제26회 한국정보처리학회 추계학술발표대회 논문집 제13권 제2호 (2006. 11)

AOP를 이용한 이동 에이전트의 투명한 이주 기법 설계1)

임원택, 김구수, 엄영익
성균관대학교 정보통신공학부
e-mail:{imcliff, gusukim, yieom}@ece.skku.ac.kr

Transparent Migration Scheme of Mobile Agent by using AOP

Wontaek Lim, Gu Su Kim, Young Ik Eom
School of Information and Communication Engineering,
Sungkyunkwan University

요 약

Java 기반의 이동 에이전트 시스템은 에이전트의 실행 상태를 이주시킬 수 없는 문제점을 가지고 있다. 이러한 문제점을 해결하기 위해 JVM을 수정하거나 이주에 필요한 소스 코드, 혹은 바이트 코드를 삽입하는 메커니즘을 이용한 이동 에이전트 시스템들이 연구되었다. 하지만 이러한 시스템들은 이식성이 떨어지거나 에이전트의 이주부분을 프로그래밍 할 수 없는 단점이 존재한다. 또한 이러한 시스템들은 플랫폼의 요청에 의한 에이전트 이주 기법인 forced migration을 지원하지 않는다. 본 논문에서는 AOP(Aspect Oriented Programming)를 이용한 에이전트의 투명한 이주 기법을 제안한다. 제안 기법에서는 에이전트를 비즈니스 로직, 이동성 코드, 상태 저장 코드로 나누어 개발하고, 이를 직조하여 이동 에이전트를 개발한다. 제안 기법을 사용하면 에이전트 개발자는 이동 에이전트의 비즈니스 로직 개발에 집중할 수 있고 에이전트의 이주 부분을 프로그래밍 함으로써, 유연한 에이전트의 이주 정책을 수립할 수 있다.

1. 서론

이동 에이전트는 서로 다른 실행 환경 사이를 이주할 수 있는 소프트웨어 컴포넌트이다[1]. 이동 에이전트는 자율성, 적응성, 비동기적 실행 등 여러 가지 장점을 가지고 있다[2]. 최근의 이동 에이전트 시스템은 대부분 자바를 기반으로 구현되고 있다[3,4]. 하지만 자바 기반 이동 에이전트 시스템은 이동 에이전트의 이주 시, 에이전트의 실행 상태를 이주시킬 수 없는 문제점을 가지고 있다.

이러한 자바 기반 이동 에이전트의 문제점을 해결하기 위해 여러 가지 시스템들이 연구되었다[5,6]. 하지만 이러한 이동 에이전트 시스템은 JVM을 수정함으로써 이식성이 떨어지거나, 소스코드 및 바이트코드를 삽입하는 기법을 사용하여 에이전트 개발자가 이동 에이전트의 이주 부분을 프로그래밍 할 수 없는 한계점을 가진다. 또한, 이러한 기법들은 플랫폼의 요청에 의한 에이전트 이주 기법인 forced migration을 지원하지 못하는 단점을 가진다.

본 논문에서는 AOP (Aspect Oriented Programming)를 이용하여 에이전트 코드와 함께 에이전트의 상태를 이주시키는 이주 기법을 제안한다. 본 기법은 에이전트의 비즈니스 로직과 에이전트의 이주 부분을 분리하여 구현함으로써 에이전트 개발의 유연성, 투명성을 제공하고, forced migration을 지원한다.

본 논문은 2장에서 이동 에이전트의 이동성에 관련된 연구와 AOP를 소개하고, 3장에서는 이동 에이전트의 구성을 설명한다. 4장에서는 이동 에이전트 구현을 위한 알고리즘을 설명하고, 5장에서는 AOP를 이용한 이동 에이전트의 간단한 예를 보인다. 마지막으로 6장에서는 제안 기법에 대한 결론과 향후 연구를 설명한다.

2. 관련 연구

본 장에서는 자바 기반 이동 에이전트의 이동성에 관한 연구를 살펴보고, AOP에 대해 간략히 소개한다.

1) 본 연구는 21세기 프론티어 연구개발사업의 일환으로 추진되고 있는 정보통신부의 유비쿼터스컴퓨팅및네트워크원천기반기술개발사업의 지원에 의한 것임 (2006-0391-0100).

