분산리스트 네트워킹 메카니즘의 최적화에 관한 연구

임동주*

A Study on Optimization of Networking Mechanism of Distributed List

Dong Ju Im

요 약

본 논문에서 소켓 기반의 분산리스트와 RMI 기반의 분산리스트 구현을 중심으로 기술한다. 먼저 소켓 기반 분산리스트에 있어서 메시지를 포장할 때 객체 스트림을 사용하여 분산리스트 인터페이스를 분산환경에 맞게 구현하고 기술한다. 둘째로 RMI로 분산리스트를 구현하는 가장 큰 목적은 약간 복잡한 애플리케이션을 구현함으로써 다른 네트워킹 메카니즘과의 장단점을 비교하는 것으로 RMI를 사용할 때의 가장 큰 장점은 애플리케이션 레벨 프로토콜을 사용하지 않고도 프로그램을 간단하게 구현할 수 있다는 것이다. 프로그램의 효율이라는 측면에서 살펴본다면 RMI를 사용한 애플리케이션은 매 업데이트마다 생성되는 많은 TCP/IP 연결로 인해 성능이 매우 떨어질 수 있다[1]. TCP/IP 연결은 RMI에 비해 매우 적은 비용을 요구하며 비록 RMI가 하나의 네트워크 연결을 사용해서 여러 메소드 호출을 처리해 주는 메카니즘을 가지고 있기는 하지만 직접 소켓을 사용하는 것보다는 효율이 떨어진다. 그러나 RMI는 HTTP 프록싱 메카니즘을 사용하여 방화벽을 넘어 통신하는 것이 가능하다[2]. 따라서 두 시스템을 비교 분석함으로써 최적화 해법을 모색하여 네트워킹 메카니즘 모델링을 제시하고자 한다.

ABSTRACT

In this paper, I describe the implementation of the distributed lists based on socket and on RMI(Remote Method Invocation). First, I describe and implement an interface of distributed list based on socket using object stream in a distributed environment, when encapsulating the message in the distributed list. Second, the major purpose to implement a distributed list in RMI is to compare advantages and disadvantages with other networking mechanisms by implementing complicated applications. The major advantage in using RMI is to implement simply the programs without using application-level protocol. In terms of program efficiency, the applications using RMI can degrade the performance due to many TCP/IP connections generated every update. TCP/IP connection requires much less cost than RMI. Even though RMI has the mechanism processing many method calls using a single network connection, RMI is less efficient than the direct use of socket. However, RMI makes it possible to communicate beyond firewall using HTTP proxying mechanism. Consequently, I present a modeling of networking mechanism in finding out optimization solution by comparing and analyzing the two systems.

* 조선대학교 전산통계학과

1. 서 론

실세계에서 인터넷의 급속한 발달과 더불어 네트워킹의 발전은 필수 불가결한 요소이다. 따라서 클라이언트/서버 환경에서 좀 더 효율적이고 빠른 네트워킹에 대한 많은 연구가 진행되어 왔다. 이러한 네트워킹을 실현시킬 수 있는 두 가지 접근방법으로 하나는 클라이언트 측면에서 빠르고 효율적인 네트워킹 기법을 실현하는 것이고 다른 하나는 서버 측면에서 클라이언트 네트워킹과 최적의 상태로 통신할 수 있는 환경을 구축하여 효율적인 네트워킹을 형성하는 것이다.

본 논문에서는 분산리스트 자료구조를 중심으로 실질적인 구현을 통해서 클라이언트/서버 환경에서 자바를 이용하여 효율적인 네트워크 프로그래밍 기법을 모색하면서 발전적인 모델링을 제시하고자 한다. 먼저 소켓 기반 분산리스트 구현을 통해서 효율적인 클라이언트/서버 네트워킹에 대하여 기술하기 위하여 분산리스트 인터페이스를 분산환경에 맞게 구현하고 메시지를 포장할 때 객체 스트림을 사용한다. 본 논문에서 설계하고 구현한 화이트보드 프레임워크는 분산리스트 자료구조를 기반으로 만들어졌으며 분산리스트는 기본적으로 벡터와 매우 흡사한 엘리멘트의 리스트이다. 엘리멘트는 목록이나 값을 통해서만 액세스할 수 있다. 리스트의 n번째 요소를 액세스하는 기능을 제공하지 않은 이유는 투명성(transparency)이 생명인 분산환경에 맞지 않기 때문이다[3]. 리스트 내부에서는 인덱스 대신에 내용과 관련되는 식별자를 사용하여 엘리멘트를 조작한다. 이 식별자라고 하는 것을 서버에 연결된 클라이언트 안에다가 저장해두기 때문에, 어떤 클라이언트가 리스트의 엘리멘트를 변경하고 싶다면 식별자를 사용하는 것이고, 모든 다른 클라이언트들은 어떤 엘리멘트가 어드레싱 되었는지를 알 수 있다.

