

Conceptos Avanzados de Sistemas Operativos: Prácticas

Aitor Acedo, Sergio Paracuellos

Índice

1. Introducción	4
2. Programacion cliente-servidor basada en paso de mensajes	4
2.1. Decisiones de diseño	4
2.1.1. Diseño del servidor	4
2.1.2. Diseño del cliente	5
2.2. Conclusiones de diseño	5
3. Código de la aplicacion	5
3.1. Código del servidor con procesos	5
3.2. Código del servidor con hilos	9
3.3. Código del cliente	12
4. Programación RPC	17
4.1. Código de la práctica de rpc	17
4.2. Código del servidor rpc	19
4.3. Codigo del cliente rpc	21

Índice de figuras

1.	Servidor de primos con fork()	8
2.	Servidor de primos con threads	12
3.	Cliente de primos	16
4.	fichero de especificación RPC	18
5.	Servidor RPC	21
6.	Cliente RPC	22

1. Introducción

Antes de comenzar con el comentario del desarrollo de las prácticas queremos introducir en esta sección como hemos planificado las tareas previas para llevar a cabo este proceso.

Hemos montado un servidor de **cv**s para poder realizar la tarea con control de las distintas versiones y poder trabajar cómodamente cada uno en nuestra casa. Hemos modificado el comportamiento de este **cv**s para que cada vez que se produzca un cambio nos lo notifique enviandonos un mail a una lista de correo que hemos montado para este propósito.

Después de estas consideraciones prerealización de las practicas y documentación, pasaremos a analizar cada una de las prácticas propuestas para esta asignatura.

2. Programacion cliente-servidor basada en paso de mensajes

El objetivo de la practica es el análisis de una aplicación y la programación de diversas soluciones cliente-servidor utilizando como mecanismos de comunicación y/o sincronización sockets. La aplicación está basada en un servidor que realizará el trabajo especificado en las funciones *cuenta_primos()*, *encuentra_primos()* y *esprimo()*. Los clientes podran pedir de estos servicios al servidor.

2.1. Decisiones de diseño

Una vez estudiado un poco lo que se nos pide para esta práctica, tuvimos que decidir cuál era el mejor diseño tanto para el cliente como para el servidor. En primer lugar veamos que decisiones se tomaron para el servidor.

2.1.1. Diseño del servidor

Tuvimos que tener en cuenta factores que nos ayudaron un poco a la hora de nuestra decisión. Por ejemplo:

- Recuperación de errores de transmisión.
- Concurrencia/paralelismo en atención de múltiples clientes por parte del servidor.

Diseñamos dos servidores. El primer servidor está basado en gestión de peticiones por medio de un proceso. Como sabemos, esto es bastante sencillo, y basta con que cada petición una vez realizado el *accept()*, su descriptor lo tome un proceso hijo distinto mediante *fork()*. Este diseño se utiliza en algunos servidores conocidos, como por ejemplo el servidor web *apache* en sus versiones anteriores al 2. Esta solución nos era válida, pero la creación de un proceso es algo que es lento ya que se tiene que preparar su contexto y otras cosas como ya sabemos. Viendo ésto, nos surgio la idea de ¿por qué en vez de procesos no usamos *threads*? Así es que el siguiente diseño del servidor lo hicimos con

”procesos ligeros.” como bien hemos dicho, *threads* (apache en sus versiones a partir de la 2). La creación de un hilo es una tarea bastante más rápida que la creación de un proceso, aunque también tiene sus problemillas y cosas a tener en cuenta a la hora de la programación. Por ejemplo, hay que tener en cuenta que si se lanzan varios hilos y tenemos una variable global, esta variable es común a todos ellos, y habría que implementar mecanismos para asegurar el acceso en exclusión mutua de dicha variable (semáforos). Otra cosa que hay que tener en cuenta, es que las funciones que lanzan los hilos, solo pueden tener un parámetro (al igual que ocurrirá con *rpc*, como ya veremos) y por tanto hay que diseñar estructuras de datos para poder pasar parámetros a la función del hilo y que todo esto funciones correctamente.

