Protocolo DBUS

Ejercicio N° 1

Objetivos	● Buenas prácticas en programación de Tipos de Datos Abstractos (TDAs) ● Modularización de sistemas ● Correcto uso de recursos (memoria dinámica y archivos) ● Encapsulación y manejo de Sockets
Instancias de Entrega	**Entrega 1:** clase 4 (4/05/2020). **Entrega 2:** clase 6 (18/05/2020).
Temas de Repaso	● Uso de structs y typedef ● Uso de macros y archivos de cabecera ● Funciones para el manejo de Strings en C ● Funciones para el manejo de Sockets
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Cumplimiento de la totalidad del enunciado del ejercicio ● Ausencia de variables globales ● Ausencia de funciones globales salvo los puntos de entrada al sistema (*main*) ● Correcta encapsulación en TDAs y separación en archivos ● Uso de interfaces para acceder a datos contenidos en TDAs ● Empleo de memoria dinámica de forma ordenada y moderada ● Acceso a información de archivos de forma ordenada y moderada

Índice

Introducción

¿Alguna vez se preguntaron cómo interactúan las distintas aplicaciones de escritorio entre sí? ¿Cómo sabe un navegador web el estado de la red, cómo funcionan los atajos de teclado, o incluso cómo aplicaciones como el editor Sublime sabe si existe una instancia abierta del mismo?

El siguiente ejercicio consiste en una implementación parcial del protocolo utilizado por el servicio D-bus. Se implementará la parte utilizada para realizar llamadas a funciones remotas, y la misma será implementada sobre el protocolo TCP.

Descripción

D-BUS Es un sistema de comunicación entre procesos y llamadas a procedimientos remotos, utilizado principalmente por sistemas Unix. El protocolo que utiliza es parte del proyecto *freedesktop.org*. Un resumen más extensivo del mismo puede encontrarse en este link.

El sistema D-Bus posee un servicio corriendo en segundo plano con 2 canales (buses) para comunicarse con el resto de las aplicaciones. Estos canales, llamados **system bus** y **session bus**. Están implementados sobre **sockets de tipo unix**.

En el siguiente trabajo práctico se deberá implementar una aplicación cliente y una servidor que se comuniquen sobre un socket TCP utilizando el mismo protocolo que D-BUS.

Cliente

El cliente debe realizar 2 acciones: Leer desde la entrada de datos las llamadas a procedimientos remotos, y enviar el pedido al servidor con el protocolo elegido

El cliente debe conectarse a un *host* y un *puerto*, utilizando el protocolo *TCP*. La entrada de datos puede ser un archivo cuya ruta es pasada en los argumentos del programa, o la entrada standard, si no se le pasó ningún argumento. Una vez conectado, el cliente procesa la entrada de datos leyendo linea por linea, envía la llamada a procedimiento remoto, y espera por la respuesta del servidor.

El protocolo con el que se comunica con el servidor se describe más adelante.

Formato de linea de comando

El cliente debe ser invocado con el siguiente parámetro:

```
./client <host> <puerto> [<archivo de entrada>]
```

Donde archivo de entrada es el nombre de un archivo de texto con las llamadas a realizar. Si este último

parámetro no es incluido, se interpretará la entrada estándar.

Servidor

El servidor debe esperar la conexión de un cliente, luego escuchar un petitorio y escuchar los mensajes del cliente. Por cada mensaje recibido se debe imprimir en pantalla la información descripta más adelante. Luego devuelve al cliente el texto "OK" (3 bytes, agregar el '\n' al final).

Formato de linea de comando

El cliente debe ser invocado con el siguiente parámetro:

```
./server <puerto>
```

Formato del protocolo D-Bus

El protocolo D-Bus es parte en texto plano (autenticación) y parte en binario (mensajes), y la especificación es bastante amplia, con una variedad de mensajes para transmitir. Nos centraremos en los mensajes de llamadas a procedimientos remotos únicamente, simplificando levemente el protocolo.

Un mensaje consiste de una cabecera (header) y cuerpo (body). La cabecera es un conjunto de valores binarios con un formato fijo. En el header se incluyen varios parámetros que definen el mensaje a enviar. En el body se envían los argumentos de los métodos que ejecutamos.

Formato de cabecera

El formato de la cabecera es

```
BYTE, BYTE, BYTE, BYTE, UINT32, UINT32, ARRAY of STRUCT of (BYTE,VARIANT)
```

La longitud de la cabecera debe ser múltiplo de 8, en caso de no serlo, se rellenan los bytes faltantes con ceros. El cuerpo no posee esta restricción.

