웹 서비스를 위한 예외 상황 기반 동적 서비스 연결 프레임워크
(Exception based Dynamic Service Coordination Framework for Web Services)

한 동 수 † 이 성 독 †† 정 종 하 †††
(Dongsoo Han) (Sungdoke Lee) (Jongha Jung)

요 약 인터넷상에서 접근 가능한 웹 서비스는 가용성 및 성능적 측면에서 신뢰성이 항상 보장되지는 못한다. 동적 서비스 연결(dynamic service coordination) 기법은 웹 서비스를 호출하는 시스템이나 응용 프로그램에서 이 같이 신뢰할 수 없는 상황에 대처할 수 있는 꼭 필요한 기술이다. 웹 서비스의 신뢰적인 호출을 보장해 주기 위해서 동적 서비스 연결 기법에서는 특정 웹 서비스가 정해진 시간 내에서 응답하지 못할 경우 실행 중에 해당 웹 서비스를 다른 웹 서비스로 대체하여 수행한다. 본 논문에서는 웹 서비스를 위한 예외 상황 기반 동적 서비스 연결 프레임워크를 제안한다. 이 프레임워크에서 동적 서비스 연결에 필요한 모든 정보들은 속성(attributes) 집합 형태에 의해서 명시적으로 기술된다. 본 논문에서 고안된 동적 서비스 연결 기법은 기술된 속성 정보를 기반으로 동적 서비스 연결이 가능한 클래스 또는 워크플로우를 자동으로 생성하고, 생성된 클래스 함수를 통해 웹 서비스를 간접적으로 호출함으로써 신뢰성 있는 웹 서비스 호출을 지원한다. 제안된 방식은 웹 서비스 호출 작업이 간접적으로 이뤄지기 때문에 이에 초래되는 약간의 성능적 손실을 피할 수는 없지만, 이 방법을 통해서 얻어지는 시스템의 유연성(flexibility)과 신뢰성(reliability)를 고려할 때, 다소의 성능적 손실은 많은 경우에 있어서 허용될 수 있을 것으로 예상된다.

키워드 : 웹 서비스, 동적 서비스 연결, 워크플로우 시스템, 예외 상황 핸들링, 웹바인

Abstract Web services on the Internet are not always reliable in terms of service availability and performance. Dynamic service coordination capability of a system or an application invoking Web services is essential to cope with such unreliable situations. In dynamic service coordination, if a Web service does not respond within a specific time constraint, it is replaced with another Web service at run time for reliable invocation of Web services. In this paper, we develop an exception based dynamic service coordination framework for Web services. In the framework, all necessary information for dynamic service coordination is explicitly specified and summarized as a set of attributes. Then classes and workflows, supporting dynamic service coordination and invoking Web services, are automatically created based on these attributes. Developers of Web services client programs can make the invocations of Web services reliable by calling the methods of the classes. Some performance loss has been observed in the indirect invocation of a Web service. However, when we consider the flexibility and reliability gained from the method, the performance loss would be acceptable in many cases.

Key words : Web services, Dynamic Service Coordination, Workflow System, Exception Handling, WebVine

1. 서 론

웹 서비스는 표준에 기반한 분산된 컴포넌트 서비스이면서 동시에 인터넷 상에서 이들 서비스의 느슨한 연결을 지원하는 플랫폼 기술이기도 하다[1]. 웹 서비스의 이러한 특성 때문에 웹 서비스 기반 응용 프로그램은 기존의 소프트웨어 컴포넌트 기반 응용 프로그램과 비교하면 보다 더 유연하게 인터넷 상에 공개된 다양한

† 종신회원 : 한국정보통신대학교 공학부 교수
dshan@icu.ac.kr
†† 정 회 원 : 한국정보통신대학교 공학부 교수
sdlee@icu.ac.kr
††† 학생회원 : 한국정보통신대학교 공학부
jongha@icu.ac.kr
논문접수 : 2005년 11월 18일
심사완료 : 2006년 5월 29일

서비스를 연계해서 프로그램을 구성할 수 있다. 즉, 비즈니스 응용 프로그램 개발 과정 중에 여러 소스에서 다양한 웹 서비스들을 선택 할 수 있으며, 느슨히 연결된 웹 서비스의 인터페이스를 통해서 서로가 쉽게 연결되고 조합된다. 웹 서비스에 기반한 응용 프로그램에서는 SOAP(Simple Object Access Protocol)[2], WSDL (Web Services Description Language)[3], UDDI (Universal Description Discovery and Integration)[4], 그리고 HTTP(Hypertext Transfer Protocol)와 같은 웹 표준 언어와 프로토콜을 이용하여 기업의 경계에 한정 받지 않고 필요한 서비스를 사용하는 것이 가능하다. 한편 웹 서비스 연계을 통하여 새로운 비지니스 프로세스를 구성하는 경우에는 웹 서비스를 비즈니스 프로세스의 한 부분으로 간주할 수 있다.

이러한 장점에도 불구하고 현재 인터넷 상에서 웹 서비스는 서비스의 가용성(availability) 및 성능적(performance) 측면에서 신뢰성이 보장되지 못하는 약점을 가지고 있다. 예를 들면, 호출한 웹 서비스에 접근이 되지 않거나, 정해진 시간 안에 응답을 얻지 못하는 경우가 자주 발생한다. 또한 예상치 못한 결과의 값을 받게 되는 경우도 있다. 다행히도, 웹 서비스 환경에서는 유사한 서비스 혹은 동일한 서비스를 지원하는 사이트들이 존재한다고 가정할 수 있다. 따라서 문제가 된 웹 서비스를 다른 웹 서비스로 대체하여 이와 같은 문제의 해결책을 시도하는 것이 가능하다. 웹 서비스가 접근 가능한 상태가 아니거나, 정해진 시간 안에 응답을 하지 못할 때, 웹 서비스의 실행 중에 다른 서비스로 대체하는 작업을 가능하게 하는 것이 바로 동적 서비스 연결의 목적이다.

응용 프로그램이나 서비스에서 동적 서비스 연결을 지원하도록 하는 방법으로 여러 가지를 생각할 수 있다. 첫번째 접근 방법으로는 모든 조건 및 상태를 검사하는 핸들링 (또는 처리) 루틴을 응용 프로그램에 내장시키는 것이다. 그러나 여러 응용 프로그램 내에서 공유될 수 있는 코드를 프로그램에 내장시키는 이 방법은 효율적이지 못하다. 이에 대한 보완으로써 다른 접근 방법은 동적 서비스 연결을 지원하는 독립된 모듈을 만드는 것이다. 그리고 독립된 모듈에는 핸들링 루틴에 해당하는 공유조건 및 상태검사 루틴들을 모듈 안에 준비해 둔다. 응용 프로그램들은 이 모듈 내의 루틴을 호출하는 간접적인 방법으로 웹 서비스를 호출하게 된다. 이 방법은 공통된 코드부분을 공유할 수 있다는 장점이 있기 때문에 개발자의 부담감을 덜어준다. 하지만 이 방법도 제공되는 서비스에 변화가 생기면 해당 모듈이 수정되어야 한다는 점에서 여전히 한계를 가지고 있다.

본 논문에서는 위와 같은 한계를 극복하기 위한 방안으로 웹 서비스를 위한 동적 서비스 연결 프레임워크를 제안한다. 동적 서비스 연결 프레임워크에서는 웹 환경의 변화를 서비스 실행 중에 반영하여 필요한 웹 서비스를 호출할 수 있다는 점에서 이전의 방법들보다 우수한 방법으로 볼 수 있다. 동적 서비스 연결 프레임워크에서는 연결에 필요한 모든 정보들을 속성 집합으로 명확히 명시하고, 연결을 지원하는 클래스와 워크플로우는 명시된 속성 값을 기반으로 하여 자동 생성된다. 생성된 클래스에는 웹 서비스 호출에 문제가 발생하였을 때 이에 대응하기 위한 루틴이 사용자가 명시한 방식으로 준비되어 있어서 서비스 실행 중에 루틴으로 제어가 이동되어 문제의 처리를 실행하게 된다. 이는 워크플로우의 예외 상황 핸들링 메커니즘[6,7]과 유사한 방식으로 실제로 본 논문에서도 이러한 예외 상황 핸들링 메커니즘을 참고하고 있다. 결과적으로 동적 서비스 연결 프레임워크 활용을 통해서 웹 서비스 클라이언트 응용 프로그램의 개발자들은 생성된 클래스 내의 메소드를 이용하여 신뢰성 있는 웹 서비스 호출을 손쉽게 구현할 수 있게 된다.