2.1.2. Diseño del cliente

Para diseñar los clientes, nos hemos basado ya en un diseño considerando la existencia de múltiples servidores. Así pues los clientes, tienen que dividir su trabajo, es decir, crear subintervalos con los números primos y cada subintervalo será mandado a un servidor diferente. De esta forma, lo único que tenemos que tener en cuenta es la selección de a que servidor mandarle el subintervalo, y por otro lado, tener cuidado con las actualizaciones de los datos a la hora de hacer subdivisiones en intervalos. Lo solucionamos con la función de librería *select()* que elige un descriptor de un conjunto de descriptors (servidores disponible) inicializado anteriormente.

2.2. Conclusiones de diseño

Tras comentar nuestras decisiones de diseño, queremos destacar que creemos que hemos alcanzado un alto grado de paralelismo y concurrencia, al igual que una aplicación que se comporta correctamente y parece robusta.

3. Código de la aplicación

Veamos en este apartado el código escrito para que todo esto funcione. Hemos puesto comentarios, así es que su comprensión creemos que es bastante sencilla.

3.1. Código del servidor con procesos

```
/*
 * SERVIDOR.C
 * Servidor de numeros primos v1.0
 * (copyleft) Aitor Acedo y Sergio Paracuellos Gutierrez
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

3.1 Código del servidor con procesos

```
#include "servidor.h"

#define MAXCONEXIONES 100
#define MAX_TAM_BUFFER 256
#define TRUE 0
#define FALSE 1

/* TRUE si es primo */

int esprimo (long n) {
    long i;
    for (i = 2; i * i <= n; i++) {
        if ((n % i) == 0)
            return TRUE;
    }
    return FALSE;
}

/* cuenta primos entre MAX Y MIN */
long cuenta_primos (long min, long max) {
    long i, contador = 0;
    for (i = min; i < max; i++)
        if (esprimo (i))
            contador++;
    return contador;
}

/* devuelve un vector con los primos */
long encuentra_primos (long min, long max, long *vector) {
    long i, contador = 0;
    for (i = min; i < max; i++)
        if (esprimo (i))
            vector[contador++] = i;
    return contador;
}

/* MAIN */
main (int argc, char *argv[]) {
    int fd, fd2; /* los ficheros descriptores */
    pid_t pid;
    struct sockaddr_in server; /* para la información de la dirección del servidor */
    struct sockaddr_in client; /* para la información de la dirección del cliente */
    int sin_size;
    int port; /*puerto para conectarse */
    FILE * log_file;
    long buffer[MAX_TAM_BUFFER];
    long longitud, i;
    long max, min, num_primos = 0;
    int numbytes;
    char peticion[MAX_TAM_BUFFER];

    /* comprobacion de argumentos */
    if (argc != 3) {
        perror ("Error en el numero de argumentos!!!\n");
        exit (1);
    }
}
```

3.1 Código del servidor con procesos

```
/* recogemos el puerto de linea de comandos */
port = atoi (argv[2]);
/* abrimos el fichero de log */
if ((log_file = fopen ("log.txt", "a")) < 0) {
    perror ("Error abriendo el fichero\n");
    exit (-1);
}

fprintf (log_file, "%s \t%s\n", argv[1], argv[2]);
fclose (log_file);

/* socket() */
if ((fd = socket (AF_INET, SOCK_STREAM, 0)) == -1) {
    perror ("error en socket()\n");
    exit (-1);
}

server.sin_family = AF_INET;
server.sin_port = htons (port);
server.sin_addr.s_addr = INADDR_ANY;
bzero (&(server.sin_zero), 8);

/* bind() */
if (bind (fd, (struct sockaddr *) &server, sizeof (struct sockaddr)) == -1) {
    perror ("error en bind() \n");
    exit (-1);
}

/* Listen() */
if (listen (fd, MAXCONEXIONES) == -1) {
    perror ("error en listen()\n");
    exit (-1);
}

printf ("Aceptando conexion.....\n");

