이동 에이전트 기반의 동적 작업 부하 균형 프레임워크

(A Dynamic Load Balancing Framework based on Mobile Agent)

김 지 균 † 김 태 윤 ††
(Ji Kyun Kim) (Tai-Yun Kim)

요 약 네트워크의 보편화와 개인용 컴퓨터의 고급화를 통한 가용 자원의 확장은 분산 컴퓨팅 환경에서 작업 부하 균형의 성능 향상을 기대할 수 있게 되었다. 하지만 이를 현실화시키기 위해서 다음과 같은 제약 사항을 극복해야 한다. 첫째, 네트워크로 연결된 각각의 시스템은 이질적인 하드웨어와 운영체제로 구성되어 있다. 둘째, 네트웍 대역폭의 격심한 변화가 존재하며 상이한 시스템 성능 차이가 존재한다. 셋째, 어플리케이션의 요구 조건이 상이하다.

본 논문에서는 작업 부하 균형에 이동 에이전트 패러다임을 적용하며 위의 문제점을 해결하기 위하여 세 가지의 사항을 추가한다. 1) 이질적인 분산 컴퓨팅 환경에 어플리케이션을 동적으로 이식하기 위하여 분산 객체 지향 미들웨어인 CORBA[1] 기반 MASIF[2]를 이용한다. 2) 유휴 자원 정보에 기반한 어플리케이션의 동적 배치를 위하여 자원 감지 모니터링을 실행한다. 3) 다양한 어플리케이션의 요구 조건을 만족시키기 위하여 다양한 모니터링 알고리즘을 동적으로 로드하는 자바 객체, MonitorHandler를 제안한다.

제안한 프레임워크의 실효성을 검증하기 위하여 프로토타입 어플리케이션을 구현하였다. 실험 결과 유휴 자원을 고려한 동적 배치가 정적 배치나 초기 정보에 의한 단 한번의 배치보다 각각 57%와 26%의 성능 향상을 보였다. 제안하는 프레임워크는 작업 부하 균형 어플리케이션의 개발을 용이하게 하며 범용성과 확장성을 제공한다.

Abstract The expansion of available resources based on the proliferation of network and the enhancement of personal computer has opened the door to new possibility of load balancing in distributed parallel computing. However, to realize it, the following constraints should be overcome. First, computing nodes consist of heterogeneous hardwares and operating systems. Second, network jitter exists and there is a constant change of system availability. Finally, the requirements of applications vary widely.

In this paper, we propose a dynamic load balancing framework based on mobile agent paradigm. The framework includes three characteristics. 1) It is based on CORBA[1] and MASIF[2], which are object oriented middleware to support the full complexity of heterogeneous distributed environment. 2) Resource detection monitoring is performed to arrange applications dynamically based on idle resource information. 3) We also introduce the concept of the MonitorHandlers, which are Java objects capable of implementing a particular monitor algorithm of target application.

We have developed and tested a prototype application as á proof of our proposed framework. Results show that dynamic placement with resource monitoring is better than static placement and one-movement placement, and has performance gain of 57% and 26%. The proposed framework provides easiness of load balancing programming, generality, and scalability. One of the main aims of proposed framework is to keep the application less complicated and still provide the basic load balancing functionality.

* 본 연구는 2000년도 한국전자통신연구원의 '인터넷 정보 가전 기술 개발'의 과제로 수행되었음.

† 비 회 원 : 고려대학교 컴퓨터학과
kjk@netlab.korea.ac.kr

†† 종신회원 : 고려대학교 컴퓨터학과 교수
: tykim@netlab.korea.ac.kr

논문접수 : 2000년 12월 23일
심사완료 2001년 4월 25일

1. 서 론

부하 균형(load balancing)이란 분산 시스템에서 과부하 상태인 프로세서로부터 부하가 적은 프로세서에게로 작업을 이동시킴으로써 평균 응답 시간의 최소화 또는 전체 시스템 처리량을 최대화하는 하나의 태스크 스케줄링 방법이다.

이질적인 분산 컴퓨팅 환경에서 작업 부하 균형을 위한 다양한 프로그래밍 모델[3,4]이 연구되었음에도 불구하고 모든 상황에 적용될 수 있는 패러다임을 정의하지는 못하였다. 각각의 모델들은 한정된 어플리케이션과 환경에 대한 해법만을 제시하였으며, 이질적인 분산 시스템의 복잡성을 해결하지는 못하였다. 즉, 각각의 모델들은 한정된 어플리케이션과 환경만을 고려한 채 다음과 같은 제약 사항을 해결하지는 못하였다. 첫째, 네트워크로 연결된 각각의 시스템은 이질적인 하드웨어와 운영체제로 구성되어 있다. 따라서, 프레임워크는 이질적인 실행 환경에서 어플리케이션을 동적으로 이식하여 실행할 수 있는 구조를 제공하여야 한다. 둘째, 시스템 가용성과 네트워 대역폭의 격심한 변화가 존재하며 상이한 시스템 성능 차이가 존재한다. 따라서, 프레임워크는 가용 자원의 지속적인 변화와 상태를 감지하여 어플리케이션의 이동에 반영하여야 한다. 셋째, 어플리케이션의 요구 조건은 상이하다. 따라서, 어플리케이션의 요구 조건에 따라 자원 감지 모니터링 알고리즘을 동적으로 실행할 수 있는 구조를 제공하여야 한다.

본 논문에서는 이동 에이전트 패러다임을 작업 부하 균형에 적용하여 그 적용의 실효성을 프로토타입 어플리케이션을 통하여 검증한다. 그림 1과 같이 이동 에이전트(mobile agent)는 클라이언트의 작업을 위임받아 서버 측으로 이동하여 로컬로 실행된다[5]. 따라서 최소한 2배수의 메시지 전송을 요구하는 원격 함수 호출(RPC; Remote Procedure Call)[14]에 비하여 네트워크로 전송되어지는 메시지의 양을 대폭 감소시킬 수 있다.

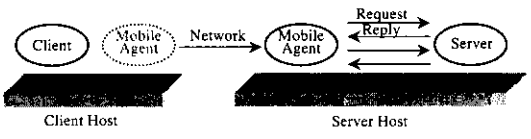

그림 1 이동 에이전트 패러다임

이러한 이동 에이전트 패러다임을 작업 부하 균형에 적용하게 된 배경은 다음과 같다. 첫째, 이동 에이전트는 이질적인 하드웨어와 운영체제로 구성된 분산 컴퓨팅 환경에 동적으로 이식이 가능하다(*portability*). 동적 이식이 가능한 이유는 이동 에이전트가 시스템의 리소스를 직접적으로 이용하지 않고 에이전트 고유의 실행 환경을 통하여 간접적으로 이용하기 때문이다. 그러므로 시스템에 비종속적이기 때문에 이질적인 실행 환경에 동적으로 이식이 가능하다. 둘째, 이동 에이전트는 특정 작업을 수행하기 위해 자신을 복제하여 작업 부하를 분산시킬 수 있다(*cloning property*). 복제는 데이타간의 연관성이 없으며 동일한 모듈을 병렬적으로 사용하는 작업에 적용될 수 있다. 셋째, 이동 에이전트가 과부하 상태의 노드에서 실행되고 있다면 저부하 상태의 노드로 이동하여 실행할 수 있다(*code mobility*)[15].

