Journal of Institute of Control, Robotics and Systems (2012) 18(8):774-779
http://dx.doi.org/10.5302/J.ICROS.2012.18.8.774 ISSN:1976-5622 eISSN:2233-4335

USN을 위한 RTT 기반 TCP 설계 및 구현

RTT based TCP Design and Implementation for USN

이 현 철, 최 준 영*
(Hyun Chul Yi[1] and Joon Young Choi[1])
[1]Pusan National University

Abstract: We design and implement a RTT (Round Trip Time) based TCP (Transmission Control Protocol) for USN (Ubiquitous Sensor Network). We adopt a basic update algorithm for window size from FAST TCP that uses the queuing delay at link as the congestion measure. The designed TCP estimates the queuing delay at link from the measured RTT in the network layer, and updates the window size based on the estimated queuing delay. The designed TCP allows to utilize the full capacity of USN links and avoids the waste of the given link capacity that is common without the flow control in the transport layer. The experiment results show that the window size of the sender converges within a small range of variations without any packet loss, and verify the stability and performance of the designed TCP.

Keywords: USN, RTT, queuing delay, TCP, flow control, network congestion

I. 서론

유비쿼터스 센서 네트워크(ubiquitous sensor network)는 다양한 환경 조건을 모니터링 하기 위한 센서와 무선 네트워크 기능을 포함하는 디바이스로 이루어진 네트워크이다. 이러한 무선 네트워크를 사용하기 위한 표준 기술은 IEEE 802.15.4 표준[1]과 이를 기반으로 상위 계층을 표준화한 ZigBee 프로토콜[2]이 있다. 일반적인 USN의 경우 제한적인 성능과 부품들로 이루어져 OSI 7 모델[3]의 전 계층을 구현하지 못하고 있다. [1]의 경우 OSI 7모델의 데이터 링크 계층까지 표준화하고, [2]의 경우 [1]을 포함하여 네트워크 계층까지 표준화하고 있다. USN에서 사용하는 OS 중 하나인 TinyOS [4,5]는 [1]을 포함한 네트워크 프로토콜을 구현하여 네트워크 계층으로 이용하고 있다. 따라서 일반적인 USN에서는 전송 계층의 역할을 기대할 수 없다. 전송 계층은 네트워크 계층의 상위 계층으로써 두 노드 사이에서 전달되는 데이터의 신뢰성과 효율성을 보장한다. 전송 계층은 이러한 역할을 수행하기 위해 오류 검출 및 복구와 흐름 제어 등을 수행한다[3].

본 논문에서는 전송 용량을 최대로 이용하고 패킷 손실을 피하기 위해 USN에서 전송 계층의 TCP 기능을 설계하고 구현하고자 한다. 네트워크 계층까지 구현된 무선 센서 네트워크는 전체 네트워크 구조나 혼잡 상태에 따라 전송 효율이 달라진다. 네트워크 전송 용량의 낭비를 피하기 위하여 전송 계층의 TCP를 적용하면 윈도우 사이즈 값의 계산에 따라 각 센서 노드의 링크 전송 용량을 최대로 이용할 수 있다. TCP의 윈도우 사이즈 업데이트는 FAST TCP 알고리즘[6]을 이용한다. FAST TCP 알고리즘은 혼잡 측정 수단으로 RTT (Round Trip Time)을 이용하기 때문에 패킷 손실이 발생하지 않는다. USN의 TCP 구현[7]에서 FAST TCP 알고리즘을 이용하면 센서 노드 사이의 패킷 손실을 피할 수 있다. USN을 위한 TCP 설계 및 구현을 위해서 TinyOS를 이용한다. TinyOS의 경우 오픈 소스로 공개되어 있기 때문에 새로운 기능의 설계와 구현이 쉽다. 네트워크 계층[8,9]까지 구현된 TinyOS에서 네트워크 구조를 분석하여 TCP 기능을 가진 전송 계층을 설계하고 구현한다. 구현한 USN의 성능 평가를 위해 TCP 적용 여부에 따라 각각의 USN 동작을 실험하고, 전송 효율 및 패킷 손실 발생, 공평성 등을 분석한다.

* 책임저자(Corresponding Author)
논문접수: 2012. 5. 15., 수정: 2012. 6. 14., 채택확정: 2012. 6. 30.
이현철, 최준영: 부산대학교 전자공학과
(schezow@pusan.ac.kr/jyc@pusan.ac.kr)
※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2011-0026769).

