Google inventa un nuevo lenguaje de programación

Preparados... Listos... ¡Go!

Google no sólo es bueno en las búsquedas, sino que también sabe hacer otras cosas. El gigante de los motores de búsqueda está dispuesto a lanzar un nuevo lenguaje de programación. ¿Tendrá éxito esta vez? **POR MARCUS NUTZINGER Y RAINER POISEL**

En un encuentro de desarrolladores de Google en 2007, los antiguos veteranos de Bell Labs, Rob Pike y Ken Thompson se preguntaban si en el arte de la programación se sufrían demasiadas esperas. Según Pike, "Todo tardaba demasiado. Demasiado tiempo para desarrollar; demasiado tiempo para compilar ..." [1].

Un problema, según Pike, era que los lenguajes de programación no han variado mucho durante los últimos años, aunque los requisitos y las expectativas no hayan dejado de evolucionar. Los programas contemporáneos han de acomodar las comunicaciones por red entre cliente-servidor, clústeres de computación masiva y procesadores multinúcleo, etcétera; y, al mismo tiempo, el desarrollador debe prestar especial atención a la seguridad y a la estabilidad. Además, los sistemas para la prueba y control de los tipos de datos no paran de complicarse.

Los desarrolladores de Google necesitaban un lenguaje tan eficiente como los lenguajes de tipo estático como C y tan fácil de usar como los lenguajes dinámicos como Python. Querían también un buen soporte para la concurrencia y un sistema de recolección de basura (como en Java o C#) para la limpieza automatizada de la memoria.

Instalación

Si hay un firewall bloqueando el acceso a Internet durante la instalación, necesitaremos encontrar una solución. Deshabilitaremos las pruebas de los subsistemas *http* y *net* en el Makefile para que el resultado de su ejecución no condicione el éxito de la instalación global [4]. Para hacerlo, añadiremos entradas *http* y *net* al valor de la variable *NOTEST* en el fichero *$GOROT/src/pkg/makefile*.

Después de varios meses de planificación y otros tantos de codificación, el equipo de Google desveló un nuevo lenguaje de programación "expresivo, concurrente y con recolector de basura" llamado *Go* [2]. Las herramientas necesarias para comenzar a usar el lenguaje de programación Go ya están disponibles desde el sitio web del proyecto.

Todo aquel que tenga ganas de probar un nuevo lenguaje de programación, puede probar suerte con este lenguaje experimental diseñado para ser usado en la programación de nueva generación.

Diseño del Entorno

El código fuente de las herramientas del lenguaje de programación Go está disponible desde un repositorio en la página de inicio del proyecto [3]. Después de compilar los distintos componentes de Go (ver el cuadro "Instalación"), lo siguiente será compilar las librerías. Este paso es ya un indicador de lo bueno que es el rendimiento del nuevo lenguaje de Google, ya que las propias librerías están escritas en Go. Nótese que una librería completa apenas tarda en compilarse un par de segundos.

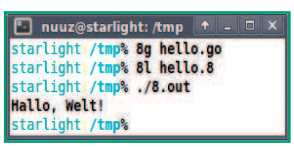

Figura 1: Los comandos para compilar y enlazar un programa en Go son parecidos a la, en un primer momento, extraña notación de Plan 9.

Actualmente hay dos compiladores disponibles, *gc* y *gccgo*. La herramienta *gccgo* interactúa con el compilador de C de GNU, pero no está tan desarrollado como las herramientas *gc*, cuyo esquema de nombrado deriva de Plan 9 [5]. El número designa la plataforma, donde el *5* representa a ARM, el *6* a los sistemas x86 de 64 bits, y el *8* a los sistemas x86 de 32 bits. La letra designa a la propia herramienta (ver Tabla 1).

La Figura 1 muestra el proceso de compilación y enlazado, mientras que el Listado 1 contiene el típico programa "Hola Mundo". La línea 3 contiene un *import*, que nos recuerda a los de Python o Java. Go requiere que los programas cuenten con una función *main()*, que será el punto de entrada, igual que en C o C++. La función *Println()* de la librería *fmt* muestra finalmente el texto. Los desarrolladores han usado deliberadamente un esquema de nombrado en el que la primera letra de las funciones es mayúscula. Una importante característica de Go es que las variables y funciones son visibles globalmente cuando sus nombres comienzan con una letra mayúscula.