Los parámetros ubicados en el array de parámetros también tienen que tener longitud múltiplo de 8.

- 1er byte: 'l' para little-endian, 'b' para big-endian. utilizaremos 'l' siempre
- 2do byte: tipo de mensaje, utilizaremos el valor 0x01 siempre, que es el valor para llamadas a métodos.
- 3er byte: flags varios, utilizaremos 0x0 siempre.
- 4to byte: versión del protocolo, utilizaremos 0x01 siempre.
- 1er entero: longitud en bytes del **cuerpo**.
- 2do entero: un número serie para identificar el mensaje. utilizaremos un valor incremental por cada mensaje enviado por el cliente. El primer mensaje tendrá el número 0x0001, el segundo 0x0002, etc...

- Por último un array de longitud variable con los parámetros necesarios según el tipo de mensaje.

Posee el siguiente formato:

- Un entero UINT32 con la longitud del array
- Por cada parámetro:
    - Un byte indicando el tipo de parámetro
    - Un byte en 1 (posiblemente indicando que tenemos sólo un "string")
    - Un byte indicando el tipo de dato (Utilizaremos sólo strings o equivalentes)
    - Longitud del dato en bytes. La longitud no toma en cuenta el padding del último elemento.

Ejemplo:

`06 01 73 00 0C 00 00 00` : Es un parámetro de tipo 6, que es 1 string (ya que 0x73 corresponde a la letra "s") y de longitud 0xC, es decir 12

Los parámetros necesarios que iran en ese array son:

- *ruta del objeto*: se identifica por el tipo 1, y su tipo de dato es "o" (a fines prácticos es un string)
- *destino*: se identifica por el tipo 6, y su tipo de dato es "s" (string UTF-8)
- *interfaz*: se identifica por el tipo 2, y su tipo de dato es "s"
- *método*: se identifica por el tipo 3 y su tipo de dato es "s"
- *firma*: opcional, se identifica por el tipo 9 y su tipo de dato es "g" (a fines prácticos también es un string). Los métodos invocados sólo tendrán argumentos de tipo string, por lo que la firma será una cadena formada por caracteres 's' y de longitud igual a la cantidad de argumentos utilizados por el método. Si el método no utiliza parámetros, no se envía. A diferencia de los otros tipos, no es seguido de un string. Ej: “`09 01 67 00 02 73 73 00`” : tipo 9 (`09`), 1 (`01`) parámetro tipo “g” (`67 00`), 2 (`02`) `parámetros: “ss” (73 73 00)`

Los strings que componen al array también tienen que tener longitud 8, y deben ser rellenados con ceros en caso contrario. Los strings deben terminar en '\0'

Los argumentos no necesariamente van a ser transmitidos en ese orden, pero notar que la longitud del array es la suma de la descripción de los parámetros más la longitud de todos los strings sin contar el padding del último.

Formato del cuerpo

Una vez leída la cabecera, si la misma posee una firma tenemos que leer el cuerpo con los parámetros que utiliza. Como mencionamos previamente, sólo utilizaremos strings. Al igual que en la cabecera los strings deben terminar en '\0’

Ejemplo de uso real

A continuación haremos un análisis de una ejecución real, observando el mensaje enviado por el siguiente comando:

```
dbus-send --type=method_call --dest=taller.hellodbus /taller/greeter
taller.DbusGreeter.printHello string:"Hola!"
```

Para inspeccionar el protocolo utilizamos strace con los siguientes argumentos

```
strace -xs 1024 -e network dbus-send [argumentos...]
```

Con la ejecución anterior podemos observar todas las llamadas a funciones de red del sistema, en particular los send y recv utilizados para comunicarse por medio de sockets. La opción "-xs 1024" es para ver en hexadecimal (-x) y que las lineas que imprime por salida estándar tengan un largo máximo de 1024 bytes