2.1 투명한 이주를 지원하는 자바 기반의 이동 에이전트 시스템

자바 기반의 이동 에이전트 시스템은 자바 가상 머신에서 제공하는 클래스 로딩과 직렬화, 리플랙션 기능을 이용하여 실행 코드와 클래스의 멤버 변수를 이주시킬 수 있지만, 에이전트의 실행 상태를 이주시킬 수는 없다. 이러한 자바 기반 이동 에이전트 시스템들이 에이전트의 실행 상태를 이주시킬 수 있도록 하기 위해서 여러 시스템이 개발되었다[5,6]. 이러한 시스템들은 JVM을 수정하거나, 프리컴파일링(pre-compiling)을 통해 소스코드나 바이트코드를 삽입하여 에이전트의 실행상태를 저장하고 복원한다. 하지만 JVM을 수정할 경우 이식성이 떨어지는 문제점을 가지며, 소스코드나 바이트코드를 삽입할 경우 에이전트의 이주 부분을 프로그래밍 할 수 없는 한계점을 가진다. 또한, 두 가기 경우 모두 플랫폼의 이주 요청에 의한 에이전트 이주 기법인 forced migration을 지원하지 못한다.

2.2 AOP(Aspect-Oriented Programming)

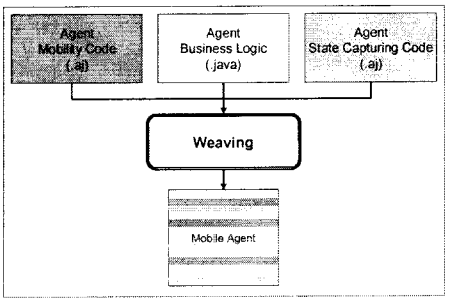
AOP는 횡단 관심사의 모듈화를 지원하기 위해 1996년 제록스 팔로 알토 연구소의 연구원인 Gregor Kiczales에 의해 발표되었다[7].

AOP는 시스템을 중요 기능인 핵심 관심사와 시스템 전반에 사용되는 부가적인 기능인 횡단 관심사로 나누어 개발한다. 횡단 관심사에는 횡단 관심사가 핵심 관심사에 삽입될 부분을 정의한 직조 규칙(weaving rule)이 존재한다. 시스템은 개발된 관심사들을 직조 규칙에 따라 직조함으로써 통합된다. 핵심 관심사에는 횡단 관심사에 대한 명시적인 호출이 없으므로, 코드의 재사용성이 증대된다. 또한 여러 관심사의 코드가 한 모듈에 나타나는 코드 혼합(code tangling)문제와 한 모듈의 구현 부분이 여러 모듈에 분산되는 코드 산재(code scattering)문제를 해결할 수 있다.

3. 이동 에이전트의 구성

본 논문에서 제안하는 이주 기법은 에이전트를 에이전트 비즈니스 로직, 에이전트 이동성 코드, 에이전트 상태 저장 코드의 세 부분으로 나눈다. 에이전트 비즈니스 로직은 에이전트가 수행해야 할 작업들에 대한 자바 코드이다. 에이전트 이동성 코드는 에이전트 이주 요청에 관한 인터럽트 체크와 플랫폼에서 제공하는 에이전트 이주 관련 함수의 호출을 수행하는 코드이다. 에이전트 상태 저장 코드는 특정 지점에서의 에이전트의 상태를 저장하는 코드이다. 에이전트 이동성 코드와 에이전트 상태 저장 코드는 Aspect의 형태로 구현된다.

세 부분으로 개발된 이동 에이전트는 직조과정을 통해 하나의 이동 에이전트로 통합된다. (그림 1)는 에이전트의 통합 과정을 보인다.



(그림 1) 이동 에이전트의 통합 과정

4. 알고리즘

본 장에서는 이동 에이전트를 구현하기 위한 직조 규칙과 에이전트 이동성 코드 및 에이전트 상태 저장 코드의 알고리즘을 설명한다.

4.1 직조 규칙

이동 에이전트의 직조를 위해서는 직조 규칙이 필요하다. <표 1>은 이동 에이전트의 직조에 필요한 직조 규칙을 나타낸다.

<표 1> 이동 에이전트의 직조 규칙

횡단 관심사	결합 지점
에이전트 상태 저장 코드	메서드 호출 직전
	루프의 시작점
에이전트 이동성 코드	메서드 실행 직후
	루프의 시작점

AspectJ에서는 루프를 나타내는 결합점을 사용할 수 없다. 따라서 에이전트 개발자는 Agent 클래스의 멤버 변수인 bMigrate를 접근함으로써 루프의 시작점임을 나타낸다.