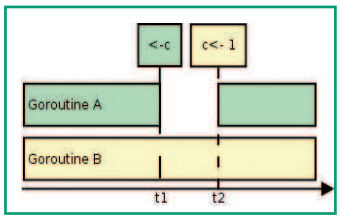

Figura 2: Las Goroutines se comunican a través de canales. En el momento *t1*, la Goroutine *A* lee desde el canal y por tanto lo bloquea hasta el momento *t2*, en el que la Goroutine *B* escribe datos a dicho canal.

Fácil Entrada

En Go, el punto y coma no finaliza una instrucción; en lugar de eso se usa como separador, como en una lista. Cuando el programa consta de una única instrucción, no hacen falta los puntos y coma. Go hereda de varios lenguajes (C, C++, Python, etc.), pero la sintaxis difiere en varios puntos. Google ofrece tanto una introducción general a la sintaxis del lenguaje [6] como un resumen de los temas más avanzados [7].

En este artículo describimos un pequeño proyecto de programación que demuestra las capacidades cliente-servidor de Go y siembra las bases para un programa de chat mínimo. El proceso servidor espera conexiones TCP en un puerto específico; cuando establecen la conexión, los clientes envían mensajes al servidor usando el formato definido y luego terminan.

Tareas del Servidor

El servidor, implementado en el Listado 2, empieza importando paquetes en la línea 3 y luego los usa como llamadas de

Make vs. New

Go incluye dos palabras clave para propósitos similares: el programador puede usar tanto *make* como *new* para reservar memoria. La palabra clave *new* (popular en lenguajes orientados a objetos como C++ o Java) reserva memoria para un nuevo objeto y devuelve un puntero a la instancia recién creada del tipo de objeto seleccionado como valor de retorno. El parámetro *new* define el tipo para el cual reservará Go el espacio en memoria.

Por contra, el desarrollador usará *make* para crear slices, mapas o canales. Esta palabra clave no soporta otros tipos. El valor devuelto aquí no es un puntero a una nueva instancia de los tipos de datos pasados, sino el propio valor. Además, el objeto resultante se inicializa internamente y, por tanto, se puede empezar a usar inmediatamente. Cuando se usa *new* con estos tres tipos de datos, Go devuelve un puntero *nil*, ya que las estructuras de datos subyacentes no están inicializadas [7].

La palabla clave *make* soporta además argumentos adicionales. En las slices, dichos argumentos son la longitud y la capacidad del campo en cuestión. El primero define la longitud actual de la slice a la hora de crearla, mientras que el segundo es la longitud del campo subyacente (es decir, la longitud hasta la cual podrá crecer la slice).

A modo de ejemplo, en la línea 50 del Listado 2 se crea una slice de bytes basada en un array de longitud *message.SenderLen*.

Incluso tratándose de diferencias lógicas, cabe preguntarse por qué los desarrolladores de Google han decidido implementar dos palabras clave distintas para la comisión de tareas tan similares. Sobre todo los recién llegados, tendrán dificultades a la hora de apreciar la diferencia.

Tabla 1: Herramientas de Go (64 bits)

Nombre	Descripción
6a	Ensamblador de Go
6c	Compilador de C de Go (*cgo* hace uso de esta herramienta)
6g	Compilador de Go
6l	Enlazador de Go
cgo	Crea paquetes y programas que llaman a código en C

Listado 1: hello.go

```
01 package main
02
03 import "fmt"
04
05 func main () {
06   fmt.Println("¡Hola Mundo!")
07 }
```

librería. La línea 15 pone de relieve el hecho de que los desarrolladores han invertido el orden de las palabras clave en las declaraciones de variables. Según la documentación del proyecto, de este modo se mejora la claridad de la sintaxis y se ahorra escritura. Go además identifica automáticamente los tipos cuando la declaración y la inicialización tienen lugar en el mismo paso, haciéndose redundantes las definiciones de tipo adicionales. El código comienza definiendo dos constantes de tipo *int* estándar y una variable global de tipo *int* *. La variable *listenPort* es por tanto un puntero. Esta característica proviene de C/C++.

