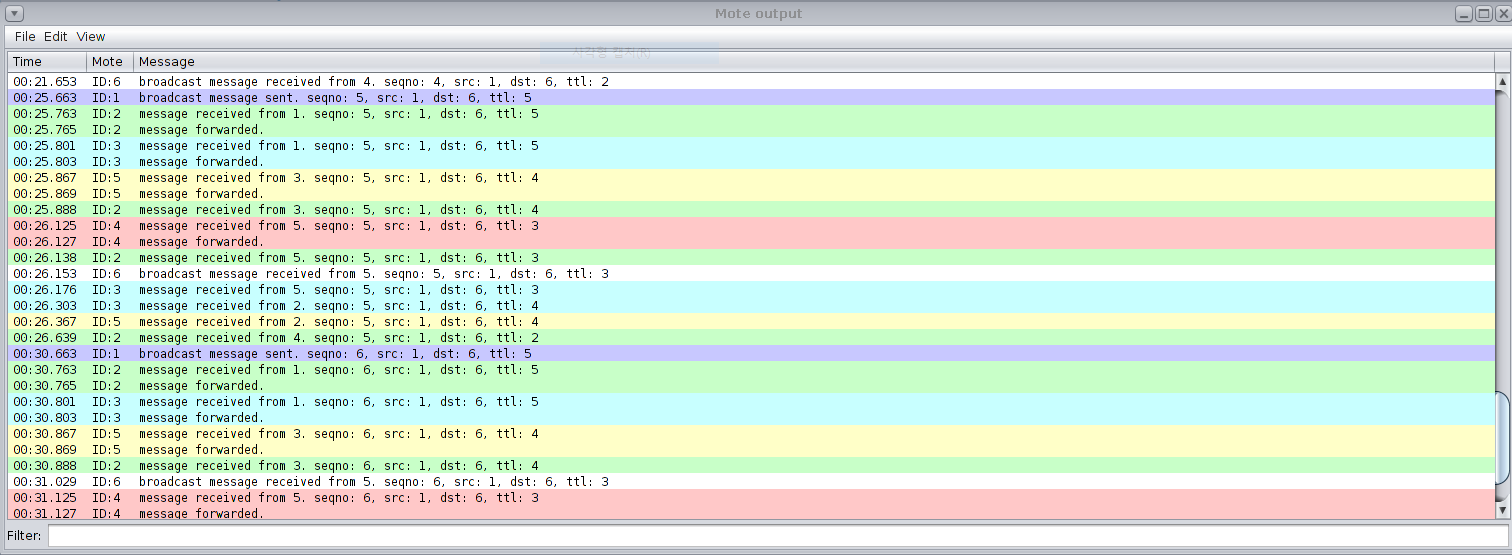
**네트워크 프로토콜 설계 및 실습 과제 4**

전기정보공학부 2016-10769

**1. Random Walk protocol 구현**

다음은 Random Walk protocol의 구현을 simulation한 결과이다. node의 배치는 4주차 ppt자료에서의 topology 1과 같다. 1이 sender node, 6이 receiver node로 구성하여 배치했다.



**그림1.** RWP구현 시뮬레이션 결과. sender node가 1이고, receiver node가 6이다.

위 그림에서 확인할 수 있듯, sender node에서 packet을 보냄에 따라 receiver node(sink node)에서 packet을 수신하는 것을 알 수 있다. 또한 중복된 packet의 경우는 무시한다. 이때 6번 node에서, 직전의 node가 4번 또는 5번으로 나타나는데 이는 4, 5번 node에서의 순서에 따라 달라진다. 먼저 수신한 packet을 받고, 나중에 수신한 packet은 무시하기 때문이다. 따라서 4, 5번 node의 duty cycle 주기나 처리 속도 등에 의해 순서가 바뀔 수 있다. node간에서 이미 forwarding한 packet을 송신하는 것을 볼 수 있는데, 이경우 이미 처리된 packet이므로 무시한다. ttl이 3인 상태로 도착하는 것으로 보아, sender와 receiver 사이에 2 개의 node가 있는 것을 알 수 있다.

다음은 구현 source code이다.