동적 서비스 연결 프레임워크는 앞에서 서술한 장점을 가지고 있긴 하지만 웹 서비스 호출 시 지연시간과 같은 성능적 손실을 초래하는 단점이 있다. 워크플로우를 이용하여 웹 서비스를 호출할 경우 평균적으로 약 0.5초 정도의 지연시간이 발생하는 것을 확인하였다. 그러나 이런 성능적 손실은 동적 서비스 연결을 통하여 얻어지는 서비스의 유연성(flexibility)과 신뢰성(reliability)을 고려한다면, 서비스 종류에 따라서 어느정도 수용 될 수도 있다고 본다. 또한 발생하는 지연시간은 최적화 된 상태에서 측정된 값이 아니기 때문에 향후 개선이 가능하다.

본 논문의 구성은 다음과 같다. 2장에서는 웹 서비스를 위한 동적 서비스 연결 기술의 일반적인 개념 및 관련연구를 설명한다. 3장에서는 응용 프로그램에서 동적 서비스 연결을 지원할 수 있도록 하는 동적 서비스 연결 프레임워크에 대한 설명을 하며, 4장에서는 동적 서비스 연결 구현에 사용한 워크플로우 시스템의 예외 상황 핸들링 메커니즘에 관하여 서술한다. 5장에서는 본 논문에서 제안한 메커니즘을 사용한 웹 서비스 호출 방식의 성능을 측정하고, 6장에서 결론을 맺는다.

2. 동적 서비스 연결

2.1 웹 서비스를 위한 동적 서비스 연결

본 논문에서는 편의상 웹 서비스 상에서 동적으로 서비스 연결을 지원하는 일련의 모든 기술들을 동적 서비스 연결(dynamic service coordination)이라 명명한다. 동적 서비스 연결 개념은 워크플로우 분야에서 사용된

동적 워크플로우 변경(dynamic workflow reconfiguration)이란 개념과 유사하다. 동적 워크플로우 변경은 서비스 실행 중에 실행 패스나 핸들러 등, 미리 정의된 프로세스와 관련된 속성들을 변경하고 실행하는 기법이다. 동적 서비스 연결에서도 동적 워크플로우 변경과 유사한데 즉, 처음에 응용 프로그램이나 서비스 프로그램에 연결되는 것으로 정의된 웹 서비스를 서비스 실행 중에 다른 웹 서비스로 대체하는 것을 의미한다. 먼저 호출하기로 정의된 웹 서비스가 적절한 응답을 하지 못하는 경우, 동적 서비스 연결 기능을 통해서 문제가 발생한 웹 서비스 호출을 다른 웹 서비스로 대체하여 호출함으로써 서비스의 안정성을 높이게 된다. 여기서 주목해야 하는 것은 이러한 대체 작업이 실행 중의 정해진 제한 시간 내에서 이루어져야 한다는 것이다. 인터넷 환경에서 제공되는 웹 서비스가 본질적으로 신뢰성이 높지 않기 때문에 동적 서비스 연결 기술은 웹 서비스 환경에서는 매우 유용하게 사용될 것으로 예상된다.

웹 서비스 환경에서 다양한 종류를 갖고, 지속적으로 변화되는 서비스를 대상으로 동적 서비스 연결 기술을 적용시키는 접근 방법으로는 크게 두 가지를 들 수 있다. 첫 번째 접근 방법으로는 서비스 연결시 동적 서비스 연결을 위해 UDDI에 의뢰하는 방식으로 특정 웹 서비스 호출에 문제가 발생하면 UDDI에 해당 사실을 통보하고, UDDI는 대체 웹 서비스를 찾아서 연결시켜 주는 방식이다. 이상적인 상황이라면 UDDI가 후보 서비스에 관한 정보를 충분히 갖고, 동적 서비스 연결에 필요한 기능을 갖추게 되면 적당한 시간에 쉽게 최상의 후보 서비스들의 위치를 찾을 수 있다는 점에서 매력적이다. 그러나 현실적으로 UDDI자체가 완벽하지 못하고, 동적 서비스 연결을 완벽하게 지원할 수 있는 표준이 아직 만들어지지 않았기 때문에 현재로서는 적절한 방식으로 보기가 어렵다. 두 번째 접근 방법으로는 웹 서비스 별로 대체 가능한 웹 서비스 후보들을 미리 사용자에게 명시하도록 하고, 웹 서비스가 적절하게 응답하지 않으면 명시된 후보 중에서 하나를 선택한 다음, 상황과 사양에 따라서 실행하는 것이다. 이 방법은 처리과정에 있어서 UDDI가 개입하지 않는다는 점에서 첫 번째 방법과 차이가 있다. 본 논문에서는 두 번째 접근 방법을 선택하였으며, 이 방식의 구현에는 워크플로우 시스템에서 사용되는 예외 상황 핸들링 메커니즘을 사용한다. 즉 특정 웹 서비스가 적절히 응답하지 못할 때는 예외 상황이 발생한 것으로 해석하고, 발생한 예외 상황은 대체 웹 서비스를 찾거나 호출함으로써 처리한다.

2.2 워크플로우 및 웹 서비스

워크플로우는 특정 비즈니스 목표를 이루기 위해서 실행되는 일련의 단계로서 이런 단계는 보통 해당 비지니스 목표를 달성할 때까지 오랜 시간 지속되는 경우가 많다. 따라서 특정 비즈니스 프로세스에 관련된 워크플로우의 작업이 종료되기까지 어느 정도 시간이 소비되는 것은 보통이다.

한편 워크플로우에도 제어흐름, 데이타흐름, 사양이 모두 기술되기 때문에 일반 프로그래밍 언어와 많은 공통된 특징을 가지고 있다. 워크플로우와 일반 프로그래밍 언어의 차이점은 각 단계에서 기술 (記述) 되는 작업의 크기에 있다. 워크플로우 내 단계들은 각 분야 전문가들이 쉽게 이해 할 수 있을 정도의 수준으로 기술되야 하기 때문에, 각 단계의 크기는 충분히 커야한다. 워크플로우의 이러한 면은 웹 서비스의 특성과도 잘 일치된다. 왜냐하면 각 웹 서비스의 크기도 원격 웹 서비스 호출 비용을 상쇄할 수 있을 정도로 충분히 커야 하기 때문이다.

충분한 크기를 갖는 워크플로우 각 단계의 특성은 워크플로우가 일반 프로그래밍 언어와 비교해서 보다 더 유연하게 제어 처리를 할 수 있는 여지를 제공한다. 즉 워크플로우에 있어서는 한 단계의 실행이 종료될 때마다 실행에 관한 제어권은 워크플로우 시스템에게 넘어간다. 그러면 워크플로우 시스템은 다음 실행될 단계를 결정하고, 제어권을 다시 해당 단계에 넘긴다. 이 같은 제어권 교환은 해당 워크플로우 실행이 종료시까지 워크플로우 시스템과 워크플로우 내 단계 사이에서 반복적으로 일어나게 된다. 결과적으로는 워크플로우 시스템은 프로세스가 실행되는 동안 개입이 용이하고, 필요한 경우에는 손쉽게 적절한 조치를 취할 수도 있다. 한편 일반 프로그램이 운영체제 상에서 수행되는 경우에는 양상이 다소 다르다. 프로그램의 각 스텝의 실행이 종료되면 언제나 그 제어권이 운영체제로 넘어가지 않기 때문이다. 즉 프로그램 내에 운영체제와 같은 외부 시스템에 제어권을 넘긴다는 명백한 명세가 없다면 제어권은 계속 프로그램 상에 남게 된다.

위와 같은 상황을 고려한다면, 웹 서비스를 워크플로우의 한 단계로 생각할 수 있다. BPEL4WS[5]의 사양에서와 같이 웹 서비스 상에서의 많은 표준화 작업과 보고서 등이 이와 같은 아이디어를 기반하여 만들어지고 있다. 따라서 프로그램에서 웹 서비스를 호출하는 경우 워크플로우가 개입하는 것은 적절한 것으로 판단된다. 즉 응용 프로그램에서 웹 서비스를 호출하는 경우 웹 서비스 호출을 담당하는 워크플로우가 생성되고, 해당 응용 프로그램은 생성된 워크플로우를 호출함으로써 원하는 웹 서비스를 호출하는 것이다.

이 방법을 사용함으로 여러 가지 장점을 기대할 수 있다. 우선, 웹 서비스를 직접 호출하는 것에서 워크플로우를 이용한 간접적인 호출 방법으로 변경함으로서,

워크플로우 시스템 상의 많은 장치들은 웹 서비스 호출 응용 프로그램을 위한 QoS(Quality of Service)와 관계된 이슈들을 처리 하는데 사용될 수 있다. 워크플로우 시스템에서 제공 가능한 트랜잭션 지원 기능, 예외 상황 핸들링 메커니즘 그리고 기타 워크플로우 시스템이 제공 가능한 기능들은 웹 서비스 호출의 신뢰성을 높이는데 사용 할 수 있다. 즉, 웹 서비스 환경에 변화가 있을 경우, 주변 시스템에 영향을 주지 않고 즉각적으로 웹 서비스 호출에 반영할 수 있다. 하지만, 이 방법은 웹 서비스의 호출 과정에서 어느 정도 성능적(예를들면, 시간) 손실을 가져온다. 이 같은 성능상의 손실은 웹 서비스를 호출하는 상황에 따라서 수용 될 수도, 그렇지 않을 수도 있다. 그렇기 때문에 우선적으로 성능적 손실의 수용 한도를 알 필요가 있다.