AspectJ에서의 직조 규칙은 애스펙트(aspect)의 구성 요소인 교차점(point cut)과 충고(advice)에 의해 정의된다. (그림 2)와 (그림 3)는 애스펙트의 형태로 나타낸 이동 에이전트의 직조 규칙을 보인다.

```
import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

public aspect StateCapture {
  public pointcut captureStates() : call(* Agent.*(..)) ||
```

```
                                        set(* Agent.bMigrate);
  before() : captureStates() {
    // 에이전트의 상태 저장 코드
  }
}
```

(그림 2) 에이전트 상태 저장 코드의 애스팩트

```
public aspect AgentMobility {
  public poincut interruptCheck() : call(* Agent.*(..)) ||
                                        set(* Agent.bMigrate);
  after() : interruptCheck(){
    // 인터럽트 체크
    // 플랫폼의 이주 관련 메서드 호출
  }
}
```

(그림 3) 에이전트 이동성 코드의 애스팩트

4.2 에이전트 상태 기록 코드의 알고리즘

에이전트 상태 기록 코드는 특정 시점에서의 이동 에이전트의 상태를 기록하는 역할을 담당하는 횡단 관심사이다. 에이전트 상태 기록 코드는 AspectJ에서 제공하는 리플렉션 기능을 이용하여 에이전트 클래스의 정보, 호출 된 메서드의 정보, 에이전트 상태 기록 코드가 직조된 위치 등의 정보를 얻어온다. 얻어진 정보는 에이전트 상태 기록 코드의 내부 스택에 저장된다. (그림 4)는 에이전트 상태 기록 코드의 알고리즘을 보인다.

```
captureStates() {
  Get Class Information;
  Get Arguments of point cut;
  Get source code information (Line);

  Information put into stack;
}
```

(그림 4) 에이전트 상태 기록 코드 알고리즘

4.3 에이전트 이동성 코드의 알고리즘

에이전트 이동성 코드는 특정 위치에서 에이전트 이주 요청 인터럽트가 발생했는지를 검사하고, 만일 에이전트의 이주 요청 인터럽트가 발생했다면 목적지 플랫폼의 정보를 얻어와 플랫폼에서 제공하는 에이전트 플랫폼에서 제공하는 에이전트 이주 함수를 호출한다. (그림 5)는 에이전트 이동성 코드의 알고리즘을 보인다.

```
interruptCheck() {
  if(Agent.has migration interrupt) {
    Get Target Platform Information
    Platform.migrate(Agent, targetPlatform);
  }
}
```

(그림 5) 에이전트 상태 기록 코드 알고리즘

5. 실행 예제

본 장에서는 직조 규칙에 따른 에이전트의 통합을 보여주는 실행 예제를 보인다. 작성된 실행 예제는 AspectJ 1.5.2를 기반으로 작성되었다. (그림 6)는 에이전트 클래스를 보인다.

```
public class Agent extends Thread{
  public boolean bMigrate = true;
  public boolean bInterrupt = false;

  public Agent(){
    bInterrupt = false;
  }
  public void run() {
    System.out.println("[Func1 is called]");
    int result = func1();
    System.out.println("Result is " + result);
  }
  private int func1(){
    int result = 0;
    for(int j = 1; j <= 3; j++){
      System.out.println("[bMigrate is set (j=" + j + ")]");
      bMigrate = true;
      result += j;
    }
    return result;
  }
}
```

(그림 6) 예제 Agent 클래스

Agent 클래스는 Java 스레드를 상속받고, 에이전트 이주의 인터럽트 체크와 상태 저장의 교차점을 지정해주는 멤버 변수인 bMigrated와 에이전트의 이주 인터럽트 상태를 나타내는 멤버변수인 bInterrupt를 가진다. 예제 코드는 run 메서드에서 func1을 호출하고 func1은 루프를 돌며 1에서 3까지의 합을 구해 반환한 후 이 값을 화면에 출력한다.

예제에서 사용된 에이전트 상태 저장 코드는 저장될 정보들을 화면에 출력해 주는 역할을 한다. 에이전트 이동성 코드는 정확한 위치에 에이전트 이동성 코드가 직조되는지를 알아보기 위한 코드이다. (그림 7)와 (그림 8)는 예제에서 사용된 에이전트 상태 저장 코드와 에이전트 이동성 코드를 보인다.

```
import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

public aspect StateCapture {
  public pointcut captureStates() : call(* Agent.*(..)) ||

  before() : captureStates(){
    System.out.println("-----------------------");
    System.out.println("JoinPoint : " + thisJoinPoint.getThis());
    StringBuffer argStr = new StringBuffer("Args: ");
    Object[] args = thisJoinPoint.getArgs();
    for (int length = args.length, i = 0; i < length; ++i)
      argStr.append(" [" + i + "] = " + args[i]);
    System.out.println(argStr);
    System.out.println("Kind: " + thisJoinPointStaticPart.getKind());
    SourceLocation sl =
                    thisJoinPointStaticPart.getSourceLocation();
```

```
    System.out.println("Source location: " +
                    sl.getFileName() + ":" + sl.getLine());
  }
}
```

