

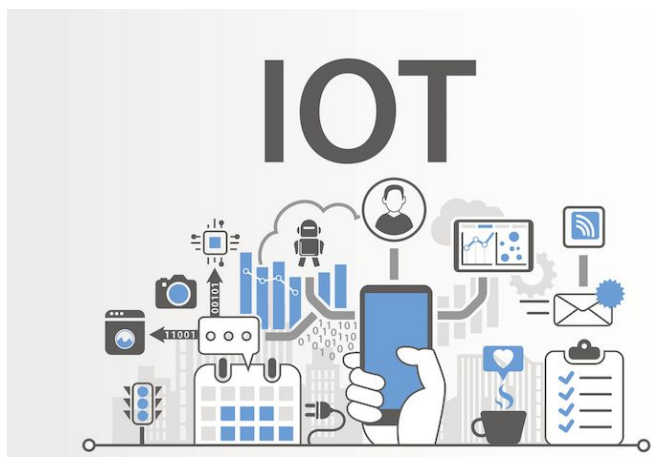


UNIVERSITÉ
DE MONTPELLIER



RAPPORT DE PROJET

Simulation de l'Internet des Objets avec COOJA



Réalisé par :

MATARISE Loan, POUJOL Maxime, SAHONET Arnaud, PERNOT Antoine

Sous la direction de :

Mr MOLNAR

Pour l'obtention du DUT Informatique

Année Universitaire : 2020/2021

REMERCIEMENTS

A l'issue de ce travail, nous tenons à exprimer notre gratitude et nos remerciements pour toutes les personnes qui ont contribué à sa réalisation.

Tout d'abord, nous voulons particulièrement remercier Monsieur MOLNAR, notre tuteur de projet, pour son aide, ses conseils mais également pour ses précieuses explications sur des notions jamais vues en cours.

Par la suite, nous remercions Madame MESSAOUI pour son aide dans la rédaction de ce projet notamment pour la mise en page et la structuration.

Pour finir, un grand merci à l'IUT de nous avoir permis de réaliser ce projet, assez différent des notions vus en cours, nous permettant d'acquérir de nouvelles connaissances.

GLOSSAIRE

6LowPAN - il s'agit de l'acronyme de IPv6 Low power Wireless Personal Area Networks ou IPv6 LoW Power wireless Area Networks.

Mote - c'est un nœud qui sert d'émetteur-récepteur sans fil combiné avec des capteurs, il reçoit et envoie des données.

LoRaWAN - c'est un protocole de télécommunication permettant la communication à bas débit, par radio, d'objets à faible consommation électrique communiquant selon la technologie LoRa et connectés à l'Internet via des passerelles, participant ainsi à l'Internet des objets.

Routeur de bordure - il s'agit du routeur à la base du réseau. Il sert le plus souvent de racine RPL ou de lien avec un autre réseau.

SOMMAIRE

REMERCIEMENTS	1
GLOSSAIRE	2
SOMMAIRE	3
Introduction	4
1 - Les protocoles de l'internet des objets	5
1. 6LowPAN	5
2. Protocole RPL	6
3. DODAG et Routage	7
2 - Contiki et Cooja	8
1. Étapes d'installation de Contiki sur VirtualBox	8
2. Contiki	13
3. Cooja	15
3 - Etude de cas	17
1. Notre étude de cas	17
2. Notre approche	18
3. Notre réalisation	19
Annexes	21

Introduction

Il existe de nombreux types d'objets connectés et leurs besoins en communication sont différents, ce qui impliquent donc divers réseaux. Ces derniers peuvent être basés sur un protocole bas débit, comme par exemple LoRaWAN* pour collecter des données ou encore sur des solutions usuelles telles que l'IP (version 6 ici).

Nous allons donc nous demander comment fonctionne l'Internet des objets.

Ici, l'enjeu essentiel est de comprendre ce mode de fonctionnement assez particulier ainsi que les communications qui ont lieu tout autour de nous chaque jour, notamment à travers l'architecture réseau.

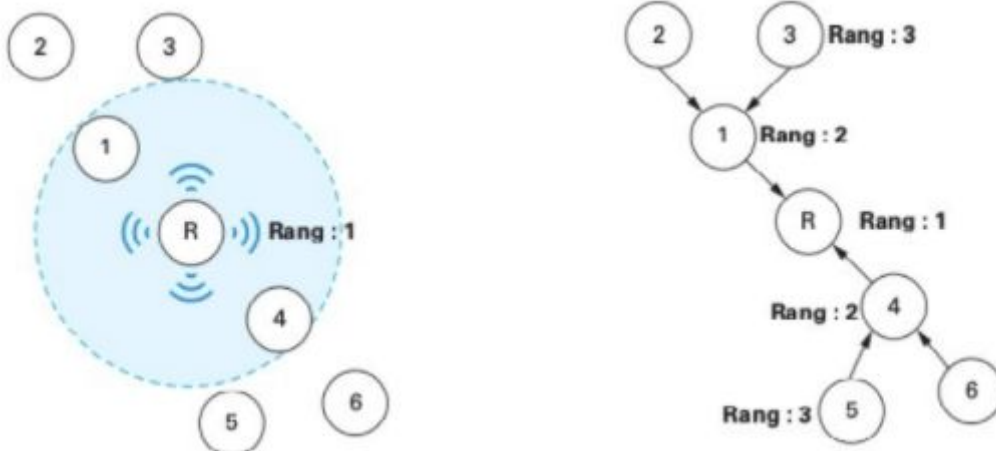
L'objectif majeur de notre projet est d'apprendre à manipuler le logiciel Cooja, installé sur le système d'exploitation Contiki. Ce qui nous permettra de conceptualiser une étude de cas.

Afin de répondre à notre problématique, nous allons étudier le protocole 6LowPAN* ainsi que les autres concepts qui y sont liés (notamment RPL et DODAG). Par la suite, nous allons voir les outils que nous avons utilisés pour réaliser des simulations d'objets interconnectés : le système d'exploitation Contiki ainsi que le logiciel Cooja. Pour finir, nous allons expliquer tous ces concepts à travers notre étude de cas sur un hypermarché, où l'on compte le nombre de personnes dans un des 3 espaces différents, que ce soit le parking, la galerie marchande ou bien l'hypermarché lui-même.

2. Protocole RPL

Le protocole de routage standardisé RPL(Routing Protocol, Protocole de Routage) a pour but de construire un réseau entre les différents objets connectés.

Il part tout d'abord du routeur de bordure pour se propager à chaque objet, et lorsque les données sont transmises d'un objet à un autre ou bien jusqu'au routeur de bordure, celui-ci intervient à chaque fois.



Lors du trafic typique dans le réseau qu'est le ramassage de données incast (multipoint-à-point, des nœuds vers le BR), le protocole permet la diffusion entre les objets et le routeur de bordure. Grâce à ce protocole de routage, on peut construire, pour chaque application, un DODAG.

Le multipoint-à-point est l'inverse du point-à-multipoints qui consiste à la communication qui par du routeur de bordure pour aller aux appareils connectés. Alors le multipoint-à-point consiste à la communication des appareils connectés au routeur de bordure.

3. DODAG et Routage

Le DODAG (Destination Oriented Directed Acyclic Graph) est un graphe orienté vers une destination. Expliquons ça plus clairement:

Nous savons qu'il existe 3 types de noeuds dans le réseau 6LowPAN vu précédemment :

- Le routeur de bordure* (aussi appelé BR)
- Le(s) routeur(s)
- Le(s) hôte(s)

Dans le principe, le DODAG va faire remonter les données des hôtes vers le routeur bordure en passant par le ou les routeurs qui sont des sortes d'étapes intermédiaires.

On peut donc le schématiser de la manière suivante :

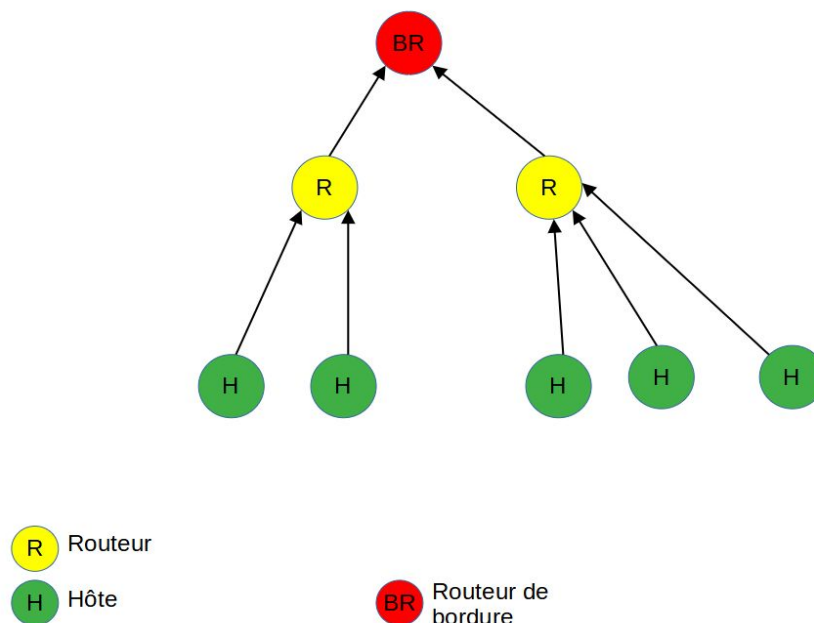


Schéma n°1 : Schématisation d'un DODAG

La construction d'un DODAG s'effectue par couche (ou rang par rang).

Tout d'abord, le BR ou la racine RPL a un rang égal à 1. Celui-ci va ensuite envoyer un message DIO (DODAG Information Object) dans un périmètre donné pour entrer en connexion avec les nœuds voisins.

Les nœuds recevant un DIO vont ensuite à l'aide d'une fonction déterminer leur parent "préféré", celui formant le plus souvent le chemin le plus court vers le BR. Puis ils émettent à leur tour un DIO et ainsi de suite jusqu'à ce que l'ensemble des nœuds soient connectés au DODAG.

