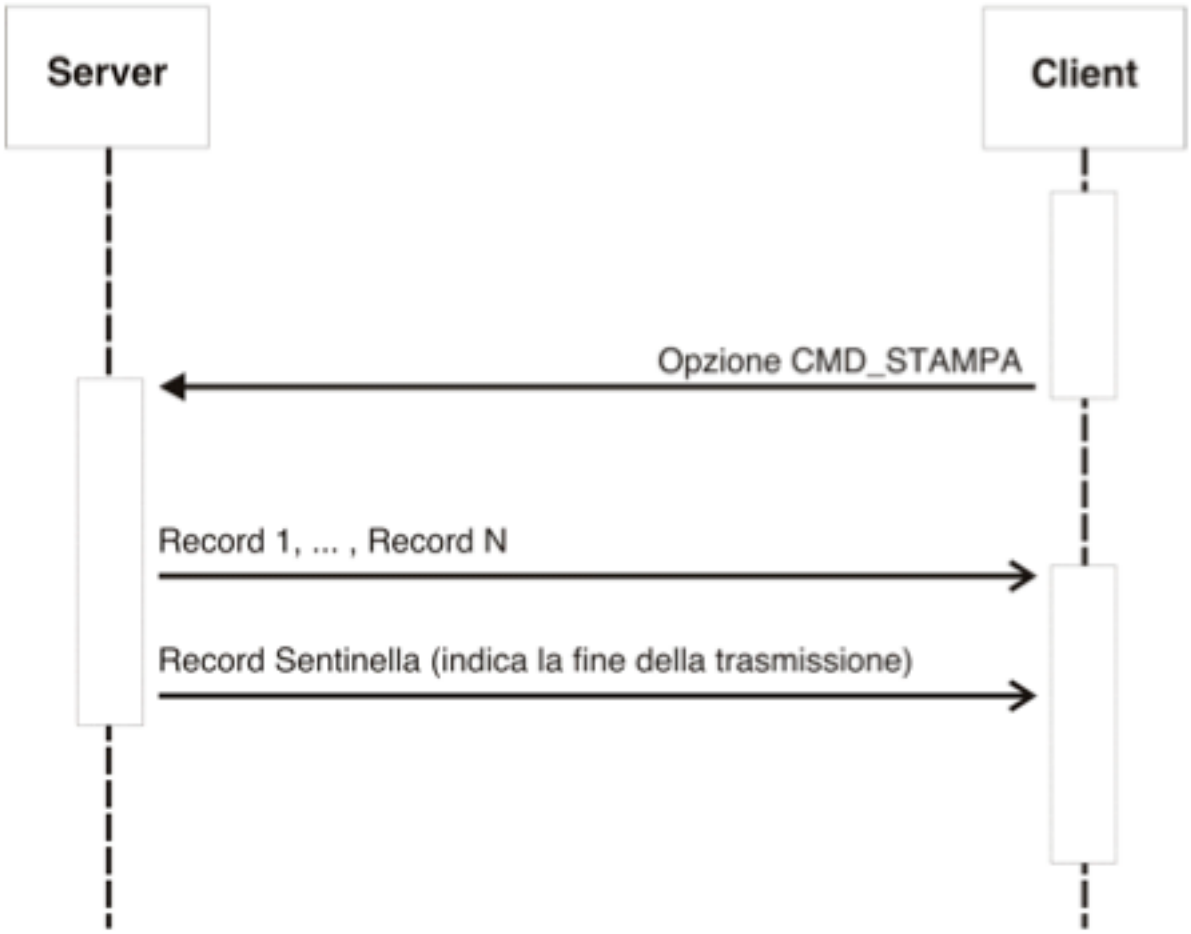
Operazione "Modifica Contatto"



Operazione "Nuovo Contatto"



Operazione "Stampa Contatti"



SERVER

ANALISI DEL PROBLEMA

Lo scopo del server è gestire un piccolo database fatto di una sola tabella. Come tutti gli altri server, anche questo è capace di accettare richieste dai client e soddisfarle nel modo più appropriato possibile.
Il server è stato sviluppato pensando a due principali problemi:

- Gestione del database su disco
- Sincronizzazione delle richieste da parte dei client.

