* 1. Założenia

Badania zostały przeprowadzone w oparciu o system Contiki OS wraz z symulatorem Cooja. Analiza ruchu sieciowego została wykonana przy użyciu programu Wireshark. Proces badawczy można przedstawić w poniższych podpunktach:

1. Przygotowanie modeli węzłów w oparciu o biblioteki Contiki OS.
2. Kompilacja modeli niezainfekowanych w oparciu o domyślne biblioteki CONTIKI\_RPL.
3. Modyfikacja bibliotek CONTIKI\_RPL ( modyfikacje opisane w rozdziale 3.)
4. Kompilacja modeli zainfekowanych.
5. Przeprowadzenie testów opisanych w rozdziale 3.
6. Zbieranie danych (struktura drzewa DODAG, zapotrzebowanie energetyczne poszczególnych węzłów, ruch sieciowy)
7. Proces przetwarzania danych – tworzenie grafu DODAG, tworzenie wykresów oraz statystyk.
8. Analiza rezultatów oraz wnioski

Czas trwania symulacji jest taki sam dla wszystkich testów i wynosi 10 minut. Został on określony na podstawie kilku przeprowadzonych prób i pozwala w odpowiedni sposób przedstawić wszystkie mechanizmy zachodzące w sieci – osiągnięcie zbieżności, mechanizm globalnej naprawy sieci, zmiany w grafie DODAG.

* 1. Modele węzłów

Na listingu 4.1 oraz listingu 4.2. - przedstawiony jest kod źródłowy modelu korzenia oraz węzłów – zainfekowanego oraz nie zainfekowanego. Kod napisany jest w języku C. Z uwagi na to, że zaimplementowane w ramach systemu Contiki modele rzeczywistych sensorów, które są emulowane w Cooja mają mocno ograniczone zasoby: głównie pamięć ROM oraz RAM – należy zwracać szczególną uwagę na optymalizację. Kod powstał na bazie gotowych modeli zaimplementowanych w ramach ContikiOS.

Kod węzła będącego rootem przedstawiony jest w listingu 4.1. W linii 21 zdefiniowana jest stała UIP\_IP\_BUF. Odnosi się ona do struktury będącej buforem na dane uip\_buf. Niżej zdefiniowane są porty dla serwera oraz klienta (linia 22-23). W linii 28 tworzony jest proces udp\_server\_process przy pomocy dyrektywy Contiki PROCESS (). Poniżej w linii 29 proces zostaje uruchomiony razem z procesem o nazwie collect\_common\_process będącym częścią aplikacji collect-view służącej do zbierania informacji z węzłów. W linii 34 wywołana zostaje funkcja powertrace\_start (CLOCK\_SECOND\*10), która uruchamia aplikację powertrace do zbierania informacji o obciążeniu energetycznym węzłów. W linii 40-43 węzeł określa, że będzie rootem (wartość root\_if przyjmuje wartość różną od 0) oraz ustala swój adres IPv6. W linii 45-53 węzeł jako root tworzy nowy DAG (lub jeśli nie jest rootem wypisuje informację o błędzie (linia 52)). Następnie w linii 59 tworzone jest połączenie między serwerem i klientem, a w linii 60 połączenie jest łączone z konkretnym numerem portu UDP\_SERVER\_PORT. W pętli while (linia 67-75) węzeł czeka na wiadomości i w razie zdarzenia (linia 69), czyli nadejścia pakietu, zostaje wywołana funkcja tcpip\_handler(), która wypisze otrzymane dane na ekranie terminala. W linii 73 znajduje się odwołanie do funkcji rpl\_repair\_root(), która inicjalizuje proces globalnej naprawy sieci.

