ISBN 978-89-6211-674-8

Spring framework 기반 Comet 서버 푸쉬 기술 설계 및 구현

한 영 만

hans@kisti.re.kr

한국과학기술정보연구원

Korea Institute of Science and Technology Information

KiSTi Korea Institute of Science and Technology Information

목 차

1. 서론 ·····1

2. 적용 기술 요소 및 구현 방법 ·····2

2.1. Spring 프레임워크 ·····2

2.2. 클라이언트 폴링 vs. Jetty Comet ·····3

2.3. Spring framework 기반 Jetty 서버 설정 ·····5

2.4. Spring framework 기반 Comet 서버 푸시 구현 ·····8

3. 맺음말 ·····13

표 차례

표 1 Continuation 기반 Comet 기술 적용시 서버 자원 소요량 ·····4

그림 차례

그림 1 Spring 프레이워크의 7가지 모듈 구성도 ·····3

그림 2. Jetty Comet 기반 생명정보 워크플로우 데이터 동기화 실행모델 ··5

그림 3 BatchJobTracker 애플리케이션 클래스 다이어그램 ·····9

1. 서론

정보 통신 기술의 발달과 더불어 웹이 등장한 이후로, 실생활의 거의 모든 분야에 막대한 영향을 끼치는 눈부신 발전을 이루어 왔다. 최근 웹 기술의 발전과 웹에서의 사용자 중심적인 쌍방향 통신의 요구에 따라 Web 2.0이라는 웹의 새로운 패러다임 전환이 이루어지고 있다. '웹 2.0'이라는 용어는 미국의 IT 전문 출판 미디어인 오라일리(O'Reilly)와 미디어 라이브 인터내셔널의 컨퍼런스 브레인스토밍 세션에서 처음으로 등장하였다. 2001년 닷컴 버블 붕괴 이후 인터넷 업계에서 사라진 기업과 살아남은 기업의 차이에 주목하여, 위기 이후에도 생존한 기업들이 가지고 있는 공통점을 지칭하는 말로 오라일리 부사장인 데일 도허티(Dale Dougherty)가 "웹 2.0"이라는 개념을 제안하였다. 웹 2.0의 특성은 크게 네 가지로 정리할 수 있다. 첫째, '플랫폼으로서의 웹'이라는 특성을 들 수 있다. 윈도우즈를 실행시켜야만 이용할 수 있었던 웹 1.0시대의 넷스케이프와는 달리, 구글처럼 판매 혹은 패키지화되지 않고 순수한 웹 애플리케이션으로 서비스되는 형태를 가지는 것이 대표적인 웹 2.0의 특징이다. 둘째, 오픈소스를 통한 기술의 자유로운 재구성이 일어난다는 점이다. 웹 2.0은 이미 존재하고 있는 기술의 변환 또는 조합을 통해 보다 사용자에게 개방적인 환경을 제공하는 흐름을 의미한다. 셋째, 사용자의 참여를 보장하도록 설계된다는 점을 들 수 있다. 사용자에 의해 변경될 수 있는 유연한 데이터를 제공하고 그로 인해 새롭게 창출되는 콘텐츠를 유통시키는 플랫폼으로 기능하는 것이 웹 2.0 서비스의 주요 특징이다. 넷째, 즉각적 피드백을 통해 끊임없는 기술적 보완과 혁신이 일어난다는 점이다[1].

