El patrón de eventos “Handler" en Arduino para la implementación de un protocolo simple de comunicación basado en eventos

Ing. Oscar Beltrán Gómez

Tecnológico Nacional de México/I.T. Chihuahua 2, DEPI.
Av. de las Industrias 11101, Complejo Industrial Chihuahua, 31130 Chihuahua, Chih., Mex.
Tels. (614) 442-50-00
oscar.bego@chihuahua2.tecnm.mx

Dr. Rafael Sandoval Rodríguez

Tecnológico Nacional de México/I.T. Chihuahua 2, DEPI.
Av. Tecnologico 2909, Tecnológico, 31200 Chihuahua, Chih. Mex.
Tels. (614) 201-20-00
rafael.saro@chihuahua2.tecnm.mx

M.I. Arturo Legarda Sáenz

Tecnológico Nacional de México/I.T. Chihuahua 2, DEPI.
Av. de las Industrias 11101, Complejo Industrial Chihuahua, 31130 Chihuahua, Chih., Mex.
Tels. (614) 442-50-00
arturo.ls@chihuahua2.tecnm.mx

Resumen

Cada vez es más frecuente ver sistemas que interactúan con algún aspecto físico y disponer de la información que estos generan o requieren puede representar un reto por la gran variedad de tecnologías tanto en “*computación física*” como en la “*computación tradicional*”.

La programación basada en eventos puede representar un punto de convergencia al despreocuparse de cómo los componentes se comportan de manera individual para lograr un comportamiento propagado por eventos.

Este artículo trata del patrón de eventos “*Handler*” en el desarrollo de una librería ejecutable en la placa Arduino para la implementación de un protocolo simple de comunicación basado en eventos.

1. Introducción

Cada vez es más frecuente ver sistemas que interactúan con algún aspecto físico y aunque contamos con conceptos como el “*Internet de las cosas*”, disponer de la información que estos elementos físicos generan o proporcionar las que estos requieren puede representar un verdadero reto para los desarrolladores.

La dificultad radica en la gran variedad de dialectos, paradigmas y tecnologías que soportan una mayor cantidad de plataformas, es decir, la implementación de sistemas que se puedan adaptar a tal cantidad de variantes es una tarea compleja; también influye que la información rara vez queda aislada, es decir, normalmente ocupamos la información “distribuida” en: bases de datos, aplicaciones de escritorio y móviles, páginas web y más recientemente en aplicaciones implementadas en lo que se conoce como computación física o IoT.

> “*La computación física implica el diseño de objetos interactivos que se pueden comunicar con las personas usando sensores y actuadores controlados por un comportamiento implementado en software que se ejecuta dentro de un microcontrolador*” [1].

Por otra parte, y para efectos de este artículo, la “*computación tradicional*”, la que de forma indirecta interactúa con el hardware que la contiene, normalmente a través de algún intermediario o sistema operativo; este “computo más tradicional” ha evolucionado para el acceso intensivo de la información, mientras que “*la computación física*” aún está en ese camino.

En consecuencia, los desarrolladores que implementan aplicaciones de cómputo físico necesitan adentrarse en áreas como redes de computadoras y electrónica para atender o

propagar la información que se presenta en forma de suceso de la contraparte física del sistema y lograr una arquitectura homogénea.

La programación basada en eventos o sucesos puede representar un punto de convergencia, para el desarrollo de sistemas que interactúen con aspectos físicos y la ya mencionada programación tradicional (CT), al proveer de:

"un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema o que ellos mismos provoquen" [2].

La propuesta de una arquitectura basada en eventos distribuidos es despreocuparse de cómo los componentes, fisicos e intangibles, de un sistema se comportan de manera individual para lograr un comportamiento propagado por eventos.

2. Objetivo

En este artículo se aborda el patrón "*Handler*" para la creación de una librería para la placa de desarrollo Arduino [3] en su variante *"Handler sin Cabeza"* para la implementación de un protocolo simple de comunicación basado en eventos entre ya mencionadas aplicaciones de CT y las desarrolladas a su vez en "computación física".

Esta arquitectura considera como puntos fundamentales:

- Un desarrollo dirigido por eventos.
- Bajo acoplamiento.
- Un enfoque de propagación de eventos en red (sockets).

3. Tecnologías utilizadas para la demostración

En la demostración e implantación de la arquitectura se utilizan las siguientes tecnologías:

- Arduino UNO R3 [4].
- Lenguaje de programación C++ [5].

4. El Patrón "*Handler"* para el desarrollo dirigido por eventos

El principal patrón en el desarrollo dirigido por eventos es el llamado "*Handler*" [6]. El patrón "*Handler"* (ver figura. 1), define un flujo de eventos, un despachador y un conjunto de manejadores.

