로봇 컴포넌트의 QoS 보장을 위한 프레임워크의 설계 및 성능분석

Design and Performance Analysis of Framework for Guaranteeing QoS of Robot Components

임재석, 조문행, 정재엽, 이철훈
충남대학교 컴퓨터공학과

Jae-Seok Lim(jslim@cnu.ac.kr), Moon-Haeng Cho(root4567@cnu.ac.kr),
Jae-Yeop Jeong(iwis023@cnu.ac.kr), Cheol-Hoon Lee(clee@cnu.ac.kr)

요약

고성능 CPU의 개발과 고속 통신 기술의 발전은 네트워크 기반의 지능형 서비스 로봇 개발에 대한 연구가 활성화 되는 데에 많은 공헌을 하였다. 로봇 소프트웨어는 로봇의 올바른 수행과 사용자의 안전을 보장해야 하며, 이를 지원하기 위해 로봇 소프트웨어를 구성하는 컴포넌트의 QoS 보장에 대한 연구가 절실히 요구된다. 로봇 컴포넌트의 QoS는 컴포넌트가 원하는 데이터의 흐름을 가능하게 하여 안정적으로 수행함을 목적으로 하며, QoS를 보장함으로써 이러한 로봇의 지능화와 안정성을 얻을 수 있다. 본 논문에서는 제한된 자원을 가진 로봇 플랫폼 상에서 운용되는 로봇 컴포넌트들이 요청한 QoS를 최대한 보장하도록 하는 QoS 프레임워크를 설계하고 구현하였다. 또한 QoS의 요청에 대한 응답시간을 측정하고 그에 따른 성능분석 결과를 제시한다.

■ 중심어 : | 지능형 서비스 로봇 | 컴포넌트 | QoS Manager | 프레임워크 |

Abstract

The growth of CPU and communication technologies have made an important contribution to the development of the network-based intelligent service robots. Robot software must guarantee the correct execution and safety of the user. To achieve this, it is highly required to research how to guarantee the QoS of the components which organize a robot software. The QoS of robot components aims to execute the component stably by processing the data stream in a correct way. By guaranteeing QoS, we can achieve the intelligence and stability of robots. In this paper, we design and implement the QoS framework to guarantee the QoS of robot components on robot platforms with limited resources. We also measure the response times of QoS requests and present the performance analysis results about it.

■ keyword : | Intelligent Service Robots | Components | QoS Manager | Framework |

I. 서 론

로봇은 인간의 작업을 대신할 수 있다는 유용성 때문에 많은 연구가 진행되어 왔지만 핵심 요소 기술이 뒷받침 되지 않아서 그 실제적 확산이 매우 더뎠다. 최근 고성능 CPU의 개발과 고속 통신 기술의 발전으로 지능

* 본 연구는 지식경제부 및 정보통신연구진흥원의 IT성장동력기술개발사업의 일환으로 수행하였음. [2008-S-030-01, RUPI-클라이언트 기술 개발]
접수번호 : #081028-006
접수일자 : 2008년 10월 28일
심사완료일 : 2008년 12월 15일
교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

형 서비스 로봇 개발에 대한 기술적 낙관론이 힘을 얻으면서 로봇기술의 혁신을 통해 로봇 시스템의 보급과 실용화를 앞당기는 것을 목표로 그에 대한 연구가 급속도로 활발해졌다. 이는 로봇을 구성하는 소프트웨어가 지능화되고 안정적이며, 사용자의 안전을 보장해야 함을 그 전제로 하고 있다.

로봇을 구성하는 소프트웨어가 지능화되기 위해서 로봇 소프트웨어 시스템은 응용 프로그램을 동적으로 적재하거나 재구성할 수 있는 동적 재배치 기능, 엄밀한 시간제약에 맞도록 태스크를 처리하는 실시간성, 로봇의 비전과 음성 등 내부 장치간의 데이터 교환 및 처리에 대한 QoS 보장, 불완전한 상황에서도 신뢰성 있게 동작하는 결함 허용성 등을 지원해야 한다. 뿐만 아니라 로봇의 특성상 고도의 자원제약성과 하드웨어의 다양성을 극복할 수 있어야 한다[1][2]. 로봇 소프트웨어 개발자가 이처럼 다양한 로봇의 요구조건을 만족시키는 응용 프로그램을 개발하기 위해서는 여러 제약사항이 따른다. 특히 thin-client 개념의 로봇 플랫폼 상에서 운용되는 응용 프로그램은 자원제약으로 인해 QoS에 대한 고려가 절대적으로 필요하다.

QoS는 적합한 수준의 데이터 전송을 위해 충족시켜야 하는 서비스 요구 사항의 집합을 의미한다. 이러한 일반적인 QoS의 정의를 로봇이라는 특정 도메인에 적용하였으며, 본 논문에서의 로봇 컴포넌트 QoS는 컴포넌트가 원하는 데이터의 흐름을 가능하게 하도록 하여 안정적으로 컴포넌트를 수행하기 위한 것으로, CPU나 메모리 등 하드웨어의 자원을 비롯해 세마포어와 같은 소프트웨어 자원에 대한 컴포넌트 요구사항의 보장을 의미한다. 시스템이 가지는 자원과 로봇 응용 프로그램을 구성하는 컴포넌트들이 요구하는 QoS에 대한 적절한 조정과 중재는 보다 안정적이고 안전한 로봇의 수행을 보장해 줄 수 있다. 하지만 기존의 QoS를 지원하기 위한 대부분의 미들웨어는 분산 시스템에서 분산된 자원에 대한 QoS를 보장하는 것이며[3-5], 단대단(end-to-end) 네트워크 상에서의 QoS를 위한 연구가 대부분이다. 이들은 CORBA와 같은 통신 미들웨어와 실시간성을 제공하는 운영체제를 사용하며, 통신 미들웨어와 운영체제가 제공하는 서비스를 이용해 네트워크 QoS 및 실시간성을 보장한다.

본 논문에서는 기존의 네트워크 QoS를 보장하는 미들웨어의 구조를 이용해, 로봇 플랫폼 상에서 운용되는 컴포넌트의 QoS를 만족시키는 프레임워크를 구현한다. 또한 컴포넌트가 요청하는 QoS에 대해, 각각의 상황에 대한 응답시간을 측정하고 분석한다.

