DOI: 10.3745/KIPSTA.2010.17A.5.249

# 로봇 어플리케이션을 위한 협업 프레임워크 설계

이 창 묵†·권 오 영††


## 요 약

로봇 어플리케이션의 활용도는 사회 전반에서 점차 확대되고 있지만 로봇들의 컴퓨팅 자원 차이로 인해 한 로봇에서 다양한 어플리케이션을 실행하기는 힘든 실정이다. 본 논문에서는 로봇이 주변 장치와의 자원 공유를 통해 자원의 제약을 극복하는 프레임워크를 제안한다. 프레임워크는 협업에 요구되는 공통 구성 요소들을 정의하고 어플리케이션을 쉽게 제작할 수 있는 API를 제공한다. 로봇과 다른 장치가 네트워크로 연결된 환경에서 동작하는 로봇을 이용한 체조 학습 어플리케이션의 예시를 통해 프레임워크의 동작 흐름을 보인다.


**키워드 : 로봇 미들웨어 프레임워크, 분산 협업, 로봇 어플리케이션 API**

# The Design of Collaboration Framework for Robot Application

Chang-Mug Lee† · Oh-Young Kwon††


## ABSTRACT

The utilization of robot application is growing up in recent years, but there is a constraint to execute various application on the robot because of difference of robot resource. This paper presents the framework in order to solve the resource constraint by sharing resources with other devices near by robot. The framework defines common factors that are needed to collaboration work and provides APIs in order to implement robot application easily. Furthermore, We show the working flow of framework with physical training application using robot by example. The application shows how to collaborated work between robot and other devices through network.


**Keywords : Robot Middleware Framework, Collaborated Working, Robot Application API**

## 1. 서 론

로봇 기술의 발전에 따라 로봇은 산업, 서비스 분야에서 점차 폭넓게 이용되고 있다. 이와 관련하여 집과 사무실 등의 공간에서 가정용 로봇을 이용한 다양한 서비스를 제공하는 어플리케이션도 최근 주목받고 있다[1, 2]. 로봇은 점차 소형화, 고성능화 되고 있지만 성능과 비례하는 비용 상승의 문제 등으로 인해 로봇에서 규모 있는 어플리케이션을 실행하기 어려운 경우가 많다. 가정용 로봇과 같은 소형 임베디드 시스템의 자원 제약 문제를 해결하기 위해 어플리케이션 코드 최적화나 커널 최적화 등의 연구들이 수행되어 왔지만 이러한 방법들은 시스템의 주어진 자원 내에서만 효율을 향상 시킬 수 있다. 본 논문은 홈 네트워크상에 다수의 장치가 존재하는 실행 환경을 대상으로 하므로 자원 제약 문제에 대한 해결책으로 로봇이 주변의 가용한 장치들과 협업을 수행하여 자원의 한계를 극복하는 모델을 고려해 볼 수 있다. 여러 장치들의 네트워크를 통한 연결과 협업을 위한 기술로는 UPNP[3, 4], RPC[5] 등의 표준화된 기술들이 존재하지만 이들은 가전기기의 편리한 이용이나 PC를 기반으로 하는 범용적인 환경에 사용하기 위한 기술로, 자원의 제약이 심할 수 있는 로봇 어플리케이션에서 협업 기능만을 위해 사용하기에 적합하지 않다. 따라서 우리는 로봇 어플리케이션에 적합한 협업 모델의 구현을 위한 협업 프레임워크를 개발했다. 프레임워크는 장치들 간의 협업에 필요한 공통 요소와 동작을 정립하고 어플리케이션 개발 단계에서 적용 가능한 인터페이스를 제공하며 로봇과 다른 장치들 사이의 협업을 위한 기능을 수행하는 미들웨어 역할을 한다. 본 논문에서는 협업 프레임워크 모델 설계와 동작 흐름, 프레임워크를 이용한 어플리케이션의 제작을 기술한다.

일반적으로 어플리케이션은 특정 기능을 수행하는 모듈, 혹은 루틴들이 결합된 형태이다. 프레임워크를 이용한 개발에서 모듈들은 개별 모듈을 담당하는 프레임워크의 구성요소에 탑재된다. 프레임워크에 탑재된 모듈은 협업을 위한

※ 이 논문은 한국기술교육대학교 교육진흥비 지원 프로그램의 지원에 의하여 수행되었음.
† 정 회 원 : 한국기술교육대학교 전기전자공학과
†† 종신회원 : 한국기술교육대학교 컴퓨터공학부 부교수
논문접수 : 2010년 3월 30일
수 정 일 : 1차 2010년 5월 31일
심사완료 : 2010년 6월 18일

기능이 추가되었을 뿐 여전히 어플리케이션을 특정 기능 수행을 담당하므로 일반적인 어플리케이션 개발과 같이 여러 모듈을 적절히 배치, 연결하여 어플리케이션을 작성한다. 이후 같은 네트워크 상에 존재하는 로봇과 여러 장치들 모두에 프레임워크를 적용하여 작성된 클론 어플리케이션이 탑재되고, 로봇이 어플리케이션을 실행하면 각 장치에 있는 클론 어플리케이션의 모듈들이 로봇의 모듈과 협업을 하게 된다.

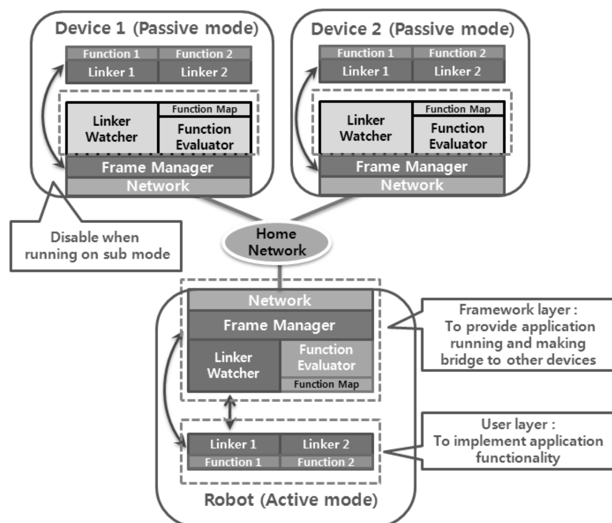
이어지는 2장에서는 프레임워크의 전체 구성과 프레임워크를 이용한 개발 관점의 접근을 설명한다. 3장에서는 프레임워크의 각 구성 요소와 동작을 상세히 설명한다. 4장에서는 프레임워크를 이용한 어플리케이션이 로봇과 여러 장치에서 협업을 하는 동작 과정을 체조 학습 시스템 어플리케이션[6, 7] 의 예를 들어 설명한다. 5장에서는 프레임워크와 다른 유사 기술과의 특징을 비교하며 프레임워크의 활용성에 대해 기술한다.

## 2. 프레임워크의 구성

### 2.1 프레임워크의 구조

전체 프레임워크의 구성은 (그림 1)에 나타나 있다. 그림에서 디바이스는 어플리케이션을 실행할 수 있는 모든 종류의 컴퓨팅 장치를 나타내며 로봇과 디바이스들은 로컬 네트워크를 통해 연결되어 있다. 디바이스와 로봇에는 (그림 1)과 동일한 구조로 작성된 클론 어플리케이션이 각각 탑재된다. 이때 로봇의 어플리케이션은 전체 실행을 주관하는 엑티브(active) 모드로 동작하고 디바이스의 어플리케이션은 로봇에서 요청한 기능만 선별적으로 처리해주는 패시브(passive) 모드로 동작하며 패시브 모드로 동작할 시 프레임워크의 구성요소 중 Linker Watcher 와 Function Evaluator 는 작동하지 않는다.