둘째로 RMI 기반 분산리스트 구현을 통해서 효율적인 클라이언트/서버 네트워킹에 대하여 기술한다. 소켓 기반 시스템에서 리스트를 갱신하는 클라이언트는 서버에게 객체 스트림으로 캡슐화된 메시지를 전송하면 서버도 동일한 방식을 사용하여 객체 스트림으로 캡슐화된 메시지로 응답한다. 이 경우 클라이언트와 서버는 모두 자신의 내부에 현재 리스트 상태의 사본을 유지하기 위해 IDList 클래스를 사용하며 RMI에 기초한 분산리스트의 구현에서는 이와 거의 비슷한 방식을 사용한다. 클라이언트와 서버는 로컬 IDList 객체에 리스트의 상태를 유지하고 있다가, 클라이언트의 리스트가 갱신되면 RMI를 사용하여 중앙 서버에 자신이 변경되었다는 것을 통지하며 만약 중앙 서버가 이 변경을 승인한다면 다시 RMI를 사용하여 등록된 모든 클라이언트에게 이 변경을 통지한다. 이 방식은 이전 소켓 방식에서 객체 스트림 메시지만을 RMI를 사용하여 바꾸어 주면 된다.

여기서 알아두어야 할 가장 중요한 점은 애플리케이션을 어떻게 디자인하느냐에 따라 적용할 네트워크 기술을 적절히 선택해야 한다는 것이다. RMI를 사용하는 경우 구현이 편리한 장점은 있겠지만, 높은 대역폭을 요구하는 애플리케이션이나 서로 많은 데이터를 주고받는 애플리케이션을 개발하기는 힘들 것이다. 물론 RMI의 장점을 충분히 살릴 수 있는 애플리케이션도 많다. RMI를 사용하면, 두 객체간의 네트워크 연결을 생성하기 위해서 원격 참조자 상에서 메소드를 호출해 주기만 하면 된다. 이처럼 RMI를 사용한 네트워크 연결이 쉽기 때문에 복잡하고 동적인 네트워크 애플리케이션을 훨씬 쉽게 작성할 수 있다. 가장 중요한 것은 어떤 애플리케이션을 작성하느냐에 따라 네트워크 프로그래밍에 사용되는 스타일이 결정된다는 것이다[4]. 스트림과 소켓에 기반한 네트워크 시스템의 경우 더 효율적이고 세세한 제어를 가능하게 해줄 것이며 RMI 프레임워크를 사용하는 경우에는 매우 쉽고 강력한 기능들이 제공될 것이다.

2. 소켓 기반 분산리스트

DistributedList는 리스트 자료구조가 제공하는 기본적인 인터페이스를 갖고 있는데 이것을 기반으로 여러 가지의 네트워크 프로토콜에 대한 분산 환경에 맞는 코드를 구현한다. DistributedList 인터페이스에는 이 리스트 자료구조의 엘리먼트를 액세스할 수 있는 여러 가지 메소드가 선언되어 있다. DistributedList에 추가되는 모든 요소는 직렬화가 가능한 객체형이라는 점이 매우 중요하다.

ChangeListener 인터페이스는 DistributedList의 변경을 알리는 객체의 골격을 나타내는 것으로 changeOccurred() 메소드는 해당 자료구조가 변경될 때마다 DistributedList의 listener에게 호출되며 실제

의 변경내용은 ChangeEvent 타입의 객체를 통해 오고 간다.

ChangeEvent 클래스는 DistributedList에 대한 변경을 알리는 이벤트를 나타내는 인터페이스이다. 변경 이벤트를 발생시키는 소스가 되는 DistributedList 객체를 이 클래스의 생성자가 받아서 바로 수퍼 클래스의 생성자로 넘겨주어 내부적으로 저장해 둔다. 상속받은 getSource() 메소드를 나중에 사용하여 변경 이벤트가 어디서 발생되었는지를 알 수 있게 된다.

