인터셉터를 이용한 CORBA 부하분산 기법

김중한[1]°, 라마크리쉬나[1], 구경이[2], 김홍식[2], 김유성[2]
[1] 광주과기원 정보통신공학과, [2] 인하대학교 전자계산공학과
[1]{s911669,rsr}@banana.kjist.ac.kr,
[2]{g9721046,g1991279}@inhavision.inha.ac.kr, yskim@inha.ac.kr

Load Distribution Mechanism in CORBA Using Interceptor

Joong-Han Kim[1]°, R.S.Ramakrishna[1], Kyung-I Ku[2], Hong-Sik Kim[2], Yoo-Sung Kim[2]
[1] DIC, KJIST, [2] Dept. of Computer Sci. and Eng., Inha Univ.

요 약

현재 분산객체 시스템 아키텍처의 표준으로 제정된 CORBA 명세에는 부하분산에 대한 기술 표준이 정의되어 있지 않다. CORBA 시스템의 부하분산 기능의 부재는 특정서버 객체에 대한 과부하 및 클러스터 내 특정 서버의 과부하를 초래한다. 기존에 제안된 CORBA 부하분산 기법들은 ORB 호환성 저해, 코드수정 필요성, 주기적인 부하보고에 따른 네트워크 부하가중 등의 문제점들을 가졌다. 본 논문은 기존 부하분산 기법들의 문제점을 해결하기 위해 인터셉터를 이용한 CORBA의 부하분산 아키텍처 및 구현기법을 제안한다. 제안된 CORBA 부하분산 아키텍처는 인터셉터의 이용으로 CORBA 시스템의 호환성을 유지하며 CORBA의 수행방식을 바꾸어 부하분산기능을 가진다. 제안된 시스템에서 서버의 부하보고는 CORBA의 이벤트 서비스를 이용하여 구현하여 기존 부하분산 기법의 주기적 부하보고의 문제점을 해결하였으며 부하분산시스템 자체의 부하분산 및 고장 감내성을 제공하는 장점을 가진다.

1. 서론

CORBA(Common Object Request Broker Architecture)는 OMG(Object Management Group)에 의해서 제시되고 표준이 제정된 분산객체 시스템 아키텍처이다[1]. CORBA에서 클라이언트와 서버객체 간의 통신은 ORB(Object Request Broker)라는 소프트웨어 객체통신버스를 통해서 이루어진다. CORBA는 초기부터 시스템상의 메시지 전달체계에 중점을 두어 설계되었기 때문에, 분산시스템을 구성하기 위해 필요한 여러 가지 기능이 부족했다. 그러한 문제점을 보완하기 위해 여러 CORBA 서비스들이 제안되었고 표준으로 제정되었다[2]. 그러한 서비스들에는 네이밍, 이벤트, 트랜잭션 서비스 등이 있으며 이러한 서비스들은 CORBA 시스템의 기능 및 성능 향상에 도움이 되었다. CORBA 서비스들 중에 분산시스템에서 중요한 요소의 하나인 부하분산을 위한 메커니즘은 CORBA 표준 사양에 제시되어 있지 않다. 부하분산 기능의 결여는 대용량의 사용자 요구를 적절히 서버들간에 분산시키지 못해 응답시간의 저하를 초래한다.

현재까지 여러 가지 형태의 CORBA 부하분산 기법들이 제안되어 왔다. 그러한 기법들은 CORBA ORB의 수정[3], 새로운 서비스의 제안[4]등에 의해 타 CORBA ORB와의 호환성 저해, 또는 기존 코드(서버객체 또는 클라이언트)의 수정을 초래하거나 부하분산 시스템 자체의 부하병목을 초래하는 단점[5]을 지니고 있다.

본 논문에서는 인터셉터를 이용한 CORBA 부하분산 시스템인 LODIN(LOad Distribution using INterceptor)의 아키텍처 및 구현방법을 제시한다. CORBA 인터셉터는 클라이언트와 서버간의 호출경로 상에서의 메시지나 사용자요구를 중간에 가로채서 적절한 연산을 수행하여 클라이언트나 서버 양측에 투명한 기능을 제공할 수 있는 소프트웨어 모듈이다[1]. CORBA 인터 셉터는 보안, QoS, 모니터링, 시각화 등의 다양한 분야에서 CORBA의 성능 향상을 위해 사용되어 왔다[6]. 본 논문에서 구현된 LODIN은 LBINT(Load Balancing INTerceptor)를 이용하여 클라이언트 및 서버객체에 투명한 부하분산을 제공한다. 서버의 부하보고는 CORBA 이벤트 서비스[2]를 이용하여 구현하였다. 기존 부하분산 기법들은 부하보고를 주기적으로 수행하여 네트워크 부하를 가중시킨 단점을 지니고 있다. 그러나, 본 논문에서 제안한 LODIN은 이벤트 서비스를 통한 부하보고채널의 구현으로 서버의 부하가 특정 임계값 이상의 변화 발생시만 Load Estimator가 Load Analyzer로 부하보고를 하여 네트워크 부하를 감소시키도록 구현되었다.

