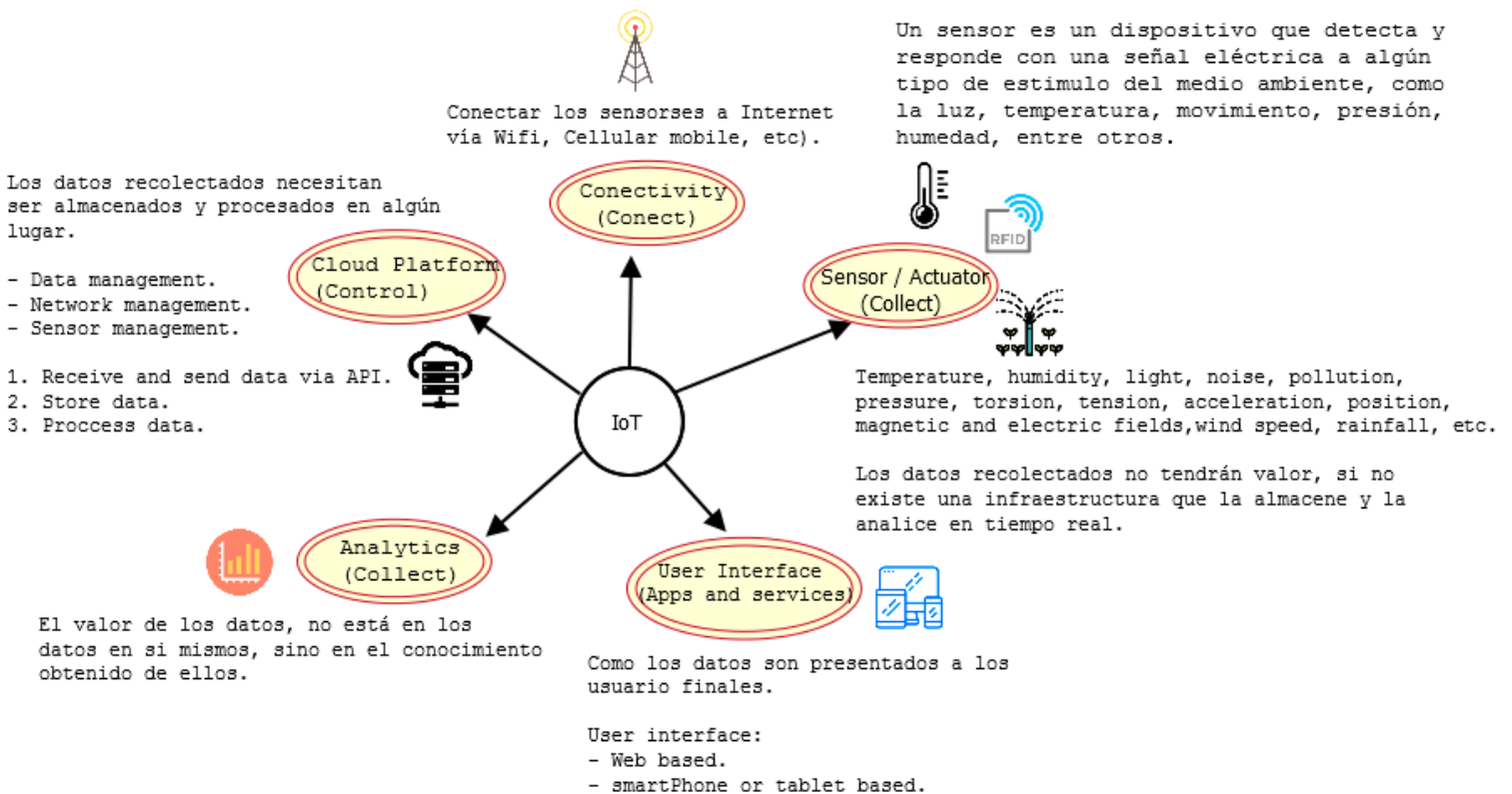


Introducción.

Una de las aplicaciones con más desarrollo en el entorno de node.js es el control de hardware a través de la web (browser o aplicación móvil). Para iniciar nuestra práctica en este interesante tema, vamos a desarrollar la siguiente aplicación que manejará un sensor de temperatura - humedad de la serie HTC11 (desde un Arduino Mega), cuyos datos serán gestionados y transferidos por un servidor basado en node.js. El usuario, a través de una plataforma de visualización de datos en la nube, podrá hacer seguimiento a la medición de temperatura y humedad del sensor en tiempo real, la cual puede ser usada por varios dispositivos simultáneamente.

Bloques de una aplicación IoT

A continuación, se observan los componentes básicos de una aplicación IoT. Para este primer desarrollo, trabajaremos principalmente en las etapas de sensores, conectividad y plataformas en la nube.



Objetivo.

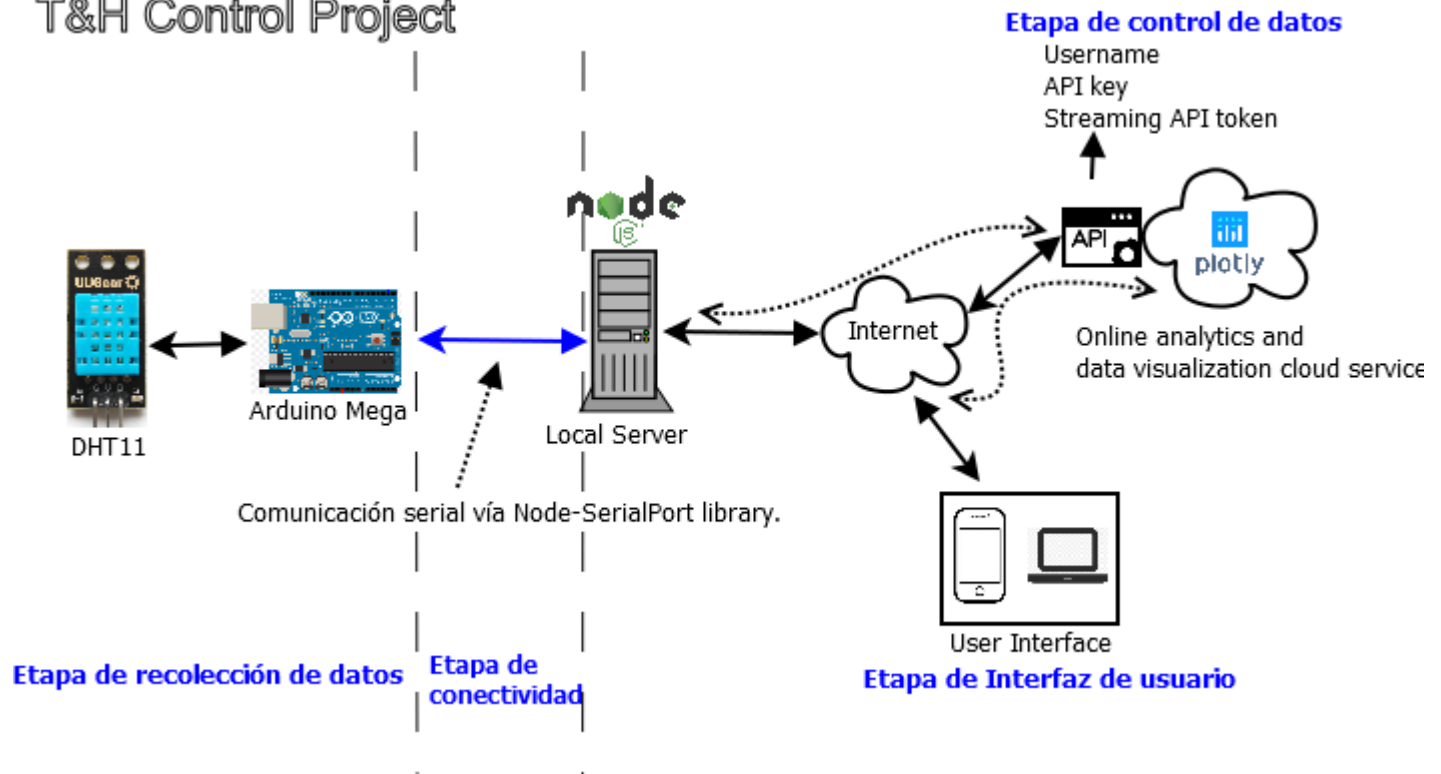
Diseñar y desarrollar una aplicación básica IoT que nos sirva de base para estudiar su arquitectura en general.

