# TD : Routage RPL avec Contiki OS

## 1. Principes de fonctionnement du protocole RPL

RPL was designed with the objective to meet the requirements spelled out in RFC5867, RFC5826, RFC5673, and RFC5548.
In order to be useful in a wide range of LLN application domains, RPL separates packet processing and forwarding from the routing optimization objective. Examples of such objectives includes minimizing energy, minimizing latency, or satisfying constraints. A RPL implementation, in support of a particular LLN application, will include the necessary Objective Function(s) as required by the application.
RPL operations require bidirectional links. In some LLN scenarios, those links may exhibit asymmetric properties. It is required that the reachability of a router be verified before the router can be used as a parent. RPL expects an external mechanism to be triggered during the parent selection phase in order to verify link properties and neighbor reachability.
RPL also expects an external mechanism to access and transport some control information, referred to as the "RPL Packet Information", in data packets. RPL provides a mechanism to disseminate information over the dynamically formed network topology. This dissemination enables minimal configuration in the nodes, allowing nodes to operate mostly autonomously.
In particular, RPL may disseminate IPv6 Neighbor Discovery (ND) information such as the RFC4861 Prefix Information Option (PIO) and the RFC4191 Route Information Option (RIO). ND information that is disseminated by RPL conserves all its original semantics for router to host, with limited extensions for router to router, though it is not to be confused with routing advertisements and it is never to be directly redistributed in another routing protocol. A RPL node often combines host and router behaviors. As a host, it will process the options as specified in RFC4191, RFC4861, RFC4862, and RFC6275. As a router, the RPL node may advertise the information from the options as required for the specific link.
For further information, please refer to **RFC 6550**, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks".

## 2. Mise en place et test

1. Ce TD utilise **`Contiki-OS`** et **`Cooja`** qui sont pré-installés dans une machine virtuelle. Récupérez la VM, composée de trois fichiers .vmdk (image du disque virtuel d'origine), .mf (empreinte SHA256 des fichiers) et .ovf (configuration de la VM) situés dans **`/net/stockage/dmagoni/contiki2.7`** et copiez les sur votre machine locale dans **`/espace/<votre-login>`**. Créez un sous dossier **`/espace/<votre-login>/vbox`**.
2. Importez l'image ci-dessus dans **Virtualbox**, grâce au fichier .ovf, en mode expert en configurant le chemin du disque virtuel sur **`/espace/<votre-login>/vbox/`** afin que la VM soit installée en local sur la machine.
3. Testez Contiki-OS en compilant un petit programme
4. Lancez la fenêtre du **terminal**.
5. Allez dans le dossier contenant l'exemple hello world :

```
cd /examples/hello-world
```

6. Compilez le code pour la plateforme native (utilisée lorsqu'aucune **mote** n'est connectée à l'ordinateur).

```
make TARGET=native
```

7. Une fois la compilation terminée, lancez le programme Hello World.

```
./hello-world.native
```

8. Vous devez voir le texte similaire suivant sur le terminal :

```
Contiki 3.0 started with IPV6, RPL
Rime started with address 1.2.3.4.5.6.7.8
MAC nullmac RDC nullrdc NETWORK sicslowpan
Tentative link-local IPv6 address fe80:0000:0000:0000:0302:0304:0506:0708
Hello, world
```

9. Le code paraîtra suspendu, cependant il tourne toujours sur Contiki, mais il ne produit aucune sortie car le programme Hello World est fini. Pressez Ctrl-C pour quitter.

## 3. Exemple de Broadcast

Cet exemple montre comment programmer une communication de type **Broadcast** à l'aide de la stack Rime (http://contiki.sourceforge.net/docs/2.6/a01798.html) et comment encapsuler des données dans des paquets.

