2006 한국컴퓨터종합학술대회 논문집 Vol. 33, No. 1(A)

효율적인 컴포넌트 실행모델을 위한 RSCA의 확장

홍두원, 이재수, 김세화, 홍성수
서울대학교 전기컴퓨터공학부
{dwhong, jslee, ksaehwa, sshong}@redwood.snu.ac.kr

Extending RSCA for Efficient Component Execution Model

Duwon Hong, Jaesoo Lee, Saehwa Kim, Seongsoo Hong
School of Electrical Engineering and Computer Science, Seoul National University

요 약

Robot Software Communication Architecture는 URC 로봇을 위해서 제안된 표준 시스템 소프트웨어 구조이다. RSCA용 응용 프로그램은 다수의 컴포넌트의 결합으로 구성되는데 RSCA의 기반이 된 SCA에서는 컴포넌트를 프로세스로 실행할 것인지 쓰레드로 실행할 것인지에 대해서 명확하게 정의하고 있지 않다. 만약 각각의 컴포넌트의 특성을 고려하여 어떻게 실행할 것인지에 대한 부가적인 정보를 기술하고 이 정보에 따라서 컴포넌트를 실행하게 된다면, 보다 효율적으로 응용 프로그램을 실행할 수 있게 된다. 본 논문에서는 RSCA를 확장하여 컴포넌트의 실행에 대한 정보를 기술하는 방법 및 기술된 정보에 따라서 컴포넌트를 실행하기 위한 방법을 제시한다.

1. 서론

RSCA (Robot Software Communication Architecture)는 URC (Ubiquitous Robotics Companion) 로봇을 위한 표준 시스템 소프트웨어 구조로써 이기종 분산 처리, 동적 시스템 재구성, 서비스 품질 및 실시간성 보장과 같은 URC 로봇의 응용 특성을 지원하는 것을 목적으로 한다[1]. RSCA는 SDR (Software Defined Radio)을 위해서 JTRS (Joint Tactical Radio System)에서 제안한 SCA (Software Communication Architecture)[2]를 기반으로 한다. RSCA는 SCA에서 불필요한 부분을 제거하고 URC 로봇을 위해 필요한 QoS와 이벤트 서비스 기능들을 추가한 것이다[3].

본 연구실에서는 RSCA의 타당성을 검증하기 위해서 Evolution Robotics[4]의 Scorpion 로봇을 위한 응용 프로그램을 작성하였다. 그리고 작성된 응용 프로그램을 로봇에서 실행하여 로봇이 잘 작동함을 확인할 수 있었다. 그런데 RSCA 응용 프로그램을 작성하고 실행하는 과정에서 RSCA의 제약점을 발견하게 되었다. RSCA 응용 프로그램은 하나 이상의 컴포넌트의 집합으로 구성되며, 각 컴포넌트 간에는 코바 통신을 이용하여 데이터를 주고 받는다. 응용 프로그램이 실행될 때, 하나의 컴포넌트는 하나의 프로세스 또는 쓰레드로 실행된다. 다수의 컴포넌트로 구성된 복잡한 응용 프로그램이 실행되는 경우는 컴포넌트의 개수에 해당하는 만큼의 프로세스 또는 쓰레드가 생성된다. 그러나 기존의 RSCA는 컴포넌트를 프로세스로 실행할지 쓰레드로 실행할지에 대해서 명확하게 정의하고 있지 않다. 또한 RSCA의 기반이 되었던 SCA 의 가장 최근의 표준[5]에서도 이것에 대해서는 언급하고 있지 않다.

응용 프로그램 컴포넌트의 특성을 고려하여 쓰레드 또는 프로세스로 실행할 수 있다면, 보다 적은 메모리를 소모하고 문맥교환시간이 짧아질 수 있다. 또한 컴포넌트간의 코바 통신의 비용이 감소할 수 있다. 본 논문에서는 이와 같은 효율적인 컴포넌트 실행을 위한 RSCA 확장 설계를 제안한다.