2 - Contiki et Cooja

1. Étapes d'installation de Contiki sur VirtualBox

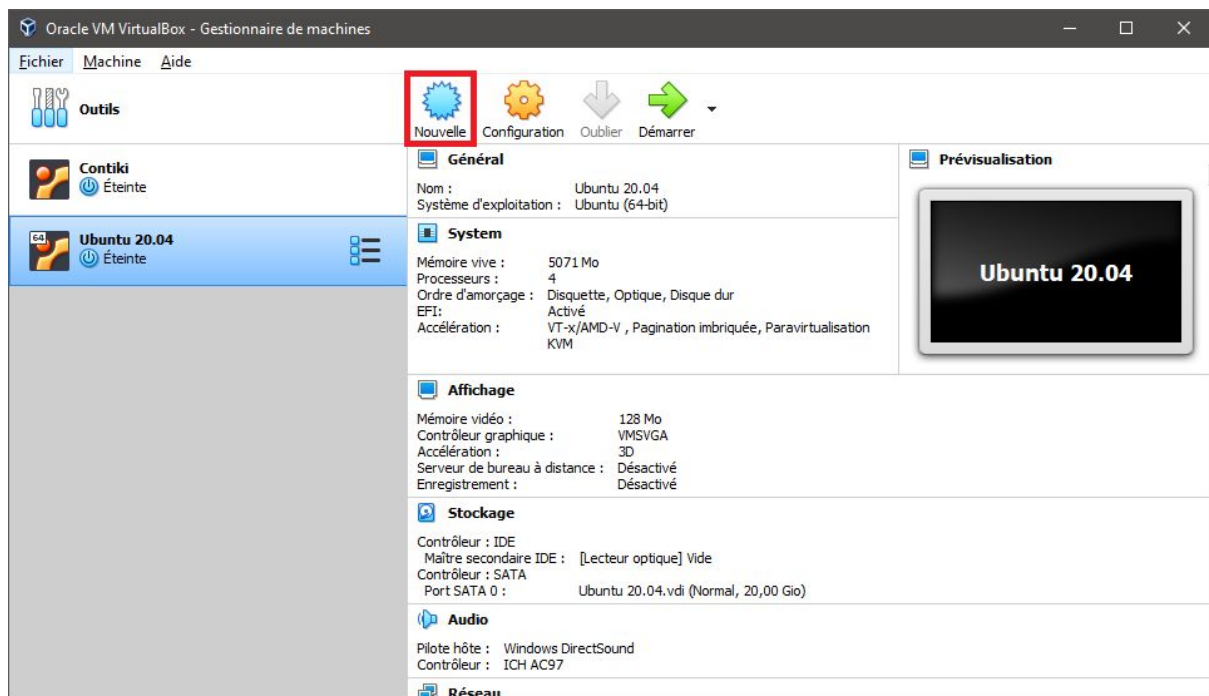
1- Téléchargez les ressources nécessaires via les liens ci-dessous :

[Dernière version de VirtualBox.](#)

[Version de Instant-Contiki pour des machines virtuelles.](#)

2- Pour l'installation de VirtualBox, suivez les instructions une à une, vous ne devriez pas rencontrer de difficulté. Extrayez les fichiers contenus dans le .zip de instant-contiki.

3- Ouvrez VirtualBox, et une fenêtre comme ci-dessous devrait s'ouvrir. Cliquez sur "Nouvelle" pour créer une machine virtuelle.



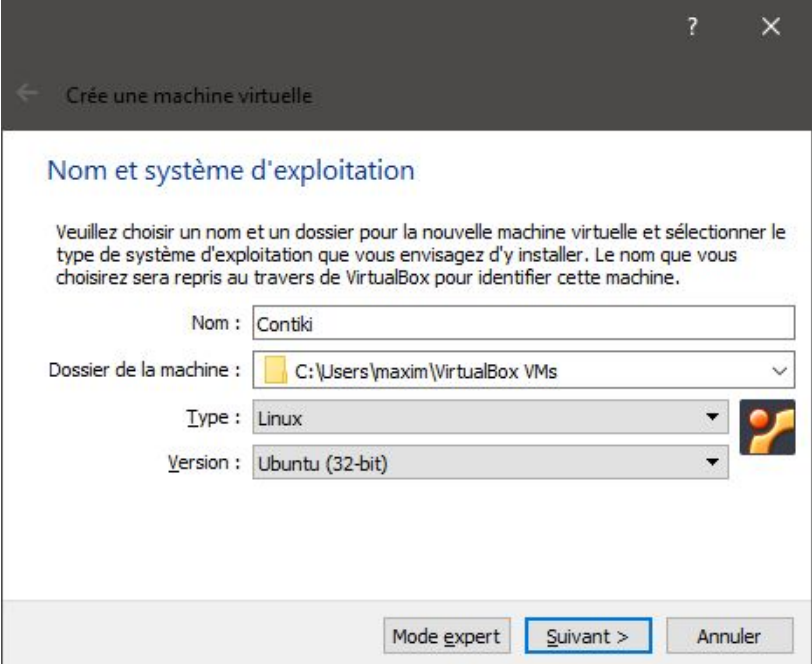
4- Sur la fenêtre suivante, remplissez les champs tels que :

“Nom “ : **nom de votre machine virtuelle.**

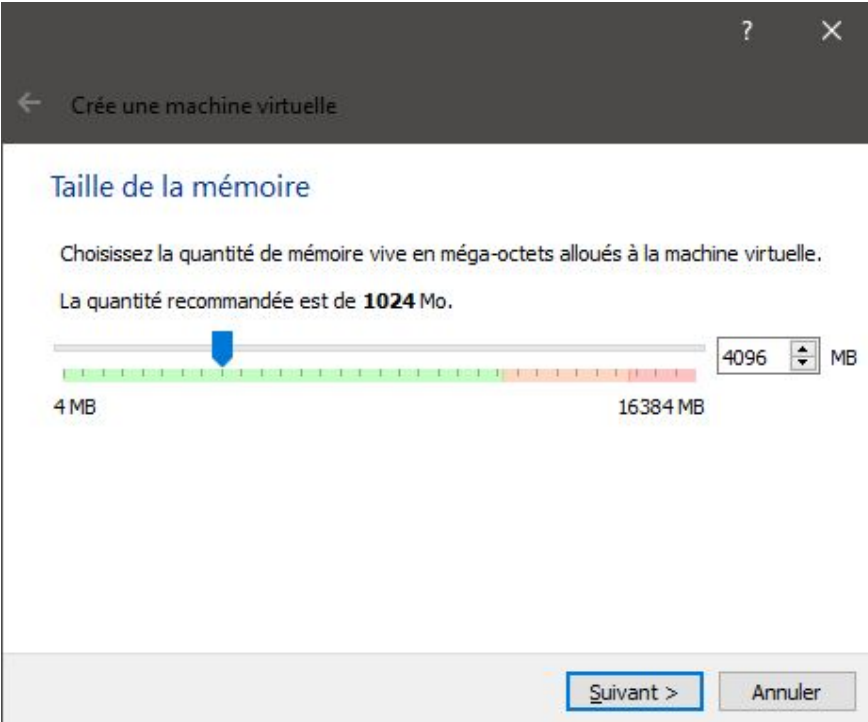
“Dossier de la machine” : **emplacement d’installation de la machine virtuelle.**