ID 클래스는 객체 식별자를 나타내기 위한 클래스로써 이 식별자는 정수형으로 저장되는데 IDList 클래스에서 이것을 사용한다. ID는 아주 단순한 직렬화가 가능한 객체이며, 생성자는 정수 식별자를 매개변수로 받아 변수 id에 저장한다. IDList 클래스는 DistributedList와 비슷하게 API를 제공하는 비분산 환경의 자료구조이지만 이 자료구조의 요소에 할당된 식별자를 알려 줄 수 있다는 점에서 중요하다. 이 클래스가 바로 분산리스트 구현의 시작이라고 할 수 있다. IDList는 직렬화가 가능하도록 구현된 클래스이기 때문에 쉽게 내보낼 수 있으며, 미리 할당된 객체 식별자를 매개변수로 받아들이는 것을 제외하면 Distributed List 인터페이스의 비슷한 메소드를 가지고 있다. 또한 이들 메소드와 함께 사용할 목적으로 새로운 식별자를 만들어 내는 allocateID()를 추가하였다.

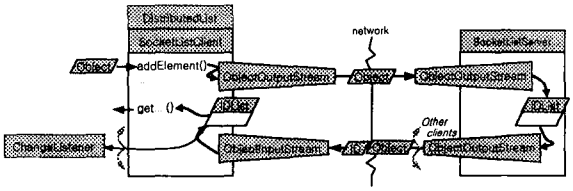

그림 1. 소켓 기반의 분산리스트 구조

2.1 분산리스트 프로토콜

그림 2와 같이 분산리스트에서 사용되는 프로토콜은 비교적 단순하지만 강력하다. IDListMsg 클래스는 클래스와 서버 사이에 타입 안정성을 지닌 메시지 포장을 가능하게 해준다. 이 클래스도 직렬화가 가능하도록 구현되었기 때문에 모든 메시지를 객체스트림에 의해 안전하게 보낼 수 있다.

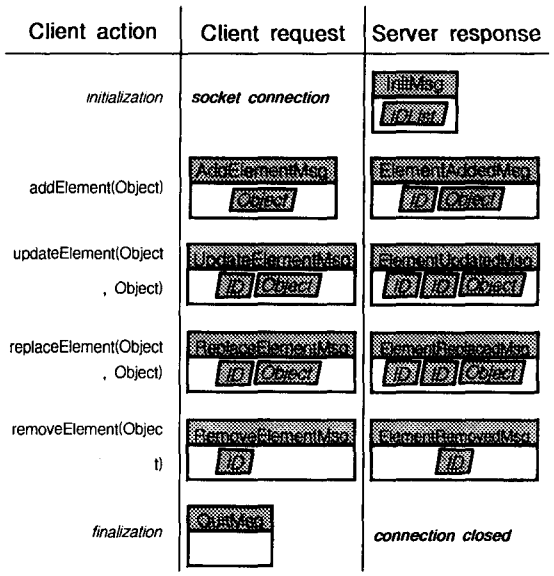

그림 2. 소켓 기반의 분산리스트 프로토콜

클라이언트가 서버에 처음 연결하게 되면 IDList의 내용을 담고 있는 InitMsg 타입의 메시지를 받게 되는데 이 메시지는 getIDList() 메소드로 추출해낼 수 있다. 만약 클라이언트가 분산리스트에다가 요소 하나를 추가하려 할 때는 AddElementMsg에다가 추가할 요소를 담아 서버로 보내면 된다. 요소 추가 명령이 받아들여지면 서버는 모든 클라이언트에게 ElementAddedMsg를 보내는데 이 메시지에 새로 추가된 요소와 할당된 식별자가 담긴다.