1. Dans la VM, lancez **`Cooja`** en double-cliquant sur l'icône sur le bureau.
2. Pour créer une nouvelle simulation, cliquez sur le menu **File**, choisissez '**New Simulation**', nommez la simulation et cliquez sur '**Create**'.
3. La nouvelle simulation montre une fenêtre '**Network**' vide en haut à gauchede l'écran car nous n'aons ajouté aucune **mote** (équipement nœud IoT) à notre simulation. La fenêtre '**Simulation Control**' possède des options pour **Start**, **Pause** et **Reload** la simulation. La fenêtre '**Notes**' permet d'ajouter des notes à notre simulation. La fenêtre '**Mote Output**' affiche toutes les activités de tous les ports série des motes connectées à la simulation. La fenêtre '**Timeline**' fournit de l'information en temps réel sur les évènements qui ont lieu dans la simulation.
4. Pour ajouter des Motes à la simulation, cliquez sur `Motes -> Add motes -> Create a new mote type ->` et selectionnez le type de mote à ajouter au réseau.
5. Dans la fenêtre '**Create Mote Type**', choisissez un nom approprié pour le type de mote. Dans l'option '**Contiki process/Firmware**', suivre le chemin `examples/ipv6/simple-udp-rpl` et selectionnez le programme `broadcast-example.c`. Compilez et cliquez sur '**Create**'.
6. Dans la fenêtre '**Add mote**' qui apparaît, entrez le nombre de motes, changez l'intervalle de position selon vos besoins et cliquez sur '**Add motes**'. Le nombre de motes ajouté au réseau est maintenant montré dans la fenêtre '**Network**'.
7. Pour démarrer la simulation, cliquez sur '**Start**' dans la fenêtre '**Simulation Control**'.
8. Pendant que la simulation s'exécute, la fenêtre '**Mote output**' imprime les information s sur les émetteurs et les récepteurs.
9. L'onglet **View** dans la fenêtre **Network** peut être utilisé pour montrer les caractéristiques des motes telles que :
    - Mote Ids - affiche l'id du mote
    - Addresses: IP or Rime - affiche l'adresse IPv6 du mote
    - Radio Traffic- affiche une animation montrant les communications entre les motes
    - Positions – affiche les coordonnées du mote
    - Radio Environment – cliquer sur un mote particulier affiche sa zone de couverture

## Explication du code

L'objectif de cet exemple est de tester la couche de diffusion (broadcast layer) dans **Rime**. Un coup d'oeil à `core/net/rime/broadcast.h` et `core/net/rime/broadcast.c` aide à comprendre les concepts sous-jacents de cet exemple. Dans le code, un processus appelé `example_broadcast_process` est démarré par AUTOSTART_PROCESSES.

### Macros et Structures

```
PROCESS(name,strname)
```

Chaque process doit être défini via la macro PROCESS. PROCESS possède deux arguments: la variable de la structure process, et un nom lisible dans une chaîne de caractère, utilisée pour le debugging.

- name: The variable name of the process structure.
- strname: The string representation of the process name.

```
AUTOSTART_PROCESS(struct process &)
```

AUTOSTART_PROCESSES démarre automatiquement les processus donnés en argument(s) lorsque le module boote.

- &name: Reference to the process name

```
broadcast_recv(struct broadcast_conn *, const rimeaddr_t *)
```

Cette fonction parse un paquet entrant et affiche le message et l'adresse de l' émetteur. En la définissant comme la fonction callback designée du broadcast, broadcast_recv est automatiquement appelée lorsqu'un paquet est reçu.

- broadcast_conn *: This structure which has 2 structures : abc_conn, broadcast_callbacks *. The abc_conn is basic type of connection over which the broadcast connection is developed. And, the broadcast_callbacks point to recv and sent functions (in this example, just recv).
- rimeaddr_t *: This is a union which has a character array u8[RIMEADDR_SIZE].

```
PROCESS_THREAD(name, process_event_t, process_data_t)
```

Un processus dans Contiki consiste en une référence unique à un "protothread". Cette fonction est utilisée pour definir le protothread d'un processus. Le processus est appelé lorsqu'un évènement à lieu dans le système. Chaque processus dans le module nécessite un gestionnaire d'évènement (event handler) sous la macro PROCESS_THREAD.

