주기 및 비주기 태스크의 효율적인 관리를 위한 실시간 센서 노드 플랫폼의 설계

김 병 훈[†] · 정 경 훈[††] · 탁 성 우[†††]

요 약

본 논문에서는 주기 및 비주기 태스크의 효율적인 관리를 제공하는 실시간 센서 노드 플랫폼을 설계하고 구현하였다. 기존 센서 노드의 소프트웨어 플랫폼은 제한된 센서 노드의 자원을 효율적으로 사용하기 위하여 메모리 및 전력 소비량의 최소화에만 초점을 두었기 때문에 태스크의 실시간성과 빠른 평균 응답시간을 보장하는 실시간 센서 노드의 소프트웨어 플랫폼에는 적합하지 않다. 이에 본 논문에서는 센서 노드의 소프트웨어 플랫폼으로 많이 사용되고 있는 TinyOS 기반에서 태스크의 실시간성과 빠른 평균 응답시간을 보장할 수 있는 기법과 한계를 분석하였으며, 모든 주기 태스크가 마감시한 내에 실행이 완료되는 것을 보장하고 비주기 태스크의 응답시간을 최소화하는 실시간 센서 노드 플랫폼을 제안하였다. 본 논문에서 제안한 플랫폼은 Atmel사의 초경량 8비트 마이크로프로세서인 Atmega128L이 탑재된 센서 보드에서 구현되었다. 구현된 실시간 센서 플랫폼의 성능을 분석한 결과, 모든 주기 태스크의 마감시한 보장을 제공함과 동시에 향상된 비주기 태스크의 평균 응답시간과 낮은 시스템의 평균 처리기 이용률을 확인할 수 있었다.

키워드 : 실시간 센서 노드 플랫폼, 태스크 기반의 경량 TCP/IP, 멀티 태스크 기반 소프트웨어 컴포넌트, 센서 네트워크, 유비쿼터스 컴퓨팅

Design of a Real-time Sensor Node Platform for Efficient Management of Periodic and Aperiodic Tasks

Byounghoon Kim[†] · Kyunghoon Jung[††] · Sungwoo Tak[†††]

ABSTRACT

In this paper, we propose a real-time sensor node platform that efficiently manages periodic and aperiodic tasks. Since existing sensor node platforms available in literature focus on minimizing the usage of memory and power consumptions, they are not capable of supporting the management of tasks that need their real-time execution and fast average response time. We first analyze how to structure periodic or aperiodic task decomposition in the TinyOS-based sensor node platform as regard to guaranteeing the deadlines of all the periodic tasks and aiming to providing aperiodic tasks with average good response time. Then we present the application and efficiency of the proposed real-time sensor node platform in the sensor node equipped with a low-power 8-bit microcontroller, an IEEE802.15.4 compliant 2.4GHz RF transceiver, and several sensors. Extensive experiments show that our sensor node platform yields efficient performance in terms of three significant, objective goals: deadline miss ratio of periodic tasks, average response time of aperiodic tasks, and processor utilization of periodic and aperiodic tasks.

Key Words : Real-time Sensor Node Platform, Task-based lightweight TCP/IP Protocol stack, Multitask-based Software Components, Sensor Networks, Ubiquitous Computing

1. 서 론

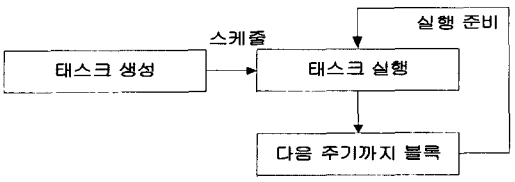
센서 네트워크는 다양한 센서와 무선 통신 장치를 탑재한 초경량 센서 노드간의 자율 통신을 형성하여 특정 지역의 상황 정보를 실시간 모니터링 하는 것이 목적이다. 센서 노드는 감지된 상황 정보를 수집 노드(Sink node)로 전송한다. 또한 자가 치유(Self-Healing) 및 자가 구성(Self-Organization)의 지원, 그리고 모든 작업의 실시간성을 보장하기 위하여 하드웨어 및 소프트웨어의 자원을 효율적으로 관리하는 것도 센서 노드의 역할이다. 이와 더불어 하드웨어 자원이 제한적인 센서 노드에서는 태스크의 실시간성 보장 및 빠른 평균 응답시간, 그리고 낮은 시스템의 평균 처리기 이용률을 제공하는 실시간 센서 노드 플랫폼이 필요하다. 실시간 센서 노드 플랫폼에서 수행되는 작업은 태스크로 구체화가

※ 본 연구는 부산대학교 자유 과제 학술연구비(2년)에 의하여 연구되었음.
† 준 회 원 : 부산대학교 컴퓨터공학과 석사과정
†† 정 회 원 : 부산대학교 U-Port 정보기술 산학공동사업단 전임연구원
††† 종신회원 : 부산대학교 컴퓨터 및 정보통신연구소 겸임 연구원 (교신저자)
논문접수 : 2007년 3월 15일, 심사완료 : 2007년 7월 2일

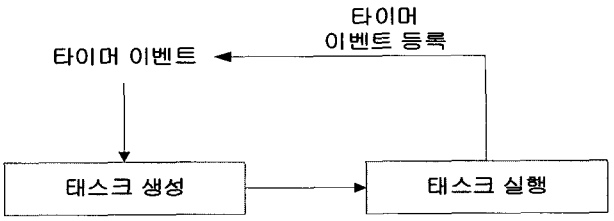
되며 태스크의 속성에 따라 주기 또는 비주기 태스크로 구분할 수 있다. 주기 태스크는 일정한 시간 간격마다 실행되어 마감 시한 내에 반드시 실행이 완료되어야 하는 태스크이다. 주기 태스크는 주기(Period), 수행 요청 시간(Release Time), 실행 시간(Computation Time) 그리고 마감 시한(Deadline)의 속성을 가진다. 반면 비주기 태스크는 주기와 마감시한이 없고 태스크의 수행 요청이 임의로 발생한다. 센서 노드는 마감시한이 없는 비주기 태스크에게 빠른 응답 시간을 제공할 수 있어야 한다. 그러나 비주기 태스크의 응답 시간을 줄이기 위하여 센서 노드의 자원을 비주기 태스크에게 우선적으로 할당하면 주기 태스크의 마감시한을 보장하지 못할 수 있다. 이와 반대로 주기 태스크에게 센서 노드의 자원을 우선적으로 할당하면 비주기 태스크의 응답 시간이 커질 수 있다. 따라서 실시간 센서 노드 플랫폼은 모든 주기 태스크가 마감 시한 내에 실행이 완료되는 것을 보장함과 동시에 비주기 태스크의 평균 응답 시간을 최소화해야 한다. 한편, 센서 노드의 배포 목적에 따라 센서 노드 간의 통신 패턴이 다양하게 나타나므로 센서 노드에 탑재되는 통신 프로토콜도 배포 목적에 맞게 최적화가 가능한 조립형 모듈로 되어야 한다. 실시간 센서 노드 플랫폼의 통신 프로토콜 스택은 배포 목적에 따라 쉬운 재구성과 태스크의 실시간성 보장을 제공하기 위하여 개별 프로토콜을 태스크로 구성되어야 한다. 또한 센서 노드의 자원은 제한되어 있으므로 실시간 센서 노드 플랫폼의 메모리 사용량과 단위 시간당 평균 전력 소비량을 최소화해야 한다. 특히 전력 소비량은 센서 노드의 수명에 직접적인 영향을 주기 때문에 센서 노드 플랫폼의 오버헤드를 최소화해야 한다. 그러나 메모리 및 전력 소비량의 최소화만을 고려하는 센서 노드의 소프트웨어 플랫폼에서는 주기 및 비주기 태스크의 실행이 요구되는 실시간 센서 노드 플랫폼의 요구 사항을 만족시킬 수 없다. 이에 본 논문에서는 주기 및 비주기 태스크의 효율적인 관리를 제공하는 실시간 센서 노드 플랫폼을 설계하고 구현하였다. 본 논문의 구성은 다음과 같다. 2장에서는 기존 센서 노드의 소프트웨어 플랫폼에 대한 문제점을 기술하였고, 3장에서는 본 논문에서 제안한 실시간 센서 노드 플랫폼의 구조에 대하여 기술하였다. 4장에서는 제안한 실시간 센서 노드 플랫폼의 성능을 분석하였으며, 마지막으로 5장에서는 결론을 기술하였다.

