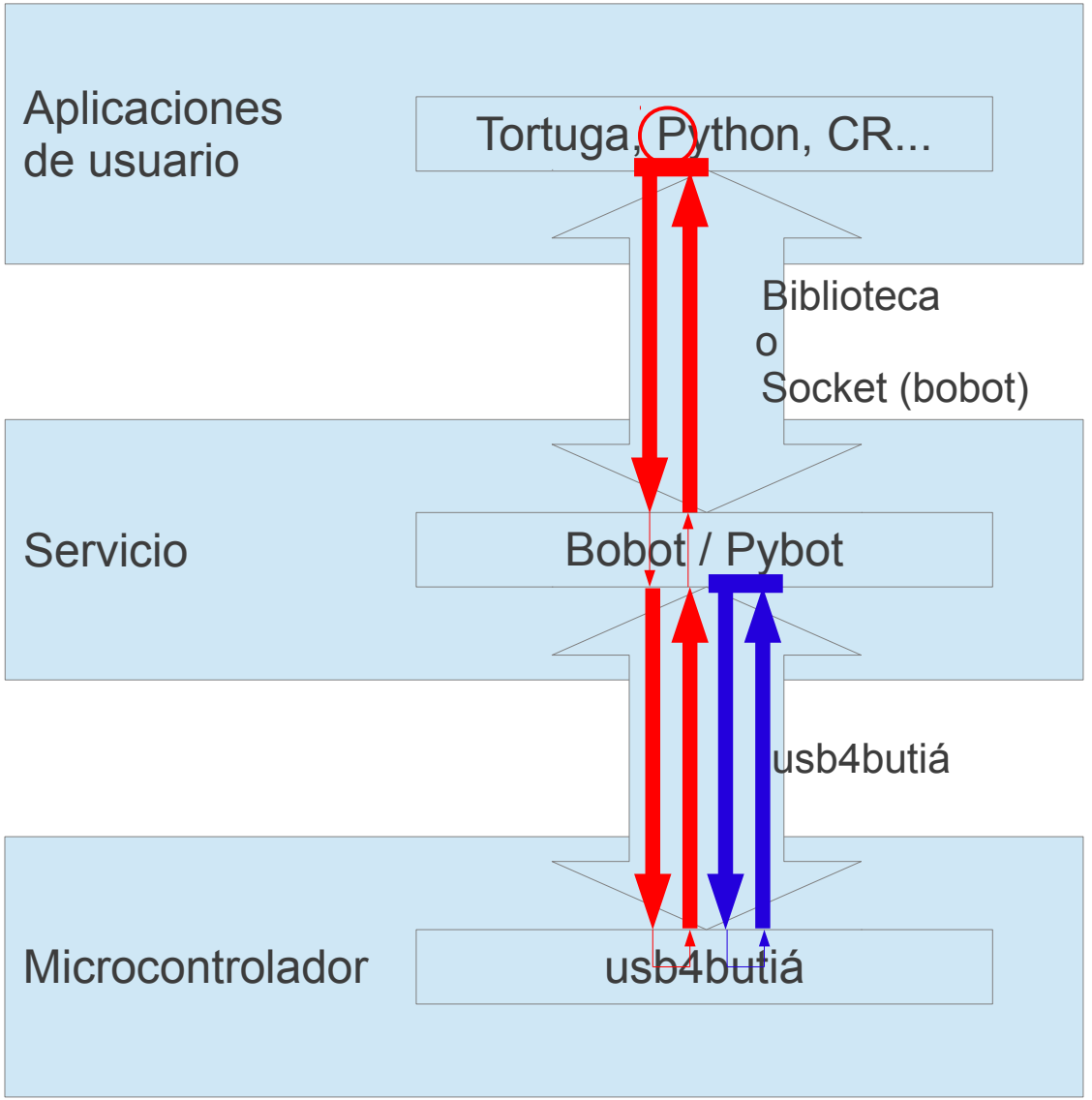
Butia 3+

Stack de software: Ideas.