2. 관련연구

CORBA 환경에서의 응용은 요구를 보내는 클라이언트와 그러한 요구에 응답하는 서버객체로 구성되어있다. 클라이언트와 서버간의 통신은 CORBA의 ORB가 담당한다. CORBA에 부하분산 기능을 삽입하는 것은 이러한 CORBA 기본 메커니즘을 토대로 여러 방식으로 구현되어 왔으며 기존에 제안된 CORBA 부하분산 기법은 다음과 같은 세 기법으로 나눌 수 있다.

• 부하분산 CORBA 서비스 구현 방안

부하분산 CORBA 서비스 구현 방안은 부하분산 기능을 가지는 CORBA 서비스를 구현한 후, 타 CORBA 서비스와 마찬가지로 클라이언트가 그 서비스를 이용할 수 있도록 하는 기법이다[3]. 이러한 서비스를 이용하여 클라이언트가 서버 객체를 사용하는 것은 부하분산 효과를 보장하며 타 ORB에서도 사용 가능한 장점을 지니고 있지만, 새로운 서비스를 사용하기 위해 클라이언트의 코드를 수정해야 하는 단점이 있다.

• **ORB에 부하분산 기능 통합 방안**
이 방법은 ORB 내부에 부하분산 기능을 통합하는 기법이다[4]. 이 기법은 클라이언트의 요구가 부하가 적은 서버의 객체에 투명하게 전달되는 것을 보장하며, 클라이언트 및 서버객체의 수정이 필요가 없으나, 여러 다른 ORB가 이용되어지는 경우 타 ORB들간의 호환성 결여로 부하분산 기능은 제대로 작동하지 않는다.

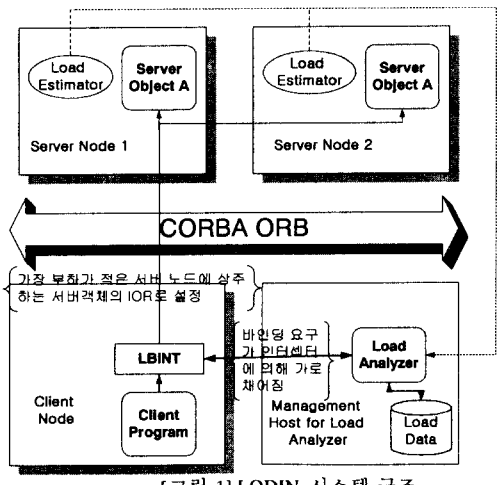
• **부하분산 기능이 추가된 CORBA 네이밍 서비스**
본 방안은 CORBA의 표준 서비스인 네이밍 서비스가 서버들의 부하를 관리하는 부하정보서버와 통신을 하여 클라이언트의 네이밍 리솔루션(Naming Resolution) 요구 시 가장 부하가 적은 노드의 서버객체의 참조자를 반환한다[5]. 이러한 기법은 개발된 부하분산 네이밍 서비스를 채택한 ORB들간의 호환성도 유지할 수 있도록 한다. 본 기법을 사용한 시스템은 부하정보서버가 고장 감내성을 갖지 않고 단일한 부하정보서버에 또 다른 부하병목 현상을 초래한다.

상기의 3가지 부하분산 기법은 각각의 단점을 가지고 있다. 또한, 노드부하의 변화에 상관없이 부하보고를 주기적으로 하여 네트워크 및 부하분석 프로세스에 새로운 부하를 가중시키는 공통의 단점도 가진다.

3. LODIN 부하분산 시스템 구조

본 논문에서 구현된 LODIN 시스템은 CORBA의 인터셉터와 이벤트 서비스를 이용해 부하 분산 문제를 해결하여 기존 부하분산 기법들의 단점을 개선하였다. LODIN은 부하분산을 위해 상호 유기적으로 연결되어 수행하는 LBINT, Load Estimator, Load Analyzer로 구성되어 있다([그림 1] 참조).