2.2 SocketListServer 클래스

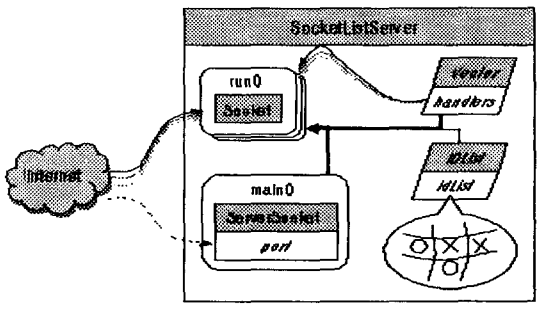

그림 3. SocketListServer 클래스

SocketListServer 클래스는 소켓 기반의 분산리스트 애플리케이션 구조에서 서버측을 담당한다. 전체 리스트 상태의 마스터 사본이 바로 이 서버 내에 저장되며 메인 쓰레드는 ServerSocket을 열고 클라이언트 연결을 기다린다. 클라이언트가 연결을 시도하면 이 클라이언트의 모든 요청을 처리하는 핸들러 객체를 새로 만들어내는데 이 경우 핸들러는 리스트의 마스터 사본을 보낸 후에 루프를 돌면서 클라이언트의 요청을 읽고 처리해 간다. 클라이언트가 리스트 변경을 요청하면 핸들러는 중앙 서버에 있는 IDList에 대한 변경을 시도하여 성공하면 연결된 모든 클라이언트에게 변경 사실이 전송된다. 그러므로 클라이언트는 리스트 변경을 요청할지라도 자기 자신이 가지고 있는 IDList를 곧바로 바꿀 수 없으며 그 대신 서버가 변경 사실을 통보할 때까지 기다려야 한다.

2.3 SocketListClient 클래스

SocketListClient 클래스는 DistributedList 인터페이스를 구현한 것으로 SocketListServer와 데이터를 주고받으며 분산리스트 자료구조를 제공하고 조작한다. SocketListClient는 중앙 서버인 SocketListServer에서 관리되는 IDList 객체의 상태를 반영하는 자체 IDList 객체를 가지고 있어 클라이언트가 SocketList Client에 변경을 요청하면 이것이 중앙 서버에 대한 요청으로 전송되며 서버는 연결된 모든 클라이언트에게 변경 메시지를 전송한다. SocketListClient는 이 변경 메시지를 받아 자체의 IDList 객체로부터 바뀐 상세 정보를 얻어내게 된다.

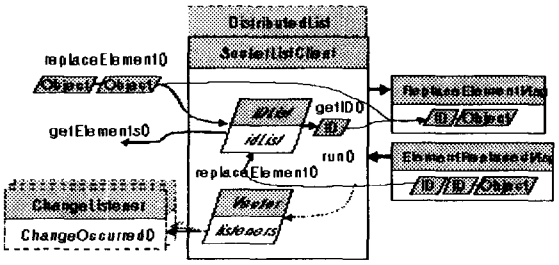

그림 4. SocketListClient 클래스

간단한 화이트보드 시스템의 구현에 분산리스트 클래스를 사용함으로써 화이트보드에 필요한 모든 자료구조를 분산리스트에 저장하고 추가 혹은 교체를 자유롭게 할 수 있었다. 중요한 것은 이 자료구조를 분산되지 않은 것처럼 사용했다는 것이다. 이 리스트는 투명성을 가지고 분산되었기 때문에 동일한 서버에 여러 개의 화이트보드를 연결했다 하더라도 애플리케이션 수준에서는 거의 아무런 노력을 들이지 않고도 협력형 애플리케이션을 구축할 수 있게 되었다.

3. RMI 기반 분산리스트

소켓 기반 시스템에서 리스트를 갱신하는 클라이언트는 서버에게 객체 스트림으로 캡슐화된 메시지를 전송하면 서버는 마찬가지 방식으로 객체 스트림으로 캡슐화된 메시지로 응답한다. 클라이언트와 서버는 모두 자신의 내부에 현재 리스트 상태의 사본을 유지하기 위해 IDList 클래스를 사용하는데 RMI에 기초한 분산리스트의 구현에서는 이와 거의 비슷한 방식을 사용한다. 클라이언트의 리스트가 갱신되면 RMI를 사용하여 중앙 서버에 자신이 변경되었다는 것을 통지한다. 이 방식에서는 이전 소켓 방식에서 객체 스트림 메시지만을 RMI를 사용하여 바꾸어 주면 된다.

