Tema 3: Representación de la información

XDR

Jose Luis Díaz

Curso 2011-2012

Contenidos

Esquema

El problema

- En un sistema distribuido cada máquina puede tener diferente arquitectura.
- Cada arquitectura puede tener una forma diferente de codificar la información.
- Al transferir esta información entre máquinas, es necesario hacer una conversión para que no se pierda *significado*

Ejemplo

```
//=== Programa que envia =====
short int i=10;
send(sock, &i, sizeof(i), 0);

//=== Programa que recibe ====
short int n;
recv(sock_dat, &n, sizeof(n), 0);
printf("Valor recibido= %d\n", n);
```

Si la arquitectura origen es little-endian y la destino es big-endian ¿Qué valor se imprimiría?

Solución

La solución más sencilla consiste en:

- *Inventar* un formato nuevo para codificar la información.
- Programar funciones que conviertan del formato nativo de cada máquina a ese formato nuevo, y viceversa.
- Cuando dos procesos quieran comunicarse:
  - El emisor convierte la información al nuevo formato y envía los bytes resultantes.
  - El receptor recibe la secuencia de bytes y debe convertirla al formato que use su máquina, antes de poder usar la información.

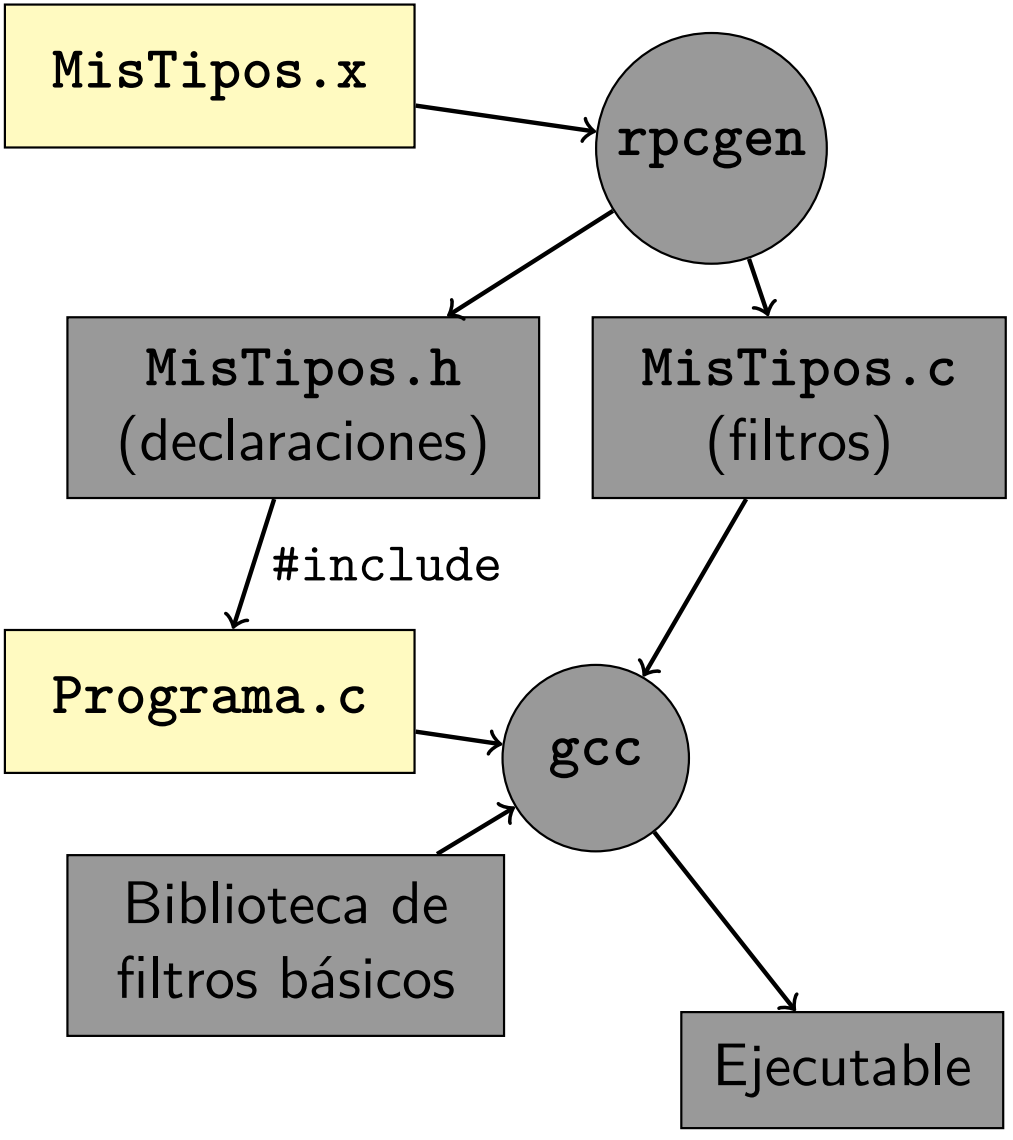
Este trabajo ya lo han hecho otros por nosotros. Veremos un caso: XDR.

