論文2002-39CI-1-1

J-JDBS: Jini를 이용한 자바 분산 일괄처리 시스템

(J-JDBS: Java Distributed Batch-processing System Using Jini)

具 槿 *, 金 正 善 **

(Geon Goo and Jungsun Kim)

요 약

분산 일괄처리 시스템은 네트워크 상에 있는 유휴상태의 컴퓨터를 활용하여 각 컴퓨터에 의하여 제출된 CPU-intensive한 작업을 수행시킴으로써 컴퓨터 자원의 활용도 및 작업의 생산성을 높일 수 있도록 하는 시스템이다. 이러한 시스템이 효과적으로 운용되기 위해서는 규모확장성, 결함 허용성, 그리고 머신 풀(pool) 구성의 유연성 등이 보장되는 것이 바람직하다. 그러나, 네트워크의 결함이나 컴퓨터의 다운 또는 컴퓨터 소유자의 자율적 의지에 의한 풀에서의 탈퇴 등으로 인하여 사용 가능한 컴퓨터가 비결정적(non-deterministic)일 수밖에 없는 동적 환경 하에서 그와 같은 특성을 보장하기는 매우 어렵다. 본 논문에서는 Jini의 core 서비스를 이용한 분산 일괄처리 시스템인 J-JDBS (Jini-based Java Distributed Batch-processing System) 시스템의 설계 및 구현을 제시함으로써, Discovery 서비스, Lookup 서비스, Lease 서비스 등의 Jini 코아 서비스가 규모확장성, 결함 허용성, 풀 구성의 유연성을 보장하는 신뢰성 있는 분산 일괄처리 시스템을 쉽고 빠르게 구축하는 데 매우 유용하게 적용될 수 있음을 보인다.

Abstract

In Distributed Batch-processing Systems (DBSs), CPU-intensive jobs are automatically transferred and executed using idle computers across a network, there by increasing the resource usage and throughput. To be successful, the systems must guarantee the scalablility, fault-tolerance, and flexibility of dynamic configurations. In practice, however, it is very difficult to provide such capabilities in a non-deterministic environment in which the available set of resources is unpredictable because of network failures, computer failures, or voluntary withdrawal from a pool by a machine owner. In this paper, we present the design and implementation of the J-JDBS (Jini-based Java Distributed Batch-processing System) system which is based on the core Jini services like Discovery service, Lookup service, Lease service and etc. We show that the Jini core services can be very effectively used to build reliable, scalable, fault-tolerant, and flexible DBS systems with little effort.

* 正會員, (株) 씽크프리 코리아 연구원
(ThingkFree)
** 正會員, 漢陽大學校 電子컴퓨터 工學部
(School of Electrical Engineering and Computer Science, Hanyang University)
接受日字:2001年8月7日, 수정완료일:2001年10月15日

I. 서 론

분산 일괄처리 시스템(Distributed Batch-processing System)이란 일반적으로 사용자와의 상호작용(interaction)이 없는 일괄처리 (batch) 형태의 작업들을 컴퓨터 풀(pool)에 속한 유휴(idle)상태의 컴퓨터에 분산시켜 수행시킴으로써, 유휴 컴퓨팅 자원의 활용도와 작

업 생산성의 향상을 목적으로 하는 시스템을 의미한다[1]. 분산 일괄처리 시스템은 워크스테이션 및 개인용 컴퓨터의 활용도가 사용자의 유형과 하루의 시간대에 따라 크게 차이가 난다는 점에 착안하였으며, 기존의 네트워크 환경을 토대로 각 컴퓨터에 최소한의 런타임 모듈만을 설치함으로써 쉽게 구축될 수 있다는 특징이 있다[2,3]. 이러한 시스템은 오랜 시간동안의 수행을 필요로 하는 자연과학 또는 공학분야의 시뮬레이션 작업을 처리하는 데 매우 효과적이며, JDBS[1], BATRUN[2,3], Condor[4,5], Utopia[6][1], 등은 지금까지 개발된 분산 일괄처리 시스템의 대표적인 예이다.

일반적으로, 분산 일괄처리 시스템은 하나의 마스터 머신과 다수의 슬레이브 머신들로 풀(pool)을 형성하며, 마스터는 풀에 등록된 각 슬레이브가 제출한 작업에 대하여 유휴 상태의 슬레이브 머신(즉, 타겟 머신)에서 그 작업이 수행될 수 있도록 한다. 그런데, 이러한 시스템에서는 네트워크의 결함이나 컴퓨터 자체의 다운 또는 컴퓨터 소유자의 자율적 의지에 의한 풀에서의 탈퇴 등으로 인하여 사용 가능한 컴퓨터 자원이 비결정적일 수밖에 없기 때문에 풀의 구성 및 관리가 동적으로 유연하게 이루어져야 하며, 결함 허용성과 규모확장성이 지원되는 것이 바람직하다. 그러나, 지금까지 개발된 분산 일괄처리 시스템에서는 정적인 풀의 구성만을 지원하여 결함 허용성이 결여되어 있거나 동적인 구성을 지원한다 하더라도 그 프로토콜과 메커니즘이 이식성이 없으며 규모 확장성 또한 용이하지 못하다.

한편, Sun microsystem에 의해 발표된 Jini(TM)[7]는 동적인 네트워크 환경에 연결된 임의의 서비스(장치 또는 소프트웨어)들 간에 서로를 발견하여 클라이언트/서버 관계를 맺을 수 있도록 하여 주는 표준 프레임워크를 제공한다. 비록 Jini는 PDA, 카메라, 프린터 등과 같이 비교적 소형의 하드웨어 장치들이 네트워크 환경에서 서로 동적으로 상호 작용할 수 있도록 하기 위하여 제안되었으나, 다음과 같은 Jini의 기본적 특성은 동적인 환경에서의 분산 일괄처리 시스템의 요건을 만족시키는 데 있어서도 매우 적합하다.

● 서비스를 구동시킬 때 이외에는 서비스가 online 또는 offline 될 때마다 관리자의 개입을 필요로 하지 않는다.

● Jini-aware 시스템은 self-healing 능력을 가지고 있다. 즉, 서비스 또는 서비스 사용자가 비결정적으로 등장하거나 사라지더라도 그 상황을 스스로 알아서 대처한다.

특히, Jini의 core를 구성하는 Discovery 서비스, Lookup 서비스, Lease 기능, Distributed Events, RMI 기능 등은 분산객체 모델을 기반으로 한 신뢰성 있고 규모확장성 있는 분산 일괄처리 시스템의 구축을 매우 용이하게 한다. 예로써, Jini의 Discovery 서비스는 비결정적으로 (non-deterministic) 유휴상태와 그렇지 않은 상태를 오가는 컴퓨터들 가운데 유휴상태의 머신들만을 이용하여 동적으로 풀을 구성하는 데 편리하게 사용되며, Lease 기능을 이용하면 다운되는 머신에 대해서도 소유자의 자율적 의지에 의한 탈퇴와 동일하게 유연하게 대처할 수 있다. 또한 Jini의 Lookup 서비스는 마스터의 구성요소인 Matchmaker[1]의 기능과 중복되는 부분이 많아 최소한의 노력으로 Matchmaker의 구현이 가능하다. 그밖에도 RMI(Remote Method Invocation)[8]와 Security Manager[9]는 각각 객체지향적 설계와 보안문제 해결에 많은 기여를 한다.

