네트워크 임베디드 Self-adaptive 소프트웨어의 상호작용 비용에 기반한 구조 분석

김영필, 유혁
고려대학교 컴퓨터학과
ypkim@os.korea.ac.kr, hxy@os.korea.ac.kr

Interaction cost based software architectural analysis for networked embedded self-adaptive system

Young-Pil Kim, Chuck Yoo
Dept. of Computer Science & Engineering, Korea University

요 약

Self-adaptive 시스템에서 adaptation은 시스템 운영의 부담으로 작용할 수 있다. Self-adaptive system의 구조적인 고려는 이러한 부담을 줄일 수 있는 방안의 힌트로써 작용 할 수 있다. 본 논문에서는 self-adaptive 시스템을 구성할 때 가능한 소프트웨어 구조인 사용자 레벨 adaptation과 커널 레벨 adaptation을 살펴보고 각 구조에서 대상에 대한 adaptation을 수행할 때 발생하는 상호작용에 따른 비용을 추정하여 비교하였다. 특히 빠른 적응과 다양한 변화에 따른 적응이 요구되는 Networked Embedded Self-adaptive system을 대상으로 하였다. 본 논문은 Self-adaptive S/W 시스템을 설계할 때 기존의 기능적인 고려 외에 구조적인 고려가 필요함을 말하고자 하며 본 논문의 분석 결과는 Self-adaptive 시스템을 설계하고자 할 때 좋은 참고 자료가 될 수 있을 것이다.

1. 서론

Self-adaptive system은 시간 혹은 조건에 의해서 어떠한 형태로든 변화하는 시스템에 대해서 관찰(observation)하고 분석(analyzing)하여 어떠한 기준조건(standardization)을 만족하지 못하는 경우 시스템 스스로가 대상(target subsystem)을 조정(manipulation)하는 시스템이다. 이러한 Self-adaptive system은 다양한 대상들을 수용하기 위해서 여러 가지 형태로 존재한다. Self-adaptive system들로 쉽게 들 수 있는 예로써, 소프트웨어 오류나 결함에 대처하는 Self-healing system이나 비행에 필요한 항로 변화나 고도 변화 등 환경 요인에 대처하는 무인 항공기(unmanned air vehicle) 제어 시스템이나 주위 온도 변화에 대해 목표 온도 수치를 유지시켜주는 온도 조절(temperature adaptation) 시스템 등이 있다.

이러한 목적과 특징으로 인해 Self-adaptive system들은 장치 제어와 응용프로그램 제어 모두에 관여할 수 있다. 기존에 장치 제어와 응용프로그램 제어는 임베디드 시스템에서 전담하고 있기 때문에, Self-adaptive 소프트웨어는 임베디드 시스템과 많은 상호 작용을 할 수 밖에 없다.

비단 임베디드 시스템만이 Self-adaptation 기능이 요구되는 것은 아니나 본 논문에서는 임베디드 시스템에 adaptation 기능이 크게 요구됨에 초점을 맞추고 있다. 그 이유는 임베디드 시스템 구성과 기능이 다양해 지고 있다는 점과 기존 네트워크 인프라의 수용으로 인해 정보 접근성이 높다는 점에 있다. 정보 접근성이 높아짐으로 인해 제조사들에 의한 장치 구성에 따른 다양성 뿐 아니라 지원 소프트웨어 및 시스템 소프트웨어 기능 변화에 따른 다양성마저 가속화 시키고 있다. 다양성의 가속화는 필연적으로 변화를 수반하며 변화에 대한 대처를 위해서는 adaptation 능력이 요구된다.

본 논문에서 다루고자 하는 것은 Networked Embedded Self-adaptive System이다. Self-adaptation은 변화에 따른 관찰과 분석 조정을 시스템 스스로가 해야 하므로 이에 따른 오버헤드(overhead)가 발생한다. 임베디드 시스템의 특징상 자원 제약을 가지고 있으며, 때문에 그 어떤 시스템 보다 효율성이 요구된다. 본 논문은 이러한 adaptation 비용을 줄일 수 있는 여지가 구조적인 측면에 있다는 통찰을 가지고 서술되었고, 어떠한 구조들이 가능하며 상호작용의 측면에서 어떠한 구조가 보다 유리할 수 있는지를 살펴보았다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구를 소개한다. 3장에서는 본 논문의 주제인 Networked Embedded System의 기본적인 구조와 adaptation에 필요한 Self-adaptive S/W의 동작에 대해 정의하고 adaptation과 상호작용 에 대해서 2가지 구조 모델을 기반으로 분석한다. 마지막으로 4장에서 결론을 맺는다.