Esquema

XDR: eXternal Data Representation

- Fue creado por *Sun Microsystems*, como parte de su mecanismo "RPC" (llamadas a procedimientos remotos)
- Proporciona un buen número de *tipos básicos*.
  - Para los tipos básicos, proporciona una biblioteca de funciones de conversión (filtros)
- Permite al programador definir sus propios tipos
  - Para definir nuevos tipos, se usa el lenguaje XDR.
  - Una vez definidos, una herramienta genera automáticamente el código de los filtros necesarios.

Esquema general de uso



1. Escribir nuevos tipos de datos (lenguaje XDR)
2. Utilizar herramienta **rpcgen**, para generar
   - Los tipos en C (.h)
   - Los filtros de conversión a XDR
3. Escribir el programa que usa estos tipos y filtros
4. Enlazar con la biblioteca de filtros básicos
5. Obteniendo el ejecutable final

Información general

- Sobre los filtros de conversión:
  - Son funciones C
  - Todas tienen la misma sintaxis.
  - Su nombre es siempre `xdr_tipo()`.
  - El mismo filtro sirve para codificar y decodificar, según el valor de uno de sus parámetros.
- Sobre la codificación XDR:
  - Una vez codificado en XDR el dato siempre ocupa un número de bytes múltiplo de 4.
  - La ordenación de los bytes será *Big-endian*.
  - La codificación incluye sólo el valor del dato, pero no su tipo.

Sintaxis genérica de un filtro

Todos los filtros tienen la misma sintaxis, cambiando *tipo* por el tipo XDR deseado. Se verán más adelante los posibles tipos XDR .

```
bool_t xdr_tipo(XDR *operacion, tipo *dato);
```

operacion Un puntero a una estructura especial que define:

- El almacén para la codificación xdr
- El sentido del filtro

dato Puntero al dato en codificación de máquina.

Retorna **TRUE** si no hubo problemas, **FALSE** si la codificación no pudo hacerse.

La estructura XDR

- Esta estructura define el sentido de la conversión y el destino u origen de los datos codificados en xdr.
- Se inicializa mediante la función **`xdrstdio_create()`**.
- Se destruye con la función **`xdr_destroy()`**

```
bool_t xdrstdio_create(XDR *operacion, FILE *fichero,
                       enum xdr_op sentido)
```

operacion Puntero a la estructura que queremos inicializar

fichero Almacén de la codificación XDR.

sentido Sentido de la conversión:

- XDR_ENCODE máquina→xdr
- XDR_DECODE xdr→maquina

Retorna TRUE si éxito, FALSE si error.

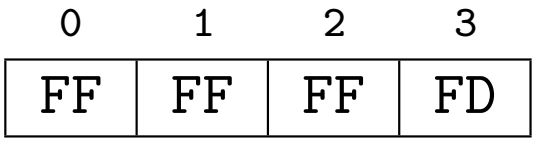
Ejemplo

```
#include <rpc/rpc.h> // Para las funciones de filtros básicos
  int dato;          // Dato a escribir y leer después
  FILE *fichero;     // Fichero donde se escribirá / leerá
  XDR operacion;     // Necesario para los filtros

  // Escribir un entero en el fichero
  fichero=fopen("Entero.dat", "w");
  if (fichero==NULL) { perror("Al abrir fichero"); exit(-1); }
  xdrstdio_create(&operacion, fichero, XDR_ENCODE);
  dato=35;
  xdr_int(&operacion, &dato); // Llamada al filtro. Codifica y guarda
  xdr_destroy(&operacion);
  fclose(fichero);
  // Leerlo seguidamente
  dato=0;
  fichero=fopen("Entero.dat", "r");
  if (fichero==NULL) { perror("Al abrir fichero para leer"); exit(-1); }
  xdrstdio_create(&operacion, fichero, XDR_DECODE);
  xdr_int(&operacion, &dato);   // Llamada al filtro. Lee y decodifica
  printf("Dato leido= %d\n", dato);
  xdr_destroy(&operacion);
  fclose(fichero);
```