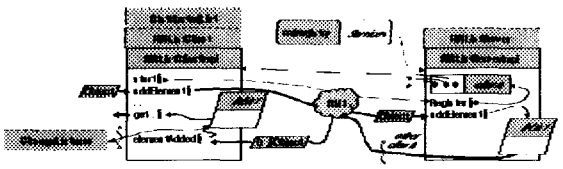

그림 5. RMI에 기반한 분산리스트

이 시스템은 4개의 클래스로 구성되어 있다. RMIListServer는 중앙 서버를 기술하는 원격 인터페이스로 클라이언트가 리스트의 변경을 요구할 때는 이 인터페이스를 사용하여 서버에 접근한다. RMIList Server는 중앙 서버를 기술하는 원격 인터페이스로써 클라이언트가 리스트의 변경을 요구할 때는 이 인터페이스를 사용하여 서버에 접근한다. RMIListClient는 클라이언트 리스트를 기술하는 원격 인터페이스로 서버가 클라이언트에게 리스트가 변경된 것을 통지하는 콜백 호출을 할 때 이 인터페이스를 사용한다. RMI ListServerImpl은 중앙 서버를 구현하는 클래스이고 RMIListClientImpl은 클라이언트를 구현하는 클래스

이다. 이 클래스는 DistributedList 인터페이스를 구현하며 RMI 기반 분산리스트의 클라이언트측 기본 인터페이스가 된다.

3.1 RMIListServer, RMIListClient 인터페이스

RMIListServer 인터페이스는 클라이언트가 RMI에 기반한 중앙 리스트 서버에 접근하기 위해 사용되는 API들을 기술한다. 이 원격 인터페이스는 클라이언트가 서버에 접근할 때 사용하는 인터페이스를 정의한다. register()와 deregister() 메소드는 클라이언트가 서버에 등록하거나 등록을 해제하는데 사용하는 것으로 클라이언트가 서버에 처음 등록하면 서버는 중앙 리스트의 사본을 만들어서 반환해 준다. 또한 RMIListClient 인터페이스는 서버가 RMI에 기반한 리스트 클라이언트에게 콜백을 할 수 있도록 해주는 원격 API를 기술한다.

3.2 RMIListServerImpl 클래스

이 클래스는 분산리스트의 서버측을 구현하는 것으로 분산되어 있는 리스트 구조의 중심에 해당하는 원격 객체이며 원격 클라이언트를 등록하고 리스트 변경이 일어났을 때 그것을 통지해 주는 역할을 해준다. 중앙 데이터 구조의 변경은 RMIListServer 원격 인터페이스에 정의된 메소드들을 통해서 일어나게 될 것이다. 생성자는 분산리스트의 중앙 원본인 IDList와 현재 등록되어 있는 원격 클라이언트들을 저장하고 있는 벡터를 생성한다. IDList는 간단한 비분산형 리스트로 이 클래스는 리스트의 모든 원소들과 연관된 식별자들을 가지고 있다. 이 식별자들은 여러 개의 클라이언트가 클라이언트 변수에 동시에 접근하는 경우나, 네트워크 접속이 해제되는 경우에도 클라이언트 변수가 무결성을 잃지 않도록 해준다.

3.3 RMIListClientImpl 클래스

이 클래스는 RMI에 기반한 클라이언트측 분산리스트 클래스이다. 이 클래스는 DistributedList를 구현하며 인터페이스는 분산리스트 구조를 구현하기 위해 RMIListServer와 통신하는데 사용된다. 이 클라이언트는 서버 측에서 중앙 데이터 리스트가 변경된 것을 클라이언트에게 콜백을 사용하여 알려 줄 수 있도록 내부적으로 RMIListClient 원격 인터페이스를 구현하는데 원격에서 접근 가능한 객체를 사용한다. 또한 이 클래스에서는 중앙 서버의 위치를 알아내고 그곳에 등록하는 작업을 해주는 start() 메소드를 제공해준다. 클라이언트가 중앙 서버에 등록되면 중앙 서버의 데이터가 변경되었을 때 서버로부터 데이터가 변경되었음을 통지 받을 수 있다. 이와는 반대로 stop() 메소드는 서버로부터 등록을 해제하고 더 이상 데이터 변경을 통지 받지 않도록 한다.