본 논문에서는 Jini 기반의 분산 일괄처리 시스템인 J-JDBS(Jini-based Java Distributed Batch-processing System) 시스템의 설계 및 구현을 제시한다. J-JDBS는 JDBS (Java Distributed Batch-processing System)[1]의 개발 경험을 토대로 Jini 기술을 접목시켜 개발한 분산 일괄처리 시스템이다. JDBS는 기존의 분산 일괄처리 시스템에 자바의 플랫폼 독립성을 접목시켜 사용 가능한 자원의 확보를 극대화하였으며 기존의 분산 배치 처리 시스템들에 비해 이식성, 확장성, 자율성 및 결함 허용 기능 뿐 만 아니라 swing을 이용한 편리한 사용자 및 관리자 인터페이스를 제공한다. 그러나, 다른 시스템과 마찬가지로 JDBS에서도 유휴 머신 풀(pool) 구성의 유연성, 규모확장성, 결함 허용성을 보장하기 위하여 고유의 메커니즘과 프로토콜을 사용하였다. J-JDBS는 JDBS의 장점을 그대로 유지하면서 최소한의 노력으로 유휴 머신 풀(pool) 구성의 유연성, 규모확장성, 결함 허용성을 더욱 신뢰성 있게 보장할 수 있도록 Jini를 기반으로 설계하고 구현하였다. 즉, J-JDBS에서는 마스터와 슬레이브가 제공하는 기능을 각각 별도의 Jini 서비스로서 정의하고, 시

1) 실행되는 작업 유형의 관점에서 보면, Utopia는 다른 시스템과는 달리 interactive한 작업의 실행까지 지원한다.

스템의 구성에 Jini의 core 서비스를 활용함으로써 결함 허용성과 규모확장성을 보장하는 동적 시스템의 구성을 용이하게 한다.

본 논문의 구성은 다음과 같다. 2장에서는 일반적 분산일괄처리 시스템의 개요를 살펴보고, 3장에서는 J-JDBS의 구조와 구현을 제시한다. 4장에서는 J-JDBS의 특징에 대하여 설명하고, 마지막으로 5장에서는 결론 및 향후 과제를 제시하도록 한다.

Ⅱ. 분산 일괄처리 시스템(Distributed Batch-processing System)의 개요

분산 일괄처리 시스템(이하 DBS라 칭함)에 의하여 수행되는 작업은 사용자와 상호작용이 없는 CPU-intensive한 일괄작업(batch job)인 것이 일반적이며 여기서는 그러한 작업을 DBS Job이라고 부르기로 한다. 이 장에서는 J-JDBS의 이해를 돕기 위하여 일반적 DBS의 구조와 DBS Job의 수행 시나리오를 간단하게 소개한다.

1. DBS의 일반적 구조

CONDOR, UTOPIA, BATRUN, JDBS등은 지금까지 개발된 대표적 시스템으로서, 이와 같은 시스템에서는 하나의 마스터 머신과 다수의 슬레이브 머신들로 풀(pool)을 형성하며, 마스터는 풀에 등록된 각 슬레이브가 제출한 작업에 대하여 유휴 상태의 슬레이브 머신(즉, 타겟 머신)에서 그 작업이 수행될 수 있도록 한다. 이때 풀의 구성은 그림 1 (a)와 같은 별(star)형 구조 또는 그림 1 (b)와 같은 클러스터 구조를 사용한다.

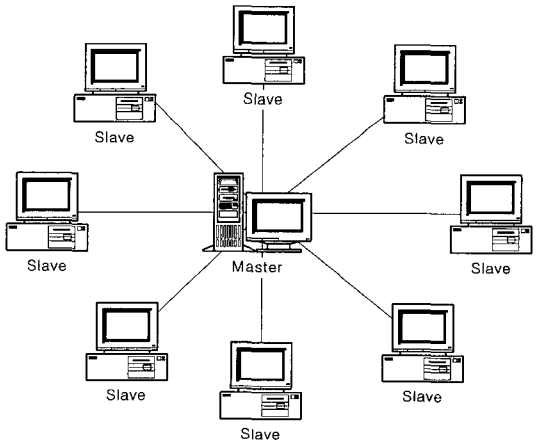


그림 1. (a). 별(star)형 구조
Fig. 1. (a). Star Structure.

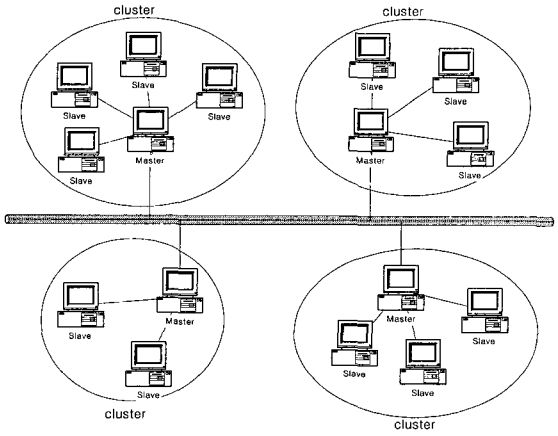


그림 1. (b). 클러스터 구조
Fig. 1. (b). Cluster Structure.

비교적 초기 시스템인 CONDOR에서 채택한 별형 구조는 하나의 마스터에 모든 슬레이브들이 연결되어 있는 중앙 집중적인 방식으로 비교적 구현이 간단하지만 마스터가 다운되는 경우 전체 시스템의 작동이 중지되기 때문에 결함허용 능력이 없고, 슬레이브가 늘어남에 마스터의 부하가 커지므로 규모 확장성이 없다. 이러한 문제를 극복하기 위하여 BATRUN과 JDBS에서는 별형 구조의 연합 (federation) 형태인 클러스터 구조를 사용한다. 클러스터 구조에서는 마스터가 다운되는 경우, BATRUN과 같이 다운된 마스터가 포함된 클러스터 전체를 가용 머신 풀에서 제외시키거나, UTOPIA와 JDBS에서와 같이 다운된 머신이 속한 클러스터내의 슬레이브들 가운데 새로운 마스터를 자동으로 선출하는 방법을 취한다. 후자의 경우, 가용머신의 수가 클러스터 단위가 아닌 머신 단위로 감소하므로 자원의 활용도에 있어 우수하다고 볼 수 있다. J-JDBS 역시 클러스터 구조를 채택하고 있다. 그러나, UTOPIA와 JDBS에서와는 달리 마스터 다운시에 각 슬레이브들로 하여금 새로운 마스터의 슬레이브로 등록되도록 하는 방법을 취하며, 이는 Jini의 core 서비스를 이용하여 쉽게 구현된다.

지금까지 DBS의 일반적 구조를 살펴 보았으나, 여기서는 특별한 구조에 상관 없이 일반적으로 수행되는 마스터와 슬레이브의 역할을 간단히 설명한다.

• 마스터(Master): 마스터의 가장 중요한 기능은 match-making이다. 즉, 제출된 DBS Job에 대하여 그 Job이 요구하는 사양을 만족하는 유휴상태의 슬레이브

를 찾아내어 작업을 제출한 머신과 연결시켜 주는 작업이다. 따라서, 마스터는 등록되어 있는 슬레이브들의 유휴 상태 여부를 감시해야 하며 각 머신의 메모리 용량과 CPU 종류 등에 대한 정보를 가지고 있어야 한다. 일반적으로 마스터는 match-making까지만 관여하며 실제 작업의 전송과 수행은 연결된 슬레이브 쌍에 의하여 처리된다.