2. 관련 연구

센서 노드의 소프트웨어 플랫폼을 위한 운영체제는 1990



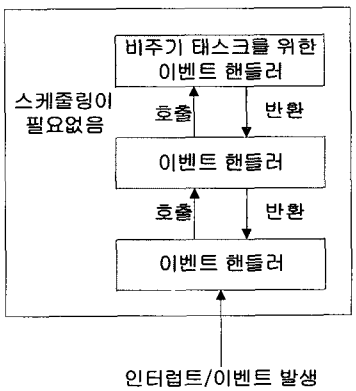
(그림 1) 블록 기반 주기 태스크 구현 기법



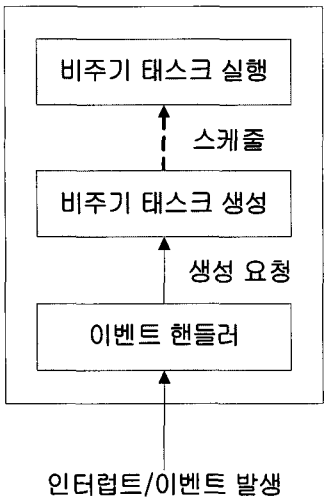
(그림 2) 이벤트 기반 주기 태스크 구현 기법

년대 말부터 시작되었으며, TinyOS[1], MANTIS[2], DCOS[3], 그리고 AvrX[4]와 같은 다양한 운영체제가 개발되었다. 이러한 센서 네트워크 운영체제는 동작 방식에 따라 이벤트 기반 운영체제와 멀티 태스크 기반 운영체제로 구분된다. 일반적으로 이벤트 기반 운영체제가 멀티 태스크 기반 운영체제에 비하여 적은 메모리와 태스크의 실행 오버헤드가 적기 때문에 소형 센서 노드에 많이 운용되고 있다. 이벤트 기반의 센서 네트워크 운영체제로는 TinyOS가 국내외에서 많이 사용되고 있다. TinyOS는 FIFO(First-In First-Out) 방식의 비선점형 태스크 스케줄링을 사용하며, 모든 태스크와 ISR(Interrupt Service Routine)이 단일 스택을 공유한다. 따라서 태스크의 문맥을 저장하는데 필요한 메모리와 문맥 전환에서 발생하는 오버헤드가 작다. 그러나 TinyOS는 태스크의 실시간성을 고려하지 않기 때문에 개발자가 직접 센서 노드 플랫폼에서 실시간성과 빠른 응답시간을 보장하는 실시간 스케줄링 기법을 구현해야 한다. (그림 1)과 (그림 2)는 TinyOS에서 주기 태스크를 구현하는 기법이다.

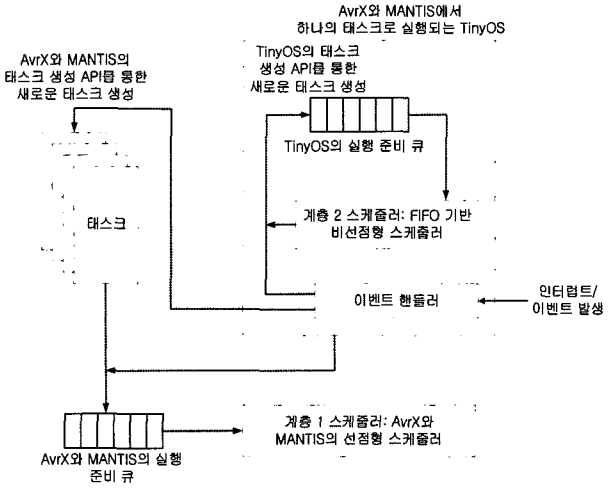
(그림 1)의 블록 기반 주기 태스크 구현 기법은 주기 태스크가 실행 완료되면 다음 주기 시작 시간까지 주기 태스크를 블록한다. (그림 2)의 이벤트 기반 주기 태스크 구현 기법은 주기 태스크가 실행 완료될 때마다 다음 주기 태스크의 생성 작업을 주기적인 타이머 이벤트에 등록하여 태스크를 생성한다. TinyOS는 비선점형 운영체제이기 때문에 실행중인 태스크를 강제 블록할 수 없다. 이에 (그림 2)에서 제시한 이벤트 기반의 주기 태스크 구현 기법을 사용해야 한다. 비록 (그림 2)에서 제시한 기법을 사용하더라도 TinyOS의 비선점형 스케줄링 기법에서는 현재 실행 중인 주기 태스크보다 마감시한이 임박한 다른 주기 태스크가 실



(그림 3) RTC (Run-To-Completion)기반 비주기 태스크 구현 기법



(그림 4) SPO(Split-Phase-Operation)기반 비주기 태스크 구현 기법



(그림 5) TinyOS를 태스크로 실행하는 AvrX와 MANTIS의 구조

행 선점을 하는 것이 불가능하기 때문에 주기 태스크의 실시간성을 보장할 수 없다. 또한 비선점형 스케줄링 기법에서 주기 및 비주기 태스크의 최적 스케줄링은 NP-hard 문제임이 이미 증명되었다[5-6]. 따라서 TinyOS에서는 주기 태스크의 마감시한을 보장할 수 없다. 한편, (그림 3)과 (그림 4)는 TinyOS에서 비주기 태스크를 구현하는 기법이다.

(그림 3)의 RTC (Run-To-Completion) 기반 비주기 태스크 구현 기법에서는 비주기 태스크가 이벤트 핸들러로 구성된다. TinyOS에서는 이벤트 핸들러가 항상 태스크를 선점하기 때문에 (그림 3)에서 제시한 기법을 사용할 경우 비주기 태스크의 응답시간을 최소화 할 수 있다. 그러나 이벤트 핸들러로 구성된 비주기 태스크가 주기 태스크를 항상 선점하기 때문에 주기 태스크의 마감 시한을 보장할 수 없다. (그림 4)의 SPO(Split-Phase-Operation) 기반 비주기 태스크 구현 기법은 비주기 태스크를 TinyOS의 태스크로 구현하여 이벤트 핸들러에 의해 비주기 태스크가 생성 및 실행된다. 생성된 비주기 태스크는 다른 태스크와 함께 FIFO 기반의 스케줄링 후 실행된다. (그림 4)에서 제시한 기법은 (그림 3)에서 제시한 RTC 기반의 비주기 태스크 구현 기법보다 비주기 태스크의 평균 응답시간은 증가하지만, 비주기 태스크가 주기 태스크의 실행을 선점할 수 없기 때문에 상대적으로 (그림 3)에서 제시한 기법 보다 주기 태스크의 마감시한 초과율이 작을 수 있다. 한편 기존의 TinyOS에서 FIFO 큐 대신에 우선순위 큐(Priority Queue)를 구현하여 우선순위 기반의 비선점형 태스크 스케줄링 기법이 있다[7]. [7]에서 제시한 기법에서는 모든 주기 태스크와 비주기 태스크에게 우선 순위를 부여한 후 비선점형 우선 순위 기반의 스케줄링이 가능하다. 따라서 비주기 태스크의 우선 순위에 따라 주기 태스크의 마감시한 초과율과 비주기 태스크의 평균 응답시간이 결정된다. <표 1>은 앞서 설명한 비주기 태스크의 구현 방법을 보여주고 있다. 그러나 앞서 기술한 바와 같이 비선점형 스케줄링 기반의 TinyOS에서는 주기 태스크의 마감시한 보장과 동시에 비주기 태스크의 빠른 평균 시간 응답을 보장하는 것은 불가능하다. 앞서 살펴본 바와 같이 주기 태스크의 마감시한 보장과 비주기 태스크의 빠른 평균 응답 시간을 제공하기 위해서는 멀티태스킹 기반의 선점형 스케줄러가 지원되는 센서 노드 플랫폼을 사용해야 한다. AvrX와 MANTIS는 비선점형 스케줄러 기반의 TinyOS에서 실행되는 태스크에게 선점형 실행을 제공하기 위하여 (그림 5)와 같이 TinyOS 자체

〈표 1〉 TinyOS에서 비주기 태스크 구현 기법의 비교

구 분	RTC기반 비주기 태스크 구현기법	SPO기반 비주기 태스크 구현 기법	우선 순위 기반 비주기 태스크 구현 기법
비주기 태스크 스케줄링 정책	필요 없음	FIFO	우선 순위 기반 스케줄링
우선 순위	비주기 태스크 >> 주기 태스크	비주기 태스크 = 주기 태스크	비주기 태스크 << 주기 태스크*
주기 태스크 마감시한 초과율	가장 큼	보통	가장 작음*
비주기 태스크의 응답 시간	가장 짧음	보통	가장 김*
TinyOS의 개선 여부	필요 없음	필요 없음	필요함

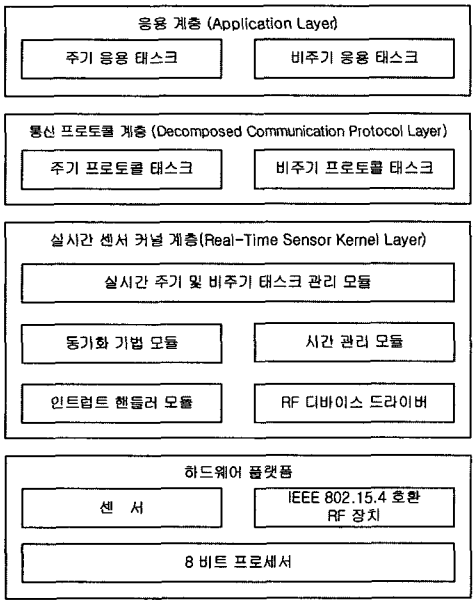
* 비주기 태스크의 우선순위에 따라 결정될 수 있음.

