creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

오류 처리 시간 단축을 통한 ROS 진단 시스템 개선

Reducing Error Handling Time of Diagnostics System in ROS

2017 년 2 월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템전공

최 우 인

오류 처리 시간 단축을 통한 ROS 진단 시스템 개선

Reducing Error Handling Time of Diagnostics System in ROS

지도 교수 홍 성 수

이 논문을 공학석사 학위논문으로 제출함
2017 년 1 월

서울대학교 융합과학기술대학원
융합과학부 지능형융합시스템전공
최 우 인

최우인의 공학석사 학위논문을 인준함
2017 년 1 월

위 원 장 _____ 박 재 홍 _____ (인)

부위원장 _____ 홍 성 수 _____ (인)

위 원 _____ 곽 노 준 _____ (인)

초 록

자동차 E/E 시스템 내의 소프트웨어의 규모는 점점 커지고 과거의 펌웨어 수준에 비해 많은 기능이 소프트웨어로 대체됨에 따라 복잡도 또한 높아지고 있다. 특히 자율주행 자동차가 개발되면서 소프트웨어의 관심이 집중되고 있다. 이러한 복잡한 시스템 내 소프트웨어의 신뢰성을 제고시키기 위해서는 주기적으로 프로그램의 행위를 모니터링하여 오류 검출 및 해결하는 작업이 필수적이다. 한편 로봇 진영에서 재사용 가능한 로봇 소프트웨어 플랫폼의 필요성이 대두되어 등장한 ROS (Robot Operating System)는 다양한 개발 지원 도구와 구독/발행의 효율적인 통신방식을 제공하는 오픈 소스 플랫폼으로, 로봇을 넘어서 자율주행 자동차 플랫폼으로의 활용이 시도되고 있다.

프로그램의 상태를 모니터링하는 메커니즘을 제공하는 ROS의 진단 시스템은 오류 검출 및 해결에 있어 비효율적인 메시지 처리 시간과, 메시지 전송 지연이라는 두 가지 지연 요소를 가진다. 오류 검출 및 해결 과정 전체에서 모든 진단 결과 메시지에 대한 리스트를 계속 유지하고 있기 때문에 비효율적인 메시지 처리가 야기되고, 진단 데이터를 수집하는 노드와 오류를 처리하는 노드가 독립적으로 실행되기 때문에 노드 간 메시지 전송 지연을 발생시킨다. 본 논문에서는 설정 가능한 오류 복구 레벨을 정의하고 오류 메시지 선별 처리, 직접 오류 처리 기법을 적용하여 문제를 해결한다. 개선된 ROS 진단 시스템은 실험 결과 기존 시스템에 비해 오류 검출 및 해결 시간을 25.1% 감소시켰다.

주요어 : 로봇운영체제, 자율주행 자동차, 진단 시스템, 오류 처리 시간, 구독-발행 통신, 런타임 소프트웨어 모니터링

학　번 : 2015-26052

목 차

제 1 장 서 론 1
제 2 장 배 경 7
제 1 절 ROS (Robot Operating System)의 특징 7
제 2 절 ROS의 진단 시스템 9
제 3 장 관련 연구 14
제 1 절 프로그램 변경을 통한 모니터링 14
제 2 절 프로그램 독립적인 모니터링 16
제 4 장 문제 정의 20
제 1 절 시스템 모델 20
제 2 절 ROS 진단 시스템의 응답시간 지연 문제 22
제 5 장 오류 처리 시간 단축을 위한 메시지 처리 개선 28
제 1 절 오류 메시지 선별 처리 29
제 2 절 직접 오류 처리 31
제 3 절 오류 복구 레벨 설정 가능한 진단 시스템 31
제 6 장 실험 및 검증 36
제 1 절 실험 환경 36
제 2 절 실험 구성 37
제 3 절 실험 평가 38
제 7 장 결 론 44

참고문헌 46

Abstract 49

표 목차

[표 1] 용어 설명 21
[표 2] 진단 데이터 수집 및 발행작업 실행시간 구성요소..... 24
[표 3] 진단메시지 처리작업 실행시간 구성요소 25
[표 4] 오류 검출 후 4가지 처리 방법..... 28
[표 5] 오류 복구 레벨 별 제안된 기법 적용 효과..... 35
[표 6] 실험 환경에 사용된 하드웨어와 소프트웨어 상세 37

그림 목차

[그림 1] ISO 26262의 결함 발견 시간과 결함 반응 시간 3
[그림 2] ROS 소프트웨어 스택 8
[그림 3] ROS의 토픽 기반 발행/구독 통신방식 9
[그림 4] ROS의 DiagnosticStatus 데이터 타입 10
[그림 5] ROS 진단 시스템의 기능 및 패키지 11
[그림 6] ROS 진단 시스템의 런타임 요소 12
[그림 7] rqt_robot_monitor의 실행 화면 13
[그림 8] 시스템 모델 21
[그림 9] 프로그램 행위 모니터링의 Gantt 차트 23
[그림 10] 진단 로직 수에 따른 오류 검출 및 해결시간 26
[그림 11] 노드간 메시지 전송: 4byte 크기 메시지 예 27
[그림 12] 프로그램 행위 모니터링의 구간별 소요시간 비율 27
[그림 13] 진단 업데이터 모듈에 추가된 변수 32
[그림 14] 모니터링 대상 노드 M의 진단 업데이터 적용 예 32
[그림 15] 개선된 진단 업데이터의 순서도(flowchart) 34
[그림 16] 하드웨어 설정: kobuki와 NotePC 연결 36
[그림 17] 관련 연구와의 비교 실험 결과 39
[그림 18] kobuki의 노드 관련 진단 로직 40
[그림 19] 진단 로직 수에 따른 오류 검출 및 해결 시간 41
[그림 20] 오류 메시지 비율 증가에 따른 오류 검출 및 해결 시간 42

제 1 장 서 론

자동차에 탑재되는 E/E (Electric & Electronic) 시스템 내의 소프트웨어의 위상은 점차 높아지고 있다. 과거 작고 단순한 펌웨어 수준이었던 자동차 소프트웨어는 운영체제, 플랫폼 수준의 방대한 양과 복잡도를 가지게 되었다. 게다가 최근 자율 주행 자동차에 대한 연구 개발 지원이 국가 수준에서 이루어지고 업체들은 앞다투어 자율 주행 자동차에 대한 기술 목표를 수립하여 발전시켜 나가고 있다. 자율 주행 자동차에 대한 하드웨어 발전도 필요하지만 대량의 센서 데이터를 기반으로 자율 주행 자동차를 동작시키는 것은 소프트웨어의 몫으로, 자동차 분야에 있어서 급격한 기술 발전의 중심은 소프트웨어에 있다고 말할 수 있다[1][2][3].

BMW는 2018년 자율주행 자동차 상용화를 위한 기술 개발에 힘쓰고 있으며 특히 자율주행의 일부분인 자동주차 기술력을 얻기 위해 원격 발렛 파킹 어시스턴트(Fully Automated Remote Valet Parking) 시스템을 개발하였다. BMW는 자율 주행 자동차를 위한 소프트웨어 플랫폼으로 ROS(Robot Operating System)를 활용하고 있다는 점이 주목할 만 하다. 자율주행을 위해 기존의 로보틱스 연구에서 이미 연구된 알고리즘을 쉽게 사용함으로써 개발 시간을 절약할 수 있다는 장점이 있고, 오픈 소스 플랫폼으로 가장 활발한 커뮤니티 등의 생태계를 가지고 있다는 점을 이유로 ROS를 채택한 것이다[4]. 나고야 대학에서는 자율주행 자동차를 위한 소프트웨어 스택으로 Autoware라는 오픈소스 소프트웨어를 공개하였다. Autoware 또한 ROS를 기반으로 개발된 것으로, 나고야 대학은 Autoware로 자율 주행

자동차 시스템 전체를 동작시키며 일본 나고야 시내에서 수백마일 주행에 성공하였다[5].

ROS (Robot Operating System)는 로봇 어플리케이션을 개발하기 위한 오픈 소스 소프트웨어 플랫폼으로, 2007년 Willow Garage 사에 의해 ROS라는 이름으로 개발이 시작되어 지금까지 주기적으로 릴리즈가 진행되고 있다. 특히 2013년 2월 오픈소스로 발표되면서 본격적으로 로보틱스 소프트웨어 내 가장 크고 활발한 커뮤니티를 가진 플랫폼으로 성장하였다[6]. 본 논문에서는 BMW와 나고야 대학의 적용 사례 등을 토대로 ROS를 자동차 소프트웨어의 플랫폼으로 선정하였다.

한편, 자동차 소프트웨어의 규모가 기하급수적으로 증가하고 복잡도 또한 증가하면서 소프트웨어의 신뢰성(Reliability) 보장을 위해서는 프로그램의 행위에 대해 이해하고 감시하는 작업의 중요성이 대두되고 있다[8][9]. 또한 자동차는 시스템의 장애가 사람에게 치명적인 원인이 될 수 있는 안전 필수 시스템(Safety-critical System)이기 때문에 시간 제약사항(Timing Constraints)이 엄격하다. 따라서 프로그램 행위 모니터링을 통해 결함이 검출되면 이를 해결하여 시스템이 다시 안전한 상태(Safe State)로 돌아갈 때까지 걸리는 시간이 해당 시스템의 시간 제약사항 내에서 이루어져야 한다. 자동차 전자 제어 장치 오작동으로 인한 사고 및 인명손실을 최소화하기 위해 2011년 제정된 자동차 기능안전 표준인 ISO 26262에서는 그림 1과 같이 시스템이 정상 동작을 하다가 결함을 검출하고 해결하는 시간에 대해 결함 검출 시간(Fault Detection Time)과 결함 반응 시간(Fault Reaction Time)을 명시하고 있다.

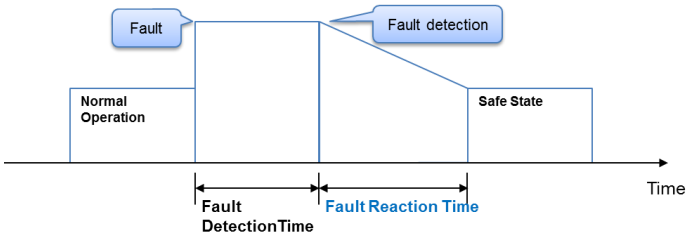

그림 1 ISO 26262의 결함 발견 시간과 결함 반응 시간

자동차 시스템 중 긴급 제동 시스템(AEB: Autonomous Emergency Braking)의 경우 요구사항에 명시된 반응 시간이 5ms로 엄격한 점을 미뤄볼 때 프로그램 행위 모니터링의 응답시간 즉, 오류 검출과 오류 처리 시간을 단축시키는 것은 필수적이다.