/* bucle indefinido de aceptacion de conexiones */
while (1) {
    sin_size = sizeof (struct sockaddr_in);
    fd2 = accept (fd, (struct sockaddr *) &client, &sin_size);
    pid = fork ();
    switch (pid) {
        case -1:
            perror ("Error creando proceso\n");
            exit (1);
            break;
        case 0:
            /* accept */
            if (fd2 < 0) {
                perror ("error en accept gestionado port proceso\n");
                exit (1);
            }
            printf
                ("Se obtuvo una conexión desde%s gestionada por un proceso con pid%d\n",
                 inet_ntoa (client.sin_addr), getpid ());

            /* recibimos la petición deseada del cliente */
```

3.1 Código del servidor con procesos

```
if ((numbytes = read (fd2, peticion, MAX_TAM_BUFFER)) == -1) {
    perror ("Error recibiendo la petición\n");
    exit (1);
}
/* recibimos el minimo */
if ((numbytes = read (fd2, &min, sizeof (long))) == -1) {
    perror ("Error en revc\n");
    exit (1);
}
/* recibimos el maximo */
if ((numbytes = read (fd2, &max, sizeof (long), 0)) == -1) {
    perror ("Error en revc\n");
    exit (1);
}
if ((strcmp (peticion, "cuenta_primos")) == TRUE) {
    /* contamos los primos en ese intervalo */
    num_primos = cuenta_primos (min, max);
    /* mandamos los primos al cliente */
    if ((write (fd2, &num_primos, sizeof (long))) == -1) {
        perror ("Error en el envio\n");
        exit (1);
    }
} else if ((strcmp (peticion, "encuentra_primos")) == TRUE) {
    /* encontramos la lista de primos en ese intervalo */
    longitud = encuentra_primos (min, max, buffer);
    /* mandamos la longitud del vector de primos al cliente */
    if ((numbytes = write (fd2, &longitud, sizeof (long))) == -1) {
        perror ("Error mandando la longitud del vector de primos\n");
        exit (-1);
    }
    /* mandamos la lista de primos al cliente */
    for (i = 0; i < longitud; i++) {
        if ((numbytes = write (fd2, &buffer[i], sizeof (long))) == -1) {
            perror ("Error en el envio de la lista de primos\n");
            exit (1);
        }
    }
} else {
    printf ("Petición no disponible de momento\n");
}
default:
    break;
}
}
close (fd2);
}
```

Figura 1: Servidor de primos con fork()

3.2. Código del servidor con hilos

```

/*
 * SERVIDOR_HILOS.C
 * Servidor de numeros primos v1.0
 * (copyleft) Aitor Acedo y Sergio Paracuellos Gutierrez
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
10

#define MAXCONEXIONES 100
#define MAX_TAM_BUFFER 256
#define TRUE 0
#define FALSE 1

/*
 * Atributos que tendremos que pasarle
 * a la funcion a la que llama el hilo.
 */
20
typedef struct atributos {
    int fd2;
    struct sockaddr_in client; /* para la información de la dirección del cliente */
} atrib;

/* prototipo de la funcion a la que llama el hilo */
void *gestiona_peticon (atrib * at);

/* TRUE si es primo */
30
int esprimo (long n) {
    long i;
    for (i = 2; i * i <= n; i++) {
        if ((n % i) == 0)
            return TRUE;
    }
    return FALSE;
}

/* cuenta primos entre MAX Y MIN */
40
long cuenta_primos (long min, long max) {
    long i, contador = 0;
    for (i = min; i < max; i++)
        if (esprimo (i))
            contador++;
    return contador;
}

/* devuelve un vector con los primos */
50
long encuentra_primos (long min, long max, long *vector) {
    long i, contador = 0;
    for (i = min; i < max; i++)
        if (esprimo (i))
            vector[contador++] = i;
    return contador;
}

```

3.2 Código del servidor con hilos

```
/* MAIN */
main (int argc, char *argv[]) {
    int fd; /* los ficheros descriptores */
    struct sockaddr_in server; /* para la información de la dirección del servidor */
    int sin_size;
    int port; /*puerto para conectarse */
    pthread_t id;
    atrib a;
    FILE * log_file;

