무선 센서 네트워크 환경에서 게이트웨이 어플리케이션의 개발 환경을 위한 패킷 처리 관점의 네트워크 프레임워크 설계 및 구현

이호응** · 최대담* · 박현주**

Network framework design and implementation of packet processing perspective for development environments of gateway application in wireless sensor network environments

Hoeung Lee** · DaeDam Choi* · Hyun-Ju Park**

본 논문은 2단계 BK21 사업과 한국연구재단(사업번호 : 2010-0010904)의 지원을 받아 수행된 연구임

요 약

IEEE 802.15.4 기반의 WSN(Wireless Sensor Network)과 TCP/IP기반의 PN(Public Network)의 중계를 담당하는 WSN Gateway는 이 두 개의 네트워크를 기반으로 하는 핵심 기술에 속한다. WSN Gateway는 하드웨어에 대한 성능을 충분히 활용하면서, 다수의 센서 노드로부터 다양한 종류의 패킷을 불특정 시간에 수신해야 하기 때문에 소프트웨어에 대한 복잡성이 높고 구현이 난해하다. 이러한 문제들을 해결하기 위해, 이 논문에서는 패킷을 식별하는 효율적인 이벤트 검출 구조와, '트랜잭션'이라는 프로토콜의 구현 단위를 제안한다. 제안한 네트워크 프레임워크를 적용한 결과, 소프트웨어의 복잡성을 낮출 수 있었다. 또한, 다양한 성능 요구 사항을 반영할 수 있는 유연한 소프트웨어 구현 환경을 제공하였다.

ABSTRACT

WSN gateway which runs transmission between WSN (Wireless Sensor Network) based on IEEE 802.15.4 and PN (Public Network) based on TCP/IP belongs to core technology of application based on two network. Because WSN Gateway receives various kinds of packet from many sensor nodes in an uncertain time as well as uses hardware's performance enough, it has high level of complexity about software and it is hard to be implemented. To solve these problems, this paper suggests both efficient event detection scheme for identify packet and implementation unit of protocol called 'Transaction'. The results of applying the proposed network framework, complexity of software reduces. And we provide software development environments of reflect various performance requirements.

키워드

무선 센서 네트워크, 프레임워크, 미들웨어

Key word

Wireless sensor network, Framework, Middleware

* 준회원 : 한밭대학교 (dami3834@hanbat.ac.kr)
** 정회원 : 한밭대학교

접수일자 : 2011. 01. 06
심사완료일자 : 2011. 01. 28

Ⅰ. 서 론

IEEE 802.15.4 프로토콜[1] 기반의 WSN (Wireless Sensor Network) 기술이 발전하면서 환경 정보 감시, 원격 제어, 빌딩 자동화 등의 다양한 응용이 등장하였다[2].

이러한 응용의 범위가 점차 확대되면서 TCP/IP 기반의 공용 네트워크(Public Network, PN)에서 WSN에 접근하고자 하는 요구가 증가하였다. 하지만 WSN은 저전력, 저속 및 저비용에 초점을 맞춘 프로토콜이므로, 이 두 개의 네트워크를 연결하는 WSN Gateway 를 이용하여 상호 통신 하는 형태가 일반적인 시스템 구조이다.

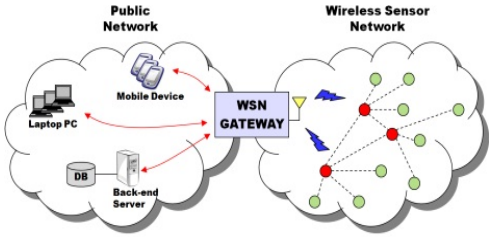


그림 1. PN와 WSN의 연결
Fig. 1 Connection of between PN and WSN

그림 1에서 WSN Gateway는 PN와 WSN의 접점에 위치하여 두 네트워크간의 통신 중계를 담당하는 핵심 기술에 속한다[3].

그림 2는 WSN Gateway의 일반적인 구조를 나타낸다. 이 그림에서 어플리케이션 영역은, 네트워크 프레임워크[4]를 기반으로 특정 어플리케이션을 위한 서비스 프로토콜을 구현하는 영역이다. 네트워크 프레임워크는, 네트워크 소프트웨어 개발을 지원하기 위한 공통된 기능들을 제공하는 소프트웨어이다. 네트워크 소프트웨어는 소프트웨어 개발의 생산성 향상을 위해 대부분 이러한 프레임워크를 기반으로 개발한다[5].

WSN Gateway에 대한 연구 동향을 살펴보면, 특정 센서 네트워크 응용을 위한 서비스 프로토콜 절차 및 그에 따른 추상적인 소프트웨어 설계에 대부분 집중하고 있다[6]. 제품에 대한 시장 적시성(Time-to-market)을 중요하게 생각하는 업계에서는, 개발자가 WSN Gateway를 위한 응용 소프트웨어를 신속하고 정확하게 구현 가능하도록 유도함과 동시에 주어진 응용에 특화된 성능 향상 정책을 쉽게 적용할 수 있는 유연한 네트워크 프레임워크도 중요한 요소로 고려한다.

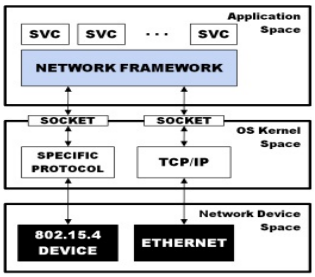


그림 2. WSN GWAY의 구조
Fig. 2 Architecture of WSN GWAY

다음은, 이러한 네트워크 프레임워크의 요구 사항과 기존의 네트워크 프레임워크에 대한 한계를 요약하고 본 논문에서 제안하는 특징을 나타낸다.

• 이벤트 감지를 위한 소프트웨어 구조

WSN Gateway는 다수의 상대측 센서 노드 및 호스트로부터 연속적인 데이터를 수신한다. 이 과정에서 네트워크 프레임워크가 데이터를 패킷 단위로 구성하고 각 패킷의 종류별로 이벤트를 감지한다면, 개발자는 패킷 이벤트 검출의 구현에 대한 작업의 부담을 줄일 수 있으므로 개발 생산성은 증가한다. 기존의 프레임워크는 이러한 기능을 지원하고 있지만, 연산 비용의 부담[7]이 있거나 필요 이상의 복잡성[8]을 내포하고 있다. 본 논문에서 제안하는 패킷 분류기는 다수의 디자인 패턴을 적용한 객체 지향적인 모듈이며, 패킷의 수신 및 패킷별 이벤트 감지를 담당한다. 또한, 패킷 분류기는 구조가 간단하고 유연하기 때문에 WSN의 다양한 패킷 종류를 쉽게 적용할 수 있다.

• 프로토콜 구현을 위한 소프트웨어 구조

다수의 센서 노드와 통신하는 WSN Gateway는 소프트웨어의 복잡도가 높은 영역에 속한다[4]. 이에 따라 프로토콜을 구현하는 개발자의 실수가 증가할 가능성이 많고, 소프트웨어에서 오류가 발생하면 찾기 어렵다. 일부 프레임워크에서는 이러한 문제점을 보완하기위한 소프트웨어 구조를 제시하고 있지만, WSN Gateway 의 응용에 적합하지 않고 복잡성을 내포하고 있다. 본 논문에서는 트랜잭션이라는 구현 단위를 제공한다. 모든 프로토콜에 대한 명세는 트랜잭션이라는 단위로 구현하도록 유도하기 때문에 구현 방법이 명확하고, 간단하며,

오류 검출이 용이하다.