프로그램의 행위를 이해하고 감시하기 위한 방법으로서 프로그램 실행 중 소프트웨어 오류 검출, 진단 및 복구를 위한 런타임 소프트웨어 모니터링은 다양한 분야에서 연구되어 왔고 그 과정에서 많은 도구들이 제안되었다[17, 18]. 먼저 Java 언어를 대상으로 결함 검출 시간 단축을 위한 기법이 적용된 JRTM (Java Run-Time Timing-constraint Monitor) [19]와 정형 명세(Formal Specification)를 바탕으로 런타임에 감시 및 복구가 가능한 프레임워크인 MOP (Monitoring-Oriented Programming) 기반의 Java-MOP이 제안되었다[20, 23]. 이들 연구는 프로그램의 코드를 수정하지 않고 논문에서 정의하는 정형 명세 기반의 모니터링 로직이 독립적으로 작성될 수 있다는 장점이 있다. JRTM은 최대한 빠른 시간 내에 결함을 검출하고자 하는 연구로서의 의미가 있으나 발견된 오류가 처리되기까지의 시간은 고려되지 않았다. 대표적인 모바일 소프트웨어 플랫폼인 안드로이드에서는 서비스를 모니터링 하는 기법이 제안되었다[21]. 이 연구에서 제안하는 모니터의 동작은 본 논문의 방식과 유사하나, 모니터가 목적에 따라 모니터링

대상의 프로세스 내에서 실행되거나 별도의 서비스로 실행되도록 선택할 수 있다는 장점이 있다. 그러나 안드로이드는 자동차와 같은 안전 필수 시스템이 아니므로 심각한 오류 시 신속한 복구 메커니즘은 고려되지 않았다. 본 논문에서 자동차 소프트웨어 플랫폼으로 선정한 ROS에서도 다양한 연구들이 진행되었다. 앞서 언급한 MOP의 연장선 상에 있는 ROSMOP[24]과, 모델 기반의 진단-복구 아키텍처가 제안되었다[16]. 또한, Drums는 ROS로 인해 추상화된 분산 로봇 시스템을 실제 호스트 기반으로 시스템의 상태를 거시적으로 모니터링하는 도구이다[22]. Drums는 프로세스, 소켓, 리소스 등의 시스템 전반의 상황을 수집하여 모니터링할 수 있다는 장점이 있지만 세부 동작에서의 오류 검출은 불가능하며, ROSMOP은 런타임에 생성되는 하나의 모니터 노드에 진단 부하가 집중되는 문제가 존재한다. 모델 기반의 진단-복구 아키텍처는 기존의 진단 시스템과의 호환성을 유지하면서 모니터링 대상 프로그램의 코드 수정이 불필요하다는 장점이 있으나 모니터링 목적에 따라 별도로 존재하는 옵저버(Observer) 노드들의 메시지 구독 및 발행은 시스템의 복잡도를 높인다. 학계 뿐 아니라 자동차 산업계에서도 프로그램 행위 모니터링 기법이 적용되었다. AUTOSAR에서는 프로그램 흐름 모니터링을 위한 와치독 매니저(Watchdog Manager)[15], EU 프로젝트 EASIS에서는 실시간 특성을 보장하기 위한 소프트웨어 와치독(Software Watchdog)이 제안되었다[12]. 특히 AUTOSAR의 와치독 매니저는 ISO 26262의 소프트웨어 결함 목록에 기반한 프로그램 흐름 모니터링 메커니즘을 제공하지만 이들은 주어진 기능 외의 다른 소프트웨어 오류 검출이 불가능해 유연성이 떨어진다는 단점이 있다.

본 논문에서는 ROS에서 프로그램 행위 모니터링을 하기 위하여 진단 시스템(Diagnostics System)을 활용한다. ROS의 진단 시스템을

활용하면 런타임 모니터링을 위해 프로그램의 코드를 수정해야 하지만 모니터링 로직의 구현 자유도가 높다는 장점이 있다. 진단 시스템은 진단 업데이터를 통해 진단 작업에 의해 판단된 시스템의 진단 상태를 진단 메시지로 발행하고, 그 메시지를 구독하는 진단 핸들러 노드가 오류에 대한 처리를 수행한다. 본 논문에서는 ROS 진단 시스템의 빠른 응답시간을 보장하기 위해 오류 검출 및 해결시간을 지연시키는 두 가지 요소인 비효율적 메시지 처리 시간과 메시지 전송 지연을 도출하였다. 이는 오류 검출 및 해결 과정 전체에서 모든 진단 결과 메시지에 대한 리스트를 계속 유지하고 있기 때문에 비효율적인 메시지 처리가 야기되고, 진단 데이터를 수집하는 노드와 오류를 처리하는 노드가 독립적으로 실행되기 때문에 노드 간 메시지 전송 지연을 발생시키는 것이다. 이 문제를 해결하기 위해 오류 메시지 선별 처리와 직접 오류 처리 기법을 제안한다. 단, 직접 오류 처리 기법은 메시지 전송 지연은 해결하지만 노드 내 기능 통합에 의한 의존성 증가라는 trade-off 문제를 가지기 때문에 개선된 ROS 진단 시스템에서는 모니터링 대상 노드의 오류 발생 시 시스템의 동작에 심각한 영향을 주는 경우에 한해서 이 기법을 선택적으로 적용할 수 있도록 오류 복구 레벨을 제공한다. 본 논문에서 제안하는 기법을 ROS Kinetic Kame 버전의 진단 시스템에 적용하여 ROS에서 제안된 다른 기법[16, 24]과 비교하였을 때 오류 검출 및 해결 시간이 평균적으로 더 짧은 것을 확인하였고 kobuki 로봇 구동 시 기존 ROS 진단 시스템 대비 오류 검출 및 해결 시간이 평균 25.1% 단축 되었음을 확인하였다.

이를 통해 본 논문의 학술적, 기술적 기여는 크게 두 가지로 볼 수 있다. 첫째, 기존의 연구들이 빠른 오류 검출 혹은 오류 검출 기능 자체에만 집중하는데 비해 본 논문은 검출된 오류에 대한 해결 후 시스템을 정상 상태로 되돌리는 시점까지로 연구의 범위를 확장하였다.

개선된 ROS 진단 시스템을 통하여 우리는 모니터링 대상의 고장(Failure)이 시스템에 미치는 영향을 고려한 오류 복구 전략을 수립할 수 있다. 오류 복구 레벨에 따른 오류 처리 메커니즘이 이러한 오류 복구 전략을 실현 가능하게 한다. 또한 런타임 모니터링 도구의 성능으로 오류 검출 및 해결 시간을 지표로 제시하여 기존에 제안된 기법 대비 성능 우위에 있음을 실험적으로 검증하였다. 둘째, 본 논문에서 정의한 문제를 해결하는 과정에서 ROS의 진단 시스템을 분석하여 동작의 특성과 한계점을 도출 및 개선하였으며 이는 오픈소스인 ROS의 개선을 위한 하나의 시도로 활용될 수 있다.

본 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 논문의 이해를 돕기 위해 ROS의 특징과 진단 시스템에 대한 배경지식을 설명하고 3장에서는 프로그램 행위 모니터링을 위해 제안된 다양한 연구 결과를 검토한다. 4장에서는 본 논문의 대상 시스템 모델을 정의하고 ROS의 진단 시스템이 가진 응답시간 지연 문제를 정의한다. 5장에서는 제안하는 해결책을 설명하고, 6장에서는 실험을 통하여 이를 검증한다. 마지막으로 7장에서 논문의 결론을 맺는다.

제 2 장 배 경

이 장에서는 자율주행 자동차 플랫폼으로 활용이 되고 있는 ROS의 주요 특징과 본 논문에서 개선의 대상인 진단 시스템에 대해 설명한다.

제 1 절 ROS (Robot Operating System)의 특징

ROS[①]는 로봇 응용 프로그램을 개발하기 위한 로봇 소프트웨어 플랫폼으로 하드웨어 추상화, 하위 디바이스 제어, 로보틱스에서 많이 사용되는 센싱, 인식(Perception), 지도 작성, 모션 플래닝 등의 기능 구현, 프로세스 사이의 메시지 패싱, 패키지 관리, 개발 환경에 필요한 라이브러리와 다양한 개발/디버깅 도구를 제공한다[13].

ROS의 전체 소프트웨어 스택은 그림 2과 같이 구성되어 있다. ROS는 운영체제인 리눅스 위에서 동작하며, roscore 명령어에 의해 ROS 런타임이 수행되고 개발자의 목적에 따라 구현된 어플리케이션들이 동작한다. ROS 런타임을 구성하는 주요 서브시스템은 내비게이션, MoveIt 등의 로봇 어플리케이션, 로봇 어플리케이션을 구동하는 데 필요한 주요 라이브러리들로 구성된 로봇 어플리케이션

① ROS는 2007년 미국의 스탠퍼드 대학 인공지능 연구소(AI LAB)가 진행하던 STAIR(Stanford AI Robot) 프로젝트를 위해 Morgan Quigley이 개발한 Switchyard이라는 시스템에서 시작하였다. 미국의 로봇 전문 기업 윌로우게러지가 이를 이어받아 2010년 ROS 1.0을 발표했고 2016년 5월에 최신 버전인 ROS 키네틱 케임(Kinetic Kame) 버전을 릴리즈했다.

프레임워크, 노드 간의 통신을 담당하는 커뮤니케이션, 다양한 언어를 지원하는 클라이언트 라이브러리, 카메라, 센서 등의 하드웨어를 관장하는 하드웨어 인터페이스와 소프트웨어 개발을 지원하는 개발 툴로 구성된다.

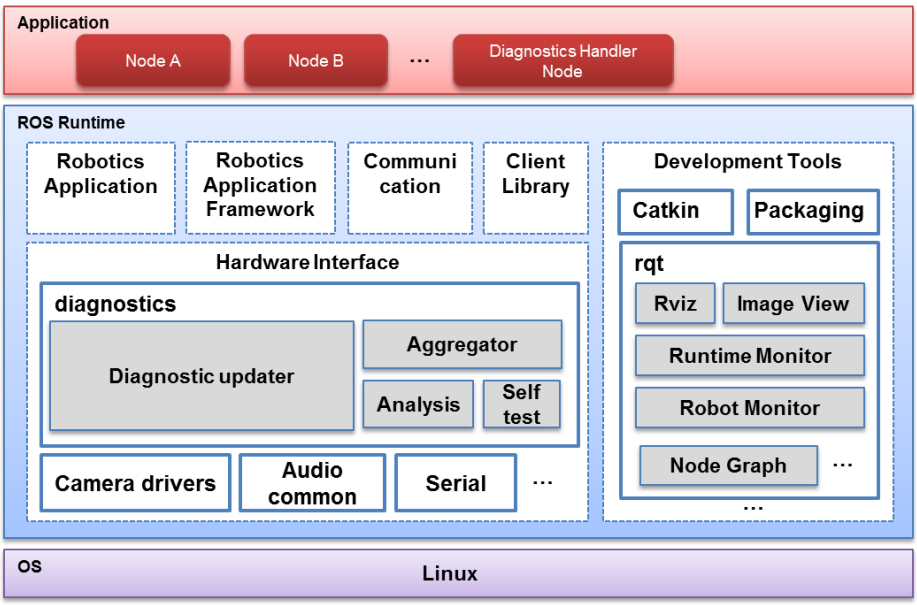

그림 2 ROS 소프트웨어 스택

ROS의 특징은 다음과 같이 다섯 가지로 꼽을 수 있다.

첫째, ROS는 리눅스와 같은 범용 운영체제 위에서 동작한다. 특히 리눅스의 한 배포판인 우분투(Ubuntu)의 프로세스 관리 시스템, 파일 시스템, 유저 인터페이스, 프로그램 유틸리티(컴파일러, 스레드 모델 등) 등을 사용하고 있다.

둘째, ROS는 런타임에서 단일 기능을 가진 최소 실행 단위인 노드들의 집합으로 구성된다. 독립적으로 실행되는 노드로 인해 컴포넌트 간, 하드웨어 간 의존성을 갖지 않고 노드 간 통신을 통해 구동되기 때문에 오류 발생 시 해결이 쉽고 재사용이 용이하다.

셋째, 메시지 전달을 위해 토픽 기반의 구독/발행 통신 방식을 제공한다. 토픽 메시지 통신은 그림 3과 같이 정보를 송신하는 발행자(Publisher)와 정보를 수신하는 구독자(Subscriber)가 토픽이라는 메시지 형태로 정보를 송수신하는 것이다. 발행자는 생성과 동시에 마스터에 자신의 정보를 등록하고, 구독자는 접속하려는 노드의 정보를 마스터로부터 취득한다. 이 정보를 활용하여 발행자와 구독자는 직접 접속하여 메시지 통신을 수행하며, 이들 간에는 N:M 통신이 가능하다.

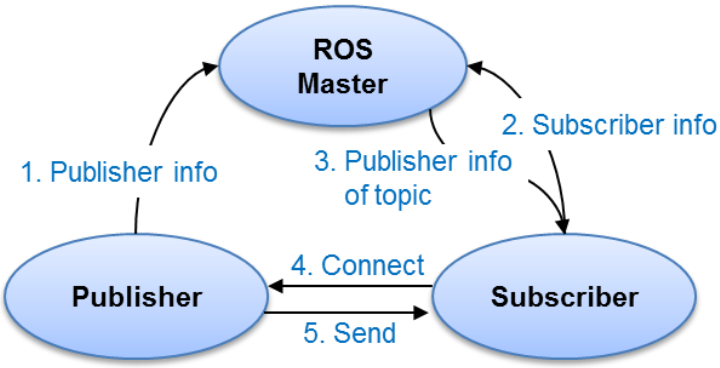

그림 3 ROS의 토픽 기반 발행/구독 통신방식

넷째, 진단 시스템을 제공하여 소프트웨어 개발 시, 혹은 로봇 구동 시 장비의 상태를 진단 및 모니터링할 수 있다.