• LBINT
클라이언트로부터 요청된 서버 객체로의 바인딩 연산을 가로챈 후, Load Analyzer로부터 최소 부하서버의 IP 주소 및 해당 서버 객체의 포트를 얻어 IOR을 수정한다.
• Load Estimator
상주하고 있는 서버노드의 부하를 Load Analyzer에게 보고한다.
• Load Analyzer
Load Estimator로부터 노드 부하정보를 얻어온다. 얻어온 부하정보를 가지고 가장 부하가 적은 노드를 결정하여 LBINT의 요구 시 반환한다.



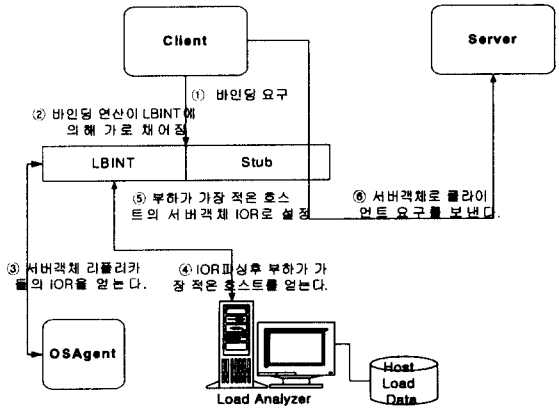
[그림 1] LODIN 시스템 구조

4. LODIN 시스템 구성요소 설계 및 구현방안

LODIN 시스템의 기본 구조에 따른 시스템 구성요소의 설계 및 구현방법은 다음과 같다.

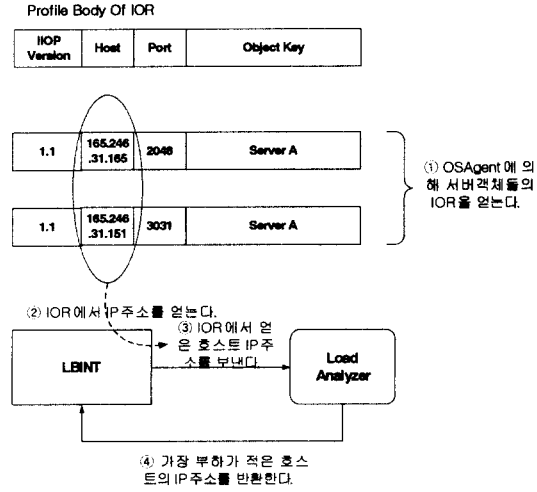
4.1 LBINT 설계 및 구현

인터셉터는 호출경로에 삽입될 수 있는 선택적인 CORBA 확장 요소로서 여러 분야에서 적용되어 왔다. LBINT는 부하분산을 위해 개발된 인터셉터이며 Visibroker for Java의 Bind Interceptor[7,8]를 이용하여 구현하였으며, 클라이언트 구동 시 함께 로드된다. 클라이언트가 서버객체에 대한 참조자를 얻는 연산을 바인딩 연산이라 하며 이 연산이 클라이언트로부터 요구될 때마다 LBINT에 의해 부하분산을 위한 여러 가지 연산들이 수행되도록 제어된다[그림 2].



[그림 2] LBINT를 이용한 바인딩 연산 수행과정

부하분산을 위한 코드는 LBINT의 Bind()함수에 삽입되었으며, 삽입된 코드는 바인딩 흐름을 제어하여 가장 부하가 적은 노드에 있는 서버객체의 IOR을 클라이언트가 얻을 수 있도록 한다[그림 3]. LBINT의 수행과정은 다음과 같다.



[그림 3] 부하분산을 위한 LBINT 내부연산

첫째, 클라이언트의 서버객체에 대한 Bind()호출만이 관심대상이기 때문에 바인딩 연산에서 고려할 필요가 없는 호출은 걸러낸다.

둘째, 클라이언트로부터 전송된 객체 리포지토리 ID 및 이름과 일치하는 모든 서버 객체들의 IOR들을 얻는다.

마지막으로, 얻어진 스트링화된 IOR들을 바이트배열 형태로 바꾼 후 파싱을 통해 IOR 내부에 포함된 서버객체의 IP 주소를 얻어 그들중에서 가장 부하가 적은 노드의 호스트를 Load Analyzer에 의해 결정한 후, IOR 을 그 노드의 서버객체의 IOR로 바꾼다. 이후 클라이언트는 가장 적은 부하를 가지는 노드의 서버객체를 호출하게 된다.

LBINT는 Load Analyzer 및 디렉터리 서비스 데몬(OSAgent)[7]과 상호 작용을 수행한다. 부하분산 전략의 수정을 원한다면 클라이언트 및 서버객체에 수정 없이 인터셉터 내에 원하는 코드를 삽입한다. 예를 들면, 사용자의 특권에 맞추어 부하 값이 특정치 이상이거나 이하인 노드를 선택하도록 할 수 있다.