2. RSCA 코어 프레임워크 및 응용 프로그램

RSCA의 핵심 부분인 코어 프레임워크는 RSCA 응용 프로그램에게 운영체제와 하드웨어에 대하여 추상화된 기능을 제공하는 인터페이스들로 구성된다.

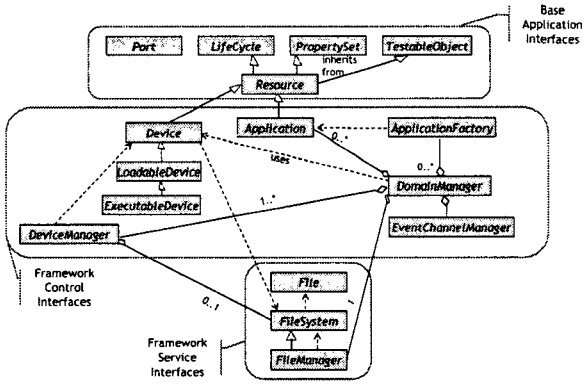

그림 1 코어 프레임워크 인터페이스

그림 1에서 알 수 있듯이, 코어 프레임워크는 크게 기본 컴포넌트 인터페이스와 프레임 제어 인터페이스, 프레임 서비스 인터페이스로 구성된다. 기본 컴포넌트 인터페이스들은 응용 프로그램의 컴포넌트들이 구현해야 하는 인터페이스를 명시한다. 프레임 제어 인터페이스는 도메인 내의 제어를 위한 것으로 도메인 관리, 장치, 장치 관리 인터페이스가 있다. 이 인터페이스들은 도메인 내의 응용 프로그램 및 하드웨어 장치의 설치 및 해제와 관리를 위한 인터페이스를 명시한다. 프레임 서비스 인터페이스는 응용 프로그램 및 코어 프레임워크가 공통으로 사용하는 파일 시스템과 로그 시스템에 대한 인터페이스를 정의한다.

코어 프레임워크에서는 도메인 프로파일이란 XML 디스크립터를 정의하였다. 이는 로봇 시스템의 하드웨어와 소프트웨어의 구성 정보를 기술하는데 사용된다. 또한 도메인 프로파일은 새로운 소프트웨어를 설치할 때 컴포넌트들의 조합에 대한 설치 정보를 기술하는 부가 정보로써 사용될 수 있다.

RSCA용 응용 프로그램은 응용 프로그램을 구성하는 컴포

닌트들의 실행파일과 각각의 컴포넌트들에 대한 정보를 기술한 도메인 프로파일을 압축한 형태의 패키지로 이루어진다. 각 컴포넌트는 전술한 코어 프레임워크의 기본 컴포넌트 인터페이스를 제공하도록 구현해야 하며 컴파일 된 실행파일의 형식은 executable 또는 shared library 형식이어야 한다.

3. 기존의 컴포넌트 실행 방법 및 한계점

본 장에서는 기존의 RSCA에서 응용 프로그램의 컴포넌트 배치 및 실행이 이루어지는 과정에 대해서 살펴본 후 기존의 방법의 한계점을 설명한다.

3.1. 기존의 컴포넌트 배치 및 실행 과정

도메인 프로파일은 응용 프로그램에 대한 정보를 기술한다. 도메인 프로파일 중 SAD 파일은 응용 프로그램이 어떻게 구성되었는지에 대한 전반적인 내용을 포함하고 있다. SAD 파일의 형식은 softwareassembly.2.1.dtd에서 정의하고 있다. 이 파일에 따르면 SAD는 도메인의 관리를 위해서 필요한 응용 프로그램의 기본적인 정보들을 제공해야 한다. DTD 파일 중에서 컴포넌트의 배치 및 실행과 연관이 있는 *partitioning* 부분은 다음 그림 2와 같다.

```
<!ELEMENT partitioning
        (componentplacement
        | hostcollocation
        )*>
<!ELEMENT componentplacement
        ( componentfileref
        , componentinstantiation+
        )>
<!ELEMENT hostcollocation
        ( componentplacement
        )+>
<!ATTLIST hostcollocation
        id ID #IMPLIED
        name CDATA #IMPLIED>
```