다섯째, 로봇 개발에 필요한 다양한 GUI 도구를 제공한다.

제 2 절 ROS의 진단 시스템(Diagnostics System)

ROS의 진단 시스템은 로봇의 하드웨어 드라이버, 소프트웨어로부터

정보를 수집하여 사용자 및 구동자(operator)가 분석, 문제 해결 및 기록을 할 수 있도록 한다. 진단 시스템은 진단 데이터의 수집, 발행, 분석, 뷰의 기능을 하는 도구들로 구성된다.

진단 시스템은 /diagnostics 토픽을 중심으로 동작한다. 이 토픽을 통하여 하드웨어 드라이버와 디바이스들은 해당 디바이스의 이름과 상태, 장비 관련 정보를 DiagnosticStatus 타입의 메시지에 실어 발행한다. 그림 4는 DiagnosticStatus의 데이터 타입을 보여준다. 진단 데이터는 그림 4와 같이 진단 대상 항목의 이름과 상태(OK/WARN/ERROR/STALE), 해당 상태를 설명하는 message를 포함하며 추가적인 정보를 {key, value} 쌍의 형태로 가질 수 있다.

```
diagnostic_msgs::DiagnosticStatus
# Possible levels of operations
byte OK=0
byte WARN=1
byte ERROR=2
byte STALE=3

byte level # level of operation
string name # a description of the component
string message # a description of the status
string hardware_id # a hardware unique string
KeyValue[] values # an array of values
```

그림 4 ROS의 DiagnosticStatus 데이터 타입

그림 5는 ROS의 진단 시스템이 제공하는 기능과 이에 해당하는 패키지를 표현한 다이어그램이다. diagnostic_updater 패키지는 노드들이 진단 데이터를 수집 및 발행하도록 하는 기능을 제공한다. 이를 통해 드라이버, 소프트웨어, 하드웨어는 장비의 연결상태나 동작 주파수 등을 모니터링할 수 있다. diagnostic_aggregator는 실시간으로 진단 데이터를 분류하고 분석한다. Plug-in 모델을 활용하여

다양한 타입의 진단 데이터를 단순화하여 분석이 가능하며, 이는 개발자들이 로봇 운영 상의 공통적인 문제와 관련된 메시지를 쉽게 이해할 수 있도록 도움을 준다. 로봇의 개발자와 로봇 구동자는 rqt_robot_monitor 패키지를 통해 분석된 진단 데이터를 볼 수 있다. diagnostic_aggregator를 활용하지 않은 로봇의 진단은 원본 진단 데이터를 시각화하여 보여주는 rqt_runtime_monitor를 활용할 수 있다. 그리고 별도의 유틸리티로 활용할 수 있는 diagnostic_analysis 패키지는 ROS 실행 로그 중 진단 정보에 해당하는 데이터를 CSV(Comma Separated Value) 형식으로 변환하여 오프라인 분석이 가능하도록 편의 기능을 제공한다.

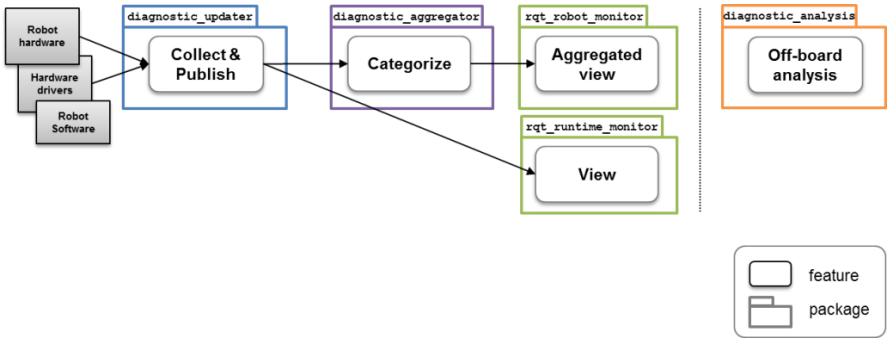

그림 5 ROS 진단 시스템(Diagnostics System)의 기능 및 패키지

위에서 언급한 패키지들이 런타임에서 실제 진단 시스템이 동작하는 모습은 그림 6과 같다. 모니터링의 대상이 되는 Node A 내에는 등록해둔 진단 업데이터가 동작하고 /diagnostics 토픽의 메시지를 발행한다.

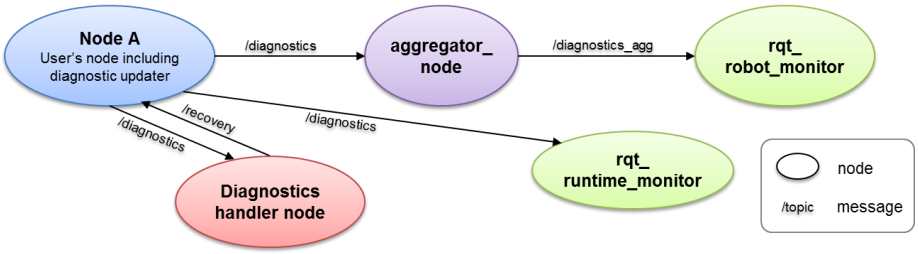

그림 6 ROS 진단 시스템의 런타임 요소

분류 및 분석을 위해 aggregator_node가 동작하며 이는 diagnostic_aggregator 패키지의 런타임 엔티티로, /diagnostics 토픽의 메시지를 받아 재분류하는 역할을 한다.

시각화를 위해서는 aggregator_node가 발행하는 /diagnostics_agg 토픽의 재분류된 진단 데이터를 입력으로 받아 그리는 rqt_robot_monitor 노드와, /diagnostics 토픽의 원본 진단 데이터를 입력으로 받는 rqt_runtime_monitor 노드가 동작한다. rqt_robot_monitor 노드를 실행하였을 때 볼 수 있는 화면은 그림 7와 같다. rqt_robot_monitor와 rqt_runtime_monitor는 ROS의 개발 툴인 rqt의 플러그인 형태로 존재하며 각 디바이스 혹은 모니터링 대상이 되는 코드의 진단 상태를 트리 형태로 보여준다.

진단 업데이터를 통해 판단된 시스템의 상태로부터 에러를 처리하는 노드는 독립적으로 실행되는 진단 핸들러 노드로, 이 또한 /diagnostics 토픽의 메시지를 구독하고 에러 처리 목적에 따라 필요한 경우 노드 A 또는 전체 시스템의 노드에 특정 명령을 전달하기 위해 /recovery 토픽의 메시지를 발행한다.

그림 7 rqt_robot_monitor의 실행 화면

제 3 장 관련 연구

이 장에서는 본 논문의 연구에 영향을 미친 관련 연구에 대해 기술한다. 자동차 소프트웨어로서 ROS의 신뢰성을 향상시키기 위한 프로그램 행위 모니터링 메커니즘에 대한 연구에 앞서, 기존의 학계 및 산업계에서 제안된 연구들을 살펴보겠다. 프로그램을 실행시키면서 동작을 감시하는 런타임 소프트웨어 모니터링은 소프트웨어 결함 검출(Detection), 진단(Diagnosis) 및 복구(Recovery)뿐 아니라 프로파일링, 성능 분석, 소프트웨어 최적화를 위해 사용되어 왔다. 그 과정에서 많은 도구들이 소프트웨어 결함 검출, 진단 및 복구의 목적으로 제안되어 왔다[17, 18]. 이러한 런타임 모니터링 도구들에 대한 연구는 크게 두 가지 방식으로 구분할 수 있다. 모니터링 대상 프로그램에 코드 레벨의 수정을 통해 모니터링 로직을 부여하는 방식과 프로그램과는 독립적으로 동작하는 모니터를 추가하는 방식이다. 이어지는 두 절에서는 두 가지 방식에 해당하는 기존 연구에 대해 각각 살펴본다.

제 1 절 프로그램 변경을 통한 모니터링

Xi Chen 등(2007)은 유럽의 자동차 제조업체, 공급업체, 연구기관에 의해 구성된 컨소시엄의 EASIS (Electronic Architecture and System Engineering for Integrated Safety Systems) 프로젝트에서 자동차 시스템의 실시간 특성을 보장하기 위해 소프트웨어

와치독(Software Watchdog)을 제안하였다. 소프트웨어 와치독은 세가지 기능을 가진다[12]. 프로그램의 생존여부와 모니터링 포인트의 도착 빈도를 감시하는 생존 감시(Heartbeat monitoring)과 프로그램의 실행 순서를 감시하는 프로그램 흐름 확인(Program flow checking) 기능, 위의 두 기능으로부터 검출된 오류의 수와 시스템의 임계값을 비교하여 프로그램의 상태를 판단하는 작업 상태 표시(Task state indication) 기능이다. 하지만 생존 감시와 프로그램 흐름 확인의 결과를 통합하여 시스템의 상태를 결정하므로 각각의 기능에 대한 오류 핸들링은 불가능하다는 단점이 있다.

AUTOSAR[②] 에서는 프로그램 흐름 모니터링을 제공하는 워치독 매니저(Watchdog Manager, WdgM)를 제안한다[15]. WdgM이 제공하는 프로그램 흐름 모니터링은 Alive Supervision, Deadline Supervision, Logical Supervision의 세 가지로 분류할 수 있다. Alive supervision은 주기적으로 수행되는 태스크에 사용되며, 이 태스크가 특정 시간 간격 동안 수행된 횟수를 측정하여 이 값이 미리 정해놓은 최소 횟수와 최대 횟수 내에 있는지 확인한다. Deadline supervision은 비주기적으로 수행되는 태스크에 사용되며, 특정 코드 구간을 수행한 시간이 미리 정해 놓은 최소 수행 시간과 최대 수행 시간 내에 있는지 확인한다. Logical supervision은 태스크의 제어 흐름을 기반으로 논리적으로 타당한 수행 순서를 지키는지 확인하며, 이를 위해 다수의 체크 포인트를 사용한다. 이들 체크 포인트를 기준으로 각 체크 포인트마다 어떤 체크 포인트로 분기될 수 있는지를 파악하도록 제어 흐름 그래프를 작성하여 이를 바탕으로 런타임에 논리적으로 타당한

---

② 2003년 BMW, Daimler AG, Continental, Bosch 등의 자동차 제조 회사, 자동차 부품 회사, 차량용 소프트웨어 회사와 반도체 회사 등 자동차 산업과 관련된 업체가 참여하여 만든 차량용 소프트웨어 플랫폼 표준으로, 최신 버전 4.2.2가 2015년 7월에 릴리즈 되었다. http://www.autosar.org/

수행이 되고 있는지 확인한다. 프로그램 흐름 모니터링의 결과가 비정상 상태일 경우 사용자 정의 콜백함수 호출, 진단 이벤트 관리자에게 보고, 파티션 종료, 하드웨어 워치독을 통한 리셋, MCU 리셋의 5가지의 오류 처리 작업을 수행할 수 있다. AUTOSAR의 프로그램 흐름 모니터링과 이에 대한 오류 처리 메커니즘은 ISO 26262에서 정의하는 시간 및 실행 측면의 소프트웨어 결함을 검출할 수 있고 단계적 오류 처리 방법을 제시하고 있으나 세 가지 감시 기능 이외의 목적으로는 추가할 수 없고 Deadline supervision과 Logical supervision은 모든 경우를 커버하지 못하는 로직 상의 약점을 가지고 있다[14].

Daniel Markart(2014)은 안드로이드의 서비스를 모니터링하는 기법을 제안하였다[21]. 대부분의 안드로이드 어플리케이션은 온라인 서비스 기반으로 동작하므로 서비스를 모니터링하고 오류를 가능한 한 빠른 시간 내에 해결하는 것이 중요하기에 그 특성을 반영한 서비스 모니터링 기법을 도출하였다. 이 연구에서 제안하는 모니터의 동작은 본 논문의 방식과 유사하나, 모니터가 목적에 따라 모니터링 대상의 프로세스 내에서 실행되거나 별도의 서비스로 실행되도록 선택할 수 있다는 장점이 있다.