- name: The variable name of the process structure.
- process_event_t: The variable of type character.If this variable is same as PROCESS_EVENT_EXIT then PROCESS_EXITHANDLER is invoked.

Dans le corps de PROCESS_THREAD il y a trois tâches principales (major tasks) :

- Initialisation
  - allocation des ressources
  - definition des variables
  - démarrage du processus
- Boucle infinie
  - while(1) est utilisé pour créer une boucle infinie dans laquelle la réponse effective à un évènement (event-handling response) a lieu

- Désallocation
    - terminaison du processus
    - désallocation des ressources

`PROCESS_EXITHANDLER(handler)`

Définit une action lorsqu'un processus termine (exits). NOTE: cette declaration doit venir immediatement avant la macro PROCESS_BEGIN().

- handler: The action to be performed.

`PROCESS_BEGIN()`

Cette macro definit le début d'un processus, et doit toujours apparaître dans une definition de PROCESS_THREAD(). Cette macro initie PT_BEGIN(), qui est declarée au point de départ d'un protothread. Toutes les instructions C situées avant l' invocation de PT_BEGIN() seront executées à chaque fois que le protothread est schedulé.

`broadcast_close(struct broadcast_conn *)`

Cette fonction ferme une connection broadcast qui a été préalablement ouverte avec broadcast_open(). Cette fonction est typiquement appelée comme gestionnaire de sortie (exit handler).

- broadcast_conn : This is same as the variable from broadcast_recv().

`PROCESS_END()`

Cette macro definit la fin d'un processus. Elle doit apparaitre dans la definition d'un PROCESS_THREAD() et doit toujours être incluse. Le processus termine lorsque la macro PROCESS_END() est atteinte. Cette macro initie PT_END(). Elle doit toujours être utilisée avec une macro PT_BEGIN() correspondante.

```
broadcast_open(struct broadcast_conn *,
    uint16_t, const struct broadcast_callbacks *)
```

Initie une connexion **broadcast** identifiée de type *best-effort*. L'appelant va allouer de la mémoire pour la structure `struct broadcast_conn`, habituellement en la déclarant comme une variable statique. Le pointeur `struct broadcast_callbacks *` pointe vers une structure contenant un pointeur vers une (ou plusieurs) fonction qui sera appelée lorsqu'un paquet arrive sur le canal. Cette fonction ouvre une connexion du type `abc_conn`. Aussi, elle pointe vers la fonction `channel_set_attributes()`.

- broadcast_conn : A pointer to a struct broadcast_conn
- uint16_t: The channel on which the connection will operate
- broadcast_callbacks : A struct broadcast_callbacks with function pointers to functions that will be called when a packet has been received

`etimer_set(struct etimer *, clock_time_t)`

Cette fonction est utilisée pour fixer un timer d'évènement pour un temps situé quelque part dans le futur. Lorsque le timer expire, l'évènement PROCESS_EVENT_TIMER sera envoyé (posted) au processus qui a appelé la fonction.

- etimer : A pointer to the event timer
- clock_time_t : The interval before the timer expires.

```
static struct broadcast_conn
```

Le module **broadcast** envoie un paquet à tous les voisins de la zone locale avec un entête qui identifie l'émetteur. Il ajoute aussi une adresse single-hop, comme attribut de paquet, aux paquets sortants. La structure `broadcast_conn` consiste en deux structures

- abc_conn struct: the abc(Anonymous Best-effort local area Broadcast) module sends packets to all local area neighbors. It uses one channel.
- broadcast_callbacks struct: this is called when a packet has been received by the broadcast module. The struct broadcast_callbacks pointer is used in broadcast_open to point to a function that will be called when a packet arrives on the channel.