#include "contiki.h"

#include "net/rime.h"

#include "random.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include "dev/light-sensor.h"

#include "dev/cc2420.h"

#include "sys/node-id.h"

#include <stdio.h>

#include <string.h>

*struct* message {

*uint16\_t* src;

*uint16\_t* dst;

*uint16\_t* seqno;

*uint16\_t* ttl;

};

/\*---------------------------------------------------------------------------\*/

PROCESS(example\_broadcast\_process, "Broadcast example");

AUTOSTART\_PROCESSES(&example\_broadcast\_process);

/\*---------------------------------------------------------------------------\*/

static *void*

broadcast\_recv(*struct* broadcast\_conn \*c, const *rimeaddr\_t* \*from)

{

}

static const *struct* broadcast\_callbacks broadcast\_call = {broadcast\_recv};

static *struct* broadcast\_conn broadcast;

/\*---------------------------------------------------------------------------\*/

PROCESS\_THREAD(example\_broadcast\_process, ev, data)

{

static *struct* etimer et;

static *uint8\_t* seqno = 1;

*struct* message msg;

PROCESS\_EXITHANDLER(broadcast\_close(&broadcast);)

PROCESS\_BEGIN();

broadcast\_open(&broadcast, 129, &broadcast\_call);

while(1) {

/\* Delay 5 seconds \*/

etimer\_set(&et, CLOCK\_SECOND \* 5);

PROCESS\_WAIT\_EVENT\_UNTIL(etimer\_expired(&et));

msg.seqno = seqno;

    // set source as me

    msg.src = node\_id;

    // destination is sink node : 6

    msg.dst = 6;

    msg.ttl = 5;

//packetbuf\_copyfrom("Hello", 6);

packetbuf\_copyfrom(&msg, sizeof(*struct* message));

broadcast\_send(&broadcast);

printf("broadcast message sent. seqno: %d, src: %d, dst: %d, ttl: %d\n", msg.seqno, msg.src, msg.dst, msg.ttl);

seqno++;

}

PROCESS\_END();

}

**코드1.** sender.c의 구현. 1번 node로 사용되어 최초로 packet을 send한다.

sender의 경우, PROCESS\_THREAD에서, seqno와 msg를 정의하고 process가 종료될 때의 동작을 broadcast channel close로 정의한다. callback함수로써 broadcast\_recv를 등록하여 packet을 받았을 때의 동작을 정의한다. sender의 경우 아무것도 하지 않는다. process가 시작되면 broadcast channel을 열고, 5초에 한번씩 msg를 packet에 실어 send한다. 이때 msg의 각 값을 초기화하는데, src를 자신의 node number인 node\_id로, dst를 sink node인 6으로, ttl을 5로 설정한다. broadcast message를 보낸 뒤에는 seqno를 증가시킨다. seqno는 1부터 시작하여 1씩 증가하도록 설정하였다.

#include "contiki.h"

#include "net/rime.h"

#include "random.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include "dev/light-sensor.h"

#include "dev/cc2420.h"

#include <stdio.h>

#include <string.h>

*struct* message {

*uint16\_t* src;

*uint16\_t* dst;

*uint16\_t* seqno;

*uint16\_t* ttl;

};

static *uint16\_t* last\_seqno = 0;

/\*---------------------------------------------------------------------------\*/

PROCESS(example\_broadcast\_process, "Broadcast example");

AUTOSTART\_PROCESSES(&example\_broadcast\_process);

/\*---------------------------------------------------------------------------\*/

static *void*

broadcast\_recv(*struct* broadcast\_conn \*c, const *rimeaddr\_t* \*from)

{

*struct* message msg;

memcpy(&msg, packetbuf\_dataptr(), sizeof(*struct* message));

if (msg.seqno <= last\_seqno)

return;

last\_seqno = msg.seqno;

printf("broadcast message received from %d. seqno: %d, src: %d, dst: %d, ttl: %d\n", from->u8[0], msg.seqno, msg.src, msg.dst, msg.ttl);

}

static const *struct* broadcast\_callbacks broadcast\_call = {broadcast\_recv};

static *struct* broadcast\_conn broadcast;

/\*---------------------------------------------------------------------------\*/

PROCESS\_THREAD(example\_broadcast\_process, ev, data)

