Microsoft ROBOTICS STUDIO

CCR 및 DSS 사용자 가이드

CCR 및 DSS 사용자 가이드는 아래의 항목으로 구성됩니다.

목차

1. 시작하기

런타임 소개(Runtime Introduction)

마이크로소프트 Robotics Studio 는 학생과 동호인 그리고 상용 소프트웨어 개발자가 손쉽게 로봇 애플리케이션을 개발하기 위한 윈도우 기반 개발 환경입니다. 그 런타임은 PC 에 직접 연결된 (시리얼 포트, 블루투스, USB 등) 로봇 에서부터 온보드 PC 를 가진 로봇, 시뮬레이션환경 하에서 작동하는 로봇까지 다양한 로봇 하드웨어를 지원합니다. 또한 런타임은 기본적인 센서 입력의 관측(observation)에서 연결에 의한 구동(drive-by-wire) 과 원격 작업(remote presence) 및 자동화된 업무, 다수의 자동화된 로봇들의 협력 등 의 다양한 종류의 로봇 애플리케이션을 지원하도록 디자인되었습니다.

마이크로소프트 Robotics Studio 런타임 환경은 다음과 같은 다양한 요구조건을 가진 애플리케이션들의 개발을 위해 디자인 되어 왔습니다:

1. 애플리케이션 작동 중 상태(state)를 모니터 하고 각 컴포넌트들과 상호작용할 수 있어야 한다.
2. 애플리케이션 작동 중 컴포넌트들을 검색(discover), 생성, 종료, 재시작 할 수 있어야 한다.
3. 다양한 센서 입력을 동시에 처리함부터 태스크(task)들 간의 의도되지 않은 충돌 위험 없이 태스크들의 입력을 오케스트레이션(orchestrate) 할 수 있어야 한다.
4. 로컬과 네트웍을 통한 로봇 애플리케이션 둘 모두 자동으로 처리할 수 있어야 한다.
5. 런타임은 다양한 종류의 환경에서 실행가능 하도록 가벼워야 한다.
6. 애플리케이션 환경은 다양한 종류의 하드웨어 소프트웨어 환경들을 모두 다룰 수 있도록 확장가능하고 유연해야 한다

마이크로소프트 Robotics Studio 런타임 은 .NET framework 을 사용하는 다양한 종류의 윈도우 플랫폼 과 .NET Compact Framework 를 사용하는 Windows CE 를 지원합니다. 런타임은 다음의 두 주된 요소로 구성됩니다.

동시처리 및 조정 기술 (CCR) 은 고도의 동시성, 쓰레스, 락, 세마포어를 사용하지 않는 메시지 조작을 통한 강력한 오케스트레이션을 지원하는 메시지 기반 프로그래밍 모델입니다. CCR 은 비동기 오퍼레이션(asynchronous operations) 관리와 동시성 처리 및 병렬처리 하드웨어와 부분적 실패(partial failure)를 처리를 손쉽게 하는 서비스기반 애플리케이션입니다. 또한 애플리케이션 디자인에서 소프트웨어 모듈이나 컴포넌트가 느슨히 연결(loosely coupled)되게 합니다. CCR 은 각각이 컴포넌트들이 런타임 환경과 다른 컴포넌트들에 관한 최소한의 가정하에서 독립적으로 개발될 수 있게 합니다. 이 접근법은 디자인 단계에서부터 프로그램에 관한 개념을 바꾸며 보다 손쉽게 동시성, 실패, 고립화(isolation)를 일관되게 처리할 수 있게 합니다. CCR 은 .NET 2.0. Common Language Runtime (CLR)을 위한 여러 프로그래밍 언어를 지원합니다.

분산화된 소프트웨어 서비스 (DSS)는 기존의 웹기반 아키텍쳐(REST 로 알려진)와 여러 웹서비스 아키텍쳐의 주요 요소를 결합한 가벼운 서비스 기반 애플리케이션 모델입니다. 이 애플리케이션 모델은 서비스를 상태(state) 와 operation 들의 집합으로 정의하며, 구조화된 데이터 조작, 이벤트 알림(notification)과 서비스 조립을 추가한 HTTP 기반 애플리케이션 모델의 확장으로 REST model 하에서 DSS 를 구성합니다.

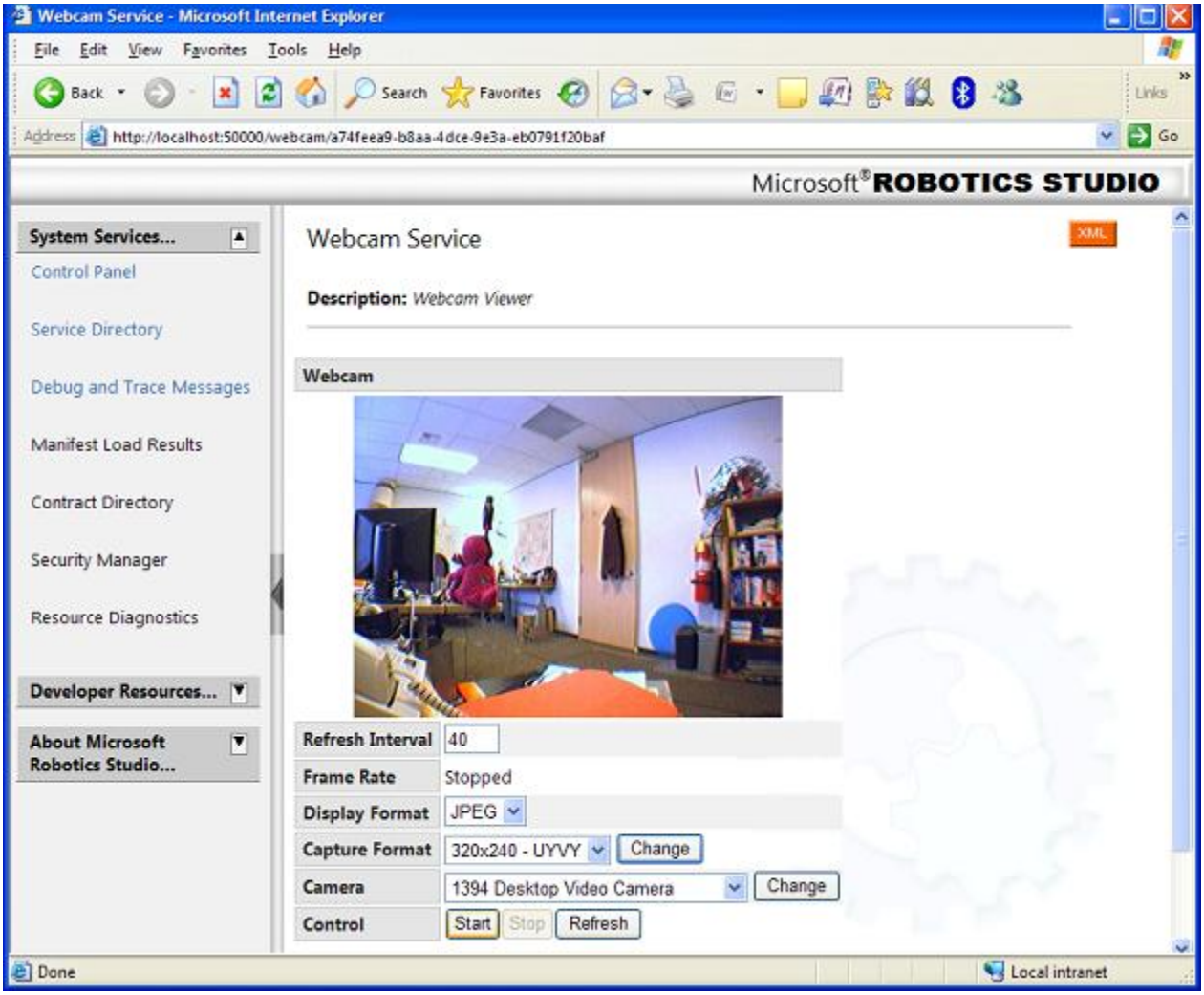
DSS 의 주된 목표는 단순성, 상호운용성과 느슨한 연결(loose coupling)을 증진함 입니다. 이는 특히 서비스가 같은 노드 혹은 네트웍 상에서 실행 됨에 관계없이 서비스들의 구성을 통해 애플리케이션을 생성하는 것을 가능하게 합니다. 이는 매우 광범위한 애플리케이션들을 제작하는데 매우 유연하면서 단순한 플랫폼입니다. DSS 는 HTTP 와 서비스들간의 상호작용을 위해 DSSP 를 사용합니다. DSSP 는 구조화된 상태(state)의 조작과 이 상태(state)의 변화에 의한 이벤트 모델을 위한 가벼운 SOAP 기반 프로토콜입니다. DSSP 는 서비스들을 조작하고 subscribing 하기 위해 사용되며 단순하고, 상태(state)기반 애플리케이션 모델을 제공하는데 있어 HTTP 와 호환됩니다.

DSS 런타임은 CCR 기반으로 제작되었으며 그 외 마이크로소프트 Robotics Studio 의 다른 컴포넌트에는 의존하지 않습니다. 또한 서비스 생성, 검색, 로깅, 디버깅, 모니터링과 보안을 위한 인프라스트럭쳐 서비스 집합과 서비스들 관리하기 위한 호스팅 환경을 제공합니다.

런타임과의 상호작용

어떤 서비스를 작성하지 않고도 단순히 시작메뉴에서 Run DSS Node 를 선택해 마이크로소프트 Robotics Studio 런타임을 실행해 볼 수 있습니다. 모든 런타임 서비스들은 자동적으로 HTTP 를 통해 보여지며 웹브라우저로 접근가능합니다. 각 서비스들은 구조화된 데이터를 교환함을 통해

통신하도록 디자인 되어져 있지만, 사용자와의 상호작용 또한 중요합니다. 서비스 들은 주로 XML 로부터 생성된 HTML 형식의 UI 를 제공하지만, WinForm, 오디오, 비디오, 이미지 및 마이크로소프트 Robotics Studio 시뮬레이션 엔진등 다양한 형식의 UI 를 제공할 수 있습니다.



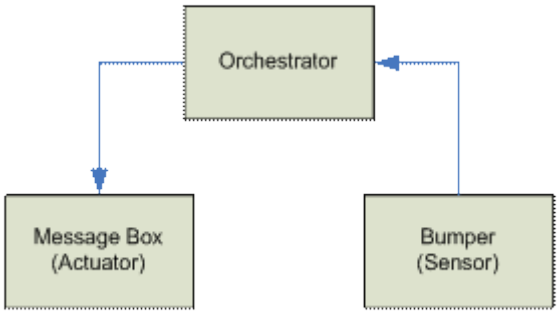
데이터와 자료 구조를 비디오와 함께 보여주는 웹캠 서비스

어떻게 CCR과 DSS가 포함된 마이크로소프트 Robotics Studio 런타임을 사용하는 지에 대한 자세한 사항은 런타임 문서와 특히 다양한 관점에서 로봇 애플리케이션 제작을 보여주는 샘플과 튜토리얼들을 읽어보십시오. 마이크로소프트 비주얼 프로그래밍 언어 언어(Visual Programming Language)와 마이크로소프트 비주얼 시뮬레이션 환경(Visual Simulation Environment)을 포함한 다른 컴포넌트에 대해서는 Microsoft Robotics Studio 소개를 읽어보십시오.

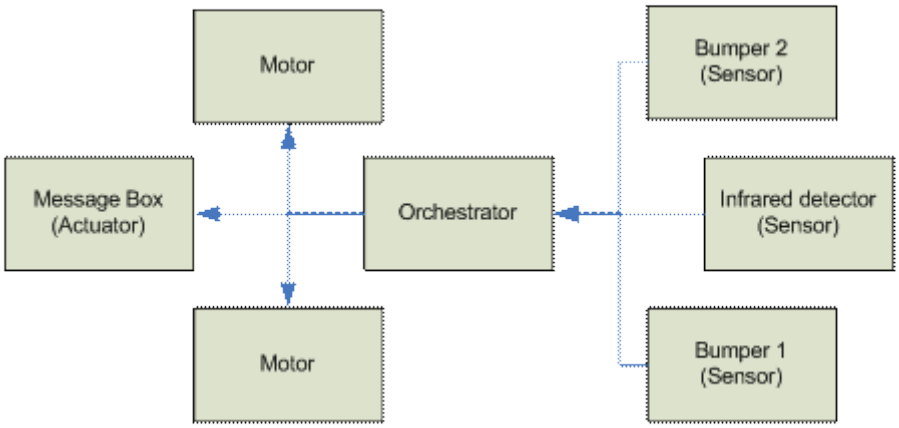
애플리케이션 모델 소개(Application Model Introduction)

로보틱스 애플리케이션들의 주된 일은 주어진 애플리케이션을 수행하기 위해 다양한 종류의 센서 입력을 처리하고 이에 대한 반응으로 엑츄에이터들의 집합을 오케스트레이션하는 것입니다.

일례로 다음의 데이터플로우 다이어그램에서 보여지는 로보틱스 애플리케이션은 센서로 단순한 범퍼(simple bumper)가 눌러지면 엑츄에이터로 메시지 박스에 메시지를 표시하는 것이며, 이때 오케스트레이터(orchestrator)는 두 부분을 연결하는 것입니다.



이 orchestration 은 단순하지만 애플리케이션이 커짐에 따라 센서와 엑츄에이터의 종류 및 복잡한 작업을 수행을 위한 오케스트레이터 또한 가 다양해질 것 입니다. 보다 복잡한 로보틱스 애플리케이션은 다음과 같습니다:



위의 데이터플로우에서 보면, 센서와 엑츄에이터는 처음 예와 비슷하지만, orchestrator 는 더 많은 컴포넌트들을 관리해야 합니다. 또한 컴포넌트(이 또한 오케스트레이터 일수 있습니다)의 증가에 따라 오케스트레이터 또한 복잡해 졌으며, 다른 "Hello world" 애플리케이션과는 다른 다음의 특징이 있습니다:

1. 센서입력을 처리와 엑츄에이터 제어는 동시에 처리되어져야하며, 그렇지 않으면 중간에 센서가 무시되거나 엑츄에어터 동작이 제한됩니다.
2. 오케스트레이션과 구성(composition)은 애플리케이션의 매우 중요한 부분입니다. 특히 센서와 엑츄에이터의 수가 많아질수록 오케스트레이션은 복잡해 집니다.
3. 자동화되고 협력적인 오케스트레이션은 네트웍상에서 접근 가능한 분산될 수 있는 컴포넌트들을 필요로 합니다

이러한 요구조건을 만족하기 위해, 마이크로소프트 Robotics Studio 런타임은 높은 동시성, 기존 웹기반 아키텍쳐와 유연하지만 가벼운 분산 애플리케이션 모델을 사용한 웹서비스를 결합한 서비스 중심 애플리케이션 모델을 제공합니다. 자연히 이러한 애플리케이션 관점은 전통적인 로보틱스 컴포넌트인 모터, 범퍼들에 제한 되어 지지는 않습니다.

웹과 웹서비스의 결합(Merging Web and Web Services)

웹아키텍쳐의 주된 초점은 단순성, 상호호환성, 느슨한 연결(loose coupling)입니다. HTTP 는 단순하고, 상태(state)중심의 애플리케이션 모델인 "REST"의 장점을 지닌 애플리케이션을 가능하게 하는데 기여하였습니다. 웹기반 애플리케이션은 HTTP 를 통해 이러한 모델이 확장가능하며 상호운용성과 다양한 시나리오를 만족하는 유연성을 줌을 시연해왔습니다. HTTP 의 성공에도 불구하고 아직 다루어지지 않은 잘 알려진 몇 개의 관점이 존재하며, 특히 다음의 세가지는 웹애플리케이션이 급성장 하는 것을 제한하고 있습니다:

1. 구조화된 데이터 조작의 미지원. 허용되는 조작은 "리소스 레벨(resource level)"의 리소스에 대해서며, 리소스의 하부구조에는 해당되지 않는 것입니다. 이것은 서비스 상태(state)의 갱신(update)을 힘들게 하며 서비스가 상호작용하는 것을 제한합니다.
2. 이벤트 알림의 미지원(event notification). HTTP 는 요구/응답(request/response) 프로토콜이며 데이터를 subscriber 들에게 제공하는 이벤트 알림 모델을 지원하지 않습니다. RSS 와 같은 메커니즘이 매우 유용하지만, 아직 pull oriented 이며 구조화된 필터링 지원이 미흡합니다.
3. 위 다어어그램에서 보여진 바와 같이 컴포넌트 사이의 관계의 종류들을 표현하지 못합니다. 다시말해 특정 오케스트레이터가 특정 센서 및 엑츄에이터와 파트너 되었다는 것을 표현할 방법이 없습니다.

마이크로소프트 Robotics Studio 런타임은 REST 기반의 애플리케이션 모델을 제공하지만 서비스들간의 구조화된 데이터 조작, 이벤트 알림, 파트너 관리를 처리하는 웹서비스 들로 확장되었습니다. 이러한 방법으로 REST 모델을 확장함으로써 애플리케이션들은 기존의 REST 인프라스트럭쳐와 상호운용성을 잃지 않으면서 이벤트 알림 모델과 구조화된 데이터 조작의 장정을 가질 수 있습니다. 결과로 노드 내 혹은 네트웍 상에서 조직된 서비스들의 구성으로 만들어진 보다 상호작용적이고 동적인 애플리케이션이 가능해 졌습니다.

REST 로 구조화 된 데이터 조작 (Structured Data Manipulation in a REST World)

HTTP 는 XML 이 대중적이기 이전에 정의되었고 구조화된 존재로 리소스를 다루지 안았습니다. 그 결과 HTTP 는 GET 과 PUT 같이 리소스 상태(state) 전체를 바꾸는 방법만을 제공합니다. URI query notation(예로 "http://www.example.com?keyword=value")이 리소스의 부분적인

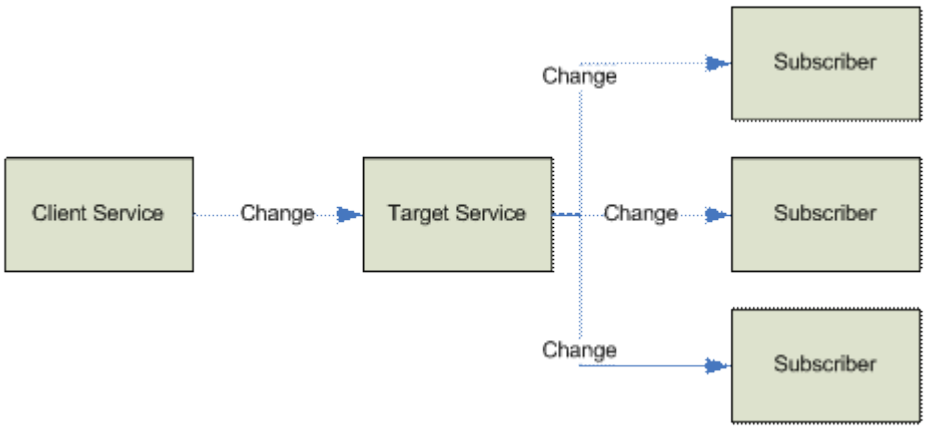
읽기(view)를 제공하지만, 이것 또한 구조화되지 않은 것이고 단지상태(state)를 읽은 것을 반영할 뿐 바꾸지 못합니다.

반면, 웹서비스들은 XSD, C#과 Java 등의 다양한 형태의 시스템으로 기술된 XML 로 표현됩니다.리소스를 구조화된 존재로 생각하는 것은 서비스의 각 부분들을 작업하는 일반적 잡업(common operations)들을 가능하게 합니다.

마이크로소프트 Robotics Studio 런타임은 리소스가 구조화된 존재이며 모든 서비스에 대해 일반적 데이터 모델 없이 추가, 삭제, 갱신, query 가능한 서비스 상태(state)들의 조작들로 정의됨을 따르고 있습니다. 이 관점에서 상태(state) 조작은 새로운 것이 아닌 REST 웹서비스의 구조화된 읽기와 결과를 결합함으로 서비스들이 상호작용하기가 보다 유연해 짐을 의미합니다.

REST 에서 이벤트 알림(Event Notification in a REST World)

마이크로소프트 Robotics Studio 런타임은 웹서비스 세상에서 이벤트알림을 가져왔습니다. 이벤트를 서비스의 상태(state)변화로 모델링함으로써 이벤트를 REST 모델로 도입하는 것이 가능해졌습니다. 예로 서비스에 대한 UPDATE 조작의 경우, 서비스의 상태(state)는 UPDATE 조작으로 변화됩니다. 또한 UPDATE 조작 자체가 상태(state) 변화를 나타내며 이것은 이벤트 알림을 단순히 UPDATE 조작으로 자연스럽게 여기게 합니다.



이벤트 알림을 서비스의 상태(state) 변화로 표현함으로써 subscriber 는 특정 서비스의 내용(semantics)에 상관없이 publisher 를 모니터링 하는 일반적 방법을 가지며, publisher 는 이벤트 알림을 단순히 모든 상태(state) 변화를 subscriber 들에게 전달하는 것으로 표현하는 일반적 방법을 가집니다.

마이크로소프트 Robotics Studio 서비스 모델(Microsoft Robotics Studio Service Model)

마이크로소프트 Robotics Studio 런타임에서 애플리케이션이란 앞서 기술한 바와 같이 서비스들의 구성이며 각 서비스는 이벤트 알림과 구조화된 데이터 조작을 추가한 REST 스타일의

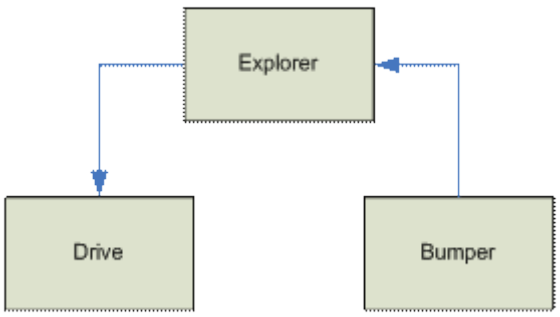
서비스 입니다. 느슨히 연결(loosely coupled)되고 가벼운 서비스들을 한 호스트 혹은 네트웍 상에서 오케스트레이션함을 통해 다양한 복잡도의 애플리케이션들은 제작되고 보다 복잡한 애플리케이션을 제작하기 위해 다시 구성될 수 있습니다.

HTTP 와의 관계 때문에 각 서비스상태(state)는 웹브라우저나 DSSP (Decentralized Software Services Protocol)라는 단순한 SOAP 기반 프로토콜을 이용한 HTTP 를 통해 직접 감독 및 조작될 수 있습니다:

1. HTTP 는 GET, PUT, POST 와 DELETE 같은 조작을 지원합니다. 이것은 서비스가 기존의 인프라스트럭쳐와 데이터 형식을 사용하는 웹브라우저 같은 다양한 UI 컨텍스트 하에서 사용되게 합니다.
2. DSSP 는 UPDATE, INSERT 와 QUERY 와 같은 내용이 담긴 구조화된 데이터 조작을 지원합니다. 또한 DSSP 는 해당 서비스 상태(state)변화와 연동된 이벤트 알림 모델을 지원합니다.

로보틱스 애플리케이션 예제(Sample Robotics Application)

다음의 단순한 마이크로소프트 Robotics Studio 애플리케이션의 예는 기본적인 로봇을 제어하기 위한 세가지 서비스로 구성됩니다. drive service (actuator)는 로봇이 다양한 속도와 방향으로 움직이게 합니다. bumper service (sensor) 는 로봇이 다른 물체에 충돌했을 때 신호를 보내며, explorer service (orchestration)는 bumper 로 들어온 입력에 따라 속도와 방향을 바꾸게 됩니다. Explorer service 는 bumper 로부터 이벤트 알림을 기다리며 drive service 의 속도와 방향을 결정합니다.



이 예제에서 Explorer service 는 Bumper service 에 subscribe 하고 Drive service 에 명령을 보냅니다. 이러한 구성을 partnering 이라고 부릅니다. 파트너관계는 서비스가 다른 서비스를 가지는 관계를 표현합니다. 파트너는 해당 파트너의 형식(type)과 파트너 서비스 인스턴스로 기술됩니다. 파트너 정보는 상태(state)로 나타나며 DSSP 의 부분인 LOOKUP 조작을 이용해 검사할 수 있습니다.

2. CCR 사용자 가이드

CCR 소개

Concurrency and Coordination Runtime (CCR)은 .NET 2.0. Common Language Runtime 을 위한 여러 프로그래밍 언어를 지원하는 managed code library (DLL) 입니다. CCR은 비동기 오퍼레이션(asynchronous operations) 관리와 동시성 처리 및 병렬처리 하드웨어와 부분적 실패(partial failure)를 처리를 손쉽게 하는 서비스기반 애플리케이션입니다. 또한 애플리케이션 디자인에서 소프트웨어 모듈이나 컴포넌트가 느슨히 연결(loosely coupled)되게 합니다. CCR은 각각이 컴포넌트들이 런타임 환경과 다른 컴포넌트들에 관한 최소한의 가정하에서 독립적으로 개발될 수 있게 합니다. 이 접근법은 디자인 단계에서부터 프로그램에 관한 개념을 바꾸며 보다 손쉽게 동시성, 실패, 고립화(isolation)를 일관되게 처리할 수 있게 합니다.

기존 프로그램 방식에서의 문제점

- **비동기성(Asynchrony)** - 네트워크 상의 프로그램동작처럼 느슨히 연결된 소프트웨어 컴포넌트간의 통신 혹은 유저 입력과 file 입출력 서브시스템을 이용한 UI 코드 통신시 비동기 조작(asynchronous operations)은 코드를 보다 확장가능하고 응답성을 높여주며 다중 조작시 실패를 처리하게 합니다. 그러나 비동기 프로그래밍은 “콜백”과 조작을 실행하는 코드 사이 로직의 단절로 인해 유저 코드의 가독성을 심각히 줄입니다. 또한 다중 조작시 실패처리를 성공적으로 구현하기가 힘듭니다.
- **동시성(Concurrency)** - 병렬 실행 가능한 여러 개의 실행 리소스들을 보다 잘 활용하는 코드는 독립적인 로직 부분(logical segment)들로 나누어져야 하며 각 부분들이 결합해 수행시 결과를 적절히 출력하기 위해 통신이 필요합니다. 종종 이런 로직 부분들은 쓰레드 OS 요소(primitive) 로 구현되고 긴 수명을 가진 iteration 그 이상이 아닙니다. 쓰레드 성능이 쓰레스 시작 시에 제한되기 때문에 쓰레드는 긴 시간 동안 활동 상태가 유지 되고 특정 패턴으로 고착됩니다: 코드는 블러킹 혹은 동기 호출(synchronous calls)을 사용한 긴 시퀀스로 구조화되고 단지 동시에 한가지 일을 처리합니다. 더 나아가 쓰레드는 그들 사이의 주된 통신이 공유메모리(shared memory)라고 가정하여 프로그래머가 공유 메모리 접근을 동기화 하기 위해 아주 명시적이고 오류가 발생하기 쉬운 방법을 사용하게 합니다.
- **조정과 실패 처리(Coordination and Failure Handling)** - 컴포넌트들 사이의 조정(Coordinating)은 대규모 소프트웨어 프로그램시 가장 복잡한 부분입니다. 상호작용 패턴들의 실수(오브젝트에 대해 메서드 호출 대 OS 시그널 요소(signaling primitives) 사용 대 큐와 signaling 사용)는 런타임 행동이 조정 접근법 사이에서 크게 변화할 때 일기 힘든 코드를 만듭니다. 이는 또한 에러 처리 접근법이 잘못 정의되고 또한 크게 변화할 때 더욱 심각합니다.

애플리케이션 모델(Application Model)

CCR 은 컴포넌트들을 메시지만을 통해 상호작용하는 부분들로 나눌 수 있는 애플리케이션 모델에 적합합니다. 이 모델에서 컴포넌트들은 메시지들 사이의 조정할 방법을 필요로 하고 복잡한 실패 시나리오들을 처리해야 하며 비동기 프로그래밍을 효율적으로 처리해야 합니다. 이 애플리케이션 모델은 또한 이기종 하드웨어 통합 방법과 네트워크 애플리케이션을 제작 방법과 매우 유사합니다.전통적 PC 프로그램에서 서버 애플리케이션과 브라우저에서 실행되는 애플릿까지 대부분의 소프트웨어 프로그램은 같은 요구사항이 있습니다. 이는 유저 입력을 조정하고 저장 입출력과 UI 표현의 조정이 필요하다는 것입니다. 대부분의 동기 API 계층에서 보면 모호하지만, 장치들이 서로 다른 속도로 동작하고 리소스 사용에서 큰 차이가 있기 때문에 비동기성은 필수적이며 일반적으로 이를 구분하는데 큐를 어떻게 사용해야 하는지를 우리는 알고 있습니다.

다음 장은 효율적이고 강인하며 확장 가능한 방법으로 위 문제 영역들을 처리하는 CCR프로그래밍 모델과 이의 구현을 소개합니다.

CCR API 개요

CCR 구현은 기능적으로 다음의 세가지 종류를 가집니다.

1. 포트(Port) 와 PortSet 큐 요소(queueing primitives) (참조 CCR 포트와 PortSet). Port 클래스는 FIFO (First In First Out) 어떤 유효한 CLR 형식(type) 이든지 가능한 아이템 큐이고 대부분의 경우 Arbiter 에 의해 감독되는 유저코드인 리시버들의 큐입니다.
2. 조정 요소(coordination primitives)는 arbiter 라 합니다. (참조 CCR 조정 요소들). 이는 유저코드를 실행하는 클래스이며, 적당한 조건이 될 때 종종 메서드의 delegate 실행합니다. 이 요소는 네스트(nest)될 수 있으며 확장가능 합니다.
3. Dispatcher, Dispatcher 큐 그리고 태스크 요소(Task primitives) (참조 CCR 태스크 스케줄링). CCR 은 어떻게 태스크가 실행되고 포함되며 어떤 리소스를 사용하는지를 요약해(abstract) 스케쥴링과 로드 밸런싱을 그 외의 구현과 분리시킵니다. CCR 은 하나임을 피하고 넓은 실행 리소스를 수행하며(wide execution resource) CLR 쓰레드 풀(thread pool)을 사용하며 프로그래머가 어떤 쓰레딩 경향이든 요약한 여러 개의 완전히 분리된 OS 쓰레드 풀(isolated pools)을 사용하게 합니다. 대부분의 경우 수많은 소프트웨어 컴포넌트는 적은 수의 OS 쓰레드에서 매우 많은 작업을 중재하는 하나의 Dispatcher 리소스를 공유합니다. Dispatcher 큐 클래스는 단지 Dispatcher 클래스와 상호작용하는 방법이며, Dispatcher 별 다중 큐는 공평한 스케쥴링 정책이 가능하게 합니다.

CCR 포트(Port)와 PortSet

CCR 포트 는 가장 일반적인 요소(common primitive)이고 어떤 두 컴포넌트 사이의 상호작용에 사용됩니다. 포트는 **IPortReceive** 와 **IPort** 두 인터페이스로 구현됩니다. 인터페이스는 아이템을 포트에 추가하고 아이템을 검색하여(retrieve) 아이템을 제거하는 코드를 장착하고 비동기 실행하는 논리적으로 구분된 방법입니다. 인터페이스는 strongly typed 가 아닌 (type object 의 인스턴스로 동작) default 구현(implementation)과 Port<> class 로 된 하나의 generic type 인수(argument)를 가집니다.

주의: 클래스 다음의 <> 표기는 generic class 를 나타냅니다. (C++의 템플릿과 유사). 예로 Port<int> 는 하나의 System.Int32 인 CLR 타입 인수(type argument)를 가진 포트 구현입니다.

아이템 포스팅(Posting Items)

아이템을 포트 에 추가하는 것은 비동기 조작입니다. Arbiter 가 포트에 장착되지 않았을 경우 아이템은 큐에 추가됩니다. Arbiters 가 존재할 경우, 그 각각은 아이템을 평가(evaluation)하고 태스크가 실행을 위해 생성 및 스케줄 될지를 결정하기 위해 호출됩니다. 이 평가는 빠르며, 비블록화(non-blocking) 조작이어서 Post 메서드는 가능한 빨리 호출자에게 조정권을 돌려줍니다. 이것이 CCR 의 비동기 행동의 핵심입니다: 프로그램은 아이템을 포트에 빨리 포스트하여 태스크가 현재의 쓰레드가 아닌 적당한 다른 쓰레드에 스케줄 되게 적당한 조건이 되면 호출되게 합니다.

예제 1

```
// Create port that accepts instances of System.Int32
Port<int> portInt = new Port<int>();
// Add the number 10 to the port
portInt.Post(10);
// Display number of items to the console
Console.WriteLine(portInt.ItemCount);
```

위 예제에서는 integer 만을 허용하는 포트 형식(type)을 생성하고 숫자 10 인 한 개의 아이템을 포트에 더한 뒤 포트의 ItemCount 가 1 로 읽어짐을 확인 합니다.

아이템 검색 (Retrieving Items)

아이템을 포트로부터 검색하는 두 개의 시나리오가 있습니다:

1. 포트를 "수동적" queue 로 사용: 아이템이 포스트 될 때 스케줄 된 태스크가 없는 경우 아이템은 Test 메서드를 호출함으로 검색됩니다. 이 메서드는 블록화(block) 되지 않습니다. 만약 아이템이 없으면 flase 를 리턴하고 출력 parameter 를 null 로 설정합니다. 이 시나리오는 어떤 이벤트발생 시 코드를 실행할 메커니즘이 있거나 단지 포트를 효율적인 FIFO 큐로 사용하고자 할 때 사용됩니다. 이것은 포트에 늦게 리시버를 장착하여 이미 있는 아이템에 대해 비동기적 태스크를 수행할 때 유용합니다.
2. 포트를 "능동적" queue 로 사용: 아이템이 포스트 되었을 때, 하나 이상의 arbiter 가 포트에 등록되어 있는 경우 태스크가 스케줄됩니다. 이것이 가장 일반적인 CCR 포트의 사용법 입니다. 아이템이 포스트될 때 유저코드가 스케줄되어 있기 때문에 CCR 의 동시성은 잠재적으로 병렬 실행이 가능하게 합니다.

예제 2

```
// Create port that accepts instances of System.Int32
Port<int> portInt = new Port<int>();
// Add the number 10 to the port
portInt.Post(10);
// Display number of items to the console
Console.WriteLine(portInt.ItemCount);
// retrieve the item using Test
int item;
bool hasItem = portInt.Test(out item);
if (hasItem)
{
    Console.WriteLine("Found item in port:" + item);
}
```

예제 1 을 확장한 위 예제에서 Test 메서드가 포스트된 아이템을 검색하는 데 사용되었습니다.아이템은 thread-safe 형식으로 제거되고 **ItemCount** 은 0 이 되어야 합니다.

예제 3

```
// Create port that accepts instances of System.Int32
Port<int> portInt = new Port<int>();
// Add the number 10 to the port
portInt.Post(10);
// Display number of items to the console
Console.WriteLine(portInt.ItemCount);

// create dispatcher and dispatcher queue for scheduling tasks
Dispatcher dispatcher = new Dispatcher(0, "sample dispatcher");
DispatcherQueue taskQueue = new DispatcherQueue("sample queue", dispatcher);
```

```
            // retrieve the item by attaching a one time receiver
            Arbiter.Activate(
                taskQueue,
                Arbiter.Receive(
                    false, // one time
                    portInt, // port to register on
                    delegate(int item) // user delegate
                    {
                        // this code executes in parallel with the method that
                        // activated it
                        Console.WriteLine("Received item:" + item);
                    }
            ));
            // any code below runs in parallel with delegate
```

위 예제는 Example 1 을 따르지만, Test 메서드 대신 단순한 리시버 arbiter 를 포트에 등록하기 위해 **Arbiter.Activate** 를 사용합니다. 여기서 리시버는 익명 메서드(anonymous method)로 인라인으로 정의된 유저 delegate 입니다. Delegate 는 제공된 Dispatcher 큐 인스턴스와 같이 dispatcher 쓰레드 중 하나에서 실행됩니다. 이 예제에서 Delegate 는 항상 주 부분(main body)과 병렬적으로 실행됨을 주의하십시오. delegate 실행 후 리시버는 자동적으로 포트로부터 제거됩니다. 리시버와 arbiters 는 다음 장에 계속됩니다.

포트 상태 검사(Inspecting Port State)

아래의 메서드와 Port<> 에 대한 프로퍼티는 런타임에 포트의 상태를 검사합니다.

- ToString - 이 메서드는 default ToString() 구현을 override 하여 가독가능한 형식으로 포트에 있는 아이템의 수와 리시버들의 계층을 포트와 함께 출력합니다.
- ItemCount - 현재 큐에 있는 아이템 수 입니다. 하나 이상의 지속된(persisted) arbiter 가 포트에 등록되어 있으면 이 값은 0 이 됨을 주의하십시오. 대부분의 Arbiter 에서는 item 이 태스크로 즉각 변환되어 실행되도록 스케줄 되어, 아이템은 포트 아이템 큐에 보이지 않습니다.

PortSet

Port<> 클래스는 오직 하나의 generic 타입 인수 를 가지므로, 종종 single container 하에서 Port<> 클래스의 다중, 독립적 인스턴스를 가져 하나의 항목(entity)으로 취급하는 것이 편리합니다. PortSet<> generic 클래스는 이런 그룹화가 가능하며 CCR 소프트웨어 컴포넌트에 “인터페이스”를 정의하는 가장 일반적인 방법입니다: public 메서드 집합 대신에 CCR 컴포넌트는 N 개의 독립된 Port<> 인스턴스 하에서 strongly typed PortSet 만으로 표현됩니다. 이 컴포넌트는 서로 다른 메시지 타입들이 어떻게 연결되어 실행될지를 조정합니다.

또한 코드는 항상 메시지들을 포스트 하는 형태로 동시에 독립적으로 실행됩니다. 이런 프로그래밍 모델은 웹 서비스, 전통적 서버 프로세스와 커널 모드 I/O 프로세서와 매우 유사합니다.

PortSet 인스턴스 생성(Constructing a PortSet instance)

PortSet 의 새 인스턴스를 생성하는 두가지 방법입니다:

1. generic 타입 인수를 사용하여 Port<> 인스턴스의 형식과 개수를 컴파일시 정의합니다. CCR 은 PortSet 의 구현(implementations)을 데스크톱 CLR 에서 최대 20 개, .NET Compact Framework 에서 최대 8 개의 generic 타입 인수로 제공합니다. 이 접근법은 포스팅시 최고의 형식(type) 안정성과 런타임 성능을 제공하지만, PortSet 개수가 제한됩니다.
2. 타입 인수들의 parameter 리스트를 가지는 초기화 생성자(initialization constructor)를 사용합니다. 이 경우 형식(type)개수에 제한이 없이, Port<> 인스턴스들이 런타임에 생성됩니다. 어떤 각 메시지 타입에 대한 Post 같은 메서드들은 지원되지 않으며 (PostUnknownType 혹은 TryPostUnknownType 이 반드시 사용되야 함) 아이템 타입이 런타임에 검사되고 비교되어 1 번 방법에 비해 성능적 저하가 있습니다.

예제 4

```
            // Create a PortSet using generic type arguments
            PortSet<int, string, double> genericPortSet = new PortSet<int, string,
double>();
            genericPortSet.Post(10);
            genericPortSet.Post("hello");
            genericPortSet.Post(3.14159);

            // Create a runtime PortSet, using the initialization
            // constructor to supply an array of types
            PortSet runtimePortSet = new PortSet(
                typeof(int),
                typeof(string),
                typeof(double)
                );

            runtimePortSet.PostUnknownType(10);
            runtimePortSet.PostUnknownType("hello");
            runtimePortSet.PostUnknownType(3.14159);
```

위 예제에서는 세가지 다른 타입을 가진 PortSet 을 생성하는 것을 보여줍니다. 처음에는 generic 타입 인수를 사용하고 그 뒤에는 런타임에 type array 를 사용합니다. strongly typed

Post 메서드는 non generic PortSet 에서 상속(derives)된 클래스가 추가되어 generic PortSet<>만큼의 안전성을 가질 수 있습니다.

예제 5

```
    /// <summary>
    /// PortSet that accepts items of int, string, double
    /// </summary>
    public class CcrConsolePort : PortSet<int, string, double>
    {
    }
    /// <summary>
    /// Simple example of a CCR component that uses a PortSet to abstract
    /// its API for message passing
    /// </summary>
    public class CcrConsoleService
    {
        CcrConsolePort _mainPort;
        DispatcherQueue _taskQueue;
        /// <summary>
        /// Creates an instance of the service class, returning only a PortSet
        /// instance for communication with the service
        /// </summary>
        /// <param name="taskQueue"></param>
        /// <returns></returns>
        public static CcrConsolePort Create(DispatcherQueue taskQueue)
        {
            CcrConsoleService console = new CcrConsoleService(taskQueue);
            console.Initialize();
            return console._mainPort;
        }

        /// <summary>
        /// Initialization constructor
        /// </summary>
        /// <param name="taskQueue">DispatcherQueue instance used for scheduling</param>
        private CcrConsoleService(DispatcherQueue taskQueue)
        {
            // create PortSet instance used by external callers to post items
            _mainPort = new CcrConsolePort();
            // cache dispatcher queue used to schedule tasks
            _taskQueue = taskQueue;
        }

        private void Initialize()
        {
            // Activate three persisted receivers (single item arbiters)
            // that will run concurrently to each other,
            // one for each item/message type
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive<int>(true, _mainPort, IntWriteLineHandler),
```

```
                Arbiter.Receive<string>(true, _mainPort, StringWriteLineHandler),
                Arbiter.Receive<double>(true, _mainPort, DoubleWriteLineHandler)
            );
        }

        void IntWriteLineHandler(int item)
        {
            Console.WriteLine("Received integer:" + item);
        }
        void StringWriteLineHandler(string item)
        {
            Console.WriteLine("Received string:" + item);
        }
        void DoubleWriteLineHandler(double item)
        {
            Console.WriteLine("Received double:" + item);
        }
    }
```

이 예제에서는 세개의 타입 인수(type arguments)가진 PortSet 에서 상속된 CcrConsolePort 클래스를 정의합니다. 이것은 generic type 정의를 반복할 필요가 없어 portset 의 차후 사용을 보다 읽을 수 있게 합니다. CcrConsoleService 클래스는 CCR 컴포넌트를 위한 일반적인 패턴을 보여줍니다. 예제는 인스턴스 object 를 생성하고 이와 통신하기 위한 private PortSet 인스턴스를 재실행할 정적 루틴(static routine)을 정의한 뒤, PortSet 의 각 메시지 형식을 위한 핸들러를 활성화합니다. 각 핸들러의 활성화는 동시에 일어납니다.

타입 안정성(Type Safety)

PortSet 클래스는 해당 클래스가 지원하는 모든 포트 인스턴스와 아이템 형식에 대해 열거(enumeration)를 지원합니다. generic PortSet 은 또한 자동적으로 올바른 Port<> 인스턴스를 invoke 하고, 만약 타입 인수(type argument) 가 PortSet 에 맞을경우 자동적으로 PortSet<> 인스턴스를 Port<> 인스턴스로 implicit conversion 하는 편리성을 제공합니다. 이것은 PortSet 의 어떤 한 포트에 리시버를 단지타입 인수 만을 이용해 등록할 때 유용합니다.

예제 6

```
            PortSet<int,string,double> portSet = new PortSet<int,string,double>();
            // the following statement compiles because of the implicit assignment
operators
            // that "extract" the instance of Port<int> from the PortSet
            Port<int> portInt = portSet;
            // the implicit assignment operator is used below to "extract" the Port<int>
            // instance so the int receiver can be registered
            Arbiter.Activate(_taskQueue,
```

```
                Arbiter.Receive<int>(true, portSet, delegate(int item) { })
            );
```

이 예제는 두 use case 를 통해 implicit operator 의 사용방법을 보여줍니다:

1. PortSet 안에서 올바른 Port<> 인스턴스를 다른 Port<> 변수에 할당함.
2. 올바른 Port<> 인스턴스를 Extracting 하여 Arbiter 에 등록하여 사용되게 함.

CCR 조정 요소들(Coordination Primitives)

비동기 프로그래밍은 부분적 실패 처리와 다중 조작을 조정해야 하고 동시 조작의 제약조건을 만족하면서 비동기 행동을 정의해야 하므로 매우 어렵습니다. CCR 은 서로 다른 코드 부분들 간의 높은 수준의 제약조건과 어떤 조정이 필요한지를 표현하는 방법을 포트간 메시지 전송으로 모델링하여 동시성 처리 가능하게 하고 이를 증진합니다.