4. 비교, 분석 및 최적화 모델링

RMI를 사용할 때의 가장 큰 장점은 애플리케이션 레벨 프로토콜을 사용하지 않아도 프로그램을 간단하게 구현할 수 있다는 것이다. 여기서는 오직 메소드 호출이라는 개념 하나만을 가지고 전체 시스템을 구현하였고 가상 머시인들 사이의 네트워크 프로토콜은 RMI 프레임워크에서 모두 처리해 주었다.

그러나 소켓 기반에서는 단지 메소드 호출이라는 개념과는 달리 여러 개념들이 구현되어져야 하며 Thread, Synchronization, Socket, Server Socket, 클라이언트의 모든 요청을 처리하는 Handler 등 많은 개념의 구현이 선행되어져야 한다. 이러한 개념들은 프로그램의 디자인에서 구현 및 테스팅에 이르기까지 복잡도를 증가시킨다. Thread에 관련된 메소드는 start(), stop(), run()이고 Synchronization에 관련된 메소드는 start(), stop()이다. 따라서 Thread와 Synchronization에 관련된 테스팅은 기본적인 테스트 5가지와 통합 테스트 2×3×3=18 가지, 총 23 가지를 테스팅한다. 테스팅 측면에서 보면 관련된 메소드 개수의 제곱이 되지만 관련된 변수의 수까지 고려하면 전체 시스템 측면에서 볼 때 복잡도는 기하급수적으로 증가한다. 소켓 기반과 RMI 기반 분산리스트 클라이언트/서버 애플리케이션을 메소드 수와 테스팅 측면을 고려한 표 1을 보면 쉽게 비교가 된다.

표 1. 소켓 기반 분산리스트 애플리케이션

클 래 스	메소드	변수	관련된 다른 클래스 메소드	Testing cases
DistributedList	7	0	0	0
ChangeListener	1	0	0	0
ChangeEvent	1	0	0	1
ID	3	1	0	3
IDList	9	3	3	27
IDListMsg	0	0	0	0
InitMsg	2	1	0	2
AddElementMsg	2	1	0	2
ElementAddedMsg	2	1	3	6
UpdateElementMsg	3	2	3	9
ElementUpdatedMsg	2	1	3	6
ReplaceElementMsg	3	2	3	9
ElementReplacedMsg	2	1	3	6
RemoveElementMsg	2	1	3	6
ElementRemovedMsg	1	0	0	1
QuitMsg	1	0	0	0
SocketListServer	10	6	9	120
SocketListClient	13	7	9	133
계	64	28	25	331

표 2. RMI 기반 분산리스트 애플리케이션

클 래 스	메소드	변수	관련된 다른 클래스 메소드	Testing cases
DistributedList	7	0	0	0
ChangeListener	1	0	0	0
ChangeEvent	1	0	0	1
ID	3	1	0	3
IDList	9	3	3	27
RMIListServer	6	0	0	0
RMIListClient	4	0	0	0
RMIListServerImpl	7	2	3	42
RMIListClientImpl	14	5	3	42
계	52	11	9	115

표 1과 표 2에서 알 수 있듯이 클래스, 메소드, 변수, 다른 클래스 메소드와의 관계, 테스팅 수 등 모든 비교 인자 측면에서 볼 때 RMI 기반 분산리스트 애플리케이션이 소켓 기반 분산리스트보다 훨씬 간단하고 다지인하기 쉽고 구현에서 유지보수에 이르기까지 훨씬 더 효율적이다[5]. 전체적으로 외형적인 비교 인자만 가지고 볼 때 RMI 기반이 소켓 기반보다 약 3배의 효율성을 가지고 있다고 볼 수 있다. 그러나 프로그램 효율성 측면에서 살펴본다면 RMI를 사용한 애플리케이션은 매 업데이트 시마다 생성되는 많은 TCP/IP 연결로 인해서 매우 성능이 떨어질 수 있다. 또한 TCP/IP 연결은 RMI에 비해 매우 적은 비용을 요구하며 비록 RMI가 하나의 네트워크 연결을 사용해서 여러 메소드 호출을 처리해 주는 메카니즘을 가지고 있기는 하지만 이 경우 역시 직접 소켓을 사용하는 것보다는 효율이 떨어진다[6]. 더욱이 서버에서 클라이언트에게 콜백을 해줄 때 생성되는 네트워크 연결은 네트워크 설정에 의해서 금지되는 경우가 많아 서버에 접속하는 클라이언트가 많으면 많을수록 RMI 기반 통신 효율성은 급격하게 떨어진다.

