RSCA 로봇 응용의 상황인식 적응을 위한 컴포넌트 이동성 지원

김소연[0], 김세화, 홍성수
서울대학교 전기컴퓨터공학부
{sykim[0], ksaehwa, sshong}@redwood.snu.ac.kr

Component Migration Support for Context-aware Adaptation of RSCA Robot Applications

Soyeon Kim[0], Saehwa Kim, Seongsoo Hong
School of Electrical Engineering and Computer Science, Seoul National University

요 약

본 연구실에서는 URC 로봇의 응용 소프트웨어를 위한 표준 시스템 소프트웨어 구조로 RSCA를 개발하였다. RSCA는 로봇 응용 소프트웨어에게 표준화된 운영 환경을 제공하고, 이들의 개발을 용이하게 하는 프레임워크도 제공한다. 하지만 RSCA 분산 미들웨어인 CORBA ORB들과 RSCA 코어프레임워크는 동적인 재구성성에 대한 지원이 미흡한 실정이다. 본 논문에서는 RSCA 로봇 응용 컴포넌트의 이동성에 대한 제약들을 살펴보고, 이와 같은 문제점을 해결하기 위해 RSCA를 확장 구현한 설계를 소개한다. 본 실험적 연구를 통해, 로봇 응용 설계자들은 컴포넌트 이동성에 대한 구현의 어려움을 없앨 수 있고, 로봇 응용의 성능을 향상시킬 수 있다.

1. 서론

서울대학교 실시간운영체제 연구실에서는 로봇 소프트웨어 통신 구조 (RSCA: Robot Software Communications Architecture) [1]를 개발하였다. 본 연구실은 RSCA를 URC 로봇의 응용 소프트웨어를 위한 표준 시스템 소프트웨어 구조로 제안하였다. RSCA는 URC 로봇 응용에게 표준화된 운영 환경을 제공하고, 이들의 개발을 용이하게 하는 프레임워크도 제공한다. 구체적으로 RSCA의 운영 환경은 실시간 운영체제와 분산 제어 미들웨어, 배치 미들웨어의 3 계층으로 구성된다. 실시간 운영체제는 다양한 하드웨어 디바이스 위에서 로봇 응용을 신뢰성 있고 안정적으로 처리하는 동시에 탄력적이고 유연성 있게 구동하기 위하여 필요한 기본적인 추상화 계층을 제공한다. 분산 제어 미들웨어인 CORBA[2]은 URC 로봇의 분산 노드들의 다양한 이기종성을 숨기고 분산 응용의 부분들이 유연하게 상호 작용할 수 있도록 분산성을 감추는 추상화 계층을 제공한다. 마지막으로 배치 미들웨어인 코어 프레임워크는 로봇 응용의 재구성성을 지원하며 분산 컴포넌트 기반 응용의 배치를 지원한다. 이는 응용의 다운로드와 설치 및 제거, 응용의 생성과 소멸, 시작과 정지를 포괄하는 응용 컴포넌트들의 재구성 과정을 지원하는 계층이다. 추가로, RSCA에는 도메인 프로파일이라는 별도의 XML 파일이 제공되어야 한다. 이는 URC 로봇의 하드웨어와 소프트웨어의 구성 정보를 저장한다. 코어 프레임워크는 이 정보를 통해 시스템 동작 중에 동적으로 컴포넌트의 배치 및 자원 관리를 수행할 수 있다.

논문의 나머지 부분은 다음과 같이 구성되어 있다. 2장에서는 컴포넌트 이동성과 상황인식 적응의 용어를 정의하고 이들이 URC 로봇에서 어떤 역할을 하는지 설명한다. 그리고 현재 ORB들과 RSCA의 컴포넌트 이동을 위한 지원에서 어떠한 제약이 있는지 기술한다. 3장에서는 RSCA를 어떻게 확장 구현하여 우리가 직면해 있는 문제점을 해결하는지를 서술한다. 마지막으로 4장에서 결론을 맺는다.

2. 연구 동기

본 장에서는 우선 컴포넌트 이동성에 관하여 서술하고 컴포넌트 이동성이 URC 로봇에서 필요한 이유를 기술한다. 그리고 현재 ORB들과 RSCA의 제약들을 서술한다.

