미들웨어 가용성 관리를 위한 프레임웍 설계

박진욱° 박재걸 채흥석
부산대학교
{jwpark°, jgpark, hschae}@pusan.ac.kr

A Framework-based Approach to Availability Management of Middleware

Jinwook Park° Jaegeol Park Heung Seok Chae
Pusan National University

요 약

미들웨어에서 지속적인 서비스를 제공하는 가용성은 기본적인 요구사항이다. 가용성을 제공하기 위해서는 장애탐지와 장애복구가 기본적인 요소이다. 그리고 개별 미들웨어의 특성에 따라 다양한 수준의 가용성을 제공하기 위해서는 여러 가지 장애 탐지및 복구방법 중에서 적절한 방법이 적용되어야 한다. 본 논문에서는 다양한 수준의 가용성을 제공할 수 있는 프레임웍 기반의 설계방안을 제시한다. 프레임웍 기반의 설계를 사용함으로써 미들웨어에서 제공되어야 하는 가용성관리를 구현하는 데에 드는 시간과 비용을 절감 할 수 있다.

1. 서 론

최근에 하루 24시간 일 년 365일 중단없이 안정적인 서비스의 제공이 요구되는 시스템이 더욱 증가하고 있다. 24시간 운영되는 쇼핑몰이나 증권정보를 제공해야 하는 증권회사의 시스템, 많은 사람들이 같은 시간에 많이 집중될 수 있는 옥션이나 온라인 게임 같은 시스템에서도 안정적인 서비스는 요구되고 있다. 가용성이란 이렇게 특정 시스템의 사용자가 원하는 서비스를 사용자가 원하는 시간에 중단 없이 제공하는 특성을 말한다[1].

시스템의 가용성을 제공하기 위해서는 두 가지 행동을 포함해야 한다. 그것은 시스템의 이상을 탐지하고 발견한 이상을 복구하는 것이다. 여기서 말하는 시스템의 이상은 사용자가 원하는 서비스 또는 작업에 대하여 잘못된 일을 하거나 잘못된 대답을 주는 경우, 혹은 아예 일을 하지 않거나 대답을 주지 않는 경우를 말하며 이를 시스템 Failure라고 한다. 그리고 시스템 Failure는 아니더라도 내부의 특정 부분이 이상을 일으키는 경우를 장애(Fault) 라고 하며, 이런 장애들이 모이게 되면 Failure를 일으킬 수 있게 된다. 즉 높은 가용성을 제공하려면 앞에서 언급한 시스템 이상, 즉 장애를 빠른 시간 내에 탐지하고, 빠른 시간 내에 복구해야 한다[2].

하지만 장애를 탐지하고 복구하는 방법에는 여러 가지 방법이 존재하며 이러한 방법은 가용성을 제공하고자 하는 Application에 따라서 다를 수 있다. 온라인 게임의 경우 서버의 가용성을 체크하기 위하여 짧은 시간 간격으로 서버의 상태 체크를 해야 하며 반대로 서버의 경우도 클라이언트의 상태를 체크하기 위해 역시 짧은 시간 간격으로 클라이언트 상태 체크를 수행해야 한다. 하지만, 항상 많은 정보가 오고가지는 않는 작은 편의점의 계산 시스템 같은 경우는 빠른 시간 내에 반복해서 서버의 상태나 클라이언트의 상태를 체크할 필요가 없다. 그리고 장애가 탐지되었을 때에도 온라인 게임의 경우는 빠른 시간 내에 복구를 요구하며 손님이 적은 가게의 계산 시스템은 조금은 더 여유로운 복구방법을 선택하여도 상관이 없다. 이렇듯 가용성을 제공하는 방법인 장애 탐지와 복구에는 여러 방법이 있고 각 Application 마다 다른 방법을 사용한다[2].

본 논문에서는 이렇게 다양한 장애 탐지 방법과 복구 방법을 컴포넌트 수준에서 제공하는 미들웨어의 가용성 관리 프레임웍을 제시한다.

2. 연구배경

본 절에서는 연구배경으로서 가용성에 필요한 장애 탐지, 복구 방법들에 대하여 알아본다. 그리고 프레임웍의 정의, 장점, 구성요소에 대하여 알아 본다.