{

PROCESS\_EXITHANDLER(broadcast\_close(&broadcast);)

PROCESS\_BEGIN();

broadcast\_open(&broadcast, 129, &broadcast\_call);

while(1) {

  PROCESS\_WAIT\_EVENT();

}

PROCESS\_END();

}

**코드2.** receiver.c의 구현. 6번 node로 사용되어 sink node 역할을 한다.

broadcast-example.c를 기반으로 하여, sender.c와 비슷하게 동작한다. PROCESS\_THREAD에서, broadcast channel을 open한뒤 대기상태에 들어간다. 이후 아무런 동작도 하지 않는다. callback함수로 등록된 broadcast\_recv함수에서, packet을 수신하면 msg에 내용을 복사한다. last\_seqno라는 global variable을 가지고 있다. last\_seqno는 packet의 sequence number를 저장하여, 자신이 이미 처리한 packet의 경우 무시한다. 따라서 msg에서의 seqno가 last\_seqno보다 작으면 아무것도 하지 않고 함수를 종료한다. last\_seqno보다 크면 새로운 packet이므로 last\_seqno에 이를 저장하고 message를 받았음을 print한다.

#include "contiki.h"

#include "net/rime.h"

#include "random.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include "dev/light-sensor.h"

#include "dev/cc2420.h"

#include "sys/node-id.h"

#include <stdio.h>

#include <string.h>

*struct* message {

*uint16\_t* src;

*uint16\_t* dst;

*uint16\_t* seqno;

*uint16\_t* ttl;

};

static *uint16\_t* last\_seqno = 0;

/\*---------------------------------------------------------------------------\*/

PROCESS(example\_broadcast\_process, "Broadcast example");

AUTOSTART\_PROCESSES(&example\_broadcast\_process);

/\*---------------------------------------------------------------------------\*/

static *void*

broadcast\_recv(*struct* broadcast\_conn \*c, const *rimeaddr\_t* \*from)

{

*struct* message msg;

memcpy(&msg, packetbuf\_dataptr(), sizeof(*struct* message));

printf("message received from %d. seqno: %d, src: %d, dst: %d, ttl: %d\n", from->u8[0], msg.seqno, msg.src, msg.dst, msg.ttl);

// filtering for restriction.

  if (msg.src == node\_id )

return;

if (msg.seqno <= last\_seqno)

return;

if (msg.ttl <= 0)

return;

// if all true, then forward packet.

last\_seqno = msg.seqno;

msg.ttl--;

packetbuf\_copyfrom(&msg, sizeof(*struct* message));

process\_post(&example\_broadcast\_process, PROCESS\_EVENT\_CONTINUE, NULL);

}

static const *struct* broadcast\_callbacks broadcast\_call = {broadcast\_recv};

static *struct* broadcast\_conn broadcast;

/\*---------------------------------------------------------------------------\*/

PROCESS\_THREAD(example\_broadcast\_process, ev, data)

{

PROCESS\_EXITHANDLER(broadcast\_close(&broadcast);)

PROCESS\_BEGIN();

broadcast\_open(&broadcast, 129, &broadcast\_call);

while(1) {

    // wait until callback function's wakeup call

    PROCESS\_WAIT\_EVENT();

// send msg

broadcast\_send(&broadcast);

printf("message forwarded.\n");

}

PROCESS\_END();