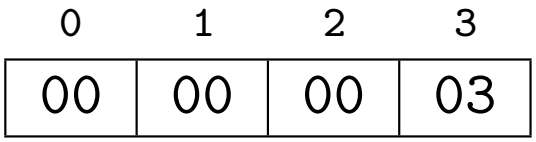
Enteros

Tipo xdr	`int`
Tipo C	`int`
Filtro	`xdr_int()`
Codificación	32 bits, complemento a 2
Ejemplo	El valor -3 se codificará en 4 bytes:



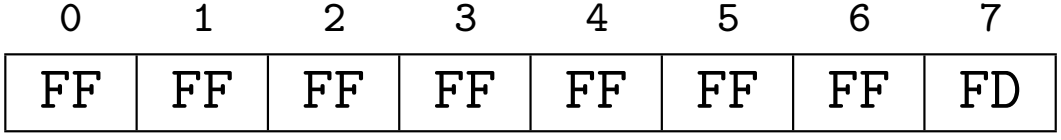
Enteros positivos

Tipo xdr	**`unsigned int`**
Tipo C	**`unsigned int`**
Filtro	**`xdr_u_int()`**
Codificación	32 bits, binario natural
Ejemplo	El valor 3 se codificará en 4 bytes:



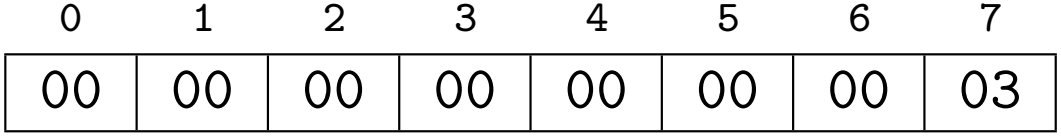
Enteros grandes

Tipo xdr	**`hyper`**
Tipo C	**`int64_t`** (un tipo definido en **`types.h`**, equivalente al **`long long int`**)
Filtro	**`xdr_hyper()`**
Codificación	64 bits, complemento a 2
Ejemplo	El valor -3 se codificará en 8 bytes:



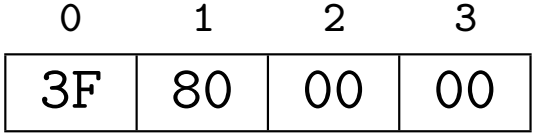
Enteros grandes positivos

Tipo xdr	**`unsigned hyper`**
Tipo C	**`uint64_t`** (un tipo definido en **`types.h`**, equivalente al unsigned **`long long int`**)
Filtro	**`xdr_u_hyper()`**
Codificación	64 bits, binario natural
Ejemplo	El valor 3 se codificará en 8 bytes:



Reales (precisión simple)

Tipo xdr	`float`
Tipo C	`float`
Filtro	`xdr_float()`
Codificación	32 bits. Coma flotante. Norma IEEE-754, precisión simple.
Ejemplo	El valor 1,0 se codificará en 4 bytes:



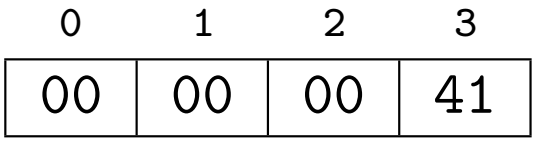
Reales (precisión doble)

Tipo xdr	**`double`**
Tipo C	**`double`**
Filtro	**`xdr_double()`**
Codificación	64 bits. Coma flotante. Norma IEEE-754, doble precisión.
Ejemplo	El valor $-1{,}0$ se codificará en 8 bytes:

0	1	2	3	4	5	6	7
BF	F0	00	00	00	00	00	00

Carácter

Tipo xdr	**`char`**
Tipo C	**`char`**
Filtro	**`xdr_char()`**
Codificación	32 bits, los 7 bits inferiores contienen el código ASCII, los restantes valen 0
Ejemplo	El carácter **`'A'`** se codificará en 4 bytes:



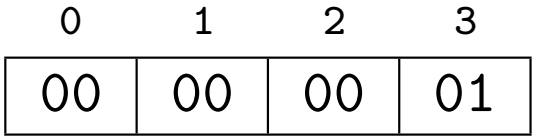
`(ASCII de 'A'=41h)`

No suele usarse

Observar el desperdicio de bytes. Para almacenar cadenas de caracteres resulta más conveniente el tipo **`string`**

Booleano

Tipo xdr	**`bool`**
Tipo C	**`bool_t`** (un tipo definido en **`rpc/types.h`**, equivalente al **`int`**)
Filtro	**`xdr_bool()`**
Codificación	32 bits. El valor **`TRUE`** se representa como el entero `1` y el valor **`FALSE`** como el entero `0`
Ejemplo	El valor **`TRUE`** se codificará como