본 논문의 구성은 2장에서 로봇을 위한 여러 가지 미들웨어에 대해 간략히 소개하고, 기존의 분산 시스템에서 구현된 QoS 미들웨어에 대해 기술하며, 시스템 자원정보를 알아내기 위한 proc 파일 시스템에 대해 살펴본다. 3장에서 로봇 컴포넌트의 QoS 보장을 위한 프레임워크의 설계 및 구현에 대해 기술하며, 4장에서는 실험 환경 및 결과를, 마지막으로 5장에서는 결론 및 향후 연구과제에 대해 기술한다.

II. 관련 연구

본 장에서는 로봇 미들웨어인 RSCA, Miro, RTC, OROCOS 등에 대한 각각의 특징을 기술하고, 이러한 로봇 미들웨어들이 지원하지 않는 QoS 보장을 위한 분산 환경에서의 미들웨어인 DeSiDeRaTa 프로젝트를 소개한다.

1. Robot Middleware

1.1 RSCA(Robot Software Communication Architecture)

RSCA는 URC(Ubiquitous Robotics Companion) 로봇을 위한 표준 시스템 소프트웨어 구조로써 이기종 분산처리, 동적 시스템 재구성, QoS 및 실시간성 보장과 같은 URC 로봇의 응용 특성을 지원하는 것을 목적으로 개발되었다. RSCA는 JTRS(Joint Tactical Radio System)에서 제안한 SCA표준을 기반으로, SCA에서 불필요한 부분을 제거하고 URC 로봇을 위해 필요한 QoS와 이벤트 서비스 기능들을 추가한 것이다.

RSCA는 크게 코어 프레임워크 인터페이스(Core Framework Interface)와 도메인 프로파일(Domain Profile)로 나눌 수 있다. 코어 프레임워크 인터페이스

는 여러 개의 응용 소프트웨어 컴포넌트들로 구성된 로봇 소프트웨어를 각 프로세싱 노드에 배치하는 역할을 하는 기능을 외부에 제공하는 인터페이스이며, 도메인 프로파일은 로봇 시스템의 하드웨어와 소프트웨어의 구성 정보를 기술하기 위한 XML 디스크립터 파일들로 구성된다. 이는 설치된 하드웨어와 소프트웨어 컴포넌트의 정보를 알아내는데 사용될 뿐만 아니라, 새로운 소프트웨어를 설치할 때 컴포넌트들의 조합에 대한 설치 정보를 기술하는 부가 정보로서 사용될 수 있다[1].

1.2 Miro

Miro는 모바일 로봇 제어를 위한 CORBA기반의 분산 객체지향 프레임워크로, Miro의 코어 컴포넌트들은 ACE(Adaptive Communication Environment)를 사용하여 Linux 환경에서 C++로 개발되었다. ACE는 동시처리방식 통신 소프트웨어의 많은 핵심 패턴들을 구현한 오픈소스 기반의 객체지향 프레임워크로, 네트워크 프로그래밍의 단점을 해결하기 위해 만들어졌다. ACE를 바탕으로 만들어진 TAO(The ACE ORB)는 Miro 프레임워크의 핵심이라고 할 수 있으며, 이로 인해 TAO가 설치된 플랫폼에 쉽게 Miro를 이식할 수 있다[2][3].

1.3 RTC(Robot Technology Component)

RTC는 로봇용 소프트웨어 컴포넌트들이 서로 조합되고 연결되기 위한 컴포넌트 간의 공동 인터페이스 기준 규격을 OMG가 표준화 한 것이다.

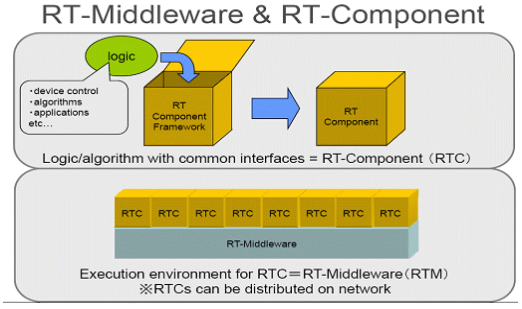

그림 1. RT-Middleware와 RTC의 구조

[그림 1]에서 보는 바와 같이 RTC 프레임워크를 통해 여러 종류의 컴포넌트가 표준형으로 포장되면, RT-Middleware가 이를 수용하여 컴포넌트를 조립해 사용한다. 로봇의 많은 소프트웨어 모듈들이 임베디드 소프트웨어이므로 경량화 한 컴포넌트 모델로 구현한 것이 RTC의 장점이다[4].

1.4 OROCOS(Open RObot COntrol Software)

OROCOS는 2001년 9월부터 EURON(European Robotics Research Network)의 지원으로 4개국의 대학에서 연구가 시작되었으며, 로봇과 기계 제어를 위한 모듈화 된 프레임워크를 제공한다. OROCOS는 로봇 애플리케이션 개발을 위한 C++ API인 RTT(Real Time Toolkit), 로봇 컴포넌트 및 애플리케이션 개발자들이 사용할 수 있는 이미 개발된 컴포넌트의 집합인 OCL(OROCOS Component Library), 로봇의 관절 제어에 실시간성을 제공하는 KDL(Kinematics and Dynamics Library), 동적인 베이스 네트워크(Bayesian Network) 상에서 재귀적인 정보처리 및 추측 알고리즘 등과 같은 추론을 위한 BFL(Bayesian Filtering Library) 등 4개의 C++ 라이브러리로 구성된다[5][6].

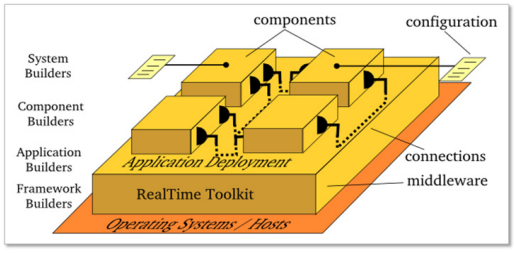

그림 2. OROCOS의 구조

전체적인 OROCOS의 구조는 [그림 2]와 같으며, 최하위 계층에 운영체제를 기반으로, RTT 라이브러리를 사용하여 로봇 컴포넌트를 개발하는 환경을 제공한다.