웹 2.0 패러다임이 대두됨에 따라 사용자 중심적이고 쌍방향 통신을 보다 원활이 하기 위한 RIA(Rich Internet Application)[2] 기술이 개발되었다. RIA는 기존의 웹 브라우저 기반의 애플리케이션을 대체하여 웹상에서 전통적인 데스크톱 애플리케이션 기능과 특징을 구현한 웹 애플리케이션이다. RIA를 구현하기 위한 대표적인 표준 기술 중 하나가 AJAX(Asynchronous JavaScript and XML)[3]이다. Ajax는 말 그대로 비동기 자바스크립트 XML'이다. Ajax는 자바스크립트 렌더링 엔진을 이용한 기술로, Ajax를 이용할 경우 플래시나 액티브엑스(ActiveX) 의존도를 많이 벗어날 수 있다. 대표적으로 구글과 야후, 아마존 등의 여러 서비스에서 Ajax 기술을 활용하고 있다. 이들 사이트의 서비스는 액티브엑스를 사용하는 사이트와 달리 윈도의 익스플로러가 아닌 다른 운영체제나 브라우저에서도 사용할 수 있다. AJAX라는 낱말은 Garrett가 2005년에 쓴 'A New Approach to Web Applications'이라는 에세이에서 'AJAX(Asynchronous JavaScript + XML)'라는 낱말로 이 기술을 소개한 이후 퍼진 것으로 알려졌다[4].

AJAX를 통한 RIA의 구현의 한계점은 클라이언트 요청에 의해서만 콘텐츠를 획득할 수 있다는 점이다. 즉, 웹과 같은 클라이언트-서버 환경에서 콘텐츠를 생산하

는 공급자가 사용자 측으로 실시간으로 정보를 제공할 수 없다는 점이다. 이러한 한계점을 극복하고 실시간 웹이라는 차세대 웹 환경을 구현하기 위해 가장 최근에 대두되고 있는 기술이 Comet과 Reverse AJAX[5]이다. Comet이란 웹 클라이언트(보통 웹 브라우저)의 명시적인 요청이 없어도 서버 푸시 방식으로 동작하는 웹 프로그래밍 모델을 말하며 차세대 웹의 근간이 될 HTML5[6] 표준에서 WebSocket 기술을 통해 지원되어 주류 기술로 인정받고 있다.

본 기술보고서는 최근 각광받는 소프트웨어 프레임워크인 Spring 프레임워크를 기반으로 Comet 기술을 구현하는 실제적인 방법에 대해 제시하고자 한다.

2. 적용 기술 요소 및 구현 방법

2.1. Spring 프레임워크

Spring 프레임워크는 Rod Johnson의 “Expert One-on-One J2EE Design and Development"[7]에서 소개된 코드를 기초로 한 J2EE(Java 2 platform Enterprise Edition) 기반 애플리케이션 프레임워크 중 하나로, 객체의 라이프 사이클을 관리하기 위하여 Dependency Injection 기법을 사용하는 경량 컨테이너이다. Spring을 활용하면 복잡한 엔터프라이즈 애플리케이션 개발을 위해 EJB(Enterprise Java Bean)와 같은 복잡한 아키텍처를 사용하지 않고도 AOP(Aspect Oriented Programming), IoC(Inversion of Control), 테스트, 트랜잭션, 원격 서비스 연결, 객체-관계 맵핑, 보안, 웹 MVC(Model-View-Controller) 등과 같은 엔터프라이즈 자바 기술을 쉽게 적용할 수 있다. Spring은 <그림 1>에서 보는 바와 같이 7개의 잘 정의된 모듈들로 구성되며 전체적으로 이들 모듈은 엔터프라이즈 애플리케이션 개발에 필요한 모든 것을 제공한다. 이는 애플리케이션이 완전히 Spring 프레임워크를 기반으로 해야 하는 것은 아니다. 즉, 애플리케이션에 적합한 모듈을 선택하여 적용하고 나머지 모듈들은 무시해도 된다. Spring 모듈은 모두 핵심 컨테이너 위에 구축되어있다. Spring의 코어 기능인 자바 빈의 생성, 설정, 관리하는 방법은 핵심 컨테이너에서 정의된다. Spring의 핵심 기능인 IoC, 즉 제어 역행화는 Martin Fowler[8]에 따르면 객체간의 의존성 제어가 객체 코드에 의하지 않고 프레임워크에 의해 관리됨을 말한다. Spring은 이러한 IoC 개념을 기반으로 한 대표적인 IoC 컨테이너이다. IoC라는 용어가 직관적이 못하기 때문에 보다 이해하기 쉬운 Dependency Injection, 즉 의존성 삽입이라고도 말한다. Spring 프레임워크는 객체간의 의존성 삽입을 XML을 통하여 설정할 수 있도록 한다. 이렇게 함으로써 객체 간 의존성을 최소화하고 객체 컴포넌트의 모듈화를 극대화하여 매우 유연하고 확장성 높은 시스템을 구현할 수 있다. 본 기술보고서에서 제시되는 대부분의 기술은 Spring 프레임워크를 기반으로 구현된다.