Módulos Útiles

La función *main()* de la línea 80 parsea los parámetros de la línea de comandos. Para ello se basa en las funciones proporcionadas por el paquete *flag*.

Una de las características especiales de Go es el sistema de canales empleado en la comunicación entre *Goroutines*, el homólogo de los hilos (ver Figura 2). La línea 83 crea un nuevo canal, que los hilos individuales pueden utilizar para intercambiar datos de tipo boleano. El cuadro titulado "Make vs. New" describe la palabra clave *make*, usada para crear nuevas instancias, y las diferencias que tiene con *new*.

Listado 2: server.go

```
001 package main
002
003 import (
004 "bytes";
005 "encoding/binary";
006 "flag";
007 "fmt";
008 "net";
009 "os";
010 "os/signal";
011 "syscall";
012 "./build/msg/msg"
013 )
014
015 const (
016  defPort = 7777;
017  bufSize = 1024
018 )
019
020 var listenPort *int =
    flag.Int("p", defPort, "puerto
    en el que esperar conexiones")
021
022 // esperamos conexiones TCP
    entrantes
023 func acceptor(listener
    *net.TCPListener, quit chan
    bool) {
024  var buf [bufSize]byte
025
026  for {
027   conn, e :=
    listener.AcceptTCP()
028   if e != nil {
029    fmt.Fprintf(os.Stderr,
    "Error: %v\n", e)
030    continue
031   }
032
033   num, e := conn.Read(&buf)
034   if num < 0 {
035    fmt.Fprintf(os.Stderr,
    "Error: %v\n", e)
036    conn.Close()
037    continue
038   }
039
040   go handleClient(conn,
    buf[0:num])
041  }
042 }
043
044 // manejamos una conexión de un
    cliente
045 func handleClient(conn
    *net.TCPConn, bytebuf []byte) {
046  message := new(msg.Message)
047  buf :=
    bytes.NewBuffer(bytebuf)
048
049  binary.Read(buf,
    binary.LittleEndian,
    &message.SenderLen)
050  s := make([]byte,
    message.SenderLen)
051  buf.Read(s)
052  message.SetSender(string(s))
053  binary.Read(buf,
    binary.LittleEndian,
    &message.DataLen)
054  d := make([]byte,
    message.DataLen)
055  buf.Read(d)
056  message.SetData(string(d))
057
058  fmt.Printf("%s connected\n >
    %s\n\n", message.GetSender(),
    message.GetData())
059
060  conn.Close()
061 }
062
063 // leemos del canal
    signal.Incoming
064 // se recibe SIGINT
065 func signalHandler(quit chan
    bool)
066 {
067  for {
068   select {
069    case sig := <-signal.Incoming:
070    fmt.Printf("Recibida señal
    %d\n", sig)
071    if sig.(signal.UnixSignal) !=
    siscall.SIGINT {
072     continue
073    }
074    quit<- true
075    return
076   }
077  }
078 }
079
080 func main() {
081  flag.Parse()
082  address :=
    fmt.Sprintf("%s:%d",
    "127.0.0.1", *listenPort)
083  quit :=make(chan bool)
084
085  socket, e :=
    net.ResolveTCPAddr(address)
086  if e != nil {
087   fmt.Fprintf(os.Stderr,
    "Error: %v\n", e)
088   os.Exit(1)
089  }
090
091  go signalHandler(quit)
092
093  fmt.Printf("A la escucha en
    %s:%d\n\n",
    socket.IP.String(),
    socket.Port)
094  go acceptor(listener, quit)
095
096  for {
097   select {
098    case <- quit:
099    fmt.Printf("Cerrando\n");
100    listener.Close()
101    return
102   }
103  }
104 }
```