2. DeSiDeRaTa

DeSiDeRaTa(Dynamic, Scalable, Dependable, Real Time)는 분산 컴퓨팅 시스템에서의 QoS 관리 기술을

제공하는 미들웨어로써, 실시간성의 제약사항이 있는 시스템을 도메인으로 삼고 있다. 기존에 연구되어 오던 분산 환경에서의 QoS 관리 특징은 변수의 주기 측정, 비주기 프로세스, 우선순위, 고장 관리와 확장성 등이다. DeSiDeRaTa는 이러한 기존의 연구에 자원관리를 통한 QoS 보장으로 실시간성을 높이려는 노력을 기울였으며, QoS 명세와 동적인 QoS 관리에 대한 기술도 포함시켰다. QoS 명세는 계층적 구조, 응용 프로그램 내부의 연결 관계성, 실행 시 제약사항 등 동적인 경로 기반 실시간 시스템의 요구사항과 특성뿐만 아니라, LAN, 호스트, 장치 간 상호연결, 장치별 특성 등의 토폴로지에 대해서도 기술해 놓는다. 이러한 특징을 가진 DeSiDeRaTa 미들웨어는 분산 환경에서 발전된 동적 경로 패러다임(Dynamic Path Paradigm)을 지원한다[7-9].

DeSiDeRaTa QoS 미들웨어에서 실시간 제어경로 응용 프로그램은 QoS Metrics 컴포넌트에게 타임스탬프 이벤트를 보내고, 이에 대한 경로 레벨을 계산한 후 QoS Monitor에게 보낸다. QoS Monitor는 요구자원에 대한 위반사항을 탐지해 응용 프로그램을 다른 호스트나 LAN으로 보내거나, 종료시한(deadline)이 지날 때까지 자원을 할당하지 않음으로써 종료시키거나 하는 등의 동작을 선택한다. 이후에 필요자원과 가용자원의 분석 및 할당을 통해 요구한 QoS를 최대한 만족시킴으로써 실시간 제어경로 서브시스템 상에서의 QoS 보장으로 실시간성을 만족시킨다[10-13]. [그림 3]은 DeSiDeRaTa QoS 미들웨어의 논리적 구조를 나타낸다.

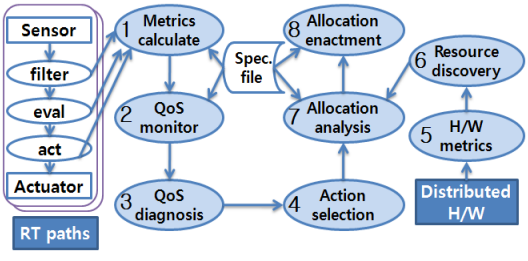

그림 3. DeSiDeRaTa QoS 미들웨어의 논리적 구조

3. proc 파일 시스템

Linux는 시스템이 실행중인 동안 커널이나 시스템을 재부팅 할 필요 없이 관리자로 하여금 커널을 변경할 수 있는 방법을 제공한다. 이는 /proc 이라는 가상파일 시스템으로 수행되며, 기본적으로 proc 파일 시스템은 실행 커널을 볼 수 있는 시각을 제공한다. 이는 퍼포먼스를 감시하고 시스템 정보를 발견하며 시스템이 어떻게 설정되었는지를 파악하고 그 설정을 변경할 때 유용하다[14].

proc 파일 시스템에서 시스템의 각종 프로세서, 프로그램 정보 그리고 하드웨어적인 정보들은 가상 파일의 형태로 존재하며 이를 쉽게 얻는 것이 목적이다. /proc 디렉토리에 존재하는 파일들은 실제 하드 디스크에 저장되지 않고 커널에 의해 커널영역의 메모리에 적재된다. 이러한 정보를 담고 있는 파일을 임의로 수정 및 변경하는 것은 시스템의 붕괴를 초래할 만큼 위험이 따른다. 본 논문에서는 단지 하드웨어 및 프로세스의 정보만을 필요로 하기 때문에 /proc 파일 시스템의 필요한 가상 파일을 읽어 그 내용을 가져오기만 하고, 직접적인 수정이나 변경을 가하지는 않는다[15]. [표 1]은 proc 파일 시스템의 주요 디렉토리와 그 역할을 보인다.

표 1. proc 파일 시스템의 하위 디렉토리

directory	Description
/meminfo	메모리와 스왑의 사용 및 가용 메모리를 저장하는 파일. 물리메모리와 가상메모리 모두의 정보를 가짐.
/cpuinfo	프로세서(CPU) 정보를 저장하는 파일
/malloc	kmalloc과 kfree 운영 정보 모니터링
/stat	시스템의 현재 상태에 대한 다양한 정보와 통계를 저장하는 파일.
/sys	중요한 커널과 네트워크 정보

III. QoS 보장을 위한 프레임워크의 설계 및 구현

1. 로봇 환경에서의 QoS 프레임워크의 구조 및 기능적 동작

로봇 컴포넌트의 QoS를 보장하기 위해 설계된 프레임워크의 기본적인 요구사항으로 다음과 같은 것들이

있다. 우선 하드웨어의 현재 상태를 파악하고 있어야 하며, 컴포넌트에 의해 요청된 QoS 요구사항에 대한 정확한 분석을 요구한다. 또한 요청된 QoS에 대해 위반사항을 예측하고 그에 대한 원인 조사를 통해 QoS 프레임워크로 하여금 적절한 동작을 하도록 명령하며, 컴포넌트에게 할당한 자원에 대해 지속적인 관리를 해주어야 한다. 여러 개의 응용 프로그램 혹은 하나의 응용 프로그램에 속한 여러 개의 컴포넌트들은 자신의 역할을 수행하기 위해 자원을 요청하게 된다. 이 때 시스템이 가진 한정된 자원으로 인해 컴포넌트 간에 충돌이 일어날 수 있으며, 이로 인해 컴포넌트는 자신의 종료 시한 내에 일을 처리하지 못하게 된다. 이러한 요구사항과 문제들을 해결하기 위해 로봇 컴포넌트와 운영체제 사이에 QoS 프레임워크를 설계 및 구현하였으며, 전체적인 구조를 [그림 4]에서 보여주고 있다.