Requerimientos:

- Sensor de temperatura y humedad DHT11
- Arduino Mega
- Node v8.6.0
- Arduino IDE 1.8.4
- Plotly node module versión 1.0.6
- SerialPort node module versión 6.0.4

Arquitectura:

T&H Control Project



Usaremos node.js como la plataforma para nuestra etapa de recolección los datos desde el arduino. Node.js es una plataforma de desarrollo de software construido sobre Chrome's V8 JavaScript engine. Node.js utiliza un paradigma orientado a eventos, usando un modelo de entradas y salidas (I/O) no – bloqueante y asíncrono. Estas características lo hacen eficiente y ligero a la hora de diseñar aplicaciones en tiempo real.

Etapa de adquisición / recolección de datos

Esta etapa consiste de un sensor de temperatura y humedad DHT11 y una placa arduino Mega 2560, la cual será la encargada de centralizar y capturar los datos de estos sensores, los cuales posteriormente se enviarán al servidor node local.

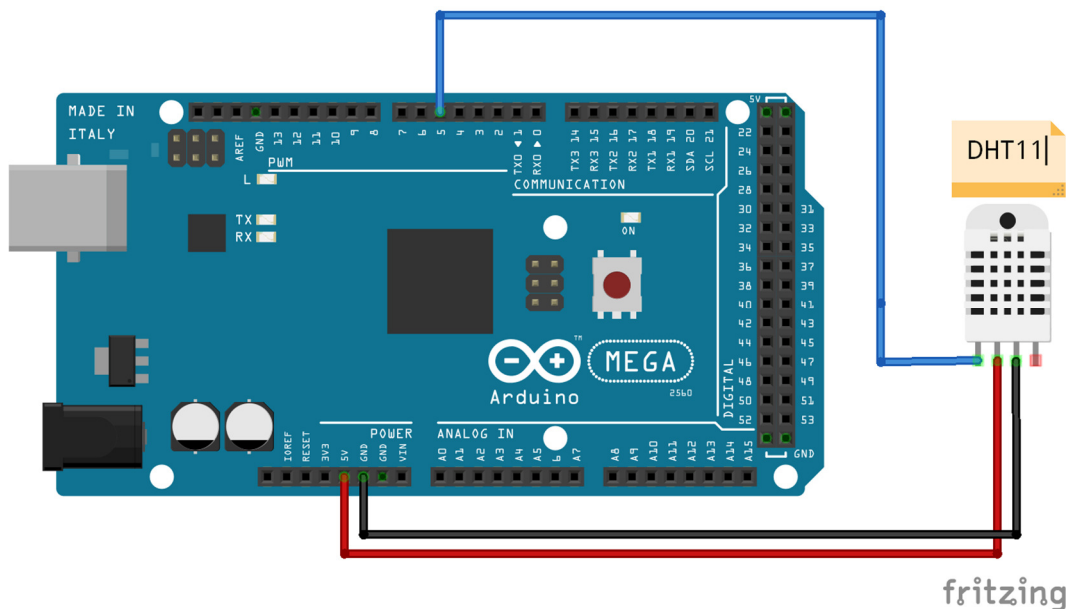
Sensor de temperatura y humedad DHT11.

El DHT11 y el DHT22 son dos sensores que miden simultáneamente temperatura y humedad relativa. De los dos el DHT22 cuenta con mejores características en cuanto a precisión y rango de medición, sin embargo, usaremos el DHT11 por ser un desarrollo de pruebas.

Características principales del DHT11:

- Temperatura: Medición de rango entre 0 a 50°C con una tolerancia de ± 2.0 °C.
- Humedad: Rango de medición entre 20 a 80% con una tolerancia del $\pm 5\%$.
- Frecuencia de muestreo: 1 Hz (1 muestra por segundo).

Como desventaja podemos citar que estos sensores tienen una baja velocidad de lectura y un tiempo de espera entre lecturas de por lo menos 2 segundos. (Aunque no es tan crítico debido a que las condiciones de temperatura y humedad no varían con mucha rapidez en el tiempo). El encapsulado del DHT11 cuenta con una resistencia de Pull-Up entre 4.7K y 10k (previene estados falsos sobre el pin de datos cuando este está en estado de reposo, durante los tiempos de espera entre lecturas) y en algunos casos con un condensador de filtrado entre Vcc y tierra.



Algunas recomendaciones y usos adicionales los pueden encontrar en:

<http://www.microprocesadores.unam.mx/assets/documentohumedadtemperatura2-1.pdf>

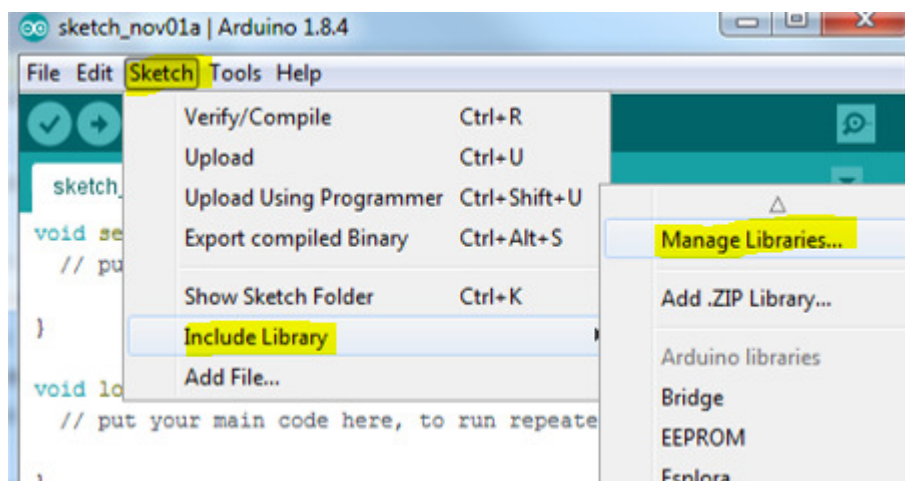
La transmisión de datos desde el sensor DHT11 hacia la placa Arduino.

El DHT11 es originalmente un dispositivo analógico, sin embargo, tiene la capacidad de realizar la conversión de estas señales analógicas a digitales y de esta manera es posible la comunicación con la placa en forma directa. El DHT11 tiene 3 pines que representan la alimentación (de 3.5 a 5V), tierra y la salida de los datos digitales. La trama de estos datos es de 40 bits, 32 de los cuales corresponden a los valores medidos de la temperatura y humedad y 8 bits de paridad para comprobar si hay errores en la trama que recibe la placa.

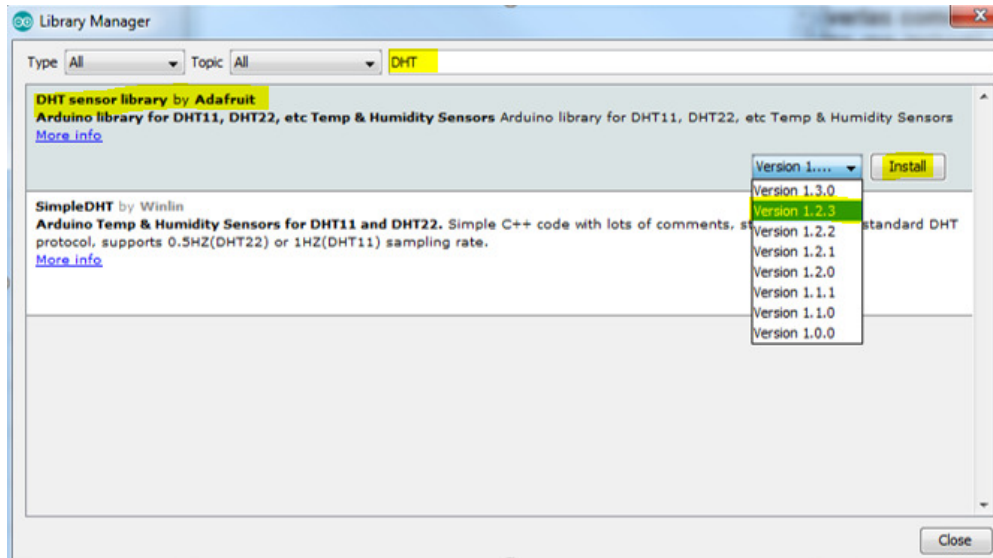
Humedad (8 bits)	Humedad (8 bits)		Bits de paridad
10110001	01011000	11110010	10001010
			01101111
		Temperatura (8 bits)	Temperatura (8 bits)