Se puede reconocer en el último send los siguientes paquetes

Header:

```
6c 01 00 01 0a 00 00 00    02 00 00 00 77 00 00 00
01 01 6f 00 0f 00 00 00    2f 74 61 6c 6c 65 72 2f
67 72 65 65 74 65 72 00    06 01 73 00 10 00 00 00
74 61 6c 6c 65 72 2e 68    65 6c 6c 6f 64 62 75 73
00 00 00 00 00 00 00 00    02 01 73 00 12 00 00 00
74 61 6c 6c 65 72 2e 44    62 75 73 47 72 65 65 74
65 72 00 00 00 00 00 00    03 01 73 00 0a 00 00 00
70 72 69 6e 74 48 65 6c    6c 6f 00 00 00 00 00 00
08 01 67 00 01 73 00 00
```

Body:

```
05 00 00 00 48 6f 6c 61 21 00
```

Hacemos un análisis byte por byte del mensaje

Observamos los primeros 4 bytes

'`6c`'-> 'l' little endian '`01`' -> llamada a método '`00`' -> sin flags '`01`' -> protocolo v1

Un entero con el largo del body

'`0a 00 00 00`' -> body 10

Un entero con el id del mensaje. Utilizaremos un id incremental por cada mensaje enviado. El primer id enviado será 1, el segundo 2, etc.

'`02 00 00 00`' -> id (2)

Un entero con el largo en bytes del array de opciones

'77 00 00 00' -> 119 Longitud del array

Ahora vemos cómo se arma el array de opciones. Leemos los primeros 8 bytes:

En los primeros 4 bytes tenemos una descripción del parámetro: El primer byte indica el tipo de parámetro, en este caso una "o" indica que es una ruta a un objeto. Luego un 1, posiblemente indicando la longitud del tipo de dato (no encontré documentación al respecto), y luego el tipo de dato, en este caso un string que es una "o". Los siguientes 4 bytes es la longitud del parámetro en bytes.

08 bytes: '01' '01' '6f 00' '0f 00 00 00' -> parámetro tipo 1 ("ruta del objeto"): 1 string tipo 0x6f ("o", string nombre de objeto) de longitud 15

Como el parámetro es de longitud 15, debo leer 16 bytes, ya que debo incluir el '\0' final y además debe ser múltiplo de 8.

16 bytes: '2f 74 61 6c 6c 65 72 2f 67 72 65 65 74 65 72 00' -> "/taller/greeter" + '\0' final + padding de ceros (sin padding)

Volvemos a leer 8 bytes, esta vez el parámetro es de tipo 6, y el tipo de dato es "s", string. La longitud es 16

08 bytes: '06' '01' '73 00' '10 00 00 00' -> parámetro de tipo 6 ("destino"): 1 string de tipo 0x73 ("s", string utf-8), de longitud 0x10 (16 en decimal)

Esta vez, como son 16 caracteres, al agregarle el '\0' final me queda 17, que no es múltiplo de 8. Redondeamos al múltiplo de 8 superior más cercano, 24.

24 bytes: '74 61 6c 6c 65 72 2e 68 65 6c 6c 6f 64 62 75 73 00 00 00 00 00 00 00 00' -> "taller.hellodbus" + '\0' final + padding de ceros (7 ceros de padding)

Repetimos para los otros parámetros enviados:

Interfaz, similar a los paquetes clases de java: enviamos "taller.DbusGreeter"

08 bytes: '02' '01' '73 00' '12 00 00 00' -> parámetro de tipo 2 ("interfaz"): 1 string de tipo 0x73 ("s", string utf-8), de longitud 0x12 (18 en decimal)
24 bytes: '74 61 6c 6c 65 72 2e 44 62 75 73 47 72 65 65 74 65 72 00 00 00 00 00 00' -> "taller.DbusGreeter" + '\0' final + padding de ceros (5 ceros de padding)

Método: el nombre de la función que vamos a invocar.

08 bytes: '03' '01' '73 00' '0a 00 00 00' -> parámetro de tipo 3 ("método"): 1 string de tipo 0x73 ("s", string utf-8), de longitud 0x0a (10 en decimal)
16 bytes: '70 72 69 6e 74 48 65 6c 6c 6f 00 00 00 00 00 00' -> "printHello" + '\0' final + padding de ceros (5 ceros de padding)

Firma del método, los parámetros que vamos a enviar. Sólo enviaremos strings, así que la firma constará de una "s" por cada parámetro.