제안하는 프레임워크는 위에서 제시한 문제점을 해결하기 위하여 세 가지의 사항을 이동 에이전트 패러다임에 추가한다. 첫째, 이질적인 분산 컴퓨팅 환경에 어플리케이션을 동적으로 이식하기 위하여 분산 객체 지향 미들웨어인 CORBA(Common Object Request Broker Architecture) 기반 MASIF(Mobile Agent System Interoperability Facility)를 이용한다. 둘째, 유휴 자원 정보에 기반한 어플리케이션의 동적 배치를 위하여 자원 감지 모니터링을 실행한다. 자원 감지 모니터링은 주문형(on-demand) 모니터링을 설계 원칙으로 한다. 주문형 모니터링이란 어플리케이션이 과부하 노드에서 실행되고 있는 경우에만 이동을 위하여 자원 감지 모니터링을 수행하는 에이전트에게 저부하 노드의 추천을 의뢰하는 것을 의미한다. 따라서, 주문형 모니터링은 적은 메시지 교환으로 시스템 전체의 통신 링크 오버헤드를 줄인다. 셋째, 다양한 어플리케이션의 요구 조건을 만족시키기 위하여 특정한 모니터링 알고리즘을 런타임에 동적으로 실행할 수 있는 MonitorHandler의 개념을 제안한다.

제안한 프레임워크는 프로토타입 어플리케이션을 구현하여 테스트한 결과 작업 부하 균형에 이동 에이전트 패러다임의 적용 가능성을 증명하였으며, 우수한 성능 향상을 나타내었다. 또한, 제안한 프레임워크는 프로그램 개발자에게 부하 균형 어플리케이션의 개발을 용이하게 하며 일반적이며, 다양한 플랫폼에서 적용 가능한 확장성을 제공한다.

본 논문의 구성은 다음과 같다. 2 장에서 기존 에이전트 시스템의 부하 균형 방법과 문제점을 기술하고, CORBA 기반 MASIF에 대하여 기술한다. 3 장에서는 제안한 프레임워크의 전체적인 구조와 구성 요소, 시나리오를 제시한다. 또한 어플리케이션의 이동 방법과 향

상된 자원 감지 모니터링, MonitorHandler를 자세하게 설명한다. 4 장에서는 프로토타입 어플리케이션의 환경과 구현 방법을 기술하고 결과를 분석하여 제안한 프레임워크의 우수성과 실효성을 검증한다. 마지막으로 5 장에서는 결론과 향후 과제를 제시한다.

2. 관련 연구

본 장에서는 기존 에이전트 시스템의 부하 균형 방법과 문제점을 기술한다. 또한 이질적인 분산 컴퓨팅 환경에서 어플리케이션을 동적으로 이식하기 위하여 CORBA 기반 MASIF를 기술한다.

2.1 부하 균형(Load Balancing)

이동 에이전트가 이용하는 부하 균형 알고리즘은 크게 두 가지 -정적, 동적 방법[3]-으로 분류할 수 있다. 첫째, 정적(static) 부하 균형 알고리즘은 작업이 실행되기 이전에 해당 작업의 이동 정보를 이용하여 임의의 프로세서에 작업을 할당한다. 즉, 시스템의 현재 부하 상태를 무시한 채 컴파일 타임에 작업이 실행될 노드를 결정한다. IBM의 Aglet[6]은 정적 부하 균형을 이용하여 이동 노드의 부하 상태를 무시한 채 이동 에이전트를 이동시킨다. 둘째, 동적(dynamic) 부하 균형 알고리즘은 실행 중에 과부하 프로세서가 초과된 태스크를 저부하 프로세서로 보낸다.

즉, 시스템의 현재 부하 정도를 고려하여 런타임에 동적으로 작업을 할당하므로 변화하는 시스템 상태에 반응하여 이동한다. EPFL의 XAgent[7]는 동적 부하 균형을 이용하여 이동 에이전트를 이동시키며, 부하 상태를 수집하기 위하여 브로드캐스팅이나 폴링(polling), 선별적 폴링(selective polling) 방법을 사용한다. 그림 2와 같이 XAgent는 Manager와 worker로 구성된다. Manager는 중앙 서버 노드로서 프로세스 스케줄링을 담당한다. worker는 백그라운드 데몬으로 동작하는 부하 계산 프로세스(load calculate process)를 생성시켜 주기적(periodic)으로 자신의 부하 정도를 계산하여 Manager에게 전송한다. 만약, 프로세스가 과부하인 경우 Manager는 분산된 worker들에게 부하 정보를 요청한다(sender initiated policy). Manager의 요청에 대하여 worker는 부하 계산 프로세스에 의하여 계산된 값을 리턴한다. Manager는 수집된 부하 정보를 인덱싱하여 부하가 적은 노드로 프로세스를 이동시킨다.

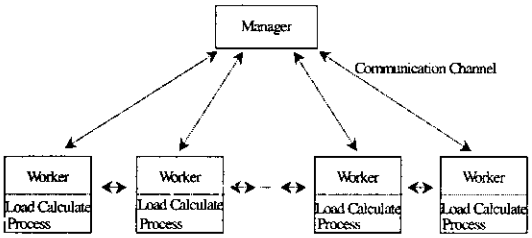

그림 2 XAgent의 동적 부하 균형

그러나, XAgent는 다음과 같은 문제점이 있다. 첫째, XAgent는 각각의 노드들이 다른 노드들의 부하 상태 정보를 수집하는데 있어서 많은 통신 오버헤드가 초래된다. 둘째, 각각의 노드는 자신의 부하 정도를 파악하기 위해 독립적으로 부하 측정 프로세스를 실행시켜 부하 측정치를 계산하여야만 한다. 셋째, 계산된 부하 측정치는 Manager에게 집중되므로 Manager가 다운되는 경우 전체 노드에 영향을 미치게 된다. 또한 중앙 집중식(centralized)의 구조로 인하여 worker의 수가 증가하는 경우 Manager의 인덱싱 시간이 증가하여 확장성을 제공하지 못한다. 넷째, worker는 오직 CPU의 부하량만을 고려하며 정적인 부하 계산 프로세스를 사용한다. 따라서, 어플리케이션의 요구 조건이 달라지는 경우 부하 계산 프로세스의 교체가 불가능하다.