를 하나의 태스크로 취급하여 실행시키는 계층적인 태스크 스케줄링 방식을 제안하였다. AvrX와 MANTIS가 제공하는 태스크 생성 API(Application Programming Interface)에 의해 TinyOS는 태스크로 생성되어 실행 준비 큐에 등록된 후에 선점형 스케줄러인 계층 1 스케줄러에 의해 실행된다. 그리고 실시간성이 요구되지 않는 태스크는 TinyOS의 태스크 생성 API에 의해 생성되어 TinyOS의 실행준비 큐에 등록된 후에 TinyOS의 FIFO 기반 비선점형 스케줄러인 계층 2 스케줄러에 의해 실행된다. 이러한 스케줄링 방식에서는 멀티태스킹기반의 선점형 스케줄링이 가능하지만 두 운영체제의 동시 실행을 위해서는 많은 하드웨어 자원이 필요하기 때문에 하드웨어 자원이 빈약한 센서 노드에서는 과도한 메모리 사용량이 요구되어 비효율적이다. 또한 주기 태스크와 비주기 태스크가 동시에 실행되는 환경을 고려하지 않았다. DCOS는 EDF(Earliest Deadline First) 기반의 선점형 스케줄러를 사용하여 주기 태스크의 실시간성만을 제공한다.

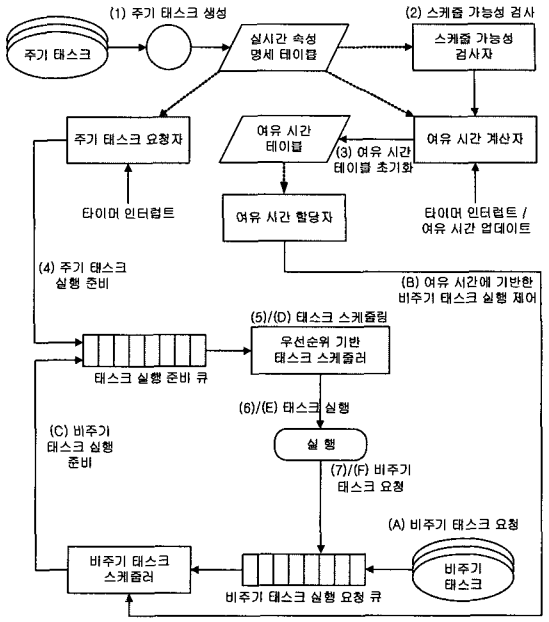
3. 실시간 센서 노드 플랫폼

3.1 실시간 센서 노드 플랫폼 구조

(그림 6)은 실시간 센서 노드 플랫폼의 구조이다. 실시간 센서 노드 플랫폼은 하드웨어 플랫폼과 소프트웨어 플랫폼으로 구분되며, 하드웨어 플랫폼은 센서 노드를 구성하는 하드웨어의 집합으로 8비트 초경량 마이크로프로세서와 센서 및 통신 장치로 구성된다. 소프트웨어 플랫폼은 실시간 센서 커널 계층(Real-Time Sensor Kernel Layer), 통신 프로토콜 계층(Decomposed Communication Protocol Layer) 그리고 센서 노드의 배포 목적을 수행하기 위한 응용 계층(Application Layer)으로 구성된다. 실시간 센서 커널 계층의 인터럽트 핸들러 모듈과 RF 디바이스 드라이버는 센서 노드의 하드웨어를 제어하고, 시간 관리 모듈은 상위 응용 계층에 시간 관련 함수 서비스를 제공한다. 실시간 주기 및 비주기 태스크 관리 모듈은 상위 계층을 구성하는 주기 태스크 및 비주기 태스크의 실시간성을 보장한다. 또한 실시간 주기 및 비주기 태스크 관리 모듈은 주기 및 비주기 태스크의 생성 인터페이스를 제공하여 주기 태스크의 실시간 속성을 명세하고, 실시간 태스크 스케줄링 가능성 여부를 판단한 후 주기 및 비주기 태스크의 실시간 스케줄링을 한다. 그리고 동기화 기법 모듈은 태스크간의 동기화를 제공한다. 통신 프로토콜 계층은 실시간 센서 커널 계층을 기반으로 구현되며, 각 통신 프로토콜의 속성에 따라 주기 및 비주기 태스크들의 집합으로 구현된다. 마지막으로 응용 계층은 센서 노드의 사용 목적에 따라 여러 개의 주기 응용 태스크와 비주기 응용 태스크로 구성된다.



(그림 6) 실시간 센서 노드 플랫폼 구조



(그림 7) 실시간 주기 및 비주기 태스크 관리 모듈 구조

3.2 실시간 주기 및 비주기 태스크 관리 모듈의 설계

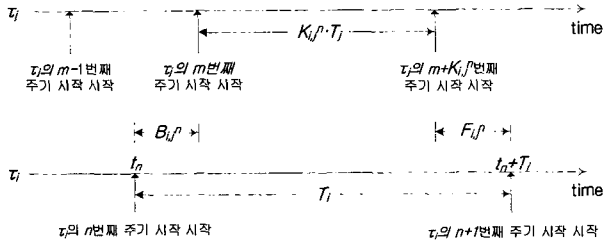
실시간 센서 커널 계층의 핵심 모듈인 실시간 주기 및 비주기 태스크 관리 기법을 기술하는데 필요한 수식 기호는 다음과 같이 정의한다. N 은 모든 주기 태스크의 수이다. τ_i는 i번째 우선순위를 가지는 주기 태스크이며, i가 낮을수록 높은 우선순위를 가진다. T_i, R_i, C_i 그리고 D_i는 τ_i의 주기, 요청시간, 실행시간, 그리고 마감시한을 나타내며, $\tau_i = (T_i, R_i, C_i, D_i)$로 표현된다. 모든 주기 태스크의 집합은 TA로 표현된다. 본 논문에서는 주기 태스크의 스케줄링을 위하여 RM (Rate Monotonic)기법을 사용한다[8]. 그리고 $S_i(t)$는 시간 t에서 실행하고자 하는 주기 태스크가 가질 수 있는 처리기 여유 시간을 나타내며, $S(t)$는 시간 t에서 계산되는 시스템 전체의 여유 시간이다. $S_i(t)$와 $S(t)$를 구하는 절차는 이 절에서 기술하고 있는 수식들 중에서 수식 (3)과

	4 bytes	4 bytes	4 bytes	4 bytes
구 분	주 기	요청 시간	실행 시간	마감 시한
태스크 1	T_1	R_1	C_1	D_1
태스크 2	T_2	R_2	C_2	D_2
...	...	...	...	...

(그림 8) 주기 태스크의 실시간 속성 명세 테이블

	4 bytes	4 bytes
구 분	남은 실행 시간	여유 시간
태스크 1	$RC_1(t)$	$S_1(t)$
태스크 2	$RC_2(t)$	$S_2(t)$
...	...	...

(그림 9) 여유 시간 테이블의 구조



(그림 10) 주기 태스크 τ_i 의 여유 시간

수식 (4)로부터 계산된다. (그림 7)은 실시간 주기 및 비주기 태스크 관리 모듈의 구성도를 보여준다. 이 모듈의 실행 절차를 살펴보면 다음과 같다. 먼저 생성되는 주기 태스크의 실시간 속성을 명시하기 위하여 주기, 요청시간, 실행시간 그리고 마감시한 값이 주기 태스크의 실시간 속성 명세 테이블에 저장된다(그림 7의 과정 1). 주기 태스크의 실시간 속성 명세 테이블의 구조는 (그림 8)과 같다. 스케줄 가능성 검사자는 실시간 속성 명세 테이블을 참조하여 주기 태스크의 스케줄 가능성을 검사한다(그림 7의 과정 2). 스케줄 가능성 검사는 주기 태스크의 실시간 스케줄링 기법에 따라 다양하게 구현 가능하다. 본 논문에서는 주기 태스크를 RM 기반으로 스케줄링하며, 여유 시간 계산자는 여유 시간 테이블을 초기화한다. 또한 여유 시간 계산자는 각 주기 태스크 τ_i에 대하여 (그림 9)와 같은 여유 시간 테이블을 만든다 (그림 7의 과정 3).

여유 시간 테이블 내의 남은 실행 시간 값과 여유 시간 값을 계산하는 과정을 살펴보면 다음과 같다. 주기 태스크 τ_i의 여유 시간 $S_i(t)$는 시간 t에서 τ_i의 실행 완료 시간이 D_i을 초과하지 않는 범위 내에서 최대 지연 가능한 시간을 의미한다. 주기 태스크는 우선순위에 따라 스케줄이 되기 때문에 주기태스크 τ_i의 여유시간 $S_i(t)$를 계산하기 위해서는 자신의 실행시간과 자신보다 우선순위가 높은 모든 태스크의 실행 시간을 고려해야 한다. (그림 10)은 주기태스크 τ_i의 여유 시간을 나타낸다. τ_j가 τ_i보다 우선순위가 크다고 가정하면 RM에 의하여 T_j가 T_i보다 작다. (그림 10)에서 τ_i의 n번째 주기 시작 시간이 t_n이라고 가정하면, n+1번째 주기 시작 시간은 $t_n + T_i$가 된다. τ_i의 n번째 주기 시작 시간인 t_n이후의 τ_j의 주기 시작 시간과의 차이를 $B^n_{i,j}$ 이라고 하고, 시간 $t_n + T_i$에서 실행 중인 τ_j의 주기 시작 시간과의 차이를 $F^n_{i,j}$ 라고 한다. t_n와 $t_n + T_i$사이에 τ_j가 실행된 횟수 $K^n_{i,j}$ 는 수식 (1)과 같이 구할 수 있다. 그리고 시간 t_n에서 실행 중인 τ_j가 마감시한 내에 실행을 완료하기 위하여 $B^n_{i,j}$ 동안 필요한 시간 $RC_j(t_n)$를 사용하여 계산된 $S_i(t_n)$는 수식 (2)와 같다.

