Sommaire

Rejoignez moi en live sur Twitch pour apprendre le tuto Arduino autrement ! (plus d'infos)

Arduino et Ethernet : client

Eskimon , le jeu. 26 mars 2015 (0 commentaire)

arduino tuto

Sommaire

Commençons doucement à découvrir ce shield en utilisant notre Arduino comme **client** . Dans ce chapitre, nous allons découvrir comment faire en sorte que notre Arduino aille interroger un site internet pour obtenir une information de sa part.

Ici nous allons interroger un site internet, mais cela aurait très bien pu être un autre service, sur un autre port (vous allez comprendre)

Afin de bien comprendre, je vous propose tout d'abord quelques explications sur le protocole HTTP et comment se passe un échange de données (requête + réponse).

Ensuite, nous verrons comment mettre en œuvre le shield et faire en sorte qu'il puisse accéder au monde extérieur.

Enfin, nous mettrons tout en œuvre pour faire une simple requête sur un moteur de recherche.

Pour terminer, je vous propose un petit exercice qui permettra de voir un cas d'utilisation : le "monitoring" d'un serveur.

Sommaire

Client et requêtes HTTP

Faisons un petit retour étendu sur la notion de client et surtout découvrons plus en détail en quoi consiste exactement une requête HTTP.

Un client, ça fait quoi ?

Le rôle du client est finalement assez simple. Son but sera d'aller chercher des informations pour les afficher à un utilisateur ou effectuer une action particulière. D'une manière générale, le client sera celui qui traite l'information que le serveur envoie.

Prenons l'exemple du navigateur web. C'est un *client* . Lorsque vous l'utilisez, vous allez générer des *requêtes* vers des serveurs et ces derniers vont ensuite renvoyer un tas d'informations (la page web). Le navigateur va alors les traiter pour vous les afficher sous une forme agréable et prévue par le développeur web.

Finalement, c'est simple d'être client, ça consiste juste à demander des choses et attendre qu'elles arrivent !

Les termes "client" et "serveur" (et même "requête") sont d'ailleurs très bien choisis car ils illustrent très bien les mêmes rôles que leur équivalent "dans la vraie vie".

Une requête HTTP, c'est quoi ?

Des requêtes HTTP il en existe de plusieurs types. Les plus classiques sont sûrement les GET et les POST, mais on retrouve aussi les PUT, DELETE, HEAD… Pour faire simple, nous allons uniquement nous intéresser à la première car c'est celle qui est utilisée la plupart du temps !

Émission

Dans la spécification du protocole HTTP, on apprend que GET nous permet de demander une ressource en lecture seule. On aura simplement besoin de spécifier une page cible et ensuite le serveur HTTP de cette page nous renverra la ressource ou un code d'erreur en cas de problème. On peut passer des arguments/options à la fin de l'adresse que l'on interroge pour demander des choses plus particulières au serveur.

Par exemple, si vous êtes sur la page d'accueil de Google et que vous faites une recherche sur "Arduino", votre navigateur fera la requête suivante : `GET /search?q=arduino HTTP/1.0` . Il interroge donc la page principale "/" (racine) et envoie l'argument "search?q=arduino". Le reste définit le protocole utilisé.

Réception

Une fois la requête faite, le serveur interrogé va vous renvoyer une réponse. Cette réponse peut être découpée en deux choses : l'en-tête (header) et le contenu. On pourrait comparer cela à un envoi de colis. L'en-tête possèderait les informations sur le colis (destinataires, etc.) et le contenu est ce qui est à l'intérieur du colis.

Typiquement, dans une réponse minimaliste, on lira les informations suivantes :

- Le code de réponse de la requête (200 si tout s'est bien passé) ;
- Le type MIME du contenu renvoyé (`text/html` dans le cas d'une page web) ;
- Une ligne blanche/vide ;

- Le contenu.

Les deux premières lignes font partie du header, puis après viendra le contenu.