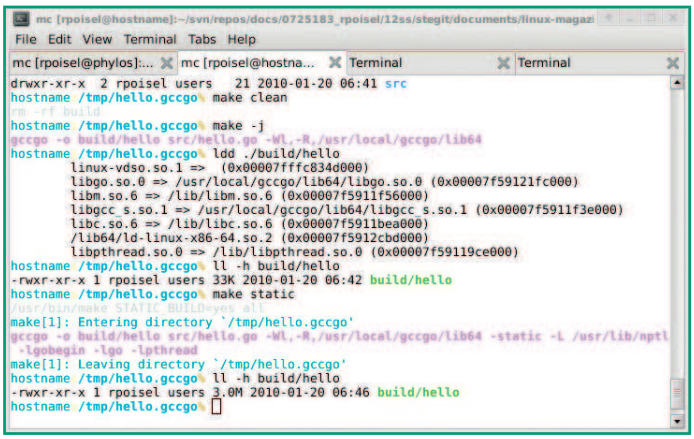

Figura 3: Chocante para los desarrolladores: Go ofrece tanto un pequeño binario de 33KB con una ingente cantidad de librerías en tiempo de ejecución, o un programa enlazado estáticamente con un peso de 3MB.

La sintaxis `:=`, que Go toma prestada de Pascal, ofrece una variante más corta para la declaración de variables con inicialización simultánea. El programa usa este idioma en las líneas 85 y 90 para crear un socket y un manejador de TCP con la ayuda de la librería *net*.

Las funciones pueden devolver varios valores en Go; así se explica la lista de variables que aparece en la parte izquierda de la asignación de la línea 85.

La palabra clave *go* inicia una Goroutine. El programa principal crea un hilo paralelo en la línea 96 con la función *signalHandler()* mientras continúa ejecutándose normalmente. El programa ejecuta por tanto dos Goroutines. Una de ellas, comenzando en la línea 66, es responsable del manejo de señales; la otra empieza en la línea 23 y escucha en el puerto TCP. El bucle infinito al final de *main()* utiliza *select* para esperar en el canal a la llegada de un mensaje *quit* entrante.

En Espera y a la Escucha

Cuando el bucle recibe el mensaje, el programa termina. El desarrollador puede usar la instrucción *select* para esperar en varios canales simultáneamente. Cuando las diferentes instrucciones llegan a varios canales, el entorno de ejecución de Go selecciona aleatoriamente un mensaje y lo procesa.

Go utiliza la palabra clave *func* para la introducción de funciones. En el caso de *acceptor()*, el programa espera conexiones TCP en el bucle infinito de la línea 23 y lee un máximo de 1024 bytes en cada conexión. La función le pasa entonces a la Goroutine *handleClient()* el número real de bytes.

El argumento de la transferencia con la sintaxis *buf[0:num]* de la línea 40 es otra de las características especiales. Go crea *slices* de este modo, los cuales juegan un rol importante al trabajar con campos en el lenguaje Go. Las slices son campos similares a los usados en otros lenguajes de programación, como C o C++. Dicho de otro modo, apuntan a un área de memoria que incluye alguna información (como pueda ser la longitud).

Para crear una slice tiene que existir un campo previamente (por ejemplo *buf*, que se crea en la línea 24). También la puede generar Go automáticamente, como en la línea 50, al crearse mediante *make*.

A la función *handleClient()*, que es llamada en cada conexión de un cliente, se le pasa una slice con los bytes (es decir, los que leyó). Las líneas 49 a la 56 dan un rodeo a través de los bytes y paquetes binarios dentro de una estructura definida para la transmisión del mensaje, que se especifica en el código del Listado 3, *msg.go*.

Formato del Mensaje

La línea 1 del archivo *msg.go* crea un paquete llamado *msg*, que en la línea 3 se define como una estructura con cuatro elementos. El código utiliza letras mayúsculas para demostrar la forma en la que Go publica los elementos. Los programadores pueden usar métodos de acceso para manipular los otros dos atributos (es decir, *sender* y *data*).

Estos métodos los introduce la palabra clave *func* (igual que las funciones); sin embargo, esperan un objetivo después de la palabra clave (*m* en este caso). El modelo de la clase difiere por tanto de los modelos de C++ y Java, en los que los métodos se enumeran dentro de las definiciones de tipo.