동적 서비스 연결 기술은 워크플로우나 응용 프로그램의 한 레벨에서 적용될 수 있다. 다시 말하면, 동적 서비스 연결을 이용하여 워크플로우의 한 레벨에서 웹 서비스를 호출하도록 의뢰할 수도 있고, 응용 프로그램이 웹 서비스를 호출 할 수 있다. 본 논문에서는 동적 서비스 연결이 응용 프로그램 레벨에서 지원이 가능하면, 워크플로우 레벨의 동적 서비스 연결 또한 쉽게 지원되기 때문에 응용 프로그램 레벨에서의 동적 서비스 연결 기법에 중점을 두어 설명한다.

2.3 관련연구

웹 서비스의 동적 서비스 연결에 대한 연구는 그 필요성에도 불구하고 많은 연구가 이루어지지 못하였다. 다만 웹 서비스 정보의 동적인 처리를 위한 응용 레벨에서의 연구가 있다. 문헌 [6]에서는 다양한 웹 콘텐츠 처리를 하기 위해서 동적 조정 모델을 구성하고, 처리 방법론에 관하여 기술하고 있다. 이 연구에서는 웹 콘텐츠 정보를 빠르고, 효과적으로 처리하기 위해서 다층 구조 조정 메커니즘을 적용하고 있다. 또한 동적 서비스 컴포넌트 연결을 위한 아키텍처 및 프로토콜에 관한 연구가 진행되고 있다[7,8].

워크플로우 동적 변경과 관련해서는 다수의 연구가 수행되었다. 많은 연구자들이 워크플로우 관리 시스템의 유연성 향상을 위하여 노력해 왔는데, 자동적이고 동적인 워크플로우 변경에 대응하기 위해서 에드 혹(ad-hoc) 변경 및 ECA(Event-Condition-Action) 를 기반으로 문제를 해결하기 위한 연구가 있었다[9-11]. 이외에 90년대 후반부터 이종성과 상호 연동 문제를 해결하기 위해서 조직간 경계선을 넘는 워크플로우 구조에 관한 연구도 진행되어 왔다[12-14]. 최근에는 동적 워크플로우 상호 연동을 위한 다중 하위 프로세스 작업 기반의 프레임워크에 관한 연구도 진행되고 있다[15].

한편 기존의 워크플로우에서 다루었던 예외 상황 핸들링 문제 및 워크플로우 예외 상황 핸들링 메커니즘을 웹 서비스에 적용하는 연구도 진행되어 왔다. 문헌 [16,17]에서는 이런 예외 상황에 적절하게 대응하기 위해서 예외 상황을 명시하고, 핸들링 루틴의 연결을 명시할 수 있는 기능 등에 관하여 기존에 워크플로우 시스템 내 예외 상황 핸들링 메커니즘에 관한 연구가 진행되어 왔다. 여기에서 명시된 예외 상황은 워크플로우 시스템에 전달 되고, 시스템은 프로세스 실행 중 명시된 예외 상황의 발생을 감지한다. 발생 된 예외 상황에 대해서는 해당 핸들링 루틴을 호출하여 수행한 뒤, 프로세스 설계자가 명시한 바에 의해서 프로세스의 적절한 곳으로 제어를 넘겨주게 된다. 문헌 [18]에서는 제한된 룰을 기반으로 한 워크플로우의 예외 상황 핸들링을 함으로써 웹 서비스로의 적용을 시도하고 있다. 여기에서는 워크플로우를 웹 서비스에 적용하기 위해서 동적인 예외 상황 핸들링에 대한 제한된 룰을 기술하고 있다. 또한 문헌 [19]에서는 웹 서비스 품질을 기반으로 한 예외 상황 핸들링의 연구가 있었는데, 웹 서비스 품질을 위해서 단지 웹 서비스 상에 몇몇 제한된 룰을 적용하였다. 그러나 이런 제한 된 룰에 의한 시도는 다양하게 변화하는 웹 서비스 환경에 적용하기에는 한계가 있었다. 이외에도 워크플로우 시스템에 멀티 에이전트 개념을 적용하여 예외 상황 핸들링을 함으로써 웹 서비스에 적용하려는 시도가 있었다[20].

3. 동적 서비스 연결 프레임워크

3.1 동적 서비스 연결을 지원하기 위한 두 가지 접근 방법

응용 프로그램이 UDDI를 활용하지 않고 동적 서비스 연결 기능을 갖기 위해서는 조건 및 상태 체크 루틴, 교체 웹 서비스의 선택 및 호출 루틴 그리고 동적 서비스 연결 호출 지원과 관련있는 기타 루틴들이 해당 응용 프로그램 내에서 적절하게 호출되어야 한다. 프로그래밍 언어에서 제공되는 예외 상황 핸들링 메커니즘을 이와 같은 루틴들을 한 데 묶는데 사용하는 것도 가능하다. 이런 메커니즘을 활용하여 동적 서비스 연결을하는 것은 직관적으로 이해하기 쉽다는 점에서는 장점이 있지만 웹 서비스 환경이 지속적으로 변화한다는 점을 고려할 때 적절한 접근 방법이 되지는 못한다. 그림 1에서 보는 바와 같이 많은 수의 새로운 웹 서비스가 끊임없이 생성되고, 기존의 웹 서비스들이 아무런 통보없이 사라지는 상황에서, 이와 같은 변화들을 응용 프로그램에 즉각 반영하는 것을 기대하기는 어렵다. 그리고 동적 서비스 연결 루틴들을 구현하는 것도 그리 간단하지 않다.

이런 결함을 피하기 위해서 생각할 수 있는 것은 웹 서비스를 호출하는 응용 프로그램에서 동적 서비스 연

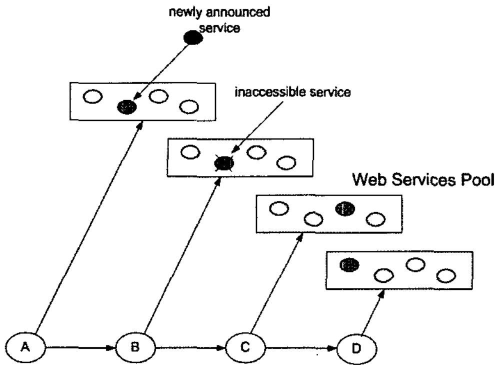


그림 1 웹 서비스 호출 응용 프로그램

결을 지원하는 부분을 분리하여 관리하는 것이다. 이 방법을 사용하면 응용 프로그램 내에서 웹 서비스를 호출하기 전에 신뢰성 있는 웹 서비스 호출을 미리 준비할 수 있다는 점에서 장점이 있다. 웹 서비스 호출에 필요한 속성들은 다이얼로그 박스와 같은 사용자 인터페이스를 통해서 명시하고, 동적 서비스 연결 기능을 지닌 웹 서비스 호출 클래스는 명시된 속성 값들을 이용하여 생성한다. 응용 프로그램은 생성된 클래스를 사용하여 웹 서비스를 호출 할 수 있고, 필요한 경우에는 클래스 내 메소드의 호출을 통해서 부가 서비스를 사용하는 것도 가능하다. 동적 서비스 연결 기능이 생성된 클래스에 이미 구현되어 있기 때문에, 웹 서비스 호출 시 문제가 발생하면 해당 클래스가 대체 웹 서비스의 호출을 시도한다. 그림 2는 이와 같은 일련의 과정을 보여주고 있다. 하지만 이 방법 역시 몇 가지 제약이 있다. 예를 들면, 사용 가능한 새로운 웹 서비스를 반영하거나 후보 리스트에서 특정 웹 서비스를 삭제하기 위해서는 웹 서비스 호출 클래스가 새로이 생성되어야만 하고, 클래스의 메소드를 호출하는 응용 프로그램은 다시 컴파일되

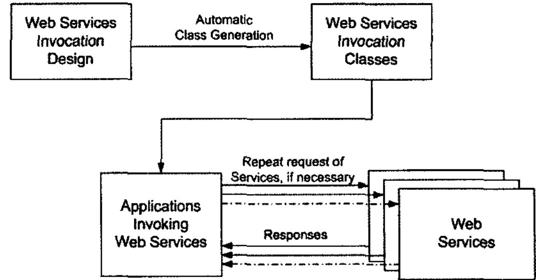


그림 2 웹 서비스 호출 클래스 및 동적 서비스 연결

어야 한다.

