# ALARM HOUSE WITH 6LoWPAN

Advanced Network Architectures and Wireless Systems
MSc. Computer Engineering, University of Pisa

Damiano Barone, Lorenzo Diana

A.Y 2015/2016

## 1 Introduction

We have implemented a house alarm system with a sensor network. We have used the library of Contiki and Californium. We also have implemented a Servlet with Tomcat server for web interface with the house owner. Through the web page the owner can monitoring the battery of sensors and activate/deactivate the alarm. If the system detect a intruder sends a message through a bot of telegram in the smart phone of the owner.

## 2 Architecture

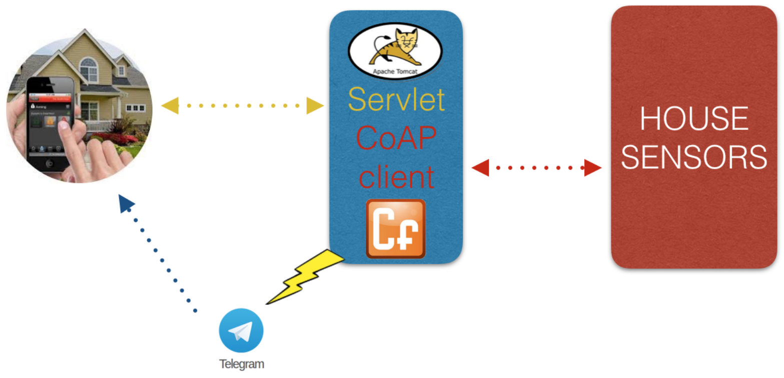

Figure 1: Architecture

### 2.1 House Sensors

Our network consist of a border router (BR), a guardian node and a variable number of generic node. Border router allow comunication between 6LoWPan network and the servlet on tomcat server. The guardian node exposes the alarm resource, this resource rapresent the alarm status and is observed by the servlet on tomcat server. Each generic node exposes the battery resource to inform the servlet about its battery status and can simulate the IR sensors by its button. The border router is the dag root and creates the instance RPL. The nodes comunicate with the java application with CoAP messages. When a infrared sensor detects that somebody has crossed the door, it sends an UDP message to the guardian node and waits an ack from it. If after a time t, it doesn't receive the ack, it resends the UDP message until it has not received an ack. When the guardian node receives a message from a node, it replies with ACK, changes the alarm variable and notify the change to the java client. So when the variable changes, the java application detects immediately the change.

## 2.2 Server

The server is a java application (servlet) that run on Tomcat server. When the server starts, it creates a thread that connect to the 6LoWPAN to monitors the battery status of the sensors and the alarm status. Each time the user connects to the server, a web page is displayed.
The web page contains information about the guardian node and all of the generic nodes, for each node are presented the battery level and the sensor status. The status indicate whether the node is working properly ("Active"), the battery is low ("Low batt."), the IR sensor is tripped ("Intruders") or the node is unreachable ("DEAD").
In the web page there is also a button to enable/disable the alarm. The button, when clicked, send a CoAP put message to the guardian node with the command for enable/disable alarm.
The web page refresh itself every *tot* seconds.
In background the servlet keep track of the battery level of the nodes, performing a CoAP get message every $n$ seconds, and the alarm status by an observe relation with the alarm resource on the guardian node.

## 2.3 WebApp

Any smartphone can connect to a Servlet with browser and through the web interface can manage the alarm system. When a user connects to a Servlet download a web page containing all status of battery percentage of all sensors. The user through a button can also send a request POST to activate or disable the alarm.

## 2.4 Telegram

When the server detects intruders at home or a sensor has a battery status at 10% or a node becomes unreachable, it sends a message to the user by a bot of telegram.