• 테스크 처리를 위한 소프트웨어 구조

WSN의 다양한 응용에 따라서, 프로토콜에 대한 스케줄 정책이 달라질 수 있다. 스케줄 정책에 유연하게 대처하려면, 프로토콜의 명세에 대한 소프트웨어의 기능적 구현과는 별도로, 구현한 기능의 성능 향상을 위한 처리 방법을 자유롭게 변경할 수 있어야 한다. 기존의 프레임워크에서는 프로토콜의 구현 부분과 이를 처리하는 스케줄러를 분리하기 어려운 구조이다. 본 논문에서 제안하는 트랜잭션이라는 구현 단위는 프로토콜에 대한 절차만 구현하도록 설계하였으며, 구현된 다수의 트랜잭션에 대한 처리는 스케줄러에서 전적으로 처리한다. 그러므로 트랜잭션별로 스케줄 정책에 변화를 줄 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 네트워크 프레임워크의 한계 및 장단점을 분석한다. 3장에서는 본 논문에서 제안하는 네트워크 프레임워크를 제안한다. 4장에서는 성능 평가를 하고, 5장에서는 결론을 맺는다.

Ⅱ. 관련 연구

• 이벤트 감지를 위한 소프트웨어 구조

현대의 네트워크 어플리케이션의 개발은 네트워크 디자인 패턴[8]을 적용한 프레임워크[9] 기반에서 개발한다. 소켓 이벤트를 감지하는 대표적인 디자인 패턴은 Reactor디자인 패턴이 있다.

Reactor 디자인 패턴[8]은 다수의 소켓에 대한 이벤트를 감지하고, 특정 소켓으로부터 이벤트가 발생하면, 미리 등록한 핸들러를 호출한다. 그 외에도 Proactor 디자인 패턴[8] 등이 있다.

Gateway는 상대측 디바이스로부터 다양한 종류의 패킷을 수신하기 때문에 각각의 패킷을 세부적으로 구분할 필요가 있다. 따라서 소켓의 기초적인 이벤트와 핸들러를 연결하는 기존의 구조와는 달리, 본 논문에서는 패킷을 식별하는 효율적인 이벤트 검출 구조를 제안한다.

• 프로토콜 구현을 위한 소프트웨어 구조

네트워크 어플리케이션의 구현 용이성을 높이기 위한 디자인 패턴 및 네트워크 프레임워크 기술은 UNIX 시스템의 STREAM 과 CLICK등이 있다. 본 논문에서 제안하는 구조는, WSN을 위한 Gateway 어플리케이션 구현에 적합한 구조를 제공한다는 점에서 위의 기법과 다르다. 제안하는 구현 단위인 트랜잭션을 이용하면, 개발자는 프로토콜의 구현과 이에 대한 실시간/동시성 처리 방법을 분리하여 작업을 진행할 수 있다.

• 테스크 처리를 위한 소프트웨어 구조

WSN 어플리케이션의 종류에 따라서 Gateway가 작업을 처리하는 테스크의 스케줄러에 대한 요구사항이 달라질 수 있다. 패킷 처리의 높은 응답성을 요구하는 응용에서는 스레드 풀(Thread Pool)[10] 패턴, 리더/팔로워(Leader/Follower)[8] 패턴 등의 기법을 제공하여 하드웨어의 성능을 극대화 할 수 있다.

동시성 및 실시간 성능 제어를 위한 소프트웨어 디자인 패턴이 다수 존재 하지만, 요구 사항이 수시로 변화하는 WSN의 개발 환경에서는, 특정 상황에 적합한 동시성/실시간성 알고리즘을 도출하는 것 보다, 여러 알고리즘을 유연하고 신속하게 적용할 수 있는 소프트웨어 구조가 더 중요하다. 본 논문에서는, 태스크 처리를 일반화할 수 있는 소프트웨어 구조를 제공하여 Gateway 프로토콜 구현에 상관없이 태스크 처리 정책을 독립적으로 변경할 수 있는 소프트웨어 구조를 제공한다.

Ⅲ. 트랜잭션 기반 네트워크 프레임워크

3.1. 네트워크 프레임워크의 소프트웨어 구조

그림 3은 제안하는 네트워크 프레임워크의 소프트웨어 구조를 나타낸다. Transaction Generator와 Transaction Scheduler는 각각 트랜잭션의 생성 및 처리를 담당한다. Transaction Timer는 예약한 시점까지 트랜잭션의 수행을 지연할 수 있는 기능을 제공한다.

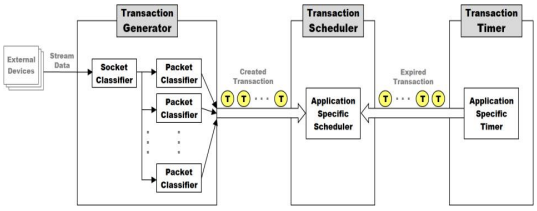


그림 3. 네트워크 프레임워크의 소프트웨어 구조
Fig. 3 Architecture of network framework

그림 3의 네트워크 프레임워크의 동작 흐름을, 외부의 디바이스로부터 패킷을 수신 했을 경우에 대한 예로 들면, 그 처리 절차는 다음과 같다.

· **트랜잭션의 생성 :** 그림 3의 왼편에서 외부 디바이스로부터 스트림 데이터를 수신하면, 소켓 분류기(Socket Classifier)와 패킷 분류기(Packet Classifier)에 의해 해당 디바이스의 프로토콜에 맞는 패킷 포맷을 기반으로, 스트림 데이터를 패킷 단위로 수신한다. 수신한 패킷에 대응하는 트랜잭션을 생성하고 이 트랜잭션을 스케줄러에게 전송하면 제너레이터의 역할은 처음으로 돌아가서 이 과정을 반복적으로 처리한다. 이 외에도 다양한 소켓 이벤트에 대해 트랜잭션과 연결할 수 있는 기능을 제공한다.

· **트랜잭션의 처리 :** 스케줄러는 제너레이터로부터 트랜잭션을 받으면 처리를 담당한다. 처리 정책은 스케줄러의 내부에 구현한 스케줄러 알고리즘 및 트랜잭션에 대한 처리 정책을 포함하는 어플리케이션에 따라 달라질 수 있다.

· **트랜잭션의 예약 :** 트랜잭션의 생성은, 제너레이터가 생성하는 경우뿐만 아니라, 트랜잭션 처리 과정 내에서 새로운 트랜잭션을 생성하는 경우도 있다. 그림 3의 타이머는 예약한 시점까지 트랜잭션의 실행을 지연한다. 타이머는 예약 시간이 만료된 트랜잭션을 스케줄러에게 넘긴다. 따라서 타이머는 트랜잭션의 예약 시점만 관리할 뿐 트랜잭션의 실행을 스케줄러에게 위임한다.