Definición de tipos propios

Un tipo definido por el programador puede ser (de menor a mayor complejidad):

- Simplemente otro nombre para uno de los tipos básicos.
- Una constante predefinida (definición) o un conjunto de ellas (enumeración)
- Una secuencia de elementos del mismo tipo (un array)
- Una cadena de caracteres (*string*)
- Una agrupación de elementos de diferentes tipos (una estructura)
- Un elemento cuyo tipo puede variar (una unión discriminada)

Dar otro nombre a un tipo existente

typedef *declaración*

declaración tiene la sintaxis de una declaración de variable en lenguaje C, pero donde iría el nombre de la variable se pone el nombre del nuevo tipo.

Ejemplo.x

```
typedef float NumeroReal;
```

Al procesar con **rpcgen** el fichero **Ejemplo.x**, generará:

Ejemplo.h con un nuevo tipo C, que podremos usar en nuestros programas.

Ejemplo_xdr.h con el código de un filtro llamado **xdr_NumeroReal()** que podremos usar para codificar y decodificar datos de este tipo.

Definiendo constantes

```
const nombre = valor
```

- La constante así definida puede usarse en otras partes del fichero xdr
- También puede usarse desde C, puesto que **rpcgen** convertirá la constante en un **\#define**

Ejemplo

```
const LONGITUD_LINEA = 80;
```

Enumeraciones

Una enumeración es un conjunto de constantes de tipo entero que tienen habitualmente valores correlativos.

```
enum nombre_del_conjunto {
   nombre_1=valor_1, ..., nombre_n=valor_n
}
```

Ejemplo.x

```
enum Palo { OROS=1, COPAS=2, ESPADAS=3, BASTOS=4 };
```

- **rpcgen** generará un tipo enumerado en C, y un filtro llamado **xdr_Palo()**
- La codificación XDR de un dato enumerado es idéntica a la de un **int**.

Enumeraciones, ejemplo

Ejemplo de uso en C

```
#include "Ejemplo.h"

  Palo pintan;  // El tipo Palo está definido en Ejemplo.h (generado)
  XDR  operacion;

  pintan=COPAS;
  // Supongamos que operacion se inicializa para escribir en el
  // fichero (XDR_ENCODE). ¿Qué almacenará la llamada siguiente?
  xdr_Palo(&operacion, &pintan);

  // Supongamos ahora que operacion se inicializa para leer del
  // fichero (XDR_DECODE)
  xdr_Palo(&operacion, &pintan);
  switch(pintan) {
   case OROS:  printf("Pintan oros\n"); break;
   case COPAS: printf("Pintan copas\n"); break;
   // etc.
  }
```

Arrays

- Un array es una secuencia de datos del mismo tipo
- XDR admite dos tipos de array
  - **Arrays de longitud fija**. El tamaño del array está prefijado, y se especifica entre corchetes `[]` en su declaración.
  - **Arrays de longitud variable**. El tamaño del array sólo se conoce en tiempo de ejecución. Se declaran poniendo ángulos `<>` en su declaración.
- Los arrays de longitud fija son iguales a los del lenguaje C
- Los arrays de longitud variable se implementan en C mediante una estructura con dos campos:
  - Un entero indicando el número de elementos
  - Un puntero al primer elemento

Arrays de longitud fija

Una declaración de array fijo en XDR tiene la siguiente sintaxis:

```
tipo_de_cada_elemento nombre_array[TAM];
```

Ejemplo.x

```
typedef int TresEnteros[3];
```

El tipo equivalente en C es un array normal, de la dimensión especificada. Se codificará como una secuencia de elementos. Ejemplo: Dada la siguiente codificación, generada por el filtro **`xdr_TresEnteros()`** ¿qué datos contenía el array?

0	1	2	3	4	5	6	7	8	9	10	11
00	00	00	02	00	00	01	02	00	00	02	01

Array de longitud variable

Una declaración de array variable en XDR tiene la siguiente sintaxis:

```
tipo_de_cada_elemento nombre_array<>;
```

Ejemplo.x

```
typedef int VariosEnteros<>;
```

La herramienta **rpcgen** genera un tipo C que es una estructura:

Tipo equivalente en C (en **Ejemplo.h**)

```
typedef struct {
  u_int VariosEnteros_len;
  int *VariosEnteros_val;
} VariosEnteros;
```

- El campo que indica cuántos elementos hay siempre se llama igual que el array declarado en XDR, más el sufijo **`_len`**
- El campo que apunta al primer elemento siempre se llama igual que el array declarado en XDR, más el sufijo **`_val`**

