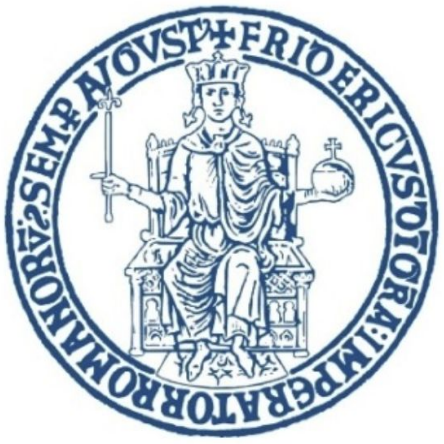


UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Università degli Studi di Napoli "Federico II"
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Corso di Laurea in Informatica

Relazione

"Remote System Monitor"

Insegnamento Laboratorio di Sistemi Operativi.
Anno Accademico 2019/2020

DOCENTE

Prof. Marco Faella

AUTORI

Ciro Gallo	N86002489
Monteiro Del Prete	N86002527

Nº Gruppo: 21

Sommario

Per la visione dei sorgenti si rimanda al repository GitHub

https://github.com/fed2-lso-remote_sysmonitor.git

GUIDA ALLA COMPILAZIONE E ALL'USO (CLIENT E SERVER):

Utilizzo

La messa in opera del sistema di monitoraggio remoto avviene lanciando a riga di comando i componenti necessari per il funzionamento: server, client (⩾ 1), agent (⩾ 1).

L'ordinamento dell'esecuzione è arbitrario:

- se viene lanciato prima il client, quest'ultimo resterà in attesa del server e successivamente della registrazione di almeno un agent (✳).
- se viene lanciato prima l'agent, quest'ultimo resterà in attesa del server e successivamente inizierà ad inviargli informazioni sull'host.
- se viene lanciato prima il server, l'agent ed il client potranno essere lanciati in qualsiasi ordine.

Per la corretta esecuzione del sistema sono necessarie delle specifiche a livello di riga di comando relative alla connessione tramite protocollo TCP/IP:

```
usage: ./run_agent <port> <ipaddress>
```

```
usage: ./run_client <port> <ipaddress>
```

```
usage: ./run_server <port_agent> <port_client>
```

Supponendo una esecuzione "ordinata" (✳), una volta avviati i componenti server e agent , questi lavoreranno in modo automatizzato e il client, successivamente lanciato, sarà l'unica interfaccia con l'utente. A seconda del numero di agents registrati, il server invierà al client una lista che verrà mostrata su STDOUT con la possibilità di scegliere il relativo host di cui si vogliono ottenere le informazioni:

```
Connected

List of registered hosts. Pick one:

   0.  90-37-218-55.ip127.fastwebnet.it connected

Choose an host to take additional info: |
```

La lista degli hosts viene costruita in modo tale da facilitarne la selezione, per cui verrà letto da STDIN l'identificativo, ad esempio `0` .
Successivamente vengono mostrare le informazioni relative all'host scelto:

host connesso

```
-------------------------------------
Requested infos about host 90-37-218-55.ip127.fastwebnet.it

uptime: ..
freeram: ..
procs: ..
-------------------------------------
```

host disconnesso

```
-------------------------------------
Requested infos about host 90-37-218-55.ip127.fastwebnet.it

last date: ..
-------------------------------------
```

(✳) i messaggi su STDOUT renderanno chiaro il contesto

Compilazione

Per facilitare la fase di compilazione si mette a disposizione dell'utente un makefile da utilizzare con il tool "make". I target disponibili sono i seguenti:

- `make server` per la compilazione del server. Il risultato è un file eseguibile run_server, lanciabile con `./run_server <port_agent> <port_client>`.

- `make client` per la compilazione del client. Il risultato è un file eseguibile run_client, lanciabile con `./run_client <port> <ipaddress>`.

- `make agent` per la compilazione dell'agent. Il risultato è un file eseguibile run_agent, lanciabile con `./run_agent <port> <ipaddress>`.

Inoltre, come targets di utility:

- `make all` per la compilazione di tutti e tre i componenti, client, agent e server.

- `make clean` per la cancellazione di tutti gli eseguibili prodotti.

PROTOCOLLO DI COMUNICAZIONE CLIENT-SERVER:

Quando il client effettua la prima connessione con il server, ottiene e stampa su STDOUT un messaggio di stato che può essere:

```
There are no agents. Waiting for at least one registered host...
```

se nessun host è ancora registrato sul server.

```
List of registered hosts. Pick one:
```

altrimenti.

