무선 센서 네트워크 운영체제 기술 동향 분석

Analysis architecture of embedded operating systems for wireless sensor network

강정훈, 유준재, 윤명현, 이명수, 임호정, 이민구, 황성일*
전자부품연구원, 맥스포*

Jeonghoon Kang, JunJae Yoo, Myunghyun Yoon, Myungsoo Lee,
Hojung Lim, Mingoo Lee, Sungil Hwang*
Korea Electronics Technology Institute, Maxfor Inc.*

Abstract - This paper presents an analysis architecture of embedded operating systems for wireless sensor network. Wireless multi-hop sensor networks use battery-operated computing and sensing device. We expect sensor networks to be deployed in an ad hoc fashion, with very high energy constraints. These characteristics of multi-hop wireless sensor networks and applications motivate an operating system that is different from traditional embedded operating system. These days new wireless sensor network embedded operating system come out with some advances compared with previous ones. The analysis is focusing on understanding differences of dominant wireless sensor network OS, such as TinyOS 2.0 with TinyOS 1.x.

Key Words : wireless, low power, sensor, network, operating system, multi-hop, ubiquitous computing

1. 서 론

센서 네트워크는 기존의 시스템들이 가지는 고려사항과는 다른 특징을 갖는다. 시스템 리소스는 응용 애플리케이션에 따라 사용여부가 결정된다. 멀티 태스킹 보다는 단일 태스킹이 사용되며, 단일 동작 보다는 다중의 협력기반 동작이 일반적이다. 이러한 새로운 특징들 때문에, 센서 네트워크 운영체제 설계는 하드웨어, 네트워크 프로토콜, 응용 서비스 전반에 대한 고려가 필요하다.

TinyOS나 MOS[1], SOS[2] 같은 경우는 시스템 flexibility를 최대한 지원한다. 오랜 시간동안 동작해야 하고 배터리 동작, RAM 등의 제한 조건을 지원하기 위해 효율적이고 flexible 하게 적용할 수 있는 컴포넌트로 구성되어야 한다. TinyOS는 센서 네트워크 특성에 적용될 수 있는 향상된 non-preemptive 스케줄러를 사용하고 있다. 애플리케이션들은 많은 컴포넌트 라이브러리로부터 구성된다. SOS는 flexiblity에 대한 고려가 더 많은데, 시스템이 동작하는 동안 컴포넌트를 추가하거나 제거할 수 있다. 이런 기능은 많은 개발자들과 연구자들에게 많은 발전의 가능성을 제공한다.

그러나 flexibility를 높이기 위해서는 필요한 기능이 있다. 효율적이기 위해서는, 컴포넌트는 사용에 대한 준비가 필요하다. 모든 사용 가능성에 대한 대비는 많은 코드 작성과 상태를 만들어낸다. 응용 애플리케이션이 여러 컴포넌트를 임의로 사용하게 되면, 이런 많은 문제점에 직면할 수 있다. 단순히 컴포넌트를 조합하여 애플리케이션을 만들어 내는 것 보다, 개발과정 중에 예상치 못했던 문제점을 찾아내야 하는 디버깅 시간과 작업의 비효율성이 생기게 된다. 새로운 플랫폼 기반으로 작업할 때는 이런 문제점이 더 크게 나타난다. 표준 API가 존재하지 않기 때문에, 새로운 플랫폼에 운영체제를 포팅 한다는 것이 명확한 작업으로 정의되지 못하는 문제이다. 애플리케이션은 많은 컴포넌트 라이브러리를 기반으로 하기 때문에, 모든 라이브러리를 모두 포팅 한다는 것은 효율적이지 못하다. 또한 어떤 부분이 OS인지를 나타내는 경계가 없기 때문에, 이런 점도 어려운 부분이다.