Ejemplo de uso del nuevo tipo

```
VariosEnteros datos;
int cuantos;

printf("Numero de datos:"); scanf("%d", &cuantos);
// Reservar memoria para el numero de datos solicitado
datos.VariosEnteros_val = malloc(cuantos*sizeof(int));
if (datos.VariosEnteros_val==NULL) {
  perror("Al reservar espacio en memoria"); exit(-1); }
// Inicializar los elementos de la estructura:
datos.VariosEnteros_len=cuantos; // Numero de ellos
for (i=0; i<cuantos; i++) {
  printf("Dato[%d]: ", i);
  scanf("%d", datos.VariosEnteros_val[i]);
}
```

Codificación del array variable

El filtro XDR codificará el array mediante una secuencia:

- Un entero sin signo que indica cuántos elementos tiene el array (4 bytes)
- Cada uno de los elementos del array, comenzando por el elemento 0, en secuencia

Ejemplo: Dada la siguiente codificación de un array de enteros ¿Cuántos elementos tiene y cuáles son?

0	1	2	3	4	5	6	7	8	9	10	11
00	00	00	02	00	00	01	02	00	00	02	01

Casos especiales de arrays

Hay dos casos especiales de tipos xdr, que son aparentemente arrays, pero se codifican diferente:

- **Arrays de datos opacos** Cada dato del array es un byte, que XDR no intenta codificar. Pueden ser de longitud fija o variable.
- **Cadenas de caracteres** Cada elemento del array es un carácter, codificado mediante su código ASCII. Siempre son de longitud variable.

El número de elementos del array puede no ser múltiplo de 4, pero las codificaciones deben serlo siempre.

- El filtro XDR añadirá ceros de relleno para lograrlo.
- Estos ceros no forman parte del array, solo de su codificación.

Arrays de opacos

```
opaque nombre_array[TAM];
opaque nombre_array<>;
```

Ejemplo.x

```
typedef opaque Datos[3];
typedef opaque OtrosDatos<>;
```

El tipo equivalente en C es un array de **char**, para el primer caso, o una estructura como se vio en los arrays de longitud variable para el segundo caso. Ejemplo: un array variable con tres datos opacos de valores 01, 02 y 03 ¿cómo se codificará?

Cadenas de caracteres

```
string nombre_array<>;
```

Ejemplo.x

```
typedef string frase<>;
```

Aunque la sintaxis es de array variable, no se convierten en una estructura con un campo **_len** y otro **_val**, como aquellos, sino simplemente en un puntero a char.

Ejemplo de uso desde el C

```
frase miFrase; // es equivalente a char *miFrase
miFrase="abc";
```

¿Cómo sería la codificación xdr de la variable **miFrase**? (El código ASCII de 'a' es **61h**)

Un string no es un array de caracteres

Aunque desde el C las cadenas se implementan como arrays de caracteres, desde el punto de vista de XDR no son equivalentes. Considerar el siguiente ejemplo (.x):

Cadenas y arrays

```
typedef string cadena<>;
typedef char   array_variable<>;
typedef char   array_fijo[5];
```

Piensa qué implementación en C tendría cada uno de estos tipos, y en cómo se codificaría una cadena que contenga “abc” para cada uno de los casos.

Estructuras

Una estructura es una secuencia de datos *(campos)* de diferentes tipos. Cada tipo en la secuencia puede ser básico o definido por el programador.

```
struct nombre_estructura {
  declaración_campo_1;
  ...
  declaración_campo_n;
}
```

Ejemplo.x

```
struct Persona {
  int edad;
  string nombre<>;
  string apellidos<>;
};
```

Observar que no requiere **typedef**. Automáticamente crea un nuevo tipo llamado **Persona**.

Estructuras (versión en C)

Una estructura XDR se convierte en C en otra estructura. Cada campo se convierte al tipo apropiado en C.

Ejemplo de uso de la estructura Persona

```
#include "Ejemplo.x"
  Persona alguien;
  XDR orden;

  alguien.edad=49;
  alguien.nombre="Bill";
  alguien.apellidos="Gates";
  // ... Inicializacion de fichero y orden XDR
  xdr_Persona(&orden, &alguien);
```

Estructuras (codificación)

La codificación XDR de una estructura no es más que la codificación en secuencia de cada uno de los campos

¿Cuántos bytes ocupará la codificación XDR de la variable **`alguien`** del ejemplo anterior? ¿Qué valores tendrán esos bytes?