한편, 만약 웹 서비스 호출 시에 발생하는 성능적 손실을 어느 정도 감안할 수 있다면, 보다 더 유연한 웹 서비스 호출이 가능하다. 그림 3은 워크플로우를 이용한 가장 유연한 웹 서비스 호출 방식을 보여주고 있다. 그림 3의 방식에서는 먼저 유사한 방식으로 응용 프로그램을 대신하여 웹 서비스 호출하는 워크플로우가 생성되고, 응용 프로그램은 생성된 워크플로우를 호출함으로써 원하는 웹 서비스를 호출한다. 웹 서비스 호출 클래스를 생성하는 것 대신에 워크플로우 호출 클래스를 생성하고, 또한 클래스 인스턴스를 호출함으로서 워크플로우 인스턴스가 생성되고 실행된다는 점에서 그림 2의 방식과 다르다.

워크플로우를 이용하여 웹 서비스를 호출하면 다음과 같은 몇 가지 장점을 기대할 수 있다. 우선, 예외 상황 핸들링 및 동적 변경과 같은 워크플로우 시스템이 제공하는 메커니즘들을 이용해서 동적 서비스 연결이 가능하다는 점이다. 예를 들어, 새로운 웹 서비스가 생기거나 또는 존재하던 웹 서비스가 없어지는 경우, 다시 말하면 웹 서비스 환경의 변화가 워크플로우 시스템의 예외 상황 핸들링 매커니즘을 통해서 쉽게 기술되고 처리될 수 있다. 즉, 웹 서비스 환경이 변화되어도 응용 프로그램에 아무런 영향을 미치지 않고 즉각적으로 변화

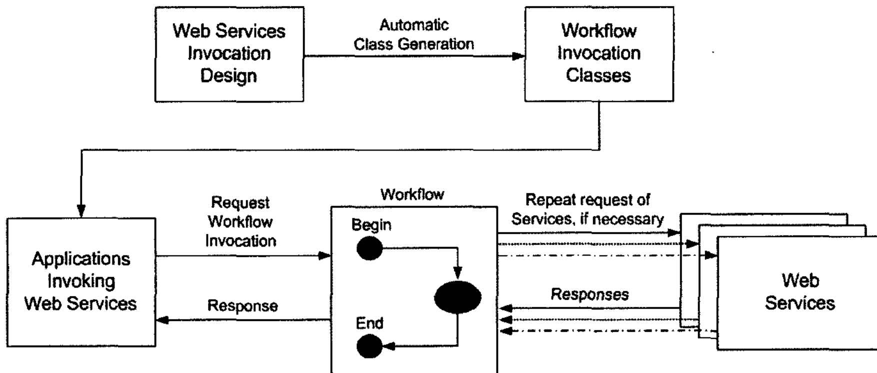


그림 3 워크플로우 호출 클래스 및 동적 서비스 연결

를 반영할 수 있다. 이 방식에서는 응용 프로그램의 컴파일 및 워크플로우 호출 클래스의 재 생성 과정은 생략 가능하다. 또한 워크플로우 시스템의 예외 상황 핸들링 메커니즘을 사용할 경우 웹 서비스를 프로세스 레벨로의 취급하는 의미를 갖게된다. 웹 서비스가 독립성이 강하면서 성긴 형태의(coarse grained) 서비스 통합을 지향한다고 볼 때, 웹 서비스를 프로세스 레벨에서 취급하는 것은 웹 서비스의 특성과 잘 일치된다. 결국, 워크프로우 시스템의 예외 상황 핸들링 메커니즘을 사용하면 동적 서비스 연결을 좀 더 유연하게 지원할 수 있음을 의미한다. 하지만 워크플로우를 이용한 웹 서비스 호출은 웹 서비스 호출 과정에서 상당한 성능적 손실을 유발시킨다는 점에서 문제가 있을 수 있다. 따라서 웹 서비스 호출 과정이 전체 성능에 큰 영향을 미친다면 이 방법은 적절하지 않게 된다.

3.2 웹 서비스 호출 설계

웹 서비스가 응용 프로그램을 이용하여 호출될 때 몇 가지 속성들의 결정 및 명세화 작업이 이뤄져야 한다. URL위치, 클래스 이름, 메소드 이름, 메소드 사용법 등은 웹 서비스를 호출하기 위해서 반드시 필요한 정보들이다. 웹 서비스 호출을 설계할 때 이와 같은 정보의 명세화 작업은 꼭 필요한 사항이다. 또한, 웹 서비스 호출에 동적 서비스 연결 기능을 추가하기 위해서는 동적 서비스 연결을 포함하는 몇 가지 관련 정보들도 명시되어야 한다. 표 1은 동적 서비스 연결을 위한 속성들을 보여준다. 표에서 속성들은 텍스트형식으로 설명되어 있지만 실제로는 다이얼 로그 박스 등을 통해서 손쉽게 명기될 수 있다. 클래스는 모든 속성이 설정된 후 명기된 속성 정보를 기반하여 생성되고, 생성된 클래스는 웹 서비스 호출 루틴 및 웹 서비스 연결 함수를 포함한다. 앞에서 언급한 바와 같이 응용 프로그램 개발자는 생성된 클래스를 이용하여 보다 쉽게 동적 서비스 연결 기능을 응용 프로그램에 접목시킬 수 있다. 다음 절에서는 설정된 속성들의 정보를 이용해 어떻게 웹 서비스 호출 클래스와 워크플로우 호출 클래스를 만들 수 있는지에 대해 알아본다.

3.3 웹 서비스 및 워크플로우 호출 클래스 생성

모든 필요한 속성값들이 다이얼로그 박스를 통하여 명기된 후, 명기된 정보들은 추후 사용을 위하여 XML (eXtensible Markup Language) 문서형식으로 저장된다. 저장된 XML문서는 앞서 기술한 바와 같이 문서 내에 명시된 조건과 옵션을 만족하는 클래스를 생성하는데 주로 이용된다. 예를 들면, 속성 값 중 클래스 타입의 값에 따라 웹 서비스 혹은 워크플로우 호출 클래스가 생성된다. 클래스의 생성은 비교적 쉽고 간단한 작업이지만, 클래스 생성을 지원하는 모듈이 준비되어야 한다. 클래스 생성 모듈은 우선, XML문서를 해석(parsing)한 후, 클래스 생성에 필요한 정보들을 뽑아내고, 뽑아낸 정보를 이용해 클래스를 생성한다.

그림 4는 웹 서비스 호출 클래스를 자동으로 생성해주는 가장 간단한 클래스 템플릿 중의 한 예를 보여주고 있다. 클래스 생성 모듈은 XML 문서의 파싱을 통해서 얻은 속성 값들을 '%'문자 사이의 문자열 위에 덮어씀으로써 완성된 코드를 얻게 된다. 그 밖의 클래스 템플릿들은 속성 값들에 따라서 다르게 표현된다. 모든 클래스 템플릿은 웹 서비스 호출 부분이 구현되어 있는 루틴 내부의 *WebServiceInvoker* 클래스의 하위 클래스가 된다. 따라서 *WebServiceInvoker*의 *public* 메소

표 1 웹 서비스 호출 디자인을 위한 속성들

속 성	설 명
웹 서비스 교체 (Replacing Web services)	웹 서비스 호출이 실패 했을 경우, 후보 웹 서비스들의 리스트를 명시한다.
실행모드 (Execution mode)	웹 서비스 실행 방법을 명시한다. 이 옵션으로 병렬적 실행 모드와 순차적 실행모드 중 하나를 설정할 수 있다.
대기시간 (Time out)	통신하고 있는 웹 서비스로부터의 응답 수신 기한 시간을 명시한다. 제한된 시간 내에 응답을 받지 못했을 경우, 후보 웹 서비스로 대체된다.
UDDI 갱신 체크 (UDDI update checking)	웹 서비스로부터 응답을 받지 못한 경우, UDDI의 갱신을 확인해야 하는 것인지 하지 않아야 하는 것인지 나타낸다.
재시도 (Retrial)	웹 서비스로의 응답을 받지 못한 경우, 모든 후보 리스트내의 웹 서비스에게 재시도를 할 것인지 하지 않을 것인지 나타낸다. 재시도 횟수 또한 명시한다.
전제/후속 조건 (Pre/Post condition)	웹 서비스 호출 전후의 조건들을 명시한다. 해당 조건이 만족되지 않는다면, 예외 상황이 발생되고 통보된다.
예외 상황 모드 (Exception mode)	예외 상황 핸들링 후의 제어 상태를 나타낸다. 시스템에서 지원하는 모드는 resume, retry, ignore, break, terminate이다.
클래스 이름 (Class name)	생성된 클래스의 이름을 명시한다.
클래스 타입 (Class type)	생성된 클래스의 타입을 명시한다. 웹 서비스 호출타입과 워크플로우 호출타입 중 하나를 사용자가 선택할 수 있다.

```
import rwsc.*;

public class RWSC_Sample extends WebServicesinvoker {

        public RWSC_Sample() {
                super();

                setPreCondition("%pre-condition%");
                setPostCondition("%post-condition%");
                setRetrialCount(%retrial-count%);
                enableUDDIChecking();
        }

          protected Object invokeBasicCall() throws Exception {
                return %class-name%.%operation-name%.(%parameter-list%);
        }

          protected Object invokeCandidateCallsAll() throws Exception {
                try {
                        return %class-name%.%operation-name%.(%parameter-list%);
                }
                catch(Exception ignored) { }

                try {
                        return %class-name%.%operation-name%.(%parameter-list%);
                }
                catch(Exception ignored) { }

                return %class-name%.%operation-name%.(%parameter-list%);
          }

}
```

그림 4 신뢰성 있는 웹 서비스 호출을 위한 클래스 템플릿 예

드가 호출되면, *RWSC_Sample* 클래스 내의 *protected* 메소드가 호출됨으로 신뢰성 있는 웹 서비스의 호출이 가능하게 된다.