소켓 기반에서 broadcast() 메소드를 사용하여 통신의 효율성을 높이는 점을 착안하여 RMI 기반에서도 broadcast() 메소드를 RMI 기반에 적용할 수 있도록 응용하여 상호보완적으로 구현하면 소켓 기반과 RMI 기반의 장점들만을 취할 수 있어 전반적인 시스템 향상을 가져올 수 있다. 즉, 개념화와 디자인 및 구현, 테스팅을 거쳐 유지보수에 이르기까지 RMI 기반 애플리케이션의 효율성을 유지하면서 broadcast() 기법을 RMI 기반에 적용하여 TCP/IP 효율성에 가까운 전체적인 시스템 향상을 가져올 수 있다는 것이다[7,8].

표 3. 소켓 기반 분산리스트 통신 효율성

클라이언트 접속수	지연시간 (ms)	효율성(%)
100	3.00	88.5
200	3.51	88.2
300	4.23	87.7
400	5.12	87.2
500	5.88	86.7
600	7.01	86.0
700	8.23	85.1
800	9.84	83.9
900	12.10	82.8
1000	14.86	81.3

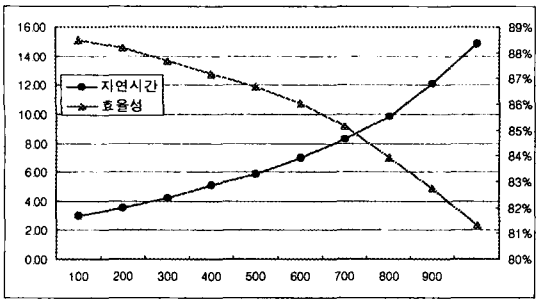

그림 6. 소켓 기반 분산리스트 통신 효율성 그래프

표 4. RMI 기반 분산리스트 통신 효율성

클라이언트 접속수	지연시간 (ms)	효율성(%)
50	3.00	88.3
100	3.32	88.0
150	3.91	87.5
200	4.49	86.2
250	5.73	84.7
300	9.45	81.7
350	15.52	77.9
400	22.34	74.1
450	29.71	70.3
500	39.19	66.1

소켓 기반과 RMI 기반 분산리스트 환경 하에서 각각 네트워크 프로그래밍을 이용하고 가상 클라이언트 수를 매개변수로 하여 서버에 접속시켜 모의실험을 해본 결과 <표 3>과 <표 4>와 같은 데이터를 얻을 수 있었다.

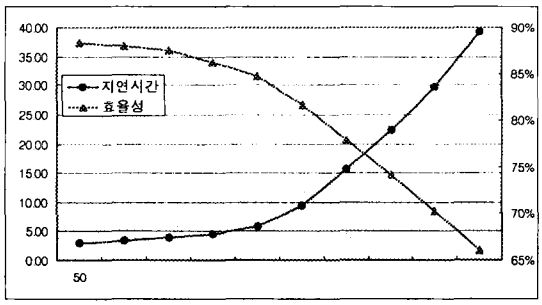

그림 7. RMI 기반 분산리스트 통신 효율성 그래프

표 5. 최적화 분산리스트 통신 효율성

클라이언트 접속수	지연시간 (ms)	효율성(%)
100	3.00	88.5
200	3.52	87.9
300	4.26	87.1
400	5.20	85.9
500	6.02	84.7
600	7.23	83.4
700	8.58	81.7
800	10.37	79.6
900	12.86	77.9
1000	16.25	74.4

