JavaSpaces 기반의 Compute Server 구현

이영지, 김태윤
고려대학교 컴퓨터학과
e-mail : yjlee@netlab.korea.ac.kr

The Implementation a ComputeServer Using JavaSpaces

Lee, young-ji, Kim, tae- yun
Dept. of Computer Science, Korea University

요 약

분산환경 시스템은 여러 가지 면에서 장점이 있지만, 여러 환경들에 있는 프로세스들을 연결하는 면에서 어려움들이 많다. JavaSpaces™ 는 이러한 분산 환경들을 간단하고 강력하고, 효과적으로 설계할 수 있게 해주는 도구이다. JavaSpaces™ 는 프로세스들이 직접적으로 통신하는 것이 아니라, *Space* 라는 공유되는 공간에 오브젝트들을 놓음으로써 간접적인 통신을 하게 한다. Space 에서 프로세스들은 원하는 오브젝트들을 찾고, 컴퓨트 서버는 이러한 JavaSpaces™ 를 이용한 작업들을 수행하는 광범위한 목적의 서비스이다. 컴퓨트 서버는 간단한 코드로 작성될 수 있고, 여러 가지 장점이 있다.

1. 서론

요즈음 대부분의 소프트웨어의 문제는 분산 애플리케이션을 설계함으로써 해결하고 있다. 하지만, 분산 소프트웨어를 디자인하는 것은 어렵다. 이종(heterogeneity), 부분적인 실패, 지연(latency)과 같은 네트웍 환경의 기본적인 특징과 서로 독립적인 여러 프로세스를 강력하고 확장성 있는 애플리케이션으로 합치는 것과 같은 많은 고려 사항들이 있기 때문이다.

JavaSpaces™ 기술은 이러한 분산 애플리케이션을 만드는 것을 쉽게 해주는 간단하고, 효과적이며, 강력한 도구이다. 프로세스들은 서로 직접적으로 통신을 하지 않고, *Spaces* 라고 불리우는 지속적인 오브젝트 저장소를 통해 서로 느슨하게 연결이 되어서 통신을 하게 된다. 이 방법은 시스템을 유연하고, 확장성 있고 신뢰성 있게 만들어 준다. 이 시스템은 간단하면서도 더 나은 분산 애플리케이션을 만드는데 강력하다. 그 중에서도 컴퓨트 서버는 JavaSpaces™ 를 이용한 서비스이다. 컴퓨트 서버는 오브젝트의 계산이나, 또는 여러 가지 광범위한 목적에 쓰이기 위해 JavaSpaces™ 를 실세계에 이용한 예이다. 앞으로 이런 종류의 애플리케이션은 여러 분야에서 많이 쓰일 것으로 보인다.

본 논문에서는 JavaSpaces™ 에 대해 알아보고, 간단한 컴퓨트 서버를 구현해 본다.

2. 관련 연구

'분산 컴퓨팅'은 분산되어 있는 머신과 작업들을 네트웍을 이용해서 공통적인 문제들을 풀 수 있게 조화로운 하나의 세트로 묶은 개념으로 디자인하고 세운다는 의미이다.

분산 컴퓨팅에는 여러 가지 장점이 있다.

·성능: 문제들은 보통 하나의 프로세스에서 돌아가는 것보다 여러 개의 프로세스에서 동시에 돌리는 것이 더 빠르게 해결될 수 있다. 분산 컴퓨팅에서는 여러 프로세스들을 서로 연결하여서 문제를 해결할 수 있다. 많은 문제들은 여러 개의 작은 부분으로 나눌 수 있는데, 각각의 프로세스에 그 작은 부분을 할당함으로써 문제를 빨리 해결할 수 있게 된다.

·확장성: 하나의 컴퓨터에서만 돌아가는 에플리케이션은 그것이 돌아가는 프로세스의 성능과 규모에 의해 좌우가 된다. 대신에 여러 개의 프로세스가 합쳐져서 그 안에서 돌아가는 애플리케이션은 그만큼의 성능 향상 뿐만 아니라, 규모도 크게 할 수 있다.