워크플로우 호출 클래스 생성의 경우에는 워크플로우 및 이와 관련된 예외 상황 핸들링 루틴들이 호출에 응할 수 있도록 미리 준비되어 있는 것으로 가정한다. 워크플로우의 예외 상황 핸들링 루틴은 전제·후속 조건들 및 예외 상황 모드의 속성값들을 기반으로 생성되며, *resume, retry, ignore, break, terminate*의 5가지 예외 상황 모드 중 하나가 선택되어 명시된다. 예외 상황에 대한 자세한 설명은 4장에서 다루기로 한다.

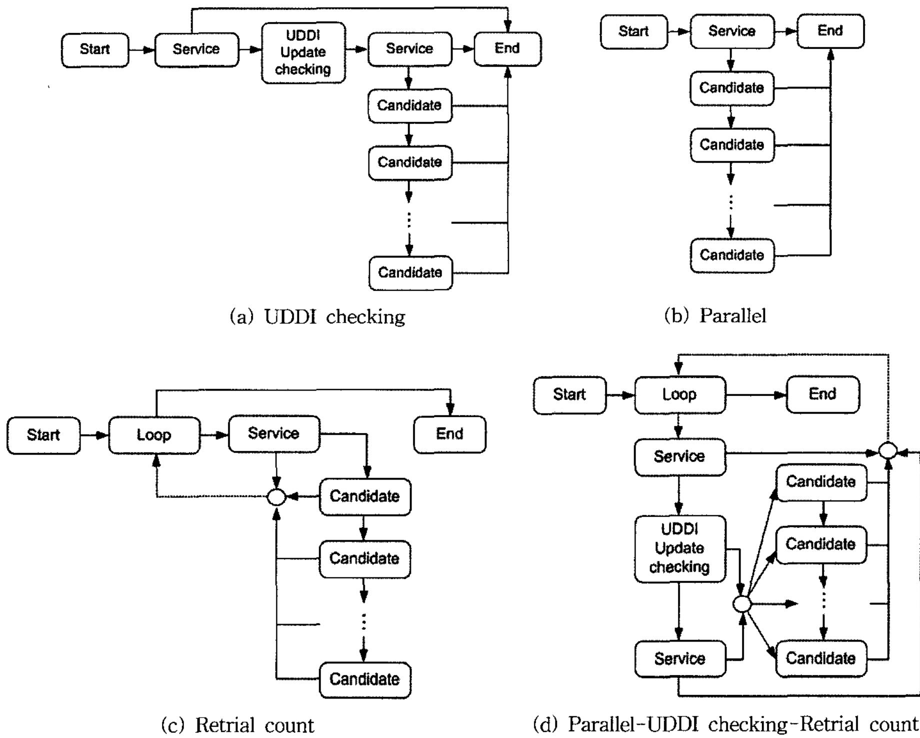
워크플로우 패턴은 총8가지가 있으며, 속성에서 선택된 내용에 따라서 그 중 하나의 패턴이 선택된다. 선택 가능한 옵션에는 UDDI 저장소 검색, 재시도 횟수(Retrial count), 병렬 모드(Parallel mode) 등이 있다. 그림 5에서는 지면 관계상 이들 8개의 패턴 중에서 4개의 패턴을 선택하여 보여주고 있다. 그림 (a)는 UDDI 저장소 검색 옵션이 선택된 경우에 나타나는 패턴이다. UDDI저장소 검색 단계는 웹 서비스가 호출되는 서비스 단계 이후에 삽입된다. UDDI 저장소 검색 단계에서는 URL위치나 연산 인터페이스가 변경된 경우와 같은 UDDI 저장소의 갱신 등을 검사한다. (b)는 병렬 옵션이 선택되었을 때를 나타낸다. 병렬 옵션이 선택되면, 명시된 모든 대체 웹 서비스들에게 동시에 서비스 요청을 하고 그 중 하나가 응답을 하면 요청이 성공한 것으로 간주한다. (c)는 재시도 횟수 옵션에 해당하는 패턴이다. 재시도 과정은 명시된 횟수 동안 반복한다. 위 세 가지 옵션을 모두 선택하면, 패턴 (d)가 선택된다. 동적 서비스 연결을 지원하기 위해서 많은 부분을 워크플로우가 담당하지만, 워크플로우 호출 클래스의 구조나 그 내용은 비교적 간단하다.

3.4 웹 서비스 호출

3.3절에서 언급한 클래스의 생성이 성공적으로 이뤄졌다면, 응용 프로그램에서는 생성된 클래스를 활용하여 웹 서비스의 신뢰성 있는 호출을 손쉽게 구현할 수 있다. 앞에서도 언급하였지만 사용자의 선택에 따라 두 종류의 클래스가 생성되는데 하나는 웹 서비스 호출 클래스이고, 다른 하나는 웹 서비스 호출 단계를 포함한 워크플로우 호출 클래스이다. 각 클래스에서 웹 서비스를 호출할 때 걸리는 오버헤드가 서로 다르기 때문에 클래스 선택에 주의를 기울여야 한다. 오버헤드에 대한 부분은 4장에서 좀 더 자세히 다루기로 한다. 사용 할 클래스가 결정되면, 생성된 클래스를 응용 프로그램에 포함시키고 클래스 내 함수를 호출하면 동적 서비스 연결 기능이 포함된 응용 프로그램 구성이 완료된다. 따라서 동적 서비스 지원 기술을 응용 프로그램에 포함시키는 일은 비교적 간단하다.

4. 워크플로우 시스템의 예외 상황 핸들링

본 논문에서는 응용 프로그램 내에서 워크플로우 호



(a) UDDI checking (b) Parallel

(c) Retrial count (d) Parallel-UDDI checking-Retrial count

그림 5 워크플로우 패턴

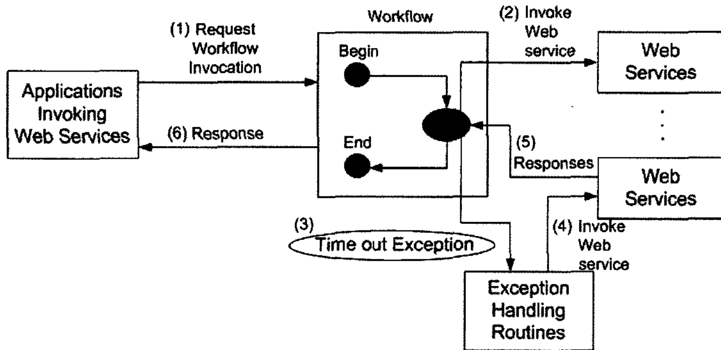


그림 6 워크플로우 시스템의 예외 상황 핸들링 메커니즘을 이용한 동적 서비스 연결 지원

출 클래스를 이용해 웹 서비스가 호출 될 때, 동적 서비스 연결을 지원하기 위해서 워크플로우 시스템 내 예외 상황 핸들링 메커니즘을 사용하기도 한다. 그림 6은 응용 프로그램에서 워크플로우 호출 클래스를 이용해 웹 서비스를 호출할 때 타임아웃 예외 상황이 발생하는 단계적 흐름을 보여준다. 그림에서 보여 주는 것은 워크플로우 예외 상황 핸들링 메커니즘이 동적 서비스 연결에 활용될 수 있다는 것을 개념적으로 보여주기 위한 것으로 동적 서비스 연결이 꼭 이런식으로 구현되어야 함을 의미하는 것은 아니다. 호출하는 동안 타임아웃 예외 상황이 발생하면, 워크플로우 시스템은 핸들링 루틴을 호출하게 되고 호출된 핸들링 루틴은 다시 대체 웹 서비스를 호출한다. 대체 웹 서비스의 호출작업이 성공하면, 결과값은 워크플로우를 통해 응용 프로그램으로 전달되고, 실패 할 경우 또 다른 대체 웹 서비스의 호출을 시도한다. 하지만 핸들러가 취할 수 있는 방법은 다양하기 때문에 웹 서비스 호출의 설계 과정에서 명시된 속성들의 정보를 이용해야 한다. 즉, 속성 정보를 기반으로 하여 핸들링 루틴이 자동 생성되고, 또한 생성된 워크플로우에 연결된다.