그림 2 softwareassembly.2.1.dtd 의 *partitioning* 부분

Partitioning 요소는 *componentplacement* 또는 *hostcollocation* 요소를 자식 요소로 가질 수 있다. *hostcollocation* 요소는 하나 이상의 *componentplacement* 요소를 자식 요소로 가질 수 있다. *componentplacement* 요소는 특정 컴포넌트의 정보를 제공하는 파일의 위치와 실행 시의 인스턴스 이름을 포함하고 있다. *hostcollocation* 요소는 동일한 실행장치(CPU)에서 실행되어야 할 컴포넌트들을 명시하는데 사용된다. 하나 이상의 *componentplacement*가 *hostcollocation* 요소의 자식으로서 존재한다면 이들 컴포넌트는 동일한 실행장치에 실행 되어야 한다는 의미이다.

응용프로그램은 *ApplicationFactory* 의 *create()* 함수를 통해서 생성된다. *create()* 함수 내부에서는 응용 프로그램의 SAD 파일의 컴포넌트 배치 정보를 분석하여 이 정보를 바탕으로 *ExecutableDevice* 의 *execute()* 함수를 호출한다. *execute()* 함수의 정의는 다음과 같다.

```
CF::ExecutableDevice::ProcessID_Type ExecutableDevice_impl::execute(
                                const char *filename,
                                const CF::Properties &options,
                                const CF::Properties &parameters)
```

그림 3 *ExecutableDevice* 의 *execute()* 함수의 정의

ExecutableDevice 는 *execute()* 함수에 전달된 컴포넌트의 실행파일이름과 옵션 및 파라미터 정보를 이용하여 새로운 프로세스 또는 쓰레드를 생성하여 컴포넌트를 실행하게 된다.

3.2. 기존의 컴포넌트 실행의 한계점

앞 절에서 설명한 것처럼 기존의 SAD 파일에는 동일한 호스트 머신에서 실행되어야 할 컴포넌트들에 대한 정보를 기술할 수가 있다. 그러나 더 이상의 응용 프로그램 컴포넌트의 특성을 고려한 실행 방법을 표현할 수가 없다. 컴포넌트의 실행방법은 컴포넌트의 특성과는 상관 없이 *ExecutableDevice* 의 *execute()* 함수를 어떻게 구현하느냐에 따라서 결정되는데 이로 인해서 효율적인 컴포넌트 실행을 하지 못하게 된다.

ExecutableDevice 의 *execute()* 함수를 구현하는 방법은 여러 가지가 있다. 그러나 기존의 *execute()* 함수를 구현하여 컴포넌트를 실행하는 데에는 효율성과 안전성에 한계가 있다. 직관적인 *execute()* 함수의 구현방법은 *fork()*와 *exec()*를 통해 컴포넌트를 프로세스로 실행하는 것이다. 그러나 모든 컴포넌트를 프로세스로 실행하는 것은 시스템에 과도한 오버헤드를 초래하는 효율적이지 못한 방법이다. 컴포넌트의 실행파일이 공유 라이브러리인 경우 효율적인 응용 프로그램 실행을 목적으로 컴포넌트를 쓰레드로 실행하도록 구현할 수도 있다. 이 때, 쓰레드는 *ExecutableDevice* 가 동작하는 프로세스와 주소영역을 공유하게 되는데 이것은 시스템의 안전성을 저해하는 요인이 된다. 왜냐하면 컴포넌트가 쓰레드로 실행되다가 문제를 일으켰을 경우 *ExecutableDevice* 도 잘못 동작할 가능성이 있기 때문이다.

4. 새로운 컴포넌트 실행 모델 및 RSCA 의 확장

본 절에서는 기존의 컴포넌트 실행방법이 가지는 제약점을 보완하는 새로운 컴포넌트 실행 모델을 제시하고, 새로운 실행 모델을 위한 RSCA 확장 방법을 제시한다.