# 4. Utilisation de RPL

Description de la pile réseau de Contiki :
http://anrg.usc.edu/contiki/index.php/Network_Stack

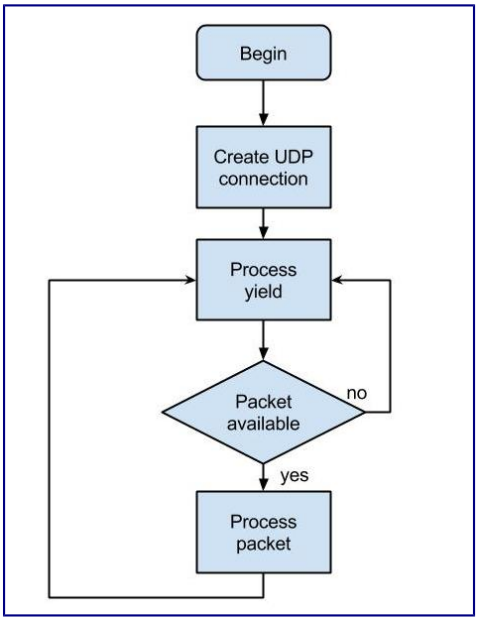
RPL est le protocole de routage IPv6 pour les réseaux LLN (Low-power Lossy Networks : réseaux à basse puissance avec pertes). LLNs sont une classe de réseaux dans lesquels les routeurs et les hôtes sont constraints. Les routeurs LLN opèrent typiquement avec des contraintes sur la puissance de calcul, la mémoire, et l'énergie. RPL fournit un mécanisme permettant le trafic multipoint-à-point depuis les équipements (devices) dans le LLN vers un point de contrôle central, de même que pour le trafic point-à-multipoint du point de contrôle central vers les devices dans le LLN. Le support du trafic point-à-point est aussi disponible.

Dans cet exemple, UDP est implémenté au dessus de RPL. Un LLN possède un serveur UDP, qui accepte les paquets disponibles, et plusieurs clients UDP, qui envoient des paquets périodiquement au serveur au travers d'un saut simple (single-hop) ou de sauts multiples (multi-hops).

Code source :

```
~/contiki-2.7/examples/ipv6/rpl-udp/udp-server.c
~/contiki-2.7/examples/ipv6/rpl-udp/udp-client.c
~/contiki-2.7/core/net/tcpip.c
~/contiki-2.7/core/net/tcpip.h
```

# Serveur UDP




Flow chart du serveur UDP

Dans l'exemple, le serveur UDP effectue trois tâches principales.
1. Initialiser le DAG RPL ;
2. Etablir une connexion UDP ;
3. Attend les paquets des clients, les reçoit et les affiche sur **stdout**.

## Initialisation du DAG RPL

```
// check whether the ADDR_MANUAL was set succefuly or not
uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
root_if = uip_ds6_addr_lookup(&ipaddr);
if(root_if != NULL) {
  rpl_dag_t *dag;
  //set the ip adress of server as the root of initial 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");
}
```

## Etablir une connexion UDP

```
//create new UDP connection to client's port
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();
}
//bing the connection to server's local port
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));
```

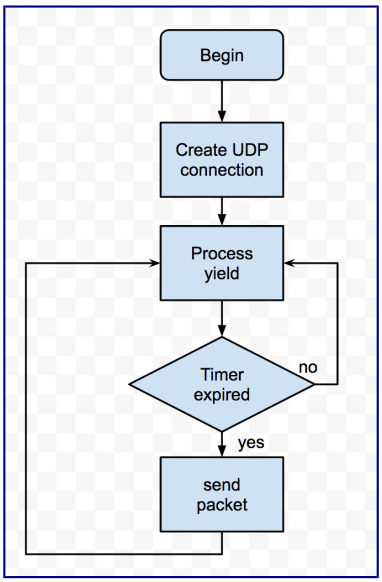
### Recevoir et traiter les paquets entrants

```
while(1) {
  PROCESS_YIELD();
  //if there is packet available
  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);
  }
}

//call this function if packet available
static void
tcpip_handler(void)
{
  char *appdata;

  if(uip_newdata()) {
    appdata = (char *)uip_appdata;
    appdata[uip_datalen()] = 0;
    //print the data of packet
    PRINTF("DATA recv '%s' from ", appdata);
    PRINTF("%d",
           UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
    PRINTF("\n");
}
```

## Client UDP




Flow chart du client UDP

Dans l’exemple, le client UDP effectue deux tâches principales.
1. Initialise une connexion UDP ;
2. Envoyer des paquets au serveur UDP periodiquement.