Il server prepara la lista degli host: si tratta di un'unica stringa in cui le informazioni relative a ciascun host - hostname/IP e stato - sono separate da '\n'; tale stringa viene processata dal client in modo da costruire un array di char * che permetta l'indicizzazione degli hosts. In particolare, nell'*i-esima* locazione dell'array ci sarà la entry della lista avente identificativo *i*.
Ottenuta la lista e costruito l'array, il client chiede all'utente di inserire su STDIN l'identificativo di un host, che viene utilizzato per accedere all'array e recuperare le informazioni ad esso associate. In particolare, da queste informazioni viene estratto l'hostname e inviato al server. Il server risolve l'hostname per ottenere l'IP corrispondente (usato come chiave) e ricercare nel binary search tree AVL il nodo contenente le informazioni associate.
Una volta ottenute le informazioni, dapprima il server comunica al client lo stato - "connected" o "disconnected" - dell'host scelto. Dopodichè, se l'host risulta in stato "connected", il server prepara e invia al client un *unsigned long* buffer di dimensione 3 con i dati recuperati:

- *Uptime* : secondi trascorsi dall'ultimo boot.
- *freeRamPercentage* : percentuale di memoria fisica disponibile.
- *Procs* : numero di processi in esecuzione.

Se invece l'host risulta in stato "disconnected" il server invia al client una stringa contenente la data e l'ora dell'ultimo aggiornamento ricevuto su di esso.

PROTOCOLLO DI COMUNICAZIONE AGENT-SERVER:

Quando l'agent effettua la connessione con il server, immediatamente entra in un ciclo in cui, ad intervalli di 3 secondi, recupera le informazioni necessarie sullo stato dell'host su cui è in esecuzione e le invia al server.
D'altra parte, il server prima identifica l'agent tramite il suo indirizzo IP (che utilizzerà come chiave per fare le ricerche nel binary search tree AVL), poi si mette in attesa di ricevere le informazioni riguardo l'host su cui l'agent è in esecuzione. Tale attesa ha un timeout fissato a 6 secondi. Se il server riceve informazioni entro il tempo fissato, allora prepara un nodo di tipo BSTNode contenente i dati ricevuti e poi effettua una serie di controlli per aggiornare il binary search tree AVL contenente tutte le informazioni su tutti gli host registrati:

- Se l'host non si è mai registrato presso il server, allora viene inserito un nuovo nodo nella struttura.
- Se l'host risulta già registrato, viene aggiornato il nodo ad esso associato.

Se invece il tempo fissato scade prima di ricevere nuove informazioni o l'agent si disconnette, allora il server imposta lo stato del relativo host a "disconnected".

Ovviamente, trattandosi di un contesto multithreaded, tutte le modifiche al binary search tree, che è condiviso tra i threads, sono sincronizzate tramite un mutex (per struttura), evitando così situazioni di race condition.

DETTAGLI IMPLEMENTATIVI

DETTAGLI IMPLEMENTATIVI COMUNI

Tutti i componenti del sistema alternano, a seconda del contesto e delle necessità implementative, le funzioni di `readn` e `writen` al posto delle system calls standard `read` e `write`. L'utilizzo di queste due funzioni nasce dall'esigenza di dover "superare", non essendo un vero errore, il comportamento di `read` e `write` su stream socket, le quali possono scrivere/leggere meno bytes di quanti ne sono richiesti. La `writen` è ripresa dal seguente riferimento (pag. 122 figure 3.16), la `readn` dallo stesso riferimento (figure 3.15) con riadattamento.

CLIENT

Successivamente alle system call di connessione associate ai relativi controlli di correttezza, il client entra in un ciclo iterativo la cui prima istruzione è una chiamata alla funzione `printAndGetUpdatedList(...)`:

> Dopo la lettura, tramite `read`, della lista degli hosts sottoforma di unica stringa, esegue in tal ordine due chiamate a funzione:
>
> ```
> char ** hosts = hostsToArray(buff,hnumber);
>
> printHosts(hosts);
> ```
>
> La funzione `hostsToArray` ha come parametri d' ingresso `buff`, che rappresenta la stringa contenente i nomi hosts, e `hnumber` che viene riempito con il numero degli hosts registrati dal server. La funzione, sulla base del protocollo di comunicazione client-server, costruisce un array di stringhe in cui ad ogni locazione corrispondono l'hostname e l'hoststate:
>
> ```
> char ** arrayHost = (char**)malloc(sizeof(char*));
> int i = 0, j = 0, k = 0;
> int bytes = 0;
> ```

```
while( buff[i] != '\0') {
      bytes++;
      if( buff[i] != '\n' ) {
          arrayHost=
          (char**)realloc(arrayHost,sizeof(arrayHost)
                    + sizeof(char*));
          arrayHost[j]=
          (char*)malloc(sizeof(char)*bytes);
          bytes = 0;
          j++;
      }
      i++;
}
*hnumber = j;

..
```