2.1. 컴포넌트 이동성 (component migration)

컴포넌트 이동성(component migration)이란 컴포넌트와 컴포넌트의 수행 문맥(execution context)을 이동시키는 능력으로 정의할 수 있다. 본 논문에서 기술되는 컴포넌트란 서로 다른 프로세서들 위에서 수행되는 분산 응용을 구성하는 기본요소를 의미한다. 단, 전제되어야 하는 조건은 컴포넌트의 소스를 이동시키는 것이 아닌 바이너리 파일을 이동시키므로, 컴포넌트가 요구하는 플랫폼의 조건들이 이동할 타겟 플랫폼의 정보와 완전 일치해야 한다는 것이다. 다음 그림 1은 컴포넌트 이동이 발생할 수 있는 한 예를 보여준다.

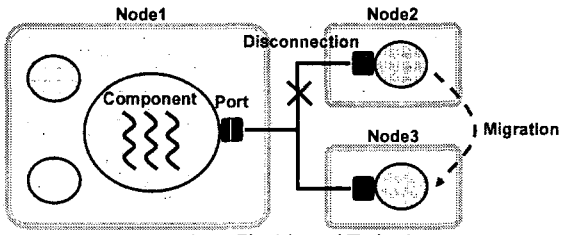

그림 1. 컴포넌트 이동의 예

컴포넌트 이동성의 구현에는 네트워크의 빈번한 끊김과 좁은 대역폭에서도 응용이 안정적으로 동작할 수 있도록 지원해 주는데 주요 목적이 있다. 이는 보통 다음과 같은 세부적인 목적들로 나뉠 수 있다[3].

- 부하 균형 (load balancing)

- 통신 성능 (communications performance)
- 이용가능성 (availability)
- 재구성 (reconfiguration)
- 특수 능력 활용 (utilizing special capabilities)

2.2. 컴포넌트 이동성의 필요성

URC 로봇에서 컴포넌트 이동성이 의미하는 바는 다음과 같다. URC 로봇의 특징은 문제해결 능력이 탁월하다는 것이다. 네트워크 환경에서 URC 로봇은 주변 사물들과 정보를 주고 받으며, 그때 그때 처한 상황(context)에 맞게 알맞게 처리하는데 능하다. URC 로봇이 사용자의 취향에 따른 맞춤형 서비스가 가능한 것도 이 때문이다. URC 로봇은 네트워크 연결이 끊어지거나 파부하된 노드에 더 이상 컴포넌트를 로드할 수 없는 상황을 인지할 수 있다. 더불어 나아가 더 이상 사용할 수 없는 컴포넌트를 이용 가능한 자원에 즉시 재구성한다면, URC 로봇은 응용의 최종 서비스를 지속적으로 제공할 수 있다. URC 로봇 응용에게 표준화된 운영 환경을 제공하는 RSCA에 위에서 언급한 기능을 제공하도록 확장 구현한다. 확장된 RSCA 기반으로 개발된 URC 로봇은 어느 영역에서나 네트워크로 연결된 로봇 단말기가 존재하는 한, 언제 어디서나 사용자를 위한 맞춤형 정보를 서비스할 수 있을 것이다.

2.3. ORB 와 RSCA 의 제약

CORBA 2.3에서는 이동성(mobility)을 위해 valuetype이라는 IDL 키워드를 사용하여 명세하였다. 그러나 이에 대한 ORB들의 지원은 미비하거나 전혀 없는 경우가 많다. 우선 응용 설계자들이 대체적으로 많이 이용하는 무료 ORB들을 살펴보겠다. TAO[4]는 CORBA 3.x를 지원하지만, valuetype을 위한 함수는 일부만 지원해준다. omniORB[5]는 CORBA 2.6을 지원하지만, valuetype의 사용을 제한한다. 마찬가지로, RSCA는 컴포넌트의 동적 재구성을 위해 파라미터를 설정할 수 있도록 인터페이스를 제공해주는데, 이런 방식의 동적 재구성에는 한계가 있다.

따라서, 동적 재구성을 위해 컴포넌트를 다시 인스턴스화하고 그에 대한 포트를 연결해주는 과정이 필요하다. 지금은 응용의 설계자가 이를 위한 인터페이스를 직접 구현해줘야 한다. 하지만 응용 설계자의 직접 구현은 구현상의 복잡성뿐만 아니라, 구현 가능한 언어에 따라 제한되므로 설계자는 많은 어려움을 겪을 수 있다. 이와 같은 응용 설계자의 어려움을 해소시키기 위해, 우리는 RSCA에서 컴포넌트 이동성을 지원하도록 코어 프레임워크를 확장한다.