제 2 절 프로그램 독립적인 모니터링

Aloysius K. Mok 등(1997)은 실시간 시스템에서 시간 제약사항을 위반한 경우를 최대한 빠른 시간 안에 검출하기 위한 기법을 제안하였다[19]. 제안하는 알고리즘을 Java 프로그램에서 적용할 수 있도록 JRTM (Java Run-Time Timing-constraint Monitor)으로

구현하였고 이는 시간 제약사항을 명세하기 위한 언어를 제공하고, 시간 제약사항 위반을 감시하는 알고리즘을 포함한다. 시간제약사항 명세를 컴파일하면서 모니터링 대상에 대한 시간 제약사항 명세를 최적화하고 시간 제약 위반 검출 시간의 상한(Upper bound)이 결정된다. 이와 같이 결함 검출 시간을 줄이기 위한 연구는 Haitao Zhu 등(2009)에 의해 심화되었다[25].

Feng Chen 등(2004)은 프로그램이 가져야 하는 특성을 문서화한 정형 명세(Formal Specification)을 런타임에 감시하고 복구하도록 하기 위한 프레임워크인 MOP (Monitoring-Oriented Programming)을 제안하였고, 이 기법을 다양한 도메인에 걸쳐 확장해 나가면서 프로그램 행위 모니터링의 한 분야를 확고히 하고 있다[20]. 이들은 Java-MOP을 제안하여 모니터링 대상 이벤트 및 그에 대한 핸들러에 대해 정형 명세를 작성하면 AspectJ를 활용하여 모니터링 대상 프로그램의 해당 위치에 모니터링 코드를 삽입하여 프로그램 실행 시 모니터링 및 오류 복구가 가능하게 하였다[23]. 이 연구의 연장선 상에서 Jeff Huang 등(2014)은 ROSMOP을 ROSRV 도구에 포함하여 ROS의 안전 관련 명세를 런타임에 모니터링할 수 있는 방법을 제안하였다[24]. MOP은 프로그램의 소스코드와 모니터링의 로직을 분리한다는 장점이 있으나 모니터링 로직이 소스코드로 변환되어 기존의 프로그램에 통합(Integration)되는 작업이 선행되어야 한다는 오버헤드가 있다. ROSMOP에서는 이 작업을 스크립트를 통해 선처리한 후 ROS를 실행시키도록 하고 있으나 매번 프로그램 실행 시마다 수행되어야 한다는 오버헤드가 된다.

S. Zaman 등(2013)은 ROS의 진단 시스템을 확장한 모델 기반의 진단-복구 아키텍처를 제안하였다[16]. 기존의 ROS의 진단 시스템은 /diagnostics 토픽을 발행하는 하드웨어 드라이버 혹은 소프트웨어

노드에 대해서만 모니터링이 가능하고, 다수의 모니터링 결과의 통합적인 분석이 불가능하여 실제 고장의 근본적인 원인(Root cause)을 찾기 힘들다는 제약사항을 설명하고 있다. 또한 자체적인 오류 복구 메커니즘을 제공하지 않기 때문에 자율주행 로봇 혹은 자동차의 진단 시스템으로의 한계점을 명시하였다. 옵저버(Observer) 기반의 진단 엔진을 통해 /diagnostics 토픽을 발행하지 않는 하드웨어 또는 소프트웨어의 행위를 모니터링하며, 다수의 노드의 상태를 동시에 체크하여 오류 여부를 종합적으로 판단할 수 있다. 또한 이러한 진단 결과가 진단 시스템이 가지고 있는 모델(Predefined Model)에 기반하여 비정상으로 판단된 경우 복구 엔진에 의해 적절한 복구 액션이 수행된다. 이 진단 엔진의 진단 결과는 기존의 진단 시스템과 호환성을 유지하기 위해 다시 /diagnostics 토픽으로 발행된다. 이 연구는 진단과 복구의 자동화를 시도했고 기존의 체크포인트 기반인 ROS의 진단 시스템에 모델 기반의 모니터링 기법을 적용했다는 점에서 의미가 있으나 제안한 기법에서 오류 검출 이후 해결까지의 소요시간에 대한 고려는 하고 있지 않다. 또한, 토픽 옵저버는 토픽의 통계 정보를 얻기 위해 모든 발행자 노드로부터 메시지를 구독함으로써 시스템의 복잡도를 높이는 단점을 가지고 있다.

Valiallah Monajjemi 등(2014)은 분산 로봇 시스템을 모니터링하는 도구인 Drums를 제안하였다[22]. Drums는 ROS와 같은 로봇 미들웨어에 의해 추상화된 실제 호스트 기반의 시스템 상황을 가시화하여 분산 시스템에서 문제의 원인이 되는 호스트를 판별할 수 있는 통합적 분석 방법을 제공한다. 이 연구는 분산 로봇 시스템의 각 호스트에서 정보를 수집하는 Collector가 동작하고, 이로부터 정보를 취합하여 시스템 오류에 대한 정확한 처리를 할 수 있게 도와준다는 강점이 있지만 그 모니터링 대상은 프로세스, 소켓, 리소스 등의 전체

시스템 수준의 정보에 초점을 맞추고 있어 프로그램 내부의 로직 상에서 오류를 검출 및 해결하는 데에는 한계가 존재한다. 또한 시스템의 외부적인 정보 수집에 의한 판단으로 인해 긍정 오류(False positive)를 도출할 수 있다는 단점이 있다.

제 4 장 문제 정의

이 장에서는 앞서 2장에서 설명한 ROS의 진단 시스템에서 개선하고자 하는 문제를 설명한다. 이를 위해 먼저 본 논문이 설명하고자 하는 대상 시스템을 설명한다. 그리고 ROS의 진단 시스템을 통한 프로그램 행위 모니터링 메커니즘에 대해 분석하여 한계점을 도출하고 본 논문에서 해결하고자 하는 문제를 정의한다.

제 1 절 시스템 모델

본 논문에서 문제를 정의하고 개선의 대상이 되는 시스템은 2장에서 상세히 설명한 ROS 의 진단 시스템으로, ROS 소프트웨어 스택 내에서는 두 부분에 나뉘어 존재한다. Hardware Interface 의 한 모듈로 diagnostics가 존재하고, 시각화 기능을 담당하는 패키지는 개발 툴 중 rqt의 플러그인으로 동작하는 runtime monitor와 robot monitor가 있다. 이 중 특히 프로그램 행위 모니터링과 연관된 모듈은 진단 업데이터와 진단 핸들러 노드로 그림 8 내에서 노란 실선으로 강조한 부분에 해당한다.

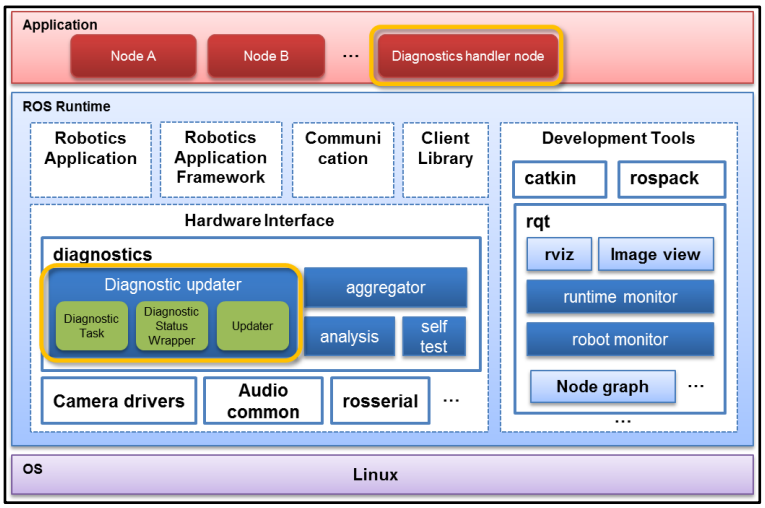

그림 8 시스템 모델

진단 업데이터와 진단 핸들러 노드를 통한 프로그램 행위 모니터링 과정을 분석하고 개선하기에 앞서 본 논문의 나머지 부분에서 다루게 될 주요 표기를 표 1과 같이 정의한다.

표 1 용어 설명

약어	설명
M	모니터링 대상 노드
H	진단 핸들러 노드
$D^M = \{d_1^M, d_2^M, \dots, d_n^M\}$	M의 진단 로직 집합
$S^M = \{s_1^M, s_2^M, \dots, s_n^M\}$	M의 D_M에 의한 진단결과 메시지 집합
E^M	S^M 중 오류(ERROR) 메시지 집합 즉, $E^M = \{s_k^M \mid s_k^M.level = ERROR\ (1 \le k \le n)\}$ $n(E^M) = m; 0 \le m \le n$
r^H	H의 실행 결과 M에 보내는 리액션 메시지
$T_{collect}^M$	M의 진단 데이터 수집 및 발행작업
T_{react}^M	M의 오류 복구작업
T_{handle}^H	H의 진단메시지 처리작업

프로그램 행위 모니터링은 두 개의 노드 즉, 모니터링 대상 노드 M과 진단 핸들러 노드 H에서 동작한다. M에는 n개의 진단 로직 D^M이 추가될 수 있고 진단 데이터 수집 및 발행작업 $T^M_{collect}$ 수행 요청이 들어오면 M에 등록된 진단 업데이터가 동작하면서 D^M의 모든 진단 로직이 순차적으로 수행되고 그 결과 n개의 진단 결과 메시지 S^M이 생성된다. 하나의 진단 결과 메시지 s^M_k는 진단 로직의 이름(name)과 상태(level), 해당 상태에 대한 설명(message) 등을 가지며 S^M의 원소 중 상태(level)가 오류인 진단결과 메시지의 집합을 E^M으로 정의한다. S^M을 H가 전달 받으면 진단메시지 처리작업 T^H_{handle}가 수행되고 그 결과 필요 시 마다 발행되는 리액션 메시지는 r^H이며 이는 다시 M이 구독한다. M이 리액션 메시지를 전달받아 수행되는 에러 처리 작업이 T^M_{react}이다.

제 2 절 ROS 진단 시스템의 응답시간 지연 문제

본 논문에서 ROS 진단 시스템의 응답시간 개선을 검증하기 위한 지표로 오류 검출 및 해결 시간을 사용한다. 오류 검출 및 해결 시간을 줄이기 위해서 기존의 ROS 진단 시스템이 가지는 다음 두 종류의 지연 요소를 줄이고자 한다.

정의 1. (Inefficient Message Handling Time) *비효율적 메시지 처리 시간*은 진단 결과 메시지 생성 및 처리 작업 내에서 비효율적인 메시지 처리에 소요되는 시간이다.

정의 2. (Message Transmission Latency) *메시지 전송 지연* $message\ transmission\ latency_{A \to B}$ 는 노드 A 의 작업 완료 후 노드 B의 작업이 시작되기까지의 지연 시간이다.

이해를 돕기 위해서 그림 9는 하나의 모니터링 대상 노드 M과 진단 핸들러 노드 H 에서 검출된 하나의 특정 오류에 대해, 오류 검출 및 해결시간을 Gantt 차트에서 도식화한다. 이 예시에서는 진단 결과 오류 상태가 반드시 하나 이상 존재함을 가정한다.

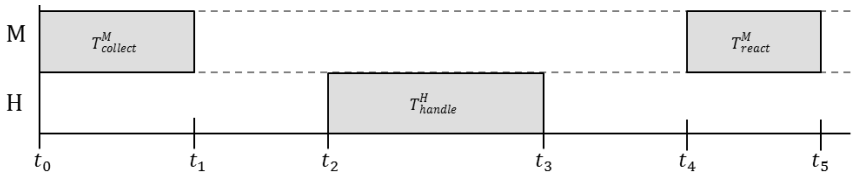

그림 9 프로그램 행위 모니터링의 Gantt 차트

모니터링 대상 노드 M 에서 `update()` 함수 호출에 의해 진단 업데이터의 동작 수행이 요청되어 t_0에 수행이 시작되면 오류여부 진단 및 진단 메시지 발행, 메시지 처리에 따른 오류 복구가 t_5에 완료된다. 이 예시에서, 오류 검출 및 해결 시간은 $t_5 - t_0$이고 그림 9에서 볼 수 있듯이 두 구간의 메시지 전송 지연 $t_2 - t_1$, $t_4 - t_3$ 이 존재한다. 비효율적 메시지 처리 시간은 왼쪽 막대 두 개가 나타내는 $T^M_{collect}$, T^H_{handle} 내부에 존재하며 그 내용은 $T^M_{collect}(t)$, $T^H_{handle}(t)$, $T^M_{react}(t)$ 와 함께 설명하겠다.