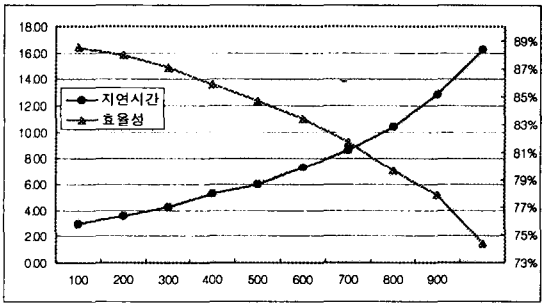

그림 8. 최적화 통합 분산리스트 통신 효율성 그래프

그림 6과 그림 7을 통해서 알 수 있듯이 소켓 기반 테스트에서는 클라이언트 접속 수가 증가함에 따라 지연시간 및 효율성은 아주 완만한 곡선을 그리고 있고 1000개의 클라이언트가 접속해도 80% 이상의 효율성을 유지하고 있다. 실험 결과 1500개 이상의 클라이언트가 접속 했을 경우에 급경사를 이루어졌으며 소켓 기반에서는 1500개의 클라이언트가 변곡점을 형성한 반면에 RMI 기반 분산리스트 환경에서는 300개 클라이언트 이후부터 급경사를 이루어 소켓 기반에 비하여 1200개의 변곡점의 차이를 나타냈고 효율성도 겨우 500개의 클라이언트에서 66%의 효율성을 나타냈다.

소켓 기반의 broadcast() 메소드를 모방한 TCP/IP 통신의 효율성에 근접한 최적화 통합 분산리스트 효율성은 소켓 기반에 비하여 그다지 큰 차이를 보이지 않고 있다. [그림 8]의 최적화 통합 분산리스트의 변곡점

은 소켓 기반과 거의 유사하였고 1000개의 클라이언트의 효율성도 74% 이상을 유지하고 있어 500개의 클라이언트에서 겨우 66%을 유지하고 있는 RMI 기반 분산리스트에 비해 전체적인 시스템성능이 전반적으로 향상되었다.

5. 결론

RMI를 사용한 경우와 여러 애플리케이션 사이에서 소켓을 사용하여 통신하는 경우를 비교해서 분석해 본 결과 애플리케이션 내부의 통신의 효율성은 소켓 기반이 월등히 우수한 반면에 시스템 디자인, 구현 및 테스팅에서는 RMI가 간단하면서도 접근하기가 훨씬 용이했다[9]. 따라서 두 시스템간의 장점만을 고려하여 통합 시스템을 구축한 결과 최적화 모델링을 구현할 수 있었고 따라서 전체적으로 20%의 시스템 향상을 가져올 수 있었다.

향후 연구 과제는 1500개의 클라이언트의 변곡점의 한계를 극복하기에는 많은 문제점을 가지고 있어 좀더 발전적인 시스템을 개발하기 위해서는 이러한 변곡점의 수치를 훨씬 더 크게 늘려 가야하며 이를 위해 좀더 통신의 하부 구조부터 상위 애플리케이션에 이르기까지 더 많은 연구가 선행되어야 할 것이라 사료된다.

참고 문헌

[1] M.R. Cutkosky, R.S. Engelmore, R.E. Fikes, M.R. Genesereth, T.R. Gruber, W.S. Mark, J.M. Tenenbaum, and J.C. Weber. PACT: An experiment in integrating concurrent engineering systems. IEEE Computer, January 1993:28-38, 1993.

[2] T. Finin, D. McKay, R. Fritzson, and R. McEntire. KQML: An information and knowledge exchange protocol. In K. Fuchi and T. Yokoi, editors, Knowledge Building and Knowledge Sharing. Ohm and IOS Press, 1994.

[3] M. Genesereth. Transmission Systems for Communication, IEEE Communications Magazine, January 1997.

[4] M. Genesereth and R.E. Fikes. Data Link Control:The Great Variety Calls for Wise and Careful Choices, Data Comm., June 1992.

[5] T.G. Gruber. A Dynamic Packet Switching System for Satellite Broadcast Channels, Proceedings of ICC, 1992.

[6] T.G. Gruber. Forward Error Correction Schemes for Digital Communications, IEEE Communication Magazine, April, 1993.

[7] R.V. Guha. Representation of defaults in Cyc. In Proc. AAAI-90, pages 608-614, 1990.

[8] R.V. Guha. Contests: Architecture and Design of a Reliable Token-Ring Network. IEEE Journal on Selected Areas in Communications, November 1991.

[9] R.V. Guha and D.B. Lenat. Connectionless Data Transmission. Computer Communications Review, November 1990.

임 동 주(Dong-Ju Im)

1985년 전남대학교 영문학과 졸업(문학사)

1993년 미국뉴욕주립대학교 대학원 전산학과(이학석사)

1994년~1999년 대불대학교 근무

1999년 조선대학교 대학원 전산통계학과(이학박사)

※주관심분야 컴퓨터네트워크, 멀티미디어, 소프트웨어엔지니어링, 데이터베이스 등