Básicamente, cualquier tipo puede ser un objetivo; dicho de otro modo, están permitidos los tipos de datos primitivos y los que no son punteros. Esto significa que el método *String()* se puede definir para tipos de datos arbitrario, pudiendo el desarrollador modi-

Listado 3: msg.go

```
01 package msg
02
03 type Message struct {
04  SenderLen uint32;
05  sender []byte;
06  DataLen uint32;
07  data []byte;
08 }
09
10 func (m *Message) GetSender()
   string {
11  return string(m.sender)
12 }
13
14 func (m *Message) SetSender()
   string {
15  m.sender = stringToBytes(s)
16  m.senderLen = uint32(len(s))
17 }
18
19 func (m *Message) GetData()
   string {
20  return string(m.data)
21 }
22
23 func (m *Message) SetData(s
   string) {
24  m.data = stringToBytes(s)
25  m.dataLen = uint32(len(s))
26 }
27
28 // función auxiliar para
   convertir una
29 // cadena dada en una slice de
   bytes
30 func stringToBytes(s string)
   []byte {
31  slice := make([]byte, len(s))
32
33  for i := 0; i < len(s); i++ {
34   slice[i] = s[i]
35  }
36  return slice
37 }
```

ficar la salida de *Println()* para cubrir sus necesidades.

El método

```
func (m *Message) String()➥
string {
 return fmt.Sprintf(
  "Sender=%s, Data=%s",
  m.sender, m.data)
}
```

define una salida personalizada para la estructura de *Message*, que aquí se emplea al llamar a la función *fmt.Println(m)*.

Signos Vitales

El programa cliente actúa de un modo similar a como lo hace el servidor. El programa, así como el resto de ejemplos, se pueden descargar desde la sección de código archivado del sitio web de Linux Magazine [8].

Una vez inicializado el socket por el cliente, éste abre una conexión TCP hacia el servidor, crea una estructura *Message* del lado del cliente con información procedente de la línea de comandos y la transmite a través del socket. Después de hacerlo, el cliente termina.

Todo aquel que disfrute con la experimentación, podría desarrollar una herramienta de chat completa con una lista para la gestión de clientes conectados. Un buen método es usar un mapa, que es el equivalente de Go a un hash de Perl. El servidor envía entonces los mensajes entrantes a todos los clientes enumerados en el mapa.

En los ejemplos se muestran algunas construcciones interesantes de Go. El lenguaje de programación viene además con una librería completa de paquetes con propósitos varios. Aunque sería absurdo esperar un ámbito como el de JEE (*Java Enterprise Edition*), a pesar de su corta edad, la lista de paquetes de Go es bastante impresionante, especialmente teniendo en cuenta que de seguro Google hará que crezca. Para disponer de una visión general, conviene echar un vistazo al directorio *src/pkg*, bajo el directorio de instalación de Go.

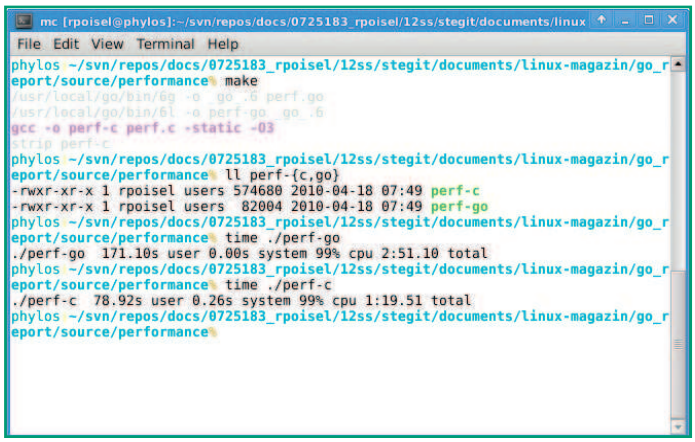

Figura 4: Rápida, pero mejorable: La implementación del Bubble Sort [12] tarda en Go aún más del doble en comparación con la implementación en C.