우선 M 의 진단 데이터 수집 및 발행작업 $T^M_{collect}$ 의 내부 로직에 따른 실행 시간은 다음과 같다.

$$T_{collect}^{M}(t) = init(t) + \sum_{i=1}^{n}\left[d_i^M(t) + append_i(t)\right] + \sum_{i=1}^{n} modify_i(t) + publish(t)$$

표 2 진단 데이터 수집 및 발행작업 실행시간 구성요소

수식	설명
$init(t)$	진단 데이터를 저장할 빈 벡터 status_vec를 초기화하는 데 걸린 시간
$append_i(t)$	status_vec 벡터에 i번째 진단 결과 메시지를 추가하는 데 걸린 시간
$modify_i(t)$	진단 결과 메시지의 이름 필드에 해당 노드의 이름을 덧붙여 수정하는 데 걸린 시간
$publish^M(t)$	/diagnostics 토픽에 DiagnosticStatus 타입의 진단 메시지 배열을 발행하는데 걸린 시간

기존의 진단 데이터 수집 및 발행 작업 시 M에 포함되어 있는 n개의 모든 진단 로직 결과가 순차적으로 메시지 배열에 추가된다. 이때 오류 복구와는 관계 없는 정상 혹은 경고 상태의 메시지도 구분 없이 발행되는 것이 노드 H에서 오류 메시지를 찾아서 리액션을 결정하는 과정에서의 비효율적인 메시지 처리를 야기한다.

H의 진단메시지 처리작업 T_{handle}^{H}의 내부 로직에 따른 실행 시간은 다음과 같다.

$$parse_1(t) + publish^H(t) \leq T_{handle}^{H}(t) \leq \sum_{i=1}^{n} parse_i(t) + publish^H(t)$$

표 3 진단메시지 처리작업 실행시간 구성요소

수식	설명
$parse_i(t)$	전송 받은 진단 메시지 배열 중 i 번째 진단 메시지를 파싱하여 그에 해당하는 리액션 로직을 결정하는데 걸린 시간
$publish^H(t)$	오류에 대한 리액션에 대한 명령 메시지를 발행하는데 걸린 시간

$T^H_{handle}(t)$ 은 노드 M 의 특정 오류 메시지 처리에 소요되는 시간이므로 그 오류 메시지가 전체 진단 결과 메시지 중 몇 번째에 위치했는지에 따라 위와 같이 계산될 수 있다. 최악의 경우 모든 진단 결과 메시지를 모두 파싱하고 마지막에 처리될 수 있다는 것이다. 빠른 오류 복구를 목표로 하는 안전 필수 시스템의 관점에서 보면 오류 메시지가 아닌 모든 진단 결과 메시지를 순차적으로 확인해야 하므로 비효율적인 메시지 처리 시간이 존재한다.

M의 오류 복구작업 T^M_{react}은 노드 H가 발행한 리액션 메시지 r^H를 전송 받으면 오류 복구를 수행하는 것이므로 별도의 수식 없이 $T^M_{react}(t)$로 표현한다.

지금까지 ROS 진단 시스템의 프로그램 행위 모니터링 메커니즘을 살펴본 결과 기존의 방식으로는 만족할 만한 수준의 진단 시스템의 응답시간을 달성할 수 없다. 그 원인은 크게 두 가지로 요약할 수 있다.

첫째, 앞서 언급했듯이 M의 진단 데이터 수집 및 발행과 H의 진단 메시지 처리작업 내의 비효율적인 로직에 의해 모니터링 대상 노드에 추가된 진단 로직이 많을수록, 오류 상태로 판단된 진단 로직의 수행 순서가 마지막에 가까울수록 비효율적 메시지 처리 시간에 의한 오류 처리 지연을 야기할 수 있다.

예 1: 모니터링 대상 노드 M에서 진단을 수행할 진단 로직 집합

D^M 의 원소 수가 증가하는 예제를 검토해 보자. 진단 로직을 하나씩 추가시키면서 오류로 검출되는 진단 로직의 순서가 첫번째인 경우와 마지막인 경우의 오류 검출 및 해결 시간이 증가하는 경향은 그림 10의 실험 결과와 같다.

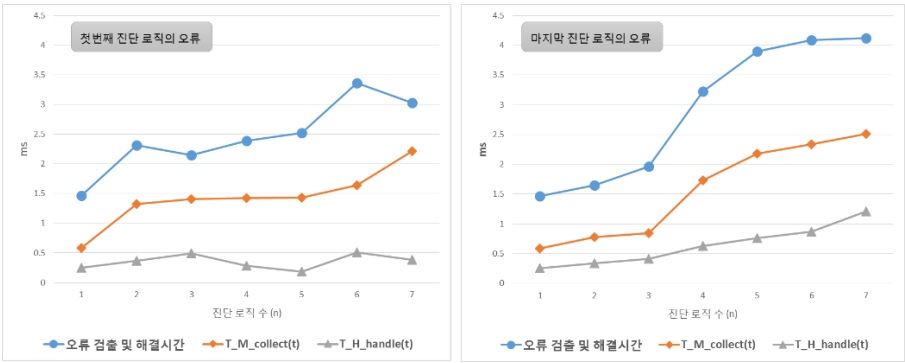

그림 10 진단 로직 수에 따른 오류 검출 및 해결시간

왼쪽 차트에서 볼 수 있듯이 첫번째 진단 로직의 오류인 경우 $T^M_{collect}(t)$ 만 진단 로직 수에 따라 증가하고, 오른쪽 차트는 오류로 검출된 진단 로직보다 먼저 처리되는 진단 로직의 수가 증가함에 따라 $T^M_{collect}(t)$와 $T^H_{handle}(t)$가 모두 증가하는 문제를 확인할 수 있다.

둘째, 진단 데이터 수집을 수행하는 노드 M 과 진단 데이터 중 오류를 식별하여 이에 대한 처리를 수행하는 노드 H 가 독립적으로 존재하기 때문에 메시지 전송 지연을 야기시킨다. 이는 2장에서 설명했던 ROS의 특징에 수반되는 현상이다. ROS는 단일 기능을 가진 최소 실행 단위인 노드들이 독립적으로 실행되기 때문에 컴포넌트 간, 하드웨어 간 의존성을 갖지 않는다. 이로 인해 오류 발생 시 해결이 쉽고 재사용이 용이하다는 장점이 있지만 실시간으로 프로그램의 행위를 모니터링 하면서 오류를 복구하고자 할 때에는 메시지 전달 메커니즘이 오버헤드를 갖게 된다. 그림 11과 같이 발행자 노드가 단순한 4bytes

크기의 메시지를 발행하여 구독자 노드가 받는 데 걸리는 소요 시간이 평균 0.47ms 이므로, 프로그램 행위 모니터링 시 필요한 메시지 전송이 2회임을 감안하면 그 이상이 소요된다는 것을 알 수 있다. 다음의 예시는 이러한 메시지 전송 지연의 문제를 보인다.

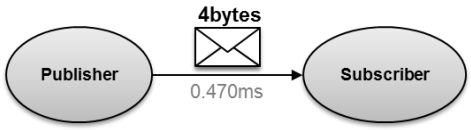

그림 11 노드간 메시지 전송: 4byte 크기 메시지 예

예 2: 임의의 진단 로직 수를 가지는 모니터링 대상 노드의 특정 오류 검출 시 각 구간 수행에 걸리는 시간의 비율은 그림 12와 같다. 진단 업데이터의 수행이 시작되는 순간부터 노드의 오류 처리가 완료되는 순간까지의 전체 구간 중에서 T_{react}^{M} 이 수행되기 직전까지를 나타내는 이 결과는 두 구간의 메시지 전송 지연 $message\ transmission\ latency_{M \rightarrow H}$, $message\ transmission\ latency_{H \rightarrow M}$ 이 각각 평균 13%, 15%를 차지하고 있음을 보여준다.

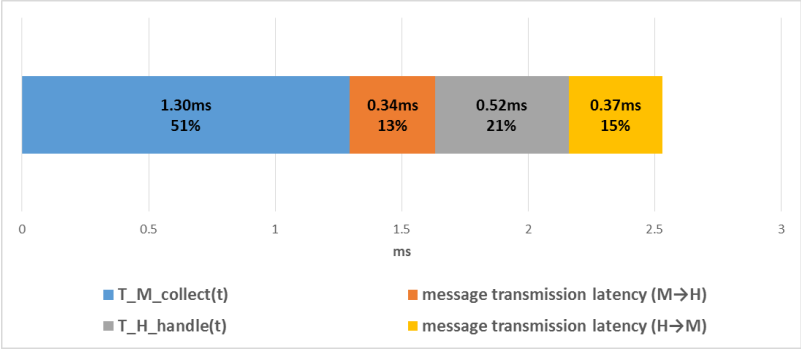

그림 12 프로그램 행위 모니터링의 구간별 소요시간 비율

제 5 장 오류 처리 시간 단축을 위한 메시지 처리 개선

이 장에서는 ROS 진단 시스템의 응답 시간을 단축시키기 위해 적용한 기법에 대해 설명한다. 4장에서 언급한 ROS 진단 시스템의 프로그램 행위 모니터링 시 두 가지 지연 요소인 비효율적 메시지 처리 시간과 메시지 전송 지연을 해결하기 위해 오류 메시지 선별 처리 기법과 직접 오류 처리 기법을 제안한다. 단, 직접 오류 처리 기법은 메시지 전송 지연은 해결하지만 노드 내 기능 통합에 의한 의존성 증가라는 trade-off 문제가 발생한다. 따라서 개선된 ROS 진단 시스템에서는 모니터링 대상 노드의 오류 발생 시 시스템의 동작에 심각한 영향을 주는 경우에 한해서 이 기법을 선택적으로 적용할 수 있도록 오류 복구 방법의 구분이 필요하다. 이에 ROS의 특정 노드를 모니터링 하다가 오류가 발생하였을 경우의 복구 레벨을 표 4와 같이 4단계로 구분하였고 각 레벨 별 오류 복구 방법을 정의하였다. 해당 모니터링 대상 노드의 오류에 대한 복구 레벨 판단은 개발자의 결정에 맡긴다.

표 4 오류 검출 후 4가지 처리 방법

레벨	오류 복구 방법
1	기존의 방식: 진단 메시지 발행 (초기값)
2	개발자가 구현한 콜백 함수 호출
3	해당 노드 종료
4	전체 ROS 노드 종료

레벨 1은 기존의 진단 시스템의 동작과 동일하게 진단 업데이터가 포함된 노드 내에서는 별도의 오류 처리를 하지 않고 진단 결과 메시지를 발행한다. 기존의 방식과의 호환성을 유지하고, 앞서 언급한 기능 의존성과 응답 시간의 trade-off 측면에서 진단 핸들러 노드를 별도로 유지하는 게 기능 의존성을 낮추는 데 우월하므로 때문에 레벨 1을 제공한다. 레벨 2는 개발자가 구현한 콜백 함수를 호출하는 단계이다. 레벨 3과 레벨 4는 해당 노드의 오류가 시스템의 동작에 심각한 영향을 주는 경우이다. 레벨 3의 경우 모니터링 대상 노드만 종료하게 되고 레벨 4는 현재 실행되고 있는 ROS의 전체 노드를 종료한다. 레벨 2부터 레벨 4는 보다 빠른 오류 해결이 요구될 경우 적용하는 단계로 모니터링 대상 노드 내에서 오류 처리가 가능하다.

이어지는 절에서는 오류 메시지 선별 처리 메커니즘과 직접 오류 처리 메커니즘 각각에 대해 설명하고 3절에서 이 기법들이 적용된 개선된 ROS 진단 시스템의 동작에 대해 설명한다.