·자원 공유: 소프트웨어의 문제뿐만 아니라, 데이터들과 자원들도 분산될 수 있다. 보통 각각의 로컬 프로세스에서 돌아가기 힘든 매우 큰 소프트웨어나 특

별한 장치를 필요로 하는 자원들은 어느 한 곳에서 자원들을 가지고 있으면 같이 공유할 수 있다. 그렇게 함으로써 불필요한 공간과 비용을 줄일 수 있다.

·에러 허용성과 유용성: 하나의 프로세스에서 돌아가는 애플리케이션은 실패했을 경우 다시 재부팅을 시키면 된다. 그것은 하나의 프로세스에만 영향을 준다. 하지만, 여러 프로세스가 서로 통신하고, 데이터를 공유하는 분산 환경에서 하나의 프로세스가 잘못 되는 경우에라도 나머지 다른 통신에 지장을 주어서는 안된다. 어느 한 부분에 문제가 생기더라도 나머지 다른 부분에는 유용성이 보장되어야 한다.

·일반적인 추세: 많은 경우에 있어서 이젠 대부분 분산 환경에서 돌아가는 시스템을 통해서 소프트웨어 문제를 해결한다. 문제는 점점 커지고 복잡해지기 때문에 작은 여러 부분으로 나눠서 해결하는 것이 더 효율적이다. 이러한 상황은 놀라운 게 아니고 요즘의 추세이다. 프로그래머들은 더 세분화되고 특성화된 일들을 해결하게 된다.

1. 분산 컴퓨팅에 대한 도전.

위와 같은 많은 장점들에도 불구하고, 분산 애플리케이션은 디자인하고 설계하고, 디버깅하는 것이 상당히 어렵다. 분산 환경은 로컬 애플리케이션에서는 문제되지 않는 여러 가지 여러움들이 나타나기 때문이다.

아마도 가장 어려운 문제는 서로 맞물려 잘 돌아가야 하는 다양한 종류의 머신 아키텍처들과 소프트웨어 플랫폼이다. 예전에는 이종(heterogeneity)의 프로세스들이 분산 애플리케이션을 개발하고 배포하는 것을 어렵게했다. 애플리케이션을 개발할 때는 모든 플랫폼에 돌아갈 수 있게끔 각각의 머신에 맞는 언어로 바꿔주어야 했다. 하지만 현재에는 자바 가상 머신이 네트웍을 통해 대부분의 플랫폼과 애플리케이션에서 실행될 수 있는 "Write Once, Run Anywhere™" 의 개념을 따른 클래스 파일들을 자동적으로 로딩해 주므로 이런 문제들을 해결해 주고 있다.

2. JavaSpaces™ 란 무엇인가?

JavaSpaces™ 는 프로세스들을 분산 애플리케이션으로 함께 실행될 수 있도록 해주는 높은 수준의 코디네이션 도구이다. 이 개념은 프로세스들 사이에서 메시지를 주고 받거나, 메소드들을 호출하는 기존의 분산 환경에서 시작되었다. 하지만 JavaSpaces 는 이와는 다르게, 하나 또는 그 이상의 *spaces* 에 오브젝트들을 넣어 *spaces* 가 오브젝트의 흐름을 제어하는 집합으로 보는 프로그래밍 모델로서의 방법을 제시하고 있다.