STRUTTURE DATI

La tabella gestita ha il seguente formato:

Nome	Cognome	Indirizzo	Telefono	Email	Marcatura
Francesco	Ferrara	Via Cardinale, 13	3405114184	fsterrar@studenti.unina.it	N
Pasquale	Napolitano	Via Garibaldi, 45	3294565834	pasqui@gmail.com	N
Nicola	D'Amore	P.zza Vittoria	3382379304	nicdam@katamail.it	X

All'interno di essa l'elemento chiave è rappresentato da nome e cognome; tutti i record sono registrati in un file di testo in modo da rendere il database consultabile anche da altri programmi come "grep", "cat", ecc…
Nel file di testo per ogni riga è stato usato il seguente formato:

"<nome>/tab<cognome>/tab<indirizzo>/tab<telefono>/tab<email>/tab<marcatura>/new_line"

Il campo marcatura è un campo speciale fatto da un solo carattere e può essere ti due tipi:

```
#define CAMPO_CANCELLATO 'X'
#define CAMPO_NUOVO 'N'
```

Se il campo della marcatura è uguale a CAMPO_CANCELLATO, allora quel record della tabella è marcato per la cancellatura che avviene fisicamente al momento della compattazione della tabella.
Un valore speciale del campo marcatura che non viene mai memorizzato sul file è CAMPO_SENTINELLA; esso indica la fine di una sequenza di record trasmessi al client.

Il server, dopo aver accettato la richiesta di connessione da parte del client, crea un thread dedicato per gestire la connessione, inserisci l'id della connessione in una lista insieme con altre informazioni in una struttura di questo tipo:

```
struct elemento {
	int connessione; //numero connessione
	int debug_info; //DEBUG
};

typedef struct nodo {
	struct elemento elem; //elemento contenuto
	struct nodo *next; //puntatore al prossimo nodo
} *lista_t;
```

Come argomento, al thread, viene passato il puntatore alla struttura "elemento"; questa è la parte del codice che svolge questa operazione:

```
while(1) {
	temp = accetta_connessione(id_socket);

	pthread_mutex_lock(&m_lista);
		lista = adtl_inserisci(lista, temp, debug_info);
		arg = adtl_elem_addr(lista, temp); //estraggo l'indirizzo da passare al thread
		pthread_create(&tid, NULL, (void *(*)()) servi_client, arg);
	pthread_mutex_unlock(&m_lista);

	pthread_detach(tid);
}
```

Altre strutture dati usate sono:

```
struct contatto {
	char nome[MAX_NOME];
	char cognome[MAX_COGNOME];
	char indirizzo[MAX_INDIRIZZO];
	char telefono[MAX_TELEFONO];
	char email[MAX_EMAIL];
	char marcatura;
};

struct ricerca_contatto {
	char nome[MAX_NOME];
	char cognome[MAX_COGNOME];
};
```

Queste strutture contengono i campi dei record della tabella, e sono utilizzate:

- Per comunicare con il client,
- Come buffer temporaneo per facilitare la scrittura e la lettura dal file.