제 1 절 오류 메시지 선별 처리(Error Message Preprocessing)

4장에서 분석한 것과 같이 모니터링 대상 노드 M의 진단 데이터 수집 및 발행작업 $T_{collect}^{M}$와 진단 핸들러 노드 H의 진단 메시지 처리작업 T_{handle}^{H} 과정에 존재하는 비효율적 메시지 처리 시간을 줄이기 위해 오류 메시지 선별 처리 기법을 제안한다. 기존의 ROS 진단 시스템에 존재하는 비효율적 메시지 처리의 원인은 모든 진단 결과 메시지 S^{M}에 대한 리스트를 계속 유지하고 있다는 점에 있다. 물론

2장에서 소개했듯이 /diagnostics 토픽의 진단 결과 메시지를 구독하는 다른 노드들이 존재하므로 – 진단 집합자(aggregator_node), 런타임 모니터(rqt_runtime_monitor)는 이 정보를 활용하여 현재의 시스템의 상태를 재분류하고 시각화한다 – 전체 진단 결과 메시지 리스트의 발행은 그대로 유지되어야 한다. 따라서 본 기법은 보다 빠른 오류 처리를 위해서 $T_{collect}^{M}$ 수행 시 오류 메시지 집합 E^{M} 에 대한 리스트를 별도로 구성하여 기존의 진단 결과 메시지보다 우선 발행한다. 그리고 진단 핸들러 노드 H 는 기존의 토픽 대신 오류 메시지를 나타내는 /diagnostics_error 토픽을 구독함으로써 메시지 파싱 로직에서 오류 여부를 확인할 필요가 없어지고 처리해야 할 메시지 수도 줄어든다. 이에 따라 $T_{collect}^{M}$과 T_{handle}^{H} 내에서 진단 결과 메시지에 대한 비효율적인 메시지 처리 시간을 줄일 수 있다. 다음은 오류 메시지 선별 처리에 따라 개선된 수행 시간 $T_{collect}^{M}{}'(t)$, $T_{handle}^{H}{}'(t)$이다.

$$T_{collect}^{M}{}'(t) = init'(t) + \sum_{i=1}^{n}\left[d_i^M(t) + append_i(t)\right] + \sum_{i=1}^{m} append_i(t) + \sum_{i=1}^{m} modify_i(t) + publish(t)$$

$$parse_1{}'(t) + publish^H(t) \leq T_{handle}^{H}{}'(t) \leq \sum_{i=1}^{m} parse_i{}'(t) + publish^H(t)$$

오류 메시지 집합의 원소 수 $n(E^M) = m$ ($0 \leq m \leq n$)이므로 기존 대비 감소할 것으로 기대된다. 단, $m = n$ 인 경우 오히려 E^M를 추가로 구성하는 오버헤드에 의해 기존 대비 $T_{collect}^{M}{}'(t)$가 증가하게 되나, 본 논문에서는 실제 시스템 상에서의 동작을 토대로 판단 결과 대부분의 진단 결과가 정상이고 오류의 비율이 낮다는 것을 전제로 한다.

제 2 절 직접 오류 처리(Direct Error Handling)

노드의 특성 상 오류 발생 시 긴급한 처리가 필요한 경우를 위해 노드 간 메시지 전송에 의한 지연을 차단하는 직접 오류 처리 기법을 제공한다. 노드의 오류 복구 레벨이 2인 경우 개발자가 구현한 콜백 함수 호출, 레벨 3의 경우 해당 노드 종료(node shutdown), 레벨 4의 경우 현재 실행되고 있는 시스템 전체의 노드들을 종료하는 메커니즘을 진단 업데이터에 내장하였다. 그 결과 진단 핸들러 노드 H로의 메시지 전송이 불필요하게 되어 두 구간에 존재하는 *메시지 전송 지연* $message\ transmission\ latency_{M \to H}$, $message\ transmission\ latency_{H \to M}$이 모두 0이 되는 효과를 가지게 된다.

제 3 절 오류 복구 레벨 설정 가능한 진단 시스템

이 절에서는 앞서 제안한 두 기법이 ROS의 개선된 진단 시스템에서 어떻게 적용되는지, 그 동작을 설명하고 그에 따른 오류 검출 및 해결 시간의 감소 효과에 대해 설명하겠다.

우선 ROS의 개선된 진단 시스템을 살펴보면, 오류 복구 레벨을 적용하기 위해 기존의 진단 업데이터의 `Updater` 클래스 내에 오류 복구 레벨을 저장하는 변수를 추가하였다. 이 변수는 4단계로 구성된 오류 복구 레벨에 대해 개발자가 식별이 용이하도록 그림 13과 같이 `lvl_recovery`의 열거형 데이터 타입으로 제공된다. 또한 이 변수에 대한 설정 API인 `setErrorRecoveryLvl()` 함수를 추가하였다.

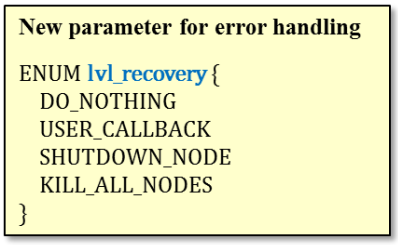

그림 13 진단 업데이터 모듈에 추가된 변수

설정 가능한 에러 복구 레벨로 인해 기존에 비해 모니터링 대상 노드 *M* 구현 시 변경되는 내용은 아래 그림 14의 음영 처리한 부분이다. 우선 진단 업데이터를 선언하고, 노드의 오류 복구 레벨을 설정하기 위해 `setErrorRecoveryLvl()`를 호출한다. 그 이후의 내용은 기존의 방식과 동일하다. 선언한 진단 업데이터에 진단 로직을 가지고 있는 진단 작업(Diagnostic Task)을 추가하고, 진단 업데이터의 동작이 필요한 부분에 `update()` 함수를 호출한다. 만약 개발자가 오류 복구 레벨을 2단계로 선택한다면, `Error_Handling_Callback()` 함수의 구현부를 추가하여 원하는 로직이 실행될 수 있도록 한다.

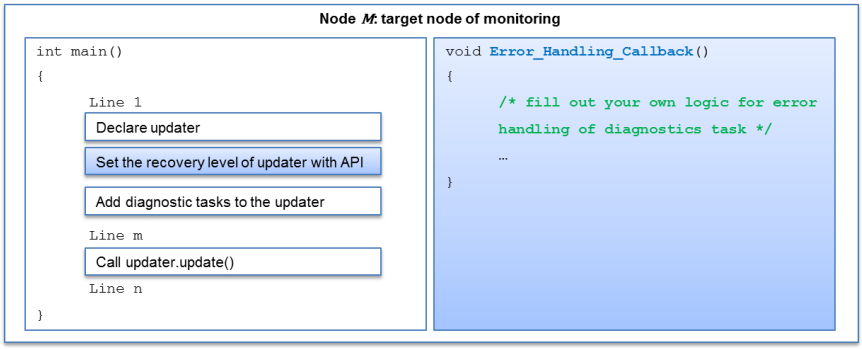

그림 14 모니터링 대상 노드 *M*의 진단 업데이터 적용 예

기존 대비 개선된 진단 업데이터의 논리 흐름은 그림 15와 같이

순서도(Flowchart)로 도식화하였다. 기존의 알고리즘에서 수정이 있거나 새로 추가된 로직은 음영을 진하게 표현하였다. 그리고 오류 메시지 선별 처리가 적용된 부분과 직접 오류 처리가 적용된 부분을 점선으로 묶어서 표현하였다. 하나의 진단 로직 수행 결과를 진단 결과 메시지 벡터에 추가시킬 때 결과가 오류인 경우 별도의 오류 벡터에 오류 메시지를 추가시킨다. 이후, 진단 업데이터 선언 시 설정한 오류 복구 레벨 값에 의해 동작이 분기한다. 진단 결과 오류가 존재한다는 가정 하에, 오류 복구 레벨이 3단계 혹은 4단계이면 오류 메시지 벡터를 순회할 필요 없이 바로 노드 종료 혹은 시스템 종료를 수행한다. 오류 복구 레벨이 1단계 혹은 2단계이면 기존의 진단 결과 메시지 벡터 status_vec가 아닌 오류 메시지만으로 구성된 error_vec을 순회하면서 에러 메시지 발행에 필요한 로직을 수행(1단계)하거나, 개발자가 구현한 Error_Handling_Callback(2단계)를 호출한다. 그리고 3, 4단계의 경우 해당 노드 혹은 시스템이 종료되므로 남은 로직이 없지만 1, 2단계의 경우 오류 메시지 벡터 처리 후 기존의 로직인 전체 진단 결과 메시지를 /diagnostics 토픽으로 발행을 수행하게 된다.

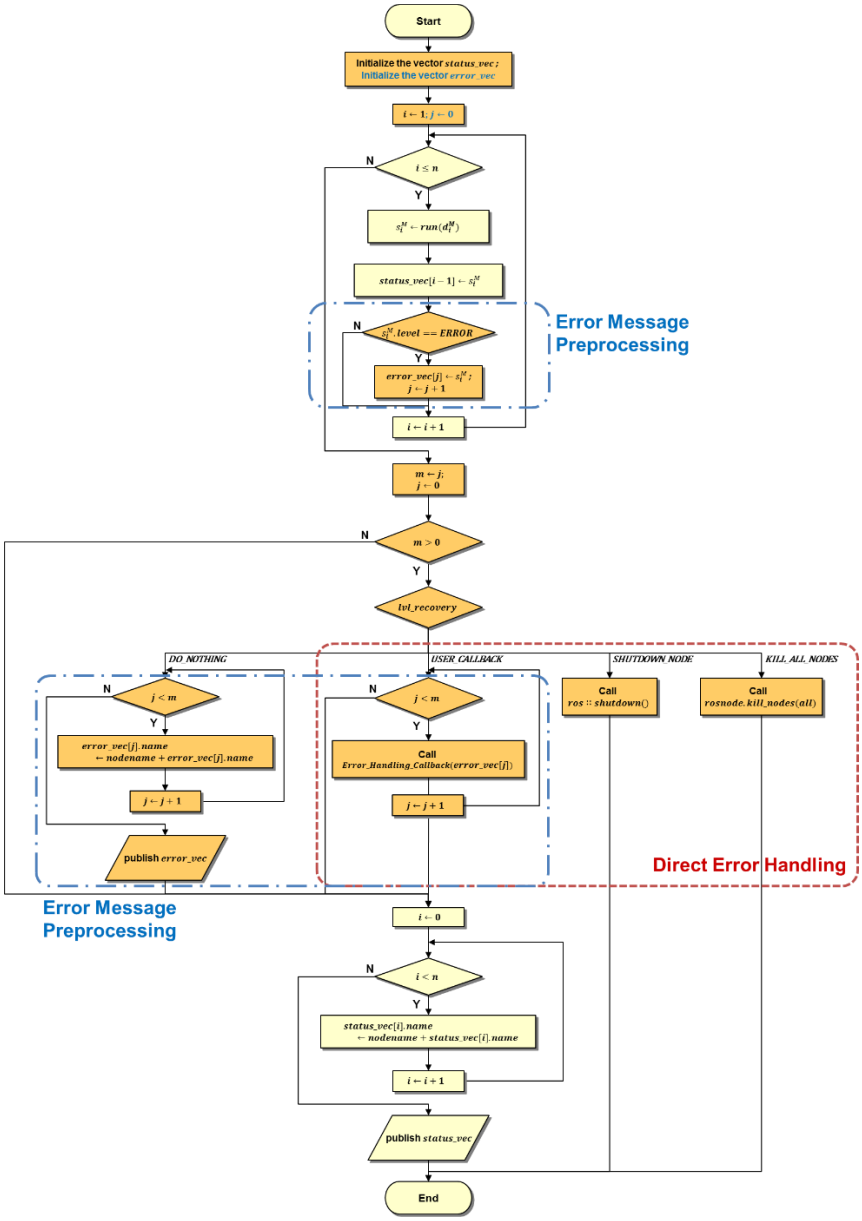

그림 15 개선된 진단 업데이터의 순서도(flowchart)

순서도에서 식별한 것과 같이 오류 복구 레벨이 설정 가능한 개선된 ROS 진단 시스템 내에서 두 개의 개선 기법이 적용된 범위를 표 5에서 보여준다. 오류 메시지 선별 처리는 단계 별로 적용되는 수준이 달라져 그 정도를 (●)의 개수로 표기하였다. 1단계에서는 별도로 구성된 오류 메시지 벡터를 순회하게 되어 비효율적인 전체 진단 결과 메시지 순회를

막을 수 있고, 그 오류 메시지 벡터가 전달된 진단 핸들러 노드에서의 메시지 순회 시간도 줄여준다. 2단계의 경우 진단 핸들러 노드에서의 작업은 불필요하므로 모니터링 대상 노드 내에서만 효율적인 메시지 처리 효과가 있다. 3, 4단계의 경우 오류 메시지 선별 로직이 추가됨에 따라 오류 존재 여부를 알 수 있게 되어 추가적인 메시지 처리가 불필요하다. 직접 오류 처리는 오류 복구 레벨 2, 3, 4단계인 경우 모니터링 대상 노드 내에서 로직 수행이 가능하도록 적용되었다.

