*

Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA
División de Ciencias Naturales e Ingeniería
Departamento de Matemáticas Aplicadas y Sistemas
$ Licenciatura en Ingeniería en Computación

Sistemas Distribuidos - 23 Otoño
Práctica 2: *llamadas a procedimientos remotos (RPC)*
Equipos de 2 personas máximo

Objetivo: comprender las ventajas de utilizar un *middleware* que haga transparente el envío y recepción de mensajes para construir una aplicación cliente/servidor.

Programa ejemplo

Enviar mensaje: bajar de aquí los archivos con los códigos fuente.

Protocolo de impresión remota (escrito en IDL)

```
/* msg.x */
program MESSAGEPROG
{
  version MESSAGEVERS
  {
    int PRINTMESSAGE(string) = 1;
  } = 1;
} = 0x20000001;
```

Implementación del procedimiento remoto (el servidor)

```
/* msg_miservidor.c: implementación del procedimiento
"printmessage" */

#include <stdio.h>
#include <rpc/rpc.h>  /*Siempre necesario*/
#include "msg.h"      /*Generado por rpcgen */

/* implementación de "printmessage" */
int *printmessage_1_svc(char **msg)
{
    static int result; /* Debe ser estática*/
    FILE *f;
    f = fopen("/dev/console", "w");
    if (f == NULL)
    {
        result = 0;
        return (&result);
    }
    fprintf(f, "%s\n", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

Cliente del servicio remoto

```c
/* msg_micliente.c: cliente del servidor. */

#include <stdio.h>
#include <rpc.h> /* Siempre se requiere para rpc*/
#include "msg.h" /* msg.h será generado por rpcgen */

int main(int argc, char *argv[])
{
   CLIENT *cl; /* referencia hacia el servidor */

   int *result;
   char *server;
   char *message;

   if (argc != 3)
   {
     fprintf(stderr,"uso: %s host mensaje\n", argv[0]);
     exit(1);
   }

   /* Salva los valores de la línea de comandos*/
   server = argv[1];
   message = argv[2];

/* Crea la referencia del Cliente para llamar MESSAGEPROG en el
* servidor pasado como parámetro. Indicamos que el protocolo a
* utilizar es "tcp" */

   cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");

   if (cl == NULL)
   { /* No se estableció comunicación con el cliente */
     clnt_pcreateerror (server);
     exit(1);
   }

/* Invocación al procedimiento remoto "printmessage" */
   result = printmessage_1(&message, cl);

  if (result == NULL)
```

```c
  { /* Error ocurrido durante la invocación */
    clnt_perror (cl, server);
    exit(1);
  }

/* Llamado exitoso */
  if (*result == 0)
  { /* No se pudo imprimir el mensaje */
    fprintf(stderr, "%s: %s no pudo imprimir su mensaje\n",
    argv[0], server);
    exit(1);
  }

/* El mensaje se imprimió en la consola del servidor */
   printf("Mensaje entregado a %s!\n", server);
   exit(0);
}
```

Para compilar

El servidor:

```
› gcc msg_miservidor.c msg_svc.c -o servidor
```

El cliente:

```
› gcc msg_micliente.c msg_clnt.c -o cliente
```

Para ejecutar

Ejecutar primero el servicio de rpcbind:

```
› sudo systemctl start rpcbind
```

El servidor:

```
› ./servidor
```

El cliente:

```
› ./cliente localhost “Hola mundo”.
```

Actividades

Un viaje más a la mini aplicación UBER

En esta práctica tienen que volver a implementar la aplicación de UBER que hicieron en la práctica anterior, pero esta vez usarán *llamadas a procedimientos remotos* (la implementación *C*-RPC). Además, deben agregar otras funcionalidades para hacer más completa su aplicación.

Los elementos de la aplicación serán los siguientes:

- **Solamente tendrán un servidor monohilado** (el *main*)**:** de manera similar a la práctica anterior, el servidor tiene como tarea principal administrar las solicitudes de viajes y llevar el registro de autos disponibles.
    - Al menos deben tener 8 autos en total, pero en cualquier caso, se recomienda usar una constante o variable ***N*** que indique el número de autos del servidor.
- En contraste con la práctica anterior, además de registrar solamente la disponibilidad de los autos, el servidor ahora debe mantener más información:
    - **Disponibilidad:** un campo booleano que indique si el auto está disponible o no.
    - **Posición**: la posición serán las coordenadas (x, y) del auto. Al arrancar el servidor, se generarán, de manera aleatoria, las posiciones iniciales de los ***N*** autos. Para simplificar la aplicación, la posición no se actualizará durante los viajes, sino hasta que un pasajero haya terminado su viaje en ese auto.
    - **Tipo de uber**: los tres tipos posibles, UberPlanet, UberXL y UberBlack.
    - **Tarifa del auto**: el costo por kilómetro de viaje. En nuestra práctica supondremos que las unidades de las posiciones son kilómetros. Es decir, cuando calculen distancias, el número resultante representará kilómetros. Las tarifas son las siguientes:
        - UberPlanet, $10 por kilómetro.
        - UberXL, $15 por kilómetro.
        - UberBlack, $25 por kilómetro.
    - **Placa del auto**: una cadena de 6 símbolos, 3 dígitos seguidos de 3 letras.
- Los servicios que ofrecerá el servidor serán los siguientes:
    - `InfoAuto` **`SolicitarViaje(`**`PosiciónPasajero`**`)`**: el servidor revisará entre los autos libres, el más cercano a la posición del pasajero (no importa el tipo de Uber). Si hay un auto libre, devolverá la información del viaje en `InfoAuto` (posición, tipo de uber, tarifa y placa). En caso de no encontrar, regresará **NULL**.

- **`void TerminarViaje(PosiciónFinal, costoViaje, placas)`**: con este procedimiento el cliente indica que terminó el viaje. El trabajo del servidor será registrar que el auto con las **`placas`** dadas ya está libre, cambiar la posición a la **`PosiciónFinal`** del pasajero y agregar el costo del viaje a la ganancia total.
- **`InfoServicio EstadoServicio( )`**: este procedimiento regresa el estado de los viajes al momento. Es decir, el número de viajes realizados, el número de autos libres, y la ganancia hasta el momento.

- **Los clientes**: nuevamente, para simplificar la implementación, en lugar de hacer dos implementaciones, deben hacer un solo programa cliente al cual le deben pasar desde la línea de comandos el rol del cliente (pasajero o administrador).
  - *Pasajero*: el pasajero debe comenzar con posiciones de origen y destino generadas de manera aleatorio. Cada coordenada de las posiciones puede ser un entero en el intervalo [0, 50].
    - Al inicio, el pasajero debe invocar **`SolicitarViaje.`**
    - Si encuentran un viaje, entonces deben simular el viaje con un *sleep*, pero **proporcional a la distancia entre las posiciones de origen y destino**.
    - Posteriormente debe invocar **`TerminarViaje.`** No olviden que el cliente debe enviar en la llamada el costo del viaje según la tarifa que le dieron y la distancia recorrida.
    - Si no consigue un auto, el cliente simplemente debe terminar indicando este hecho.
  - *Administrador*: este tipo de cliente debe estar en un ciclo sin fin invocando cada 2 segundos el servicio **`EstadoServicio`** y mostrando en la consola el estado que le regresen.

Importante

Hay 2 retos que deben resolver:

1. Para los argumentos de entrada y salida de los procedimientos del servidor, deben investigar cómo definir variables compuestas en IDL, así como también cómo usarlos en su código.
2. También deben encontrar la manera de cargar la lista de autos disponibles. Recuerden que el servidor RPC no tiene un procedimiento *main* accesible, sino que consiste de una serie de procedimientos que invocan los clientes.

Entregables

Un reporte que debe contener:

1. Una descripción breve (2-3 líneas) de al menos 4 aplicaciones importantes que usen RPC (¿cómo lo usan?). Por ejemplo, el sistema *Network File System* (NFS) está implementado con RPC.
2. Manual técnico: explicación de las partes más importantes de su código. Por ejemplo, cómo resolvieron los 2 retos mencionados arriba o cómo buscan el mejor auto disponible. También deben incluir capturas de pantalla de un ejemplo donde los procesos servidor y cliente(s) estén en computadoras diferentes.
3. Deben enviar un video corto donde se muestre al servidor recibiendo peticiones desde otras computadoras en red. No es necesario poner audio en el video.