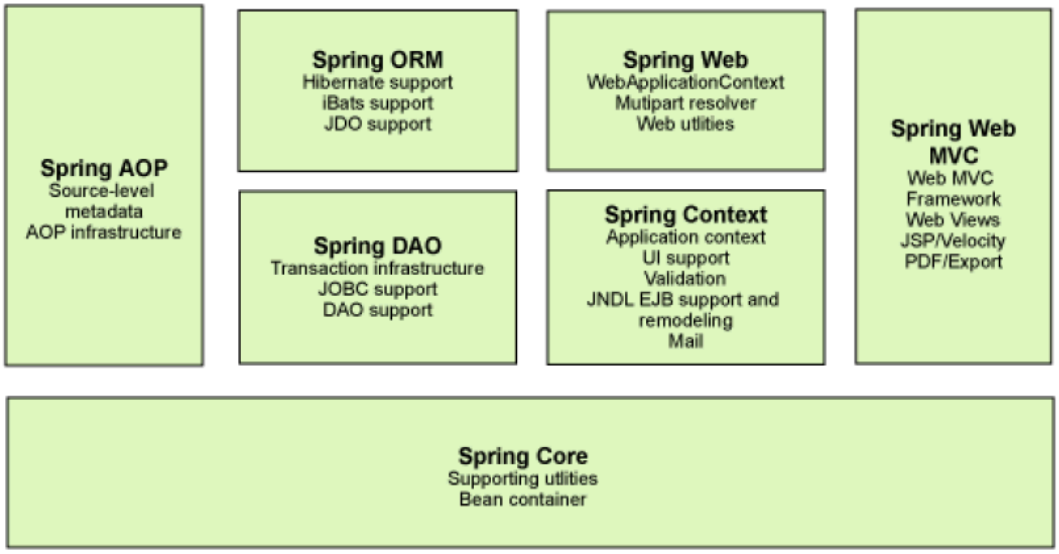

그림 1 Spring 프레이워크의 7가지 모듈 구성도

2.2. 클라이언트 폴링 vs. Jetty Comet

앞서 제시한 클라이언트-서버 환경에서 대용량의 데이터를 상호 동기화하기 위한 방법으로는 일정 시간 간격으로 클라이언트에서 서버 측으로 특정 데이터를 요청하여 처리하는 클라이언트 폴링(Polling) 방식과 서버 측에서 클라이언트 측으로 특정 상태의 데이터를 밀어 넣는 서버 푸시(Push)방법이 있다. 클라이언트 폴링 방식의 가장 큰 단점은 많은 클라이언트가 서버 측으로 데이터를 동시 요청할 경우 생성되는 트래픽 부하와 서버 측 리소스 부하에 있다. 최악의 경우 실행시간이 긴 데이터 동기화를 클라이언트 폴링하면 거의 매번 동일한 데이터를 서버 측에서 가져오게 되는 결과를 초래한다. 이는 클라이언트 측이나 서버 측 모두 심각한 자원의 낭비라 할 수 있겠다. 물론 서버 측 부하는 폴링 간격을 늘림으로써 경감될 수 있으나, 서버 측과 클라이언트 측간의 데이터 동기화의 지연을 초래한다. 따라서 데이터 동기화의 빈도수 가 낮고 실시간으로 동기화가 필요한 경우 클라이언트-서버 간 데이터 동기화는 서버 푸시 기술을 적용하는 것이 바람직하다.