프레임워크는 Function, Linker, Function Evaluator, Linker Watcher, Frame Manager로 구성된다. Frame Manager는 다른 구성요소들의 동작을 통제하고 조율하며 네트워크를 통해 다른 장치의 클론 어플리케이션과 연결한다. Function 과 Linker 는 어플리케이션의 기능이 직접 구현되는 부분이며 프레임워크에서 제공하는 틀 안에 사용자가 직접 루틴을 작성하여 기능을 구현할 수 있다. Function Evaluator 와 Linker Watcher 는 어플리케이션의 실행 흐름에 따라 연결된 장치들 중 최적의 효율을 보인 장치를 선별하여 기능 요소를 실행한다.




(그림 1) 프레임워크의 구성

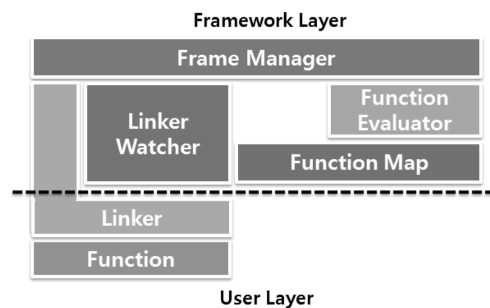
### 2.2 프레임워크의 계층별 구조

프레임워크를 구성하는 요소들의 전체 구조를 계층별로 나타내면 (그림 2)와 같다.

그림과 같이 최상위 레벨 요소는 전체 동작을 조절하는 Frame Manager이다. Frame Manager 의 다음 계층이며 직접 정보를 교환하는 요소로는 Linker Watcher와 Function Evaluator가 있다. Function Map 은 Function Evaluator에 포함된 요소이지만 Frame Manager가 아닌 Linker Watcher 에 직접 정보를 제공한다. 이 요소들은 모두 프레임워크 계층에 속하며 사용자의 어플리케이션 구현 방법과 관계없이 자체적으로 동작하는 기능 요소들이다.

사용자 계층에 속하는 요소로는 Linker 와 Function 이 있다. 두 가지 요소는 위에서 설명한 기능들을 기본적으로 제공하지만, 사용자의 어플리케이션 구현에 따라 세부적인 수행 기능이 달라지는 요소이다. 프레임워크 전체의 계층 관점에서 보면 Linker 와 Function 은 Linker Watcher 와 상호작용하는 하부 계층에 속하지만, 입출력 데이터의 전달과 Linker state 변경 동작 시에 Linker는 예외적으로 Frame Manager와 직접 정보를 교환한다.

프레임워크 구성요소들의 상세한 설명은 3장에서 설명한다.




(그림 2) 계층별 프레임워크 구조

### 2.3 프레임워크의 구현

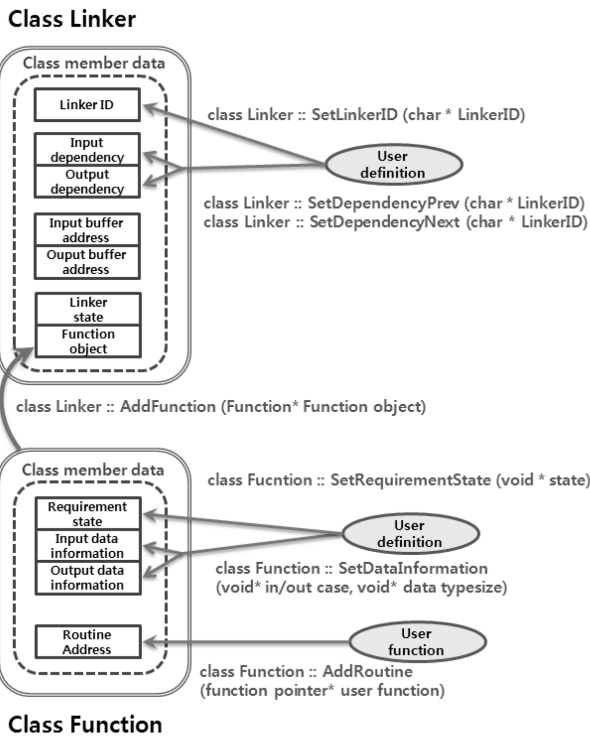
2.2 절의 (그림 2)와 같이 어플리케이션에서 사용자가 직접 구현하게 되는 구성요소는 Function 과 Linker 이다. 프레임워크는 Function 과 Linker 의 기능 구현을 위한 인터페이스를 제공한다. <표 1>은 프레임워크에서 제공하는 클래스별 API이다.

<표 1>의 API를 이용한 구현 과정은 (그림 3)에 나타나

〈표 1〉 프레임워크 API

| Class Function | |
|---|---|
| SetRequirementState(void* state) | 수행 능력 요구사항 설정 |
| SetDataInformation(void* in/out case, void* data typesize) | 루틴의 입출력 데이터 형식 등록 |
| AddRoutine(function pointer* user function) | 루틴을 Function 에 등록 |
| Class Linker | |
| SetLinkerID(char* Linker ID) | Linker ID를 Linker 객체 내부에 등록 |
| SetDependencyPrev(char* LinkerID) | Linker 의 선행 종속성 설정 |
| SetDependencyNext(char* LinkerID) | Linker 의 후속 종속성 설정 |
| Class FrameManager | |
| AddLinker(Linker* linker) | FrameManager 에 Linker 의 주소 등록 |

있다. 사용자는 먼저 Function 클래스의 사용자 정의 요소들을 클래스 멤버 함수를 통해 정의한다. 각 Function의 기능 수행을 담당할 Routine 부분은 프로그래밍 언어의 일반적인 단일 함수 형태로 작성한 후 Function 의 멤버 함수 AddRoutine 을 통해 Function 에 등록한다. 이때 Function 클래스의 Routine address 에는 사용자 작성 함수의 포인터가 저장된다. 다음으로 사용자 작성 함수의 입출력 데이터 형식과 수행 성능 측정을 위한 변수들도 제공되는 Function 멤버 함수를 통해 등록한다. 하나의 Function 은 이러한 과정을 거쳐 만들어지며 차후 Function 이 수행될 때 내부적




(그림 3) Function 과 Linker 의 구현

으로 입력 데이터를 사용자가 정의한 입력 데이터 형식으로 캐스팅한 후 루틴(함수 포인터) 을 호출하여 실행한다.

하나의 Function을 관리하고 실행시키는 도구는 Linker 이므로 각각의 Function 에 대응하는 Linker 역시 제공되는 Linker 클래스를 사용하여 제작한다. Linker 클래스 역시 구현을 위한 멤버 데이터와 함수를 제공한다. 사용자가 제작한 Function 은 AddFunction 함수를 통해 Linker 에 등록된다. Linker의 Dependency 와 ID 또한 사용자가 직접 정의하지만, Linker 클래스의 멤버 데이터 중 In/Output buffer address 와 Linker state는 사용자가 정의하지 않고 Frame Manger 에 의해 어플리케이션 실행 중 제어된다. 어플리케이션에 여러 Function 이 존재한다면 위의 과정을 반복한다.

이 후에 모든 Linker 들은 각 Function을 소유하고 있는 상태가 된다. Linker 는 여러 장비에 존재할 수 있는 동일한 Linker 들을 분류하기 위해 Linker ID를 사용하지만, 시스템 상에서 실제 Linker에 접근하기 위해서는 Linker 객체의 메모리 주소를 알아야 한다. 따라서 Linker를 제작한 이후에는 Frame Manager 클래스에서 제공하는 API를 사용해 Linker 를 등록해 주어야 하며 Frame Manager 에서는 내부적으로 Linker ID 와 실제 주소를 저장하는 테이블을 유지한다.

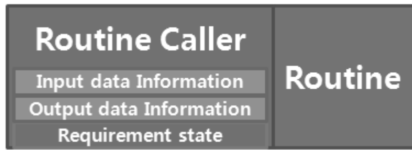
사용자의 프레임워크에 대한 접근은 이 단계까지이며 Linker를 실행하면 내부적으로 Linker Watcher에 Linker ID 가 등록된 후 다른 장치와의 협업을 진행하게 된다.