물론 워크플로우 시스템마다 각각 자체의 예외 상황 핸들링 메커니즘이 있기 때문에 예외 상황을 핸들링하기 위한 메커니즘 또한 서로 다를 수 있다. 본 장에서는 워크플로우 시스템 중 WebVine[16]이라는 시스템의 예외 상황 핸들링 메커니즘을 중심으로 동적 서비스 연결 방식을 워크플로우 시스템과 연계시키는 방안에 대하여 설명한다.

WebVine 예외 상황 핸들링 메커니즘은 두가지 중요한 부분으로 구분되는데 예외 상황 명세화 부분과 핸들링 부분이다. 예외 상황 명세화 부분에서는 *modes, models, triggering conditions, exception handler*가 기술 되며, 핸들링 부분에서는 실행 중에 예외 상황이 발생하면 그에맞는 예외 상황 핸들러가 호출되고, 미리 준비되어 있는 예외 상황 핸들링 루틴을 수행하게 된다. 다음의 두 절에서는 이 두 부분을 좀 더 구체적으로 설명하고 동적 서비스 연결을 지원하기 위해서 기존의 예외 상황 핸들링 메커니즘을 적용 또는 확장하는 방안에 대해서 기술한다.

4.1 WebVine의 예외 상황 명세화

WebVine의 예외 상황 명세화 부분에서는 각 예외 상황을 정의하기 위해 다양한 속성들이 정의되어 있다. 동기 모드(triggering mode), 예외 상황 발생을 위한 조건들, 예외 상황 핸들링 모드 및 그에 따른 단계들이 정의되어 있다(예외 상황 속성에 대한 자세한 설명은 문헌[7]을 참고). 사용자가 정의한 예외 상황 핸들링 루틴 역시 이곳에 포함되어 있다. 동적 서비스 연결은 다음과 같은 *Handling Mode, Handling Model, Exception Handler, Timeout Condition* 들을 적절히 결합하여 명시한다.

- *Handling Mode* - 예외 상황 핸들링 프로세스를 동기 하거나 종료 할 때의 자동화 정도를 명시한다. *Automatic Mode*는 예외 상황이 워크플로우 시스템에서 완벽하게 조정되는 모드를 말하며, *Manual Mode*는 사용자에 의해 예외 상황이 조정 된다. 동적 서비스 연결 스텝에서 기존의 웹 서비스와 잘 대응하는 대체 웹 서비스가 자동적으로 선택되기 때문에 이 속성값은 *Automatic Mode*로 설정되어야 한다.
- *Handling Model* - 그림 7에서 보는 바와 같이, 예외 상황을 핸들링하는 모델을 5가지 모델(*resume, retry, ignore, break, terminate*)로 분류할 수 있다. 서비스의 대체작업이 이뤄진 후에도 동적 서비스 연결은 계속해서 워크플로우를 실행하기 때문에 resume 모드로 설정되어야 한다.
- *Exception Handler* - 감시 블록(guarded block) 내에서 액티비티들이 실행되는 동안에 예외 상황이 감지되었을 경우, 이때 실행되는 엑티비티의 집합을 예외 상황 핸들러라 정의한다. 동적 서비스 연결에서 사용되는 예외 상황 핸들러는 적합한 웹 서비스를 선택하고, 도중에 전달되는 결과 값들을 저장하며, 기존의 서비스를 새것으로 대체하는 작업과 저장한 결과 값들을 이용해 새로운 웹 서비스를 활성화 시키는 등의 작업을 한다.

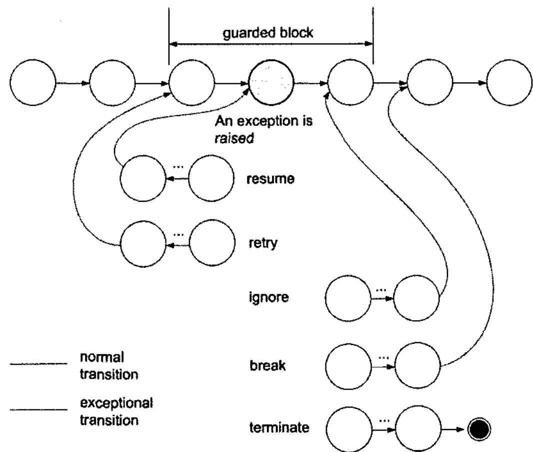


그림 7 WebVine 시스템에서 지원하는 5가지 예외 상황 핸들링 모델

- *Timeout Condition* - 이 속성은 WebVine이 지원하는 예외 상황 핸들링 메커니즘에서 동적 서비스 연결에 꼭 필요한 조건이다. WebVine은 각 단계의 수행 시간을 체크한다. 명시된 시간 내에 해당 단계 수행이 종료되지 않으면 타임아웃 예외 상황이 발생한다. 이 속성은 워크플로우 내 한 단계에서 이루어진 웹 서비스가 호출 요청에 대한 응답이 지연되는 경우에 예외 상황을 발생시키는 데 활용된다.

그림 8의 DTD(Document Type Definition)는XPDL (XML Process Definition Language)에서의 예외 상황 속성을 표현하기 위한 구문이다.

워크플로우 내에서 동적 서비스 연결을 표현하기 위해 필요한 주요 요소들은 다음과 같다.

- **Activity** - Activity에 의해서 예외 상황은 동적 서비스 연결을 활성화 시킨다. OnExceptions의 요소는 0개 혹은 한개 이상의 속성을 지닐 수 있다. 각각의 OnException 요소는 Id, Name, RetryCount와 같은 속성들을 지닌다.
- **Transitions** - ExceptionalTransition의 요소는 예외 상황이 발생한 경우, 동적 서비스 연결 단계로 이르는 길을 명시한다.
- **Exceptions** - 이 요소를 통해서 동적 서비스 연결을 동기화하는 예외 상황의 집합이 정의되며, 각각의 예외 상황은 Exception의 요소 내에서 표현된다.

```
...
<!ELEMENT Package (PackageHeader, ···, Exception?, ExtendedAttributes?)>
<!ELEMENT WorkflowProcess (ProcessHeader, ···, Exceptions?,
        GuardeBlocks?, ExtendedAttributes?)>
<!ELEMENT Activity (Description?, ···, PreCondition, PostCondition,
        ExtendedAttributes?)>
<!ELEMENT PreCondition (#PCDATA | Xpression)*>
<!ELEMENT PostCondition (#PCDATA | Xpression)*>
<!ELEMENT Transitions (Transition*, ExceptionalTransition*)>
<!ELEMENT ExceptionalTransition (Condition?, Description?,
        ExtendedAttributes?)>
<!ATTLIST ExceptionalTransition
        Id NMTOKEN #REQUIRED
        From NMTOKEN #REQUIRED
        To NMTOKEN #REQUIRED
        Loop (NOLOOP | FROMLOOP | TOLOOP) #IMPLIED
        Name CDATA #IMPLIED>

<!ELEMENT Exceptions (Exception*)>
<!ELEMENT Exception (Description?)>
<!ATTLIST Exception
        Id NMTOKEN #REQUIRED
        Name CDATA #IMPLIED>
<!ELEMENT GuardedBlocks (GuardedBlock*)>
<!ATTLIST GuardedBlock
        Id NMTOKEN #REQUIRED
        From NMTOKEN #REQUIRED
        To NMTOKEN #REQUIRED
        Name CDATA #IMPLIED>
<!ELEMENT OnExceptions (OnException*)>
<!ELEMENT OnException (RetryCount?)>
<!ATTLIST OnException
        Id NMTOKEN #REQUIRED
        Name CDATA #IMPLIED>
<!ELEMENT RetryCount (#PCDATA)>
...
```

그림 8 XPDL내의 예외 상황 속성들을 표현하기 위한 DTD

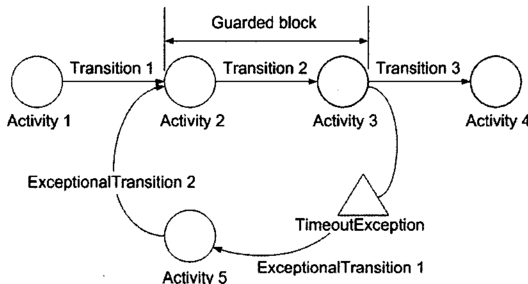


그림 9 가이드 블록 내에서 예외 상황 핸들링 시나리오

- **GuardedBlocks** - 예외 상황이 발견되고, 동적 서비스 연결 단계가 활성화되는 영역을 명시한다. 액티비티 범위는 각 감시 블록 내 From, To로서 정해진다.