    /* comprobacion de argumentos */
    if (argc != 3) {
        perror ("Error en el numero de argumentos!!!\n");
        exit (1);
    }

    /* recogemos el puerto de linea de comandos */
    port = atoi (argv[2]);

    /* abrimos el fichero de log */
    if ((log_file = fopen ("log.txt", "a")) < 0) {
        perror ("Error abriendo el fichero\n");
        exit (-1);
    }
    fprintf (log_file, "%s \t%s\n", argv[1], argv[2]);
    fclose (log_file);

    /* socket() */
    if ((fd = socket (AF_INET, SOCK_STREAM, 0)) == -1) {
        perror ("error en socket()\n");
        exit (-1);
    }

    server.sin_family = AF_INET;
    server.sin_port = htons (port);
    server.sin_addr.s_addr = INADDR_ANY;
    bzero (&(server.sin_zero), 8);

    /* bind() */
    if (bind (fd, (struct sockaddr *) &server, sizeof (struct sockaddr)) == -1) {
        perror ("error en bind() \n");
        exit (-1);
    }

    /* Listen() */
    if (listen (fd, MAXCONEXIONES) == -1) {
        perror ("error en listen()\n");
        exit (-1);
    }

    printf ("Aceptando conexion.....\n");

    /* bucle indefinido de aceptacion de conexiones */
    while (1) {
        sin_size = sizeof (struct sockaddr_in);
        a.fd2 = accept (fd, (struct sockaddr *) &a.client, &sin_size);
    }
}
```

3.2 Código del servidor con hilos

```
    /* creamos el hilo que llama a gestiona_peticion() */
    pthread_create (&id, 0, (void *(*)(void *)) gestiona_peticion, &a);
    /* esperamos a que el hilo acabe */
    pthread_join (id, 0);
}

close (a.fd2);
}
120

/* funcion a la que llama el hilo */
void *gestiona_peticion (atrib * at) {
    long buffer[MAX_TAM_BUFFER];
    long longitud, i;
    long max, min, num_primos;
    int numbytes;
    char peticion[MAX_TAM_BUFFER];

    /* accept */
    if (at->fd2 < 0) {
        perror ("error en accept gestionado port proceso\n");
        exit (1);
    }

    printf ("Se obtuvo una conexión desde%s\n",
            inet_ntoa (at->client.sin_addr));

    /* recibimos la peticion deseada del cliente */
    if ((numbytes = read (at->fd2, peticion, MAX_TAM_BUFFER)) == -1) {
        perror ("Error recibiendo la petición\n");
        exit (1);
    }

    /* recibimos el minimo */
    if ((numbytes = read (at->fd2, &min, sizeof (long))) == -1) {
        perror ("Error en revc\n");
        exit (1);
    }

    /* recibimos el maximo */
    if ((numbytes = read (at->fd2, &max, sizeof (long), 0)) == -1) {
        perror ("Error en revc\n");
        exit (1);
    }

    if ((strcmp (peticion, "cuenta_primos")) == TRUE) {
        /* contamos los primos en ese intervalo */
        num_primos = cuenta_primos (min, max);
        /* mandamos los primos al cliente */
        if ((write (at->fd2, &num_primos, sizeof (long))) == -1) {
            perror ("Error en el envio\n");
            exit (1);
        }
    }

    else if ((strcmp (peticion, "encuentra_primos")) == TRUE) {
        /* encontramos la lista de primos en ese intervalo */
        longitud = encuentra_primos (min, max, buffer);
    }
}
130
140
150
160
```

3.3 Código del cliente

```
/* mandamos la longitud del vector de primos al cliente */ 170
if ((numbytes = write (at->fd2, &longitud, sizeof (long))) == -1) {
    perror ("Error mandando la longitud del vector de primos\n");
    exit (-1);
}
/* mandamos la lista de primos al cliente */
for (i = 0; i < longitud; i++) {
    if ((numbytes = write (at->fd2, &buffer[i], sizeof (long))) == -1) {
        perror ("Error en el envio de la lista de primos\n");
        exit (1);
    }
}
} else {
    printf ("Petición no disponible de momento\n");
}
}
```

Figura 2: Servidor de primos con threads

3.3. Código del cliente