2. 관련연구

본 연구에서는 Self-adaptive system의 구조를 평가한다. 구조(architecture)라는 말은 여러 가지 의미로 다양한 분야에서 사용될 수 있는데, 본 논문에서

말하는 구조는 Self-adaptive system을 구성하고 있는 형태와 상호작용에 기인한 각 구성요소와의 관계를 일컫고 있다. Self-adaptive software의 구성요소를 어떻게 둘 것인가에 대한 연구는 오래 전부터 논의되어 왔다. [5]에서 self-adaptive software의 정의나 adaptation 관리의 행위(behavior)들이 정의되었다. adaptation의 대상에 따라 어떠한 구성요소가 추가적으로 더 요구되는가의 연구들도 존재한다. 예를 들면, [1]에서는 그 대상이 소프트웨어 폴트와 그에 따른 시스템 구성이다. Self-adaptation을 가능하게 하기 위한 동적인 소프트웨어 구조 명세에 대한 연구들[3]도 활발하게 진행되어 왔다. 이러한 연구들이 시사하는 바는 소프트웨어에 Self-adaptive한 속성을 부여하는 것이 실현되었다는 것이다. 때문에 현 연구 단계에서 더 요구되는 것은 활용에 요구되는 Self-adaptive S/W 자체에 대한 평가이다. Self-adaptive S/W의 평가에 관련된 연구[2]로 진행된 바 있으나 Stability나 Robustness등 기본적인 속성에만 국한하고 있다. 본 논문에서는 좀더 실제적인 평가를 시도한다는 점에서 의의가 있으며, 구조에 따라 실행 성능에 어떠한 영향을 끼칠 수 있는가를 살펴 보려 한다. Self-adaptive System에 대해서 이러한 시도는 아직 행해진 바가 없으며, 구조에 따른 성능상의 논의는 오히려 전통적인 운영체제 연구[4]에서 더 많이 나타난다. 본 연구는 전통적인 운영체제 연구에서 바라보는 구조적인 관점을 본 연구의 대상인 Networked embedded self-adaptive system에 적용하였다.

3. Networked Embedded Self-adaptive System 구조 분석

Networked Embedded Self-adaptive System(이후 NESS)의 구조적인 분석과 평가를 위해서 먼저 시스템의 기본적인 구성요소들을 정의할 필요가 있다.

3.1 Self-adaptive 구성요소가 없는 NESS의 기본 구조 정의

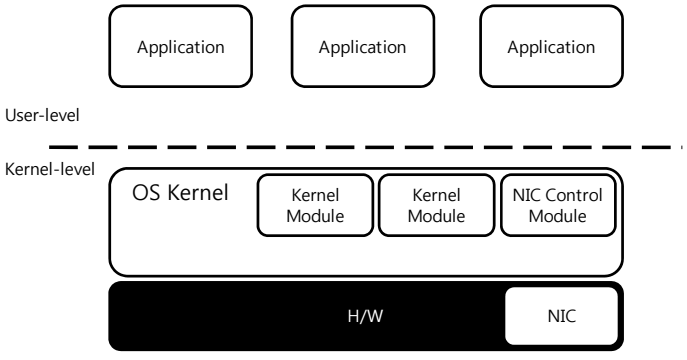

그림 1 Networked embedded system의 구조

그림 1은 NESS의 기본적인 소프트웨어 구조를 나타내고 있다. 기본적으로 본 논문에서 가정하고 있는 임베디드 시스템은 실행 모드에 있어 우선권이 하드웨어적으로 지원 가능한 시스템(kernel-level mode, user-level mode로 구분)을 대상으로 한다. 이는 운영체제 커널과 같은 관리 및 제어 소프트웨어들은 응용프로그램과는 다른 우선권을 가지고 좀 더 강력하고 특수한 명령어들을 수행 할 수 있다는 의미이다. 이러한 관점에서, 주요 구성요소에 대해서 각각 살펴보면 다음과 같다.