4.1. SAD 의 확장을 통한 새로운 컴포넌트 실행 모델

먼저 컴포넌트를 실행하는데 유연성을 부여하기 위해서 SAD 파일을 확장하여 각 컴포넌트를 쓰레드로 실행할지 프로세스로 실행할지에 대한 정보를 파일에 기술 하도록 했다. 즉, 각각의 컴포넌트를 실행하는 방법이 컴파일 타임에 결정되는 것이 아니라 런타임에 SAD 에 기술된 정보에 의해서 정해질 수 있게 하여, 유연성을 제공한다. 이 목적을 위해서 기존의 *partitioning* 요소에 *processcollocation* 요소를 추가하여 다음의 그림 4 와 같이 확장한다. 음영 처리된 부분이 추가된 부분이다.

```
<!ELEMENT partitioning
        (componentplacement
        | hostcollocation
        | processcollocation
        )*>
<!ELEMENT componentplacement
        ( componentfileref
        , componentinstantiation+
        )>
<!ELEMENT hostcollocation
        ( componentplacement
        | processcollocation
        )+>
<!ATTLIST hostcollocation
        id ID #IMPLIED
        name CDATA #IMPLIED>
<!ELEMENT processcollocation
        ( componentplacement
        )+>
<!ATTLIST processcollocation
        id ID #IMPLIED
        name CDATA #IMPLIED>
```

그림 4 확장된 SAD의 DTD 파일

processcollocation 요소를 통해서, 이 요소의 자식인 *componentplacement*에서 정의된 컴포넌트들은 동일한 프로세스 안에서 쓰레드로 실행되어야 하는 제약조건을 표현할 수 있게 되었다. 또한 *hostcollocation* 요소에도 *processcollocation* 요소를 추가하여 동일한 호스트 머신 위에서 생성되어야 할 프로세스 제약조건을 기술할 수 있도록 한다.

예를 들어 A, B, C, D, E, F, G의 일곱 개의 컴포넌트로 구성된 응용 프로그램이 있다고 가정을 해보자. 다음 그림 5와 같이 컴포넌트 (A, B, C)와 (D, E)를 동일한 호스트 머신에서 각각 프로세스로 생성하고, 컴포넌트 (F, G)를 제약 조건이 없이 프로세스로 생성하려고 할 때, 이에 대한 XML 파일은 다음 그림 6과 같다. 단, *componentplacement*에서 정의하고 있는 각 컴포넌트에 대한 자세한 내용은 생략한다.

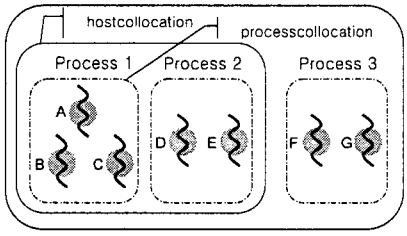

그림 5 컴포넌트의 배치 제약 조건

```
<partitioning>
    <hostcollocation>
        <processcollocation>
            <componentplacement>
                <!--.. A 컴포넌트에 대한 내용  -->
            </componentplacement>
            <componentplacement>
                <!--.. B 컴포넌트에 대한 내용  -->
            </componentplacement>
            <componentplacement>
                <!--.. C 컴포넌트에 대한 내용  -->
            </componentplacement>
        </processcollocation>
        <processcollocation>
            <componentplacement>
                <!--.. D 컴포넌트에 대한 내용 -->
            </componentplacement>
            <componentplacement>
                <!--.. E 컴포넌트에 대한 내용  -->
            </componentplacement>
        </processcollocation>
    </hostcollocation>
    <processcollocation>
        <componentplacement>
            <!--.. F 컴포넌트에 대한 내용  -->  ...
        </componentplacement>
        <componentplacement>
            <!--.. G 컴포넌트에 대한 내용 -->
        </componentplacement>
    </processcollocation>
</partitioning>
```