A questo punto arrayHost è allocato della dimensione necessaria e si prosegue al riempimento, carattere per carattere:

```
..

i = 0;
j = 0;
while(buff[i] != '\0'){
   if(buff[i] != '\n') {
      arrayHost[j][k] = buff[i];
      k++;
   } else {
      arrayHost[j][k] = '\0';
      k = 0;
      j++;
   }
   i++;
}
return arrayHost;
```

La funzione `printHosts` utilizza la variabile globale `hostsnumber`, precedentemente assegnata da `hostsToArray`, per scorrere l'array così ottenuto e mostrare su STDOUT la lista.

Successivamente, di ritorno dalla funzione `printAndGetUpdatedList`, la chiamata

```
g_hostname = getHostName(hosts[choice]);
```

viene utilizzata per ottenere il nome dell'host ed inviarlo al server; la variabile "choice" contiene l'indice dell'host scelto dall'utente.
Nel dettaglio, la funzione alloca inizialmente lo spazio necessario per copiare, carattere per carattere, il nome nella variabile hostname

```
if(hostinfo[pos] == 'c'){
   hostname = (char*)malloc(sizeof(char)*(dim - 9));
} else {
   hostname = (char*)malloc(sizeof(char)*(dim - 12));
}
```

per poi completare il riempimento, ricordando di non includere l'hoststate (separato dall'hostname con uno spazio)

```
while(hostinfo[i] != ' ') {
   hostname[i] = hostinfo[i];
   i++;
}
hostname[i] = '\0';

return hostname;
```

SERVER

Memorizzazione delle informazioni

Il server deve memorizzare un pacchetto di informazioni per ogni agent registrato e poiché il numero di agents che si connetteranno non è prefissato o comunque non prevedibile, si rende necessario l'utilizzo di una **struttura dati dinamica**. Inoltre, sarebbe preferibile che la struttura fosse **ottimizzata** nelle operazioni di **ricerca** e **inserimento,** cioè quelle che il server effettua più frequentemente.

Per onorare tali requisiti, si è scelto di utilizzare un **binary search tree AVL** con autobilanciamento mediante rotazioni, avente come chiave di ricerca e quindi di ordinamento gli IP degli hosts (convertiti da *char** a *long* tramite la funzione `long parseIP(char * IP)`). In particolare, un nodo del bst, definito come tipo `BSTNode`, ha la seguente struttura:

```
typedef struct BSTNode BSTNode;

struct BSTNode{
    long key;
    bool connected;

    char * idhost;
    char * time;

    unsigned long uptime;
    float freeRamPercentage;
    unsigned long procs;

    int height; //Utilizzato per il bilanciamento
    BSTNode * dx;
    BSTNode * sx;
};
```

Il riferimento alla radice dell'albero e al mutex ad esso associato è memorizzato invece in una variabile globale di tipo `BSTHostInfo *`. Il tipo `BSTHostInfo` è definito come segue:

```
typedef struct BSTHostInfo{
```

```
        BSTNode * root;
        pthread_mutex_t mutex;
    } BSTHostInfo;
```

Creazione e distruzione della struttura sono gestite mediante le seguenti funzioni:

```
void destroyBSTHostInfo(BSTHostInfo * bstInfo){
    pthread_mutex_destroy(&bstInfo->mutex);
    bstDestroy(bstInfo->root);

    free(bstInfo);
}

BSTHostInfo * initBSTHostInfo(void){
    BSTHostInfo * bstInfo = (BSTHostInfo
*)malloc(sizeof(BSTHostInfo));
    bstInfo->root = NULL;

    pthread_mutex_init(&bstInfo->mutex,NULL);

    return bstInfo;
}
```

Poiché la struct contenente il riferimento al BST è globale, quindi condivisa tra tutti i threads, è stato necessario associare ad essa un mutex, evitando così race conditions sugli accessi.

La libreria `<list.h>` definisce una struct `node` utilizzata per la creazione di una lista (con le relative funzioni di gestione) che possa mantenere i TID dei singoli thread creati dai thread stub, permettendo così al main thread di attendere la loro terminazione mediante `pthread_join`.