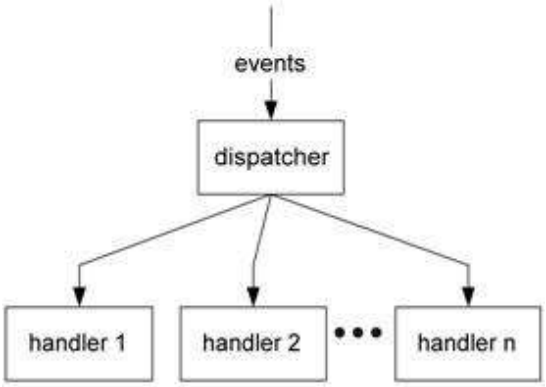

Figura 1. Patrón *Handler*.

Podríamos decir que el despachador es quien rutea los eventos a uno o varios manejadores (*Handler's*), los cuales realizan alguna función de acuerdo al tipo de evento; esta lógica de negocio incluye un ciclo para seguir atendiendo los eventos entrantes.

De las diferentes variantes del patrón "*Handler*" son: "*Handler Extendido*", "*Handler con Colas de Eventos*" y el "*Handler sin Cabeza*" [7], figura 2; este último se ajusta más al modelo que en este artículo se describe.

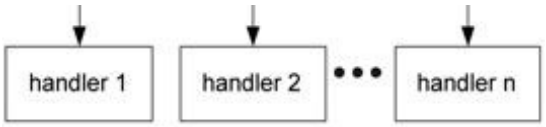

Figura 2. Patrón *Handler sin Cabeza*.

El "*Handler sin Cabeza*", ó en inglés "*Headless Handlers Pattern*" [7], oculta al despachador, dejando únicamente visible la lista de manejadores.

Los sistemas basados en cómputo físico pueden tratar flujos de eventos infinitos, sin embargo, en

la mayoría de los sistemas basados en eventos el flujo de eventos es finito. La lógica de la aplicación debe de contar con la capacidad de salir del ciclo de eventos ó bien ingresar al mismo.

```
1:  int socketClientEvent::on (char dato, void    (*callb)())
2:  {
3:     Elemento *nuevo_elemento;
4:
5:     if ( (nuevo_elemento =
6:           (Elemento *)
7:            malloc (sizeof (Elemento))) == NULL
8:        )
9:          return -1;
10:
11:    nuevo_elemento->dato = dato;
12:    nuevo_elemento->callb = callb;
13:
14:    nuevo_elemento->siguiente = lista->inicio;
15:    lista->inicio = nuevo_elemento;
16:
17:    return 0;
18:  }
19:
20:
21:
22:
```

Figura 3. Método "*on*".

5. Implementación de la librería "socketClientEvent"

La librería "*socketClientEvent*" [8] consta de tres mecanismos principales: el primero es el método llamado "*on*", fig. 3, este método asigna una correspondencia de un evento a una función específica o manejador definiendo una ruta. Este método, de forma interna, agrega a una "*Lista Enlazada*" el par evento-manejador. El evento está definido, en este caso y por facilidad, como un carácter o "*char"*, mientras que el manejador está representado por un apuntador a función.

```
1:  void socketClientEvent::disparar
2:  (char c) {
3:
4:  Elemento *actual;
5:  actual = lista->inicio;
6:
7:    while (actual != NULL) {
8:      if(actual->dato == c )
9:          actual->callb();
10:
11:     actual = actual->siguiente;
12:   }
13: }
```

Figura 4. Método "*disparar*".

El segundo mecanismo, fig. 4, es el disparador. Cuando es llamado este método ocupa el paso de un parámetro tipo "*char"*, correspondiente al evento, para buscarlo en la "*Lista Enlazada"* y posterior mente llamar a la función que apunta.

```
1:  void socketClientEvent::listener() {
2:
3:    _cliente = _server->available();
4:
5:    if (_cliente) {
6:
7:      while (_cliente.connected()) {
8:
9:        char data = _cliente.read();
10:
11:       disparar(data);
12:
13:       switch(data)  {
14:
15:         case 'x':
16:             _cliente.print("adiós :)
17: ");
18:             _cliente.stop();
19:           break;
20:
21:         default:
22:           break;
23:       }
24:     }
25:   }
26:
27:   _cliente.stop();
28: }
```

Figura 5. Método "*listener*".

El tercer mecanismo, fig. 5, representa al despachador; este método, que continuamente está

en espera de clientes (clientes sockets) rutea sus peticiones a algún método a disparar con los datos que recibe de la misma petición.

"En código de la figura 6, líneas 9, 10 y 11, muestra como de forma desacoplada se definen los manejadores". Este tipo de ensamblaje es apropiado ya que el mismo evento puede estar ligado a más de un manejador, líneas 25 y 28.

