동적 환경에 적합한 자율 이동 에이전트의 이주 기법

(A Migration Technique for Autonomous Mobile Agents in Dynamic Environments)

복 경 수 † 여 명 호 †† 유 재 수 †††
(Kyoung Soo Bok) (Myung Ho Yeo) (Jae Soo Yoo)

요 약 본 논문에서는 동적 환경에 적합한 자율 이동 에이전트의 이주 기법을 제안한다. 제안하는 이주 기법은 네트워크 및 시스템의 상태 변화에 적절하게 대응할 수 있도록 에이전트의 이주 경로를 동적으로 생성한다. 에이전트의 이주 과정에서 전송해야 할 데이타량 및 에이전트의 인스턴스를 생성하기 위한 시간을 감소시키기 위해 다음에 이동할 목적지에 프리패칭 메시지를 전달하여 필요한 데이타를 미리 수신할 수 있도록 한다. 제안하는 이동 기법은 에이전트의 이주 과정에서 발생한 결함에 대처할 수 있도록 체크포인트 기법을 이용하여 에이전트의 상태를 유지하고 복구할 수 있도록 하는 기능을 제공한다. 제안하는 이주 기법의 우수성을 입증하기 위해 다양한 실험을 통해 기존 방법의 비교 분석을 수행한다.

키워드 : 자율 이동 에이전트, 동적 경로, 이주, 프리패칭

Abstract In this paper, we propose a migration technique for autonomous mobile agents suitable to dynamic environments. The proposed migration technique dynamically creates the itinerary of agents that considers states of networks and systems. In the migration of the agent, it first sends prefetching message to the next system. The system receives necessary data for migration in advance. Through this, we reduce the amount of the sending data and save the time for creating the instance of the agent. And it improves the execution efficiency by using the checkpoint-based recovery method that does not execute the agent again and recovers the process states even though the errors take place. To show superiority of the proposed technique, we compare the proposed method with the existing methods through various simulations.

Key words : Autonomous Mobile Agent, Dynamic Itinerary, Migration, Perfetching

1. 서 론

에이전트는 사용자를 대신해서 사용자가 원하는 작업을 자동적으로 수행하여 주는 소프트웨어이다. 이러한 에이전트는 일반적인 소프트웨어와 달리 자율성을 가지고 능동적으로 작업을 수행한다[1]. 초기에 에이전트는 인공지능 분야에서 사용자를 대신하여 자율적으로 작업을 수행할 수 있는 지능형 에이전트에 대한 연구가 시작되었다. 그러나 복잡한 사용자의 요구를 해결하기 위해 여러 에이전트들 사이에 분산 협동의 필요성이 증가됨에 따라 80년대 말부터 독립적인 연구 분야로 분리되어 연구되었다. 특히, 90년대 인터넷의 급속한 발전과 함께 다양한 사용자의 서비스를 수행하기 위해서 에이전트의 필요성이 증가되고 있다. 에이전트에 대한 정의는 매우 다양하지만 일반적으로 에이전트는 다음과 같은 특성을 갖는다[2,3].

- 에이전트는 사용자나 다른 에이전트의 직접적인 지시나 간섭 없이도 스스로 판단하여 수행할 수 있는 자율성을 갖는다.
- 에이전트는 작업을 수행하거나 다른 에이전트에 도움을 주기 위해 다른 에이전트 또는 사용자와 상호 관계성을 갖는다.

· 본 연구는 한국과학재단 목적기초연구(특정기초연구 과제번호:R01-2003-000-10627-0) 지원 및 산업자원부의 지역혁신 인력양성사업의 연구결과로 수행되었음
† 비 회 원 : 한국과학기술원 전산학과
ksbok@dbserver.kaist.ac.kr
†† 비 회 원 : 충북대학교 정보통신공학과
mhyeo@netdb.cbnu.ac.kr
††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@cbucc.chungbuk.ac.kr
(Corresponding author임)
논문접수 : 2004년 2월 27일
심사완료 : 2005년 9월 28일

- 에이전트는 환경의 변화를 인지하고 변화에 대응하는 행동을 수행하며 경험을 바탕으로 학습하는 기능을 갖는다.
- 에이전트는 행동의 결과로 환경의 변화를 가져올 수 있으며 에이전트의 행동은 한번에 끝나는 것이 아니라 지속적으로 이루어진다

최근 클라이언트/서버 환경의 문제점인 지연 및 대역폭 문제와 네트워크 연결 문제를 해결하기 위해 에이전트의 한 분야로 이동 에이전트에 대한 연구가 활발히 연구되고 있다. 이동 에이전트는 네트워크를 통해 수행에 필요한 코드, 데이타 또는 상태 정보를 특정 시스템으로 이주하면서 작업을 수행하고 최종 결과를 원래 작업을 요청한 시스템으로 전달된다. 이동 에이전트는 작업을 수행할 코드 자체를 이동하기 때문에 원격지에 있는 서버와의 통신을 감소시켜 네트워크의 부하와 트래픽을 감소시킨다. 또한 수행에 필요한 코드와 데이타를 한번에 이동하여 작업을 수행하기 때문에 네트워크 지연에 따른 대기 시간을 감소시켜 분산 시스템 응용 개발에 적합하다. 이동 에이전트는 네트워크 상에서 객체를 캡슐화하여 이동하는 것이 가능하여 독자적인 프로토콜 구현 및 통신이 가능하다. 또한 이동 에이전트는 자율적이고 비동기적으로 수행이 가능하기 때문에 회선 접속 상태가 원활하지 않은 곳에서 효율적이며 주위 환경 변화에 능동적으로 반응하여 자기 자신을 이동시킬 수 있기 때문에 최적화된 성능을 발휘할 수 있다[4,5].

이러한 이동 에이전트는 네트워크를 통해 코드나 데이타를 다른 시스템으로 이동하기 때문에 이동할 데이타의 크기 및 네트워크의 상태 또는 다음에 이동할 시스템의 상태에 따라 많은 영향을 받는다[6]. 이동 에이전트의 상호 운용성을 지원하기 위해 FIPA와 OMG에서는 표준화를 진행 중에 있다. FIPA에서는 FIPA98 표준에서 이동 에이전트 이동성을 지원하기 위한 "Agent Management Support for Mobility"라는 표준을 발표하였다[7]. 또한 OMG에서는 1998년 MASIF (Mobile Agent System Interoperability Facility)라는 표준을 발표하였다[8]. 그러나 이러한 표준화는 실제적인 에이전트의 이주를 위한 시스템 구성에 중심이 되어 이루어지고 있으며 실제적인 에이전트의 이주 기법에 대한 상세한 표준화는 진행되지 않고 있다.

기존에 개발된 이동 에이전트 시스템인 JAMES[9], Ajanta[10], ASDK[11], Grasshopper[12] 등에서는 에이전트의 이주를 위한 다양한 기법들이 제안되었다. 그러나 기존 에이전트의 시스템들에서 제공하는 이주 기법은 동적인 네트워크 및 시스템의 상태 변화에 적응하지 못할 뿐만 아니라 이주 과정에서 전송해야 할 데이타의 크기를 고려하지 못하고 있다. 또한 이주 과정에서 발생할 수 있는 여러 가지 결함에 대한 처리 기능을 제공하지 못하기 때문에 네트워크 및 시스템의 결함이 발생할 경우 이동하는 에이전트가 손실될 수 있는 문제점이 있다.

본 논문에서는 동적 환경 변화에 적합한 이동 에이전트의 새로운 이주 기법을 제안한다. 제안하는 이주 기법은 네트워크 및 시스템의 상태 변화에 적절히 대응할 수 있도록 에이전트의 이주 경로를 동적으로 생성하는 기능을 제공한다. 또한 에이전트의 이주 과정에서 전송해야 할 데이타량을 감소시키기 위해 이동할 목적지에 프리패칭 메시지를 전달하여 필요한 데이타를 미리 수신할 수 있도록 한다. 에이전트 코드를 수신한 시스템에서는 에이전트의 인스턴스를 미리 생성하고 필요한 데이타를 메모리에 적재하기 때문에 에이전트의 수행 시간을 단축시킬 수 있다. 제안하는 이주 기법은 에이전트의 이주 과정에서 발생한 결함에 대처할 수 있도록 체크포인트 기법을 이용하여 에이전트의 상태를 유지하고 복구할 수 있도록 하는 기능을 제공한다.

본 논문의 구성은 다음과 같다. 2장에서는 이동 에이전트의 개념과 기존에 제안된 에이전트 시스템에 사용된 이주 기법의 특징을 기술한다. 3장에서는 제안하는 이주 기법에 필요한 시스템의 특성을 기술하고 4장에서 제안하는 에이전트의 이주 기법을 기술한다. 5장에서는 실험을 통해 기존에 제안된 이주 기법과의 비교 분석을 수행하고 마지막 6장에서는 결론 및 향후 연구 방향에 대해 기술한다.

2. 관련 연구

2.1 이주 기법

이동 에이전트는 다수의 시스템을 이주하면서 사용자가 요청한 작업을 자율적으로 수행하고 수행된 결과만을 원래 작업을 요청한 시스템으로 전달한다. 이동 에이전트는 사용자의 작업 요청을 하나의 시스템에서 처리할 수 없을 때에는 해당 작업을 수행하기 위한 이주 경로를 생성하고 이주 경로에 따라 시스템을 이주하면서 사용자의 작업 요청을 수행한다. 일반적으로 이동 에이전트는 다음과 같은 과정을 수행하면서 이주를 진행한다. 먼저, 에이전트가 이동할 목적지 시스템을 결정해야 한다. 이동할 목적지가 결정되면 현재 에이전트가 수행되고 있는 시스템에서 수행 중인 에이전트의 실행 상태를 중지한다. 실행 중인 에이전트가 중지되면 전송에 필요한 상태와 데이타들을 생성하기 위해 에이전트를 직렬화(serialization)를 수행한다. 직렬화가 수행되면 전송 프로토콜에 적합하도록 부호화를 수행하여 네트워크를 통해 에이전트를 전송하게 된다. 에이전트를 수신한 시스템에서는 비직렬화(deserialization)를 수행하여 에이

전트를 복원하고 에이전트를 재실행하게 된다.