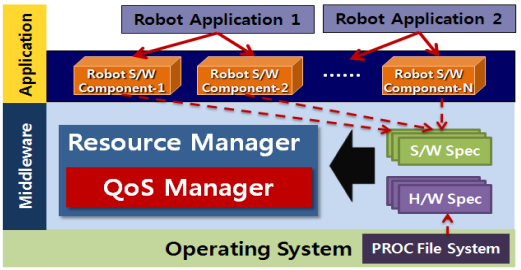

그림 4. QoS 프레임워크의 구조

[그림 4]의 QoS 프레임워크는 크게 QoS 관리자(QoS Manager)와 자원관리자(Resource Manager)로 나누어진다. QoS 프레임워크는 시스템의 가용한 자원량에 대한 정보를 알아내 하드웨어 프로파일에 저장한다. 본 논문에서는 리눅스 2.6.15 운영체제 상에서 구현을 하였기 때문에 리눅스에서 제공하는 proc 파일 시스템을 이용해 현재 시스템의 자원 정보를 얻어온다. 로봇 컴포넌트가 요청한 QoS는 구조체 형태로 QoS 관리자의 메시지 큐로 전달되며, QoS 관리자는 컴포넌트가 요청한 QoS와 현재 가용 자원량을 기준으로 자원 협상을 통해 최대한의 자원 할당을 함으로써 QoS를 보장해준다.

그림 5. QoS 프레임워크의 모듈구성

이러한 QoS 프레임워크의 전체적인 동작을 그림 5에서 보여주고 있다. QoS 자원협상(QoS Negotiator) 모듈을 기준으로 왼쪽의 QoS 모니터(QoS Monitor)와 QoS 위반사항진단(QoS Diagnosis)은 QoS 관리자의 모듈을, 오른쪽의 자원복구(Resource Discovery), 실행가능성 분석(Feasibility Analysis), 자원 할당(Resource Allocation)은 자원관리자의 모듈을 각각 나타낸다.

로봇 컴포넌트는 QoS 관리자에게 자신의 수행을 위한 QoS를 요청한 후 다음 동작에 대한 판단을 위해 QoS 관리자의 응답을 기다린다. QoS 관리자는 요청된 QoS를 QoS 위반사항진단 모듈을 통해 자원요청에 대한 위반사항을 탐지한다. 만약 위반사항이 발생하지 않았다면 즉시 자원을 할당해주지만, 위반사항이 발생하면 위반사항의 원인이 되는 자원의 종류와 이유를 컴포넌트에게 알려 자원협상의 수행을 유도한다. QoS 관리자로부터 메시지를 받은 컴포넌트는 자원협상이 필요하면 직접 QoS 자원협상 모듈을 호출한다. 만약 자원협상에 실패한다면 컴포넌트는 수행에 필요한 자원을 얻지 못하므로 해당 자원을 기다리는 상태로 변경되고, 자원협상에 성공한다면 해당 자원을 할당받기 위해 자원 할당 모듈을 호출해 실제 자원을 얻는다. 위와 같은 기능을 가지고 동작하는 QoS 프레임워크에 대한 설계와 구현을 다음 장에서부터 자세히 설명하도록 한다.

2. 로봇 컴포넌트의 QoS 보장을 위한 프레임워크 설계 및 구현

로봇 컴포넌트의 QoS 보장을 위한 프레임워크의 설계를 위해서는 우선 컴포넌트가 요구하는 QoS에 대한 정의가 필요하다. 본 논문에서는 QoS의 정의를 하드웨어 프로파일과 소프트웨어 프로파일로 나누어 정의하

여 [표 2]에 제시하였다.

표 2. 하드웨어 및 소프트웨어 프로파일의 정의

Profile		Description
H/W	CPU Utilization	현재 시스템의 CPU 사용량
	Memory Usage	시스템의 메모리 사용량
	Available Memory	컴포넌트가 사용 가능한 메모리의 크기
S/W	Execution Time	로봇 컴포넌트의 WCET(Worst Case Execution Time)
	Execution Period	로봇 컴포넌트의 실행 주기
	Requested Memory	QoS 관리자에게 요청할 메모리의 크기

하드웨어 프로파일은 현재 시스템이 가지는 자원에 대한 사용량을 proc 파일 시스템으로부터 얻어와 기술하며, 소프트웨어 프로파일은 로봇 컴포넌트의 QoS 요구사항을 파일 형태로 기술해 놓은 것이다. 하드웨어 프로파일 정보를 가져오는 함수의 프로토 타입은 다음과 같다.

*int getSystemInfo(QoSManager *qosm);*

로봇 컴포넌트는 소프트웨어 프로파일에 QoS 요구사항을 기술하여 QoS 프레임워크에게 전달한다. 이 때 QoS 프레임워크가 정보를 사용할 수 있도록 요청 큐(Request Queue)에 RequestQ 구조체 형태로 변형시켜 삽입한다.

표 3. QoS/Resource Manager의 주요 구조체 필드

Field		Description
QoS 관리자	int qmStatus	QoS 관리자의 전이 상태
	int ref_qos_file	컴포넌트가 요청한 QoS 참조 파일
	RequestQ requested	메시지큐에 삽입된 QoS
	int violation	QoS 위반사항이 발견됐는지 여부를 저장
	int neededRsc	요청된 자원의 종류(bit로 표시)
	int nego_count	자원 관리자와의 자원협상을 위한 필드
자원 관리자	int rmStatus	자원 관리자의 전이 상태
	int nego_count	QoS 자원협상을 위한 필드

QoS 프레임워크를 구성하는 주요 구조체는 QoSManager와 ResourceManager 이며 [표 3]은 각 구조체의 주요 필드를 나타낸 것이다. 이들은 QoS 프레임워크의 주체가 되는 구조체로써, 컴포넌트가 요청한 QoS 참조 파일의 이름이나 RequestQ 구조체 형태로 저장된 QoS 정보, 자원할당에 대한 정보 등을 가지고 있다. 또한 컴포넌트가 요청한 자원의 종류와 관리자의 상태 등을 담고 있다.

2.1 QoS Monitor

QoS Monitor 모듈은 로봇 컴포넌트가 요구하는 하드웨어 및 소프트웨어에 대한 QoS 요청을 감지하는 모듈이다. 각각의 로봇 컴포넌트는 주어진 역할을 수행하기 위해서 QoS 관리자에게 시스템이 가지는 자원에 대한 접근 및 사용에 대한 요청을 하게 된다. QoS 프레임워크의 함수 중에서 컴포넌트에게 유일하게 제공하는 함수가 바로 QoS 요청을 요청 큐에 삽입하는 함수이며, 프로토 타입은 다음과 같다.

int addRequest(char req_file_name);*

컴포넌트는 자신이 요청하는 QoS를 기술한 소프트웨어 프로파일의 이름을 인자로 하여 QoS 관리자에게 QoS 요구사항을 전달하고 [표 4]의 RequestQ 구조체 형태로 만들어 QoS 모니터 모듈에게 요청된 QoS가 있음을 알린다.

