Cátedra de Sistemas Operativos II

Trabajo Práctico N° III

Piñero, Tomás Santiago

4 de Junio de 2020

Índice

1. Introducción

1.1. Objetivo

El objetivo del trabajo es lograr una visión *end to end* de una implementación básica de una *RESTful API* sobre un sistema embedido.

1.2. Definiciones, Acrónimos y Abreviaturas

- *API*: *Application Programming Interface*. Es un conjunto de funciones ofrecidas para ser utilizadas por otro software.
- *REST*: *Representational State Transfer*. Es un diseño de software para los servicios web. Estos servicios web son llamados *RESTful Web services* y proveen operaciones entre computadoras en Internet. [1]
- *Framework*: es una estructura conceptual y tecnológica de soporte definido, normalmente con módulos de software concretos, que puede servir de base para la organización y desarrollo de software.[2]

2. Descripción general

2.1. Restricciones

A las restricciones dadas (ver `Enunciado.pdf`) se le agregaron las siguientes:

1. El servidor en el que se instalen los servicios deben tener instaladas las dependencias necesarias:
    - *Ulfius library*[3].
    - *Nginx*[4].
2. El usuario con el cual que se ejecuten los servicios, `usertp3`, ya debe existir en el servidor. El *Makefile* no se encarga de crear dicho usuario.
3. El usuario mencionado en el ítem anterior debe tener permisos para ejecutar el comando `useradd` sin la petición de contraseña.
4. Debe existir el archivo `/etc/.nginxpasswd`. Este archivo es el que *Nginx* utiliza para permitir el acceso a los servicios.
5. El ciente debe tener en `/etc/hosts` la dirección IP del servidor asociada al dominio `tp3.com`.

2.2. Esquema del proyecto

El proyecto está dividido en las siguientes carpetas:

- **bin**: contiene los archivos ejecutables de los servicios.
- **inc**: contiene el *header 'utilities.h'*, que contiene las funciones útiles para los servicios: escribir el *log* de eventos y la obtención del *timestamp*.
- **config**: contiene los archivos de configuración para *Nginx* y los servicios para *systemd*.
- **src**: contiene los códigos fuente.
  - *status_service.c*: crea el servicio de estado en el puerto `8081`.
  - *users_service.c*: crea el servicio de usuarios en el puerto `8080`.
  - *utilities.c*: implementa las funciones del *utilities.h*.

2.3. Requisitos futuros

- Crear una página que consuma los datos devueltos por los servicios.

3. Diseño de solución

Como se mencionó en la sección anterior, se organizó el proyecto en varias carpetas (ver **2.2**).

Se decidió crear dos archivos `.c`, uno para cada servicio. "*users_service*" se encargará de proveer los servicios de `/api/users` y "*status_service*" proveerá los servicios de `/api/servers/hardwareinfo`.

Para realizar el archivo *log*, se creó `utilities.c`, que contiene la función para escribir el archivo y otras funcionalidades que sean compartidas entre los servicios.

3.1. *Makefile*

Las recetas disponibles en el *Makefile* son las siguientes:

1. *all*: genera el ejecutable del sistema.
2. *check*: corre *cppcheck* sobre el directorio `src`.
3. *doc*: genera la documentación del código en HTML utilizando *doxygen* en la carpeta llamada `Doc` y la abre en el navegador *Firefox*.[1]
4. *install*: se encarga de copiar los archivos de configuración de *nginx* y de los servicios para *systemd* a los directorios correspondientes.
5. *clean*: elimina los directorios `bin` y `Doc`.

[1] Si no se encuentra instalado, se sugiere cambiar el navegador en la receta.

4. Implementación y resultados

4.1. Servicios

En un principio se utilizó el archivo *helloworld* ofrecido por *Ulfius* para comprender el funcionamiento del *framework*.

Posteriormente, se crearon los archivos mencionados en la sección anterior, uno para cada serivicio, con sus *endpoints* correspondientes. Cada uno de éstos consistía simplemente en devolver un mensaje de tipo *string* para corroborar el funcionamiento.

4.1.1. *Users*

Para obtener el listado de usuarios, se implementó la funcón *get_ users()*, que devuelve un objeto *JSON* con la información de los usuarios.

Esta función es utilizada en el *callback_ users_ get*, que se ejecuta cada vez que el servicio recibe una *request HTTP* mediante el método *GET*.

Código 1: Función '*get_ users()*'.

```
1  json_t *get_users()
2  {
3    int users = 0;
4    json_t *json_users_object = json_object();
5    json_t *json_users_array  = json_array();
6
7    json_object_set_new(json_users_object, "data", json_users_array);
8
9    struct passwd *passwd_info = getpwent();
10
11   while(passwd_info)
12     {
13       users++;
14       json_t *json_array_data;
15       json_array_data = json_pack("{s:i, s:s}", "user_id", passwd_info->
             pw_uid,
16                                   "username", passwd_info->pw_name);
17
18       json_array_append(json_users_array, json_array_data);
19
20       passwd_info = getpwent();
21     }
22
23   endpwent();
24
25   syslog(LOG_NOTICE, "Usuarios listados: %d\n", users);
26   char msg[BUFF];
27   sprintf(msg, "Usuarios listados: %d\n", users);
28   write_log(get_date(), USER, msg);
29
30   return json_users_object;
31 }
```

En cuanto al método para agregar un usuario nuevo, lo primero que se realiza al recibir una petición, es obtener un objeto *JSON* de la misma y verificar que el formato es el correcto. De no serlo, se evnía una respuesta con código 400 y una descripción "*Bad request*". Caso contrario, se revisa si el usuario ya existe en el sistema, sino existe, se lo crea y se devuelve la información.