II. TinyOS 분석 및 TCP 설계

1. TinyOS 네트워크 모델 분석

TCP를 설계하기 위하여 TinyOS의 네트워크 구조를 분석한다[10,11]. 각각의 기능이 구현된 컴포넌트의 연결을 분석하면 전체 네트워크 구조를 파악할 수 있다. 그 구조는 그림 1과 같다. 데이터 링크 계층에 대한 기능은 ActiveMessageC 컴포넌트, 네트워크 계층에 대한 기능은 CtpP 컴포넌트에 구현되어 있다. CtpP 컴포넌트는 CTP (Collection Tree Protocol)라는 트리 기반의 라우팅 프로토콜 기능이 구현된 컴포넌트이다[8,9].

2. TCP 설계

그림 2는 그림 1의 네트워크 구조를 분석하여 TCP를 설계하고 구현할 위치를 추가한 구조이다. 전송 계층은 FAST TCP를 통해 패킷 저장 큐를 관리한다[6]. FAST TCP는 윈도우 사이즈 업데이트를 위해 RTT 값을 필요로 한다. 계산에 필요한 RTT 값은 네트워크 계층에서 측정하여 전송 계층으로 전달된다. 전송 계층에서는 전달된 RTT 값을 이용하여 윈도우 사이즈를 업데이트하고 패킷 저장 큐에 윈도 사이즈만큼 패킷을 저장하도록 조절한다.

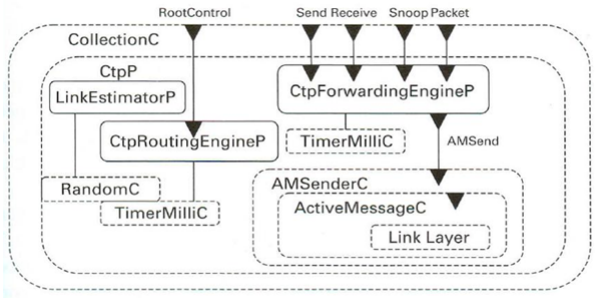

그림 1. TinyOS의 네트워크 구조.
Fig. 1. Network structure of TinyOS.

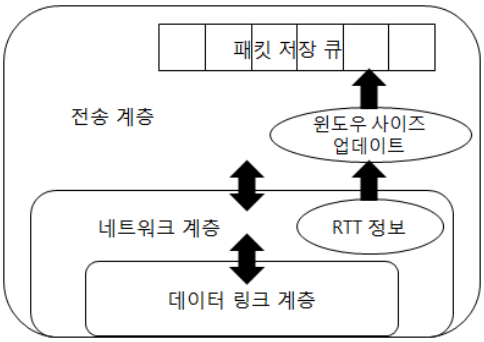

그림 2. TCP가 적용된 네트워크 계층 구조.
Fig. 2. TCP applied network layer structure.

RTT 측정을 위해서는 RTT 시작 시간과 RTT 종료 시간 측정이 필요하다. RTT 시작 시간은 패킷 전송 노드에서 패킷을 전송하기 직전에 측정하고, RTT 종료 시간은 패킷 수신 노드에서 전송한 ACK의 수신 직후에 측정한다. 이렇게 구한 RTT 시작 시간과 종료 시간의 차이를 통해 RTT를 측정할 수 있다.

FAST TCP에서 이용하는 윈도우 사이즈 업데이트 알고리즘은 다음 식 (1)과 같다[6,12].

$$WND_{new} = -(1-\gamma)\, WND_{old} + \gamma\left(\frac{RTT_{\min}}{RTT_{avg}} WND_{old} + \alpha\right) \quad (1)$$

식 (1)에서 WND_{new}는 현재 계산된 윈도우 사이즈, γ는 윈도우 사이즈 적용 비율, WND_{old}는 이전 윈도우 사이즈, $RTT_{\min}$은 현재까지 측정된 RTT 값 중 가장 작은 값, RTT_{avg}는 현재까지 측정된 평균 RTT 값, α는 윈도우 사이즈 평형 상수 값이다.

III. USN을 위한 TCP 구현

II 장의 TinyOS 분석 및 TCP 설계를 통해 실제 TinyOS에 TCP의 기능을 구현할 수 있다. 네트워크 계층에서는 RTT 측정을 위한 기능을 구현하고 전송 계층에서는 TCP 큐와 윈도우 사이즈 업데이트 기능을 구현한다.