• 슬레이브(Slave): 실제로 CPU를 공유해 주어 DBS Job을 실행시키는 역할을 한다. 이를 위해서는 로컬머신의 상태를 마스터에게 알려 주어야 하는 데, 일반적으로 처음에 마스터에 등록할 때는 CPU의 종류나 전체 메모리 크기 등과 같이 변하지 않는 정보를 등록시키고 그 후에는 정기적으로 CPU의 부하나 가용 메모리 크기 등의 가변적인 정보를 주기적으로 보내거나 또는 마스터의 요청이 있을 때마다 보내도록 한다.

2. DBS Job의 수행 시나리오

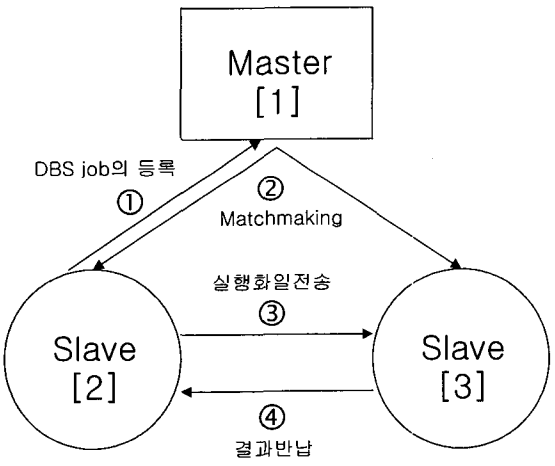


그림 2. DBS의 동작 시나리오
Fig. 2. Operational scenario of DBS.

그림 2는 DBS Job이 처리되는 시나리오를 나타내는 그림이다. 여기서는 간단하게 단일 클러스터 내에서의 기본적인 동작 과정만을 나타내도록 한다. 대괄호 안의 숫자는 편의상 컴퓨터에 번호를 붙인 것으로서, 현재 2번 머신(제출 머신)을 통하여 제출된 DBS Job이 3번 머신(타겟 머신)에서 수행되는 과정을 보여 주고 있다.

① DBS Job의 등록: 제출 머신(2번)은 수행시키기 원하는 DBS Job에 대한 정보를 마스터(1번)에게 등록시킨다.

② Match-making: 2,3번 머신을 비롯한 모든 슬레이브들은 busy/idle을 포함한 자신의 상태정보를 주기적으로 마스터에게 보내게 되며, 마스터는 슬레이브들이 제출한 상태정보를 검사하여 ①에서 제출한 작업의 요구환경을 만족하는 유휴(idle)상태 머신이 발견되면 제출 머신과 타겟머신에게 서로가 조건이 맞았음을 알려준다.

③ 실행 파일 전송: 제출 머신은 타겟 머신이 Job을 수신할 수 있는 지 확인한 후에 타겟 머신에게 DBS Job의 실제 실행 파일과 데이터를 전송하게 된다.

④ 결과 반납: DBS Job의 수행이 완료되면 타겟 머신은 제출 머신에게 결과를 넘겨준다.

Ⅲ. Jini 기반 분산 일괄처리 시스템 (J-JDBS)의 구조 및 구현

이 장에서는 Jini 기반 분산 일괄처리 시스템인 J-JDBS의 구조 및 구현내용과 구현시 고려사항 등에 대하여 기술한다.

1. J-JDBS의 구조

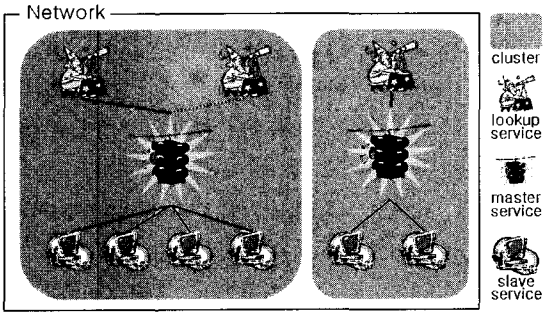


그림 3. J-JDBS의 구조
Fig. 3. Structure of J-JDBS.

J-JDBS는 기존 DBS시스템에서의 마스터와 슬레이브의 역할을 각각 Master Service와 Slave Service라는 새로운 Jini 서비스로서 정의하였다. 따라서, J-JDBS에서는 Master Service를 실행하는 임의의 머신 (이후 "마스터 서버" 또는 "마스터"라 칭함)이 일반적 DBS에서의 마스터 역할을 수행하게 되고, Slave Service를 실행하는 임의의 머신(이후 "슬레이브 서버" 또는 "슬레이브"라 칭함)이 일반적 DBS에서의 슬레이브 역할을 수행하게 된다. 그러나, J-JDBS에서 실제 마스터의 기

능은 마스터 서버와 Lookup 서버(즉, Jini의 core 서비스인 Lookup Service를 실행하는 머신)의 공조에 의하여 수행된다. 이는 기존 시스템에서 마스터가 수행하던 match-making 작업의 많은 부분이 Jini의 core 서비스인 Lookup Service를 통하여 쉽게 구현할 수 있기 때문이며, 이렇게 함으로써 Jini의 특성을 최대한 활용함과 동시에 코드의 작성을 최소화할 수 있게 된다.

J-JDBS는 시스템의 확장성과 결함 허용성을 위하여 그림 1 (b)의 클러스터 구조를 채택하였다. 그림 3은 현재 네트워크 상에 2개의 클러스터가 생성되어 있는 J-JDBS의 상태를 보여준다. 그림 3에서 보는 바와 같이, J-JDBS에서는 하나의 클러스터 내에 하나의 Master Service와 다수의 Slave Service, 그리고 하나 이상의 Lookup Service가 존재한다. 클러스터 내의 모든 머신은 Master Service와 Slave Service를 모두 제공할 수 있는 대칭적 구조를 갖기 때문에 언제든지 마스터 서버 또는 슬레이브 서버로서의 역할을 수행할 수 있다.

한편, 각 머신에서의 Job의 제출과 관리, 작업 큐의 관리 기능 등은 Job Manager에 의하여 수행되도록 하였다. Master Service와 Slave Service와는 달리 Job Manager는 Jini의 서비스로서 정의되지 않았지만 각각의 Jini 클라이언트로서 동작해야 하기 때문에 Jini의 runtime을 필요로 한다.

1) 1 Lookup Service

Lookup Service는 클러스터의 동적 구성 및 match-making을 위한 핵심 서비스를 제공하는 데, J-JDBS에서는 이 서비스를 위하여 Jini의 Lookup Service를 그대로 적용한다. 이 서비스가 클러스터의 동적 구성과 match-making에 어떻게 사용되는가는 뒤에서 설명하기로 한다. J-JDBS에서는 하나의 네트워크에 하나의 Lookup Service만 존재하여도 동작하지만 이 경우 Lookup 서버가 다운되는 경우 시스템 전체가 다운되기 때문에 결함 허용을 위하여 둘 이상의 인스턴스가 존재하도록 하는 것이 바람직하다.