Para la programación del sensor desde el IDE de Arduino, vamos a trabajar con la librería “DHT-sensor-library” (ver: <https://github.com/adafruit/DHT-sensor-library>). Esta librería nos proporciona una serie de funciones que podemos usar para hacer las lecturas del sensor mucho más sencillas. Para cargar la librería y tenerla disponible en el IDE, seguiremos los siguientes pasos:

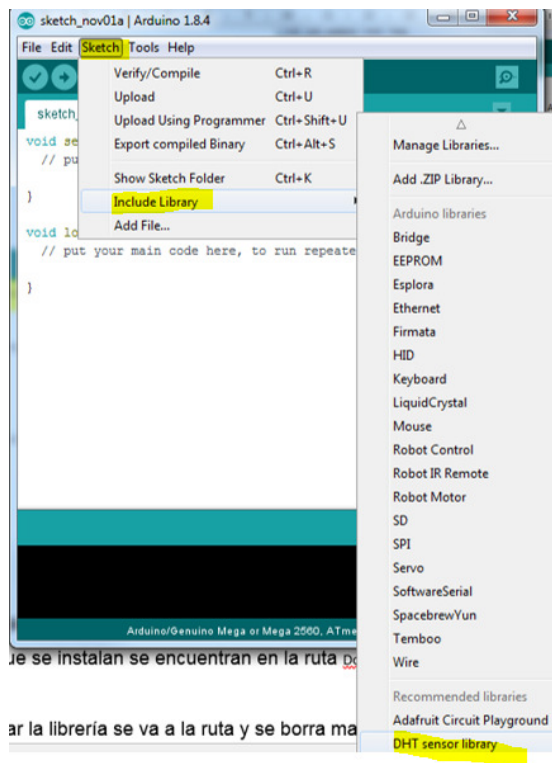
1. Abrir el IDE de Arduino y usar el gestor de librerías (repositorio oficial de Arduino):



2. Buscamos la librería que necesitamos y procedemos a instalarla:



3. Añadimos la librería a nuestro sketch principal:



```
sketch_nov01a $  
#include <DHT.h>  
#include <DHT_U.h>  
  
void setup() {  
    // put your setup code here, to run once:  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```

- las librerías que vienen por defecto están en la ruta **C:\Program Files (x86)\Arduino\libraries**.
- Las librerías que se cargan, se encuentran en la ruta **Documents\Arduino\libraries\"nombre_de_la_librería"**
- Para desinstalar la librería se va a la anterior ruta y se borra manualmente la carpeta que la contiene.
- Si la librería se aloja en otra carpeta, entonces debemos indicarla en `#include <ruta\library.h>`

Es importante tener en cuenta el tipo y el tamaño de las librerías que vamos a cargar en nuestro sketch. Se debe verificar que el tamaño de la librería no comprometa la cantidad memoria flash de la placa. Adicionalmente, es importante realizar un chequeo al rendimiento de la librería, esto es por ejemplo si cargamos una librería que realiza múltiples operaciones matemáticas, de la cual solo usaremos una o dos, entonces sería recomendable ingresar al código e intentar optimizar o eliminar las partes del código que no usaremos o en su defecto copiar las operaciones que necesitemos de la librería, directamente en nuestro sketch principal.

Etapas de conectividad.

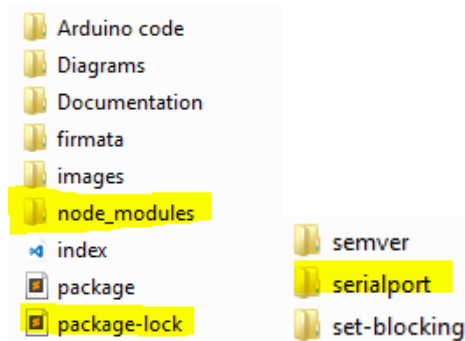
Para realizar la conexión entre la placa Arduino y el servidor local Node.js usaremos la librería estándar de Arduino “Serial Library” del lado de la placa. Del lado del server podemos usar la librería “Node-SerialPort”, para instalarla, adicionamos esta dependencia al archivo package.json:

```
{
  "name": "Arduino-node",
  "version": "1.0.0",
  "description": "DHT11 project with Arduino-node.js",
  "license": "MIT",
  "dependencies": {
    "plotly": "^1.0.6",
    "serialport": "latest"
  },
  "devDependencies": {}
}
```

Y desde la consola de node, ejecutamos:

```
npm install --save
```

A partir de la versión 5 de NPM, al momento de realizar la instalación de los módulos, NPM crea un fichero adicional llamado package-lock.json, el cual se explica en detalle en <https://elabismodenuell.wordpress.com/2017/07/07/el-fichero-package-lock-json-tengo-que-versionarlo/>



En la carpeta node-modules se cargan todos los modulos estándar de node, más las dependencias incluidas en el fichero package.json. Dentro de esta podemos ver nuestro módulo serialport. Este módulo le permite a Node acceder al hardware del puerto serial de dispositivos externos. Tiene métodos útiles y sencillos de usar, siendo “read,” “write,” “open,” “close” los usados principalmente. Adicionalmente incluye un conjunto de “parsers” que ayudan a gestionar el flujo de datos recibido.

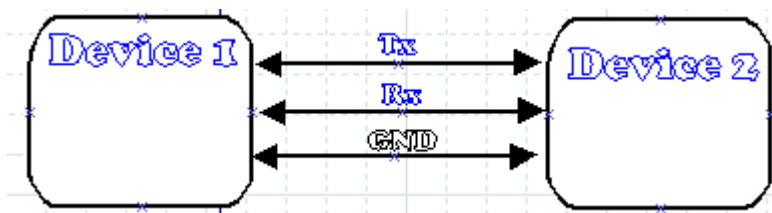
Nota: Es importante verificar la versión de los módulos instalados y ver la documentación para esa versión en específico, ya que algunos métodos y funcionalidades pueden cambiar de una a otra:

```
"serialport": {
  "version": "6.0.4",
  "resolved": "https://registry.npmjs.org/serialport/-/serialport-6.0.4.tgz",
  "integrity": "sha512-ohiyBppkw0rRbd7CksNSsH8kTlx5Fdh1TRL0",
  "requires": {
    "bindings": "1.3.0",
    "commander": "2.11.0",
    "debug": "3.1.0",
    "nan": "2.7.0",
    "prebuild-install": "2.3.0",
    "promirepl": "1.0.1",
    "safe-buffer": "5.1.1"
  }
}
```

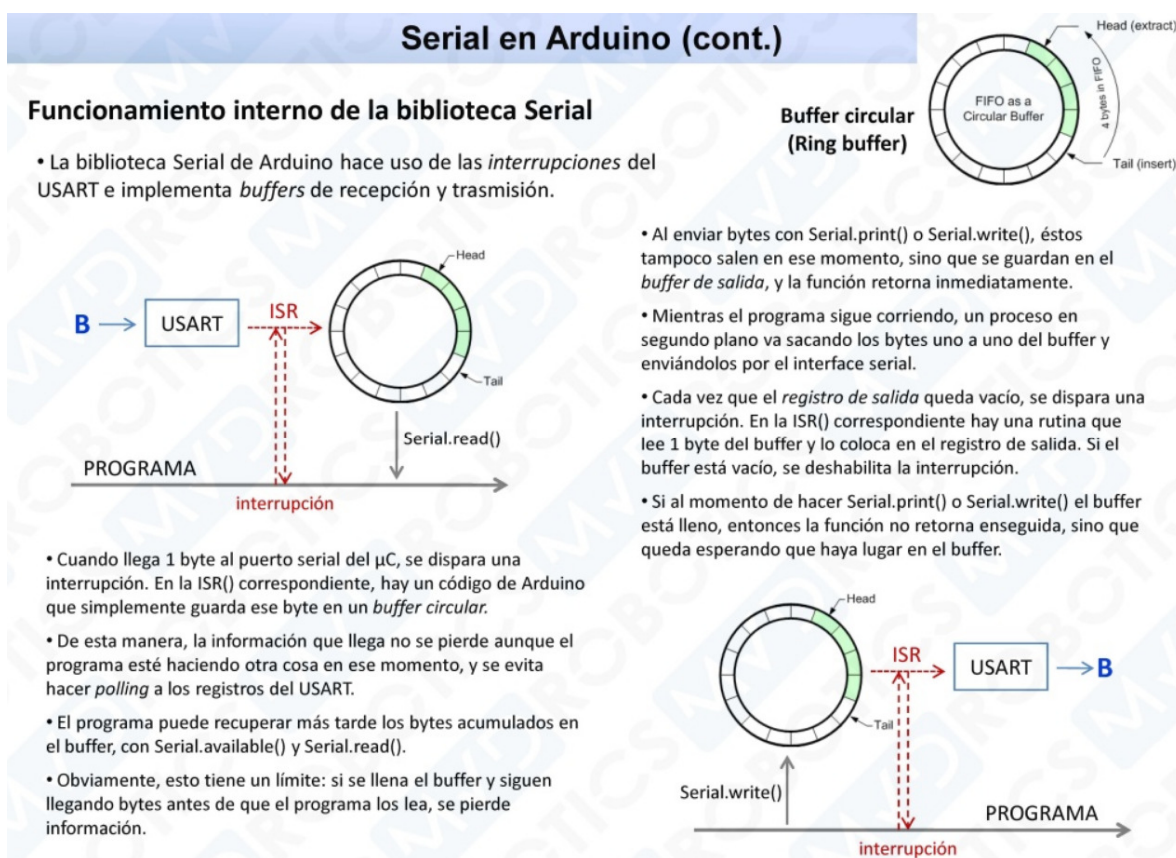
```
"plotly": {
  "version": "1.0.6",
  "resolved": "https://registry.npmjs.org/plotly/-/plotly-1.0.6.tgz",
  "integrity": "sha1-smcsPfiDM2Jb32hJgtDj9lIdeqo=",
  "requires": {
    "mkdirp": "0.5.1"
  }
}
```


UART (Universal Asynchronous Receiver-Transmitter)

UART (Universal Asynchronous Receiver/Transmitter) es un conjunto de hardware y protocolos (por lo general es un sistema embebido en un microcontrolador μC) responsables de implementar y controlar la comunicación serial entre la placa arduino y el exterior. Dentro de sus funciones se encuentran el manejo de las interrupciones involucradas en la Tx/Rx serial, convertir los datos paralelos provenientes del sistema a datos seriales para su posterior transmisión y viceversa, también pueden contar con un buffer de Tx/Rx, en donde los datos pueden permanecer hasta que el μC los lea o los transmita.



Para el manejo de la comunicación serial, Arduino cuenta con una librería estándar, la cual hace uso del objeto "Serial" con sus respectivos métodos, tales como `begin()`, `read()`, `write()`, `print()`, `available()`, `flush()`, entre otros. A continuación se observa una muy buena explicación del funcionamiento interno de la librería serial de la placa Arduino, preparada por Pablo Gindel. Para ver la presentación completa, puede remitirse a la siguiente url: <https://es.slideshare.net/pablogindel/microcontroladores-4-comunicacin-uart>



Para la transmisión de información serial a través de la placa Arduino, usamos el método `Serial.println(val, format)`, el cual envía a través del puerto serial un valor "val" de cualquier tipo de dato (int, char, etc.) seguido de un retorno de carro '\r' y un salto de línea '\n'. Opcionalmente con "format" indicamos en que base será enviado el valor (BIN, HEX, OCT, BYTE, DEC por defecto). Si el valor a enviar es de tipo float, este parámetro representara en número de decimales, después de la coma, que serán enviados (por defecto son dos).

En vez de esperar a que los bytes de datos hayan sido transmitidos, `Serial.println(val, format)` escribe estos en un buffer de memoria, los cuales son transmitidos en segundo plano un carácter a la vez de forma asíncrona, involucrando a las interrupciones (UDRE - USART, Data Register Empty y TXCn, Tx-Complete). Este método permite que el sketch principal continúe ejecutándose sin bloqueos.

Este protocolo es orientado a bytes, lo que quiere decir que cualquier cosa enviada, tiene que ser convertida primero a uno o más bytes, para luego ser enviada secuencialmente. El método **Serial.println(val, format)** y sus variantes, realizan primero la conversión apropiada de “val” al formato definido en “format”, para luego transmitir los bytes individuales en formato Ascii.

Serial.write(58) → “00111010”

Serial.print(58, DEC) → 5356 (53 y 56 corresponden en Ascii “5” y “8” respectivamente).

Serial.print(58, DEC) → 53561310 (1310 corresponden en Ascii “\r” y “\n” respectivamente).

Como lo mencionamos anteriormente, UART es de tipo asíncrono, esto quiere decir que los dispositivos involucrados en la comunicación serie no se sincronizan previamente antes de la transmisión. Entonces es posible que el transmisor (Tx) intente enviar datos cuando el receptor (Rx) no está en modo de escucha y viceversa. En estos casos, Tx y Rx mantienen los datos en un buffer, un espacio en memoria donde los datos entrantes son almacenados antes de que estos sean leídos por el receptor. Es importante mantener un control sobre el contenido de estos buffers ya que, en muchas implementaciones, aunque el receptor no esté en modo de escucha, el buffer puede haber almacenado datos enviados desde el transmisor en una ejecución anterior, y al momento de iniciar la lectura, puede obtener datos erróneos.

Para nuestro proyecto, anteriormente mencionábamos que podíamos obtener los datos de humedad, temperatura e índice de calor, mediante la librería DHT-sensor-library de Arduino. Estos datos son enviados de forma serial hacia el servidor de Node, mediante **Serial.println()** codificado en ASCII en formato decimal, con una terminación de línea para cada variable:

Humedad: 49,32\r\n (dato enviado en Ascii) → 52574651501310

Temperatura: 23,00\r\n (dato enviado en Ascii) → 50514648481310

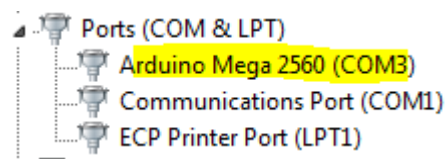
Índice de calor: 22,66\r\n (dato enviado en Ascii) → 50504654541310

La placa Arduino está continuamente enviando este flujo de datos y el servidor Node, mediante **transformation stream**, se mantiene en espera hasta que un carácter de final de línea '\r\n' es recibido, entonces realiza una lectura completa de su buffer.

El primer paso para establecer la comunicación, es inicializar y abrir el puerto serial en cada uno de los dispositivos:

Paso 1: Configurar el módulo SerialPort. Nombre del puerto serial que será usado en el servidor de Node.

Para conocer el puerto serial usado por la placa Arduino, en Windows podemos verificarlo en el administrador de dispositivos:



Y lo incluimos en el index.js, mediante:

```
var serialPort = require('serialport');
const portName = 'COM3';
var port = new serialPort(portName, {
  baudRate: 9600
});
```

O pasarlo como argumento directamente desde la consola de node:

```
node index.js COM3
```

```
var SerialPort = require('serialport');
//const portName = 'COM3';
const portName = process.argv[2];
var port = new SerialPort(portName, {
  baudRate: 9600
});
```

O podemos verificarlo a través del método `list()` disponible en la librería `SerialPort`. Este método entrega una lista de puertos seriales disponibles con metadatos de los mismos.