### Etablir une connexion UDP

```
/* 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));
```

### Envoyer des paquets

```
//set time interval by SEND_INTERVAL
etimer_set(&periodic, SEND_INTERVAL);
while(1) {
  PROCESS_YIELD();
  if(ev == tcpip_event) {
    tcpip_handler();
  }
  //send packet every SEND_INTERVAL
  if(etimer_expired(&periodic)) {
    etimer_reset(&periodic);
    ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);

static void
send_packet(void *ptr)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN];

  seq_id++;
  PRINTF("DATA send to %d 'Hello %d'\n",
         server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], seq_id);
  sprintf(buf, "Hello %d from the client", seq_id);
  //send packet through client_conn to UDP server
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                        &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
```

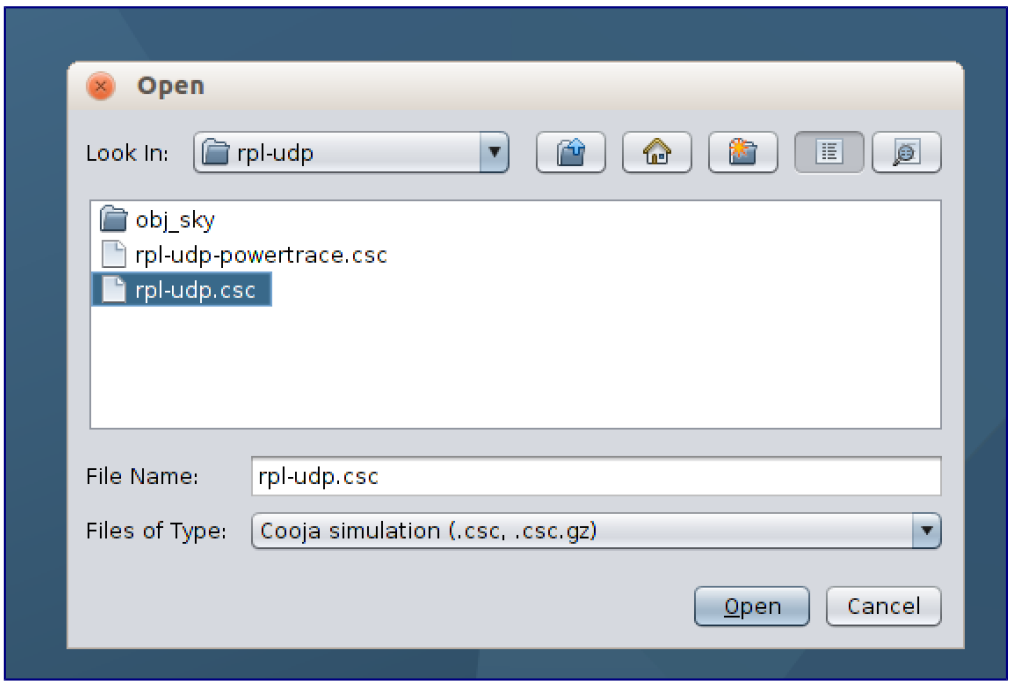
# 5. Simulation avec Cooja

Le modèle DGRM est utilisé.
Les étapes suivantes décrivent la création d’une nouvelle simulation :

#### 1. Démarrer Cooja

Pour démarrer le simulateur, allez dans le dossier Contiki, naviguez vers le dossier `/tools/cooja` et exécutez ‘**ant’** pour lancer le simulateur :