```
/*
 * CLIENTE.C
 * Cliente de números primos v1.0
 * (copyleft) Aitor Acedo y Sergio Paracuellos Gutiérrez
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>

#define MAX_SIZE_BUFFER 256
#define MAX_FD 10
#define MAX_PORT 10
#define TRUE 0
#define FALSE 1

/*
 * El protocolo que vamos a seguir es el de poner como segundo parametro
 * el numero de servidores a los que vamos a lanzar trozos de
 * los calculos que tenemos que hacer.
 */

int
main (int argc, char *argv[]) {
    int fd[MAX_FD], numbytes; /* ficheros descriptores */
}
```

3.3 Código del cliente

```
struct hostent **he; /* estructura que recibirá información sobre el nodo remoto */
struct sockaddr_in server; /* información sobre la dirección del servidor */
long i, j, longitud = 0;

FILE *log_file; /* fichero de log */

int puerto;
int puertos[MAX_PORT]; 40

int encontrado;
int num_serv;

struct nodo {
    char *ip;
    int port;
};

struct nodo *servidores; 50

long min = 1L;
long max = 100L;
long num_primos = 0;
long intervalo;
long numero;

long buffer[MAX_SIZE_BUFFER]; 60

char peticion[MAX_SIZE_BUFFER];

/*select */
/* leera el conjunto de descriptors de archivos */
fd_set readset;
int num_ready;
int num_now;
int num_fds;
int k;
/* control de hora */ 70
time_t hora1, hora2;
struct tm *hora_entrada;
struct tm *hora_salida;

/* comprobacion de argumentos */
if (argc < 4 && argc < (atoi (argv[2]) + 1)) {
    perror ("Error en el paso de parametros.\n");
    exit (1);
} 80

/* reservamos memoria para el hostname y el puerto */
num_serv = atoi (argv[2]);

/*
 * Hemos reservado el tamaño total para el vector y despues
 * tendremos que utilizar el indice para la selección
 * de una posicion
 */
```

3.3 Código del cliente

```
servidores = (struct nodo *) malloc (num_serv * sizeof (struct nodo));           90

k = 3; /*control de parametros */

for (i = 0; i < num_serv; i++) {
    servidores[i].ip = argv[i + k];
    servidores[i].port = atoi (argv[i + k + 1]);
    k++;
}

                                                                                   100

max = max / num_serv;
intervalo = max; /* dividimos entre los disponibles el numero m"ximo para repartirlo */

he = (struct hostent **) malloc (num_serv * (sizeof (struct hostent)));

for (i = 0; i < num_serv; i++) {
    if ((he[i] = gethostbyname (servidores[i].ip)) == NULL) {
        printf ("gethostbyname() error en servidor%d", i + 1);
    }
}                                                                                   110

/*SELECT*/
/* numero de descriptors = numero de servidores */
num_fds = num_serv;
num_now = num_fds;

printf ("Pulse ENTER para conectar. . .\n");

while (num_now > 0) {
    /* limpiamos el valor de set */
    FD_ZERO (&readset);
                                                                                   120

    for (j = 0; j < num_fds; j++) {
        fd[j] = j;
        if (fd[j] >= 0) {
            /* a"adimos el descriptor a set */
            FD_SET (fd[j], &readset);
        }
    }
                                                                                   130

    /* select necesita el descriptor de archivos maximo */
    if ((num_ready = select (num_serv, &readset, NULL, NULL, NULL)) == -1) {
        perror ("Error en select\n");
        exit (1);
    }
    printf ("Servidores disponibles: %d", num_ready);

    /* hora de entrada */
    time (&hora1);
    hora_entrada = localtime (&hora1);
    printf ("\tHora de comienzo: %d:%d:%d\n", hora_entrada->tm_hour,
        hora_entrada->tm_min, hora_entrada->tm_sec);
                                                                                   140

    for (j = 0; j < num_fds && num_ready > 0; j++) {
        /* si hay descriptor y esta establecido en set */
        if ((fd[j] >= 0) && FD_ISSET (fd[j], &readset)) {
```

3.3 Código del cliente