서버 푸시 기술은 초기 웹브라우징 환경을 선도했던 넷스케이프 사에서 최초로 제안한 이래로 최근의 Web 2.0 환경에서의 AJAX 등의 RIA(Rich Internet Application)기술의 발전과 더불어 Comet이라는 기술로 각광받고 있다. Comet이란 웹 클라이언트(보통 웹 브라우저)의 명시적인 요청이 없어도 서버 푸시 방식으로 동작하는 웹 프로그래밍 모델을 말하며 차세대 웹의 근간이 될 HTML5 표준에서 WebSocket 기술을 통해 지원되어 주류 기술로 인정받고 있다. Comet 기술의 동작 방식은 일반적으로 Long Polling 방식으로 클라이언트가 서버에 접속 하면 서버는 계속 접속을 유지하고 있다가 서버 측 이벤트가 발생하면 클라이언트로 이를 전송

하고 HTTP 트랜잭션을 마친다. 서버 측은 클라이언트 접속을 유지하며 이러한 과정을 반복한다. Comet 을 효율적으로 지원하기 위해서는 하나의 스레드가 여러 클라이언트 접속을 유지하며 각 접속에서 여러 개의 요청을 처리할 수 있어야 한다. 현재의 자바 기반 웹 애플리케이션 서버 환경(Java Servlet 환경)에서는 이것이 불가능하다. 따라서 이에 대한 대안이 여럿 등장했다. 자바 서블릿 서버 환경 중에는 Jetty 서블릿 컨테이너가 최초로 Continuation 방식의 Comet 기술을 구현했다. 현재 초기 드래프트 리뷰 상태에 있는 자바 서블릿 3.0에서 Jetty의 Continuation과 비슷한 방식으로 지원이 논의되고 있다. Continuation[9]는 scheme 등의 언어에서 지원하던 개념인데, Jetty 서블릿 컨테이너에서의 Continuation 지원 방식은 이와 유사한 메커니즘으로 버전 6.0부터 Continuation API를 제공하고 있다. Jetty Continuation의 동작방식은 서버에 접속한 클라이언트의 요청 처리를 중지(suspend)시킨 후 재개(resume)하면 바로 중지시킨 그 지점에서 다시 진행하는 것이 아니라 요청 처리 체인(FilterChain)을 다시 처음부터 진행한다는 점에서 기존 Continuation과 약간의 차이점이 있다. <표 1>는 일반적인 웹 서비스(Web 1.0), AJAX와 같은 Web 2.0 기술과 Comet 기술을 적용한 웹 서비스, 그리고 Jetty Continuation 기술을 함께 적용했을 시의 웹 서비스의 성능 개선 내용을 보여준다. Jetty Continuation 기반의 Comet 기술을 활용하면 웹서비시스 기반의 클라이언트-서버 환경에서의 서버의 부하를 최소화하면서 클라이언트에서 실시간으로 서버 측에서 변경된 데이터를 동기화 할 수 있다. <그림 2>은 Jetty Comet 기반으로 서버 측의 배치 작업 데이터를 동기화하는 실행 모델을 도식화 한 것이다.

	Formula	Web 1.0	Web 2.0 + Comet	Web 2.0 + Comet + Continuations
Users	u	10000	10000	10000
Requests/Burst	b	5	2	2
Burst period (s)	p	20	5	5
Request Duration (s)	d	0.200	0.150	0.175
Poll Duration (s)	D	0	10	10
Request rate (req/s)	rr=u*b/20	2500	4000	4000
Poll rate (req/s)	pr=u/d	0	1000	1000
Total (req/s)	r=rr+pr	2500	5000	5000
Concurrent requests	c=rr*d+pr*D	500	10600	10700
Min Threads	T=c T=r*d	500 -	10600 -	- **875**
Stack memory	S=64*1024*T	32MB	694MB	**57MB**

표 1 Continuation 기반 Comet 기술 적용시 서버 자원 소요량(출처: http://www.webtide.com/downloads/whitePaperAjaxJetty.html)