비동기와 동시성을 함께 구현하는 것이 중요합니다: 느슨한 연결과 빠른 초기화 상호작용을 위한 일정한 큐의 사용은 종속성을 잘 정의하며 확장 가능한 소프트웨어 디자인을 가능하게 합니다. 따라서 CCR 은 비동기 프로그래밍의 단점을 보완하고 소프트웨어 컴포넌트를 위해 적합한 방식입니다..

CCR 에서 제공되는 조정 요소(coordination primitives)에 대한 주된 두 개의 시나리오입니다:

1. 오랜 시간 지속되는 서비스기반 컴포넌트를 위한 내부 요청(requests)의 조정. 대표적인 예는 HTTP 요청을 기다리는 웹서비스로 CCR 포트를 이용해 각 요청을 서비스하는 핸들러를 장착하고 내부요청을 포스트해 처리합니다. 또한 다른 핸들러들이 동작 중일 때 어떤 핸들러가 실행 하지 못하게 하는 요소도 사용할 수 있습니다.
2. 다양한 리턴 타입을 가지는 요청의 조정. 한 예로 성공 혹은 실패를 PortSet 결과로 기다리는 요청입니다: 요청이 처리완료 될 때, 성공 혹은 실패 아이템이 포스트 되며, 적당한 코드가 실행됩니다. 다른 예는 다중 요청을 한번에 보내고 하나의 요소를 이용해 하나 혹은 여러 개의 응답 포트에 도착한 응답들을 그 순서에 상관없이 모으는 것입니다.

Arbiter 를 간단히 설명한 뒤, 위 시나리오에 대한 자세한 설명이 제공됩니다.

Arbiter 정적 클래스(Arbiter Static Class)

Arbiter 정적 클래스는 탐지가능하며 type safe 한 CCR Arbiter 클래스 인스턴스 생성하는 것을 도와줍니다. 아래에 기술된 메서드들은 이 클래스의 맴버들 입니다. Arbiter 정적 메서드는 CCR arbiter 생성의 모든 방법은 아닙니다. 보다 상세한 설정으로, 각 arbiter 클래스는 적절한 생성자 인수를 사용해 직접 생성될 수 있습니다. 다음은 가장 일반적인 arbiter 클래스 메서드가 호출될 때 어떤 CCR 클래스가 생성되는지를 보여줍니다.

- Arbiter.FromTask -> **Task** 인스턴스를 생성
- Arbiter.Choice -> **Choice** 인스턴스를 생성
- Arbiter.Receive -> **Receiver** 인스턴스를 생성
- Arbiter.Interleave -> **Interleave** 인스턴스를 생성
- Arbiter.JoinedReceive -> **JoinReceiver** 인스턴스를 생성

- Arbiter.MultipleItemReceive -> **JoinSinglePortReceiver** 인스턴스를 생성

싱글 아이템 리시버(Single Item Receiver)

single item 리시버 는 Port<T>의 인스턴스를 가진 T 형식의 인수 하나를 가진 유져 delegate 를 연결합니다. 지속하기 옵션(persist option)이 true 인 경우, 매 번 아이템이 포스트될 때 마다, 리시버 가 사용자 delegate 의 인스턴스를 실행합니다. 지속하기 옵션(persist option)이 false 인 경우 리시버는 포스트 된 한 아이템을 처리할 사용자 delegate 를 실행한 뒤 이를 포트에서 제거합니다.

주의: 이미 Port<T> 인스턴스 큐에 이미 아이템들이 있는 경우, 리시버는 큐된 아이템들의 시점과 상태에 따라 이를 신뢰할만한 방법으로 처리합니다.

Example 7

```
Port<int> port = new Port<int>();
Arbiter.Activate(_taskQueue,
   Arbiter.Receive<int>(
       true,
       port,
       delegate(int item) { Console.WriteLine(item); }
   )
);
// post item, so delegate executes
port.Post(5);
```

위 예제는 Port<int> 인스턴스를 생성하고 한 개의 아이템 리시버를 활성화 해 아이템이 포트에 포스트될 때 마다 사용자 delegate 를 실행합니다. 리시버가 지속됨(persist)에 주의하세요.

예제 8

```
// alternate version that explicitly constructs a Receiver Arbiter
Arbiter.Activate(_taskQueue,
   new Receiver<int>(
       true, // persisted
       port,
       null, // no predicate
       new Task<int>(delegate(int item) { Console.WriteLine(item); }) // Task
   )
);
```

위 예제는 예제 7 번 과 실행 결과가 똑 같지만, Arbiter.Receive() 메서드가 단지 리시버 arbiter 의 wrapper 로 쓰여졌습니다.

Choice Arbiter

Choice Arbiter 는 하나의 branch 을 실행하고, 해당 포트에서 nest 된 다른 Arbiter 들을 자동으로 제거합니다. 이것은 하나의 선택만이 실행됨을 보장하고 branching behavior 를 표현하며 성공/실패 응답을 처리하고 race 문제를 피하게 합니다.

예제 9

```
            // create a simple service listening on a port
            ServicePort servicePort = SimpleService.Create(_taskQueue);
            // create request
            GetState get = new GetState();
            // post request
            servicePort.Post(get);
            // use the helper on the Arbiter class that creates a choice
            // given two types found on one PortSet. This a common use of
            // Choice to deal with responses that have success or failure
            Arbiter.Activate(_taskQueue,
                Arbiter.Choice(get.ResponsePort, // PortSet with ports to listen on
                delegate(string s) { Console.WriteLine(s); }, // delegate for success
                delegate(Exception ex) { Console.WriteLine(ex); } // delegate for failure
            ));
```

위 예제는 **Choice** arbiter 를 사용해 PortSet 에서 받아진 메시지에 따라 두개의 다른 delegates 를 처리하는 것입니다. Choice 가 2 개뿐 아니라 임의의 수의 리시버를 가지고 이를 조정할 수 있음에 주의하십시오. Arbiter.Choice 는 Choice arbiter 생성 대신으로 쓰였으며, 두 리시버 arbiter 를 생성했습니다.

주의: choice arbiter 는 "부모" arbiter 의 예입니다: single item 리시버 나 join 과 같은 arbiter 는 Choice 에 의해 nest 될 수 있습니다. Arbiter 는 계층화와 유저 핸들러가 실행되기 이전에 올바른 순서로 각 계층의 arbiter 호출되게 설계되었습니다. 이것은 프로그래머에게 몇 줄의 코드로 복잡한 조정기능을 표현할 수 있게 합니다.

Join 과 다중 아이템 리시버(Multiple Item Receiver)

다중 아이템 리시버 arbiter 는 다음 두 종류로 나뉩니다:

1. **Join** 들로 알려진(OS 에서는 WaitForMultiple) 리시버들은 하나 이상의 포트에서 수신을 시도하고 그 중 하나가 실패할 경우 아이템들을 다시 포스트하고 올바른 조건이 될

때까지 다시 시 기다립니다. 이 두 부분의 로직은 타입 안전(type safe)하고 데드락(deadlock) 없는 메커니즘입니다. 이것은 아이템의 도착 순서가 중요하지 않기에 데드락 걱정 없이 다중 리소스를 접근을 보장합니다. 아이템의 과 포트의 수가 실행 시에 결정될 수 있다는 점은 CCR 이 다른 형태의 typed join 들에 대해 제공하는 중요한 확장입니다.

2. 리시버는 해당 포트에 들어온 아이템을 계속 처리하고 총 아이템 수가 만족되면 사용자 delegate 를 실행합니다. 이것은 빠르지만 리소스 동기화 요소로 사용되어서는 안됩니다. 종종 이것은 다중 대기 요청의 결과를 모으는데 사용됩니다(scatter/gather scenarios).

Join

예제 10

```
Port<double> portDouble = new Port<double>();
Port<string> portString = new Port<string>();

// activate a joined receiver that will execute only when one
// item is available in each port.
Arbiter.Activate(_taskQueue,
    Arbiter.JoinedReceive<double, string>(
    false,// one time
    portDouble, // first port to listen
    portString, // second port to listen
    delegate(double value, string string Value)
    {
        value /= 2.0;
        stringValue = value.ToString();
        // post back updated values
        portDouble.Post(value);
        portString.Post(stringValue);
    })
);

// post items. The order does not matter, which is what Join its power
portDouble.Post(3.14159);
portString.Post("0.1");
//after the last post the delegate above will execute
```

위 예제는 간단한 "정적(static)" join (컴파일 시 포트개수와 형식이 정해지는)의 예입니다. 여기서는 두 포트에 대해 join 리시버를 활성화 한 뒤, 각 포트에 아이템을 포스트 합니다. Join 로직은 필요한 모든 것을 결정하고 실행될 delegate 를 스케줄 합니다.

예제 11

```
            Port<int> portInt = new Port<int>();
            Port<double> portDouble = new Port<double>();
            Port<string> portString = new Port<string>();

            // activate a joined receiver that will execute only when one
            // item is available in each port.
            Arbiter.Activate(_taskQueue,
                Arbiter.JoinedReceive<double, string>(
                false,// one time
                portDouble, // first port to listen
                portString, // second port to listen
                delegate(double value, string stringValue)
                {
                    value /= 2.0;
                    stringValue = value.ToString();
                    // post back updated values
                    portDouble.Post(value);
                    portString.Post(stringValue);
                })
            );

            // activate a second joined receiver that also listens on portDouble
            // and on a new port, portInt. Because the two joins share a common port
            // between them (portDouble), there is contention when items are posted on
            // that port
            Arbiter.Activate(_taskQueue,
                Arbiter.JoinedReceive<double, int>(
                false,// one time
                portDouble, // first port to listen
                portInt, // second port to listen
                delegate(double value, int intValue)
                {
                    value /= 2.0;
                    intValue = (int)value;
                    // post back updated values
                    portDouble.Post(value);
                    portInt.Post(intValue);
                })
            );

            // post items.
            portString.Post("0.1");
            portInt.Post(128);
            // when the double is posted there will be a race
            // between the two joins to determine who will execute first
            // The delegate that executes first will then post back a double,
            // allowing the delegate that "lost", to execute.
            portDouble.Post(3.14159);
```

위 예제는 이전 예제의 간단한 확장으로 join 경쟁(contention) 의 예를 보여 줍니다: 두 독립적인 delegate 가 같은 포트를 청취(listen)합니다. Join 이 두 부분으로 구성되므로 공유 포트에서 값이 나오는 대로 동시에 실행됩니다. 순서가 중요하지 않으므로 race 는 결과에 영향을 주지 않습니다. 지금까지 CCR 을 이용해 다중 리소스에 대해 전통적 lock 문제를 단순히 스케줄링 종속(scheduling dependency)으로 푸는 방법을 설명하였습니다. 메시지는 다중 동시 접속 시 보호되는 리소스이며 필요 시 코드 실행을 발생시키는 시그널 입니다.

주의: Join 의 사용은 nest 된 lock 사용의 좋은 대안이지만, 여전히 리소스 제어에 있어 에러를 야기할 만 합니다. 뒤에 설명될 Interleave 요소는 보다 단순하고 에러 없는 빠른 대안입니다.

예제 12

```
int itemCount = 10;
Port<double> portDouble = new Port<double>();
// post N items to a port
for (int i = 0; i < itemCount; i++)
{
    portDouble.Post(i * 3.14159);
}
// activate a JoinSinglePortReceiver that
// waits for N items on the same port
Arbiter.Activate(_taskQueue,
    Arbiter.MultipleItemReceive<double>(
    false,// one time
    portDouble, // port to listen
    itemCount, // total number of items
    delegate(double [] items)
    {
        foreach (double d in items)
        {
            Console.WriteLine(d);
        }
    })
);
```

위 예제는 간단한 "동적(dynamic)" join 의 예입니다: 실행 시에만 아이템의 수를 알 수 있으며(itemCount 에 저장됨), 아이템들은 한 포트에서 읽어 집니다. 예제는 join 의 한 종류인 **JoinSinglePortReceiver** 를 사용해 한 포트로 들어온 N 개의 item 에 대한 핸들러를 처리합니다.

다중 아이템 리시버

다중 아이템 리시버는 한포트에 대해 경쟁이 없을 때 적당합니다. 이는 다중 요청에 대해 응답을 모을 때 사용합니다.

예제 13

```
            // create a simple service listening on a port
            ServicePort servicePort = SimpleService.Create(_taskQueue);
            // shared response port
            PortSet<string, Exception> responsePort = new PortSet<string, Exception>();
            // number of requests
            int requestCount = 10;
            // scatter phase: Send N requests as fast as possible
            for (int i = 0; i < requestCount; i++)
            {
                // create request
                GetState get = new GetState();
                // set response port to shared port
                get.ResponsePort = responsePort;
                // post request
                servicePort.Post(get);
            }

            // gather phase:
            // activate a multiple item receiver that waits for a total
            // of N responses, across the ports in the PortSet.
            // The service could respond with K failures and M successes (K+M == N)
            Arbiter.Activate(_taskQueue,
                Arbiter.MultipleItemReceive<string, Exception>(
                responsePort, // port set used to gather success or failure
                requestCount, // total responses expected
                delegate(ICollection<string> successes, ICollection<Exception> failures)
                {
                    Console.WriteLine("Total     received:"     +     successes.Count     +
failures.Count);
                })
            );
```

위 예는 다중 대기 비동기 조작(multiple pending asynchronous operation)시 그 결과를 한 개의 delegate 로 처리하는 일반적 경우를 보여줍니다. 어떤 N 개의 조작에 대해 K 개가 실패, M 개가 성공할 때를 가정(K+M = N)하면, CCR **MultipleItemReceiver** 는 순서와 형식에 상관없이 모든 결과를 모으는 좋은 방법입니다. 여기서 K 개의 실패와 M 개의 성공을 가지는 두 모음을 가진 한 개의 delegate 가 불려집니다. Arbiter.MutipleItemReceive 메서드는 두 서로 다른 형식에 사용되지만, **MultipleItemGather** CCR arbiter 는 어떤 형식 개수와 상관없이 수행됩니다.

서비스 기반 컴포넌트의 조작(Coordination for service-oriented components)

지속된 싱글 아이템 리시버(Persisted Single Item Receivers)

CCR 은 내부(inbound) 메시지 처리 핸들러 활성화와 메시지 큐를 청취(listen)하는 컴포넌트를 효율적으로 런타임에 실행하는 데 목적을 둡니다. 이 매우 단순한 작업은 지속적(persist) 모드의 리시버 arbiter 를 사용해 포트를 청취하고 아이템이 포스트 될 때 핸들러를 활성화 시키는 것입니다.

예제 14

```
    /// <summary>
    /// Base type for all service messages. Defines a response PortSet used
    /// by all message types.
    /// </summary>
    public class ServiceOperation
    {
        public PortSet<string, Exception> ResponsePort = new PortSet<string, Exception>();
    }
    public class Stop : ServiceOperation
    {
    }
    public class UpdateState : ServiceOperation
    {
        public string State;
    }
    public class GetState : ServiceOperation
    {
    }
    /// <summary>
    /// PortSet that defines which messages the services listens to
    /// </summary>
    public class ServicePort : PortSet<Stop, UpdateState, GetState>
    {
    }
    /// <summary>
    /// Simple example of a CCR component that uses a PortSet to abstract
    /// its API for message passing
    /// </summary>
    public class SimpleService
    {
        ServicePort _mainPort;
        DispatcherQueue _taskQueue;
        string _state;

        public static ServicePort Create(DispatcherQueue taskQueue)
        {
            SimpleService service = new SimpleService(taskQueue);
            service.Initialize();
            return service._mainPort;
        }

        private void Initialize()
        {
```

```
            // using the supplied taskQueue for scheduling, activate three
            // persisted receivers, that will run concurrently to each other,
            // one for each item type
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive<UpdateState>(true, _mainPort, UpdateHandler),
                Arbiter.Receive<GetState>(true, _mainPort, GetStateHandler)
            );
        }

        private SimpleService(DispatcherQueue taskQueue)
        {
            // create PortSet instance used by external callers to post items
            _mainPort = new ServicePort();
            // cache dispatcher queue used to schedule tasks
            _taskQueue = taskQueue;
        }
        void GetStateHandler(GetState get)
        {
            if (_state == null)
            {
                // To demonstrate a failure response,
                // when state is null will post an exception
                get.ResponsePort.Post(new InvalidOperationException());
                return;
            }
            // return the state as a message on the response port
            get.ResponsePort.Post(_state);
        }
        void UpdateHandler(UpdateState update)
        {
            // update state from field in the message
            _state = update.State;
            // as success result, post the state itself
            update.ResponsePort.Post(_state);
        }
    }
```

위 예제는 소프트웨어 컴포넌트를 위한 일반적인 CCR pattern을 보여줍니다:

- 컴포넌트와 상호작용에 사용될 메시지 형식의 정의
- 정의된 메시지 형식을 받을 PortSet 파생 클래스 정의. 항상 PortSet 에서 파생되어야 하진 않지만 특정 개수의 형식의 PortSet을 재사용하는데 용이.
- 컴포넌트의 인스턴스를 초기화하고 해당 컴포넌트의 인스턴스와 통신에 사용되는 PortSet 인스턴스를 돌려주는 정적 생성 메서드.
- 해당 서비스와 통신시 외부 코드를 이용하는 public PortSet 에 사용할 Arbiter 장착을 위한 private 초기화 메서드.

다른 핸들러 사이에 동시성에 제약이 없을 경우, 간단하고 지속된(persist) 한 개의 아이템 리시버가 사용됩니다.

Interleave Arbiter

포트를 청취하는 컴포넌트는 동시 접근에 있어 신중히 보호 되야 하는 곳에 private 리소스를 사용하는 것과 같습니다. 다중 update 가 필요한 내부 저장 데이터 구조가 한 요소로 취급되는 것이 그 한 예입니다. 다른 시나리오는 어떤 외부 요청이 존재할 때 선점되어서는 안되는 복잡한 다중 단계프로세스로 구성된 컴포넌트 구현입니다. CCR 은 프로그래머가 복잡한 프로세스 구현에 집중하게 도와주고, 프로세스가 완료될 때까지 큐에 쌓여진 요청들과 핸들러 활성화를 처리합니다. **Interleave** arbiter 를 코드에서 보호가 필요한 부분을 정의하는 데 필수적으로 사용됩니다. interleave arbiter 는 쓰레드 프로그래밍에서 **reader/writer lock** 과 매우 유사지만, 특정 객체에 lock 을 사용하는 대신 code 일정 부분을 보호합니다. lock 에서의 경쟁문제를 피해, interleave 는 스케줄링 종속성(scheduling dependencies)을 생성할 내부 큐를 사용하며 동시에 수행할 수 있게 관리하고, 태스크가 배타적으로 수행되고 다른 태스크가 먼저 완료되도록 기다립니다.

예제 15

```
    /// <summary>
    /// Simple example of a CCR component that uses a PortSet to abstract
    /// its API for message passing
    /// </summary>
    public class ServiceWithInterleave
    {
        ServicePort _mainPort;
        DispatcherQueue _taskQueue;
        string _state;

        public static ServicePort Create(DispatcherQueue taskQueue)
        {
            ServiceWithInterleave service = new ServiceWithInterleave(taskQueue);
            service.Initialize();
            return service._mainPort;
        }

        private void Initialize()
        {
            // activate an Interleave Arbiter to coordinate how the handlers of the
service
            // execute in relation to each other and to their own parallel activations
            Arbiter.Activate(_taskQueue,
                Arbiter.Interleave(
                new TeardownReceiverGroup(
                // one time, atomic teardown
                    Arbiter.Receive<Stop>(false, _mainPort, StopHandler)
```

```
                ),
                new ExclusiveReceiverGroup(
                // Persisted Update handler, only runs if no other handler running
                    Arbiter.Receive<UpdateState>(true, _mainPort, UpdateHandler)
                ),
                new ConcurrentReceiverGroup(
                // Persisted Get handler, runs in parallel with all other activations of
itself
                // but never runs in parallel with Update or Stop
                    Arbiter.Receive<GetState>(true, _mainPort, GetStateHandler)
                ))
            );
        }

        private ServiceWithInterleave(DispatcherQueue taskQueue)
        {
            // create PortSet instance used by external callers to post items
            _mainPort = new ServicePort();
            // cache dispatcher queue used to schedule tasks
            _taskQueue = taskQueue;
        }
        void GetStateHandler(GetState get)
        {
            if (_state == null)
            {
                // when state is null will post an exception
                get.ResponsePort.Post(new InvalidOperationException());
                return;
            }
            // return the state as a message on the response port
            get.ResponsePort.Post(_state);
        }
        void UpdateHandler(UpdateState update)
        {
            // update state from field in the message
            // Because the update requires a read, a merge of two strings
            // and an update, this code needs to run un-interrupted by other updates.
            // The Interleave Arbiter makes this guarantee since the UpdateHandler is in
the
            // ExclusiveReceiverGroup
            _state = update.State  + _state;
            // as success result, post the state itself
            update.ResponsePort.Post(_state);
        }
        void StopHandler(Stop stop)
        {
            Console.WriteLine("Service stopping. No other handlers are running or will run
after this");
        }
    }
```

위 예제는 다양한 핸들러를 실행할 리시버들을 조정하는데 Interleave Arbiter 를 사용하는 예제입니다. Interleave 는 nest 된 다양한 리시버들을 가진 “부모” Arbiter 입니다. 예제는 동시성 측면에서 특정 핸들러를 독립적으로 다른 것은 아니게 수행하게 프로그래밍 할 수 있습니다. CCR 은 어떤 리소스 혹은 다중 단계 프로세스에 배타적인 접근이 필요한지 알 필요가 없습니다. 단지 어떤 코드 핸들러가 보호되어야 하는지를 알아야 합니다. 이 예제에서 핸들러는 매우 단순하지만, 이후에 소개될 Iterator 핸들러는 여러 단계로 수행되는 복잡한 코드를 어떻게 보호하는지를 보여줍니다.

CCR 태스크 스케줄링

CCR 의 세번째 주요요소는 태스크 스케줄링으로 활성화 된 리시버를 가진 포트에 메시지가 도착했을 때 이를 처리할 태스크를 어떻게 해당 머신의 실행 리소스에 로드 밸런싱 하는가 입니다. CCR 스케줄링에는 세 가지의 클래스가 있습니다:

- **ITask** 인터페이스와 태스크 및 IterativeTask 구현. ITask 를 구현하는 클래스 만이 스케줄 됩니다. Arbiters 는 또한 ITask 가 스케줄되고 적절히 활성화되도록 구현합니다.
- **DispatcherQueue** 클래스. DispatcherQueue 는 태스크의 FIFO 큐입니다. Dispatcher queues 는 태스크 스케줄링을 위해 CLR 쓰레드 풀을 사용하거나(매우 드문 경우) CCR Dispatcher 의 인스턴스를 사용합니다.
- **Dispatcher** 클래스. Dispatcher 는 OS 쓰레드를 관리하고 하나 이상의 DispatcherQueue 인스턴스에 있는 태스크들의 로드 밸런스를 관리합니다.

예제 16

```
1:          Dispatcher dispatcher = new Dispatcher(
                0, // zero means use one thread per CPU, or 2 if only one CPU present
                "sample dispatcher" // friendly name assigned to OS threads
                );

2:          DispatcherQueue taskQueue = new DispatcherQueue(
                "sample queue", // friendly name
                dispatcher // dispatcher instance
                );

3:          Port<int> port = new Port<int>();
4:          Arbiter.Activate(taskQueue,
               Arbiter.Receive<int>(
                   true,
                   port,
                   delegate(int item) { Console.WriteLine(item); }
               )
            );
            // post item, so delegate executes
5:          port.Post(5);
```

위 예제는 dispatcher 와 dispatcher queue 를 생성하고 태스크 스케줄에 사용하는 것을 보여줍니다. 다음은 예제의 단계별 설명입니다:

1. 쓰레드 개수를 0 으로 하여 Dispatcher 인스턴스를 생성합니다. 이것은 CCR 이 쓰레드 수를 OS 에 보고된 CPU cores 수에 따라 결정하게 합니다. 디폴트값으로 사용되는 CPU 당 쓰레드 수는 Dispatcher class 의 정적(static) 프로퍼티인 **ThreadsPerCpu** 로 제어됩니다.

2. 단계 1 에서 생성된 Dispatcher 인스턴스를 인수로 DispatcherQueue 인스턴스를 생성합니다. 이는 dispatcher 를 큐에 장착하는 것을 의미합니다.
3. Port<int> 인스턴스를 생성합니다. 이 포트는 아이템을 포스트 하고 delegate 를 가진 리시버를 장착하는데 사용합니다.
4. Arbiter.Activate 메서드는 dispatcher 큐 인스턴스를 전달하고 포트에 리시버 arbiter 를 장착하여 해당 아이템이 도착하면 실행할 delegate 를 추가하는데 사용합니다.
5. int 형식의 아이템을 포트에 포스트 합니다.

아이템이 리시버가 장착된 포트에 포스트 될 때 다음 과정이 포트구현 내에서 이루어 집니다.:

1. 값을 포스트 하기 위해 container 가 생성됩니다. container 클래스인 IPortElement 는 CCR 이 아이템을 형식에 관계없이 큐에 넣고 태스크 인스턴스에 할당 되도록 합니다.
2. container 인스턴스가 큐됩니다.
3. 리시버리스트가 null 이 아니고 적어도 한 개 이상의 리시버가 있다면, 포트는 ReceiverTask.Evaluate 메서드를 검색해 리시버와 그 arbiter 계층이 아이템이 사용될지를 결정할 수 있게 합니다. 예로 리시버는 Evaluate 로부터 true 를 리턴하고 아이템과 유저 delegate 를 인수로 Task<int> 인스턴스를 생성합니다.
4. 포트 로직은 리시버에 있는 Evaluate 메서드로 부터 리턴 된 태스크를 지닌 taskQueue.Enqueue 를 호출합니다. 리시버가 먼저 활성화된 경우, Arbiter.Activate 메서드가 제공한 dispatcher queue 인스턴스와 연결됨에 주의하십시오.

위 단계 4 뒤, 생성된 태스크 인스턴스는 스케줄링 로직에 의해 처리됩니다.

예제 17

```
            Task<int> task = new Task<int>(
              5,
              delegate(int item) { Console.WriteLine(item); }
              );
          taskQueue.Enqueue(task);
```

위 예제는 예제 16 에서와 같이 delegate 를 스케줄링 코드지만 포트에 어떤 것도 포스팅하지 않습니다. 태스크 인스턴스를 명시적으로 생성하는 것은 데어터가 사용가능하고 코드가 즉시 처리될 수 있을 때 유용합니다. CCR 은 Port.Post 호출의 범위에서 리시버가 불려질 때 와 유사한 태스크를 수행합니다.

아이템을 dispatcher 큐에 들어가면, 다음과 같이 됩니다:

1. dispatcher 큐는 dispatcher 인스턴스에 신호를 보내고 새로운 태스크 실행에 이용할 수 있게 연결됩니다.
2. dispatcher 는 TaskExecutionWorker 클래스의 하나 이상의 인스턴스에 통지합니다. 각 태스크 실행자는 1 개의 OS 스레드를 관리합니다. 아이템이 스케줄링에 이용할 수 있을 때 dispatcher 으로부터 신호를 기다리면서 스레드를 효율적인 슬립 상태에 넣습니다.
3. TaskExecutionWorker 의 인스턴스는 DispatcherQueue.Test 메서드에 큐로부터 태스크를 검색하도록 요구한다. 만약 태스크를 이용가능 하면, 작업자는 ITask.**Execute** 를 호출합니다.
4. 태스크와 연관 된 ITask.Execute 는 delegate 를 호출하고, 태스크와 하나 이상의 매개 변수를 건네줍니다. 예제에서는 값 5 를 가진 하나의 매개 변수가 콘솔에 출력하는 delegate 에 전해집니다.

쓰로틀링(Throttling)

CCR DispatcherQueue 구현은 미리 지정된 정책에 따라 태스크 실행 쓰로틀링(throttling)을 허용합니다. 태스크 쓰로틀링 프로그램이 큰 메시지 부하를 처리하고, CCR 스케줄러가 큰 큐를 관리하는 복잡성을 처리하게 하는 CCR 의 주된 특별한 장점입니다. dispatcher 큐가 만들어질 때 쓰로틀링 정책이 지정됩니다. 다른 dispatcher 큐가 조정 요소의 활성화에 사용될 수 있기 때문에, 다른 핸들러를 위해 프로그래머는 다른 정책을 적용할 수 있습니다.

Task execution policy enumeration

예제 25

```
/// <summary>
    /// Specifies dispatcher queue task scheduling behavior
    /// </summary>
    public enum TaskExecutionPolicy
    {
        /// <summary>
        /// Default behavior, all tasks are queued with no constraints
        /// </summary>
        Unconstrained = 0,
        /// <summary>
        /// Queue enforces maximum depth (specified at queue creation)
        /// and discards tasks enqueued after the limit is reached
        /// </summary>
        ConstrainQueueDepthDiscardTasks,
        /// <summary>
        /// Queue enforces maximum depth (specified at queue creation)
        /// but does not discard anny tasks. It forces the thread posting any tasks after
the limit is reached, to
        /// sleep until the queue depth falls below the limit
```

```
        /// </summary>
        ConstrainQueueDepthThrottleExecution,
        /// <summary>
        /// Queue enforces the rate of task scheduling specified at queue creation
        /// and discards tasks enqueued after the current scheduling rate is above the
specified rate
        /// </summary>
        ConstrainSchedulingRateDiscardTasks,
        /// <summary>
        /// Queue enforces the rate of task scheduling specified at queue creation
        /// and forces the thread posting tasks to sleep until the current rate of task
scheduling falls below
        /// the specified average rate
        /// </summary>
        ConstrainSchedulingRateThrottleExecution
    }
```

```
        void ThrottlingExample()
        {
            int maximumDepth = 10;
            Dispatcher dispatcher = new Dispatcher(0, "throttling example");
            DispatcherQueue             depthThrottledQueue             =             new
DispatcherQueue("ConstrainQueueDepthDiscard",
                  dispatcher,
                  TaskExecutionPolicy.ConstrainQueueDepthDiscardTasks,
                  maximumDepth);

            Port<int> intPort = new Port<int>();
            Arbiter.Activate(depthThrottledQueue,
               Arbiter.Receive(true, intPort,
               delegate(int i)
               {
                   // only some items will be received since throttling will discard most
of them
                   Console.WriteLine(i);
               })
            );

            // post items as fast as possible so that the depth policy is activated and
discards
            // all the oldest items in the dispatcher queue
            for (int i = 0; i < maximumDepth * 100000; i++)
            {
                intPort.Post(i);
            }
        }
```

위 예에서 Dispatcher 인스턴스와 DispatcherQueue 를 생성했하고, 하나의 가능한 태스크 실행 옵션으로 그 다음 지속된 리시버를 접속하고, 가능한 한 빨리 100 만개 아이템에 알립니다. 만약 정책이 지정되지 않았다면, 100 만개의 태스크들은 각각이 얼마나 빨리 스케줄 되는가에 따라 해당 머신의 모든 CPU 코어에 스케줄 되고 dispatcher 큐의 깊이도 증가됩니다. 정책이 지정에 따라 CCR 은 가장 오래된 태스크를 버리고, 단지 마지막 10 개의 태스크를 실행을 위해 보관합니다. 이것은 단지 가장 최근의 N 개 메시지가 유용한(알림, 타이머, 등) 상황에서 매우 유용하다

주의: 각 아이템 포스트를 위한 Task 인스턴스를 생성할 하나의 아이템 리시버와 함께 depthThrottledQueue 인스턴스는 Arbiter.Activate 메서드에 공급됨에 주의하십시오.

정책 시나리오(Policy scenarios)

- ConstrainSchedulingRateDiscardTasks-CCR 핸들러가 규칙적인 일부 비율로 (초당 메시지수) 도착하는 메시지를 처리하는데 적당합니다. 이 경우 모든 메시지 보관은 중요하지 않지만 가장 최근의 메시지를 유지하는 것은 중요합니다. 이 정책은 메시지가 불규칙하게 도착하더라도 코드가 고정 비율로 실행되는 센서 알림, 타이머 이벤트에 적합합니다.
- ConstrainQueueDepthDiscardTasks- 마지막 N개 메시지는 보존되고 나머지는 버려도 되는 경우에 적합합니다. 가장 오래된 태스크를 버리고 가장 최근의 N 태스크를 스케줄하는 것을 이 정책은 보증합니다. 가장 유효한 깊이 임계치(depth threshold)는 1입니다, 왜냐면 그것이 가장 최근의 마지막 메시지/태스크를 유지기 때문입니다. 잘 알려진 평균 비율(메시지는 주기적이 아님)이 없는 critical 메시지, 센서 데이터, 종료 타이머를 처리하는 데 유용합니다.
- ConstrainSchedulingRateThrottleExecution- 주기적인 메시지의 소스가 같은 OS 프로세스내의 또 다른 스레드에 있을 때 적합합니다. 쓰로틀링은 리시버와 dispatcher 큐와 연관된 포트에 대한 Post 메서드 호출자의 Thread.Sleep을 야기합니다. 따라서 이 정책은 메시지 발신자의 속도를 낮추며, 이때 어떤 태스크도 버리지 않습니다.
- ConstrainQueueDepthThrottleExecution- 위와 비슷하지만 메시지가 어떠한 주기도 없고, 대신 랜덤 간격에 도착할 때 적당합니다. 이는 여러 다른 머신들에서 혹은 네트웍에서 메시지를 받는 포트에 일반적입니다. 이 경우도 네트워크 인바운드 조작을 하는 스레드를 제외한 다른 태스크들은 드롭 되지 않고 쓰로틀링 되며, 초당 메시지비율이 줄어진 메세지들이 CCR 포트에서 전달됩니다.

CCR 이터레이터(Iterator)

이터레이터는 CCR 에서 뛰어난 방법으로 사용되어 지는 C#2.0 언어의 특징입니다: 비동기 동작(또한 콜백으로서 알려져 있는)을 네스팅하기 위해 이용하는 delegate 대신 프로그래머가 CCR arbiter 혹은 다른 CCR 태스크에 yield 를 하면서 순차적인 방법으로 코드를 작성할 수 있게 합니다. 여러 단계의 비동기 로직은 이 후 하나의 이터레이터 메서드로 작성되어 코드의 가독성을 매우 개선시키며, 비동기 동작을 유지하고, 어떠한 OS 스레드도 yield 동안 블록화 되지 않으므로 대기된 수많은 조작에 적용가능 합니다.

예제 18

```
        void StartIterator()
        {
            // create an IterativeTask instance and schedule it
            Arbiter.Activate(_taskQueue,
                Arbiter.FromIteratorHandler(IteratorExample)
            );
        }
        /// <summary>
        /// Iterator method scheduled by the CCR
        /// </summary>
        IEnumerator<ITask> IteratorExample()
        {
            // accumulator variable
            int totalSum = 0;
            Port<int> portInt = new PortSet<int>();
            // using CCR iterators we can write traditional loops
            // and still yield to asynchronous I/O !
            for (int i = 0; i < 10; i++)
            {
                // post current iteration value to a port
                portInt.Post(i);
                // yield until a delegate executes in some other thread
                yield return Arbiter.Receive<int>(false, portInt, delegate(int j)
                {
                    // this delegate can modify a stack variable, allowing it
                    // communicate the result back to the iterator method
                    totalSum += j;
                });
            }
            Console.WriteLine("Total:" + totalSum);
        }
```

위에 예에서 StartIterator 메서드는 이터레이터 delegate 로부터 태스크를 만드는 Arbiter 클래스를 사용하고, 그 다음 Arbiter.Activate 를 사용해 스케줄링을 위해 그것을 제출합니다. 두 번째 메서드는 이터레이터 메서드로 그 내부에 실행을 제어하기 위해 yield return 과 yield

break C# 명령문을 사용할 수 있습니다. 이 메서드와 일반적인 C# 메서드의 차이점은 리턴 값입니다:

```
IEnumerator<ITask> IteratorExample(){
```

리턴 값은 이것이 CCR ITask 인스턴스에 대한 C# 이터레이터인것을 나타내고, 메서드내에 yield 명령문을 포함할 수 있다는 것을 컴파일러에 알립니다.

```
yield return Arbiter.Receive<int>(false, portInt, delegate(int j) {
```

위의 yield return 명령문은 CCR 스케줄러에 제어를 리턴합니다. 이것은 또한 Arbiter.Receive 헬퍼를 부를 때 만들어지는 ReceiverTask 에 의해 구현되는 ITask 인터페이스의 인스턴스를 리턴합니다. CCR 스케줄러는 리턴 된 Task 를 가동시키고 또한 이터레이터를 태스크와 연결시킵니다. 태스크가 실행을 완료할 때, 그것은 yield 이후 다음 코드 명령문에 이터레이터를 진행할 지 선택할 수 있습니다. 프로그래머는 이 코드에 대해 yield 를 동기점으로 생각할 수 있습니다: yield가 만족될 때 까지 이터레이터 메서드가 실행을 멈출 것입니다.

중요: 이터레이터 내에서 지속된(persist) 리시버에 절대 yield 해선 안됩니다. 위의 yield 명령문에서, Arbiter.Receive 의 첫번째 인수인 지속하기(persist) 가 false 로 설정되어 호출되었습니다. 만약 리시버가 지속된 경우, yield 는 절대 만족되지 않으며 이터레이터가 yield 이후 다음 명령문을 결코 진행하지 않을 것입니다.

로컬 변수를 이용한 암묵적 매개 변수 전달

C# 이터레이터와 CCR을 사용하는 능력은 두개의 다른 C#언어의 특징에 기인합니다:

1. 익명 메서드(Anonymous methods)- 인라인 delegate 로 핸들러를 정의하기 위해 예제에서 이 특징을 사용하고 있습니다.
2. 컴파일러는 익명의 메서드 내부에서 참조되는 이터레이터 메서드안의 모든 로컬 변수를 "캡처합니다". 이것은 delegate 가 부모 메서드에 정의된 로컬 변수를 사용하게 합니다. 항상 어떤 다른 스레드에서 실행되는 delegate 가 명시적 매개 변수 전달 없이 부모 이터레이터에 결과를 알릴 수 있습니다

```
// this delegate can modify a stack variable, allowing it // communicate the result back to the iterator method totalSum += j;
```

예제 18 의 delegate 의 내용을 위에 다시 나타냈습니다. totalSum 변수는 부모 이터레이터에 정의되었으나 delegate 내에서 변경하게 됩니다. 변수는 여러단계의 비동기 조작의 결과를 표시하는 결과 변수로 생각될 수 있으며 반복의 종료로 사용됩니다(이터레이터 메서드의 종료).

```
Console.WriteLine("Total:" + totalSum);
```

예 18 의 이터레이터 메서드 IteratorExample 안의 위 라인은 메서드의 마지막 명령문으로 암묵적으로 반복의 종료를 나타냅니다. delegate 에서 변경된 변수를 사용함에 주목하십시오.

조정 요소에 yield하는 것

예제 19

```
void StartIterator2()
        {
            Port<string> portString = new Port<string>();
            Arbiter.Activate(
                _taskQueue,
                Arbiter.ReceiveWithIterator(false, portString, StringIteratorHandler)
            );
        }
        IEnumerator<ITask> StringIteratorHandler(string item)
        {
            Console.WriteLine(item);
            yield break;
        }
```

위의 예는 리시브 조작의 결과로 이터레이터 메서드를 지정하는 방법을 보여줍니다. 지금까지 우리는 아비터가 만족될 때 실행하는 메서드를 위한 전통적 방법들을 사용해 왔습니다. 모든 주요한 조정 요소(JoinReceiver, MultipleItemGather, Choice, Receiver, 등)들에 대해 일반 메서드 또는 이터레이터 메서드를 프로그래머가 선택해 사용할 수 있습니다. 아비터 구현은 유저 delegate 의 타입을 숨기는 ITask 의 인스턴스와 같이 동작하므로 이터레이터와 비이터레이터 메소드 둘 다 작업 가능합니다.

예제 20

```
IEnumerator<ITask> IteratorWithChoice()
        {
            // create a service instance
            ServicePort servicePort = ServiceWithInterleave.Create(_taskQueue);

            // send an update request
            UpdateState updateRequest = new UpdateState();
            updateRequest.State = "Iterator step 1";
            servicePort.Post(updateRequest);

            string result = null;
```

```
            // wait for EITHER outcome before continuing
            yield return Arbiter.Choice(
                updateRequest,
                delegate(string response) { result = response; },
                delegate(Exception ex) { Console.WriteLine(ex); }
            );
            // if the failure branch of the choice executed, the result will be null
            // and we will terminate the iteration
            if (result == null)
                yield break;
            // print result from first request
            Console.WriteLine("UpdateState response:" + result);
            // now issue a get request
            GetState get = new GetState();
            servicePort.Post(get);
            // wait for EITHER outcome
            yield return Arbiter.Choice(
                get.ResponsePort,
                delegate(string response) { result = response; },
                delegate(Exception ex) { Console.WriteLine(ex); }
            );
            // print result from second request
            Console.WriteLine("GetState response:" + result);
        }
```

위의 예는 이터레이터의 일반적 사용을 보여줍니다: 서비스에 대한 다중 비동기 요청 결과는 성공 혹은 실패로 나타납니다. 실재 문제에서 N-1 개의 요청 결과로부터 N 개 요청들이 데이터 의존성이 있을 때 그들이 차례로 실행되어져야 합니다. Arbiter.Choice 의 리턴 값에 yield 하는 능력은 비동기 입출력 동작의 여러 결과를 간결하게 처리하기 위해 이터레이터 사용해 모든 인라인으로 처리하게 합니다. 이터레이터 메서드는 choice 실행 중 한 분기로 실행을 계속함에 주의하십시오. 이것은 또한 결과 스택(result stack) 변수를 요청의 결과를 검색하기 위해 사용합니다.

이터레이터의 네스팅

이터레이터 메서드가 쉽게 관리되기에 너무 커진 경우에는 작은 이터레이터 메서드로 분해시킬 수 있습니다. CCR 스케줄러는 언제 이터레이터가 yield break 에 도착하거나, 반복의 마지막 단계를 실행을 완료하는지를 알 수 있습니다.

예제 21

```
        IEnumerator<ITask> ParentIteratorMethod()
        {
            Console.WriteLine("Yielding to another iterator that will execute N
asynchronous steps");
            yield return Arbiter.ExecuteToCompletion(
```

```
                _taskQueue,
                new IterativeTask<int>(10, ChildIteratorMethod)
            );
            Console.WriteLine("Child iterator completed");
        }

        IEnumerator<ITask> ChildIteratorMethod(int count)
        {
            Port<int> portInt = new Port<int>();
            for (int i = 0; i < count; i++)
            {
                portInt.Post(i);
                yield return Arbiter.Receive(false, portInt, delegate(int j) { });
            }
        }
```

위에 예에 우리는 Arbiter.ExecuteToCompletion 메서드를 사용하여 이터레이터를 네스팅한 경우를 보여줍니다:

- 부모 이터레이터 메서드는 Arbiter.ExecuteToCompletion 에 의해 리턴 되는 태스크 인스턴스 실행을 yield 합니다.
- Arbiter.ExecuteToCompletion 은 제공된 dispatcher 큐 인스턴스의 IterativeTask 인스턴스를 스케줄합니다. Arbiter.FromIteratorHandler 는 명시적 IterativeTask 를 만드는 대신에 사용될 수 있습니다.
- 자식 이터레이터 메서드는 dispatcher 스레드 중의 하나에서 실행되고, 단순한 리시버 조작에 대해 10 번을 yield 합니다. 루프는 이터레이터가 비동기식 조작을 매우 쉽고 읽을 수 있게 해줌을 보여줍니다. 또한 부모 이터레이터는 복잡한 자식 이터레이터들이 얼마나 많은 비동기 단계를 실행하는지 알지 않고서도 yield 합니다.

Arbiter.ExecuteToCompletion 은 비 이터레이터 태스크도 실행가능 합니다.

중요: Arbiter.ExecuteToCompletion 호출의 문맥 내에서 실행된 핸들러에서 예외(Exceptions)가 발생하면 부모 이터레이터가 여전히 호출될 것입니다. 예외는 이터레이터를 암묵으로 종료하고 CCR 이 적절하게 그 경우 처리합니다.