3.2. 트랜잭션의 기본 개념

트랜잭션은 본 논문에서 제안하는 프로토콜 구현 단위이자 스케줄러의 처리 단위로 사용하는 C++ 클래스이다. 그림 4는 트랜잭션의 클래스 구조를 나타낸다. 트랜잭션 클래스는 Execute() 메소드를 포함하는 추상 클래스이다. 프로토콜에 대한 트랜잭션의 구현은 이 클래스를 상속받아서 Execute() 메소드를 재정의 하여 구현할 수 있다.

하나의 트랜잭션에서 포함하는 처리 범위는, 특정한 패킷 수신에 대해 종속적인 일련의 프로토콜 처리 과정들이다. 트랜잭션은 네트워크 프레임워크의 제너레이터에 반드시 등록해야 사용할 수 있다. 그림 5는 트랜잭션의 등록 방법을 나타낸다.

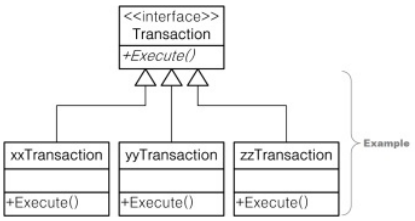


그림 4. 트랜잭션의 클래스 구조
Fig. 4 Class architecture of Transaction

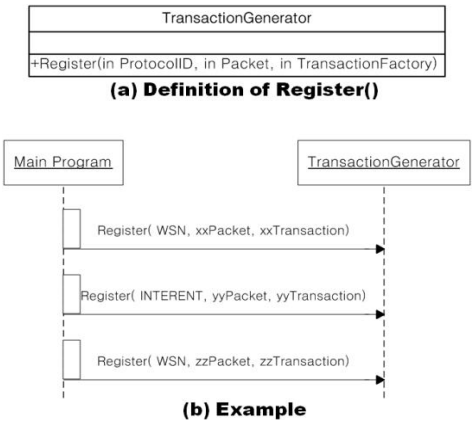


그림 5. 트랜잭션의 등록 방법
Fig. 5 Registration method of Transaction

그림 5에서 제너레이터의 Register() 메소드는 트랜잭션의 등록 요청을 처리하며, 파라미터들은 다음과 같이 두 가지로 분류할 수 있다.

· **패킷 :** Register() 메소드의 첫 번째 파라미터와 두 번째 파라미터의 결합으로, 트랜잭션에 대응하는 패킷을 나타낸다. 두 번째 파라미터는 패킷의 실제 내용을 포함하고 있는 버퍼 오브젝트이며, 첫 번째 파라미터는 이 패킷에 대한 프로토콜 식별자를 나타낸다. 프로토콜마다 고유의 패킷 포맷이 있으므로, 프로토콜 식별자를 활용하여 패킷의 내용을 해석할 수 있다.

· **트랜잭션 :** 세 번째 파라미터는 등록하는 트랜잭션을 나타낸다. 제너레이터는 패킷을 수신 했을 때 이 패킷에 대응하는 트랜잭션을 생성해야 하므로 트랜잭션 팩토리 오브젝트를 사용한다.

트랜잭션을 등록하면, 제너레이터는 프로토콜 종류에 따른 패킷 수신을 진행한 후 이 패킷에 대응하는 트랜잭션을 생성할 수 있다.

3.3. 트랜잭션 제너레이터의 설계

제너레이터는 다수의 소켓 이벤트를 감지하여 이에 대응하는 트랜잭션을 생성하는 역할을 담당한다. 그림 6은 제너레이터의 클래스 구조를 나타낸다. 제너레이터는 하나의 소켓 분류기와 다수의 패킷 분류기 오브젝트를 포함한다. 소켓 분류기는 다수의 소켓에 대한 이벤트를 감지한다.

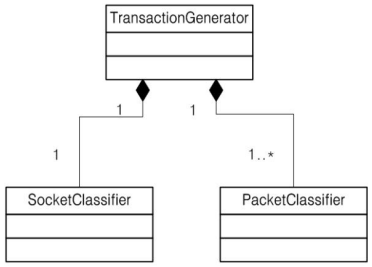


그림 6. 트랜잭션 제너레이터의 클래스 구조
Fig. 6 Class architecture of Transaction Generator

패킷 분류기는, 특정 프로토콜의 포맷에 따라 패킷을 수신하고, 이 패킷에 대응하는 트랜잭션을 생성하는 역할을 한다. 그림 7은 소켓 분류기와 패킷 분류기의 역할을 나타내는 개념도이며, 처리 과정은 다음과 같다.

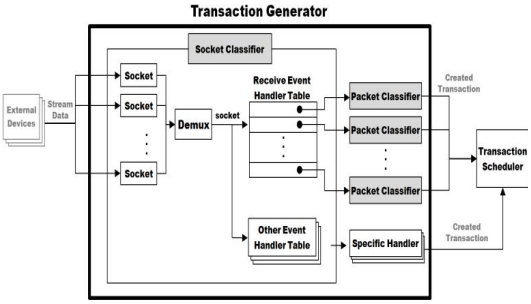


그림 7. 소켓 분류기와 패킷 분류기의 역할에 대한 개념도
Fig. 7 Concept about Socket Classifier and Packet classifier role

수신 이벤트가 감지되면 ReceiveEvent Handler Table에서 그에 해당하는 하나의 핸들러를 실행한다. 수신 이벤트에 대한 핸들러의 구현은 패킷 분류기이다. 패킷 분류기는 특정 프로토콜에 종속적인 패킷 포맷을 기반으로, 소켓으로부터 오는 연속적인 데이터를 패킷단위로 수신하는 역할을 한다. 패킷 분류기의 개수는, 이 네트워크 프레임워크 기반에서 접근하는 네트워크 프로토콜 종류의 개수만큼 존재해야 한다. 즉, WSN와 PN을 연결하는 WSN Gateway의 경우, 2개의 패킷 분류기가 필요하다.

· **소켓 분류기 :** 그림 8은 소켓 분류기의 클래스 구조를 나타낸다. 소켓에서 송신, 수신, 종료 등의 이벤트들이 발생할 때, 이에 대응하는 핸들러를 실행한다. 또한 소켓 분류기는 각 이벤트에 대한 다수의 핸들러를 저장하는 이벤트 테이블들이 존재하고 테이블은 핸들러를 0번부터 시작하는 슬롯 번호로 저장한다. 슬롯 번호는 특정 테이블에서 핸들러를 호출하기 위한 인덱스이다.

소켓 클래스는 자신의 소켓 이벤트에 대한 처리를 담당하는 핸들러 정보를 슬롯 번호로 저장한다. 그림 8에서 소켓 클래스의 멤버변수인 ReceiveEventSlot, SendEventSlot, CloseEventSlot, ExceptionEventSlot 은 각각 소켓의 수신, 송신, 종료 및 예외 이벤트에 대응하는 핸들러의 슬롯 번호이다. 그림 9는, 소켓 분류기의 실행 절차를 나타낸다. 소켓의 수신 이벤트가 발생한 경우를 예로 들면 다음과 같다.