Uniones discriminadas

Una unión discriminada es un dato cuyo tipo no está prefijado, sino que puede variar entre varios posibles tipos, según cuál sea el valor de un *discriminante*.

```
union nombre_union switch (declaración_discriminante) {
  case valor_1: declaración_tipo_1;
  ...
  case valor_N: declaración_tipo_N;
  default: declaración_tipo_defecto; /* Opcional */
}
```

Ejemplo.x

```
union Respuesta switch (int tipo)  {
  case 1:  int    entero;
  case 2:  double real;
  default: string cadena<>;
};
```

Observar que tampoco requiere **typedef**. Automáticamente crea un nuevo tipo llamado **Respuesta**, que puede comportarse como un entero, un real o una cadena, según el valor del discriminante.

Uniones discriminadas (versión en C)

En el código C, este tipo de datos se convierte en una estructura con dos campos:

- Un campo es el discriminante. Su nombre y tipo son los declarados en el fichero **`.x`**
- Otro campo es una unión C. El nombre de este campo es igual al nombre declarado en el fichero **`.x`**, más el sufijo **`_u`**.
- Los campos de la unión son los declarados en el **`.x`**

Ejemplo del tipo generado por la unión discriminada anterior

```
struct Respuesta {
        int tipo;
        union {
                int entero;
                double real;
                char *cadena;
        } Respuesta_u;
};
```

Uniones discriminadas (ejemplo en C)

Ejemplo de uso

```
#include "Ejemplo.h"
   Respuesta a, b, c;
   a.tipo=1;                    // Elijo el tipo entero para a
   a.Respuesta_u.entero=14;          // y le asigno un valor
   b.tipo=2;                     // Elijo el tipo real para b
   b.Respuesta_u.real=3.1415926;  // y le asigno un valor
   c.tipo=0;              // Elijo el tipo por defecto para c
   c.Respuesta_u.cadena="mar";      // y le asigno un valor

   // Una variable tipo Respuesta puede cambiar de tipo
   a.tipo=0;               // Ahora "a" tiene el tipo por defecto
   a.Respuesta_u.cadena="un texto";    // debo asignar de nuevo

   // Para imprimir una variable de tipo Respuesta, debo mirar
   // antes qué tipo tiene.
   switch(a.tipo) {
        case 1: printf("%d\n", a.Respuesta_u.entero); break;
        case 2: printf("%f\n", a.Respuesta_u.real); break;
      default: printf("%s\n", a.Respuesta_u.cadena);
   }
```

Uniones discriminadas (codificación)

Se codifica en primer lugar el valor del discriminante, y seguidamente el valor del dato elegido por el discriminante.

Por ejemplo, ¿cuál sería la codificación xdr de la variable "`a`" al principio del programa anterior? ¿Y la variable "`c`"?

Punteros, listas enlazadas, . . .

El programador de C a menudo usa punteros para crear estructuras de datos que pueden crecer dinámicamente. Por ejemplo, una lista enlazada. Una lista enlazada se implementa mediante una

estructura que contiene:

- Uno o más campos para los datos asociados con un elemento de la lista
- Un puntero al siguiente elemento de la lista (con el valor `NULL` si es el último elemento)

El programa suele tener un puntero al primer elemento de la lista, y utlizar algoritmos para recorrer esta lista elemento por elemento, o para añadirle elementos al principio o al final

¿Cómo implementar un dato así en XDR?

Tipos opcionales XDR

XDR permite definir un campo de una estructura como "opcional". La sintaxis consiste en poner un asterisco delante del nombre del campo. Esto es:

```
tipo *nombre_campo;
```

La sintaxis recuerda la declaración de un puntero en C, pero XDR no tiene punteros. El asterisco indica que el valor de ese campo se codificará o no, opcionalmente.

El tipo equivalente en C tendrá un puntero en ese campo.

Ejemplo.x

```
struct Elemento {
  string Nombre<>;
  int   *EnteroOpcional;
};
```

Ejemplo.h

```
struct Elemento {
  char *Nombre;
  int  *EnteroOpcional;
};
```

Datos opcionales (codificación)

El campo opcional, al ser en C un puntero, puede ser:

NULL En este caso se codificará como el entero 00000000

No NULL En este caso se codificacrá como el entero 00000001 seguido de la codificación del dato a que apunta.

Ejemplo (en C)

```
Elemento dato1, dato2;
dato1.Nombre=.Ej1"; dato1.EnteroOpcional=NULL;
dato2.Nombre=.Ej2";
dato2.EnteroOpcional=malloc(sizeof(int)); // Reservar espacio
*dato2.EnteroOpcional=15; // Asignarle el dato
```

¿Cómo se codificarán **dato1** y **dato2**?