ORGANIZZAZIONE DEL PROGRAMMA

Il server è in grado di accettare connessioni da parte di più client; esso è organizzato in modo da evitare che operazioni contemporanee da parte di più client possano creare incoerenze. Per fare ciò è stato utilizzato l'algoritmo "dei lettori e degli scrittori" strutturato in questo modo:

```
pthread_mutex_t scrittura; //mutex per i thread che intendono scrivere
pthread_mutex_t lettura; //mutex per i thread che intendono leggere
int num_lettori = 0; //numero di thread lettori che stanno leggendo il file contemporaneamente

        .
        //inizializzo i mutex
        pthread_mutex_init(&scrittura, NULL);
        pthread_mutex_init(&lettura, NULL);
        .
```

```
//SCRITTORE

pthread_mutex_lock(&scrittura);
.
.//Sezione critica
.
pthread_mutex_unlock(&scrittura);
```

```
//LETTORE

pthread_mutex_lock(&lettura);
        num_lettori++;
        if (num_lettori == 1)
                pthread_mutex_lock(&scrittura);
pthread_mutex_unlock(&lettura);
.
.//Sezione critica
.
pthread_mutex_lock(&lettura);
        num_lettori--;
        if (num_lettori == 0)
                pthread_mutex_unlock(&scrittura);
pthread_mutex_unlock(&lettura);
```

Cosi facendo, c'è la possibilità di dare accesso in scrittura ad un solo client, ma l'accesso in lettura può essere concesso a più client contemporaneamente.

Una volta avviato il server esso entra in un ciclo infinito nel quale soddisfa le richieste dei client; per terminare questo ciclo si deve inviare al programma il segnale SIGINT (un modo per farlo è digitare Control+C). Questo è il codice relativo:

```
main_thread = pthread_self();
.
.
.
if (signal(SIGINT,handler_term) == SIG_ERR) //definisco l funzione da eseguire per SIGINT
                perror("signal"), exit(EXIT_FAILURE);
```

```
static void handler_term(int signo) {
        int elemento;

        if(pthread_equal(main_thread,pthread_self()) != 0) { //se sono il thread principale
                fprintf(stdout,"Chiusura del server in corso...\n");

                devo_terminare=1; //dico agli altri thread di terminare
                arresta_server(id_socket);
                pthread_exit(EXIT_SUCCESS); //termina il programma
        }

        /*Gli altri thread torneranno normalmente alle loro operazioni fino a terminare
        in un punto sicuro*/
}
```

Dopo aver inviato il segnale al processo, esso fa terminare solo il thread principale (cioè il thread che resta in ascolto di nuove connessioni da parte dei client), ma prima di farlo imposta ad 1 la variabile "devo_terminare". Questa variabile viene testata da tutti thread alla conclusione di ogni operazione e se è settata ad uno provoca la terminazione del thread in modo pulito; questo è il codice eseguito dai thread quando gestiscono le connessioni

```
do {
	if(ricevi(id_c, &opzione, DIM_OPZIONE))
		fatal_error(id_c, pthread_self(), debug);

	if (debug)
		fprintf(stdout,"[server] opzione %c da %d\n",opzione , id_c);

	switch (opzione) { //controllo cosa devo fare
		case CMD_STAMPA:
			stampa_contatti(id_c, debug); //stampa a video dei contatti
			break;
		case CMD_CERCA:
			cerca_contatto(id_c, debug); //cerco un contatto
			break;
		case CMD_MODIFICA:
			modifica_contatto(id_c, debug); //modificare un contatto
			break;
		case CMD_NUOVO:
			nuovo_contatto(id_c, debug); //inserire un nuovo contatto
			break;
		case CMD_CANCELLA:
			cancella_contatto(id_c, debug); //cancellare un contatto nella rubrica
			break;
		case CMD_COMPACT:
			compact_tabella(id_c, debug); //compattare la tabella
			break;
		case CMD_EXIT:
			if(trasmetti_opzione(id_c, SIG_TERM)) //uscita dal programma
				fatal_error(id_c, pthread_self(), debug);
			break;
	}

	if (devo_terminare) {
		if(trasmetti_opzione(id_c, SIG_TERM))
			fatal_error(id_c, pthread_self(), debug);
		flag=0;
	}
	else
		if(trasmetti_opzione(id_c, SIG_NULL))
			fatal_error(id_c, pthread_self(), debug);

} while((opzione != CMD_EXIT) && (flag));
```

Tutto questo è stato fatto per chiudere le connessioni in una maniera pulita. Inoltre ogni client è servito separatamente dagli altri e quindi se uno di essi si sconnette improvvisamente, gli altri non si accorgono dell'accaduto.