이러한 이동 에이전트의 이주는 이주 과정에서 전송되는 정보에 따라 강 이주(strong migration)와 약 이주(weak migration)로 구분된다[13]. 강 이주 기법은 목적지에 실행에 필요한 코드, 데이타 그리고 에이전트의 실행 상태 모두를 목적지 시스템에 전송한다. 강 이주에서 전송된 코드와 실행 상태는 목적지 시스템에서 이주가 시작한 시점을 기준으로 쓰레드(thread)를 활성화하여 에이전트의 작업을 계속 진행할 수 있다. 이에 반해 약 이주는 에이전트의 실행 상태는 전송하지 않고 에이전트가 수행한 데이타와 코드만을 전송하게 된다.

이동 에이전트는 다수의 시스템을 이주하면서 작업을 수행하기 때문에 시스템 환경에 독립적으로 수행할 수 있어야 한다. 따라서, 대부분의 에이전트 시스템은 JVM(Java Virtual Machine) 상에서 자바를 이용하여 개발하고 있다. 그러나 자바는 코드 이동 및 데이타 이동을 지원하지만 스택에 직접 접근이 불가능하기 때문에 완벽한 에이전트의 실행 상태 즉, 쓰레드 이동을 지원하지 못한다. 따라서 에이전트의 강 이주 기법을 지원하기 위해서는 JVM의 바이트 코드를 수정하여 스택 정보를 가져오는 방법을 사용하거나 응용 레벨에서 실행 상태정보를 자바의 객체에 프레임별로 저장하여 이동시키는 방법으로 개발되고 있다[14,15].

2.2 에이전트 시스템의 이주 기법

지난 몇 년 동안, 사용자의 요청을 대신하여 수행할 에이전트의 실행 환경을 제공하기 위한 많은 에이전트 시스템들이 개발되었다. 현재 개발된 에이전트 시스템은 표준의 지원 여부에 따라 FIPA 계열과 OMG의 MASIF 계열로 구분되고 있다. IBM의 도쿄 연구실에서 개발한 ASDK(Aglets Software Development Kit)은 자바로 개발된 이동 인터넷 에이전트(mobile internet agent)이다[11]. ASDK은 인터넷을 통해 서로 다른 시스템을 이동할 수 있는 자바 객체로 Aglet을 생성하고 GUID(Globally Unique Identifier)라는 고유한 식별자를 부여받는다. ASDK는 MASIF 계열의 에이전트 시스템으로 Tahiti 서버라 불리는 GUI 기반의 자바 실행 환경을 제공하여 이동 에이전트를 쉽게 개발할 수 있다. ASDK에서는 에이전트의 이주를 위해 dispatch, retractAglet, MobilityListener 기능을 제공한다. dispatch는 직렬화를 통해 목적지 시스템에 에이전트를 전달하고 인스턴스를 생성한다. retractAglet는 dispatch와 같은 기능을 하지만 dispatch와는 다르게 풀(Pull) 방법을 이용하여 이전 시스템으로 에이전트가 돌아가게 한다. MobilityListener는 두 가지의 메소드 dispatch, retractAglet를 제공한다. 이러한 ASDK는 이주 시 사용자의 정보 및 이동할 주소 그리고 Aglet에서 사용된 클래스에 대한 바이트 코드 복사본을 바이트 배열에 저장하여 목적지에 이주시킨다. 즉, 자바 객체 직렬화 기법과 비직렬화 기법을 사용하여 Aglet의 코드와 스택 정보는 보내지 않고 힙에 있는 상태 정보를 스트림으로 보내고 받는 약 이주 기법을 사용하였다[16]. ASDK는 단순히 에이전트 코드를 필요에 따라 전송하기 때문에 동적인 환경 변화에 적절히 대응하지 못하며 이주에 필요한 데이타의 크기 그리고 결함에 대해 적절히 대처하지 못하는 문제점이 있다.

Minnesota 대학에서는 보안 및 강건한 하부구조를 만들기 위한 목적으로 Ajanta라는 이동 에이전트 시스템을 개발하였다. Ajanta는 자바 직렬화, 반사(reflection), RMI(Remote Method Invocation) 기능을 이용하여 이동 에이전트를 구현하였다[10]. 에이전트의 이주는 네트워크 통해 이동할 에이전트를 자바의 직렬화 기법을 이용하여 상태 정보를 이주하고 클래스 로딩 기법을 이용하여 요구한 클래스만을 이동시키는 약 이주 기법을 사용한다. 프록시 중재(proxy interposition)를 통해 시스템 자원에 접근하는 이주 기법을 제공한다. Ajanta에서 제공하는 클래스 로딩 기법은 수행에 필요한 코드만을 이동하는 방법을 사용하여 전송할 데이타 양을 최소화함으로써 네트워크 부하를 감소시킨다. 또한 에이전트의 수행에 적합한 이주 패턴을 선택하고 선택된 이주 패턴에 따라 에이전트를 수행함으로써 이주 성능을 향상시킬 수 있다. 그러나 Ajanta의 이주 기법은 요구한 클래스만을 이동시키는 방법을 사용했지만 이주 데이타의 크기를 고려하지 못하는 문제점과 결함에 대한 처리를 고려하지 못한다는 문제점이 있다.

기존의 클라이언트/서버 환경에서는 네트워크 트래픽 및 지연시간과 같은 문제가 발생한다. 이러한 문제를 개선하고자 이동 에이전트에 대한 연구가 진행되었고 이에 따라 강건하고 쉬운 갱신 및 네트워크 트래픽 감소의 효과를 가져오게 되었다. 이에 JAMES는 네트워크 관리 및 통신에 있어 성능을 향상시키고자 개발된 이동 에이전트 시스템이다[9]. JAMES는 고성능의 수행 효율, 효과적인 코드이주, 강건한 네트워크 관리의 특성이 있다. 효과적인 코드이주를 위해 코드 프리패칭, 캐쉬 스키마, 쓰래드 및 소켓 폴링 기법을 적용하였다. JAMES의 이주기법은 클라이언트가 서비스를 요청하게 되면 중앙 호스트에서 이동 에이전트가 생성되고 이주 경로를 생성한다. 이주 정보에 따라 필요한 코드를 각 에이전트 시스템에 메시지 형태로 전송하고 메시지를 수신한 에이전트 시스템에서는 자신이 필요한 코드를 자신의 캐쉬메모리, 이전 에이전트 시스템의 캐쉬메모리, 자신의 보조기억장치에서 찾는다. 만약 코드가 없는 경우에는 JAMES 코드 서버에 필요한 코드를 요청하여

미리 코드를 받는다. 이러한 프리패칭 기법을 이용한 JAMES 시스템은 이주할 때 발생할 수 있는 네트워크 트래픽을 개선하였다. 그러나 JAMES의 이주 기법은 동적인 환경변화 즉, 목적지 시스템의 자원 그리고 네트워크의 상태를 고려하지 못한다는 문제점과 이주 과정에서 발생하는 결함에 대한 처리를 제공하지 못한다.

Grasshopper는 GMD FOKU와 IKV++에서 이동 에이전트 실행환경을 개발한 것으로 FIPA(Foundation for Intelligent Physical Agents) 스팩에 따라 최초로 개발된 시스템이다[8]. Grashopper는 소켓, RMI, CORBA, Socket+SSL(Secure Sockets Layer), RMI+SSL 기술과 자바를 사용하여 시스템이나 하드웨어를 수정하지 않고 에이전트를 개발할 수 있는 시스템으로 실행 상태를 제외한 코드 및 데이타를 목적지 시스템에 이동시키는 약 이주 기법을 사용한다. 그러나 Grasshopper는 목적지 시스템의 자원 및 네트워크의 상태 변화에 적절히 대응하지 못할 뿐만 아니라 이주할 데이타의 크기 그리고 결함에 대한 처리를 고려하지 못한다.

3. 최적의 이주 기법을 지원하기 위한 에이전트 시스템

이 장에서는 이동 에이전트에 대한 동적 환경 변화에 적응할 수 있는 새로운 이주를 지원하기 위한 에이전트 시스템에 대해 기술한다. 제안하는 이주 기법은 네트워크 및 시스템의 상태 변화에 따라 동적으로 이동 경로를 생성하고 이주 과정에서 발생할 수 있는 다양한 문제에 적응할 수 있다. 먼저 에이전트의 이주를 위한 에이전트 시스템의 특성을 기술하고 실제 에이전트의 이주를 위해 필요한 기능에 대해 기술한다.