Si l'on veut chercher des informations, en général on le fera dans le contenu.

Utilisation du shield comme client

Maintenant que l'on en sait un peu plus, on va pouvoir faire des requêtes et aller interroger le monde…

Préparation minimale

Pour commencer, il va falloir configurer notre module Ethernet pour qu'il puisse travailler correctement. Pour cela, il va falloir lui donner une adresse MAC et une adresse IP (qui peut être automatique, nous y reviendrons). On va utiliser 4 variables différentes :

- Un tableau de `byte` (ou char) pour les 6 octets de l'adresse MAC ;
- Un objet `IPAddress` avec l'adresse IP que l'on assigne au module ;
- Un objet `EthernetClient` qui nous servira à faire la communication ;
- Une chaîne de caractères représentant le nom du serveur à interroger.

L'adresse MAC doit normalement être écrite sur un autocollant au dos de votre shield (sinon inventez-en une !)

De manière programmatoire, on aura les variables ci-dessous.

```
// Ces deux bibliothèques sont indispensables pour le shield
#include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira à la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "leserveur.com"
```

Démarrer le shield

Maintenant que nous avons nos variables, nous allons pouvoir démarrer notre shield dans le `setup()` . Pour cela, il suffira d'appeler une fonction bien nommée : `begin()` , eh oui, comme pour la voie série ! Cette fonction prendra deux paramètres, l'adresse MAC et l'adresse IP à utiliser. C'est aussi simple que cela !

```
// Ces deux bibliothèques sont indispensables pour le shield
```

```
#include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira à la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "perdu.com"

void setup() {
  // On demarre la voie série pour déboguer
  Serial.begin(9600);

  // On démarre le shield Ethernet
  Ethernet.begin(mac, ip);
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
}
```

Simple non ?

le DHCP

Si vous bricolez à domicile, il est fort probable que vous soyez en train d'utiliser un routeur ou votre box internet. Bien souvent, ces derniers ont par défaut la fonction DHCP activée. Cette technologie permet de donner une adresse IP automatiquement à tous les équipements qui se connectent au réseau. Ainsi, plus besoin de le faire manuellement !

Du coup, on peut faire évoluer notre script d'initialisation pour prendre en compte cela.

```
// Ces deux bibliothèques sont indispensables pour le shield
#include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira a la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "perdu.com";

void setup() {
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
```

```
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  Serial.println("Pret !");
}
```

Envoyer une requête

Une fois que le shield est prêt, nous allons pouvoir commencer à l'utiliser ! Accrochez-vous, les choses amusantes commencent !

Requête simple

Pour débuter, il va falloir générer une requête pour interroger un serveur dans l'espoir d'obtenir une réponse. Je vous propose de commencer par une chose simple, récupérer une page web très sommaire, celle de http://perdu.com !

C'est là que notre variable "client" de type `EthernetClient` entre enfin en jeu. C'est cette dernière qui va s'occuper des interactions avec la page. Nous allons utiliser sa méthode `connect()` pour aller nous connecter à un site puis ensuite nous ferons appel à sa méthode `println()` pour construire notre requête et l'envoyer.

```
void setup() {
  // ... Initialisation précédente ...

  // On connecte notre Arduino sur "perdu.com" et le port 80 (defaut pour l'http)
  erreur = client.connect(serveur, 80);

  if(erreur == 1) {
      // Pas d'erreur ? on continue !
      Serial.println("Connexion OK, envoi en cours...");

      // On construit l'en-tête de la requête
      client.println("GET / HTTP/1.1");
      client.println("Host: perdu.com");
      client.println("Connection: close");
      client.println();
  } else {
    // La connexion a échoué :(
    Serial.println("Echec de la connexion");
    switch(erreur) {
      case(-1):
        Serial.println("Time out");
        break;
      case(-2):
        Serial.println("Serveur invalide");
        break;
      case(-3):
        Serial.println("Tronque");
        break;
      case(-4):
        Serial.println("Reponse invalide");
```

```
            break;
        }
        while(1); // Problème = on bloque
    }
}
```

Étudions un peu ces quelques lignes.