$ sudo ant run




**2. Ouvrir un fichier d'une simulation existante**

Dans l'interface graphique, selectionnez `File->Open simulation->Browse...`

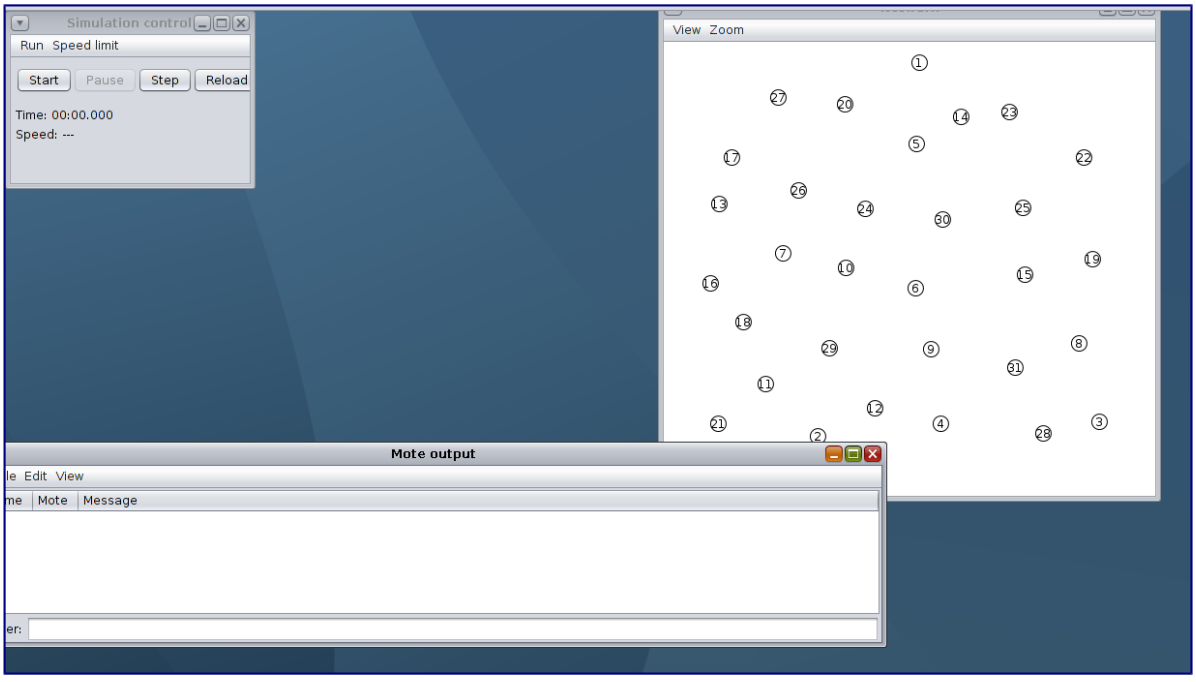
Une fois la boite de dialogue ouverte, ouvrez :

```
home/contiki-2.7/examples/ipv6/rpl-udp/rpl-udp.csc
```

***Note***: si une erreur de compilation apparait, lancez :

```
$ cd contiki/examples/ipv6/rpl-udp
$ make
```

Vous devriez voir apparaître la simulation comme ci-dessous :