Código 2: *Endpoint /users, POST method.*

```
int callback_users_post(const struct _u_request * request,
                        struct _u_response * response, void * user_data)
{
  json_t *json_request = ulfius_get_json_body_request(request, NULL);
  json_t *body;

  const char *user   = json_string_value(json_object_get(json_request, "
     username"));
  const char *passwd = json_string_value(json_object_get(json_request, "
     password"));

  if(user == NULL || passwd == NULL)
    {
      body = json_pack("{s:s}", "description", "Bad request");
      ulfius_set_json_body_response(response, 400, body);
      return U_CALLBACK_CONTINUE;
    }

  struct passwd *info = getpwnam(user);
  if(info != NULL)
    {
      body = json_pack("{s:s}", "description", "El usuario ya existe");
      ulfius_set_json_body_response(response, 409, body);
      return U_CALLBACK_CONTINUE;
    }

  char cmd[BUFF];
  sprintf(cmd, "sudo useradd %s -p $(openssl passwd -5 '%s')", user, passwd
     );
  char *tmp = get_popen(cmd);
  free(tmp);

  char *timestamp = get_date();

  struct passwd *passwd_info = getpwnam(user);

  body = json_pack("{s:i, s:s, s:s}", "id", passwd_info->pw_uid,
                   "username", user, "created_at", timestamp);

  ulfius_set_json_body_response(response, 200, body);

  syslog(LOG_NOTICE, "Usuario %d creado", passwd_info->pw_uid);

  char msg[BUFF];
  sprintf(msg,"Usuario %d creado\n", passwd_info->pw_uid);
  write_log(timestamp, USER, msg);
  free(timestamp);

  return U_CALLBACK_CONTINUE;
}
```

4.1.2. *Status*

La obtención de los datos pedidos se guardan un una estructura llamada *hwinfo*. Esta, a su vez, contiene un dato tipo *upinfo*, que es otra estructura que contiene los datos que indican el tiempo transcurrido desde que se inició el sistema.

Código 3: Estructura '*upinfo*'.

```
typedef struct uptime   /** Estructura para obtener el uptime. */
{
  int time; /** Uptime total en segundos. */
  int h;    /** Horas de uptime. */
  int min;  /** Minutos de uptime. */
  int seg;  /** Segundos de uptime. */
} upinfo;
```

Código 4: Estructura '*hwinfo*'.

```
typedef struct hardinfo /** Estructura para obtener la informacion del
    hardware. */
{
  char *kernelVersion; /** Version de kernel. */
  char *processorName; /** Procesador. */
  int totalCPUCore;    /** Cantidad de nucleos del procesador. */
  int totalMemory;     /** Cantidad de memoria RAM total en MB. */
  int freeMemory;      /** Cantidad de memoria RAM libre en MB. */
  int diskTotal;       /** Capacidad del disco en GB. */
  int diskFree;        /** Espacio libre del disco en GB. */
  float loadAvg;       /** Carga promedio del sistema en 1 minuto. */
  upinfo uptime;       /** Cuanto tiempo hace que se inicio el sistema. */
} hwinfo;
```

La función '*get_hwinfo*' se encarga de completar estas estructuras, que son utilizados en la función *callback_status*, donde se arma un objeto *JSON* con esta estructura.

Código 5: Función '*get_hwinfo*'.

```
void get_hwinfo(hwinfo *hard)
{
  hard->kernelVersion = get_popen("uname -r");

  char *tmp = get_popen("cat /proc/loadavg");
  sscanf(tmp, "%f", &hard->loadAvg);

  tmp = get_popen("cat /proc/uptime");
  sscanf(tmp, "%d", &hard->uptime.time);

  hard->uptime.h   = hard->uptime.time/60/60%24;
  hard->uptime.min = hard->uptime.time/60%60;
  hard->uptime.seg = hard->uptime.time%60;

  tmp = get_popen("free --mega | grep Mem");
  sscanf(tmp, "%*s %d %*d %d", &hard->totalMemory, &hard->freeMemory);

  tmp = get_popen("lscpu | grep name");
  tmp = remove_spaces(tmp);
  hard->processorName = tmp;

  tmp = get_popen("lscpu | grep socket");
```

```
23    tmp = remove_spaces(tmp);
24    sscanf(tmp, "%d", &hard->totalCPUCore);
25
26    tmp = get_popen("df . | grep /");
27    sscanf(tmp, "%*s %d %*d %d", &hard->diskTotal, &hard->diskFree);
28    hard->diskTotal /= 1024000;
29    hard->diskFree /= 1024000;
30
31    free(tmp);
32 }
```

4.2. *Nginx*

Para la configuración de *nginx*, se siguió la documentación provista por los desarrolladores[5], creando un archivo en el directorio `/etc/nginx/sites-available` llamado `tp3so2`.