1. RTT 측정을 위한 구현[13]

TinyOS에서 사용하는 변수의 경우 최대 표현 범위가 16비트이기 때문에 RTT 측정 과정에서 오버플로우 문제가 발생한다. RTT 종료 시간이 65535보다 클 경우 실제 저장

```
IF ( RTT_END < RTT_START )
    RTT_TOTAL = Variable's Limitation Value
                - RTT_START + RTT_END
ELSE
    RTT_TOTAL = RTT_START - RTT_END
```

그림 3. RTT 측정 의사 코드.
Fig. 3. RTT estimation pseudo code.

```
//RTT Setter
command void setRtt(uint16_t rtt);

//RTT Getter
command error_t getRtt(uint16_t* rtt);
```

그림 4. RTT 값 저장 및 불러오기 함수 원형.
Fig. 4. Store and load function prototypes for RTT value.

```
//Rtt Start Time
 RttStart = call Timer.getNow();
```

그림 5. RTTSTART 값 저장 코드.
Fig. 5. Store code for RTTSTART value.

```
if(ackPending && call
PacketAcknowledgements.wasAcked(msg))
{
    RttEnd = call RetxmitTimer.getNow();
      if (call RootControl.isRoot())
          call CtpInfo.setRtt(0);
    else if ( RttEnd < RttStart )
          call CtpInfo.setRtt(0xFFFF - RttStart + RttEnd);
    else
          call CtpInfo.setRtt(RttEnd - RttStart);
}
else
          call CtpInfo.setRtt(0xFFFF);
```

그림 6. RTT 값 저장을 위한 구현 코드.
Fig. 6. Implementation codes of RTT value store.

되는 RTT 종료 시간은 RTT 시작 시간보다 작아진다. 이 문제를 해결하기 위해서 다음 그림 3과 같은 의사 코드가 필요하다. 실제 구현에서 변수의 최댓값은 0xFFFF가 된다.

그림 4는 CtpInfo.nc 파일에 추가된 RTT를 저장하고 불러오는 코드이다. CtpInfo.nc 파일은 TinyOS에서 구현된 컴포넌트 간의 인터페이스 역할을 한다. 네트워크 계층과 전송 계층은 CtpInfo 인터페이스 통해 RTT 값을 저장하고 그 값을 불러올 수 있다. setRtt 함수를 통해 RTT 값을 내부 변수에 저장하고, getRtt 함수를 통해 저장된 RTT 값을 불러올 수 있다.

그림 5는 RTT 시작 시간을 측정하기 위한 코드로써 CtpForwardingEngineP.nc 파일에 추가된 코드이다. CtpForwardingEngineP.nc 파일은 패킷 전송과 관련된 기능이 구현되어 있다. 그림 5의 코드는 패킷 전송 함수기 호출되기 직전에 실행되어 RTT 시작시간을 저장한다.

그림 6은 전송 노드가 패킷을 전송하여 수신 노드로부터 ACK 신호를 받은 경우 호출되는 코드이다. 전송 노드는 ACK 신호를 받은 후 RTT 종료 시간을 측정하고, 그림 5에서 측정한 RTT 시작 시간과 차이를 계산하여 RTT 값으로 저장한다.

2. TCP 큐 설정 및 FAST TCP 알고리즘 구현

전송 계층은 III 장 1절에서 언급한 RTT 측정을 통해 윈도우 사이즈를 업데이트 하고, 그 값에 따라 TCP 큐를 제어한다[13]. 전송 계층에 TCP 큐를 설정하고 제어하기 위해서 그림 7, 8과 같은 코드 추가가 필요하다. 그림 7은 TCP 큐를 사용하기 위한 코드로써 message_t 변수는 TinyOS에서 사용하는 패킷 변수이고, 20은 큐의 크기가 된다. 설정한 TCP 큐를 사용하기 위해서는 그림 8과 같은 인터페이스를 설정한다. 큐 인터페이스와 TCP 큐를 연결하고 인터페이스를 통해 TCP 큐를 제어한다.

TCP 큐를 이용하여 패킷의 전송이 이루어지는 구현 코드는 그림 9와 같다. TinyOS는 각 기능의 동작이 타이머를 이용하여 이루어지기 때문에 그림 9에서 Timer는 패킷 전송을 담당하는 타이머이다. 타이머가 동작하면 TCP 큐의 상태를 확인하여 큐에 저장된 패킷을 불러온다. 이 패킷은 네트워크 계층과 연결된 인터페이스를 통하여 전송 계층으로 전달된다. 이 과정에서 큐는 패킷을 삭제하지 않고 그대로 유지한다. 전송 노드는 수신 노드에서 전송한 ACK 신호를 수신하면 큐에서 이 패킷을 삭제한다. 이 과정을 통해 USN에서 패킷 손실이 발생하는 것을 방지할 수 있다.