Tout d'abord, on va essayer de connecter notre client au serveur de "perdu.com" sur son port 80 qui est le port par défaut du protocole HTTP :

```
client.connect("perdu.com", 80);
```

Ensuite, une fois que la connexion semble correcte, on va construire notre requête pas à pas. Tout d'abord, on explique vouloir faire un `GET` sur la racine ("/") du site et le tout sous le protocole HTTP version 1.1.

```
client.println("GET / HTTP/1.1");
```

Ensuite, on redéclare le site qui devra faire (héberger, "host" en anglais) la réponse. En l'occurrence on reste sur perdu.com.

```
client.println("Host: perdu.com");
```

Puis, on signale au serveur que la connexion sera fermée lorsque les données sont transférées.

```
client.println("Connection: close");
```

Enfin, pour prévenir que l'on vient de finir d'écrire notre en-tête (*header*), on envoie une ligne blanche.

```
client.println();
```

J'ai ensuite rajouté un traitement des erreurs pour savoir ce qui se passe en cas de problème.

Requête avec paramètre

Et si l'on voulait passer des informations complémentaires à la page ? Eh bien c'est simple, il suffira juste de modifier la requête GET en lui rajoutant les informations ! Par exemple, admettons que sur perdu.com il y ait une page de recherche "recherche.php" qui prenne un paramètre "m" comme mot-clé de recherche. On aurait alors :

```
client.println("GET /recherche.php?m=monmot HTTP/1.1");
```

Ce serait équivalent alors à aller demander la page "perdu.com/recherche.php?m=monmot", soit la page "recherche.php" avec comme argument de recherche "m" le mot "monmot".

Lire une réponse

Lorsque la requête est envoyée (après le saut de ligne), le serveur va nous répondre en nous renvoyant la ressource demandée. En l'occurrence se sera la page d'accueil de perdu.com.

Eh bien vous savez quoi ? On fera exactement comme avec une voie série. On commencera par regarder si des données sont arrivées, et si c'est le cas, on les lira une à une pour en faire ensuite ce que l'on veut (recomposer des lignes, chercher des choses dedans…)

Dans l'immédiat, contentons-nous de tout afficher sur la voie série !

Première étape, vérifier que nous avons bien reçu quelque chose. Pour cela, comme avec Serial, on va utiliser la méthode `available()` qui nous retourne le nombre de caractères disponibles à la lecture.

```
client.available()
```

Si des choses sont disponibles, on les lit une par une (comment ? Avec `read()` bien sûr ! ) et les envoie en même temps à la voie série :

```
char carlu = 0;
// on lit les caractères s'il y en a de disponibles
if(client.available()) {
  carlu = client.read();
  Serial.print(carlu);
}
```

Enfin, quand tout a été lu, on va vérifier l'état de notre connexion et fermer notre client si la connexion est terminée.

```
// Si on est bien déconnecté.
if (!client.connected()) {
  Serial.println();
  Serial.println("Deconnexion !");
  // On ferme le client
  client.stop();
  while(1); // On ne fait plus rien
}
```

Globalement, voici le code pour lire une réponse :

```
void loop()
{
  char carlu = 0;
  // on lit les caractères s'il y en a de disponibles
  if(client.available()) {
    carlu = client.read();
    Serial.print(carlu);
  }

  // Si on est bien déconnecté.
  if (!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion !");
    // On ferme le client
    client.stop();
    while(1); // On ne fait plus rien
  }
}
```

Avez-vous remarqué, plutôt que de lire tous les caractères avec un `while` , on n'en lira qu'un seul à la fois par tour dans `loop()` . C'est un choix de design, avec un while cela marcherait aussi !