```
/* socket() */
if ((fd[j] = socket (AF_INET, SOCK_STREAM, 0)) == -1) {
    printf ("socket() error\n");
    exit (-1);
}

server.sin_family = AF_INET;
/* htons() es necesaria nuevamente */
server.sin_port = htons (servidores[j].port);
/*he->h_addr pasa la información de *he a "h_addr" */
server.sin_addr = *((struct in_addr *) he[j]->h_addr);
bzero (&(server.sin_zero), 8);

/* connect() */
if (connect(fd[j], (struct sockaddr *) &server,sizeof (struct sockaddr)) == -1) {
    printf ("connect() error\n");
    exit (-1);
}

printf ("Mandando datos al servidor. . .\n");

strcpy (peticion, argv[1]);

/* mandamos al servidor la peticion */
if (numbytes = write (fd[j], peticion, MAX_SIZE_BUFFER) == -1) {
    perror ("Envío de la peticion.\n");
    exit (1);
}

/* mandamos al servidor el valor minimo del intervalo */
if (numbytes = write (fd[j], &min, sizeof (long)) == -1) {
    perror ("Envío del mínimo.\n");
    exit (1);
}

/* mandamos al servidor el valor maximo del intervalo */
if (numbytes = write (fd[j], &max, sizeof (long)) == -1) {
    perror ("Envío del maximo.\n");
    exit (1);
}

if ((strcmp (peticion, "cuenta_primos")) == TRUE) {
    /* recibimos la respuesta del servidor con el numero de primos */
    if ((numbytes = read (fd[j], &num_primos, sizeof (long))) == -1) {
        perror ("Error en revc\n");
        exit (1);
    }

    num_ready--; /* atendida y por tanto uno menos disponible */

    printf("El numero de primos que me ha mandao el servidor es:%d\n",num_primos);

    /* actualizamos valores */
    numero += num_primos;
    min = max;
    max += intervalo;
}
```

```

num_now--;

/* hora de salida */
time (&hora2);
hora_salida = localtime (&hora2);
printf ("\tHora de finalizacion:%d:%d:%d\n",
        hora_salida->tm_hour, hora_salida->tm_min,
        hora_salida->tm_sec);
fflush (stdout);
fd[j] = -1;
/* si la peticion es devolver el vector de primos */
} else if ((strcmp (peticion, "encuentra_primos")) == TRUE) {
/* recibimos la longitud del vector de primos */
if ((numbytes = read (fd[j], &longitud, sizeof (long))) == -1) {
    perror ("Error en revc\n");
    exit (1);
}

printf ("La lista de primos obtenida del servidor es: ");