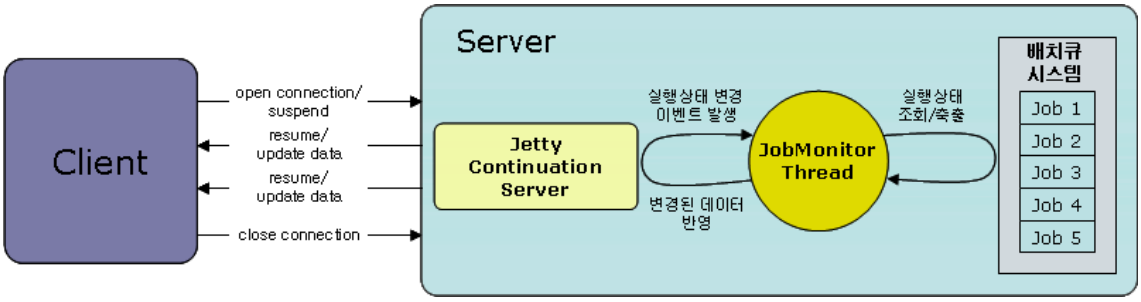

그림 2. Jetty Comet 기반 생명정보 워크플로우 데이터 동기화 실행모델

<그림 2>에서 볼 수 있듯이 초기에 클라이언트가 서버에 접속하면 접속 상태는 유지되면서 요청처리는 중지된 상태로 남아 있다. 이 때 Continuation 객체 내에서 요청처리에 대한 객체는 유효한 상태로 남아 있다. 서버 측의 배치 작업 모니터를 위한 JobMonitor는 일정 간격으로 배치큐에 적재되어 있는 배치 작업을 모니터링 하다가 각 실행 작업의 상태가 변경되면 Jetty 서버에 워크플로우 실행 상태 변경에 대한 이벤트를 발생시키고 변경된 워크플로우 데이터를 전달하게 된다. Jetty 서버가 워크플로우 상태 변경 이벤트를 받으면 Continuation 객체는 중지되었던 요청처리를 resume하고 변경된 데이터를 클라이언트 측으로 반환하게 된다.

2.3. Spring framework 기반 Jetty 서버 설정

Spring 프레임워크를 기반으로 Comet 기술을 구현하기 위해서는 먼저 Comet 기술을 지원하는 Jetty와 같은 웹 애플리케이션 서버를 Spring 프레임워크 위해 의존성 삽입 방식으로 통합해야 한다. 다음은 Spring 프레임워크에 Jetty를 통합하기 위한 XML 설정 내용이다 [10].

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="server.Server" class="org.mortbay.jetty.Server" destroy-method="stop">
        <property name="threadPool">
            <bean class="org.mortbay.thread.QueuedThreadPool">
                <property name="maxThreads" value="100" />
            </bean>
        </property>
```

```
        <property name="connectors">
            <list>
                <bean class="org.mortbay.jetty.nio.SelectChannelConnector">
                    <property name="port" value="8080" />
                    <property name="maxIdleTime" value="30000" />
                </bean>
            </list>
        </property>
        <property name="handler">
            <bean class="org.mortbay.jetty.handler.HandlerCollection">
                <property name="handlers">
                    <list>
                        <ref local="server.ContextHandlerCollection" />
                        <bean class="org.mortbay.jetty.handler.DefaultHandler" />
                        <bean class="org.mortbay.jetty.handler.RequestLogHandler">
                            <property name="requestLog">
                                <bean class="org.mortbay.jetty.NCSARequestLog">
                                    <constructor-arg
value="cfg/logs/jetty-yyyy_mm_dd.log" />
                                    <property name="extended" value="false"/>
                                </bean>
                            </property>
                        </bean>
                    </list>
                </property>
            </bean>
        </property>

        <property name="userRealms">
            <list>
                <bean class="org.mortbay.jetty.security.HashUserRealm">
                    <property name="name" value="Test Realm" />
                    <property name="config" value="cfg/etc/realm.properties" />
                </bean>
            </list>
        </property>
```

```
        <property name="stopAtShutdown" value="true" />
        <property name="sendServerVersion" value="true"/>
    </bean>

    <bean
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject" ref="server.Server" />
        <property name="targetMethod" value="addLifeCycle" />
        <property name="arguments">
            <list><ref local="server.ContextDeployer" /></list>
        </property>
    </bean>