Code complet

En résumé, voici le code complet de la lecture de la page "perdu.com"

```
// Ces deux bibliothèques sont indispensables pour le shield
#include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira à la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "perdu.com";

void setup() {
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  Serial.println("Pret !");

  // On connecte notre Arduino sur "perdu.com" et le port 80 (defaut pour l'http)
  erreur = client.connect(serveur, 80);

  if(erreur == 1) {
      // Pas d'erreur ? on continue !
      Serial.println("Connexion OK, envoi en cours...");

      // On construit l'en-tête de la requête
      client.println("GET / HTTP/1.1");
      client.println("Host: perdu.com");
      client.println("Connection: close");
      client.println();
  } else {
    // La connexion a échoué :(
    Serial.println("Echec de la connexion");
    switch(erreur) {
      case(-1):
        Serial.println("Time out");
```

```
        break;
      case(-2):
        Serial.println("Serveur invalide");
        break;
      case(-3):
        Serial.println("Tronque");
        break;
      case(-4):
        Serial.println("Reponse invalide");
        break;
    }
    while(1); // On bloque la suite
  }
}

void loop()
{
  char carlu = 0;
  // on lit les caractères s'il y en a de disponibles
  if(client.available()) {
    carlu = client.read();
    Serial.print(carlu);
  }

  // Si on est bien déconnecté.
  if (!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion !");
    // On ferme le client
    client.stop();
    while(1); // On ne fait plus rien
  }
}
```

Et voici le résultat que vous devez obtenir dans votre terminal série. Remarquez la présence du header de la réponse que l'on reçoit.

```
HTTP/1.1 200 OK
Date: Thu, 26 Mar 2015 16:14:15 GMT
Server: Apache
Last-Modified: Tue, 02 Mar 2010 18:52:21 GMT
ETag: "cc-480d5dd98a340"
Accept-Ranges: bytes
Content-Length: 204
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

<html><head><title>Vous Etes Perdu ?</title></head><body><h1>Perdu sur l'Internet ?</h1><h2>Pas de
panique, on va vous aider</h2><strong><pre>    * <----- vous &ecirc;tes
ici</pre></strong></body></html>
```

Faire des requêtes en continu

Faire une requête en début de programme c'est bien, mais ce serait mieux de pouvoir la faire quand on veut. Faire évoluer notre programme ne va pourtant pas être si simple…

Pour commencer, on va ajouter quelques nouvelles variables. La première servira à se souvenir de quand date (via `millis()`) la dernière requête faite. Ce sera donc un `long()` . La

seconde sera une constante servant à indiquer le temps minimal devant s'écouler entre deux requêtes. Enfin, la dernière variable sera un booléen servant de *flag* pour indiquer l'état de la connexion (ouverte ou fermée) entre deux tours de boucle `loop()` .

```
// moment de la dernière requête
long dernierеRequete = 0;
// temps minimum entre deux requêtes
const long updateInterval = 10000;
// mémorise l'état de la connexion entre deux tours de loop
bool etaitConnecte = false;
```

Ensuite, on va créer une fonction sobrement nommée `requete()` qui se chargera de construire et envoyer la requête. Aucune nouveauté là-dedans, c'est le code que vous connaissez déjà.

```
void requete() {
  // On connecte notre Arduino sur "perdu.com" et le port 80 (defaut pour l'http)
  char erreur = client.connect(serveur, 80);

  if(erreur == 1) {
      // Pas d'erreur ? on continue !
      Serial.println("Connexion OK, envoi en cours...");

      // On construit l'en-tête de la requête
      client.println("GET / HTTP/1.1");
      client.println("Host: perdu.com");
      client.println("Connection: close");
      client.println();

      // On enregistre le moment d'envoi de la dernière requête
      dernierеRequete = millis();
  } else {
    // La connexion a échoué :(
    // On ferme notre client
    client.stop();
    // On avertit l'utilisateur
    Serial.println("Echec de la connexion");
    switch(erreur) {
      case(-1):
        Serial.println("Time out");
        break;
      case(-2):
        Serial.println("Serveur invalide");
        break;
      case(-3):
        Serial.println("Tronque");
        break;
      case(-4):
        Serial.println("Reponse invalide");
        break;
    }
  }
}
```

La seule différence ici est que l'on ferme le client si la connexion a un problème et aussi on enregistre le moment auquel a été faite la requête.