“Type” : **Linux.**

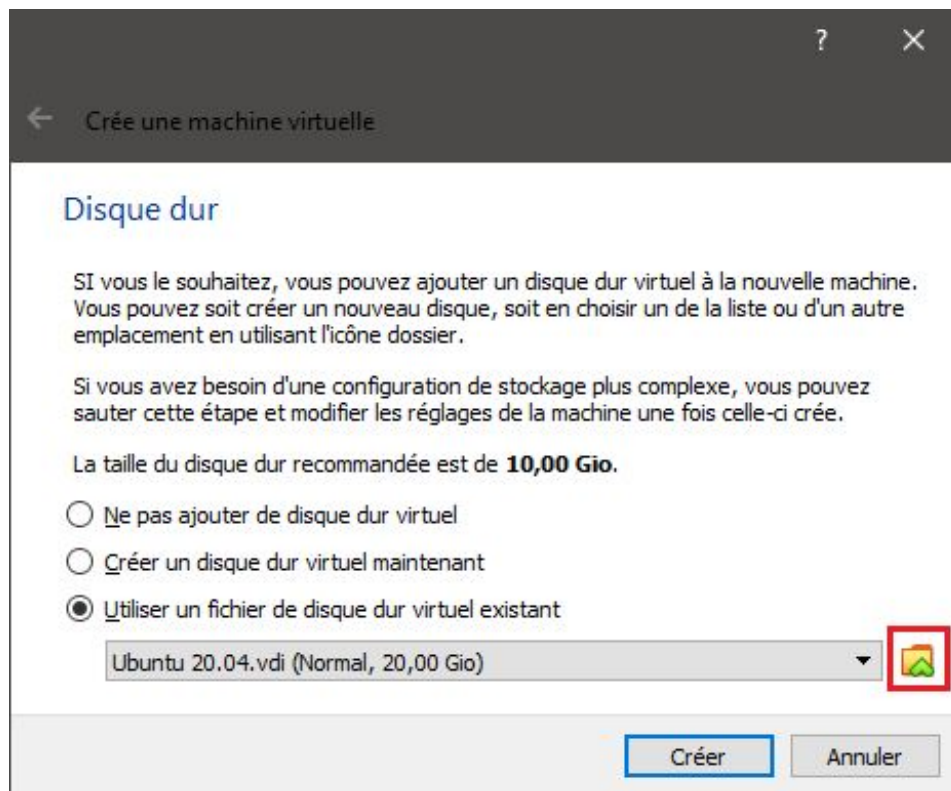
“Version” : **Ubuntu (32-bit).**



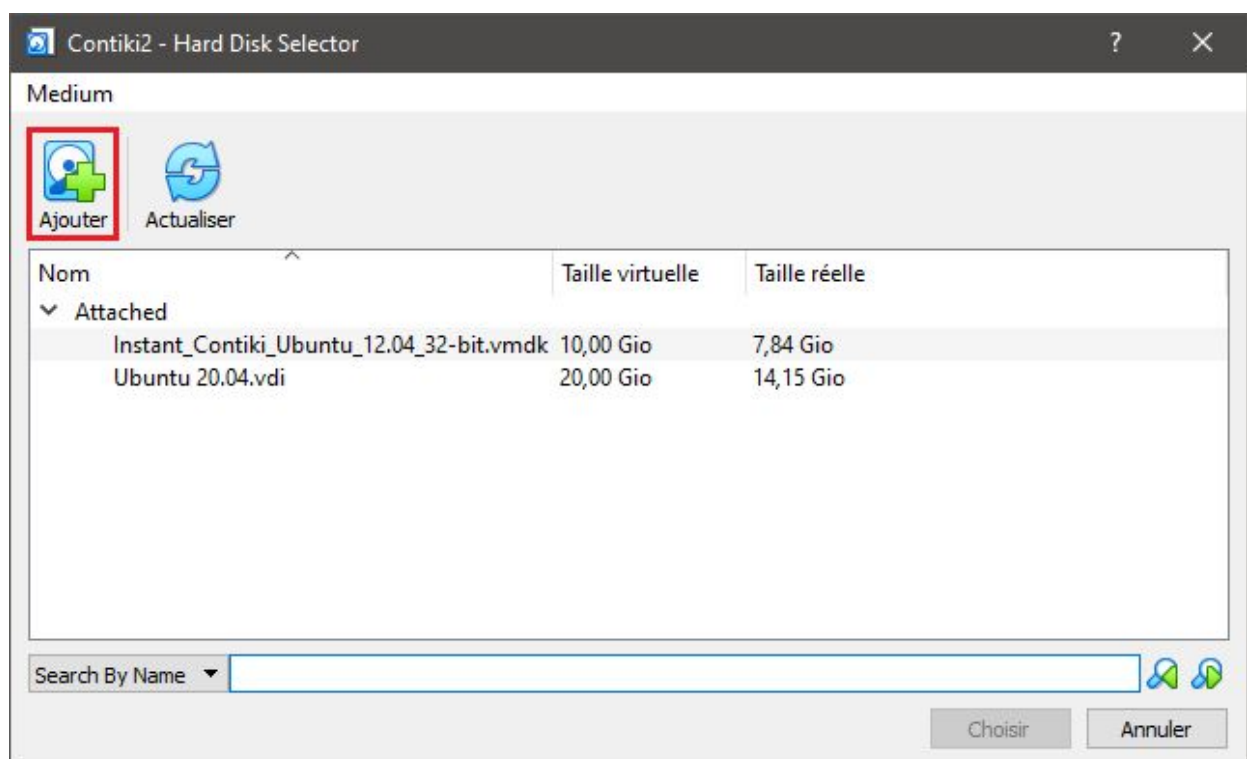
5- Sur la fenêtre suivante, indiquez la taille de la mémoire (RAM) de votre machine virtuelle (**minimum 2048 MB, recommandé 4096 ou plus MB**).



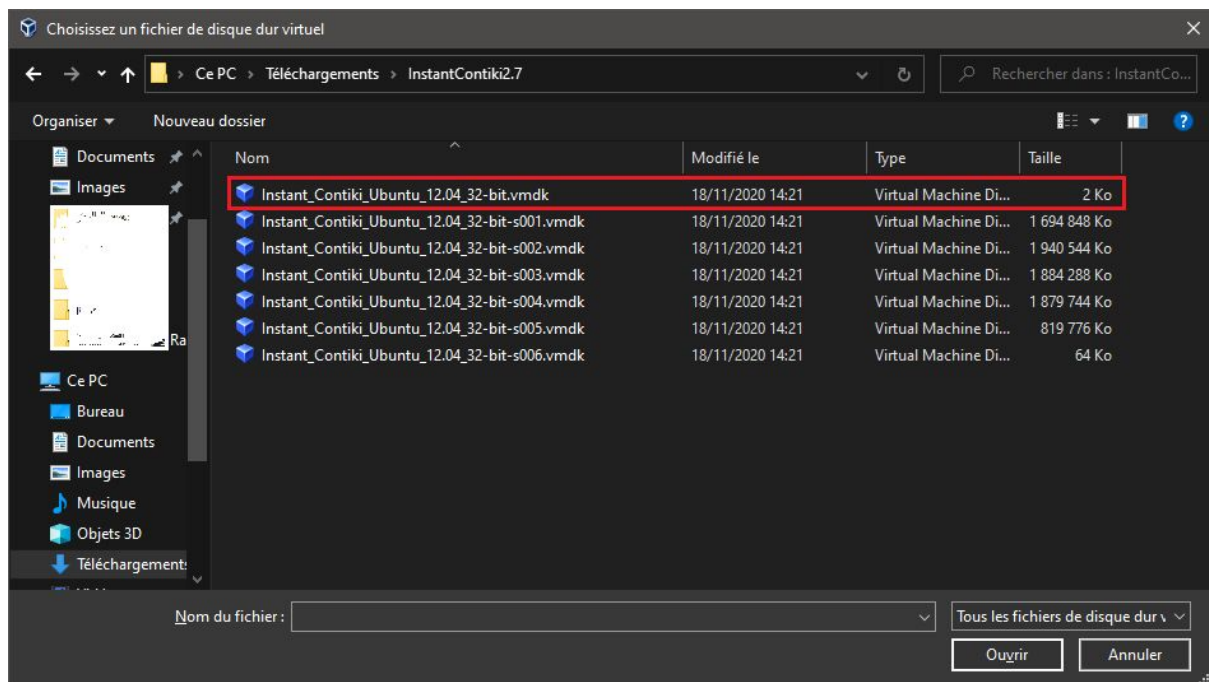
6- Sur la fenêtre suivante, sélectionnez “**Utiliser un fichier de disque dur virtuel existant**”, puis cliquez sur l'**icône de fichier** en bas à droite.



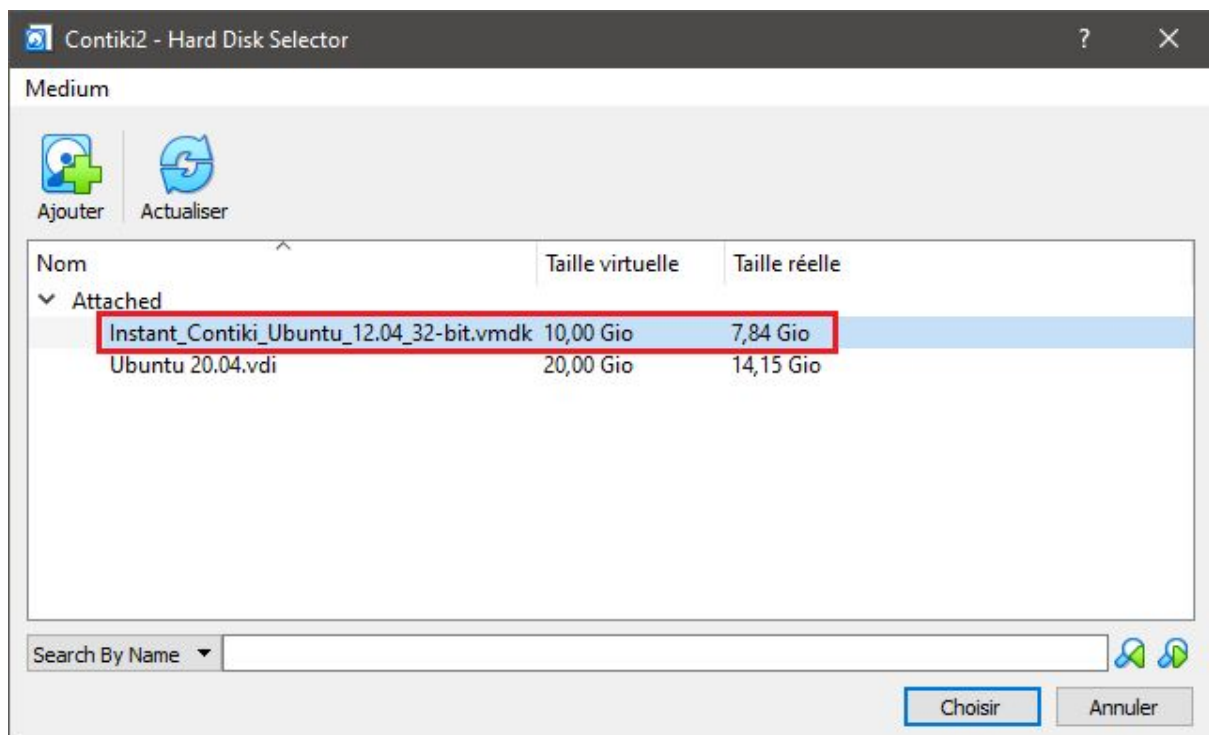
7- Sur la fenêtre suivante, cliquez sur **ajouter** en haut à gauche.



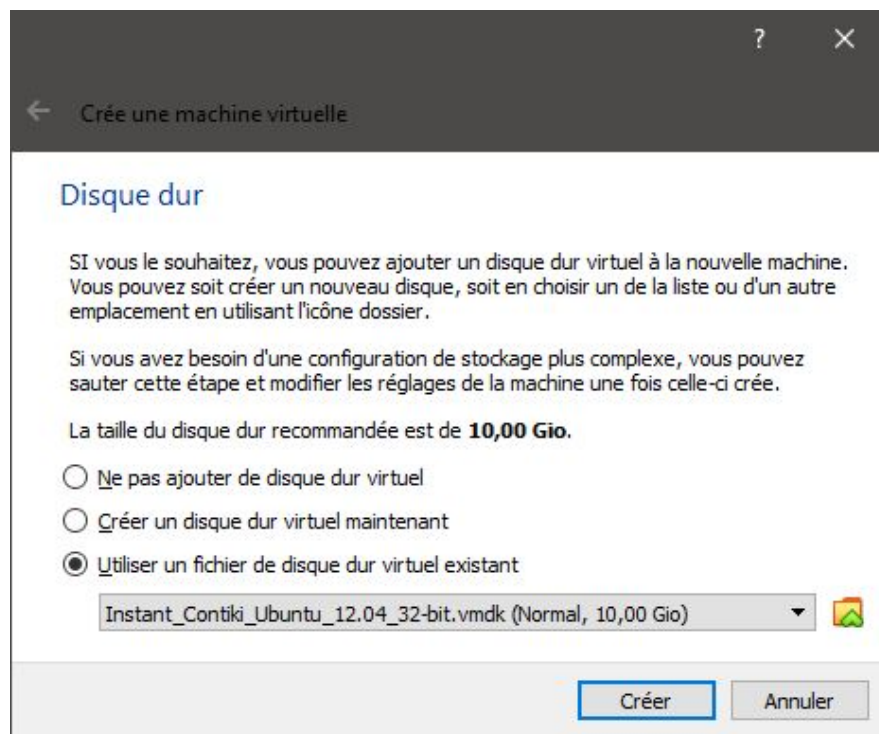
8- Sur la fenêtre suivante, sélectionnez “Instant_Contiki_Ubuntu_12.04_32-bit.vmdk” puis cliquez sur **Ouvrir** en bas à droite. **NE PAS CHOISIR “... 12.04.32-bit-s00 ...”**.