$$K^n_{i,j} = \{T_i - B^n_{i,j} - F^n_{i,j}\} / T_j \quad (1)$$

$$S_i(t_n) = T_i - \sum_{j<i}^{N} RC_j(t_n) - \sum_{j\le i}^{N} K^n_{i,j} \cdot C_j - \sum_{j<i}^{N} \min(F^n_{i,j}, C_j) \quad (2)$$

여유 시간 동안 비주기 태스크가 실행 가능하기 때문에 시간 t-1에서 비주기 태스크가 실행되면 시간 t에서의 여유 시간은 1 감소하여, $S_i(t) = S_i(t-1)$이 된다. 반면 시간 t-1에서 주기 태스크가 실행되는 경우 시간 t에서의 여유 시간은 변하지 않기 때문에 $S_i(t) = S_i(t)$이 된다. 한편 τ_i의 여유시간 $S_i(t)$는 매주기 마다 수식 (3)과 같이 계산되며, 시스템의 여유 시간 $S(t)$는 모든 주기 태스크의 여유 시간 중 최소값이며 수식 (4)와 같이 계산된다.

$$S_i(t) = T_i - \sum_{j<i}^{N} RC_j(t) - \sum_{j\le i}^{N} K^n_{i,j} \cdot C_j - \sum_{j<i}^{N} \min(F^n_{i,j}, C_j) \quad (3)$$

$$S(t) = \min\{S_1(t), S_2(t), S_3(t), ..., S_{N-1}(t), S_N(t)\} \quad (4)$$

한편, 주기 태스크 요청자는 실시간 속성 명세 테이블의 주기 값에 기반하여 각 주기 태스크를 매주기 시작시간에 대기 상태에서 실행 준비 상태로 전환한다 (그림 7의 과정 4). 실행 준비 상태가 된 주기 태스크는 태스크 실행 준비 큐에 등록되며 태스크 스케줄러가 우선순위에 기반하여 태스크를 실행한다 (그림 7의 과정 5와 6). 요청된 비주기 태스크는 비주기 태스크 요청 큐에 등록된다(그림 7의 과정 7). 시스템의 여유 시간이 있을 경우 비주기 태스크 스케줄러에게 여유 시간을 할당한다 (그림 7의 과정 B). 여유 시간을 할당 받은 비주기 태스크 스케줄러는 FIFO 방식으로 비주기 태스크 요청 큐에 등록된 태스크를 선택하여 실행 준비 상태로 전환시킨다 (그림 7의 과정 C). 실행 준비 상태로 전환된 비주기 태스크는 태스크 스케줄러에 의하여 선택된 후 실행된다(그림 7의 과정 D와 E). 실행되는 비주기 태스크는 계속해서 다른 비주기 태스크의 실행을 요청할 수 있다(그림 7의 과정 F). 우선 순위 기반 태스크 스케줄러는 주

〈표 2〉 실시간 주기 및 비주기 태스크 관리 모듈의 상세 정보

구 성 요 소	소요 메모리(Byte)	구 성 요 소	명령어 수 (cycle)	소요 메모리(Byte)
실시간 속성 명세 테이블	16 × N	스케줄 가능성 검사자	511	320
여유 시간 테이블	8 × N	여유시간 계산자	458	532
비주기 태스크 요청 큐	12 × 8	여유시간 할당자	82	112
* N은 주기 태스크의 총 개수		비주기 태스크 스케줄러	435	170

기 태스크와 비주기 태스크가 동시에 실행 가능할 경우, 여유 시간이 존재하면 항상 비주기 태스크를 최우선 순위로 선택하여 비주기 태스크의 응답시간을 줄이고자 한다. 만약 여유 시간이 부족하여 비주기 태스크의 실행이 완료되지 않은 경우에는 비주기 태스크를 다시 대기 상태로 전환하여 실행 준비중인 주기 태스크를 실행시켜 주기 태스크의 실시간성을 보장할 수 있도록 한다. 이 절에서 설명한 각 모듈을 C언어로 구현하여 계산된 결과 값을 정리하면 <표 2>와 같다.

3.3 비주기 태스크의 응답시간 분석

비주기 태스크 스케줄링 기법에는 인터럽트 기반의 실행 기법, 폴링 서버(Polling server) 기법, 백그라운드 서버(Background server) 기법, 그리고 슬랙 스틸링(Slack Stealing) 기법 등이 있다[9]. 인터럽트 기반 실행 기법은 비주기 태스크의 실행을 인터럽트 핸들러로 구현하는 방법이며 응답시간이 가장 짧다. 그러나 인터럽트 기반 실행 기법에서 비주기 태스크는 주기 태스크를 항상 선점하기 때문에 주기 태스크가 마감시한을 초과하는 경우가 발생할 수 있다. 백그라운드 서버 기법은 실행 준비 상태의 주기 태스크가 없을 때에만 비주기 태스크를 처리하여 주기 태스크의 실시간성을 보장하고자 한다. 따라서 백그라운드 서버 기법에서는 비주기 태스크의 응답시간이 가장 크다. 폴링 서버 기법인 경우 주기 태스크가 비주기 태스크의 실행 요구가 있는지를 주기적으로 폴링하여 주기 태스크가 대신에 수행을 하며, 그 주기 태스크의 우선순위에 따라 비주기 태스크의 응답시간이 결정된다. 마지막으로 슬랙 스틸링 기법은 주기 태스크가 마감시한 내에 실행이 완료되어야 하는 시점의 여유 시간 동안에 비주기 태스크를 먼저 실행시켜 비주기 태스크의 응답 시간을 개선하고자 한다. 본 논문에서는 비주기 태스크의 응답시간을 최소화하기 위하여 슬랙 스틸링 기법의 개념을 사용하였으며, 백그라운드 서버 기법 및 폴링 서버 기법과 성능을 비교하였다. 이 절에서는 폴링 서버 기법과 백그라운드 서버 기법을 모델링 하여 본 논문에서 구현한 기법과 비교하여 실험 결과의 정확성을 설명하는데 활용하였다. 백그라운드 서버 기법은 실행 준비 상태인 주기 태스크가 존재하지 않을 때 비주기 태스크를 처리하기 때문에 비주기 태스크는 모든 주기 태스크보다 우선순위가 낮은 태스크이다. 백그라운드 서버 기법에서 실행되는 비주기 태스크의 응답 시간을 계산하는데 사용되는 수식 기호는 다음과 같다. $\overline{T_{ap}}$는 비주기 태스크의 응답 시간이다. $\overline{W_{ap}}$는 비주기 태스크가 큐에서 기다린 시간이며 $\overline{C_{ap}}$는 비주기 태스크의 평균 실행시간이다. λ_i와 λ_{ap}는 각각 주기 태스크 τ_i와 비주기 태스크 τ_{ap}의 평균 도착율이며 $1/\mu_i$ 과 $1/\mu_{ap}$은 τ_i와 τ_{ap}의 평균 실행 시간이다. ρ_i와 ρ_{ap}는 τ_i와 τ_{ap}의 처리기 이용률이며 $\rho_i=\lambda_i/\mu_i$와 $\rho_{ap}=\lambda_{ap}/\mu_{ap}$로 표현된다. $\overline{T_{ap}}$는 수식 (5)와 같이 구할 수 있다. 수식 (6)에서의 $\overline{W_{ap}}$는 비주기 태스크가 요청되었을 때 큐에 이미 존재하는 주기 태스크와 비주기 태스크가 처리되는 시간 $\overline{W'_{ap}}$와 비주기 태스크가 요청된 후 준비 상태가 된 주기 태스크가 선점되어 큐에서 기다리는 시간 $\overline{W''_{ap}}$의 합으로 계산된다. 따라서 수식 (5)는 수식 (6)과 같이 전개된다.

$$\overline{T_{ap}} = \overline{W_{ap}} + \overline{C_{ap}} \tag{5}$$

$$\overline{T_{ap}} = \overline{W'_{ap}} + \overline{W''_{ap}} + \overline{C_{ap}} \tag{6}$$

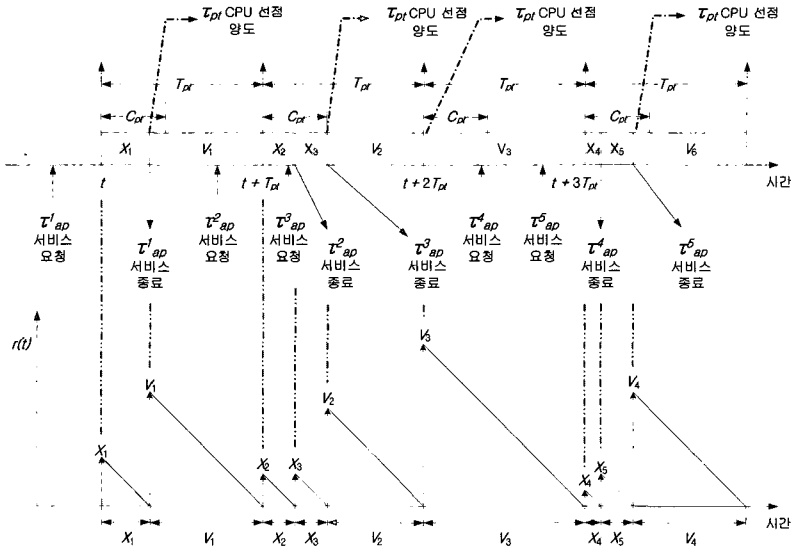
$\overline{W'_{ap}}$는 비주기 태스크가 생성되었을 때 시스템에 있는 모든 주기 태스크가 실행될 때까지 소요되는 시간이며 수식 (7)과 같이 계산된다. 그리고 $\overline{W''_{ap}}$는 $\overline{T_{ap}}$ 동안 도착하는 주기 태스크들의 처리 시간이며 수식 (8)과 같이 계산된다. 따라서 수식 (6)부터 수식 (8)에 의해서 $\overline{T_{ap}}$는 수식 (9)과 같이 계산된다.