## 3. 프레임워크의 구성 요소

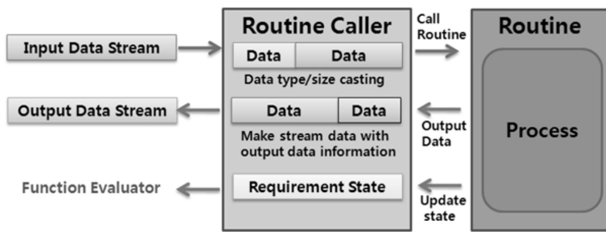
### 3.1 Function

Function은 전체 어플리케이션을 기능별로 분할했을 때 하나의 기능을 말한다. (그림 4)의 구조는 각각 하나의 Function 에 대응한다. Function 의 구조는 (그림 4)와 같다.

Routine Caller 는 Function 의 기능이 구현된 루틴을 직접 실행하는 부분이다. Routine Caller에는 세 가지 요소가 포함되어 있는데, In/Output data information 은 루틴의 실행에 필요한 입출력 데이터의 타입, 크기, 메모리 주소 정보가 기록된다. Requirement state 에는 Function 의 루틴이 실행되었을 때의 실행 결과를 측정한 값이 기록된다. 측정 기준은 사전에 정의되어 있는 것이 아니며 어플리케이션 구현 시 각 Function 에 요구되는 기능에 따라 사용자에 의해 정의된다. 예를 들어 어떤 Function 이 카메라를 통해 영상을 입력받는 기능을 수행한다고 가정하자. 카메라에 관련된 측정 기준은 카메라의 유무, 해상도, 초당 프레임 등이 있을 수 있다. 카메라의 유무를 측정 기준으로 선택한다면 Requirement state 는 단순히 TRUE/FALSE 값만을 나타내는 것으로 충분하다. 이 경우 루틴에는 카메라 유무를 확인하여 Requirement state에 기록하는 코드가 포함되어야 한다. 측정 기준을 루틴의 수행 시간으로 정의하고 싶다면, Requirement state 는 시간을 나타낼 실수형 데이터로 정의할 수 있고 루틴의 시작과 끝에는 시간을 측정하여 Requirement state 변수에 기록하는 코드가 포함되어야 한다. Requirement




(그림 4) Function 의 구조




(그림 5) 루틴의 실행 과정

state 에 기록된 결과는 로봇의 Function Evaluator 로 전송된다. 루틴은 Function 에 요구되는 기능을 직접 구현하는 코드 부분이다. 루틴에는 기능 구현 부분 외에 앞서 설명한 바와 같이 Requirement state 를 측정하여 기록하는 부분이 반드시 포함되어야 한다. Function에서 실제 루틴이 실행되는 과정은 (그림 5)에 나타나 있다.

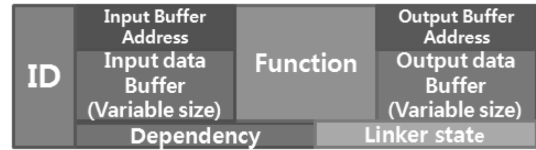
Function 에 필요한 입력 데이터는 Linker에 의해 스트림 형태로 전달된다. Routine Caller 는 이 데이터를 Input data information을 이용해 루틴에 요구되는 입력 형태로 캐스팅하고 루틴을 실행한다. 루틴이 실행된 후 나온 출력 데이터는 다시 Routine Caller 로 전달되며 Output data information 정보에 따라 스트림 형태로 변환되어 Linker로 전달된다. 이때 In/Output data information 에는 데이터의 형식과 크기가 정의되어 있으며 Routine Caller 는 해당 정보에 따라 데이터의 크기를 확인한 후 정의된 대로 데이터를 캐스팅한다. 루틴이 실행될 때마다 당회 측정된 루틴의 성능 평가 결과가 Requirement state 에 업데이트 된다.

### 3.2 Linker

Linker는 Function을 실행하기 위해 호출하고 전체 어플리케이션의 실행 흐름에 따라 Function 끼리 연결하는 역할을 한다. Linker의 구조는 (그림 6)과 같다.

ID는 사용자에 의해 Linker 에 부여되는 유일한 식별자로 로봇과 디바이스들에 여러 Linker 들이 존재할 때 Linker 들을 식별하기 위해 사용된다. Linker 는 Function을 감싸는 형태로 존재하며 하나의 Linker 는 하나의 Function 을 포함한다.

입출력 버퍼는 로봇과 디바이스 사이의 Function 별 입출력 데이터 전송 시 네트워크 버퍼 역할을 수행하거나 아래에서 설명할 종속성에 따라 Linker 가 대기 상태에 놓이면 입출력 데이터를 저장하는 버퍼 역할을 수행한다. 다음의 경우를 보자. Function 은 어플리케이션의 단일 기능을 수행하는 단위이고 Linker는 하나의 Function을 호출하는 주체



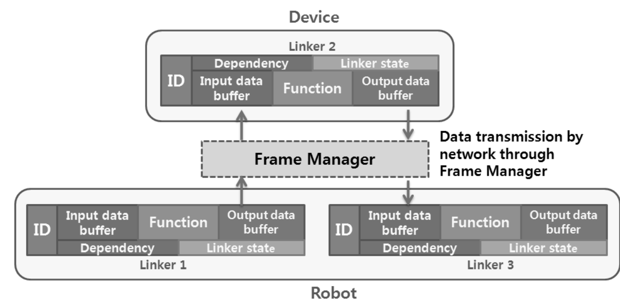

(그림 6) Linker 의 구조

이므로 전체 어플리케이션의 진행은 Linker 들이 서로 연결되어 수행되게 된다. (그림 7)은 로봇과 디바이스에 존재하는 Linker 들이 연결된 예를 나타낸다.

Linker 의 출력 데이터는 연결된 다른 Linker 의 입력 데이터가 되는데, 이때 연결된 두 Linker가 서로 다른 장치에 존재하고 네트워크를 통해 입출력 데이터를 주고받는다면 입력 데이터 수신이 끝나기 전에는 연결된 다음 Linker 가 실행될 수 없고, 마찬가지로 데이터를 다음 Linker 에 모두 전송하기 전에는 자신의 출력 데이터도 삭제되어서는 안된다. 따라서 모든 Linker 는 입출력 데이터 버퍼를 통해 서로 데이터를 교환한다. 장치간의 데이터 교환 시 로봇과 디바이스의 시스템이 다를 수 있으므로 데이터 교환에는 머신 독립성(Machine Independency) 이 요구된다. 이를 위해 네트워크를 통한 전송에는 RPC의 XDR (External Data Representation) 프로토콜을 사용한다.

입출력 버퍼의 데이터는 바이너리 스트림 형태로 존재하며 (그림 5)의 과정을 거쳐 Function 에 요구되는 형태로 변환된다. Function 별로 요구되는 데이터 형식은 Function 내부에 정의되어 있지만 데이터의 크기는 연산 결과에 따라 가변적일 수 있으므로 Linker 사이의 데이터 전송 시 데이터의 크기도 함께 통보한다. Linker 의 입출력 버퍼는 할당될 때마다 크기와 메모리 상의 위치가 변할 수 있으므로 Linker 는 가변적인 크기의 입출력 버퍼에 항상 접근할 수 있도록 입출력 버퍼의 주소를 소유한다. (그림 7)에서 Linker 에는 입출력 버퍼가 존재하지만 이것은 각 Linker 가 직접 메모리 공간을 소유하고 있다는 의미가 아니다. 실제로 Linker 는 자신의 입출력 버퍼가 있는 메모리 공간의 주소만 소유하게 되며 이에 관해서는 3.5 절에서 상세히 설명한다.