Alla fine di tutte le operazioni ogni thread esegue in modo individuale le seguenti operazioni:

```
pthread_mutex_lock(&m_lista); //accedo alla lista in maniere esclusiva
	chiudi_connessione(id_c);
	lista = adtl_cancella(lista, id_c); //cancello l'elemento dalla lista
pthread_mutex_unlock(&m_lista); //rilascio la lista

pthread_exit(NULL);
```

PRINCIPALI PROCEDURE

Queste sono le funzioni più importanti:

```
int avvia_server(int max_connessioni, int portnum);
```

Avvia il server aprendo la porta PORTNUM, ma non ancora si mette in ascolto dei client.

```
int accetta_connessione(int id_socket);
```

Ascolta se c'e' un client che vuole connettersi, stabilisce la connessione, e ritorna id.

```
int chiudi_connessione(int id_connessione);
```

Chiude una connessione con il server e restituisce 0. In caso di errore restituisce 1

```
void arresta_server(int id_socket);
```

Chiude il socket principale del server.

```
void reset_database(void);
```

Crea e/o azzera il file utile ad server per gestire la tabella. Questa procedura è chiamata quando si invoca il server con l'opzione "-r" o "--reset".

```
int apri_file_r(void);
```

Apre il file PATHNAME e ritorna il file descriptor.

```
int leggi_record(int fd, struct contatto *record, int *posizione);
```

Legga un record di tipo "struct contatto" dal file e lo memorizza in "record". Ritorna 0 in caso di successo e 1 quando il file e' finito.
In caso di errore di lettura è causata la terminazione del programma.

```
void chiudi_file(int fd);
```

Chiude il file individuato dal file descriptor
Le funzioni: int apri_file_r(void), void chiudi_file(int fd), int leggi_record(int fd, struct contatto *buffer), servono al programma chiamante per leggere il file fermandosi ogni record. Tutte le altre funzioni che operano sul file hanno incorporato al loro interno l'apertura e la chiusura del file.

```
int nuovo_record(struct contatto *new);
```

Inserisce all'interno del file il record "new". Ritorna 0 in caso di successo, 1 in caso di conflitto (cioe' esiste un altro contatto con lo stesso nome).

```
int trova_record(struct ricerca_contatto *cerco);
```

Dato "nome" e "cognome" di un contatto contenuti nella struttura ricerca_contatto questa procedura cerca il primo contatto nel file con gli stessi dati e ritorna la posizione nel file dove esso è memorizzato. Se non è stato trovato nessun contatto è ritornato -1.

```
int modifica_record(struct contatto *new, struct ricerca_contatto *cerca);
```

Dato un nuovo contatto "new", questa procedura lo memorizza sostituendolo al contatto memorizzato nella posizione individuata dal secondo parametro passato alla procedura. Ritorna 0 in caso di successo, e 1 in caso di fallimento. Si ha il fallimento quando c'e' un conflitto con altri nomi della rubrica oppure quando il record non è più esistente.

```
int cancella_record(struct ricerca_contatto *cancello);
```

Dati "nome" e "cognome" di un contatto contenuti nella struttura ricerca_contatto, viene cancellato dal file il record con lo stesso nome e cognome, e viene ritornato 0. Viene ritornato 1 in caso di record inesistente.

```
int compatta_tabella(void);
```

Vengono fisicamente cancellati i record marcati per la cancellazione dal file. Se il file non necessitava di compattazione viene ritornato 1, 0 altrimenti.

```
lista_t adtl_crealista();
```

Inizializza la lista.

```
lista_t adtl_inserisci(lista_t lista, int elemento, int debug_info);
```

Inserisce l'elemento nella lista.

```
lista_t adtl_cancella(lista_t lista, int elemento);
```

Cancella un elemento dalla lista.