Otras interfaces

Las interfaces, una característica del lenguaje con la que los programadores de Java ya estarán familiarizados, requieren que se les dedique una investigación algo más detallada. Aunque el concepto de las interfaces tiene significado en Go, la implementación es diferente. Go carece de palabras clave como los *implements* de Java. El código soporta la interfaz una vez implementa todos los métodos definidas por ésta.

Un ejemplo bien conocido de esto es la interfaz *Reader* del paquete *io* [9]. Dicha interfaz define un método *Read()* público, que acepta una slice de bytes y devuelve dos valores. Cada clase que contenga un *Read()* con esta firma en concreto implementa automáticamente la interfaz *Reader*. De este modo el desarrollador puede pasar sus propios tipos de variable a funciones que esperen un *io.Reader*, ahorrando tecleos adicionales y facilitando el intercambio flexible de implementaciones.

El ejemplo del chat usa esta técnica. Dado que el Listado 3 contiene un método *string()* para el tipo *Message*, dicho tipo implementa la interfaz *Stringer* [10]. De aquí en adelante, las funciones de salida usarán este método auto definido para formatear los objetos.

Listado 4: perf.go

```
01 package main
02
03 const (
04  SIZE = 200000;
05  MULT = 1103515245;
06  MASK = 4096;
07  INC = 12345;
08 )
09
10 func bubbleSort(numbers
     []uint32) {
11  for i := (len(numbers) - 1); i >=
     0; i-- {
12   for j := 1; j <= i; j++ {
13    if numbers[j-1] > numbers[j] {
14     numbers[j-1], numbers[j] =
15      numbers[j], numbers[j-1]
16    }
17   }
18  }
19 }
20
21 func main() {
22  var arrayOfInt [SIZE]uint32
23  var lNext uint32 = 1;
24
25  for i := 0; i < len(arrayOfInt);
     i++ {
26   lNext = lNext * MULT + INC;
27   arrayOfInt[i] =
     (uint32)(lNext/MASK) % SIZE;
28  }
29  bubbleSort(&arrayOfInt)
30 }
```

¡GCC Go!

El compilador, con GCC como backend, proporciona una alternativa al compilador nativo de Go, *gc*. El proyecto Go proporciona notas sobre la instalación [11]. No plantea tareas imposibles para programadores experimentados, pero descarga más de 65000 archivos desde el repositorio fuente, ocupando no menos de 1.3GB de espacio en disco.

Viendo el tamaño del código generado, puede que se prefiera instalar el compilador alternativo. Mientras que los binarios generados con el compilador nativo pesan varios cientos de kilobytes, el tamaño de los ejecutables

creados con *gccgo* y las librerías enlazadas dinámicamente ocuparán lo que un típico programa en C. El aspecto negativo es que los programas dependerán de la librería en tiempo de ejecución de Go así como de otras herramientas (ver Figura 3).

¡Despegamos!

La velocidad de ejecución de los binarios difiere infinitamente, aunque depende en gran medida de la configuración usada y de las optimizaciones hechas. A fin de conseguir una velocidad de datos objetiva y compatible, el programa de prueba del Listado 4 evita esperas incalculables provocadas por las operaciones de entrada y salida.

Este simple programa de medición y prueba implementa un algoritmo *Bubble Sort* [12] llenando un array con una secuencia reproducible de números pseudoaleatorios de 32 bits. El programa reorganiza entonces los valores en orden ascendente. Este ejemplo evita en la medida de lo posible cualquier característica específica del lenguaje.

El ejemplo en C implementa los arrays con punteros (ver Listado 5), mientras que el ejemplo de Go hace uso de slices. En ambos casos, los compradores crearon binarios estáticos sin símbolos de depuración.

Como hemos mencionado anteriormente, el código fuente de los programas a los que hace referencia este artículo está disponible desde el sitio web de Linux Magazine en [8]. En la Figura 4 se puede apreciar cómo el ejemplo en C puro sigue siendo mucho más rápido.