08 bytes: '08' '01' '67 00' '01 73 00 00' -> parámetro de tipo 8 ("firma"): 1 string de tipo "g" (string firma de método). La firma es de longitud 1, y es de tipo 0x73 ("s", un string), 1 byte de padding

Formato de los archivos de entrada

Cliente

El archivo de entrada es un archivo de texto con el siguiente formato

```
<destino> <path> <interfaz> <metodo>([<parametro1>,<parametroN>...])
```

Ejemplo:

```
taller.server /tp1/server com.taller.tp1 saludar(juanin,juan,harry)
```

El destino es "taller.server", la ruta (path) es "/tp1/server", la interfaz es "com.taller.tp1", el método es "saludar", y los parámetros son "juanin", "juan" y "harry"

Todas las llamadas terminan con el caracter de fin de linea, se puede asumir que las lineas están bien formadas. Notar que interfaz y método están separadas.

Servidor

El servidor no posee archivos de entrada

Formato de los archivos de salida

Cliente

El cliente debe imprimir por cada procedimiento llamado una linea con el id seguido de la respuesta del servidor:

```
<id en hexadecimal, 4 dígitos, con prefijo "0x">: <respuesta del servidor>
```

Como la respuesta del servidor es siempre "OK\n" (3 bytes), la salida del cliente será del estilo:

```
0x0001: OK
0x0002: OK
```

El "OK" **DEBE** provenir del servidor, no se acepta que sea el cliente el que genera esta respuesta.

Servidor

El servidor deberá imprimir por cada llamada a procedimiento remoto las siguientes lineas:

```
* Id: <id en hexadecimal, 4 dígitos, con prefijo "0x">
* Destino: <destino>
* Path: <path>
* Interfaz: <interfaz>
* Método: <método>
```

Si el método posee parámetros, se agregan las siguientes lineas

```
* Parámetros:
    * <parámetro1>
    * <parámetroN>
```

Por último, una linea en blanco después de los parámetros o el método si estos no están presentes.

Aclaración: los parámetros son antecedidos por 4 espacios.

Utilizando el ejemplo anterior, si leemos la linea

```
taller.server /tp1/server com.taller.tp1 saludar(juanin,juan,harry)
```

Imprimimos

```
* Id: 0x0001
* Destino: taller.server
* Path: /tp1/server
* Interfaz: com.taller.tp1
* Método: saludar
* Parámetros:
    * juanin
    * juan
    * harry
```

Sugerencias y Recomendaciones

- En caso de notar algún defecto en el enunciado o en las pruebas, por favor avisar lo antes posible para que sea corregido de la manera más inmediata.
- Encarar el trabajo de manera modular, en el orden que resulte más cómodo, pero no arrancar programando funciones demasiado extensas después recortarlas, ya que esa manera de encarar los problemas conduce a mucho retrabajo y bugs.
- Hacer un paneo de las preguntas de otros cuatrimestres antes de empezar a trabajar, ya que hay respuestas muy útiles para facilitar el trabajo.
- **No programar demás**, hacer módulos con responsabilidades claras y cohesivas, y evitar el acoplamiento. Definir las interfaces antes de las implementaciones.

- Las páginas de manual de unix son especialmente útiles para programar con sockets. (Leer **man send** y **man getaddrinfo**, por ejemplo).
- Hacer uso de **netcat** en modo escucha para verificar los datos enviados por el cliente. También se lo puede usar para simular la conexión de un cliente, redirigiendo a su entrada un archivo binario.
- Hacer uso de **strace**, para observar las llamadas a *syscalls* . Puede ser muy útil para saber si una aplicación se queda bloqueada en, por ejemplo, una lectura.

Penalizaciones

La siguiente es una lista de restricciones técnicas exigidas no obligatorias para la aprobación:

1. Las funciones de más de 20 líneas requieren una justificación clara de su extensión.
2. La lectura del archivo de entrada debe ser en bloques de 32 bytes (es decir, utilizar un buffer de lectura). No se puede trabajar con todo el contenido del archivo en memoria.

Restricciones

La siguiente es una lista de restricciones técnicas exigidas de caracter obligatorio:

1. El sistema debe desarrollarse en ISO C (C99). No se puede usar macros que alteren el standard del código excepto en el “.c” de sockets.
2. Está prohibido el uso de variables y funciones globales. La función **main** no puede contener lógica del negocio, solamente debe servir como punto de entrada del flujo del programa.
3. El informe debe contener al menos un diagrama que represente alguna parte del diseño. No hace falta que sea UML, pero sí que sea descriptivo.
4. El protocolo de comunicación es obligatorio, no sugerido.

Bibliografía

- Especificación de D-BUS: https://dbus.freedesktop.org/doc/dbus-specification.html