2.1 가용성을 위한 장애 탐지와 복구 방법

가용성제공을 위해 요구되는 장애탐지와 복구 방법을 소개한다.

- 장애 탐지 방법

장애를 탐지하는 방법은 Ping/Echo 방법과 Heartbeat 방법이 대표적이다[2].

Ping/Echo는 장애 탐지의 대상이 되는 컴포넌트에게 모니터가 Ping 을 컴포넌트에게 Ping을 받은 컴포넌트 스스로가 그에 대한 Echo를 모니터에게 다시 보내는 방법이다. 일정 시간 간격으로 보낼 수도 있으며 특정 이벤트가 있을 때에만 보낼 수도 있다. Ping 이후에 적절한 Echo가 없으면 해당 컴포넌트에 이상이 있다고 모니터가 판단한다.

Heartbeat는 장애 탐지의 대상이 되는 컴포넌트 스스로가 모니터에게 일정 시간간격으로 계속해서 heartbeat 메시지를 보내고, Heartbeat 메시지를 받는 모니터는 일정 시간 내에 메시지가 도착하기를 기다리는 방법이다. 모니터는 장애탐지를 위해 등록된 컴포넌트의 heartbeat가 제 시간에 오지 않으면 해당 컴포넌트에 이상이 있다고 판단한다. 앞에서 소개한 Ping/Echo 방법보다는 네트웍 부하가 비교적 크다.

- 장애 복구 방법

장애를 복구하는 방법은 크게 Fail-over 방법과 재시작 방법으로 나누어 볼 수 있다[2][3].

Fail-over는 장애가 일어난 컴포넌트에서 하던 일을 정상적으로 동작하는 다른 컴포넌트에게 옮김으로써 일을 계속할 수 있게 하는 방법이다. 예를 들어, A라는 컴포넌트에서 장애가 탐지되었을 때 같은 일을 할 수 있는 컴포넌트 B가 A가 하던 일을 대신해 주게 된다. Fail-over는 중복기법을 사용한다. 중복기법에는 수동적 중복기법과 능동적 중복기법이 있다. 수동적 중복기법은 대기하는 컴포넌트를 생성해 두고 특정 컴포넌트의 장애가 탐지되었을 때에 대기중인 컴포넌트의 기능이 동작하도록 하는 방법이다. 능동적 중복기법은 항상 일을 하던 컴포넌트가 장애가 탐지된 컴포넌트의 일까지 처리하는 방법이다. Fail-over를 위해서는 중복을 두는 방법을 사용해야 하며, 이런 중복을 위한 컴포넌트를 만들어 두어야 하므로 시스템 자원이 낭비될 수 있다.

재시작(restart)은 장애가 탐지된 컴포넌트를 다시 시작하도록 하는 방법이다. 앞에서 소개한 중복기법은 장애가 탐지된 컴포넌트를 대신할 수 있는 컴포넌트를 두어 장애가 탐지되었을 때에 장애가 발견된 컴포넌트를 바로 대체하여 아주 짧은 시간내에 시스템이 복구된다. 반면에 재시작 기법은 재시작을 하는 컴포넌트의 기능이 재시작을 하는 동안에는 동작하지 않는 단점이 있다. 대신에 중복기법을 사용하지 않으므로 시스템 자원이 Fail-over 경우보다 적게 사용된다는 장점이 있다. Fail-over를 사용하는 경우에도 장애가 탐지된 컴포넌트는 재시작을 통해 다시 복구될 가능성을 가진다.

2.2 프레임웍

프레임웍의 정의, 장점, 구성요소를 소개하고 가용성 프레임웍의 필요성을 소개한다.

프레임웍이란 "연관된 문제들에 대한 솔루션의 디자인을 구체화한 클래스의 집합" 이다[4]. 즉, 특정 영역에 대한 문제를 해결하기 위한 주요 디자인을 구체화 한 틀을 말한다. 유명한 프레임웍으로는 ACE(Advanced Computing Environment), MFC (Microsoft Foundation Classes), DCOM(Microsoftware's Distributed Common Object Model), RMI(Remote Mothod Invocation) 등이 있다[5].