또한 새로운 하드웨어 리소스는 여러 가지 스케줄러 알고리즘에 대한 문제도 나타내는데, 초소한의 코드로 동작하는 non-preemptive, 지연된 호출 기법을 이용하는 경우, 시스템 오류의 기본적 문제를 제공한다. OS는 일정시간 동안의 재부팅, 워치독 타이머, grenade 타이머 등을 사용하여 재가동될 수 있다. 그러나 OS의 성능적인 측면에서 이런 방법은 완전한 해결 방안이라고 볼 수는 없다.

세 가지 문제점인 애플리케이션 복잡도, 새로운 플랫폼에 대한 포팅 작업, 신뢰성 문제는 운영체제의 근본적인 문제에 해당하지는 않는다. 대신 센서 네트워크가 발전되면서 급격히 요청되고 있는 사항이다. 어떤 abstraction과 레이어의 경계가 제공되어야 하는지에 대한 많은 요구사항이 정의되고 있다.

본문에서는 2세대 운영체제인 T2[3]를 설명한다. T2는 5년간의 TinyOS 커뮤니티 경험에서 만들어진 것이고, 애플리케이션에서 요구하는 시스템 flexibility를 정의한다. component architecture는 컴포넌트의 연결과 신뢰성 향상, 다양한 플랫폼 제공, 단순한 애플리케이션 개발을 제공한다.

T2는 TinyOS을 개선시킨 모델이다. 컴포넌트 기반으로 nesC로 개발되었다. 단일 쓰레드 컨트롤 형식을 가지고 있

으며 non-blocking 호출을 지원한다. 개념적으로는 비슷한 모델이지만, 세부적으로 T2는 많은 다른 특징을 갖고 있다. 좀 더 한정적인 concurrency 모델을 가지고 있으며, 새로운 boot sequence, 새로운 인터페이스를 지원한다. 이런 점들은 이전의 TinyOS에는 존재하지 않았던 특징들이다.]

2. 관련 기술 동향

2.1 TinyOS

TinyOS는 컴포넌트 기반, 이벤트 기반의 운영체제이다. 블로킹되는 호출은 존재하지 않는다. 대신 블로킹이 필요할 정도의 장시간의 동작을 수행하는 호출은 즉시 리턴을 보내주고, 리턴을 보내준 추후에 동작을 수행하고 수행이 종료되었을 때, 종료 상황을 알려준다. 따라서 호출은 두개의 방향으로 진행되는데, 이것을 split-phase라고 한다. 이 방법에서는 모든 컴포넌트들이 하드웨어 블록처럼 동작을 한다. 컴포넌트들은 명령을 받고, 이 명령이 종료되면 종료되었다는 이벤트를 상위로 알려준다. TinyOS의 concurrency 모델은 non-preemptive, 지연된 함수 호출 방식의 태스크에 기반하고 있다. 컴포넌트들은 추후의 실행을 위해 스케줄러에게 태스크를 호출하여 등록할 수 있다. TinyOS 스케줄러는 FIFO 방식의 고정 길이의 큐를 기반으로 한다.

모든 TinyOS의 코드들은 nesC로 작성되어 있다. nesC는 컴포넌트 형태의 C 언어이다. 프로그래밍을 할 때, 애플리케이션을 만들기 위해 TinyOS의 부트 시퀀스에 필요한 컴포넌트들을 연결한다.

2.2 The nesC programming language

nesC 프로그래밍 언어는 3가지 특성을 갖고 있다. 컴포넌트, 인터페이스, concurrency 모델이다. 컴포넌트는 두 가지 부분으로 구성되어 있는 소프트웨어이다. 두 가지 부분은 선언(specification)과 구현(implementation)이다. specification은 인터페이스에 대해 선언하는 것이고, implementation은 인터페이스의 logic을 구현하는 부분이다.