$$\overline{W'_{ap}} = \frac{\sum_{i=1}^{N}\lambda_i\overline{C_i^2} + \lambda_{ap}\overline{C_{ap}^2}}{2(1-\rho_1-\rho_2-\rho_N-\rho_{ap})} = \frac{\sum_{i=1}^{N}\frac{\overline{C_i^2}}{T_i} + \lambda_{ap}\overline{C_{ap}^2}}{2(1-\sum_{i=1}^{N}\rho_i-\rho_{ap})} \tag{7}$$

$$\overline{W''_{ap}} = \sum_{i=1}^{N}\frac{1}{\mu_i}\lambda_i\overline{T_{ap}} = \sum_{i=1}^{N}C_i\frac{1}{T_i}\overline{T_{ap}} = \sum_{i=1}^{N}\rho_i\overline{T_{ap}} \tag{8}$$

$$\overline{T_{ap}} = \frac{2\overline{C_{ap}}(1-\sum_{i=1}^{N}\rho_i-\rho_{ap}) + \sum_{i=1}^{N}\frac{\overline{C_i^2}}{T_i} + \lambda_{ap}\overline{C_{ap}^2}}{2(1-\sum_{i=1}^{N}\rho_i)(1-\sum_{i=1}^{N}\rho_i-\rho_{ap})} \tag{9}$$

폴링 서버 기법은 비주기 태스크의 처리를 담당하는 폴링 서버가 주기적으로 실행 준비중인 비주기 태스크가 존재하는지를 폴링하여 비주기 태스크를 주기적으로 처리한다. 따라서 모든 비주기 태스크는 폴링 태스크와 같은 우선순위를



(그림 11) 폴링 서버 기반의 비주기 태스크 처리 예

부여 받게 된다. 비주기 태스크의 응답시간을 최소화하기 위해서는 폴링 서버의 우선순위를 가장 높게 설정해야 한다. 폴링 서버의 응답 시간을 계산하는데 필요한 수식 기호는 다음과 같다. τ_{pt}는 폴링 서버이며, 폴링 서버의 주기와 실행시간은 T_{pt}와 C_{pt}로 표현된다. τ^i_{pt}는 i번째에 생성된 비주기 태스크이며, X_i는 τ^i_{pt} 의 실행 시간을 나타낸다. 그리고 V_i는 폴링 태스크 τ_{pt}가 각 주기 마다 대기 상태로 있는 시간이다. 나머지 수식 기호는 백그라운드 서버 기법에서 설명된 내용과 동일하다. (그림 11)은 폴링 서버 기반의 비주기 태스크를 처리하는 예이다. (그림 11)에서 비주기 태스크 τ^1_{pt}가 요청된 시점에서는 폴링 서버가 실행 준비 상태가 아니므로 τ^1_{pt}가 즉시 처리될 수 없다. 시간 t에서 폴링 서버의 새로운 주기가 시작되면, 비주기 태스크 τ^1_{pt}을 처리하고 다시 실행 대기 상태가 된다. τ^3_{pt}가 요청된 시간에는 폴링 서버가 실행 준비 상태이지만 τ^2_{pt}의 처리가 아직 끝나지 않았기 때문에 실행 대기 상태가 된다. 한편, 비주기 태스크가 생성되더라도 현재 실행 중인 비주기 태스크가 처리될 때까지 소요되는 시간 또는 폴링 서버가 실행 준비 상태로 전환될 때까지 기다려야 하는 시간을 R로 표기한다. 그리고 τ^5_{pt}와 같이 폴링 서버가 실행 준비 상태로 전환될 때까지 기다리는 시간뿐 아니라 먼저 이미 실행 준비 큐에서 대기 중인 비주기 태스크의 실행시간도 고려해야 한다. 비주기 태스크 실행 요청 큐에 있는 평균 비주기 태스크의 수는 N_Q로 나타내며 비주기 태스크의 평균 실행 시간을 $\overline{C_{ap}}$로 나타낸다. 한 비주기 태스크가 도착하여 비주기 태스크 실행 요청 큐에서 평균 대기하는 시간은 $N_Q \times \overline{C_{ap}}$가 된다. 폴링 서버 기법에서 비주기 태스크의 응답 시간은 수식 (10)과 같이 계산된다. 이 때 비주기 태스크가 큐에서 기다리는 시간 W_{ap}는 R과 $N_Q \times \overline{C_{ap}}$ 합으로 계산된다. *Little's Law* 에 의해 수식 (11)이 성립한다.

$$\overline{T_{ap}} = \overline{W_{ap}} + \overline{C_{ap}} \tag{10}$$

$$\overline{W_{ap}} = \overline{R} + N_Q \frac{1}{\mu_{ap}} = \overline{R} + N_Q \overline{C_{ap}},\ N_Q = \lambda_{ap} W_{ap} = \lambda_{ap} \frac{\overline{R}}{1-\rho_{ap}} \tag{11}$$

(그림 11)의 $r(t)$는 R의 분포 함수이며, 수식 (12)와 같이 계산된다. 여기서 $M(t)$는 시각 t까지 완료된 비주기 태스크의 수이고 $L(t)$는 시각 t까지 폴링 서버의 주기가 반복된 횟수이다. 수식 (12)의 양변에 극한을 취하면, $\overline{R}$을 수식 (13)과 같이 계산된다. 따라서 수식 (11)과 수식 (13)에 의해 $\overline{W_{ap}}$와 $\overline{T_{ap}}$를 수식 (14)와 같이 계산된다.

$$R = \frac{1}{t}\int_0^t r(\upsilon)d\upsilon = \frac{M(t)}{t}\frac{\sum_{i=1}^{M(t)} \frac{1}{2} X_i^2}{M(t)} + \frac{L(t)}{t}\frac{\sum_{i=1}^{L(t)} \frac{1}{2} V_i^2}{L(t)} \tag{12}$$

$$\lim_{t\to\infty} R = \overline{R} = \lambda_{ap}\frac{\sum_{i=1}^{M(t)} \frac{1}{2} X_i^2}{M(t)} + \frac{(1-\rho_{pt})}{\overline{V}}\frac{\sum_{i=1}^{L(t)} \frac{1}{2} V_i^2}{L(t)} = \frac{\lambda_{ap}\overline{X^2}}{2} + \frac{(1-\rho_{pt})\overline{V^2}}{2\overline{V}},$$
$$\text{where } \lim_{t\to\infty}\frac{M(t)}{t} = \lambda_{ap}, \lim_{t\to\infty}\frac{L(t)}{t} = \frac{(1-\rho_{pt})}{\overline{V}} \tag{13}$$

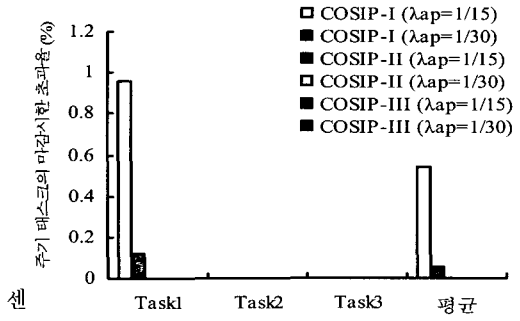
$$\overline{T_{ap}} = \frac{\lambda_{ap}\overline{X^2}}{2(1-\rho_{ap})} + \left(\frac{1-\rho_{pt}}{1-\rho_{ap}}\right)\frac{\overline{V^2}}{2\overline{V}} + \overline{C_{ap}},\ \text{where } \overline{T_{ap}} = \overline{W_{ap}} + \overline{C_{ap}}$$
$$\text{and } \overline{W_{ap}} = \frac{\lambda_{ap}\overline{X^2}}{2(1-\rho_{ap})} + \left(\frac{1-\rho_{pt}}{1-\rho_{ap}}\right)\frac{\overline{V^2}}{2\overline{V}} \tag{14}$$

4. 성능 분석 및 비교 평가

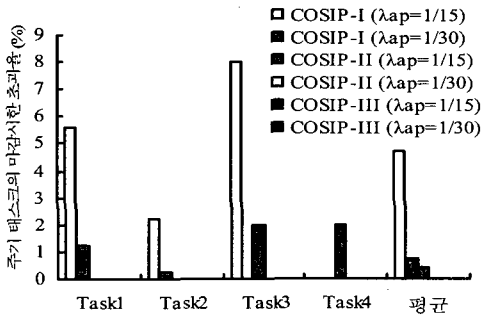
본 논문에서 제안한 실시간 센서 노드 플랫폼의 성능을 측정하기 위하여 다음과 같은 실험 환경을 사용하였다.