그림 9는 예외 상황 핸들링 메커니즘이 어떻게 동적 서비스 연결을 지원하기 위해 사용되는지를 설명하는 시나리오이다. 먼저TimeoutException이 감시 블록에 첨가된다. TimeoutException은 액티비티 3(Activity 3)이 실행되는 도중에 발생하게 되면, ExceptionTransition 1로 지정된 경로를 따라서 제어가 실행되어 예외 상황 핸들러(동적 서비스 연결이자)인 액티비티 5(Activity 5)를 호출하게 된다. 이 그림에서 원은 액티비티를 나타내며 삼각형은 예외 상황을 나타낸다. 웹 서비스 대체 작업이 한번에 하나의 엑티비티 만을 수행할 수 있기 때문에 한 감시 블록에 하나의 엑티비티 만을 포함하게 된다.

```
<Package Id="1" Name="Sample Model">
...
<WorkflowProcesses>
 <WorkflowProcess Id="1" Name="Sample Process1">
...
<Activities>
 <Activity Id="1" Name="Activity1"> ··· </Activity>
 <Activity Id="2" Name="Activity2"> ··· </Acticity>
 <Activity Id="3" Name="Activity3"> ··· </Acticity>
 <Activity Id="4" Name="Activity4"> ··· </Acticity>
</Activities>
<Transitions>
 <Transition Id="1" From="1" To="2" Name="Transition1"/>
 <Transition Id="2" From="2" To="3" Name="Transition2"/>
 <Transition Id="3" From="3" To="4" Name="Transition3"/>
 <ExceptionalTransition Id="3" From="3" To="5" Name="ExceptionalTransition1"/>
 <ExceptionalTransition Id="5" From="5" To="2" Name="ExceptionalTransition2"/>
</Transitions>
<Exceptions>
 <Exception Id="1" Name="TimeoutException">
 <Description> Timeout occurred </Description>
 </Exception>
</Exceptions>
<GuardedBlocks>
 <GuardedBlock Id="1" From="2" To="3">
<OnExceptions>
```

그림 10 XPDL로 표현한 예외 상황 핸들링 예

그림 10은 위의 예외 상황 핸들링 시나리오를 XPDL로 표현하였다.

4.2 동적 서비스 연결에 WebVine 예외 상황 핸들링 메커니즘의 적용

4.1절에서 사용자가 명시한 속성 값을 기반으로 예외 상황 발생 조건이나 핸들링 루틴이 반영된 웹 서비스 호출 워크플로우가 만들어지면 다음 절차에 따라 동적 서비스 연결이 이루어진다. 그 과정을 설명하면 다음과 같다.

- 웹 서비스를 호출하는 단계를 포함한 워크플로우를 생성한다.
- 이때 웹 서비스 호출의 설계 과정에서 미리 정해진 속성 값들이 워크플로우 생성에 반영된다. 예외 상황 핸들링 루틴도 이 속성에 따라 생성되고 생성된 워크플로우와 연결된다.
- 응용 프로그램을 대신해 워크플로우에서 웹 서비스 호출을 일으킨다.
- 웹 서비스 호출이 성공적이면 종료한다.
- 그렇지 않으면 예외 상황이 발생하고 제어를 예외 상황 핸들러에 넘기고 처리 의뢰. 예외 상황 핸들러의 처리가 종료되면 제어는 모드에 따라 워크플로우 적절한 곳으로 이동한다.

5. 구현 및 평가

5.1 구현 환경 및 사양

본 논문에서 제안한 프레임워크를 이용하여 웹 서비스 호출이 이루어지는 경우의 경과되는 시간을 측정하기 위하여 일부 시스템을 구현하였다. 다음은 시스템 구현 환경 및 측정 시 사용된 소프트웨어 및 하드웨어 사양이다.

• 소프트웨어 환경 및 사양
  - 이클립스 (Eclipse)
  - 자바 (J.D.K) v1.4
  - 웹 서비스 서버: Axis v1.1
  - 웹 서버: 톰켓 (Tomcat) v5.0
  - 워크플로우 엔진: 웹바인 (WebVine) v1.0
• 하드웨어 환경 및 사양
  - Local 테스트: 펜티엄4, 2.4-GHz, 1-CPU/1G-메모리
  - 원격LAN 테스트: 펜티엄3, 1-GHz, 2-CPU/1G-메모리
  - 원격Internet 테스트: 펜티엄4, 2-GHz, 2-CPU/1G-메모리

5.2 평가

본 논문에서 제안한 동적 서비스 연결 프레임워크를 이용하여 웹 서비스 호출에 걸리는 경과 시간을 측정한 결과 및 평가를 실행하였다. 웹 서비스 호출을 위한 디자인 과정에서 사용되는 다이얼로그 박스는 Lavadora 프로젝트(http://lavadora.sourceforge.net/)에서 제공하는 툴을 이용하여 구현하였다. Lavadora프로젝트의 목표는 웹 서비스 분야의 상업용 개발환경이 제공하는 기능을 반영하는 이클립스 플랫폼을 위한 플러그인(plug-in)을 만드는 것이다. 웹 서비스 클라이언트 코드 자동생성, 클래스에서 웹 서비스로의 변환, 톰캣 서버의 로컬 설치를 위해 생성된 웹 서비스의 배치 또는 이와 상반되는 상황 등이 이 툴에서 지원된다.

그림 11은 동적 서비스 연결을 위한 각 속성을 명시하는 사용자 인터페이스를 보여준다. 웹 서비스 호출 클래스나 워크플로우 호출 클래스는 이 정보를 기반으로 자동 생성된다. 웹 서비스가 동적 서비스 연결 작업을 통해서 간접적으로 호출되기 때문에 직접 호출 되는 방법과 비교해서 성능적인 면에서 손실이 있을 수 있다. 만약 워크플로우를 통해서 웹 서비스를 호출하는 경우에는 이 같은 성능적 손실은 더 클 것으로 예상된다. 따라서 각 방식의 성능 손실의 정도를 보다 정확히 이해하기 위해서 각 방법을 사용한 웹 서비스 호출의 경과 시간을 측정했다. 모두 세 가지의 상황을 가정하였는데, 첫째는 웹 서비스와 클라이언트 응용 프로그램이 동일 컴퓨터 상에 있는 경우이고, 둘째는 LAN환경 내에서 연결되어 있는 경우이며, 마지막으로는 인터넷 환경에서 웹 서비스를 원격 호출해야 하는 경우이다. 요구(request)에 대한 응답(respond) 이외의 다른 작업은 웹 서비스 내에서는 이뤄지지 않기 때문에 웹 서비스 내의 서비스 시간은 포함 시키지 않았다.

표 2는 측정결과를 보여준다. 예상했던 것과 같이 동일 컴퓨터 내에서 직접 호출한 경과 대기 시간 값이 원격장소에서 간접적으로 호출한 대기 시간 값에 비해 월등히 빠르다는 것을 알 수 있다. 한 가지 특이한 사항은 원격장소에서 간접적으로 호출했을 때의 대기 시간이 동일 컴퓨터 내에서 간접적으로 호출했을 경우에 비해 더 짧다는 것이다. 이와 같은 결과가 나온 이유는 동일 컴퓨터 내에서는 웹 서비스 및 많은 리소스가 필요한 워크플로우 시스템이 같이 설치되었기 때문에 실행에 따른 대기 시간이 조금 더 걸린 것이다. 표에서 워크플로우를 통해서 웹 서비스를 호출할 때 응답 시간은 평균적으로 약 0.5초 정도 소요되었다. 표에서 괄호 안의 값은 표준편차를 나타낸다. 한 가지 지적 사항은 이 결과 값이 최적화된 환경에서 측정된 것은 아니라는 것이다. 다시 말하면, 동적 서비스 연결만을 지원하도록 최적화 시킨 워크플로우를 사용하였을 경우에는 훨씬 좋은 결과 값을 얻을 수 있을 것으로 예상된다. 그렇지만 실시간 응용 프로그램에서는 위와 같은 성능상 손실 문제가 상당히 심각한 문제를 야기할 수도 있다. 이 경우에는 생성된 클래스를 통해서 간접적으로 호출하는 것이 동적 서비스 연결을 지원하는데 적절한 방법이 될 수 있다. 동적 서비스 연결을 지원하는 클래스를 통해서 간접적으로 웹 서비스를 호출하는 것이 직접 호출할 때와 비슷한 성능 결과를 보일 것으로 예상된다. 간접 웹 서비스 호출방법을 택함으로 얻어지는 유연성과 신뢰성을 고려한다면 특별한 경우가 아닌 경우 본 논문에서 제안한 방식이 안전한 웹 서비스 호출에 도움이 될 것으로 기대된다.