인터페이스는 컴포넌트의 양방향 연결에 대해 정의한다. downcall과 upcall의 split-phase가 어떻게 서로 연결되어 있는지를 정의한다. Downcall (command)를 호출하기 위해서는 컴포넌트가 upcall (event)를 정의해야 한다. 반대로 컴포넌트가 downcall을 구현해야만, upcall을 호출할 수 있다. 두 방향의 인터페이스를 구현하여 연결하는 것을 wiring이라 한다. 인터페이스로 컴포넌트의 두 방향이 연결되어 있기 때문에, nesC 프로그램은 함수 포인터가 필요 없다. 그리고 컴파일러는 양방향의 함수 호출을 최적화 할 수 있다.

nesC 동시성(concurrency) 모델은 TinyOS의 태스크 abstraction에 기반하고 있다. nesC 태스크는 다른 컴포넌트와 구별되어 독립적으로 실행된다. 태스크는 리턴 값이 없으며, 구동에 필요한 인자도 없는데, 구동에 필요한 인자는 일반적으로 태스크의 컴포넌트에 저장된다. 기본적으로 코드 호출은 태스크에서 시작된다. 인터럽트 핸들러는 async 키워드인 코드만을 실행시킬수 있다. async 함수의 예들은 A/D converter와 LED 구동 코드들이다. 태스크/async 특성은 인터럽트 핸들러가 필요할 때에 태스크를 실행하도록 한다.

2.3 802.15.4 and the CC2420

802.15.4는 저전력 무선 통신 규격이다. 비교적 높은 비트율을 낮은 전력인 15-20mA에서 구현 가능하기 때문에 센서 네트워크 시스템들에서 데이터 링크를 위한 기술로 사용된다. 현재 칩콘의 CC2420의 가장 유명한 하드웨어 이다.

CC2420은 패킷 인터페이스를 지원한다. 패킷이 수신되면 인터럽트를 발생시켜서 수신여부를 알려준다. 하드웨어가 성공적으로 패킷을 받으면 CC2420은 자동적으로 데이터 링크 레벨로 알려준다. 소프트웨어 스택은 수신된 바이트들을 칩에서 데이터 버스를 통해 전달 받는 역할을 수행한다. 만일 메모리가 오버플로우되면 무선 통신부는 패킷 수신을 멈추게된다. 또한 데이터를 송신할 때는 송신할 데이터를 CC2420의 메모리에 입력하고, 송신 명령을 내린다.

이런 단순한 하드웨어 인터페이스는 TinyOS가 복잡해지도록 하는 원인이 되고 있다.

2.4 TinyOS limitations

TinyOS의 세 가지 기본적인 한계점은 다양한 새로운 플랫폼 지원, 애플리케이션 구성, 신뢰성 있는 동작이다.

새로운 플랫폼에 TinyOS를 포팅하는 것은 많은 파일들을 복사하여 플랫폼에 맞도록 재구성해야 한다는 것이다. 새로운 플랫폼으로 옮겨가는 것이 기존의 플랫폼에서 속성을 이어 받아 재구성되는 방향이 아니라, 플랫폼들이 새로운 칩들로 구성되는 경우가 많기 때문에 새롭게 경우별로 구현을 해야 하는 문제점이 있다. 또한 TinyOS는 각 레이어별로 명확한 구분을 지어놓지 않았기 때문에, 직접 하드웨어를 제어하는 코드가 많아지게 되었다.

TinyOS 애플리케이션들은 컴포넌트들의 조합에서 오류가 발생할 수 있다. 많은 수의 컴포넌트를 사용하는 것이 예상치 못한 에러를 발생시키기도 한다. 예를 들면, Telos 플랫폼은 CC2420 무선 하드웨어와 플래쉬 저장 하드웨어를 SPI 버스를 통해 공유한다. 이 핀들은 외부 센서에도 연결될 수 있다. 애플리케이션에서는 이 모든 경우를 고려하여 제어할 수 있어야 한다. 예를 들어, 부트 시퀀스에서 통신 하드웨어와 플랫쉬 저장 하드웨어를 동시에 초기화하면 하나의 하드웨어에서는 실패한다.