3. RSCA 코어 프레임워크의 설계 및 구현

본 장에서는 RSCA가 컴포넌트 이동성을 지원해 주도록 코어 프레임워크 구성요소들의 인터페이스를 확장한 설계를 제안한다. 본 실험적 연구의 목적은 응용 설계자의 컴포넌트 이동에 대한 설계상의 편리성과 응용의 성능 향상을 가져오는 것이다. 각 그림에서 음영으로 처리된 부분은 우리가 추가 구현한 인터페이스를 나타낸다.

3.1. *LifeCycle* 인터페이스의 확장

컴포넌트 이동을 위한 구현은 컴포넌트의 생성부터 소멸까지 컴포넌트의 라이프사이클을 관리하는 *LifeCycle* 인터페이스에 추가하는 것이 바람직하다. 이를 통해, 여러 노드에 분산된 각 로봇 응용의 컴포넌트를 일괄적으로 이동시키거나 이동된 컴포넌트를 제거할 수 있다.

LifeCycle 인터페이스에 추가된 함수들을 살펴보기 이전에, *LifeCycle*과 *Resource*, *Application*의 인터페이스들의 관계에 대하여 간단히 살펴본다. *LifeCycle* 인터페이스는 기본 응용의 인터페이스로서, 컴포넌트 인스턴스를 초기화하거나 해제하는 함수들을 정의한다. 응용을 구성하는 각 컴포넌트(*Resource*)의 시작과 정지 동작에 대한 구현은 *LifeCycle* 인터페이스에서 제공하고 *Resource*와 *Application*이 각각 상속한다. 다음은 *LifeCycle* 인터페이스에 추가한 함수들을 기술한 것이다[6]. 컴포넌트를 이동시키는 구현 부분은 Boost Library의 C++ Serialization[7]을 참조하여 RSCA에 최적화하였다.

- *LifeCycle LifeCycle::copy()*: 컴포넌트의 복사본을 이 함수를 호출한 노드에 생성한 후 컴포넌트 참조를 반환한다. 이 인터페이스는 장애방지(fault-tolerant)를 위해 사용된다.
- *LifeCycle LifeCycle::move()*: 컴포넌트의 복사본을 이 함수를 호출한 노드에 생성한 후 객체를 제거한다. 결과값으로 생성한 컴포넌트 참조를 반환한다. 이 인터페이스는 부하 균형을 위해 사용된다.
- *void LifeCycle::remove()*: 함수를 호출한 노드에 생성되어 있는 컴포넌트의 복사본을 제거한다.

3.2. *Application* 인터페이스의 확장

컴포넌트 이동으로 재구성된 응용을 위한 구현은 응용의 시작과 정지를 관리하는 *Application* 인터페이스에 추가한다. 그리고 응용의 생성부터 소멸까지 각 응용의 라이프사이클을 관리하며, 특정 컴포넌트가 요구하는 후보 디바이스들의 위치를 파악할 수 있는 *ApplicationFactory* 인터페이스에서 사용하는 것이 바람직하다. 이를 통해, 여러 노드에 분산된 각 로봇 응용을 일괄적으로 상황인식 적응을 할 수 있다.

Application 인터페이스에 추가된 함수들을 살펴보기 이전에, *Application*의 인터페이스에 대하여 간단히 살펴본다. *Application*은 프레임워크 제어 인터페이스로서, 사용자가 새로운 소프트웨어(응용)를 설치했을 때 코어 프레임워크 내에 자동으로 생성되는 컴포넌트이다. 이는 사용자가 설치한 응용을 시작하고 정지할 수 있는 인터페이스를 제공한다.

Application 인터페이스에 특정 컴포넌트를 적재하고 실행할 수 있는 디바이스를 다시 결정하거나 새로 통신할 포트를 연결하는 과정을 추가하였다. 다음은 *Application*에 추가한 인터페이스들을 기술한 것이다.