프레임웍을 사용하게 되면 인터페이스를 이용하여 세부 구현을 캡슐화 할 수 있으므로 modularity를 높일 수 있고, 제공되는 안정된 인터페이스를 통해서 reusability를 향상 시킬 수 있다. 또한, 안정적인 인터페이스를 확장할 수 있는 hook method[6]를 통해서 확장성을 향상

이 논문은 교육인적자원부 지방연구중심대학육성사업
(차세대물류IT기술연구사업단)의 지원에 의하여 연구되었음.

시킬 수 있다[5].

프레임웍은 핵심 프레임웍 디자인, 프레임웍 내부 증분, 응용-특화 증분으로 구성된다[5].

- 핵심 프레임웍 디자인은 추상, 실제 클래스를 포함하며 실제 클래스일 경우에 프레임웍 사용자에게는 보여지지 않는다. 프레임웍 사용자는 핵심 프레임웍 디자인 중에서 보이는 추상 클래스를 서브 클래싱하여 사용할 수 있으며 이런 경우 이 부분이 hot spot[7]이 된다. hot spot은 프레임웍에서 그 구조만 제공하며 실제 구현 application 마다 다르게 구현해야 하는 부분을 말한다.
- 프레임웍 내부 증분은 핵심 프레임웍 디자인이 사용할 수 있는 클래스 라이브러리들을 말한다. 핵심 프레임웍 디자인의 공통된 부분을 구현한 부분이다.
- 응용-특화 증분은 프레임웍을 구현하는 application에서 직접 구현해야 하는 내용을 나타낸다. 핵심 프레임웍 디자인에 존재하는 추상 클래스들을 서브 클래싱하여 application에 맞는 내용으로 다시 구현하는 부분이다.

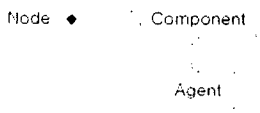
미들웨어 시스템의 가용성을 제공하기 위한 장애 탐지 방법과 장애 복구 방법을 직접 구현하는 데에는 많은 시간과 비용이 소모 된다. 본 논문에서 제공하는 프레임웍을 사용하여 개발하면 장애 탐지 및 복구 방법을 구현하는 데에 드는 시간과 비용을 절약 할 수 있다.

3. 가용성 관리 프레임웍

본 절에서는 가용성 관리 프레임웍을 설명하기 위하여 가용성 관리의 대상이 되는 개체에 대한 모델과 개체모델의 한 요소인 Agent의 상태 모델을 설명한다. 그리고, 가용성 관리 프레임웍의 전체 흐름을 알아보고 프레임웍을 통한 장애탐지 시나리오와 장애복구 시나리오를 설명한다.

3.1 가용성 관리 개체 모델

가용성 관리 프레임웍에 필요한 개체 모델은 크게 3가지로 구분된다. 장애 탐지의 대상이 되는 컴포넌트들이 있는 Node, 실제로 장애 탐지의 대상이 되는 컴포넌트, 그리고 이런 컴포넌트의 장애를 탐지하고 복구하는 일을 대신하는 Agent로 나누어서 볼 수 있다. [그림1]은 가용성 관리 개체 모델을 보여준다.

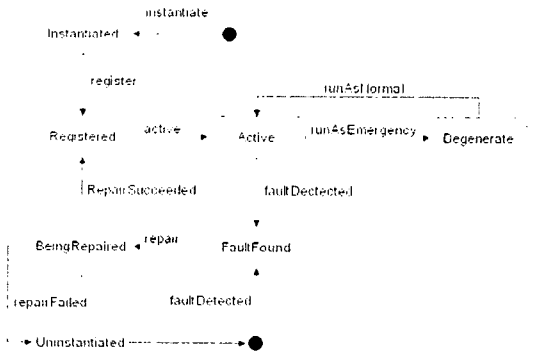


[그림1] 가용성 관리 개체 모델

프레임웍에서 인터페이스를 제공해야하는 것은 Node와 Agent까지 이며 실제 컴포넌트는 응용-특화 증분에 해당한다.

3.2 Agent의 상태 모델

컴포넌트에 직접 가용성과 관련된 코드를 삽입하지 않으므로 Agent의 상태에 따라서 컴포넌트 관련 내용을 프레임웍에 구성해야 한다. [그림2]는 ComponentAgent의 상태 모델을 보여준다.