CCR 실패 처리(Failure Handling)

전통적인 실패 처리 방법은 다음 패턴을 따릅니다:

1. 메서드의 동기 호출을 위해, 호출자는 메서드로부터 하나 이상의 리턴 값을 확인합니다. 메서드는 호출자 실행 문맥(종종 이것은 스레드입니다)을 사용합니다
2. 구성된 예외 처리를 사용해 호출자는 try/catch/finally 명령문의 동기식 메서드 호출을 감싸고, 에러 처리를 위해 catch{} 블록화를 사용하거나 catch {} 블록화와 메서드의 결과 확인을 조합해 사용합니다.
3. 스레드 전반에 문맥 흐름제어를 위해 OS 기반 구조와 복잡한 메커니즘같이 컴포넌트가 호출됨에 따른 다양한 보상 구현 방법을 사용합니다.

모든 위의 방법은 동시형, 비동기 실행에 쉽게 적용될 수 없습니다. 특히 위 두 개의 경우는 비동기 프로그램으로 작성이 어렵습니다. 메서드는 잠재적으로 호출자와 나란히 실행되는 임의의 문맥(context)에서 실행됩니다. 비동기식 조작의 결과를 모으는 블로킹 조작이 있지 않는 한, 호출자 문맥에서 스레드가 블록화 되어 실패 또는 성공의 결과가 쉽게 결정될 수 없습니다. CLR 의 비동기 프로그래밍 모델은 Begin/End 호출을 사용하여 조작의 결과에 관계없이 하나의 콜백에서 부가적인 호출과 예외 처리를 통해 실패 처리를 수행하므로 부가적인 복잡도를 가집니다. 비동기 조작은 실패가 처리되는 곳이 아니므로 코드가 읽기 쉬워 집니다.

CCR 은 다음 두 개의 방법으로 실패를 처리합니다:

1. **Choice 와 MultipleItemGather** arbiter **를 사용하여 명시적 혹은 로컬 실패를 처리합니다. 처리.** CCR 은 프로그래머가 성공과 실패 경우를 두 개의 메서드로 나누어 처리하고 그 공통된 부분으로 이어짐으로 이터레이터와 조합하여 실패를 다루는 타입 안전하고 강인한 방법을 제공합니다. **예제 8,13 과 20 은 각각 이터레이터에서 Choice, MultipleItemGather 와 Choice 를 사용해 명시적으로 에러를 처리하는 것을 보여줍니다.**
2. **암묵적이거나 분산형 실패 처리는 여러 개의 비동기 조작에 대해 네스팅과 확장이 가능한 인과율(**Causalities**)에 의거해 처리됩니다. 이는 논리적인 문맥 개념 또는 조작의 그룹화를 트랜잭션과 같이 처리하고 동시적 비동기 환경에 이를 확장합니다.** 이 메커니즘은 또한 머신 사이의 경계를 가로질러 확장 될 수 있습니다

인과율(Causalities)

인과율은 한 개의 원인에 논리적 근원을 두는 실행의 트리를 만드는 fork 와 join 같은 다중 실행 문맥에 적용되는 조작의 시퀀스를 나타냅니다. 이 논리적인 그룹화는 인과율 문맥이라 불리고, 메시지 전송자로부터 리시버(receiver)로 암묵적 흐름입니다. 리시버에서 활성화된

코드에 의해 보내진 어떤 메시지라도 이 인과율로 전송되고, 어떤 새로운 수신기에라도 메시지는 전파됩니다.

인과율은 스레드를 위해 구현된 구조화된 예외 메커니즘의 확장입니다. 이것은 네스팅을 허용하면서도 다른 인과율(예를 들어 join 의 의한)과 합쳐진 동시에 일어나는 다중 실패를 처리합니다.

예제 22

```
        void SimpleCausalityExample()
        {
1:          Port<Exception> exceptionPort = new Port<Exception>();
            // create a causality using the port instance
2:          Causality exampleCausality = new Causality("root cause", exceptionPort);
            // add causality to current thread
3:          Dispatcher.AddCausality(exampleCausality);

            // any unhandled exception from this point on, in this method or
            // any delegate that executes due to messages from this method,
            // will be posted on exceptionPort.

            Port<int> portInt = new Port<int>();
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false,portInt,IntHandler)
            );

            // causalities flow when items are posted or Tasks are scheduled
4:          portInt.Post(0);

            // activate a handler on the exceptionPort
            // This is the failure handler for the causality
5:          Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, exceptionPort,
                delegate (Exception ex)
                {
                    // deal with failure here
                    Console.WriteLine(ex);
                })
            );
        }

        void IntHandler(int i)
        {
            // print active causalities
            foreach (Causality c in Dispatcher.ActiveCausalities)
            {
                Console.WriteLine(c.Name);
            }
            // expect DivideByZeroException that CCR will redirect to the causality
6:          int k = 10 / i;
```

```
        }
```

위에 예에서 우리는 인과율이 비동기 조작에 대해 에러 처리를 도와주는 간단한 시나리오를 보여줍니다. 코드는 다음 주요 단계를 수행합니다:

1. Port<Exception> 인스턴스는 인과율 문맥 내에서 생기는 어떤 예외라도 저장하기 위해 생성됩니다.
2. 새로운 인과율 인스턴스는 친화적 이름이 더해진 예외 포트를 사용해 생성됩니다.
3. 인과율은 Dispatcher 가 현재의 실행 문맥(OS 스레드)를 위해 보관하는 인과율 리스트에 추가 됩니다.
4. 아이템이 포트에 포스트됩니다. 포스트가 인과율의 문맥 내에서 일어나기 때문에, 아이템이 포스트되는 곳에 인과율을 암묵적으로 연결할 것입니다. 핸들러가 이 아이템을 실행할 때, 아이템에 붙여진 인과율은 리시버를 실행하는 스레드에 추가될 것입니다. 리시버의 어떤 예외라도 인과율과 관련된 예외 포트에 포스트될 것입니다.
5. Receive 가는 예외 포트에 가동되고 따라서 인과율 내에서 어떤 비동기 조작으로 처리되지 않는 어떤 예외라도 수신하게 됩니다.
6. portInt 인스턴스에 포스트된 아이템으로 인해 핸들러가 스케줄되고 DivideByZeroException 을 던집니다. CCR 스케줄러는 인과율과 관련된 예외 포트로 예외를 리다이렉트(방향 시정)합니다.

예는 인과율이 실패를 다루는 가장 강력한 방법임을 보여줍니다. 만일 프로그래머가 다른 핸들러가 병렬로 실행하게 되는 부가적인 비동기식 조작을 추가하면 실패 처리는 변경하게 되지 않습니다. 게다가 이 핸들러가 하는 어떤 조작이라도 또한 원래의 인과율 범위 내에서 처리되며 예외 포트에 예외를 보냅니다.

네스트된 인과율

인과율을 네스팅하는 것은 종종 적합합니다. 따라서 각기 다른 실패 핸들러는 서로 다른 수준에서 실패를 보상합니다. 구조화된 예외와 유사하지만 동시 설정에 일반화되어 있는 인과율은 네스팅할 수 있으며, 프로그래머가 내부 인과율의 예외를 다루거나 어떤 부모 인과율에 예외를 명백히 포스팅하는 것 중에서 선택하게 해줍니다.

예제 23

```
        public void NestedCausalityExample()
        {
            Port<Exception> exceptionPort = new Port<Exception>();
```

```
            Causality parentCausality = new Causality(
                "Parent",
                exceptionPort);
            Dispatcher.AddCausality(parentCausality);

            Port<int> portInt = new Port<int>();
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, portInt, IntHandlerNestingLevelZero)
            );
            portInt.Post(0);

            // activate a handler on the exceptionPort
            // This is the failure handler for the causality
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false,
                exceptionPort,
                delegate (Exception ex)
                {
                    // deal with failure here
                    Console.WriteLine("Parent:" + ex);
                })
            );
        }

        void IntHandlerNestingLevelZero(int i)
        {
            // print active causalities
            foreach (Causality c in Dispatcher.ActiveCausalities)
            {
                Console.WriteLine("Before child is added:   " + c.Name);
            }

            // create new child causality that will nest under existing causality
            Port<Exception> exceptionPort = new Port<Exception>();
            Causality childCausality = new Causality(
                "Child",
                exceptionPort);
            Dispatcher.AddCausality(childCausality);

            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false,
                exceptionPort,
                delegate (Exception ex)
                {
                    // deal with failure here
                    Console.WriteLine("Child:" + ex);
                })
            );

            // print active causalities
            foreach (Causality c in Dispatcher.ActiveCausalities)
            {
```

```
                Console.WriteLine("After child is added:    " + c.Name);
            }

            // attach a receiver and post to a port
            Port<int> portInt = new Port<int>();
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, portInt, IntHandlerNestingLevelOne)
            );
            portInt.Post(0);
        }

        void IntHandlerNestingLevelOne(int i)
        {
            throw new InvalidOperationException("Testing causality support. Child
causality will catch this one");
        }
```

예제의 콘솔 출력:

```
Before child is added:   Parent
After child is added:    Child
Child:System.InvalidOperationException: Testing causality support. Child causality will
catch this one
   at          Examples.Examples.IntHandlerNestingLevelOne(Int32          i)          in
C:WmriWmainWCCRWtestsrcWUnitTestsWccruserguideexamples.cs:line 571
   at                        Microsoft.Ccr.Core.Task`1.Execute()                        in
C:WmriWmainWCCRWsrcWCoreWTemplatesWGeneratedFilesWTaskWTask01.cs:line 301
   at   Microsoft.Ccr.Core.TaskExecutionWorker.ExecuteTaskHelper(ITask   currentTask)   in
C:WmriWmainWCCRWsrcWCoreWscheduler_roundrobin.cs:line 1476
   at        Microsoft.Ccr.Core.TaskExecutionWorker.ExecuteTask(ITask&        currentTask,
DispatcherQueue p) in C:WmriWmainWCCRWsrcWCoreWscheduler_roundrobin.cs:line 1376
   at              Microsoft.Ccr.Core.TaskExecutionWorker.ExecutionLoop()              in
C:WmriWmainWCCRWsrcWCoreWscheduler_roundrobin.cs:line 1307
```

위에 예에서 다음과 같은 네스팅된 비동기 순서를 볼 수 있습니다:

1. NestedCausalityExample 메서드는 인과율을 추가하고 리시버를 가진 포트에 포스트 시킵니다.
2. IntHandlerNestingLevelZero 메서드는 NestedCausalityExample 메서드와 나란히 비동기적으로 실행되고 단계 1 에서 보낸 인과율 아래에서 네스팅된 새로운 인과율을 추가합니다
3. IntHandlerNestingLevelZero 메서드는 다른 리시버를 가진 새로운 포트를 만들고, 메시지를 포스트합니다

4. NestedCausalityExample 메서드에서 원래 포스트되어 간접적으로 발생한 IntHandlerNestingLevelOne 메서드가 실행되며 두 개의 인과율을 활성화 합니다. 스택의 최하단은 IntHandlerNestingLevelZero 메서드에 의해 만들어지는 자식 인과율입니다
5. 예외는 IntHandlerNestingLevelOne 메서드에서 발생하고 예외를 그 부모로부터 숨기므로 자식 인과율 핸들러에 의해 처리됩니다.

정적인 프로퍼티인 Dispatcher.ActiveCausalities 를 사용해 높은 수준으로 실패를 선택적으로 전파하면서 인과율 예외 처리기는 그 부모 인과율에 예외를 명백하게 알릴(포스트) 수 있습니다. 예는 IntHandlerNestingLevelZero 메서드의 최상위에 모든 활성화된 인과율을 인쇄하기 위해 이 모음을 사용합니다.

Join 과 인과율

CCR 인과율 구현의 독특한 특징은 두 개의 다른 실행 경로에서 오는 인과율을 조합할 수 있는 것입니다. 하지만 Join 혹은 다중 아이템 리시버가 만족되어 실행되는 핸들러에서 인과율을 처리해야 합니다. 그러한 시나리오에서 CCR 은 인과율을 네스트하지 않지만 join 된 아이템을 가진 핸들러에 이를 각각 추가합니다. 만일 예외가 던져지면, 그것은 두 개의 독립 인과율 스택을 따라 동시에 전파되어 각 아이템을 위한 핸들러에서 넘겨집니다.

예제 24

```
        public void JoinedCausalityExample()
        {
            Port<int> intPort = new Port<int>();
            Port<int> leftPort = new Port<int>();
            Port<string> rightPort = new Port<string>();

            Port<Exception> leftExceptionPort = new Port<Exception>();
            Port<Exception> rightExceptionPort = new Port<Exception>();

            // post twice so two handlers run
            intPort.Post(0);
            intPort.Post(1);

            // activate two handlers that will execute concurrently and create
            // two different parallel causalities
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, intPort,
                delegate (int i)
                {

                    Causality leftCausality = new Causality("left", leftExceptionPort);
                    Dispatcher.AddCausality(leftCausality);
                    // post item on leftPort under the context of the left causality
                    leftPort.Post(i);
```

```
                })
            );

            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, intPort,
                delegate (int i)
                {
                    Causality rightCausality = new Causality("right", rightExceptionPort);
                    Dispatcher.AddCausality(rightCausality);
                    // post item on rightPort under the context of the right causality
                    rightPort.Post(i.ToString());
                })
            );

            // activate one join receiver that executes when items are available on
            // both leftPort and rightPort

            Arbiter.Activate(_taskQueue,
                Arbiter.JoinedReceive<int, string>(false, leftPort, rightPort,
                delegate (int i, string s)
                {
                    throw new InvalidOperationException("This exception will propagate to
two peer causalities");
                })
            );

            // activate a handler on the exceptionPort
            // This is the failure handler for the causality
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, leftExceptionPort,
                delegate(Exception ex)
                {
                    // deal with failure here
                    Console.WriteLine("Left causality:  " + ex);
                })
            );

            // activate a handler on the exceptionPort
            // This is the failure handler for the causality
            Arbiter.Activate(_taskQueue,
                Arbiter.Receive(false, rightExceptionPort,
                delegate(Exception ex)
                {
                    // deal with failure here
                    Console.WriteLine("Right causality:  " + ex);
                })
            );
        }
```

예제의 콘솔 출력:

```
Left causality:  System.InvalidOperationException: This exception will propagate to two peer causalities
Right causality:  System.InvalidOperationException: This exception will propagate to two peer causalities
```

위 예에서 익명의 메서드는 가독성(readability)을 위해 한 개의 메서드 내에서 모든 로직을 유지하도록 무겁게 사용됩니다. 예제는 다음을 보여줍니다:

- 두 개의 독립 핸들러가 두 개의 인과율("왼쪽"와 "오른쪽"으로 이름 지어진)을 만듭니다.
- 두 개의 다른 포트 각각에 한 개의 메시지를 포스트합니다.
- 세 번째 핸들러가 이 두개의 포트(leftPort 와 rightPort)에 대해 만족하는 join 을 위해 실행됩니다.
- join 핸들러는 양쪽 인과율 문맥에 예외를 던집니다

요점은 join 핸들러가 실행되고 예외를 던질 때, 두 개의 동등한 인과율이 활성화되고 둘 다 독립적으로 그들 각각의 예외 포트에 포스트된 예외를 받게 되는 것입니다.

중요한: 인과율은 일반적 CCR 포트를 예외를 받기 위해 사용하기 때문에, CCR 조정 프리미티브를 다중 인과율에 대한 예외 처리를 조합하기 위해 사용할 수 있습니다. Join, interleave 와 choice 들은 모두 동시형, Multi-tiered 비동기 조작에 대한 실패 처리를 조합하는 적합하고 강력한 방법입니다.

비CCR 코드와 CCR의 상호운영

스레드 아파트 제약조건(Thread apartment constraints)

몇 개의 전통적 윈도우 컴포넌트들, 특히 COM 객체들은 그들과 상호 작용할 때 특정 스레드 아파트 정책을 필요로 합니다. 훨씬 더 최근의 프레임워크인 .NET WinForm 들도 WinForm 을 호스트하는 스레드에 SingleThreadedApartment 정책을 필요로 합니다.

CCR 은 STA 컴포넌트들을 쉽게 호스트하고 상호운영할 수 있습니다: 컴포넌트들은 한 개의 스레드와 그 생성자에 적합한 스레드 아파트 정책을 가지고 CCR Dispatcher 클래스의 인스턴스를 만듭니다. DispatcherQueue 인스턴스는 그 다음 전통적 객체와 상호운영하기 위한 핸들러를 가동시킬 때 이 dispatcher 를 사용할 수 있습니다. 단순히 일반적인 CCR 포트에서 포스트하는 다른 CCR 컴포넌트에 STA 컴포넌트의 동질성(affinity)을 숨기고 포트에 붙여지는 핸들러가 어떤 dispatcher 을 사용하는지 상관없이 이 핸들러들은 COM 또는 winform 객체에 안전하게 액세스할 수 있습니다.

CCR WinForm 어댑터 라이브러리는 CCR(Ccr.Adapters.Winforms.dll) 내에서 .NET 윈도우 폼을 호스트하는 편리한 방법 중 하나입니다.

메인 애플리케이션 스레드 조정(Coordination with main application thread)

CCR 소프트웨어 컴포넌트들은 종종 독립형 실행화일 같이 CLR 애플리케이션의 문맥에서 실행됩니다. .NET 런타임은 프로그램들을 한 개의 OS 스레드를 사용해 시작하고, 스레드가 종료될 때 그들을 종료합니다. CCR 애플리케이션들이 비동기성과 동시성을 가지므로, 메시지가 전송되지 않을 때 활성화 되지 않으며 어떤 스레드도 거의 블록화 하지 않습니다. CCR dispatcher 는 스레드를 효율적인 슬립 상태로 유지할 것입니다. 그러나 만일 이 스레드가 백그라운드에서 생성된 경우에는 CCR 이 아이템을 처리하는 동안이라도 애플리케이션이 종료될 수 있습니다.

CLR 시작의 동기적 부분들을 인터페이스 하는 가장 일반적인 유형 중 하나는 CCR 애플리케이션이 완료될 때까지 System.Threading.AutoResetEvent 를 사용하고 메인 애플리케이션 스레드를 블록화 하는 것입니다. 이벤트는 어떤 CCR 핸들러에 의해서도 호출될 수 있습니다.

예제 25

```
        void InteropBlockingExample()
        {
            // create OS event used for signalling
            AutoResetEvent signal = new AutoResetEvent(false);
            // schedule a CCR task that will execute in parallel with the rest of
            // this method
```

```
            Arbiter.Activate(
                _taskQueue,
                new Task<AutoResetEvent>(signal,SomeTask)
            );
            // block main application thread form exiting
            signal.WaitOne();
        }
        /// <summary>
        /// Handler that executes in parallel with main application thread
        /// </summary>
        /// <param name="signal"></param>
        void SomeTask(AutoResetEvent signal)
        {
            try
            {
                for (int i = 0; i < 1000000; i++)
                {
                    int k = i * i / 10;
                }

            }
            finally
            {
                // done, signal main application thread
                signal.Set();
            }
        }
```

위에 예에서는 OS 이벤트를 사용해 메인 애플리케이션 스레드를 블록화 하는 일반적인 경우를 보여줍니다.

.NET 비동기 프로그래밍 모델 어댑터(Programming Model Adapters)

CCR 은 APM Begin/End 비동기 패턴을 구현하는 클래스를 감쌀(wrap) 수 있습니다. 작은 CCR 클래스는 APM 패턴의 세부사항과 복잡도를 요약한 포트(또는 PortSet)를 리턴할 수 있습니다. MSDN 문서 중 Concurrent Affairs 는 이 상호운용 패턴을 매우 상세해 보여줍니다.

다른 동시성 접근법들

CCR 은 (필요할 경우 간단한 헬퍼 메서드와 함께) 다양한 다른 동시 실행이 방법들을 표현할 수 있습니다. 그 예는 다음과 같습니다.

1. **.NET 의 비동기 프로그래밍 모델(APM):** The Concurrent Affairs 문서는 APM API 로 시스템 라이브러리를 관리하기 위한 CCR 어댑터 헬퍼의 몇 가지 예를 보여줍니다.
2. **Futures:** Futures 는 클래스로부터 포트를 리턴하고 병렬로 핸들러를 가동시키는 것으로 표현됩니다. 호출자 병령로 코드를 실행하는 dispatcher 큐를 사용하고 그 뒤 결과(동기식 향후 패턴을 가정하는 것)를 기다리는 퍼블릭 메서드를 블록화 하기 위해 OS 이벤트를 사용합니다. 향후 작업 아이템을 스케줄하기 위해 CCR 스케줄러 설계는 CPU 처리 능력을 적절히 사용하고 쓰로틀링과 로드 밸런싱에 대해 사용자가 지정된(user specified) 적절한 정책을 사용합니다. Dispatcher 큐는 FIFO 이지만 DispatcherQueue 클래스의 구현은 가상 메서드를 사용해 LIFO, 랜덤 삽입 등의 구현을 위한 상속 클래스를 가능하게 합니다.
3. **Join:** Join 프로그래밍은 운영 체제에서 오랜기간 사용되어져 왔으며(NT 의 WaitForMultiple, I/O Completion port 등) CCR 은 동적 join 과 다중 아이템 리시버를 이용해 이것을 한층 더 확장했습니다.
4. **구독자/작성자 잠금(Reader/Writer Locks), 잠금(Lock), 모니터와 같은 전통 스레딩 요소들:** CCR 은 공유 메모리 생성에 대한 명백한 락 대신에 포트와 Arbiter 를 통하여 스레딩 동기 요소들을 표현할 수 있습니다. 문제는 이 후 공유 메모리를 보호하는 대신에 스케줄링과 프로토콜 설계 중 하나가 됩니다.

3. DSS 사용자 가이드

프로그램 언어 선택

당신에게 맞는 올바른 프로그램 언어를 선택하는 것은 중요합니다. 마이크로소프트 Robotics Studio 는 전통적 .NET 언어인 C#과 VB.Net 에서 스크립트 언어인 Python 에서 새로운 시각적 프로그램 언어인 마이크로소프트 비주얼 프로그래밍 언어(VPL) 까지 다양한 프로그램 언어를 지원합니다.

C# 과 VB.Net

C# 및 VB.Net 은 일반적인 .Net 언어 이고 튜토리얼 및 견본 중 많은 것이 이 공통 언어로 쓰여져 있습니다. C#그리고 VB.Net 코드 작성을 위해 마이크로소프트 비주얼 스튜디오 Express 버전을 포함하여 모든 버전의 비주얼 스튜디오를 이용할 수 있습니다.

> 비주얼 스튜디오 Express 버전은 한 개의 언어를 지원합니다. 예를 들어, 비주얼 스튜디오 Visual basic 2005 Express 판이 있으면 VB.Net 코드 만을 사용하며, 다른 비주얼 스튜디오 Express 버전도 동일 합니다. 다른 Net 언어를 사용하고 싶으면 동일한 머신에 각각의 비주얼 스튜디오 Express 를 나란히 설치할 수 있습니다.

Python

Python 은 대중적인 스크립트 언어 이고 IronPython 과 함께 지금.Net 언어를 지원합니다. Python 을 이용해 당신의 로보틱스 어플리케이션을 다루고 오케스트레이션 할수 있고, 기본적인 로보틱스 튜토리얼은 마이크로소프트 로보틱스 스튜디오와 Python 을 어떻게 사용하는지를 보여줍니다.

C++

C++은 방대한 라이브러리를 가진 잘 갖춰진 대중적인 프로그램 언어입니다.. 마이크로소프트 로보틱스 스튜디오와 함께 C++을 사용할 수 있는 몇가지 방법은 다음과 같습니다.

1. C++을 위한 managed 확장을 사용한 DSS 서비스 작성
2. .Net COM 상호호완성을 이용한 기존 C++ 라이브러리를 사용하거나 managed 코드로부터 native 함수를 부르는 방법

보다 자세한 정보를 원하시면 아래의 온라인 레퍼런스를 방문하세요. 기본적인 로보틱스 튜토리얼은 마이크로소프트 로보틱스 스튜디오와 C++를 사용하는 몇가지 예제를 보여줍니다.

마이크로소프트 비주얼 프로그래밍 언어 (VPL)

VPL 은 그래픽 데이터 플로우에 기반한 프로그램 모델로 고안된 어플리케이션 개발 환경입니다. 이는 연속해서 수행되는 명령들의 시리즈이기 보다는 데이터 플로우 프로그램은 물자가 도착할 때 그들에게 할당한 업무를 하는 작업 라인의 일련의 노동자 같은 것입니다. 그 결과로 VPL 은 다양한 동시성과 분산 프로세싱 시나리오를 프로그래밍하는데 적합합니다.

VPL 은 변수와 로직같은 개념의 기본적인 이해에서 출발하는 초보 프로그래머를 위해 사용됩니다. 하지만 VPL 은 초보로 그용도가 제한하지 않습니다. 이 프로그램 언어의 조합적 성격은 빠른 prototyping 이나 코드 개발을 위해 상급 프로그래머에게 필요할 것입니다. 또한 이 언어의 주된 사용이 로봇 어플리케이션 이지만, 근본적인 언어의 구조는 로봇으로 제한되지는 않으며 다른 어플리케이션에도 적용될 수 있습니다. 그 결과 VPL 은 학생, enthusiasts/동호인을 포함해 웹 개발자와 전문 프로그래머까지 광범위한 사용자에게 필요할 것입니다. 기본적인 로보틱스 튜토리얼은 VPL 의 실재 사용 예제들을 보여주며, 튜토리얼들을 통해 쉽게 VPL 을 배울 수 있습니다.

서비스의 이해

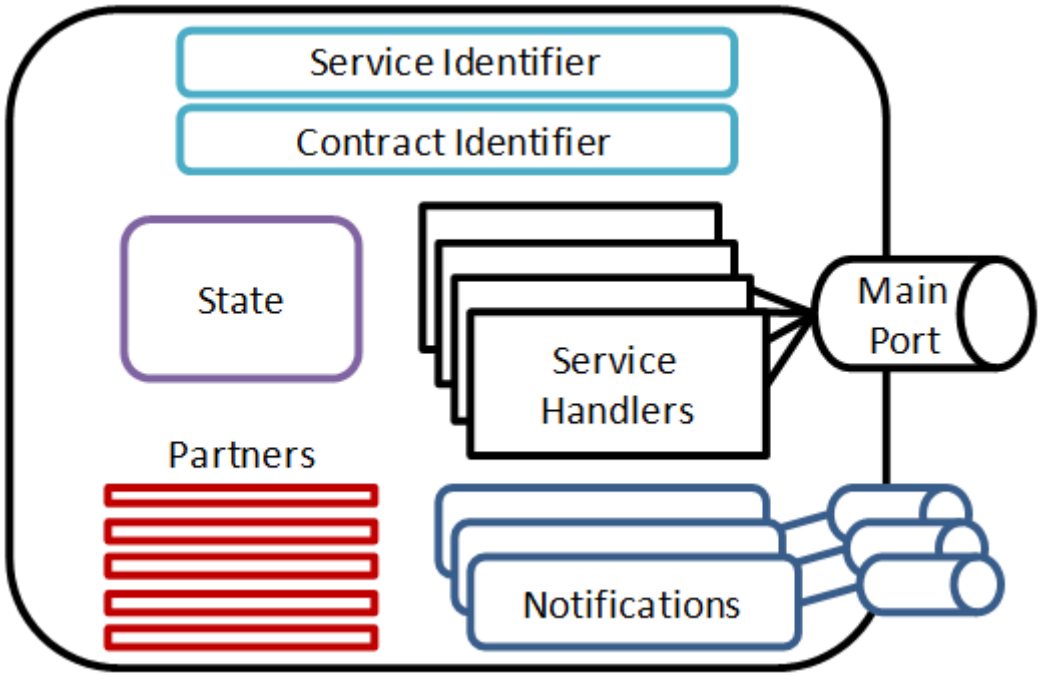
서비스 컴포넌트(Service Components)

서비스는 마이크로소프트 Robotics Studio 를 이용해 애플리케이션을 제작하기 위한 기본적인 빌딩 블록이며 DSS 애플리케이션의 모델의 주요 구성입니다. 서비스 튜토리얼(Service Tutorials Overview)은 시위를 한다 ? 여기에 기술된 개념을 잘 보여줍니다. 다음에 제한되지는 않지만 서비스는 다음을 포함한 어떤 것도 표현하기 위해 사용될 수 있습니다:

1. 센서와 엑츄에이터와 같은 하드웨어 구성 기기
2. 유저 인터페이스, 기억 장치, 디렉터리 서비스, 등과 같은 소프트웨어 구성 요소
3. 집단화: 센서 융합, mash-ups, 등

서비스는 DSS 노드 문맥(context)내에서 실행됩니다. DSS 노드는 삭제 되든지 DSS 노드가 멈출 때 때까지 서비스를 생성 및 관리해주는 호스팅 환경입니다. 서비스는 본래적으로 네트웍 사용 가능하며 같은 DSS 노드 내에서 또는 네트워크를 가로질러 실행됨에 상관없이 균일한 방법으로 서로 의사소통할 수 있습니다.

DSS 서비스 모델은 서비스의 재사용이 용의하고, 서비스들 사이의 매우 느슨한 연결되어 있으면서 서로 조립하고 사용하기 용이하게 디자인되었습니다. 이 섹션에 기술된 모든 DSS 서비스는 아래와 같이 컴포넌트의 공통 집합으로 구성됩니다.



서비스 식별자(Service Identifier)

서비스 인스턴스가 DSS Node 내에서 만들어질 때, 그것은 Constructor Service 에 의해 URI 를 동적으로 할당됩니다. 이 서비스 식별자는 특정 DSS 노드에서 실행되는 서비스 식별자의 인스턴스를 가르킵니다. 서비스 식별자는 다른 서비스와 의사소통하고 웹브라우저를 통하여 서비스를 접근하게 합니다. 서비스 식별자는 단지 서비스 인스턴스의 존재(identity)를 제공합니다. 서비스 상태, 동작 또는 문맥에 대하여 정보를 알리지 않습니다.

서비스 식별자는 다음 단계에 의해 결정됩니다:

1. 만일 서비스 인스턴스 이름이 Service Manifests 에서 표시하게 되면 그 이름은 사용된다
2. 만일 서비스 인스턴스 이름이 Create 에 표시되면, 이 요청은 직접 System Services 에 직접 보내어져 그 이름은 사용된다

서비스 구현이 다음과 같은 ServicePort 속성을 가질 경우,

```
[ServicePort("/mysample")]
```

그 이름은 "/mysample" 이 추가된 DSS 노드의 호스트와 포트 이름입니다. 예를 들면

```
http://machine:port/mysample
```

3. 서비스 구현이 AllowMultipleInstances = true 로 ServicePort 속성을 가질 경우,

```
[ServicePort("/mysample", AllowMultipleInstances = true)]
```

서비스 인스턴스의 이름은 위와 같지만 다른 인스턴스와 이 인스턴스를 좀더 구별하기 위해 추가된 GUID 를 가지고 있다:

```
http://machine:port/mysample/d88441c9-8319-407c-b554-0b0bfd90050b
```

컨트랙트 식별자(Contract Identifier)

컨트랙트는 다른 서비스가 주어진 컨트랙트(Contract)가 가리키는 서비스를 조립하고 재사용 할 수 있게 동작을 기술하는 서비스 구현과 압축된 설명입니다. 서비스의 컨트랙트는 DSS Contract Information Tool (DssInfo.exe)를 사용하여 조사할 수 있습니다. 컨트랙트는 직접 서로 연결되는 것이라기 보다 서로 다른 서비스간 대화에 대한 서비스 연결인 DSS proxy DLL 들을 생성하기 위해 사용됩니다.

컨트랙트 식별자는 서비스의 컨트랙트를 유일하게(uniquely) 구분하는 URI 입니다. 서비스 프로젝트를 만들 때 컨트랙트 식별자는 자동적으로 생성됩니다. 이의 예는 DSS New Service Generation Tool(DssNewService.exe)를 사용해 새로운 서비스를 생성하는 경우입니다. 컨트랙트는 식별자는 다음 형식으로 구성되어 변경됩니다.

```
http://schemas.tempuri.org/[year]/[month]/[name].html
```

서비스 상태(Service State)

서비스 상태는 어떤 주어진 시간에서 서비스의 표현입니다. 서비스 상태는 서비스의 현재 컨텐츠를 기술하는 문서로 생각할 수 있습니다. 상태 문서의 예는 다음과 같습니다:

1. 모터를 표현하는 서비스 상태는 분당 회전수, 온도, 유압과 연료 소비량으로 구성될 수 있습니다.
2. 작업 큐를 표현하는 서비스는 큐에 들어있는 작업 아이템과 그들의 현재 상태 목록을 포함할 수 있습니다. 작업 아이템들 그들 자신은 단순히 그들의 존재들로 단순히 참조되는 작업 큐로 서비스될 수 있습니다.
3. 키보드를 표시하고 있는 서비스는 키가 눌려진 정보를 포함할 수 있습니다.

DSS 서비스의 일부로서 검색 되거나, 변경하게 되거나, 감시 되는 어떤 정보라도 서비스 상태의 일부로서 표현되어야 합니다.

서비스의 현재의 부분을 캡처하는 것과 더불어, 상태는 서비스의 UI 를 동작하기 위해 사용될 수 있습니다.

서비스 파트너(Service Partners)

DSS 애플리케이션 모델의 중요한 부분 중 하나는 보다 높은 수준의 기능을 제공하기 위해 서비스의 조립을 가능하게 한 것입니다. 이것은 서비스가 서로를 효과적으로 서로를 이용하기 위해, 서비스들은 탐지(discover)할 수 있으며 서로와 효율적이고 예측할 수 있는 방법으로 통신가능 해야 합니다. 하지만 서비스들이 느슨히 연결된 경우, 한 서비스가 다른 서비스에 의존하는 지와 심지어 이 다른 서비스가 어디에 있는지, 사용가능한지에 대해 미리 알 수 없습니다. 이 문제를 해결하기 위해, DSS 애플리케이션 모델은 파트너 서비스의 개념을 제공합니다.

파트너 서비스는 서비스가 적절하게 상호 작용하고, 잘 동작하기 위해 의존하는 다른 서비스들을 말합니다. 다른 서비스들을 파트너들로 선언함을 통해 서비스들은 서로 연결되며, 이 서비스들은 런타임에 해당 서비스 그 자체의 생성과정의 일부가 됩니다. 각 파트너와 관계를 잘 통제하도록 풍부한 설정을 옵션을 가진 Partner 속성을 사용하여 파트너를 선언합니다. 예를 들면, 서비스는 파트너가 서비스가 동작하기 위해 필요하다라고 단언할 수 있고, 만일 그 파트너가 찾아질 수 없으면 시작에 대한 서비스를 시작하지 못한다고 선언할 수 있습니다. 파트너는 또한 런타임에 옵션으로 정의될 수 있지만, 생성과정이 실패할 경우 무시됩니다.

메인 포트(Main Port)

메인 포트는 다른 서비스로부터의 메시지가 도착하는 CCR 포트입니다. 서비스 구현이 서로 직접 연결되지 않기 때문에, 서비스는 단지 또 다른 서비스에게 메인 포트로 메시지를 보내는 것에

의해 통신하게 됩니다. 메인 포트 그 자체가 서비스 클래스의 private 멤버이며 ServicePort 속성에 의해 구분 되어집니다. 다음은 메인 포트 선언 예입니다.

```
[ServicePort("/mysample")]?Orivate MySampleOperations _mainPort = new MySampleOperations()
```

메인 포트에서 받아진 메시지는 포트의 타입에 의해 정의되며, 위의 경우 MySampleOperations 타입에 의합니다. 메인 포트에서 정의된 모든 동작은 DSSP 혹은 HTTP(웹브라우저를 통하여 Accessing Services 를 본다)에 의해 정의된 메시지 조작을 따라야만 합니다. DSSP 조작을 지원하는 메인 포트 정의의 예는 LOOKUP, DROP, GET 와 REPLACE 이고, HTTP 조작의 경우는 GET 와 POST 입니다.

```
public class MySampleOperations: PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, Replace, HttpGet, HttpPost> ㅣㅣ { }
```

서비스 핸들러(Service Handlers)

메인 포트에서 정의된 DSSP 조작을 위해, 서비스 핸들러는 그 포트에 등록되어 도착 메시지를 처리합니다. 유일한 예외는 DSS 런타임이 디폴트 핸들러를 등록한 DsspDefaultLookup 과 DsspDefaultDrop 입니다. 예를 들면, 위에 MySampleOperations 정의의 경우, MySample 서비스는 Get, Replace, HttpGet 와 HttpPost 를 위해 핸들러를 등록할 것입니다.

서비스 핸들러는 ServiceHandler 속성을 사용하여 선언적으로 등록될 수 있습니다. 다음은 DSSP GET 조작을 위해 서비스 핸들러를 등록한 예입니다:

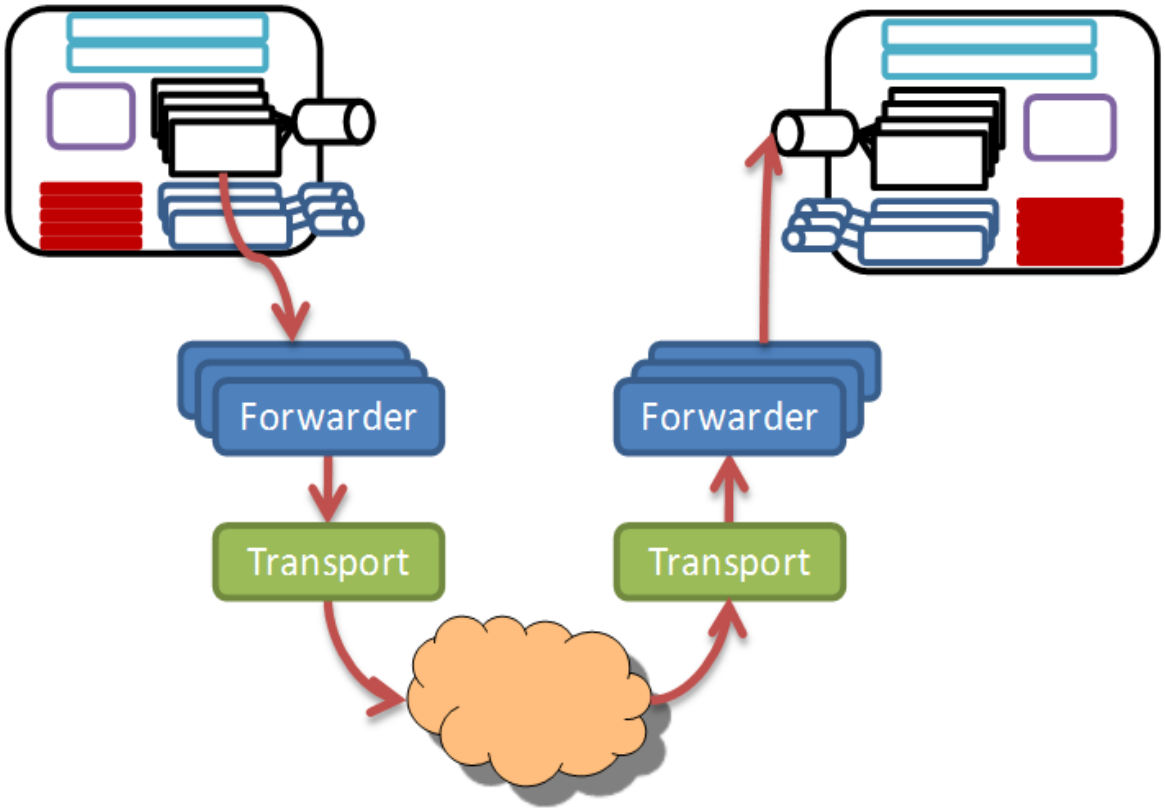
```
[ServiceHandler]
public IEnumerator<ITask> GetHandler(Get get)
{
get.ResponsePort.Post(_state);
yield break;
}
```

서비스 핸들러의 문맥 내에서 서비스가 다른 서비스에 메시지에 보낼 수 있습니다. 이때 서비스가 메시지를 보내는 두가지 방법은 다음과 같습니다:

1. 다른 서비스에 보내질 요청 메시지형식.
2. 이벤트 알림을 생성하는 서비스 내에서 상태 변경의 결과로서 서브스크라이버(subscriber)에 보내지는 이벤트 알림 형식.

둘 중 어느 쪽이든, 메시지는 원격 서비스의 메인 포트를 나타내는 로컬 CCR Port 인 서비스 포워더(*service forwarder*)를 통해 전해집니다. 메시지가 서비스 포워더를 통하여 보내질 때 런타임을 통해 네트웍으로 메시지를 다른 서비스의 트랜스포트(transport) 로 연결(route)하는

트랜스포트 에 도달할 때까지 포워드 될 것입니다. 여기서 메시지는 수신측 서비스의 메인 포트에 도착할 때까지 런타임을 통하여 유지됩니다.



이벤트 알림(Event Notifications)

DSS 서비스에서 사용되는 공통 유형은 서비스가 다른 서비스를 서브스크립션(subscribing)하는 것입니다. 서비스는 자신의 상태 변경의 결과로 이벤트 알림을 생성한다. 서비스가 다른 서비스에 서브스크립션하여, 서비스는 개별 CCR 포트에 대한 이벤트 알림을 수신합니다. 서로 다른 포트마다 각각 서브스크립션을 함을 통해 각각의 이벤트 알림을 구분하고 이벤트 알림이 관계된 서브스크립션을 결정하는 것이 가능합니다.

이벤트 알림이 서비스의 상태 변경에 결과로 생성되므로 상태 변경은 DELETE, INSERT, UPDATE, 등과 같은 DSSP 조작의 결과로 나타납니다. 실제 이벤트 알림 메시지들은 상태 변경에 영향을 미치는 메인 포트에 대한 조작들 입니다. 그 결과로 이벤트 알림이 도착하는 로컬 알림 포트는 그 특정 서브스크립션과 연관된 메인 포트 같은 타입입니다.

서비스 데이터 모델(Service Data Model)

서비스 상태는 특정 주어진 시간에서의 서비스의 표현입니다. 서비스 상태는 서비스의 현재 컨텐츠를 기술하는 문서로 생각할 수 있습니다. 서비스의 현재의 부분을 캡처하는 것과 더불어, 상태는 또한 서비스의 웹 기반의 UI 를 동작하는 데 사용된다.

서비스의 상태를 정의할 때 CLR 클래스, 필드, 속성들을 DataContractAttribute, DataMemberAttribute 와 DataMemberConstructorAttribute 를 이용해 추가적으로 기술해야 합니다. 이는 컴파일시 자동으로 생성될 DSS 프록시 DLL 에 타입을 포함하기 위함 입니다. DSS 프록시 생성은 명백하게 기술되지 않은 타입을 프록시 DLL 에 포함되지 않도록 하기위해 "opt-in" 모델을 사용합니다.

이 속성을 사용하는 방법은 서비스 튜토리얼을 참고하시기 바랍니다.

DataContractAttribute

DataContractAttribute 는 Classes, Structs 와 Enumerations 에 퍼블릭 컨트랙트(Contract)의 일부로서 서비스 구현을 위해 생성되는 DSS 프록시 DLL 에 이 타입들이 포함되도록 하기 위해 사용됩니다.

DataMemberAttribute

DataMemberAttribute 는 Fields 와 속성에 퍼블릭 컨트랙트의 일부로서 서비스 구현을 위해 생성하게 되는 DSS 프록시 DLL 에 이 구성 요소가 포함되는 포함 되도록 하기 위해 사용됩니다.

DataMemberConstructorAttribute

DataMemberConstructorAttribute 는 초기화 매개변수를 포함하고 있는 생성자 상속이 프록시 DLL 속에 자동적으로 만들어지도록 할 때 사용됩니다. 속성은 DataContractAttribute 또는 DataMemberAttribute 와 조합되어 다음과 같이 사용됩니다:

- DataContractAttribute 와 조합. 이 속성은 DataMemberAttribute 라고 표시된 모든 필드 혹은 속성을 가진 생성자(constructor)를 만듭니다.
- DataMemberAttribute 와 조합. 이 속성은 만들어진 생성자에 리스트된 필드 또는 속성에 순서를 조정합니다. 만약 순서가 0(디폴트)으로 설정된 경우, 어휘 순서는 보존된며, 1 로 설정되면 필드 또는 속성은 만들어진 생성자에 명시적으로 포함되지 않습니다.