〈표 3〉 성능 평가를 위한 태스크 집합

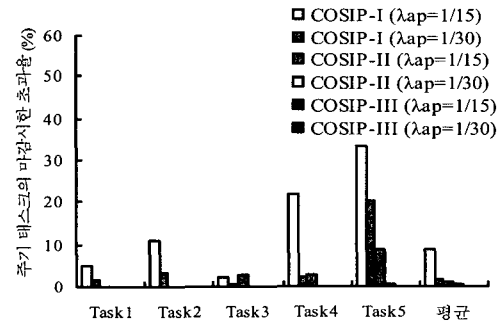
주기 태스크		비주기 태스크 도착율
처리기 이용률	태스크 집합 정의	
ρ = 0.2	*TA*1 = { *Task*1:(0,1,10,10), *Task*2: (0,1,20,20), *Task*3: (0,2,40,40) }	1/15 또는 1/30
ρ = 0.5	*TA*2 = { *Task*1: (0,1,5,5), *Task*2: (0,1,10,10), *Task*3: (0,2,20,20), *Task*4: (0,4,40,40) }	1/15 또는 1/30
ρ = 0.8	*TA*3 = { *Task*1: (0,1,5,5), *Task*2: (0,3,10,10), *Task*3: (0,2,20,20), *Task*4: (0,4,40,40), *Task*5: (0,5,50,50) }	1/15 또는 1/30



(그림 12) *TA*1에서 주기 태스크의 마감시한 초과율



(그림 13) *TA*2에서 주기 태스크의 마감시한 초과율



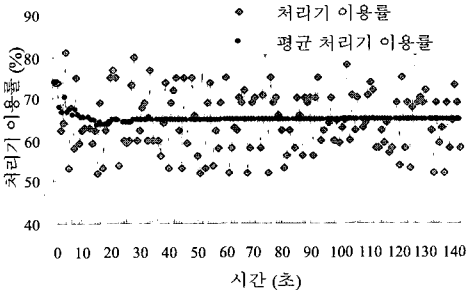
(그림 14) *TA*3에서 주기 태스크의 마감시한 초과율

서 노드의 하드웨어 플랫폼은 Chipcon사의 CC2420DB를 사용하였다. 실험에서 사용한 CC2420DB는 Atmel사의 8비트 마이크로프로세서인 Atmega128L과 IEEE 802.15.4 표준을 지원하는 RF 장치, 그리고 1개의 내부 센서(온도 센서)와 2개의 외부 센서(조도 센서와 습도 센서)로 구성된다. 또한 4KB 내부 메모리와 16KB의 외부 메모리를 가진다. 본 논문에서는 실시간 센서 노드 플랫폼의 성능을 평가하기 위하여 uC/OS-II와 uIP 버전 0.9를 사용하였다 [10-11]. 본 실험에서 사용한 TCP/IP는 센서 네트워크에 적용하기에는 매우 복잡한 구조를 가지고 있지만 제안한 실시간 센서 노드 플랫폼이 TCP/IP 프로토콜 기반에서 우수한 성능을 보여주면, 차후 TCP/IP 프로토콜보다 경량화되고 표준화 될 센서 네트워크 프로토콜에서도 우수한 성능을 보여줄 것으로 확신한다. 본 논문에서 수행할 센서 노드 플랫폼은 3 개의 모델로 구현되었다. 먼저, COSIP-I 모델에서는 uC/OS-II 버전 2.52와 uIP 버전 0.9를 수정없이 사용하여 센서 노드 플랫폼을 구축하였다. 그리고 COSIP-II 모델에서는 기존의 uC/OS-II기반에서 uIP내의 TCP/IP 프로토콜을 태스크 단위로 구현하여 센서 노드 플랫폼을 구축하였다. TCP/IP 프로토콜을 태스크 단위로 구현된 상세 정보는 <표 4>에서 설명한다. 마지막으로 COSIP-III 모델에서는 uC/OS-II에 3장에서 설명한 주기 및 비주기 태스크 관리 모듈을 구현하였으며, TCP/IP 프로토콜을 태스크 단위로 구현하여 통신 모듈을 탑재한 센서 노드 플랫폼이다. COSIP-III 모델과 다르게 COSIP-I 모델과 COSIP-II 모델은 주기 및 비주기 태스크 관리를 하기 위하여 사용자가 직접 구현해야 한다. COSIP-I 모델과 COSIP-II 모델에서는 블록 기반의 주기 태스크 구현 기법을 사용하여 주기 태스크를 관리하였다. COSIP-I 모델은 RTC 기반 비주기 태스크 구현 기법을 사용하여 비주기 태스크를 관리하였다. COSIP-II 모델은 우선순위 기반 비주기 태스크 구현 기법을 사용하여 비주기 태스크를 관리하였다. 제안된 COSIP-I/II/III 모델의 성능을 평가하는데 사용한 태스크 집합의 속성은 <표 3>과 같다. 성능 평가를 위하여 사용한 태스크 집합은 처리기 이용률이 각각 0.2, 0.5, 0.8인 주기 태스크 집합과 1/15, 1/30의 평균 도착율을 가지는 비주기 태스크를 정의하였다. 그리고 COSIP-II 모델에서 비주기 태스크의 우선순위는 *Task*2와 *Task*3 사이로 지정하였다.

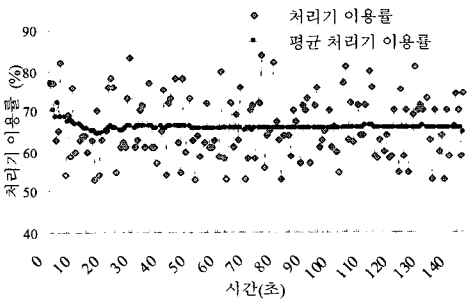
(그림 12)부터 (그림 14)는 COSIP-I 모델과 COSIP-II 모델 및 COSIP-III 모델에서 주기 태스크의 마감시한이 초과한 비율을 보여준다. 주기 태스크의 처리기 이용률이 0.2인 *TA*1이고 비주기 태스크의 도착율이 1/15일 때 COSIP-I 모델에서 주기 태스크의 평균 마감시한의 초과 비율은 0.5%로 나타났다. 그리고 비주기 태스크의 도착율이 1/15이고 *TA*2일 때, COSIP-I 모델과 COSIP-II 모델에서 주기 태스크의 평균 마감시한의 초과 비율은 4.6%와 0.4%로 증가 하였으며, *TA*3에서는 8.7%와 1.0%로 증가하였다. 이와 같은 결과가 나온 이유는 처리기 이용률이 작을수록 주기 태스크의 마감시한 초과 비율이 작게 나타나며, 비주기 태스크의 평균 도착율이 클수록 주기 태스크의 마감시한 초과 비율이 높게 나타난다. 다시 말하면, 비주기 태스크의 도착율이 높을수록 주기 태스크의 실행 시간이 상대적으로 지연되어 주

〈표 4〉 성능 평가를 위한 태스크들의 상세 정보

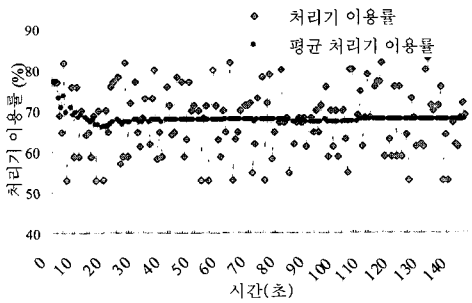
태스크 이름		우선 순위	주기 (ms)	실행시간 (ms)		
				COSIP-I	COSIP-II	COSIP-III
응용 태스크	온도 측정 태스크	16	200	40	40	40
	조도 측정 태스크	17	300	20	20	20
	습도 측정 태스크	18	400	20	20	20
	응용 비주기 태스크	8	비주기	200	200	200
네트워크 태스크	ARP 태스크	9	비주기	2	2	2
	IP 태스크	10	비주기	2.7	2.7	2.8
	TCP-OUT 태스크	11	비주기	3.1	3.3	3.3
	TCP-IN 태스크	12	비주기	3.0	3.0	3.0
	UDP-OUT 태스크	13	비주기	3.0	3.0	3.0
	UDP-IN 태스크	14	비주기	3.0	3.0	3.0
	ARP-UPDATE 태스크	19	500	2	2	2



(그림 15) COSIP-I 모델



(그림 16) COSIP-II 모델

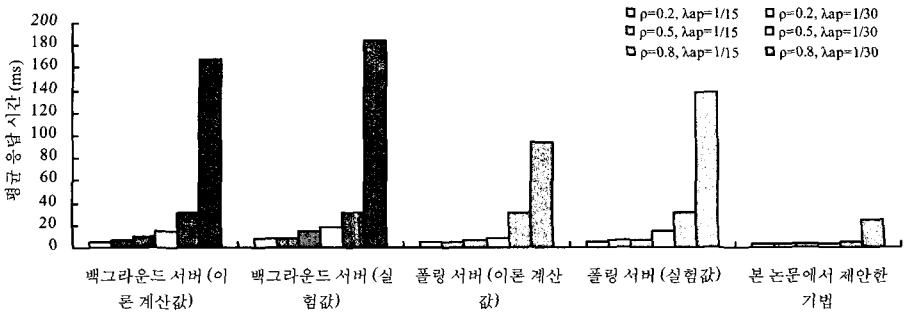


(그림 17) COSIP-III 모델

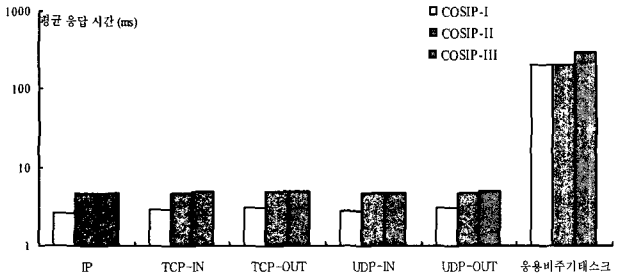
기 태스크의 마감시한을 초과할 경우가 많기 때문이다. 반면, COSIP-III 모델에서는 주기 태스크의 처리기 이용률과 비주기 태스크의 평균 도착율이 증가하더라도 모든 주기 태스크의 마감시한이 초과하지 않음을 확인하였다.