표 5 오류 복구 레벨 별 제안된 기법 적용 효과

레벨	오류 복구 방법	오류 메시지 선별 처리	직접 오류 처리
1	기존의 방식: 진단 메시지 발행	●●●	
2	개발자가 구현한 콜백 함수 호출	●●	●
3	해당 노드 종료	●	●
4	전체 ROS 노드 종료	●	●

제 6 장 실험 및 검증

이 장에서는 앞서 제안한 ROS의 개선된 진단 시스템이 4장에서 정의한 문제점을 해결하는지 검증하기 위해 실험 환경을 구축하고 실험 시나리오를 구성하며 그 결과를 보인다.

제 1 절 실험 환경

ROS의 개선된 진단 시스템과 실험 시나리오를 구현하고 검증하기 위하여 표 6과 같이 하드웨어와 소프트웨어 실험 환경을 구성하였다. 하드웨어는 유진로봇 사의 kobuki와 이를 제어하는 Note-PC를 그림 16과 같이 연결하였다. Note-PC에 올라가는 소프트웨어로 ROS의 최신 버전인 Kinetic Kame에 호환되는 리눅스 배포 버전인 우분투 16.04을 설치하였다. 또한 ROS Kinetic Kame의 소스코드에서 본 논문에서 제안하는 해결책을 구현하기 위해 진단 업데이터 코드를 수정하여 설치하고, kobuki를 제어하기 위한 kobuki 관련 패키지를 설치하였다.

그림 16 하드웨어 설정: kobuki와 NotePC 연결

표 6 실험 환경에 사용된 하드웨어와 소프트웨어 상세

하드웨어			소프트웨어	
로봇	kobuki		운영체제	Ubuntu 16.04
Note-PC	프로세서	i5-4210U	플랫폼 (ROS)	Kinetic Kame
	메모리	DDR3L 4GB		Diagnostics updated patch
	스토리지	SSD 128GB	Kobuki 어플리케이션	Kobuki
				Kobuki_soft
				Kobuki_node

제 2 절 실험 구성

개선안의 오류 처리 시간 단축을 검증하기 위해 본 논문에서는 메트릭으로 오류 검출 및 해결 시간을 정의하였다. 오류 검출 및 해결 시간은 진단 업데이터의 update()가 실행되기 시작하는 지점부터 오류에 대한 리액션 수행이 완료될 때까지의 소요시간이다. 이와 같은 메트릭을 측정하기 위해 기존 시스템과 개선된 시스템에서는 동일하게 ROS에서 제공하는 로깅 API인 ROS_INFO를 사용하였다. ROS 로깅 API는 출력 메시지와 함께 해당 코드가 수행된 시점을 [second.nanosecond] 형식으로 출력해준다는 점을 활용하여 측정하고자 하는 각 코드의 주요 지점에 ROS_INFO API를 삽입하여 코드가 수행된 시점을 기록할 수 있었다. 또한 이 측정 데이터는 ROS에서 제공하는 개발 툴 중 하나인 ROS 콘솔(rosconsole)을 통하여 로깅 API가 호출된 시점에 대한 데이터를 추출하였다.

제 3 절 실험 평가

이 절에서는 두 가지 관점에서의 실험적 검증 결과를 보여준다. 본 논문에서 제안하는 기법의 오류 검출 및 해결 시간을 기존의 연구들에서 제안된 기법과 비교하고, 개선된 진단 시스템이 본 논문에서 문제점으로 정의한 기존의 ROS 진단 시스템이 가지는 2가지 지연요소를 해결하는지 검증한다.

■ 기존 연구와의 비교 검증

3장에서 살펴본 런타임 소프트웨어 모니터링에 대한 다양한 연구 중 본 논문의 기법과 동일한 조건에서 비교 가능한 ROS에서의 연구를 살펴보자. Drums[22]는 시스템 전반의 정보를 모니터링하는 용도이므로 비교검증에서 제외하고, S. Zaman이 제안한 모델 기반의 진단-복구 아키텍처[16]와 J. Huang이 제안한 ROSMOP[24]와의 비교 실험을 통해 본 논문에서 개선한 ROS 진단 시스템이 오류 검출 및 수행 시간 측면에서 우세함을 확인할 수 있었다. 본 논문의 솔루션을 포함한 세가지 기법들은 감시할 수 있는 로직이 완벽히 동일하지 않으므로[26, 27] 공통적으로 구현하여 비교 측정이 가능한 시나리오로 토픽의 발행 빈도(Frequency)가 기준치를 벗어날 경우 오류로 검출하고 이에 대한 복구 액션으로 해당 모니터링 대상 노드를 종료시키는 실험을 수행하였다. 한편 ROS 진단 시스템과는 달리 나머지 두 기법은 3장에서 언급했듯이 프로그램의 변경이 불필요한 독립적인 모니터링 메커니즘을 가진다. 따라서 이들 기법은 이미 노드에서 메시지가 발행되고 나서 그 메시지를 받고 난 후 모니터의 판단이 시작된다. 이

실험에서는 비교의 공정성을 위해 메시지 발행(Publish) 시점과 복구 액션인 노드 종료(Shutdown) 직전에 ROS_INFO 로깅을 하여 그 소요시간을 측정하였고 그 결과는 그림 17과 같다.

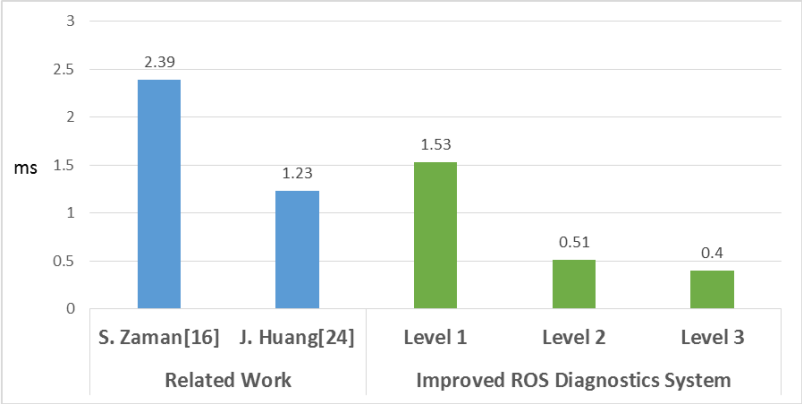

그림 17 관련 연구와의 비교 실험 결과

개선된 ROS의 진단 시스템은 오류 복구 레벨에 따라 오류를 검출하여 해결하기까지의 시간 차이가 존재하지만 평균적으로 기존의 두 연구에서 제안하는 모니터링 기법에 비해 응답시간이 빠르다는 것을 확인할 수 있었다. 단, 위와 같이 단순한 시나리오에서 J. Huang의 기법은 본 논문의 솔루션 중 1단계에 비해서는 빠른 결과를 보이나 시스템이 복잡해지고 모니터링 대상이 많아질 경우 J. Huang의 기법은 하나의 모니터 노드에 감시의 역할이 집중되어 현격한 성능 저하를 가져온다. 한편 S. Zaman의 기법은 옵저버, 진단 엔진, 복구 엔진이 별도로 구성되어 각 활동을 거쳐서 마지막으로 복구 액션 메시지가 전달되기까지의 오류 검출 및 해결의 총 소요시간은 다른 기법에 비해 상대적으로 길다. 따라서 S. Zaman의 기법은 시간 요구사항이 엄격한 자동차 소프트웨어에서는 적절치 않는 것으로 보인다.

■ 제안된 기법의 대표 시나리오 검증

본 논문에서 제안한 기법이 4장에서 정의한 2가지 지연 요소를 해결하는 지 검증하기 위한 대표 시나리오 실험 구성은 다음과 같다. 우선 프로그램 행위 모니터링의 대상은 로봇 kobuki를 제어하는 kobuki_node 패키지의 mobile_base_nodelet_manager 노드로 지정하였다. 해당 노드에는 총 10개의 진단 로직을 추가할 수 있다. 그 목록은 그림 18과 같이 kobuki 노드의 헤더파일 /kobuki_node/include/kobuki_node/kobuki_ros.hpp 일부에서 확인할 수 있다. 기존의 진단 시스템과 개선된 진단 시스템의 성능 비교를 위해 기존의 시스템을 통한 오류 처리는 /diagnostics 토픽을 구독하는 진단 핸들러 노드를, 개선된 진단 시스템을 위해서는 /diagnostics_error 토픽을 구독하는 진단 핸들러 노드를 두어 각 진단 로직 수행 결과에 해당하는 진단 상태가 오류인 경우에 대해 처리하는 로직을 구현하였다.

Diagnostic Task for kobuki

```
  /**********************
  ** Diagnostics
  **********************/
  diagnostic_updater::Updater updater;
  BatteryTask      battery_diagnostics;
  WatchdogTask    watchdog_diagnostics;
  CliffSensorTask   cliff_diagnostics;
  WallSensorTask   bumper_diagnostics;
  WheelDropTask     wheel_diagnostics;
  MotorCurrentTask  motor_diagnostics;
  MotorStateTask    state_diagnostics;
  GyroSensorTask     gyro_diagnostics;
  DigitalInputTask dinput_diagnostics;
  AnalogInputTask  ainput_diagnostics;
};

} // namespace kobuki

#endif /* KOBUKI_ROS_HPP_ */
```

그림 18 kobuki의 노드 관련 진단 로직

각 오류 복구 레벨 별 개선된 진단 시스템의 오류 검출 및 해결 시간이 달라지므로 실험에서는 1, 2, 3단계 세 경우에 대해 각각 측정하였다. 4단계는 3단계와 같이 해당 노드가 종료되므로 동일한 측정 조건으로 간주하고 생략하였다.

먼저, 노드에서 모니터링을 수행하는 진단 로직의 수를 증가시키면서 진단 로직의 마지막 수행 결과만 오류로 판단되는 경우를 만들었다. 진단 로직 수가 증가함에 따라 개선된 진단 시스템의 오류 검출 및 해결 시간 개선 효과가 어떻게 나타나는 지 보이기 위한 실험이다. 같은 조건에 대해 오류 복구 레벨을 1에서 3으로 각각 수행하면서 기존의 진단 시스템에서의 결과와 비교한 결과는 그림 19과 같다.

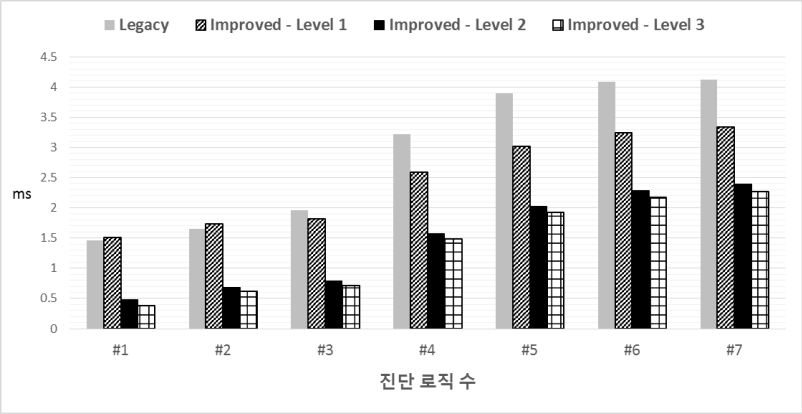

그림 19 진단 로직 수에 따른 오류 검출 및 해결 시간

진단 로직 수를 1부터 7까지 증가시키면서 마지막 진단 로직 결과가 오류인 경우의 오류 검출 및 해결 시간을 측정한 결과이다. 평균 오류 검출 및 해결 시간은 40.4% 단축되었음을 볼 수 있다. 각 오류 복구 레벨 별 결과가 다른 양상을 보이는데 그 분석은 다음과 같다. 전체 진단 결과 메시지 수와 오류 메시지의 수의 차이가 없을 때 즉,

진단 로직이 2개 이하일 경우 개선된 ROS 진단 시스템에서 오류 복구 레벨을 1단계로 선택하면 오류 메시지 벡터 생성에 의한 오버헤드에 의해 오히려 오류 검출 및 해결 시간이 증가함을 볼 수 있다. 하지만 진단 로직 수가 3 이상부터는 감소 폭이 커져 효과가 증대됨을 확인할 수 있었다. 오류 복구 레벨 2, 3단계는 노드 간 메시지 전송이 불필요하므로 모든 실험에서 확연한 시간 단축 효과를 보여준다.