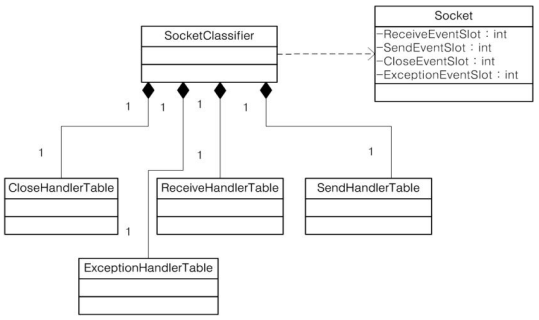


그림 8. 소켓 분류기와 소켓의 클래스 구조
Fig. 8 Class architecture of Socket Classifier and Socket

그림 9의 1단계에서, 멀티플렉싱[11] 기법에 의해 이벤트가 발생한 소켓을 추출한다. 2단계에서는, 소켓에서 어떠한 이벤트가 발생했는지 판단한다. 수신 이벤트가 발생한 경우라면, 수신 이벤트의 핸들러는 Receive HandlerTable에 슬롯 번호 순서로 저장하고 있으므로, 이 소켓에 대응하는 핸들러를 실행하려면 슬롯 번호가 있어야 한다. 그러므로 그림 9의 10라인에서는 소켓 오브젝트의 ReceiveEventSlot 값을 인덱스로 활용하여

ReceiveHandlerTable 에서 핸들러를 찾아 실행한다. 수신 이벤트 뿐만 아니라 다른 이벤트도 이와 동일한 방식으로 동작한다.

```
1 : void SocketEventLoop()
2 : {
3 : - Step 1 -
4 :         Socket = Demultiplexing();
5 :
6 : - Step 2 -
7 :         Event = Socket.Event;
8 :
9 : - Step 3 -
10 :        switch( Event )
11 :        {
12 :            case RECEIVE:
13 :                Slot = Socket.ReceiveEventSlot;
14 :                ReceiveHandlerTable[ Slot ].Execute( Socket );
15 :                break;
16 :            case SEND:
17 :                Slot = Socket.SendEventSlot;
18 :                SendHandlerTable[ Slot ].Execute( Socket );
19 :                break;
20 :            case CLOSE:
21 :                Slot = Socket.CloseEventSlot;
22 :                CloseHandlerTable[ Slot ].Execute( Socket );
23 :                break;
24 :            case EXCEPTION:
25 :                Slot = Socket.ExceptionEventSlot;
26 :                ExceptionHandlerTable[ Slot ].Execute( Socket );
27 :                break;
28 :        }
29 : }
```

그림 9. 소켓 분류기의 처리 절차
Fig. 9 Handling procedures of Socket Classifier

· 패킷 분류기 : 패킷 분류기는 소켓 오브젝트를 입력받아서 패킷 수신 처리를 한 후, 이 패킷에 대응하는 트랜잭션을 생성하는 역할을 담당한다. 그림 10은 이러한 패킷 분류기의 클래스 구조를 나타낸다. 패킷 분류기 클래스는 Receiver, Converter, Invoker 클래스를 포함하며, 하부 오브젝트들을 선택하여 조립할 수 있는 전략 패턴 구조이다. 하부 오브젝트들은, 특정한 프로토콜에 종속적인 오브젝트와 독립적인 오브젝트로 나눌 수 있다.

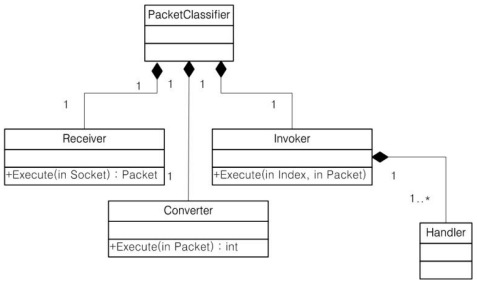


그림 10. 패킷 분류기의 클래스 구조
Fig. 10 Class architecture of Packet Classifier

예를 들어, Receiver와 Converter의 역할은 각각 특정한 프로토콜을 기반으로 패킷의 수신과 분석을 담당하므로 프로토콜에 종속적이다. Invoker는 인덱스에 대응하는 핸들러를 수행하므로 Invoker는 프로토콜에 독립적이다. 이러한 모듈들을 조합하여 특정 프로토콜을 위한 패킷 분류기를 완성할 수 있다.

그림 11은 패킷 분류기가 하부 모듈을 활용하는 처리 절차에 대한 개념도이다. 패킷 분류기의 실행은 Receiver가 소켓 입력을 받으면, 소켓에서 스트림 데이터를 패킷 단위로 수신한 후, 패킷 내용을 담은 패킷 오브젝트를 생성하여 반환한다. 반환된 패킷 오브젝트를 Converter 오브젝트에게 전달하고 Converter는 패킷의 내용을 분석하여 이 패킷의 종류를 식별하는 인덱스 값을 반환한다. 인덱스 값은 Invoker에게 전달하고 Invoker는 이에 대응하는 핸들러를 수행한다.

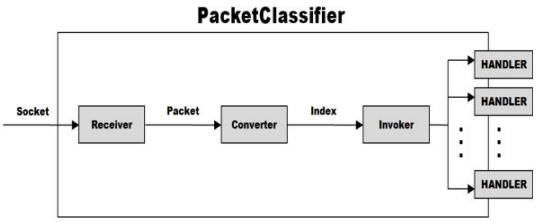


그림 11. 패킷 분류기의 처리 과정에 대한 개념도
Fig. 11 Concept about process of Packet Classifier

끝으로 이 핸들러는 자신의 맴버 변수에 있는 트랜잭션 팩토리 오브젝트를 이용해 트랜잭션을 생성하고 스케줄러에게 전달한다. 이러한 처리 과정을 반복하는 것이 패킷 분류기의 역할이다.

이러한 처리 과정에서 Invoker의 궁극적인 역할은, 입력받은 패킷에 대응하는 트랜잭션의 생성이다. 이러한 처리 과정을 만족하려면 트랜잭션의 생성을 담당하는 핸들러를 Invoker에 등록해야 한다. 그림 12는 트랜잭션의 처리 과정과 이에 대한 처리 과정을 시퀀스 다이어그램으로 나타낸다. 그림 12의 처리 절차는 다음과 같다.

1) 인덱스 값 추출 : 연결하고자 하는 패킷의 내용과 동일한 패킷 오브젝트를 생성한 후, Converter를 이용하여 이 패킷 오브젝트에 대한 인덱스 값을 받는다.
2) 트랜잭션의 팩토리 오브젝트 생성 : 연결하고자 하는 트랜잭션의 팩토리 오브젝트를 생성한다.

3) 트랜잭션의 등록 요청 : 1)에서 추출한 인덱스 값과 2)에서 생성한 팩토리 오브젝트를 Invoker의 Register() 메소드 호출의 인자 값으로 넘긴다.
4) 트랜잭션 생성을 위한 핸들러의 배치 : Register() 메소드는 매개 변수로 받은 인덱스 값에 대한 핸들러를 배치한다. 핸들러 오브젝트는 Register() 메소드의 두 번째 매개 변수인 트랜잭션 팩토리 오브젝트를 맴버 변수로 포함한다.