MANUALE UTENTE

Per avviare questo programma non c'è bisogno di nessun parametro:

```
emac:~/Desktop/LabSO/progetto ferrara$ server
Server avviato... premere <Control+C> per uscire.
```

Per terminare il processo bisogna digitare Control+C:

```
emac:~/Desktop/LabSO/progetto ferrara$ server
Server avviato... premere <Control+C> per uscire.
^C
Chiusura del server in corso...
emac:~/Desktop/LabSO/progetto ferrara$
```

Tuttavia c'è la possibilità di avviare il processo in diverse modalità passandogli opportuni parametri. Il primo è "-h" o "--help" con il quale viene visualizzata sullo schermo una mini guida:

```
emac:~/Desktop/LabSO/progetto ferrara$ server -h
Usa: server [OPZIONE 1 ... OPZIONE N]

OPZIONE puo' essere:
        -h oppure --help        Per visualizzare questa guida
        -r oppure --reset       Per resettare il database dell'agenda
        -v oppure --verbose     Per visualizzare le informazioni di debug
        -p <numero porta>
        oppure
        --port <numero porta>   Per scegliere una porta diversa da quella di default

Progetto di Laboratorio di Sistemi Operativi (2004/2005)
Ferrara Francesco Saverio - 566/811 - fsterrar@studenti.unina.it

emac:~/Desktop/LabSO/progetto ferrara$
```

In quest'esempio è stato avviato il server in modalità "verbose" in modo da tenerci aggiornati sulle operazioni compiute facendogli stampare tutti i messaggi a video:

```
emac:~/Desktop/LabSO/progetto ferrara$ server -v
Server avviato... premere <Control+C> per uscire.
[CONNESSO] client numero 4
[server] opzione g da 4
[DISCONNESSO] client n?4 di indirizzo 3146048
[CONNESSO] client numero 4
[server] opzione a da 4
[thread 25167872] stampa contatti
[server] opzione d da 4
[thread 25167872] nuovo contatto
-------------Ricevuto record da 25167872:
Nome: Pasquale  Cognome: Napolitano
Indirizzo: via Cinthia, n 432
Telefono: 081/234623    E-Mail: pasnap@yahoo.com
Fine record-----------------------------
^C
Chiusura del server in corso...
```

In quest'altro esempio, invece, avviamo il server facendolo stare in ascolto su una porta diversa da quella di default:

```
emac:~/Desktop/LabSO/progetto ferrara$ server -p 2334
Server avviato... premere <Control+C> per uscire.
```

CLIENT

ANALISI DEL PROBLEMA

Il client è molto più semplice del server poiché deve solamente:

- Connettersi al server,
- Inviare le richieste,
- Stampare i risultati a video.

STRUTTURE DATI

Le uniche strutture dati usate sono i record destinati a contenere i dati dei contatti presenti nella rubrica:

```
struct contatto {
        char nome[MAX_NOME];
        char cognome[MAX_COGNOME];
        char indirizzo[MAX_INDIRIZZO];
        char telefono[MAX_TELEFONO];
        char email[MAX_EMAIL];
        char marcatura;
};

struct ricerca_contatto {
        char nome[MAX_NOME];
        char cognome[MAX_COGNOME];
};
```

Nella maggior parte delle operazioni il client scrive o legge dalla socket una sequenza di questi record.

ORGANIZZAZIONE DEL PROGRAMMA

Siccome l'attività prevalente di questo programma è scambiare informazioni sulla rete è stata fatta particolare attenzione a rilevare prima possibili problemi di rete. Ogni qualvolta ci sono problemi con la connessione il processo invoca la funzione fatal_error la quale provvede a stampare a video un opportuno messaggio d'errore:

```
void fatal_error(int id_connessione) {
        fprintf(stdout,"Errore di rete! Verificare il cavo e/o il programma server\n");
        chiudi_connessione(id_connessione); //chiudo la connessione
        exit(EXIT_FAILURE);
}
```

