L’esercizio di programmazione richiede di progettare un sistema a tre livelli:

1) Processi client
2) Gestore delle richieste (sulla macchina client)
3) Server (remoti)

Il punto (1) chiede di descrivere il protocollo di comunicazione adottato dall’applicazione.
Assumendo che la connessione fra il gestore locale e il server remoto avvenga attraverso TCP/IP, e’ possibile costruire un semplice protocollo di comunicazione, che contenga:
- un identificativo del protocollo
- un identificativo della richiesta/risposta
- un insieme di codici che identificano il tipo dei parametri (e.g., 0=stringa, 1=intero, 2=float) e dei valori di ritorno

Esempio:

Richiesta di funzione
"My_RPC_Protocol\n"
"Function Request\n"
"functionname;0:foo;2:0.3\n"

Richiesta dell’elenco dei servizi offerti
"My_RPC_Protocol\n"
"Info Request\n"
"\n"

Risposta negativa
"My_RPC_Protocol\n"
"Error Response\n"
(segue un codice d’errore, non e’ necessario entrare in ulteriori dettagli)

Risposta positiva alla richiesta di funzione
"My_RPC_Protocol\n"
"Function Response\n"
valore di ritorno (e.g.: "0:bar\n", oppure "1:27\n")

Risposta positiva alla richiesta di informazioni
"My_RPC_Protocol\n"
"Info Response\n"
"fname1;fname2;fname3\n"

Il punto (2) richiede di implementare il server.
Questo e’ del tutto analogo al server HTTP visto a lezione (riportato qui con le modifiche necessarie, e senza le parti che controllano la correttezza dell’esecuzione ed eseguono il log, che per semplicita’ possiamo ignorare):

```
void server_run (struct in_addr local_address, uint16_t port)
{
  struct sockaddr_in socket_address;
  int rval;
  struct sigaction sigchld_action;
  int server_socket;

        /* Create a TCP socket. */
        server_socket = socket (PF_INET, SOCK_STREAM, 0);
        memset (&socket_address, 0, sizeof (socket_address));
        socket_address.sin_family = AF_INET;
        socket_address.sin_port = port;
        socket_address.sin_addr = local_address;
```



```
        rval = bind (server_socket, &socket_address, sizeof (socket_address));
        rval = listen (server_socket, 10);

        while (1) {
          struct sockaddr_in remote_address;
          socklen_t address_length;
          int connection;
          pid_t child_pid;
        address_length = sizeof (remote_address);
        connection = accept (server_socket, &remote_address, &address_length);

    child_pid = fork ();
    if (child_pid == 0) {
      close (STDIN_FILENO);
      close (STDOUT_FILENO);
      close (server_socket);
                        handle_connection(connection)
                close (connection);
      exit (0);
    }
    else if (child_pid > 0) {
      close (connection);
    }
  }
}


int handle_connection(int connection) {
          char buffer[256];
                        ssize_t bytes_read;
                        char **args;
                        int i=0,nargs=0;
                        char *func;
                        char *tmp;

                        /* Leggiamo la linea relativa all’header del protocollo
*/
                        bytes_read = read (connection, buffer, sizeof (buffer) -
 1);
                buffer[bytes_read] = ‘\0’;
                        if (strcmp(buffer,"My_RPC_Protocol")!=0)
                                /* Protocollo non valido, non e’ richiesta una g
estione particolare */
                                exit(1);
                        /* Leggiamo la linea relativa alla richiesta */
                        bytes_read = read (connection, buffer, sizeof (buffer) -
 1);
                buffer[bytes_read] = ‘\0’;
                        if (strcmp(buffer,"Function Request")==0) {
                                bytes_read = read (connection, buffer, sizeof (b
uffer) - 1);
                          buffer[bytes_read] = ‘\0’;
                                nargs=conta_separatori(buffer); /* ci sono tanti
 parametri quanti ’;’ in buffer; l’implementazione di questa parte non e’ necess
aria */
                                args=malloc(sizeof(char *)*nargs);
                                tmp=strtok(buffer,";");
                                func=strdup(tmp);
                                while (tmp=strtok(NULL,";")!=NULL) {
                                        tmp[i]=’\0’
                                        args[i]=convert(atoi(tmp[1]),strdup(tmp[
2])); /* La funzione di conversione non deve essere implementata */
```

```
                                        i++
                                }
                                tmp=handle_request(func,nargs,args); /* di quest
a funzione non e' richiesta l'implementazione! */
                                write(connection,"My_RPC_Protocol\nFunction Resp
onse\n",34);
                                write (connection, tmp, strlen (tmp)); /* rispos
ta */
                        }
                        else if (strcmp(buffer,"Info Request")==0) {
                                tmp=handle_info_request(); /* di questa funzione
 non e' richiesta l'implementazione! */
                                write(connection,"My_RPC_Protocol\nInfo Response
\n",29);
                                write (connection, tmp, strlen (tmp)); /* rispos
ta */
                        }
}


L'ultima parte dell'esercizio richiede di implementare il gestore delle richiest
e sul lato client.
La parte server del gestore e' sostanzialmente analoga a quella del server visto
 al punto 2 (ovviamente essendo una comunicazione esclusivamente locale, e' poss
ibile usare PF_LOCAL/PF_UNIX).
La qualita' bloccante o meno del client e' in prima approssimazione irrilevante
(dal momento che comunque un processo separato gestisce ogni richiesta).
Cambia, ovviamente, la parte di gestione (handle connection), che deve aprire un
a connessione con uno dei server.
In questo caso, c'e' un solo tipo di operazione.

Possiamo supporre di disporre delle seguenti strutture e funzioni (da implementa
re solo se avanza tempo):
int servers[NSERVERS]; /* serve anche una struttura che mantiene la lista di fun
zioni per ogni server */
int actives[NSERVERS];
int check(int server, char *func); /* puo' server eseguire func? */
int update(int server); /* update status of currently inactive server */
union semun {
   int val;
   struct semid_ds *buf;
   unsigned short int *array;
   struct seminfo *__buf;
};

Sara' necessario definire un semaforo che gestisca l'accesso in mutua esclusione
 alle strutture condivise dai processi del gestore, nonche' l'apertura della con
nessione con i server remoti.
```