1) Application - 사용자 레벨에서 동작하는 응용프로그램은 adaptation의 목표가 될 수 있는 구성요소이다. Self-adaptive S/W가 관찰(observing)을 하거나 제어를 하는 대상(target)이다.

2) OS kernel - 특권 레벨인 커널 레벨에서 동작하는 운영체제 커널은 하드웨어의 관리와 응용프로그램의 관리 그리고 관련 자원 관리의 추상계층(abstraction)을 제공하고 있다. Linux를 포함하여 현대의 대부분의 커널들은 확장 가능성을 지원하므로 커널 모듈(kernel module)을 지원하는 커널을 가정한다. 이는 모듈화된 모놀리식 커널(monolithic kernel) 구조이다. 현대의 운영체제 커널 구조 중에는 운영체제 기능을 커널 레벨에서 동작시키지 않는 마이크로 커널 구조(micro kernel)도 존재한다. 그러나 임베디드 시스템에서는 자원제약으로 인해 다소 복잡한 마이크로 커널 구조보다 단순한 모놀리식 구조가 실제적으로 더 많으므로 본 논문에서는 마이크로 커널 형태는 배제하였다.

3) Kernel Module - 커널 모듈은 운영체제 기능의 일부를 구현하고 있으며 커널 레벨에서 동작하고 응용프로그램과는 격리된 커널 주소공간에서 동작한다. 대표적인 커널 모듈은 커널 레벨 디바이스 드라이버(kernel-level device driver)이며, 그림 1에서 NIC control module이 네트워크 장치를 제어할 수 있는 추상계층을 제공하는 장치에 해당한다. 커널 레벨 디바이스 드라이버로 언급한 이유는 마이크로 커널 구조에서는 사용자 레벨의 디바이스 드라이버 구조도 가능하기 때문이다. 본 논문에서는 커널 모듈 역시 adaptation의 대상으로 본다. 그 이유는 커널 모듈은 커널의 내부 데이터의 변화에 대응하여 동작하고 시스템 전체에 막강한 영향을 끼치므로 비정규적상황(anomaly)에 대처해야 할 필요성이 크기 때문이다. 상위 응응프로그램들과 하부의 장치에 치명적인 영향을 끼칠 수 있으므로 빠른 대처가 요구된다.

4) NIC - 하드웨어 구성요소 가운데 프로세서, 메모리

등의 기본적인 장치 외에 NIC(Network Interface Card) 구성요소가 기본 구성요소에 포함된다. 응용프로그램과 마찬가지로 NIC 역시 adaptation의 대상이 된다. 기본적인 통신에서 비롯되는 이벤트 처리 뿐 아니라, 네트워크 환경 변화를 감지하기 위한 관찰(observing), 그리고 이에 따른 성능 저하나 에러 회피를 위한 대역폭 조절 등의 조정(manipulation)이 가능하다.

3.2 Self-adaptive S/W 동작(behavior)의 정의

본 논문에서는 Self-adaptive software를 다음과 같은 동작(behavior)를 가지는 소프트웨어 구성요소로 정의하였다.

1) Observing
- adaptation의 대상으로부터 확보할 수 있는 정보를 수집하는 단계이며, 정보 수집의 원천이 어느 위치에 있는지에 따라 상호작용에 따른 비용이 달라진다. 본 논문에서는 기본적으로 observing 대상을 수정하지 않는 것을 가정하였다.

2) Analyzing
- observing 단계의 다음은 adaptation의 decision making 단계이다. 이러한 behavior는 self-adaptive component가 보유한 정보를 기반으로 computational하게 진행되므로 다른 구성요소들과의 부가적인 interaction은 발생하지 않는다.

3) Manipulation
- 이 단계에서는 adaptation의 결과를 반영하기 위해서, 대상에 직접 접근을 하던지 아니면, 운영체제 커널을 경유하여 대상에 영향을 끼칠 수 있는 서비스를 요청하게 된다.

그림 1에서 정의한 기본구조에서 adaptation의 대상이 될 수 있는 구성요소들은 application, kernel module, NIC이다. 이 중 NIC은 실제로 장치제어를 담당하는 부분인 NIC Control module에 의해서 제어 또는 관찰 된다. 본 논문에서 다루고자 하는 것은 Self-adaptive S/W에 의해서 adaptation이 수행될 때 각 대상들과 요구되는 연산 혹은 상호작용에 의해서 발생 할 수 있는 비용을 계산하여 가능한 구조들을 비교하는 것이다.