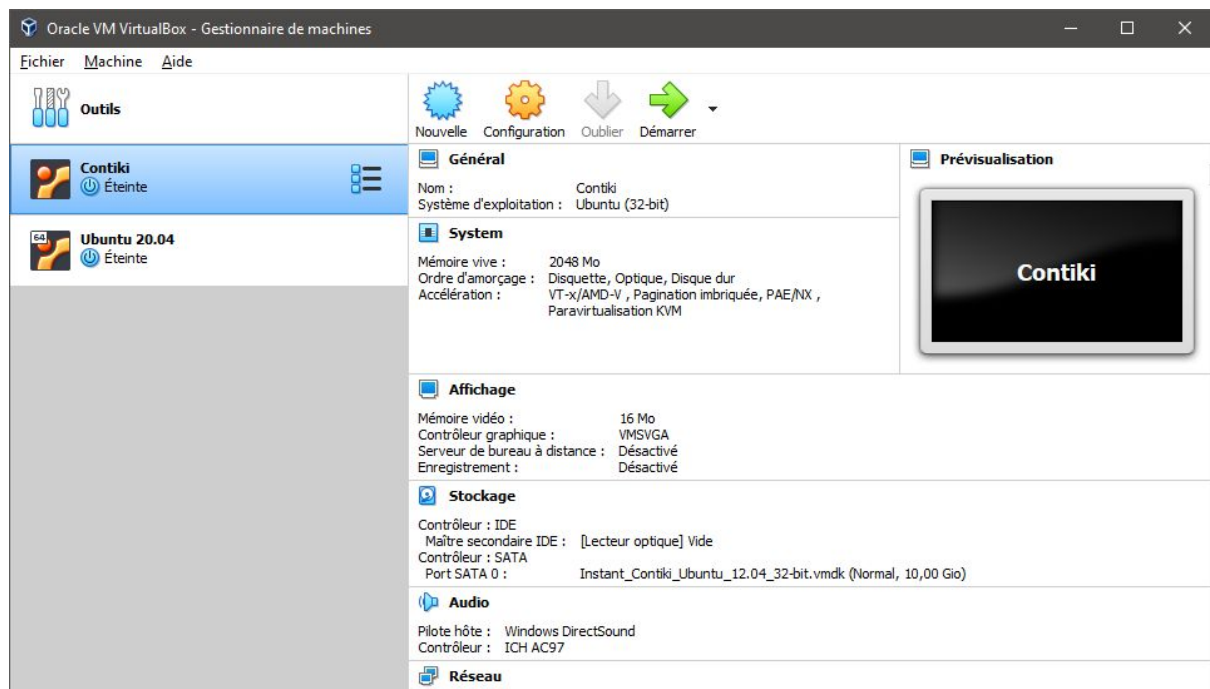
9- Sur la fenêtre suivante, sélectionnez le **fichier choisi précédemment** puis cliquez sur **Choisir** en bas à droite.



10- Vérifiez que le fichier sélectionné est le bon puis cliquez sur **Créer**.



11- Enfin, sélectionnez Contiki sur le côté gauche. Si vous voulez **configurer** la machine virtuelle, cliquez sur le **petit engrenage orange "Configuration"**. Sinon, pour la **lancer**, cliquez sur la **flèche verte "Démarrer"**.



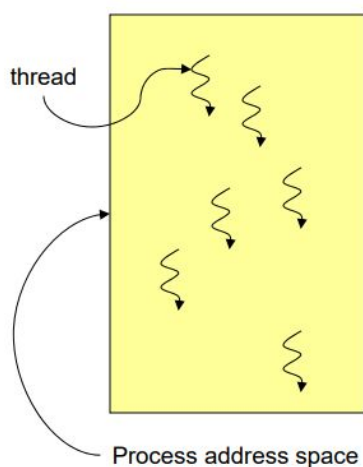
2. Contiki

Contiki n'est pas véritablement un OS Linux même s'il est basé sur l'OS Ubuntu : c'est un "exécutif" (un seul exécutable OS + Application). La première version a été développée et publiée le 10 mars 2003 par Adam Dunkels, un développeur indépendant. Cet OS est utilisé pour sa faible utilisation de la mémoire vive pour fonctionner (2 Ko de RAM) contrairement à Ubuntu (2Go de RAM au minimum). Pour plus de détails sur les différences d'architecture entre Contiki et un OS Linux, vous trouverez des images en annexes permettant la comparaison.

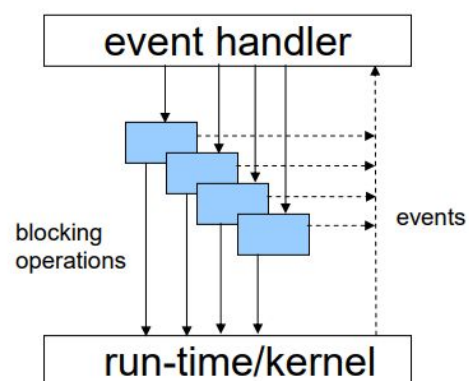
Contiki est surtout très utilisé pour son émulateur préinstallé Cooja dont on parlera en détails dans la partie qui suit.

La majorité des OS sont multithread, c'est-à-dire qu'ils disposent de plusieurs threads qui peuvent s'exécuter en parallèle : chacun dispose d'une pile et d'un contexte. Ce système nécessite une programmation complexe car les threads sont indépendants, il est donc très difficile de les faire communiquer entre eux et ils doivent être tous synchronisés entre eux.

Contiki, quant à lui, utilise un modèle événementiel : lorsque le programme fonctionne, il se bloque et attend qu'un événement se produise. Ces événements sont appelés "Event-handler" et sont exécutés par le noyau, jusqu'à la fin. Lorsqu'un événement est détecté, il va alors effectuer les actions qui lui sont liées. Ce modèle, contrairement au modèle multithread, n'est pas un système préemptif : en informatique, un système préemptif a la capacité d'interrompre une tâche en cours pour une autre tâche de plus importante.



thread model



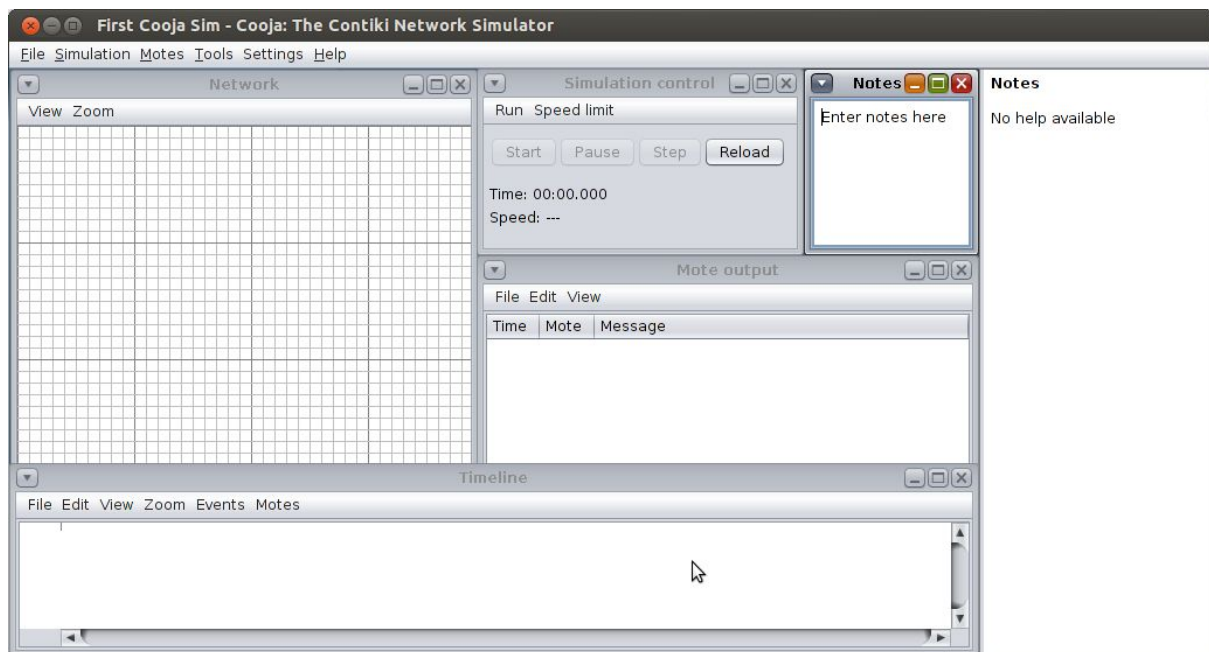
event model

Sur les modèles événementiels récents, on trouve le “XNextEvent()” qui utilise des protothreads. Ceux sont des modèles de thread, mais simplifié qui disposent de grands atouts :

- Ils partagent le même contexte d'exécution.
- Ils sont basés sur des macros.
- Ils n'ont pas de pile donc pas de variables locales.
- Ils sont portables car ils sont écrits en C.
- L'attente est bloquante sans changement de contexte.

3. Cooja

Cooja est un logiciel disponible dans l'environnement de développement Instant Contiki, c'est un simulateur de réseau. Ce simulateur permet l'émulation de différents capteurs sur lesquels seront chargés un système d'exploitation et des applications. Cooja permet ensuite de simuler les connexions réseaux et d'interagir avec les capteurs.