서비스 프로젝트 관리하기

서비스 프로젝트 생성(Creating Service Projects)

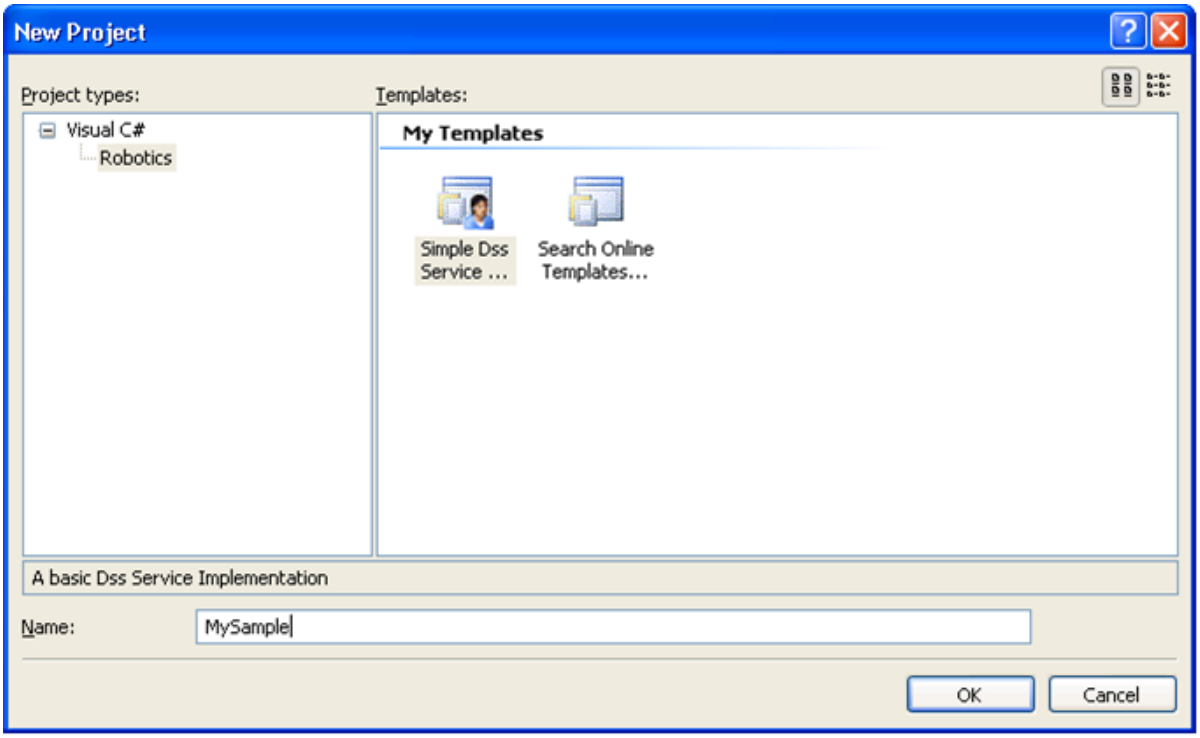
서비스 프로젝트는 커맨드 라인 인터페이스, 비주얼 스투디오에 통합된 솔루션, 마이크로소프트 비주얼 프로그래밍 언어(VPL) 등 유저가 즐겨 사용하는 여러 방법으로 만들어 집니다.

DSS New Service 생성 툴

DSS New Service 생성 툴(DssNewService.exe)은 마이크로소프트 Robotics Studio 커맨드 프롬프트로부터 실행될 수 있는 커맨드 라인 툴입니다. 이 도구는 프로젝트의 속성을 결정하는 여러 가지 옵션을 사용해 VB.Net, C#과 C++를 위한 서비스 프로젝트 만들 수 있습니다. 또한 윈도 CE 플랫폼을 목표로 하는 서비스 프로젝트를 생성하기 위해 사용됩니다.

마이크로소프트 비주얼 스투디오 새 프로젝트 위저드

비주얼 스투디오 안에서, "파일(File)" 메뉴에서 "새 프로잭트(New Project)"를 선택한 뒤, 원하는 CLR 언어(C#또는 VB.Net)를 고르고 "Simple DSS Service Template"을 선택함으로 DSS 서비스 프로젝트를 생성할 수 있고. 템플릿의 수에 따라 다르지만 다음과 유사하게 보일 것입니다:



서비스 프로젝트 개요

서비스 프로젝트 생성에 기술된 바와 같이, 서비스 프로젝트는 마이크로소프트 비주얼 프로그래밍 언어 또는 여러 가지 .Net 언어를 사용하여 만들어질 수 있습니다. 그러나, 서비스 프로젝트가 만들어지는 방법에 관계없이, 그것은 아래와 같은 컴포넌트의 공통 요소들을 가집니다.

서비스 프로젝트 파일

비록 같은 방법이 다른 프로그래밍 언어에도 적용될 것이지만, 설명을 위해 우리는 DSS New Service 생성 툴(DssNewService.exe)를 사용해 기본 예제 C#프로젝트를 만듭니다. 또한 비주얼 스투디오 내에서 "New Project"를 선택 후 적합한 마이크로소프트 Robotics Studio 프로젝트 템플릿(참조 서비스 프로젝트 생성)을 선택해 새로운 서비스 프로젝트를 직접 만들 수 있습니다.

마이크로소프트 Robotics Studio 커맨드 프롬프트에서 예제 C#서비스 프로젝트를 만들기 위해 다음을 입력하세요:

```
DssNewService.exe /s : SimpleSample ||
```

SimpleSample 라고 불리는 새로운 디렉터리가 생성됨을 다음 라인을 입력해 조사할 수 있습니다.

```
cd SimpleSample dir
```

결과적인 출력은 다음(주의, 타임 스탬프와 크기가 바뀔지도 모름)과 유사하게 보일 것 입니다.

```
10:00 10:00의 06/03/07 674 10:00 AssemblyInfo.cs 06/03/07 2,229 10:00 SimpleSample.cs
06/03/07 5,048 10:00 SimpleSample.csproj 06/03/07 1,233 10:00 SimpleSample.csproj.user
06/03/07 398 10:00 SimpleSample.manifest.xml 06/03/07 909 SimpleSample.sln 06/03/07
5,375 SimpleSampleTypes.cs
```

서비스 프로젝트는 3 개의 카테고리: 비주얼 스투디오 파일, 소스 코드 파일과 매니페스트 파일로 구성되며, 아래와 같이 나타납니다:

비주얼 스투디오 파일

SimpleSample.csproj, SimpleSample.csproj.user 와 SimpleSample.sln 파일들은 비주얼 스투디오에 의해 서비스 프로젝트를 관리하기 위해 사용됩니다. 비주얼 스투디오는 솔루션과 프로젝트라고 불리는 2 종류의 컨테이너를 이용해 서비스를 개발에 필요한 참조, 폴더, 파일과 종속성 같은 요소들을 관리합니다. 또한, 비주얼 스투디오는 Solution Folders 를 관련된 프로젝트를 그룹으로 구성하기 위해 제공하고, 그 다음 프로젝트의 그 그룹에 대해 동작을

수행합니다. 컨테이너들과 관련된 아이템들을 보고, 관리하기 위한 인터페이스인 솔루션 탐색기는 비주얼 스투디오 통합 개발 환경(IDE)의 부분입니다.

비주얼 스투디오 내에서 서비스 프로젝트를 열기 위해 스타트 메뉴로부터의 비주얼 스투디오를 실행하거나 다음 예와 같이 마이크로소프트 Robotics Studio 커맨드 프롬프트의 커맨드 라인에서 프로젝트 또는 솔루션의 이름을 입력합니다.

```
SimpleSample.sln
```

만일 비주얼 스투디오 IDE, 프로젝트와 솔루션에 익숙하지 않으면, MSDN 라이브러리 내에서 비주얼 스투디오 문서를 참조하십시오.

소스 코드 파일

소스 코드 파일인 SimpleSample.cs, SimpleSampleTypes.cs 와 AssemblyInfo.cs 는 실제 서비스 구현의 시작입니다. 서비스 프로젝트가 이 점에서 일부 코드를 이미 포함하지만, 이것은 단지 완전한 서비스 구현 방법을 보여주는 템플릿입니다. AssemblyInfo.cs 파일은 평상 시 변경할 필요가 없는 프로젝트 일부 속성을 포함합니다.

SimpleSample.cs 파일은 초기화와 종료 코드, 서비스 핸들러와 이벤트 알림 핸들러로 이루어진 서비스 컴포넌트의 핵심요소들을 포함합니다. 그리고 SimpleSampleTypes.cs 파일은 서비스의 컨트랙트 식별자와 서비스 상태 정의 그리고 메인 포트에서 허용되는 메시지들을 포함합니다.

매니페스트 파일

매니페스트는 Manifest Loader Service 에 의해 읽혀지는 시작되어 될 서비스들의 집합을 기술한 XML 파일입니다. 매니페스트는 DSS Node 의 수명 동안 언제라도 로드될 수 있지만, 전형적으로 DSS Node 시작 시 실행할 서비스들을 구성하기 위해 사용됩니다. 매니페스트는 DSS Host Tool(DssHost.exe)를 사용하여 DSS Node 를 시작할 때 명령 행에서 지정될 수 있습니다. 매니페스트는 또한 시스템 서비스중의 하나인 콘트롤 패널 서비스를 이용해 로드 될 수 있고, 매니페스트 로더 서비스를 직접 접근하여 프로그램에서 로드할 수 있습니다.

서비스 프로젝트 컴파일

비주얼 스튜디오 프로젝트와 솔루션은 비주얼 스투디오 안에서 또는 직접 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 비주얼 스투디오와 .Net Framework SDK 의 한 부분인 MSBuild 를 사용해 컴파일 됩니다. 커맨드 라인에서 MSBuild 를 사용하는 방법은 MSDN 라이브러리 내에서 MSBuild 문서를 참조하십시오.

비주얼 스투디오에서 솔루션이 열려있을때, "Build Solution "(일반적으로 F6 을 누름) 으로 컴파일하고 다음 3 개의 어셈블리가 생성 됩니다:

1. 서비스 프로젝트에 포함되는 소스 코드에 의거하는 서비스 구현 어셈블리.
2. 다른 서비스에서 사용 가능한 서비스의 컨트랙트(contract)이 보여지는 DSS Proxy 어셈블리. 프록시 어셈블리는 서비스에 의해 노출되는 퍼블릭 조작과 상태 타입을 포함합니다. 서비스가 다른 파트너를 파트너링 할 때 직접 연결보다 프록시를 이용합니다.
3. DSS Transform 어셈블리는 서비스 구현 어셈블리에서 형식 정의와 프록시 어셈블리 사이의 매핑을 제공합니다. 서비스 트랜스폼 어셈블리는 마이크로소프트 Robotics Studio 런타임에 의해 자동적으로 로드되고, 단지 서비스 구현에 관련되며 서비스 프록시를 사용하는 다른 서비스에는 관여하지 않습니다.

위 3 개의 어셈블리는 마이크로소프트 Robotics Studio 런타임(서비스를 시작하고, 웹 브라우저로 상호 작용하는 방법을 위해 서비스 실행을 봅니다)에 의해 불려지는 공통 bin 폴더에 복사됩니다.

서비스 구현 어셈블리의 이름은 DSS New Service 생성 툴(DssNewService.exe)과 연결해 커맨드 라인 옵션을 사용하거나 비주얼 스투디오에서 프로젝트 설정을 변경하는 것에 의해 제어됩니다. 디폴트 어셈블리 이름 형식은 다음과 같습니다.

```
<ServiceName>.Yyyyy.Mmm
```

yyyy 는 4 개 숫자로 된 년도, mm 은 2 개 숫자로 된 월입니다. 규약에 의해, 프록시와 트랜스폼 어셈블리는 둘 다 이와 유사하게 이름 지어지며, "proxy" 혹은 "transform"이 뒤에 붙여 집니다.

서비스 프로젝트의 확장

서비스 프로젝트는 위에 열거된 소스 코드 파일들에만 제한되지 않습니다. 서비스 프로젝트의 전형적인 확장은 XSLT 파일을 이용한 서비스 상태로의 HTML 을 생성과 이미지, 음성 파일, 스크립트, 등을 포함합니다. 이 같은 추가 파일은 Mount Service 를 사용해 다른 서비스에 액세스 가능한 출력 폴더 카피될 수 있으며, 또한 Embedded Resource Service 를 사용해 임베디드 리소스로 서비스 구현 어셈블리에 포함될 수 있습니다.

여러 개의 서비스 프로젝트 관리

서비스 프로젝트를 만들 때 디폴트로 하나의 비주얼 스투디오 프로젝트 내에 하나의 서비스를 포함합니다:

솔루션	프로젝트	서비스
SimpleSample.sln	SimpleSample.csproj	SimpleSample.cs SimpleSampleTypes.cs SimpleSample.manifest.xml

그러나, 프로젝트는 여러 서비스를 솔루션은 여러 프로젝트를 포함할 수 있습니다. 다중 프로젝트. 이 특징은 서비스 수가 점점 증가할수록 서비스 종속성을 관리하고 서비스 DLL 수를 줄이는데 유용합니다.

여러 개 프로젝트를 솔루션에 추가하기

다중 서비스 프로젝트 사이에서 종속성을 명백하게 설정하기 위해 같은 솔루션에 서비스 프로젝트들을 쉽게 추가할 수 있습니다. 먼저, 두번째 서비스 프로젝트를 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 다음과 같이 생성합니다.

```
cd %MRI_INSTANCE_DIR%
DssNewService.exe /s:SimpleSampleTwo
```

새로운 서비스 프로젝트를 포함하고 있는 SimpleSampleTwo 라고 불리는 폴더가 생성되었을 것입니다. 여기서 첫번째 서비스 프로젝트가 있는 SimpleSample.sln 솔루션을 열고, 비주얼 스투디오 솔루션 탐색기(Solution Explorer)에서 "Adding Existing Project"를 선택해 솔루션에 SimpleSampleTwo 를 추가합니다:

Solution	Project	Service
SimpleSample.sln	SimpleSample.csproj	AssemblyInfo.cs SimpleSample.cs SimpleSampleTypes.cs SimpleSample.manifest.xml
	SimpleSampleTwo.csproj	AssemblyInfo.cs SimpleSampleTwo.cs SimpleSampleTwoTypes.cs SimpleSampleTwo.manifest.xml

서비스 구현은 서로에 직접 연결되기 보다 해당 서비스와 연결된 프록시 어셈블리를 오히려 사용해 연결되어 서비스 구현들이 독립적으로 배포되고 개선되도록 합니다. 그러나 프록시 어셈블리가 솔루션 탐색기에 직접 노출되지 않기에 프로젝트 종속성(Project Dependencies) 위저드 이용해 사용되는 서비스 프로젝트들 사이의 종속성을 표시해야 합니다.

프로젝트에 여러 개의 서비스 추가

위 2 개의 서비스를 된 솔루션을 컴파일 하면, SimpleSample 서비스와 SimpleSampleTwo 서비스를 위한 두개의 서비스 어셈블리가 생성됩니다. 하지만 서비스 수가 증가할 수록, 여러 서비스들을 어셈블리 하나로 만드는 것이 합리적입니다.

Solution	Project	Service
SimpleSample.sln	SimpleSample.csproj	AssemblyInfo.cs SimpleSample.cs SimpleSampleTypes.cs SimpleSample.manifest.xml SimpleSampleTwo.cs SimpleSampleTwoTypes.cs SimpleSampleTwo.manifest.xml

합쳐진 프로젝트를 컴파일 후 생성되는 양쪽 서비스 구현을 가진 서비스 어셈블리는 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 DSS Contract Information Tool(DssInfo.exe)를 사용하여 다음과 같이 확인할 수 있습니다.

```
cd %MRI_INSTANCE_DIR%\bin
DssInfo.exe /v:m SimpleSample.Y2007.M06.dll
```

서비스 프로젝트 마이그레이션

한 머신에 마이크로소프트 Robotics Studio 의 여러 버전은 나란히 설치할 수 있습니다. DSS Project Migration 툴(DssProjectMigration.exe)은 다른 버전의 마이크로소프트 Robotics Studio 에서 만들어진 서비스 프로젝트를 현재의 Robotics Studio 버전으로 변환해 줍니다. 프로젝트의 변환은 다음 2 개의 단계로 일어납니다:

- 마이크로소프트 Robotics Studio 런타임의 현재 버전을 사용하도록 프로젝트 파일 변환.
- API 가 마이크로소프트 Robotics Studio 의 버전 사이의 API 변경을 처리하는 소스 코드 변환.

서비스 프로젝트를 변환한 뒤에, 다시 프로젝트 변환 작없없이는 현재의 마이크로소프트 Robotics Studio 에서만 사용가능 합니다. 따라서 안전한 변환 방법은 현재의 마이크로소프트 Robotics Studio 의 설치 폴더 아래로 제작된 서비스 프로젝트 디렉터리를 복사하는 것입니다. 프로젝트 디렉터리를 복사한 뒤, 다음과 같이 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 변환을 수행합니다:

```
DssProjectMigration <serviceproject1> <serviceproject2> ...
```

만일 프로젝트가 Robotics Studio 설치 폴더 밖이나 다른 드라이브 혹은 폴더에 위치할 경우 프로젝트가 발견될 수 있는 폴더를 지정하십시오:

```
DssProjectMigration "d:₩dev₩GpsTest" "d:₩dev₩MyRobotTest"
```

위 명령은 소스 코드와 프로젝트가 있는 폴더를 스캔하고, 프로젝트를 변경하며 다음과 같은 출력을 보여 줄 것입니다:

```
Microsoft Robotics Studio Project Migration Utility Root Directory: C:₩Microsoft Robotics Studio Searching Directory: d:₩dev Updating GpsTest.csproj ... Updating GpsTest.csproj.user ... Fixing Contract.Namespace reference in file: d:₩dev₩GpsTest₩GpsTest.cs Fixing Contract.Namespace reference in file: d:₩dev₩GpsTest₩GpsTest.manifest.xml Fixing Contract.Namespace reference in file: d:₩dev₩GpsTest₩GpsTestTypes.cs Updating MyRobotTest.csproj ... Updating MyRobotTest.csproj.user ... Fixing Contract.Namespace reference in file: d:₩dev₩MyRobotTest₩MyRobotTest.cs Fixing Contract.Namespace reference in file: d:₩dev₩MyRobotTest₩MyRobotTest.manifest.xml Fixing Contract.Namespace reference in file: d:₩dev₩MyRobotTest₩MyRobotTestTypes.cs Done C:₩Microsoft Robotics Studio>
```

런타임 구성

애플리케이션 설정 파일

메시지 로깅을 디버그 메시지의 생성과 서비스 요청 타임 아웃 처리를 포함한 DSS 런타임 특징들은 애플리케이션 설정 파일을 통하여 설정될 수 있습니다. 애플리케이션 설정은 애플리케이션과 같은 이름의 .config 확장자를 가진 애플리케이션과 같은 폴더에 위치 하는 XML 파일입니다.

DSS Host 툴(DssHost.exe)를 사용해 서비스를 실행하는 경우 애플리케이션 설정 파일은 dsshost.exe.config 로 불립니다. 그러나 DssEnvironment 정적 클래스로 DSS 런타임을 시작하는 자체 실행가능한 애플리케이션의 설정 파일은 이름이 추가된.config 입니다.

애플리케이션 설정 파일은 애플리케이션 구동 시 읽어집니다. 이것은 애플리케이션이 다시 시작되기 전에는 애플리케이션이 실행하는 동안 애플리케이션 설정 파일 변경이 영향을 미치지 못함을 의미합니다.

메시지 캡처링과 로깅

DSS 런타임은 서비스 사이의 메시지를 직렬화하고 그리고 이를 각 메시지를 주고받는 서비스 인스턴스의 URI 로 이름지어진 로그 파일에 저장하게 설정될 수 있습니다. 메시지 캡처링과 로깅의 레벨을 통제하기 위해, 애플리케이션 설정 파일을 열어 appSettings 부분을 찾아 Microsoft.Dss.Services.Forwarders.MessageCapture 를 확인합니다:

```
<appSettings>  <add  key="Microsoft.Dss.Services.Forwarders.MessageCapture"  value=/>
</appSettings>
```

이 옵션에 사용 가능한 값들은 다음과 같습니다:

CaptureInbound

CaptureInbound 는 모든 메시지가 서비스에 의해 수신되고, XML 로 직렬화 되고 디스크에 저장되는 옵션입니다.

```
<add key="Microsoft.Dss.Services.Forwarders.MessageCapture" value="CaptureInbound"/>
```

CaptureOutbound

CaptureOutbound 는 서비스가 보내는 모든 메시지가 XML 에 직렬화하 되고 디스크에 저장되는 옵션입니다.

```
<add key="Microsoft.Dss.Services.Forwarders.MessageCapture" value="CaptureOutbound"/>
```

CaptureInboundOutbound

CaptureInboundOutbound 는 서비스가 송수신하는 모든 메시지가 XML 로 직렬화 되고 디스크에 저장되는 옵션입니다.

```
<add                              key="Microsoft.Dss.Services.Forwarders.MessageCapture"
value="CaptureInboundOutbound"/>
```

DSS 노드가 실행하는 동안 로그 파일은 storeWlogs 디렉터리에 저장되고 액세스 될 수 있습니다. DSS Directory Service 와 같이 고정 URI 를 가지는 서비스들은 항상 같은 로그 파일을 사용합니다. 각 인스턴스 마다 다른 URI 를 사용하는 서비스는 각기 다른 로그 파일을 사용합니다.

디버그와 Trace 메시지

DSS 런타임은 System.Diagnostics.Trace 클래스를 사용해 실행하는 동안 여러 가지 상태 메시지를 로그합니다. Trace 메시지는 *verbose*, *informational*, *warning*, to *error* 로 심각성 레벨에 따라 메시지를 보여주게 설정되는 각 trace 스위치의 집합으로 형성됩니다. trace 스위치에 쓰이는 메시지는 다음 두 개 중 한 방법으로 조사됩니다:

1. DSS 노드가 실행 중일 때 디버그와 trace 메시지는 웹 브라우저에서 액세스 가능한 시스템 서비스 인 Console Output 서비스로 확인할 수 있습니다.
2. 메시지는 DebugView 와 같은 여러 가지 도구로 캡쳐될 수 있습니다

trace 스위치는 다음과 같은 마이크로소프트 로보틱스 런타임으로 정의됩니다:

Microsoft.Ccr.Core

CCR 에서 생성되는 메시지를 포함합니다.

Microsoft.Dss.Core

DSS 에서 생성되는 메시지를 포함합니다.

Microsoft.Dss.Services.TestBase

Service Testing Infrastructure 에서 생성되는 메시지를 포함합니다.

Microsoft.Dss.Services

디렉터리와 컨트랙트 디렉터리 같은 시스템 서비스에 의해 생성되는 메시지를 포함합니다.

Microsoft.Dss.Services.Transports

DSS 런타임의 네트워크 전송 계층에 의해 생성되는 메시지를 포함합니다

Microsoft.Dss.Services.Forwarders

DSS 서비스와 트랜스포트 사이에 있는 메시지 포워더에 의해 생성되는 메시지를 포함합니다

trace 스위치를 설정하고 애플리케이션 설정 파일을 열어 system.diagnostics 라고 불리는 큰 부분의 일부분인 switches 를 찾습니다:

```
<switches>
<add name="Microsoft.Ccr.Core" value="2" />
<add name="Microsoft.Dss.Core" value="3" />
<add name="Microsoft.Dss.Services.TestBase" value="3" />
<add name="Microsoft.Dss.Services" value="3" />
<add name="Microsoft.Dss.Services.Transports" value="2" />
```

```
<add name="Microsoft.Dss.Services.Forwarders" value="2" />
</switches>
```

각 스위치 값은 다음 중 하나입니다:

Value	Description
0	trace 스위치를 끕니다.
1	trace 레벨 오류 메시지만 표시합니다.
2	trace level 오류와 경고 메시지를 표시합니다.
3	trace level 오류, 경고, 정보를 표시합니다.
4	trace level 오류, 경고, 정보와 *verbose* 를 표시합니다.

서비스 저자는 System.Diagnostics.TraceSwitch .Net Framework 클래스를 사용해 자신의 trace 스위치를 정의할 수 있습니다. 마이크로소프트 Robotics Studio 에 의해 제공되는 trace 스위치 설정과 비슷한 방법으로 애플리케이션 설정 파일에 이를 추가하여 위에서 설명된 방법으로 사용할 수 있습니다.

서비스 요청 타임아웃 트래킹

DSS 런타임은 로컬에 있는지 원격에 있는지 상관없이 다른 서비스로 보내는 모든 outbound 요청에 대해 타임아웃을 추적하도록 설정될 수 있습니다. 이것은 서비스가 다른 서비스의 요구에 응답하지 않거나, deadlock 을 일으킬 때를 감지합니다. 애플리케이션을 위한 서비스 타임아웃을 제어하기 위해 애플리케이션 설정 파일을 열고 appSettings 부분을 찾아 Microsoft.Dss.Services.Forwarders.DsspTimeoutTracking 부분을 확인합니다:

```
<appSettings> <!-- DsspTimeoutTracking enables/disables expiration on all DsspOperations,
--> <add key="Microsoft.Dss.Services.Forwarders.DsspTimeoutTracking" value="true"/>
</appSettings>
```

값을 "true"로 설정하는 것으로 정적인 필드 DsspOperation.DefaultShortTimeout(30 초)에서 지정된 간격 내에서 완료하지 못하는 모든 outbound 요청들이 Soap.Fault 응답으로 완료될 것입니다. Fault.Code.Subcode 필드는 Timeout 값을 포함하며 요청자가 다른 실패와 타임아웃 실패를 구분하게 해줍니다.

또한 여러분의 outbound 요청에 대한 명백한 타임 아웃을 DsspOperatiom.TimeSpan 필드를 outboud 요청을 포스트하기 전에 설정함으로 지정할 수 있습니다:

```
    using cons = Microsoft.Dss.Services.Constructor;
    // Create a service using its contract identifier      cons.Create create = new
cons.Create(new ServiceInfoType(Contract.Identifier));
   // Set timtout value to 40 seconds     create.TimeSpan = TimeSpan.FromSeconds(40); //
Issue request and check for timeout     yield return Arbiter.Choice(
        create.ResponsePort,
        delegate(CreateResponse createResponse) { // Successful service creation },
        delegate(Fault fault)
        {
   if (d.Code.Subcode.Value == dssp.DsspFaultCodes.ResponseTimeout) {
               LogError("Timeout received");            }        } );
```

서비스 시작과 실행

서비스 실행

서비스는 DSS 노드에서 실행됩니다. DSS 노드는 서비스가 삭제되거나 DSS 노드가 멈추게 될 때까지 서비스를 생성하고 관리하는 호스팅 환경입니다. DSS 노드는 DSS Host 툴(DssHost.exe)을 사용하는 독립형 애플리케이션으로 시작하거나 DSS Environment 정적 클래스를 사용하는 다른 애플리케이션의 일부로서 시작할 수 있습니다.

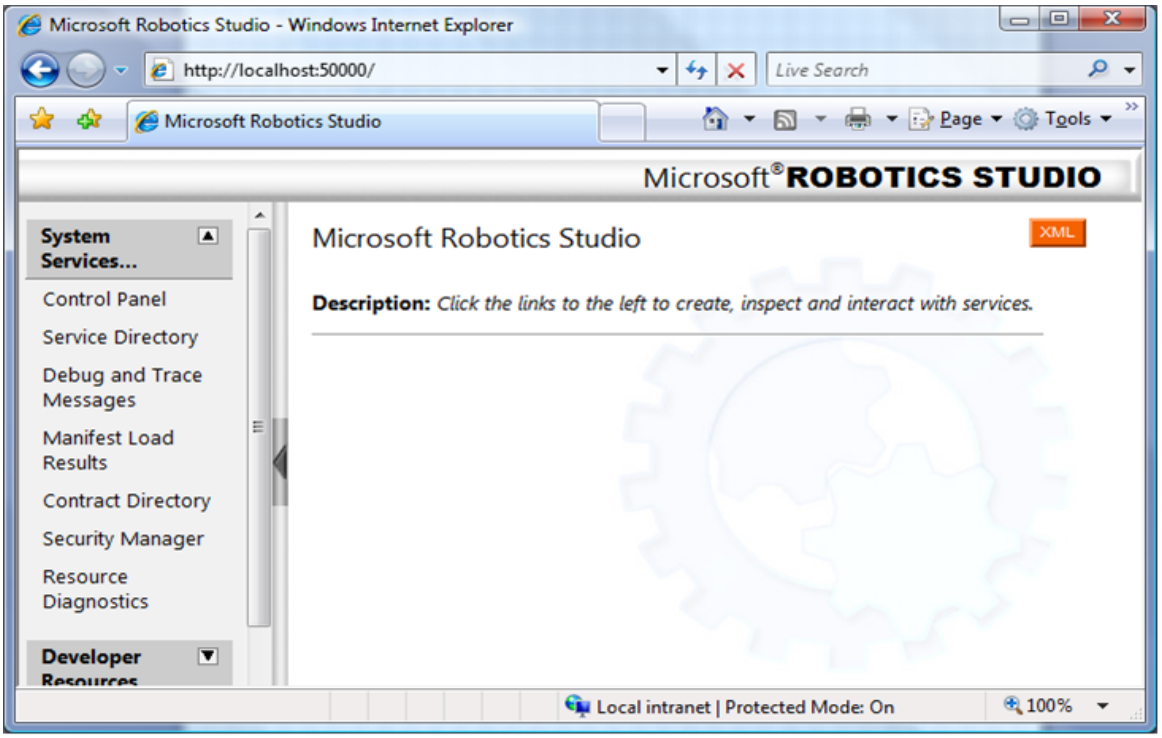
독립적으로 실행되는 DSS 노드를 DSS Host Tool(DssHost.exe)를 이용해 시작하려면 시작 메뉴에서 마이크로소프트 Robotics Studio 커맨드 프롬프트를 열어 다음을 실행합니다.

```
Dsshost /p: 50000
```

DSS 노드가 시작되는 방법에 상관없이, 웹 브라우저를 통해 서비스를 조사하고, 서비스와 상호작용할 수 있습니다. 위에 예에서 사용되는 URI 는 다음과 같습니다.

```
http://localhost:50000/
```

브라우저 위 URI 를 입력하면 다음 페이지를 볼 수 있습니다.



이제 컨트롤 패널 서비스에서 각 서비스들을 제어합니다. 어떤 서비스들이 이용가능 한 지 알아보십시오.

서비스 매니페스트

매니페스트는 매니페스트 로더 서비스(Manifest Loader Service)에 의해 읽어질 수 있는 실행 될 서비스들을 나타내는 XML 파일입니다. 매니페스트가 DSS 노드 실행 중 언제라도 로드될 수 있지만 일반적으로 DSS 노드 시작 시 서비스들을 실행하기 위해 사용됩니다. 매니페스트는 커맨드 라인에서 DSS Host 툴(DssHost.exe)를 사용해 DSS Node 실행하거나, 정적인 DssEnvironment 클래스의 일부로서 시작할 때 같이 입력됩니다. 또한 그들은 또한 시스템 서비스 중의 하나인 컨트롤 패널 서비스나 매니페스트 로더 서비스를 직접 액세스하는 프로그래밍을 동해 로드 될 수 있습니다.

DSS 매니페스트 는 어떤 XML 에디터나 혹은 매니페스트를 만들고, 변경하기 위한 비주얼 에디터인 DSS Manifest Editor(참조 DSS 매니페스트 편집기 소개)를 사용해 편집가능 합니다.

DSS Host 툴(DssHost.exe)과 정적인 DssEnvionment 클래스 둘다 동시에 여러 개의 매니페스트들을 정의할 수 있습니다. 이는 매니페스트를 하나로 통합하지 않고도 각 매니페스트로 지정된 여러 서비스들을 같이 시작하는 것을 쉽게 합니다. 이 기능은 로보틱스 튜토리얼에서 특정 로보틱스 플랫폼을 지정하는데 사용됩니다.

매니페스트는 생성되고 파트너 되어져야 할 각 서비스들을 지정하는 레코드의 집합입니다. 매니페스트 로더 서비스(Manifest Loader Service)는 각 매니페스트를 서비스 인스턴스를 생성하는 생성자 서비스(Constructor Service)에 보닙니다(참고 시스템 서비스). 처음 서비스를 매니페스트로 생성한 후 부터는 생성자 서비스가 직접 해당 서비스와 상호작용하므로, 새로운 서비스 인스턴스를 생성에 매니페스트는 필요하지 않습니다.

매니페스트의 주 목적은 다음 두 개입니다:

1. 기존 코드를 변경하지 않고 한 세트의 서비스들을 선언함
2. 해당 서비스 시작 시 필요한 디폴트 파트너 서비스들을 덮어 씀.

매니페스트는 서비스들의 복합한 오케스트레이션을 위해서는 추천되지 않습니다. 많은 다른 서비스들을 오케스트레이션하는 애플리케이션은 생성자 서비스를 직접 사용하거나 자동으로 서비스 인스턴스를 생성하는 파트너 속성을 사용해 서비스들을 생성해야 합니다.

매니페스트 편집과 생성

편의 을 위해 모든 DSS 서비스 프로젝트의 한 부분으로 단순한 매니페스트가 만들어집니다. 그러나, 서비스들으 항상 매니페스트에 리스트 될 필요는 없으며, 서비스들이 한개의 매니페스트에 리스트어야 하는 것도 아닙니다. 매니페스트는 비주얼 스투디오 에디터등의 어떤 XML 에디터로도 편집될 수 있습니다. 다음의 메니페스트의 예 입니다.

```
<Manifest xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
xmlns:s="http://schemas.tempuri.org/2007/04/mysample.html"> <CreateServiceList>
<ServiceRecordType>
<dssp:Contract>http://schemas.tempuri.org/2007/04/mysample.html</dssp:Contract>
</ServiceRecordType> </CreateServiceList> </Manifest>
```

이 매니페스트는 만드는 서비스의 컨트랙트를 포함하고 있는 하나의 ServiceRecordType 요소를 포함합니다. 매니페스트는 어떤 개수의 ServiceRecordType 요소를 포함할 수 있으며, 각각은 컨트랙트에 나타난 서비스의 인스턴스를 생성합니다. 다음은 다른 시나리오들을 위해 ServiceRecordType 요소를 어떻게 지정하는 지를 보여줍니다.

서비스 인스턴스 이름 변경하기

DSS 서비스는 서비스를 정의의 일 부분인 ServicePort 속성을 사용하여 지정된 디폴트 이름을 가집니다(Service Tutorial 1(C#)를 봅니다-서비스 생성). 그러나 매니페스트를 통하여 이 이름은 다음과 같이 서비스 요소를 사용해 다른 이름으로 덮어 쓸 수 있습니다:

```
<ServiceRecordType>
<dssp:Contract>http://schemas.tempuri.org/2007/04/mysample.html</dssp:Contract>
<dssp:Service>http://localhost/MyNewSampleName</dssp:Service> </ServiceRecordType>
```

초기 상태 파트너 설정하기

초기 상태 파트너를 이용하는 서비스를 위해(참조 서비스 튜토리얼 3(C#)-상태 유지하기) 초기 상태로 사용할 자원 또는 파일을 나타내는 매니페스트를 사용할 수 있습니다. 초기 상태는 스테이트서비스(StateService) 이름을 가진 파트너와 추가하고 자원 혹은 파일을 나타내는 URI 를 값으로 가지는 서비스 요소를 추가하여 구성합니다.

```
<ServiceRecordType>
<dssp:Contract>http://schemas.tempuri.org/2007/04/mysample.html</dssp:Contract>
<dssp:PartnerList> <dssp:Partner> <dssp:Service>mystate.xml</dssp:Service>
<dssp:Name>dssp:StateService</dssp:Name> </dssp:Partner> </dssp:PartnerList>
</ServiceRecordType>
```

서비스 이름이 URI 이고, 매니페스트 그 자체에 대한 해결된 관계입니다. 위에 예에 이것은 초기상태를 저장한 mystate.xml 가 매니페스트 그 자체와 같은 디렉터리에 존재함을 의미합니다.

다른 파트너들 설정하기

다른 파트너들은 초기 상태 파트너에 사용된 것과 파트너 이름만 다른 같은 모델을 사용해 설정됩니다. 예를 들어 컨트랙트가 http://schemas.tempuri.org/2007/04/mysample.html 인 서비스가 MyPartner 이름의 파트너를 가진다면 매니페스트는 다음과 같습니다.

```
<ServiceRecordType>
<dssp:Contract>http://schemas.tempuri.org/2007/04/mysample.html</dssp:Contract>
<dssp:PartnerList>                                                    <dssp:Partner>
<dssp:Service>http://localhost/myPartnerService</dssp:Service>
<dssp:Name>s:MyPartner</dssp:Name>          </dssp:Partner>          </dssp:PartnerList>
</ServiceRecordType>
```

이 경우 myPartnerService 서비스는 이미 실행되고 있거나 매니페스트에서 그것을 별도의 serviceRecordType 요소르 열거해 생성할 수 있어야 할 것입니다.

교차 노드 매니페스트

매니페스트는 같은 DSS 노드에서 실행하는 서비스와 파트너 참조에만 제한되지 않습니다. 시큐리티(Security)가 허용되는 경우 서비스는 어떤 노드에서라도 생성되고 파트너로 연결될 수 있습니다. 다음 예는 이미 다른 노드에서 실행되고 있는 서비스와 파트너 관계를 맺을 수 있는 방법을 설명합니다. 파트너 요소의 일부로서 서비스디렉토리(ServiceDirectory) 요소를 사용하는 것으로 원격 서비스의 특정 서비스 인스턴스 URI 는 원격 노드에서 실행되는 디렉터리 서비스(Directory Service)에서 검색될 수 있습니다.

```
<ServiceRecordType>
<dssp:Contract>http://schemas.tempuri.org/2007/04/mysample.html</dssp:Contract>
<dssp:PartnerList>          <dssp:Partner>          <dssp:Name>s:MyPartner</dssp:Name>
<dssp:ServiceDirectory>http://example.org:50000/directory</dssp:ServiceDirectory>
</dssp:Partner> </dssp:PartnerList> </ServiceRecordType>
```

서비스 튜토리얼 7(C#)-고급 토픽들에서 DSS 노드들 사이의 서비스 생성과 파트너 관계 생성에 대한 방법을 자세히 설명합니다.

매니페스트 로드 결과

매니페스트는 어떤 구문 오류라도 보고하는 매니페스트 로더 서비스(Manifest Loader Service)에 의해 로드 되기 전에 XML 스키마(Schema) 정의에 따라 오류가 검사 됩니다. 매니페스트 로더 서비스(Manifest Loader Service)가 매니페스트의 로드를 완료했을 때 그 응답으로 결과를 보고합니다. 그러나, 매니페스트 로드 결과는 또한 실행되고 있는 DSS 노드를 웹 브라우저로 열어 조사할 수 있고, 통상 다음과 같이 매니페스트 로더 서비스를 발견할 수 있습니다:

```
http://localhost:50000/manifestloaderclient
```

시스템 서비스

여러분이 서비스 개발을 위해 어떤 프로그램을 사용하던지 DSS 노드의 한 부분인 시스템 서비스들을 이용할 수 있습니다. 시스템 서비스는 파일 시스템에 액세스하는 디렉터리 서비스로부터 로깅, 진단, 씨큐리티와 새로운 서비스 인스턴스 생성까지 공통된 기능들을 제공합니다. 시스템 서비스는 DSS 노드에서 실행되는 어떤 서비스에라도 의해 사용될 수 있습니다. 이것은 DSS 가 본래 분산되어 있으며, 여러분의 시큐리티 설정에 딸 이들 서비스가 원격으로 액세스되기 때문입니다.

마이크로소프트 Robotics Studio 커맨드 프롬프트에서 다음을 입력하거나

```
dsshost /p:50000
```

혹은 시작메뉴에서 "Run DSS Node" 를 선택하여 하여 DSS Node 를 실행한 후 시스템 서비스 중의 대다수를 다음 URI 를 웹 브라우저에 입력하여 확인할 수 있습니다.

```
http://localhost:50000
```

컨트롤 패널 서비스(Control Panel Service)

컨트롤 패널은 수동으로 서비스를 시작하고 정지하고 사용 가능한 서비스 컨트랙트를 조사하기에 유용한 서비스입니다.

생성자 서비스(Constructor Service)

생성자 서비스는 서비스의 새로운 인스턴스를 만듭니다. 서비스가 생정자 서비스를 직접 사용하는 경우에도 매니페스트 또는 파트너와 같은 상급의 요소를 종종 사용합니다.

매니페스트 로더 서비스(Manifest Loader Service)

매니페스트 로더 서비스는 매니페스트를 로드하고 생성자 서비스에 서비스 인스턴스를 만들기 위한 요청을 보냅니다.

서비스 디렉토리 서비스(Service Directory Service)

서비스 디렉터리 서비스는 DSS 노드에서 실행되는 서비스 인스턴스들의 단순한 디렉터리입니다. 디폴트 동작으로 모든 서비스 인스턴스가 디렉토리에 등록되게 됩니다. 서비스 디렉터리의 일부로서, 각 서비스 인스턴스가 연관된 파트너들을 보는 것이 가능합니다.

컨트랙트 디렉터리 서비스(Contract Directory Service)

DSS 노드에서 이용할 수 있는 서비스 어셈블리를 찾기 위해 컨트랙트 디렉터리는 bin 폴더(혹으 다른 폴더)를 검색합니다. 서비스 어셈블리가 발견되는 서비스만이 DSS 노드에서 만들어질 수 있습니다.

마운트 서비스(Mount Service)

마운트 서비스는 서비스가 DSS 노드를 통하여 파일들을 보여주고 이 파일들을 저장하고 불러들이게 합니다. 그리고 마이크로소프트 Robotics Studio 의 설치 폴더 아래 있는 것만 파일만이 이처럼 노출됩니다.

임베디드 리소스 서비스(Embedded Resource Manager Service)

임베디드 리소스 서비스는 서비스가 DSS 노드를 통하여 임베디드 리소스를 보여주게 합니다. 디폴트로 임베디드 리소스는 DSS 노드를 통해 사용되지 못하지만 임베디드 리소스 서비스를 이용해 명백하게 등록한 경우 읽기 전용으로 다른 서비스에서 액세스할 수 있습니다.

서브스크립션 매니저 서비스(Subscription Manager Service)

서브스크립션 매니저 서비스는 각 서비스가 서브스크립션을 자체적으로 가질 필요없이 이를 관리할 수 있는 헬퍼 서비스입니다. 서브스크립션 매니저는 각 서비스에서 파트너로 설정하여 사용합니다(서브스크립션 매니저 사용과 관리를 위해 서비스 튜토리얼을 참고하십시오).

파트너 매니저 서비스(Partner Manager Service)

파트너 매니저 서비스는 서비스 인스턴스와 연결된 활성화된(active) 파트너를 관리하는 또 다른 헬퍼 서비스입니다.

콘솔 출력 서비스(Console Output Service)

콘솔 출력 서비스는 DSS 노드가 실행될 때 디버그와 로그 메시지를 보여줍니다. 콘솔 서비스는 효과적으로 구성된 콘솔로서 임의의 복잡도를 가진 메시지를 보낼 수 있고 관심있는 메시지를 정확히 여과하도록 할 수 있습니다.

리소스 진단 서비스(Resource Diagnostics Service)

리소스 진단 서비스는 시스템에서 처리 혹은 대기된 메시지의 양에 관한 현재의 정보를 제공합니다.

시큐리티 매니저 서비스(Security Manager Service)

보안 관리자 서비스는 노드를 위한 보안 정책 설정을 가능하게 합니다.

서비스와의 상호작용

웹브라우저를 통한 서비스 접근

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Accessing Services through a Web Browser

Well-Known 서비스 URI

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Well-known Service URIs

DSSP를 통한 서비스 접근

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Accessing Services through DSSP

서비스 기반 사용자 인터페이스

서비스 사용자 인터페이스를 위한 방안

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Service User Interface Alternatives

기본 DSS XSLT 템플릿을 사용하는 튜토리얼

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Tutorial for Using Default DSS XSLT Template

서비스 시험하기

서비스 시험 기반구조

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Service Testing Infrastructure

시험용 서비스 생성

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Creating Test Services

테스트 스크립트

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Test Scripts

서비스 배포

서비스 배포 방안

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Service Deployment Alternatives

서비스 배포하기

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Deploying Services

Windows CE용 서비스 개발

로보틱스 Windows CE 장치에서 작동하는 서비스와 애플리케이션 개발을 지원합니다. 이 문서는 개발자가 Robotics Studio 를 이용해 Windows CE 용 프로그래밍 중 만나는 가장 일반적인 시나리오를 보여줍니다. 이 시나리오들은 PC 프로그래밍보다 많은 과정을 필요로 하지만, 실재 애플리케이션이 디바이스에서 동작하게 해줍니다.