2.2 CORBA 기반 이동 에이전트 시스템: MASIF

OMG(Object Management Group)에서는 이동 에이전트 플랫폼간의 상호 운용성을 지원하기 위해 CORBA 기반 MASIF를 추가하였다. MASIF는 특정 플랫폼에 의존하지 않는 중립적인 구조를 지향하고 있으며, 시스템 차원에서 에이전트 플랫폼간의 상호 운용성을 제공하고자 하는 목표를 가지고 있다[2]. 그림 3에서와 같이 MASIF는 기존 에이전트 플랫폼에서 사용되는 장소(place)와 지역(region)의 개념을 도입했다. place는 이동 에이전트의 실행 환경을 제공하며, 출발점과 도착점 기능을 한다. region은 하나의 권한(authority)에 속하는 에이전트들의 집합을 기술해 놓음으로써 플랫폼 관리를 편리하게 한다. CORBA의 ORB (Object Request Broker)는 분산 환경에서의 통신을 담당하는 제반 구조로서 분산 어플리케이션의 개발을 단순화시킨다. 또한, MASIF는 이동 에이전트 시스템간의 상호연동을 위해 MAFAgentSystem 인터페이스와 MAFFinder 인터페이스를 정의하고 있다. 전자는 에이전트 생성(create), 수신(receive), 중지(suspend), 종료(terminate) 등과 같은 에이전트 관리를 위한 기본적인 인터페이스를 정의하며 후자는 에이전트의 등록(register), 등록 해제(unresister), 위치 확인(lookup) 등과 같은 네이밍 서비스(Naming Service)를 위한 인터페이스를 제공한다.

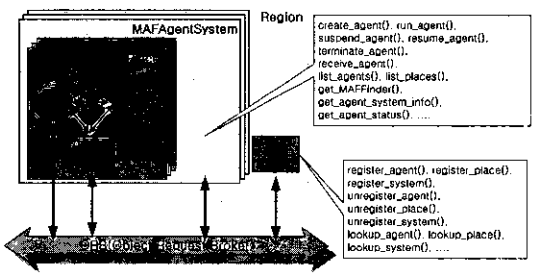

그림 3 MASIF 구조

3. 제안하는 작업 부하 균형을 위한 이동 에이전트 기반 프레임워크

제안한 프레임워크는 이질적인 분산 컴퓨팅 환경에서 어플리케이션을 동적으로 이식하기 위하여 MASIF를 기반으로 설계하였다. 또한, 이동 에이전트를 이용한 자원 감지 모니터링을 수행한다. 선별적인 폴링 방법을 사용하는 XAgent와는 달리 통신 오버헤드와 부하 계산 프로세스의 오버헤드를 줄이기 위하여 이동 모니터 에이전트를 추가하였다. 자바 객체, MonitorHandler는 다양한 어플리케이션의 요구 조건을 수용하여 모니터링 알고리즘을 동적으로 실행할 수 있는 구조를 제공한다.

3.1 전체적인 프레임워크 구조

그림 4는 제안한 작업 부하 균형을 위한 이동 에이전트 프레임워크의 전체 구성도이다. 프레임워크의 주요 구성 요소와 시나리오는 다음과 같다.

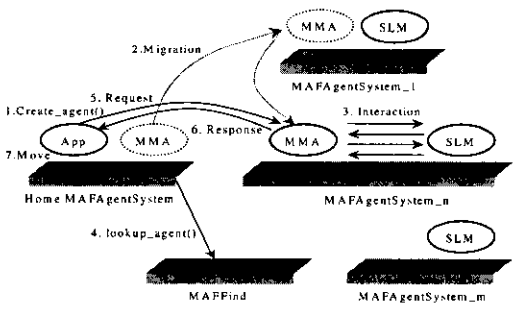

그림 4 제안된 프레임워크의 전체 구성도

3.1.1 프레임워크 구성요소

이동 에이전트 기반의 동적 작업 부하 균형 프레임워크에서 이동 에이전트의 성격을 이용하는 부분은 크게 두 부분으로 나뉜다. 하나는 사용자 어플리케이션에 이동성을 제공하는 부분이고 다른 하나는 분산 컴퓨팅 환경내의 노드들을 순회하면서 노드들 각각의 자원을 감지하여 모니터링하는 부분이다.

- Application : 사용자 어플리케이션으로서 성능 향상을 위하여 이동이 가능하다. 이동 에이전트 플랫폼 위에서 하나의 이동 객체로서 동작한다.
- MAFAgentSystem : 사용자 어플리케이션은 MAF AgentSystem의 제어 아래에서 생성된다. 성능 향상을 위한 이동(move)이나 복사(clone)를 MAFAgent System에게 통보 받아서 자신의 내부 상태를 정리하고 이동을 준비한다. 이동을 위하여 저장해야 하는 상태 정보는 어플리케이션의 논리에 따라서 결정되므로, MAFAgentSystem은 어플리케이션에게 이동/복사할 것이라는 것만을 통보하고 실제로 내부 실행 상태 저장은 어플리케이션에 위임하다.
- MAFFinder : 과부하 상태의 어플리케이션은 CORBA 네이밍 서비스나 MAFAgentSystem. get_MAF Finder() 메소드를 사용하여 MAFFind에게서 이동 모니터 에이전트의 객체 참조를 구하여 통신을 실행한다.
- 이동 모니터 에이전트(Mobile Monitor Agent; MMA) : 네트워크로 연결된 노드들을 방문 리스트에 따라 순회하면서 모니터링 정보를 수집하고 그것을 바탕으로 추천 노드 선정 알고리즘을 적용하여 성능 향상을 위한 이동/복사의 대상이 되는 노드들을 추천하는 기능을 수행한다. 확장성을 제공하기 위하여 MMA는 복사될 수 있다.
- 정적 로컬 모니터 에이전트(Stationary Local Monitor Agent; SLM) : 정적인 에이전트로 로컬 노드에서 자원 사용 현황만을 조사한다. 각각의 노드에서 독립적으로 자원 사용 현황을 조사하여 부하 측정치를 계산하는 XAgent와는 달리 SLM과 MMA의 부하 측정을 위한 상호 작용을 통하여 부하 계산 프로세스를 분산시켰다. 즉, SLM과 MMA의 상호 작용이란 MMA가 SLM과 통신하여 해당 노드의 부하 정보를 얻은 후, 최선의 이동 노드를 갱신하는 것을 의미한다. 이에 대한 자세한 내용은 3.3 절에서 기술한다.

3.1.2 프레임워크 시나리오

기본적인 이동 형태는 다음과 같은 순서로 진행된다.

1. Create_Agent() : 어플리케이션과 SLM, MMA를 생성한다. 이동 에이전트들은 MAFAgent System을 통하여 생성되고, 자신의 ID와 IOR을 MAFFind에 등록시킨다.
2. Migrate_Agent() : 생성된 SLM은 MMA의 순회 리

스트의 모든 노드로 이동하여 작업 부하량을 모니터링한다. MMA는 순회 리스트를 따라 이동하며 다른 노드로 이동하는 경우 MAFFind에게 자신의 이동 정보를 갱신한다. Home MAFAgent System의 어플리케이션은 작업을 시작한다.