3.1 이동 에이전트 시스템

이동 에이전트는 서로 다른 시스템을 이주하면서 사용자의 작업 요청을 수행하게 된다. 이러한 이동 에이전트가 동적 환경 변화에 적응하면서 자율적으로 시스템을 이주하기 위해서는 시스템의 상태 정보는 물론 현재 수행 중이거나 시스템에 등록되어 있는 에이전트의 상태를 관리해야 한다. 또한 이주할 목적지 시스템의 위치를 관리하고 통신을 수행하기 위한 기능을 제공해야 한다. 그림 1은 에이전트 시스템의 구조를 나타낸 것이다. 에이전트 시스템은 실제적인 사용자의 작업을 대신 수행하는 에이전트 계층(agent layer)과 에이전트가 실행하기 위해 필요한 다양한 기능을 제공하는 에이전트 서비스 계층(agent service layer)으로 구성된다. 에이전트 계층은 사용자의 작업 요청을 처리하기 위해 이동 경로를 따라 시스템을 이주하면서 자율적으로 작업을 수행한다. 에이전트 서비스 계층은 에이전트가 자율적으로

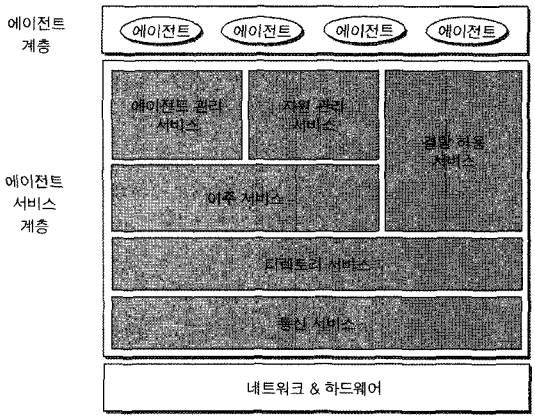


그림 1 에이전트 시스템

사용자의 작업을 수행할 수 있는 실행 환경을 제공하는 것으로 에이전트를 관리하고 에이전트의 수행에 필요한 서비스를 제공한다. 이러한 에이전트 서비스 계층은 이주 서비스(migration service), 에이전트 관리 서비스(agent management service), 자원 관리 서비스(resource management service), 결함 허용 서비스(fault-tolerant service), 디렉토리 서비스(directory service) 그리고 통신 서비스(communication service)로 구성되어 있다.

에이전트 관리 서비스는 사용자의 작업 요청에 따라 에이전트를 생성하고 시스템에서 동작하는 에이전트의 상태 및 기능을 관리하는 역할을 수행한다. 에이전트가 생성되면 에이전트 관리 서비스는 생성된 에이전트를 시스템에 등록하고 에이전트에 쓰레드를 할당한다. 쓰레드가 할당된 에이전트가 실제 작업을 수행하기 위해 에이전트 관리 서비스는 에이전트 스케쥴링을 수행하여 에이전트의 작업 순서를 결정하고 작업에 대한 우선 순위를 결정한다. 또한 에이전트 관리 서비스는 에이전트의 실행 상태 즉, 에이전트의 일시 정지, 정지, 재시작과 같은 에이전트의 실행을 제어한다.

이동 에이전트는 네트워크의 부하 및 실행의 최적화를 위해 자율적으로 시스템을 이동할 수 있는 이주 기능이 있다. 이주 서비스는 에이전트의 이주에 관련된 기능을 제공하는 서비스로 이주에 필요한 이동 경로(itinerary) 생성, 필요한 코드의 이동에 대한 비용을 최소화하기 위해 필요에 따라 동적으로 이동 경로 생성 그리고 생성된 이주 경로에 프리패칭 기능을 제공한다. 에이전트 관리 서비스에서 에이전트의 스케쥴링에 의해 작업 순서가 결정되면 이주 서비스에서는 시스템의 상태 및 네트워크의 상태에 따라 이주 경로를 생성한다. 에이전트의 이주 경로가 결정되면 에이전트의 수행 시간 및 코드의 이동에 따른 네트워크의 부하를 최소화하

기 위해 다음에 이주할 시스템에 필요한 코드를 프리패칭한다.

자원 관리 서비스는 로컬 또는 원격지 시스템에서 사용되는 자원을 관리하고 모니터링하는 기능을 제공한다. 하나의 시스템은 다수의 에이전트들이 동시에 특정 작업을 수행할 수 있다. 자원 관리 서비스에서는 각각의 에이전트들이 사용하는 CPU 및 메모리 점유율, 사용 가능한 메모리, 쓰레드 수 및 네트워크의 상태를 관리한다. 이러한 자원의 상태 정보는 일정 간격으로 모니터링되어 계속적으로 갱신되며 에이전트의 이주 과정에서 특정 시스템에 이동이 가능한지를 판별할 때 사용된다.

결함 허용 서비스는 에이전트 또는 시스템의 결함에 대해 에이전트의 실행이 중지되거나 에이전트의 수행 결과가 손실되는 문제를 해결하기 위한 기능을 제공하는 것으로 체크포인트 기능과 복제 기능을 제공한다. 체크포인트 기법은 에이전트 수행 과정에서 발생할 수 있는 다양한 결함에 대처하기 위해 에이전트의 수행 상태를 안전한 저장 장치에 기록하고 이에 대한 로그 정보를 기록한다. 그러나 체크포인트 기법은 에러에 대한 복구가 수행되는 동안 에이전트의 수행이 중지되기 때문에 에이전트의 복사본을 동시에 수행시키는 복제 기법을 동시에 제공한다.

디렉토리 서비스는 원격지에 존재하는 시스템의 위치 및 상태를 유지하고 시스템에서 수행 중인 에이전트와 협력하는 에이전트의 위치를 관리한다. 또한 에이전트가 다른 시스템으로 이주할 필요가 있을 때 시스템의 위치를 검색하는 기능을 제공한다. 하나의 에이전트가 다른 시스템으로 이주를 수행하기 위해서 각 시스템에서 제공하는 서비스 및 상태 정보를 이용하여 이주 경로를 생성한다. 디렉토리 서비스는 이러한 에이전트의 이주를 위해 필요한 에이전트 또는 시스템의 위치 및 상태를 검색하는 기능을 제공한다.

통신 서비스는 로컬 또는 원격지에 존재하는 시스템 또는 에이전트들 사이의 물리적인 통신을 수행하는 기능을 제공한다. 통신 서비스는 시스템 식별자, 시스템에서 제공하는 서비스의 종류 및 에이전트의 상태를 바탕으로 로컬 또는 원격지에 존재하는 시스템 또는 에이전트에 메시지 및 데이타를 전송한다. 일반적으로 에이전트 또는 시스템 사이의 통신은 메시지 전송에 의해 수행되며 각 시스템은 메시지 전송에 따른 응답을 수신하여 에이전트 및 시스템의 상태를 모니터링할 수 있다. 에이전트의 이주를 위해서는 디렉토리 서비스를 통해 이주에 적당한 시스템을 선택하고 통신 서비스를 통해 메시지를 전송하거나 실제 에이전트의 코드를 전송한다.

3.2 이주에 필요한 기능

이동 에이전트는 사용자가 요청한 작업을 수행하기 위해 이주 경로를 따라 다수의 시스템을 이동하면서 작업을 완료하게 된다. 이러한 이동 에이전트의 이주를 수행하기 위해서는 에이전트의 코드를 이동하기 위한 직렬화 기법이 필수적이다. 직렬화란 데이타가 아니라 객체를 저장하고 읽어 들이는 방법으로 객체의 내용을 바이트 단위로 변환하여 파일 또는 네트워크를 통해서 송수신 스트림이 가능하게 만들어 주는 것을 말한다. 즉, 복잡한 데이타 구조를 갖는 객체를 직렬화시켜 파일이나 배열에 저장하고 이를 객체로 다시 복원해 내는 것이다. 이 기술은 더 나아가 네트워크 상에서 코바 기술을 이용하여 객체를 전송하는데 활용할 수 있다. 그림 2는 객체 직렬화 기법을 보여주고 있다[17].

에이전트가 다른 시스템으로 이주하였을 때에는 에이전트 수행에 필요한 에이전트 코드를 시스템에 적재하는 기능이 필요하다. 이러한 기능을 제공하기 하는 것이 동적 클래스 로더(Dynamic class loader)이다. 유닉스 시스템의 .os파일, 윈도우 시스템의 DLL파일은 동적으로 링크되는 파일이다. JVM의 경우에는 모든 프로그램이 동적으로 링크되어 있다. 모든 클래스들은 한 번에 하나의 클래스만 적재하고 필요할 때마다 새로운 클래스를 적재한다. 즉, 프로그래머는 모든 프로그램이 동적으로 적재되므로 별다른 구분 없이 프로그램 내의 모든 부분들에 대해서 동일하게 접근할 수 있다. 그림 3은 클래스 로더의 구조이다. JVM이 클래스 파일을 동적으로 로드할 수 있는 것은 java.lang.ClassLoader 라는 클래스에 의해서 가능하다. 클래스 로더는 적재, 연결, 초기

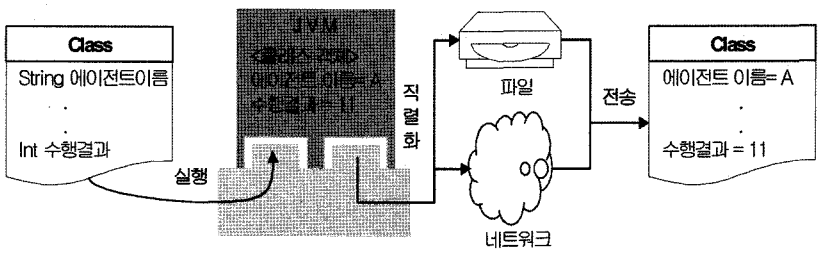


그림 2 객체 직렬화

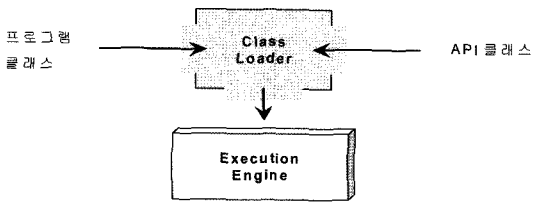


그림 3 자바 클래스 로더 구조

화의 과정을 거쳐 클래스를 사용할 준비를 한다. 적재는 클래스의 이름을 사용하여 클래스 파일 형태의 바이트를 찾고, 클래스의 구현 정보를 JVM에게 알리는 작업을 수행한다. 적재 단계를 통해 JVM은 클래스의 이름과 클래스 계층도상에서 클래스 위치 및 그 클래스의 필드와 메소드의 종류를 알게 된다. 연결은 클래스가 기본적인 형태를 제대로 갖추고 있는지, JVM의 보안과 관련된 제약조건에 맞는지에 대한 클래스의 검증하는 단계, 그리고 스택 초기화를 위한 모듈을 수행한다. 이 검증 과정의 부가적 영향으로 여러 클래스들이 함께 적재되며, 이 작업이 끝나면 클래스를 사용할 준비가 완료된다.