Robotics Studio 를 이용해 .NET Compact Framework 2.0 으로 호스트되는 분산 서비스를 개발할 수 있습니다. 이것은 Robotics Studio 로 제작된 애플리케이션을 Windows CE 임베디드 디바이스, Windows Mobile Pocket PC 와 SmartPhone 등 다양한 임베디드 플랫폼에 배포할 수 있게 합니다..

요구사항

.Net Compact Framework 애플리케이션을 개발하고 CE 디바이스에 배포하기 위해 다음 사항이 필요합니다:

- Windows CE 5.0 이상이 탑재된 디바이스. Windows CE 는 Windows Mobile 디바이스에 통합되어 있습니다. 애플리케이션을 테스트하기 위해 Windows CE emulator 와 VS2005 를 항상 이용할 수 있습니다.
- .Net Compact Framework 2.0 SP2 Redistributable 을 다운로드하고 PC 와 Windows CE 5.0 이상의 디바이스에 설치합니다.
- Visual Studio 2005 Standard 버전 이상. Visual Studio Express 버전은 모바일과 임베디드 개발을 지원하지 않습니다.
- 목표하는 CE platform 이 ICOP eBox 임베디드 시스템과 같은 싱글보드컴퓨라면 최신 보드 지원 패키지(BSP)를 제조사의 웹사이트에서 받아 설치해야 합니다.

이 가이드에서는 임베디드 디바이스에 운영체제가 설치되어 있다고 가정합니다. OS 이미지를 준비 설치하는 방법은 해당 디바이스의 제조사에 문의하십시오.

CF 용 DSS 서비스 생성

DSS 서비스는 먼저 .Net Framework 2.0 을 이용해 Windows XP 에서 서비스를 개발한 뒤 이를 .Net Compact Framework 2.0 에서 동작하는 Windows CE 용 서비스로 바꾸는 것이 CF 와 데스크탑 서비스 통신에 사용될 proxy 와 transform 어셈블리를 생성하기 위해 필요합니다.

예로 다음과 같은 데스크탐용 서비스를 DSSNewService 로 생성합니다:

```
>cd samples
>DssNewService.exe /service:simpleExample
```

이제 iRobot Create 을 사용하도록 다음과 같이 수정합니다:

- 생성된 "simpleExample.csproj" 을 Visual Studio 에서 엽니다.
- Robotics Studio 의 bin 디렉터리에 있는 "IRobot.Y2007.M01.proxy"를 참조로 추가합니다.
- simpleExample.cs 를 열어 다음과 같이 변경합니다:
  - 다음 "using" 명령을 맨 윗부분에 추가합니다:

```
using lite = Microsoft.Robotics.Services.IRobot.Lite.Proxy;
using sensorupdates = Microsoft.Robotics.Services.IRobot.SensorUpdates.Proxy;
using irobot = Microsoft.Robotics.Services.IRobot.Roomba.Proxy;
using create = Microsoft.Robotics.Services.IRobot.Create.Proxy;
```

  - iRobot 파트너 서비스와 몇 개의 전역변수를 SimpleExample 클래스 선언에 추가합니다:

```
 [Partner("irobotlite", Contract = lite.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry,
Optional = false)]
lite.IRobotLiteOperations _irobotPort = new lite.IRobotLiteOperations();
lite.IRobotLiteOperations _irobotNotify = new lite.IRobotLiteOperations();

private int _initialized = 0;
```

  - iRobot 서비스에대한 서브스크립션을 "Start()" 메서드에 추가합니다:

```
_irobotPort.Subscribe(_irobotNotify);
Activate<ITask>(
  Arbiter.Receive<sensorupdates.UpdateMode>(true, _irobotNotify, UpdateModeHandler),
  Arbiter.Receive<sensorupdates.UpdatePose>(true, _irobotNotify, UpdatePoseHandler),
  Arbiter.Receive<sensorupdates.UpdateBumpsCliffsAndWalls>(true, _irobotNotify, UpdateBumpsHandler)
```

```
);
```

○ notification 메서드를 위와 같이 추가합니다:

```
private void UpdateModeHandler(sensorupdates.UpdateMode mode)
{
  if (mode.Body.RoombaMode == irobot.RoombaMode.Full)
  {
        if (_initialized == 0)
        {
        _initialized = 1;
        _irobotPort.CreateDriveDirect(200, 200);
        }
  }
  LogInfo(LogGroups.Console,          "Mode:          "          +
mode.Body.RoombaMode.ToString());
}
private void UpdatePoseHandler(sensorupdates.UpdatePose pose)
{
  LogInfo(LogGroups.Console,          "Distance:          "          +
pose.Body.Distance.ToString()      +      "      Angle:      "      +
pose.Body.Angle.ToString());
  if (_initialized == 1 && pose.Body.Distance > 200)
  {
        _initialized = 2;
        _irobotPort.CreateDriveDirect(-100, 100);
        _irobotPort.RoombaPlaySong(2);
  }

  if (_initialized == 2 && (pose.Body.Angle > 90 || pose.Body.Angle
< -90))
  {
        _initialized = 3;
        _irobotPort.CreateDriveDirect(0, 0);
```

```
        _irobotPort.RoombaPlaySong(1);
  }
}

private                                                    void
UpdateBumpsHandler(sensorupdates.UpdateBumpsCliffsAndWalls update)
{
  if                ((update.Body.BumpsWheeldrops                &
(irobot.BumpsWheeldrops.BumpLeft                               |
irobot.BumpsWheeldrops.BumpRight)) != 0)
  {
        LogInfo(LogGroups.Console,
update.Body.BumpsWheeldrops.ToString());
  }
}
```

- "simpleExample.manifest.xml"를 열어 알맞은 시작정보를 매니페스트에 추가합니다:

```
<?xml version="1.0" ?>
<Manifest
        xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
        xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"

xmlns:example="http://schemas.tempuri.org/2007/04/simpleexample.html"
        >
  <CreateServiceList>
        <ServiceRecordType>

<dssp:Contract>http://schemas.tempuri.org/2007/04/simpleexample.html</dssp:C
ontract>
          <dssp:PartnerList>
                <dssp:Partner>
                  <dssp:Name>example:irobotlite</dssp:Name>
                </dssp:Partner>
```

```
          </dssp:PartnerList>
        </ServiceRecordType>
        <ServiceRecordType>

<dssp:Contract>http://schemas.microsoft.com/robotics/2007/02/irobotlite.html
</dssp:Contract>
          <dssp:Service>http://localhost:0/irobotlite</dssp:Service>
          <dssp:PartnerList>
                <!--Initialize Roomba config file -->
                <dssp:Partner>
                  <dssp:Service>irobot.ce.config.xml</dssp:Service>
                  <dssp:Name>dssp:StateService</dssp:Name>
                </dssp:Partner>
          </dssp:PartnerList>
          <Name>example:irobotlite</Name>
        </ServiceRecordType>
  </CreateServiceList>
</Manifest>
```

- "irobot.ce.config.xml"라는 새 xml file 을 프로젝트에 추가하고 이 파일에 다음 설정 정보를 추가합니다:

```
<?xml version="1.0" encoding="utf-8"?>
<RoombaState xmlns:s="http://www.w3.org/2003/05/soap-envelope"

xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"

xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.html"

xmlns="http://schemas.microsoft.com/robotics/2007/01/irobot.html">
  <BaudRate>57600</BaudRate>
  <Name>Dave</Name>
  <SerialPort>1</SerialPort>
  <IRobotModel>Create</IRobotModel>
```

```
  <ConnectionType>CreateSerialPort</ConnectionType>
  <MaintainMode>Full</MaintainMode>
  <PollingInterval>201</PollingInterval>
  <WaitForConnect>false</WaitForConnect>
  <CreateNotifications />
</RoombaState>
```

Robotics Studio 커맨드 프롬프트에서 다음 명령을 입력해 이 서비스를 컴파일할 수 있습니다:

```
>msbuild simpleExample.csproj
```

제작된 데스크탑용 DSS 서비스 C# 프로젝트를 DssNewService 와 "/generateCFproject" 인수를 이용해 다음과 같이 CF 프로젝트로 변환합니다:

```
>DssNewService.exe /generateCFproject:simpleExample.csproj
```

결과로 제작된 같은 코드를 가지고 CF 에서 컴파일 되는 프로젝트로 변환됩니다. 생성된 CF 프로젝트는 Windows CE 5.0 플랫폼에서 동작합니다. 이 새 프로젝트를 다음과 같이 컴파일 합니다.

```
>msbuild "cf.simpleExample.csproj"
```

다음과 같이 define 명령문을 이용해 x86 과 CE 코드 부분을 구분할 수 있습니다:

```
#if !URT_MINCLR
// Desktop specific code
#endif
```

URT_MINCLR 전처리 심볼은 DssNewService 에서 생성한 CF 프로젝트에만 정의되어 있습니다.

CE 에 배포를 위한 DssDeploy 패키지 생성

CF DSS 서비스 배포는 데스크탑에서와 같이 DssDeploy 를 사용합니다.

DssDeploy 는 자동실행 압축파일(self extracting executable)을 생성합니다. 패키지를 배포하기 위해 생성된 파일을 CE 디바이스에 복사하고 이를 실행합니다. 실행 후 파일이 복사된 디렉터리에 압축된 파일이 풀리게 됩니다.

배포 패키지를 생성하기 위해 해당 컨트랙트를 위한 매니페스트 파일을 지정합니다. 또한 자동으로 찾아지지 않는 문서, native 참조, 미디어 파일등을지정하여 추가할 수 있습니다. 모든 서비스가 CF 를 위해 사용될 수 있는지를 확인해야 합니다. 이는 그 어셈블리가 Robotics Studio 의 binWCF 에 있는지로 확인할 수 있습니다.예로 위의 Simple Example 서비스를 패키지로 만들어 보겠습니다.

"samplesWsimpleExample"에 있는 "SimpleExample.manifest.xml" 매니페스트를 사용해 다음과 같이 패키지를 만듭니다:

```
>DssDeploy.exe /p /cf /m:"SimpleExample.manifest.xml" mySimpleIRobot.exe
```

이것은 mySimpleIRobot.exe 라는 CE 디바이스용 자동실행 압축파일을 생성합니다.

CE 디바이스에 패키지 배포

DssDeploy 로 만들어진 패키지는 .Net Compact Framework 2.0 SP2 제외한 디바이스에서 실행에 필요한 모든 것을 가집니다.

주목: 패키지 배포 전에 .Net Compact Framework 2.0 SP2 (CF) 이 디바이스에 설치되어 있어야 합니다. 디바이스에 서비스를 배포하기 위해 몇 가지 방법이 있습니다. 그 중 한 예는 ActiveSync 를 사용해 설치하는 것입니다. 디바이스가 ActiveSync 를 지원하지 않으면, 적절한 버전의 "cab" 파일을 찾아야 합니다. 예로 "C:WProgram FilesWMicrosoft.NetWSDKWCompactFrameworkWv2.0WWindowsCEWwce500Wx86WNETCFv2.wce5.x86.cab" (Windows CE 5.0 용)을 디바이스에 복사하고 실행해 설치합니다. 각 CPU 마다 지정된 CF 설치 파일이 있습니다. 디바이스 매뉴얼을 읽고 프로세서가 XSCALE, ARM, x86, SH, MIPS 등 어떤 것인지를 확인하고 올바른 CF 2.0 cab 파일을 찾아 디바이스에 설치하십시오. 파일을 설치하기 위해서 How to: Install the .NET Compact Framework 를 참조하십시오.

CF 가 설치되어 있으면 제작한 패키지를 ActiveSync 나 기타 여러분의 디바이스로 파일을 보내는 방법을 이용해 복사합니다. 배포 패키지는 DSS 와 CCR 런타임과 같이 서비스 실행에 필요한 파일들을 가지고 있으므로 어느 디렉터리나 설치해 실행할 수 있습니다. 하지만 일관성을 위해 이 문서에서는 "WProgram FilesWMSRS" 를 디바이스의 배포 디렉터리로 사용합니다.

생성된 패키지는 풀 때 어떤 옵션도 필요하지 않습니다. "binWCF" 디렉터리의 파일들이 이제 "bin" 디렉터리에 위치됨을 주목하십시오. 그 외의 파일들은 Robotics Studio 설치에서의 트리 위치와 동일 할 것입니다.

CE 디바이스에서 DSS 서비스 실행

CE 디바이스에서 DSS 서비스를 실행하는 것은 데스크탑에서와 매우 비슷합니다. DssHost 에 HTTP 와 TCP/IP 포트와 매니페스트 등의 인수를 입력해 실행해야 합니다. 아래는 그 한 예입니다:

```
>cd WProgram FilesWMSRS
>binWcf.dsshost /p:50000 /t:50001 /m:..WsamplesWsimpleExampleWsimpleExample.manifest.xml
```

위 명령에서 데스크탑과는 달리 매니페스트가 현재 디렉터리에서가 아니라 cf.dsshost.exe 가 있는 디렉터리에서 상대적으로 지정됨에 주의하십시오. 또한 CE 용 Dsshost 에서는 매니페스트 경로에 인용부호가 없음에 주의합니다.

여러분의 CE 디바이스가 DssHost 를 실행할 커맨드 라인이 없을 때에는 Visual Studio (VS)를 이용해 원격으로 디바이스의 서비스를 실행할 수 있습니다. 이를 위해 다음 부분을 읽어보세요. 디버그 상태로 실행하지만 break point 를 지정해서는 안됩니다.

서비스가 iRobot Create 과 연동하기 위해 디바이스와 로봇을 Bluetooth 나 유선 serial 로 연결하고 "irobot.ce.config.xml"에서 **SerialPort** 와 d **ConnectionType** 을 설정해야 합니다.

데스크탑용 DSS 노드에서 실행되는 서비스와 CF 상에서 실행되는 CE 디바이스용 서비스 통신시에는 데스크탑 노드의 security 를 반드시 disabled 로 설정해야 합니다. 더 자세한 사항은 보안(security) 부분을 보십시오.

Visual Studio 를 이용한 CE 디바이스 DSS 서비스 디버깅

CE 디바이스의 서비스 디버깅을 위해 다음 단계가 필요합니다. 예에서는 CF 용 iRobot 서비스를 이용합니다:

- Visual Studio 에서 CF 프로젝트인 cf.simpleExample.csproj 를 엽니다.
- 솔루션 탐색기에서 프로퍼티를 선택해 오른쪽 마우스를 클릭해 프로젝트 프로퍼티를 엽니다.
- "디버그" 탭으로 이동합니다.
- "Start External Program" 옵션에 다음을 입력합니다:

  ```
  WProgram FilesWMSRSWbinWcf.dsshost.exe
  ```

  여기서 배포된 패키지가 디바이스의 "WProgram FilesWMSRS" 폴더에 있다고 가정합니다.
- "Command Line Arguments"에 다음을 입력합니다:

  ```
  /p:50000 /t:50001
  /m:file:///Program%20Files/MSRS/samples/simpleexample/simpleexample.manifest
  .xml
  ```

- 솔루션 탐색기 창에서 맨 위의 솔루션 노드를 오른쪽 마우스 마우스를 클릭해 프로퍼티를 선택합니다

- "Configuration PropertiesWConfiguration"에서 "cf.simpleExample.csproj" 프로젝트 옆의 deploy and build를 **uncheck** 합니다.
- Simple Example 서비스의 시작 루틴에 breakpoint 를 추가합니다.
- F5를 눌러 디바이스에서 서비스를 시작합니다.
- CE 디바이스에서 새 윈도우가 나타나고 노드가 시작됨을 볼 수 있습니다.

  주목: 배포 후 처음 시작 시 서비스 캐쉬가 생성되므로 ICOP eBox2300 임베디드 시스템에서 약 30초간 기다려야 합니다.

- 서비스가 시작되고 breakpoint에서 멈춥니다.

보안(Security)

이 부분은 DSS 노드의 보완을 위한 메커니즘을 설명합니다. DSS 노드 보완은 다음 세 부분으로 구성됩니다:

- 인증(Authentication)
- 서비스 어셈블리 로딩(Service Assembly Loading)
- 네트웍 액세스 퍼미션(Network Access Permissions)

설정

보안은 노드가 시작될 때 XML Security Settings 파일에서 설정됩니다.

인증

노드의 인증이 사용 가능하게 되어 있으면 모든 입력 transport 연결은 NTLM 인증을 필요로 합니다. 인증된 사용자명은 노드에 액세스하는 것을 허락 받은 사용자와 그룹의 리스트와 비교됩니다. 그리고 액세스가 리스트에 근거해 허가되거나 거절됩니다.

서비스 어셈블리 로딩

서비스 어셈블리 로딩은 단지 서명이 로컬 사용자 또는 머신의 인증 리스트에서 인증서가 연결된 Authenticode 로 서명된 어셈블리만 허용됩니다.

네트워크 액세스 퍼미션

서비스에 이용되는 .NET Code Access Security(CAS) 퍼미션을 제한하기 위해 노드는 표준 툴을 사용해 구성될 수 있습니다. DSS 런타임은 네트워크 자원에 접근을 통제하는 CAS 퍼미션을 가집니다. 만일 서비스가 이 접근권한을 가지지 않으면 단지 같은 노드 내에서의 서비스만 액세스할 수 있습니다.

설정

보안을 사용 가능하게 하기 위해, DSS Node 는 보안 설정 파일의 경로와 함께 설정되어야 합니다. 만일 노드가 이 경로를 지정하지 않으면, 어떠한 보안 기능도 그 노드에서 사용되지 않을 것입니다. 만일 어떠한 파일도 지정된 경로에 존재하지 않으면, 노드는 필요한 인증과 제한없는 서비스 어셈블리 로딩을 실행합니다.

- DssHost
- 보안 설정 파일

  - AuthenticationRequired
  - OnlySignedAssemblies
  - 사용자
- 보안 관리자 서비스

DssHost

DSS Host 툴(DssHost.exe)을 사용할 때, 보안 설정 파일의 위치를 다음 두 개의 방법으로 지정합니다.

- 커맨드 라인 인수 /s (혹은 /settings)를 사용함:

```
dsshost.exe /p:50000 /t:50001 /s:storeWSecuritySettings.xml
```

- DssHost.exe.config 파일에서-appConfig 부분에 SecuritySettings 키를 사용함:

```
<appConfig>
<add key="Security" value=.. WstoreWSecuritySettings.xml"/>
</appConfig>
```

> 만일 상대 경로가 DSS Node에 입력되면 그 경로는 DSS노드(dsshost.exe)의 위치에서 상대적으로 결정됩니다.

보안 설정 파일

보안 설정 파일은 다음 세 개의 아이템을 포함합니다:

- AuthenticationRequired
  인증이 필요함 설정.
- OnlySignedAssemblies
  단지 서명된 어셈블리만 로드함 설정.
- Users
  노드에서 인증될 수 있는 사용자의 리스트

다음은 보안 설정 파일의 예입니다:

```
<?xml version="1.0" ?>
<SecuritySettings
```

```
  xmlns="http://schemas.microsoft.com/robotics/2006/10/security.html">
  <AuthenticationRequired>true</AuthenticationRequired>
  <OnlySignedAssemblies>false</OnlySignedAssemblies>
  <Users>
    <UserPermission>
      <Rights>All</Rights>
      <UserName>MSRS_TEST</UserName>
    </UserPermission>
    <UserPermission>
      <Rights>All</Rights>
      <UserName>MSRS_TESTA</UserName>
    </UserPermission>
    <UserPermission>
      <Rights>All</Rights>
      <Sid>S-1-5-32-544</Sid>
    </UserPermission>
  </Users>
</SecuritySettings>
```

AuthenticationRequired

이 요소는 true 혹은 false 어느 쪽으로도 설정될 수 있습니다. 이것이 true 로 설정될 때 노드는 인증을 필요로 합니다. 그렇지 않으면 노드는 인증이 필요없는 접속을 채택합니다.

OnlySignedAssemblies

이 요소는 true 혹은 false 어느 쪽으로도 설정될 수 있습니다. 이것이 true 로 설정될 때 서명된 Authenticode 로 서명된 어셈블리의 서비스만 로드될 것입니다. 그렇지 않으면 노드는 어떤 어셈블리의 서비스라도 로드할 것입니다.

Users

이 요소는 노드에 액세스할 퍼미션이 있는 사용자의 리스트를 포함합니다. AuthenticationRequired 요소가 true 일 경우만 이 리스트가 관련됩니다. 이것은 UserPermission 레코드의 리스트입니다. 이 레코드의 각각은 다음과 같이 최대 세 개의 요소를 가집니다:

요소	설명
Rights	이 필드는 사용자의 노드에 대한 퍼미션을 지정합니다. 현재 버전의 퍼미션 레벨은 All과 None입니다
Sid	이것은 사용자를 위한 표준 윈도우 Security Identifier(SID)로 Security Descriptor Definition Language(SDDL)[1]로 표현되어 있습니다, 이 요소는 선택적입니다.
UserName	이것은 사용자를 위한 사용자명입니다. 이 요소는 선택적이지만 UserPermission 레코드가 Sid 또는 UserName 요소 중 하나를 반드시 포함해야 합니다.

보안 설정 파일을 직접 만들 때 일반적으로 UserName 요소를 사용자 또는 그룹에 퍼미션을 지정하기 위해 사용하는 것이 편리합니다.

잘 알려진 사용자를 위해, UserName은 Windows에 로컬라이즈 되어있습니다. 예를 들면, BUILTINWAdministrators(로컬 머신 관리자 권한을 가지는 모든 사용자를 포함하는 그룹) 사용자명은 단지 윈도의 영어 버전에서만 정확히 지원될 것입니다. 보안 설정 파일이 머신 사이에서 사용될 때 Sid 필드는 잘 알려진 사용자를 위해 사용될 필요가 있습니다. 위 예에서는 UserPermission 레코드에 UserName인 BUILTINWAdministrators 대신에 Sid, S-1-5-32-544가 사용되었습니다.

보안 관리자 서비스

DSS Node 가 보안 설정 파일이 지정되어 시작될 때 보안 관리자 서비스가 시작됩니다.

보안 관리자 서비스는 최종 사용자가 웹 인터페이스를 통하여 보안 설정을 조작하게 해주며 DSSP 를 사용하는 다른 서비스에서의 프로그램적인 보안 설정 변경을 가능하게 해줍니다.

주요 설정은 AuthenticationRequired 와 OnlySignedAssemblies 로 노드 시작 시 읽어집니다. 노드가 실행 중인 동안 이것을 변경하는 것은 영향을 미치지 않습니다. 오직 보안 설정 파일을 사용하도록 시작된 노드는 새로운 설정을 얻을 것입니다.

런타임에 사용자 리스트를 변경하여 노드에 대한 퍼미션을 가지는 그룹과 사용자 리스트를 변경할 수 있습니다.

인증

입력 연결이 만들어 지면 노드는 NTLM 을 사용해 원격 측을 확인합니다. 인증이 사용 가능하게 설정된 경우 연결이 다른 DSS Node 로부터 DSSP.TCP 나 HTTP 를 사용하는 지에 관계없이 인증을 수행합니다.

내부적으로 DSS Node 는 연결과 원격 사용자를 위한 위장토큰(impersonation token)[2]을 연결시킵니다. 위장토큰은 원격 사용자의 SID 와 원격 사용자가 멤버인 그룹 리스트(SID 들의 리스트로 표현된)를 포함합니다.

원격 사용자가 현재 노드에 액세스할 퍼미션이 있는지 확인하기 위해 다음 검사를 수행합니다:

1. 원격 사용자가 노드가 실행하는 사용자 문맥과 같은 경우, 현재 사용자는 항상 노드에 대한 모든 권한을 가집니다. 이것은 아무도 액세스 액세스할 퍼미션을 가지지 않은 상태에서 노드가 실행됨을 방지합니다.
2. 원격 사용자의 SID 또는 그 그룹의 SID 가 UserPermissions 리스트에 있으면, 사용자는 일치되는 항목에 대한 모든 권한을 갖게 됩니다.
3. 사용자가 All 권한을 가지면, 연결을 허용하며 그렇지 않으면 입력 접속을 거절합니다.

UserPermissions 리스트는 허용 리스트입니다. 사용자 또는 그룹에 액세스를 거부하는 메커니즘이 존재하지 않습니다.

서비스 어셈블리 로딩

서비스가 생성될 때 그 서비스가 구현되어 있는 어셈블리가 로드됩니다. 서비스가 초기화되기 전에, 어셈블리는 Authenticode 가 서명되어 있는지 검사됩니다.

만일 OnlySignedAssemblies 가 true 로 설정되어 있으면 서명되지 않은 어셈블리는 로드되지 않을 것입니다.

이 경우 믿을 만한 발행인(로컬 사용자와 머신의 인증서 기억 장치에 있는)이 만든 인증서로 서명된 어셈블리들만 로드될 것입니다.

네트워크 액세스 퍼미션

- 서비스간 통신
- 네트워크 액세스를 제한
- 코드 액세스 보안 설정

서비스간 통신

서비스가 또 다른 서비스와 통신을 구축하는 세 개의 방법이 있습니다:

- 서비스 디자인 시 Partner 어트리뷰트를 사용

- 서비스 시작 시 매니페스트를 사용
- DsspServiceBase.ServiceForwarder<T> 또는 런타임에 DsspServiceBase.ServiceForwarderUnknownType()으로 생성된 포워더를 사용

이 옵션 중 세 번째가 사용되면, 서비스는 다른 노드의 서비스에 포워더를 생성하기 위해 DssNetworkPermission CAS 퍼미션이 필요합니다. 만일 서비스가 FullTrust 로 로드된 경우는 이 퍼미션을 가집니다.

네트워크 액세스를 제한하는 것

DSS Service 는 Code Access Security(CAS)정책을 생성해 서비스의 네트웍 액세스를 제한할 수 있습니다. 만일 서비스를 포함하고 있는 어셈블리가 DssNetworkPermission 을 가지지 않으면 서비스는 원격 서비스에 포워더를 만들 수 없습니다.

서비스가 단지 Execution 퍼미션을 가지게 제한되면, 서비스는 DsspServiceBase 클래스에서 이용할 수 있는 인터페이스를 통하여 네트웍을 액세스할 수 있습니다. 이것은 서비스가 다른 서비스와 상호 작용하게 하지만 DssNetworkPermission 클래스 없이 네트워크에 데이터를 보낼 수 없습니다.

코드 액세스 보안 설정

코드 액세스 보안(Code Access Security) 정책은 .NET Framework 2.0 설정 도구를 사용해 생성되고 편집될 수 있습니다. 설정 정책을 구성하는 자세한 방법은 MSDN 의 .NET Framework Configuration Tool(Mscorcfg.msc)에 페이지에서 알 수 있습니다.

런타임 사용법

- 보안 관리자 서비스
- 인증

보안 관리자 서비스

서비스는 보안 관리자 서비스를 런타임에 보안 설정을 변경하기 위해 사용할 수 있습니다.

사용자 리스트는 InsertUser UpdateUser 과 DeleteUser 메시지를 전송하는 것으로 변경될 수 있습니다. 이 메시지는 그 몸체에 UserPermission 타입을 가집니다.

UserPermission 타입은 사용자의 텍스트 표현을 사용합니다. 다음과 유사한 SID 코드로부터 그것을 생성합니다.

```
string UserNameFromSid(SecurityIdentifier sid)
{
    NTAccount account = sid.Translate(typeof(NTAccount)) as NTAccount;
    if (account != null)
    {
        return account.Value;
    }
    return null;
}
```

잘 알려진 계정과 그룹명은 Windows 에 로컬라이즈 되어 이습니다. 그래서 이를 하드코딩(hard-coding)하는 것은 서비스를 다른 머신에서 신뢰할 수 없게 할 것입니다. 잘 알려진 SID 로부터 텍스트명을 만들기 위해, 다음과 유사한 코드를 사용합니다.

```
string UserNameFromWellKnownSid(WellKnownSidType wellKnownSid)
{
    SecurityIdentifier sid = new SecurityIdentifier(wellKnownSid, null);
    NTAccount account = sid.Translate(typeof(NTAccount)) as NTAccount;
    if (account != null)
    {
        return account.Value;
    }
    return null;
}
```

예를 들어 다음은 Everyone 그룹을 표시하는 UserPermission 객체를 안전하게 생성합니다:

```
    UserPermssion user = new UserPermission();
    user.Rights = DsspRights.All;
    user.User = UserNameFromWellKnownSid(WellKnownSidType.WorldSid);
```

이터레이터 함수 내의 사용자 리스트에 이 사용자를 추가합니다:

```
Fault fault = null;

yield return Arbiter.Choice(
    _secMgr.InsertUser(user),
    delegate(DefaultInsertResponseType success){},
    delegate(Fault f)
    {
        fault = f;
    }
);

if (fault != null)
{
    LogError(f);
    yield break;
}
```

AuthenticationRequired 또는 OnlySignedAssemblies 설정을 변경할 때는 오로지 Replace 메시지를 사용해야 하며 현재 사용자 리스트를 삭제하지 않아야 합니다. 변화는 다음 DSS Node 시작 때 반영됩니다.

인증

AuthenticationRequired 가 사용 가능하게 설정될 때, 인증된 전송에서 도착한 메시지는 헤더를 가집니다. 이 헤더는 요청을 한 사용자를 확인해 줍니다. 이것은 서비스가 다른 사용자를 위해 다른 동작을 선택하게 해줍니다.

일반적인 DSSP 조작을 위해, 헤더는 메시지에 대해 GetHeader 메서드를 호출하여 검색될 수 있습니다.

```
sec = Microsoft.Dss.Runtime.Security를 사용하는 것
```

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> GetHandler(Get get)
{
    sec.UserPermission user = get.GetHeader<sec.UserPermission>();

    if (user != null)
    {
        // process per user
    }
```

HttpGet HttpPost 와 HttpQuery 메서드를 이용해 웹 클라이언트에서 온 메시지를 처리하기 위해, HttpListenerContext 의 일부로서 사용자 정보가 전달 됩니다. 이것은 FromIdentity 메서드를 이용해 UserPermission 으로 바꿀 수 있습니다.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> HttpGetHandler(HttpGet get)
{
    IPrincipal principal = get.Body.Context.User;

    if (principal != null)
    {
        sec.UserPermission user = sec.UserPermission.FromIdentity(
            principal.Identity
        );

        if (user != null)
        {
            // process per user
        }
    }
```

[1] SID 에 관한 자세한 정보는 MSDN 의 Security Identifiers 항목을 보십시오

[2] DSS Node 에서 사용되는 위장레벨(impersonation level)은 Identify 입니다.

4. DSS 툴 및 유틸리티

Command Line 옵션들

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Command Line Arguments

DSS 툴 및 유틸리티

DSS Contract 정보 툴 (DssInfo.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- DSS Contract Information Tool (DssInfo.exe)

DSS 배포 툴 (DssDeploy.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- DSS Deploy Tool (DssDeploy.exe)

DSS Host Tool (DssHost.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- DSS Host Tool (DssHost.exe)

DSS 신규 서비스 생성 툴 (DssNewService.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- DSS New Service Generation Tool (DssNewService.exe)

DSS 프로젝트 마이그레이션 툴 (DssProjectMigration.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- DSS Project Migration Tool (DssProjectMigration.exe)

DSS Proxy 생성 툴 (DssProxy.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- DSS Proxy Generator Tool (DssProxy.exe)

보안 툴 및 유틸리티

HTTP 네임스페이스 점유 툴 (HttpReserve.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- HTTP Namespace Reservation (HttpReserve.exe)

시험 툴 및 유틸리티

CLR Runtime 정보 툴 (ClrRuntimeInfo.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- CLR Runtime Information Tool (ClrRuntimeInfo.exe)

Execute Command 툴 (ExedcuteCommand.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Execute Command Tool (ExecuteCommand.exe)

Random 파일 생성 툴 (RandFiles.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Random File Generator Tool (RandFiles.exe)

Web Get 툴 (WebGet.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Web Get Tool (WebGet.exe)

Web Publish 툴 (WebPublish.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Web Publish Tool (WebPublish.exe)

Xml 편집 툴 (XmlReplace.exe)

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Xml Editing Tool (XmlReplace.exe)

5. DSS 서비스 튜토리얼

서비스 튜토리얼 1(C#) - 서비스 생성하기

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 서비스 생성하기

새로운 서비스를 만드는 것으로 시작합니다.

시작 메뉴를 열어 모든 프로그램에서 마이크로소프트 Robotics Studio 를 선택한 후 커맨드 프롬프트를 선택하십시오. 이것은 Robotics Studio 설치 경로의 루트 디렉터리로 하는 Robotics Studio 를 위한 커맨드 프롬프트 창을 엽니다.

당신의 첫번째 서비스를 만들도록 Samples 디렉터리로 이동하여 아래처럼 매개 변수를 사용해 DssNewService 도구를 실행합니다. 그리고 ServiceTutorial1(ServiceTutorial<Number One>)으로 이동합니다. 이 과정은 여러분의 시작을 돕기 위한 템플릿을 자동으로 만듭니다.

```
cd Samples
dssnewservice /namespace:Robotics /service:ServiceTutorial1
cd ServiceTutorial1
```

여기서 ServiceTutorial1.sln 이름의 비주얼 스투디오 Solution 이 ServiceTutorial1 디렉터리에서 만들어집니다. C#에디터를 사용해 이 솔루션을 로드합니다.

```
start ServiceTutorial1.sln
```

다음으로, 솔루션을 빌드합니다. 비주얼 스투디오에서 빌드 메뉴를 열고, 그 다음 솔루션 빌드(또는 F6 을 누름)를 선택하여 빌드합니다. 또한 커맨드 프롬프트에서 다음 명령을 입력해 컴파일할 수 있습니다

```
msbuild ServiceTutorial1.sln
```

단계 2: 서비스 시작하기

마이크로소프트 Robotics Studio 디렉터리로 이동합니다.

```
cd ..W..
```

여러분은 bin 디렉터리에서 방금 프로젝트를 빌드했을 때 만들어진 파일을 다음과 같이 보아야 합니다.

```
.
.
.
ServiceTutorial1.Y2007.M07.dll
ServiceTutorial1.Y2007.M07.pdb
ServiceTutorial1.Y2007.M07.Proxy.dll
ServiceTutorial1.Y2007.M07.Proxy.pdb
ServiceTutorial1.Y2007.M07.proxy.xml
ServiceTutorial1.Y2007.M07.transform.dll
ServiceTutorial1.Y2007.M07.transform.pdb
```

주의사항:

기본적으로, DssNewService 는 생성된 서비스의 어셈블리 파일의 이름에 현재 년도와 월을 Y2007.M07 와 같이 추가합니다. 현재 날짜에 따라 상기의 예에서 보는 것과 다른 이름을 가질 것입니다. Robotics Studio 에 설치된 ServiceTutorial1 프로젝트와 다른 날짜인 여러분의 ServiceTutorial1 파일을 볼 수 있습니다.

서비스를 실행하기 위해, DSS 호스팅 애플리케이션인 DssHost.exe 를 실행하여 DSS 노드를 우선 실행해야 합니다. DssHost 는 여러분의 서비스를 시작합니다. 서비스들은 DssHost 로 시작하는 3 개의 방법이 있습니다:

- 커맨드 라인 플래그 /manifest 를 사용한 매니페스트에 의해
- 커맨드 라인 플래그 /dll 을 사용한 어셈블리 이름에 의해
- 커맨드 라인 플래그 /contract 를 사용하면서 컨트랙트에 의해

매니페스트 파일은 서비스를 시작하기 위해 필요한 정보를 포함하는 XML 파일입니다. DssNewService 는 DssHost 로 서비스를 시작하는 데 필요한 정보를 포함하는 ServiceTutorial1.manifest.xml 파일을 자동적으로 만듭니다.

매니페스트를 사용해 DssHost 를 시작합니다.

```
binWdsshost                                                    /port:50000
/manifest:"samplesWServiceTutorial1WServiceTutorial1.manifest.xml"
```

주의사항:

설치된 튜토리얼 프로젝트를 위한 매니페스트 파일은 samplesWConfig 폴더에 있습니다. samplesWServiceTutorialsWServiceTutorial1 폴더에서 ServiceTutorial1 프로젝트를 사용하고 있으면 samplesWConfig 로부터 올바른 매니페스트로 DssHost 를 실행해야 합니다.

binWdsshost /port:50000 /manifest:"samplesWConfigWServiceTutorial1.manifest.xml"

그러나, 설치된 튜토리얼 프로젝트는 각 튜토리얼을 위해 완성된 코드를 포함합니다. 따라서 튜토리얼의 모든 단계가 완성된 후 서비스가 올바로 실행됩니다.

해당 매니페스트를 이용한 서비스 로드 결과를 아래와 같이 볼 수 있습니다:

```
.
.
.
*   Starting manifest load: .../ServiceTutorial1.manifest.xml
*   Service uri:  ...[http://localhost:50000/servicetutorial1]
*   Manifest load complete ...[http://localhost:50000/manifestloaderclient]
```

이제 웹 브라우저를 열러 다음 주소를 입력합니다.

http://localhost:50000/servicetutorial1

새롭게 만들어진 서비스인 ServiceTutorial1State 의 XML 직렬화(표현)는 SOAP envelope 에 포함되어(encapsulated) 아래와 같이 나타납니다.

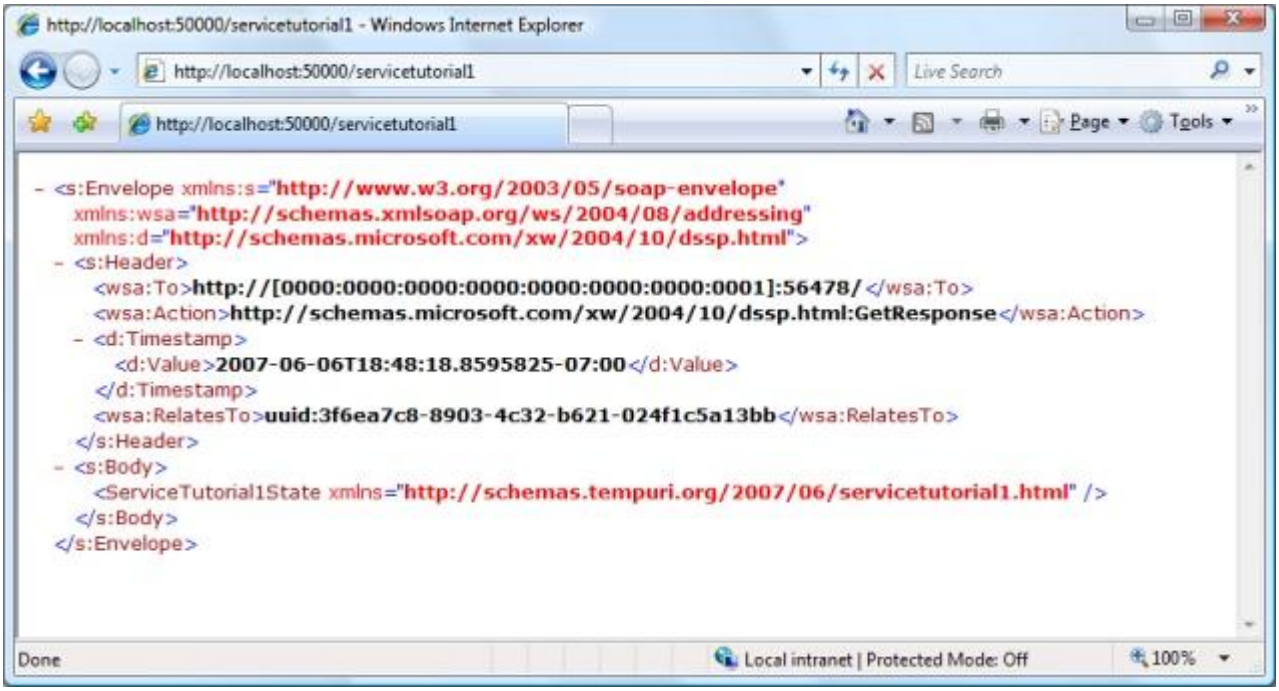


그림 1- 브라우저에서 http://localhost:50000/servicetutorial1 는 envelope 에 포함된 서비스의 상태를 표시합니다.

커맨드 프롬프트 창에서 CTRL+C 를 누르면, " Shutdown complete " 메시지가 나타나고 DssHost 는 종료 됩니다.

단계 3: HTTP GET 지원하기

웹 브라우저에 상태를 보낼 때, 서비스는 선택적으로 SOAP envelope 를 전송하는 것을 피할 수 있고, 그 상태의 XML 직렬화를 다음과 같이 대신 전송할 수 있습니다.

ServiceTutorial1Types.cs 파일은 서비스의 컨트랙트(Contract)를 정의합니다. 그것은 이 서비스의 컨트랙트 식별자, 상태, 조작 포트, 조작 메시지와 요청/응답 타입을 포함합니다. 이 튜토리얼을 진행함에 따라 서비스의 컴포넌트에 대해 자세히 알 수 있습니다.

파일 ServiceTutorial1Types.cs 를 다음과 같이 변경하십시오:

1. Microsoft.Dss.Core.DsspHttp 네임스페이스를 using 지시자로 추가합니다. 이 네임스페이스는 서비스가 표준 웹 브라우저 같은 표준 HTTP 의뢰인의 요청에 응답하기 위한 메시지 정의를 포함하고 있습니다.

```
using Microsoft.Dss.Core.DsspHttp
```

2. 다음으로, ServiceTutorial1State 클래스에 다음의 어트리뷰트를 추가하십시오, 이것은 직렬화된 데이터를 정확히 보게 합니다. 서비스 튜토리얼 6(C#)에서 서비스간 정보를 전송하기 위해 ServiceTutorial1State 클래스를 사용할 것입니다.

```
private string _member = "This is my State!";

[DataMember]
public string Member
{
    get { return _member; }
    set { _member = value; }
}
```

DataContract 어트리뷰트는 ServiceTutorial1State 클래스가 XML 직렬화가능(serializable) 을 지정합니다. DataContract 어트리뷰트라고 하는 특징이 있는 타입 내에서, 개개의 프로퍼티와 필드에 DataMember 어트리뷰트를 사용해 XML 직렬화가능(serializable)을 명백하게 표시해야 합니다. 오직 이 선언이 정의된 퍼블릭 프로퍼티과 필드만이 직렬화 될 것입니다. 또한 프로퍼티 멤버(property member)를 직렬화를 위해 set 과 get 메서드 구현이 필요할 것입니다.

주목:

*어트리뷰트*는 프로그램 작성에 키워드형식의 주석을 추가하는 .NET 의 특징입니다. 이때 프로그램 가능한 멤버는 예컨대 타입, 필드, 메서드와 프로퍼티입니다. .NET 어트리뷰트에 대해 더 알기 위해서는 .NET Framework Developer's Guide 의 Attributes Overview 를 봅니다.

3. 이제, 서비스의 포트로 지원되는 메시지의 리스트에서 HttpGet 메시지를 추가합니다. 포트는 메시지가 서비스 사이에서 전달되는 통로 메커니즘입니다. PortSet 은 단지 포트들의 집합(collection)입니다.

```
[ServicePort]
public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup,
DsspDefaultDrop, Get, HttpGet>
{
}
```

4. ServiceTutorial1.cs 파일에, HttpGet 메시지 지원을 추가하십시오. ServiceTutorial1.cs 서비스의 동작을 정의합니다.

여기도 DsspHttp 를 사용하기 위해 using 명령문을 추가하십시오.

```
using Microsoft.Dss.Core.DsspHttp;
```

5. 그리고 ServiceTutorial1 클래스에, HttpGet 메시지를 위한 메시지 핸들러를 추가하십시오.

```
/// <summary>
/// Http Get Handler
/// </summary>
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> HttpGetHandler(HttpGet httpGet)
{
    httpGet.ResponsePort.Post(new HttpResponseType(_state));
    yield break;
}
```

이 핸들러는 HttpGet 메시지 응답 포트에 HttpResponseType 메시지를 보냅니다. DSS 노드 내의 HTTP 기반 구조는 XML 에서 제공된 상태를 직렬화할 것이고, HTTP 요청에 대한 응답을 보냅니다.

고급:

HttpResponseType 생성자는 여기 사용된 것과 다른 많은 상속(overload)들을 가집니다. 이 중 하나는 서비스 저자가 웹 클라이언트에 의해 사용될 수 있는 XSLT 파일의 경로를 지정하게 합니다. (참조 서비스 튜토리얼 6(C#)).

6. 서비스를 빌드하고 실행(F5 를 누르거나, 디버그 > 디버깅 시작 메뉴 커맨드를 선택합니다)하십시오. 그리고 웹 브라우저에 http://localhost:50000/servicetutorial1을 입력해 다음을 볼 수 있습니다.