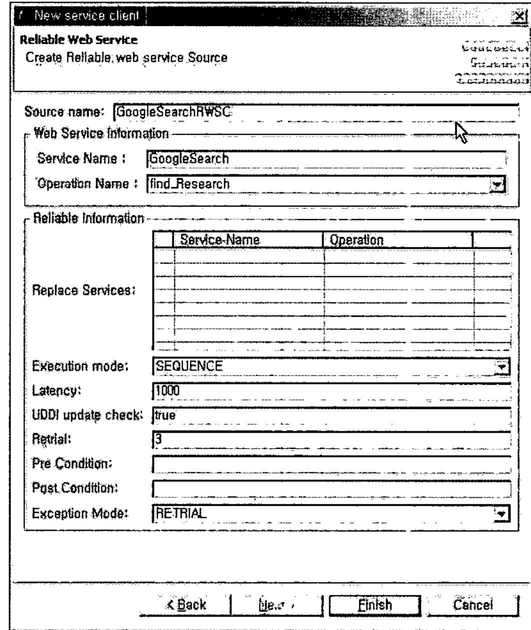


그림 11 속성 선택 다이얼로그 박스

표 2 웹 서비스 호출을 위한 평균 대기시간 (단위: 초)

웹 서비스 위치	직접 콜	간접 콜 (via Workflow)
동일 컴퓨터 사이트	0.010(0.008)	0.55(0.091)
원격 사이트 (LAN)	0.081(0.020)	0.53(0.086)
원격 사이트 (Internet)	0.100(0.045)	0.57(0.087)

6. 결 론

웹 서비스 기술의 미래는 밝다. 하지만 아직 극복해야 하는 장애물들은 많이 남아있다. 예를 들어, 인터넷상의 웹 서비스는 서비스의 이용 가능성 및 성능적인 면에서 항상 신뢰성을 보장해 주지는 못하는 문제를 가지고 있다.

본 논문에서는 이와 같은 문제를 해결하기 위해서 웹 서비스를 위한 예외 상황에 기반한 동적 서비스 연결 프레임워크를 제안하였다. 웹 서비스 호출 클래스와 워크플로우 호출 클래스가 신뢰성 있는 웹 서비스의 호출에 사용 되었으며, 시스템 구현을 위해서는 워크플로우 시스템의 예외 상황 핸들링 메커니즘을 사용하는 것을 제안하였다. 본 논문에서 제안한 방법을 기반으로 하여 예외 상황 핸들링 메커니즘을 지원하는 모든 워크플로우 시스템은 각각에 맞는 솔루션을 개발 할 수 있다.

웹 서비스를 직접적으로 호출하는 방법에서 간접적으로 호출하는 방법으로 변환함으로 몇 가지 이득을 기대할 수 있지만, 성능적인 면에서 어느 정도의 손실이 발생한다. 웹 서비스 호출에서 대기시간(latency)이 중요한 문제가 된다면 클래스를 통해서 직·간접적인 호출을 하는 것이 적절한 방법이다. 오랜 시간 동안 실행되는 업무 프로세스가 웹 서비스의 주요한 서비스라는 웹 서비스의 특성을 고려할 때, 웹 서비스 호출 시에 발생하는 약간의 성능적 손실은 어느 정도 수용 될 수 있다. 그런 면에서 앞으로는 안전성이 보장된 간접 웹 서비스 호출 방식이 주목 받을 것으로 예상된다.

UDDI는 동적 엑서스 포인트(dynamic access point) 관리에도 유용하다고 알려져 있다. 본 논문의 동적 서비스 연결 기능이 UDDI의 기능과 연계된다면 더욱 개선된 동적 서비스 연결이 가능할 것이다. 따라서 향후에는 앞선 UDDI기능과 본 논문에서 제안한 동적 서비스 연결 프레임워크를 통합하는 작업이 필요하다.

참 고 문 헌

[1] Application Architecture: A Critical Foundation for Service - Oriented Development and Web Services. White Paper, The Stencil Group and Wakesoft, http://www.wakesoft.com/product/WhitePapers.html, August 2003.

[2] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau and H. F. Nielsen, "SOAP Version 1.2 World Wide Web Consortium Recommendation," http://www.w3.org/TR/soap12-part1/, 2003.

[3] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1 World Wide Web Consortium note," http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.

[4] D. Ehnebuske, B. McKee and D. Rogers, "UDDI Version 2.04 API Specification," http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, 2002.

[5] BPEL4WS Specification: Business Process Execution Language for Web Services Version 1.1, http://www-128.ibm.com/developerworks/library/specification/ws-bpel, 2003.

[6] I. Y. Ko, K. T. Yao and R. Neches, "Dynamic Coordination of Information Management Services for Processing Dynamic Web Content," Proceedings of the 11th International World Wide Web Conference, Honolulu, Hawaii, USA, May 7-11 2002.

[7] L. J. Zhang, Q. Zhou and T. Chao, "A Dynamic Services Discovery Framework for Traversing Web Services Representation Chain," Proceedings of the IEEE Int. Conf. on Web Services (ICWS'04), June 6-9 2004.

[8] M. M. B. Tariq and T. Kawahara, "Introducing Dynamic Distributed Coordination in Web Services for Next Generation Service Platforms," Proceedings of the IEEE Int. Conf. on Web Services (ICWS'04), June 6-9 2004.

[9] C. Ellis, K. Keddara and G. Rozenberg, "Dynamic change within workflow systems," In Proc. of the Conf. on Organizational Computing Systems, August, pp. 10-21, 1995.

[10] M. Reichert and P. Dadam, "ADEPTflex-supporting dynamic changes in workflows without loosing control," Journal of Intelligent Information Systems , Vol.10, No.2, pp. 93-129, March 1998.

[11] G. Kappel, S. Rausch-Schott and W. Retschitzegger, "Coordination in Workflow Management Systems - A Rule-based Approach, Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents," LNCS 1364, pp. 99-120, 1998.

[12] A. Lazacano, G. Alonso, H. Schuldt and C. Schuler, "The WISE Approach to Electronic Commerce," International Journal of Computer Systems, Science, and Engineering, Vol.13. No.5, pp. 345-357, Sept. 2001.

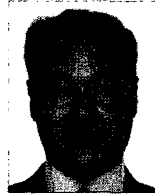
[13] M. Merz, B. Liberman and W. Lamersdorf, "Using Mobile Agents to Support Interorganizational Workflow-Management," International Journal on Applied Artificial Intelligence, Vol.11, No.6, pp. 551-572, Sept. 1997.

[14] M. Merz, B. Liberman and W. Lamersdorf, "Crossing Organizational Boundaries with Mobile

Agents in Electronic Service Markets," Integrated Computer-Aided Engineering, Vol.6 No.2, pp. 91-104, 1999.
[15] J. Y. Shim, M. J. Kwak and D. S. Han, "A Framework Supporting Dynamic Workflow Interoperation," International Workshop on Modeling Inter-Organizaional Systems and Interoperability of Enterprise Software and Applications (MIOS-INTEROP'05), Cyprus, 31 Oct.-4 Nov. 2005.
[16] D. S. Han, J. Y. Goo, S. D. Song, S. D. Lee and B. S. Seo, "Design of a Web Services Based eAI Framework," 6th International Conference on Advanced Communication Tecnnology (ICACT 2004), Phoenix Park, Korea, Feb. 2004.
[17] Y. K. Song and D. S. Han, "Exception Specification and Handling in Workflow Systems," LNCS 2642, pp. 495-506, 2003.
[18] Y. Shi, L. J. Zhang and B. Shi, "Exception Handling of Workflow for Web Services," Proceedings of the 4th Int. conf. on Computer and Information Technology, pp. 273-277, 2004.
[19] U. Greiner and E. Rahm, "Quality-Oriented Handling of Exceptions in Web services-Based Cooperative Processes," Proc. Of EAI-Workshop 2004-Enterprise Applocation Integration, Oldenburg. GITO-Verlag, Berlin, pp. 11-18, 2004.
[20] M. Kelin, A. Juan, Rodriguez-Aguilar and C. Dellarocas, "Using Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems: The Case of Agent Death," Autonomous Agents and Multi-Agent Systems 7, pp. 179-189, 2003.
[21] P. Johannesson, B. Wangler and P. Jayaweera, "Application and Process Integration - Concepts, Issues, and Research Directions," Information Systems Engineering Symposium '02, eds. S. Brinkkemper, E. Lindencrona, and A. Sölvberg, Springer Verlag, 2000.
[22] C. Szypersky, D. Gruntz and S. Murer, "Component Software: Beyond Object-Oriented Programming," 2nd edition, ACM Press, 2002.

한 동 수

1989년 서울대학교 계산통계학과(학사) 1991년 서울대학교 계산통계학과(석사) 1996년 일본 교토대학교 정보공학과(박사). 1996년 4월~1996년 7월 일본 NEC C&C 중앙연구소 연구원. 1996년 9월~1997년 10월 ㈜현대정보기술 정보기술연구소 책임연구원. 1997년 11월~현재 한국정보통신대학교 공학부 교수



이 성 독

1988년 전북대학교 전자공학과(학사). 1991년 전북대학교 전자공학과(석사). 2002년 일본 토호쿠대학교 정보과학연구과(박사). 1991년 5월~1993년 7월 군산대학교 전기공학과 조교. 2002년 4월~2003년 3월 일본 토호쿠대학교 전기통신연구소 연구원. 2003년 8월~현재 한국정보통신대학교 공학부 연구조교수

정 종 하

2005년 한동대학교 전산전자공학부 졸업(학사). 2005년 2월~현재 한국정보통신대학교 공학부 Computer System and Theory track 석사과정