에이전트 이주의 성능을 향상시키기 위해서는 에이전트의 수행에 필요한 코드를 미리 전송하고 수행에 필요한 인스턴스 및 데이타를 메모리에 적재하는 프리패칭 기법이 필요하다. 일반적으로 프리패칭은 가까운 미래에 참조될 가능성이 높은 데이타 블럭을 예측하여 이를 미리 프로세서에 인접한 메모리로 가져와서 메모리 접근 시간을 줄인다. 프리패칭을 수행하기 위해서는 메시지 전송을 수를 증가시키고 이를 처리하기 위해 로컬 시스템의 부담을 증가시킬 있다. 그러나 프리패칭은 에이전트의 이주 과정에서 전송해야 할 네트워크의 부하를 줄이고 작업 수행에 있어 네트워크 내에서 빠른 응답시간과 지연시간 감소로 원격지 영역에서 수행되는 작업의 효율을 기대할 수 있다. 이러한 프리패칭은 로컬 영역의 저장 장치, 메모리, 네트워크 전송량 등의 절약할 수 있으며 서버 측으로의 데이타 접근 횟수와 처리량을 늘릴 수 있다. 즉, 네트워크상에서 이동 에이전트의 전체적인 수행에 있어 수행에 필요한 코드를 미리 이동할 시스템으로 이동 후 인스턴스화 하면 에이전트 이주 시 전송해야 할 데이타 량을 감소시켜 이주 시간을 감소시킬 수 있고 전체적인 수행 시간을 감소시킬 수 있다.

시스템의 에러 또는 에이전트 자체의 문제로 인해 에이전트는 해당 작업을 수행하지 못하고 중지되는 상황이 발생할 수 있다. 특히, 이동 에이전트의 이주 과정하면서 에러가 발생하거나 네트워크의 문제로 인해 이주하는 에이전트의 데이타가 손실되는 경우에는 심각한 문제를 발생시킨다. 따라서, 다양한 에러에 대처할 수 있도록 하기 위해 주기적으로 체크포인트를 수행하여 에이전트의 수행을 복구할 수 있는 기능을 제공한다. 이러한 체크포인트 과정은 이주 과정에서 시스템의 부하를 증가시킬 수 있다. 그러나 체크포인트를 수행하지 않을 경우 이주 과정에서 발생하는 에러에 대처할 수 없으며 현재까지 수행된 데이타 또는 에이전트의 코드를 손실 할 수 있다. 또한, 목적지 시스템으로 이주가 완료되기 전에 에러가 발생할 경우 전송되어야 할 데이타량이 증가된다. 따라서, 제안하는 이주 기법에서는 실제 이주 과정에서 발생하는 다양한 에러에 대처할 수 있도록 하기 위해 목적지 시스템으로 이주하기 전에 체크포인트를 수행하고 목적지 시스템에서 에이전트의 이주가 완료되기 전에 체크포인트를 수행한다.

4. 제안하는 이주 기법

4.1 특징 및 이주 절차

제안하는 이주 기법은 데이타 크기, 네트워크의 상태 그리고 이동할 시스템의 상태를 고려하여 동적 환경에 적합하도록 설계한다. 제안하는 이주 기법은 에이전트의 수행 효율을 증가시키기 위해 에이전트 생성시 각 시스템에서 수행 가능한 서비스, 통신 비용, 시스템 상태 그리고 사용 가능한 자원 등에 대한 비용 검사를 통해 최소의 비용으로 이주가 가능한 시스템의 리스트를 생성한다. 또한, 에이전트 수행 중 네트워크 부하 및 시스템 자원 부족 등의 문제가 발생 시 동적으로 이주 경로를 생성함으로써 실행 환경변화에 적절하게 대응할 수 있으며 성능 향상을 가져온다. 시스템 또는 에이전트의 결함에 대처할 수 있도록 이주 후 코드 및 데이타 그리고 실행상태를 안정된 저장장치에 체크포인트 기법을 이용하여 저장한다. 에이전트의 실행 시간을 최소화하고 네트워크의 트래픽을 감소시키기 위해 필요한 코드를 미리 각 시스템에 이동하여 코드의 이동하는 프리패칭 기법을 제공한다. 제안하는 에이전트 시스템에서 제공하는 이주 과정은 그림 4와 같이 8단계의 과정을 거쳐 수행된다.

사용자의 서비스 요청에 따라 에이전트 관리 서비스는 CreateAgent()를 수행하여 작업을 수행할 에이전트를 생성한다. CreateAgent()는 에이전트를 생성하고 에이전트가 수행할 작업 순서를 결정하기 위한 스케줄링 작업을 수행한다. 스케줄링 작업에 의해 작업 순서가 결정되면 이주 서비스에서 제공하는 CreateItinerary()를 이용하여 해당 작업을 수행할 이주 경로를 생성한다. CreateItinerary()는 자원 관리 서비스와 디렉토리 서비스를 이용하여 이주 비용을 최소로 할 수 있는 이주 경로를 생성한다. 에이전트의 이주 경로가 결정되면 RequestPrefetching()을 이용하여 다음에 이주할 N개의

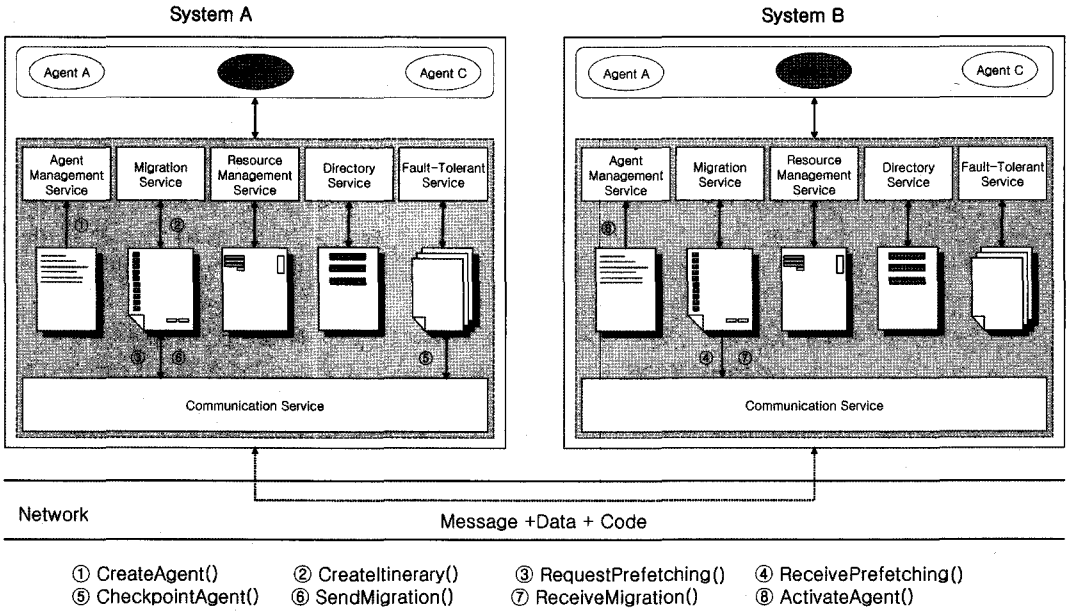


그림 4 이동 에이전트 이주 절차

시스템에 프리패칭 메시지를 전달한다. 프리패칭 메시지는 통신 서비스를 이용하여 네트워크를 통해 전달된다. 통신 서비스를 통해 전달된 프리패칭 메시지는 ReceivePrefetching()을 통해 메시지를 수신하고 에이전트 관리 서비스를 통해 현재 시스템에 해당 코드를 포함하고 있는지를 확인한다. 만약 프리패칭 메시지를 수신한 시스템에 해당 코드가 없을 경우에는 디렉토리 서비스를 통해 필요한 코드를 수신할 적당한 시스템을 선택하고 해당 시스템에 필요한 코드를 전송하라는 메시지를 전달한다. RequestPrefetching()과 ReceivePrefetching()은 비동기적으로 수행되며 프리패칭 메시지를 전달하는 RequestPrefetching()은 단순히 에이전트 수행에 필요한 코드가 무엇인지에 대한 메시지만을 전달하고 실제적으로 필요한 코드의 수신은 ReceivePrefetching()에 의해 수행된다. 에이전트의 코드가 수신되면 해당 시스템은 코드에 대한 인스턴스를 생성하고 필요한 데이타를 메모리에 적재한다. 프리패칭 메시지를 수신한 시스템은 해당 코드에 대한 수신 여부를 프리패칭 메시지를 전달한 시스템에 전달한다.

프리패칭 메시지를 전달한 시스템은 프리패칭 메시지의 수신 여부와 독립적으로 에이전트의 작업을 수행한다. 에이전트는 작업을 수행하면서 시스템 또는 에이전트의 결함에 대처할 수 있도록 주기적으로 체크포인트를 수행한다. 에이전트의 작업이 완료되면 다음에 이주할 시스템에 에이전트를 이동하기 위해 SendMigration()을 수행한다. SendMigraion()에서는 에이전트 관리 서비스를 통해 에이전트의 수행 상태를 중지시키고 CheckItinerary()를 이용하여 다음에 이동할 시스템에 이주가 가능한지를 판별한다. 에이전트의 이주 여부가 판별되면 CheckpointAgent()를 이용하여 에이전트의 수행 상태를 안전한 저장 장치에 기록하고 필요에 따라 새로운 이주 경로를 생성한다. 에이전트의 수행 상태가 저장되면 에이전트 이주를 위해 필요한 데이타를 직렬화한다.