Bien. Maintenant, il faut revoir notre loop.

Le début ne change pas, on récupère/affiche les caractères s'ils sont disponibles.

Ensuite, on avait l'étape de fermeture du client si la connexion est fermée. Dans notre cas actuel, on ne doit pas l'arrêter n'importe quand, car sinon une connexion pourrait de nouveau avoir lieu sans que celle-là soit finie. On va donc la fermer QUE si elle était déjà fermée à la toute fin du tour précédent. Voici comme cela peut-être représenté :

```
void loop()
{
  // traite les caractères
  // ...

  // SI on était connecté au tour précédent
  // ET que maintenant on est plus connecté
  // ALORS on ferme la connexion
  if (etaitConnecte && !client.connected()) {
    Serial.println();
    Serial.println("Deconnexion !");
    // On ferme le client
    client.stop();
  }

  // ...
  // bricole, traite, calcul, manigance, complote...
  // ...

  // enregistre l'état de la connexion (ouvert ou fermé)
  etaitConnecte = client.connected();
}
```

Maintenant, il ne nous reste plus qu'à redéclencher une requête si nos dix secondes se sont écoulées et que la précédente connexion a fini de travailler :

```
// Si on est déconnecté
// et que cela fait plus de xx secondes qu'on a pas fait de requête
if(!client.connected() && ((millis() - derniereRequete) > updateInterval)) {
  requete();
}
```

C'est tout clair ? Ci-dessous le code complet pour vous aider à y voir un peu mieux dans l'ensemble  .

Le code complet

```
// Ces deux bibliothèques sont indispensables pour le shield
#include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira a la communication
EthernetClient client;
```

```
// Le serveur à interroger
char serveur[] = "perdu.com";

// pour lire les caractères
char carlu = 0;
// moment de la dernière requête
long derniereRequete = 0;
// temps minimum entre deux requêtes
const long updateInterval = 10000;
// mémorise l'état de la connexion entre deux tours de loop
bool etaitConnecte = false;

void setup() {
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  Serial.println("Pret !");
}

void loop()
{
  // on lit les caractères s'il y en a de disponibles
  if(client.available()) {
    carlu = client.read();
    Serial.print(carlu);
  }

  // SI on était connecté au tour précédent
  // ET que maintenant on est plus connecté
  // ALORS on ferme la connexion
  if (etaitConnecte && !client.connected()) {
    Serial.println();
    Serial.println("Deconnexion !");
    // On ferme le client
    client.stop();
  }

  // Si on est déconnecté
  // et que cela fait plus de xx secondes qu'on a pas fait de requête
  if(!client.connected() && ((millis() - derniereRequete) > updateInterval)) {
    requete();
  }

  // enregistre l'état de la connexion (ouvert ou fermé)
  etaitConnecte = client.connected();
}

void requete() {
```

```
  // On connecte notre Arduino sur "perdu.com" et le port 80 (defaut pour l'http)
  char erreur = client.connect(serveur, 80);

  if(erreur == 1) {
      // Pas d'erreur ? on continue !
      Serial.println("Connexion OK, envoi en cours...");

      // On construit l'en-tête de la requête
      client.println("GET / HTTP/1.1");
      client.println("Host: perdu.com");
      client.println("Connection: close");
      client.println();

      // On enregistre le moment d'envoi de la dernière requête
      derniereRequete = millis();
  } else {
    // La connexion a échoué :(
    // On ferme notre client
    client.stop();
    // On avertit l'utilisateur
    Serial.println("Echec de la connexion");
    switch(erreur) {
      case(-1):
        Serial.println("Time out");
        break;
      case(-2):
        Serial.println("Serveur invalide");
        break;
      case(-3):
        Serial.println("Tronque");
        break;
      case(-4):
        Serial.println("Reponse invalide");
        break;
    }
  }
}
```

L’intérêt d’un client

En lisant tout ça, vous vous dites peut-être qu’utiliser le shield Ethernet en mode client est un peu inutile. C’est vrai, après tout on ne peut même pas afficher les pages reçues !