3. Interact_Agent() : MMA는 노드들을 방문한 후 SLM과 접속하여 모니터링 결과를 얻는다. 얻어진 정보를 바탕으로 내부에 구현된 알고리즘을 적용하여 성능 향상을 위한 이동이나 복사의 대상이 되는 추천 노드를 갱신한다.
4. Lookup_Agent() : Home MAFAgentSystem에서 실행하고 있는 어플리케이션이 사용자의 요구 조건을 만족시키지 못하는 경우 즉, 이동 사건(migration event)이 발생한 경우 MAFFind에게 MMA의 위치 정보를 요청한다.
5. Request_Agent() : MAFAgentSystem 혹은 어플리케이션은 Lookup_Agent()를 통하여 얻어진 MMA의 위치로 추천 노드를 요구한다.
6. Respond_Agent() : 요구한 추천 노드를 Home MAFAgentSystem 혹은 어플리케이션에게 전달한다.
7. Move_Application() : 어플리케이션은 추천 노드의 MAFAgentSystem으로 이동하여 재실행한다.

3.2 사용자 어플리케이션의 이동

MAFAgentSystem을 이용하여 어플리케이션의 이동을 제어하는 과정은 그림 5와 같다. MAFAgentSystem은 initialize 함수를 호출하여 사용자 어플리케이션을 생성시킨다. 이동이나 복사와 같은 이벤트 처리를 위해 CloneListener, MobilityListener, PersistenceListener 등의 객체를 addListener 함수를 통하여 추가한 후 startWork 함수를 실행시킨다. 만약 실행 중에 어플리케이션이 이동을 결정하면, 사용자 어플리케이션의 stopWork 함수를 호출하여 실행을 멈추고 내부 상태를 저장하도록 한다. 또한 내부 상태를 저장하기 위하여 어플리케이션의 코드와 데이타를 포함하는 메시지를 생성한다. 어플리케이션 데이타란 어플리케이션이 관리하는 객체들을 의미한다. 이러한 객체들을 네트워크상으로 전송하기 위하여 자바의 객체 직렬화(object serialization) 메커니즘을 이용하여 바이트 스트림 형태로 변환한 후 메시지에 저장한다. 객체 직렬화를 위해 클래스의 인스턴스는 *Serializable* 인터페이스를 구현하여야 하며, *writeObject*와 *readObject*의 두가지 메소드를 사용하여 직렬화와 역직렬화한다. stopWork 함수가 예외 상황이 없이 리턴되면 MMA에 의하여 추천된 사이트로 위에서 생성된 메시지를 전송하고 원래의 어플리케이션은 소멸시킨다. 추천 노드의 MAFAgentSystem은 전송된 메시지를 수신한다. 이 과정에서 어플리케이션의 클래스와 관련된 클래스 로더(ClassLoader)를 생성하고 어플리케이션 코드를 초기화한다. 어플리케이션 데이타는 역직렬화(object de-serialization) 메커니즘을 이용하여 객체를 재구성한 후 resumeWork 함수를 호출하여 어플리케이션을 재실행한다.

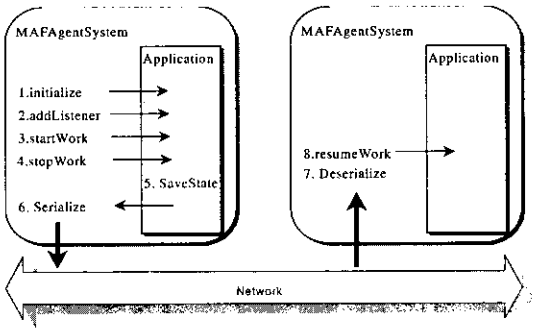

그림 5 어플리케이션의 이동

어플리케이션이 태스크를 분할하여 네트워크상의 노드로 분산되어 병렬적으로 실행된다면 태스크를 수행하는 프로세스의 동기적인 이동(synchronous migration)을 지원해야 한다. 이것은 분할된 태스크가 메시지 교환을 통하여 상호 작용을 수행하는 경우, 이동 프로세스에게 전송되는 메시지 손실은 잘못된 결과를 초래할 수 있기 때문이다. 또한, 프로세스의 동기적 이동은 다수의 과부하 노드에서 프로세스의 비동기적 이동(asynchronous migration)으로 인한 무익한 이동(useless migration)의 가능성을 제거한다. 프로세스의 동기적인 이동은 그림 6과 같이 4 단계로 구성된다.

1. 프로세스1(P_1)이 과부하 노드(host1)에서 동작하는 경우 MAFAgentSystem 혹은 P_1은 이동 사건(migration event)을 발생시킨다.
2. 과부하 노드의 MAFAgentSystem은 작업 부하 균형을 위해 구성된 노드들(혹은 P_1과 통신을 하는 노드들만)의 MAFAgentSystem과 MMA에게 P_1의 이동을 통보하는 메시지를 전송한 후 ACK를 받을 동안 기다린다(waiting). P_1, P_2, ..., P_n 이 병렬적인 태스크를 수행하는 경우 ACK를 통하여 이동 프로세스는 자신에게 도착하는 모든 메시지의 수신을 보장받는다.
3. 각각의 노드로부터 ACK를 받은 후, P_1은 자신의 상태 정보를 저장한 후 MMA가 추천한 노드로 이동한

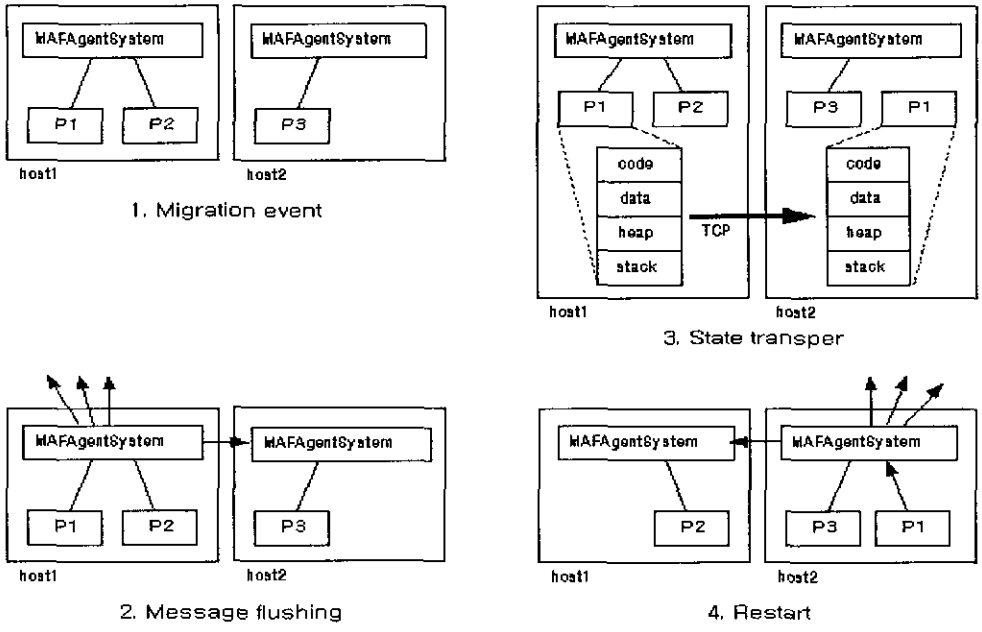

그림 6 동기적인 이동(*synchronous migration*)

다.