표 4. RequestQ의 구조체 필드

Field	Description
char* qosProfile	컴포넌트가 요청한 QoS 프로파일
int comPriority	컴포넌트의 우선순위
int comPid	컴포넌트의 프로세스 ID
int req_rsc	컴포넌트가 요청한 자원의 종류
int req_mem	요구 메모리의 크기
int req_min_mem;	최소 요구 메모리의 크기
int req_deadline	컴포넌트의 종료시한

QoS 모니터 모듈은 컴포넌트가 요청한 QoS에 대해 위반사항을 탐지하기 위해 QoS 위반사항진단 모듈을 호출하는데, QoS 위반사항이 발견되면 QoS 자원협상 모듈을, 그렇지 않으면 자원 관리자에게 해당 자원할당을 요청한다.

2.2 QoS Diagnosis

QoS 위반사항이란 컴포넌트 입장에서는 시스템이 약속된 서비스를 제공하지 않아 응용 프로그램을 실행할 수 없는 상태를, 시스템 입장에서는 요청된 자원의 사용으로 인해 약속된 QoS를 지원하지 못하는 상태로 정의할 수 있다. QoS 위반사항진단 모듈은 컴포넌트로부터 요청된 QoS에 대한 위반사항이 발생했는지 여부를 판단한다. 또한 위반사항에 대한 원인을 찾아내고 QoS 관리자로 하여금 위반사항을 해결하기 위한 다음 행동을 지시하는 역할을 수행한다.

QoS 위반사항진단 모듈은 우선 요청 큐에 저장된 QoS 요구사항을 순차적으로 가져오는데, 그 함수의 프로토 타입은 다음과 같다.

RequestQ getRequest();

이렇게 가져온 QoS 요구사항의 정보로부터 어떤 자원이 원인이 되어 위반사항이 발생되었는지를 알림으로써 QoS 관리자에게 QoS 위반사항을 해결하기 위한 기본적인 방안을 제시한다.

2.3 QoS Negotiator

QoS 자원협상 모듈은 비록 QoS 위반사항이 발견되어 당장 자원을 할당해 줄 수는 없지만, 최대한의 노력을 통해 QoS를 보장하기 위한 모듈이다. QoS 관리자와 자원 관리자 사이에서 가능한 자원을 할당하기 위해 요청된 QoS의 요청자원 감소(Degradation), 즉 QoS 프로파일에 미리 명시한 로봇 컴포넌트의 수행에 필요한 최소한의 자원으로 다시 QoS를 요청함으로써 QoS에 대한 합의점을 찾는다.

QoS 자원협상 모듈은 크게 QoS 관리자에서의 동작과 자원관리자에서의 동작으로 나누어 동작하며, [그림 6]은 QoS 관리자에서의 자원할당을 위한 협상 동작을 순서도로 나타낸 것이다.

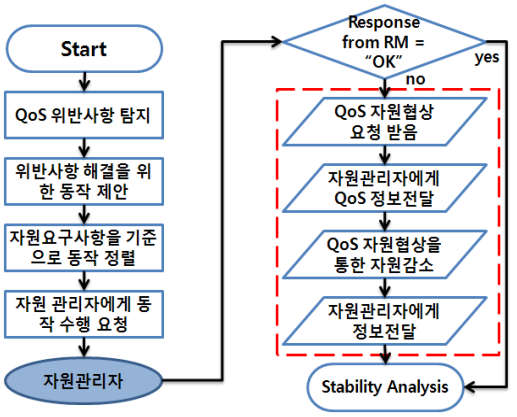

그림 6. QoS 관리자의 자원협상

QoS 위반사항진단 모듈로부터 위반사항을 탐지한 QoS 관리자는 자원 관리자에게 위반사항에 대한 내용을 알리고 협상이 시작되었음을 알린다. 자원 관리자가 QoS 관리자의 자원요청을 수락하지 않은 경우, 컴포넌트의 수행에 필요한 최소한의 자원량을 찾기 위해 QoS 요청자원 감소를 감행한다. 그러나 요구한 컴포넌트의 입장에서 볼 때, 요구한 QoS 만큼의 자원이 컴포넌트의 수행에 절대적인 영향을 미친다고 판단되면 컴포넌트는 QoS 요청자원 감소를 하지 않고 원하는 자원을 얻을 때까지 자원대기 리스트에서 QoS 요구사항 정보를 가진 채 기다린다. 만약 QoS 요청자원 감소를 해도 컴포넌트의 수행에 무리가 없다고 판단된다면, QoS 관리자는 QoS 요구사항에 대한 감소를 허가하는 메시지를 자원 관리자에게 전달함으로써 QoS 자원협상을 수행한다. QoS 관리자의 자원협상 과정에서 안정성 분석(stability analysis)은 자원 관리자에 의해 자원할당이 이루어진 후에 QoS를 요청한 컴포넌트가 종료시한 내에 자원을 할당받았는지 여부를 검사하는 역할을 한다. 만약 종료시한이 지난 컴포넌트에게 자원할당이 되었다면 안정성 분석 모듈에서 할당된 자원을 다시 반납하도록 함으로써 자원의 효율성을 증가시킨다. 이는 자원 관리자 내에서 수행되는 모듈로써 QoS 관리자는 자원 관리자로부터 안정성 분석에 대한 결과만을 반환값으로 받게 된다. 자원 관리자의 QoS 자원협상에 대한 동작은 [그림 7]에 나타내었다.