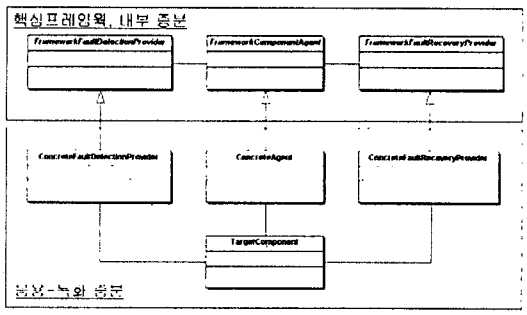
[그림 2] Component Agent의 상태 다이어그램

ComponentAgent의 상태 모델은 크게 컴포넌트 생성/등록, 장애 처리, 의존 컴포넌트 장애 처리의 3가지 부분으로 나누어 진다.

- **컴포넌트 생성/등록** 부분은 컴포넌트를 생성하고 가용성 관리에 등록되는 상태들이다. 처음 상태에서 Agent의 객체가 만들어지면 Instantiated 상태로 전이한다. Agent의 객체를 만드는 것은 응용-특화 증분에 해당하는 부분으로 Application에서 직접 해야 한다. Agent가 만들어진 상태에서 장애 탐지의 대상임을 알려주기 위해 프레임웍에 등록되면 Registered 상태로 전이한다. Registered 상태에서 프레임웍으로부터 컴포넌트의 일을 시작하라는 메시지가 도착하면 Active 상태로 전이하며 프레임워크는 이때부터 Agent를 통해서 컴포넌트의 Fault 탐지를 시작한다.
- **컴포넌트 장애 처리** 부분은 해당 컴포넌트 자체에 장애가 발생했을 때의 처리를 위한 상태들이다. Active 상태에서 컴포넌트의 장애를 발견하면 FaultFound 상태로 전이한다. FaultFound 상태로 전이 할 때는 장애 탐지 행동을 멈추어야 한다. FaultFound 상태에서 복구가 행해지면 BeingRepaired 상태로 전이한다. BeingRepaired 상태에서 복구가 제대로 이루어졌는지를 조사하여, 복구가 제대로 이루어졌으면(repair succeeded 이벤트) 다시 Registered 상태로 전이한다. 그러나, 복구가 제대로 이루어지지 않았으면(repairFailed 이벤트) Failed 상태로 전이하게 되고 프레임웍으로부터 destroy 이벤트를 받아 사라지게 된다.
- **의존 컴포넌트 장애 처리** 부분은 Active 상태에서 컴포넌트 자신의 장애가 탐지된 것이 아니라, 컴포넌트와 의존 관계에 있는 다른 컴포넌트의 장애가 탐지된 경우의 처리를 위한 상태들이다. 의존 컴포넌트의 장애가 탐지되면 runAsEmergency 이벤트가 발생하고 Degenerate 상태로 전이한다. 그리고 Fault가 탐지된 의존관계에 있는 컴포넌트가 복구되거나, 다른 컴포넌트로 대체되면(Fail-over) 다시 runAsNormal 이벤트를 통해서 Active 상태로 전이한다.

3.3. 가용성 프레임웍 전체 흐름

본 연구에서 제안하는 프레임웍은 미들웨어의 말단인 컴포넌트 각각의 가용성을 관리하기 위한 방법들을 지원하고 있다. [그림3]은 전체 프레임웍의 흐름을 보여준다.