jvisca@fing.edu.uy - 2013

Arquitectura Butiá <=2.0



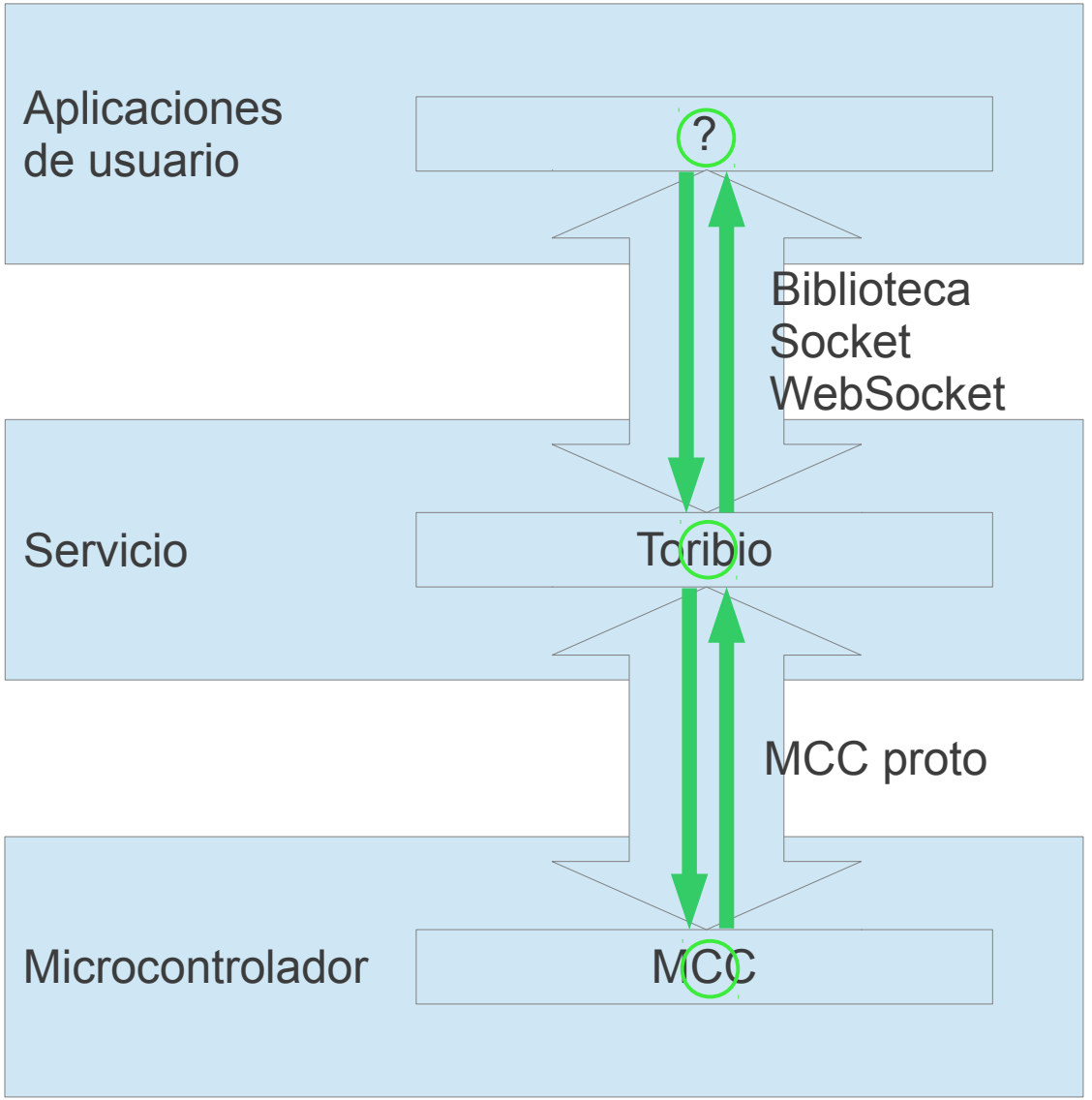
- Modelo RPC
  - En cada capa, mapeado 1 a 1 en el recorrido
- API de usuario basada en getters y setters
  - Muestreo por polling
  - Bloqueantes
- Gestión de PNP como polling desde capa de Servicio

```
while (true) {
    if (getButton()) {
        SetSpeed(0);
    }
}
```