두번째 실험은 진단 로직의 수행 결과 정상인 메시지의 비율에 따라 개선된 진단 시스템의 오류 검출 및 해결 시간 개선 효과에 대해 검증하였다. 그림 20은 진단 로직이 7개로 고정된 조건에서 결과가 오류가 되는 경우를 하나씩 증가시키면서 오류 검출 및 해결 시간을 측정한 결과이다.

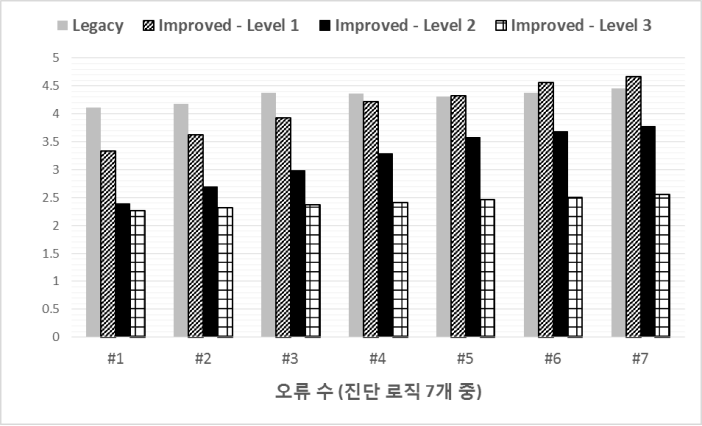

그림 20 오류 메시지 비율 증가에 따른 오류 검출 및 해결 시간

실험 결과 평균 오류 검출 및 해결 시간은 25.1% 단축되었다. 전체 진단 로직의 수보다 오류 메시지의 비율이 낮은 경우 즉 오류 수가 4개보다 더 작을수록 개선된 시스템의 오류 검출 및 해결 시간 개선효과가 높다는 것을 확인할 수 있었다. 특히 오류 복구 레벨 3단계는 오류 메시지 수에 상관없이 오류가 하나 이상 존재하면 바로

복구 로직이 수행되므로 오류 수 증가에 따라 오류 검출 및 해결시간 증가 비율이 매우 낮다. 진단 로직의 대부분의 결과가 오류인 경우는 오류 메시지 리스트 구성 및 발행의 오버헤드가 커져 응답시간이 더 길어지는 결과가 나왔지만 보통 프로그램 행위 모니터링 시 대부분 정상 동작임이 확인되고 일부 로직에 대해서만 오류가 검출되는 시나리오에서는 개선된 진단 시스템이 효과적일 것으로 기대된다.

이번에는 본 논문에서 제안하는 기법에 의해 생기는 오버헤드에 대해 살펴보자. 5장에서 소개한 개선된 진단 업데이터의 순서도에서 확인할 수 있듯이 기존의 로직인 전체 진단 결과 메시지를 /diagnostics 토픽으로 발행하는 시점이 오류 메시지를 선별하여 구성하고 발행한 후로 지연됨을 확인할 수 있다. 긴급한 오류 메시지에 대한 처리는 먼저 수행됨을 보장하나 진단 집합자, 로봇 모니터 등의 기존의 다른 노드들로의 전송이 지연되는 것이다. 이는 오류 복구 레벨 3, 4단계에서는 동작 중이던 해당 노드가 종료되므로 /diagnostics 토픽으로의 메시지 발행이 수행되지 않고 오류 복구 레벨 1, 2단계의 실험 결과만으로 판단한다. 임의의 진단 로직 수와 오류 수를 가지는 개선된 진단 시스템에서의 /diagnostics 토픽 메시지의 발행 시간이 평균 0.42ms 지연됨을 확인하였다. 기존의 로직 수행이 개선된 로직에 의해 지연되었으나 이는 오류를 선 처리하고 현재의 진단 로직 결과를 재분류하거나 단순히 시각화하여 보여주는 정보 제공 차원의 로직이 미뤄지는 결과를 야기하는 것이므로 시스템의 동작에는 영향을 끼치지 않는다고 볼 수 있다. 그보다 오류에 대한 처리를 우선 적용함으로써 자동차와 같은 안전 필수 시스템에 부합하는 모습으로 개선되었다고 판단된다.

제 7 장 결 론

본 논문에서는 자율주행 자동차의 소프트웨어 플랫폼으로 대두되고 있는 ROS에 대해 살펴 보았다. ROS는 본래 로봇 어플리케이션 개발을 위한 오픈소스 로봇 소프트웨어 플랫폼이다. 활성화된 오픈소스 커뮤니티와 학계 및 산업계에서 많은 연구를 진행하고 있고 발행-구독 통신 방식을 지원하기 때문에 많은 센서 데이터의 프로세싱 및 연산이 필요한 자율주행 자동차의 소프트웨어 플랫폼으로 적절하다.

한편 이와 같이 자동차 시스템 내에서 점차 증가하는 소프트웨어의 규모와 복잡도에 따라 소프트웨어 신뢰성 향상을 위해 런타임에서 주기적으로 프로그램의 오류를 검출하여 이를 복구하는 프로그램 행위 모니터링이 필수적이다.

ROS에서 프로그램 행위 모니터링을 제공하는 진단 시스템에는 두 가지 지연 요소가 존재한다. 본 논문에서는 비효율적인 메시지 처리 시간과 메시지 전송 지연을 줄이기 위해 오류 메시지 선별 처리와 직접 오류 처리 기법을 제안하였다. 개선된 ROS 진단 시스템은 진단 메시지 수집 과정에서 오류 메시지를 선별 처리함으로써 비효율적 메시지 처리 시간을 단축시켰고 진단 업데이터 내에 오류 복구 로직을 내장함으로써 메시지 전송 지연을 제거하였다. 단, 노드 간 의존성과 빠른 응답시간의 trade-off가 존재하는 직접 오류 처리 기법은 노드의 오류에 대한 취약성 정도에 따라 설정 가능하도록 오류 복구 레벨을 제공한다.

제안된 해결책은 우선 ROS에서 제안된 다른 기법들보다 빠른 오류 검출 및 해결시간을 가지며 모니터링 대상 노드에 진단 로직이 많이 추가될 수록, 진단 결과 정상 메시지의 비율이 오류의 비율보다

높을수록 큰 개선 효과를 보인다. 진단 로직 수가 증가함에 따라 최선의 경우 오류 검출 및 해결 시간이 평균 40.4% 단축되었고, 동일한 진단 로직 수에 대해 임의의 오류 메시지에 대한 오류 검출 및 해결 시간이 평균 25.1% 단축되었음을 보였다.

본 논문에서 제안하는 기법이 적용된 ROS 진단 시스템은 프로그램 행위 모니터링의 응답시간 개선을 가져오며 자동차 소프트웨어에 적용 시 엄격한 시간 제약사항 내에서의 오류 복구를 통해 신뢰성 향상에 기여할 수 있다고 기대한다.

참고 문헌

[1] “European Roadmap Smart Systems for Automated Driving”, *EPoSS*, 2015.

[2] “Global Test Sites and Incentive Programs for Automated Cars”, *Frost & Sullivan Market Research*, 2016.

[3] “Navigant Research Leaderboard Report: Autonomous Vehicle OEMs”, *Navigant Research*, 2015.

[4] M. Aeberhard et al., “Automated Driving With ROS at BMW”, *Roscon*, 2015.

[5] S. Kato et al., “An Open Approach to Autonomous Vehicles”, *IEEE Micro,* vol. 35, no. 6, pp. 60-68, 2015.

[6] M. Quigley et al., “ROS: an open-source Robot Operating System,” *IEEE International Conference on Robotics and Automation Open-Source Software workshop*, 2009.

[7] Aeberhard et al., “Experience, Results and Lessons Learned from Automated Driving on Germany‘s Highways”, *IEEE Intelligent Transportation Systems Magazine*, pp. 42-57, 2015.

[8] M. Broy, “Challenges in automotive software engineering”, *Proceedings of the 28th international conference on Software engineering*, pp. 33-42, 2006.

[9] A. Pretschner et al., “Software Engineering for Automotive Systems: A Roadmap”, *Future of Software Engineering*, 2007.

[10] K. Tindel et al., “Safe Automotive Software Development”, *Proceedings of the Design, Automation and Test in Europe*

Conference and Exhibition, 2003.

[11] V. Venkatasubramanian et al, "A review of process fault detection and diagnosis Part I: Quantitative model-based methods", *Computers and Chemical Engineering*, vol. 27, Issue 3, pp.293-311, 2003.

[12] X. Chen et al, "Application of Software Watchdog as a Dependability Software Service for Automotive Safety Relevant Systems", *Dependable Systems and Networks*, 2007.

[13] ROS Wiki, http://wiki.ros.org

[14] AUTOSAR, "Overview of Functional Safety Measures in AUTOSAR", http://www.autosar.org, 2015.

[15] AUTOSAR, "Specification of Watchdog Manager," http://www.autosar.org, 2015.

[16] S. Zaman et al., "An Integrated Model-Based Diagnosis and Repair Architecture for ROS-Based Robot Systems", *IEEE International Conference on Robotics and Automation*, 2013.

[17] N. Delgado et al., "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools", *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[18] N. Asadi et al., "Run-Time Monitoring of Timing Constraints: A Survey of Methods and Tools", *International Conference on Software Engineering Advances*, 2013.

[19] A. Mok et al., "Efficient Run-Time Monitoring of Timing Constraints", *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pp. 252-262, 1997.

[20] F. Chen et al., "A Formal Monitoring-based Framework for

Software Development and Analysis", *International Conference on Formal Engineering Methods*, pp. 357-372, 2004.

[21] D. Markart, "Implementing reliable Android applications", *Master's Thesis*, Graz University of Technology, 2014.

[22] V. Monajjemi et al., "Drums: a Middleware-Aware Distributed Robot Monitoring System", *Canadian Conference on Computer and Robot Vision*, 2014.

[23] Feng Chen et al., "Java-MOP: A Monitoring Oriented Programming Environment for Java", *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 546-550, 2005.

[24] J. Huang et al., "ROSRV: Runtime Verification for Robots", *Runtime Verification*, pp. 247-254, 2014.

[25] H. Zhu et al, "Predictable Runtime Monitoring", *Euromicro Conference on Real-Time Systems*, pp. 173-183, 2009.

[26] S. Zaman et al., tug_ist_model_based_diagnosis http://wiki.ros.org/tug_ist_model_based_diagnosis?distro=fuerte

[27] Formal Systems Laboratory, Illinois Univ., http://fsl.cs.illinois.edu/index.php/ROSMOP

Abstract

Reducing Error Handling Time of Diagnostics System in ROS

Wooin Choi

Department of Transdisciplinary Studies

Graduate School of Convergence Science and Technology

Seoul National University

In automotive electric & electronic systems, as a lot of the functionalities of automotive system are implemented with software, the software size is increasing and their complexity becomes higher than the past firmware-level source code. Especially development trends of autonomous vehicles move the attention to the software. Thus monitoring and understanding program behavior become mandatory to improve reliability of software in complicated systems. As robot developments are increasing, there have been huge needs of robot software platform which is reusable and hardware-independent. ROS (Robot Operating System) is a representative open source robot software platform. It provides efficient communications with publish/subscribe system and powerful development toolset. Over the robotics boundary, ROS is currently adapted to the autonomous driving software platform by the car manufactures.

ROS has diagnostics system providing mechanisms for program behavior monitoring. However ROS diagnostics system has two

limitations of response time, one is inefficient message handling time and the other is message transmission delay. Maintaining all the diagnostics message incurs inefficient message handling and message transmission between node of collecting diagnostics data and node of error handling incurs a delay of error handling.

This paper proposes error message preprocessing and direct error handling to solve the problem. To provide the choice of trade-off between reducing functional dependency and rapid error handling, this paper also proposes configurable error recovery level in the improved diagnostics system. The experimental results demonstrate the improved diagnostics system reduces error detection and reaction time up to 25.1% than existing diagnostics system.

Keywords : Robot OS, Fault Reaction Time, Diagnostics System, Error Handling, Runtime Software Monitoring

Student Number : 2015-26052