# 3 RPL - Integrity

An alarm system needs the integrity of a network, if a node is down, the network must immediately notice it. Contiki doesn't implement a mechanism of fasting notice if a node is down, so we have to change the mechanism of RPL in Contiki. We want to exploit the DIO message to notice if a node is down. Therefore we must force the algorithm trickle. The algorithm trickle was thought for increasing the interval to send a DIO messages in time, because they want to decrease the consumption of sensors, but if we need the integrity, the sensors must send a message periodically (at fix intervals) to notice to the neighbours node that they are still alive. This obviously increases the power consumption. Contiki in DIO messages sends a parameter for the configuration of the trickle, so we have used this function to force the trickle to have fix intervals. When the Border router activates the alarm, it sends DIO messages in broadcast with specific parameter for the trickle, with very big K and initial interval around 2 seconds. So we have fix intervals of 2 seconds. The node that receives the DIO message, sets its interval parameter and forwards the message to the neighbours. Further the parameter of trickle we send another data, the flag to activate the alarm. The node needs this flag because when the flag is active, it monitors all neighbours with greater ranks than its. Each node monitors the nodes under itself in the tree, so the node hasn't too much information about its neighbours. Every node has a table with the ip and a timer for each neighbour that have a greater rank than itself. If the timer associated to the neighbour expires, it means that the neighbour doesn't send a DIO message by a time timer. So the node sends a DIS to that neighbour (solicitation to DIO message) and waits for another timer interval, if the timer expires again the node sends a message to a border router that the node is dead. But if the node that receives the DIS message is not dead, it replies in broadcast the DIO message and every node which has this node in the table resets the timer.

## 3.1 Code

We created a MACRO to activate our function of integrity on contiki the name is:

```
KEEP_ALIVE_GUARD
```

By just including this macro in the makefile you enable the integrity. The files of RPL are in the folder:

```
contiki/core/net/rpl
```

The list of file that we have changed:

- rpl-conf.h
- rpl-dag-root.h
- rpl-dag.c
- rpl-timers.c
- rpl.h
- rpl.c
- rpl-icmp6.c

In detail:

### 3.1.1 rpl-conf.h

Public configuration and API declarations for ContikiRPL.
In this file we declared the constant used by KAG. The number of entries for the neighbour table and the parameters for the trickle sets when KAG is actived. The trickle parameters are broadcasted via a particular DIO option of contiki. (line 287)

```
#ifdef KEEP_ALIVE_GUARD
#define RPL_KAG_MAX_NEIGHBOUR_WITH_LOWER_RANCK 10    /* ↩
    number NWLR entries */
#define RPL_KAG_INTMIN 11 /* interval for DIO messages when ↩
    KAG is active */
#define RPL_KAG_REDUNDANCY ↩
    RPL_KAG_MAX_NEIGHBOUR_WITH_LOWER_RANCK*10 /* redundancy ↩
    parameter for trickle when KAG is active. We supppose ↩
    that the only 1/10 of out neighbours have rank greater ↩
    than our */
#endif /* KEEP_ALIVE_GUARD */
```

### 3.1.2 rpl-dag-root.h

In this file we declared the 2 function for activate/disable alarm. (line 41)

```
#ifdef KEEP_ALIVE_GUARD
void start_kag(); //start keep_alive_guard  mode
void stop_kag();
#endif /* KEEP_ALIVE_GUARD */
```