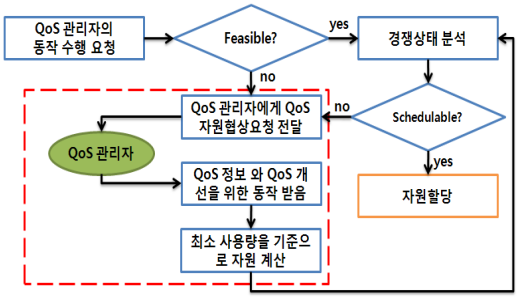

그림 7. 자원 관리자의 자원협상

자원 관리자는 QoS 관리자로부터 전달받은 QoS 보장을 위한 자원할당 동작에 대해, 현재 가용한 자원 및 동일한 자원을 기다리는 다른 컴포넌트와의 종료시한을 비교한다. 이 때 방금 요청받은 컴포넌트의 QoS에 대해 즉시 응답해 줄 수 없는 경우, 즉 요구된 자원보다 시스템의 자원이 부족하거나 요구된 자원을 기다리는 컴포넌트들 중에서 종료시한이 더 시급한 컴포넌트가 있는 경우에는 이를 QoS 관리자에게 알림으로써 QoS 요청자원 감소를 수행한다. QoS 관리자는 컴포넌트의 정보를 참조해 요청자원 감소에 대한 수락여부를 결정하고, 만약 수락한다면 자원 관리자는 컴포넌트가 감소시킨 QoS에 대해 최소 자원 사용량(Minimum Utility)를 기준으로 할당할 자원을 계산한다. 이 때 최소 자원 사용량은 컴포넌트가 수행될 때 가장 필요한 자원에 대한 최소한의 자원량을 의미하며, 컴포넌트에 의해 소프트웨어 프로파일에 이미 선언되어 있기 때문에 자원 관리자는 이를 이용해 할당 자원량을 계산하게 된다.

동일한 자원에 대해 경쟁 상태에 있는 컴포넌트들 중에서 종료시한을 넘지 않으면서 자원할당이 가능한 컴포넌트에 한해 자원할당 모듈로 하여금 요구한 QoS를 만족시키기 위한 자원할당을 하도록 한다.

2.4 Resource Allocator

자원할당 모듈(Resource Allocator)은 QoS 위반사항이 발생하지 않은 컴포넌트의 요청에 대해, 원하는 자원을 할당해주는 기능을 한다. 자원할당 모듈은 QoS 관리자로부터 해당 컴포넌트의 자원 소모량을 판단하고, 요청한 자원의 최근 활용 정도를 파악한 후, 해당 자원을 기다리는 다른 컴포넌트들과의 경쟁 상태를 분석한다. 이때 자원을 기다리는 컴포넌트들 중에서 지연시간으로 인한 종료시한 침범을 당하는 요청처리를 거부함으로써 응답시간을 최소화한다.

QoS를 요청한 컴포넌트에게 자원을 할당해 준 후, 자원할당 모듈은 하드웨어 프로파일에 해당 자원의 정보를 갱신함으로써 다른 컴포넌트의 QoS 요청에 대한 위반사항을 빠르고 정확하게 판단할 수 있게 한다.

IV. 실험 환경 및 결과

1. 실험 환경

일반적으로 로봇 환경은 리눅스와 같은 범용 운영체제를 사용하거나 혹은 thin-client와 같이 한정된 자원을 가진 도메인에서의 풋프린트를 고려한 실시간 운영체제를 기반으로 한다. 본 논문에서 구현한 QoS 프레임워크는 [표 5]에서 보이는 하드웨어 스펙과 리눅스 기반의 운영환경에서 구현하였으며, 분산 환경에서의 원격지 노드가 요청하는 QoS는 배제하였다. 또한 실험에서의 가용 메모리의 크기는 proc 파일 시스템으로부터 얻은 유저모드의 MemFree 필드의 값을 이용한다.

표 5. 실험 운영환경

Environment	Description
CPU	Intel(R) Pentium IV 2.66Ghz
Cache size	512 KB
RAM	512 MB
Linux Version	2.6.15
GCC Version	4.1.0

본 논문에서는 리눅스 상의 POSIX 라이브러리를 이용해 생성한 쓰레드를 하나의 로봇 컴포넌트라고 가정하였다. 각 로봇 컴포넌트의 CPU 점유는 리눅스의 스케줄링 기법에 의존하며, 컴포넌트의 QoS 요청에 대해서는 환형 큐(Circular queue)를 사용한 선입선출(First-In First-Out)을 기준으로 처리한다. 또한 로봇 컴포넌트가 요청하는 QoS 요구사항 중에서 컴포넌트

실행에 필요한 메모리의 요구량을 기준으로 실험을 진행하였다.

2. QoS 프레임워크의 동작 시나리오

QoS 프레임워크는 실행이 됨과 동시에 사용하는 모든 변수를 초기화한다. 이 때, 리눅스의 proc 파일 시스템을 통해 현재 시스템의 자원 상태를 저장하며, 이러한 시스템의 정보는 컴포넌트가 QoS를 요청하는 순간마다 업데이트 되도록 설계되었다.

proc 파일시스템으로부터 얻은 가용 메모리의 크기 MemFree 필드의 값은 실험결과에 대한 평균 측정을 위해 8160KB 로 고정하였으며, QoS 프레임워크가 로봇 컴포넌트에 대한 요청을 처리할 준비가 완료되면 [표 6]에서와 같이 각각의 QoS 요구사항을 가진 4개의 로봇 컴포넌트를 생성한다. 각 컴포넌트는 무작위 시간으로 생성되고 즉시 QoS를 요청한다.

표 6. 생성된 로봇 컴포넌트의 QoS 요구사항

컴포넌트	QoS 요구사항
Component [1]	– 요구 메모리 : 2900KB – 최소 요구 메모리 : 2750KB
Component [2]	– 요구 메모리 : 3100KB – 최소 요구 메모리 : 3000KB
Component [3]	– 요구 메모리 : 2900KB – 최소 요구 메모리 : 2800KB
Component [4]	– 요구 메모리 : 2200KB – 최소 요구 메모리 : 2000KB