Propuesta de Arquitectura

- Polling en CPU implica:
  - Aumento en la taza de muestro atado al uso de CPU
  - Alto uso de CPU → alto consumo
  - Latencia y jitter alta
  - *Impedance mismatch*: E/S del SO orientado a interrupciones, aplicación orientada a polling.
  - Si la aplicación es single-thread, los problemas se componen al aumentar los objetos muestreados.
- El lugar correcto para hacer polling es el uC:
  - Uso dedicado / tiempo real
  - Adyacente al objeto muestreado

Propuesta de Arquitectura



- Modelo de mensajes asincrónicos
- API de usuario flexibles:
  - Getter/Setter
  - Callbacks
  - Mensajes
- Gestión de PNP implementada dentro del uC.

```
when (getButton()) {
    SetSpeed(0)
}
```

Capa uC: MCC

- Microcontroller controller / MC2

u4a / u4b	MCC
Exportan modulos conectados a la placa-uc para ser accedidos desde el host.	Los procesos son autónomos, accedidos por un protocolo asincrónico.
Protocolo sobre usb compacto, intenta aprovechar transporte.	Sólo hay tráfico cuando hay datos. Corre sobre un enlace serial.
Todos los accesos son iniciados por el host, de la forma “request/response”.	Protocolo es asíncrono, tanto el host como la placa-uc pueden emitir mensajes.
Protocolo binario.	Protocolos binario, plano usando Bencode.
pnp implementado mediante una extensión al protocolo, con separación de módulos e instancias, y polling sobre un comando de listado.	pnp implementado nativamente en el protocolo. Soportada por mensajería asincrónica.
Cada módulo representa un dispositivo, o un servicio. Puede accederse varias instancias de un mismo módulo.	Idem.

Capa uC: MCC

- Mensajes:
  - Host a Placa-uc: *[target_pid, opcode, data]*
  - Placa-uc: *[source_pid, opcode, data]*
- Bencode:
  - Codificación: eficiente, biyectiva, 8bit-clean.
  - Numero *n* → '`i`*`n`*`e`', string s → '*`length`*`:`*`s`*', lista → '`l`*`contente`*', diccionario → 'd*`keyvalue`*`e`'
  - Ping:
    - *[0, 1, 'data']* → `'li0ei1e4:datae'`
  - Call:
    - No params: *[2, 1]* → `'li2ei1ee'`
    - Encoded params: *[2, 1, [mode=9]]* → `'li2ei1ed4:modei9eee'`

Capa uC: MCC

Diseñando APIs asincrónicas... Observación: la mayoría de las lecturas de sensores van directo a una comparación (==, <, >)

- Módulo digital:

Host → Placa-uC

[*pid*, QUERY]

- Consultar estado

Placa-uC → Host

[*pid*, STATE, *value*]

- Se emite en los cambios de estado o al ser solicitado

Capa uC: MCC

Diseñando APIs asincrónicas...

- Módulo analógico:

Host → Placa-uC

[*pid*, QUERY]

- Consultar estado

[*pid*, THRESH_UP, *values*]
[*pid*, THRESH_DN, *values*]

- Setea umbrales a vigilar

Placa-uC → Host

[*pid*, STATE, *value*]