SendMigratoin()은 이주에 필요한 데이타를 생성하기 전에 이전에 프리패칭 메시지에 의해 전달된 프리패칭 상태를 확인한다. 이주 서비스에서는 프리패칭 메시지를 전달하면 이에 대한 수신 여부를 응답하도록 되어있다. 이 정보를 이용하여 프리패칭 메시지에 의해 다음에 이동할 시스템에 필요한 코드가 존재하는지를 확인한다. 이에 따라 SendMigration()은 필요한 코드와 데이타를 생성하여 다음에 이동할 시스템에 전달하게 한다. 이주할 시스템에서는 에이전트의 수행에 필요한 코드 및 데이타가 전달되면 ReceiveMigration()에 의해 전송된 코드 및 데이타를 수신한다. 수신된 코드 및 데이타는 에러 검사를 수행하고 에이전트 수행에 필요한 코드를 에이전트 관리 서비스에 등록한다. 에이전트 관리 서비스에 에이전트가 등록되면 체크포인트를 수행하여 에이전트의 현재 상태를 안전한 저장 장치에 기록하고 ActiveAgent()를 수행하여 에이전트를 활성화한다.

4.2 이주 경로 생성

사용자의 서비스 요청에 따라 에이전트 관리 서비스에서는 에이전트를 생성하고 에이전트 수행에 필요한 작업 순서를 결정하기 위한 스케쥴링을 수행한다. 에이전트의 작업 순서가 결정되면 에이전트 관리 서비스는 이주 서비스에 작업 순서에 따른 이주 경로 생성을 요청한다. 에이전트의 이주 경로를 생성하기 위해서는 에이전트 서비스에서 제공하는 디렉토리 서비스와 자원 관리 서비스가 필요하다. 에이전트의 이주에 따라 해당 작업을 수행할 시스템을 선택하기 위해 디렉토리 서비스에서는 해당 시스템의 상태를 관리해야 한다. 표 1은 디렉토리 서비스에서 관리하는 시스템의 상태 정보를 나타낸 것이다. 시스템의 상태 정보에서는 시스템에서 제공하는 서비스의 유형과 위치 정보를 관리하여 에이전트의 이주 또는 에이전트들 사이의 상호 협동 작업이 필요할 경우 에이전트의 작업을 수행할 시스템을 선택할 수 있다. 이러한 시스템의 상태 정보에 표현된 서비스의 이름은 에이전트의 스케쥴링에 의해 선택된 작업 유형을 수행할 수 있는 기능을 나타낸 것이다.

표 1 시스템의 상태 정보

시스템 이름	서비스 이름	위치
Sys1	S1, S2	210.115.147.43
Sys2	S2, S3, S4	210.225.13.42
Sys3	S1, S3, S5	204.125.157.34
Sys4	S1, S4, S5	205.126.24.53

에이전트의 이주에 필요한 최적의 경로를 생성하기 위해서는 에이전트의 작업을 수행할 수 있는 시스템 중 에이전트 실행 또는 이주에 필요한 비용을 최소로 하는 시스템을 선택해야 한다. 따라서 자원 관리 서비스에서는 각 시스템에서 에이전트의 수행에 필요한 자원 및 네트워크를 상태를 관리하고 이주 경로에 필요한 비용을 최소로 하는 시스템을 선택할 수 있어야 한다. 표 2

표 2 시스템의 자원 정보

시스템 이름	쓰레드 가용성	메모리 가용성	네트워크 가용성
Sys1	50%	60%	60%
Sys2	70%	55%	40%
Sys3	90%	85%	60%
Sys4	20%	30%	90%

는 자원 관리 서비스에서 관리되는 시스템의 자원 정보를 나타낸 것이다. 에이전트가 생성되거나 이주하여 사용자에 의해 요청된 작업을 수행하기 위해서는 쓰레드를 할당하고 시스템에서 특정 메모리를 사용한다. 따라서, 에이전트의 실행 가능성을 증가시키고 실행 시간을 감소시키기 위해서는 에이전트의 작업에 할당 가능한 쓰레드 및 사용 가능한 메모리의 상태를 관리해야 한다. 또한 에이전트는 네트워크를 통해 다수의 시스템을 이동하기 때문에 에이전트의 이주 시간을 감소시키기 위해서는 네트워크 전송 비용을 관리해야 한다.

이주 서비스에서 제공하는 CreateItinerary()는 에이전트의 생성과 함께 이주 경로를 생성한다. 그림 5는 에이전트의 이주 경로를 생성하는 CreateItinerary()의 수행 과정을 나타낸 것이다. 에이전트 관리 서비스에 의해 생성된 에이전트는 Scheduler()에 의해 에이전트의 작업 순서가 결정된다. 에이전트의 작업 순서가 결정되면 이주 서비스에서는 CreateItinerary()를 수행하여 에이전트의 이주 경로를 생성한다. 에이전트의 이주 경로를 생성하기 위해서는 먼저 CheckService()를 수행하여 디렉토리 서비스에서 관리되는 시스템의 상태를 확인하고 에이전트의 작업이 수행 가능한 시스템을 선택한다. 만약 에이전트 시스템의 상태 정보가 표 1과 같을 때, 에이전트가 수행 해야할 작업이 Job1에서 Job5까지 존재하고 Jobi의 수행이 Si에 의해 가능하다고 할 때 표 3은 수행 가능한 시스템을 선택한 예를 나타낸다. CheckService()에 에이전트의 각 작업이 수행 가능한 시스템이 선택되면 CheckResource()를 수행하여 수행 가능한

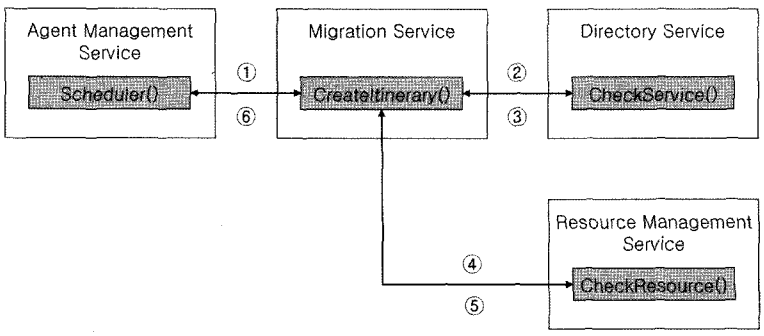


그림 5 이주 경로 생성

표 3 작업이 수행 가능한 시스템

작업 이름	서비스 이름
Job1	Sys1, Sys3, Sys4
Job2	Sys1, Sys2
Job3	Sys2, Sys3
Job4	Sys2, Sys4
Job5	Sys3, Sys4

에이전트의 각 작업에 대한 실제적인 이주 경로를 결정한다. CheckResource()에서는 에이전트의 작업을 수행하기 위한 필요한 시스템의 자원을 사용하기 위해 필요한 비용을 계산하여 각 작업을 수행할 수 있는 시스템을 정렬한다. CreateItinerary()는 에이전트의 이주 비용에 따라 정렬된 시스템에 따라 최적의 이주 경로를 생성한다. 생성된 이주 경로를 Scheduler()에 전달되어 에이전트 관리 서비스에 등록된다.

4.3 프리패칭

에이전트의 이주 경로가 결정되면 에이전트 시스템은 다음에 이주할 N개의 시스템에 프리패칭 메시지를 전송하여 에이전트 수행에 필요한 코드를 미리 수신하고 인스턴스를 생성한다. 이러한 프리패칭은 에이전트의 수행에 필요한 인스턴스를 미리 생성하여 메모리에 적재하기 때문에 에이전트의 수행 시간을 감소시킬 수 있다. 또한, 실제 이주 과정에 전송해야 할 데이타의 크기를 감소시킬 수 있기 때문에 네트워크 부하 및 지연에 따른 문제를 해결한다. 그림 6은 프리패칭 과정을 나타낸 것이다. 에이전트의 이주 경로에 프리패칭을 수행하기 위해서는 먼저 이주 서비스에서는 RequestPrefetching()을 수행한다. RequestPrefetching()은 프리패칭 메시지를 전송할 N개의 시스템에 필요한 코드가 무엇인지를 판별하고 해당 코드의 이름과 해당 코드를 전송 받기에 가장 적당한 시스템의 위치 정보를 판별한다. 수신해야 할 코드와 위치 정보가 판별되면 이주 서비스는 통신 서비스에 각 시스템에 해당 메시지를 전송하기 위한 Message()를 수행한다. 이때, 에이전트 시스템은 RequestPrefetching()의 수행 결과와 독립적으로 메시지를 전송만을 수행하고 에이전트의 해당 작업을 수행한다. Message()는 각 시스템에 프리패칭할 코드와 위치 정보를 전송한다.

프리패칭 메시지를 수신한 시스템은 Receive-Prefetching()을 수행하여 프리패칭할 코드의 상태를 확인하고 필요한 코드를 수신한다. 각 시스템은 시스템에 등록된 코드와 등록된 코드의 기능 및 상태를 관리한다. 따라서 프리패칭할 코드가 시스템에 등록되어 있는지를 확인하기 위해 CheckCode()를 수행한다. 만약 프리패칭 요청에 대한 코드가 등록되어 있다면 통신 서비스를 통해 프리패칭 메시지를 전송한 시스템에 응답메시지를 전송한다. 이를 통해 불필요한 코드의 전송을 제거할 수 있다. 만약 프리패칭 요청에 대한 코드가 등록되어 있지 않다면 프리패칭 메시지에 포함된 코드를 수신할 위치 정보를 이용하여 해당 코드를 요청한다. 코드 전송에 대한 요청을 수신한 시스템은 SendCode()를 수행하여 요청된 코드에 대한 직렬화를 수행하여 코드를 요청한 시스템에 코드를 전송한다. 코드를 수신한 시스템은 TestCode()를 수행하여 수신한 코드에 에러 검사를 수행하고 에러가 없을 경우 RegiesterCode()를 통해 코드를 등록하고 필요한 인스턴스를 생성하여 메모리에 적재한다.