4. P_i은 추천 노드(host2)의 MAFAgentSystem에게 이동을 통보한 후 재실행한다. 추천 노드의 MAFAgent System은 작업 부하 균형을 위해 구성된 노드들(혹은 P_i과 통신을 하는 노드들만)의 MAFAgent System과 MMA에게 재실행을 통보하는 메시지를 전송한다.
5. MMA는 전송받은 메시지를 통하여 이동 전의 노드와 이동 후의 노드로 이동(jumping)한 후, 부하량을 측정하여 전체 노드의 정확한 상태 정보(global state)를 유지한다.

3.3 자원 감지 모니터링

로컬 노드의 자원 감지 모니터링은 노드 각각마다 존재하는 SLM에 의하여 실행된다. 백그라운드 데몬으로 동작하는 SLM은 다양한 어플리케이션의 요구 조건(CPU 중심적, 메모리 중심적, 네트워크 중심적 조건 등)을 만족시키기 위하여 네트워크 지연 시간, 네트워크 대역폭, 서버의 작업량(서버에 접속한 연결 수), CPU 사이클 등의 다양한 자원 감지 모니터링을 수행한다. 특히, 통신 링크 자원은 어플리케이션의 논리에 따라서 모니터링의 범위가 다양하므로 통신 자원 모니터링 모듈은 융통성과 확장성을 필요로 한다.

MMA은 주문형(On-Demand) 모니터링을 설계 원칙으로 한다. 주문형 모니터링이란 어플리케이션이 실행되고 있는 노드의 부하 상태가 과부하 상태로 전환되어질 때, 다른 노드의 상태 정보를 파악하기 위해 MMA에게 저부하 노드를 추천할 것을 요청하는 메시지를 전송하는 것을 의미한다. 따라서, Home MAFAgnetSystem 혹은 어플리케이션의 요구에 대하여만 메시지 교환을 실행하므로, 시스템 전체의 통신 링크 오버헤드를 막는다. MMA는 방문 리스트내의 노드들을 차례로 방문하면서 SLM이 수집한 작업 부하와 통신 링크 모니터링 정보를 얻은 후 식 (1), (2), (3)을 이용하여 결과 값(ts)이 작은 추천 노드를 선정한다.

$$t_s = t_{CPU} + t_{net} \tag{1}$$

$$t_{CPU} = CPU_{load} \frac{No.\ of\ operations\ required}{CPU_{speed}} \tag{2}$$

$$t_{net} = LT(A,B) + \frac{No.\ of\ bytes\ required}{BW(A,B)} \tag{3}$$

where

t_s : 로컬 노드의 추천 값(sec)

t_{CPU} : 작업을 수행하기 위하여 요구되는 프로세싱 타임(sec)

t_{net} : 어플리케이션을 이동시키는데 걸리는 전송 시간(sec)

CPU_{load} : CPU를 공유하고 있는 활성화된 job의 갯수
No. of operation required : 작업을 위해 요구되는 오퍼레이션의 수
CPU_{speed} : CPU 사이클(MIPS)
$LT(A,B)$: 노드 A, B 간의 통신 지연 시간
No. of bytes required : 전송에 필요한 바이트 수
$BW(A,B)$: 통신 노드 A, B간의 대역폭(bps)
A, B : 통신 노드

가용 자원 정보와 부하 상태(CPU_{load})는 시스템 데몬들(System Configuration)로부터 구하며, 프로세서의 MIPs(CPU_{load}), 네트워크 대역폭($BW(A,B)$)과 지연시간($LT(A,B)$)은 시스템 자원 정보를 담고 있는 데이타베이스(Resource Information DB)에서 얻는다[8]. 얻어진 정보를 바탕으로 식 (2), (3)을 이용하여 어플리케이션이 이동하는 경우 어플리케이션을 이동시키는데 걸리는 전송 시간(t_{net})과 이동한 후 작업을 수행하기 위하여 요구되는 프로세싱 타임(t_{CPU})를 계산한다. 계산된 값을 MMA에게 전송하고 MMA는 식 (1)의 값을 최소화하는 노드를 선정한다.

그림 7은 MMA가 SLM으로부터 얻은 로컬 노드의 정보를 사용하여 저부하 상태의 노드를 선정하는 알고리즘이다. MMA는 식 (2), (3)의 tCPU와 tnet를 각각 time_comp와 time_comm에 저장한 후 식 (1)을 사용하여 time_total을 계산한다. 구해진 값이 min 보다 적은 경우 추천 노드, m_best 의 값을 현재 노드로 업데이트한 후 다음 노드로 이동을 하여 위의 알고리즘을 반복적으로 수행한다.

```
Algorithm Recommend
begin
  initialize min = α /*a very large number*/
  m_best = none
  while the system is active
    for each machine m_i
    /* find the best suitable machine, m, among all
       nodes for the task t */
      calculate time_comm   /* from SLM (from
        System Configuration)*/
      calculate time_comp  /* from SLM (from
        Resource Info. DB)*/
      time_total = time_comp + time_comm
      if (time_total < min) then
        min = time_total
        m_best = m_i
      end if
      move m_i+1
    end for
  until not kill_message   /*while not owner's kill
        message*/
endwhile
```

그림 7 MMA의 추천 노드 선정 알고리즘

3.4 MonitorHandler

MonitorHandler 는 특정한 모니터링을 수행하는 자바 객체들의 라이브러리이다. 특정 모니터링 알고리즘을 구현한 MonitorHandler 객체는 객체 직렬화(object serialization)를 통하여 MMA와 함께 이동한다. 또한 어플리케이션은 MonitorHandler 객체를 동적으로 생성시키고, 동적으로 사용할 시기와 장소를 결정한다. 즉, MonitorHandler로부터 어떤 객체를 초기화 시켜야 하는지는 어플리케이션의 요구조건에 따라 런타임에 동적으로 결정된다. 이러한 구조는 CORBA의 메타 레벨 구조(Meta-level Architecture)에 기반하는데 본 논문에서는 JAVA Reflection API와 메타 객체인 익명 클래스(anonymous class)를 이용한다. 메타 객체의 기본 개념은 객체의 동작 또는 의미(semantics)를 하나로 제한하지 않으며, 하나 이상의 의미를 갖는 객체가 있을 때, 메타 객체가 어떤 순간에 객체가 가지는 의미를 결정하도록 하는 것이다. Reflection은 런타임에 로드된 객체의 필드값, 메소드, 생성자 정보를 얻을 수 있으며, 반영된 필드값과 메소드, 생성자를 사용하여 객체를 동작시킬 수 있는 메커니즘이다. 이것은 보다 확장된 Class 클래스와 java.lang.reflect 패키지의 클래스들로 구성된다. 우선 특정한 모니터링 알고리즘 객체, MonitorHandler를 익명 클래스를 이용하여 동적으로 로드하고 메모리에 런타임 객체의 형태로 생성한다. 생성된 런타임 객체를 대상으로 invoke함수를 이용하여 Reflection API를 실행시키면 런타임 객체내의 모니터링 알고리즘을 구현한 함수가 실행된다. 이러한 일련의 과정을 통하여 교체된 모니터링 알고리즘이 동적으로 로드되어 실행된다. 이때 교체된 알고리즘을 구현한 객체에 대한 정보를 통보하기 위하여 어플리케이션이나 MASIFAgent System으로부터 스트링 형태의 메세지로 클래스의 이름을 알려주거나 객체 직렬화를 이용하여 객체 자체를 전송한다. 또한 모니터링 작업을 쓰레드를 이용하여 이차적인 병렬작업의 수행이 가능하므로 빠른 모니터링 작업을 수행한다. 현재 MonitorHandler abstract class - 파생 클래스를 위한 공통적인 인터페이스만을 제공하는 추상 클래스 - 만을 포함하고 있지만, 다른 MonitorHandler 객체도 포함할 수 있다. 이 경우 구현된 객체는 Monitor Handler abstract class를 상속받아야만 한다. 그림 8은 MonitorHandler의 계층도를 도식화한 것이다.