TCP 큐에서 패킷을 전송되는 부분과 저장하는 부분을 분리하기 위하여 타이머를 추가한다. 추가한 타이머는 일정 주기마다 TCP 큐에 패킷 저장을 수행한다. 수행 과정은 그림 10과 같다. 큐에 저장된 패킷 수를 파악하여 윈도우 사이즈보다 작으면 반복을 통해 패킷을 큐에 저장한다. 반대로 큐에 저장된 패킷 수가 윈도우 사이즈보다 크면 패킷을 큐에 저장하지 않는다.

```
components new QueueC(message_t, 20) as TQUEUE;
MVizC.TQUEUE -> TQUEUE;
```

그림 7. TCP 큐 설정.
Fig. 7. Setting of TCP queue.

```
interface Queue<message_t> as TQUEUE;
```

그림 8. TCP 큐 인터페이스.
Fig. 8. Interface of TCP queue.

```
event void Timer.fired() {
... //생략
if(call TQUEUE.empty())
        return;

sendbuf = call TQUEUE.head();

if (call Send.send(&sendbuf, sizeof(local)) == SUCCESS){
        sendbusy = TRUE;
}
else
        report_problem();
 ... //생략
}
```

그림 9. 패킷 전송 타이머의 동작.
Fig. 9. Operation of packet send timer.

```
event void TQueueTimer.fired() {
    ... //생략
  for( i = call TQUEUE.size() ; i < qwnd ; i++ ) {
    if (call Read.read() != SUCCESS)
        fatal_problem();

    memcpy(o, &local, sizeof(local));
    call TQUEUE.enqueue(sendbuf);
  }
}
```

그림 10. TCP 큐 타이머 동작.
Fig. 10. Operation of TCP queue timer.

```
event void Send.sendDone(message_t* msg, error_t error) {
 ... //생략
 call TQUEUE.dequeue();

 rtt = call CtpInfo.getRtt();

 if ( baseRTT > rtt && rtt != 0 )
        baseRTT = rtt;

 avgRTT = (avgRTT + rtt)*10;
 avgRTT = avgRTT / 20;

 temp = new_wnd * baseRTT / avgRTT;

 new_wnd = temp + alpha * 10 - new_wnd;
 new_wnd = new_wnd / 2;

 qwnd = new_wnd/10;
}
```

그림 11. 윈도우 사이즈 업데이트의 구현.
Fig. 11. Implementation of window size update.

그림 11은 윈도우 사이즈를 계산하기 위해서 식 (1)을 구현한 그림이다. CtpInfo 인터페이스를 이용하여 RTT를 측정하고, 평균 RTT 값을 구한다. 이 값을 이용하여 새로운 윈도우 사이즈를 계산하고 그 결과를 윈도우 사이즈 변수에 저장한다. TinyOS에서 사용하는 변수는 모두 양수이기 때문에 실수형 변수에 비해 정밀도가 떨어진다. 정밀도를 높이기 위하여 계산 과정에 10을 곱하고 결과 값을 이용할 때 10을 나누어 적용한다. 식 (1)에서 필요한 γ값은 0.5, α 값은 실험에 따라 다양한 양수 값을 적용할 수 있다.

그림 11에서 큐에 저장된 패킷의 삭제가 이루어진다. 이때 삭제하는 패킷은 그림 9에서 전송하고 수신 노드에서 보낸 ACK 신호가 확인이 된 패킷이다. 전송이 제대로 이루어지지 않거나 ACK 신호가 없을 경우 TCP 큐에서 패킷을 삭제하지 않기 때문에 큐의 크기가 변하지 않는다. 크기 변화가 없으므로, 그림 10의 TCP 큐 타이머가 동작해도 큐에 새로운 패킷을 저장하지 않는다.

IV. USN 실험

1. 실험 환경

실험에서 사용한 USN 하드웨어는 IRIS 플랫폼[14]이다. IRIS에 대한 사양은 다음 표 1과 같다. IRIS에서 사용하는 TinyOS의 버전은 2.x 버전이며 USN 상황을 GUI로 볼 수 있는 MViz를 이용한다. USN 구성은 전송 노드의 역할을 하는 센서 노드 6개와 수신 노드의 역할을 하는 싱크 노드 1개를 이용하여 구성한다. 2, 3, 4절의 실험에서는 센서 노드 1개와 싱크 노드 1개를 이용하여 실험하고 5, 6절의 실험에서는 센서 노드 6개와 싱크 노드 1개를 이용하여 실험한다.