4.4 에이전트의 이주

에이전트가 생성되어 특정 시스템에서 필요한 작업에 대한 수행이 완료되면 에이전트는 이주 경로를 따라 다음 목적지 시스템으로 이동한다. 에이전트의 이주를 수행을 위해 이주 서비스는 이동에 필요한 데이타를 직렬화하여 전송하기 적절한 형태로 변환하고 네트워크를 통해 목적지 시스템으로 전송한다. 목적지 시스템에서는 이를 수신하여 시스템에 등록하고 에이전트의 작업을 계속 수행하기 위해 에이전트를 재시작한다. 에이전트의 이주를 수행하기 위해 이주 서비스는 SendMigration()

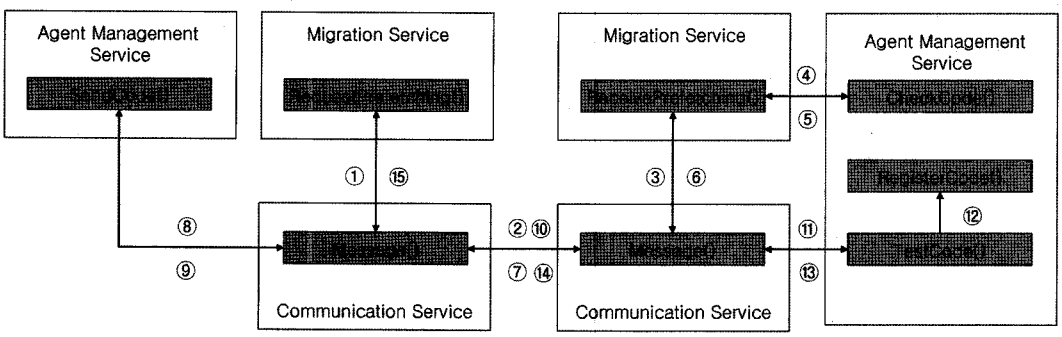


그림 6 프리패칭 과정

을 수행하여 에이전트의 수행에 필요한 코드와 데이타를 전송한다. 그림 7은 SendMigration()의 수행 과정을 나타낸 것이다. 에이전트의 이주를 수행하기 위해서는 SuspendAgent()를 수행하여 현재 수행 중인 에이전트의 상태를 중지시키고 CheckpointAgent()를 수행하여 현재 상태를 안전한 저장 장치에 기록한다. 에이전트의 상태가 기록되면 이주 경로를 따라 다음에 이주할 목적지 시스템에 필요한 코드가 수신되었는지를 확인하기 위해 CheckPrefetching()을 수행한다. 프리패칭 메시지는 다음에 수행할 작업을 최소의 이주 비용을 이동할 수 있는 다음 N개의 목적지 시스템에 필요한 코드 수신을 요청하고 이에 대한 코드 수신 여부를 기록하였다. 이를 통해 필요한 코드의 수신 여부를 확인한다. 이러한 과정은 실제 과정에서 전송해야 할 데이타 량을 감소시킬 수 있으며 중복된 코드 및 데이타의 전송을 방지할 수 있다. 필요한 코드의 수신 여부를 확인하면 해당 목적지 시스템에 에이전트를 이주할 수 있는지를 확인하기 위해 CheckItinerary()를 수행한다. CheckItinerary()에서는 이주 경로에 따라 다음에 이주할 목적지 시스템의 상태를 확인하고 에이전트가 이주하여 수행이 가능한지를 판별한다. 에이전트의 생성 과정에서 설정된 정적 경로에 의해 에이전트의 수행이 불가능한 시스템에 에이전트를 이주할 경우 에이전트는 해당 작업을 수행하지 못하고 중지된 상태로 존재하게 된다. 이러한 중지 상태는 에이전트의 전체적인 수행 시간을 증가시키게 되어 성능이 저하되는 문제점이 있다. 따라서, 에이전트가 네트워크 또는 시스템의 문제에 의해 에이전트가 이주할 수 없거나 목적지 시스템에서 할당 가능한 자원이 없어 에이전트의 수행이 불가능한 경우에는 다음에 수행해야 할 작업이 수행 가능한 시스템 중 에이전트의 이주가 가장 적합한 시스템을 선택하여 이주 경로를 다시 생성한다.

에이전트의 이주 경로가 확인되면 만약 다음에 이동할 목적지 시스템에 프리패칭 메시지가 전달되지 못했거나 프리패칭 메시지를 수신했지만 필요한 코드를 수신하지 못한 경우에는 다음에 수행에 필요한 코드를 모두 직렬화하여 전송한다. 그러나 목적지 시스템이 프리패칭 메시지에 의해 필요한 코드를 수신한 경우에는 수행에 필요한 일부 코드와 데이타만을 전송한다. Serialization()과 MakeSendData()는 목적지 시스템에서 수행할 코드를 직렬화하여 필요한 데이타를 생성한다. 에이전트 이주에 필요한 코드와 데이타가 모두 생성되면 SendData()를 수행하여 다음 목적지 시스템에 수행에 필요한 코드, 데이타 및 이주 경로를 함께 전송한다.

에이전트가 이주되면 목적지 시스템에서는 이주된 에이전트의 코드와 데이타를 수신하고 필요한 코드를 등록하고 에이전트 수행에 필요한 초기화 작업을 수행한다. 그림 8은 목적지 시스템에서 이주된 에이전트를 수신하여 에이전트를 수행하는 ReceiveMigration()의 수행 과정을 나타낸 것이다. 네트워크를 통해 수신된 코드와 데이타는 ReceiveData()를 통해 수신하여 이를 ReceiveMigration()에 전달한다. ReceiveMigration()은 Deserialization()을 수행하여 직렬화되어 전송된 에이전트 코드에 대해 비직렬화를 수행하고 TestData()를 수행하여 수신된 코드와 데이타에 대한 바이러스 및 오류를 검사한다. 만약 수신된 코드 및 데이타에 문제가 발

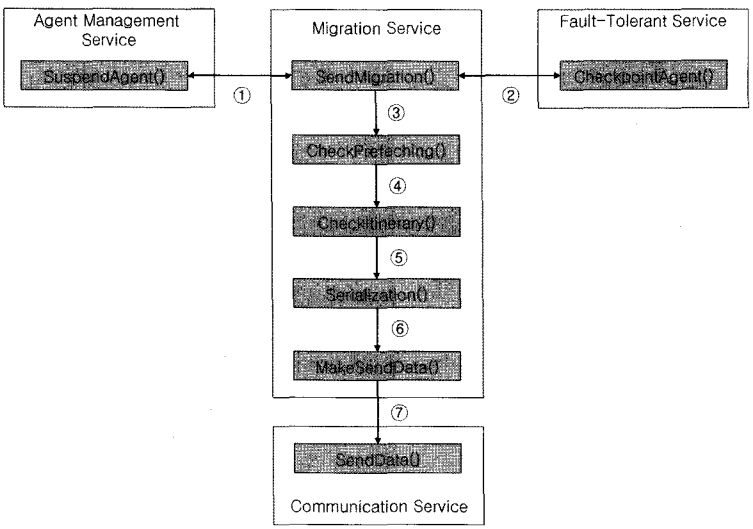


그림 7 에이전트 전송

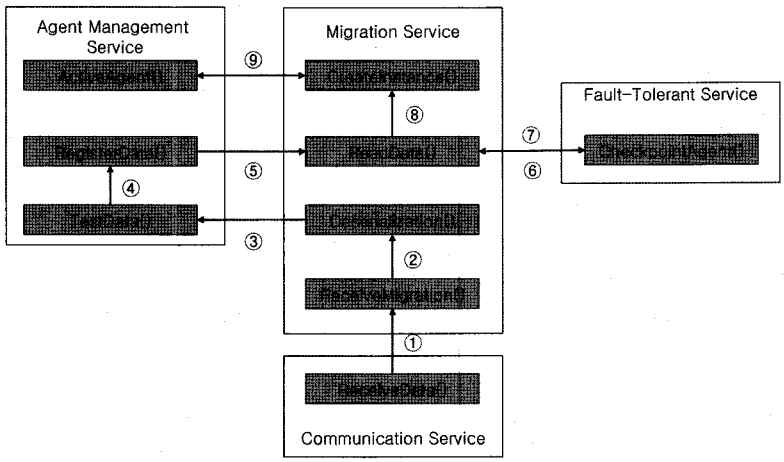


그림 8 에이전트 수신

생할 경우 필요한 내용을 다시 전송하도록 메시지를 전달하고 이를 수신한다. 수신된 코드 및 데이타에 문제가 없을 경우 RegisterData()를 수행하여 수신된 코드를 시스템에 등록하고 에이전트의 수행 상태를 시스템에 기록한다.

필요한 코드가 등록되면 ReadData()를 통해 에이전트 수행에 필요한 코드와 데이타를 읽고 에이전트가 수행할 작업을 판별한다. 에이전트가 수행할 작업이 판별되면 시스템 및 에이전트의 결함에 대처할 수 있도록 CheckpointAgent()를 수행하여 수신된 에이전트의 상태를 안전한 장치에 기록하고 완료 메시지를 에이전트를 전송한 시스템에 전송한다. 목적지 시스템에서 에이전트의 상태를 체크포인트를 통해 저장할 경우 에이전트가를 수신한 목적지 시스템에서 시스템이 다운되거나 시스템 오류가 발생하여 수신된 에이전트의 코드 및 데이타가 손실되는 문제를 해결할 수 있다. 에이전트 수행에 필요한 작업이 완료되면 이주 서비스는 CreateInstance()를 통해 기존에 프리패칭 메시지를 통해 수신된 코드와 이주 서비스를 통해 전송된 코드를 이용하여 에이전트의 작업에 필요한 인스턴스를 생성한다. ActiveAgent()는 에이전트에 쓰레드를 할당하고 에이전트의 수행 과정을 재시작한다. 만약 기존 프리패칭에 메시지에 의해 전송된 코드에 대한 인스턴스가 미리 생성되어 있는 경우에는 ActiveAgent()는 에이전트의 재시작 과정만을 수행한다. 이러한 과정을 통해 에이전트의 수행을 빠르게 처리할 수 있다.