Self-adaptive S/W 역시 소프트웨어 구성요소이며 이 구성요소는 실행 우선권이 구분되어 있는 시스템의 경우 보호 영역(protection domain)을 기준으로 응용프로그램의 실행 권한을 가지는 사용자 레벨, 운영체제 커널 동작 시에 요구되는 커널 레벨에서 동작 가능하다. 사용자 레벨에서 동작할 때는 또 다른 응용프로그램의 형태로써 미들웨어(middleware)나 서버(server)의 형태로 구성 가능하며, 커널 레벨에서 동작할 때는 커널 모듈(kernel module) 중 하나의 형태로 제공될 수 있다.

3.3 Adaptation 대상에 따른 상호 작용 분석

이후 구조들에서 비용의 계산은 다음의 세 가지 adaptation 대상의 관점에서 Self-adaptive S/W behavior의 세 가지 단계를 수행하는 경우를 살펴본다. 여기서 Analyzing의 단계는 observing한 데이터의 위치에 의존하는 것으로 내부의 상호작용만을 요구하고 구조 비교에 따른 비용 관점에서는 미비하므로 제외하였다.

1) Application management의 관점
- adaptation의 대상이 응용프로그램인 경우 응용프로그램의 상태를 observing 하기 위해서는 그 내부 데이터에 직접 접근하던지 응용프로그램이 영향을 끼치는 데이터 변화를 추적(tracing)하여 간접적으로 유추 해야 한다. 커널 레벨에서는 원하는 응용프로그램이 메모리에 적재되어있다면 직접 접근하는 것이 가능하고, 사용자 레벨에서는 IPC를 이용하거나, 시스템이 제공하는 프로세스 상태 정보(예, Linux /proc)들을 이용해야 한다. 이때, 이에 필요한 충분한 권한 확보가 요구된다.
- Manipulation의 단계는 응용프로그램에 대한 제어가 수반되므로, 사용자 레벨에서 이를 수행하기 위해서는 운영체제 커널의 중재가 필요하다. 커널 레벨에서 역시 운영체제 커널이 지원하는 서비스를 이용해야 하며, 차이가 있다면 사용자 레벨에서는 보호 영역간의 전환 비용이 더 수반된다는 점이다.

2) Kernel Module management의 관점
- adaptation의 대상이 kernel module인 경우 module의 상태를 observing 하기 위해서는 module 내부 변수에 직접 접근 하던지, 아니면 module에서 노출시키는 데이터를 참조해야 한다. Self-adaptive S/W가 커널 레벨로 구현된 경우 대상 커널 모듈이 같은 커널 주소 공간 안이고 실행 레벨이 같으므로 접근에 무리가 없지만 사용자 레벨로 실행되는 경우에는 시스템 콜이나 메모리 공유 등의 특수한 방법이 요구되며 이는 일반적인 데이터 접근이나 제어 접근보다는 비용이 크다.
- Manipulation의 단계에서는 observing 단계 보다 더 제약이 따르는데, 커널 레벨로 실행되는 경우가 아니라면, 사용자 레벨에서 수행하는 adaptation은 운영체제 커널이 제어 서비스를 제공해야 하며, 이는 보호 영역 전환에 따른 오버헤드가 발생하게 된다.

3) Hardware device management의 관점

- 하드웨어 장치의 상태를 observing 하기 위해서는 그 하드웨어를 제어하고 있는 디바이스 드라이버 모듈의 도움을 받아야 한다. 이는 커널 레벨에서는 큰 문제가 되지 않지만, 사용자 레벨에서는 디바이스 드라이버에서 상태 정보 접근에 대한 인터페이스를 제공하지 않으면 원천적으로 불가능하다. 제공하는 경우(예, IOCtl 인터페이스)에는 디바이스 메모리나 레지스터에 또는 상태 플래그(status flag)에 접근하여 데이터를 확보할 수 있다. 장치와의 통신은 event-driven 방식의 인터럽트 기반의 방법과 polling 방식의 PIO 방법을 이용할 수 있다.
- Manipulation의 단계에서는 역시 하드웨어 제어를 담당하는 디바이스 드라이버 모듈을 거쳐야 한다. 커널 레벨에서는 직접 호출을 하여 가능하지만, 사용자 레벨에서는 I/O 요청 처리를 수행해야 하므로 운영체제 커널 서비스의 도움을 받아야 하며, 이로 인한 추가적인 부담이 부가된다.