다음으로 종속성이 있다. 종속성은 Linker 들 사이에 미리 정의된 종속성에 따라 선행 Linker 의 종료를 기다리거




(그림 7) Linker 들의 연결 관계

나 독립적으로 실행함을 결정한다. Linker 들은 서로 연결되어 있으므로 선행 Linker 의 출력 데이터가 자신의 입력 데이터로 사용된다면 이 Linker는 반드시 선행 Linker 가 종료된 후에 실행될 수 있다. (그림 7)에서 Linker 2는 Linker 1에, Linker 3은 Linker 2에 종속되어 있다. 그림의 경우 모든 Linker 들은 순차적으로 실행되어야 하지만 Linker 1 과 3은 서로 종속성은 없다.

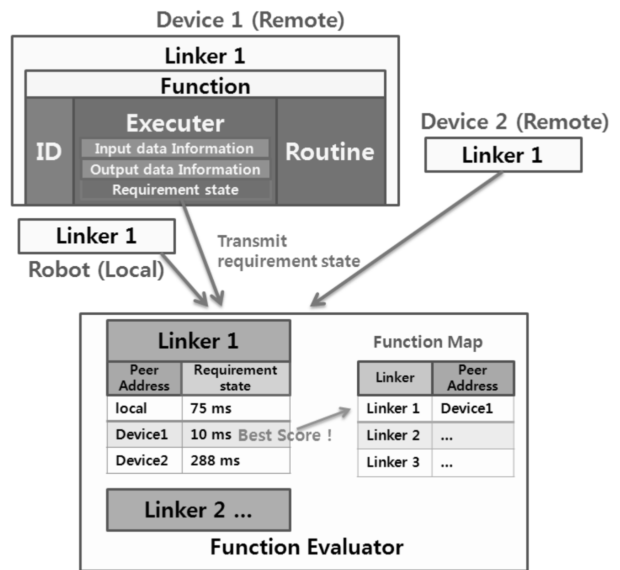
마지막으로 Linker state 는 Linker 의 현재 상태를 나타내는 것으로 Linker Watcher 에 의해 사용된다. Linker state 에는 Input data waitint, Output data waiting, Ready 의 세 가지 상태가 있다. Input data waiting 은 Linker 의 입력 데이터가 준비되었는지를 확인하기 위한 것으로 선행 Linker 의 출력 데이터가 자신에게 입력되지 않았을 경우에도 해당되지만, 주로 어플리케이션 사용자에 의한 직접적인 데이터 입력을 기다리는 상황이 이에 해당한다. Ouput data waiting 은 Linker 사이의 종속성과 관계가 있다. 종속성이 설정된 Linker 들은 선행 Linker 의 출력 데이터가 후속 Linker 의 입력 데이터로 사용되므로 선행 Linker 의 출력 데이터가 모두 완료되었는지를 확인하기 위한 것이다. Ready 는 모든 문제가 해결되어 Linker 가 즉시 실행 가능한 상태를 나타낸다.

Linker 가 Linker Watcher에 최초 등록될 때는 모든 상태가 불가능 (FALSE) 상태로 등록되며, 차후 설명할 Frame Manager 에 의해 상태 정보가 업데이트 되며 Ready 상태로 변경된다. 단, 선행 Linker 에 대한 종속성이 존재하지 않고 사용자 입력이 필요 없는 Linker 는 즉시 Ready 상태로 등록된다. Linker Watcher 는 개별 Linker의 state를 감시하여 Ready 상태가 된 Linker 만 실행한다.

### 3.3 Function Evaluator

Function Evaluator는 각 Linker 에 속한 Function 의 Requirement state, 즉 Function 수행 능력 측정 결과를 종합하여 평가한다. (그림 8)과 같이 하나의 Function에 대해 다수의 디바이스들과 로봇의 수행 능력 측정 결과 중 가장 좋은 성능의 보인 결과를 선택하고 이 결과를 보낸 장치가 어디인지를 Function Evaluator 에 포함된 Function Map에 기록한다. 그림에서는 원격지의 디바이스에서 실행된 두 개의 Linker 1과 로컬에서 실행된 로봇의 Linker 1의 Function 수행 능력 측정 결과를 비교하고 이 중 가장 좋은 결과를 보인 디바이스 1을 선정하여 Function map 에 기록했다. 측정 기준은 루틴의 실행 시간으로 가정했다. 다수의 Linker 가 존재하는 경우도 마찬가지로 다수의 장치에 존재하는 동일한 Linker 들의 성능 평가를 거친 후 Function map 에 최적의 수행 결과를 기록한다. 각 디바이스에서 Function 이 실행될 때 마다 Requirement state 는 갱신되고 이 결과가 Function Evaluator 에 업데이트 되면 재평가 작업을 거친 후 Function Map을 업데이트한다.

어플리케이션이 처음 실행되는 순간에는 장치들에 존재하는 어떠한 Function 도 수행된 적이 없으므로 Function




(그림 8) Function Evaluator 의 동작

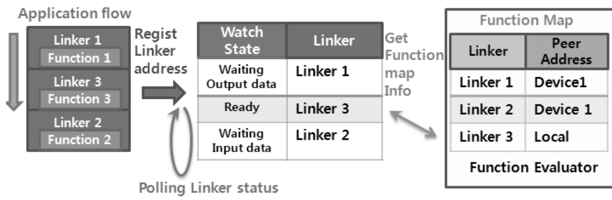
Evaluator 는 어떠한 Requirement state 도 존재하지 않는다. 따라서 어플리케이션의 최초 실행을 위한 기본 Requirement state 값은 사용자가 직접 어플리케이션이 구동될 시스템의 특성에 따라 지정해 주어야 한다. 지정 시 사용되는 기준은 특정 기능에 요구되는 장비의 유무, 시스템의 연산 속도를 기준으로 한다.

Function Map은 최적의 수행 성능 측정 결과를 보인 Function 이 존재하는 장치의 주소와 Linker ID로 구성된다. 장치의 주소에는 선정 결과에 따라 최적 Function이 디바이스의 것이라면 디바이스 IP가, 로봇의 것이라면 로컬 IP가 기록된다. Function Map 각 Linker 들이 어떤 장치에서 실행되는 것이 가장 효율적인지를 쉽게 참조하기 위해 유지되며 아래에서 설명할 Linker Watcher는 Function Map을 참조하여 Linker를 실행한다.

### 3.4 Linker Watcher

Linker Watcher 는 어플리케이션의 동작 흐름에 따라 Linker 의 실행을 결정하는 역할을 한다. 어플리케이션에 구현된 Linker 들은 순차적으로 실행될 수도 있고 특정 이벤트 발생에 따라 실행될 수도 있다. Linker 가 어플리케이션에서 실행되는 과정은 (그림 9)에 나타나 있다. 어플리케이션 흐름이 특정 Linker 가 위치하는 곳에 도달하면 Linker 는 자신을 Linker Watcher 에 등록한다. 이것은 실제 Linker 가 복사되는 것이 아니며 Linker ID 만 등록되는 것이다. Linker Watcher 는 등록된 여러 Linker 들의 Watch state 를 순차적으로 감시하며 실행 준비가 완료된 Linker 의 실행을 결정한다. Watch state 는 각 Linker 의 Linker state를 확인하여 입출력 데이터와 종속성의 대기 상태, 실행 준비 완료 등을 나타낸다. Linker state 의 종류는 Linker 에 설명된 바와 같다.

등록된 목록 중 실행 준비가 완료된 Linker를 확인하면




(그림 9) Linker Watcher 에 의한 Linker 실행

Linker Watcher 는 Function Evaluator 에 위치한 Function Map을 참조하여 해당 Linker 에 대해 최적의 효율을 나타낸 장치의 주소를 가져온다. 위의 그림에서는 Linker 3이 실행 준비 완료 상태이므로 Function Map을 참조하여 Linker 3을 로컬에서 실행하도록 결정한다. 실행이 결정된 Linker ID 는 Linker 가 실행될 장치의 주소와 함께 Frame Manager 로 보내지며 이후 해당 Linker ID는 Linker Watcher 목록에서 삭제된다.