```
SerialPort.list(function (err, ports) {
  ports.forEach(function(port) {
    console.log("comName: "+port.comName);
    console.log("pnpId: "+port.pnpId);
    console.log("manufacturer: "+port.manufacturer);
  });
});
```

```
comName: COM1
pnpId: ACPI\PNP0501\1
manufacturer: <Standard port types>
comName: COM3
pnpId: USB\VID_2341&PID_0042\5563930363435191B0C0
manufacturer: Arduino LLC <www.arduino.cc>
```

Después de tener identificado el puerto, procedemos a crear y abrir el puerto serial, mediante:

```
'use strict'
const SerialPort = require('serialport');
const portName = 'COM3'; //path

var options = {
  // autoOpen: false,
  baudRate: 9600,
  dataBits: 8,
  stopBits: 1,
  parity: 'none'
};

var port = new SerialPort(portName, options);
```

Paso 2: Abrir el puerto serial del módulo `SerialPort`.

Con `new SerialPort(path, [options], [openCallback])` creamos una nueva instancia del módulo `SerialPort`. “portName” es el nombre del puerto serial identificado anteriormente (COM3) y mediante el objeto `options`, realizo la configuración de los parámetros típicos de un puerto serial. Al momento de crear un nuevo puerto, este por defecto se abre automáticamente, si deseamos abrirlo manualmente, debemos configurar la opción `autoOpen` en `false` (por defecto esta en `true`). `OpenCallback` es llamado una vez la conexión esté abierta (Este callback no será llamado si la opción `autoOpen` es puesta en `false`). En este caso no lo vamos a usar y en su lugar, si llega a generarse un error en la apertura del puerto, usaremos el método `error.on()` para identificarlo.

La librería “serialport” está basada en eventos, por ejemplo, cuando se realiza una conexión, se reciben datos a través de la conexión o se abren o cierran los puertos, etc, estas acciones producen eventos, los cuales, generan el llamado de una función de callback para manejar una respuesta ante su aparición. Con `serialport.on('event', callback())` le indicamos a node que al producirse el evento 'event', llame a su respectiva función de callback.

Por ejemplo con `port.on('open', callback())` cuando un puerto serial es abierto, se emitirá el evento 'open' y se ejecutará su correspondiente función de callback(), la cual para nuestro caso, muestra por consola el mensaje:

Port opened: COM3 y baud rate: 9600

```
port.on('open', onOpen);

function onOpen() {
  console.log('Port opened:\t', port.path);
  console.log('baud rate: ' + port.baudRate);
}
```

Paso 3: Parser el stream recibido por el módulo SerialPort.

Transformation stream en NodeJS

El término '*transform stream*', en node, se refiere a todos los procesos que le permiten a un flujo de datos (stream) ser transformado, mediante bloques intermedios antes de llegar a su destino final, por ejemplo, comprimir/descomprimir o encriptar/desencriptar un flujo de datos. Para mayor información, pueden dirigirse a las siguientes fuentes:

Stream: <https://www.codementor.io/davidgatti/understanding-streams-in-node-js-weupiug0a>

Readable and writable stream: <https://es.slideshare.net/kushallikhi/streams-in-node-js?ref=>

<https://community.risingstack.com/the-definitive-guide-to-object-streams-in-node-js/> y

<https://medium.freecodecamp.org/node-js-streams-everything-you-need-to-know-c9141306be93>



Ver: <http://codewinds.com/blog/2013-08-20-nodejs-transform-streams.html>

SerialPort.parsers es un conjunto de Transform streams de la librería "SerialPort" de node, que analizan los datos entrantes de diferentes maneras para transformarlos. Por ejemplo la propiedad "ReadLine" es una Transform stream que lee el flujo de datos desde el buffer y cuando encuentra un salto de línea (en este caso cuando recibe la combinación de los caracteres '\r\n') lo transforma en un string de datos.

Pipe (en sistemas), se refiere a una técnica usada para pasar información desde un proceso a otro. En Node.js usamos pipes dentro del código, para pasar el resultado de una función a la otra. "ReadLine" usa el método "pipe" para "canalizar" la fuente de datos origen de un extremo a otro. En este caso la fuente de datos proviene de "port" y el método pipe canaliza los datos hacia el parser (Readline).

```

'use strict'
const SerialPort = require('serialport'); // incluimos la libreria Node-SerialPort
const parsers = SerialPort.parsers; /*Parser: The default Parsers are Transform streams
that parse data in different ways to transform
incoming data.*/

const portName = 'COM3'; //path

const parser = new parsers.Readline({
  delimiter: '\r\n' //Use a '\r\n' as a line terminator
});

var options = {
  baudRate: 9600,
  dataBits: 8,
  stopBits: 1,
  parity: 'none'
};

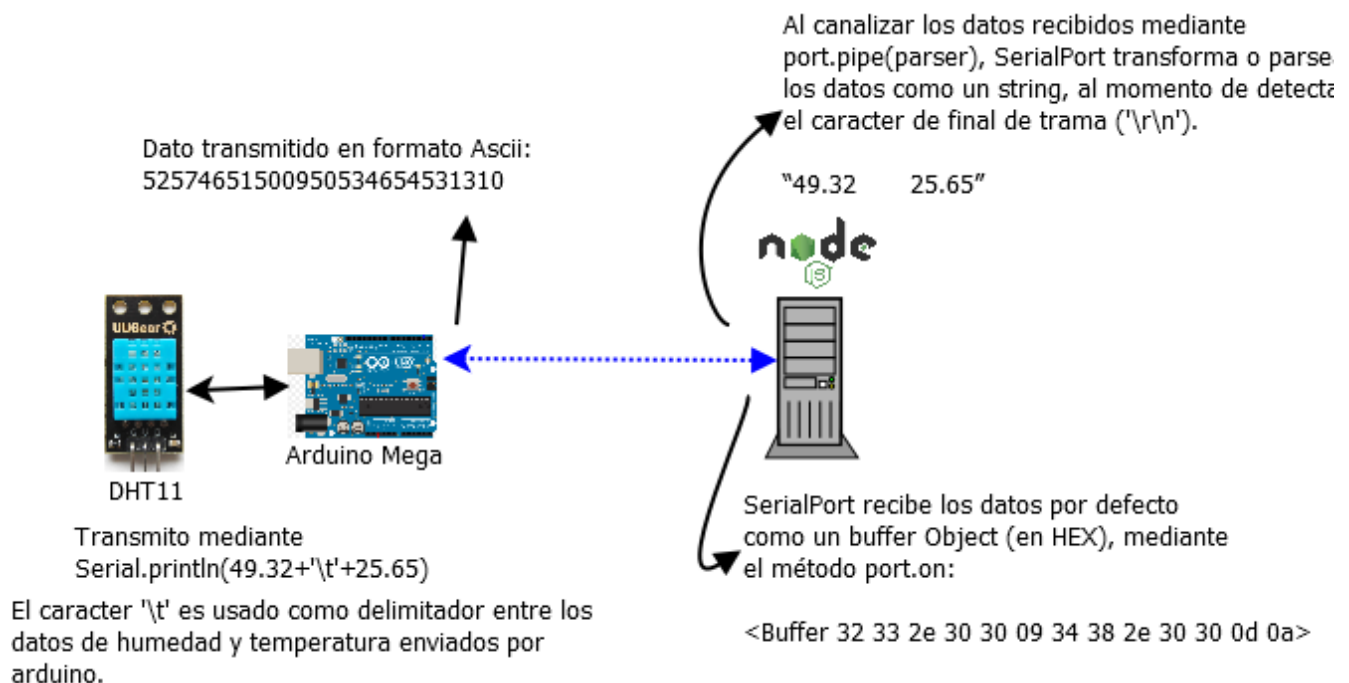
var port = new SerialPort(portName, options);
port.pipe(parser);

/* ***** */
port.on('open', function () {
  console.log('Port opened:\t', port.path);
});
/* ***** */
/* Read data when be emitted as an data event */
parser.on('data', function (data){
  console.log('Received:\t', data); // get a buffer of data from the serial port
});
/* ***** */
/* Open errors will be emitted as an error event */
port.on('error', function (err){
  console.log('Error: ', err.message);
});
/* ***** */

```

Los datos recibidos en **port** son canalizados hacia el proceso **parser**. Cuando hay datos disponibles en el buffer, se dispara el evento “data” desde el método **parser.on** el cual llama al callback indicando el dato que fue recibido.

Si no se canaliza el stream a través del método **parser**, **SerialPort** recibe el stream por defecto como un Buffer object¹ (en HEX), usando el método **port.on** directamente, esto es:



¹ <https://docs.nodejitsu.com/articles/advanced/buffers/how-to-use-buffers/>

Con el método `port.flush(callback())` limpiamos cualquier dato antiguo de los buffers antes de leer o escribir un nuevo dato. Datos que fueron recibidos pero no leídos o escritos pero no enviados.