3.4 상호 작용에 영향을 끼치는 요소 분석

1) 함수 호출 및 반환 (function call C_F/return R_F)
- 같은 도메인 안에 존재하는 함수에 대한 호출 및 반환. 파라미터 및 복귀 주소 처리 등의 기본적인 연산에 따른 비용이 수반된다. 일반적인 함수 호출에 해당한다.

2) 보호 영역 호출 및 반환 (protected domain call C_D/return R_D)
- 서로 다른 도메인 안에 존재하는 함수에 대한 호출 및 반환. 함수 호출에 따른 기본 비용 뿐 아니라, 도메인 전환 메커니즘과 파라미터 전달 방법 그리고 기본적인 검증 및 확인 작업, 필요한 경우 하드웨어적인 도메인 플러싱(domain flushing) 연산이 필요하다. SysCall(system call), IPC(inter-process call) 또는 RPC(remote procedure call) 또는 IDC(inter-domain call)등에 해당한다.

3) 인터럽트 호출 및 반환(interrupt call C_I/return R_I)
- 인터럽트 호출은 프로세서의 IRQ 핀(interrupt request pin)에 전기적인 신호에 의해서 인터럽트 핸들러가 호출되는 것을 말하며, 그 원천은 타이머의 주기적인 이벤트 혹은 장치의 비주기적인 이벤트 등이 될 수 있다. 반환은 인터럽트의 처리가 끝나 인터럽트가 발생하여 context switching 된 context로 restoring을 하는 것을 의미한다.

각각에 대한 개별적인 수치는 시스템 및 하드웨어 마다 차이가 있으나 일반적으로 값의 상대적인 차이는 나타낼 수 있다. 본 논문에서 상호작용 요인에 대한 비용은 연산을 수행하기 위해 소모되는 처리 시간(processing time)으로 간주한다. 이 관점에서 상대적인 크기 차이를 기술하면 다음과 같은 관계로 표현할 수 있다.

$$Calling\ relative\ cost:\ C_F < C_I << C_D$$
$$Returning\ relative\ cost:\ R_F < R_I << R_D$$
$$All\ C_X >= 0$$

이러한 상관관계의 근거는 다음과 같다. 먼저, 임베디드 장비에서 많이 사용하는 ARM과 같은 RISC 구조를 가정하여 명령어 수행 시간의 차이가 없다고 가정한다. 또한 동일한 수의 파라미터의 처리를 가정한다. 이때, 기본적인 함수 호출에는 복귀 주소의 저장과 분기 명령(call or branch) 수행 시간(*call_time*)이 걸린다. 인터럽트 호출에는 기본 함수 호출에 따른 복귀 주소 처리뿐 아니라, 예외 혹은 IRQ 모드로 전환 해야 하므로 특수 레지스터(예, ARM의 CPSR등)의 저장 및 복원 등의 처리 시간(*exception_time*)이 더해진다. 보호 영역의 호출에는 기본적으로 모드 전환이 필요하고, 부가적인 연산 (IPC의 경우 message 가공, RPC의 경우 marshalling 등)(*extra_time*)이 더 소모된다. 때문에 처리시간에 따른 상대적인 비용차이가 생긴다. 이를 정리하여 CF와 CI 그리고 CD의 상관관계를 좀 더 자세히 표현하면 다음과 같다.

$$CF = call\_time$$
$$CI = call\_time + exception\_time$$
$$CD = call\_time + exception\_time + extra\_time$$

이러한 상관관계는 이후 모델 분석에서 이용될 것이다. 또한, 같은 연산에 대한 비용계산의 관점에서는 호출(Calling)과 반환(Returning)의 구분이 무의미 하므로(같은 상관관계를 보이기 때문) 이후의 절에서는 호출 연산만을 기준으로 설명한다.