2) Master Service

마스터 서버에 의하여 제공되는 서비스이다. 마스터 서버는 Lookup Service에 자신을 마스터로서 등록 할 때, 자신이 관장하게될 클러스터의 이름을 알려주며 이 때 주어진 클러스터의 이름은 슬레이브 서버가 클러스터에 가입할 때 나열되어 선택되어진다. 이를 위해 슬레이브 서버가 시작되기 전에 최소한 1개의 마스터가 이미 동작하고 있어야 한다. 만약, 마스터 서버가 존재하지 않는 다면, Master Service를 Lookup Service에 등록시켜 그 스스로를 새로운 클러스터의 마스터로 지정할 수 있다. 한편, Master Service는 Lookup Service의 event listener로 등록되어 슬레이브 서버들의 가입/탈퇴 상황을 알 수 있으며 그에 따라 동적으로 클러스터를 구성한다. 그리고 제출된 J-JDBS Job에 대하여 Jini Lookup Service의 template matching을 사용하여 match-making을 수행한다. 시스템의 결함허용 능력을 위해 클러스터를 2개 이상 만들어 두는 것이 좋다. 왜냐하면 J-JDBS에서는 하나의 Master Service가 다운되었을 경우 그와 연결되어 있던 모든 슬레이브들은 각기 다른 Master Service를 찾아 옮겨가기 때문이다.

3) Slave Service

슬레이브 서버에 의하여 제공되는 서비스로서 J-JDBS Job이 수행될 수 있는 환경을 제공하게 된다. 마스터 서버는 Slave Service를 통하여 주기적으로 슬레이브 서버의 상태정보를 요청하며, 요청을 받은 슬레이브 서버는 자신의 부하정보를 조사하여 idle/busy의 상태를 결정한다. Idle일 경우 슬레이브 서버는 자신을 Master Service에 등록시킨 후 주기적으로 자신의 상태를 속성 셋(attribute set)에 등록시킨다. 그러나 busy의 상태가 되면 Master Service에서 탈퇴시키고 속성 셋의 갱신을 중지시킨다.

4) Job Manager

Slave Service와 함께 슬레이브 서버에 의하여 수행되는 객체로서 J-JDBS Job의 제출 및 제출된 job의 취소와 모니터링에 관련된 인터페이스를 제공해 주는 기능을 한다. Job Manager는 Jini 서비스로서 정의하지 않았으나 내부적으로 Jini 런타임을 필요로 한다.

2. J-JDBS의 구현

1) 마스터 컴포넌트

Master Service에 관련된 UML(Unified Modeling Language)[10] class diagram은 그림 4와 같다.

인터페이스로 정의된 MasterService는 마스터 서버가 제공하는 Jini 서비스를 나타내며, 실제 서비스의 구현은 마스터 서버에 의하여 생성되는 MasterServiceImpl 객체에 의하여 제공된다. MasterServiceProxy는 MasterService 구현 객체인 MasterServiceImpl 객체를 대신하여 Lookup Service에 등록되어질 서비스 객체로서, 내부적으로 MasterServiceImpl과 RMI를 통

한 메시지 전달을 수행하지만 MasterServiceImpl의 Stub 클래스는 아니며 내부적으로만 RMI Stub을 사용하게 된다. 마스터 서버는 MasterServiceImpl 객체를 먼저 생성하고 MasterServiceProxy의 생성자에 인자로 이를 넣어 주게 된다. 이를 위한 Master Server의 개략적인 코드는 그림 5와 같다. 여기서 exception의 처리는 생략하였다.

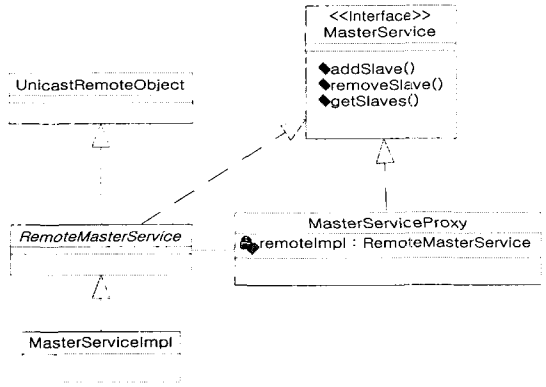


그림 4. Master Service
Fig. 4. Master Service.

```
public class MasterServer implements LeaseListener {
  MasterServiceImpl impl;
  MasterServiceProxy proxy;
  LookupManager lm;
  ServiceRegistrar sr;
  LeaseRenewalManager leaseManager;

  public MasterServer() {
    lm = new FooLookupManager(5);   // 5초간 Lookup 검색
    impl = new MasterServiceImpl("FooCluster",lm);
    proxy = new MasterServiceProxy(impl);
    sr = impl.getServiceRegistrar();
    ServiceRegistration reg =
      sr.register(new ServiceItem(null, proxy, null),
Lease.FOREVER);
    leaseManager = new LeaseRenewalManager( reg.getLease(),
Lease.FOREVER, this);  // Lease를 자동갱신해 준다
  } // constructor

  public void notify(LeaseRenewalEvent evt) {
    System.out.println("Lease Renewal Failed");
  } // Lease의 갱신에 실패 하였을 경우 호출됨

  public void static main(String[] args) throws Exception {
    System.setSecurityManager(new RMISecurityManager());
    new MasterServer();
  }
}
```

그림 5. 마스터 머신의 동작
Fig. 5. Operations of Master Machine.

LookupManager는 Jini의 Discovery protocol을 사용하여 네트워크 상의 Lookup Service를 검색해 내며, 결과로 얻은 Service Registrar들의 집합(set)을 관리하는 객체로서 ServiceRegistrar를 집합에 추가/삭제하는 메쏘드, 이들 중 하나만을 선택하여 주는 selectOne(), 관리하고 있는 모든 ServiceRegistrar들을 반납하는 getRegistrars()를 제공한다. selectOne()은 동일한 Lookup Service가 두 개 이상의 Master Service에 의해 사용되지 않도록 하기 위해 Master Service가 등록되어 있지 않은 Lookup Service만을 반납한다. 이들을 관리하는 방법과 selectOne()을 구현하는 방법이 다양하게 선택할 수 있도록 인터페이스로 작성되어 있다. 여기서 사용된 실제 implementation 클래스인 FooLookupManager는 생성자의 인자로 받아들인 waitTime(초) 시간안에 응답해 오는 ServiceRegistrar들을 최대 10개까지 저장하며 selectOne()에서는 보유하고 있는 ServiceRegistrar중 가장 index가 낮은 항목을 반납해 주는 간단한 Lookup Manager이다.

MasterServiceImpl은 마스터 서버 내에서 실제로 MasterService의 기능을 수행하는 객체로서, 마스터 서버에 의하여 생성될 때 생성자의 인자로서 클러스터의 이름과 사용하게 될 Lookup Service를 위한 Service Registrar 객체를 넘겨 받는다. 그림 6은 MasterServiceImpl 클래스와 그 주변 클래스들과의 관계를 나타내고 있다.

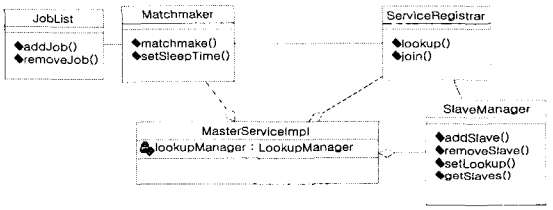


그림 6. Master Service의 구현 관련 클래스
Fig. 6. Class Diagram for Master Service.

여기서는 Slave Manager와 Matchmaker에 대하여 간단히 설명하도록 한다.