spaces 는 프로세스에 의해 공유될 수 있고, 네트웍을 통해서 접근할 수 있는 오브젝트 저장소이다. 프로세스는 직접 통신하는 대신에 지속적인 오브젝트 저장소나 메커니즘을 교환하는 곳으로 *spaces* 를 이용하고 있다. 프로세스는 *spaces* 에 새로운 오브젝트를 넣을 때 write 라는 간단한 명령어를 이용한다. *spaces* 에서 오브젝트를 꺼내올 때는 take 나 read 를 사용한다. take 는 원하는 오브젝트가 있는 경우 그걸 가져오고 나서 *spaces* 에서 그 오브젝트를 제거하고, read 는 복사본을 만들어서 *spaces* 에 그 오브젝트가 여전히 남아 있게 하는 명령어이다. 오브젝트를 가져올 때 프로세스는 원하는 오브젝트를 찾을 수 있는 간단한 value – matching lookup 을 이용한다. 만일 원하는 오브젝트가 없을 경우는 그 오브젝트가 *spaces* 에 들어올 때까지 기다린다. 오브젝트를 변경시키기 위해서는 직접 업데이트를 시켰던 기존의 오브젝트 저장소와는 달리 먼저 *spaces* 에서 그 오브젝트를 제거하고, 업데이트를 한 다음, 다시 *spaces* 에 올려놓아야 한다. Notify 는 사용자가 원하는 오브젝트가 없는 경우, 나중에 그 오브젝트가 *spaces* 에 올라왔을 때 그것을 알려주는 메소드이다. Snapshot 은 엔트리나 템플릿이 사용될 때마다 그것의 시리얼라이제이션이 매번 일어나는 것을 최소화 하기 위해 사용된다. *spaces* 기반의 애플리케이션을 설계하기 위해서는 분산 데이터 구조와 분산 프로토콜을 디자인해야 한다.

2.1 JavaSpaces™ 의 주된 특징.

JavaSpaces™ 에는 아래와 같이 여러 가지 특징이 있다.

·공유성: *Spaces* 는 많은 원격의 프로세스들이 동시에 상호 작용할 수 있는 네트웍으로 연결된 'Shared Memory'이다. *spaces* 는 동시에 일어나는 오브젝트의 접근들의 세세한 부분들을 제어한다. 'Shared Memory' 는 오브젝트들을 사용하여 동시에 분산 데이터 구조들을 설계할 수 있도록 해주고 있다.

·지속성: *spaces* 는 오브젝트를 위한 신뢰성 있는 저장소를 제공한다. 한번 *spaces* 에 저장이 되면 그 오브젝트는 제거되기 전까지 남아있게 된다. 프로세스는 오브젝트가 제거되는 시간(lease time)을 지정할 수도 있다. 그 시간이 지나면 오브젝트는 자동적으로 *spaces* 에서 제거가 된다.

·상호 작용성: 오브젝트는 메모리 로케이션이나 식별자보다 *associative lookup* 에 의해 배치된다. Associative lookup 은 누가 그 오브젝트를 만들었고, 누가 호출을 하고, 어디에 저장되어 있는지에 대한 정보를 알 필요 없이, 사용자가 찾고자 하는 오브젝트를 요구하는 내용에 의해서 쉽게 찾아준다. 오브젝트를 찾기 위해서는 요구하는 사항들을 적은 *template* 를 작성한다. *spaces* 에 있는 오브젝트가 그 *template* 와 적합하면 그 오브젝트를 반환하고, 만일 요구사항에 맞는 오브젝트가 없다면 *spaces* 에 그 오브젝트가 올라올 때까지 기다린다. *Template* 는 null 로 작성될 수도 있다.

·작업의 안전 보장: JavaSpaces™ 는 한 *spaces* 에서 일어나는 명령들이 한번에 수행이 되거나 다른 요소들에 의해서 실패하면 다시 실행하기 전의 상태로 복귀시켜 주는 트랜잭션 작업을 보장해준다.

·실행 데이터 교환: *spaces* 가 프로세스들이 통신하는 세부 사항을 관리하는 반면에 오브젝트들은 수동적인 데이터이다. 사용자는 오브젝트를 변경하거나 그

것의 메소드들을 호출할 수 없다. 다만 오브젝트를 *spaces* 에 읽거나 쓸 때는 로컬에 복사본이 생기게 된다. 그 다음에는 보통의 오브젝트들처럼 원하는 대로 변경할 수 있게 된다.

3. Compute server

컴퓨트 서버는 JavaSpaces™ 를 이용한 강력하고 광범위한 목적의 컴퓨팅 엔진이다. 작업들이 *space* 안에 들어오면 프로세스는 그것들을 실행하고 그 결과는 다시 *spaces* 로 되돌아온다. 컴퓨팅 서버는 결과를 계산할 뿐만 아니라, 그 작업이 완전하게 마쳐지게 하는 자원들을 관리한다.