2. TCP 적용 여부에 따른 큐 사이즈의 변화

TCP 적용 여부에 따라 실험을 진행한다. TCP 큐의 크기는 20, 패킷 전송 타이머 주기는 128 tick, TCP 큐 저장 타이머 주기는 128 tick으로 하여 실험한다. TCP를 적용했을 때 α값은 30으로 한다. 1024 tick은 대략 1초의 시간을 가지기 때문에 128 tick은 대략 1/9초의 시간이 된다. 각각의 실험에서 큐 사이즈의 변화를 확인해보면 그림 12와 같다.

TCP를 적용하지 않을 경우에는 TCP 큐에 대한 제어가 불가능하다. 패킷 전송 타이머와 TCP 큐 타이머는 동일한 주기로 동작하지만 각각의 동작은 독립적으로 이루어진다. 각 타이머의 동작을 확인해 보면 패킷 전송 타이머는 ACK 신호를 확인할 때까지 새로운 패킷을 전송하지 않는다.

표 1. IRIS 사양.
Table 1. IRIS specification.

IRIS H/W Spec.	
Model	XM2110CA
MCU	ATmega1281
Radio Chip	IEEE 802.15.4 RF Transceiver
RAM	8K bytes
Freq. Band	2.4 ~ 2.48GHz
TX data rate	250 kbps
S/W Spec.	
Version	TinyOS 2.X
Library	MViz

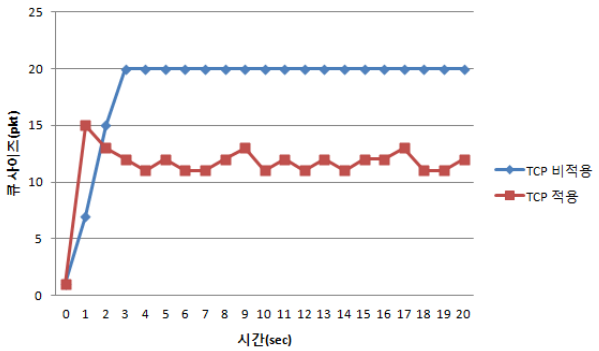

그림 12. TCP 비적용과 적용에 따른 큐 사이즈.
Fig. 12. Queue sizes without TCP and with TCP.

TCP 큐 타이머는 전송에 상관없이 주기적으로 큐에 패킷을 저장한다. 따라서 패킷 전송으로 큐에서 제거되는 패킷보다 저장되는 패킷이 더 많아지게 된다. TCP를 적용하지 않을 경우에는 TCP 큐는 일정 시간이 지나면 항상 20의 패킷을 저장하게 된다.

TCP를 적용하면 TCP 큐에 대한 제어가 가능하다. ACK를 통해 패킷의 전송이 완료되면 그림 11의 윈도우 사이즈 업데이트가 수행된다. 일정 시간이 지나면 그림 10의 TCP 큐 저장 타이머가 동작한다. 현재 TCP 큐의 크기를 확인하여 윈도우 사이즈보다 작으면 윈도우 사이즈만큼 새로운 패킷을 큐에 저장한다. 큐의 크기가 윈도우 사이즈보다 크면 패킷을 큐에 저장하지 않는다. 전송 타이머가 동작하여 큐의 크기가 윈도우 사이즈보다 작아질 때까지 새로운 패킷을 저장하지 않는다.

3. α 값에 따른 큐 사이즈의 변화

α 값의 변화에 따라 실험을 진행한다. TCP 큐의 크기는 20, 패킷 전송 타이머 주기와 TCP 큐 저장 타이머 주기는 128 tick으로 하고, α 값은 5, 10, 20, 30, 40, 50으로 하여 실험한다. 각각의 실험에서 큐 사이즈의 변화를 확인해보면 그림 13과 같다.