Linker Watcher 에 등록된 목록의 감시는 목록을 순차적으로 반복 탐색하여 이루어지며 네트워크 전송이나 로컬에서의 Linker 루틴 실행에 영향을 받지 않도록 독립된 쓰레드로 작동한다.

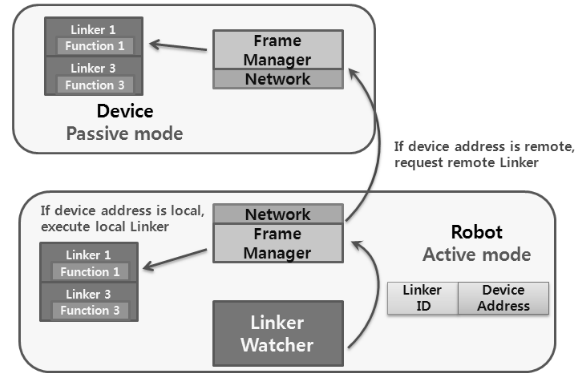
### 3.5 Frame Manager

Frame Manager 는 전체 프레임워크의 동작을 연결 및 조율하며 네트워크를 통해 다른 장치와 통신하는 일을 담당한다.

Frame Manager가 하는 일은 크게 세 가지가 있다. 먼저 Linker 의 최종적인 실행이다. Frame Manager는 실행 모드에 따라 동작이 조금씩 달라질 수 있는데, 먼저 액티브 모드를 기준으로 설명한다. Linker Watcher는 등록된 Linker 들의 상태를 감시하며 실행이 결정된 Linker의 ID와 실행 대상 장치의 주소를 Frame Manager로 보낸다. 이때 보내진 Linker 는 종속성 해결과 입력 데이터 준비가 완료된 상태이기 때문에 Frame Manager는 이 정보를 받아 대상 장치가 로컬이라면 자신의 Linker를 직접 실행하고, 대상 장치가 다른 디바이스라면 장치의 IP 주소로 Linker ID, 입력 데이터와 함께 실행 요청 메시지를 보낸다. 이 과정은 아래의 (그림 10)에 표현되어 있다.

Frame Manager가 패시브 모드에서 동작할 때는 어플리케이션의 실행 주체가 아니므로 앞에서 언급했듯 Linker Watcher와 Function Evaluator 는 동작하지 않는다. 따라서 패시브 모드에서는 로봇에서 액티브 모드로 동작하는 Frame Manager 가 전송한 Linker ID와 입력 데이터를 수신하여 디바이스 자신에게 존재하는 Linker만 단순히 실행하게 된다.

다음은 입출력 데이터의 전달과 Linker state 의 업데이트이다. 역시 엑티브 모드를 기준으로 설명한다. Linker 의 실행에 필요한 입출력 데이터는 개별 Linker 의 입출력 데이터 버퍼에 존재하는데, Frame Manager는 이 버퍼를 직접 접근하고 관리한다. 위 그림을 예로 들어보자. Frame Manager



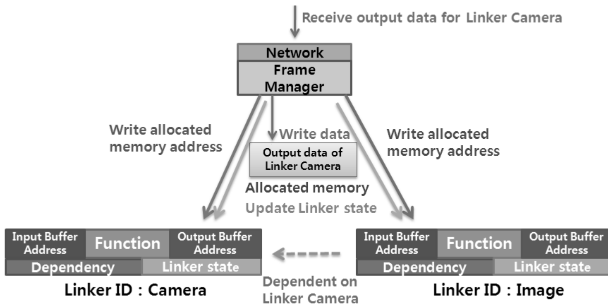

(그림 10) Frame Manager 의 동작

은 디바이스에 Linker 1 의 실행을 요청했으므로 Linker 1 의 출력 데이터는 네트워크를 통하여 로봇으로 전달된다. 이때 로봇의 Frame Manager 는 출력 데이터의 전체 크기만큼 메모리를 할당한 후 할당받은 메모리에 네트워크를 통해 수신된 출력 데이터를 모두 기록한다. 데이터 수신이 종료되면 로컬의 Linker 1 내부에 존재하는 출력 데이터 버퍼 주소에 실제 수신한 메모리의 주소를 기록한다. 즉 개별 Linker 의 실제 입출력 데이터 버퍼가 존재하는 메모리 공간은 Frame Manager 에 의해 결정되며 Linker 들은 이 공간의 주소에 대한 정보를 소유하게 된다. Frame Manager는 입출력 데이터 송수신 과정에서 각 Linker 의 Linker state 의 업데이트 종속성 검사도 병행한다. 설명한 바와 같이 네트워크를 통한 출력 데이터 수신이 완료되었다면 해당 Linker 의 Linker state를 출력 데이터 완료로 변경한다. 이때 종속성의 검사도 병행하여 만약 후속 링크가 존재한다면 후속 Linker 의 입력 데이터 버퍼 주소에 선행 Linker 의 출력 데이터 버퍼 주소를 입력하고, 후속 Linker 의 Linker state 도 입력 데이터 준비 완료로 업데이트한다.

Linker 의 입력 데이터를 읽어 들일 경우도 마찬가지로 Frame Manager는 해당 Linker의 입력 데이터 버퍼 주소에 접근하여 실제 입력 데이터가 존재하는 공간에 접근한다. 이 과정을 다음의 (그림 11)로 예를 들어 설명한다.

(그림 11)에는 Linker ID : Image 와 Linker ID : Camera 의 두 개 Linker 가 존재한다. 프레임워크가 로봇에서 동작하는 상황에서 Image 는 Camera 의 출력 영상을 받아서 실행되는 종속 관계이다. 로봇에는 카메라가 없다고 가정하여 Camera 는 원격지의 카메라가 장착된 디바이스에 실행을 요청한 상태이다.

Camera 의 출력 데이터가 네트워크를 통해 로봇에 수신되면 로봇의 Frame Manager는 새로 메모리를 할당받아 이 데이터를 기록한다. 데이터가 수신되는 동안 Image 의 Linker state 는 Input data waiting, Camara 는 Output data waiting 상태이며 이후 데이터가 모두 수신되면 Frame Manager는 Camara 의 출력 데이터 버퍼 주소에 실제 데이터를 수신한 메모리 주소를 기록하고 동시에 Camara 의 Linker state 도 출력 데이터 완료 상태로 업데이트 한다.




(그림 11) Frame Manager의 입출력 데이터 관리

이 상황에서 Camera는 실제 출력 데이터가 기록된 버퍼 공간을 직접 소유하고 있는 것은 아니며 버퍼에 접근할 수 있는 주소값을 가지고 있는 상태이다.

다음으로 Frame Manager 는 Camara 의 종속성 검사를 수행하는데, Image 가 Camara 에 종속되어 있으므로 Image 의 입력 데이터는 Camera 의 출력 데이터와 동일하다. 따라서 Frame Manager 는 Image 의 입력 데이터 버퍼 주소에 데이터를 수신한 메모리 주소를 기록한 후 Image 의 Linker state 를 입력 데이터 준비 완료 상태로 업데이트한다. 즉, Camera 의 출력 데이터 버퍼 주소와 Image 의 입력 데이터 버퍼 주소는 동일한 메모리 주소를 가리키고 있다.

이 단계에서 Image 는 실행 준비가 완료된 상태이며 Camera 는 실행이 완전히 종료되었다. 할당된 메모리 공간은 모든 종속성이 해결된 최종 Linker 의 실행이 종료될 때까지 유지되며 그림의 경우에서 Image 의 실행이 끝나면 실제 데이터가 위치한 메모리는 해제된다.