(그림 7) StateCapture.aj의 코드

```
public aspect AgentMobility {
  public pointcut interuptCheck() : call(* Agent.*(..)) ||
                                    set(* Agent.bMigrate);
  after() : interruptCheck() {
    System.out.println("Interrupt checking!!");
  }
}
```

(그림 8) AgentMobility.aj의 코드

예제를 실행시킨 결과를 (그림 9)에서 보인다.

```
1   -----Agent Thread Create-----
2   ===Agent State Capture===
3   JoinPoint : Thread[Thread-0,5,main]
4   Args:  [0] = true
5   Kind: field-set
6   Source location: Agent.java:4
7   ===Agent Interrupt Check===
8   -----Agent Thread Start-----
9   ===Agent State Capture===
10  JoinPoint : null
11  Args:
12  Kind: method-call
13  Source location: TestAgent.java:14
14  ===Agent Interrupt Check===
15  [Func1 is called]
16  ===Agent State Capture===
17  JoinPoint : Thread[Thread-0,5,main]
18  Args:
19  Kind: method-call
20  Source location: Agent.java:11
21  [bMigrate is set (j=1)]
22  ===Agent State Capture===
23  JoinPoint : Thread[Thread-0,5,main]
24  Args:  [0] = true
25  Kind: field-set
26  Source location: Agent.java:18
27  ===Agent Interrupt Check===
28  [bMigrate is set (j=2)]
29  ===Agent State Capture===
30  JoinPoint : Thread[Thread-0,5,main]
31  Args:  [0] = true
32  Kind: field-set
33  Source location: Agent.java:18
34  ===Agent Interrupt Check===
35  [bMigrate is set (j=3)]
36  ===Agent State Capture===
37  JoinPoint : Thread[Thread-0,5,main]
38  Args:  [0] = true
39  Kind: field-set
40  Source location: Agent.java:18
41  ===Agent Interrupt Check===
42  ===Agent Interrupt Check===
43  Result is 6
```

(그림 9) 예제 실행 결과

에이전트 스레드가 생성되고, 시작 될 때 각각 한 번씩 에이전트 상태 저장과 인터럽트 체크가 일어난다(1~14 줄). 이것은 에이전트 스레드의 생성자와 run 메서드가 호출되기 때문이다. func1이 호출되어 에이전트의 상태가 저장된다(15~20 줄). func1에서 루프를 돌며 에이전트 상태를 저장하고 인터럽트 체크를 하고(21~41줄), func1이 종료될 때 인터럽트 체크를 하고(42줄) 수행된 결과를 화면에 출력한다(43줄).

6. 결론

본 논문에서는 AOP를 이용한 에이전트의 투명한 이주 기법을 제안하였다. 제안 기법은 에이전트를 에이전트가 수행해야 할 작업인 에이전트 비즈니스 로직을 핵심 관심사로, 에이전트의 상태 저장과 에이전트 이주 관련 코드를 횡단 관심사로 분리하였다. 에이전트 개발자는 에이전트 비즈니스 로직에 집중하여 에이전트를 개발할 수 있다. 또한 에이전트 이주에 관련된 부분을 프로그래밍 할 수 있도록 함으로써, 보다 유연한 에이전트 이주를 가능하게 하였다.

본 논문에서 제안된 기법을 본 연구실의 경량 이동 에이전트 플랫폼인 KAgentSystem에 적용하여 투명한 에이전트 이주 기법을 지원하는 이동 에이전트 시스템의 프로토타입을 제작할 예정이다.

참고문헌

[1] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility," IEEE Transactions on Software Engineering, 24(5), May 1998.

[2] Danny B. Lange and Mitsuru Oshiama, "Seven good reasons for mobile agents," Communcations of the ACM, 42(3), Mar. 1999.

[3] D.B. Lange, M. Oshima, G. Karjoth, and K. Kosaka, "Aglets: Programming Mobile Agents in Java," Proceedings of International Conference Worldewide Computing and Its Applications, LNCS 1274, Mar. 1997.

[4] F. Bagci, J. Petzold, M. Trumler, and T. Ungerer, "Ubiquitous Mobile Agent System in a P2P-Network," In UbiSys-Workshop at the Fifth Annual Conference on Ubiquitous Computing, Oct. 2003.

[5] Ashish Malgi, Neelesh Bansod, and Byung Kyu Choi, "STRING: Efficient Implementation of Strongly Migrating Mobile Agents in Java," Proceedings of International Conference on Parallel and Distributed Computing Systems, Sep. 2005.

[6] H. Peine, "Run-Time Support for Mobile Code," Doctoral thesis, University of Kaiserslautern, Oct. 2002.

[7] G. Kiczales, "Aspect-oriented programming," Proceedings of European Conference on Object-Oriented Programming, 1997.