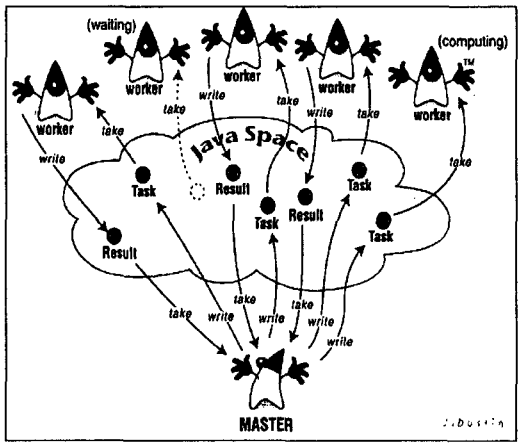
컴퓨트 서버는 작업을 받아들이고, 그것들을 수행하고, 결과를 되돌려주는 서비스를 제공한다. 서버는 작업을 더 빨리 수행하기 위해 여러 개의 프로세서나 특별한 목적의 하드웨어를 사용할 수도 있다. 컴퓨트 서버는 특별한 목적과 일반적인 목적의 두 가지 종류가 있다. SETI@home 프로젝트는 특별한 목적의 컴퓨트 서버의 대표적인 예인데, 미리 지정된 여러 개의 프로세스들을 연결시킨다. 평소에는 그 프로세스들이 각자의 서로 다른 일을 수행하고 사이클 시간이 비는 때에 특정한 일을 수행하는 것이다.

일반적인 목적의 컴퓨트 서버는 사용자가 어떤 일이라도 수행할 수 있게 한다. 사용자가 새로운 일을 첨가하면 서버는 자동적으로 그들의 작업에 그 새 일을 추가 시킨다. 서버는 현재 연결되어 있는 네트웍상에서 작업을 수행하는데 사용할 수 있는 프로세스를 찾는다.

컴퓨트 서버는 몇 개의 코드로 간단하게 설계할 수 있다. 하지만, 강력하고, 신뢰성 있는 서버를 만들려면 JavsSpace 와 JiniAPI 에 대해 많은 지식이 있어야 한다.

일반적인 *space* 기반의 컴퓨트 서버는 아래의 그림과 같다. 주 프로세서는 여러 개의 작업들을 활성화시키고 그것들을 *space* 에 올려놓는다. 여기서 말하는 작업(task)이라는 것은 특정한 작업을 설명하고, 필요한 계산들을 수행하는 메소드들을 포함하고 있다. 하나 또는 그 이상의 수행 프로세서가 *spaces* 를 모니터하고 있다가 작업이 올라오면 그것들을 계산해서 결과를 되돌려 준다. 결과(results)는 계산 이후에 나오는 값(output)을 포함하고 있는 데이터들이다.

컴퓨트 서버에는 여러 가지 장점이 있다. 첫째로 자연스럽게 규모를 조정할 수 있다. 대체로 수행 프로세스가 많으면 일을 빨리 처리할 수 있는데, 컴퓨트 서버는 런타임 때 수행 프로세스를 줄이거나 늘일 수 있고, 마지막 하나의 수행 프로세스가 돌 때까지 작업을 수행한다. 두 번째로 컴퓨트 서버는 작업이 각 프로세스마다 잘 분배될 수 있도록 로드 밸런싱을 조절해준다. 속도가 느린 프로세스에는 적은 양의 작업이 주어지고, 반면 빠른 프로세스에는 많은 작업을 분배해서 빨리 처리되게 하고, 한 프로세스에는 많은 작업이 몰리는 반면에 다른 프로세스는 쉬고 있는 불균형이 일어나지 않도록 한다. 또 주 프로세스와 작업 프로세스는 서로 강하게 묶여져 있지 않다. 주 프로세스는 작업들이 들어오면 그것들을 수행 프로세스에게 균등하게 분배하는 일을 한다. 수행 프로세스는 주 프로세스나 작업에 대해 전혀 알 필요가 없다. 단지 작업을 수행만 하면 되는 것이다. 마지막으로 *spaces* 는 이런 모든 작업들이 자연스럽게 이루어지도록 한다.