그림 6 확장된 예시 SAD 파일

4.2. *ExecutableDevice* 의 확장

다음으로 시스템의 안전성을 보장하며 SAD 기술된 제약사항을 만족시키는 컴포넌트 실행을 위해서 *ExecutableDevice*의 *execute()* 함수를 확장한 *threadexecute()* 함수를 다음 그림 7과 같이 선언한다.

```
CF::ExecutableDevice::ProcessID_TypeExecutableDevice_imple::threadexecute(
                                vector<string> FileNames,
                                vector<CF::Properties> options,
                                vector<CF::Properties> paramenters)
```

그림 7 새로 선언한 *threadexecute()* 함수

threadexecute() 함수는 하나 이상의 컴포넌트에 대한 파일 이름과, 옵션, 파라미터에 대한 정보를 전달 받는다. 이 함수는 먼저 *fork()* 함수를 통해서 새로운 프로세스를 생성하고, 생성된 프로세스 안에서 *threadexecute()* 함수의 매개변수를 이용하여 각 컴포넌트에 해당하는 새로운 쓰레드를 생성 한다. 이것을 통해서 SAD 파일에서 정의 하고 있는 대로 컴포넌트들을 쓰레드로 실행 할 수 있게 한다. *threadexecute()* 함수를 통해서 생성된 프로세스와 쓰레드는 *ExecutableDevice*와 서로 다른 프로세스로 동작하게 된다. 그래서 응용 프로그램 컴포넌트의 실행 중에 문제가 발생할지라도 *ExecutableDevice*에는 영향을 미치지 않게 되었으므로, 시스템의 안전성이 보장되게 되었다.

4.3. *ApplicationFactory* 의 확장

SAD 파일에 기술된 *processcollocation* 제약사항을 위해서 4.2절에서 선언한 함수를 사용하기 위해서는 *ApplicationFactory*의 확장이 필요하다. 기존의 RSCA에서는 응용 프로그램을 실행할 때, *ApplicationFactory*의 *create()* 함수에서 SAD 파일의 컴포넌트 배치 정보를 분석하여, *ExecutableDevice*의 *execute()* 함수를 호출한다. 그러나 기존의 *execute()* 함수는 새로 정의한 SAD 파일의 *processcollocation* 요소 부분을 제대로 실행할 수가 없다. 따라서 SAD 파일에 *processcollocation* 요소로 표현된 컴포넌트 정보가 기술되어 있다면, 이 정보를 분석하여 새로 구현한 *threadexecute()* 함수를 호출 하도록 *ApplicationFactory*의 *create()* 함수를 확장한다. 이를 통해서 기존에 작성된 응용 프로그램뿐만 아니라 *processcollocation* 제약 사항을 가지는 새로운 응용 프로그램도 문제없이 작동할 수 있도록 한다.

5. 결론 및 향후 연구 방향

본 논문에서는 RSCA 용 응용 프로그램의 수행 성능을 개선시키기 위한 방법에 대해서 제안하였다. SAD 파일을 확장하여 응용 프로그램의 실행 시 컴포넌트가 어떻게 실행되어야 할지를 명시할 수 있도록 하였다. 또한 SAD 파일에 기술된 제약사항을 만족시키기 위한 RSCA 의 확장 방법에 대해서도 언급하였다.

본 연구실에서는 C++ 언어로 RSCA를 구현하여 0.4.6 베타 버전을 공개하였다[6]. 현재 본 논문에서 제안한 확장된 RSCA를 구현 중에 있다. 추후 구현이 완료되면 다양한 응용 프로그램을 이용한 실험을 통해서 성능향상의 정도를 측정해 볼 계획이다.

참고 문헌

1. RSCA: URC 로봇 플랫폼에서 분산성, 이기종, 실시간성을 지원하는 통합 임베디드 미들웨어
2. Joint Tactical Radio Systems. "Software Communications Architecture Specification V2.2." November 2002
3. RSCA Specification, Version 0.6.4, http://rsca.snu.ac.kr
4. Evolution Robotics, http://www.evolution.com
5. Joint Tactical Radio Systems. "Software Communications Architecture Specification V3.0."
6. http://rsca.snu.ac.kr/download/