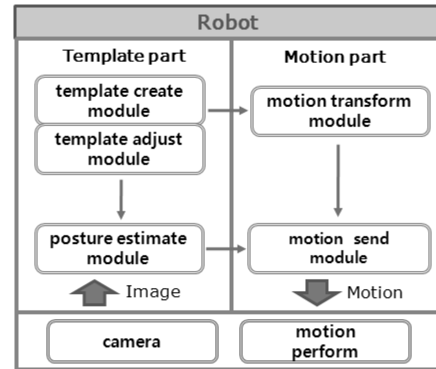
패시브 모드로 동작할 경우에는 단지 엑티브 모드의 요청에 따라 개별 Linker 만 실행될 뿐이므로 엑티브 모드에서 전송한 입력 데이터는 Linker 의 연산 종료 후 폐기된다.

Frame Manager 의 마지막 기능은 Function 의 성능 측정 평가 결과를 전송하고 Function Evaluator 에 업데이트 하는 것이다. 패시브 모드에서는 각 Linker 가 실행될 때 마다 측정된 평가 결과를 엑티브 모드 Frame Manager 로 전송하고, 엑티브 모드에서는 패시브 모드에서 보내온 평가 결과를 Function Evaluator 에 업데이트한다.

## 4. 프레임워크의 동작

프레임워크를 구성하는 개별 요소들의 동작을 상세하게 살펴보았다. 이제 요소들이 결합된 전체 프레임워크 흐름을 실제 어플리케이션의 예를 들어 설명한다.

어플리케이션은 로봇을 이용한 체조 학습 시스템이다. 이 어플리케이션은 로봇이 체조 동작의 특정 자세를 취한 후 사람이 이 자세를 따라하면, 로봇에 내장된 카메라로 사람의 영상을 입력받아 자세의 유사도를 측정한다. 측정결과 올바른 자세로 판단될 경우 다음 체조동작으로 진행하고 그렇지 않다면 사용자에게 부정확한 자세라는 것을 알린 후 현재 체조동작을 재시도하여 사용자가 올바른 체조 동작을 하도록 유도한다. 어플리케이션에 관한 자세한 내용은 [3, 4]에 있다.




(그림 12) 체조 학습 시스템의 모듈별 구조

어플리케이션의 구조를 기능별로 표현하면 (그림 12)와 같다. 어플리케이션에는 카메라를 통한 영상 획득과 영상처리를 위한 연산 능력이 요구되지만, 로봇의 종류에 따라 요구 조건을 충족시키지 못할 수도 있다. 이런 경우 프레임워크를 적용하여 각 기능들을 다른 장치들에 분산시키면 어플리케이션의 동작이 가능해진다.

그림에서 모듈은 곧 기능요소를 의미하므로, 프레임워크 적용 시 하나의 모듈은 하나의 Function 을 포함한 Linker 로 정의된다. 정의된 모든 Linker 의 기능과 Linker 사이의 종속성은 <표 2>와 (그림 13)에 나타나 있다.

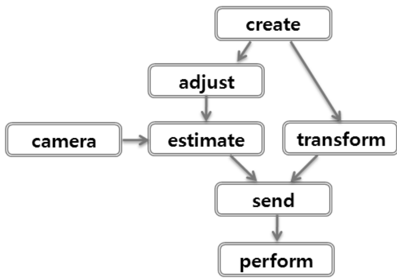
각 Linker 들은 입출력 데이터의 흐름에 따라 종속성으로 연결되어 있다. 화살표의 방향은 선행 Linker 의 출력 데이터가 후속 Linker 의 입력 데이터로 전달되는 종속성을 나타내며, create 와 camera 는 종속성이 없다.

어플리케이션의 동작 환경은 한 대의 로봇과 두 대의 PC 가 홈 네트워크로 연결되어 있다. 로봇은 낮은 연산 능력을 보유하고 있고 카메라가 존재하지 않는다. 카메라는 두번째 PC 에만 장착되어 있다.

로봇의 어플리케이션이 실행되면 PC 로 Wake-up 메시지가 전달되고 PC 에서는 로봇에 탑재된 것과 동일한 어플리케이션을 패시브 모드로 실행한다. 로봇과 PC의 어플리케이션에는 최초 실행을 위해 각자가 가진 Linker 의 수행 능력이 기본값으로 설정되어 있다. 수행 능력은 특정한 하드웨어의 유무와 연산 능력을 기준으로 어플리케이션이 실행

〈표 2〉 체조 학습 시스템의 모듈별 기능

| Linker ID | 기능 |
|---|---|
| create | 체조 자세 템플릿 생성 |
| adjust | 체조 자세 템플릿 조절 |
| estimate | 획득한 사용자 영상으로 자세 생성 |
| transform | 체조 템플릿을 로봇 모션 데이터로 변환 |
| send | 모션 실행 결정 |
| camera | 사용자의 영상 획득 |
| perform | 모션을 실행 |




(그림 13) 체조 학습 시스템의 모듈 종속성

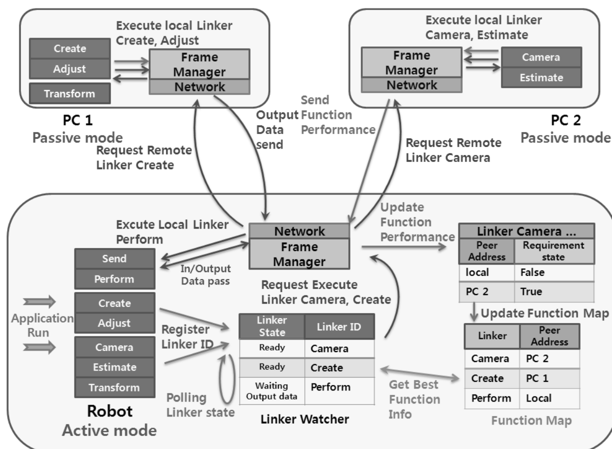
될 장치의 특성에 따라 사용자가 직접 설정해 주어야 한다.

어플리케이션을 엑티브 모드로 실행하는 주체는 로봇이므로 패시브 모드로 동작하는 두 대의 PC는 기본값으로 설정된 수행 능력을 즉시 로봇으로 전송한다.

로봇의 Function Evaluator는 PC들이 보낸 결과들을 종합한 후 Function Map을 작성하여 어플리케이션을 진행한다. 로봇의 Function Evaluator 는 최초 실행 시 장치들에 기본으로 설정된 수행 능력 측정값을 사용하지만, 어플리케이션의 진행에 따라 Linker 들이 장치에서 최소 한번이라도 실행된 후에는 실제 수행된 능력 측정값을 전송받아 사용한다. 이후 어플리케이션이 동작하는 과정은 (그림 14)와 같다.

최초 어플리케이션은 사용자의 영상을 얻기 위해 Camera 를 Linker Watcher 에 등록한다. 이 과정에서는 Linker ID 만 등록할 뿐 Linker 내의 루틴은 실행하지 않으므로 어플리케이션 흐름은 즉시 다음의 Estimate 로 이동하여 등록한다. 동시에 미리 정의된 체조 동작을 읽어들이는 Create 와 로봇 모션을 실행하는 Perform 도 등록되었다. Linker 가 Linker Watcher 로 등록되는 것은 실제 루틴이 실행되는 과정이 아니므로 어플리케이션이 실행된 즉시 존재하는 모든 Linker 가 등록된다. 실제 루틴의 실행은 Framer Manager 가 종속성에 따른 Linker state를 판단하여 이루어지게 된다.