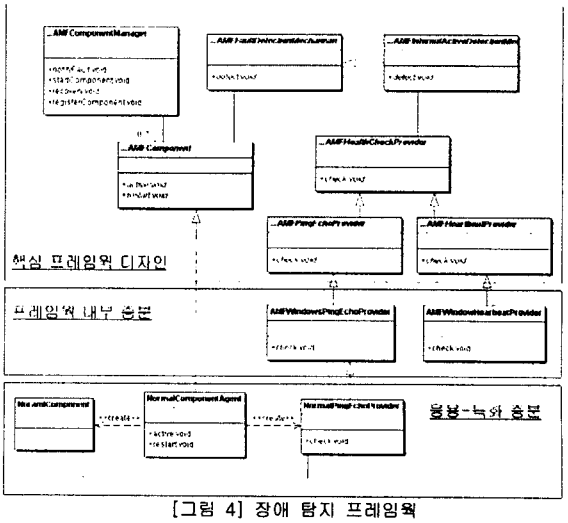


[그림 3] 가용성 프레임웍 전체 흐름

장애탐지의 대상이 되는 컴포넌트(*TargetComponent*)를 프레임웍에 등록하면 *ConcreteAgent*와 *ConcreteFaultDetectionProvider*에 의해서 감시된다. 장애가 발생하면 그 내용을 *ConcreteAgent*가 *FrameworkComponentAgent*를 통해 프레임웍에 전달하고, 프레임웍으로부터 복구 명령이 *ConcreteAgent*에게 전달되어 FaultRecoveryProvider에 의해서 처리된다.

3.4 장애 탐지 시나리오

장애 탐지와 관련된 핵심 프레임웍과 프레임웍 내부 증분, 응용-특화 증분은 [그림4]와 같다.

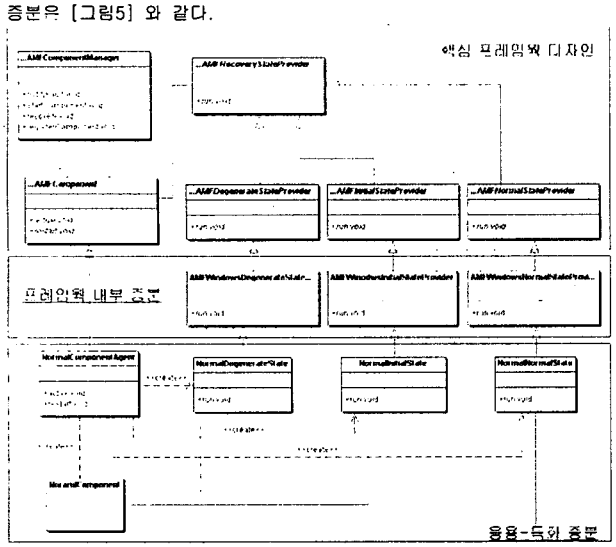


[그림 4] 장애 탐지 프레임웍

- 장애 탐지의 핵심 프레임웍 에서 주목해야 할 것은 *AMFComponent* 와 *AMFFaultDetectionMechanism*이다. *AMFComponent*는 응용-특화 증분에서 실제 *ComponentAgent*가 되는 부분이다. *ComponentAgent*는 장애 탐지 방법(*AMFFaultDetectionMechanism*)을 가지며 그 방법에는 여러 가지가 있지만 [그림4]에서는 *AMFInternalActiveDetectionMechanism*을 사용하고 있다. *AMFInternalActiveDetectionMechanism*은 다시 상태체크 Provider (*AMFHealthCheckProvider*)를 참조하며 이부분이 응용-특화 부분에서 상속받아 사용할 수 있는 hotspot이 된다. *NormalCompoenntAgent*는 자신이 가지고 있는 컴포넌트의 장애를 탐지하기 위해서 *NormalPingEchoProvider*를 사용한다.
- 프레임웍 내부 증분은 시스템 플랫폼에 따라 구현이 변경될 수 있는 부분에 대한 구현이다. [그림4]에서 프레임웍 내부 증분은 윈도우 시스템에서의 탐지 Provider를 정의하고 있다 (*AMFWindowsPingEchoProvider*, *AMFWindowsHeartbeatProvider*). 이러한 프레임웍 내부 증분을 구현함으로써 응용-특화 부분에서 OS에 의존하는 코드를 구현하지 않을 수 있도록 한다.
- 응용-특화 증분은 *AMFComponent*와 실제 장애 탐지 대상인 Component, 그리고 컴포넌트의 장애 탐지를 수행하는 Provider의 실제 구현으로 이루어 진다. [그림4]의 응용-특화 증분에서 확인할 수 있듯이, *NormalComponentAgent*가 *NormalComponent*와 *NormalPingEchoProvider*를 생성하여 연결해 준다. *NormalComponentAgent*에게 장애 탐지 명령이 내려오면 *NormalPingEchoProvider*에게 탐지 시작이 지시되고 *NormalPingEchoProvider* 는 *NormalComponent*에대한 장애 탐지를 시작한다. 만약 *NormalPingEchoProvider*가 *NormalComponent*의 장애를 탐지하면 *NormalComponentAgent*를 통해서 프레임웍으로 전달된다.