4.5 동적 경로 생성

이주 서비스는 디렉토리 서비스, 에이전트 관리 서비스 그리고 자원 관리 서비스를 이용하여 각 시스템에서 수행 가능한 서비스 검사, 통신 비용, 시스템 상태, 자원 할당 비용 등을 검사하여 이주 경로를 생성한다. 이러한 이주 경로를 에이전트의 생성과 함께 생성되는 것으로 본 논문에서는 정적 경로(static itinerary)라 한다. 정적 경로는 이동 에이전트 생성과 함께 한번만 생성되기 때문에 에이전트 수행 중 시스템 또는 네트워크의 변화에 적응할 수 없다. 즉, 네트워크의 부하 및 지연과 같은 네트워크의 환경 변화 또는 이주할 시스템의 할당 가능한 쓰레드 및 메모리와 같은 시스템의 상태 변화에 적절하게 대응하지 못한다. 최악의 경우 시스템의 결함에 의해 시스템이 중지되어 있는 경우에는 더욱 심각하다. 따라서, 이주할 시스템과 네트워크의 상태 변화를 검사하여 에이전트 수행 가능 여부를 확인하고 에이전트 실행환경에 적합한 이동 경로의 생성이 필요하다. 이러한 이주 경로를 동적 경로(dynamic itinerary)라 한다.

그림 9는 동적 경로를 생성하기 위한 CheckItinerary()의 수행 과정을 나타낸 것이다. 동적 경로 생성을 위해서는 먼저 목적지 시스템에서 수행해야 할 작업이 무엇인지를 판별하기 위해 CheckJob()를 수행한다. 기존에 생성된 정적 경로에는 작업 순서에 따라 해당 작업이 수행 가능한 시스템을 이주 비용에 따라 내림차순으로 정렬하고 각 작업이 수행 가능한 시스템 중에서 가장 최소의 비용으로 이주할 시스템에 프리패칭 메시지를 전달한 상태를 관리하고 있다. 이를 이용하여 ReadItinerary()는 다음에 수행해야할 작업이 수행 가능한 이주 경로의 목록을 읽어온다. 정적 경로를 생성에 의해 생성된 목적지 시스템은 동적인 시스템의 상황에 따라 이주를 수행할 수 없거나 이주에 많은 비용을 소요한다. 특히, 네트워크를 통해 다음 목적지 시스템에 데이타를 전송할 수 없거나 시스템의 다운 또는 에이전트 수행을 위해 할당 가능한 자원이 없을 경우에는 큰

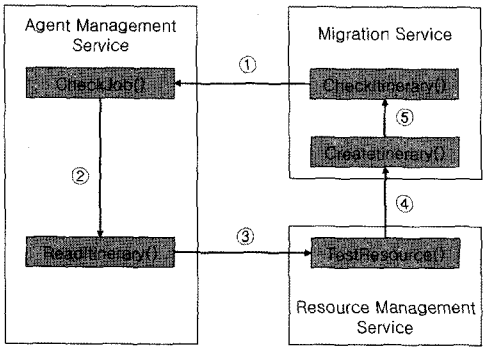


그림 9 이주 경로 생성

문제를 발생시킨다. 따라서 기존에 이동할 목적지 시스템의 상태를 확인하고 에이전트의 이주에 따라 문제가 발생할 경우 동적으로 이주 경로를 생성해야 한다. TestResource()에서는 수행할 작업이 수행 가능한 시스템에 대한 네트워크 및 시스템의 상태를 확인한다. 네트워크 및 시스템의 상태를 확인하면 CreateItinerary()에서는 다음 작업을 수행할 시스템의 상태에 따라 최소의 비용으로 이주할 다음 목적지 시스템의 경로를 생성한다. 새로운 이주 경로를 생성하기 위해서는 기존에 정적 경로에 의해 이주할 목적지에는 프리패칭 메시지에 의해 필요한 코드를 전송할 필요가 있는지를 확인하고 시스템의 상태 및 네트워크의 상태에 따라 이주 비용을 다시 계산한다.

4.6 결함허용(Fault-Tolerant) 에이전트

에이전트의 수행 중에 발생할 수 있는 결함은 에이전트 또는 시스템 자체의 결함에 의해 발생할 수도 있고 네트워크 상의 문제에 의해 발생할 수도 있다. 에이전트 또는 시스템에 결함이 발생할 경우 에이전트의 수행 작업이 중지되거나 에이전트의 수행 작업이 손실될 수 있다. 특히, 이동 에이전트는 네트워크를 통해 여러 시스템으로 이주하면서 실행하기 때문에 결함이 발생할 경우 수행 중인 에이전트를 손실할 가능성이 높다. 따라서 에이전트의 이주 전과 이주 후 에이전트의 코드 및 데이타를 안전한 저장 장치에 기록하는 체크포인트 과정을 수행한다.

그림 10은 에이전트의 이주 과정에서 체크포인트를 수행하는 과정을 나타낸 것이다. 시스템 A에서 수행 중인 에이전트 B가 시스템 B로 이주한다고 할 때 SendMigration()에서는 CheckpointAgent()를 이용하여 에이전트의 수행 상태를 안전한 저장 장치에 기록한다. 에이전트의 수행 상태를 안전한 저장 장치에 기록하면 에이전트의 이주가 완료될 때까지 에이전트의 수행 상태 및 코드를 기록하고 있다. 이주할 목적지 시스템에서는 에이전트의 수행에 필요한 코드 및 데이타가 수신되면 ReceiveMigration()은 전송된 코드 및 데이타를 수신하고 CheckpointAgent()를 이용하여 수신된 코드 및 수행 상태를 안전한 저장 장치가 기록한다. 목적지 시스템으로 이주가 완료되면 이주가 완료되었다는 메시지를 시스템 A에 전송할 때까지 체크포인트에 의해 저장된 상태는 유지한다.

5. 실험 및 성능평가

제안하는 이동 에이전트의 이주 기법의 우수성을 입증하기 위해 실험을 통해 기존에 제안된 Ajanta와 JAMES에서 제안된 이주 기법들과 성능 비교를 수행한다. 실험 환경은 인텔 펜티엄Ⅳ 1.8GHz CPU와 256MB의 메인 메모리를 갖는 시스템에서 윈도우 2000 서버 운영체제를 사용한다. 실험은 AweSim 3.0 버전을 이용하여 시뮬레이션을 수행한다. 성능 평가는 다음과 같은 네 가지 관점에서 수행한다. 첫째, 제안하는 이주 기법과 기존에 제안된 시스템의 이주 기법을 사용하여 에이전트 이주에 따른 처리 시간을 분석한다. 둘째, 제안하는 이주 기법에서 사용되는 프리패칭 기법, 동적 경로 기법 각각을 이용한 이주 기법과 프리패칭과 동적 경로 생성을 혼합한 이주 기법을 구분하여 처리 시간을 분석한다. 셋째, 이주할 에이전트 크기에 따라 제안하는 방

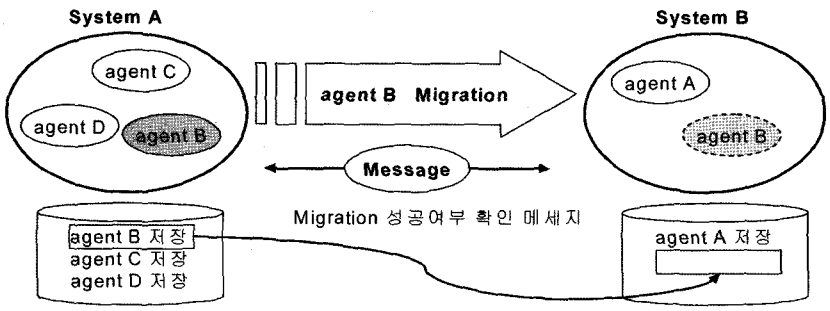


그림 10 체크포인트 기법

법과 기존 시스템의 이주기법을 비교한다. 마지막으로 이주 횟수에 따라 제안하는 방법과 기존 시스템의 이주 기법의 성능을 분석한다. 이동 에이전트의 수행 성능은 이동 데이타 크기, 네트워크 상태, 시스템 자원을 얼마나 고려했는가에 따라 처리시간이 다를 수 있다. 따라서, 성능 평가 부분에서 체크포인트를 수행하는 시간은 제외한다. 실험에서 사용한 파라미터는 표 4와 같다. 이주횟수에 따른 성능 평가를 제외한 실험은 도착율 및 네트워크 상태를 식 1의 지수분포에 따라 수행한다. 식 1에서 사용된 λ는 3 ~ 20으로 변화시켜 가면서 실험을 수행한다.

$$f(x) = \lambda e^{-\lambda x} \quad (1)$$

표 4 성능평가 파라미터

파라미터	값
이주 횟수	3~5
네트워크 상태	지수분포(λ1: 0.1 ~ 0.25)
코드 크기	4KB, 100KB, 1024KB
에이전트 도착율	지수분포(λ2: 3 ~ 20)

그림 11은 기존에 제안된 에이전트 시스템에서 사용된 이주 기법과 제안하는 이주 기법에 대한 실험 결과를 나타낸다. 제안하는 이주 기법은 전송되는 에이전트의 크기 및 동적 환경 변화에 따라 적절한 이주 경로를 생성하여 이동하기 때문에 기존에 제안된 Ajanta보다 약 90% 정도 성능이 향상되었으며 JAMES보다는 약 120% 성능이 향상되었다.