Application Cooja au démarrage

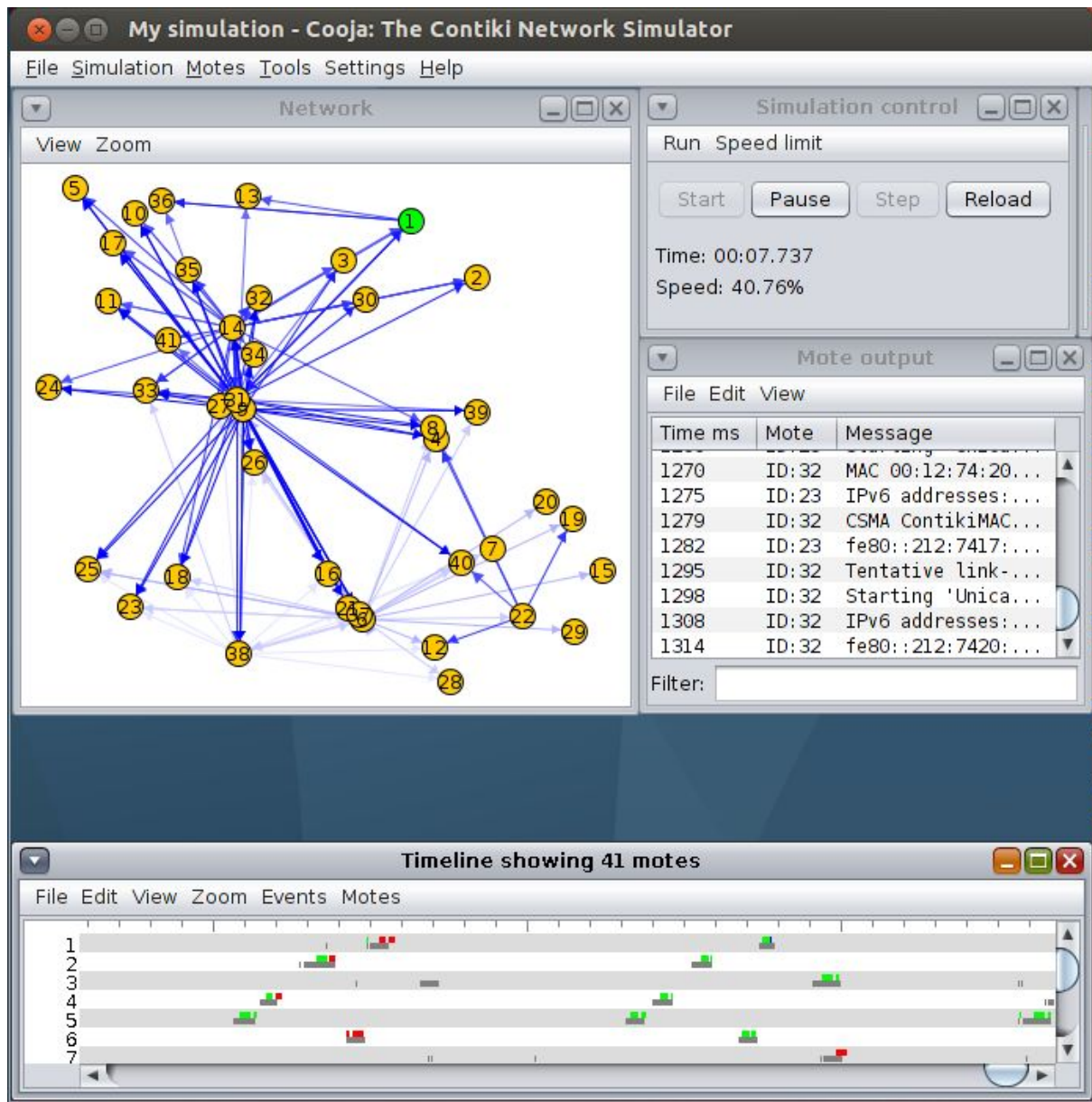
Dans la simulation d'un réseau IdO, nous pouvons utiliser les éléments intégrés dans Cooja (appelés des motes*) qui font appel à Contiki. Pour développer de nouvelles solutions, par exemple pour compléter ou modifier le routage, on peut utiliser les fichiers supplémentaires (par exemple de RPL) écrits en C, afin de modifier un comportement ou des parties du système.

Qu'est-ce qu'une mote ?

Une grande partie de l'Internet des objets (IdO) se compose d'émetteurs-récepteurs sans fil combinés à des capteurs, qui peuvent résider dans des appareils électro-ménagers ou pas, des vêtements, des machines, des bâtiments, à peu près tout ce qui est physique.

Bien sûr, l'expression « émetteur-récepteur sans fil combiné avec des capteurs » est peu maniable, de sorte qu'un tel nœud de l'IdO est appelé une mote (abréviation de *remote*).

Tout mot digne de ce nom doit avoir une adressabilité, la condition d'être non seulement identifiable de manière unique, mais également trouvable. Le système qui le gère s'appelle l'identité des objets (IDdO ou en anglais IDoT).



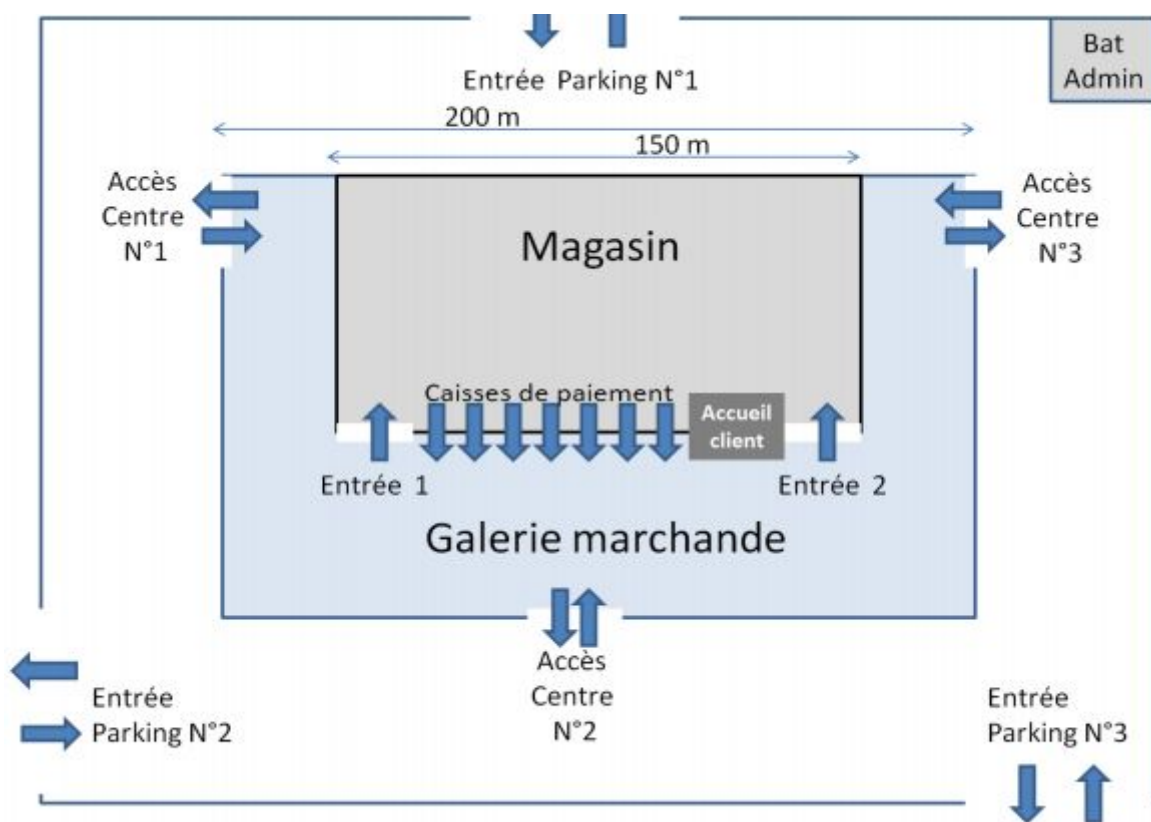
Exemple de simulation de réseau de capteurs sans fil avec Cooja

3 - Etude de cas

1. Notre étude de cas

Notre étude de cas consiste à analyser la fréquentation d'un grand hypermarché en temps réel. Pour cela, nous voulons compter le nombre de personnes venant en voiture, venant uniquement pour la galerie marchande ou venant pour l'hypermarché.

Pour plus de clarté, voici un petit schéma :



2. Notre approche

Notre approche va consister à placer des capteurs aux entrées et aux sorties (flèches bleus sur le schéma ci-dessus) et faire 3 compteurs :

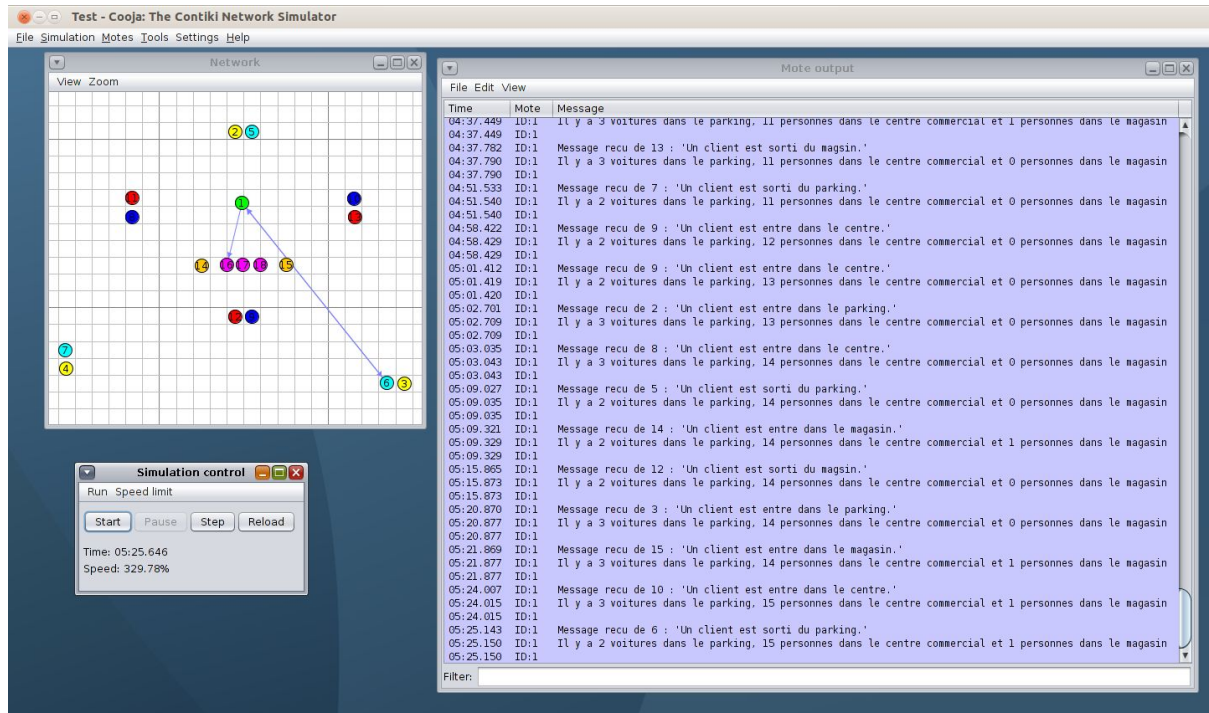
Rôle	Règles
1 compteur pour entrées/sorties du parking.	Incrémenté ou décrétementé si une personne entre ou sort du parking en voiture pour se garer.
1 compteur pour les entrées/sorties de la galerie marchande.	Incrémenté ou décrétementé si une personne entre ou sort de la galerie marchande ou si elle entre ou sort de l'hypermarché.
1 compteur pour les entrées/sorties de l'hypermarché	Incrémenté ou décrétementé si une personne entre ou sort de l'hypermarché.