Uso de datos opcionales para implementar listas

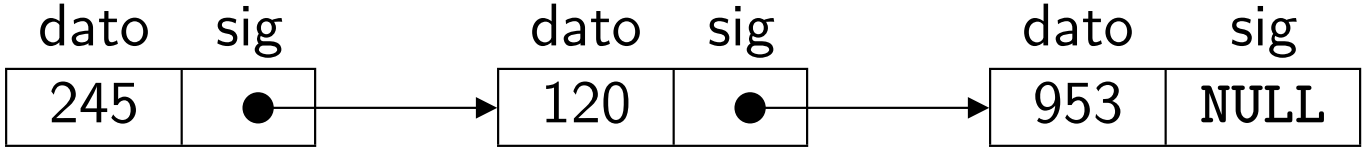
Una lista se implementa como una estructura con dos partes

- Los datos asociados al elemento
- Otro elemento opcional

Ejemplo: lista.x

```
struct Elemento {
  int     dato;
  Elemento *sig; // El dato opcional es de tipo Elemento
};
```

En cada elemento de la lista, el campo **`sig`** apunta a otro elemento, excepto en el último elemento de la lista, en el que será



Implementación de la lista en C

```
#include "lista.h"
  Elemento *primero, *aux;
  XDR operacion; FILE *f;

  // Crear un elemento y asignarle un dato
  primero=malloc(sizeof(Elemento));
  primero->dato=953;     primero->sig=NULL;
  // Insertar otro elemento en la cabeza
  aux=malloc(sizeof(Elemento));
  aux->dato=120;    aux->sig=primero;   primero=aux;
  // Insertar otro elemento en la cabeza
  aux=malloc(sizeof(Elemento));
  aux->dato=245;    aux->sig=primero;   primero=aux;
  // La lista tiene ahora tres elementos.
  // Para codificar la lista completa en formato XDR basta con
  // llamar una vez al filtro, para el primer elemento
  f=fopen("Datos.dat", "w");
  xdrstdio_create(&operacion, f, XDR_ENCODE);
  xdr_Elemento(&operacion, primero);
```

Codificación XDR de la lista

Para descubrir cómo codificará la lista el filtro **`xdr_Elemento()`**, basta seguir las reglas de codificación ya expuestas.

- El tipo **`Elemento`** es una estructura, luego se codificarán sus campos en secuencia:
  - El primer campo es un entero. Se codificará con 4 bytes.
  - El siguiente campo es un opcional. Ya que es distinto de **`NULL`** se codificará como `00000001` seguido del dato al que apunta.
  - El dato a que apunta es, de nuevo, de tipo **`Elemento`**, luego se aplica de nuevo esta misma regla, recursivamente.

Por tanto ¿qué codificación se generará finalmente?

Decodificación de datos con punteros

Hemos visto que algunos tipos XDR generan en C tipos de datos que involucran un puntero:

- Los `strings` se convierten en puntero a carácter.
- Los arrays de longitud variable se convierten en una estructura, con un campo que es un puntero
- Los campos opcionales se convierten en punteros

Reserva de memoria

¿Qué ocurre cuando queremos leer (o recibir) un dato XDR de uno de estos tipos? ¿Debemos reservar previamente memoria en el programa en C?

Decodificación de datos con punteros

Durante la decodificación, el filtro XDR pueden encontrar que un puntero:

- Es distinto de `NULL`. En ese caso guarda el dato decodificado en la dirección a que apunta el puntero
- Es `NULL`. En ese caso reserva memoria, asigna el puntero a la nueva dirección, y guarda en ella el dato decodificado

Por tanto...

Lo más sencillo para el programador es asegurarse de que los punteros valen `NULL` (o cero, que es lo mismo), antes de llamar a un filtro, y que éste se ocupe de reservar memoria.

Ejemplo

Decodificación de una lista

```
#include "lista.h"
  Elemento *primero, *aux;
  XDR operacion; FILE *f;  int i=0;

  // Solo es necesario reservar memoria para el primer elemento
  // para los demás los reservará el filtro a medida que los recibe
  primero=malloc(sizeof(Elemento));
  // Debemos inicializar con ceros toda la estructura, para asegurarnos
  // de que los punteros que contenga sean NULL
  memset(primero, 0, sizeof(Elemento));
  // Para decodificar la lista completa, basta llamar una vez al filtro
  // el cual irá reservando memoria para cada sucesivo elemento
  f=fopen("Datos.dat", "r");
  xdrstdio_create(&operacion, f, XDR_DECODE);
  xdr_Elemento(&operacion, primero);
  // Mostraremos ahora los elementos leídos
  aux=primero;
  while (aux!=NULL) {
    printf("Elemento %d = %d\n", i++, aux->dato);
    aux=aux->sig;  }
```

Liberar la memoria

Cuando un dato ya no es necesario, hay que liberar la memoria que ocupaba.