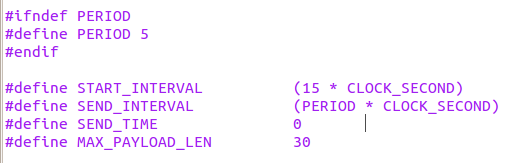
}

**코드3.** forwarder.c 구현. 2, 3, 4, 5번 node로 구성되어 있다.

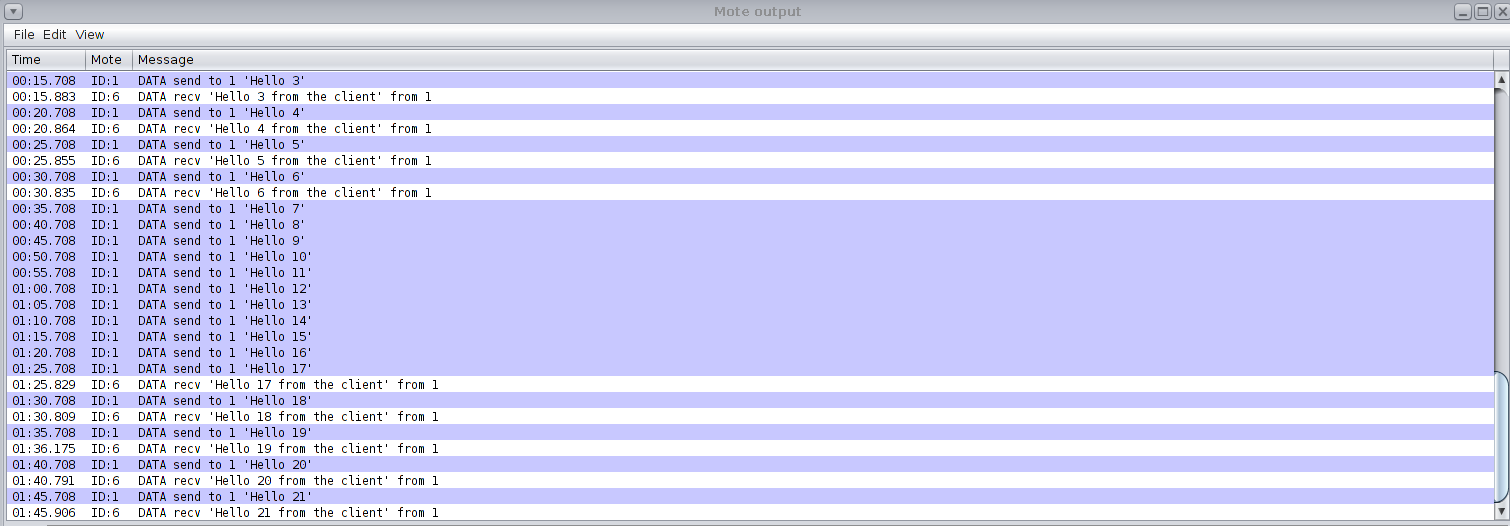
forwarder의 경우, process가 시작된 후 broadcast channel을 open한다. 이후 EVENT를 기다리도록 PROCESS\_WAIT\_EVENT()를 사용하였다. 이는 callback함수가 EVENT를 보냈을 때 message를 송신하기 위함이다. callback 함수인 broadcast\_recv에서, packet을 수신하면 이를 msg에 저장한다. 이후 message에 대한 정보를 print한다. src가 자신으로 설정되어 있으면 packet을 무시하기 위해 return하고, seqno가 last\_seqno보다 작으면 이미 처리한 packet으로 간주하여 무시한다. ttl이 0이거나 0보다 작으면 만료된 packet이므로 무시한다. 위의 if문에 모두 걸리지 않았다면 3개의 조건을 만족시킨 것이므로, last\_seqno를 msg.seqno로 설정하고 ttl을 1 감소시킨 후에 packet buffer에 msg를 copy한다. 이후 process에 PROCESS\_EVENT\_CONTINUE를 post하여 event를 기다리고 있는 process를 진행시킨다. process는 이후 packetbuf의 내용을 broadcasting한다.

**2. dynamic topology에서의 RPL과의 성능 비교**

다음은 RPL에서 topology를 바꿨을 때의 결과이다. 이 때, sender node에서 configure를 다음과 같이 하여 RWP와 같이 sender에서 5초에 한번씩 packet을 송신하도록 하였다.