QoS 프레임워크의 동작은 다음의 시나리오에 따라 수행된다. 우선 컴포넌트 1과 2는 요구 메모리에 대해 즉시 할당 가능하므로 QoS 프레임워크는 자원 관리자에게 자원할당을 요청한 후 남은 가용 자원량을 갱신한다. 컴포넌트 3은 가용한 시스템 메모리 크기보다 큰 메모리를 요청하기 때문에 QoS 위반사항을 발생시킨다. QoS 프레임워크는 이러한 위반사항을 탐지해 위반의 원인을 판단하고 이를 해결하기 위해 자원협상 모듈로 진입한다. 그러나 컴포넌트 3의 최소 요구 메모리의 크기조차 가용한 시스템 메모리의 크기보다 크기 때문에 요청한 QoS를 만족시킬 수 없음을 컴포넌트 3에게 알려준다. 마지막으로 컴포넌트 4 역시 QoS 위반사항을 발생시키지만, 최소 요구 메모리의 크기가 즉시 할당가능하기 때문에 이를 기준으로 자원할당에 성공한다.

```
++ Initialization Resource Manager!!

+++++ QoS Monitor Start +++++
        [System Memory] : 8160KB
Component 1 : QoS is Requested before 1sec.
++ [QoS Diagnosis] - QoS is detected!!
++ [Resource Allocation]
    -- MEMORY allocation : 2900KB complete!!
    -- Current System Memory : 5260KB
=> SUCCESS!! QoS processing complete!! : Component[1]
        ** QOS MANAGER's Diagnosis Time    : 9usec
        ** QOS MANAGER's Negotiation Time  : 0usec
        ** COMPONENT[1]'s Negotiation Time : 31usec
        ** Total Response Time of COMPONENT [1]: 40usec
Component 2 : QoS is Requested before 3sec.
++ [QoS Diagnosis] - QoS is detected!!
++ [Resource Allocation]
    -- MEMORY allocation : 3100KB complete!!
    -- Current System Memory : 2160KB
=> SUCCESS!! QoS processing complete!! : Component[2]
        ** QOS MANAGER's Diagnosis Time    : 10usec
        ** QOS MANAGER's Negotiation Time  : 0usec
        ** COMPONENT[2]'s Negotiation Time : 30usec
        ** Total Response Time of COMPONENT [2]: 40usec
Component 3 : QoS is Requested before 5sec.
++ [QoS Diagnosis] - QoS is detected!!
    -- Diagnosis result : SYS_MEM: 2160KB    req_mem: 2900KB
++ [QoS Negotiation]
    --Current MEMORY usage = 2160KB
     :: Negotiation Rupture - REQ_MIN_MEM 2800KB
        FAILED : System cannot satisfy Minimum QoS!
=> FAILED!! QoS processing incomplete!! :  Component[3]
        ** QOS MANAGER's Diagnosis Time    : 16usec
        ** QOS MANAGER's Negotiation Time  : 4usec
        ** COMPONENT[3]'s Negotiation Time : 63usec
        ** Total Response Time of COMPONENT [3]: 83usec
Component 4 : QoS is Requested before 7sec.
++ [QoS Diagnosis] - QoS is detected!!
    -- Diagnosis result : SYS_MEM: 2160KB    req_mem: 2200KB
++ [QoS Negotiation]
    --Current MEMORY usage = 2160KB
    --Minimum MEMORY Request = 2000KB
++ [Resource Allocation]
    -- MEMORY allocation : 2000KB complete!!
    -- Current System Memory : 160KB
=> SUCCESS!! QoS processing complete!! : Component[4]
        ** QOS MANAGER's Diagnosis Time    : 16usec
        ** QOS MANAGER's Negotiation Time  : 4usec
        ** COMPONENT[4]'s Negotiation Time : 73usec
        ** Total Response Time of COMPONENT [4]: 93usec

++ [QoS Monitor] Thread is desroyed!!
++ [req_thread] Threads are destroyed!!

 Program END.
[root@cwchoi0-linux qos_manager]#
```

그림 8. QoS 프레임워크의 동작

[그림 8]의 QoS 프레임워크의 동작 실험결과에서 보는 바와 같이, QoS 프레임워크는 QoS 요청에 대해 QoS를 만족시킬 가용한 자원이 있다면 즉시 자원을 제공함으로써 QoS를 보장하도록 설계되었다. 하지만 컴포넌트 3의 경우와 같이 요청된 QoS를 만족시킬 수 없는 경우, QoS 프레임워크는 이러한 사항을 컴포넌트에

게 즉시 알림으로써 컴포넌트로 하여금 다음 수행에 대한 결정을 내리는 데 중요한 정보를 제공한다. 이는 QoS 프레임워크가 컴포넌트의 QoS 요청에 대한 100% 보장이 목적이 아니라, QoS 자원협상과 같이 현재 자원 상황에 따른 최대한의 보장을 위한 노력이 목적임을 나타낸다.

3. QoS 프레임워크의 성능측정

위에서 설명한 시나리오를 토대로 본 논문에서 구현한 QoS 프레임워크 상에서의 QoS 요청과 그에 대한 수행시간을 측정하였다. 측정횟수는 각 성능측정실험당 5번을 한 세트로 하여 10 세트를 수행, 총 50회의 실험을 통해 평균 시간을 얻었으며, 단위는 μs(micro second) 이다. 또한 sleep()함수를 사용해 1초의 딜레이를 주고 리눅스 스케줄러를 이용한 쓰레드의 문맥교환(context switch)을 발생하게 하였다. 또한 proc 파일시스템으로부터 하드웨어 정보를 얻어오는 시간은 QoS 위반사항 진단이나 자원협상에 관계없이 평균 116μs로 항상 일정하게 동작하기 때문에 QoS의 수행시간에 포함시키지 않았다.

표 7. QoS 프레임워크의 수행시간

단위(μs)	컴포넌트 1	컴포넌트 2	컴포넌트 3	컴포넌트 4
위반사항 진단	9.74	9.82	15.48	15.6
QoS관리자의 자원협상	0	0	3.42	3.5
자원관리자의 자원협상	31.04	31.06	63.9	72.36
총 수행시간	40.78	40.88	82.8	91.46

[표 7]의 결과에서 컴포넌트의 QoS 요청에 대해 컴포넌트 1과 2처럼 즉시 허용 가능한 경우 QoS 프레임워크의 총 수행시간은 평균 40μs으로 측정되었다. QoS 자원협상이 이루어지는 경우는 크게 두 가지로 나누어지는데, 컴포넌트 3과 같이 최소 요구 자원량을 만족시킬 수 없는 경우 약 83μs의 수행시간이, 컴포넌트 4에서 최소 요구 자원량을 만족시킬 수 있는 상황에서는 약 91μs의 수행시간이 각각 측정되었다.

컴포넌트 3과 4의 응답시간이 컴포넌트 1과 2에 비해 길어진 것은 QoS 위반사항 진단 모듈에서의 QoS 위반사항 탐지로 인한 수행과 QoS 자원협상 모듈의 수행을 거치기 때문이다. 또한 컴포넌트 3과 4 사이에도 수행시간에서 차이가 발생하는데, 이는 QoS 자원협상 모듈 내부에서 수행되는 QoS 요청자원 감소 수행여부 때문이다. QoS 요청자원 감소는 최소 요구 자원량을 만족시킬 수 있는 지의 여부에 대한 검사가 이루어지고, 만약 가능하다면 자원 할당을 위해 자원할당 모듈을 호출하기 때문에 그만큼의 오버헤드가 발생하는 것이다.