- *Port ApplicationFactory:: replaceComponent(in Resource name, in Properties initConfiguration, in DeviceAssignmentSequence deviceAssignments)*: 컴포넌트를 재수행하는데 필요한 디바이스를 결정하거나 새로운 포트를 연결한다.

3.3. *ApplicationFactory* 인터페이스의 확장

ApplicationFactory 인터페이스에서는 특정 컴포넌트가 요구하는 후보 디바이스들의 위치를 파악하고 컴포넌트의 포트를 연결할 수 있기 때문에, 앞 절에서 서술한 *Application* 인터페이스는 *ApplicationFactory* 인터페이스를 통해 사용되는 것이 바람직하다. 다음은 *ApplicationFactory*에 추가한 인터페이스를 기술한 것이다.

- *void ApplicationFactory::initializeAdaptation()*: 장애방지를 위해, 응용의 컴포넌트들의 요구조건을 만족하는 후보 디바이스들에 각각 복사해 둔다.
- *Port ApplicationFactory::adaptApplication(in Resource name, in Properties initConfiguration, in DeviceAssignmentSequence deviceAssignments)*: *Application*의 인터페이스를 이용하여 컴포넌트를 재수행하는데 필요한 디바이스를 결정하거나 새로운 포트를 연결한다.

다음 그림 2는 UML 클래스 다이어그램을 사용하여 앞에서 언급한 *LifeCycle*과 *Resource*, *Application*,

ApplicationFactory 인터페이스들을 기술하고 그들의 관계를 보여준다.

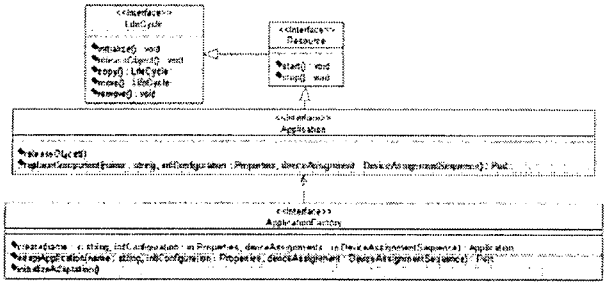

그림 2. *LifeCycle*과 *Resource, Application, ApplicationFactory* 인터페이스

3.4. 실험용 수행 시나리오

본 연구의 실험에서는 다음과 같은 환경을 지원한다. 공개 소프트웨어인 Linux와 omniORB를 운영체제와 CORBA ORB로 각각 사용하였다. RSCA 코어 프레임워크는 C++ 언어를 이용하여 컴포넌트 이동성을 구현 중이다.

본 연구에서는 두 가지의 수행 시나리오가 있다. 하나는 시스템의 부하 균형을 위해 사용될 수 있는 시나리오이다. 다른 하나는 장애방지(fault-tolerant)를 위해 사용될 수 있는 시나리오이다. RSCA가 지원하는 컴포넌트 이동성을 검증하기 위해, 본 논문에서는 간단하게 두 번째 시나리오가 수행되는 과정을 기술한다.

실험용 수행 시나리오를 설명하기 앞서서, 실험에 대한 몇 가지 조건들을 가정한다. 모든 컴포넌트들은 SAD 도메인 프로파일에서 요구되는 조건들(프로세서, 운영체제, 디바이스 등)을 만족시키는 환경에 분산 배치되도록 구성되어야 한다. 그리고 RSCA 코어 프레임워크에서는 컴포넌트의 상황(context)을 전달하는 인터페이스를 고려하지 않았다. 따라서 이를 어떻게 효과적으로 전달할지에 대해서는 사용자 레벨에서 구현해야 한다.

ApplicationFactory 인터페이스는 *Application*을 생성하는 인터페이스를 제공한다. *ApplicationFactory*의 *create*() 함수의 수행과정 중 각 *Resource*의 *Port*를 연결한 후 프로퍼티를 설정하는 11번 단계가 끝나고 상황인식 적응을 위한 초기 동작이 12번 단계로 추가되었다. 다음에는 컴포넌트의 이동성을 추가한 12번의 내부 수행과정을 번호 순으로 기술한다. 오른쪽 그림 3은 *ApplicationFactory*의 *create*() 함수가 호출될 때 수행되는 동작들을 보여준다.