Afin de réaliser au mieux cette étude de cas, nous avons beaucoup travaillé sur les exemples d'utilisation du logiciel que nous avons pu trouver dans les dossiers d'installation. Nous nous sommes essentiellement concentrés sur l'exemple nommé "rpl-udp" qui consiste à montrer les échanges entre un serveur et des clients.

En partant de ce principe, nous avons considéré l'installation d'un serveur au sein de l'hypermarché ayant pour clients les capteurs installés aux entrées et aux sorties.

3. Notre réalisation

Voici à quoi ressemblé Cooja à la fin de notre développement :



Dans la fenêtre appelée "Network", on a une visualisation du projet ou l'on peut voir les communications qui s'effectuent entre les clients et le serveur (flèches bleues).

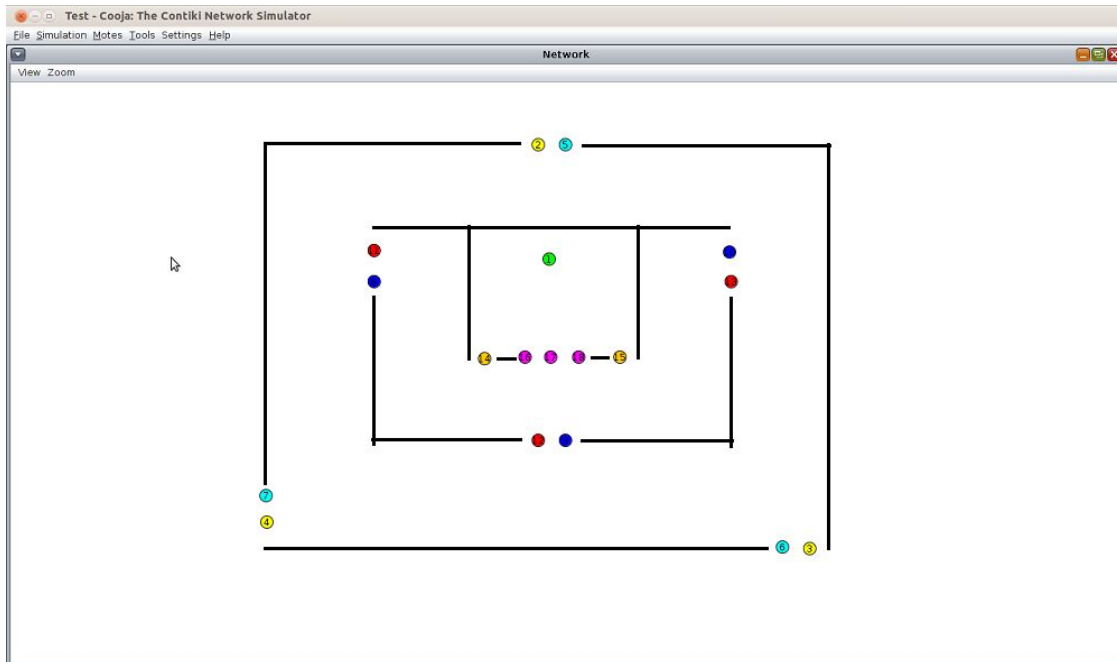
Dans la fenêtre "Mote Output", on peut voir les messages que s'échangent les clients et le serveur. Pour plus de clarté, nous avons mis uniquement ce que reçoit et envoie le serveur, car voir des "+1" et "-1" ne semblait pas très pertinent.

Lorsque le serveur reçoit un message, il affiche "Message bien reçu de X : "MESSAGE"" ou X correspond au numéro de mote du client et MESSAGE correspond au message envoyé par le client.

Par la suite, le serveur indique combien de voitures sont dans le parking, combien de personnes sont dans la galerie marchande et combien de personnes sont dans l'hypermarché.

Pour plus de détails, voir Annexe 3.

Ci-dessous, vous pouvez voir la représentation de notre hypermarché où chaque rond est une mote qui va permettre de compter.



En vert, au milieu, il y a le routeur de bordure, connecté au serveur.

Le jaune clair et le bleu clair sont respectivement les entrées et sorties du parking.

Le bleu foncé et le rouge sont respectivement les entrées et sorties de la galerie marchande.

Le jaune foncé est l'entrée de l'hypermarché.

Le rose est la caisse de l'hypermarché et donc les sorties.

Tous les nœuds, hormis le vert, sont des clients.

Notre code pour les parties client et la partie serveur est disponible en Annexe 5 et 6.