CC2420 무선통신 하드웨어는 신뢰성을 위한 몇 가지 방법이 존재하기는 하지만, 궁극적인 해결방법을 제시하지는 못 한다. TinyOS CC2420 스택이 패킷을 수신하였을 때, 인터럽트에 따라 SPI를 통해 데이터를 전송받고 상위 레벨의 컴포넌트에게 패킷 도착을 태스크를 호출하여 알리게 된다. 태스크 큐는 여러 컴포넌트에 의해 공유되고 있는 리소스이기 때문에 호출된 태스크가 제대로 등록되어 실행되지 못할 수도 있다. CC2420은 하드웨어에서 패킷을 제대로 수신하여 통신 스택에 알려주려 하지만, 통신 스택에서 데이터를 전달 받아서 상위 애플리케이션에 제대로 전달하지 못 할수도 있다. 이런 태스크 호출을 지속적으로 시도하는 컴포넌트가 존재하지 않고 있다. 이전 플랫폼들의 경우는 일정한 주기의 인터럽트 처리가 시도되었지만, CC2420은 그렇지 못하다. 한가지 방법은 타이머를 사용할 수 있지만, 타이머도 태스크 큐를 사용하기 때문에 오류에서 안전하지 않다. 다른 방법은 수신할 때까지 인터럽트를 기다리는 것인데, 수신부의 메모리가 오버플로우 되면 스택은 인터럽트를 더 이상

받아들일 수 없게 된다. TinyOS에서는 이런 경우 하드웨어로 패킷이 수신되어도 처리하지 않고 버리게 된다.

더욱이 무선 통신 뿐 아니라 다른 컴포넌트의 동작에도 영향을 줄 가능성이 있다. 이런 split-phase 방식으로 이벤트를 시그널링 하는 경우는 이런 문제에 영향을 받는다. 이런 오류는 컴포넌트 사이에 계속해서 전달될 수 있고, 상위 레벨에 블로킹이나 신호 경쟁 문제를 발생 시킨다.

3. New Design Principles

3.1 Telescoping Abstraction

T2는 telescoping abstraction을 제공하여 일반적인 경우와 특수한 경우의 애플리케이션을 모두 만족 시킨다. Telescoping abstraction은 vertical과 horizontal 방향으로의 분석을 가능하게 한다. Vertical 방향은 각 하부시스템을 연결한다. 이 경우, 하부는 하드웨어와 관련된 인터페이스를 제공하며, 상위는 단순화된 하드웨어 독립적인 인터페이스를 제공한다. 이러한 vertical 방향의 분석은 개발자에게 자신이 구현하고 있는 부분에 대한 정확한 위치와 다양한 플랫폼에서 동작시켜야하는 애플리케이션을 쉽게 정의할 수 있다.

Horizontal 방향은 하부시스템의 컴포넌트들을 다른 플랫폼 하드웨어용으로 쉽게 포팅할수 있는 가능성을 제공한다. Mote 하드웨어들은 표준 하드웨어를 이용하여 구성되어 있다. 이런 하드웨어에 대해 정의된 인터페이스들은 다른 하드웨어에 포팅될때 쉽게 구현될 수 있다. 데이터 버스 같은 경우 다른 플랫폼에도 사용될 수 있다.

3.2 Partial Virtualization

T2는 세 가지 부분으로 나누어질 수 있다. Telescoping abstraction의 최상위 부분은 virtual 이고 shared로 사용된다. 이런 virtual 과 shared 특징은 nesC의 Service Instance pattern에 의해 지원된다. Telescoping abstraction의 최하위 레이어는 한 사용자에만 할당되는 physical 과 dedicated 특성을 갖는다. 세 번째는 physical 과 shared 이다. 위의 두개와는 다르게 이 경우는 실행 중에 특성이 결정된다. 먼저 설명된 두 가지는 컴파일시에 특성이 결정된다. 이런 특성 때문에 동시에 사용될 수 없어서, 사용하기 전에는 사용 신청을 해야 하고, 사용 후에는 사용 완료를 알려야 한다. 일반적으로 physical 과 shared 리소스는 Resource 인터페이스를 제공한다. 이것은 abstraction을 요청하고 점유완료에 대한 command가 정의되어 있다.