(그림 15)부터 (그림 17)은 COSIP-I 모델과 COSIP-II 모델 및 COSIP-III 모델에서의 처리기 이용률을 비교한 결과이다. 실험 평가를 위하여 사용한 태스크 집합은 *TA2*이고 비주기 태스크의 도착율은 1/30이다. 실험 결과, COSIP-I 모델의 평균 처리기 이용률은 64.66%, COSIP-II 모델에서는 65.71%, COSIP-III 모델에서는 67.46%로 나타났다. COSIP-III 모델은 COSIP-I 모델과 COSIP-II 모델에 비하여 여유 시간 계산을 위한 오버헤드가 있기 때문이다. 그러나 COSIP-III 모델의 처리기 이용률은 COSIP-I 모델과 COSIP-II 모델에 비하여 약 3%만 증가하였으며, 이는 주기 및 비주기 태스크 관리 모듈의 오버헤드가 적음을 보여준다. (그림 18)은 <표 3>에서 제시한 태스크 집합을 적용한 COSIP-III 모델에서 백그라운드 서버 기법과 폴링 서버 기법, 그리고 본 논문에서 제안한 비주기 태스크 스케줄링 기법을 사용하여 나온 비주기 태스크의 평균 응답시간을 비교한 결과이다. 폴링 서버 기법을 사용한 실험에서는 폴링 서버를 담당하는 태스크의 우선순위를 가장 높게 설정하여 비주기 태스크의 응답시간을 최소화하도록 하였다. (그림 18)에서 보는 바와 같이 세 조건 모두에서 본 논문에서 제안한 기법이 폴링 서버 기법과 백그라운드 서버 기법보다 비주기 태스크의 평균 응답 시간이 낮게 나타났다. 또한, 주기 태스크가 차지하는 처리기 이용률이 증가할수록 비주기 태스크의 응답 시간이 증가하는 것을 알 수 있다. 이는 주기 태스크의 마감시한을 보장하기 위하여 비주기 태스크의 처리를 지연시키기 때문이다.

TCP/IP 프로토콜을 태스크 단위로 구현된 통신 모듈을 탑재한 COSIP-II 모델과 COSIP-IIII 모델의 성능평가를 하기 위한 실험 환경은 다음과 같다. 구현된 통신 모듈은 ARP 태스크, ARP-UPDATE 태스크, IP 태스크, UDP-IN 태스크, UDP-OUT 태스크, TCP-IN 태스크 그리고 TCP-OUT 태스크로 구성된다. 주요 시스템 서비스를 태스크 단위로 구현할 경우 실시간 센서 노드 플랫폼을 쉽게 구현할 수 있다. 이는 주요 시스템 서비스를 태스크 단위로 제어 가능하기 때문에 실시간 서비스를 요구하는 태스크가 마감 시한 내에 실행이 완료되는 것을 실시간 주기 및 비주



(그림 18) 비주기 태스크의 평균 응답 시간



(그림 19) COSIP-I/II/III 모델에서 비주기 태스크의 평균 응답 시간

기 관리 모듈에 의해서 제어가 가능하다. 또한 태스크 단위로 구현된 통신 모듈에서는 새로운 통신 프로토콜의 추가 모듈을 독립적인 태스크 형태로 구현 가능하기 때문에 서비스의 확장성을 용이하게 제공할 수 있다. 성능 평가를 위하여 사용된 태스크들의 속성은 <표 4>와 같다. 응용 태스크는 센싱을 담당하는 3개의 주기 태스크 (온도, 조도, 습도 측정 태스크)와 1개의 비주기 태스크로 구성된다. 온도 측정 테스크는 200ms 마다 주기적으로 온도를 측정하며, 측정 결과가 설정된 임계 온도를 초과할 경우 LED를 깜빡거린다. 조도 측정 테스크는 300ms 마다 주기적으로 현 조도를 감지한 후 TCP를 사용하여 원격지에 있는 싱크 노드에게 센싱된 조도 값을 전송한다. 습도 측정 태스크는 400ms 마다 주기적으로 현재 습도를 감지한 후 UDP를 사용하여 싱크 노드에게 센싱된 습도 값을 전송한다. 센싱 데이터 프레임의 구조는 1바이트의 노드 ID과 4바이트의 데이터로 구성된다. <표 4>에서 제시된 각 태스크의 우선순위는 다음과 같은 규칙에 의해 할당되었다.

(1) 센서 노드 플랫폼에서 무선 센서 노드가 감지 및 제어하고자 하는 데이터는 일정한 종단간 지연 시간 내에 전달되어야 의미 있는 정보가 되기 때문에 데이터의 송수신을 담당하는 네트워크 태스크의 우선순위가 응용 태스크보다 높다.

(2) 응용 태스크간의 우선순위는 응용 서비스의 중요도에 따라 결정되지만, 본 논문에서는 태스크의 주기가 짧을수록 중요하다고 판단하여 주기가 짧은 태스크에게 높은 우선순위를 배정하였다.

(3) 비주기 응용 태스크가 주기 태스크의 마감시한과 다른 비주기 태스크의 응답시간에 미치는 영향을 확인하기 위하여 다른 응용 태스크보다 높은 우선순위를 가진다.

(4) 본 논문에서 가정한 네트워크 태스크 간의 우선 순위 할당 규칙은 다음과 같다.

A ARP 캐쉬 테이블을 주기적으로 갱신하는 ARP-UPDATE 태스크의 기능은 TCP/IP 프로토콜에서 선택적인 기능이기 때문에 가장 낮은 우선순위를 할당하였다.

B 통신의 상위 계층 프로토콜의 실행이 완료되기 위해서는 하위 계층 프로토콜의 실행이 먼저 완료되어야 함으로 하위 계층 프로토콜의 기능을 담당하는 태스크는 상위 계층 프로토콜의 기능을 담당하는 태스크보다 높은 우선순위를 가진다.

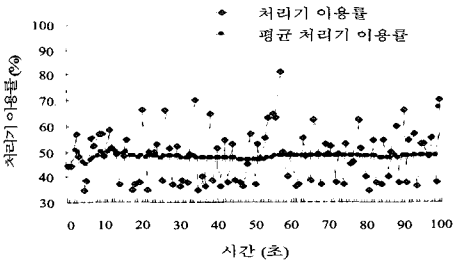
C 신뢰성을 제공하는 TCP 태스크는 UDP 태스크 보다 높은 우선 순위를 가진다.

D 센서 노드에서는 감지된 데이터의 전달이 빈번하게 실행되기 때문에 패킷의 송신 작업이 더 많이 발생한다. 이에 TCP-OUT 태스크와 UDP-OUT 태스크가 TCP-IN 태스크와 UDP-IN 태스크 보다 높은 우선순위를 가진다.

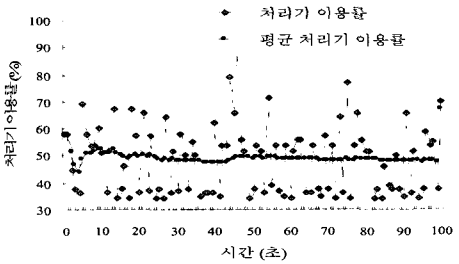
개별 IP가 할당된 2대의 CC2420DB 센서 노드를 사용하여 실험 환경을 구축하였으며, 동작 시험의 결과로는 주기 태스크의 마감 시한 초과 비율과 비주기 태스크의 평균 응답시간 및 평균 처리기 이용률을 측정하였다. <표 5>는 COSIP-I 모델과 COSIP-II 모델 및 COSIP-III 모델의 데이터와 명령어 코드 크기를 보여준다. COSIP-I 모델은 비주기 태스크의 처리를 인터럽트 핸들러 기반으로 수행하므로 부가적인 실시간 주기 및 비주기 태스크 관리 모듈이 필요 없다. 따라서 코드와 데이터의 크기가 가장 작다. COSIP-II 모델에서는 통신 모듈을 태스크 단위로 구현하였기 때문에 비주기 태스크를 위한 스택 공간의 할당이 필요하여 데이터 코드 크기가 증가하였으며, 태스크간의 동기화를 위한 코드가 필요하므로 명령어 코드 크기도 증가하였다. 그리고 COSIP-III 모델은 본 논문에서 제안한 실시간 주기 및 비주기 태스크 관리 모듈을 탑재하기 위하여 추가적인 코드와 데이터 메모리가 필요하기 때문에 COSIP-I 모델과 COSIP-II 모델에 비하여 코드의 크기가 증가하였다. 그러나 비주기 태스크를 위한 스택 공간을 모두 공유하기 때문에 COSIP-II에 비하여 데이터 코드 크기가 적다. Atmel사의

〈표 5〉 성능 평가를 위한 태스크 집합

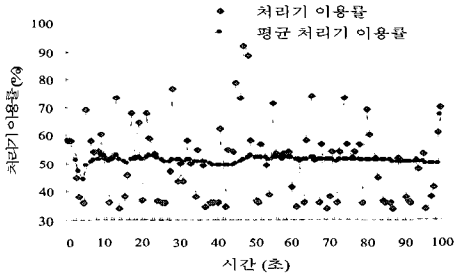
실험 모델	COSIP-I 모델	COSIP-II 모델	COSIP-III 모델
코드 크기 (Kbytes)	20.168	24.838	25.822
데이터 크기 (Kbytes)	3.243	4.363	3.953



(그림 20) COSIP-I 모델



(그림 21) COSIP-II 모델



(그림 22) COSIP-III 모델

Atmgea128L을 비롯한 대부분의 센서 노드용 마이크로프로세서는 128KB 명령어 메모리와 4KB 내부 데이터 메모리를 장착하고 있다. 따라서 COSIP-III은 초경량 8비트 마이크로프로세서를 탑재한 센서 노드 플랫폼에 적합하다.