3.5 구조 모델 분석

NESS의 소프트웨어 구조는 보호 영역을 경계로 한 Self-adaptive 소프트웨어 구성요소의 위치에 따라서 사용자 레벨, 커널 레벨의 두 가지의 소프트웨어 구조 모델이 가능하다. 이후의 절에서는 adaptation의 대상(application, kernel module, hardware device)에 대해서 adaptation behavior를 수행할 때 상호작용에 따른 비용에 대해 추정해 본다. 여기서 Observation은 주기적으로 여러 번 수행해야 하지만, 비교를 위해서 각 대상에 대해서 one-shot으로 한번만 수행하는 경우를 가정하였다. 또한 adaptation에 제한이 없도록 권한은 최대로 가지고 있음을 가정한다.

1) 사용자 레벨의 Self-Adaptive S/W 구조

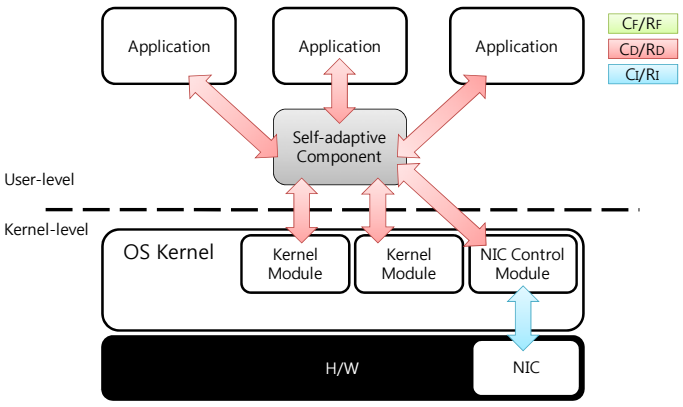

그림 2 사용자 레벨 adaptation 구조와 상호작용

그림 2는 응용프로그램 레벨의 미들웨어(middleware) 혹은 서버(server)로 Self-adaption S/W를 구현하는 경우에 adaptation 대상과의 상호작용을 도식한 것이다. 사용자 레벨에서 adaptation을 수행하게 되면, adaptation 기능의 확장이나 수정에 유연성을 줄 수 있고, 시스템 자원 소모 면에서 이득이 있으며 잘못된 adaptation으로 인한 영향력을 최소화 할 수 있다. 이러한 구조에서 adaptation을 위한 두 가지 연산(Observing과 Manipulation)에 3.4에서 살펴본 interaction flow의 영향 요소로 분석하면 다음과 같다.

a) *Observing cost = {C_D }+ {C_D }+ {(((C_D+C_I) OR C_D)}*
- 응용프로그램: 라이브러리 형태의 미들웨어로 adaptation을 구현한다면, API를 경유하여 수집할 수 있다. 때문에 응용프로그램에 대한 비용은 C_F가 된다. 하지만 이는 응용프로그램의 수정(API를 써서 재작성)을 요구한다. 본 논문에서는 수정을 가정하지 않았으므로 개별적인 서버형태의 응용프로그램으로 adaptation을 수행한다면, 응용프로그램 상태 확보를 위한 접근 비용이 소모된다. 상태 정보를 파일이나 가상자원(Linux의 /proc 정보) 등의 어떠한 시스템 자원이라고 하더라도 응용프로그램에서는 보호영역 전환이 필요하다 때문에 그 비용은 C_D이다.
- 커널 모듈: 응용프로그램에서 커널 모듈에 대한 정보를 직접 접근할 수 있는 방법은 없다. 제한적인 방법으로, 커널 모듈이 명시적으로 시스템 자원에 기록하는 것을 읽어와야 한다. 응용레벨에서 자원 접근이므로 그 비용은 C_D이다.
- 하드웨어 장치: 장치에 대한 직접 접근은 장치 드라이버가 개별적인 인터페이스 제공하는 경우(Linux의 IOCTL 인터페이스 등)에 한해서 가능하다. 이 경우에는 장치 드라이버에 대한 접근 비용 C_D와 장치에서 정보를 읽어오는 비용 C_I 이 요구된다. 또는 운영체제 자체에서 장치에 대한 제한적인 정보들을 시스템 자원으로 기록해 놓는 경우에는 C_D의 비용이 소모된다.