**그림2.** sender.c의 configure



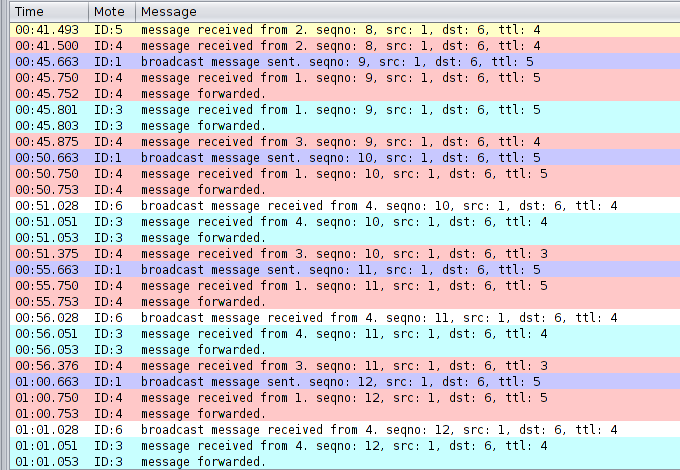
**그림3.** RPL의 dynamic topology 시뮬레이션 결과

그림 2에서 확인할 수 있듯, Hello 6을 수신한 뒤에 topology2로 바꾸어 Hello 7부터 packet을 받지 못하기 시작하였고, 이후 Hello 16까지 총 10개의 packet loss가 발생하였다. 따라서 sink node가 sender node의 packet을 성공적으로 수신하는 시간이 50초정도임을 알 수 있다.

위와 같이 RPL에서 topology가 바뀌었을 때 다시 packet을 송신하는데 시간이 소모되는 이유는 다음과 같다. RPL은 transmission을 위한 routing을 미리 설정하여 tree의 구조를 형성하는데, 이를 위해 packet을 송신하기 전에 DIS, DIO, DAO message를 이용한다. 이로 인해 topology가 바뀌었을 때, 새로운 tree 구조를 형성하기 위해 부가적인 packet 교환이 필요하게 된다. 이로 인한 overhead로 인해 sender에서의 packet loss가 발생한다.

자세히 서술하자면, tree의 root가 되는 6번(sink node)에서 DIS를 통해 다른 node들이 이를 detect하여 DIO를 보내 tree를 형성한다. 따라서 각 node는 자신에게 DIS를 보낸 node중 가장 가까운 node에만 붙어 routing을 확보하게 되는데, 2번과 5번 node가 사라짐에 따라 기존의 tree 구조가 깨지면서 routing이 불가능하게 된다. 따라서 이후에는 6번에서 다시 DIS를 보내고, 이를 통해 4번에서 DIO를 통해 tree에 붙고, 4번이 DIS를 보내서 5번이 tree에 붙고, 5번에서 1번이 tree에 붙는 과정을 반복해야 한다. 이러한 과정은 overhead가 되고, 이로 인해 소모되는 시간동안 sender node에서는 sink node까지의 routing을 알지 못하므로 message는 모두 loss된다.

다음은 Random Walk protocol에서의 결과이다.



**그림4.** RWP에서의 dynamic topology 시뮬레이션 결과

그림 4에서 확인할 수 있듯, Random Walk protocol(이하 RWP)에서는 packet loss가 일어나지 않았다. 즉, RWP에서는 topology가 변한 후에도 성공적으로 수신할 수 있었다.

RWP의 경우 RPL과는 다르게 routing을 위한 구조를 형성하지 않는다. broadcast를 통해 packet을 전달하고, 마지막의 sink node에서 자신이 dst임을 확인하면 이를 accept한다. 따라서 이론적으로 topology가 변하더라도, 구조를 다시 형성하는 overhead가 없으므로 forwarding node 사이의 거리가 멀어 전달되지 않는 구간이 없다면 바로 sender에서 보낸 packet을 sink node가 수신할 수 있다.

따라서 RWP의 경우 RPL보다 dynamic topology에 적합함을 알 수 있다. 또한 power의 관점에서도, RWP는 tree를 형성하는 overhead packet이 없기 때문에 더 효율적이라고 할 수 있다. unicast와 broadcast에 있어서 power는 차이가 거의 없으므로, RWP가 더 효율적이다. 하지만 RWP의 경우 모든 node가 packet을 broadcast하기 때문에, traffic이 과하게 많아질 수 있다. 즉, 좁은 영역에 node가 밀집되어 있으면 RPL의 경우 하나의 routing경로만이 있어 traffic이 크지 않지만, RWP의 경우 모든 node에서 한번씩 packet을 send하므로 traffic이 과도할 수 있다. 이경우에는 overhead를 감안해도 power의 관점에서 RPL이 더 효율적이라고 생각할 수 있다.