### 3.1.3 rpl-dag.c

Logic for Directed Acyclic Graphs in RPL.
Initialize structures (line 87) and define the two function start_kag() and stop_kag(). (line 1423)

```
#ifdef KEEP_ALIVE_GUARD //init the structure of kag
struct kag_global_data kag_data= {NULL,0,0,0,0};
struct kag_nwlr_element nwlr_table[↩
    RPL_KAG_MAX_NEIGHBOUR_WITH_LOWER_RANCK];
struct ctimer timer_random;
extern void call_new_dio_interval(rpl_instance_t *instance);
#endif /* KEEP_ALIVE_GUARD */
```

```
#ifdef KEEP_ALIVE_GUARD
```

```
/*
 * Initialize the nwlr_table, save the current trickle ↩
     parameter an set
 * the trickle parameter for the KAG.
 */
void
start_kag()
{
  rpl_instance_t *instance;
  instance = rpl_get_default_instance();
  kag_data.alarm_activated=1;
  nwlr_table_init();

  ctimer_stop(&instance->dio_timer);

  kag_data.redundancy=instance->dio_redundancy;
  kag_data.intdouble=instance->dio_intdoubl;
  kag_data.intmin=instance->dio_intmin;

  instance->dio_intdoubl=0;
  instance->dio_redundancy=RPL_KAG_REDUNDANCY;
  instance->dio_intmin=RPL_KAG_INTMIN;
  instance->dio_counter=0;
  instance->dio_intcurrent=instance->dio_intmin + instance->↩
      dio_intdoubl;

  call_new_dio_interval(instance);
}

void avoids_collisions(void* instance)
{
  dio_output((rpl_instance_t *) instance, NULL); /* ↩
      multicast DIO */
}

/*
 * Disable the KAG and restore the old value of the trickle ↩
     parameters.
 * doi_intcurrent is set equal to dio_intmin so the trickle ↩
     is resetted.
 */
void stop_kag()
{
  rpl_instance_t *instance;
  instance = rpl_get_default_instance();
  kag_data.alarm_activated=0;
  nwlr_table_stop_timers();

  ctimer_stop(&instance->dio_timer);
  instance->dio_redundancy = kag_data.redundancy;
  instance->dio_intdoubl = kag_data.intdouble;
  instance->dio_intmin =kag_data.intmin;
  instance->dio_counter=0;
  instance->dio_intcurrent=instance->dio_intmin;

  ctimer_set(&timer_random, ((uint32_t)rand())%((1UL << ↩
      RPL_KAG_INTMIN)*CLOCK_SECOND/1000/4), &↩
      avoids_collisions, instance);
  call_new_dio_interval(instance);
}
#endif /* KEEP_ALIVE_GUARD */
```

### 3.1.4 rpl-timers.c

RPL timer management.
In this file we save in the structure kag_data only the ticks of the new interval (line 97) and define the function called call_new_dio_interval() to set the DIO timer (line 432).

```
#ifdef KEEP_ALIVE_GUARD
  kag_data.next_dio_msg=ticks;
#endif /* KEEP_ALIVE_GUARD */
```

```
#ifdef KEEP_ALIVE_GUARD
void
call_new_dio_interval(rpl_instance_t *instance)
{
    new_dio_interval(instance);
}
#endif /* KEEP_ALIVE_GUARD */
```

### 3.1.5 rpl.h

Public API declarations for ContikiRPL.
We added the struct for the table that hold the information about neighbours with rank greater of us, and other global variables.
Line 236:

```
#ifdef KEEP_ALIVE_GUARD
/* element of table */
struct kag_nwlr_element {
    uip_ipaddr_t ip;
    struct ctimer keep_alive_timer;
};
```

```
/*global variable */
struct kag_global_data
{
    void (*found_dead_node)(void*);
    uint8_t alarm_activated;
    uint8_t redundancy, intdouble, intmin;
    clock_time_t next_dio_msg;
};

extern struct kag_nwlr_element nwlr_table[];
extern struct kag_global_data kag_data;
extern void start_kag();
extern void stop_kag();
#endif /* KEEP_ALIVE_GUARD */
```

Line 317:

```
#ifdef KEEP_ALIVE_GUARD
void nwlr_table_init();
int nwlr_table_add (uip_ipaddr_t* new_ip, clock_time_t t);
struct kag_nwlr_element* nwlr_table_find(uip_ipaddr_t* ip);
void last_chance (void* expired_element);
void set_handler_function(void (*call_back)(void *));
#endif /* KEEP_ALIVE_GUARD */
```