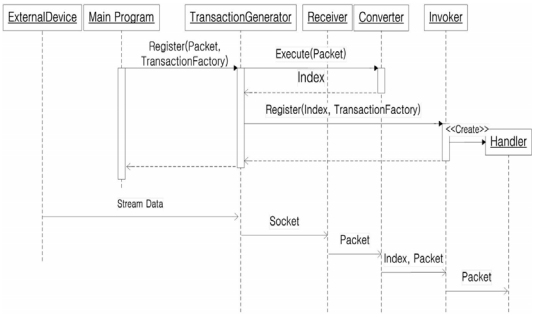


그림 12. 트랜잭션의 등록 절차
Fig. 12 Registration procedure of Transaction

이와 같이, Converter 는 네트워크 프레임워크의 초기 실행 단계에서 트랜잭션을 등록하는 용도와 네트워크 프레임워크의 실행 중에 수신 패킷에 대응 하는 트랜잭션 생성 과정에서 패킷을 분류하는 용도로 사용된다.

3.4. 트랜잭션 스케줄러의 설계

그림 13은 스케줄러의 클래스 구조이다. 스케줄러 클래스는 추상 클래스이기 때문에 이 클래스를 상속받아 구체적인 스케줄러를 구현할 수 있다.

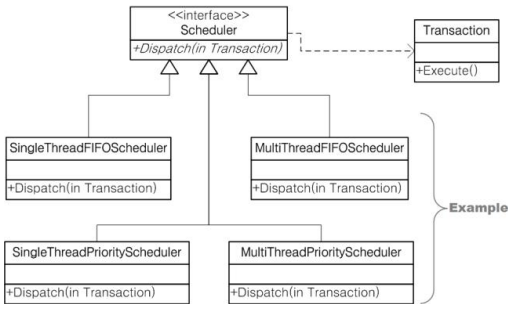


그림 13. 스케줄러의 클래스 구조
Fig. 13 Class architecture of Scheduler

스케줄러 클래스에서 제공하는 인터페이스는 Dispatch() 메소드이다. Dispatch() 메소드는 매개변수로 입력받은 트랜잭션을 수행한다. 스케줄러 클래스를 상속 받는 클래스들은 이 메소드를 재 정의하여 각 스케줄러 고유의 트랜잭션 처리 알고리즘 및 정책을 구현한다. 이러한 구조는 트랜잭션의 구현과 독립적이므로 개발자는 추후 여러 가지 스케줄러를 자유롭게 적용할 수 있다.

3.5. 트랜잭션 타이머의 설계

그림 14는 타이머의 클래스 구조이다. 타이머는 예약한 시점까지 트랜잭션의 실행을 지연하는 것이다. 타이머 클래스는 스케줄러 클래스와 마찬가지로 추상 클래스이기 때문에 이 클래스를 상속받아서 구체적인 타이머를 구현할 수 있다. 타이머에서 제공하는 대표적인 메소드는 Insert()와 Delete()이다. Insert() 는 트랜잭션의 수행을 일정 시점까지 지연하는 요청을 받는 메소드이다. Delete() 는 이미 타이머에 등록한 트랜잭션 예약을 취소할 때 사용한다. 개발자는 각 자료구조의 특성과 어플리케이션의 요구 사항을 반영하여 이에 적합한 타이머를 선택할 수 있다.

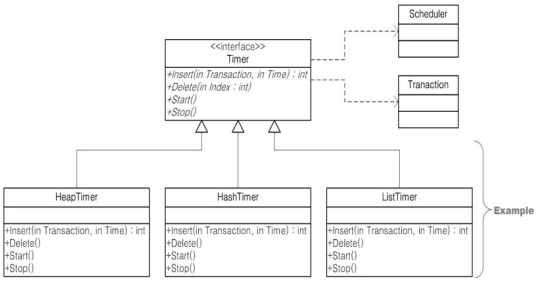


그림 14. 타이머의 클래스 구조
Fig. 14 Calss architecture of Timer

Ⅳ. 실험 및 고찰

본 장에서는, 제안하는 네트워크 프레임워크의 효용성을 검증한다. 먼저, 프로토콜의 구현에 대한 효용성을 확인하기 위해, 4.1절에서는 트랜잭션 및 패킷 분류기의 구현 예제를 보인다. 4.2절에서는 다양한 스케줄러 적용에 대한 실험 결과를 나타낸다. 이 실험 결과에서는, 트랜잭션에 대한 처리 방법을 사용자의 요구사항에 따라

유연하게 적용할 수 있음을 확인한다.

4.3절에서는 본 논문에서 제안하는 네트워크 프레임워크의 효용성에 대해 종합적으로 고찰한다.

4.1. 예제

4.1.1. 트랜잭션의 구현

본 절에서는 WSN Gateway의 응용 계층에 대한 대표적인 프로토콜 트랜잭션 구현 방법을 다음과 같이 제시한다.

• 유형 : 센싱 데이터의 수집

그림 15는 프로토콜 절차를 나타낸다. 이 유형의 예로는 센서 노드가 주변 환경 정보를 취득하여 이러한 데이터를 PN에 있는 백엔드 서버의 데이터베이스에 전송하는 경우를 들 수 있다.

그림 15의 시퀀스 다이어그램에서 네모 상자로 표시한 부분은 트랜잭션 구현 단위이다. 즉, 센서 노드로부터 수신한 패킷 종류에 따라 별도의 트랜잭션으로 처리할 수 있다. 그림 16은 그림 15에서A 유형의 패킷 수신에 대한 트랜잭션의 구현 예이다.

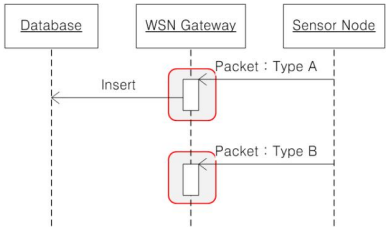


그림 15. 유형 1의 예제 : 센싱 데이터의 수집
Fig. 15 Example of Type 1 : Collection of Sensing data

```
- Step 1 : Implementing a Transaction
void GatheringTransaction::Execute()
{
    Data := Extract Data From the Packet with Type A;
    Insert To Database( Data );
    ...
}

- Step 2 : Registering the Transaction
{
    ProtocolID := WirelessSensorNetwork;
    Packet := MakeTypeAPacket();
    TransactionFactory := CreateFactoryForGatheringTransaction();

    TranactonGenerator.Register( ProtocolID, Packet, TransactionFactory)
}
```

그림 16. 트랜잭션 구현 예제
Fig. 16 Transaction implementation example

그림 16의 첫 번째 단계는, 트랜잭션의 구현이며, 수신 패킷의 내용을 분석하여 이 정보를 데이터베이스로 수집한다. 두 번째 단계는, 첫 번째 단계에서 구현한 트랜잭션을 등록하는 절차를 나타내며, 패킷과 트랜잭션을 연결한다.

4.1.2. 패킷 분류기의 구현

본 절에서는 패킷 분류기를 구성하는 하부 오브젝트들 중에서 프로토콜에 종속적인 Receiver와 Converter의 구현 방법을 제시한다.

그림 17은 패킷 분류기의 구현 대상으로 사용할 패킷 포맷의 정의에 대한 예이다. 그림 17의 패킷은 헤더와 페이로드로 구성하고, 헤더의 첫 번째 필드는 뒤에 있는 페이로드의 크기를 나타내는 필드로 정의한다. 이어서 패킷의 종류를 나타내는 Type 및 기타 여러 가지 필드들을 정의하고, 헤더의 길이는 고정한다. 페이로드는 헤더에서 정의한 Type 필드에 따라 내용이 달라질 수 있다.