Kod węzła sender/malicious-sender jest zbliżony jednak ograniczony w stosunku do roota. Różnica między sender, a malicious-sender polega na tym, że w przypadku węzła zainfekowanego był on kompilowany w oparciu o biblioteki (do których pliki nagłówkowe są zdefiniowane w linii 1-5) zmodyfikowane w sposób opisany w rozdziale 3. W przypadku zwyczajnego węzła typu sender kompilacja odbyła się w oparciu o standardowe biblioteki Contiki. Parametry wykorzystanych węzłów sensorowych podano w tabeli 4.1.

Tabela 4.1.Parametry Tmote Sky oraz Zolteria Z1

|  |  |  |
| --- | --- | --- |
| Nazwa urządzenia | Rozmiar pamięci RAM/ROM | Częstotliwość taktowania CPU |
| Tmote Sky | 10 kB/48 kB | 3.9 MHz |
| Zolteria Z1 | 8 kB/64 kB | 8 MHz |

Zapotrzebowanie energetyczne zostało liczone przy pomocy aplikacji powertrace. Otrzymane dane zostały następnie przetworzone w sposób opisany wzorem (1) [2,22]:

(1)

t - czas trwania cyklu pracy, wartość ta jest obliczana jako różnica między wartością z poprzedniego i obecnego cyklu pracy.

U – wartość napięcia zasilania dla modelu Tmote Sky wynosi 3 V

I – natężenie dla modelu Tmote Sky, wartość ta jest różna dla każdego z 4 trybów pracy:

Aktywny – 2 mA

LPM (ang. Low Power Mode ), uśpiony – 55 μA

Aktywny, TX – 18 mA

Aktywny, RX – 20 mA

T – dla badanych modeli okres przesyłania informacji z węzła wynosi 10 s

– stała określająca liczbę akcji w trakcie 1 sekundy, wynosi ona 32768

P – zapotrzebowanie energetyczne [mW]

Powyższe wartości zostały określone w dokumentacji Contiki [2] na podstawie dokumentacji oraz danych technicznych Tmote Sky [22].

Dane dotyczące drzewa DODAG zostały pozyskane bezpośrednio z Cooja. Następnie na podstawie informacji zostały utworzony graf.

W przypadku Z1, który ma nieco większe możliwości, nie dało się zaimplementować w kodzie niektórych funkcji: collect view (aplikacja odpowiedzialna za zbieranie oraz przesyłanie informacji o węzłach), power tracer (aplikacja służąca do zbierania informacji na temat zapotrzebowania energetycznego węzłów). Dlatego w pracy zostały użyte modele Tmote Sky.

1 #include "contiki.h"

2 #include "contiki-lib.h"

3 #include "contiki-net.h"

4 #include "net/ip/uip.h"

5 #include "net/rpl/rpl.h"

6 #include "net/linkaddr.h"

7 #include "net/netstack.h"

8 #include "dev/button-sensor.h"

9 #include "dev/serial-line.h"

10 #include "powertrace.h"

11 #include "dev/uart1.h"

12 #include <stdio.h>

13 #include <stdlib.h>

14 #include <string.h>

15 #include <ctype.h>

16 #include "collect-common.h"

17 #include "collect-view.h"

18 #define DEBUG DEBUG\_PRINT

19 #include "net/ip/uip-debug.h"

20

21 #define UIP\_IP\_BUF ((struct uip\_ip\_hdr \*)&uip\_buf[UIP\_LLH\_LEN])

22 #define UDP\_CLIENT\_PORT 8775

23 #define UDP\_SERVER\_PORT 5688

24

25 static struct uip\_udp\_conn \*server\_conn;

26 static uip\_ipaddr\_t ipaddr;

27

28 PROCESS(udp\_server\_process, "UDP server process");

29 AUTOSTART\_PROCESSES(&udp\_server\_process,&collect\_common\_process);

30 PROCESS\_THREAD(udp\_server\_process, ev, data)

31 {

32 uip\_ipaddr\_t ipaddr;

33 struct uip\_ds6\_addr \*root\_if;

34 PROCESS\_BEGIN();

35 powertrace\_start(CLOCK\_SECOND\*10);