Preparazione della lista degli hosts

Dopo la connessione di un client, il server deve preparare la lista degli hosts da scrivere sulla socket: si tratta di un'unica stringa in cui le informazioni relative a ciascun host - hostname/IP e stato - sono separate da '\n'. Tale operazione viene effettuata tramite la funzione ricorsiva

```
char * bstGetHosts(BSTNode * root)
```

Tale funzione effettua una visita in post order del bst, concatenando ad ogni chiamata ricorsiva le informazioni locali a quelle recuperate dai sottoalberi destro e sinistro (se esistono) e separandole con il carattere '\n'. Tale carattere sarà utilizzato dal client per organizzare le stringhe in un array.

Terminazione del server

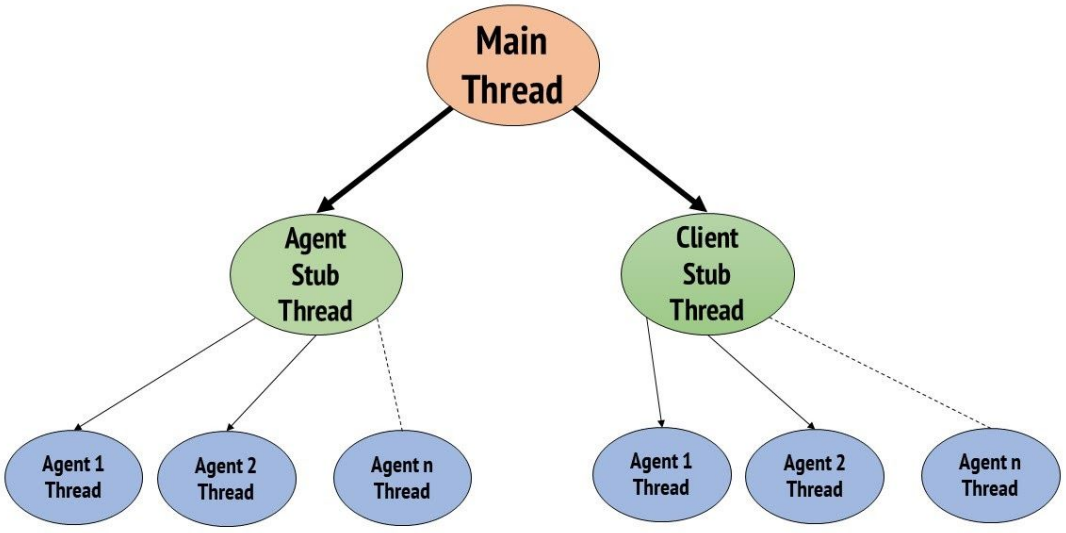
Il server può raggiungere lo stato di terminazione solo tramite segnale SIGINT, generato dal comando CTRL-C da tastiera, al quale corrisponde l'esecuzione dell'handler `handleSigInt()`. Questo provoca l'attivazione del flag globale `serverKilled` notificando i thread della terminazione del server e permettendo loro una spontanea e pulita terminazione. I thread stub e i singoli thread da essi creati reagiscono diversamente all'attivazione del flag:

- i thread stub accettano le connessioni tramite system call accept non bloccanti grazie alle quali, in un ciclo while avente come condizione `serverKilled`, possono verificare costantemente la terminazione o meno del server. In caso positivo, la memoria allocata verrà deallocata dai singoli thread ed il thread stub terminato verrà raccolto dal main thread tramite system call `pthread_join()`.

- i singoli thread servono/gestiscono i client/agent mediante un'iterazione avente come condizione il flag `serverKilled`. Terminato il server, i thread deallocano le proprie variabili locali e quelle passate dai thread stub. Anche in questo caso il main catturerà la loro terminazione con `pthread_join`.

Gestione dei threads

Il server è di tipo concorrente, ovvero è in grado di servire più client e gestire più agent simultanemente. La gestione delle connessioni è realizzata infatti mediante una gerarchia di threads, ciascuno con un compito ben preciso.