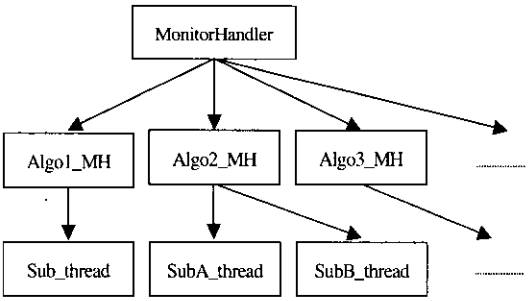

그림 8 MonitorHandler 라이브러리의 계층도

4. 프로토타입 어플리케이션의 구현

본 논문에서는 이동 에이전트 패러다임의 적용 가능성과 우수성을 검증하기 위하여 프로토타입 어플리케이션을 구현하였다. 어플리케이션은 Java[9]로 구현하였으며, 이동할 때 실행 상태 정보를 저장하기 위하여 IBM의 Aglet1.0.3을 이용하였다. 에이전트의 이동은 연약한 이동(weak migration)을 지원한다. 즉, 에이전트가 이동할 때 데이타의 상태만을 유지하며 보내는 정책으로 이동을 지원한다. 또한 Aglet에서 지원하는 코드 베이스(Code_Base)를 이용하여 필요한 클래스를 동적으로 보내 사용할 수 있다[6].

4.1 프로토타입 어플리케이션의 환경 및 구현

프로토타입 어플리케이션은 멀티미디어 데이타를 멀티캐스트 전송하는 어플리케이션으로 Music Station 서버, 클라이언트, Mobile RTP Proxy, SLM, MMA로 구성되며 부하 균형을 위한 어플리케이션은 Mobile RTP Proxy이다. Mobile RTP Proxy가 작업 부하가 많은 노드에서 실행되고 있을 때 작업 부하가 적은 노드로 복사/이동하여 다수의 클라이언트를 동시에 만족시키는 것을 목적으로 한다. 통신 링크는 10 Mbps LAN을 이용하며, 프로토타입 환경은 표 1과 같다.

표 1 프로토타입 환경

구분	Music Station	Mobile RTPproxy1	Mobile RTPproxy2	Mobile RTPproxy3	Client_1
기종	PentiumIII 450	PentiumIII 450	PentiumIII 450	PentiumII 400	Pentium200
운영체제	Win98	Win98	NT Server	Win98	Win98
메모리	128M	128M	64M	64M	72M
호스트명	ulsi	netlab	mullab20	mullab10	kjk
도메인명	korea.ac.kr	korea.ac.kr	korea.ac.kr	korea.ac.kr	korea.ac.kr
JDK	JDK1.2.2	JDK1.1.6	JDK1.1.6	JDK1.1.6	

MusicStation 서버는 음악 파일을 디스크로부터 읽어 Mobile RTP Proxy와 TCP 연결을 한 후, UDP 데이타에 RTP(Reliable Transmission Protocol) 헤더를 추가하여 스트림 형태로 전송을 한다. RTP는 화상회의 등의 응용 프로그램에서 멀티캐스트 데이타그램을 신뢰성있게 전송하기 위해 전송 순서(Sequence number)와 전송 시간(timestamp)을 실어 보낸다[10]. 또한 다중 스레드를 사용하여 다중 접속을 지원하였다. 그림 9는 RTP의 헤더 정보를 나타낸 것이다.

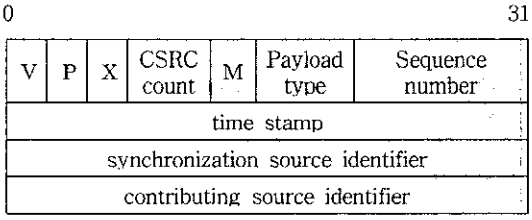

그림 9 RTP 헤더

Mobile RTP Proxy는 작업 부하 균형의 주체로서 이동 에이전트의 성격을 가지고 있다. 즉, Mobile RTP Proxy가 상주한 노드의 부하 정도가 증가하면 모든 클라이언트에게 만족할 만한 서비스를 제공할 수 없으므로 이동을 하게 된다. Mobile RTP Proxy는 Music Station으로부터 전송받은 멀티미디어 스트림을 멀티캐스트[11]하여 멀티캐스트 그룹에 가입한 클라이언트에게 전송한다. Mobile RTP Proxy는 MBONE[12]의 역할을 대행하여 수행하는 일종의 MBONE 라우터이다. n개의 서브네트로 구성된 네트워크의 경우 n-1개의 Mobile RTP proxy를 통하여 서비스를 제공받는다. Mobile RTP Proxy가 이동하는 경우 핸드오프로 인한 모든 클라이언트에서의 일시적인 서비스 중단을 막기 위하여 Mobile RTP Proxy가 이동하기 전, 자식 Mobile RTP Proxy를 복사하여 생성한 후 서비스를 위임한다. Mobile RTP Proxy는 MusicStation과의 접속을 일시 중단하고 이동하여 활동 재개를 준비한다. 자식 Mobile RTP Proxy에게 준비가 되었음을 통보하여 실행중인 자식 Mobile RTP Proxy를 소멸시킨 후 새로운 연결을 통하여 서비스를 실행한다.

클라이언트는 FreeAmp Player 2.0[13]을 사용하였으며, 클라이언트의 패킷 손실률을 조사하기 위하여 소스 코드를 수정하였다. 모든 클라이언트는 Mobile RTP Proxy의 이동과는 상관없이 투명하게 서비스를 제공받는다. Mobile RTP Proxy는 MusicStation 서버에서 실

어 보낸 RTP 헤더 정보를 통하여 자신의 부하 정도를 판단하고, 패킷 손실률이 큰 경우 모든 클라이언트에게 원활한 서비스를 제공하지 못하는 경우로 판단하여 MMA에게 추천 노드를 의뢰하여 이동을 결정한다. 그림 10은 프로토타입 어플리케이션의 구조를 나타낸다.