- Se emite al cruzar uno de los umbrales, o al ser solicitado

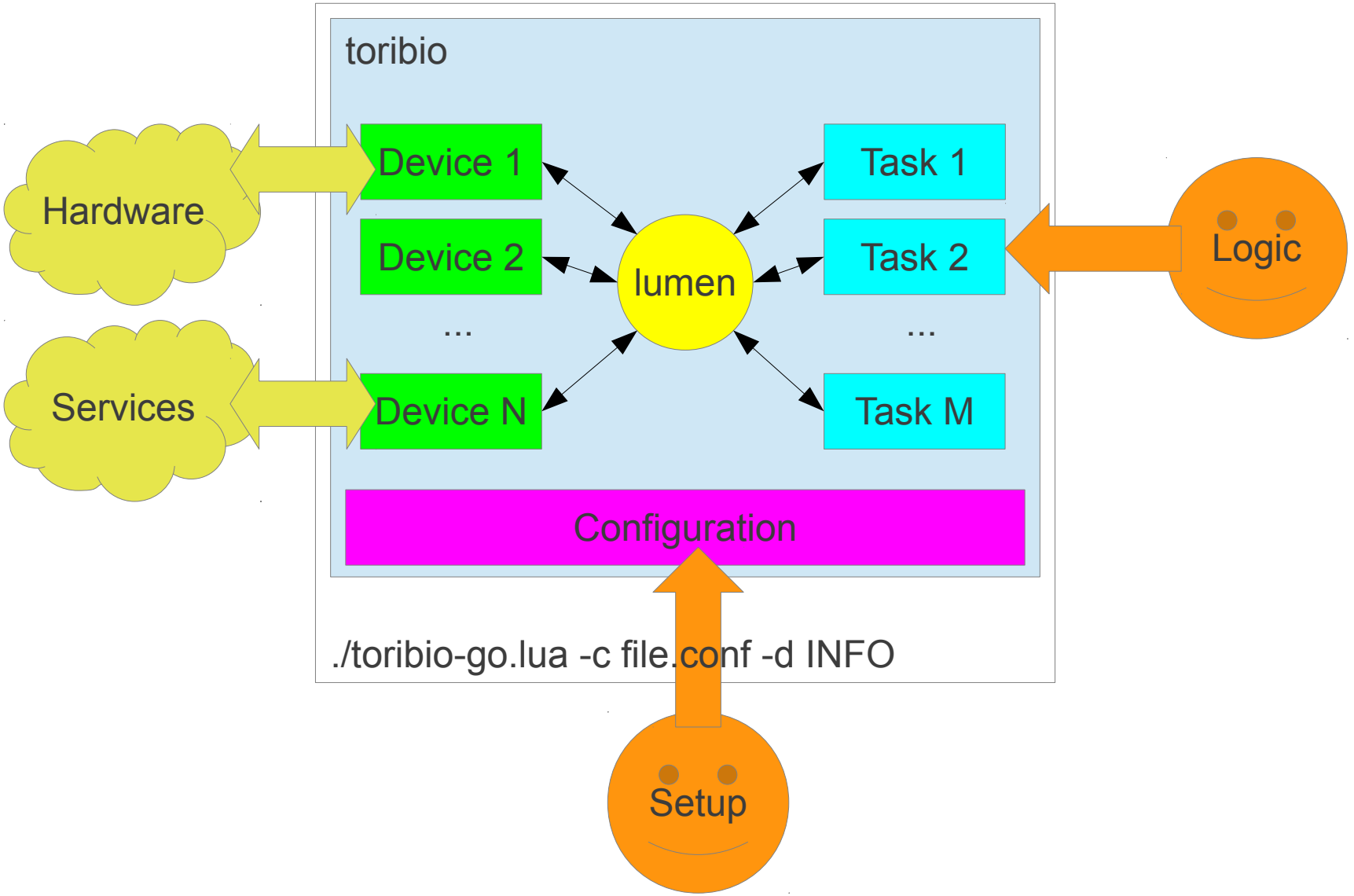
Capa uC: MCC

Demo

Capa Servicio: Toribio

- Plataforma objetivo:
  - Piso de 32Mb RAM
  - Portable entre arquitecturas (ARM, x86, MIPS...)
  - Linux
- Escrito en Lua, dependencias nativas mínimas (nixio, LuaJIT planeado)
- Implementa concurrencia cooperativa
- Orientado a concurrencia regulada por E/S
  - En particular, usando fd / sockets
- Licencia MIT

Capa Servicio: Toribio



Capa Servicio: Toribio

- Lumen: scheduler cooperativo. Basado en el concepto de señales y su captura.

```
sched.signal('event!',1)  |  ev,n=sched.wait{'event!'}
```

- Devices: abstracción de periféricos y servicios. Publican API y emiten señales.
- Tasks: tareas de usuario. Publican API y emiten señales.
  - Selector: E/S de archivos y sockets.
- Configuration: repositorio central, controla el funcionamiento.
- Herramientas: logging, servidor http, consola remota, proxys para señales remotas, etc.