• Slave Manager: 마스터가 생성한 클러스터에 속하는 모든 슬레이브들의 서비스 객체를 관리한다. 마스터가 사용중인 Lookup Service에 슬레이브가 추가/삭제되면 Lookup Service로부터 그 사실을 통보 받아 해당되는 슬레이브의 서비스 객체를 리스트에서 추가/삭

제한다. 이후, 주기적으로 슬레이브들의 부하정보를 조사하여 유휴 상태의 머신만을 Lookup Service에 유지시키고 사용중인 상태의 머신은 Lookup Service에서 탈퇴시킴으로써 유휴상태의 머신만이 match-making의 대상이 되도록 한다. 그밖에도 Slave Manager는 Lookup Service의 결함 허용을 위해 다음과 같이 사용된다. 현재 마스터가 사용중인 Lookup Service에 오류가 발생하는 경우, 마스터는 문제를 일으킨 ServiceRegistrar를 lookupManager로부터 삭제시키고 다시 selectOne()을 호출하여 새로운 LookupService를 사용하게 된다. 이 순간에 새로운 Lookup Service에는 슬레이브들이 등록되어 있지 않기 때문에 SlaveManager의 setLookup()을 호출하여 슬레이브들을 등록시킨다.

• Matchmaker: 별도의 thread에 의하여 동작하며 Job List에 있는 작업들 중에 실행 조건을 만족하는 머신이 있는지를 점검하는 matchmake()를 정기적으로 수행한다. matchmake()의 내부 동작은 Lookup Service의 lookup()을 사용한다. lookup()은 Jini에서 서비스를 검색하기 위해 제공되는 메쏘드로서 *template matching* 기법을 사용한다. 이는 원하는 서비스를 클래스나 인터페이스의 타입으로 검색이 가능하며 이렇게 검색된 서비스들 중에 원하는 속성(attribute)을 만족하는 서비스를 추려내는 방식으로 사용된다. Jini에서는 서비스의 속성을 나타내는데 속성셋을 사용하게 되는데 이는 동시에 만족해야 하는 속성의 집합이다. 속성셋은 Entry 클래스를 상속받으며 Serializable을 만족해야하며 클래스의 public field 각각이 하나의 속성을 나타내게 된다. 하나의 서비스 객체에 편의를 위해 여러 속성셋을 사용할 수도 있다. Template matching은 이들 여러 속성셋 가운데 하나만 만족하면 match 시킨다. Jini의 속성에 대한 template matching은 같거나 다르다의 두 가지 경우로만 처리가 되므로(wildcard는 제외한다) 우리는 모든 속성을 각각 별도의 클래스로 만들고 equals() 메쏘드를 재정의(override) 하였다. J-JDBS에서 작업의 속성은 Operating System, Instruction Set Architecture, Total Memory, CPU Load, Free Disk, Free Memory의 항목으로 정의된다. 그림 7은 Total Memory 속성을 표현하는 클래스이다. 그 외의 다른 속성들도 각각 클래스로 구성하여 성격에 맞도록 equals() 메쏘드를 재정의 하여야 한다. 속성셋은 이런 속성들을 public field로 가지며 Entry라는 인터페이스를 갖는 클래스이며 각 필드에서 null은 wildcard 동작한다.

```
public class TotalMemory implements Serializable {
  public int _totalMemory;
  public JDBS_TotalMemory(int mem) {
    _totalMemory = mem;
  }
  public boolean equals(Object target) {
    if(target instanceof JDBS_TotalMemory) {
      JDBS_TotalMemory t=(JDBS_TotalMemory)target;
      if(_totalMemory>=t._totalMemory) return true;
    }
    return false;
  }
}
```

그림 7. Total Memory 속성을 위한 클래스
Fig. 7. Class for Total Memory Attribute.

```
public class MasterServiceProxy implements MasterService,
Serializable {
  RemoteMasterService master;
  public MasterProxy(RemoteMasterService impl) {
    master = impl;                    // don't necessary
RMI's lookup()
  } // Constructor
  public void addSlave(SlaveService slave) {
    master.addSlave(slave);
    System.out.println("new slave added to cluster ["+slave+"]");
  }
  public void removeSlave(SlaveService slave) {
    master.removeSlave(slave);
    System.out.println("one slave deleted from cluster
["+slave+"]");
  }
  ...
}
```

그림 8. Master Proxy
Fig. 8. Master Proxy.

setSleepTime()는 match-making에 너무 많은 CPU clock을 소모하는 것을 방지하기 위해 matchmake() 수행간에 삽입되는 sleep 시간을 초 단위로 지정한다. 이

값이 너무 짧으면 CPU에 걸리는 부하가 커지며 너무 긴 경우는 Job의 수행이 가능한 자원을 놓치게 된다.

MasterServiceProxy는 Master Service의 구현 객체를 대신하여 Lookup Service에 등록되는 서비스 객체이다. Master Service의 클라이언트인 슬레이브 서버와 Lookup Service로 이동되는 객체이기 때문에 Serializable interface를 구현하여야 한다. 내부적으로 RMI stub을 사용하여 MasterServiceImpl과 통신을 한다. 여기서는 일반적으로 RMI에서 원격객체의 레퍼런스를 얻는 방법과는 약간 다른 방법을 사용하는데 이는 원격객체를 사용하는 클라이언트인 MasterService-Proxy가 실제 MasterServiceImpl 객체의 레퍼런스를 생성자를 통해 넘겨받아 미리 알 수 있기 때문에 RMI의 bind와 lookup 과정이 필요하지 않기 때문이다. 이 방법은 Jini 프로그래밍에서는 자주 사용되는 방법이다. 이를 사용한 MasterServiceProxy의 개략적인 코드는 그림 8과 같다.

2) 슬레이브 컴포넌트

그림 9는 J-JDBS에서의 슬레이브 컴포넌트를 위해 필요한 클래스들의 구조를 상위레벨에서 나타낸 class diagram이다. 기본적으로 슬레이브 컴포넌트의 구조는 마스터 컴포넌트의 구성을 위해 사용되던 구조와 동일한 구조를 가짐으로 자세한 메커니즘에 관한 설명은 생략하도록 한다.

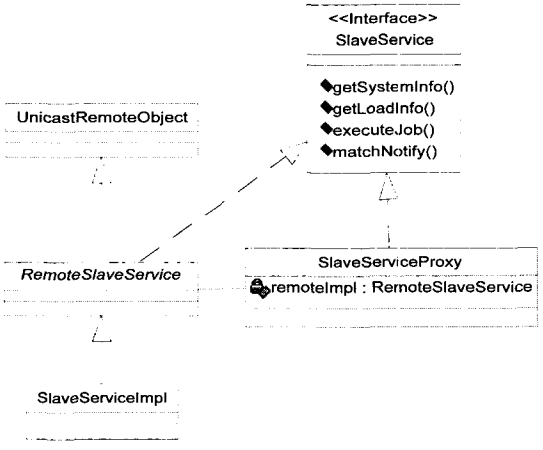


그림 9. Slave Service
Fig. 9. Slave Service.