Frame Manager와 Linker Watcher 는 독립된 쓰레드로 동작하고 있으므로 등록된 Linker 들의 상태를 어플리케이션 흐름과 관계없이 감시하고 실행을 결정한다. Camera 와




(그림 14) 전체 어플리케이션의 동작

Create 는 선행 종속 Linker 가 없기 때문에 즉시 실행 가능한 상태로 등록되었지만 Perform 은 send 에 종속되어 있으므로 send 의 출력 데이터 대기 상태에 놓여 있다. 이제 Linker Watcher는 Linker Camera, Create 에 대해 Function Map을 참조하여 최고의 성능 평가를 보인 장치를 선정하고 해당 장치에 실행을 의뢰한다. 그림의 Function Map에서 PC 2에만 카메라가 존재하므로 Camera 는 PC 2(원격) 에 실행을 요청한다. 마찬가지로 Create 는 수행 능력 기준이 연산 처리 시간이므로 연산 속도가 가장 빠른 PC 1에 실행을 요청한다.

PC 2 에서 실행이 종료된 Camera 의 영상 출력 데이터가 로봇에 도착하면 Frame Manager는 새로 할당받은 메모리에 수신한 데이터를 저장한 후 이 메모리의 주소를 Camera 의 출력 데이터 버퍼 주소에 기록하고 Camera 의 Linker state를 출력 데이터 완료로 업데이트한다. 동시에, Camera 는 Estimate 의 선행 Linker 이므로 Estimate 의 입력 데이터 버퍼 주소에도 동일한 메모리 주소를 기록하고 Estimate 의 Linker state를 입력 데이터 완료로 업데이트한다. 이제 Camera 의 실행은 완전히 종료되었으므로 Camera 는 Linker Watcher 목록에서 삭제되지만, Cemera 의 출력 데이터는 종속된 Estime 에서 계속 사용하게 되므로 할당된 출력데이터 메모리는 Estimate 의 실행 종료까지 해제되지 않는다. PC 1에서 실행된 Create 도 같은 과정을 거친다. 현재까지의 과정에서 로봇의 Frame Manager 는 Camera 와 Create 각각에 대해 두 개의 출력 데이터 메모리를 할당했고 두 Linker 모두 후속 Linker 가 존재하므로 어떤 메모리도 해제되지 않았다.

Camera와 Create 가 PC에서 실행이 종료되면 당회 실행 과정에서 재 측정된 성능 측정 평과 결과가 로봇의 Frame Manager 로 도착하고 이것은 Function Evaluator 에 통보된다. 변동 사항이 발생하면 Function Evaluator 는 재평가 과정을 통해 Function Map을 업데이트한다. Camera 는 PC 2에만 존재하므로 그대로 유지되고 Create 는 당회 연산 시간에 따라 로봇과 두 대의 PC 중 연산 속도에 대한 기댓값이 가장 높은 장치를 선택하게 된다.

이어서 Estimate, Transform 과 Adjust 도 동일한 과정을 거쳐 Function Map 상에서 가장 좋은 성능 측정을 보인 장치에서 실행된다. Transform 과 Estimate 의 실행이 끝나면 두 Linker 의 후속 Linker 인 Send 가 실행된다. Send 는 어플리케이션의 최초 실행 시 Linker Watcher 에 등록되었지만 종속성을 충족시키지 못해 이 시점까지 Input data waiting 상태로 대기했다. Send 는 Estimate 와 Transform 의 출력 데이터를 바탕으로 로봇의 체조 모션 실행을 결정하고 Send 의 후속 Linker 이자 실제로 모션실행을 담당하는 Perform 이 최종적으로 로봇에서 실행된다.

## 5. 프레임워크의 활용

논문에서 제안한 프레임워크의 목적은 자원의 제약이 있

을 수 있는 로봇에서 주변 장치와의 협업을 통하여 어플리케이션을 원활하게 실행시키는 것이다. 이때 어플리케이션의 실행에 소요되는 자원은 협업을 통해 충족할 수 있지만, 이를 위해서는 먼저 프레임워크가 로봇에서 원활하게 동작할 수 있어야 한다. 실제 본 논문에서 사용된 MIRAI SPC-101C 로봇은 64MB 의 메모리와 133Mhz 의 연산능력만을 가지고 있으므로 프레임워크 동작에 소모되는 자원 또한 중요한 고려사항이 된다.

네트워크를 통해 다른 장치에 접근하여 기능을 제어하고 이용하는 기술로는 Microsoft 사의 UPnP 와 Sun Systems 의 JINI[8] 를 들 수 있다. UPnP 는 네트워크 환경에서 표준화된 방법으로 장비간의 연결을 제공하며 개발이 쉽다는 장점이 있지만 범용의 인터넷 기반 프로토콜을 다수 사용하기 때문에 사용 모듈의 크기와 처리에 따른 시스템의 부담이 있을 수 있다. 또한 JINI 의 경우 연결된 모든 장치들에 자바 가상 머신이 탑재되어 있어야 한다.

프레임워크는 가상 머신과 같은 특정한 실행 환경이 필요하지 않고, 네트워크를 통한 다른 장치로의 접근에 반드시 필요한 기능과 프로토콜만을 정의하였으므로 최소한의 연산과 메모리 자원만을 소비한다. 이로 인해 가전기기의 제어와 같은 범용적인 분야에 대해서는 표준화된 기술에 비해 적용성이 떨어질 수 있지만, 최소한의 자원만으로 동작이 가능하므로 로봇 어플리케이션의 협업에 있어서는 장점으로 작용한다. 또한 프레임워크는 사용자가 어플리케이션을 자유롭게 구현할 수 있도록 지원한다. 프레임워크의 협업 동작은 다른 장치에 탑재된 Function을 호출함으로써 이루어지는데, 이것은 RPC(Remote Procedure Call) 기술의 동작과 유사하다. RPC는 메시지 전달 메커니즘으로 동작하는 다른 컴퓨터에 있는 프로시저를 쉽게 호출할 수 있게 하는 웹 서비스 접근 방법으로, 개발이 쉽고 서로 다른 데이터 모델이나 프로세스를 가진 프로그램들을 연결할 수 있다. 하지만 RPC에서 사용되는 메시지는 사용자가 임의로 정의할 수 없으며 정형화된 프로시저 이름과 매개변수를 사용해야 하기 때문에 개발 유연성, 유지 보수 등에 제약이 생길 수 있다. <표 3>과 <표 4>는 UPnP 와 JINI, RPC 기술을 특징별로 프레임워크와 비교한 것이다. 이 기술들은 네트워크를 통한 다른 장치와의 협업이라는 측면에서 공통점이 있지만, 기술의 상세한 목적은 모두 다르기 때문에 프레임워크와 수평적인 비교는 불가능하다. 프레임워크는 수행 기능적 측면에서 크게 두 가지로 나눌 수 있는데, 각각은 UPnP 와 JINI, 그리고 RPC 기술과 유사한 기능을 가진다. 즉, 네트워크를 통한 다른 장치의 자원 접근은 UPnP, JINI 기술과 비교하고 기능 호출을 통한 협업의 수행은 RPC 기술과 비교하였다.

프레임워크에서 사용자가 구현한 어플리케이션 루틴은 프레임워크에 함수 포인터 형태로 등록되고 루틴의 입출력 데이터 형식과 크기를 모두 사용자가 정의할 수 있다. 즉, 프레임워크 인터페이스는 사용자가 자유롭게 작성한 어플리케이션의 각 함수들과 일대일로 대응되는 래퍼(wrapper) 클래스 형태로 제공된다. 따라서 어플리케이션 개발 시 프레임워크의 구조적 특성에 따른 설계 변경이 필요하지 않고 사용자 개인의 개발 방식에 따라 직관적이고 간단하게 사용할 수 있다. 4장에서 설명한 체조 학습 시스템은 UPnP나 JINI를 적용하여 네트워크상에서 다른 장치와 상호 작용하게는 할 수 있지만, 프로토콜 스택에 어플리케이션 기능 정의를 위한 구현이 추가적으로 필요하며 기능 루틴들도 따로 모듈화 되어야 한다. UPnP나 JINI 등의 기술은 단순한 기능을 네트워크로 동작시키는 것에는 편리하게 적용 가능하지만, 구현을 위해 기본적으로 유지되어야 하는 틀이 프레임워크와 비교하여 엄격하고 무겁기 때문에 어플리케이션의 세부 기능별로 긴밀하게 협업하는 구조에는 적용이 용이하지 않다. 프레임워크는 네트워크가 연결된 임베디드 환경에서 협업을 통한 성능 향상이라는 제한된 범위의 목적을 가지고 설계되었기 때문에 어플리케이션 단위 개발의 유연성, 협업 동작에 소모되는 자원의 양, 다양한 언어에 적용할 수 있는 범용성, 네트워크 상의 장치 중 최적의 효율을 얻을 수 있는 대상을 선택할 수 있는 면에서 가장 효율적이다.