b) *Manipulation cost = {2*C_D }+ {C_D + C_F}+ {C_D+C_I }*
- 응용프로그램: 일반적으로 응용프로그램에 대한 직접적인 제어를 다른 응용프로그램 할 수 는 없다. Sleep이나 Kill등의 스케쥴링 등을 위해서는 시그널링(signaling)등의 시스템 기능을 이용해야 하며, 내부 변수의 직접적인 조작은 어렵다. 이를 기반으로 시스템 기능 요청에 따른 비용과 시스템에 의한 조작 비용은 모두 보호영역의 전환이 요구되므로 최소 2C_D의 비용이 소모된다.
- 커널 모듈: 커널 모듈에 대한 접근은 응용레벨에서는 시스템 기능에 대한 요청이 부가되어야 한다. 더불어 시스템 내부의 접근 비용이 포함되어야 하므로, CD+CF의 비용이 소모된다.
- 하드웨어 장치: 장치 상태를 직접 변경하는 경우는 Observing에서 장치 드라이버를 이용하여 레지스터에 접근하는 경우와 같다고 볼 수 있다. 때문에 그 비용은 C_D+C_I이다.

2) 커널 레벨의 Self-adaptive S/W 구조

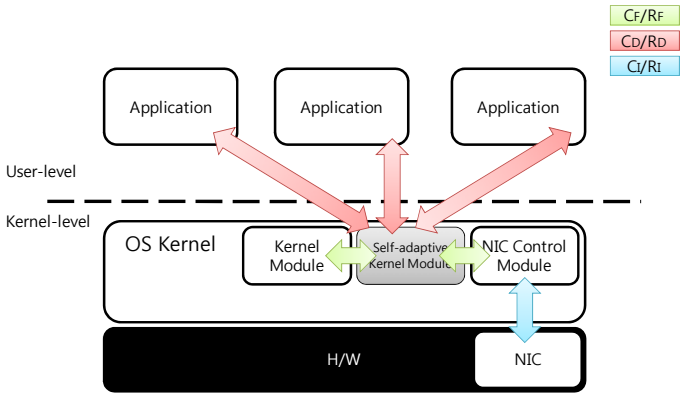

그림 3 커널 레벨의 adaptation 구조와 상호작용

그림 3은 커널 레벨의 컴포넌트에서 adaptation을 수행하는 경우를 보이고 있다. 커널 주소 공간 안에 직접 적재되므로 adaptation 기능에 대한 신뢰가 가정되어야 하며, 사용자 응용과는 달리 스왑되지도 못하므로 자원 부담이 크다. 대체적으로 사용자 레벨의 경우와 비슷하지만, 최소한 커널 모듈과 장치에의 접근 비용에 유리한 면이 있다. 이를 분석하면 다음과 같다.

a) *Observation cost = {C_F}+{C_F}+{C_F+C_I}*
- 응용프로그램: 커널 레벨에서 응용프로그램에 대한 접근은 기본적인 제어 뿐 아니라 내부 변수까지도 접근할 수 있다. 이는 데이터에 대한 접근이고, 커널 레벨이 사용자 레벨보다 높은 우선권을 가지고 있으므로 이러한 접근에는 보호 영역 전환이 필요하지 않다. 데이터 접근에 커널 인터페이스를 이용하게 되면 그 비용은 C_F가 된다.

- 커널 모듈: 동일한 도메인의 접근이므로 그 비용은 C_F이다.
- 하드웨어 장치: 장치에 대해 직접적인 접근이 가능하지만 이는 추상 계층을 파괴하는 것이므로 커널 내 인터페이스를 사용하는 경우를 장치 인터페이스의 접근 C_F와 장치 접근 C_I의 비용이 소모된다.

b) *Manipulation cost = {C_F}+{C_F}+{C_F+C_I}*
- 응용프로그램과 커널 모듈 그리고 하드웨어 장치에 대한 Manipulation에 필요한 상호작용에서 사용자 레벨과의 차이는 시스템 요청에 따른 비용이 불필요하다는 것이다. 때문에, 사용자 레벨의 Manipulation cost에서 시스템 요청 비용을 제거한 나머지가 커널 레벨에서의 Manipulation cost가 된다. 여기에 더불어 커널 레벨에서는 응용프로그램에 대한 접근 비용이 C_F가 되고, 장치 드라이버에 대한 접근도 C_F가 되므로 총 비용은 Observation 비용과 같게 된다..