하나의 머신이 슬레이브 서버로서 Slave Service를 Lookup Service에 등록하기 위해서는 네트워크 상에 수행중인 Master Service가 반드시 하나 이상 존하여야 한다. 슬레이브 서버는 Slave Service의 구현 객체인 SlaveServerImpl 객체를 생성 할 때 Master Manager를 사용하여 가입할 클러스터를 선택하게 된다. Master Manager는 Master Service가 사용중인 Lookup Manager의 getLookups()를 이용하여 네트워크에 존재하는 Lookup Service들을 얻은 후 각각에 등록된 모든 Master Service를 관리한다. 이는 네트워크 상에서 Master Service의 결함허용을 고려한 것이다. Slave Service가 현재 사용중인 Master Service에 오류가 발생할 경우 해당 Master Service를 Master Manager에서 제외시킨 후 selectOne()을 호출하여 새로운 Master Service에 가입하게 된다. MasterManager 역시 LookupManager와 같은 이유로 Interface만을 정의한다.

SlaveServiceImpl은 그림 9에서의 SlaveService 인터페이스를 구현한다. Slave Service가 제공하는 기능들에 대해 살펴보면 다음과 같다.

1) 부하정보 검출과 전송: 부하정보는 머신이 유휴상태인지 사용중 상태인지를 결정하는 지표가 된다. J-JDBS는 CPU load, 가용메모리, 가용디스크공간을 부하정보로 보며 이들의 조합으로 idle/busy 상태를 결정한다. 이들 부하정보중 가용메모리는 자바의 API에서 제공이 되지만 CPU load와 가용디스크 용량은 제공되지 않기 때문에 각 플랫폼별로 C언어로 작성하여 JNI(Java Native Interface)[11]를 사용하여 구현하였다. J-JDBS에서 자바로 구현되지 않은 유일한 부분으로서 새로운 플랫폼의 지원은 위의 두 가지 함수의 구현만으로 완성된다. 부하정보의 전달은 Master Service의 SlaveManager가 주기적으로 getSlaveInfo()를 호출하여 주는 방식을 취하게 된다. 이 때 동시에 모든 슬레이브로부터 정보를 얻어오면 Master Service에 순간적으로 큰 네트워 트래픽과 계산부하가 걸리기 때문에 하나씩 순서대로 호출하도록 하였다.

2) match 결과의 notify: Master Service에서 match-making에 성공하여 작업이 수행될 머신으로 결정되면 MatchNotifyEvent 타입의 이벤트 객체를 받으며 matchNotify()가 수행된다. MatchNotifyEvent에는 수행될 Job에 대한 정보와 J-JDBS Job을 제출한 머신에 대한 정보가 들어있다. 이를 참조하여 제출한 머신의 Job Manager로부터 수행할 작업을 받아 올 수 있게 된다.

3) J-JDBS Job의 실행: matchNotify()가 수행되면 JobExecutor 객체가 생성되어 작업을 수행시킨다. JobExecutor는 수행되는 작업에 1:1로 대응하여 생성되며 JAR 파일의 전송과 작업의 수행, 결과 반납을 담당한다. 이는 별도의 쓰레드로 수행되어 작업이 수행되는 동안에도 슬레이브의 동작에는 영향을 주지 않으며 2개 이상의 작업의 수행도 가능하다.

다음으로, 자바 2에서 새롭게 적용되는 보안정책은 sand-box 방식의 완고한 보안정책에서 벗어나 사용자에게 유연성을 최대한 보장해 준다. Slave Service는 확인되지 않은 코드를 다운로드 하여 수행시켜주는 역할을 하기 때문에 보안상의 권한설정을 하는 policy file을 작성해 주어야 한다. 여기서 사용되는 policy file은 2가지가 있다. 이들은 JDK에서 제공되는 policytool을 사용하여 편집가능하다.

1) 시스템 정책화일: J-JDBS 자체가 RMI를 기반으로 하기 때문에 RMISecurityManager가 적용된다. 따라서 보안정책 파일을 만들어 주지 않으면 socket연결이나 code download 등의 행동에 제약을 받게 된다. 그림 10은 Jini서비스와 Web Server를 통한 code downloading을 가능케 하는 시스템 보안정책 파일의 예를 보여준다.

```
grant {
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";
    permission java.net.SocketPermission "*:1024-", "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    permission java.util.PropertyPermission
                        "java.rmi.server.hostname", "read";
    permission java.util.PropertyPermission
                        "net.jini.discovery.*", "read";
    permission net.jini.discovery.DiscoveryPermission "*";
};
```

그림 10. J-JDBS 시스템 정책화일
Fig. 10. System Policy File for J-JDBS.

2) 실행 정책파일: JobExecutor가 전송 받아 실행하는 JAR 파일 형식의 코드는 검증되지 않은 코드이다. 따라서 파일 삭제 등의 보안상 위험이 있을 수 있기 때문에 이에 대응하는 policy 파일을 작성해 주어야 한다. 수행하는 작업에 따라 보안 관리자가 설치되어 있지 않은 프로그램들도 있으나 JobExecutor가 Java VM을 실행시킬 때 command line 상에서 보안관리자를 지정하여 주기 때문에 머신 제공자는 안전하게 머신을 공유해 줄 수 있다. 그림 11은 "\JDBS\Work\" 이하의 디렉토리와 파일에만 접근 권한을 주는 실행 정책파일의 예를 보여준다. 그 외에도 Socket 권한 AWT 권한 등 자바 2에서 사용 가능한 권한의 미세한 조절까지도 가능하다.

```
grant {
  ...
  permission java.io.FilePermission
                        "/JDBS/Work/-", "read,write,delete"
  ...
};
```

그림 11. J-JDBS 실행 정책화일
Fig. 11. Execution Policy File for J-JDBS.

3) Job Manager

Job Manager는 J-JDBS Job의 제출에 필요한 인터페이스를 제공하며 제출된 작업들에 대한 관리가 행하여진다. 관리는 2단계의 큐를 두어 결함에 의한 작업의 손실을 막아준다. Job Manager는 J-JDBS의 클라이언트 역할을 하기 때문에 수행되지 않더라도 J-JDBS 자체의 동작에는 영향을 끼치지 않지만 작업을 제출하는 측에서는 제출부터 수행의 완료시점까지는 Job Manager가 수행되고 있어야 한다.

• 대기큐(waiting queue): 사용자로부터 J-JDBS Job을 지정받으면 이는 Master Service에게 제출되고 바로 대기큐에 추가된다. 대기큐는 match-making에 아직 성공하지 못한 작업들을 저장하고 있다. Master Service가 match되었음을 notify해 주면 그 J-JDBS Job은 대기큐에서 삭제되며 실행큐로 넘어간다. 그 후에 별도의 thread로 구성되는 JobExecution 객체가 생성된다. 이는 Slave Service에서 존재하는 JobExecutor와 쌍을 이루며 생성되어 Job의 전송에서 결과 반납까지의 과정을 담당한다.

• 실행큐(execution queue): 현재 실행중인 J-JDBS Job들의 목록을 관리한다. JobExecutor가 실행의 완료

를 알려주면 실행큐에서 제거된다. 하지만 작업의 수행 중에 어떤 이유로든 중단되는 작업은 실행큐에서 삭제되고 대기큐로 다시 넘어가게 된다. 이는 작업의 중단으로 인한 유실을 막기 위함이다.

J-JDBS Job은 그림 12와 같다. name은 사람이 알아보기 편한 작업의 이름이며 임의의 문자열로 지정이 가능하다. options는 작업의 실행에 command-line에서 들어갈 option이다. path는 JAR 파일을 읽어 들일 수 있는 경로이다. JDBS에서 수행시킬 작업은 미리 JAR 파일로 패키징되어 있어야 한다. requirement는 작업이 수행될 곳의 조건을 지정해 주는데 systemInfo와 loadInfo의 각 항목중 don't care 항목에는 null을 지정할 수 있다.