3.3 Static allocation / binding

nesC 컴파일 모델은 완전한 프로그램 분석을 지원하기 때문에, T2는 가능한 많은 할당과 연결을 컴파일 시간에 완료한다. 이런 설계는 flexibility를 제한하지만, 많은 OS의 기능들을 결정할 수 있다. Dynamic 방법은 어떤 경우에는 잘 맞지 않는다. 예를 들어 모든 컴포넌트들이 단 하나의 상태만을 필요로 하는 경우. 오랜 시간의 많은 수의 제어할 수 없는 환경은 이렇게 정해지지 않은 원인들에 의해 제대로 작동되지 않을 수 있다. Static 할당은 컴포넌트들이 자신들이 필요한 상태를 미리 할당할 수 있음을 의미한다. 어떤 컴포넌트가 한 패킷을 전송할 때, 패킷에 필요한 버퍼를 할당해야 한다. 어떤 경우는 많은 컴포넌트들이 서로 동시에 메세지를 전송하지 않도록 설계되어 있고, 그래서 하나의 버퍼만 필요할 수 있다. 그러나 이런 경우 최대 경우를 대비하여 버퍼가 static으로 할당되어야 한다. Static으로 할당하는 것은 많은 인터페이스와 패러미터, 함수들을 컴파일 시에 결정하도록 한다.

3.4 Service Distribution

이런 컴포넌트 조합 방식은 개발자들에게 애플리케이션 개발 시에 많은 편의성을 제공한다. 반면 이런 방식은 일정 규모의 애플리케이션 개발 시에, 컴포넌트들 간의 충돌 때문에 많은 시간 소모를 유발할지 모른다. T2 컴포넌트는 다양한 애플리케이션들에서 사용될 수 있어야 하기 때문에, 기본적인 동작기능과 특징들을 상위레벨에 제공해야 한다. 이런 모든 기능을 컴포넌트에 입력할수록, 애플리케이션의 복잡도는 증가하게 된다. Distribution은 내부적으로 정상동작을 가정하고 하부 컴포넌트들과 연결된다. 결국, service distribution은 서로의 특성을 공유하게 되며, 애플리케이션들은 이런 동작을 수행하지 않아도 된다.

4. 결론

TinyOS는 현재 가장 많이 사용되고 있는 센서네트워크 운영체제이다. nesC를 사용하고 있으며, 다른 센서네트워크 운영체제인 MOS 와 SOS는 C 기반으로 기존의 운영체제 구현 방법을 취하고 있다.

센서 네트워크 기술은 적용 애플리케이션 분야가 다양하기 때문에 하드웨어 지원과 다양한 응용 서비스 특성을 지원할 수 있도록 운영체제가 설계되어야 한다. nesC를 사용한 새로운 형태의 운영체제 제안은 센서네트워크 애플리케이션 개발에 많은 장점을 제공한다. 그러나 센서 네트워크 운영체제의 사용 관점에서는 보다 쉬운 개발과 사용을 위한 운영체제 기능에 대한 추가적 연구가 향후 필요하다.

참고문헌

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors. In Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), 2003.

[2] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In Proceedings of the Third Internaltiona Conference on Mobile Systems, Appliactions, and Services (Mobisys), 2005.

[3] Philip Levis., David Gay, Vlado Handziski, Jan-Hinrich Hauer, Ben Greenstein. Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, Joe Polastre, Philip Buonadonnao, Lama Nachman, Gilman Tolleo, David Cullero, and Adam Wolisz, T2: A Second Generation OS For Embedded Sensor Networks. Technical Report TKN-05-007, http://csl.stanford.edu/~pal/pubs/tkn-05-007.pdf