```
function onOpen() {
  console.log('Port opened: ' + port.path + ', ' + 'baud rate: ' + port.baudRate);
  console.log("flushing...");
  port.flush(function(err){
    if(err){console.log("Flush error: ",err)};
    //flush discard data received but not read, and written but not transmitted.
  });
  // onWrite(delay);
}
```

Paso 4: Transmitiendo datos desde el servidor node a la placa arduino.

Para escribir datos, usamos el método `serialPort.write(data, [encoding], [callback])`

Data es el dato que se transmitirá, el cual puede ser del tipo string, array o un Buffer object. **Encoding** por defecto es utf-8. También acepta 'ascii', 'base64', 'binary', 'hex'. La función de callback es llamada una vez la operación de escritura finaliza.

```
function onWrite(delay){
  setTimeout(function() {
    console.log("waiting...");
    command = command + '#'
    port.write(command, function(err) {
      if (err) {
        console.log('Error on write: ', err.message);
      }
      console.log('Sending...');
    });
  }, delay);
  port.drain(function(err){
    if(err){console.log("Drain error: ",err)};
    // .drain will wait for the port to open and the write to finish.
    // If you wish to ensure the contents of the transmit queue are sent to the port.
  });
}
```

Algunos dispositivos como Arduino, hacen un reset cuando se abre una conexión sobre ellos. En estos casos, realizar inmediatamente una operación de escritura, causará una pérdida de datos debido a que estos no están listos aún para recibirlos. Para mitigar esto, usamos el método `setTimeout` que espera un tiempo definido en `delay` para ejecutar la operación de escritura.

En algunas ocasiones cuando la operación de escritura finaliza y `serialPort.write` retorna, es posible que algún dato permanezca aun en el buffer de transmisión sin enviar, para asegurarnos de que todos los datos sean transmitidos por el puerto serial usamos `serialPort.drain([callback])`.

Del lado de la placa Arduino, usamos el comando `Serial.available()` para iniciar el proceso de lectura de datos. Este comando retorna el número de bytes disponibles para ser leídos desde el buffer. Si hay algún dato disponible, estos pueden ser leídos mediante `Serial.read()` caracter por caracter.

```

void loop() {
  if (Serial.available() > 0) {
    incomingByte = Serial.read();

    if ((char)incomingByte != stopByte) {          // stopByte: '\n' or '#' or '\r\n'
      bufferIN[byteCount++] = (char)incomingByte;  //assemble the bytes into bufferIN
    }
    else {
      bufferIN[byteCount] = '\0'; // null to indicate the string's final.
      byteCount = 0;             // reset counter for new message.
      Serial.println(bufferIN); //Send back the message.
    }
  }
}

```

Dentro de la sentencia if de **Serial.Available()** se interpretarán los bytes que ingresan a la placa. Primero leemos el primer byte que ingresó desde el buffer FIFO (**First-in, First-out buffer**) de lectura, esto indica que el primer byte enviado desde el servidor, es el primer byte que será leído por Arduino, mediante **Serial.read()**. Estos bytes deben ser concatenados en un string para armar el dato completo, para esto, mientras no recibamos un carácter de final de línea (que puede ser \n ó \r\n ó #) el cual indica que llegamos al final de mensaje, se concatenará el dato, de lo contrario posicionamos el carácter null para indicar el final del mensaje y detener la concatenación.

Nota: El dispositivo que transmite los datos deberá incluir al final del mensaje un carácter de final de línea para que el receptor pueda reemsamblar el mensaje completamente.

Etapas de Application domain of IoT Cloud Platforms.

La nube es una base fundamental para IoT, esta puede ofrecer muchos servicios específicos dependiendo del área de aplicación, permitiendo gestionar, analizar, monitorear o visualizar datos, desde sensores o actuadores conectados a esta. El profesor **Partha Pratim Ray** de la universidad de Sikkim (India) escribió un interesante artículo sobre las diferentes áreas de aplicación de las plataformas en la nube para IoT, las cuales se observan a continuación:

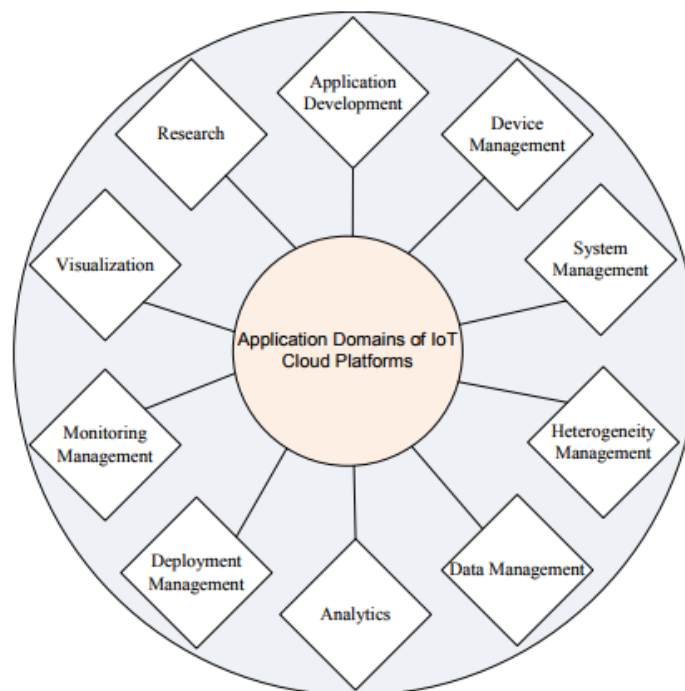
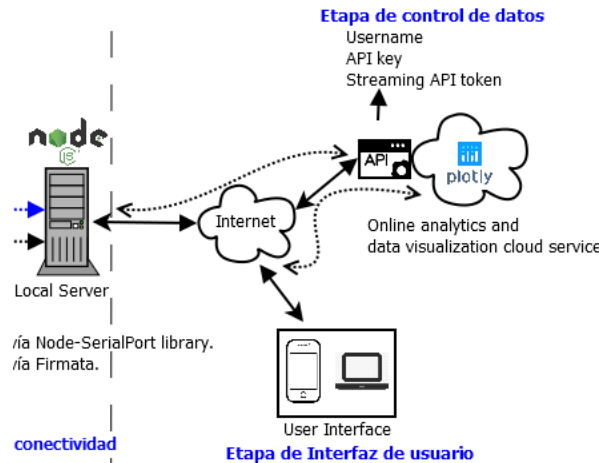


Fig. 1. Application domains of IoT cloud platforms.

<http://www.sciencedirect.com/science/article/pii/S2314728816300149>

Dentro de las plataformas de IoT en la nube para visualización de datos descritas en el documento, encontramos **Plotly** (<https://plot.ly>). Plotly es una herramienta de visualización y análisis de datos online basada en la nube, la cual hace uso de una API² para esta tarea.



El primer paso para usar esta librería es crear una cuenta desde <https://plot.ly> y desde la configuración de la cuenta, tomamos los siguientes datos:

API Settings

Username

The same as your Plotly username.

API Key

Note that generating a new API key will require changes to your `~/.plotly/.credentials` file.

Regenerate Key

Streaming API Tokens

Use one streaming token per data-stream. Check out the [documentation](#) to learn more about the Plotly streaming API.

You don't have any streaming tokens yet.

Add A New Token

² Application Programming Interface (API), la cual permite el intercambio de información o datos entre dos aplicaciones, permitiéndole a uno acceder a las funciones o servicios del otro, sin dejar de ser independientes. Plotly usa una API REST (Representational State Transfer) la cual estructura APIs usando URLs, el protocolo HTTP y el formato de datos JSON, esto le permite a cualquier dispositivo o cliente, que entienda HTTP, usar el servicio. Sobre API REST, plotly construye una Streaming API, la cual puede ser usada para crear gráficos en tiempo real.

Ahora procedemos a instalar la librería en el servidor de nodejs:

```
npm install plotly --save
```

```
npm WARN Arduino-node@1.0.0 No repository field.  
+ plotly@1.0.6  
added 1 package in 6.257s
```

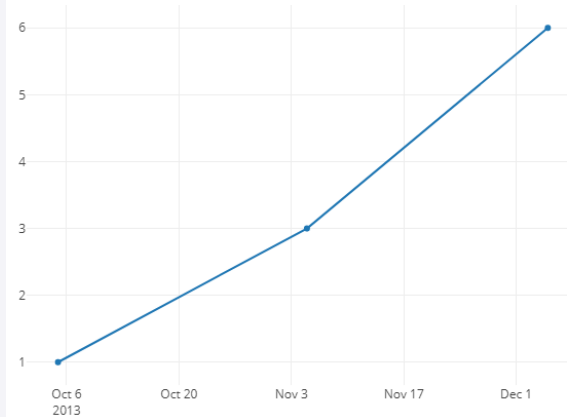
Desde node.js, a través de esta librería, se tomarán los datos medidos desde la placa Arduino para que estos sean visualizados en una gráfica.

Primero se llama la librería plotly desde node, con el username y el API key generados anteriormente:

```
const plotly = require('plotly')('Plotly_UserName', 'Plotly_API'),
```

Cada gráfica en plotly tiene objetos y propiedades que nos permiten configurar aspectos en su comportamiento y apariencia como el color, el tipo de gráfica, un label, etc.

```
require('plotly')(username, api_key);  
  
var data = [  
  {  
    x: ["2013-10-04 22:23:00", "2013-11-04 22:23:00", "2013-12-04 22:23:00"],  
    y: [1, 3, 6],  
    type: "scatter"  
  }  
];  
var graphOptions = {filename: "date-axes", fileopt: "overwrite"};  
plotly.plot(data, graphOptions, function (err, msg) {  
  console.log(msg);  
});
```



<https://plot.ly/nodejs/time-series/>

El objeto “**Data object**” contiene un array de objetos cada uno relacionado con información sobre los datos y estilo de la traza para las gráficas, entre otros. El objeto “**graphOptions**” contiene propiedades relacionadas con el estilo, como el color o el título, entre otras. Con **plotly.plot(“Data Object”, “graphOptions”, [callback])** inicializa y crea una nueva gráfica.

Data Object:

```
[{  
  "x": [],  
  "y": [],  
  "type": "scatter",  
  "mode": "markers",  
  "marker": { "color": " " },  
  "line": { "color": " " }  
}]
```

graphOptions Object:

```
{  
  "filename": "MQTT Stream",  
  "fileopt": "extend",  
  "layout": { "title": },  
  "world_readable": true  
}
```


Para graficar un streaming de datos, plotly ha incluido una capa adicional sobre su REST API para este fin. Entonces debemos seguir los mismos pasos para crear una gráfica básica, instanciando una gráfica base con el layout e incluyendo el “streaming token” en el objeto de datos:

Data Object:

```
[{
  "x": [],
  "y": [],
  "type": "scatter",
  "mode": "markers",
  "marker": { "color": " " },
  "line": { "color": " " }
  "stream": {
    "token": "xxxxyyxxxi",
    "maxpoints": 500
  }
}]
```

Por defecto, al graficar los datos en tiempo real, se muestran los 30 puntos (coordenadas x y y) más recientes a la vez. Para modificar este valor, usamos la propiedad **"maxpoints"**.

Este streaming es enviado desde node a plotly, mediante HTTP hasta su endpoint (<http://stream.plot.ly>) Para correlacionar el streaming de datos con el objeto de datos de plotly, se adjunta un token a la cabecera HTTP request desde node hacia el endpoint, este token permite identificar el streaming de origen (pueden existir varios streaming de datos sobre una misma gráfica, para cada uno se requiere un token diferente). Una vez que la conexión se establezca, se puede iniciar el streaming de datos mediante:

```
plotly.plot(data, graphOptions, (err, msg) => {
  if (err) return console.log(err);
  console.log(msg);

  var stream = plotly.stream('0x903xdc16', (err, res) => {
    console.log(err, res);
  });
  /* Read data when be emitted as an data event and then, run the callback function */
  parser.on('data', (input) => {

    if(isNaN(input) || input > 1023) return;
    var streamObject = JSON.stringify({ x : getDateString(), y : input });
    // con JSON.stringify convertimos un objeto estandar de JS al formato JSON
    console.log(streamObject);
    stream.write(streamObject+'\n');
  });
});
```

Con el método **plotly.stream()** creamos el proceso para transmitir el streaming de datos y con el método **stream.write**, escribimos el streaming en tiempo real hacia el endpoint adicionando un carácter de terminación de línea ('\n' el cual se usa para separar los strings JSON transmitidos).

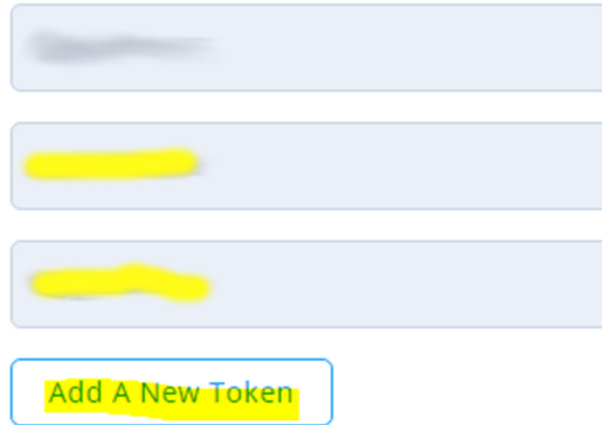
Nota:

- Es importante que estos objetos JSON sean escritos en intervalos de tiempo no mayores a 50ms, ya que podría resultar en una pérdida de datos.
- Es posible visualizar multiples streaming sobre la misma gráfica, anidando tokens diferentes dentro del correspondiente objeto de datos.
- Se deben enviar datos por lo menos cada minuto, de lo contrario plotly considerará el streaming obsoleto y este será descartado. Si el streaming se transmite a una tasa baja, es posible enviar un carácter de terminación de línea '\n', dentro de la ventana de 1 minuto para indicarle a plotly que el streaming continua activo y mantener así la conexión abierta.

Según lo mencionado anteriormente, es posible enviar multiples streamings de datos sobre la misma gráfica, anidando diferentes tokens dentro del correspondiente objeto de datos. Esto nos es útil ya que debemos tomar 2 streams al mismo tiempo (Humedad y Temperatura). Para esto, debemos crear primero un nuevo token desde la configuración de nuestra cuenta en plotly:

Streaming API Tokens

Use one streaming token per data-stream. Check out the [API Tokens](#) page for more information.



Con el fin de mantener un código más limpio y ordenado, creamos un archivo JSON, en el cual incluimos los datos básicos de configuración de la cuenta en plotly:

```
{
  "plotly_username" : "username",
  "plotly_api_key" : "Api_key",
  "plotly_tokens" : ["Token_1", "Token_2"]
}
```

Y accedemos a sus propiedades mediante:

```
const username = config['plotly_username'],
      apiKey = config['plotly_api_key'],
      tokens = config['plotly_tokens'];
```

Posteriormente creamos los objetos de datos, **layout** y el **graphOptions**:

```

var layout = {
  title: 'Temperature and Humidity',
  plot_bgcolor: 'rgba(200,255,0,0.1)',
  xaxis: {
    title: 'Time',
    showline: true,
    mirror: "ticks",
    autorange: true,
    gridcolor: "#E0E0E0",
    gridwidth: 2
  },
  yaxis: {
    title: 'Temperature (C)',
    showline: true,
    mirror: "ticks",
    autorange: true,
    gridcolor: "#E0E0E0",
    gridwidth: 2,
    domain: [0, 0.35]
  },
  xaxis2: {
    title: 'Time',
    showline: true,
    mirror: "ticks",
    anchor: "y2",
    autorange: true,
    gridcolor: "#E0E0E0",
    gridwidth: 2
  },
  yaxis2: {
    title: 'Humidity(%)',
    showline: true,
    mirror: "ticks",
    autorange: true,
    gridcolor: "#E0E0E0",
    gridwidth: 2,
    domain: [0.6, 1]
  }
};

var data = [
  {
    name: 'Temperature',
    type: "scatter",
    x: [],
    y: [],
    mode: 'lines+markers',
    stream: {
      token: tokens[0],
      maxpoints: 500
    }
  },
  {
    name: 'Humidity',
    type: "scatter",
    x: [],
    y: [],
    xaxis: "x2",
    yaxis: "y2",
    mode: 'lines+markers',
    stream: {
      token: tokens[1],
      maxpoints: 500
    }
  }
];

var graphOptions = {
  layout: layout,
  fileopt: "extend",
  filename: "iot-test"
};

```

Con `plotly.plot()` inicializamos la gráfica con los datos creados en los anteriores objetos y creamos los stream de datos.