    <bean
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject" ref="server.Server" />
        <property name="targetMethod" value="addLifeCycle" />
        <property name="arguments">
            <list><ref local="server.WebAppDeployer" /></list>
        </property>
    </bean>

    <bean id="server.ContextHandlerCollection"
class="org.mortbay.jetty.handler.ContextHandlerCollection" />

    <bean id="server.ContextDeployer"
class="org.mortbay.jetty.deployer.ContextDeployer">
        <property name="contexts" ref="server.ContextHandlerCollection" />
        <property name="configurationDir">
            <bean class="org.mortbay.resource.FileResource">
                <constructor-arg value="file://./cfg/contexts" />
            </bean>
        </property>
        <property name="scanInterval" value="1" />
    </bean>

    <bean id="server.WebAppDeployer"
```

```
class="org.mortbay.jetty.deployer.WebAppDeployer">
            <property name="contexts" ref="server.ContextHandlerCollection" />
            <property name="webAppDir" value="cfg/webapps" />
            <property name="parentLoaderPriority" value="false" />
            <property name="extract" value="true" />
            <property name="allowDuplicates" value="false" />
            <property name="defaultsDescriptor" value="cfg/etc/webdefault.xml" />
      </bean>
</beans>
```

설정된 Spring 프레임워크 기반 Jetty 서버를 구동하기 위해 자바 애플리케이션 코드는 다음과 같다.

```
public class CometServer{
      public static void main(String[] args) throws Exception {
            ApplicationContext context = new
FileSystemXmlApplicationContext("applicationContext.xml");
            Server server = (Server)context.getBean("server.Server");
            server.start();
            server.join();
      }
}
```

2.4. Spring framework 기반 Comet 서버 푸시 구현

Spring 프레임워크를 기반으로 Comet 서버 푸시 구현에 대해 설명하기 위해 간단한 배치큐 시스템의 작업 상태를 동기화하는 간단한 BatchJobTracker 애플리케이션을 제시하고자 한다. BatchJobTracker 애플리케이션은 특정 배치 작업의 상태가 변경되는 시점에 연결된 웹 클라이언트 프로그램에 해당 작업의 상태를 알려주는 애플리케이션이다. <그림 3>은 BatchJobTracker 애플리케이션의 클래스 다이어그램이다. 먼저 특정 배치작업에 대한 상태를 배치큐 시스템으로부터 얻어오는 것은 JobStatusMonitor가 수행한다. JobStatusMonitor는 startMonitor 메소드가 호출된 이후부터 주기적으로 특정작업에 대한 상태를 얻어온다. 배치 작업의 상태가 변경되면 JobStatusMonitor는 작업상태 변경 이벤트를 발생하고 등록된 JobStatusListener에 변경된 작업 상태를 전달한다. 다음은 JobStatusMonitor의 작업 상태 변경 이벤

트가 발생하는 부분의 코드이다.

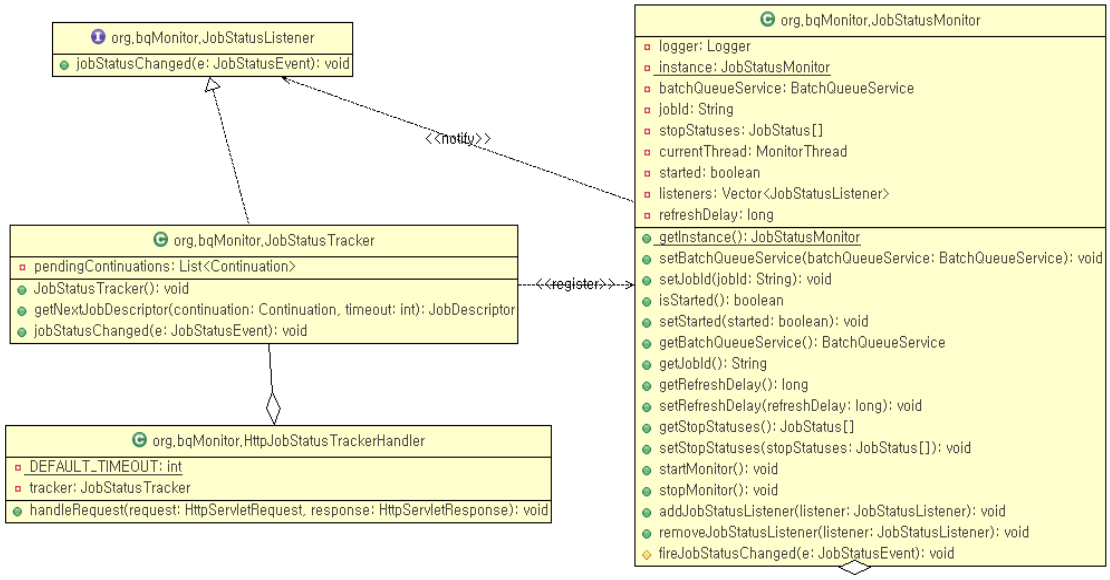

그림 3 BatchJobTracker 애플리케이션 클래스 다이어그램