"Hay que recordar que por la forma en que está programada la *"Lista Enlazada"* los eventos desencadenarán los manejadores en el orden en que fueron ingresados con el método "*on*". Lo anterior en las líneas de la 25 a la 28.

```
1:  #include <SPI.h>
2:  #include <Ethernet.h>
3:  #include <socketClientEvent.h>
4:
5:  byte mac[]  = {  0xDE, 0xED, 0xBA,
6:  0xFE, 0xFE, 0xED };
7:  byte ip[]   = { 192, 168, 2, 2 };
8:
9:  void f()  {
10:    Serial.println("se llamó a f");
11: }
12: void f1() {
13:    Serial.println("se llamó a f1");
14: }
15: void f2() {
16:    Serial.println("se llamó a f2");
17: }
18:
19: EthernetServer server(23);
20: socketClientEvent sc =
21: socketClientEvent(server);
22:
23: void setup() {
24:
25:   Serial.begin(9600);
26:   Ethernet.begin(mac, ip);
27:
28:   sc.on('a', f);
29:   sc.on('1', f1);
30:   sc.on('2', f2);
31:   sc.on('c', f);
32:
33:   //podemos hacer algo antes de
34:   //cerrar
35:   sc.on('x', f1);
36: }
37:
38: void loop() {
39:
40:   sc.listener();
41: }
```

Figura 6. Código para la placa Arduino.

La librería se ajusta al patrón "*Handler sin Cabeza*" ya que de forma natural oculta la figura del despachador dejando muy a la vista la colección de manejadores.

6. Conclusiones

La arquitectura "*Handler*" de eventos es claramente una forma de la implementación del patrón cliente-servidor ya que el servidor funciona como un despachador de eventos que espera a la escucha de una o varias peticiones de clientes; estas solicitudes son tratadas como eventos que son re-enviados a uno o varios manejadores para posteriormente devolver un resultado.

Aunque los sistemas basados en eventos suelen ser multihilos, "para atender eventos en paralelo", "los sistemas como el Arduino solo cuentan con un procesador, esta es una consideración a tomar en cuenta".

La una arquitectura basada en eventos podría tomar encueta no solo los eventos que se provengan de la red, sino según el caso, el despachador de eventos podría rutear más de un solo tipo de sucesos, por ejemplo, sucesos como la presión de un botón o el cambio de estado de un sensor.

Algunos patrones de diseño de comportamiento como el "*Observer*"[9], el "*Visitor*"[10] y "*Reactor*" [11] roporcionan, también, una pauta en la mayoría de los proyectos basados en eventos.

Ahora el mercado informático ofrece y cuenta con muchas herramientas enfocadas a la integración de aplicaciones a partir del paso de mensajes, que habitualmente suelen representar eventos [12, 13, 14, 15, 16, 17].

La librería denominada "*socketClientEvent*" es solo una demostración del patrón "*Handler sin Cabeza*" en la placa Arduino; se recomienda su uso solo para fines didácticos y demostrativos.

7. Referencias

[1] Introducción a Arduino, Massimo Banzi, Mayo 2012, O'REILLY/ANAYA, I.S.B.N.: 978-84-415-3177-2.

[2]Sarmiento J, Díaz de León J, Chimal J. TESIS: UN PARADIGMA PROACTIVO ORIENTADO A OBJETOS, INSTITUTO POLITÉCNICO NACIONAL, CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN, México D.F. a Junio 2009,

en: http://www.cic.ipn.mx/sitioCIC/images/sources/cic/tesis/A050821.pdf.
[3] Arduino - Libraries, Home Page: http://arduino.cc/en/pmwiki.php?n=Reference/Libraries
[4] Arduino - ArduinoBoardUno, Home Page: http://arduino.cc/en/Main/arduinoBoardUno
[5] Cómo programar en C++ - Harvey M. Deitel, Paul J. Deitel Pearson Educación, 2003.
[6] Programación en timepo real y bases de datos: un enfoque práctico, Josefina López Herrera, Universitat Politécnica de Cataluña, pag. 36.
[7] Event-Driven Programming: Introduction, Tutorial, History. Home Page: http://www.cnblogs.com/alex-tech/archive/2011/10/27/2227058.html.
[8] SocketClientEvent - Library, Repository: https://github.com/oscarbego/socketClientEvent.
[9] E. Gamma, et al., Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley 1995, ISBN 0-201-63361-2.
[10] The essence of the Visitor pattern, Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International, Aug 1998, pp 9 - 15.
[11] Douglas C. Schmidt, Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching, 1995.
[12] Apache Camel, Home Page: http://camel.apache.org/, Julio 2012.
[13] FuseSource Integration Everywhere, Home Page: http://fusesource.com/, Julio 2012.
[14] RabbitMQ: Spring Source, Home Page: http://www.rabbitmq.com/, Julio 2012.
[15] MuleSoft, Home Page: http://www.mulesoft.org/, Julio 2012.
[16] Apache ServiceMix, Home Page: http://servicemix.apache.org/index.html, Julio 2012.
[17] Spring Integration, Home Page: http://www.springsource.org/spring-integration/, Julio 2012.