예를 들어 그림 17의 예제에서 나타낸 바와 같이 헤더의 Type 필드에서 이 페이로드의 종류가 명령어임을 나타낸다면, 페이로드는 그림 17의 아랫부분과 같이 페이로드의 첫 번째 필드가 구체적인 명령의 종류를 나타내고 그 다음의 페이로드 부분은 해당 명령어의 종류에 따라 다시 해석할 수 있다. 그림 18은 패킷 포맷에 대한 Receiver의 구현 예이다.

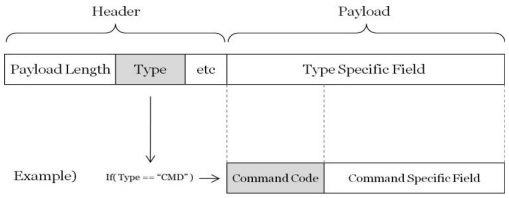


그림 17. 패킷 포맷 예제
Fig. 17 Packet format example

```
Packet exReceiver::Execute (Socket InputSocket)
{
    InputSocket.Receive( HeaderLength, Packet );

    PayloadLength := ExtractHeaderLength(Packet);

    InputSocket.Receive( PayloadLength, Packet );

    return Packet;
}
```

그림 18. Receiver의 구현 예제
Fig. 18 Implementation example of Receiver

Receiver에서는 패킷 단위로 스트림 데이터를 수신하는 역할이므로 이 패킷의 크기를 알아야한다. 위의 패킷 포맷 정의에 의해 헤더는 고정 길이이고 페이로드는 헤더의 PayloadLength 필드에 의해 추출 가능하다. 그러므로 그림 18의 Receiver 에서는 이러한 정보를 이용하여 패킷 단위로 수신한다.

그림 19는 그림 17의 패킷 포맷에 대한 Converter의 구현 예이다. Converter는 패킷의 내용을 분석하여 이 패킷을 유일하게 나타내는 식별자를 반환한다.

그림 19에서는 두 가지의 구현 예제를 제시한다. 첫 번째 예제는 헤더의 Type 정보만 구분한다. 따라서 이러한 Converter를 사용하는 패킷 분류기 환경에서는, Type을 구분하는 수준의 패킷 종류와 트랜잭션을 연결할 수 있다. 하지만 그림 19의 두 번째 예제는, 헤더와 Type과 페이로드의 Command 필드를 구분한다. 즉, 헤더의 Type 정보가 명령어를 나타냈을 때, 페이로드의 첫 번째 필드를 Command 필드로 인식하고 이 정보를 식별한다. 그리고 이 Converter의 경우 첫 번째 Converter보다 훨씬 더 정교하게 패킷과 트랜잭션을 연결할 수 있다.

```
int TypeConverter::Execute(Packet InputPacket)
{
    Type := ExtractType(Packet);

    Index := ConvertToInteger(Type);

    return Index;
}
```

(1)

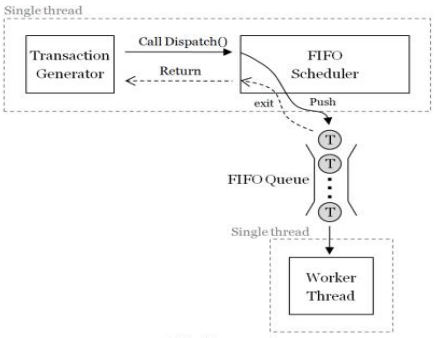
```
int TypeCmdConverter::Execute(Packet InputPacket)
{
    Type := ExtractType(InputPacket);
    if( Type == "CMD" )
        CommandCode := ExtractCmdCode(InputPacket);
    else
        CommandCode := "None";

    Index := ConvertToInteger(Type + CommandCode);

    return Index;
}
```

(2)

그림 19. Converter의 구현 예제
Fig. 19 Implementation example of Converter

4.1.3. 스케줄러의 구현

본 절에서는 스케줄러의 구현 예제를 제시한다. 다음에 제시한 스케줄러 외에도 콜백 스케줄러, 우선순위 스케줄러 등 다양한 스케줄러를 쉽게 응용에 따라 구현 가능하다.



(1) Basic concepts

```
void FifoScheduler::Dispatch(Transaction aTransaction)
{
    FifoQueue.Push(aTransaction)
}
```

(2) Pseudo code of Dispatch()

```
void WorkerThread::Execute()
{
    while(true)
    {
        aTransaction := FifoScheduler::FifoQueue.Extract();
        aTransaction.Execute();
        DestroyTransaction(aTransaction);
    }
}
```

(3) Pseudo code of WorkerThread

그림 20. 선입선출 스케줄러의 개념도와 구현 예제
Fig. 20 Concept and Implementation example of FIFO Scheduler

· **선입선출 스케줄러 :** 그림 20은 단일 스레드 스케줄러의 개념도를 나타낸다. 이 스케줄러는 스케줄러 내부에 트랜잭션 처리를 전용으로 담당하는 하나의 워커 스레드(Worker-Thread)가 있다. 제너레이터가 스케줄러에게 트랜잭션을 보내면 이 트랜잭션은 스케줄러의 내부에 있는 FIFO(First-In Fist-Out) Queue에 쌓이고, 다른 한 쪽에서는 워커 스레드가 이 Queue에 쌓인 트랜잭션을 추출하여 하나씩 차례대로 실행한다. 그러므로 제너레이터는 스케줄러에게 트랜잭션을 전달한 후, 즉시 자신의 일을 담당할 수 있다.

4.2. 스케줄러 변경에 따른 트랜잭션의 수행 결과

본 절에서는, 다양한 스케줄러의 적용에 따라 트랜잭션의 처리 결과가 달라질 수 있음을 확인한다. 그림 21은 실험을 위한 네트워크 구성을 나타낸다. 그림 21에서는 PN과 WSN을 포함하고 이 두 개의 네트워크를 연결하는 Gateway가 존재한다. PN의 내부에는 하나의 모바일 디바이스(Mobile Device)가 있고, 이 디바이스와 통신하는 Gateway가 다수의 센서 노드와 통신하는 구

조이다. 그림 21에 대한 센서 노드의 구성은, 실제 환경에서 수 백개 이상의 센서 노드를 실제로 구축해야 하는 한계를 보완하기 위해, 센서 노드의 동작을 모방한 가상의 소프트웨어를 이용하여 트래픽을 발생하는 것으로 대체한다.

또한, PN에서 동작하는 모바일 디바이스도 마찬가지로, 실제의 모바일 디바이스에 대한 동작을 모방하는 가상의 소프트웨어에서 트래픽을 발생시킨다. 표 1은 실험에서 사용한 사양을 나타낸다. 그림 22는 실험 시나리오를 나타낸다. A 영역은 Gateway가 센서 네트워크의 노드들로부터 센싱 데이터를 수신하여 이 정보를 데이터베이스에 수집하는 동작이다.