그림 12. J-JDBS Job의 구조
Fig. 12. Strucure of J-JDBS Job.

Ⅳ. J-JDBS의 특성

J-JDBS는 JDBS의 개발 경험을 토대로 개발된 DBS 시스템이다. JDBS는 기존의 분산 일괄처리 시스템에 자바의 플랫폼 독립성을 접목시켜 사용 가능한 자원의 확보를 극대화 시킨 시스템으로서, 기존의 분산 배치처리 시스템들에 비해 이식성, 확장성, 자율성 및 결함 허용 기능 뿐 만 아니라 swing을 이용한 편리한 사용자 및 관리자 인터페이스를 제공한다. 그러나, 다른 시스템과 마찬가지로 JDBS에서도 유휴 머신 풀(pool) 구성의 유연성, 규모확장성, 결함 허용성을 보장하기 위하여 고유의 메커니즘과 프로토콜을 사용한 것에 비하여 J-JDBS는 JDBS의 장점을 그대로 유지하면서도 유휴 머신 풀(pool) 구성의 유연성, 규모확장성, 결함 허용성이 보다 쉽고 신뢰성있게 보장될 수 있도록 Jini를 최대한 활용하여 설계하고 구현하였으므로 다음과 같은 특성을 갖는다.

• 플랫폼 독립성: 자바의 플랫폼 독립성을 활용하여 작업을 수행시킬 컴퓨팅 자원의 확보를 극대화 시켜준다. J-JDBS는 수행시키는 DBS Job이 자바 애플리케이션일 뿐만 아니라 J-JDBS 자체도 거의 대부분이 자바로 되어있기 때문에 여러 기종의 컴퓨터에 포팅도 간단히 이루어질 수 있다. 현재, 자바만으로 전체 메모리의 크기와 사용 가능한 메모리의 사이즈는 알아 낼 수 있지만 CPU의 부하는 구하지 못한다. 이를 구하는 루틴은 하드웨어에 종속적인 code가 불가피하게 되는데, 이 루틴만 C언어로 각 플랫폼에 맞게 작성해 주고 JNI(Java Native Interface)로 자바 애플리케이션과 연결해 주면 J-JDBS를 수행시킬 수 있다.

• 안정성: J-JDBS는 그림 1 (b)의 클러스터 구조를 사용한다. 클러스터 내에서 마스터가 다운되었을 경우, 그 클러스터 내의 슬레이브들이 자동적으로 마스터가 살아 있는 임의의 클러스터에 가입되도록 함으로써 시스템을 안정적으로 유지시킨다. 이를 위하여 J-JDBS에서는 Jini의 Lookup Service와 event 처리 메커니즘을 활용하기 때문에 클러스터 구조를 사용하는 BATRUN이나 JDBS에서의 결함허용 메커니즘에 비하여 매우 간단하면서도 효과적으로 시스템의 안정성을 제공할 수 있다.

• 확장성: 하나의 마스터에 연결되는 슬레이브들이 증가할 경우 마스터가 받는 부하가 계속적으로 증가하기 때문에 시스템의 확장에 제한이 생기게 된다. 이를 보완하기 위해 J-JDBS에서는 그림 1 (b)와 같이 여러 마스터를 두어 부하를 분산시킬 수 있도록 하는 클러스터 구조를 취하며, 클러스터의 구성과 클러스터로의 가입과 탈퇴시 Jini의 표준 Lookup, Join, Discovery 등의 서비스를 활용하므로 확장성이 용이하다.

• 안전성: 안전성이란 시스템의 crash가 일어나지 않는다는 특징을 일컫는 말이다. J-JDBS는 자바가 프로그래밍 언어 차원에서 지원하는 안전성의 특성을 최대한 활용한다. 특히, 분산 일괄처리 시스템에서 자원을 제공해 주는 컴퓨터의 소유자는 타인의 작업이 crash를 일으켜 자신이 피해 입는 것을 원치 않기 때문에 이는 중요하게 생각되어야 한다. 이 특성은 자바언어 자체의 특성과 잘 결부되는데, pointer가 없다는 점이나 자동 자원 회수(auto garbage collection), 실행시간 배열 범위 검사, exception handling 같은 기능은 J-JDBS의 안전성을 보장해 주는데 충분한 역할을 한다. 뿐만 아니라 안전성 보장을 위해 Jini의 장점도 최대한 활용하고 있다. Jini의 임대(lease)는 사용 불가능한 자원에 request를 보냄으로서 생기는 오류문제를 해결하여 주며 클러스터 구성에 필요한 Lookup, Join, Discovery

기능을 하는 Jini의 클래스 라이브러리는 J-JDBS에 상당부분 재사용 될 수 있는데 이는 많은 검증이 끝난 라이브러리이기 때문에 보다 안정적인 시스템의 개발이 가능해 진다.

• 보안성: 컴퓨터의 소유자가 J-JDBS Job으로 인한 crash를 원치 않는 것과 마찬가지로 자신의 컴퓨터에서 파일 등의 정보를 유출해 가거나 삭제하는 것을 원치 않는다. 따라서 자바 2의 보안정책을 적용시켜 소유자가 원하는 만큼의 보안권한의 설정이 가능하다. 일반적인 경우라면 J-JDBS가 지정해 놓은 작업용 디렉토리 이하의 파일들에만 읽기/쓰기 권한을 주도록 한다.

• 효율성: J-JDBS 자체를 수행하는 것만으로 큰 오버헤드가 소요된다면 유휴 상태의 자원을 공유한다는 본연의 취지가 상실될 수 있다. 이를 방지하기 위해 J-JDBS에서는 다음과 같은 사항을 고려하였다.

1) JDBS Job의 압축 전송: J-JDBS Job을 수행시킬 수 있는 컴퓨터가 발견되면 그 유휴 상태의 컴퓨터로 자바 애플리케이션인 JDBS Job이 전송된다. 이 때 J-JDBS Job을 자바의 표준 압축 포맷인 JAR(Java ARchive) 파일로 압축하여 전송한다. 이는 네트워크의 통화량(traffic)을 감소시키며 보다 빠른 전송을 가능케 한다. 뿐만 아니라 작업에 필요한 추가적인 파일들까지 전송하기 때문에 Job을 전송한 곳의 파일에 접근하기 위해 사용되던 remote system call이나 공유화일 시스템이 요구되지 않으며 이에 따른 오버헤드도 없앨 수 있다.

2) Multi-thread의 사용: J-JDBS에서는 자바에서 지원하는 multi-thread 기능을 활용하여 프로세스 생성과 자원의 공유에서 오는 오버헤드를 최소화 하였다.

• 편의성: 기존의 DBS들이 일반적으로 Unix에 기반을 두고 제작되었기 때문에 GUI가 제공되지 않거나 있더라도 세련하지 못하였지만 J-JDBS는 자바의 Swing library를 사용하여 세련된 인터페이스를 제공한다. GUI는 직관적으로 사용법을 알아낼 수 있게 할 뿐 아니라 오타로 인한 오류를 줄여준다는 장점도 가지고 있다. 또한 J-JDBS의 내부동작은 3개의 Jini 서비스 -- Lookup Service, Master Service, Slave Service -- 로서 구성되어 있어 RMI를 사용한 호출이 가능하기 때문에 GUI 혹은 Text의 여러 종류의 클라이언트 제작이 용이하여 사용자에게 편의를 제공해 줄 수 있다.