[그림 1] A space-based compute server

3.1 컴퓨트 서버 구현

```
package computeserver;

import net.jini.core.entry.Entry;

public interface Command extends Entry {
        pubic Entry execute();
}
```

[표 1] compute server

컴퓨트 서버를 구현하기 위해서는 command 라는 패턴을 사용하는데 이 패턴은 작업을 호출하는 오브젝트와 그 작업을 수행하는 오브젝트를 서로 분리한다. 오브젝트가 공간에 쓰여지기 위해서는 Entry 인터페이스를 구현하는 클래스에 의해 활성화가 되어야 한다.

The Worker

Worker 는 작업을 찾고 그것을 *space* 로부터 가져오고 계산한 다음에 결과를 돌려주는 역할을 한다. SpaceAccesor.getSpace 는 *jsbook.util* 패키지에서 제공된 것인데 이 메소드는 JavaSpace 라는 이름으로 로컬 지니 네트워크에서 JavaSpace 서비스를 찾는 역할을 한다. 서비스를 찾은 다음에 그것에 관한 조절권을 갖고 그것을 결과값으로 넘겨준다.

```
package computeserver;
```

```
import jsbook.util;

import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class Worker {
        JavaSpace space;

   public static void main (String[] args) {
       Worker worker = new worker ( );
       worker.startWork ();
 }
public Worker ( ) {
     space = SpaceAccessor.getSpace ( );
}
public void startWork ( ) {
     taskEntry template = new TaskEntry ( );

   For ( ; ; ) {
      try {
        TaskEntry task = (TaskEntry)
        space.take(template, null, Long.MAX_VALUE);
        Entry result = task.execute();
        if   (result !=null) {
             space.write (result, null, 1000*60*10);
      }
}catch (Exception e) {
     System.out.println ("Task Cancelled");
                        }
                }
      }
}
```

표 2. The Worker

이것으로 간단한 컴퓨트 서버는 완성되었다. 물론 서버의 성능을 향상시키기 위해서 몇가지 코드를 더 첨가할 수 있다. 하지만 위의 코드로도 서버의 역할을 할 수 있다는 것을 알게 되었다.

4. 향후 발전 방향 및 연구 과제

컴퓨트 서버는 광범위한 목적에 사용될 수 있고, 런타임에 프로세스들을 추가하거나, 삭제할 수 있고, 규모를 자동적으로 설정할 수 있고, 작업 균형을 잘 조절할 수 있는 등 여러 가지 장점이 있지만 그럼에도 불구하고 단점도 있다. 첫째, 지역적인 실패들을 고려하지 않는 점이다. 여러 개의 프로세스에서 돌아가는 JavaSpaces™ 인 경우에 그런 고려 사항들이 해결되어야 한다. 또 다른 단점으로는 하나의 CPU 에서 돌아가는 하나의 JavaSpace 를 사용할 때이다. 어떤 환경에서 하나의 *space* 에 의존하는 것은 병목현상을 일으킬 수 있다.

그러므로 이 간단한 컴퓨트 서버에 여러 가지 기능을 향상 시키고, 좀 더 강력한 도구로 만드는 것에 관심을 두어야 한다.

참고문헌

[1] Eric Freeman, Susanne Hupfer, and Ken Arnold "JavaSpaces™ Principles, Patterns and Practice", Addison-Wesley, 1999

[2] "JavaSpaces™ Specification", Sun, 1999

[3] Susanne Hupfer, "Make room for JavaSpaces, part1", JavaWorld, 1999

[4] Sussane Hupfer, "The Nuts and Bolts of Compiling and Running JavaSpaces Programs", Sun, 1999

[5] Sussane Hupger, "Make room for JavaSpaces, part3", JavaWorld, 2000

[6]"Jini™ Specification", Sun, 1998

[7] Erich Gamma, "Design Patterns, Elements of Reusable object-Oriented Software", Addison-Wesley, 1995