Capa Servicio: Toribio

```
$ cat tutorial.conf
deviceloaders.mice.load = true
tasks.tutorial.load = true
```

```
$ cat tasks/tutorial.lua
local toribio = require 'toribio'
local M = {}
M.init = function(conf)
   local mice = toribio.wait_for_device({module='mice'})
   toribio.register_callback(mice, 'leftbutton', function(v)
      print('left:',v)
   end)
   toribio.register_callback(mice, 'move', function(x, y)
      print('move:',x,y)
   end)
end
return M
```

```
$ ./toribio-go.lua -c tutorial.conf
```

Capa Servicio: Toribio

```
local M = {}
M.init = function()
    local sched = require 'sched'
    local toribio = require 'toribio'
    local mcc = toribio.wait_for_device({module='mcc'})

    sched.sigrun({mcc.events[0]}, function(_, m)
        print (m.spid, m.opcode, m.data)
    end)

    sched.run(function()
        for i=1, 100 do
            mcc.send(mcc.ADMIN, mcc.PING,'data'..i)
            sched.sleep(1)
        end
    end)
end
return M
```

Capa Servicio: Toribio

Demo

Capa Aplicación: IFlow

- Solamente un ejemplo de lo distinto que se puede pensar.
- Programación no imperativa, vagamente funcional: *Dataflow Programming*.
- En vez de especificar el flujo del programa, se especifica el flujo de los datos.
- Se adapta bien (?) a programar de forma reactiva.

Capa Aplicación: IFlow

- Un programa es una colección de fuentes de datos, filtros y destinos de datos.
- Los datos “fluyen” de los origenes a los destinos y son transformados en los filtros.
- Sintaxis de un proceso:

[parámetros de entrada] > filtro > [parámetros de salida]

Capa Aplicación: IFlow

- Parámetros:
  - Constantes (numéricas: `1.5`, strings: `'texto'`, booleanos: `true`)
  - Etiquetas: permiten interconectar filtros. (Las salidas solo pueden ser etiquetas)
- Filtros:
  - Funciones que transforman la información.
  - Son invocadas cuando llega un dato.
  - Pueden generar salida de forma asincrónica:
    - No están obligadas a emitir como respuesta a una entrada.
    - Pueden emitir en cualquier momento, aunque no hayan recibido datos.
  - Pueden ser invocadas de una biblioteca (p.ej. `sum`, `max`, `min`, `threshold`, `timer`...), o ser definidas en linea.

Capa Aplicación: IFlow

- Ejemplo: reloj.

```
1 > timer > x
x > function(s) return s%60 end > sec
x > function(s) if s%60==0 then return s/60 end end > min
'seconds', sec > print >
'minuntes', min > print >
```

- Ejemplo: consumir eventos de Toribio.

```
'mice:/dev/input/mice', 'leftbutton' > toribio_event > click
'mice:/dev/input/mice', 'move' > toribio_event > x, y
click,x,y > function(a,b,c) if a then return b,c end > xc,yc
'click',xc,yc > print >
```

Capa Aplicación: IFlow

- Ejemplo: interactuar con usb4all

```
'gris:1', 0.1 > sensor_poll > gris_izq
'gris:2', 0.1 > sensor_poll > gris_der
gris_izq, '>', 50 > threshold > avanzar_izq
gris_der, '>', 50 > threshold > avanzar_der
avanzar_izq, avanzar_der > motores >
```