Per tutto il tempo questo processo si trova in un ciclo che dura fin quando l'utente non decide di uscire (digitando l'opzione 7). Qui si vede il codice sorgente:

```
fprintf(stdout,"Benvenuto, sei connesso con la rubrica\n");
do {
	fprintf(stdout,"\n\nPremere INVIO per continuare");
	getchar();
	stampa_menu();
	scanf("%d", &scelta);
	getchar();

	switch (scelta) {
		case 1:
			fprintf(stdout,"[]=-------=[STAMPA CONTATTI]=-------=[]\n\n");
			pthread_create(&tid, NULL, r_stampa_contatti, (void *) &id_connessione);
			stampa_contatti(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
				fprintf(stderr,"Impossibile aspettare il tread\n");
				exit(EXIT_FAILURE);
			}
			break;
		case 2:
			fprintf(stdout,"[]=-------=-=[CERCA CONTATTO]=-=-=-----=[]\n\n");
			pthread_create(&tid, NULL, r_cerca_contatto, (void *) &id_connessione);
			cerca_contatto(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
				fprintf(stderr,"Impossibile aspettare il tread\n");
				exit(EXIT_FAILURE);
			}
			break;
		case 3:
			fprintf(stdout,"[]=-------=[MODIFICA CONTATTO]=-=-=-=[]\n\n");
			pthread_create(&tid, NULL, r_modifica_contatto, (void *) &id_connessione);
			modifica_contatto(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
				fprintf(stderr,"Impossibile aspettare il tread\n");
				exit(EXIT_FAILURE);
			}
			break;
		case 4:
			fprintf(stdout,"[]=-------=-=[NUOVO CONTATTO]=-=-=-=---=[]\n\n");
			pthread_create(&tid, NULL, r_nuovo_contatto, (void *) &id_connessione);
			nuovo_contatto(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
				fprintf(stderr,"Impossibile aspettare il tread\n");
				exit(EXIT_FAILURE);
			}
			break;
		case 5:
			fprintf(stdout,"[]=-------=[CANCELLA CONTATTO]=-=-=-=[]\n\n");
			pthread_create(&tid, NULL, r_cancella_contatto, (void *) &id_connessione);
			cancella_contatto(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
				fprintf(stderr,"Impossibile aspettare il tread\n");
				exit(EXIT_FAILURE);
			}
			break;
		case 6:
			fprintf(stdout,"[]=-------=-=[COMPACT TABELLA]=-=-=-=---=[]\n\n");
			pthread_create(&tid, NULL, r_compact_tabella, (void *) &id_connessione);
			compact_tabella(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
				fprintf(stderr,"Impossibile aspettare il tread\n");
				exit(EXIT_FAILURE);
			}
			break;
		case 7:
			pthread_create(&tid, NULL, r_program_exit, (void *) &id_connessione);
			program_exit(id_connessione);
			if(pthread_join(tid, NULL) != 0) {
```

```
                                fprintf(stderr,"Impossibile aspettare il tread\n");
                                exit(EXIT_FAILURE);
                        }
                        break;
                default:
                        fprintf(stdout,"[ERRORE] Scelta errata!\n");
                        break;
        }

        if(ricevi_opzione(id_connessione, &risultato))
                fatal_error(id_connessione);

        if(risultato == SIG_TERM) {
                fprintf(stdout,"\n\n\nERRORE DI RETE\nA causa dell'indisponibilità del server");
                fprintf(stdout," il programma verrà terminato\n");
                scelta=7;
        }
} while (scelta != 7); //ciclo finche' non chiedo di uscire

chiudi_connessione(id_connessione); //chiudo la connessione
```

Notiamo che per ogni operazione viene creato un altro thread che partecipa all'esecuzione dell'operazione assieme a thread principale. In questo modo un thread resta sempre in ascolto sulla socket, mentre l'altro invia le richieste; così facendo il thread che si trova in ascolto si accorge immediatamente quando cade la connessione e può invocare la funzione fatal_error().