전송 노드에서 패킷 전송이 발생하고 큐 사이즈를 업데이트 하면 초기에는 큐 사이즈 수렴 값보다 높은 값으로 증가하지만 다음 주기에서 수렴 값으로 감소한다. 시간이 지나면 큐 사이즈는 1~2 pkt(packet)의 범위 안에 작게 진동한다. α값을 5로 했을 경우 큐 사이즈 값은 1~2 pkt에서

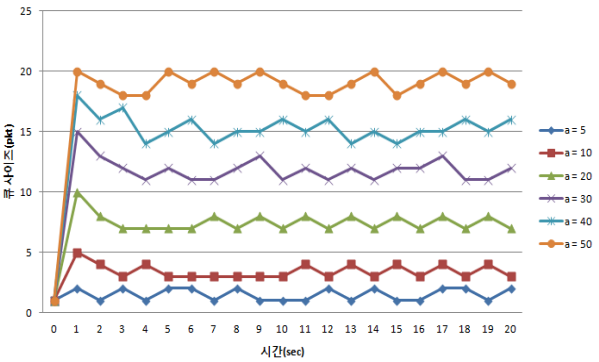

그림 13. α 값에 따른 큐 사이즈.
Fig. 13. Queue sizes for various α values.

진동한다. α값이 커질수록 수렴하는 큐 사이즈 값은 증가하고, α값이 50이 되면 큐 사이즈의 최대 크기 근처에서 작게 진동한다.

4. α 값에 따른 패킷 전송률의 변화

일정 시간 동안 패킷을 전송하고 ACK 신호를 받은 패킷 수를 분석하면 패킷 전송률을 파악할 수 있다[15]. TinyOS에서 사용하는 패킷 변수 형식은 messaget_t 로써, 51 byte의 크기를 가진다. 1초 동안 전송되는 패킷 수를 구하여 51 byte를 곱하면 전송률 단위는 Bps (bytes per second)가 된다. 실험을 통해 얻은 전송률은 그림 14와 같다.

그림을 보면 TCP를 적용하지 않은 경우 전송률이 가장 크다. 실험에서 패킷 저장 타이머와 패킷 전송 타이머를 동일 값으로 했기 때문에 패킷이 큐에서 지연되는 시간이 짧다. TCP를 적용했을 때 큐 사이즈가 윈도우 사이즈보다 작으면 윈도우 사이즈만큼 패킷을 저장하는 지연 시간이 발생한다. TCP를 적용하지 않으면 이러한 지연 시간이 발생하지 않지만, 패킷 저장 타이머의 주기에 영향을 받는다. 패킷 저장 타이머 주기가 전송 타이머 주기보다 느리면 큐에 저장된 패킷이 없는 경우가 발생한다. 이때 전송 타이머는 전송할 패킷이 없기 때문에 다음 주기에 동작한다. 큐에 패킷을 저장하면 전송 타이머가 동작하여 곧바로 전송하므로, 큐 사이즈는 항상 1의 값을 가진다. 전송 타이머 주기는 패킷 저장 타이머 주기보다 빠르지만 패킷 저장 타이머 주기에 맞춰 동작하게 된다. 실험에서는 패킷 저장 타이머와 전송 타이머 주기 값이 동일하여 그림 14와 같은 결과를 얻었지만, 두개의 타이머 주기가 다를 경우에는 TCP를 적용한 USN에 비해 더 낮은 전송률을 가지게 된다. TCP를 적용한 경우에는 항상 일정한 큐 사이즈 값을 유지하고 패킷 전송 타이머와 패킷 저장 타이머가 독립적으로 동작하므로 항상 최적의 전송률을 가진다.

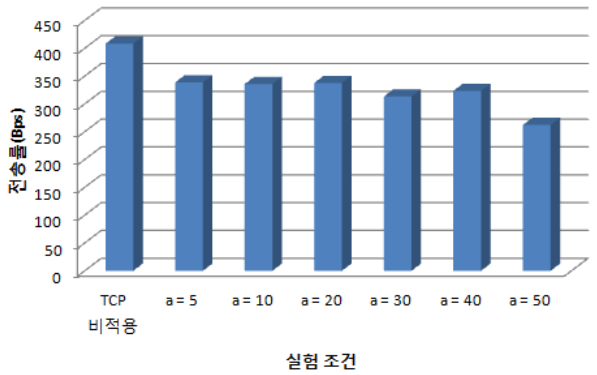

그림 14. α 값에 따른 전송률.

Fig. 14. Throughputs for various α values.

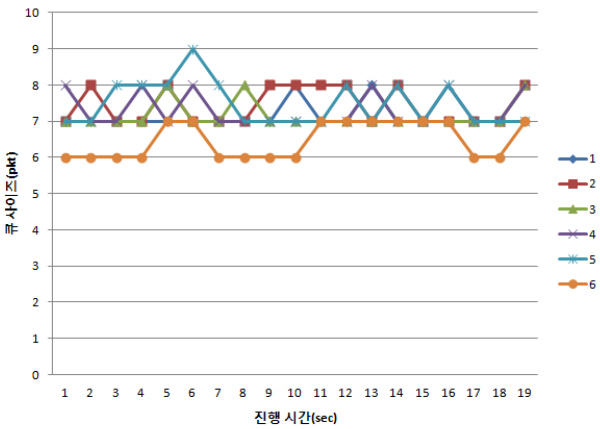

그림 15. 각 노드의 큐 사이즈.