Annexes

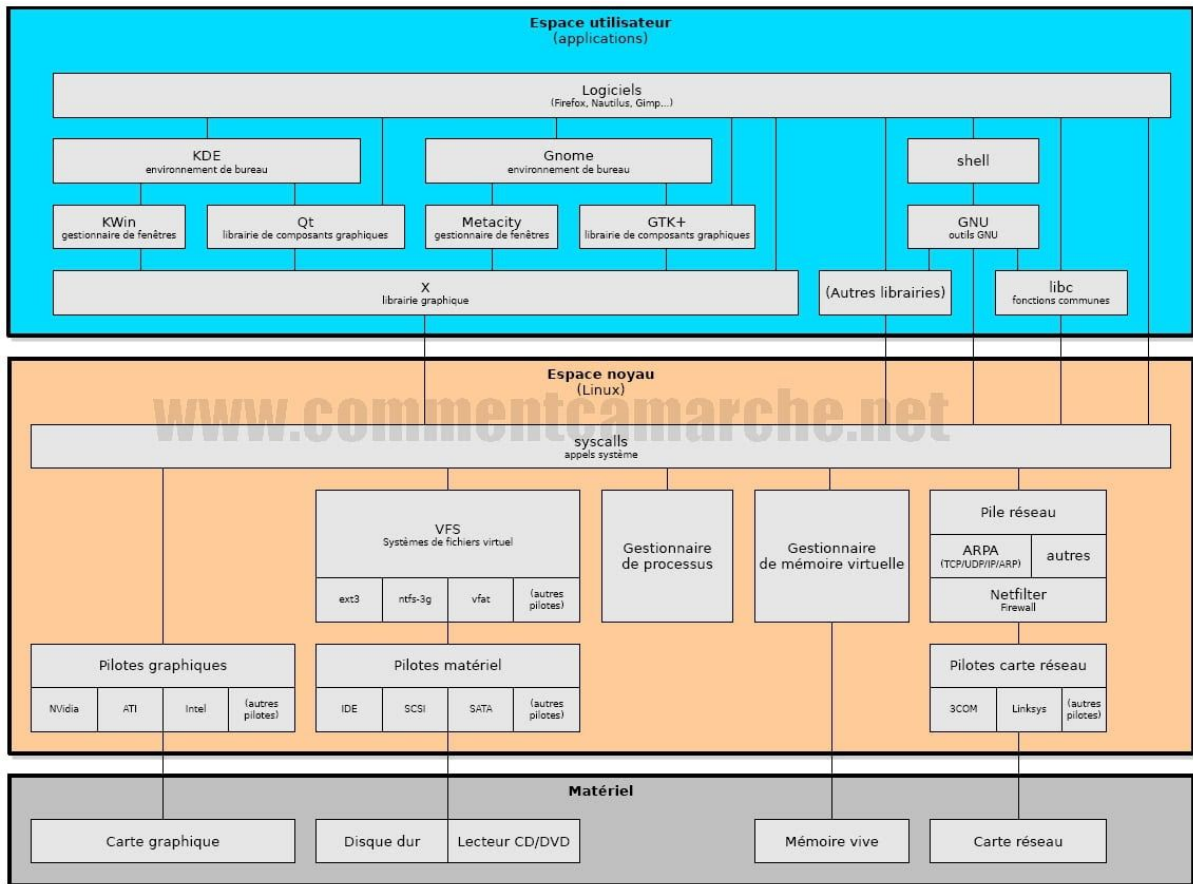


Figure 1 - Architecture du système d'exploitation Linux

Source : <https://www.commentcamarche.net/faq/9593-l-architecture-de-linux>

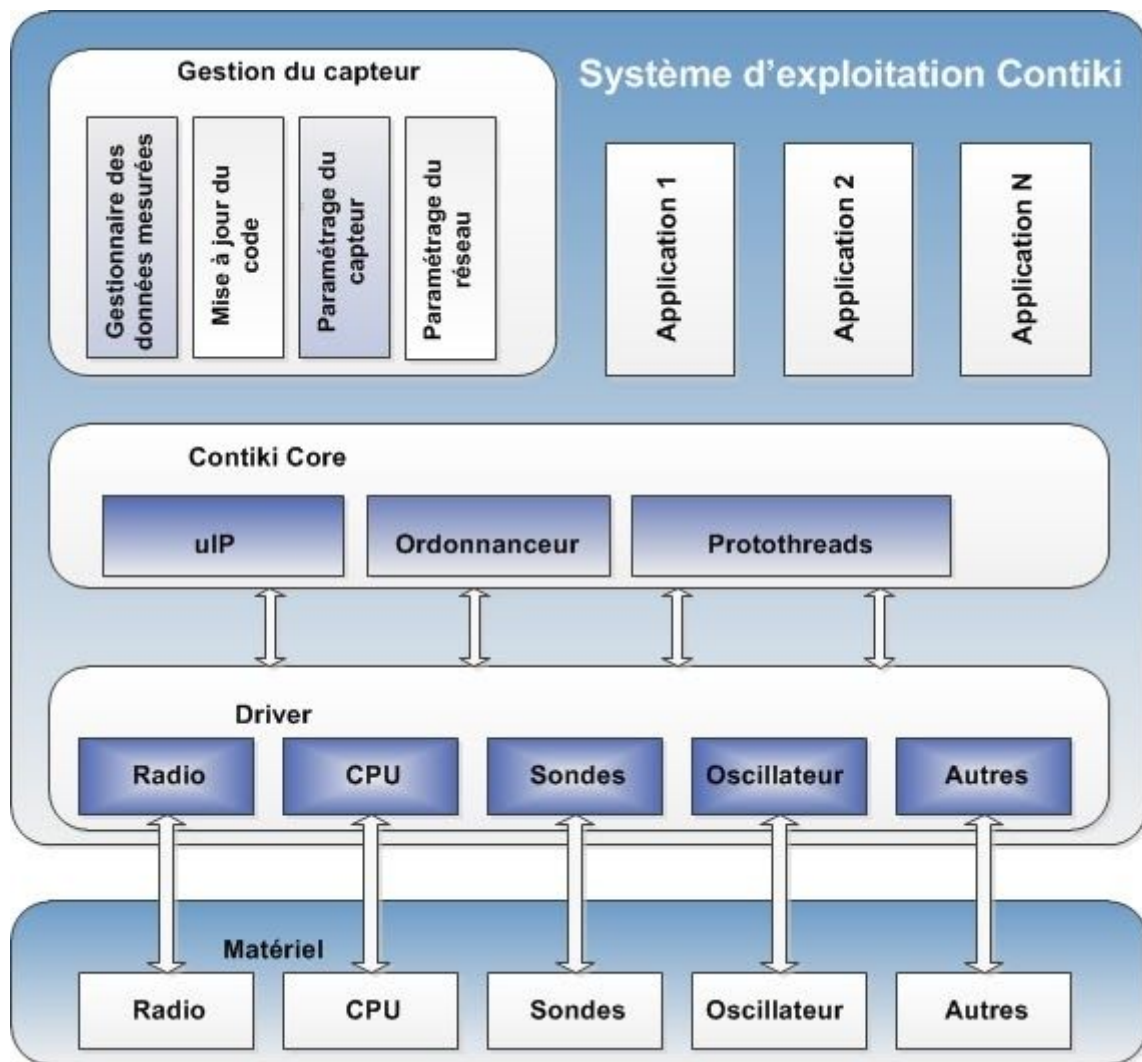


Figure 2 - Architecture du système d'exploitation Contiki.

Source : <https://fr.wikipedia.org/wiki/Contiki>

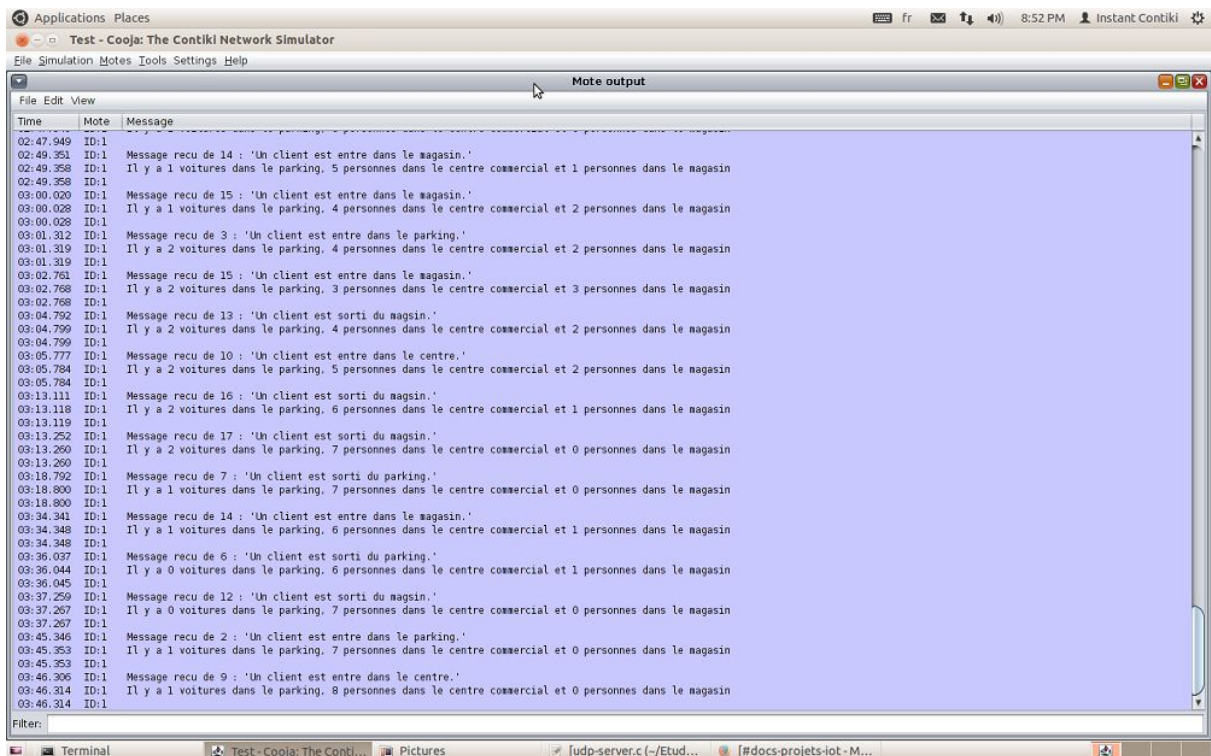


Figure 3 - Exemple des sorties lors d'un test de notre étude de cas.

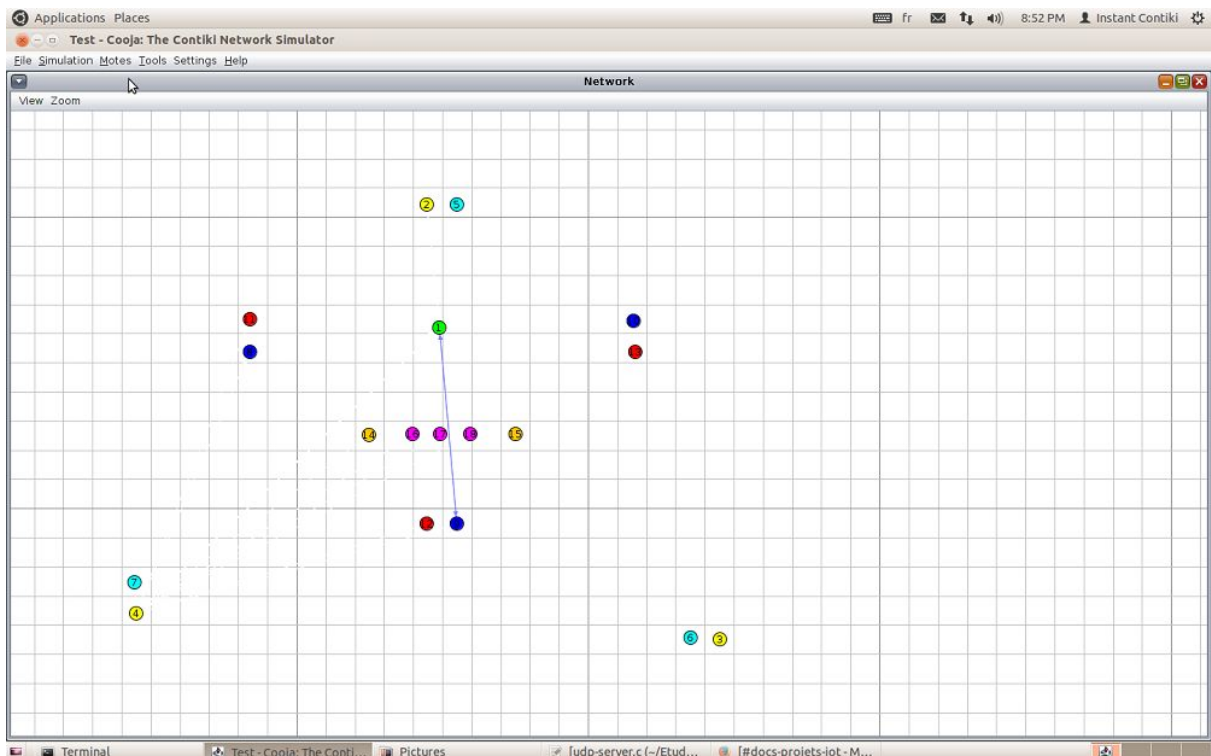


Figure 4 - Modélisation de notre étude de cas sur Cooja.


```

#include "contiki.h"
#include "lib/random.h"
#include "sys/ctimer.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "sys/ctimer.h"
#ifdef WITH_COMPOWER
#include "powertrace.h"
#endif
#include <stdio.h>
#include <string.h>

#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678

#define UDP_EXAMPLE_ID 190

#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#ifndef PERIOD
#define PERIOD 60
#endif

#define START_INTERVAL (15 * CLOCK_SECOND)
#define SEND_INTERVAL (PERIOD * CLOCK_SECOND)
#define SEND_TIME (random_rand() % (SEND_INTERVAL))
#define MAX_PAYLOAD_LEN 30

static struct uip_udp_conn *client_conn;
static uip_ipaddr_t server_ipaddr;

PROCESS(udp_client_process, "UDP client process");
AUTOSTART_PROCESSES(&udp_client_process);

static void tcpip_handler(void) {
    char *str;
    if(uip_newdata()) {
        str = uip_appdata;
        str[uip_datalen()] = '\0';
        printf("DATA recv '%s'\n", str);
    }
}

static void send_packet(void *ptr) {
    char buf[MAX_PAYLOAD_LEN];
    sprintf(buf, "centre e");
    uip_udp_packet_sendto(client_conn, buf, strlen(buf), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}

/*-----*/

static void print_local_addresses(void) {
    int i;
    uint8_t state;

    PRINTF("Client IPv6 addresses: ");
    for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
        state = uip_ds6_if.addr_list[i].state;
        if(uip_ds6_if.addr_list[i].isused &&
            (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
            PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
            PRINTF("\n");
            /* hack to make address "final" */
            if (state == ADDR_TENTATIVE) {
                uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
            }
        }
    }
}