Nuevos Objetivos

En el futuro, las utilidades de Go (*Go Utils*) incorporarán un depurador. Google está tratando también de fomentar la cooperación entre Go y C. Los diseñadores del lenguaje aún siguen filosofeando sobre si se va a permitir la inclusión de aspectos provenientes de lenguajes orientados a objetos, como puedan ser las excepciones y los genéricos. Se están planteando escribir las futuras versiones del compilador de Go en el propio lenguaje Go, ya que tanto el analizador léxico como el parseador están disponibles en forma de librerías de Go. Estos y otros detalles se pueden consultar en la hoja de ruta del proyecto [13].

Muchos de los conceptos de Go parecen bastante prometedores, pero el lenguaje aún tendrá que demostrar su valía con proyectos de mayor envergadura. Su sintaxis, simple pero potente, y a medio camino entre las de Java y C, interesará a todo aquel que esté familiarizado con alguno de estos lenguajes. El pragmático diseño de la librería promete resultados rápidos. Hay detalles del lenguaje Go que parecen bastante académicos, pero el lenguaje en sí es innegablemente interesante. El entorno de compilación y ejecución, así como los parámetros principales, como la velocidad de ejecución, necesitan aún cierto lustre, pero el equipo de Go está ya trabajando para solucionar esos problemas. La única cuestión pendiente es si Go será adoptado por los desarrolladores de la corriente dominante. ■

Listado 5: perf.c

```
01 #include <stdint.h>
02 #include <stdlib.h>
03
04 #define SIZE 200000
05 #define SEED 1
06 #define MULT 1103515245L
07 #define MASK 4096
08 #define INCR 12345
09
10 void bubbleSort(int numbers[],
   int array_size)
11 {
12  int i, j, temp;
13
14  for (i= (array_size - 1); i >= 0;
   i--)
15  {
16   for (j = 1; j <= i; j++)
17   {
18    if (numbers[j-1] > numbers[j])
19    {
20     temp = numbers[j - 1];
21     numbers[j - 1] = numbers[j];
22     numbers[j] = temp;
23    }
24   }
25  }
26 }
27
28 int main(void)
29 {
30  uint32_t *lArray = NULL;
31  uint32_t lCnt = 0;
32  uint32_t lNext = SEED;
33
34  lArray = (uint32_t*)
    malloc(sizeof(uint32_t) * SIZE);
35
36  for (lCnt = 0; lCnt < ARRAY_SIZE;
    lCnt++)
37  {
38   lNext = lNext * MULT + INCR;
39   lArray[lCnt] = (uint32_t)
    (lNext / MASK) % SIZE;
40  }
41  bubbleSort(lArray, ARRAY_SIZE);
42  return free(lArray);
43 }
```

LOS AUTORES

Marcus Nutzinger y Rainer Poisel forman parte del equipo científico del Instituto para la Investigación sobre Seguridad en TI de la Universidad de St. Pölten, Austria. Dentro del ámbito del proyecto "StegIT-2", investigan métodos para prevenir la inclusión de mensajes secretos en llamadas de voz sobre IP. Ambos autores enseñan seguridad TI.

RECURSOS

[1] Xkcd: *http://xkcd.com/303*
[2] Google Tech Talk sobre el lenguaje de programación Go: *http://www.youtube.com/watch?v=rKnDgT73v8s*
[3] Howto de instalación del proyecto: *http://golang.org/doc/install.html*
[4] Fallos en cliente-servidor *HTTP: http://code.google.com/p/go/issues/detail?id=5*
[5] Plan 9 de los laboratorios Bell: *http://plan9.bell-labs.com/plan9/*
[6] Tutorial de Go: *http://golang.org/doc/go_tutorial.html*
[7] Effective Go: *http://golang.org/doc/effective_go.html*
[8] El código fuente de este artículo: *http://www.linux-magazine.es/Magazine/Downloads/64*
[9] Interfaz Reader *io.go*: *http://golang.org/src/pkg/io/io.go*
[10] Interfaz Stringer *print.go*: *http://golang.org/src/pkg/fmt/print.go*
[11] Instalando y configurando *gccgo*: *http://golang.org/doc/gccgo_install.html*
[12] Bubble Sort: *http://www.sorting-algorithms.com/bubble-sort*
[13] Hoja de ruta para Go: *http://go.googlecode.com/hg/doc/devel/roadmap.html*