3) 사용자 레벨과 커널 레벨 adaptation의 비교
비교를 위해서 3.4에서 분석한 상관관계를 이용하여 각 비용을 표현하면 다음과 같다.

a) Observation cost 비교
- *사용자 레벨 = {C_D }+ {C_D }+ {((CD+CI) OR CD)} = 4*call_time+4*exception_time+3*extra_time OR 4*call_time + 4*exception_time + 4*extra_time*
- *커널 레벨 = {C_F}+{C_F}+{C_F+C_I} = 5*call_time+exception_time*
- *사용자 레벨 - 커널 레벨 = - call_time+3*exception_time+3*extra_time OR - call_time+3*exception_time+4*extra_time*

여기서, 함수 분기 비용인 call_time의 경우 레지스터 저장 및 복원 연산 비용인 exception_time이나 메시지 가공 비용 등의 extra_time보다 작거나 최소한 같기 때문에 위 식은 늘 0보다 크므로, observation cost의 면에서 adaptation의 구조적인 위치는 사용자 레벨이 커널 레벨보다 상호작용에 따른 비용이 더 크다고 말할 수 있다. 또한 분석 결과에서 사용자 레벨의 경우 각 상호작용 연산 비용이 고르게 분포되는 반면, 커널 레벨의 경우에는 함수 호출 비용에 집중됨을 알 수 있다.

b) Manipulation cost 비교
- *사용자 레벨 = {2*C_D }+ {C_D + C_F}+ {C_D+C_I } = 6*call_time+5*exception_time+4*extra_time*
- *커널 레벨 = {C_F}+{C_F}+{C_F+C_I} = 5*call_time+exception_time*
- *사용자 레벨 - 커널 레벨 = call_time+4*exception_time+4*extra_time*

Observation cost와 동일한 방식으로 추정해보면, Manipulation cost역시 사용자 레벨이 커널 레벨보다 크다.

4. 결론

본 논문에서는 Self-adaptive 시스템을 구성할 때 adaptation을 담당하는 소프트웨어 구성요소와 대상간의 상호작용 비용이 소프트웨어 구조에 따라서 어떠한 차이를 보이는가를 다루고 있다. 구조적인 비교에서 사용자 레벨 기반의 adaptation구조와 커널 레벨 기반의 adaptation 구조는 그 장단점이 있으나 상호작용의 측면에서는 커널 레벨의 구조가 유리함을 알 수 있다. 커널 레벨에 기능을 넣는 것은 시스템 자원을 소모하고 그만큼 응용프로그램에 가용한 자원이 소모되는 위험부담을 가진다. 하지만, adaptation에 필요한 상호작용인 Observation과 Manipulation에서의 비용차이는 간과할 수 없다. 특히, Observation 연산이 주기를 가지고 반복적으로 수행되므로 그 차이는 실제로는 훨씬 더 커질 것으로 예상된다.
본 논문은 Self-adaptive S/W 시스템을 설계할 때 기존의 기능적인 고려 외에 구조적인 고려가 필요함을 말하고자 하며 본 논문의 분석 결과는 Self-adaptive 시스템을 설계하고자 할 때 좋은 참고 자료가 될 수 있을 것이다.

참고문헌

[1] Rogerio de Lemos and Jose Luiz Fiaderio, “An Architectural Support for Self-adaptive Software for Treating Faults”, In proceedings of the international Workshop on Self-healing Systems 2002.
[2] Alex C. Meng, “On Evaluating Self-Adaptive Software”, In proceedings of the International Workshop of Self-Adaptive Software 2000.
[3] Jeremy S. Bradbury, James R. Cordy, Juergen Digel, Michel Wermelinger, “A Survey of Self-Management in Dynamic Software Architecture Specifications”, In proceedings of the International Workshop on Self-Managed Systems 2004.
[4] G. Denys, F. Piessens and F.Matthijs, “A Survey of Customizability in Operating Systems Research”, ACM Computing Surveys Vol. 34, No. 4, December 2002, pp. 450-468.
[5] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum and Alexander L. Wolf, “An Architecture-Based Approach to Self-Adaptive Software”, IEEE Intelligent Systems and Their Applications, Vol. 14, Issue 3, May-June 1999, pp. 54-62.