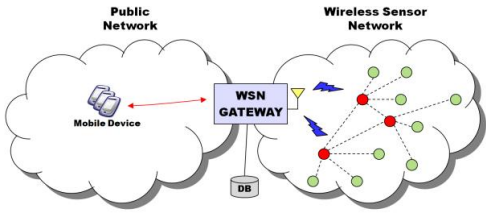


그림 21. 네트워크의 구성 및 실험을 위한 컴퓨터 구성
Fig. 21 Computer composition for experiments and composition of network

B 영역 은 PN의 호스트가 Gateway에게 데이터베이스에 수집된 정보를 요청하면, Gateway는 데이터베이스로부터 정보를 추출하여 응답 패킷으로 전송하는 흐름이다.

표 1. 실험 환경
Table. 1 Experiments environment

Mobile Device	CPU	Intel Core2 Duo 2.4Ghz
	Memory	2048MB
	WinCE	Fedora Core 8
WSN Gateway	CPU	Intel Core2 Duo 2.4Ghz
	Memory	2048MB
	OS	Fedora Core 8
	Database	MySQL 5.1
Virtual Sensor Node	CPU	Intel Core2 Duo 2.4Ghz
	Chip	2048MB
	OS	Fedora Core 8

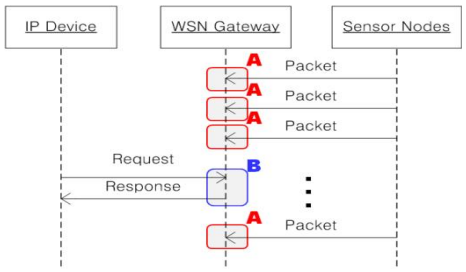


그림 22. 실험을 위한 프로토콜
Fig. 22 Protocol for experiments

이 실험에서는 Gateway가 센서 노드들로부터 수신 하는 센서 데이터의 증가로 인해 트래픽이 혼잡한 상황에서 PN의 호스트가 요청하는 명령에 대한 응답성을 각 스케줄러별로 확인한다. 특히, 우선순위 스케줄러의 경우 각 트랜잭션에 대한 우선순위를 부여할 수 있다. 이 실험에서 우선순위 스케줄러는 B 영역을 처리하는 트랜잭션에 대한 우선순위를 가장 높게 설정하였다. 표 2 ~ 3에 A 트랜잭션을 증가시켰을 때, B 영역의 응답성에 대한 실험 결과를 나타낸다.

표 2. 실험결과 : B 영역의 응답 시간 측정(단위 ms), A 트랜잭션 10,000개 발생.
Table. 2 Experiments result : Response time measurement of B area(unit ms), A Transaction 10,000 occurs

	1st	2nd	3rd	4th	5th	Average
FIFO Scheduler	2,326	2,168	1,617	2,325	1,921	2,019
Priority Scheduler	406	305	305	305	305	334

표 3. 실험결과 : B 영역의 응답 시간 측정(단위 ms), A트랜잭션 100,000개 발생.
Table. 3 Experiments result : Response time measurement of B area(unit ms), A Transaction 100,000 occurs

	1st	2nd	3rd	4th	5th	Average
FIFO Scheduler	32,230	32,762	31,316	31,417	31,925	31,786
Priority Scheduler	205	305	406	305	411	335

실험 결과, 선입선출 스케줄러는 트래픽의 증가에 따라 응답 시간이 길어지는 것을 확인할 수 있다. 반면, 우선순위 스케줄러는 B 영역에 대한 처리 우선순위를 높게 설정한 결과 트래픽의 변화에 영향을 받지 않고 짧은 응답 시간을 보인다. 그러므로 처리 성능에 대한 다양한 요구 사항을 해결하기 위해, 그러한 요구 사항을 지원할 수 있는 스케줄러를 적용하여 대응할 수 있다.

4.3. 고찰

위의 4.1절에서는, 제안한 네트워크 프레임워크 기반에서 프로토콜의 구현 예제를 보였고, 4.2절에서는 구현한 프로토콜 대해 요구 사항의 변화에 따라 다양한 스케줄러를 적용할 있음을 확인하였다. 본 절에서는 위의 이러한 결과들이 WSN을 위한 Gateway 개발 환경에 어떠한 이점을 제공하는지 네트워크 프레임워크 관점에서 고찰한다.

• 이벤트 감지를 위한 소프트웨어 구조

제너레이터의 하부 모듈인 소켓 분류기와 패킷 분류기는, 프로토콜을 인식하고 패킷을 수신하며 트랜잭션을 생성하는 역할을 자동화한다. 4.1.1절에서 제시한 트랜잭션의 구현에서는 개발자가 제너레이터를 이용하여 이벤트 처리 방법에 대한 기술적인 부분에 상관없이 프로토콜의 절차적인 동작에만 집중할 수 있는 환경을 제공하고 있음을 보였다. 또한 패킷 분류기가 패킷의 종류를 분석하는 상세 수준은, 프로토콜에 종속적인 최소한의 부분만 수정하면 구현할 수 있도록 구조를 제공하고 있으므로, 개발자는 전체적인 소프트웨어 구현의 틀을 유지하면서 패킷 이벤트 수신에 절차 및 알고리즘을 변경할 수 있다.

• 프로토콜 구현을 위한 소프트웨어 구조

WSN Gateway는 다수의 이종 프로토콜을 지원해야 하고 수많은 디바이스로부터 오는 패킷을 동시에 처리해야 하므로 소프트웨어의 복잡도가 높다. 제안하는 프레임워크에서는 이러한 복잡성을 낮추기 위해 트랜잭션이라는 개념을 도입하였다. 4.2절의 예제에서 살펴본 바와 같이, 트랜잭션은 한 가지 일을 명확히 전담하는 소규모 모듈이다. 그리고 이러한 트랜잭션이 여러 개가 모여서 전체 프로토콜의 구현을 완성한다.

복잡성에 대한 관점으로 4.2절의 트랜잭션 예제에서 주목해야 하는 주요 특징은, 트랜잭션이 오직 하나의 수신 패킷과 대응한다는 점이다. 만약 이러한 개념과는 반대로 하나의 트랜잭션이 다수의 패킷을 수신하는 구조로 정의한다면, 개발자는 트랜잭션의 구현에 대한 자유도는 높지만 단일 트랜잭션에서 다수의 역할을 담당할 가능성이 있다. 이러한 구조는 개발자가 소프트웨어의 기능 추가 및 수정을 반복하는 과정에서 특정한 방식으로 구현을 유도할 수 있는 규칙이 없다. 그러므로 소프트웨어의 복잡성을 낮추려면 개발자의 세심한 노력이 필요할 것이다. 하지만 제안하는 트랜잭션은, 오직 하나의 수신 패킷에 대한 종속적인 부분만 처리하도록 규칙을 적용하므로 특정한 프로토콜에 대한 소프트웨어의 구현이 직관적이고 단순한 구조로 소프트웨어를 개발할 수 있도록 유도한다. 결국 이러한 규칙을 기반으로, 단순하고 직관적인 소프트웨어 구조를 유지할 수 있기 때문에 구현한 소프트웨어에 대해 오류가 발생할 경우 신속하게 검출할 수 있다.