```

plotly.plot(data, graphOptions, (err, msg) => {
  if (err) return console.log(err);
  console.log(msg);
  /*
   Con el método plotly.stream() creamos el proceso para transmitir el streaming de datos
   y con el método stream.write, escribimos el streaming en tiempo real hacia el endpoint.
  */

  var streams = { // creamos dos streams de datos
    'temperature' : plotly.stream(tokens[0], function (err, res) {
      if (err) console.log(err);
      console.log(err, res);
    }),
    'humidity' : plotly.stream(tokens[1], function (err, res) {
      if (err) console.log(err);
      console.log(err, res);
    })
  };
};

```

Finalmente, leemos el streaming recibido mediante `parser.on()` y los transmitimos hacia el endpoint de plotly mediante el método `.write()`, el cual recibe un string JSON con los pares x (time) – y (stream), junto con el carácter de final de línea '\n'.

```

/* Read data when be emitted as an data event and then, run the callback function */
parser.on('data', (data) => {

    // data = 23.48\t45.62
    var values = data.split('\t'); //Split a string into an array of substrings ('\t' is used as the separator)
    // values = [23.48,45.62]

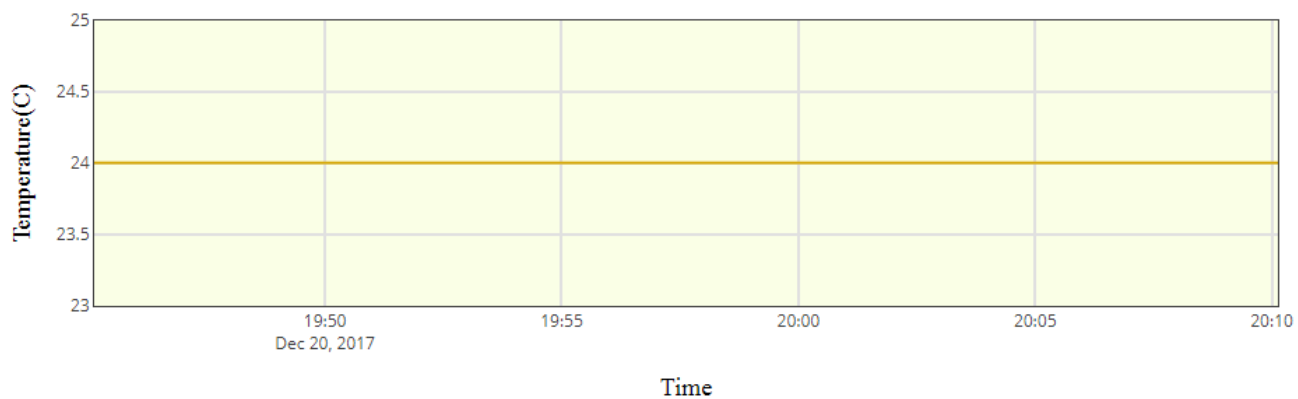
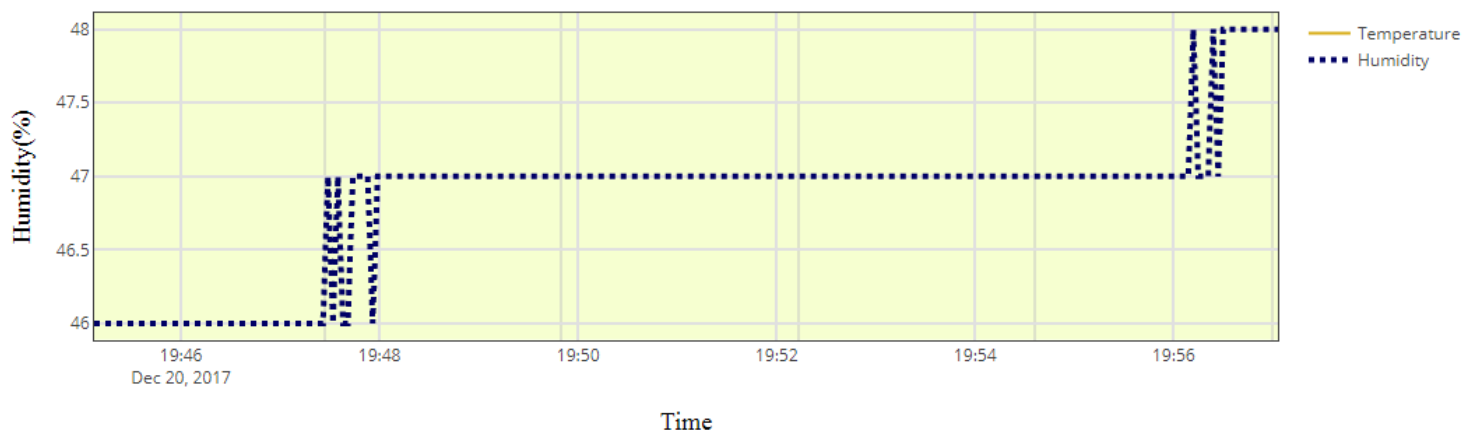
    // writing the temperature stream
    var tempStreamObject = JSON.stringify({ x : getDateString(), y : values[0] });
    console.log('temperatureObject: ' + tempStreamObject);
    streams['temperature'].write(tempStreamObject + '\n');

    // writing the humidity stream
    var humStreamObject = JSON.stringify({ x : getDateString(), y : values[1] });
    console.log('humidityObject: ' + humStreamObject);
    streams['humidity'].write(humStreamObject + '\n');

});
});

```

Temperature and Humidity



Referencias:

- <https://plot.ly/streaming/>
- <https://gist.github.com/alexander-daniel/3a12e3fddf5e7e1a50d5>
- https://github.com/plotly/Streaming-Demos/tree/master/node_examples/simple-signal-stream
- <https://os.mbed.com/users/AndyA/code/plotly/docs/d4f705ba2ea5/classplotly.html>
- <http://www.instructables.com/id/Plotly-Arduino-Data-Visualization/>
- <http://www.instructables.com/id/Real-Time-Temperature-Logging-With-Arduino-NodeJS-/>
- <https://codepen.io/etpinard/pen/yzaqmJ?editors=0010>
- <https://plot.ly/python/streaming-line-tutorial/>
- <https://github.com/plotly/plotly-nodejs/blob/master/examples/streaming-multiple-traces.js>
- <https://github.com/plotly/plotly-nodejs>
- <https://code.tutsplus.com/es/tutorials/create-interactive-charts-using-plotlyjs-getting-started--cms-29029>
- <https://plotlyblog.tumblr.com/>
- https://ac.els-cdn.com/S2314728816300149/1-s2.0-S2314728816300149-main.pdf?_tid=2bedb720-c99d-11e7-8e07-00000aab0f27&acdnat=1510706434_7469f1428b85a0eed2c7e9cb9f9d3db0
- <http://adilmoujahid.com/posts/2015/07/practical-introduction-iot-arduino-nodejs-plotly/>
- <http://www.ladyada.net/learn/arduino/lesson4.html>
- <https://desarrolloweb.com/articulos/buffer-en-nodejs.html>
- <https://programarfacil.com/blog/arduino-blog/sensor-dht11-temperatura-humedad-arduino/>