36 PROCESS\_PAUSE();

37 SENSORS\_ACTIVATE(button\_sensor);

38 PRINTF("UDP server started\n");

39

40 #if UIP\_CONF\_ROUTER

41 uip\_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);

42 uip\_ds6\_addr\_add(&ipaddr, 0, ADDR\_MANUAL);

43 root\_if = uip\_ds6\_addr\_lookup(&ipaddr);

44

45 if(root\_if != NULL) {

46 rpl\_dag\_t \*dag;

47 dag= rpl\_set\_root(RPL\_DEFAULT\_INSTANCE,(uip\_ip6addr\_t \*)&ipaddr);

48 uip\_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);

49 rpl\_set\_prefix(dag, &ipaddr, 64);

50 PRINTF("created a new RPL dag\n");

51 } else {

52 PRINTF("failed to create a new RPL DAG\n");

53 }

54 #endif

55

56 print\_local\_addresses();

57 NETSTACK\_RDC.off(1);

58

59 server\_conn = udp\_new(NULL, UIP\_HTONS(UDP\_CLIENT\_PORT), NULL);

60 udp\_bind(server\_conn, UIP\_HTONS(UDP\_SERVER\_PORT));

61

62 PRINTF("Created a server connection with remote address ");

63 PRINT6ADDR(&server\_conn->ripaddr);

64 PRINTF(" local/remote port %u/%u\n",UIP\_HTONS(server\_conn->lport),

65 UIP\_HTONS(server\_conn->rport));

66

67 while(1) {

68 PROCESS\_YIELD();

69 if(ev == tcpip\_event) {

70 tcpip\_handler();

71 } else if (ev == sensors\_event && data == &button\_sensor) {

72 PRINTF("Initiaing global repair\n");

73 rpl\_repair\_root(RPL\_DEFAULT\_INSTANCE);

74 }

75 }

76 PROCESS\_END();

77 }

Listing 4.1. Węzeł typu root

1 #include "contiki.h"

2 #include "net/ip/uip.h"

3 #include "net/ipv6/uip-ds6.h"

4 #include "net/ip/uip-udp-packet.h"

5 #include "net/rpl/rpl.h"

6 #include "dev/serial-line.h"

7 #include "dev/uart1.h"

8 #include "collect-common.h"

9 #include "collect-view.h"

10 #include <stdio.h>

11 #include <string.h>

12

13 #define UDP\_CLIENT\_PORT 8775

14 #define UDP\_SERVER\_PORT 5688

15

16 #define DEBUG DEBUG\_PRINT

17 #include "net/ip/uip-debug.h"

18 static struct uip\_udp\_conn \*client\_conn;

19 static uip\_ipaddr\_t server\_ipaddr;

20

21 PROCESS(udp\_client\_process, "UDP client process");

22 AUTOSTART\_PROCESSES(&udp\_client\_process,&collect\_common\_process);

23 PROCESS\_THREAD(udp\_client\_process, ev, data)

24 {

25 PROCESS\_BEGIN();

26 powertrace\_start(CLOCK\_SECOND\*10);

27 PROCESS\_PAUSE();

28 set\_global\_address();

29 PRINTF("UDP client process started\n");

30 print\_local\_addresses();

31

32 client\_conn = udp\_new(NULL, UIP\_HTONS(UDP\_SERVER\_PORT), NULL);

33 udp\_bind(client\_conn, UIP\_HTONS(UDP\_CLIENT\_PORT));

34

35 PRINTF("Created a connection with the server ");

36 PRINT6ADDR(&client\_conn->ripaddr);

37 PRINTF(" local/remote port %u/%u\n",

38 UIP\_HTONS(client\_conn->lport), UIP\_HTONS(client\_conn->rport));

39 while(1) {

40 PROCESS\_YIELD();

41 if(ev == tcpip\_event) {

42 tcpip\_handler();

43 }

44 }

45 PROCESS\_END();

Listing 4.2. Węzeł typu sender/malicious-sender