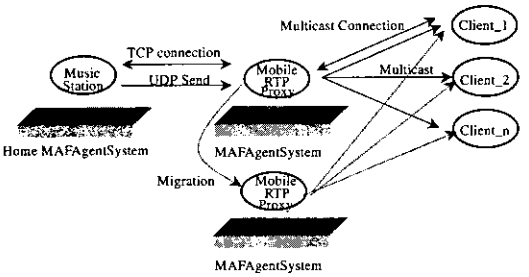

그림 10 프로토타입 어플리케이션의 구조

4.2 프로토타입 어플리케이션의 결과 및 분석

평가는 두 가지의 주목적이 있다. (1) 자원 감지를 고려하지 않은 배치에 대하여 자원 감지 배치의 성능 이득을 결정하기 위함이고, (2) 온라인 자원 감지 모니터링에 기반한 동적 배치가 초기 정보에 의한 단 한번의 이동에 비해 상당한 성능 이득을 제공하는지를 결정하기 위함이다. Mobile RTP Proxy는 다음의 배치 전략에 따라 이동한다. 첫째, 정적 배치(static placement)는 어떠한 이동도 지원하지 않으며, 자원 감지 모니터링도 제공하지 않는다. 둘째, 초기에 최선의 위치를 찾기 위해 자원 정보를 사용하였다(one-movement placement). 셋째, 동적 배치(dynamic placement)로서 온라인 모니터링 정보를 사용하여 최적의 장소로 계속적인 이동을 지원한다. 성능 평가를 위하여 주기적으로 Mobile RTP proxy가 상주하는 노드에서 새로운 프로세스를 생성시켜 부하를 증가시켰으며, Mobile RTP proxy가 상주하지 않는 노드에서는 생성된 프로세스를 소멸시켜 작업 부하를 변화시켰다.

그림 11은 각각의 배치 전략에 대한 클라이언트의 누적 패킷 손실 갯수를 나타낸 것이며, 그림 12는 구간별 패킷 손실 갯수를 나타낸다. 성능 분석 결과는 다음과 같다.

프로토타입 환경의 Mobile RTP Proxy3에서 실행된 정적 배치의 경우 주기적인 작업 부하의 증가로 클라이언트의 패킷 손실이 증가하여 클라이언트의 요구 조건을 만족시키지 못하였다. 평균 패킷 손실률은 83.4%로서 시간이 지남에 따라 패킷 손실률은 더욱 큰 증가폭을 보이고 있다.

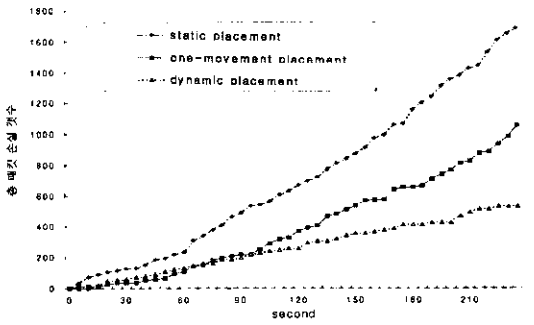

그림 11 배치 전략에 대한 클라이언트의 총 패킷 손실 수

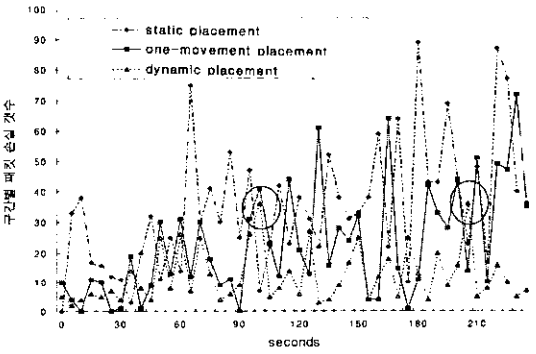

그림 12 배치 전략에 대한 클라이언트의 구간별 패킷 손실 개수

Mobile RTP Proxy1에서 실행된 One-movement 배치의 경우에는 처음에는 안정된 서비스를 제공하였으나, 정적 배치와 마찬가지로 부하가 증가함에 따라 만족할 만한 서비스를 제공하지 못하였다. 52.2%의 평균 패킷 손실률을 기록하였다. 그러나, 동적 배치의 경우 자원 감지 모니터링을 통하여 Mobile RTP Proxy가 작업 부하가 적은 노드로 지속적으로 이동하여 원활한 서비스를 제공하였으며, 모든 클라이언트에게 동등한 서비스를 제공하였다. Mobile RTP Proxy는 두 번 이동하였으며, 26%의 평균 패킷 손실률을 기록하여 성능 향상을 보였다.

표 2는 기존의 에이전트 시스템과 제안한 이동 에이전트 기반의 동적 부하 균형 프레임워크의 비교를 나타낸다. Aglet은 자원 감지 모니터링을 수행하지 않으며, 유휴 자원에 기반한 동적 배치를 제공하지 않는다. XAgent의 경우 자원 감지를 통한 동적 배치를 제공하고 있지만, 과다한 통신 오버헤드와 중앙 집중형의 통신

구조로 확장성이 결여되어 있다. 그리고, 자원의 사용 현황 조사와 추천 노드 선정 알고리즘을 각각의 노드에서 독립적으로 실행하는 고비용의 자원 감지 모니터링을 한다. 또한, 고정된 부하 계산 프로세스를 사용하므로 다양한 어플리케이션의 요구 조건을 만족하지 못한다. 반면 제안한 프레임워크는 SLM과 MMA의 상호 작용을 통한 저비용의 자원 감지 모니터링을 실행하여 어플리케이션의 동적 배치를 제공하며 확장성을 제공한다. 또한 MonitorHandler 객체를 사용하여 다양한 어플리케이션의 요구 조건을 수용할 수 있다.

Aglet과 XAgent는 자바로 구현되어 언어 독립적이지 못하며, MASIF를 준수하지 않아 서로 다른 에이전트 시스템간에 상호 운용성을 제공하고 있지 못하다. 반면 제안한 프레임워크는 CORBA 기반 MASIF를 이용하여 설계되어 상호 운용성을 제공하며, 언어 독립적이므로 구현된 언어와는 상관없이 구현이 가능하다.

표 2 기존 에이전트 시스템과의 비교

	Aglet[6]	XAgnet[7]	제안한 프레임워크
자원 감지	X	○	○
동적 배치	X	○	○
확장성	X	X	○
자원 감지 효율성	X	X	○
자원 감지 다양성	X	X	○
상호 운용성	X	X	○
언어 독립성	X	X	○

표 3은 부하 상태 측정과 이동을 위한 의사 결정 알고리즘 측면에서[16] 기존 작업 부하 균형 알고리즘과 제안한 프레임워크에서 사용한 알고리즘을 비교하였다. Aglet은 시스템의 부하 상태를 무시한 채 태스크의 성격에 따라 태스크를 정적으로 할당한다. XAgent는 시스템의 현재 부하 정도를 고려하여 동적으로 태스크를 할당한다. 그러나, 고비용의 모니터링과 중앙 집중식의 부하 상태 측정과 이동을 위한 의사 결정 알고리즘을 수행하여 확장성과 결함 포용성(fault tolerance)을 제공하지 못한다. 반면, 제안한 프레임워크는 저비용의 주문형 모니터링을 사용하며 부하 상태 측정과 이동을 위한 의사 결정을 분산시키어 확장성과 결함 포용성을 제공한다.