/* recogemos la lista de primos que nos manda el servidor */
for (i = 0; i < longitud; i++) {
    if ((numbytes = read (fd[j], &buffer[i], sizeof (long))) == -1) {
        perror ("Error recibiendo la lista de primos\n");
        exit (1);
    }
    printf ("%d ", buffer[i]);
}
printf ("\n");
num_ready--;
numero += num_primos;
min = max;
max += intervalo;
num_now--;
fd[j] = -1;
} else {
    printf ("No existe esa opcion\n");
    exit (1);
}
}
}
}
close (fd[j]);
/* liberamos memoria */
free (servidores);
}

```

Figura 3: Cliente de primos

4. Programación RPC

Las llamadas a procedimientos remotos (Remote Procedure Call) es conocido por el nombre de RPC. Digamos que se trata de una nueva forma de comunicación cliente-servidor. El RPC utiliza un formato XDR para la compatibilidad de los datos. El XDR son filtros que se encargan de traducir los datos al formato que la red necesita. Un programa RPC necesita ser registrado para que el cliente sepa que servicios le proporciona el servidor. Para ello necesitamos conocer (como clientes) del servidor lo siguiente:

- Nombre de la función.
- Parámetros que hay.
- Tipos de parámetros.

Por decirlo de alguna forma, esto constituiría la interfaz entre el cliente y el servidor (Interfaz Definition Languaje IDL). Para identificar los servicios RPC necesitamos 3 números:

- Número de programa: número de servidor RPC que existe.
- Número de versión.
- Número de procedimiento: identificador a cada una de las funciones que están en la máquina.

El servicio para realizar esto es el *portmapper*.

Tras esta pequeña introducción a lo que es el RPC, entraremos en materia. Debemos especificar los datos en un fichero con extensión *.x* y tras ejecutar *rpcgen fichero.x* nos generara archivos de cabecera y algunas cosas que utilizaremos en lo que es nuestra programación.

Tenemos que repetir la práctica anterior, pero esta vez utilizando llamadas a procedimientos remotos. Debemos introducir, además, comprobacion de errores y autenticación de los clientes en el servidor para ver si tienen permiso y cuentas para ejecutar dichos servicios.

En cuanto a las decisiones de diseño, hemos seguido el *Unix Distributed Programming* con sus ejemplos, que hemos comprendido y programado sin muchos problemas.

4.1. Código de la práctica de rpc

En primero lugar veamos el fichero de especificación *funcs.x*

```
/* Maximo tamaño del vector de primos que podemos devolver */
const MAXPRIMES = 1000;

/*
 * Devolucion de los valores del estado
 */
enum prime_status {
    STAT_OK = 1,
```

4.1 Código de la práctica de rpc

```
    STAT_BAD_RANGE = 2,
    STAT_BAD_AUTH = 3
};

/*
 * Parametros de entrada en una estructura ya que
 * solo podemos manejar un parámetro en llamadas RPC
 */
struct prime_request {
    int min;
    int max;
};

/*
 * Parámetros de salida para encontrar primos.
 * - Si es ok, entonces devolvemos el vector
 * - Si falla por algo, devolvemos void
 */
union find_result switch(prime_status status) {
    case STAT_OK:
        int array<MAXPRIMES>;
    case STAT_BAD_RANGE:
        void;
    case STAT_BAD_AUTH:
        void;
};

/*
 * Parámetros de salida para contar los primos.
 * - Si es ok, entonces devolvemos el contador
 * - Si algo falla por algo, devolvemos void
 */
union count_result switch(prime_status status) {
    case STAT_OK:
        int count;
    case STAT_BAD_RANGE:
        void;
    case STAT_BAD_AUTH:
        void;
};

/* definicion del programa */
program PRIMEPROG {
    version PRIME_AUTH_VERS {
        find_result FIND_PRIMES(prime_request) = 1;
        count_result COUNT_PRIMES(prime_request) = 2;
    } = 2; /* numero de version */
} = 0x2000009a; /* numero de programa */
```

Figura 4: fichero de especificación RPC

4.2. Código del servidor rpc

```

/* Servidor RPC */

#include <rpc/rpc.h>
#include <pwd.h>
#include <stdio.h>
#include "funcs.h"

/*
 * el _1 es el numero de la version
 */
find_result *find_primes_2 (prime_request *request,
                            struct svc_req *rqstp) {
    static find_result result;
    static int prime_array[MAXPRIMES];
    int i, count = 0;

    /* Validamos al usuario */
    if (validate_user(rqstp) == 0)
        result.status = STAT_BAD_AUTH;
    /* Validamos los parametros de entrada */
    else if (request->min >= request->max)
        result.status = STAT_BAD_RANGE;
    else {
        /* bucle dentro del rango mirando los primos */
        for (i=request->min;i<request->max;i++)
            if (isprime(i))
                prime_array[count++] = i;
        /* aqui sera correcto */
        result.status = STAT_OK;
        result.find_result_u.array.array_len = count;
        result.find_result_u.array.array_val = prime_array;
    }
    /* devolvemos el puntero a la estructura */
    return &result;
}

count_result *count_primes_2 (prime_request *request,
                              struct svc_req *rqstp) {
    static count_result result;
    int i, count = 0;