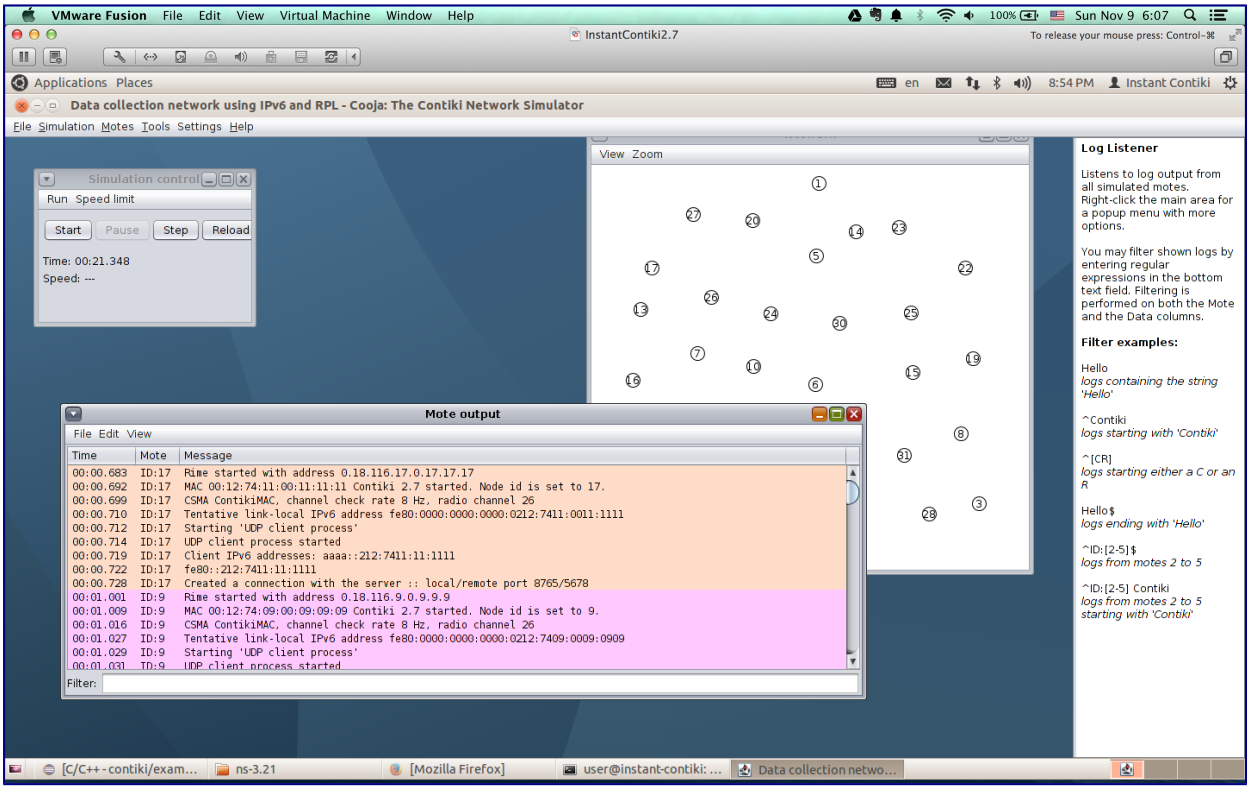


### 3. Exécuter la simulation

Lancez la simulation en utilisant l'option *Start* dans la fenêtre *Simulation Control*. Cela va initialiser les motes et leur allouer de nouvelles adresses **Rime** ainsi que d'autres processus d'initialisation.

### 4. Sortie

Les messages de sortie et de debug générés par les motes peuvent être vus dans la fenêtre *Motes Output*. Vous pouvez filtrer les sorties avec l'ID d'un nœud : *node_id* afin d'observer un nœud particulier. Vous pouvez aussi observer des messages de debug particulier en les filtrant. D'autres fonctions utiles des *Motes Output* sont *File*, *Edit* et *View*. L' option *File* permet de sauver la sortie dans un fichier. L' option *Edit* permet de copier la sortie – complète ou bien uniquement les messages sélectionnés. Vous pouvez aussi effacer les messages en utilisant l' option *Clear all messages*.
Les messages de sortie sauvegardés dans un fichier peuvent être utilisésultérieurement pour effectuer des observations et tracer des graphiques selon les objectifs de l'expérience.




### 5. Etude à réaliser

1. A l'aide des traces des messages envoyés par les motes visualisées dans **Network** avec une vitesse ralentie et dans **Mote Output** et **Radio Messages**, determinez l'arbre de routage du réseau. Donnez les valeurs du rang de chaque mote.
2. Supprimez 4 motes au hasard parmi celles situées au centre du réseau.
3. A l'aide des traces des messages envoyés par les motes restantes visualisées dans **Network** avec une vitesse ralentie et dans **Mote Output** et **Radio Messages**, determinez le nouvel arbre de routage du réseau.Donnez les valeurs du rang de chaque mote.
4. Que peut on en déduire sur la construction du **DODAG** ?