Fig. 15. Queue size of each node ID.

5. TCP를 적용한 USN의 노드별 큐 사이즈

2, 3, 4절의 실험에서는 싱크 노드와 센서 노드를 1개씩 사용하여 실험했지만, 5, 6절에서는 1개의 싱크 노드와 6개의 센서 노드를 사용하여 실험한다. TCP 큐의 크기는 20, 패킷 전송 타이머 주기와 TCP 큐 저장 타이머 주기는 128 tick, α 값은 20으로 설정한다. 네트워크 토폴로지가 안정되어 전체 노드의 패킷 전송이 성공적으로 이루어지는 동안 각 노드의 큐 사이즈를 확인한다. 실험을 통해 얻은 각 노드의 큐 사이즈 값을 확인해보면 그림 15와 같다.

6개 센서 노드의 큐 사이즈를 보면 최소 6 pkt에서 최대 9 pkt의 사이에서 값이 변하는 것을 확인할 수 있다. 6번 노드를 제외한 나머지 노드는 대부분의 시간 동안 7~8 pkt 사이에서 값이 변한다. 6번 노드의 경우에도 6~7 pkt 사이에서 값이 변하므로 각 노드의 큐 사이즈 차이는 0~2 pkt 사이에서 변한다.

6. TCP를 적용한 USN의 노드별 패킷 전송률

전체 노드의 패킷 전송이 성공적으로 이루어지는 동안 각 노드에서 전송하는 패킷 수를 분석하면 각 노드별 패킷 전송률을 구할 수 있다. 4절에서 사용한 계산법을 이용하여 전송률을 계산하면, 각 노드의 패킷 전송률은 그림 16과 같다.

그림을 보면 TCP를 적용한 USN의 센서 노드는 비슷한 전송률을 가진다. 6개 노드의 평균 전송률은 336.18 Bps이고 최대 전송률은 1번 노드의 347.37 Bps, 최소 전송률은 6번 노드의 319.88 Bps이다. 최댓값과 최솟값의 차이가 27.49 Bps로 큰 차이가 없다. 패킷 크기가 51 byte이므로, 1초당 패킷 1개보다 작은 차이를 가진다. 그림 14와 비교해보면, 센서 노드 6개 실험의 전송률은 센서 노드 1개 실험의 전송률과 큰 차이가 없다. 따라서 TCP를 적용한 USN은 여러 개의 센서 노드를 사용해도 안정적으로 동작하며, 높은 네트워크 공평성을 가진다.

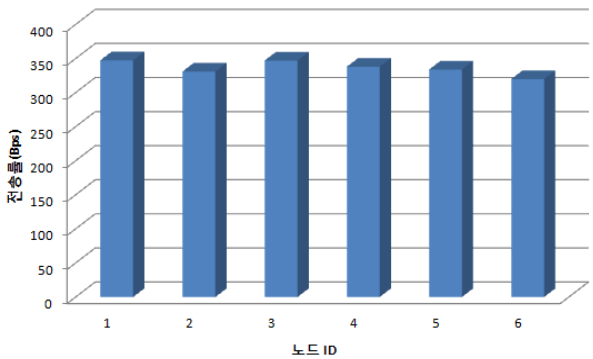

그림 16. 각 노드의 패킷 전송률.

Fig. 16. Throughput of each node ID.

V. 결론

본 논문에서는 전송 용량을 최대로 이용하고 패킷 손실을 피하기 위해 USN에서 전송 계층의 TCP 기능 설계 및 구현을 제안하였다. 구현 결과를 실험하기 위해 TCP 기능을 적용하지 않은 USN과 TCP 기능을 적용한 USN의 큐 사이즈 변화를 분석하고 α값을 달리 하여 전송 효율을 분석하였다. 좀 더 복잡한 네트워크 환경에서 실험하기 위해 센서 노드 6개를 사용함으로써, 각 노드의 큐 사이즈 값과 전송률을 분석하고 복잡한 네트워크에서 TCP 기능이 제대로 동작하는지 확인하였다.