    /* validamos usuario */
    if (validate_user(rqstp) == 0)
        result.status = STAT_BAD_AUTH;
    /* validamos parametros de entrada */
    else if (request->min >= request->max)
        result.status = STAT_BAD_RANGE;
    else {
        /* miramos los primos */
        for (i=request->min;i<request->max;i++)
            if (isprime(i))
                count++;
        /* metemos el resultado */
        result.status = STAT_OK;
        result.count_result_u.count = count;
    }
}

```

```

    }
    /* devolvemos el resultado */
    return &result;
}
60

int isprime (int n) {
    int i;
    for (i=2;i*i<=n;i++) {
        if ((n % i) == 0)
            return 0;
    }
    return 1;
}

int validate_user(struct svc_req *rqstp) {
70
    struct authunix_parms *ucred;
    struct passwd *pwent;
    char *client_name;
    FILE *fd;
    char name_in_file[50];

    /* Sacamos por pantalla las credenciales para debuggear */
    fprintf(stderr, "Autorizacion flavor = %d\n",
        rqstp->rq_cred.oa_flavor);
    if (rqstp->rq_cred.oa_flavor != AUTH_UNIX)
80
        return 0; /* no es correcto */
    ucred = (struct authunix_parms *) (rqstp->rq_clntcred);
    /* debug */
    fprintf(stderr, "host = %s\n", ucred->aup_machname);
    fprintf(stderr, "uid = %s\n", ucred->aup_uid);
    fprintf(stderr, "gid = %s\n", ucred->aup_gid);

    /* mapeamos uid en el nombre de login de la maquina servidor */
    pwent = getpwuid (ucred->aup_uid);
    if (pwent == NULL)
90
        return 0; /* no tiene cuenta el usuario en la maquina */
    client_name = pwent->pw_name;
    /* debug */
    fprintf(stderr, "client name is = %s\n", client_name);

    if (fd = fopen("/etc/primeusers", "r") == NULL)
        return 0; /* No hay fichero de autentificación */

    /* leemos el fichero de autentificación */
    while (fscanf(fd, "%s", name_in_file) != EOF) {
100
        /* debug */
        fprintf(stderr, "Comparing %s with %s\n", client_name, name_in_file);
        if (strcmp(client_name, name_in_file) == 0) {
            fclose(fd);
            return 1; /* autentificación realizada con éxito */
        }
    }
    fclose(fd);
    return 0; /* usuario no esta en la lista */
}
110

```

Figura 5: Servidor RPC

4.3. Código del cliente rpc

```

/* cliente RPC */

#include <rpc/rpc.h>
#include "funcs.h"

main (int argc, char *argv[]) {
    int i;
    CLIENT *cl; /* manejador del cliente */
    count_result *result;
    prime_request request;

    /* comprobacion de argumentos */
    if (argc != 4) {
        printf("Uso: %s host min max\n", argv[0]);
        exit(1);
    }

    /* creamos conexion */
    cl = clnt_create(argv[1], PRIMEPROG, PRIME_AUTH_VERS, "tcp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(2);
    }

    /* creamos las credenciales UNIX */
    cl->cl_auth = authunix_create_default();

    /*
     * contruimos el paquete con los valores
     * min y max de la linea de comandos.
     */
    request.min = atoi(argv[2]);
    request.max = atoi(argv[3]);

    /* RPC CALL */
    result = count_primes_2(&request, cl);
    if (result == NULL) {
        clnt_perror(cl, argv[1]);
        exit(3);
    }

    /* Chequeamos el resultado */
    switch (result->status) {
        case STAT_OK:
            printf("Numero de primos encontrados %d\n",
                result->count_result_u.count);
            break;
        case STAT_BAD_RANGE:
            printf("Servidor nos rechaza: Rango Invalido\n");
            break;
    }
}

```

4.3 Código del cliente rpc

```
    case STAT_BAD_AUTH:
        printf("Servidor nos rechaza: Autorización Invalida\n");
        break;
}

/* liberamos memoria */
xdr_free(xdr_count_result, result);
}
```

Figura 6: Cliente RPC