Código 6: Configuración del sitio `tp3.com`.

```
server {
        listen :80;

        server_name tp3.com;

        auth_basic            "TP3 Area";
        auth_basic_user_file /etc/.nginxpasswd;

        location /api/users {
                proxy_pass http://localhost:8080/users;
        }

        location /api/servers/hardwareinfo {
                proxy_pass http://localhost:8081/hardwareinfo;
                proxy_set_header X-Real-IP $remote_addr;
        }
}
```

El Código **6** establece el nombre del servidor, el nombre del área protegida a la cual únicamente los usuarios registrados en `/etc/.nginxpasswd` tienen acceso y los *locations* del servidor.

4.3. *systemd*

Por último, con *nginx* y los binarios de los servicios funcionando correctamente se procedió a crear los archivos necesarios para habilitarlos como servicios. El archivo `.service` tiene la siguiente forma:

Código 7: Estructura del servicio.

```
[Unit]
Description=[Descripcion del servicio]
Requires=nginx.service
After=nginx.service

[Service]
Type=simple
ExecStart=/usr/bin/[ejecutable_servicio]
User=usertp3

[Install]
WantedBy=multi-user.target
```

Los dos archivos correspondientes a los servicios se ubican en el directorio `/etc/systemd/system/`.

Además del *log* que se pidió, se realiza un *log* en *journal* mediante la función *syslog()*. Esto se realizó para obtener tener una copia en otro directorio del sistema, `/var/log/` y cumplir con el formato requerido por la cátedra.

4.4. Resultados

El sistema se encuentra corriendo en una *Raspberry Pi 3B+* conectada a la misma red WiFi.

La siguiente figura muestra el resultado de las pruebas implementadas en *Postman*.

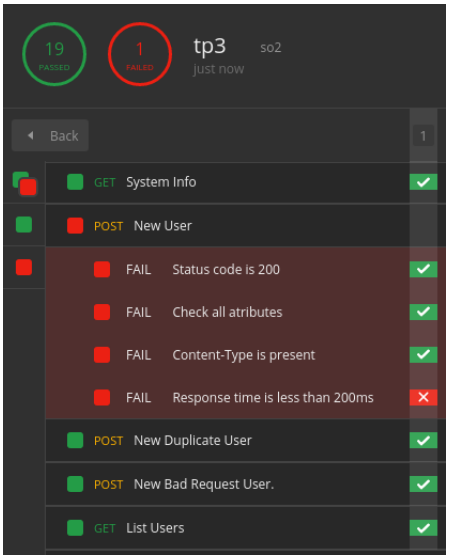

Figura 1: Resultados de los *tests* en *Postman*.

Como se puede ver, el tiempo menor a 200ms de requerimiento para el método de agregar un usuario nuevo no se cumple. Se realizaron varios cambios en la implementación y el tiempo mínimo que se pudo lograr fue de 300ms.

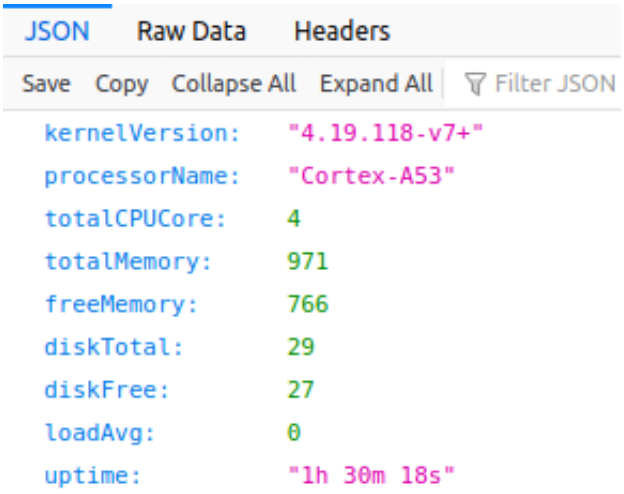

Figura 2: Información del hardware del servidor.

La Fig. **2** muestra el resultado de la consulta a *hardwareinfo*. Los valores correspondientes a la memoria se muestran en [MB] y los valores de disco en [GB].

5. Conclusiones

El trabajo práctico no fue complicado de implementar, a diferencia de los anteriores.

El único requerimiento que no se pudo cumplir es el del tiempo de respuesta de la solicitud para agregar el usuario, que puede deberse tanto a detalles de la implementación como de la latencia en la red.

Referencias

[1] *Wikipedia, REST*,
`Link to Wikipedia`

[2] *Wikipedia, Framework*,
`Link to Wikipedia`

[3] *Github, Ulfius*,
`Link to Ulfius`

[4] *Nginx*,
`Link to Nginx`

[5] *Nginx, Documentation*,
`Link to Nginx documentation`