3.5 장애 복구 시나리오

장애 복구와 관련된 핵심 프레임웍과 프레임웍 내부 증분, 응용-특화 증분은 [그림5] 와 같다.



[그림 5] 장애 복구 프레임웍

- 장애 복구의 핵심 프레임웍에서 주목해야 할 것은 *AMFComponent* 와 *AMFRecoveryStateProvider*이다. *AMFComponent*는 앞에서 설명하였듯이 응용-특화 증분에서 실제 *ComponentAgent*가 되는 부분이다. *AMFComponent*는 *AMFRecoveryStateProvider*를 가지며 Provider는 *AMFComponent*의 상태의 변화에 따른 여러 가지 Provider를 가질 수 있다. 프레임웍 상위로부터 *AMFComponent*에게 재시작 명령이 전달될 경우에만 *AMFComponent*가 직접 컴포넌트에게 재시작을 지시하며, 그 외의 Agent 상태 모델에서 나타나는 Degenarate 상태, Active 상태, Registered 상태는 3가지 Provider에 대응되어 실행된다(*AMFDegenerateStateProvider*, *AMFInitialStateProvider*, *AMFNormalStateProvider*). 복구와 관련된 명령이 프레임웍 상단에서 내려오면 그에 맞는 적절한 Provider를 생성하고 사용해야 한다. 이것은 State 패턴과 매우 유사하다.
- 프레임웍 내부 증분은 각 장애 복구 상태 Provider에 대해서 다른 시스템에서의 구현을 지원하기 위한 부분이다. [그림5]에서는 윈도우 시스템에서의 장애 복구 Provider를 구현할 때 윈도우 OS에 의존하는 코드를 미리 작성해 놓음으로써 응용-특화 부분에서 OS관련 내용을 구현하지 않을 수 있도록 한다 (*AMFWindowsDegenerateStateProvider*, *AMFWindowsInitialStateProvider*, *AMFWindowsNormalStateProvider*).
- 응용-특화 증분은 *AMFComponent*와 복구 대상이 되는 컴포넌트, 그리고 *AMFComponent*의 복구 상태에 따라 컴포넌트 제어 방법을 제공하는 Provider들의 실제 구현으로 이루어 진다. [그림5]의 응용-특화 증분에서 확인 할 수 있듯이, *NormalComponentAgent*는 각 장애 복구 상태에 따른 Provider를 생성하며(*NormalDegenerateState*, *NormalInitialState*, *NormalNormalState*) 컴포넌트와 Provider들을 연결하여 준다. *NormalComponentAgent*에게 장애 복구와 관련된 명령이 전달되면 *NormalComponentAgent*는 전달된 명령에 따른 적절한 복구 상태 Provider를 할당하고 Provider에게 행동을 지시한다.

4. 결론 및 향후 연구 방향

본 논문에서는 미들웨어에서의 가용성을 제공하기 위한 가용성 관리 프레임웍을 제안하였다. 가용성 관리 프레임웍은 장애 탐지와 복구에 대한 다양한 방법을 모두 지원하며 각 미들웨어에 특화된 가용성 관리를 응용-특화 증분에서 직접 구현하였다. 본 논문에서의 가용성 탐지 대상은 컴포넌트였다. 향후에는 미들웨어에서의 노드에 대한 가용성 관리를 제공 할 수 있도록 프레임웍을 보강할 계획이다.

참고문헌

[1] Evan Marcus and Hal Stern, Blueprints for High Availability 2nd Edition, 2003
[2] SA Forum, Sevice Availability Forum Application Interface Specification Volume 2:Availability Management Framework SAI-AIS-AMF-B.01.01, http://www.saforum.org
[3] Len Bass, Paul Clements, and Rick Kazaman, Software Architecture in Practice 2nd Edition, 2003
[4] Ralph E. Johnson and B. Foote, Designing reusable classes. Journal of Object-Oriented Programming 1(2), June 1988
[5] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, Building Application Frameworks, 1999
[6] Pree, Wolfgang. Design Patterns for Object-Oriented Software Development. Reading, MA:Addison-Wesley, 1995
[7] Pree, Wolfgang. Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design. Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, July, 1994