Il main thread ha il compito di creare i threads stub - agent stub thread e client stub thread - e le socket su cui essi accetteranno le connessioni in entrata.

```
socket(PF_INET,SOCK_STREAM | SOCK_NONBLOCK,0);
```

In particolare, le socket sono dichiarate non bloccanti (tipo `SOCK_NONBLOCK`) per evitare che i threads stub rimangano bloccati sulla system call `accept()`, non potendo così controllare il flag `serverKilled`, settato a true dall'handler del segnale `SIGINT` per ottenere una terminazione pulita dei thread. Inoltre, tramite la funzione `setsockopt()`, alla socket del client stub e a quella dell'agent stub è stato associato un timer, per finalità e motivi diversi. In particolare:

- **Socket degli agents** - `sdAgent`.

```
struct timeval timer_agent;

timer_agent.tv_sec=6;
timer_agent.tv_usec=0;

//Set timeout of 6 seconds on read calls.
setsockopt(sdAgent,SOL_SOCKET,SO_RCVTIMEO,(const char
*)&timer_agent,sizeof(timer_agent))
```

Le operazioni di lettura sulla socket hanno un timeout di 6 secondi, terminati i quali - nella funzione `handleAgent()` - lo stato dell'host associato all'agent viene posto a "disconnected".

Nella funzione handleAgent():

```
//If agent closes socket or read remains blocked for more
than 6 seconds set hoststate to "disconnected".

if(read(socketAgent,read_buffer,sizeof(read_buffer)) <= 0){
    pthread_mutex_lock(&bstHostInfo->mutex);
        bstSetState(bstHostInfo->root,localKey,false);
    pthread_mutex_unlock(&bstHostInfo->mutex);

    break;
}
```

- **Socket dei clients** - `sdClient`

```
struct timeval timer_client;

timer_client.tv_sec=1;
timer_client.tv_usec=0;

//Set read calls non-blocking. If fails, kill server.
setsockopt(sdClient,SOL_SOCKET,SO_RCVTIMEO,(const char
   *)&timer_client,sizeof(timer_client))
```

Le operazioni di lettura sulla socket hanno un timeout di 1 secondo, terminati i quali il server controlla semplicemente il flag serverKilled:

- Se il flag ha valore *true*, allora l'operazione di lettura viene interrotta e il thread terminato.
- Se il flag ha valore *false*, allora l'operazione di lettura viene ripresa.

Nella funzione handleClient():

```
while(!serverKilled){
    //Read the choice from client
    while((nread=read(socketClient,read_buff,BUFFSIZE)) < 0){
        if(serverKilled)
            break;
    }
    if(serverKilled || nread==0)
        break;
}
//Clean termination with release of resources used by current
thread
```

I threads stub quindi hanno il compito di accettare le connessioni in entrata tramite la system call `accept()`, e creare i thread che si occuperanno di gestirle tramite le funzioni `handleClient()` e `handleAgent()`.

AGENT

Obiettivo principale dell'agent è ottenere le informazioni di sistema dell'host su cui è in esecuzione. Per farlo utilizza la libreria `<sys/sysinfo.h>`

```
struct sysinfo info;
..
if(sysinfo(&info) != 0) {
   error("Error fetching system
   informations\n",STDOUT_FILENO,ESYS_INFO);
}
```

Oltre a riempire i campi della struct `info` dei valori `uptime` e `procs` , viene calcolata la percentuale di memoria fisica libera mediante la funzione `ramToPercentage`.

GESTIONE DEGLI ERRORI

La gestione degli errori avviene mediante la funzione

```
void error(char* msg, const int std, int err);
```

che definisce il messaggio, lo standard stream su cui trascriverlo e il tipo di errore da restituire alla terminazione del processo (mediante funzione `exit()`), a sua volta definito nella libreria `<utility.h>` mediante una famiglia di costanti. Per la descrizione delle costanti usate si consiglia di consultare l'apposito dizionario.

Il server non invia alcun output su STDOUT e redirige i messaggi di errore sullo STDERR solo in caso di terminazione. Client e agent stampano su STDOUT gli errori che potrebbero essere rilevanti per l'utente.

DIZIONARIO ERRORI

Nome	Valore	Descrizione
ESOCK_CREATE	5	Errore nella creazione della socket.
ESOCK_BIND	6	Errore durante il binding della socket.
ESOCK_LISTEN	7	Errore durante la preparazione della socket ad accettare le connessioni.
ESOCK_CONN	8	Errore durante la connessione ad una socket.
ESOCK_OPT	9	Errore durante l'impostazione delle opzioni di una socket.
ESYS_INFO	10	Errore durante il recupero delle informazioni sul sistema.
EWRITE	11	Errore fatale durante la scrittura sul file.
EREAD	12	Errore fatale durante la lettura da file.
EIP_NOTVALID	13	Errore durante la conversione ASCII to NETWORK.
EARGS_NOTVALID	14	Errore nel passaggio dei parametri da riga di comando.
EPORT_NOTVALID	15	Porta fuori dal range 1024-65535.
ETHREAD_CREATE	16	Errore durante la creazione del thread.
ESIGNAL	17	Errore durante la definizione di un handler per il segnale.