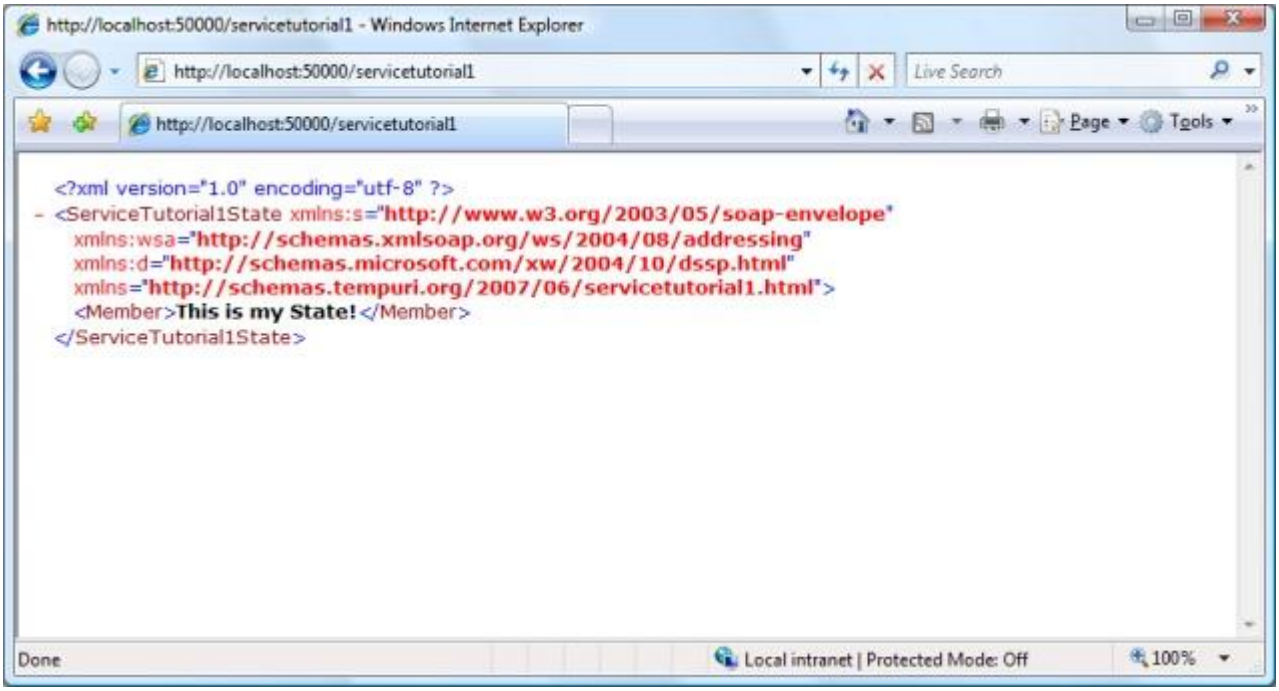


그림 2 - 웹브라우저에서 보여진 서비스의 상태, ServiceTutorial1State 와 그 멤버

단계 4: 컨트롤 패널을 사용하기

서비스를 시작하기 위해 마이크로소프트 Robotics Studio 의 자체 DSS 서비스 중 하나인 컨트롤 패널을 사용할 수 있습니다. 이것을 시험해 보기 위해, 현재의 DSS 노드가 있다면 CTRL+C 를 눌러 종료하십시오. 그리고 매니페스트를 입력하지 않고 DssHost 를 다시 실행합니다.

```
binWdsshost /port:50000
```

이제 브라우저에서 http://localhost:50000 를 엽니다. 페이지가 로드될 때, 왼쪽 메뉴 위쪽에 컨트롤 패널을 클릭하십시오. 현재 노드에서 볼 수 있는 서비스들의 테이블이 브라우저에 나타납니다.

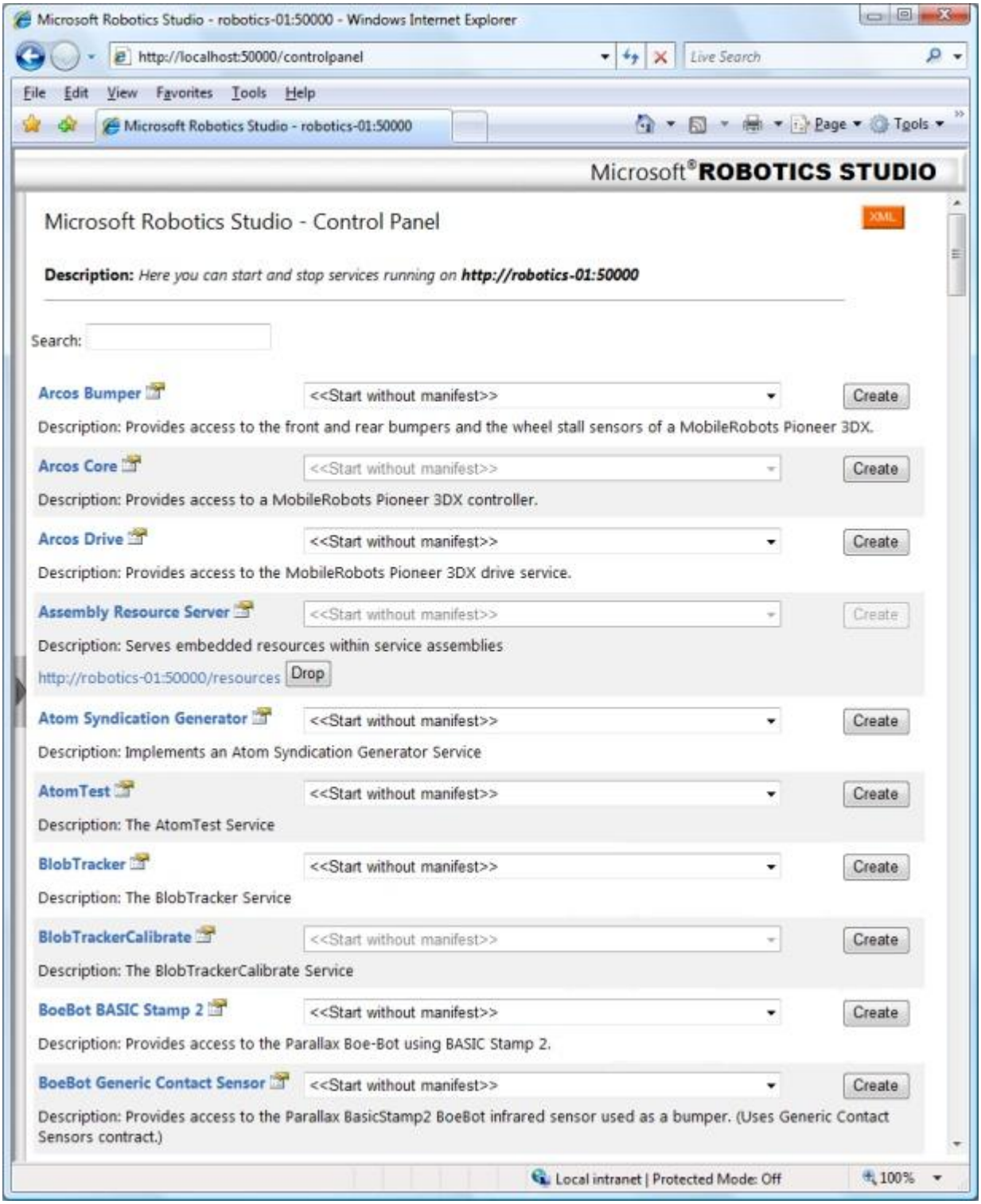


그림 3 - 브라우저에서 본 마이크로소프트 Robotics Studio 컨트롤 패널

표의 각 행은 서비스의 이름으로 시작되며, 그 옆에는 서비스를 실행할 수 있는 매니페스트를 선택하는 dropdown 리스트가 있습니다. 만일 현재 실행하고 있는 그 서비스가 있으면, 서비스의 Description 아래에서 그 인스턴스를 위한 URL 을 볼 수 있습니다. 실행되고 있는 서비스는

URL 의 오른쪽에 Drop 버튼이 활성화 되고, 버튼을 클릭하여 해당 서비스에 Drop 메시지를 보냅니다. 이 메시지는 서비스를 멈춥니다.

페이지를 아래로 스크롤하여 ServiceTutorial1 을 위한 항목을 발견하거나, Search 박스에 servicetutorial1을 입력해 서비스를 찾을 수 있습니다.

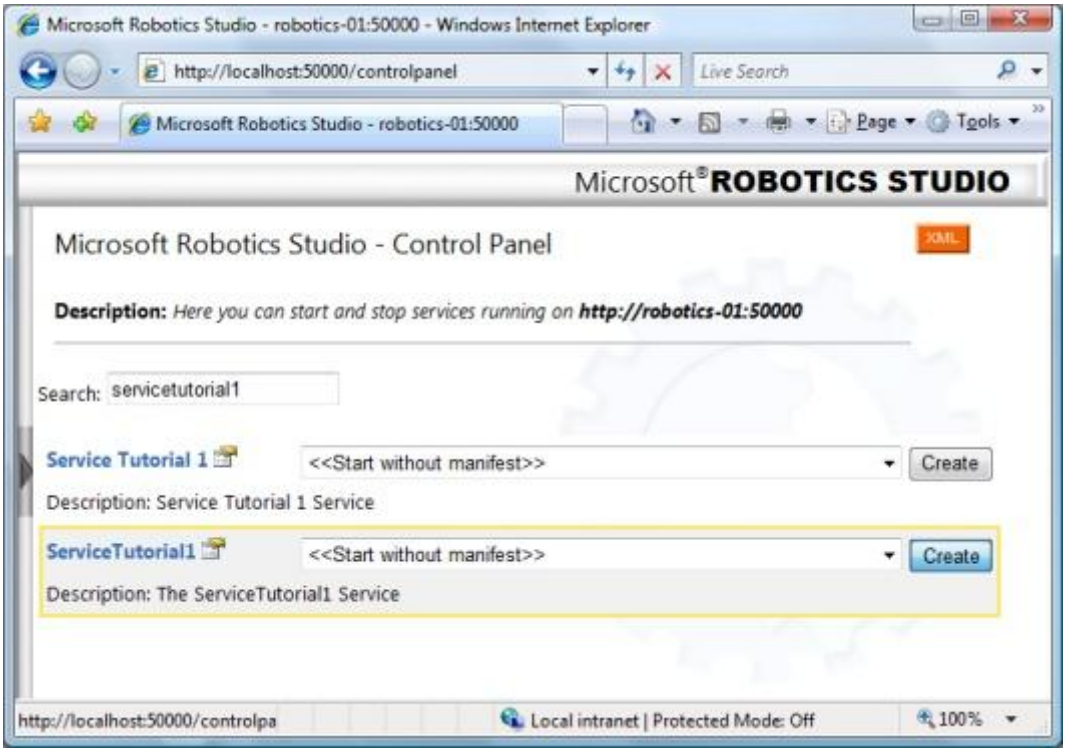


그림 4-컨트롤 패널의 ServiceTutorial1을 위해 Entry

Search 박스에 입력 후 두 개의 결과를 볼 것입니다. 그 중 하나는 마이크로소프트 Robotics Studio 와 같이 설치된 완성된 서비스 튜토리얼 1 입니다. 이들은 dropdown 리스트에서 매니페스트의 위치로 구분할 수 있으며, 직접 어셈블리(dll 파일)를 manifest 없이 <<Start without manifest>>를 선택해 직접 로드할 수 있습니다. 매니페스트에 열거된 일련의 파트너 서비스들을 해당 서비스와 같이 실행할 경우 매니페스트 실행이 필요합니다.

서비스는 Create 버튼을 클릭하여 실행하십시오.

이제 왼쪽 탐색 구역에서 Service Directory 를 선택합니다. 현재 실행되고 있는 서비스 리스트에서 /servicetutorial1 을 볼 수 있습니다. /controlpanel 을 포함해 실행되고 있는 다른 서비스들이 DSS 런타임의 각기 다른 컴포넌트고, DSS 환경이 초기화될 때 디폴트로 시작되는 것을 알 수 있습니다. 각 서비스의 상태는 URL 을 클릭하거나 직접 브라우징 하여 조사될 수 있습니다.

주의사항:

노드가 시작할 때 컨트롤 패널 서비스는 항상 시작됩니다. 여러분은 원하는 서비스를 DSS 노드를 다시 시작하지 않고 실행하고 멈출 수 있습니다. 그러나 DSS 노드는 서비스 어셈블리를 로드하기 때문에 서비스를 다시 빌드한 경우에는 DSS 노드를 종료하고 재시작 해야만 합니다.

단계 5: 서비스 멈추기

서비스가 실행되는 중, 웹브라우저에서 http://localhost:50000/controlpanel 를 엽니다.

주의사항:

DSS 노드에서 실행되는 서비스를 업데이트하기 위해 컨트롤 패널 서비스를 리프레시할 수 있습니다.

이전 섹션에 기술된 바와 같이 servicetutorial1 을 위한 항목을 찾으십시오.

그림 5-컨트롤 패에 있는 ServiceTutorial1 서비스 인스턴스와 Drop 버튼

URL 을 클릭해 서비스의 상태를 조사하거나, Create 버튼을 클릭해 새로운 인스턴스를 만들거나, Drop 버튼을 클릭해 서비스를 멈출 수 있습니다.

단계 6: Replace 지원하기

Replace 메시지는 서비스의 현재 상태를 교체하기 위해 사용됩니다. Replace 메시지가 전송 될 때 서비스의 현재 상태는 Replace 메시지의 보디에 지정된 상태 객체로 교체됩니다. 이것은 새로운 상태로 서비스를 초기화하거나, 서비스가 동작하는 어떤 때라도 이전에 저장된 상태로 복구하는 것을 허락합니다.

서비스에서 Replace 를 지원하기 위해, ServiceTutorial1Types.cs 에서 Replace 타입을 정의하십시오.

```
    public class Replace : Replace<ServiceTutorial1State,
PortSet<DefaultReplaceResponseType, Fault>>
    {
    }
```

그 다음 Portset 에 Replace 를 더합니다.

```
    /// <summary>
    /// ServiceTutorial1 Main Operations Port
    /// </summary>
    [ServicePort]
    public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup,
DsspDefaultDrop, Get, HttpGet, Replace>
    {
    }
```

Replace 핸들러를 ServiceTutorial1.cs 에서 더합니다.

```
        /// <summary>
        /// Replace Handler
        /// </summary>
        [ServiceHandler(ServiceHandlerBehavior.Exclusive)]
        public IEnumerator<ITask> ReplaceHandler(Replace replace)
        {
            _state = replace.Body;
            replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
            yield break;
        }
```

상기의 코드에 ServiceTutorial1State 타입인 Replace 메시지의 보디(Body)를 이 서비스의 _state 에 할당합니다. 그 다음, DefaultReplaceResponseType 타입의 성공 응답을 Replace 메시지의 ResponsePort 에 포스트합니다.

Replace 조작은 서비스들 사이에서 값을 교환하기 위해 나중에 사용할 것입니다.

부록 A: 코드

ServiceTutorial1Types.cs

ServiceTutuorial1Types.cs 파일은 서비스 컨트랙트를 정의합니다. 컨트랙트는 .NET CLR 네임스페이스와 서비스가 지원하는 메시지의 집합과 관련된 고유한 텍스트 열로 정의됩니다.

다음과 같이 이 파일에서 사용되는 네임스페이를 정의합니다.

```
using Microsoft.Ccr.Core;
```

```
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.Core.DsspHttp;
using Microsoft.Dss.ServiceModel.Dssp;

using System;
using System.Collections.Generic;
using W3C.Soap;

using servicetutorial1 = RoboticsServiceTutorial1;
```

컨트랙트 클래스

```
    /// <summary>
    /// ServiceTutorial1 Contract class
    /// </summary>
    public sealed class Contract
    {
        /// <summary>
        /// The Dss Service contract
        /// </summary>
        public const String Identifier =
"http://schemas.tempuri.org/2006/06/servicetutorial1.html";
    }
```

이 컨트랙트 클래스는 고유한 문자열인 Identifier 를 정의합니다. URI(고유한 Resource Identifier)는 각 서비스 별로 고유한 이름을 지정하기 위해 사용하는 것으로 XML 문서에서 사용되는 규약을 따릅니다. 사용되는 디폴트 메커니즘은 URI 를 호스트명(DssNewService 에 매개변수로 제공됨), 경로(이 예에서 비어 있음), 년, 월과 서비스의 이름을 조합하는 것입니다. 만약 서비스 저자가 일부 관리 수준을 가지는(예를 들면, http://spaces.live.com 의 계정 주소를 사용)호스트명과 경로와 더불어 날짜와 서비스명의 조합으로 네임스페이스가 조립되면 서비스에 관한 소량의 정보를 포함하게 되는 이점과 고유성을 사용자에게 제공합니다. URI 는 웹 페이지에 대한 어떠한 요구 사항도 없으며 서비스 저자가 이와 연결되는 페이지를 만들 필요도 이유도 없습니다.

고급:

만일 불투명한 고유 Identifier 가 필요할 경우(가능하지만, 권고 되지 않습니다) GUID(GuidGen.exe 툴을 사용하여 생성)를 이용해 "urn: uuid: 4de060f3-f665-11da-95e7-00e08161165f" 형식의 ID 를 사용하십시오

ServiceTutorial1State 클래스

```
    /// <summary>
    /// The ServiceTutorial1 State
    /// </summary>
    [DataContract]
    public class ServiceTutorial1State
    {
        private string _member = "This is my State!";

        [DataMember]
        public string Member
        {
            get { return _member; }
            set { _member = value; }
        }
    }
```

ServiceTutorial1State 클래스는 멤버라는 한 개의 퍼블릭 프로퍼티를 가지며, 직렬화를 위해 사용되는 DataMember 어트리뷰트이 멤버에 정의됩니다. 디폴트로, null 인 프로퍼티 또는 필드는 직렬화하게 되지 않을 것입니다.

ServiceTutorial1Operations 클래스

```
    /// <summary>
    /// ServiceTutorial1 Main Operations Port
    /// </summary>
    [ServicePort]
    public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup,
DsspDefaultDrop, Get, HttpGet, Replace>
    {
    }
```

이 클래스는 서비스가 지원하는 퍼블릭 메시지를 정의합니다.

메시지	설명
DsspDefaultLookup	모든 서비스는 Lookup<TBody,TResponse>에서 비롯되는 메시지를 지원해야 합니다. 기반 구조는 DsspDefaultLookup을 정의합니다. DsspServiceBase 클래

	스(서비스 구현 클래스가 파생된)는 Lookup을 위해 디폴트 메시지 핸들러를 정의합니다. 실재로 모든 서비스 조작(operations) PortSet에 DsspDefaultLookup을 추가해 이를 정의합니다. 서비스는 그 자체에 대해 기본 정보로 Lookup 메시지에 응답합니다. 디폴트 구현은 DsspServiceBase로부터 ServiceInfo 프로퍼티를 리턴합니다.
DsspDefaultDrop	서비스에 Drop 메시지가 전송될 때, 서비스는 종료됩니다. 서비스는 이것을 구현할 필요가 없습니다. 디폴트 구현은 DsspServiceBase에서 제공됩니다. 일반적으로 서비스는 조작 PortSet에 DsspDefaultDrop를 추가해 Drop을 지원합니다.
Get	서비스는 그 현재 상태를 Get 메시지로 응답해야 합니다. 서비스는 Get 메시지에 응답하여 그 상태를 변경해서는 안됩니다. Get 메시지가 옵션이지만, 대부분의 서비스들은 이를 구현합니다.
HttpGet	위에서 논의한 것과 같이, 이것은 Get 조작과 같습니다. 그러나 HttpGet 조작은 서비스가 웹 브라우저 클라이언트와 직접 의사소통하게 합니다.
Replace	서비스는 그 전체 상태를 Replace메시지의 보디로 교체해야 합니다. Replace 메시지는 옵션이며 모든 서비스가 이를 구현해야 하지는 않습니다.

Get 과 Replace 는 적합한 디폴트 선언을 가지고 있지 않지만, 다음과 같이 이 서비스는 두 메시지를 지원합니다.

- Get 의 경우, 이것은 Get 메시지에서 주 응답이 서비스의 상태여야 하기 때문입니다. 이 서비스의 경우 는 상태 타입인 ServiceTutorial1State 입니다.
- 이 서비스에서 Replace 를 위한 메시지의 보디는 ServiceTutorial1State 입니다.

```
        /// <summary>
        /// ServiceTutorial1 Get Operation
        /// </summary>
        public Get()
        {
        }

        /// <summary>
        /// ServiceTutorial1 Get Operation
        /// </summary>
        public Get(Microsoft.Dss.ServiceModel.Dssp.GetRequestType body) :
                base(body)
        {
        }

        /// <summary>
        /// ServiceTutorial1 Get Operation
```

```
        /// </summary>
        public          Get(Microsoft.Dss.ServiceModel.Dssp.GetRequestType          body,
Microsoft.Ccr.Core.PortSet<ServiceTutorial1State,W3C.Soap.Fault> responsePort) :
                base(body, responsePort)
        {
        }
    }
    public        class        Replace        :        Replace<ServiceTutorial1State,
PortSet<DefaultReplaceResponseType, Fault>>
    {
    }
```

ServiceTutorial1.cs

ServiceTutorial1.cs 파일은 서비스 구현 클래스를 포함합니다.

```
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.Core.DsspHttp;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Xml;
```

이것은 이 클래스에서 사용될 네임스페이스입니다.

서비스 구현 클래스

코드의 이 섹션은 DsspServiceBase 에서 파생된 ServiceTutorial1 클래스를 선언합니다. 모든 서비스 구현 클래스는 DsspServiceBase 에 유래합니다.

```
    /// <summary>
    /// Implementation class for ServiceTutorial1
    /// </summary>
    [DisplayName("Service Tutorial 1")]
```

```
[Description("Service Tutorial 1 Service")]
[Contract(Contract.Identifier)]
public class ServiceTutorial1Service : DsspServiceBase
{
```

컨트랙트 어트리뷰트는 이 클래스와 컨트랙트 Identifier 사이를 직접 연결시켜주는 선언입니다. DisplayName 과 Description 은 서비스를 기술하는 어트리뷰트입니다. DisplayName 은 타이틀과 유사한 짧은 설명입니다. Description 은 보다 상세한 설명입니다.

ServiceTutorial1State 는 서비스의 상태를 유지합니다. 서비스 상태를 읽고, 변경하는 조작 포트를 통해 다른 서비스의 상태에 접근할 수 있습니다.

```
        /// <summary>
        /// Service State
        /// </summary>
        private ServiceTutorial1State _state = new ServiceTutorial1State();
```

이 예에서 서비스는 다음을 지원합니다

- 전체 상태를 읽습니다(Get 와 HttpGet). 그리고
- 전체 상태를 교체합니다(Replace).

ServicePort 어트리뷰트는 _mainPort 필드가 이 서비스의 메인 조작 포트인 것을 선언합니다.

```
        /// <summary>
        /// Main Port
        /// </summary>
        [ServicePort("/servicetutorial1", AllowMultipleInstances=false)]
        private ServiceTutorial1Operations _mainPort = new ServiceTutorial1Operations();
```

이것은 서비스가 /servicetutorial1 을 디폴트 경로로 지정하게 합니다. 이것은 또한 서비스가 단지 한 개의 인스턴스를 한번에 실행하게 명기합니다. 만일 AllowMultipleInstances = true 로 지정되면, 서비스의 인스턴스가 만들어질 때 마다 독특한 접미부가 경로 뒤에 추가됩니다.

초기화

서비스가 만들어질 때 서비스 튜토리얼 4, 5(C#)에서 논의된 것 같이 파트너 서비스가 만들어지는 두 단계의 생성 과정이 있습니다.

```
        /// <summary>
        /// Default Service Constructor
        /// </summary>
        public ServiceTutorial1Service(DsspServiceCreationPort creationPort) :
                base(creationPort)
        {
        }
```

이 생성자는 생성 과정의 첫 부분에 사용되고, 서비스가 올바로 생성되도록 보여주게 해야 합니다.

Start 메서드는 두 단계 생성 과정의 마지막 동작으로 불립니다.

```
        /// <summary>
        /// Service Start
        /// </summary>
        protected override void Start()
        {
            base.Start();
            // Add service specific initialization here.
        }
```

base.Start() 과정 동안, 서비스는 다음 세가지를 수행합니다. 이는 또한 서비스가 수동으로 수행할 수 있습니다.

1. ActivateDsspOperationHandlers 를 호출해 DsspServiceBase 가 메인 서비스 포트에서 지원되는 각 메시지들을 핸들러에 장착하게 합니다.

   핸들러를 선언하는 방법은 아래를 참조하십시오.

2. DirectoryInsert 를 호출해 해당 서비스를 위한 서비스 레코드가 디렉터리에 삽입하게 되게 합니다. 디렉터리 그 자체가 서비스이며 이 메서드는 디렉터리 서비스에 Insert 메시지를 보냅니다.

   DssHost 가 실행 중일 때, 웹브라우저에 http://localhost:50000/directory 를 입력해 디렉터리 서비스를 확인할 수 있습니다.

3. LogInfo 는 Insert 메시지를 /console/output 서비스에 전송합니다 (http://localhost:50000/console/output). 메시지의 카테고리는 콘솔입니다. 이는 메시지가 커맨드 창 콘솔에 출력되게 합니다. 서비스의 URI 는 출력에 자동적으로 추가됩니다.

다음 코드는 base.Start()에 의해 수행되는 세개의 태스크를 요약합니다. 그러나, 대부분의 경우 이것은 서비스 시작을 수동으로 초기화하는 대신에 base.Start()를 사용해 수행됩니다.

```
        // Listen on the main port for requests and call the appropriate handler.
        ActivateDsspOperationHandlers();

        // Publish the service to the local node Directory
        DirectoryInsert();

        // display HTTP service Uri
        LogInfo(LogGroups.Console, "Service uri: ");
```

메시지 핸들러

다음은 Get 메시지를 위한 핸들러입니다. 이 핸들러는 단순히 메시지의 응답 포트에 서비스의 상태를 포스트합니다.

```
        /// <summary>
        /// Get Handler
        /// </summary>
        /// <param name="get"></param>
        /// <returns></returns>
        [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
        public virtual IEnumerator<ITask> GetHandler(Get get)
        {
            get.ResponsePort.Post(_state);
            yield break;
        }
```

ServiceHandler 어트리뷰트는 base.Start()에서 불리는 메서드인 ActivateDsspOperationHandlers 에 의해 해당 멤버 함수가 메시지 핸들러가 되게 합니다.

ServiceHandlerBehavior.Concurrent 는 메시지 핸들러가 단지 서비스의 상태에 Read-Only 접근만 하도록 지정합니다. 이것은 상태를 변경하지 않는 메시지 핸들러가 동시에 실행되는 것을 허락합니다.

중요:

서비스 상태에 쓰기 접근을 필요로 하는 메시지 처리기는 ServiceHandlerBehavior.Exclusive 를 사용해야 합니다. 이것은 단지 한 개의 핸들러가 서비스 상태를 같은 시간에 변경할 수 있도록 합니다.

메시지 처리기는 일반적으로 Microsoft.Ccr.Core.IteratorHandler<T> 서명을 가집니다

```
public delegate IEnumerator<ITask> IteratorHandler<T>(T parameter);
```

이터레이터(.NET 2.0 의 신기능)는 핸들러에 비동기 동작의 시퀀스를 스레드를 블록없이 사용하게 해줍니다. 이것은 서비스 튜토리얼 3(C#)에서 설명됩니다.

```
        /// <summary>
        /// Http Get Handler
        /// </summary>
        [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
        public IEnumerator<ITask> HttpGetHandler(HttpGet httpGet)
        {
            httpGet.ResponsePort.Post(new HttpResponseType(_state));
            yield break;
        }
```

다음은 Replace 메시지를 위한 핸들러입니다. 이 핸들러가 서비스 상태를 변경할 것을 나태내는 ServiceHandlerBehavior.Exclusive 로 선언된다라는 점에 주목하십시오. 단지 한 개의 Exclusive 핸들러만이 한번에 실행될 것입니다. 그리고 Exclusive 핸들러가 실행되는 동안 어떠한 동시형 핸들러도 실행될 수 없습니다.

```
        /// <summary>
        /// Replace Handler
        /// </summary>
        [ServiceHandler(ServiceHandlerBehavior.Exclusive)]
        public IEnumerator<ITask> ReplaceHandler(Replace replace)
        {
            _state = replace.Body;
            replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
            yield break;
```

```
        }
```

서비스 튜토리얼 2(C#) - 상태 업데이트하기

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 서비스에 새 메시지 추가

서비스는 계수기를 상태에 저장할 것이며, 새로 정의될 메시지에 의해 계수기가 증가될 것입니다. 이 모든 변경은 ServiceTutorial2Types.cs 에서 일어납니다.

여러분이 할 첫번째 변경은 서비스 상태 타입에 정수 프로퍼티, Ticks 를 추가합니다.

```
private int _ticks;

[DataMember]
public int Ticks
{
    get { return _ticks; }
    set { _ticks = value; }
}
```

다음으로, Tick 프로퍼티를 늘리기 위해 메시지를 정의해야 합니다.

아래 코드 부분은 IncrementTick 메시지를 정의합니다. 메시지는 DsspOperation 에서 비롯되고, 세 개의 요소를 가집니다: 동작(action), 보디(body)와 응답 포트(response port). 선언된 IncrementTick 메시지는 동작을 Update(specifically, UpdateRequest)로 설정하는 Update<TBody TResponse> 제네릭 클래스에서 비롯됩니다. 보디 타입은 IncrementTickRequest 라고 선언됩니다. 응답 포트는 PortSet<DefaultUpdateResponseType, Fault>로서 정의됩니다.

Update 는 서비스 상태의 부분 집합을 변경할 메시지에 사용되는 조작입니다. 이것은 단지 Tick 프로퍼티에 영향을 미칠 것입니다.

보디는 서비스가 이 메시지 동작을 수행하는 요청할 때 사용됩니다. 이 경우, IncrementTick 메시지가 상태의 Tick 프로퍼티를 1 씩 증가시키므로 보디는 필드를 포함할 필요가 없습니다. IncrementTick 메시지에 대한 생성자가 보디를 설정하고 생성하기 위해 정의됩니다.

응답 포트는 서비스에 성공 또는 실패를 호출자에게 알립니다. 이 메시지에서도 대부분의 Update 메시지와 같이 DefaultUpdateResponseType 는 업데이트가 성공적이었다라고 신호 보내기에 충분합니다. IncrementTick 메시지가 실패 경로를 가지지 않으면, Fault 타입은 서비스에 메시지를 보내는 데 문제가 있는 경우에 DSS 기반 구조가 호출자에 알릴 때 사용됩니다.

다음 메시지와 요청 타입을 ServiceTutorial2Types.cs 에 추가하십시오.

```
public class IncrementTick : Update<IncrementTickRequest,
PortSet<DefaultUpdateResponseType, Fault>>
{
    public IncrementTick()
        : base(new IncrementTickRequest())
    {
    }
}

[DataContract]
public class IncrementTickRequest
{
}
```

이제 IncrementTick 메시지를 서비스가 지원하는 메시지의 리스트에 추가하십시오. 만일 IncrementTick 메시지가 이 서비스에 바로 보내어지면, 서비스가 종료될 때까지 메시지는 포트에 남아 있을 것입니다. 이는 메모리 누수를 초래합니다. 이것은 우리가 메시지 핸들러를 아직 정의하지 않았기 때문입니다.

```
/// <summary>
/// ServiceTutorial2 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial2Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
Get, HttpGet, Replace, IncrementTick>
{
}
```

단계 2: 메시지 핸들러 구현

다음 단계는 IncrementTick 메시지를 위한 핸들러를 구현합니다. ServiceTutorial2.cs 파일에서 있는 ServiceTutorial2 클래스에 핸들러를 추가하십시오.

이 핸들러는 서비스 상태의 Tick 프로퍼티를 증가시키고, 그 변경을 기록하고, 호출자에게 성공 응답을 보냅니다.

서비스 튜토리얼 1(C#)에 기술된 바와 같이-ServiceHandler 어트리뷰트가 이 메서드가 메인 서비스 포트에 보내지는 메시지 타입을 처리함을 선언합니다. 이 경우, 이 핸들러가 서비스 상태를 변경하기 때문에 배타적인 액세스를 필요합니다. 그러므로, 핸들러는 ServiceHandlerBehavior.Exclusive 로 선언됩니다.

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> IncrementTickHandler(IncrementTick incrementTick)
{
    _state.Ticks++;
    LogInfo("Tick: " + _state.Ticks);
    incrementTick.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}
```

서비스는 이제 서비스 상태에 새로운 필드를 가지고 그 필드를 늘리라고 서비스에 말하는 메시지와 서비스 상태를 실제로 변경하는 핸들러를 가집니다. 다음 단계에서 상태가 변하게 됩니다.

단계 3: 메시지 내부 포스트하기

서비스는 초당 Tick 프로퍼티를 한 번 늘립니다. 이를 위해 타이머를 필요합니다. .NET 프레임워크는 타이머 메커니즘을 제공합니다. 하지만 CCR 은 포트와 리시버를 가진 CCR 시스템 내에서 타이머를 직접 사용하는 메커니즘을 직접 제공합니다.

처음 해야 하는 것은 타이머 발화마다 메시지가 매번 전송되는 포트를 선언하는 것입니다. 다음 라인은 DateTime 이 포스트되는 포트를 선언합니다. 서비스 클래스에서 _mainPort 선언 후 이 멤버 필드를 추가합니다.

```
private Port<DateTime> _timerPort = new Port<DateTime>();
```

다음 두개의 라인을 Start 메서드에 추가하십시오:

- 방금 선언한 포트에 DateTime 을 포스트합니다. DateTime 포스트 값은 비록 그것이 디버깅에 유용할 수 있지만 실행에는 중요하지 않습니다.

- _timerPort 포트를 위해 핸들러를 가동시킵니다.

_timerPort 필드가 메인 서비스 포트의 일부분이 아니기 때문에 ServiceBehavior 선언을 핸들러 메서드에 추가해 핸들러를 가동시킬 수 없습니다. Activate 메서드는 태스크의 리스트를 가지고 이를 가동시킵니다. 여기에 Arbiter.Receive(...)라고 선언된 태스크는 포트에 대한 단순한 receiver 입니다. 이번 경우 TimerHandler 메서드는 메시지가 _timerPort 포트에 도착할 때 호출됩니다. 처음 매개변수(Boolean true)는 리시버가 지속적으로 수행됨(persistent)을 나타내고, 이 포트 도착된 모든 메시지가 처리됩니다.

CCR 은 Activate()로 호출이 넘겨지는 모든 태스크를 관리합니다. CCR 은 DSS 위에 지어진 병행 태스크 디스패치 기반구조입니다.

Start()메서드는 다음과 같습니다:

```
/// <summary>
/// Service Start
/// </summary>
protected override void Start()
{
    base.Start();

    _timerPort.Post(DateTime.Now);
    Activate(Arbiter.Receive(true, _timerPort, TimerHandler));
}
```

TimerHandler 는 메시지가 _timerPort 에 도착할 때 호출되며 다음 두 가지를 수행합니다:

- TimerHandler 는 IncrementTick 메시지를 메인 서비스 포트합니다. 이것은 IncrementTickHandler 를 실행하게 합니다.
- TimerHandler 는 1000mSec 타임아웃을 가동시킵니다. 이 메서드는 리시버에 대한 핸들러로 선언되지 않고 익명(anonymous) delegate(.NET 2.0 의 신기능)를 사용합니다. 이것은 Arbiter.Receive 호출에서 핸들러를 인라인으로 작성하게 해줍니다. 리시버는 첫 매개 변수가 false 로 설정되어 지속적이지 않음(not persistent)에 주목하십시오. 이것은 TimeoutPort()메소드가 생성한 포트가 정해진 간격 (이 경우에 1000mSec)이 지나한개의 메시지를 받기 때문입니다.

Activate()메서드가 호출되면, 이것은 Arbiter.Receive 호출로 정의된 태스크를 이 서비스를 위한 활동 태스크 리스트에 추가합니다. 이경우 이미 메시지를 포스트 한 Activate 호출 내에서

정의된 조작은 현재의 스레드와 독립적으로 스케줄됩니다. 위 경우는 이미 메시지를 포스트한 후이므로 Activate 호출 직후 TimerHandler 메서드가 수행됩니다.

Arbiter.Receive()메서드는 지정된 시간(이 경우 1000mSec 이후)이 지난 후 전송된 메시지로 하여금 세 번째 매개변수로 정의된 익명 delegate 를 한번 실행합니다.

```
void TimerHandler(DateTime signal)
{
    _mainPort.Post(new IncrementTick());

    Activate(
        Arbiter.Receive(false, TimeoutPort(1000),
            delegate(DateTime time)
            {
                _timerPort.Post(time);
            }
        )
    );
}
```

단계 4: 실행하면서 서비스 관찰하기

서비스를 빌드하고 실행하십시오.

이제 http://localhost:50000/servicetutorial2 를 열어 보면, 다음과 같은 서비스 상태를 볼 수 있습니다:

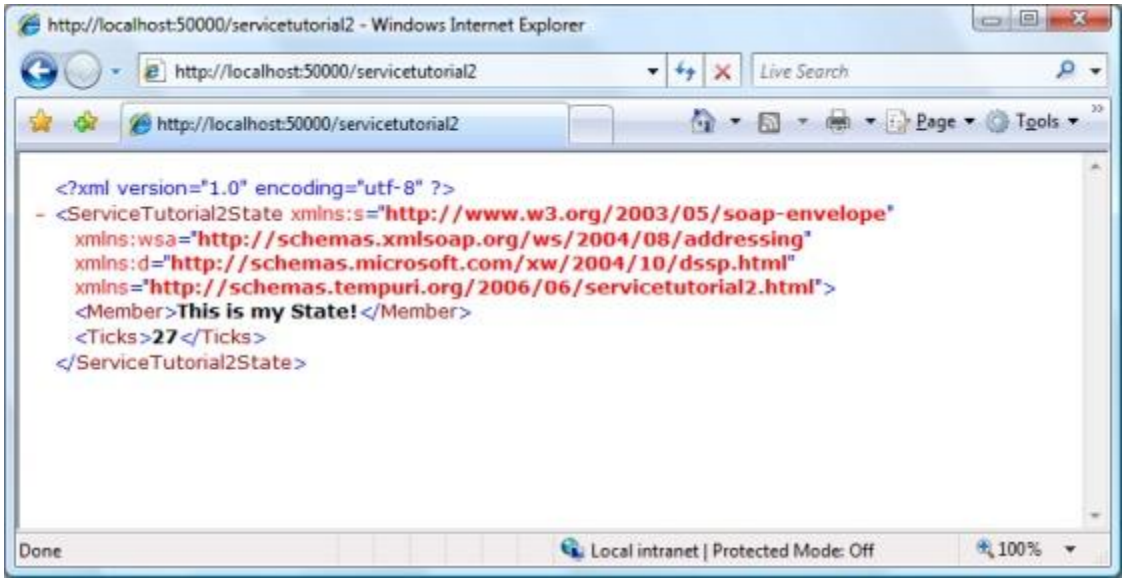


그림 1 – 브라우저에서 ServiceTutorial2 상태 보기

Ticks 프로퍼티를 살펴봅니다. 브라우저를 리프레시해보면 시간이 지남에 따라 숫자가 증가하는 것을 볼 수 있습니다.

이제 http://localhost:50000/console/output 을 열어 서비스가 보낸 로깅 메시지들을 확인해 봅니다:

Row	Time	Level	Description (mouse-over for details)
0	14:50:55	**	Validating Contract Directory Cache
1	14:50:57	**	Contract Directory is initialized
2	14:50:57	**	Attempting to load manifest: file:///C:/MSRS/samples/config/ServiceTutorial2.manifest.xml
3	14:50:58	**	Service uri: http://localhost:50000/servicetutorial2
4	14:50:57	**	Initial manifest loaded successfully.
5	14:50:59	**	Tick: 1
6	14:50:59	**	Tick: 2
7	14:51:00	**	Tick: 3
8	14:51:01	**	Tick: 4
9	14:51:02	**	Tick: 5
10	14:51:03	**	Tick: 6
11	14:51:04	**	Tick: 7
12	14:51:05	**	Tick: 8

13	14:51:06	**	Tick: 9
14	14:51:07	**	Tick: 10
15	14:51:08	**	Tick: 11
16	14:51:09	**	Tick: 12
17	14:51:10	**	Tick: 13
18	14:51:11	**	Tick: 14
19	14:51:12	**	Tick: 15
20	14:51:13	**	Tick: 16

각 행을 클릭하면 그 메시지에 대한 세부사항이 열립니다:

16	14:51:09	**	**Category**	StdOut
			Level	Info
			Time	2006-06-08T14:51:09.8709299-07:00
			Subject	Tick: 12
			Source	http://localhost:50000/servicetutorial2
			CodeSite	Boolean MoveNext()() at line:92, fileC:WMSRSWSamplesWServiceTutorialsWServiceTutorial2WServiceTutorial2.cs

DsspServiceBase 클래스는 로그의 Category, Level 과 Subject 필드를 변경하기 위해 여러 가지 로깅 메서드의 상속을 정의합니다.

서비스 튜토리얼 3 (C#) - 상태 지속하기

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 파일로부터의 초기 상태 읽어오기

파일로부터 서비스의 초기 상태를 읽기 위해, 상태 구성 멤버에 대한 선언을 서비스 구현 클래스의 추가하고 초기 상태 파일이 없을 경우를 처리해야 합니다.

ServiceTutorial3.cs 를 다음과 같이 변경합니다:

1. InitialStatePartner 어트리뷰트를 서비스 상태 선언에 추가합니다. 이것은 DSS 생성자가 상태의 초기값을 설정하게 합니다. InitialStatePartner 는 서비스 파트너의 특별한 사용입니다(파트너는 서비스 튜토리얼 4(C#)와 6(C#) 에서 좀더 상세하게 기술됩니다).
2. ServiceUri 매개 변수는 초기 상태 파일의 경로를 지정합니다. 이것은 store 디렉터리에서부터 상대적인 URI 로 표현됩니다. 이 튜토리얼의 경우, 초기 상태 파일은 store/ServiceTutorial3.xml 입니다.

고급:

서비스 매니페스트는 어떤 파트너 선언에 대해서든 런타임 상속을 제공합니다. 이것은 서비스가 다른 위치로부터 상태를 로드하고 지속하게 합니다.

상태 선언을 다음과 같이 변경하십시오:

```
/// <summary>
/// Service State
/// </summary>
[InitialStatePartner(Optional = true, ServiceUri = "ServiceTutorial3.xml")]
private ServiceTutorial3State _state = new ServiceTutorial3State();
```

초기 상태 파트너를 선언할 때 Optional = true 로 지정했기 때문에 어떠한 초기 상태 파일이 발견되지 않더라고 서비스는 시작될 것입니다. 이 경우 _state 필드는 null 일 것입니다. 다음을 start 메서드에서 base.Start() 호출 전에 디폴트 초기화를 제공하기 위해 추가합니다.

```
if (_state == null)
```

```
{
    _state = new ServiceTutorial3State();
}
```

단계 2: 상태 지속하기

현재 서비스가 시작될 때 상태 파일을 로드를 시도하지만 파일이 찾지 못합니다. 다음은 같은 위치에 상태를 지속하기 위한 방법입니다.

언제 서비스를 지속해야 하는지를 서비스의 성질에서 따라 달라질 것입니다. 이 서비스에서는 tick 이 10 의 배수가 될 때 마다 상태를 지속하기 위해 저장합니다. 현재 상태를 저장하기 위해 DsspServiceBase 클래스의 SaveState 메서드를 부릅니다. 이 호출은 마운트 서비스(파일-시스템 능력을 기능을 제공하는)에 Replace 메시지에 포스트하는 비동기 처리를 수행합니다. 상태 객체가 마운트 서비스에 포스트되면 객체는 복사됩니다. 이것은 직렬화 과정에서 서비스 상태가 훼손되는 방지하고 서비스 조작과 동시에 서비스 상태 변경이 가능하게 합니다.

다음 코드를 IncrementTickHandler 메서드에서 tick 카운터 로그 라인 이후에 추가하십시오.