1. *ApplicationFactory*는 *initializeAdaptation*() 함수를 호출한다.
2. 위 함수 내부에서는 대상 컴포넌트의 *Resource::copy*() 함수가 호출되어 수행 요구조건을 만족하는 후보 디바이스에 컴포넌트를 복사한다(이 컴포넌트를 복사본 컴포넌트라 부름).

응용이 수행하는 도중에 한 컴포넌트의 네트워크 연결이 끊어진다. 그런데 클라이언트는 네트워크 연결이 끊어진 컴포넌트의 포트로 통신을 시도한다. 이때 네트워크 연결이 끊어져 있기 때문에, 컴포넌트 폴트가 발생한다.

3. *ApplicationFactory*에게 timeout exception 메시지가 전달되면, *ApplicationFactory*는 *adaptApplication*()을 호출한다.
4. 위 함수의 내부에서는 *Application::replaceComponent*() 함수를 호출한다. 우선 응용을 중지하고, *name*이란 복사본 컴포넌트에 대한 참조를 획득한다. 필요한 자원을 설정하고, 클라이언트와 새로운 컴포넌트 참조와의 포트를 연결한다.

클라이언트는 새로 연결된 포트로 대상 컴포넌트와 통신한다. 응용이 수행을 끝내고 *Application*이 종료될 때, 다음과 같이 처리된다.

5. *Application*은 소멸시 대상 컴포넌트의 *remove*()를 호출하여, 복사본 컴포넌트에 대한 자원을 해제하고 복사본 컴포넌트를 제거한다.

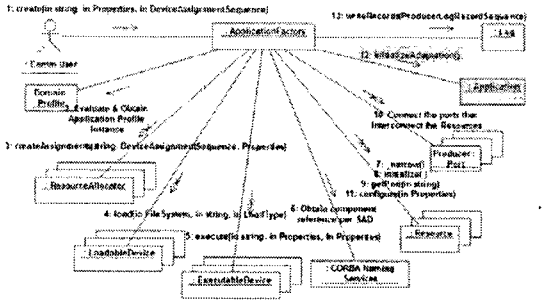

그림 3. *ApplicationFactory* 동작

4. 결론

본 논문에서는 RSCA의 상황인식 적응을 위한 컴포넌트 이동성에 대한 제약들을 살펴보고, 이를 해결하는 모델로 RSCA를 확장 구현한 설계를 제안하였다. 본 실험적 연구를 통해 다음과 같은 결과들을 기대한다. 우선, 응용 설계자는 컴포넌트의 이동에 관련된 복잡한 구현없이, 확장된 RSCA를 기반으로 편하게 설계할 수 있다. 이로 인해 응용의 개발 시간을 단축시킬 수 있다. 더 나아가, 컴포넌트 이동성을 제공함으로써 전체 시스템 부하의 균형을 맞출 수 있으며, 그 결과 분산된 실행 바이너리의 성능을 충분히 향상시킬 수 있다. 본 연구실은 vSLAM 응용 소프트웨어를 테스트 베드에 탑재하여 RSCA의 타당성과 성능을 검증하였다. 또한 본 논문에서 제시한 확장된 RSCA를 로봇에 적용 가능하도록 구현 중이다. 앞으로는 본 논문에서 실험용 수행 시나리오들을 통해 본 연구의 타당성을 검증하고, 계속해서 성능 개선 및 보안, QoS 등에 대한 연구를 진행할 계획이다.

참고문헌

1. RSCA Specification, Version 0.6.4, http://rsca.snu.ac.kr
2. The Common Object Request Broker: Architecture and Specification, Version 3.0, Object Management Group, June 2002
3. Lakshmikanth S. Jonnalagadda, "The Role of Network Traffic Statistics in Devising Object Migration Policies," Department of Electrical and Computer Engineering, Rutgers University, May 1997
4. TAO, http://www.cs.wustl.edu/~schmidt/TAO.html
5. omniORB, http://omniorb.sourceforge.net
6. Y. Peter, "Mobility Management in CORBA: A Generic Implementation of the LifeCycle Service," XXIV-th Seminar on Current Trends in Theory and Practice of Informatics, 1997
7. Boost C++ Library http://www.boost.org/
8. 홍성수, "RSCA: 분산 로봇 플랫폼에서 임베디드 소프트웨어의 동적 재구성을 지원하는 통합 미들웨어," 로봇공학회지, 제1권 제1호, pp. 35-52, 2004. 10.