그림 12는 제안하는 이주 기법에서 사용되는 프리패칭 기법, 동적 경로 생성 기법에 대한 각각의 성능을 비교 평가한 결과이다. 제안하는 알고리즘에 프리패칭 기법과 동적 경로 생성 기법을 모두 고려할 경우 프리패칭만 사용한 경우보다 평균 60%, 동적 경로만을 사용한 경우보다 평균 40% 성능이 향상된다. 동적 경로가 프리패칭 기법보다 우수한 성능을 발휘하는 것은 실시간적으로 변화하는 시스템 환경 및 네크워크의 환경을 고려

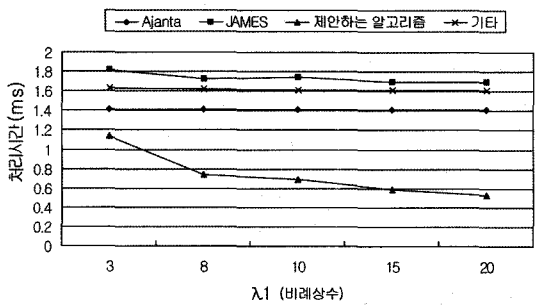


그림 11 에이전트 도착율에 따른 처리시간

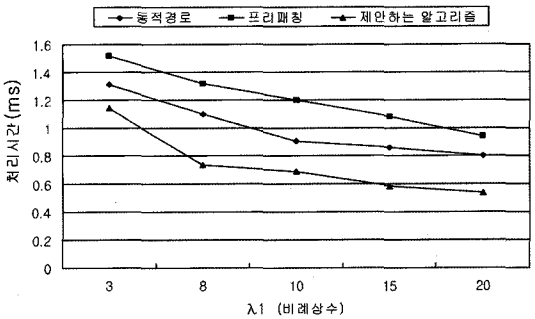


그림 12 각 기능별 처리 시간

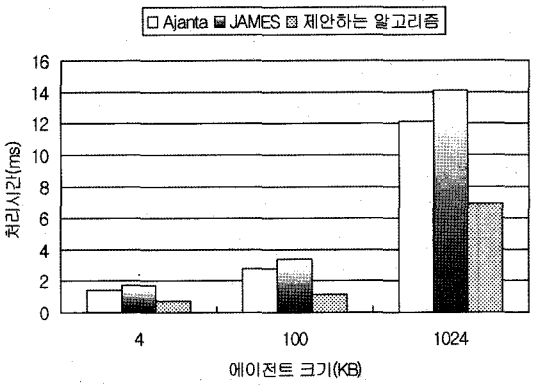


그림 13 에이전트 크기에 따른 처리시간

하기 때문에 이동 에이전트의 수행 성능을 많이 좌우한다.

그림 13은 이주할 에이전트의 크기에 따라 성능을 비교 분석한 결과이다. λ는 10, 이주 횟수는 3으로 하여 에이전트 크기를 4KB, 100KB, 1024KB 크기로 변화했을 때의 기존의 시스템 이주기법과 제안하는 방법의 성능 결과이다. 제안하는 방법이 Ajanta 보다는 평균 110% 향상되었고, JAMES보다는 평균 160%가 향상되었다. 이동 에이전트의 수행 성능은 이동 데이타 크기, 네트워크 상태, 시스템 자원을 얼마나 고려했는가에 따라 처리시간이 다를 수 있다.

그림 14는 에이전트의 이주 횟수에 따라 성능 평가 결과이다. λ는 10, 에이전트크기는 4KB, 이주 횟수를 3회, 4회, 5회로 하여 Ajanta와 JAMES에서 제안된 기법과의 비교 분석을 수행한다. 실험 결과 제안하는 방법은 Ajanta 보다는 평균 100% 향상되었고, JAMES 보다는 평균 140%가 향상되었다.

6. 결론 및 향후 연구

이동 에이전트의 이주 성능은 이동 데이타의 크기, 네트워크 및 시스템의 상태에 많은 영향을 받는다. 기존에 제안된 이주 기법들은 이동 에이전트의 성능인자들을

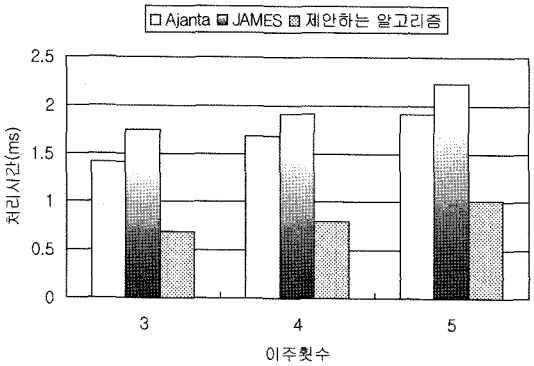


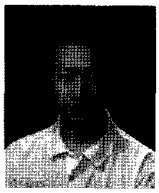
그림 14 이주 횟수에 따른 처리시간

모두 고려하지 못했다. 즉, 네트워크 트래픽 및 시스템의 자원 그리고 일관성 있는 에이전트 이주를 보장하지 못했다. 이러한 문제점을 해결하기 위해 본 논문에서는 동적 환경에 적합한 자율 이동 에이전트의 이주 기법을 제안하였다. 제안하는 이주 기법은 네트워크 및 시스템의 상태에 따라 동적으로 이동 경로를 생성하고 다음에 이동할 목적지에 필요한 에이전트의 코드를 전송하는 프리패칭 기법을 사용한다. 또한 이주 과정에서 발생한 결함에 대처할 수 있도록 체크포인트 기법을 이용하여 에이전트의 상태를 유지하고 복구할 수 있도록 하는 기능을 제공한다. 제안하는 이주 기법은 기존에 제안된 Ajanta에 비해 평균 100% 정도 성능이 향상되었고 JAMES에 비해서는 평균 140% 정도의 성능이 향상되었다. 또한, 이주할 시스템 수 또는 이동할 데이타의 크기가 증가할수록 성능이 더 우수하다는 것을 입증하였다. 향후 연구 방향으로 본 논문에서 제안한 이주기법을 기반으로 실제적인 에이전트 시스템을 개발할 예정이다.

참 고 문 헌

[1] M. Wooldridge, "Agent-based Software Engineering," IEEE Proceedings on Software Engineering, Vol.144, No.1, pp.26-37, 1997.
[2] M. Wooldridge and P. Ciancarini, "Agent-Oriented Software Engineering : The State of the Art," The First International Workshop on Agent-Oriented Software Engineering, pp.1-28, 2000.
[3] 최중민, "에이전트의 개요 및 연구방향", 한국정보과학회지, 제15권, 제3호, pp.7-16, 1997.
[4] 김수중, 윤용익, "모바일 에이전트 시스템 기술 동향", 정보처리학회지, 제8권, 제5호, pp.111-118, 2001.
[5] G. P. Picco, "Mobile Agents : An Introduction," Journal of Microprocessors and Microsystems, Vol.25, No.2, pp.65-74, 2001.
[6] 조수현, 김영학, "결함 허용을 고려한 효율적인 이동 에이전트 전송방법", 한국정보과학회 추계학술대회 논문집(III), 제28권, 제2호, pp.550-552, 2001.
[7] FIPA, "www.fipa.org"
[8] MASIF, "http://www.fokus.gmd.de/research/cc/ecco/masif"
[9] L. M. Silva and P. Simoes, "JAMES : A platform of Mobile Agents for the Management of Telecommunication Networks," The International Workshop on Intelligent Agents for Telecommunication Applications, pp.76-95, 1999.
[10] A. R. Tripathi, N. M. Karnik, T. Ahmed, R. D. Singh, A. Prakash, V. Kakani, M. K. Vora and M. Pathak, "Design of the Ajanta System for Mobile Agent Programming," Journal of Systems and Software, Vol.62, No.2, pp.123-140, 2002.
[11] Aglets, "http://www.trl.ibm.com/aglets"
[12] Grasshopper, "http://www.grasshopper.de/"
[13] A. Fuggetta and G. Vigna, "Understanding Code Mobility," IEEE Transactions on Software Engineering, Vol.24, No.5, pp.342-361, 1998.
[14] T. Ilmann, T. Krueger, F. Kargl and M. Weber, "Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture," Proc. 5th International Conference on Mobile Agents, pp.198-212, 2001.
[15] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java," Proc. International Symposium on Agent Systems and Applications/Mobile Agents, pp.29-43, 2000.
[16] D. B. Lange, M. Oshima, G. Karjoth and K. Kosaka, "Aglets : Programming Mobile Agents in Java," Proc. International Conference on Worldwide Computing and Its Applications, pp.253-266, 1997.
[17] S. Bouchenak, D. Hagimont and N. D. Palma, "Techniques for Implementing Efficient Java Thread Serialization," Proc. ACS/IEEE International Conference on Computer Systems and Applications, 2003.

복 경 수
1998년 2월 충북대학교 수학과(이학사) 2000년 2월 충북대학교 정보통신공학과(공학석사). 2005년 2월 충북대학교 정보통신공학과(공학박사). 2005년 3월~현재 한국과학기술원 전산학과 Postdoc. 관심분야는 시스템 소프트웨어, 이동 객체 데이타베이스, 자료 저장 시스템, 멀티미디어 데이타베이스, 센서 네트워크 및 RFID 등



여 명 호

2004년 2월 충북대학교 정보통신공학과 (공학사). 2004년 3월~현재 충북대학교 정보통신공학과 석사과정. 관심분야는 이동 객체 데이타베이스, 모바일 에이전트, 메인메모리 데이타베이스 시스템 등

유 재 수

1989년 2월 전북대학교 컴퓨터공학과 공학사. 1991년 2월 한국과학기술원 전산학과 공학석사. 1995년 2월 한국과학기술원 전산학과 공학박사. 1995년 2월~1996년 8월 목포대학교 전산통계학과 전임강사. 1996년 8월~현재 충북대학교 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소 부교수. 관심분야는 데이타베이스 시스템, 정보검색, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅 등