표 8. QoS 자원협상 시간

단위(μs)	컴포넌트1	컴포넌트2	컴포넌트3	컴포넌트4
실험1	31.2	31.4	67	76.7
실험2	30.8	30.8	66.8	75.5
실험3	31.2	31.6	67.8	76.3
실험4	30.6	31	67.2	75.5
실험5	31.2	31	67.4	76.3
실험6	31.6	31.6	67.2	75.7
실험7	30.4	30.6	67.2	75.7
실험8	31.4	31.2	67	75.7
실험9	30.6	30.6	67.6	75.5
실험10	31.4	30.8	67.8	75.7
평균	31.04	31.06	67.3	75.86

QoS의 요청에 대한 총 자원협상 시간을 측정한 것이 [표 8]에 나타나 있다. 컴포넌트 1과 2는 QoS 위반사항이 발생하지 않았기 때문에 자원 관리자의 자원협상의 수행시간만 적용되고, 컴포넌트 3과 4는 QoS 관리자의 자원협상 수행시간과 자원 관리자의 자원협상 수행시간을 모두 포함한 것이다.

V. 결론

본 논문에서는 로봇 플랫폼 상에서 로봇을 구성하는 컴포넌트 간의 QoS 보장을 위한 프레임워크를 설계하고 구현하였다. 로봇 컴포넌트의 전체 수행시간의 관점에서 보았을 때, 로봇 컴포넌트가 요청하는 QoS가 QoS 프레임워크를 지남으로써 발생하는 수행시간은 QoS

프레임워크를 지나지 않을 때보다 분명히 오버헤드가 발생한다. 하지만 이러한 수행시간의 오버헤드를 감수함으로써 컴포넌트의 QoS를 최대한 만족시킬 수 있기 때문에 컴포넌트의 수행 완료에 있어서 안정성을 보장할 수 있게 된다.

향후 연구과제로는 로봇 컴포넌트의 일반적인 특성을 분석하여 로봇 QoS에 대한 모델링을 정의하고 이를 QoS 프로파일에 적용하는 연구가 이루어져야 한다. 일반적으로 QoS 프로파일은 XML 파일 형식으로 정의되며, 이러한 XML 파일에 대한 문장해석(parsing) 시간은 QoS의 처리에 또 다른 오버헤드가 된다. 따라서 QoS 프로파일을 최소한의 오버헤드를 가지는 형태로 저장하는 방법에 대한 연구도 필요하다. 또한 thin-client 의 개념을 가진 로봇의 특성을 고려한 자원 관리를 통해 로봇 시스템 가용 자원의 효율적인 할당에 대한 연구를 추가적으로 진행함으로써 더 높은 신뢰성과 안정성을 보장해야 한다.

참 고 문 헌

[1] 홍성수, “RSCA : 분산 로봇 플랫폼에서 임베디드 소프트웨어의 동적 재구성을 지원하는 통합 미들웨어”, 한국통신학회지, 제21권, 제10호, pp22-35, 2004.

[2] Dept. of Neural Information Processing, Univ. of Ulm, Germany, "Miro Manual ver0.9.4," 2006.

[3] Hans Utz, S. Sablatnôg, S. Enderle, and G. Kraetzschmar, "Miro-Middleware for Mobile Robot Applications," IEEE Transactions on robotics and automation, Vol.18, No.4, pp.493-497, 2002.

[4] OMG, "Robotic Technology Component(RTC) Specification," 2006.

[5] EURON(the EUropean RObotics Network), "The Orocos Component Builder's Manual ver1.4.1", 2008.

[6] http://www.orocos.org

[7] L. R. Welch, "Distributed, Scalable, Dependable Real-Time Systems : Middleware Service and Applications," Parallel and Distributed Processing, 13th International, 1999.

[8] R. Binoy, "Resource Management Middleware for Dynamic, Dependable Real-Time Systems," Real-Time Systems Vol.20, pp.183-196, 2001.

[9] The Open Group Research Institute System/ Technology Development Corp., "Managing Quality of Service within Distributed Environments," 2002.

[10] B. Shirazi, "QoS Middleware Support for Pervasive Computing Applications," System Sciences, Proceedings of the 37th Annual Hawaii International Conference, 2004.

[11] C. D. Cavanaugh and L. R. Welch, "Quality of Service Negotiation for Distributed, Dynamic, Real-Time Systems," Lecture Notes in Computer Science, Vol.1800, 2000.

[12] D. A. Menascé and H. Ruan, H. Gomaa, "QoS management in service-oriented architectures," Performance Evaluation, Vol.64, pp646-663, Issue 7-8, 2007.

[13] A. D. Ferdinando, P. Ezhilchelvan, M. Dales, and J. Crowcroft, "A QoS-Negotiable Middleware System for Reliably Multicasting Messages of Arbitrary Size," Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pp.253-260, 2006.

[14] http://www.opentech.at

[15] http://www.osx86.org/study/linux/procfs.txt

저 자 소 개

임 재 석(Jae-Seok Lim) 준회원

- 2008년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2008년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템, 로봇 미들웨어

조 문 행(Moon-Haeng Cho) 정회원

- 2004년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2006년 2월 : 충남대학교 컴퓨터공학과(공학석사)
- 2008년 2월 : 충남대학교 컴퓨터공학과(공학박사수료)
- 2008년 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템, URC 로봇

정 재 엽(Jae-Yeop Jeong) 준회원

- 2007년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2007년 3월 ~ 현재 : 충남대학교 컴퓨터 공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템, 로봇 미들웨어

이 철 훈(Cheol-Hoon Lee) 정회원

- 1983년 2월 : 서울대학교 전자공학과(공학사)
- 1988년 2월 : 한국과학기술원 전기및전자공학과(공학석사)
- 1992년 2월 : 한국과학기술원 전기및전자공학과(공학박사)
- 1983년 3월 ~ 1986년 2월 : 삼성전자 컴퓨터사업부 연구원
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터사업부 선임연구원
- 1994년 2월 ~ 1995년 2월 : Univ. of Michigan 객원연구원
- 1995년 2월 ~ 현재 : 충남대학교 컴퓨터공학과 교수
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 초빙연구원

<관심분야> : 실시간시스템, 운영체제, 로봇 미들웨어