전송 계층에서 패킷을 전송할 때, TCP 큐에 저장된 패킷을 바로 삭제하지 않고 전송이 확인이 된 경우에 삭제함으로써 패킷 손실이 발생하지 않게 하였다. TCP를 적용한 경우, TCP 큐 사이즈는 일정한 값에서 작게 진동 하였으며, α값의 차이에 따라 그 값도 일정하게 변하였다. 윈도우 사이즈 업데이트 알고리즘으로 이용하는 FAST TCP는 RTT를 이용하기 때문에 USN에서 발생하는 혼잡을 측정할 수 있다. TCP 기능을 적용한 USN의 전송률을 보면 α값의 변화에도 비슷한 전송률을 유지하는 것을 확인하였다. 마지막으로 여러 개의 센서 노드를 사용하여 실험 한 결과, 복잡한 네트워크에서도 TCP가 제대로 동작하며, 네트워크 공평성도 높다는 것을 확인하였다.

향후 연구과제로, 시뮬레이션을 통한 더 복잡한 네트워크 실험과 전송 효율 증가를 위한 알고리즘 개선에 대한 연구가 있다.

참고문헌

[1] IEEE Std 802.15.4-2006, IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local metropolitan area networks - Specific requirements, Part 15.4: Wireless Medium Access Control(MAC) and Physical Layer(PHY) Specifications for Low-Rate Wireless Personal Area Networks(WPANs), IEEE, NY, USA, Sep. 2006.

[2] "ZigBee Alliance," http://www.zigbee.org/

[3] "OSI 7 Model," http://en.wikipedia.org/wiki/Osi_7_layer_model

[4] "TinyOS," http://www.tinyos.net/

[5] M. Kim and N. H. Phong, "Reliable message routing protocol for periodic messages on wireless sensor networks," *Journal of Institute of Control, Robotics and Systems (in Korean)*, vol. 17, no. 2, pp. 190-197, Feb. 2011.

[6] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *Networking, IEEE/ACM Transactions on*, vol. 14, no. 6, pp. 1246-1259, Dec. 2006.

[7] H. Byun, "Improving TCP performance for wireless networks based on successive ECN," *Journal of Institute of Control, Robotics and Systems (in Korean)*, vol. 16, no. 8, pp. 816-822, Aug. 2010.

[8] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: a self-regulating algorithm for code maintenance and propagation in wireless sensor networks," *In Proc. of the First USENIX Conference on Networked Systems Design and Implementation (NSDI)*, USENIX Associatio Berkeley, CA, USA, pp. 1-15, 2004.

[9] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis, "Four bit wireless link estimation," *In Proc. of the Sixth Workshop on Hot Topics in Networks (HotNets VI)*, pp. 1-6, Nov. 2007.

[10] R. Gao, H. Zhou, and G. Su, "Structure of Wireless Sensors Network Based on TinyOS," *Control, Automation and Systems Engineering (CASE), 2011 International Conference on*, pp. 1-4, Jul. 2011.

[11] Y. Li, H. Chen, R. He, R. Xie, and S. Zou, "ICTP: An improved data collection protocol based OnCTP," *Wireless Communications and Signal Processing (WCSP), 2010 International Conference on*, pp. 1-5, Oct. 2010.

[12] J. Choi, K. Koo, and J. Lee, "Global exponential stability of FAST TCP with heterogeneous time-varying delays," *IEICE Transactions on Communications*, vol. E94B, no. 7, pp. 1868-1874, Jul. 2011.

[13] H. Yi, H. Kim, Y. Kim, and J. Choi, "RTT estimation method in WSN," *2012 International Conference on Information and Computer Networks (ICICN 2012)*, IACSIT Press, Singapore, vol. 27, pp. 17-21, 2012.

[14] MEMSIC. 2012, http://www.memsic.com/

[15] J. Carnley, B. Sun, and S. K. Makki, "TORP: TinyOS opportunistic routing protocol for wireless sensor networks," *Consumer Communications and Networking Conference (CCNC)*, 2011 IEEE, pp. 111-115, Jan. 2011.

이 현 철

2010년 부산대학교 정보컴퓨터공학부 졸업. 2012년 동 대학원 전자전기공학과 석사. 현재 동 대학원 박사과정. 관심 분야는 임베디드 시스템, 무선센서 네트워크.

최 준 영

1994년 포항공과대학교 전자전기공학과 졸업. 1996년 동 대학원 석사. 2002년 동 대학 박사. 현재 부산대학교 전자전기공학부 부교수. 관심분야는 임베디드 시스템, 제어 시스템.