```
if (_state.Ticks % 10 == 0)
{
    LogInfo("Store State");
    base.SaveState(_state);
}
```

이제 서비스는 실행한 뒤 10 초 이후부터 현재 상태가 파일로 저장되어 지속되게 됩니다. DSS 노드를 멈추고 서비스를 다시 시작하면 서비스는 마지막 상태에서부터 동작합니다.

위에 언급된 것같이 상태가 storeWServiceTutorial3.xml 파일에 저장됩니다. 서비스 잠시 동안 실행하면서 이 상태 파일의 내용을 관찰해 보십시오

```
      <?xml version="1.0" encoding="utf-8"?>
      <ServiceTutorial3State
xmlns="http://schemas.tempuri.org/2006/06/servicetutorial3.html">
        <Member>This is my State!</Member>
        <Ticks>530</Ticks>
      </ServiceTutorial3State>
```

이제 브라우저에서 http://localhost:50000/servicetutorial3 를 열고 비교해 보십시오.

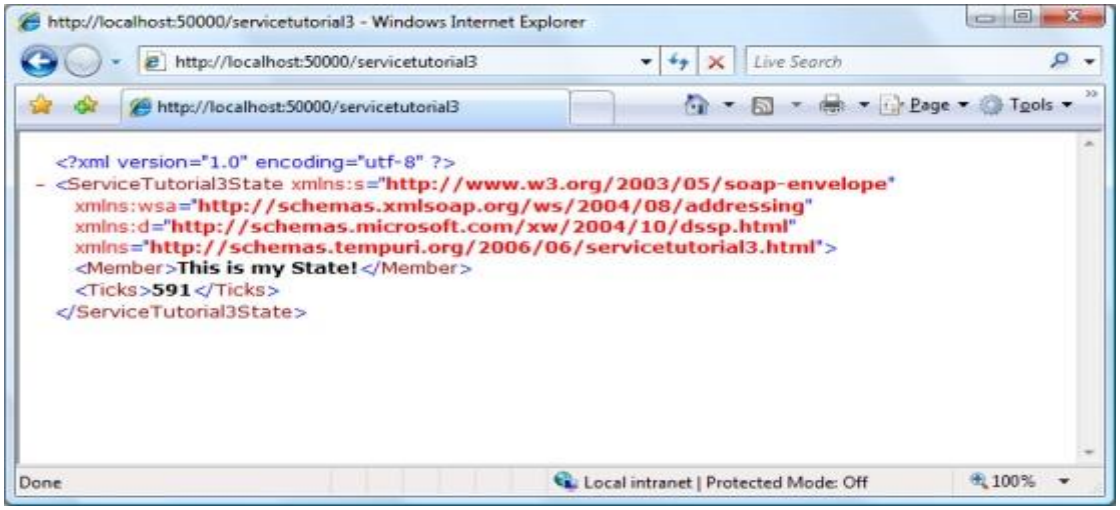


그림 1 - 브라우저에서 ServiceTutorial3 의 상태를 본 화면

파일로 직렬화된 상태는 HTTP 로 직렬화된 상태의 형식과 동일합니다. 이제 서비스를 멈추고, 그것을 다시 시작합니다. 마지막 상태 저장에서부터 Tick 카운터가 시작됩니다.

단계 3: 비동기 응답 관리하기

위와 같이 상태를 지속하는 동안 오류의 가능성을 무시했습니다. SaveState 가 성공했는지 실패했는지를 확인하도록 간단한 변경을 다음과 같이 추가합니다.

C#키워드인 yield return 과 Choice 태스크 생성을 이용해 태스크의 결과를 리턴합니다.

yield return 으로 태스크를 리턴하는 것은 핸들러가 태스크가 완료될 때까지 대기 상태로 있게 합니다. 이 기능은 여러분이 비동기 조작들의 시퀀스를 작성할 수 있게 합니다.

SaveState 메서드는 마운트 서비스에 Replace 메시지를 포스트 합니다. SaveState 메서드는 그 메시지에 대해 Response 포트를 리턴합니다.

Choice 태스크는 한 개 이상의 메시지 타입을 가진 PortSet 에 포스트된 첫번째 메시지를 처리할 핸들러를 실행합니다. 이경우 ResponsePort 는 두개의 메시지 타입을 가집니다.

- DefaultReplaceResponseType - 성공인 경우 사용
- Fault - 실패인 경우 사용

IncrementTickHandler 에서 base.SaveState(_state)를 부르는 라인을 아래 코드로 교체합니다. 여기서 핸들러는 각 메시지 타입을 위한 두 개의 익명의 delegate 입니다. 만약 실패가 보고되면, 오류를 로그합니다.

```
yield return Arbiter.Choice(
    base.SaveState(_state),
    delegate(DefaultReplaceResponseType success) { },
    delegate(W3C.Soap.Fault fault)
    {
        LogError(null, "Unable to store state", fault);
    }
);
```

이 변경의 효과를 보기 위해 storeWServiceTutorial3.xml 의 파일속성을 읽기 전용으로 변경합니다. 그 후 서비스를 실행하게 되면 SaveState 는 실패 메시지를 보내고 이 때의 오류 메시지가 로그됩니다. ConsoleOutput 을 보십시오:

Row	Time	Level	Description (mouse-over for details)
0	14:50:55	**	Validating Contract Directory Cache
1	14:50:57	**	Contract Directory is initialized
2	14:50:57	**	Attempting to load manifest: file:///C:/MSRS/samples/config/ServiceTutorial3.manifest.xml
3	14:50:58	**	Service uri: http://localhost:50000/servicetutorial3
4	14:50:57	**	Initial manifest loaded successfully.
5	14:50:59	**	Tick: 371
6	14:50:59	**	Tick: 372
7	14:51:00	**	Tick: 373
8	14:51:01	**	Tick: 374
9	14:51:02	**	Tick: 375
10	14:51:03	**	Tick: 376
11	14:51:04	**	Tick: 377
12	14:51:05	**	Tick: 378
13	14:51:06	**	Tick: 379
14	14:51:07	**	Tick: 380
15	14:51:07	**	Store State
16	14:51:07	***	Unable to store state: http://www.w3.org/2003/05/soap-envelope:Receiver -> http://schemas.microsoft.com/xw/2004/10/dssp.html:OperationFailed
17	14:51:08	**	Tick: 381
18	14:51:09	**	Tick: 382

19	14:51:10	**	Tick: 383
20	14:51:11	**	Tick: 384
21	14:51:12	**	Tick: 385
22	14:51:13	**	Tick: 386

서비스 튜토리얼 4(C#) - 서브스크립션 지원하기

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 메시지 서브스크립션 지원하기

서브스크립션을 지원하는 첫번째 부분은 Subscribe 메시지 지원입니다.

ServiceTutorial4Types.cs 파일에 Subscribe 메시지의 정의를 추가하고, 서비스에서 지원되는 메시지의 리스트에 이를 포함시킵니다. SubscribeRequestType 와 SubscribeResponseType 클래스는 Microsoft.Dss.ServiceModel.Dssp 네임스페이스에 정의된 표준 타입입니다.

Subscribe 메시지의 정의:

```
public class Subscribe : Subscribe<SubscribeRequestType, PortSet<SubscribeResponseType,
Fault>>
{
}
```

Subscribe 메시지를 서비스의 조작 포트에 포함한 코드는 다음과 같습니다:

```
/// <summary>
/// ServiceTutorial4 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial4Operations : PortSet<
    DsspDefaultLookup,
    DsspDefaultDrop,
    Get,
    HttpGet,
    Replace,
    IncrementTick,
    Subscribe>
{
}
```

주의사항:

코드를 읽기 쉽도록 조작 포트를 여러 줄로 펼쳐 두었습니다.

ServiceTutorial4.cs 파일에 있는 서비스 구현 클래스에 SubscribeHandler 메서드를 추가합니다.

이 핸들러는 현재 가입자(subscriber)의 주소만을 로그합니다. 다음 과정은 가입자의 리스트를 유지하는 방법을 설명합니다.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> SubscribeHandler(Subscribe subscribe)
{
    SubscribeRequestType request = subscribe.Body;
    LogInfo("Subscribe request from: " + request.Subscriber);

    yield break;
}
```

단계 2: 서브스크립션 매니저 사용하기

서브스크립션을 지원하기 위한 다음 과정은 가입자(subscriber)의 리스트을 유지하는 것입니다. 서브스크립션을 지원하는 것은 서비스를 위한 공통 기능이기 때문에, 가입자(subscriber)를 관리하기 위해 사용하는 표준 DSS 컴포넌트인인 서브스크립션매니저 (SubscriptionManager)가 있습니다. 따라서 서브스크립션매니저를 파트너 서비스로 포함시켜 간단히 서브스크립션을 지원할 수 있습니다.

ServiceTutorial4.cs 를 서브스크립션매니저를 추가하기 위해 여십시오. 서브스크립션매니저가 표준 서비스이기 때문에, 새로운 참조를 프로젝트에 추가할 필요가 없습니다. 타이핑을 최소화하기 위해 using 라인에사용하고 있는 라인을 submgr 를 에일리어스(alias)로 추가합니다. 다른 서비스를 사용할 때 해당 서비스 클래스를 위해 에일리어스를 사용하는 것이 일반적입니다. 예를 들면, 모든 클래스는 컨트랙트(Contract) 클래스를 가지고 대부분의 경우 Get 클래스 등을 가지고 있습니다. 서비스 튜토리얼 5(C#)에서 이를 볼 수 있습니다.

```
using submgr = Microsoft.Dss.Services.SubscriptionManager;
```

서비스 구현 클래스에 파트너 서비스를 선언하는 코드를 다음과 같이 추가하십시오.

파트너 어트리뷰트를 다음과 같이 지정합니다:

- 파트너를 위한 고유이름

- 파트너로 사용하는 서비스의 컨트랙트(Contract)
- 생성 정책(creation policy)

이 튜토리얼을 위해, 컨트랙트 이름을 http://schemas.microsoft.com/xw/01/2005/subscriptionmanager.html 로 가지는 파트너(SubMgr)를 생성합니다. 서비스가 시작되면 이 파트너의 새로운 인스턴스가 생성됩니다. 만일 생성이 실패하면, 서비스 생성도 실패하며 Start 메서드는 호출되지 않을 것입니다.

서비스 구현 클래스에 _mainPort 를 정의하는 라인 이후 다음 정의를 추가하십시오:

```
[Partner("SubMgr", Contract = submgr.Contract.Identifier, CreationPolicy = PartnerCreationPolicy.CreateAlways)]
private submgr.SubscriptionManagerPort _submgrPort = new submgr.SubscriptionManagerPort();
```

여기에서 파트너(Partner) 어트리뷰트는 _submgrPort 필드에 적용됩니다. 이 _submgrPort 필드는 서브스크립션매니저의 메인 서비스 포트로 선언됩니다. 이 포트에서 어떤 메시지 포스트하면 이 서비스에서 생성한 서브스크립션매니저 인스턴스에 전송됩니다.

단계 3: 가입자(Subscriber) 리스트 유지하기

파트너로 서브스크립션매니저를 가지기 때문에 여기에 메시지를 보낼 수 있습니다. 가입자(subscriber)의 리스트를 유지하면서 시작하십시오. DsspServiceBase 는 메서드가 여러분이 필요한 것들을제공합니다. SubscribeHandler 에서 yield break 바로 전에 다음 코드를 추가하십시오.

```
SubscribeHelper(_submgrPort, request, subscribe.ResponsePort);
```

이 메서드는 서브스크립션매니저에 Insert 메시지를 보냅니다. 그 후 서브스크립션매니저는 서브스크립션 리스트를 관리하는 책임을 가집니다.

단계 4: 알림(Notification) 전송하기

서브스크립션을 지원하는 마지막 부분은 가입자(subscriber)에게 알림을 보내는 것 있습니다. 이 서비스가 서브스크립션매니저를 사용하기 때문에, 해야 할 일은 단지 서브스크립션매니저에게 상태 변경을 포스트하는 것뿐입니다. 서브스크립션매니저는 이 서비스에 서브스크립션한 파트너(가입자)에게 알림을 보냅니다.

상태가 변하는 두 개의 위치는 IncrementTickHandler 와 ReplaceHandler 메서드입니다.

상태가 전적으로 교체될 때, 가입자들에게 이 새로운 서비스 상태를 알림으로 보냅니다. DsspServiceBase 는 헬퍼 함수인 SendNotification 을 제공합니다.

ReplaceHandler 에서 _state 를 변경하는 라인 이후 다음 라인을 추가하십시오:

```
base.SendNotification(_submgrPort, replace);
```

마찬가지로, 상태가 IncrementTick 메시지를 처리하는 서비스에 의해 업데이트 될 때, 가입자에게 이 이벤트의 알림을 전송합니다.

IncrementTickHandler 에서 LogInfo 호출 이후 다음 라인을 추가하십시오:

```
base.SendNotification(_submgrPort, incrementTick);
```

서브스크립션 모델은 대칭성을 가집니다. 서비스는 어떤 상태라도 메시지로 변경하여 가입자에게 알릴 수 있습니다. 따라서 이 튜토리얼 서비스의 경우, 서비스 상태를 변경하는 유일한 두 개의 메시지는 Replace 와 IncrementTick 입니다. 서비스 튜토리얼 5(C#)는 이 서비스(서비스 튜토리얼 4)에 서브스크립션하는 방법을 보여줍니다.

단계 5: 새 가입자(Subscriber)와 동기화하기

IncrementTick 업데이트는 현재의 tick 카운트를 포함하지 않습니다. 단지 Tick 카운트가 1 씩 증가되는 시맨틱 컨텐츠를 전송할 뿐입니다. 이 때문에 가입자들은 tick 카운트를 알 수 없으며, 단지 그들의 서브스크립션한 이후 발생한 tick 의 개수만을 알 수 있습니다. 필요하다면 현재의 tick 카운트를 새로운 가입자에게 Replace 알림으로 전송할 수 있습니다.

새로운 가입자가 성공적으로 추가하게 될 시간을 찾는 SubscribeHandler 메서드를 다음과 같이 변경합니다.

SubscribeHelper 메서드는 서브스크립션매니저 서비스에 Insert 메시지를 보냅니다. 이것은 비동기식 조작입니다.

SubscribeHelper 는 SuccessFailurePort 를 리턴합니다. 이것은 두개의 메시지 타입인 SuccessResult 와 Exception 을 처리할 수 있는 PortSet 입니다. 이 결과를 처리하기 위해, 우리는 yield return 을 사용하는 Choice 태스크를 리턴합니다. 그 Choice 태스크는 그 다음 두 개의 delegate 를 가집니다: SuccessResult 를 처리하는 delegate 와 Exception 을 처리하는 delegate.

SuccessResult 메시지가 리턴 될 때, 서브스크립션이 성공적으로 성립되었다는 것을 알게 됩니다. 그리고 가입자에게 서비스(Replace 알림으로)의 현재 상태를 전송합니다. 만일 Exception 메시지가 리턴 되면, 이 오류를 로그합니다.

주의사항:

아래와 같이 사용된 SendNotification 메서드 호출에서, 보통의 두 개가 아닌 세 개의 매개변수가 사용되었습니다. 이 경우, 두 번째 매개 변수는 가입자의 주소입니다. 이것을 이용해 서브스크립션매니저가 특정 가입자에게 알림을 보내게 할 수 있습니다. 현재 코드에서 생성된 Replace 메시지가 없으므로, 이 호출은 SendNotification 에 대한 상속을 사용합니다. 이것은 어떤 타입의 메시지를 알람으로 전송할 지를 나타내는 제네릭 타입-매개 변수(이 경우, Replace)를 메시지의 보디로 가짐으로 구현되어집니다.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> SubscribeHandler(Subscribe subscribe)
{
    SubscribeRequestType request = subscribe.Body;
    LogInfo("Subscribe request from: " + request.Subscriber);

    yield return Arbiter.Choice(
        SubscribeHelper(_submgrPort, request, subscribe.ResponsePort),
        delegate(SuccessResult success)
        {
            base.SendNotification<Replace>(_submgrPort, request.Subscriber, _state);
        },
        delegate(Exception e)
        {
            LogError(null, "Subscribe failed", e);
        }
    );

    yield break;
}
```

부록 A: Subscription Manager Message Flow

다음은 ServiceTutorial5 가 ServiceTutorial4 에 서브스크립션할 때의 메시지 흐름을 표시합니다.

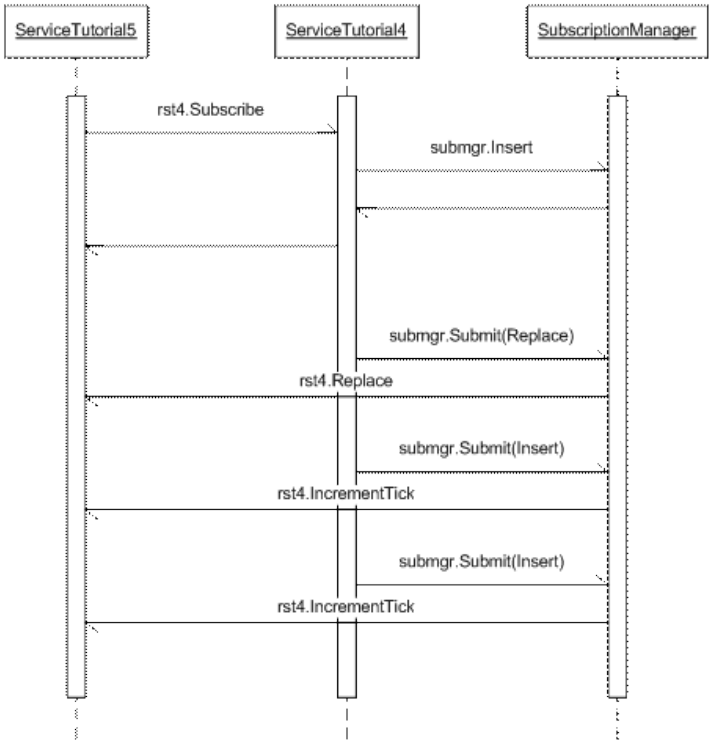


그림 1 – 두 개 서비스 사이의 서브스크립션 메시지 흐름도

서비스 튜토리얼 5(C#) - 서브스크립션하기(Subscribing)

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 서비스 프록시(Service Proxy)를 참조(Reference)에 추가하기

ServiceTutorial4 와 파트너 관계를 확립하고, 그 다음 그 서비스에 서브스크립션합니다. 첫번째 단계로 파트너 서비스를 참조에 추가합니다. 이를 위해 프럭시에 대해 조금 알아볼 필요가 있습니다.

서비스가 빌드될 때, 다음 세 개의 어셈블리가 만들어집니다.

- 메인 서비스 (또는 구현) 어셈블리(assembly)
  예, ServiceTutorial5.Y2006.M06.dll--서비스의 기능을 포함합니다.
- 프록시 어셈블리
  예, ServiceTutorial5.Y2006.M06.proxy.dll--서비스의 모든 퍼블릭 인터페이스를 위한 사항들을(stubs) 포함합니다.
- 변환(transform) 어셈블리
  예, ServiceTutorial5.Y2006.M06.transform.dll --타입이 프록시와 서비스 어셈블리 사이에서 변환 되게 하는 코드를 포함합니다.

다른 서비스에 메시지를 보내기 위해, 구현 어셈블리를 직접 사용하는 것보다 서비스의 프록시 어셈블리를 참조해 사용하십시오. 이것은 여러 이유에서 비롯 됩니다: 해당 서비스를 다른 노드에서 실행할 때, 로컬 머신에는 구현 어셈블리의 복사본을 가지고 있지 않을지도 모르며, 파트너 서비스는 서비스의 문맥 내에서 실행하는 코드를 가지지 않으면서도, 프록시를 사용하는 것에 의해 다른 노드에서 자연스럽게 실행됩니다.

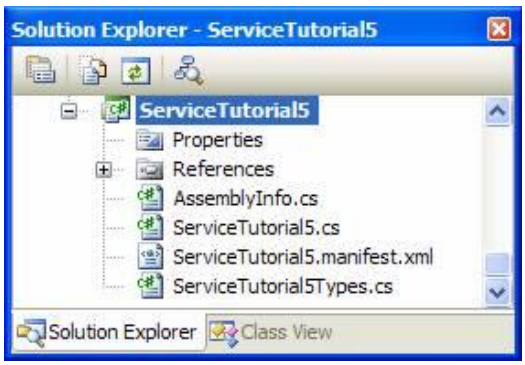


그림 1-솔루션 탐색기에서 프록시 어셈블리를 참조함

ServiceTutorial4 를 파트너로 사용하기 위해, 프로젝트에 어셈블리(ServiceTutorial4.Y2006.M06.proxy.dll)를 참조에 추가해야 합니다. 마이크로소프트 비주얼 스투디오에서 프로젝트> 참조 추가 메뉴를 선택하거나 솔루션 탐색기의 프로젝트를 클릭하여 참조 추가를 팝업메뉴에서 선택합니다.

이 후 참조 추가 대화 상자가 나타나며, 찾아보기(Browse) 탭을 선택하고, 마이크로소프트 Robotics Studio 의 설치 디렉터리 아래에서 빈 디렉터리로 이동하십시오. ServiceTutorial4.Y2006.M06.proxy.dll 파일 찾아 선택하십시오.

주의사항:

ServiceTutorial4.Y2006.M06.proxy.dll 파일은 ServiceTutorial4 프로젝트를 위한 프록시입니다. 만약 여러분이 이 튜토리얼을 따라서 작업하고 있고, DssnewService 도구를 사용하여 별도 프로젝트를 만든 경우에는 해당 프로젝트 디렉토리에서 프록시를 찾으면 됩니다. 이때 프록시의 이름은 다를 수 있습니다.

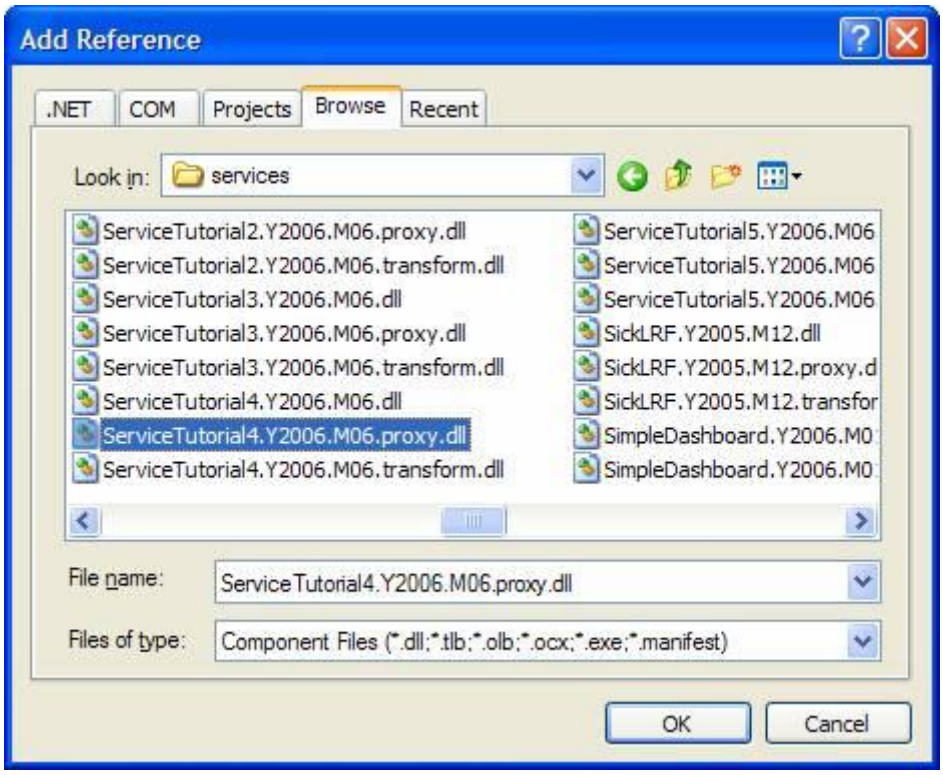


그림 2- bin 디렉터리에서 프록시 어셈블리를 찾아보기(Browse)

솔루션 탐색기의 참조 리스트에서 참조된 어셈블리를 선택하십시오. 참조 속성 창에서 로컬복사(Copy Local)와 특정 버전(Specific Version) 필드를 False 로 설정하십시오.

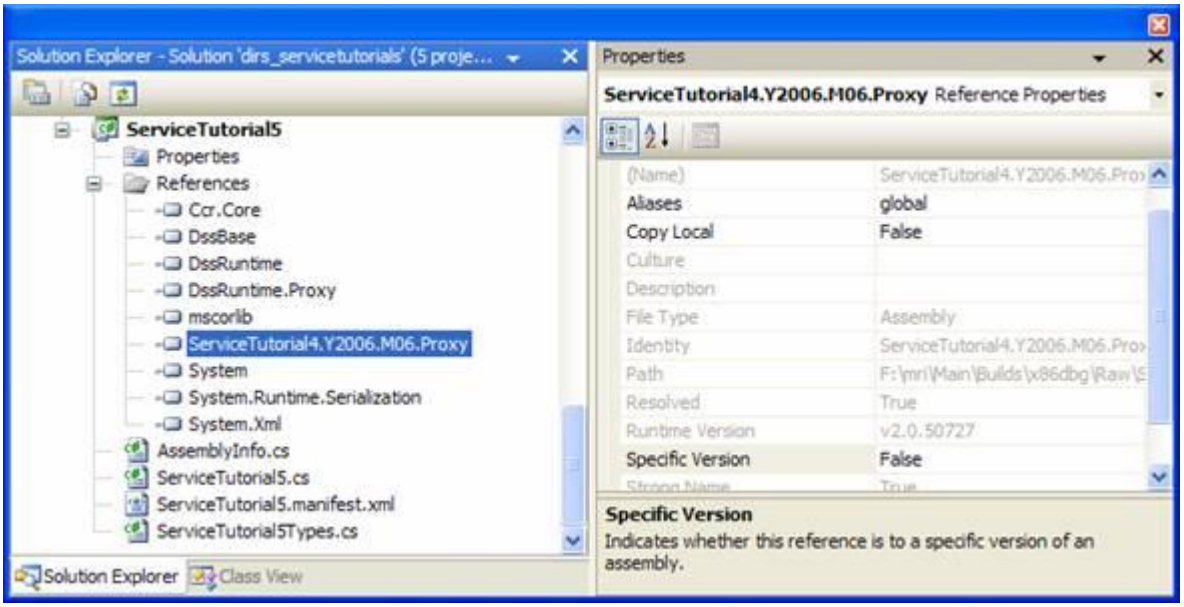


그림 3-참조 속성 창에서 프록시 참조 속성 설정

같은 방법으로 DssRuntime.Proxy.dll 를 참조해 사용하십시오. 이 DLL 은 또한 bin 디렉터리에 있습니다.

단계 2: 파트너 서비스에 서브스크립션하기

ServiceTutorial5.cs 을 다음과 같이 변경합니다.

서비스 튜토리얼 4 프록시의 에일리어스를 다음과 같이 추가하십시오. 프럭시는 구현 어셈블리 안의 서비스 네임스페이스로부터 그 네임스페이스를 도출합니다. Proxy 접미부는 그 네임스페이스에 추가하게 되고, 프록시에 할당됩니다.

```
using rst4 = Robotics.ServiceTutorial4.Proxy
```

서비스 튜토리얼 4 에서 서브스크립션매니저 파트너를 추가하는 것과 같은 방법으로 파트너를 추가합니다. 서비스 튜토리얼 4 의 조작포트인 rst4.ServiceTutorial4Operations 타입으로 _clockNotify 필드를 생성합니다. 이 필드는 서브스크립션을 만들었을 때 이 서비스가 알림을 수신하는 포트입니다.

```
[Partner("Clock", Contract = rst4.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.UseExistingOrCreate)]
rst4.ServiceTutorial4Operations _clockPort = new rst4.ServiceTutorial4Operations();
rst4.ServiceTutorial4Operations _clockNotify = new rst4.ServiceTutorial4Operations();
```

이제 파트너 서비스에 다음과 같이 서브스크립션합니다.

서비스 튜토리얼 4(C#)를 상기해 보면, 서비스가 지원할 메시지인 Subscribe 를 생성했었습니다. 프록시가 생성될 때, 지원할 각 메시지에 대한 유틸리티 함수들이 생성됩니다. 이 튜토리얼의 경우는 서비스가 Subscribe 메시지에 대한 유틸리티 함수를 호출합니다. 서브스크립션이 알림을 보낼 알림 포트를 매개 변수로 서브스크립션 유틸리티 함수를 호출합니다.

Start 메서드 끝에 다음 코드를 추가하십시오.

```
_clockPort.Subscribe(_clockNotify);
```

이 서비스를 빌드하고 실행하면, http://localhost: 50000/console/output 로그에서 다음 메시지를 봅니다.

```
      .
      .
      .
      Starting                              manifest                              load:
file:///C:/MSRS/samples/config/ServiceTutorial5.manifest.xml
      Manifest load complete
      Service uri: http://localhost:50000/servicetutorial5
      Service uri: http://localhost:50000/servicetutorial4
      Subscribe                              request                              from:
http://localhost:50000/servicetutorial5/NotificationTarget/7cf955e7-73ff-47cd-bdc0-
d3a3a8251c49
      Tick: 21
      Tick: 22
      Tick: 23
      Tick: 24
```

이 로그에서 주목해 볼 것: ServiceTutorial4 와 ServiceTutorial5 서비스가 실행됨과, 서브스크립션 요청이 처리 되었다는 로그 메시지. 여기에 넘어가는 URI 는 알림 포트의

주소입니다. 이것은 _clockNotify 포트에 알림을 보내기 위해 DSS 기반 구조 내에서 내부적으로 사용됩니다.

단계 3: 알림 처리하기

서비스에 서브스크립션하는 목적은 그 서비스에서 전송 되는 알림을 받기 위함입니다. ServiceTutorial4 서비스가 IncrementTick 와 Replace 메시지에 대해 알림을 전송하는 것을 상기해 보십시오. 서비스 튜토리얼 4 의 상대편인 이 서비스 튜토리얼 5 는 다음 단계로 그 알림을 수신하여 처리합니다.

알림을 수신할 핸들러를 추가합니다. 알림은 이 서비스의 메인 서비스 포트로 보내지지 않으므로, ServiceHandler 어트리뷰트를 사용할 수 없습니다. 대신 다음과 같이 여러분이 처리할 각 알림 메시지를 위해 리시버(receiver) 태스크를 가동합니다. 이것은 Start 메서드에 추가됩니다.

```
protected override void Start()
{
    base.Start();

    Activate<ITask>(
        Arbiter.Receive<rst4.IncrementTick>(true, _clockNotify, NotifyTickHandler),
        Arbiter.Receive<rst4.Replace>(true, _clockNotify, NotifyReplaceHandler)
    );
    _clockPort.Subscribe(_clockNotify);
}
```

두개의 리시버는 다음을 명기합니다:

- 처리하는 메시지 타입:
    - rst4.IncrementTick
    - rst4.Replace
- 메시지가 도착하는 포트:
    - _clockNotify
- 메시지를 위한 핸들러 메서드.

다음으로, 이 메시지 타입 각각을 위한 핸들러를 구현합니다.

ServiceTutorial5 의 메인 서비스 구현에 클래스에 다음 메서드를 추가합니다.

```
private void NotifyTickHandler(rst4.IncrementTick incrementTick)
{
    LogInfo("Got Tick");
}

private void NotifyReplaceHandler(rst4.Replace replace)
{
    LogInfo("Tick Count: " + replace.Body.Ticks);
}
```

주의사항:

이 핸들러는 처리할 메시지에 응답을 전송하지 않습니다. 이 핸들러는 응답을 기다리지 않는 알림 메시지를 처리합니다.

이제 이 서비스를 실행하면 여러분은 콘솔에서 모든 Tick: [틱수] 메시지와 Subscribe request from: [알림 포트] 뒤에 하나의 Tick Count : [틱수] 메시지 뒤에 Got Tick 메시지를 봅니다.

단계 4: 상태 동기화하기

ServiceTutorial5 서비스는 이제 서비스 튜토리얼 4(C#)를 파트너로 하여 그 상태를 동기화하기 위한 모든 정보를 가지게 되었습니다.

상태를 동기화 하기 위해 ServiceTutorial5Types.cs 를 다음과 같이 변경합니다.

Tick 카운트를 서비스 상태에 포함시킵니다. 서비스 튜토리얼 4 와 구별되게 하기 위해 Member 프로퍼티를 삭제하고, TickCount 라고 불리는 정수 프로퍼티를 만드십시오.

```
private int _tickCount;

[DataMember]
public int TickCount
{
    get { return _tickCount; }
    set { _tickCount = value; }
}
```

이제 지원하는 조작 포트를 변경하십시오:

- ServiceTutorial4 에서 정의된 IncrementTick 메시지
- 이 서비스를 위한 새 메시지인 SetTickCount 를 추가하기

```
/// <summary>
/// ServiceTutorial5 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial5Operations : PortSet<
    DsspDefaultLookup,
    DsspDefaultDrop,
    Get,
    HttpGet,
    Replace,
    rst4.IncrementTick,
    SetTickCount>
{
}
```

단계 2 의 코드와 같은 방법으로 using 지시자를 사용해 파트너 에일리어스를 ServiceTutorial5Types.cs 에 추가합니다.

ServiceTutorial5Types.cs 에 SetTickCount 메시지와 그 보디 타입인 SetTickCountRequest 를 다음 코드와 같이 선언합니다.

```
public class SetTickCount : Update<SetTickCountRequest,
PortSet<DefaultUpdateResponseType, Fault>>
{
    public SetTickCount()
    {
    }

    public SetTickCount(int tickCount)
        : base(new SetTickCountRequest(tickCount))
```

```
    {
    }
}

[DataContract]
[DataMemberConstructor]
public class SetTickCountRequest
{
    public SetTickCountRequest()
    {
    }

    public SetTickCountRequest(int tickCount)
    {
        _tickCount = tickCount;
    }

    private int _tickCount;

    [DataMember]
    public int TickCount
    {
        get { return _tickCount;}
        set { _tickCount = value;}
    }
}
```

[DataMemberConstructor] 어트리뷰트 [DataContract]와 같이 사용되어 생성된는 프록시에서 [DataMember] 구성 요소의 각각을 생성하는 초기화 생성자를 지정합니다. 즉 [DataMemberConstructor]가 SetTickCountRequest 타입에서 삭제되면 프록시 타입에서 단지 디폴트 SetTickCountRequest() 생성자만을 가지게 됩니다..

ServiceTutorial5.cs 의 서비스 구현 클래스에 있는 이 두 개의 메시지에 대한 핸들러를 추가합니다. 이들 메시지는 상태를 변경하기 때문에 ServiceHandlerBehavior.Exclusive 로 선언됩니다.

```
ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> IncrementTickHandler(rst4.IncrementTick incrementTick)
{
    _state.TickCount++;
    incrementTick.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> SetTickCountHandler(SetTickCount setTickCount)
{
    _state.TickCount = setTickCount.Body.TickCount;
    setTickCount.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}
```

마지막으로, 알림 핸들러로부터 메인 서비스 포트로 적합한 메시지를 전송하여 이 메시지 핸들러에 알림 핸들러를 연결합니다.

NotifyTickHandler 와 NotifyReplaceHandler 메서드는 다음과 같습니다:

```
private void NotifyTickHandler(rst4.IncrementTick incrementTick)
{
    LogInfo("Got Tick");
    _mainPort.Post(new rst4.IncrementTick(incrementTick.Body));
}

private void NotifyReplaceHandler(rst4.Replace replace)
{
    LogInfo("Tick Count: " + replace.Body.Ticks);
    _mainPort.Post(new SetTickCount(replace.Body.Ticks));
}
```

주의사항:

NotifyTickHandler 메서드에서 IncrementTick 타입의 메시지가 처리되고 메인 서비스 포트에 전송됩니다. IncrementTick 인수가 단지 직접 메인 포트로 보내어지지 않는 이유는 ResponsePort 가 정확하게 생성됨을 보증하는 새로운 메시지가 만들어질 필요가 있기 때문입니다.

고급:

NotifyReplaceHandler 메서드에서 rst4.Replace 메시지가 처리되고 SetTickCount 메시지가 메인 포트에 전송됩니다. 새로운 메시지 타입이 필요한 이유는 rst4.Replace 메시지가 DSSP 동작 중 하나인 Replace 를 사용하기 때문입니다. 이것은 서비스의 상태를 교체(replace)하는 ServiceTutorial4 에서 적합합니다. 그러나 개념적으로 서비스 상태 중 한 개의 구성 요소를 변경하는 ServiceTutorial5 에서는 부적절합니다. 이 이유 때문에 SetTickCount 메시지는 Update 를 사용하여 정의되고 상태를 변경하기 위해 사용됩니다.

서비스 튜토리얼 6 (C#) - 상태 검색(Retrieving)과 XML 변환(Transform)으로 보여주기

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 다른 서비스 상태를 가져오기

ServiceTutorial6Types.cs 의 ServiceTutorial6State 클래스에 Clock 프로퍼티와 InitialTicks 를 추가하십시오. 이 포로퍼티는 다른 서비스의 상태를 가져오는 방법을 보여주는데 사용됩니다.

```
private string _clock;

[DataMember]
public string Clock
{
    get { return _clock; }
    set { _clock = value; }
}

private int _initialTicks;

[DataMember]
public int InitialTicks
{
    get { return _initialTicks; }
    set { _initialTicks = value; }
}
```

ServiceTutorial6.cs 파일안의 Start 메서드에서 _clockPort.Subscribe 를 호출하는 라인을 제거하고, 다음 코드로 교체합니다. 이것은 이터레이터 메서드인 OnStartup 이 서비스의 실행과 동시에 실행되게 합니다.

```
SpawnIterator(OnStartup)
```

이제 OnStartup 메서드를 작성합니다. 이 메서드가 하는 첫번째 일은 ServiceTutorial4 파트너 서비스에 Get 메시지를 전송하는 것입니다. 이 메시지를 전송하기 위해, _clockPort 필드의

Get 메서드를 호출합니다. 이것은 Get 메시지를 생성하고 서비스에 보냅니다. 이터레이터는 그 다음 yield return 을 사용하는 Choice 태스크를 리턴합니다. 이때 ServiceTutorial4 의 Get 메시지에 의해 정의된 두 개의 가능한 응답 메시지 타입이 있습니다. 일반적으로 Get 메시지는 해당 서비스의 상태를 리턴하거나 Fault 를 리턴합니다.

여기에서는, 성공적인 경우 서비스의 상태는 로컬 변수 상태에 저장되고 실패(Fault)인 경우는 오류 경로에 로그됩니다.

주의사항:

다음 함수는 파일 맨 위에 선언된 W3C.Soap 을 사용합니다.

```
{
    rst4.ServiceTutorial4State state = null;

    yield return Arbiter.Choice(
        _clockPort.Get(GetRequestType.Instance),
        delegate(rst4.ServiceTutorial4State response)
        {
            state = response;
        },
        delegate(Fault fault)
        {
            LogError(null, "Unable to Get state from ServiceTutorial4", fault);
        }
    );
```

만일 성공적인 경우이면, state 는 null 이 아닐 것입니다. 이 경우 코드는 현재 상태를 위한 새로운 ServiceTutorial6State 를 만들고 ServiceTutorial4 서비스의 상태에 있는 Ticks 의 현재 값으로 InitialTicks 프로퍼티를 설정합니다. 코드는 또한 Clock 파트너를 위한 파트너 정의를 찾고, 파트너의 서비스 주소로 Clock 프로퍼티를 설정합니다.

그 다음 replace 메시지가 이 서비스의 메인 포트에 새로운 상태 값을 설정하기 위해 보내집니다.

```
if (state != null)
    {
        ServiceTutorial6State initState = new ServiceTutorial6State();
        initState.InitialTicks = state.Ticks;
```

```
        PartnerType partner = FindPartner("Clock");
        if (partner != null)
        {
            initState.Clock = partner.Service;
        }

        Replace replace = new Replace();
        replace.Body = initState;

        _mainPort.Post(replace);
    }
```

ServiceTutorial4 서비스의 상태를 얻은 후, OnStartup 메서드는 ServiceTutorial4 서비스에 Subscribe 메시지를 보내고 실패(Fault)를 로그합니다.

```
yield return Arbiter.Choice(
        _clockPort.Subscribe(_clockNotify),
        delegate(SubscribeResponseType response) { },
        delegate(Fault fault)
        {
            LogError(null, "Unable to subscribe to ServiceTutorial4", fault);
        }
    );
}
```

단계 2: 파트너 서비스 상태에 XSLT 를 사용하기

서비스 튜토리얼 1(C#)에서 서비스 상태를 웹브라우저에서 XML 표현으로 표시하는 방법을 배웠습니다. 상태는 사용자에게 보다 잘 보여지기 위해 HTML(또는 텍스트 또는 다른 XML 표현)로 변형될 수 있습니다. 이를 위해 다음 세 개의 단계가 필요합니다:

1. 상태를 변형시키는 방법을 기술하는 XSLT 파일을 작성합니다. 이것은 이 튜토리얼의 범위에서 벗어납니다. http://msdn.microsoft.com 에서 XSLT 작성에 관한 정보를 찾아 볼 수 있으며, 이 튜토리얼의 부록에 XSLT 의 예제가 있습니다.
2. 서비스 어셈블리의 포함 리스소(embedded resource)로 XSLT 를 추가합니다.
3. 상태의 직렬화된 XML 표현인 XSLT 에 링크를 제공합니다.

첫번째로 마운트 서비스(Mount Service)를 마이크로소프트 Robotics Studio 설치 디렉터리의 storeWtransforms 서브디렉터리에 있는 XSLT stylesheet 를 참조하는데 사용하는 방법을 보여 줄 것입니다.

ServiceTutorial6.cs 파일에는 _transform 이라고 불리는 스트링(string) 타입을 선언하고, 그것을 마운트 서비스를 통하여 XSLT 파일에 상대 경로로 지정합니다.

```
string _transform = "/mountpoint/store/transforms/ServiceTutorial6.xslt";
```

주의사항:

DSS 노드가 시작되기 전까지 DssHost 포트 번호을 알 수 없습니다. 따라서 변환(transform)을 위한 URI 는 런타임에 생성되거나 상대 경로로 지정되어야 합니다.

HttpGetHandler 메서드안의 HttpResponseType 생성자에 XSLT 경로를 넣어줍니다.

```
httpGet.ResponsePort.Post(new HttpResponseType(System.Net.HttpStatusCode.OK, _state, _transform));
```

ServiceTutorial6.xslt 파일을 storeWtransforms 디렉터리에 복사한 다음 서비스를 빌드하고 실행합니다. 이후 다음과 같은 페이지를 볼 것입니다.

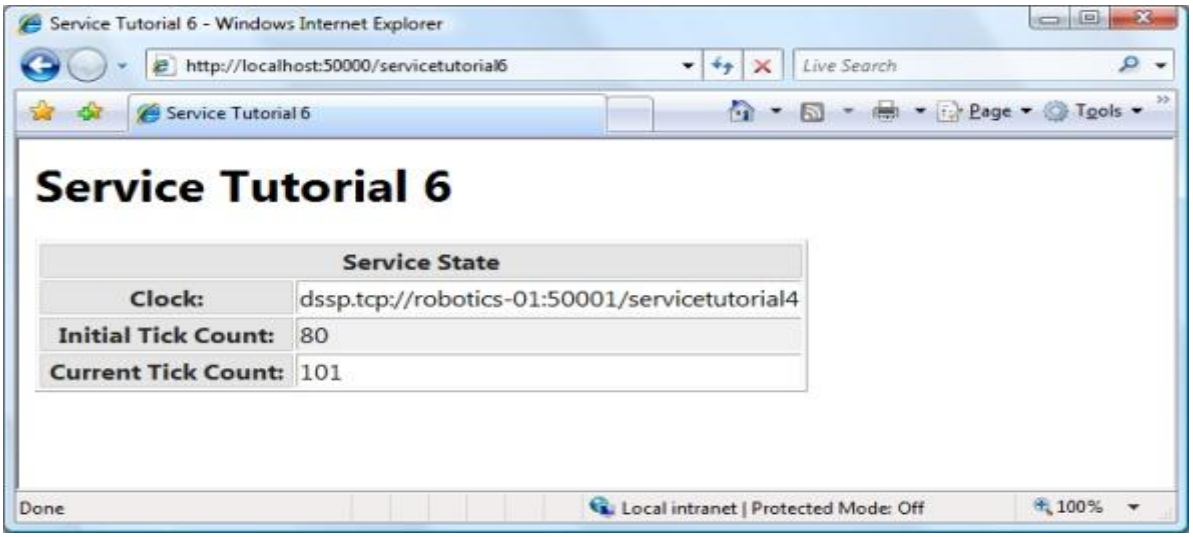


그림 1 - 브라우저에서 XSLT stylesheet 이 서비스의 상태에 적용된 화면

중요:

인터넷 익스플로러에서 XSLT 를 디버그할 때 변경된 것을 보기 위해 새로운 브라우저 윈도 또는 탭을 열어 확인합니다. IE 의 캐싱은 Ctrl+F5 를 누르는 경우에도 때때로 페이지를 새로 고치지 못합니다. 새 윈도우를 여는 것은 항상 페이지를 다시 로드합니다.

단계 3: 포함 리소스(Embedded Resources) 사용하기

XSLT 파일을 store₩transforms 디렉터리에 카피하는 것이 항상 적합하지는 않습니다. 이는 서비스를 배포하는 데 복잡도를 증가시킵니다.

마이크로소프트 Robotics Studio 는 어셈블리에 리소스를 포함시키는 것을 지원합니다. XSLT 파일을 포함시키기 위해 비주얼 스투디오의 솔루션 탐색기에서 해당 XSLT 파일을 선택합니다. 아래의 속성 창에서 빌드 작업 항목을 '내용' 에서 '포함 리소스' 로 변경합니다. 리소스는 파일의 디폴트 프로젝트 네임스페이스와 파일 이름의 접합으로 이름 지어집니다. 이 경우에는 Robotics.ServiceTutorial6.ServiceTutorial6.xslt 가 됩니다.

ServiceTutorial6.cs 파일에 EmbeddedResource 어트리뷰트와 XSLT 파일 경로를 추가합니다.