### 3.1.6 rpl.c

In this file we have defined the function to manage the integrity mechanism.
Line 318:

```
#ifdef KEEP_ALIVE_GUARD
// inizialeze the nwlr_table table
void
nwlr_table_init()
{
  memset(nwlr_table, 0, sizeof(struct kag_nwlr_element) * ↩
      RPL_KAG_MAX_NEIGHBOUR_WITH_LOWER_RANCK);
  kag_data.next_free_index=0;
}

/*
 * Add a new element to the nwlr_table if possible and ↩
      starts the timer of this
 * new element, function last_chance() will be called when ↩
      the timer expire.
 *
 * @paramenter new_ip, the ip of the new element in the ↩
      table.
 *          time, time after the timer will expire, in CPU ↩
     ticks.
 *
 * @return 1 if the element was added, 0 otherwise.
 */
int
nwlr_table_add(uip_ipaddr_t* new_ip, clock_time_t time)
{
  if (kag_data.next_free_index < ↩
      RPL_KAG_MAX_NEIGHBOUR_WITH_LOWER_RANCK) {
    memcpy(&nwlr_table[kag_data.next_free_index].ip, new_ip,↩
         sizeof (uip_ipaddr_t));
    ctimer_set(&nwlr_table[kag_data.next_free_index].↩
        keep_alive_timer, time, last_chance, &nwlr_table[↩
        kag_data.next_free_index]);
    ++kag_data.next_free_index;
    return 1; // element added
  }
  return 0; // table is full !
}

/*
 * Search an element in the nwlr_table using the ip passed ↩
     as parameter.
 * If the element exist retrun a pointer to the element, ↩
     otherwise NULL
 * is returned.
 *
 * @paramenter ip, the ip we are looking for.
 *
 * @return a pointer to the searched element, or NULL if the↩
      element
 *        is not in the table.
 */
struct kag_nwlr_element *
nwlr_table_find(uip_ipaddr_t* ip)
{
  unsigned int i;
  for (i=0; i<kag_data.next_free_index; ++i)
    if (uip_ipaddr_cmp(ip, &nwlr_table[i].ip))
      return &nwlr_table[i];
  return NULL;
}

/*
 * This function is called when the timer of an element in ↩
     the nwlr_table
 * expires. This function send a DIS message to the node ↩
     whose timer is expired
 * and reset the timer to call the function set by the user.
 *
 * @paramenter expired_elem, the element in the table whose ↩
     timer is expired.
 */
void
last_chance(void* expired_elem)
{
  struct kag_nwlr_element* expired_element= (struct ↩
      kag_nwlr_element*) expired_elem;
  dis_output(&expired_element->ip);
  ctimer_set(&expired_element->keep_alive_timer, kag_data.↩
      next_dio_msg, kag_data.found_dead_node, &↩
      expired_element->ip);
}

/*
 * This function hould be called before start_kag().
 * Set the function to be called if a node becomes ↩
     unreachable.
 *
 * @paramenter call_back, the function to be called if a ↩
     node becomes
 *          unreachable.
 */
void
set_handler_function(void (*call_back)(void *))
{
  kag_data.found_dead_node=call_back;
}

/*
 * Stop all timer in the nwlr_table when the KAG is disable.
 */
void
nwlr_table_stop_timers()
{
  int i;
  for (i=0; i<kag_data.next_free_index;i++) {
  ctimer_stop(&nwlr_table[i].keep_alive_timer);
  }
}
#endif /* KEEP_ALIVE_GUARD */
```

### 3.1.7 rpl-imcmp6.c

ICMP6 I/O for RPL control messages.
Here we added a new byte at the end of the option used by Contiki to broadcast the trickle parameters. This new byte inform if the alarm is active or not, so when the guardian node active the alarm it sets this byte and the KAG trickle

paramenter and when the DIO is send this info is broadcasted to every node in the network, in this way we can enable/disable the integrity starting from one single node.
line 66:

```
#ifdef KEEP_ALIVE_GUARD
extern void call_new_dio_interval(rpl_instance_t *instance);
#endif /* KEEP_ALIVE_GUARD */
```

We modified the function dis_input() because before we received a dis input sent a DIO message in unicast, but we want to send it in multicast, so that the other nodes receive the message and reset the timer. In this way the node reduces the power consuption. (line 179)

```
#ifdef KEEP_ALIVE_GUARD
/*receve a DIS so it must send a DIO message in multicast ↩
    and reseet dio timer*/
          ctimer_stop(&instance->dio_timer);
          call_new_dio_interval(instance);
          /*send always multicast DIO*/
          dio_output(instance,NULL);
#else
          dio_output(instance, &UIP_IP_BUF->srcipaddr);
#endif /* KEEP_ALIVE_GUARD */
```