```

```

    }
}
}

static void set_global_address(void) {
    uip_ipaddr_t ipaddr;
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0);
    uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
    uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

    /* The choice of server address determines its 6LoPAN header compression.
     * (Our address will be compressed Mode 3 since it is derived from our link-local address)
     * Obviously the choice made here must also be selected in udp-server.c.
     *
     * For correct Wireshark decoding using a sniffer, add the /64 prefix to the 6LowPAN protocol preferences,
     * e.g. set Context 0 to aaaa::. At present Wireshark copies Context/128 and then overwrites it.
     * (Setting Context 0 to aaaa::1111:2222:3333:4444 will report a 16 bit compressed address of
     * aaaa::1111:22ff:fe33:xxxx)
     *
     * Note the IPCMV6 checksum verification depends on the correct uncompressed addresses.
     */

    #if 0
    /* Mode 1 - 64 bits inline */
    uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 1);
    #elif 1
    /* Mode 2 - 16 bits inline */
    uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0xff, 0xfe00, 1);
    #else
    /* Mode 3 - derived from server link-local (MAC) address */
    uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0x0250, 0xc2ff, 0xfea8, 0xcd1a); //redbee-econotag
    #endif
}

PROCESS_THREAD(udp_client_process, ev, data) {
    static struct etimer periodic;
    static struct ctimer backoff_timer;
    #if WITH_COMPOWER
    static int print = 0;
    #endif

    PROCESS_BEGIN();
    PROCESS_PAUSE();
    set_global_address();
    PRINTF("UDP client process started\n");
    print_local_addresses();
    /* new connection with remote host */
    client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);

    if(client_conn == NULL) {
        PRINTF("No UDP connection available, exiting the process!\n");
        PROCESS_EXIT();
    }

    udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

    PRINTF("Created a connection with the server ");
    PRINT6ADDR(&client_conn->ripaddr);
    PRINTF(" local/remote port %u/%u\n",
        UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));

    #if WITH_COMPOWER
    powertrace_sniff(POWERTRACE_ON);
    #endif

    etimer_set(&periodic, SEND_INTERVAL);
    while(1) {
        PROCESS_YIELD();

```

```

if(ev == tcpip_event) {
    tcpip_handler();
}

if(etimer_expired(&periodic)) {
    etimer_reset(&periodic);
    ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);

#ifdef WITH_COMPOWER
    if (print == 0) { powertrace_print("#P"); }
    if (++print == 3) { print = 0; }
#endif
}
}
PROCESS_END();
}

```

Figure 5 - Code des clients de notre étude de cas.

```

#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/uip.h"
#include "net/rpl/rpl.h"

#include "net/netstack.h"
#include "dev/button-sensor.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#define UIP_IP_BUF ((struct uip_ip_hdr *)&uip_buf[UIP_LLH_LEN])

#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678

#define UDP_EXAMPLE_ID 190

int PARKING = 0;
int CENTRE = 0;
int MAGASIN = 0;

static struct uip_udp_conn *server_conn;

PROCESS(udp_server_process, "UDP server process");
AUTOSTART_PROCESSES(&udp_server_process);

static void tcpip_handler(void) {
    char *appdata;

    if(uip_newdata()) {
        appdata = (char *)uip_appdata;
        appdata[uip_datalen()] = 0;

        if(strcmp(appdata, "magasin e") == 0 && CENTRE > 0){
            MAGASIN += 1;
            CENTRE -= 1;
            PRINTF("Message reçu de %d : 'Un client est entre dans le magasin.' \n",
UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
            PRINTF("Il y a %d voitures dans le parking, %d personnes dans le centre commercial et %d
personnes dans le magasin \n", PARKING, CENTRE, MAGASIN);
            PRINTF("\n");
        }else if (strcmp(appdata, "magasin s") == 0 && MAGASIN > 0){
            MAGASIN -= 1;
            CENTRE += 1;
            PRINTF("Message reçu de %d : 'Un client est sorti du magsin.' \n",
UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
            PRINTF("Il y a %d voitures dans le parking, %d personnes dans le centre commercial et %d
personnes dans le magasin \n", PARKING, CENTRE, MAGASIN);
            PRINTF("\n");
        }

        if (strcmp(appdata, "centre e") == 0){
            CENTRE += 1;
            PRINTF("Message reçu de %d : 'Un client est entre dans le centre.' \n",
UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
            PRINTF("Il y a %d voitures dans le parking, %d personnes dans le centre commercial et %d
personnes dans le magasin \n", PARKING, CENTRE, MAGASIN);
            PRINTF("\n");
        }else if (strcmp(appdata, "centre s") == 0 && CENTRE > 0){
            CENTRE -= 1;
            PRINTF("Message reçu de %d : 'Un client est sorti du centre.' \n",
UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);

```

```

        PRINTF("Il y a %d voitures dans le parking, %d personnes dans le centre commercial et %d
personnes dans le magasin \n", PARKING, CENTRE, MAGASIN);
        PRINTF("\n");
    }

    if (strcmp(appdata, "parking e") == 0){
        PARKING += 1;
        PRINTF("Message reçu de %d : 'Un client est entre dans le parking.' \n",
UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
        PRINTF("Il y a %d voitures dans le parking, %d personnes dans le centre commercial et %d
personnes dans le magasin \n", PARKING, CENTRE, MAGASIN);
        PRINTF("\n");
    }else if (strcmp(appdata, "parking s") == 0 && PARKING > 0){
        PARKING -= 1;
        PRINTF("Message reçu de %d : 'Un client est sorti du parking.' \n",
UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
        PRINTF("Il y a %d voitures dans le parking, %d personnes dans le centre commercial et %d
personnes dans le magasin \n", PARKING, CENTRE, MAGASIN);
        PRINTF("\n");
    }

    #if SERVER_REPLY
        PRINTF("DATA sending reply\n");
        uip_ipaddr_copy(&server_conn->ripaddr, &UIP_IP_BUF->srcipaddr);
        uip_udp_packet_send(server_conn, "Reply", sizeof("Reply"));
        uip_create_unspecified(&server_conn->ripaddr);
    #endif
}

/*-----*/
static void print_local_addresses(void) {
    int i;
    uint8_t state;

    PRINTF("Server IPv6 addresses: ");
    for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
        state = uip_ds6_if.addr_list[i].state;
        if(state == ADDR_TENTATIVE || state == ADDR_PREFERRED) {
            PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
            PRINTF("\n");
            /* hack to make address "final" */
            if (state == ADDR_TENTATIVE) {
                uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
            }
        }
    }
}

PROCESS_THREAD(udp_server_process, ev, data) {
    uip_ipaddr_t ipaddr;
    struct uip_ds6_addr *root_if;

    PROCESS_BEGIN();

    PROCESS_PAUSE();

    SENSORS_ACTIVATE(button_sensor);

    PRINTF("UDP server started\n");

    #if UIP_CONF_ROUTER
        /* The choice of server address determines its 6LoPAN header compression.
        * Obviously the choice made here must also be selected in udp-client.c.
        *
        * For correct Wireshark decoding using a sniffer, add the /64 prefix to the 6LowPAN protocol preferences,
        * e.g. set Context 0 to aaaa::. At present Wireshark copies Context/128 and then overwrites it.
        * (Setting Context 0 to aaaa::1111:2222:3333:4444 will report a 16 bit compressed address of

```

```

aaaa::1111:22ff:fe33:xxxx)
 * Note Wireshark's IPCMV6 checksum verification depends on the correct uncompressed addresses.
 */

#ifdef 0
/* Mode 1 - 64 bits inline */
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from link local (MAC) address */
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
    uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
#endif

uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
root_if = uip_ds6_addr_lookup(&ipaddr);
if(root_if != NULL) {
    rpl_dag_t *dag;
    dag = rpl_set_root(RPL_DEFAULT_INSTANCE, (uip_ip6addr_t *)&ipaddr);
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
    rpl_set_prefix(dag, &ipaddr, 64);
    PRINTF("created a new RPL dag\n");
} else {
    PRINTF("failed to create a new RPL DAG\n");
}
#endif /* UIP_CONF_ROUTER */

print_local_addresses();
/* The data sink runs with a 100% duty cycle in order to ensure high
   packet reception rates. */
NETSTACK_MAC.off(1);

server_conn = udp_new(NULL, UIP_HTONS(UDP_CLIENT_PORT), NULL);
if(server_conn == NULL) {
    PRINTF("No UDP connection available, exiting the process!\n");
    PROCESS_EXIT();
}
udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));

PRINTF("Created a server connection with remote address ");
PRINT6ADDR(&server_conn->ripaddr);
PRINTF(" local/remote port %u/%u\n", UIP_HTONS(server_conn->lport),
        UIP_HTONS(server_conn->rport));

while(1) {
    PROCESS_YIELD();
    if(ev == tcpip_event) {
        tcpip_handler();
    } else if (ev == sensors_event && data == &button_sensor) {
        PRINTF("Initiaing global repair\n");
        rpl_repair_root(RPL_DEFAULT_INSTANCE);
    }
}
PROCESS_END();
}

```

Figure 6 - Code du serveur de notre étude de cas.

Source pour la partie explicative de Contiki :

<https://fr.wikipedia.org/wiki/Contiki#Comparaison>

https://fr.wikipedia.org/wiki/M%C3%A9moire_morte

http://pficheux.free.fr/pub/tmp/03_Contiki.pdf

<http://courses.cs.vt.edu/cs5204/fall09-kafura/Presentations/Threads-VS-Events.pdf>

Source pour la partie explicative de Cooja :

http://anrg.usc.edu/contiki/index.php/Cooja_Simulator

<https://fr.m.wikipedia.org/wiki/Fichier:Contiki-ipv6-rpl-cooja-simulation.png>

<https://spectrum.ieee.org/telecom/internet/the-internet-of-things-is-full-of-motes-domotics-and-bans>