```
public class JobStatusMonitor {
        private Logger logger = Logger.getLogger(JobStatusMonitor.class);
        private static JobStatusMonitor instance = new JobStatusMonitor();

        private BatchQueueService batchQueueService;
        private String jobId;
        private MonitorThread currentThread = null;
        private boolean started = false;
        private Vector<JobStatusListener> listeners = new Vector<JobStatusListener>();
        private long refreshDelay = 1000;

        private class MonitorThread extends Thread{
                private boolean kill = false;

                public void run() {
```

```
        try{
                logger.debug("Start MonitorThread.run for job id: " + jobId);
                JobStatus currentStatus = null;
                while(!kill){
                  sleep(refreshDelay);
                  JobDescriptor newJobDescriptor =
                                batchQueueService.getJobDescriptor(jobId);
                   if(currentStatus == null||
                     !currentStatus.equals(newJobDescriptor.getStatus())){
                        currentStatus = newJobDescriptor.getStatus();
                        // check job status for killing thread
                        if(ArrayUtils.contains(stopStatuses, currentStatus)){
                          stopMonitor();
                        }

                        fireJobStatusChanged(new JobStatusEvent(newJobDescriptor));
                     }
                  }
                  logger.debug("Done MonitorThread.run for job id: " + jobId);

         }
        catch(Exception e){
          logger.error("Error in JobStatusMonitor.MonitorThread.run:" + e);

        }
}
}

protected void fireJobStatusChanged(JobStatusEvent e) {
    for(JobStatusListener listener: listeners ){
      if(listener != null){
```

```
            listener.jobStatusChanged(e);
          }
        }
    }
}
```

JobStatusListener는 JobStatusMonitor에서 발생되는 이벤트를 수신하여 jobStatusChanged 메소드를 콜백하는 자바 인터페이스이다. 이 예제에서는 JobStatusListener를 구현하여 Continuation을 적용하는 JobStatusTracker를 클래스를 다음과 같이 구현한다.

```
public class JobStatusTracker implements JobStatusListener {

        private List<Continuation> pendingContinuations = new
ArrayList<Continuation>();

        public JobStatusTracker(){
                JobStatusMonitor.getInstance().addJobStatusListener(this);
        }

        public JobDescriptor getNextJobDescriptor(Continuation continuation, int
timeout) {

                synchronized (this) {
                        if (!continuation.isPending()) {
                                pendingContinuations.add(continuation);
                        }

                        // Wait for next update
                        continuation.suspend(timeout * 1000);
                }

                return (JobDescriptor) continuation.getObject();
        }
```

```
        @Override
        public void jobStatusChanged(JobStatusEvent e) {

                synchronized (this) {
                        for (Continuation continuation : pendingContinuations) {

                                continuation.setObject(e.getJobDescriptor());
                                continuation.resume();
                        }