Variable decleration in dio_input() at line 234:

```
#ifdef KEEP_ALIVE_GUARD
  rpl_dag_t *dag;
  rpl_instance_t *instance;
  uint8_t new_alarm_status=0; // alarm flag read from the ↩
      received DIO
  clock_time_t remaining_clock; /* cloks remaining to the ↩
      end of the current
                   DIO interval */
  uip_ipaddr_t guardian_node_adress;
  struct kag_nwlr_element *neighbor_dio;
#endif /* KEEP_ALIVE_GUARD */
```

The node must read an extra data in the buffer of DIO message, because we send the alarm flag.

```
#ifdef KEEP_ALIVE_GUARD
/* with kag we receve a extra information */
      if(len != 17) {
#else
      if(len != 16) {
```

```
#endif /* KEEP_ALIVE_GUARD */
```

In line 414 we read the alarm flag in the DIO message.

```
#ifdef KEEP_ALIVE_GUARD
      new_alarm_status=buffer[i + 16];
#endif /* KEEP_ALIVE_GUARD */
```

Line 447 of dio_input(). Here we update the timer related to the sender of the DIO received and turn on/off KAG according to the alarm flag readed at line 414.

```
#ifdef KEEP_ALIVE_GUARD
instance=rpl_get_instance(dio.instance_id);
  dag=instance->current_dag;

  uip_ip6addr(&guardian_node_adress, 0xfe80, 0, 0, 0, 0xc30c↩
      , 0, 0, 2);
  if (new_alarm_status) { // if alarm is active
    // if in this node KAG is disable we must activate it
    if (!kag_data.alarm_activated && (DAG_RANK(dio.rank,↩
        instance) <= DAG_RANK(dag->rank,instance) || !memcmp↩
        (&from,&guardian_node_adress,sizeof(uip_ipaddr_t))))↩
         {
      nwlr_table_init();
      kag_data.alarm_activated=1;
      instance->dio_intdoubl = dio.dag_intdoubl;
      instance->dio_intmin = dio.dag_intmin;
      instance->dio_redundancy = dio.dag_redund;
      instance->dio_intcurrent = instance->dio_intmin + ↩
          instance->dio_intdoubl;
      ctimer_stop(&instance->dio_timer);
      PRINT6ADDR(&from);
      call_new_dio_interval(instance);
    }

    // if the DIO comes from a node that we must monitor, ↩
        reset its timer
    if (DAG_RANK(dio.rank,instance) > DAG_RANK(dag->rank,↩
        instance)) {
      neighbor_dio=nwlr_table_find(&from);
      if (!neighbor_dio) {
        if (nwlr_table_add(&from, kag_data.next_dio_msg)==0)
          PRINTF("ERROR! Keep alive guard table full!\n");
      }
      else {
        remaining_clock=timer_remaining(&neighbor_dio->↩
            keep_alive_timer.etimer.timer);
        ctimer_stop(&neighbor_dio->keep_alive_timer);
        ctimer_set(&neighbor_dio->keep_alive_timer,kag_data.↩
            next_dio_msg + remaining_clock,last_chance, ↩
            neighbor_dio);
      }
    }
  }
  else // if the DIO message have the alarm flag turned off,↩
       stop the KAG
    if (kag_data.alarm_activated) {
      stop_kag();
    }
#endif /* KEEP_ALIVE_GUARD */
```

In dio_output() function the node must extend the DIO message to send the alarm flag. (line 586)

```
#ifdef KEEP_ALIVE_GUARD
    buffer[pos++] = 15; /* the length of the option with the↩
        KAG is
                one byte bigger */
#else
  buffer[pos++] = 14;
#endif /* KEEP_ALIVE_GUARD */
```

Line 607:

```
#ifdef KEEP_ALIVE_GUARD
  buffer[pos++]=kag_data.alarm_activated; // send our KAG ↩
      status
#endif /* KEEP_ALIVE_GUARD */
```

## 4 Conclusion

We have developed an alarm system, but the changes for integrity in RPL can be used for many scenarios. Just change the function handler to be call when the network detects an unreachable node. Also change the fix interval for DIO message according to the needs.