• 동적 클러스터 구성의 용이성: J-JDBS에서는 클러스터의 가입과 탈퇴가 매우 용이하다. 마스터는 유동적인 머신 상태하에서 현재 유휴 상태인 머신들만의 목록(active list)을 유지할 수 있어야 하는 데, 이를 위해 Jini의 Discovery, Join, Lease을 사용하여 클러스터를 동적으로 유연하게 관리한다.

• 객체지향 설계: 분산된 환경에서의 UDP나 socket을 이용한 메시지 전달은 객체지향 설계기법에 곧바로 적용하는데는 어려움이 있어 각종 프레임워크나 디자인패턴의 장점을 이용하기 어렵기 때문에 조잡한 프로그램이 되기 쉬우며 확장성이나 유지보수에 문제를 겪게된다. 그러나 마스터와 슬레이브를 각각 서비스로 구성하여 객체로 보고 RMI를 사용하여 원격 객체와 로컬 객체간의 위치 투명성을 제공하면 객체지향의 설계의 장점을 쉽게 채용할 수 있다.

• Customization: Jini에서 사용하는 RMI는 객체의 위치 투명성을 보장해 준다. 때문에 J-JDBS의 구성에 필요한 Lookup Service, Master Service, Slave Service, 웹 서버 등을 모든 머신에 설치해야 될 필요가 없으며 각 머신의 부하상태나 시스템의 안정성, 네트워크 트래픽 등을 고려하여 원하는 대로 customize 할 수 있다. 즉, Master Service와 Lookup Service는 같은 머신에 있는 것이 효율적이지만 머신에 걸려있는 부하가 큰 경우 서로 다른 머신에 배치 할 수도 있으며 안정성을 부각시키고 싶은 경우는 Lookup Service를 두 개 이상 수행시킬 수도 있다. 웹서버는 서비스를 사용하는데 필요한 서비스 객체를 다운로드 하기 위해 사용된다.

• 구현의 용이성: JDBS의 구현시 직접 작성하였던 클러스터 관리, 다운된 머신들의 처리, Matchmaker 등이 Jini의 core 서비스와 상당부분 중복되기 때문에 J-JDBS 시스템 구현시 Jini를 사용함으로써 소스코드가 간결하고 이해가 쉽다.

V. 결론 및 향후 과제

본 논문에서는 네트워크에 연결된 유휴 상태의 컴퓨팅 자원의 활용을 위해 Jini기술을 적용한 J-JDBS (Jini-based Java Distributed Batch-processing System)을 소개하였다. J-JDBS와 유사한 시스템으로 Condor, UTOPIA, BATRUN 등이 있으나 이들은 수행할 작업의 플랫폼과 같은 플랫폼의 머신에서만 수행되기 때문에 이기종간의 CPU 공유는 수행하지 못한다.

이에 비해 J-JDBS는 Java의 플랫폼 독립성을 활용하여 플랫폼을 초월하여 CPU 공유를 가능토록 만든 분산 일괄처리 시스템이다. 자바의 사용으로 인하여 얻는 이점은 활용 가능 자원의 극대화에만이 있는 것이 아니라 새 플랫폼에 대한 쉬운 포팅, 안전성, 보안성, 효율성, 세련된 GUI 까지도 얻을 수 있다. 클러스터 구조를 채택하여 확장성과 안정성을 고려하였으며 JAR 패키지를 이용한 작업 전송을 사용하여 네트워크의 트래픽 감소에도 기여를 하였다.

한편, J-JDBS의 클러스터 구성에는 Jini 기술을 사용하였다. Jini의 동적 서비스 연합 구성 기능이 유휴 상태와 사용중 상태를 오가는 머신들을 한데 묶는 분산 일괄처리 시스템의 구현에 유용하게 적용될 수 있을 것으로 판단하였으며, 이는 실제로 J-JDBS의 구현의 기반 기술로 적합하다는 결론을 얻을 수 있었다. Jini 기술을 적용함으로써 동적인 클러스터 구성을 안정적이고 단순하게 이룰 수 있었으며 각각의 컴포넌트들을 시스템의 관리자의 의도대로 customize가 가능해 지는 등의 장점을 얻을 수 있었다. 그러나 Jini 자체의 부하가 크다는 문제점도 지적되었다.

현재 자바의 Security 정책을 기반으로 하여 보안정책이 사용되고 있지만 앞으로는 J-JDBS사용자에게 Log-in을 하고 사용토록 하여 보다 신뢰성 있는 환경을 제공하여 자원공유에 쉽게 참여할 수 있도록 하는 작업등이 수행되어야 한다.

참 고 문 헌

[1] 구건, 전진수, 김정선, "JDBS: 워크스테이션 네트워크를 이용한 자바 분산 배치 처리 시스템," 정보과학회 논문지(C) Vol 5, No 5, pp. 581-594, Oct. 1999.

[2] S. C. Kothari, F. Tandiary, and A. Dixit, "Design of BATRUN distributed Processing System," ISU Computer Science Department Technical Report 95-07, May 1995.

[3] S. C. Kothari, F. Tandiary, and A. Dixit, "BATRUN Distributed Processing System(DPS): Utilizing Idle Workstations for Large-scale Computing," IEEE Parallel and Distributed Technology: Systems and Applications, pp. 41-48, 1996.

[4] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter for idle workstations," Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 104-111, San Jose, June 1988.

[5] T. Tannenbaum and M. Litzkow, "The condor distributed processing system," Dr. Dobb's Journal, Feb. 1995.

[6] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "Utopia: A load sharing facility for large, heterogeneous distributed computer systems," Software-Practive and Experience, pp. 1305-1336, 1993.

[7] Sun Microsystems, "Jini(tm) Technology White Papers & Other Documentation," URL: http://www.sun.com/jini/whitepapers/. .

[8] Sun Microsystems, "Java Remote Method Invocation (RMI)," URL: http://java.sun.com/products/jdk/rmi/.

[9] L. Gong, "Inside Java 2 Platform Security," Addison-Wesley, 1999.

[10] G. Booch, I. J. Rumbaugh, and Jacobson, "The Unified Modeling Language User Guide," Addison-Wesley, 1999.

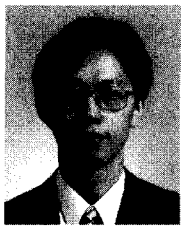
[11] R. Gordon, "Essential JNI: Java Native Interface," Prentice Hall, 1998.

[12] J. Meyer and T. Downing, Java Virtual Machine, O'Relly, 1997.

저 자 소 개

具 楗(正會員)

1998년 한양대학교 전자계산학과 졸업 (학사). 2000년 한양대학교 전자계산학과 졸업 (석사). 2000년~현재 (주)씽크프리 코리아 연구원. 관심분야: XML 기반 분산객체컴퓨팅, 객체지향 모델링/프로그래밍, 오피스웨어



金 正 善(正會員)

1986년 서울대학교 컴퓨터공학과 졸업 (학사). 1988년 Iowa State University 전기 및 컴퓨터공학과 졸업 (석사). 1994년 Iowa State University 전기 및 컴퓨터공학과 졸업 (박사). 1994~1996년 한국전자통신연구원 (ETRI) 선임연구원. 1996년~현재 한양대학교 전자컴퓨터공학부 조교수. 관심분야: Parallel/Distributed Processing, Distributed Object Computing, Component Based Development