복잡성 이외에 유연성 측면도 고려 대상이다. 즉, WSN을 기반으로 하는 다양한 응용이 지속적으로 등장하면서, 기존에 개발한 응용 프로토콜을 변경해야 하는 요구 사항에 신속하게 대응할 수 있어야 한다. 이러한 경우에도 트랜잭션을 이용한 프로토콜의 구현은 장점을 발휘한다. 즉, 트랜잭션의 작성 규칙에 의해 하나의 트랜잭션에서 처리할 수 있는 작업의 영역이 명확하므로 트랜잭션간의 종속성이 낮다. 이러한 특징으로 인해 기능을 추가하거나 삭제할 때 다른 트랜잭션에 영향을 최소화할 수 있다. 따라서 이러한 요구 사항 변화에 신속하게 대응할 수 있으므로 유연성도 만족할 수 있다.

• 테스크 처리를 위하 소프트웨어 구조

개발자가 Gateway 프로토콜을 구현할 때 절차적인 측면과 성능적인 측면을 동시에 고려해야 한다면 상당히 난해한 작업이 될 것이다.

제안하는 프레임워크는 개발자가 프로토콜을 구현할 때 한 번에 하나의 일에만 집중할 수 있도록 하여 이러한 문제점을 해결한다. 4.1.1절에서는 트랜잭션을 이용한 프로토콜의 구현 예제를 보였다. 이 예제에 나타낸 바와 같이, 트랜잭션을 구현할 때 이 트랜잭션의 실행 방법이나 동시성 문제는 트랜잭션 내에서 고려할 필요가 없으며 이러한 문제는 스케줄러에서 전적으로 담당한다. 그리고 4.1.3절의 예제에서, 스케줄러는 트랜잭션에 대한 구체적인 실행 정책을 결정할 수 있음을 보였다. 즉, 트랜잭션을 구현할 시점에는 프로토콜의 절차적인 동작에 집중하게 하고, 스케줄러를 구현할 때에는 이에

대한 처리 방법에 집중하게 함으로써, 개발자에게 한 시점에 하나의 관점에만 초점을 맞출 수 있도록 유도한다. 이러한 특징에 따라, 개발자는 다수의 측면을 동시에 고려해야 하는 부담이 없다.

위에서 기술한 장점 이외에도, 기능적인 측면과 성능적인 측면을 분리하는 구조로 인해, 빈번한 요구 사항의 변경에 유연하게 대처할 수 있다는 부수적인 장점이 있다. 예를 들어, 프로토콜의 일부 절차에 대한 실시간 처리 요구 사항이 변한다면, 제안하는 프레임워크에서는 단지 스케줄러를 이용하여 해당 부분에 대한 처리 정책을 변경하는 것으로 대응할 수 있다. 이 과정에서 실시간 처리 처리 정책을 세분화하기 위해 트랜잭션을 더욱 작은 단위로 분할하는 경우가 아니라면, 트랜잭션에 대한 변경은 발생하지 않으므로, 실시간 처리 요구 사항의 변화에 신속히 대응할 수 있다.

V. 결 론

본 논문에서는 WSN Gateway를 위한 네트워크 프레임워크를 제안하였다. 제안하는 프레임워크의 구조는 제너레이터, 스케줄러, 타이머로 구성하였다. 네트워크 프레임워크를 구성하는 이러한 하부 모듈들은 트랜잭션 중심으로 각자의 역할을 담당하도록 하였다. 제너레이터에서는 다수의 프로토콜을 지원할 수 있는 구조이며, 소켓과 패킷 이벤트를 관리하여 이에 대응하는 트랜잭션을 생성한다. 특히 다수의 프로토콜 지원을 위한 개발자의 작업 부담을 줄이기 위해 프로토콜 종속적인 부분과 독립적인 부분을 명확히 분할하였고, 개발자는 프로토콜에 종속적인 최소한의 부분만 구현할 수 있도록 하였다. 트랜잭션 개념을 이용한 프로토콜 구현 기법은 소프트웨어 구현 및 처리 관점에서 다양한 장점을 제공하였다. 먼저 구현 관점에서 보면, 전체 프로토콜을 분할하여 '한 가지 일을 명확히 처리 하는 소규모 모듈' 단위로 구현을 유도하므로, 개발자에 대한 구현의 편의성을 제공하였다. 처리 관점에서 살펴보면, 프로토콜의 절차적인 구현과 이에 대한 처리 방법을 명확히 분할하여 개발할 수 있는 구조이므로, 개발자가 한 시점에 한 가지 관점에 집중할 수 있게 하였고, 이미 구현한 트랜잭션에 대한 실시간 처리 요구 사항이 변할 경우에도 유연하게 대처할 수 있는 구조를 제공하였다.

참고문헌

[1] Local and 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 (LRWPANS),IEEE Std. 802.15.4-2003, IEEE, 2003.

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

[3] Se-Jin Oh, Chae-Woo Lee, "u-Healthcare SensorGrid Gateway for connecting Wireless Sensor Network and Grid Network", The 10th International Conference on Advanced Communication Technology, February 2008.

[4] Douglas C. Schmidt, "C++ Network Programming, Volume I: Mastering Complexity with ACE and Patterns", Addison-Wesley, December 2001.

[5] Douglas C. Schmidt, "Applying a Pattern Language to Develop Application-level Gateways", Design Patterns in Communication Software, 1st edition, Cambridge University Press, September 2001.

[6] Yaoming Chen, Wei Shen, Hongwei Huo, Youzhi Xu, "A Smart Gateway for Health Care System Using Wireless Sensor Network", Fourth International Conference on Sensor Technologies and Applications, July 2010.

[7] Texudo Middleware, "http://www.oracle.com".

[8] Douglas Schmidt, "Pattern-Oriented Software Architecture, Volume 2 : Pattern for Concurrent and Networked Objects", WILEY, September 2000.

[9] ACE Network Framework, "http://www.ace.org".

[10] Grand, M., Patterns in Java, Volume 3: Java Enterprise Design Patterns. Second Edition, John Wiley & Sons (2002).

[11] W. Richard Stevens, "Unix Network Programming, Volume 1 : The Sockets Networking API", Addison-Wesley, November 2003.

저자소개

이호응(Hoeung Lee)

2004년 한밭대학교 정보통신공학과 학사
2006년 한밭대학교 정보통신전문대학원 정보통신공학과 석사
2011년 한밭대학교 정보통신전문대학원 전파공학과 박사
※관심분야 : 무선 MAC, 임베디드 소프트웨어, 임베디드 리눅스, 네트워크 시뮬레이터

최대담(DaeDam Choi)

2010년 한밭대학교 전파공학과 학사
2010년~ 한밭대학교 정보통신전문대학원 전파공학과 재학(석사)
※관심분야 : 임베디드 소프트웨어, 임베디드 리눅스, 무선통신 프로토콜, 네트워크 시뮬레이터, etc.

박현주(Hyun-Ju Park)

1997년 서울대학교 전산과학과 이학박사
2004년~ 2009년 한밭대학교 정보통신컴퓨터공학부 부교수
2009년~ 한밭대학교 정보통신 컴퓨터공학부 교수
※관심분야 : 공간 데이터베이스, 위치 추정 알고리즘, 실내 위치기반 서비스, 임베디드 리눅스, etc.