Cependant, avec un peu d’imagination on pourrait facilement voir des utilisations plus que pratiques !

Télécharger une configuration

La première idée par exemple pourrait être de mettre à disposition des fichiers de paramètres à l’Arduino. Imaginons par exemple que je fais une application qui dépend des saisons. Mon application pourrait utiliser des moteurs et des capteurs pour faire certaines actions, mais ces dernières pourraient être différentes en fonction du moment de l’année. Plutôt que de stocker 4 ensembles de paramètres dans l’Arduino, cette dernière pourrait vérifier régulièrement en ligne (en interrogeant un petit script que nous aurions fait) pour savoir si quelque chose doit changer. Et voilà, configuration à distance !

Enregistrer des données en ligne

J'ai souvent lu sur des forums que des gens aimeraient pouvoir sauvegarder des choses dans une base de données. Bien entendu, l'Arduino seule ne peut pas le faire, gérer une BDD est bien trop compliqué pour elle. En revanche, elle pourrait facilement interroger des pages en insérant des paramètres dans sa requête. La page qui reçoit alors la requête lirait ce paramètre et s'occuperait de sauvegarder les infos. Par exemple, on pourrait imaginer la requête `http://mapageweb.com/enregistrer.php?&analog1=123&analog2=28&millis=123456` . Cette requête possède deux paramètres, **analog1** et **analog2** qui pourraient être les valeurs des entrées analogiques et un dernier "millis" qui pourrait être la valeur de millis() d'Arduino. Notre page "enregistrer.php" ferait alors la sauvegarde dans sa base de données ! Pour construire une telle requête, le code suivant devrait faire l'affaire :

```
// On construit l'en-tête de la requête
client.print("GET /enregistrer.php/?analog1="); //attention, pas de saut de ligne !
client.print(analogRead(A1));
client.print("&analog2=");
client.print(analogRead(A2));
client.print("&millis=");
client.print(millis());
// On finit par le protocole
client.println(" HTTP/1.1"); //ce coup-ci, saut de ligne pour valider !
// on aurait alors :
// "GET /enregistrer.php/?analog1=<valeur-de-A1>&analog2=<valeur-de-A2>&millis=<valeur-de-millis()>
HTTP/1.1"
client.println("Host: mapageweb.com");
client.println("Connection: close");
client.println();
```

Sûrement plein d'autres choses !

Il existe surement un paquet d'autres idées auquel je n'ai pas pensé !

Exercice, lire l'uptime de Eskimon.fr

Pour finir ce chapitre je vous propose un petit exercice, allez lire l'uptime (temps écoulé depuis la mise en route du système) de mon blog, eskimon.fr.

Consigne

Pour cela, je vous ai concocté une petite page juste pour vous qui renvoie uniquement cette information, sans tout le bazar HTTP qui va avec une page classique. Vous obtiendrez ainsi simplement le header de la requête et la valeur brute de l'uptime.

La page à interroger pour votre requête est : `http://eskimon.fr/public/arduino.php`

Essayer de modifier votre code pour afficher cette information 

Vous êtes maintenant en mesure de vous balader sur Internet pour aller y chercher des informations. Bienvenue dans l'IoT, l'Internet of Things !

---

tuto-arduino-801-découverte-de-lethernet-sur-arduino

tuto-arduino-803-arduino-et-ethernet-serveur