PRINCIPALI PROCEDURE

Queste sono le funzioni più importanti:

```
int connetti(char *ip, int portnum);
```

Dato un ip in input la funzione stabilisce una connessione con la macchina individuata dall'ip alla porta PORTNUM. Alla fine viene ritornato l'id della connessione

```
void chiudi_connessione(int id_connessione);
```

Questa funzione chiude la connessione individuata dalla variabile id_connessione.

```
void inserisci_record(struct contatto *new);
```

Dato un puntatore alla struttura contatto, chiede all'utente di inserire tutti i dati necessari a riempire la struttura.

```
void inserisci_record_ricerca(struct ricerca_contatto *new);
```

Dato un puntatore alla struttura ricerca_contatto, chiede all'utente di inserire tutti i dati necessari a riempire la struttura.

MANUALE UTENTE

Il processo si può lanciare semplicemente senza opzioni in questo modo:

```
emac:~/Desktop/LabSO/progetto ferrara$ client
Benvenuto, sei connesso con la rubrica


Premere INVIO per continuare
```

Facendo così si assume di volersi connettere con il server avviato su "localhost" in ascolto sulla porta di default.
Una volta avviato, viene visualizzato il menù e da questo momento in poi è possibile inviare le richieste al server. Quando si vuole uscire basta scegliere l'opzione n°7:

```
[]=-=-=-=-=-=-=-=-=-=-=-=-=[MENU]=-=-=-=-=-=-=-=-=-=-=-=-=[]
Scegli una delle seguenti opzioni:

        1) Stampa tutti i contatti
        2) Cerca un contatto
        3) Modifica un contatto
        4) Nuovo contatto
        5) Cancella un contatto
        6) Compact tabella
        7) ESCI

Inserisci la tua scelta: 7

Arrivederci
emac:~/Desktop/LabSO/progetto ferrara$
```

Possiamo anche avviare il client passandoci delle opzioni, ad esempio "`-h`" per visualizzare la guida:

```
emac:~/Desktop/LabSO/progetto ferrara$ client -h
Usa: client <numero ip> <numero porta>

Se i parametri non vengono inseriti, verranno utilizzati quelli di default

Progetto di Laboratorio di Sistemi Operativi (2004/2005)
Ferrara Francesco Saverio - 566/811 - fsterrar@studenti.unina.it

emac:~/Desktop/LabSO/progetto ferrara$
```

Dalla guida si vede che è possibile far avviare il programma istruendolo per farlo connettere a un ip diverso da "127.0.0.1" e ad una porta diversa da quella di default (3456). Qui vediamo un esempio:

```
emac:~/Desktop/LabSO/progetto ferrara$ client localhost 2334
Benvenuto, sei connesso con la rubrica


Premere INVIO per continuare
```

ESEMPIO COMPILAZIONE

Questa prova di compilazione è stata fatta su un sistema "*Mac OS X*" aggiornato alla versione 10.3:

```
emac:~/Desktop/LabSO/progetto ferrara$ uname -a
Darwin emac.local 7.4.1 Darwin Kernel Version 7.4.1: Wed May 12 18:54:39 PDT 2004; root:xnu/xnu-
517.6.11.obj~5/RELEASE_PPC  Power Macintosh powerpc
```

Dopo aver decompresso il file, ci spostiamo nella directory creata…

```
emac:~/Desktop/LabSO/progetto ferrara$ pwd
/Users/ferrara/Desktop/LabSO/progetto
emac:~/Desktop/LabSO/progetto ferrara$ ls
Makefile         README           doc              rubrica          src_client       src_common
src_server
```

E lanciamo la compilazione con il comando "make"

```
emac:~/Desktop/LabSO/progetto ferrara$ make
cc  -c src_server/s_main.c
cc  -c src_server/s_socket.c
cc  -c src_server/s_lista.c
cc  -c src_server/s_dbagenda.c
cc  -c src_common/x_dbagenda.c
cc  -c src_common/x_socket_rw.c
cc  -o server s_main.o s_socket.o s_lista.o s_dbagenda.o x_dbagenda.o x_socket_rw.o -lpthread
cc  -c src_client/c_main.c
cc  -c src_client/c_socket.c
cc  -c src_client/c_dbagenda.c
cc  -o client c_main.o c_socket.o c_dbagenda.o x_dbagenda.o x_socket_rw.o -lpthread
emac:~/Desktop/LabSO/progetto ferrara$
```