표 3 작업 부하 균형 알고리즘의 비교

	Aglet [6]	XAgent [7]	제안된 프레임워크의 작업 부하 균형 알고리즘
system state estimation	static	dynamic centralized periodic cpu load	dynamic decentralized on demand cpu load+network
decision making	deter-ministic	centralized sender initiated first fit	decentralized sender initiated best fit

표 4은 기존 작업 부하 균형 모델과의 비교를 나타낸다. 기존 모델은 한정된 어플리케이션과 실행 환경에 대해서만 작업 부하 균형을 제공한다. 하지만, 제안한 프레임워크는 이동 에이전트의 본래의 장점에 MASIF, 동적인 분산 자원 모니터링, MonitorHandler를 추가하여 이질적인 분산 컴퓨팅 환경의 복잡성을 작업 부하 균형 어플리케이션 개발자로부터 은닉한다. 특히, MonitorHandler는 객체로 구현되므로 범용성과 코드 재사용을 가능하게 한다. 따라서, 제안한 프레임워크는 이동 에이전트 패러다임을 적용하여 이질적인 분산 컴퓨팅 환경에서의 동적인 작업 부하 균형 어플리케이션 개발을 용이하게 하며 다양한 플랫폼에서 적용 가능한 프레임워크를 제공한다.

표 4 기존 작업 부하 균형 모델과의 비교

	기존 작업 부하 균형 모델	이동 에이전트 패러다임을 적용한 작업 부하 균형 프레임워크
이질적 환경	△	○
확장성	X	○
범용성	X	○
코드 재사용성	X	○
통신 프로토콜	변형이 어려움	변형이 쉬움

5. 결론 및 향후 연구 과제

본 논문에서는 이동 에이전트 패러다임을 작업 부하 균형에 적용하여 그 실효성을 프로토타입 어플리케이션을 통하여 검증하였다. 제안한 프레임워크는 MASIF를 기반으로 설계되어 이질적인 분산 컴퓨팅 환경에서 어플리케이션을 동적으로 이식한다. 또한, 자원 감지 모니터링을 수행하여 유휴 자원 정보에 기반한 작업 부하 균형 어플리케이션을 동적으로 배치한다. MonitorHandler는 CORBA의 메타 레벨 아키텍처를 이용하여 다양한 모니터링 알고리즘을 동적으로 실행시키고 필요

에 따라 동적으로 교체하여 융통성과 확장성을 제공하며, 특정 어플리케이션에 종속되지 않는다(application-nonspecific). 구현한 프로토타입 어플리케이션의 실험 결과 이질적인 분산 컴퓨팅 환경에서 유휴 자원을 고려한 동적 배치가 정적 배치나 초기 정보에 의한 단 한번의 배치보다 57%와 26%의 성능 향상을 보였다. 따라서 제안하는 프레임워크는 작업 부하 균형에 이동 에이전트 패러다임의 적용 가능성을 증명하였으며, 우수한 성능 향상을 나타내었다. 프로토타입 어플리케이션 이외에도 여러 개의 프로세싱 파워를 요구하는 병렬 처리 어플리케이션의 경우에도 이동 에이전트를 복사하여 병렬적으로 수행할 수 있다.

그러나 제안한 프레임워크는 단일 에이전트(single MA)를 기반으로 하고 있으며, 이로 인해 순회해야할 서버의 수가 증가함에 따라 네트워크 환경에서의 여러 가지 변화를 실시간(real-time)으로 반영한 효율적인 관리, 운영의 한계가 존재한다. 이를 해결하기 위하여 멀티 에이전트(MultiMA) 기반의 연구가 요구되며, 현재 멀티 에이전트 시스템의 연구와 멀티 에이전트간의 상호 협동을 포함시킨 연구를 진행 중이다.

참 고 문 헌

[1] OMG, CORBA Service: Common Object Services Specification, 1988.

[2] OMG, Mobile Agent System Interoperability Facilities Specification, November, 1997.

[3] Niranjan G. Shivaratri, Phillip Krueger, Mukesh Singhal, "Load Distributing for Locally Distributed Systems," IEEE Computer, Vol. 25, No. 12, pp. 33-44, December, 1992.

[4] Chokchai Leangsuksun, Jerry Potter, Stephen Scott, "Dynamic Task Mapping Algorithms for a Distributed Heterogeneous Computing Environment," Heterogeneous Computing Workshop, Vol. 3, No. 2, pp. 42-50, April, 1995.

[5] Davis Chess, Colin Garrison, Aaron Kershenbaum. "Mobile agents: Are they a good idea?," in Mobile Object Systems: Towards the programmable internet, Springer-Verlag, Vol. 6, No. 1, pp. 46-48, April, 1997.

[6] IBM Tokyo Research Labs, Aglet Warkbench: Programming Mobile Agents in Java, http://www.trl.ibm.co.jp/aglets/, 1999.

[7] EPFL's Artificial Intelligence Labs, "XAgent : Mobile Agents phase2," http://liawww.epfl.ch/~willmott/XAgents/work_done.html, 1999.

[8] John F. Karpovich, "Support for Object Placement in Wide Area Heterogeneous Distributed Systems," Dissertation Proposal; University of Virginia Dept. of Computer Science Technical Report CS-96-03, January 1996.

[9] Jim Farley, JAVA Distributed Computing, O'Reilly, 1998.

[10] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, Van Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 1889, January, 1996.

[11] Stephen E. Deering, "Host Extensions for IP Multicasting," RFC1112, August, 1989.

[12] Thomas Maufer, Deploying IP Multicast in the enterprise, Prentice Hall. 1997.

[13] FreeAmp, http://www.freeamp.org/, 1999.

[14] Paradeep K. Sinha, Distributed Operating Systems : Concepts and Design, IEEE Computer Society press, 1997.

[15] Danny B. Lange, Mitsuru Oshima, Lange Programming and Deploying Java Mobile Agents with Aglets, Addison Wesley, 1998.

[16] H. G. Rotithor, "Tanonomy of dynamic task scheduling schemes in distributed computing systems," IEE Proc.-Comput. Digit. Tech, Vol. 141, No. 1, January, 1994.

김 지 균

1997년 경희대학교 산업공학과 학사. 1998년 ~ 현재 고려대학교 컴퓨터학과 석사과정 재학. 관심분야는 네트워크, 분산시스템, RTOS, 보안

김 태 윤

정보과학회논문지 : 정보통신
제 28 권 제 1 호 참조