- Si la memoria la reservó el programador con **`malloc()`** se liberará llamando a **`free()`**
- Si la memoria la reservó el propio filtro al recibir debe liberarse con **`xdr_free()`**

```
void xdr_free(xdrproc_t xdr_filtro, char *dato);
```

xdr_filtro	Es el nombre del filtro que se usó para decodificar el dato
dato	Es un puntero al dato
NOTA	No se libera la memoria a que apunta **`dato`**, sino la memoria a que apuntan los punteros que **`dato`** pudiera contener. Para liberar la memoria apuntada por **`dato`** se usaría **`free()`**

Liberar la memoria

Ejemplo de cómo liberar la lista

```
#include "lista.h"
  Elemento *primero, *aux;
  // ...

  // Para liberar la lista, primero liberamos todos los elementos
  // que creó el filtro
  xdr_free((xdrproc_t) xdr_Elemento, (char *)primero);

  // Finalmente liberamos los elementos que creó el programa
  // con malloc, (es decir, el primero)
  free(primero);
```

Combinar XDR con sockets

- Hasta ahora hemos visto que XDR puede codificar y decodificar datos, pero los datos XDR siempre iban o procedían de un fichero.
- El sistema operativo trata sockets y ficheros de la misma forma.
- Por tanto, ¿podríamos especificar un socket en lugar de un fichero a la hora de inicializar xdr?

La respuesta es que sí, pero hay una dificultad.

- Cuando creamos el socket, obtenemos un descriptor de tipo **`int`**
- Sin embargo la función **`xdrstdio_create()`** espera un descriptor de tipo **`FILE`**

¿Por qué esta discrepancia?

Descriptores de fichero

Desde C se pueden usar dos tipos diferentes para acceder a los ficheros:

- Para el sistema operativo un descriptor es un entero (`int`). Existen funciones C para manejar este tipo de descriptores.
- El lenguaje C, además, proporciona el descriptor de tipo `FILE*` como una forma más cómoda de acceder a los ficheros, con otro conjunto de funciones.

Existen funciones para convertir unos en otros:

fileno() Recibe un `FILE*` obtenido con `fopen()` y devuelve un entero que se refiere al mismo fichero.

fdopen() Recibe un `int` obtenido con `open()`, o si es un socket con `socket()` o `accept()`, junto con un modo de apertura (`"r"` o `"w"`) y retorna un `FILE*` que se refiere al mismo fichero o socket.

Ejemplo: Envío de un dato por un socket usando XDR

```
#include <rpc/rpc.h>
#define PUERTO 33333
  int  sock;     // El socket de comunicación
  FILE *fsock;   // El mismo socket, visto "como FILE"
  double dato=3.1415926; // El dato a enviar
  XDR  operacion;

  // Creacion del socket (debe ser TCP)
  sock=socket(PF_INET, SOCK_STREAM, 0);
  // Conexión con el otro extremo (no se muestra)
  // Una vez conectado con exito, covierto el socket en FILE
  fsock=fdopen(sock, "w");  // Para escribir (enviar)
  // Inicializar estructura XDR
  xdrstdio_create(&operacion, fsock, XDR_ENCODE); // encode
  // Enviar el dato consiste simplemente en aplicarle el filtro
  xdr_double(&operacion, &dato);
  // Para asegurar el envío, vaciar el buffer del FILE asociado
  fflush(fsock);
  // Desconectar
  fclose(fsock); close(sock);
```

Ejemplo: Recepción del dato

```
#include <rpc/rpc.h>
#define PUERTO 33333
  int  e_sock;  // El socket de escucha
  int  sock;    // El socket de datos
  FILE *fsock;  // El mismo socket (datos), visto "como FILE"
  double dato;  // El dato a recibir
  XDR  operacion;

  // Creacion del socket (debe ser TCP)
  e_sock=socket(PF_INET, SOCK_STREAM, 0);
  // Asignación de puerto y puesta en escucha (no se muestra)
  // Esperamos conexión:
  sock=accept(e_sock, NULL, NULL); // NULL cuando no interesa el cliente
  // Una vez conectado con exito, covierto el socket en FILE
  fsock=fdopen(sock, "r");   // Para leer (recibir)
  // Inicializar estructura XDR
  xdrstdio_create(&operacion, fsock, XDR_DECODE); // decode
  // Recibir el dato consiste simplemente en aplicarle el filtro
  xdr_double(&operacion, &dato);
  // Imprimir el dato recibido:
  printf("Recibido %f\n", dato);
```