A questo punto sono stati creati due file eseguibili nella directory corrente, e cioè "`server`" e "`client`"

```
emac:~/Desktop/LabSO/progetto ferrara$ ls
Makefile        c_main.o        doc             s_lista.o       server          src_server
README          c_socket.o      rubrica         s_main.o        src_client      x_dbagenda.o
c_dbagenda.o    client          s_dbagenda.o    s_socket.o      src_common      x_socket_rw.o
emac:~/Desktop/LabSO/progetto ferrara$
```

Se si vuole installare il programma nel sistema, e si hanno i permessi del superutente, si può usare il comando:

```
make install
```

Per cancellare tutti i file creati (gli eseguibili e i file oggetto), si può usare il comando:

```
make clean
```

La compilazione è stata effettuata con successo anche su sistemi Linux con kernel 2.4.x e 2.6.x. Sul sistema Compaq Tru64 UNIX P5.1-10 (OSF1 Rev. 397) sono stati riportati diversi "warning", ma sono del tutto normali:

```
studenti.unina.it> make
cc  -c src_server/s_main.c
cc  -c src_server/s_socket.c
cc  -c src_server/s_lista.c
cc  -c src_server/s_dbagenda.c
cc  -c src_common/x_dbagenda.c
cc  -c src_common/x_socket_rw.c
cc: Warning: src_common/x_socket_rw.c, line 22: In this statement, performing pointer arithmetic
on a pointer to void or a pointer to function is not allowed.  The compiler will treat the type
as if it were pointer to char. (badptrarith)
                        buffer += byte_letti; //aggiorno la posizione del buffer
------------------------^
cc: Warning: src_common/x_socket_rw.c, line 39: In this statement, performing pointer arithmetic
on a pointer to void or a pointer to function is not allowed.  The compiler will treat the type
as if it were pointer to char. (badptrarith)
                        buffer += byte_scritti; //aggiorno la posizione del buffer
------------------------^
cc: Warning: src_common/x_socket_rw.c, line 58: In this statement, performing pointer arithmetic
on a pointer to void or a pointer to function is not allowed.  The compiler will treat the type
as if it were pointer to char. (badptrarith)
                        buffer += byte_letti; //aggiorno la posizione del buffer
------------------------^
cc: Warning: src_common/x_socket_rw.c, line 77: In this statement, performing pointer arithmetic
on a pointer to void or a pointer to function is not allowed.  The compiler will treat the type
as if it were pointer to char. (badptrarith)
                        buffer += byte_scritti; //aggiorno la posizione del buffer
------------------------^
cc  -o server s_main.o s_socket.o s_lista.o s_dbagenda.o x_dbagenda.o x_socket_rw.o -lpthread
cc  -c src_client/c_main.c
cc  -c src_client/c_socket.c
cc  -c src_client/c_dbagenda.c
cc  -o client c_main.o c_socket.o c_dbagenda.o x_dbagenda.o x_socket_rw.o -lpthread
studenti.unina.it>
```

Questi avvisi da parte del compilatore ci dicono che nel codice sorgente incrementiamo puntatori a "void" che non hanno un'aritmetica definita: è vero, ma in quel contesto si stava ragionando in byte e quello che si vuole è incrementare il puntatore di un byte.

RIFERIMENTI

Giovanni Verzino
"Slide delle lezioni di Sistemi Operativi"
http://staff.marscenter.it/verzino/lso/

Simone Picardi
"GaPiL"
(Guida alla Programmazione in Linux)

Richard Stevens
"Advanced Programming in the UNIX Environment"
"Advanced UNIX Programming"
Addison Wesley

Andrew S. Tanenbaum
"Modern Operating Systems"
Prentice Hall