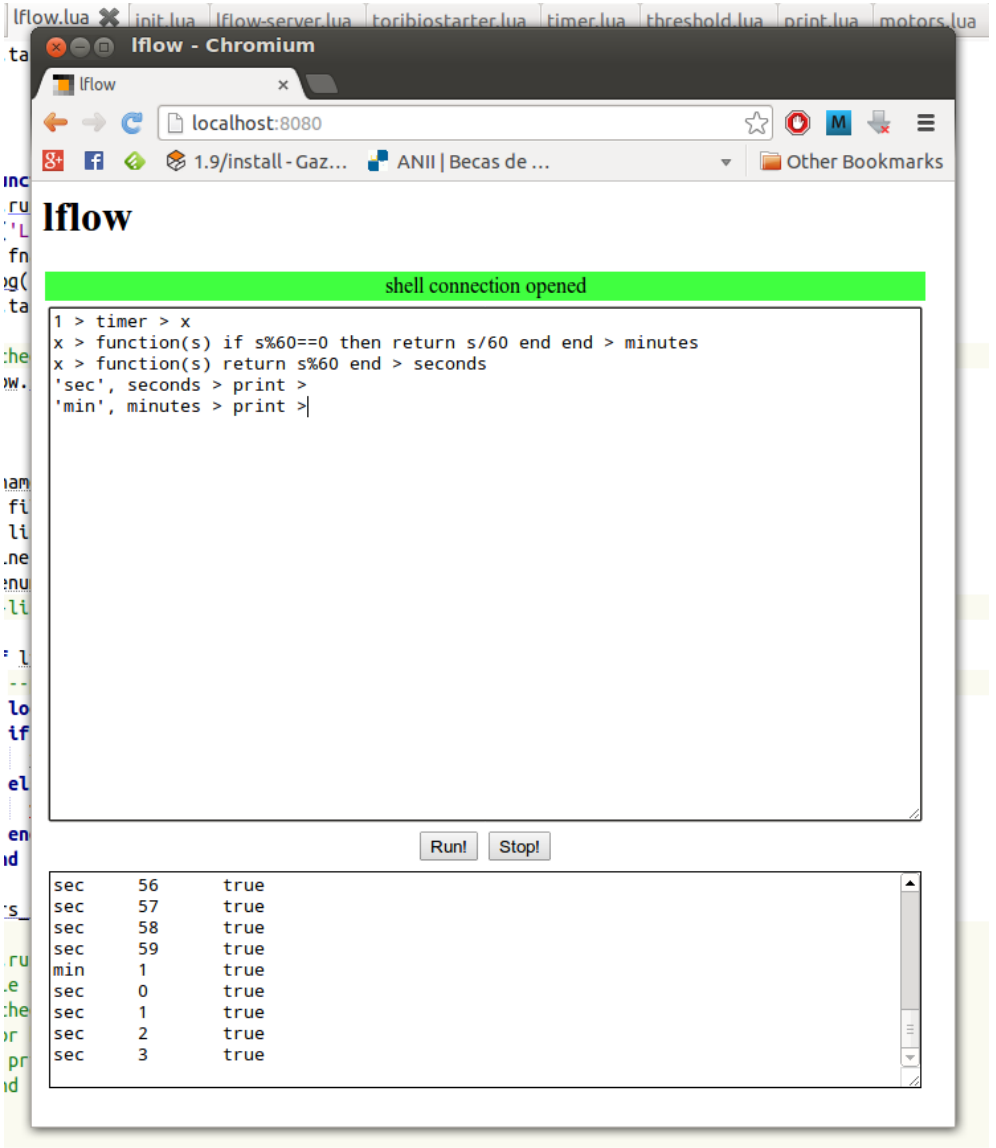
- La combinación sensor_poll + threshold se puede reemplazar con eventos desde MCC:

```
'gris:1', '>', 50 > sensor_watch > avanzar_izq
# o tal vez...
'gris:1', 50 > sensor_greater > avanzar_izq
```

Capa Aplicación: IFlow

- Toribio incorpora un servidor http, con soporte para WebSockets...
- ...todo lo necesario para implementar interfaces de usuario usando HTML5 / javascript
- Ejemplos de IDEs de ese estilo:
  - WebIDE de Raspberry Pi
  - [proyecto de grado, sobre Toribio]
- ¿Cómo hacerlo gráfico?

Capa Aplicación: IFlow



Referencias

- MCC: https://github.com/xopxe/MCC.git
- Toribio: https://github.com/xopxe/Toribio.git
- lflow: https://github.com/xopxe/lflow.git