(그림 19)는 COSIP-I 모델, COSIP-II 모델, 그리고 COSIP-III 모델에서 측정된 비주기 네트워크 태스크들의 평균 응답 시간을 보여준다. COSIP-I 모델은 비주기 태스크가 항상 주기 태스크를 선점하므로 비주기 태스크의 응답시간이 낮게 나타났다. COSIP-II 모델에서는 태스크간의 문맥 전환과 동기화로 인한 과부하가 발생하여 COSIP-I 모델에 비하여 다소 증가하였다. COSIP-III에서는 COSIP-I과 COSIP-II에 비하여 평균 비주기 태스크의 응답 시간이 증가하였는데, 이러한 이유는 COSIP-III 모델에서는 비주기 태스크의 실행시간이 주기 태스크의 여유 시간보다 클 경우 비주기 태스크의 실행을 지연시키고 주기 태스크가 마감시한 내에 실행 완료할 수 있도록 보장하기 때문이다. (그림 19)의 결과를 보여주는 실험 환경에서 발생된 온도, 조도, 습도 측정을 담당하는 주기 태스크의 마감시한 초과율은 다음과 같다. COSIP-I 모델에서는 2%, 23.3%, 15.4%였으며, COSIP-II 모델에서는 0.2%, 2.6%, 1.1%의 마감 시한 초과율이 발생되어 주기 태스크의 실시간성을 보장할 수 없었다. 그러나 COSIP-III 모델에서는 모든 주기 태스크가 마감시한 내에 실행 완료 가능함을 확인하였다.

(그림 20)부터 (그림 22)까지는 1초마다 측정된 태스크의 처리기 이용률과 평균 처리기 이용률의 변화량을 보여준다. (그림 20)에서의 시간이 55초가 되는 지점과 (그림 21)에서의 45초가 되는 지점, 그리고 (그림 22)에서 45초가 되는 지점에서 처리기 이용률이 순간적으로 증가되는 부분은 많은 실행 시간을 소모하는 응용 비주기 태스크가 실행되는 지점이다. 이 실험에서 COSIP-I 모델의 평균 처리기 이용율은 47.8%, COSIP-II 모델은 48.0% 마지막으로 COSIP-III 모델의 평균 처리기 이용률은 50.0%로 나타났다. (그림 20)부터 (그림 22)의 결과에서 보는 바와 같이 평균 처리기 이용률의 차이는 3%내외로 COSIP-III 모델은 COSIP-I 모델과 COSIP-II 모델과 비교하였을 때 처리기 이용률의 증가가 매우 미미하였다. 동작 시험을 수행한 결과, 본 논문이 제안한 실시간 센서 노드 플랫폼은 모든 주기 태스크의 마감시한을 보장하고 기존의 비주기 태스크 스케줄링 기법보다 향상된 평균 응답 시간을 보여주었다. 또한 실시간성을 보장하기 위해 추가된 실시간 주기 및 비주기 관리 모듈의 처리 과부하가 작아서 초경량 실시간 센서 노드 플랫폼에 탑재될 수 있음을 확인하였다.

5. 결 론

기존의 센서 노드 플랫폼은 실시간성을 지원하지 못하거나 주기 태스크들의 스케줄링만 지원한다. 이와 같은 플랫폼 환경에서는 주기 태스크의 마감시한 보장과 비주기 태스크의 빠른 응답시간이 요구되는 응용 서비스를 제공하기 위하여 개발자가 응용 계층에서 별도의 실시간 스케줄링 기법을 설계 및 구현해야 하는 문제점이 발생한다. 또한 기존의 센서 노드 플랫폼에서는 새로운 응용 및 네트워크 서비스의 추가와 삭제가 어렵기 때문에 빠른 개발 공정주기를 요구하는 센서 네트워크 산업에서 다양한 실시간 응용 서비스의 개발 및 관련 제품의 출시가 지연될 수 있다. 이에 본 논문에서는 멀티 태스크 기반의 확장성과 주기 및 비주기 태스크 관리 기법을 효율적으로 제공할 수 있는 실시간 센서 노드 플랫폼을 설계하고 구현하였다. 제안된 실시간 센서 노드 플랫폼을 상용 무선 센서 보드에서 동작 시험을 수행한 결과, 본 논문에서 제안한 실시간 태스크 관리 기법은 멀티

태스크 기반의 확장성과 주기 및 비주기 태스크의 실시간 관리 기법을 제공하기 위해 요구되는 부가적인 과부하가 있음에도 불구하고 주기 태스크의 마감시한 보장과 더불어 비주기 태스크의 빠른 응답시간과 효율적인 시스템의 평균 처리기 이용률을 확인하였다.

참 고 문 헌

[1] P. Levis, et al., "The Emergence of Networking Abstractions and Techniques in TinyOS," Proc. of the First USENIX/ACM Symposium on Networked Systems Design and Implementation, San Francisco, California, USA, pp. 1-14, March 2004.

[2] S. Bhatti, et al., "Mantis OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," ACM Kluwer Mobile Networks and Applications Journal, Special Issue on Wireless Sensor Networks, Volume 10, Issue 4, pp. 563-579, August, 2005.

[3] T.J. Hofmeijer, S.O. Dullman, P. G. Jansen and P. J. Havinga, "DCOS, A Real-time Light-weight Data Centric Operating System," Proc. of International Conference on Advances in Computer Science and Technology, St. Thomas, Missouri, USA, pp. 259-264, November 2004.

[4] P. Ganesan and A.G. Dean, "Enhancing the AvrX Kernel with Efficient Secure Communication using Software Thread Integration," Proc. of the 10th Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, pp. 265-275, May 2004.

[5] K. Jeffay and C.U. Martel, "On Non-preemptive Scheduling of Periodic and Sporadic Tasks," Proc. of the 12th IEEE Real-Time Systems Symposium, San Antonio, Texas, USA, pp. 129-139, December 1991.

[6] L. Georges and P. Muehlethaler, and N. Rivierre, "A Few Results on Non-preemptive Real-Time Scheduling," INRIA Research Report nRR3926, 2000.

[7] V. Subramonian, H. Huang-Ming, S. Datar and L. Chenyang, "Priority Scheduling in TinyOS - A Case Study," Washington University Technical Report (WUCSE-2003-74), Washington University, USA, December 2003.

[8] J. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," Proc. of the 10th IEEE Real Time Systems Symposium, Santa Monica, California, USA, pp. 166-171, December 1989.

[9] J. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-aperiodic Tasks in Fixed-priority Preemptive Systems," Proc. of the 13th IEEE Real-Time Systems Symposium, Phoenix, Arizona, USA, pp. 110-123, December 1992.

[10] J. J. Labrosse, *MicroC/OS-II The Real-Time Kernel*, 2nd Edition, CMP Books, 2002.

[11] A. Dunkels, "Full TCP/IP for 8bit Architectures," Proc. of the First ACM/Usenix International Conference on Mobile Systems, California, San Francisco, USA, pp. 85-98, May 2003.

김 병 훈

e-mail : bhkim81@gmail.com

2006년 2월 부산대학교 정보 컴퓨터 공학부 (학사)

2006년~현재 부산대학교 컴퓨터공학과 석사과정

관심분야 : 메쉬 네트워킹, TCP/IP 통신, 실시간 시스템, 큐잉 이론

정 경 훈

e-mail : jungkh@pknu.ac.kr

1996년 부경대학교 전자계산학과(학사)

1998년 부경대학교 전자계산학과(석사)

2006년 부경대학교 전자계산학과(박사)

2006년~현재 부산대학교 U-Port 정보기술 산학공동사업단 전임연구원

관심분야 : 센서네트워크, 임베디드 운영체제, 실시간 스케줄링 분산병렬처리

탁 성 우

e-mail : swtak@pusan.ac.kr

1995년 부산대학교 컴퓨터공학과(학사)

1997년 부산대학교 컴퓨터공학과(석사)

2003년 미국미주리주립대학교 Computer Science(박사)

2004년~현재 부산대학교 정보컴퓨터공학부 조교수

2004년~현재 부산대학교 컴퓨터 및 정보통신연구소 겸임 연구원

관심분야 : 유무선 네트워크, SoC 설계, 실시간 시스템, 위치인식, 최적화 기법, 그래프 이론, 큐잉 이론