                        pendingContinuations.clear();
                }
        }
}
```

클라이언트가 Continuation과 함께 getNextJobDescriptor를 호출하면, isPending 메소드는 요청이 이 지점에서 재시도 되지 않는지를 검사하고, 이를 작업 상태를 대기하는 Continuation 컬렉션에 추가한다. 그러고 나서 Continuation은 suspend 된다. 한편, 작업 상태가 변경될 때 호출되는 jobStatusChanged 메소드는 중지된 Continuation을 반복하고, 새로운 작업 상태를 설정하며, 이들을 resume한다. 각각의 재시도 된 요청은 getNextJobDescriptor의 실행하고 Continuation에서 변경된 작업 상태를 조회한 후 이것을 클라이언트에게 반환한다. 여기에서 동기화가 필요한 이유는, pendingContinuations 컬렉션의 일관성 없는 상태로부터 보호하고 새롭게 추가된 Continuation이 suspend 되기 전에 시작 되지 않도록 하기 위해서이다.

마지막으로 클라이언트의 HTTP 요청을 처리하는 HTTPJobStatusTrackerHandler 이다.

```
public class HttpJobStatusTrackerHandler implements HttpRequestHandler{
    private static final int DEFAULT_TIMEOUT = 60;
    private JobStatusTracker tracker = new JobStatusTracker();


        public void handleRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
```

```
                String jobId = request.getParameter("jobId");
                if(!StringUtils.isEmpty(jobId)){
                        JobStatusMonitor.getInstance().setJobId(jobId);

                        Continuation c =
ContinuationSupport.getContinuation(request, null);
                        JobDescriptor jobDescriptor = tracker.getNextJobDescriptor(c,
DEFAULT_TIMEOUT);

                        String json = JSON.toString(jobDescriptor);
                        response.getWriter().print(json);

                }
        }

}
```

이 HTTP 요청 처리 코드는 아주 작은 일을 수행한다. 요청의 Continuation을 획득하고 getNextJobDescrptor 메소드에 Continuation을 설정하여 호출한다. JobStatusTracker로부터 변경된 작업 상태를 받아 JSON[11] 형태로 변환한 후 HTTP 응답 객체에 출력한다.

3. 맺음말

최근 Web 2.0의 등장으로 정적이고 단방향의 웹은 사용자 참여적이고 쌍방향 통신이 가능한 형태로 진화하고 있다. Web 2.0 패러다임의 중심에서 차세대 실시간 웹을 구현하기 위한 대표적 기술 중 하나는 Comet이다.

본 기술보고서는 최근 많은 IT 분야에서 각광받고 있는 유연하고 확장성 높은 Spring 프레임워크를 기반으로 하여 Comet 기술을 실제 구현하는 방법을 제시하였다. 소개된 구현 기술은 Spring 프레임워크를 기반 프레임워크로 하는 차세대 실시간 웹 환경을 지향하는 다양한 응용 시스템에 쉽게 적용되어 활용되어 질 수 있을 것이라 기대된다.

참고문헌

1. 이호영 외, *웹2.0시대 디지털 콘텐츠의 사회적 확산 경로 연구*, 연구보고 07-03, 정보통신정책연구원, 2007

2. RIA web wiki page, *http://en.wikipedia.org/wiki/Rich_Internet_application*

3. AJAX web wiki page, *http://ko.wikipedia.org/wiki/Ajax*

4. Jesse James Garrett, *Ajax: A New Approach to Web Applications,* *http://www.adaptivepath.com/ideas/essays/archives/000385.php*, 2005

5. Dave Crane, Phil McCarthy, *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*, ISBN 978-1-59059-998-3, Apress, 2009.

6. W3C HTML5 web page, *http://dev.w3.org/html5/spec/Overview.html*

7. Rod Johnson, *Expert One-on-One J2EE Design and Development,* Wrox book, ISBN: 978-0-7645-4385-2, 2002.

8. Martin Fowler, *Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html*, 2004.

9. Gerald Jay Sussman and Guy L. Steele, Jr. *Scheme: An interpreter for extended lambda calculus,* AI Memo, 349:19, 1975

10. Embedding Jetty 6 in Spring: http://coffeaelectronica.com/blog/2008/08/embedding-jetty-6-in-spring/

11. JSON web page, http://www.json.org