〈표 3〉 프레임워크와 UPnP/JINI 의 비교

| Performance | |
|---|---|
| Framework | 최소 요구 기능만이 구현된 소수의 API를 사용하여 빠름 |
| UPnP | API를 이용한 언어별 구현에 따라 다름 |
| JINI | 최적화된 자바 프로세스를 이용하지 않는 한 느림 |
| Cost | |
| Framework | 하나의 어플리케이션을 간단한 API를 이용하여 작성 |
| UPnP | 모든 장치와 플랫폼에 API를 이용하여 작성 |
| JINI | 모든 장치에 자바 가상 머신 요구 |
| Easy to use | |
| Framework | 쉽게 사용 가능 |
| UPnP | API 구현에 따라 다름 |
| JINI | 쉽게 사용 가능 |
| Independency | |
| Framework | 운영체제와 구현 언어 |
| UPnP | 운영체제와 구현 언어 |
| JINI | 운영체제 |

〈표 4〉 프레임워크와 RPC 의 비교

| Feature | Frame work | RPC |
|---|---|---|
| Easy to use | ○ | ○ |
| Developer defined message types | ○ | × |
| Developer defined data types | ○ | × |
| Can specify recipient | ○ | × |
| Detailed fault handling | × | ○ |

프레임워크는 집안에서 다양한 오락, 교육용으로 사용될 수 있는 홈 네트워크 환경의 가정용 로봇 어플리케이션을 대상으로 한다. 현재 홈 네트워크 환경에서 작동하는 소규모 자원을 가진 가전기기도 점차 늘어나고 있는 추세이므로 이들 스마트 가전기기는 프레임워크의 협업 동작을 위한 좋

은 대상 장치가 될 수 있다. 또한 프레임워크가 목적으로 하는 가정용 로봇은 네트워크가 가능한 소규모 연산장치를 의미하기 때문에 임베디드 가전기기 상에서 동작하는 어플리케이션에도 같은 방법으로 적용될 수 있으므로 전체적인 홈 네트워크 환경의 활용도를 높일 수 있다.

## 6. 결 론

각종 산업과 엔터테인먼트 분야에서 로봇은 더 이상 희귀한 장치가 아니게 되었으며 로봇을 활용한 어플리케이션의 요구도 높아지고 있다. 하지만 로봇마다 천차만별인 컴퓨팅 자원의 차이로 인해 한 로봇에서 다양한 어플리케이션을 실행하기 힘들다. 이러한 문제를 극복하기 위해 본 논문에서는 주변 컴퓨팅 장치들의 자원을 이용해 어플리케이션의 자원 요구를 충족시킬 수 있는 프레임워크를 제안했다. 프레임워크는 협업을 위한 다른 범용 기술들과 달리 로봇 어플리케이션의 협업을 위해 설계되었으므로 매우 제한적인 자원을 가진 로봇에서도 동작이 가능하다.

프레임워크는 장치들 사이의 협업에 요구되는 공통 구성 요소들을 정의하여 협업 어플리케이션의 개발에 사용될 수 있는 모델을 제공한다. 또한 어플리케이션 개발 단계에서 적용 가능한 인터페이스를 제공하며, 어플리케이션 기능을 모듈별로 프레임워크에 탑재하여 로봇과 다른 장치들 사이의 협업을 위한 기능을 제공한다.

프레임워크의 동작을 위해 로봇을 이용한 체조 학습 어플리케이션에 프레임워크를 적용한 후 어플리케이션이 동작하는 상세한 과정을 보였다.

## 참 고 문 헌


[1] Duckki Lee, Tatsuya Yamazaki, Sumi Helal, “Robotics Companions for Smart Space Interactions,” IEEE Pervasive Computing, pp.78-84, April-June, 2009.

[2] Kim, H, Cho, Y.-J, Oh, S.-R, “CAMUS:A middleware supporting context-aware services for network-based robots,” IEEE Workshop on Advanced Robotics and Its Social Impacts, Nagoya, Japan, 2005.

[3] A.M. Brent, N. Toby, T. Charlie, and D.W. Mark, “Home networking with Universal Plug and Play,” IEEE Communications Magazine, Vol.39, No.2, pp.104-109, Dec., 2001.

[4] http://www.upnp.org

[5] Andrew D. Birrell, Bruce Jay Nelson, “Implementing Remote Procedure Calls,” ACM Transactions on Computer Systems, Vol.2, No.1, pp.39-59, Feb., 1984.

[6] Chang-Mug Lee, Oh-Young Kwon, “Posture Recognition for Physical Training System with Adjusted Edge Template,” The 31th KIPS Spring Conference, pp.110-113, 2009.

[7] Chang-Mug Lee, Oh-Young Kwon, “Automatic Motion Creating from the Posture Template for the Robot Physical Training System,” ICHIT 2009.

[8] http://www.jini.org/

[9] Jin Nakazawa Tokuda, H. Edwards, W.K. Ramachandran, U, “A Bridging Framework for Universal Interoperability in Pervasive Systems,” Distributed Computing Systems, 2006.

[10] H. Kim, H. Yang, R. Bose and A. Helal, “Enhancing the Sentience of URC using Atlas Service-Oriented Architecture,” Proceedings of 8th International Workshop on Human-friendly Welfare Robotic Systems (HWRS 2007), Korea, October, 21-23, 2007.

[11] G. Biggs, B. Macdonald, “A Survey of Robot Programming Systems,” in proceedings of the Australasian Conference on Robotics and Automation, December 1-3 2003, http://www.araa.asn.an/acra/acra2003/papers/27.pdf

[12] S. Ahn, K. Lim, J. Lee, H. Ko, Y. kwon and H. Kim, “UPnP Robot Middleware for Ubiquitous Robot Control,” The 3rd International Conference on Ubiquitous Robots and Ambient Intelligence, Oct., 2006.

[13] Nader Mohamed, Jameela Al-Jaroodi and Imad Jawhar, “Middleware for Robotics : A Survey,” IEEE International Conference on Robotics, Automation and Mechatronics, pp.736-742, Sep., 2008.

[14] Y. Ha, J. Sohn, Y. Cho and H. Yoon, “Towards Ubiquitous Robotics Companion: Design and Implementation of Ubiquitous Rebotics Service Framework,” ETRI Jounal, Vol.27, No.6, pp.666-676, 2005.

이 창 묵

e-mail : atlantis13@kut.ac.kr

2008년 한국기술교육대학교 정보기술공학부(학사)

2008년~현 재 한국기술교육대학교 전기전자공학과

관심분야 : 고성능 컴퓨터 구조, 홈 네트워킹 미들웨어

권 오 영

e-mail : oykwon@kut.ac.kr

1990년 연세대학교 이과대학 전산과학과(이학사)

1992년 연세대학교 컴퓨터과학과(이학석사)

1997년 연세대학교 컴퓨터과학과(공학박사)

1997년~2000년 ETRI 네트워크컴퓨팅연구부 분산컴퓨팅 연구팀 선임연구원

2000년~현 재 한국기술교육대학교 컴퓨터공학부 부교수

관심분야 : 병렬화 컴파일러, 고성능 컴퓨터 구조, 그리드 컴퓨팅, 홈 네트워킹 미들웨어