```
[EmbeddedResource("Robotics.ServiceTutorial6.ServiceTutorial6.xslt")]
string _transform = null;
```

_transform 필드의 값은 런타임에 서비스 시작에서 결정됩니다.

주의사항:

네임스페이스 충돌을 피하기 위해 마이크로소프트 Robotics Studio 로 설치한 서비스 튜토리얼 6 은 위에서 보여진 네임스페이스와 다른 네임스페이스를 가집니다. 여러분 프로젝트의 디폴트 네임스페이스을 보려면 프로젝트 속성에서 응용 프로그램 탭에 있는 기본 네임스페이스를 보십시오.

단계 4: DSS XSLT 템플릿(Template) 사용하기

위 그림에 있는 상태 표현이 다른 DSS 서비스의 공통 배치를 가지고 있지 않는 것을 알 수 있습니다. 마이크로소프트 Robotics Studio 는 모든 DSS 서비스를 위해 공통 XSLT 템플릿을 일관되게 사용합니다. 여러분의 서비스가 디폴트 DSS XSLT 템플릿을 사용하기 위한 방법은 Tutorial for Using Default DSS XSLT Template 에 있습니다.

주의사항:

위의 것은 고급 단계입니다. 따라서 위 단계를 건너 뛰어 다음 튜토리얼로 이동해도 좋습니다. 그러나 제품을 위한 서비스를 개발할 때에는 위의 단계에 지정된 가이드라인을 따르기를 권장합니다.

부록 A: ServiceTutorial6.xslt

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:rst6="http://schemas.tempuri.org/2006/06/servicetutorial6.html">

  <xsl:output method="html"/>

  <xsl:template match="/rst6:ServiceTutorial6State">
    <html>
      <head>
        <title>Service Tutorial 6</title>
        <link                        rel="stylesheet"                        type="text/css"
href="/resources/dss/Microsoft.Dss.Runtime.Home.Styles.Common.css" />
      </head>
      <body style="margin:10px">
        <h1>Service Tutorial 6</h1>
        <table border="1">
          <tr class="odd">
            <th colspan="2">Service State</th>
          </tr>
          <tr class="even">
            <th>Clock:</th>
            <td>
              <xsl:value-of select="rst6:Clock"/>
            </td>
          </tr>
          <tr class="odd">
            <th>Initial Tick Count:</th>
            <td>
              <xsl:value-of select="rst6:InitialTicks"/>
            </td>
```

```
          </tr>
          <tr class="even">
            <th>Current Tick Count:</th>
            <td>
              <xsl:value-of select="rst6:TickCount"/>
            </td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

서비스 튜토리얼 7 (C#) - 고급 주제

마이크로소프트 Robotics Studio 를 사용하여 애플리케이션을 만드는 것은 서비스들 사이에서 입출력을 오케스트레이션하는 단순한 작업입니다. 서비스는 소프트웨어 또는 하드웨어의 인터페이스를 나타내며 특정 기능을 수행하는 프로세스들 사이에서 의사소통을 가능하게 합니다.

단계 1: 다른 Node 에서 실행 중인 서비스에 서브스크립션하기

서비스 튜토리얼 5(C#)에서 만들어져 서비스 튜토리얼 6(C#)에서 변경된 서비스에서 파트너는 "Clock"이란 이름으로 정의되어져 있습니다. 이파트너는 다음 프로퍼티로 선언됩니다:

- 이름 Clock
- 서비스 튜토리얼 4(C#)에서 생성된 서비스 컨트랙트
- UseExistingOrCreate 의 파트너 생성 정책

서비스 튜토리얼 5 또는 6 이 시작될 때, 이 파트너 선언은 서비스 튜토리얼 4 가 현재 노드에서 이미 실행되고 있지 않을 때 이를 실행하게 합니다.

매니페스트 파일은 서비스가 가지고 있는 파트너를 구성할 수 있습니다. 다음 매니페스트는 현재 노드가 아닌 다른 노드에서 파트너 서비스를 찾는 방법을 보여줍니다.

```
<?xml version="1.0" ?>
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns:st6="http://schemas.tempuri.org/2006/06/servicetutorial6.html">
  <CreateServiceList>
    <!-- Service Tutorial 6 -->
    <ServiceRecordType>
      <d:Contract>http://schemas.tempuri.org/2006/06/servicetutorial6.html</d:Contract>
      <d:PartnerList>
        <d:Partner>
          <d:Service>http://localhost:40000/servicetutorial4</d:Service>
          <d:Name>st6:Clock</d:Name>
        </d:Partner>
      </d:PartnerList>
    </ServiceRecordType>
  </CreateServiceList>
```

```
</Manifest>
```

이를 보여주기 위해 다음의 두 노드에서 서비스를 실행합니다:

1. http 를 위한 40000 번 포트에서 서비스 튜토리얼 4 를 실행하는 노드
2. http 를 위한 50000 번 포트를 사용하며 위 매니페스트를 실행하는 노드

이들 두 노드를 시작하기 위해 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 다음을 실행합니다.

첫번째 노드를 실행합니다:

```
dsshost /p:40000 /t:40001 /m:"SamplesWConfigWServiceTutorial4.manifest.xml"
```

이것은 포트 40000 의 노드를 시작하고, 서비스 튜토리얼 4 서비스를 시작합니다.

.WSamplesWConfig 디렉터리에 위의 매니페스트를 ServiceTutorial7.manifest.xml 로 저장하십시오

두 번째 노드를 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 실행합니다:

```
dsshost /p:50000 /t:50001 /m:"SamplesWConfigWServiceTutorial7.manifest.xml"
```

위 매니페스트를 사용하여 포트 50000 의 노드에서 서비스 튜토리얼 6 을 시작합니다. 그리고 포트 40000 의 노드에서 Clock 파트너를 찾습니다.

주의사항:

Clock 파트너는 네트워크에 있는 다른 머신에서 실행될 수 있습니다. 여기서는 명료함을 위해 같은 머신에서 동작하는 것을 보여줍니다.

단계 2: 다른 Node 에서 서비스 실행하기

매니페스트를 사용해 다른 노드에서 동작하는 서비스를 파트너로 가질 수 있습니다. 하지만 이 방법으로 다른 노드의 서비스를 시작하지는 못합니다. 다른 노드의 파트너 서비스를 실행하기 위해 서비스 튜토리얼 6 의 코드를 변경해야 합니다.

첫 번째로 clock 파트너로부터 파트너 어트리뷰트를 제거합니다. 이것은 서비스가 시작될 때 파트너가 더 이상 성립되지 않음을 의미합니다. 서비스는 파트너 관계를 자체적으로 성립해야 합니다.

_clockPort 의 맨 위에 있는 다음 어트리뷰트를 제거합니다:

```
[Partner("Clock", Contract = rst4.Contract.Identifier, CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
```

두 번째로 서비스가 시작될 노드를 찾습니다. 이것은 매니페스트 파트너에 디렉터리 서비스 경로를 지정하는 것으로 이루어 집니다.

```
<?xml version="1.0" ?>
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns:st6="http://schemas.tempuri.org/2006/06/servicetutorial6.html">
  <CreateServiceList>
    <!-- Service Tutorial 6 -->
    <ServiceRecordType>
      <d:Contract>http://schemas.tempuri.org/2006/06/servicetutorial6.html</d:Contract>
      <d:PartnerList>
        <d:Partner>
          <d:Service>http://localhost:40000/directory</d:Service>
          <d:Name>st6:Remote</:Name>
        </d:Partner>
      </d:PartnerList>
    </ServiceRecordType>
  </CreateServiceList>
</Manifest>
```

이 정보는 base.FindPartner(원격)를 호출하여 런타임에 서비스에 의해 읽어지고, 리턴 된 PartnerType 객체의 Service 필드는 매니페스트에서 지정한 URI 를 가지고 있을 것입니다.

이 서비스 URI 는 base.ServiceForwarder<T>()를 사용하는 서비스 포워더를 생성하기 위해 사용됩니다.

이를 위해 디렉토리 네임스페이스를 using 지시자로 선언합니다. 이와 같이 생성자(Constructor) 네임스페이스를 같이 선언합니다.

ServiceTutorial6.cs 에 다음 에일리어스를 추가하십시오:

```
using ds = Microsoft.Dss.Services.Directory; using cs =
```

```
Microsoft.Dss.Services.Constructor;
```

다음으로 OnStartup()에 다음 변경사항을 추가해 원격 디렉토리에 포워더를 생성합니다.

```
PartnerType remote = FindPartner("Remote");
ds.DirectoryPort remoteDir = DirectoryPort;

if (remote != null && !string.IsNullOrEmpty(remote.Service))
{
    remoteDir = ServiceForwarder<ds.DirectoryPort>(remote.Service);
}
```

이제 서비스는 원격 디렉터리 서비스에 대한 포워더를 가지거나 실패한 경우는 로컬 디렉터리 서비스를 가집니다.이것은 생성자 서비스의 인스턴스를 위한 디렉터리를 조회(query)할 수 있습니다.

OnStartup()의 마지막에 다음코드를 추가합니다:

```
ds.Query query = new ds.Query(
    new ds.QueryRequestType(
        new ServiceInfoType(cs.Contract.Identifier)
    )
);

remoteDir.Post(query);

cs.ConstructorPort remoteConstructor = ConstructorPort;

yield return Arbiter.Choice(
    query.ResponsePort,
    delegate(ds.QueryResponseType success)
    {
        remoteConstructor                                                    =
ServiceForwarder<cs.ConstructorPort>(success.RecordList[0].Service);
    },
    delegate(Fault fault) { }
);
```

이 코드는 생성자 서비스를 얻기 위해 Query 메시지를 디렉터리 서비스에 전송합니다. 실패할 경우 로컬 생성자 서비스를 사용으로 대신합니다.

최종적으로 생성자 서비스에 Create 메시지를 보냅니다.

```
cs.Create create = new cs.Create(new ServiceInfoType(rst4.Contract.Identifier));

string clockService = null;

remoteConstructor.Post(create);

yield return Arbiter.Choice(
    create.ResponsePort,
    delegate(CreateResponse success)
    {
        clockService = success.Service;
    },
    delegate(Fault f) { }
);

if (string.IsNullOrEmpty(clockService))
{
    yield break;
}

_clockPort = ServiceForwarder<rst4.ServiceTutorial4Operations>(clockService);
```

만일 생성자가 서비스 생성에 성공하면 이 생성된 서비스에 포워더를 만들어 사용합니다. OnStartup()을 다음과 같이 변경해 서비스 튜토리얼 4 의 정확한 경로를 설정합니다.

다음 코드를

```
PartnerType partner = FindPartner("Clock");
if (partner != null)
{
    initState.Clock = partner.Service;
}
```

아래 줄로 교체하십시오:

```
initState.Clock = clockService;
```

주의사항:

서비스 튜토리얼 7 의 최종판은 서비스 튜토리얼 6(C#)과 위에 설명된 것 부분에서 차이가 있으며 이는 다음 섹션에서 설명됩니다.

이를 실행하는데 다음이 필요합니다:

- 서비스 튜토리얼 6 에 위 변경사항을 적용
- 위 매니페스트를 .₩Samples₩Config₩ServiceTutorial7.manifest.xml 에 저장
- 다음 두 노드를 실행:
    - HTTP 를 위한 포트 40000 의 노드
    - HTTP 를 위한 다른 포트(50000 으로 설정됨)에서 위 매니페스트를 실행하는 노드

이 두 노드를 실행하기 위해 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 다음을 입력합니다.

첫 번째로:

```
dsshost /p:40000 /t:40001
```

이것은 포트 40000 에서 노드를 실행합니다.

주의사항:

이것은 시스템 서비스 이외의 어떤 서비스도 시작하지 않습니다.

두 번째로 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 다음을 실행합니다:

```
dsshost /p:50000 /t:50001 /m:"Samples₩Config₩ServiceTutorial7.manifest.xml"
```

위 매니페스트를 이용해 수정된 서비스 튜토리얼 6 을 포트 50000 에서 실행하여 포트 40000 에서 동작중인 노드에 있는 생성자 서비스를 찾도록 합니다.

잠시 후 다음 두라 라인이 콘솔에 나타납니다.

```
* Service uri: [http://localhost:50000/servicetutorial6]
* Service uri: [http://localhost:40000/servicetutorial4]
```

주의사항:

생성자 서비스는 네트워크에 있는 다른 머신에서 실행될 수 있습니다. 여기서는 명료함을 위해 같은 머신에서 동작하는 것을 보여줍니다.

단계 3: 다중 서비스 구성하기

서비스 튜토리얼 7의 최종판은 서비스 튜토리얼 6과 다음 사항에서 다릅니다.

1. ServiceTutorial7Types.cs 파일에 ServiceTutorial7State 타입은 string 리스트와 TickCount 객체 리스트를 포함합니다.

```
[DataContract]
public class ServiceTutorial7State
{
    private List<string> _clocks = new List<string>();

    [DataMember(IsRequired = true)]
    public List<string> Clocks
    {
        get { return _clocks; }
        set { _clocks = value; }
    }

    private List<TickCount> _tickCounts = new List<TickCount>();

    [DataMember(IsRequired = true)]
    public List<TickCount> TickCounts
    {
        get { return _tickCounts; }
        set { _tickCounts = value; }
    }
}

[DataContract]
public class TickCount
{
```

```
    public TickCount()
    {
    }

    public TickCount(int initial, string name)
    {
        _initial = initial;
        _count = initial;
        _name = name;
    }

    private string _name;
    [DataMember]
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    private int _count;
    [DataMember]
    public int Count
    {
        get { return _count; }
        set { _count = value; }
    }

    private int _initial;

    public int Initial
    {
        get { return _initial; }
        set { _initial = value; }
    }
}
```

2. ServiceTutorial7Types.cs 파일에서 조작 포트(ServiceTutorial7Operations)는 서비스 튜토리얼 7 에 정의된 IncrementTick 메시지를 사용합니다(서비스 튜토리얼 6 에서는 서비스 튜토리얼 4 에 정의된 것을 사용).

```
[ServicePort]
public class ServiceTutorial7Operations : PortSet<
    DsspDefaultLookup,
    DsspDefaultDrop,
    Get,
    HttpGet,
    Replace,
    IncrementTick,
    SetTickCount>
{
}
```

IncrementTick 메시지와 IncrementTickRequest 요청 타입을 위한 정의:

```
public class IncrementTick : Update<IncrementTickRequest,
PortSet<DefaultUpdateResponseType, Fault>>
{
    public IncrementTick()
    {
    }

    public IncrementTick(string source)
        : base(new IncrementTickRequest(source))
    {
    }
}

[DataContract]
[DataMemberConstructor]
public class IncrementTickRequest
{
    public IncrementTickRequest()
    {
    }
```

```
    public IncrementTickRequest(string name)
    {
        _name = name;
    }

    private string _name;
    [DataMember]
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

3. ServiceTutorial7Types.cs 파일에 SetTickCount 메시지는 TickCount 객체를 그 보디로 사용합니다.

```
public class SetTickCount : Update<TickCount, PortSet<DefaultUpdateResponseType,
Fault>>
{
    public SetTickCount()
    {
    }

    public SetTickCount(int tickCount, string source)
        : base(new TickCount(tickCount, source))
    {
    }
}
```

4. ServiceTutorial7.cs 파일에 "로컬" 파트너가 정의됩니다.

```
[Partner("Local", Contract = rst4.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.UsePartnerListEntry)]
rst4.ServiceTutorial4Operations _localClockPort = new
rst4.ServiceTutorial4Operations();
rst4.ServiceTutorial4Operations _localClockNotify = new
rst4.ServiceTutorial4Operations();
```

5. ServiceTutorial7.cs 파일에 Start()메서드에 notifcation 핸들러가 로컬 clock 파트너를 위해 구성됩니다. 로컬과 원격 파트너로부터 알림 메시지를 받기 위한 핸들러들이 각각 구성됩니다.

Start()메서드에서 활성화되는 핸들러들:

```
Activate<ITask>(
    Arbiter.Receive<rst4.IncrementTick>(true,                    _clockNotify,
RemoteNotifyTickHandler),
    Arbiter.Receive<rst4.Replace>(true,                          _clockNotify,
RemoteNotifyReplaceHandler),
    Arbiter.Receive<rst4.IncrementTick>(true,                _localClockNotify,
LocalNotifyTickHandler),
    Arbiter.Receive<rst4.Replace>(true,                      _localClockNotify,
LocalNotifyReplaceHandler)
);
```

원격 핸들러들:

```
private void RemoteNotifyTickHandler(rst4.IncrementTick incrementTick)
{
    LogInfo("Got Tick from remote");
    _mainPort.Post(new IncrementTick("Remote"));
}

private void RemoteNotifyReplaceHandler(rst4.Replace replace)
{
    LogInfo("Remote Tick Count: " + replace.Body.Ticks);
    _mainPort.Post(new SetTickCount(replace.Body.Ticks, "Remote"));
}
```

로컬 핸들러들:

```
private void LocalNotifyTickHandler(rst4.IncrementTick incrementTick)
{
    LogInfo("Got Tick from local");
    _mainPort.Post(new IncrementTick("Local"));
}
```

```
private void LocalNotifyReplaceHandler(rst4.Replace replace)
{
    LogInfo("Local Tick Count: " + replace.Body.Ticks);
    _mainPort.Post(new SetTickCount(replace.Body.Ticks, "Local"));
}
```

6. ServiceTutorial7.cs 파일에 로컬과 원격 파트너를 종료하는 드롭 핸들러가 구현되어 있습니다.

```
[ServiceHandler(ServiceHandlerBehavior.Teardown)]
public IEnumerator<ITask> DropHandler(DsspDefaultDrop drop)
{
    _clockPort.DsspDefaultDrop();
    _localClockPort.DsspDefaultDrop();

    base.DefaultDropHandler(drop);
    yield break;
}
```

이 변경은 서비스가 두 개의 파트너를 가지게 합니다: 하나는 서비스 튜토리얼 5 와 6 과 같이 로컬 파트너로 동작함. 다른 하나는 단계 2 에서 처럼 원격 파트너를 사용해 동작함

이 서비스는 다음 매니페스트를 사용합니다:

```
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns:s="http://schemas.tempuri.org/2006/06/servicetutorial7.html"
    >
  <CreateServiceList>
    <ServiceRecordType>
      <d:Contract>http://schemas.tempuri.org/2006/06/servicetutorial7.html</d:Contract>
      <d:PartnerList>
        <d:Partner>
          <d:Service>http://localhost:40000/directory</d:Service>
          <d:Name>s:Remote</d:Name>
        </d:Partner>
        <d:Partner>
          <d:Name>s:Local</d:Name>
```

```
        </d:Partner>
      </d:PartnerList>
    </ServiceRecordType>
    <ServiceRecordType>
      <d:Contract>http://schemas.tempuri.org/2006/06/servicetutorial4.html</d:Contract>
      <d:Service>http://localhost/localclock</d:Service>
      <d:PartnerList>
        <d:Partner>
          <d:Service>ServiceTutorial7.LocalClock.config.xml</d:Service>
          <d:Name>d:StateService</d:Name>
        </d:Partner>
      </d:PartnerList>
      <Name>s:Local</Name>
    </ServiceRecordType>
  </CreateServiceList>
</Manifest>
```

이 매니페스트 다음의 방법들로 서비스를 구성합니다:

1. http://localhost: 40000directory 에서 찾을 수 있는 원격 인스턴스 디렉터리를 지정
2. /localclock 서비스 패스를 가지는 로컬 파트너를 지정
3. 로컬 파트너가 ServiceTutorial7.LocalClock.config.xml 파일에 지정된 초기 상태를 사용하도록 지정

주의사항: config 파일 경로는 매니페스트 경로에서 상대적입니다. 따라서 config 파일은 ./Samples/Config/ServiceTutorial7.LocalClock.config.xml 에서 발견될 것입니다

원격 파트너가 시작될 때 서비스 튜토리얼 3(C#)에서 기술된 것과 같이 InitialStatePartner 어트리뷰트에 의해 선언된 초기 상태 구성 파일을 사용합니다. 서비스 튜토리얼 4 의 경우는 ./store/ServiceTutorial4.xml 입니다.

비록 양쪽 노드가 같은 배포 디렉터리로부터 서비스 튜토리얼 4 서비스의 두 인스턴스를 실행한다 하더라도, 포트 40000 과 포트 50000 에서 실행하는 두 노드는 다른 설정 파일을 사용할 것입니다. 이는 두 개 인스턴스의 Ticks 와 Member 필드를 변경한 후 각 서비스 상태를 조회하여 확인가능 합니다.

예를 들어 ./Samples/Config 디렉터리에 다음을 ServiceTutorial7.LocalClock.Config.xml 로 복사합니다

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceTutorial4State
xmlns="http://schemas.tempuri.org/2006/06/servicetutorial4.html">
  <Member>Local Clock Configuration</Member>
  <Ticks>1000</Ticks>
</ServiceTutorial4State>
```

그 다음 마이크로소프트 Robotics Studio 커맨드 프롬프트 중 하나에서 HTTP 포트 40000 을 사용하는 노드를 다음과 같이 시작합니다.

```
dsshost /p:40000 /t:40001
```

이 노드가 실행되고 있을 때, 또 다른 마이크로소프트 Robotics Studio 커맨드 프롬프트에서 포트 50000 의 다른 노드를 매니페스트를 지정하여 시작하십시오.

```
dsshost /p:50000 /t:50001 /m:"SamplesWConfigWServiceTutorial7.manifest.xml"
```

두 번째 노드가 다음을 보여줄 것입니다.

```
* Service uri: [http://localhost:50000/localclock]
* Service uri: [http://localhost:50000/servicetutorial7]
```

이후 첫 번째 노드가 다음을 보여줄 것입니다.

```
* Service uri: [http://localhost:40000/servicetutorial4]
```

http://localhost:50000/localclock 과 http://localhost:40000/servicetutorial4 를 웹브라우저에서 열어 서비스가 다른 설정으로 시작됨을 확인할 수 있습니다.

주의사항:

이 튜토리얼이 localhost 라고 언급한 곳에는 현재의 머신 이름이 표시될 것입니다

6. DSS 호스팅 튜토리얼

호스팅 튜토리얼 1(C#) - DSS환경 시작과 정지하기

DSS 서비스를 실행하기 위해 DssHost.exe 툴을 사용하거나 DSS 노드를 호스트하는 애플리케이션을 만들 수 있습니다. 호스팅 튜토리얼에서는 이 중 후자의 예를 보여줍니다. DssEnvironment.dll 어셈블리와 DssEnvironment 정적인 클래스를 사용해 여러분의 애플리케이션이나 .NET 원도우 서비스 내에서 DSS 런타임을 초기화할 수 있습니다.

DSS 노드를 호스트하는 콘솔 애플리케이션 제작

이 예제를 위한 비주얼 스투디오 솔루션은 마이크로소프트 Robotics Studio 설치 디렉터리 아래의 SamplesWHostingTutorialsWTutorial1WCSharpWHostingTutorial1.sln 에 있습니다. 솔루션 파일을 열거나 비주얼 스투디오의 새로운 C# 콘솔 애플리케이션 프로젝트를 만든 후 다음 코드로 해당 파일의 내용을 교체합니다.

다음 코드는 DSS 노드를 호스팅 하는 것을 보여줍니다. 애플리케이션은 DSS 노드를 실행하고, 20 초 동안 기다린 뒤 노드를 종료합니다.

```
using System;

using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Hosting;
using Microsoft.Dss.ServiceModel.Dssp;

namespace HostingTutorial1
{
    class Program
    {
        /// <summary>
        /// Main entry to the program
        /// </summary>
        static int Main()
        {
            // Initialize DSS node on port 50000 (HTTP) and 50001 (TCP)
            DssEnvironment.Initialize(50000, 50001);
```

```
            // Post a timer message that expires in 20 seconds
            TimeSpan timeout = new TimeSpan(0, 0, 20);
            Port<DateTime> pTimeout = new Port<DateTime>();
            DssEnvironment.TaskQueue.EnqueueTimer(timeout, pTimeout);

            // Wait for timeout to happen and shutdown node
            Arbiter.Activate(DssEnvironment.TaskQueue,
                Arbiter.Receive<DateTime>(false, pTimeout, delegate(DateTime t)
                {
                    DssEnvironment.LogInfo("Received timeout signal");
                    DssEnvironment.Shutdown();
                })
            );

            // Wait here until DSS Environment has been shut down
            DssEnvironment.WaitForShutdown();

            // Exit program
            return 0;
        }
    }
}
```

애플리케이션을 실행하는 동안 여러분은 http://localhost:50000 에서 동작중인 DSS 노드를 확인할 수 있습니다. 노드는 20 초 동안 동작함에 주의하십시오.

주의사항

정적 Microsoft.Dss.Hosting.DssEnvironment 클래스는 새로운 DSS 노드를 지정된 HTTP 와 TCP 포트에서 시작하는 초기화(Initialize) 메서드와 실행할 매니페스트 파일의 옵션 리스트를 제공합니다.

DssEnvironment 클래스도 서비스의 인스턴스를 위한 Directory Service 를 검색하거나, 로그 서비스를 사용하는 것과 같은 런타임 서비스를 위한 헬퍼 메서드를 제공합니다. DSS 노드가 초기화 될 때 서비스에 대한 태스크 스케줄링을 위해 CCR dispatcher 와 dispatcher 큐가 할당됩니다. 할당된 DispatcherQueue 는 DssEnvironment.TaskQueue 프로퍼티를 통하여 액세스 될 수 있습니다.

DSS 노드가 실행되는 동안 여러분의 애플리케이션에서 동시 처리 프로그래밍을 처리하는 서비스 클래스와 유사한 CCR 패턴을 사용할 수 있습니다. 그러나 여기에서는 Arbiter.Activate 를 호출할 때 Arbiter 에 DispatcherQueue 를 명백하게 전달해야 합니다.

DssEnvironment.WaitForShutdown()을 Main() 메서드의 끝에 사용해 애플리케이션이 DSS 가 동작 중에 종료되는 것을 피할 수 있습니다.

호스팅 튜토리얼 2(C#) - 호스트 된 DSS노드에서 서비스 검색하기

DSS 서비스를 실행하기 위해 DssHost.exe 툴을 사용하거나 DSS 노드를 호스트하는 애플리케이션을 만들 수 있습니다. 호스팅 튜토리얼에서는 이 중 후자의 예를 보여줍니다. DssEnvironment.dll 어셈블리와 DssEnvironment 정적인 클래스를 사용해 여러분의 애플리케이션이나 .NET 원도우 서비스 내에서 DSS 런타임을 초기화할 수 있습니다.

호스트 된 DSS 노드내에서 서비스 검색하기(Query)

이 예제를 위한 비주얼 스투디오 솔루션은 마이크로소프트 Robotics Studio 설치 디렉터리 아래의 SamplesWHostingTutorialsWTutorial2WCSharpWHostingTutorial2.sln 에 있습니다. 솔루션 파일을 열거나 비주얼 스투디오의 새로운 C#콘솔 애플리케이션 프로젝트를 만들고, 다음 코드로 대상 파일의 내용을 교체합니다. 또한 프로젝트에 ServiceTutorial1 서비스의 프록시 어셈블리를 참조로 추가해야 합니다.

다음 코드는 DSS 노드를 호스트 하고 서비스를 검색하는 것을 보여줍니다. 애플리케이션은 ServiceTutorial1 매니페스트를 가지고 DSS 노드를 실행한 뒤 ServiceTutorial1 서비스를 검색을 Service Directory 에 요청합니다. 만일 서비스가 발견되면 ServiceTutorial1State 객체의 Member 프로퍼티의 값을 로그하고 그 상태를 검색하기 위해 ServiceTutorial1 서비스에 Get 요청을 보냅니다. 만일 이 프로세스 동안 요청이 실패하면 애플리케이션은 오류를 기록하고 DSS 노드를 종료합니다.

```
using System;
using System.Collections.Generic;

using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Hosting;
using Microsoft.Dss.ServiceModel.Dssp;

using st1 = RoboticsServiceTutorial1.Proxy;

namespace HostingTutorial2
{
    class Program
    {
        // Used to set exit code
        static int _status = 1;
```

```
        /// <summary>
        /// Main entry to the program
        /// </summary>
        /// <returns>0 if program succeeds, otherwise a value greater than 0</returns>
        static int Main()
        {
            // Initialize DSS node on port 50000 (HTTP) and 50001 (TCP) with a manifest
            // starting an instance of the Service Tutorial 1 service.
            DssEnvironment.Initialize(50000, 50001,
                LayoutPaths.RootDir          +          LayoutPaths.SampleDir          +
@"configWServiceTutorial1.manifest.xml");

            // Spawn RunTask
            Arbiter.Activate(DssEnvironment.TaskQueue,
Arbiter.FromIteratorHandler(RunTask));

            // Wait here until DSS Environment has been shut down
            DssEnvironment.WaitForShutdown();

            // Exit program
            return _status;
        }

        /// <summary>
        /// The runtask function uses iterators to mimic sequential programming even
though
        /// it is actually running asynchronously.
        /// </summary>
        /// <returns></returns>
        static IEnumerator<ITask> RunTask()
        {
            // Operations port for service tutorial 1
            st1.ServiceTutorial1Operations pServiceTutorial1 = null;

            // First find Service Tutorial 1 in the local directory
            yield                                                                return
```

```
Arbiter.Choice(DssEnvironment.DirectoryQuery(st1.Contract.Identifier),
                delegate(ServiceInfoType success)
                {
                    // Request succeeded.
                    DssEnvironment.LogInfo("Found service tutorial 1 service: " +
success.Service);

                    System.Uri addr;
                    try
                    {
                        // Create URI from service instance string
                        addr = new System.Uri(success.Service);

                        // Now create service forwarder to service tutorial 1
                        pServiceTutorial1 =

DssEnvironment.ServiceForwarder<st1.ServiceTutorial1Operations>(addr);
                    }
                    catch (Exception e)
                    {
                        DssEnvironment.LogError("Could not create URI: " + e.Message);
                    }
                },
                delegate(W3C.Soap.Fault failure)
                {
                    // Request failed.
                    DssEnvironment.LogError("Could not find service tutorial 1");
                }
            );

            // Check that we got a forwarder. If not then we stop here.
            if (pServiceTutorial1 == null)
            {
                DssEnvironment.Shutdown();
                yield break;
            }
```

```
            // Send a GET request to service tutorial 1
            yield return Arbiter.Choice(pServiceTutorial1.Get(),
                delegate(st1.ServiceTutorial1State success)
                {
                    // GET Request succeeded
                    DssEnvironment.LogInfo("Service Tutorial1 state: " +
success.Member);

                    // Set exit code to 0 for success
                    _status = 0;
                },
                delegate(W3C.Soap.Fault failure)
                {
                    // GET Request failed.
                    DssEnvironment.LogError("Could not retrieve state.");
                }
            );

            // Shut down runtime
            DssEnvironment.LogInfo("Run task completed");
            DssEnvironment.Shutdown();
        }
    }
}
```

애플리케이션을 실행하는 동안 여러분은 http://localhost:50000 에서 동작중인 DSS 노드를 확인할 수 있습니다.

호스팅 튜토리얼 3(C#) - 동시처리 요청 전송과 Unknown 응답 타입 처리하기

DSS 서비스를 실행하기 위해 DssHost.exe 툴을 사용하거나 DSS 노드를 호스트하는 애플리케이션을 만들 수 있습니다. 호스팅 튜토리얼에서는 이 중 후자의 예를 보여줍니다. DssEnvironment.dll 어셈블리와 DssEnvironment 정적인 클래스를 사용해 여러분의 애플리케이션이나 .NET 원도우 서비스 내에서 DSS 런타임을 초기화할 수 있습니다.

동시 요청 전송과 알려지지 않은 응답 타입 처리하기

이 예제를 위한 비주얼 스투디오 솔루션은 마이크로소프트 Robotics Studio 설치 디렉터리 아래의 SamplesWHostingTutorialsWTutorial3WCSharpWHostingTutorial3.sln 에 있습니다. 솔루션 파일을 열거나 비주얼 스투디오의 새로운 C# 콘솔 애플리케이션 프로젝트를 만들고, 다음 코드로 대상 파일의 내용을 교체합니다. 또한 프로젝트에서 ServiceTutorial1 과 ServiceTutorial2 서비스의 프록시 어셈블리를 참조로 추가해야 합니다.

다음 코드는 DSS 노드와 문의 다중이 서비스하는 호스팅을 보여줍니다. 애플리케이션은 ServiceTutorial1 과 ServiceTutorial2 매니페스트로 두 개의 서비스를 로드하기 위해 DSS 노드를 실행합니다. 그 뒤 서비스 실행 전에 두 개의 동시 요청을 서비스 디렉터리에 전송하고 이에 대한 두 개의 응답을 기다립니다. 두 개의 ServiceInfoType 응답이 디렉터리로부터 받게 되자마자 이 응답들은 각 서비스가 메시지를 전송하게 하는 개개의 서비스 포워더를 생성하는 데 사용됩니다. 그 다음으로 DsspDefaultGet 메시지는 ServiceTutorial1 과 ServiceTutorial2 서비스 각각에 포스트 되고 양 쪽의 응답 상태 타입을 다시 기다립니다.

만일 이 프로세스 동안 요청이 실패하면 애플리케이션은 오류를 기록하고 DSS 노드를 종료합니다.

```
using System;
using System.Collections.Generic;
using System.Threading;

using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Hosting;
using Microsoft.Dss.ServiceModel.Dssp;

using ds = Microsoft.Dss.Services.Directory;
using st1 = RoboticsServiceTutorial1.Proxy;
using st2 = RoboticsServiceTutorial2.Proxy;

namespace HostingTutorial3
```

```
{
    class Program
    {
        // Used to set exit code
        static int _status = 0;

        /// <summary>
        /// Main entry to the program
        /// </summary>
        /// <returns>0 if program succeeds, otherwise a value greater than 0</returns>
        static int Main()
        {
            // Initialize DSS using some manifests
            DssEnvironment.Initialize(50000, 50001,
                LayoutPaths.RootDir          +          LayoutPaths.SampleDir          +
@"configWServiceTutorial1.manifest.xml",
                LayoutPaths.RootDir          +          LayoutPaths.SampleDir          +
@"configWServiceTutorial2.manifest.xml");

            // Spawn RunTask
            Arbiter.Activate(DssEnvironment.TaskQueue,
Arbiter.FromIteratorHandler(RunTask));

            // Wait here until DSS Environment has been shut down
            DssEnvironment.WaitForShutdown();

            // Exit program
            return _status;
        }

        /// <summary>
        /// The runtask function uses iterators to mimic sequential programming even
though
        /// it is actually running asynchronously.
        /// </summary>
        /// <returns></returns>
        static IEnumerator<ITask> RunTask()
```

```
        {
            // Keep track of how many services we find
            int serviceCount = 0;

            // Port for holding directory results
            DsspResponsePort<ServiceInfoType>       directoryResults       =       new
DsspResponsePort<ServiceInfoType>();

            // Start looking in directory for service tutorial 1
            DssEnvironment.DirectoryQuery(st1.Contract.Identifier,
DsspOperation.DefaultShortTimeSpan, directoryResults);

            // Start looking in directory for service tutorial 2
            DssEnvironment.DirectoryQuery(st2.Contract.Identifier,
DsspOperation.DefaultShortTimeSpan, directoryResults);

            // Response port to use for receiving GET responses
            // We explicitly use 'object' as we don't know the types
            PortSet<W3C.Soap.Fault, object> serviceResults = new PortSet<W3C.Soap.Fault,
object>();

            // Wait for results for both queries
            yield return Arbiter.MultipleItemReceive(directoryResults, 2,
                delegate(ICollection<ServiceInfoType>                          successes,
ICollection<W3C.Soap.Fault> failures)
                {
                    // Issue GET on the results we receive from the directory
                    foreach (ServiceInfoType s in successes)
                    {
                        // Increment our counter
                        Interlocked.Increment(ref serviceCount);

                        // First create a forwarder to the service instance.
                        // We use unknown type as we just issue GET requests
                        IPort                        servicePort                        =
DssEnvironment.ServiceForwarderUnknownType(new System.Uri(s.Service));
```

```
                    // Post GET request to service. Again, we don't know the
response type so we use 'object'
                    servicePort.PostUnknownType(new DsspDefaultGet(typeof(object),
serviceResults));
                }

                // Log failures
                foreach (W3C.Soap.Fault f in failures)
                {
                    DssEnvironment.LogWarning("Could not find instance: " +
failures.ToString());
                }
            }
        );

        // Check that we found services
        if (serviceCount == 0)
        {
            Cleanup(1, "Did not find any service instances.");
            yield break;
        }

        // Do GET on the service instances we found
        yield return Arbiter.MultipleItemReceive(serviceResults, serviceCount,
            delegate(ICollection<W3C.Soap.Fault> failures,
            ICollection<object> successes)
            {
                foreach (object obj in successes)
                {
                    if (obj is st1.ServiceTutorial1State)
                    {
                        st1.ServiceTutorial1State state = obj as
st1.ServiceTutorial1State;
                        DssEnvironment.LogInfo("Service 1 state: " + state.Member);
                    }
                    else if (obj is st2.ServiceTutorial2State)
                    {
```

```
                            st2.ServiceTutorial2State      state      =      obj      as
st2.ServiceTutorial2State;
                            DssEnvironment.LogInfo("Service 2 state: " + state.Member +
" " + state.Ticks);
                        }
                        else
                        {
                            DssEnvironment.LogInfo("Unknown state");
                        }
                    }

                    foreach (W3C.Soap.Fault f in failures)
                    {
                        DssEnvironment.LogError("Failure: " + f.ToString());
                    }
                }
            );
            Cleanup(0, "Completed run task");
        }

        static void Cleanup(int status, string message)
        {
            _status = status;
            if (_status != 0)
            {
                DssEnvironment.LogError(message);
            }
            else
            {
                DssEnvironment.LogInfo(message);
            }

            // Shut down DSS node
            DssEnvironment.Shutdown();
        }
    }
}
```

호스팅 튜토리얼 4(C#) - 동적으로 서비스 인스턴스 생성하기

DSS 서비스를 실행하기 위해 DssHost.exe 툴을 사용하거나 DSS 노드를 호스트하는 애플리케이션을 만들 수 있습니다. 호스팅 튜토리얼에서는 이 중 후자의 예를 보여줍니다. DssEnvironment.dll 어셈블리와 DssEnvironment 정적인 클래스를 사용해 여러분의 애플리케이션이나 .NET 원도우 서비스 내에서 DSS 런타임을 초기화할 수 있습니다.

Service 의 Instance 를 만드는 것

이 예제를 위한 비주얼 스투디오 솔루션은 마이크로소프트 Robotics Studio 설치 디렉터리 아래의 SamplesWHostingTutorialsWTutorial4WCSharpWHostingTutorial4.sln 에 있습니다. 솔루션 파일을 열거나 비주얼 스투디오의 새로운 C# 콘솔 애플리케이션 프로젝트를 만들고, 다음 코드로 대상 파일의 내용을 교체합니다. 또한 프로젝트에서 ServiceTutorial1 서비스의 프록시 어셈블리를 참조로 추가해야 합니다.

다음 코드는 DSS 노드를 호스팅하고 검색할 서비스의 인스턴스를 생성하는 것을 보여줍니다. 이 예제는 호스팅 튜토리얼 2 와 유사하지만 DSS 노드가 매니페스트 없이 시작되고 프로그램에서 직접 DssEnvironment.CreateService 메서드를 호출하여 ServiceTutorial1 서비스의 인스턴스를 생성하는 점이 다릅니다. 그 다음 를 ServiceTutorial1 서비스에 상태 검색을 위해 Get 요청을 보내고 ServiceTutorial1State 객체의 Member 프로퍼티 값을 로그합니다. 만일 이 프로세스 동안 요청이 실패하면 애플리케이션은 오류를 기록하고 DSS 노드를 종료합니다.

```
using System;
using System.Collections.Generic;

using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Hosting;
using Microsoft.Dss.ServiceModel.Dssp;

using ds = Microsoft.Dss.Services.Directory;
using st1 = RoboticsServiceTutorial1.Proxy;

namespace HostingTutorial4
{
    class Program
    {
        // Used to set exit code
```

```
        static int _status = 1;

        /// <summary>
        /// Main entry to the program
        /// </summary>
        /// <returns>0 if program succeeds, otherwise a value greater than 0</returns>
        static int Main()
        {
            // Initialize DSS without any manifests
            DssEnvironment.Initialize(50000, 50001);

            // Spawn RunTask
            Arbiter.Activate(DssEnvironment.TaskQueue,
Arbiter.FromIteratorHandler(RunTask));

            // Wait here until DSS Environment has been shut down
            DssEnvironment.WaitForShutdown();

            // Exit program
            return _status;
        }

        /// <summary>
        /// The runtask function uses iterators to mimic sequential programming even
though
        /// it is actually running asynchronously.
        /// </summary>
        /// <returns></returns>
        static IEnumerator<ITask> RunTask()
        {
            // Operations port for service tutorial 1
            st1.ServiceTutorial1Operations pServiceTutorial1 = null;

            // Create service tutorial 1 instance
            yield                                                          return
Arbiter.Choice(DssEnvironment.CreateService(st1.Contract.Identifier),
                delegate(CreateResponse success)
```

```
                {
                    // Create Request succeeded.
                    DssEnvironment.LogInfo("Created service tutorial 1 service: " +
success.Service);

                    System.Uri addr;
                    try
                    {
                        // Create URI from service instance string
                        addr = new System.Uri(success.Service);

                        // Now create service forwarder to service tutorial 1
                        pServiceTutorial1 =

DssEnvironment.ServiceForwarder<st1.ServiceTutorial1Operations>(addr);
                    }
                    catch (Exception e)
                    {
                        DssEnvironment.LogError("Could not create URI: " + e.Message);
                    }
                },
                delegate(W3C.Soap.Fault failure)
                {
                    // Request failed.
                    DssEnvironment.LogError("Could not find service tutorial 1");
                }
            );

            // Check that we got a forwarder. If not then we stop here.
            if (pServiceTutorial1 == null)
            {
                DssEnvironment.Shutdown();
                yield break;
            }

            // Send a GET request to service tutorial 1
            yield return Arbiter.Choice(pServiceTutorial1.Get(),
```

```
                delegate(st1.ServiceTutorial1State success)
                {
                    // GET Request succeeded
                    DssEnvironment.LogInfo("Service Tutorial1 state: " +
success.Member);

                    // Set exit code to 0 for success
                    _status = 0;
                },
                delegate(W3C.Soap.Fault failure)
                {
                    // GET Request failed.
                    DssEnvironment.LogError("Could not retrieve state.");
                }
            );

            // Shut down runtime
            DssEnvironment.LogInfo("Run task completed");
            DssEnvironment.Shutdown();
        }
    }
}
```

Atom/RSS 신디케이션 서비스 개요

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Atom/RSS Syndication Service Overview

Atom/RSS 신디케이션 서비스 튜토리얼

본 자료의 세부적인 내용은 아래 영문 내용을 참고하시기 바랍니다.

- Atom/RSS Syndication Service Tutorial