4.2 Load Estimator와 Load Analyzer의 설계 및 구현

Load Estimator와 Load Analyzer는 데몬 프로세스로서 각각의 역할을 수행하며 통신채널은 이벤트 서비스로 구성된다. 실험에서 사용된 서버들은 인텔 펜티엄 333MHz를 탑재한 Windows 시스템이며, 시스템 의존적인 연산을 위해 C++언어 및 Visibroker for C++[9]를 Java언어와 함께 사용하여 구현하였다.

• Load Estimator

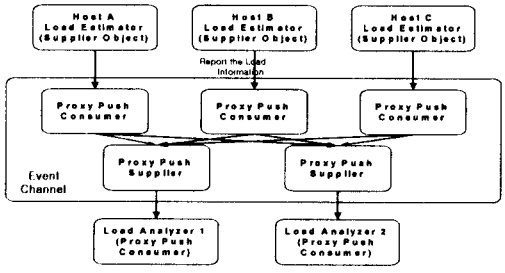
본 논문에서 제안된 LODIN 시스템 내의 각 노드의 Load Estimator는 부하정보를 Load Analyzer에게 보고한다. 부하 정보는 가용 메모리 크기 및 CPU 이용도이다. 가용메모리 크기는 Win32 API인 GlobalMemoryStatus[10] 함수를 사용하여 메가 바이트 단위로 얻는다. CPU 이용도는 백분율로 나타내어지며 Windows Platform SDK[11]의 Performance Data Helper 유틸리티를 이용하여 측정한다.

• Load Analyzer

본 논문에서 제안된 LODIN 시스템 내의 Load Estimator로부터 전송된 부하 정보들을 가지고 부하가 가장 적은 서버 노드를 판단한다. 우선, CPU 이용도가 가장 낮은 노드를 부하가 가장 적은 노드로 결정한다. CPU 이용도가 동일한 경우 가용 메모리가 가장 많은 노드로 결정한다.

• Load Estimator 와 Load Analyzer의 통신설계 및 구현

CORBA 이벤트 서비스를 이용하여 Load Estimator가 Load Analyzer에게 부하보고를 하도록 한다[그림 4]. 이벤트 서비스 중에서 푸시 모델을 사용하여 CPU 이용도가 특정 임계치 이상 변한 경우만 보고한다.실험에서는 10%로 설정하였다. 이러한 설계는 기존의 주기적 보고방식이 부하 값의 변화가 없거나 극히 작은 경우에도 부하정보를 보고하던 단점을 보완한다.



[그림 4] 이벤트 서비스기반 Load Estimator 와 Load Analyzer의 통신구조

하나의 Load Analyzer는 그 자체도 부하병목을 야기시키거나 Load Analyzer의 작동정지 시 부하분산기능이 정지하는 단점을 지닌다. 이벤트 서비스를 이용하는 것은 Load Analyzer가 중복되는 것을 허용하여 고장 감내성을 갖게 하며 라운드 로빈(Round-Robin) 방식으로 Load Analyzer의 부하를 분산시키는 구조를 갖게 한다. 이러한 구조는 클라이언트나 서버객체 또는 LBINT, Load Estimator등의 시스템 요소에게 투명하게 구성되어진다.

5. 결론 및 향후 연구과제

CORBA는 부하분산 기능에 대한 표준의 부재로 특정 서버에 부하가 편중되는 단점을 지닌다. 기존에 제안된 부하분산 기법들은 ORB간의 호환성 저해, 기존 코드의 수정과 같은 각각의 문제점들을 내포하고 있으며, 주기적 부하보고로 네트워크 부 하를 가중시키는 공통의 단점을 지니고 있다. 본 논문에서는 인터셉터 및 CORBA 이벤트 서비스를 이용하여 CORBA 부하분산 시스템을 설계 및 구현하였다. 인터셉터의 이용은 클라이언트 및 서버객체에 투명한 부하분산 서비스를 제공하는데 사용되었으며, 이벤트 서비스는 Load Estimator와 Load Analyzer간의 부하정보 보고를 위한 통신 채널로 사용되었다. 이것은 통신부하를 감소시키는 장점이 있으며, Load Analyzer의 중복을 허용하여 Load Analyzer가 고장 감내성을 갖게 하였으며 부하병목의 문제점을 해결하였다.

본 실험에서는 동등한 성능을 가지는 서버들을 사용하여 부하를 측정하였다. 이러한 노드 부하측정은 다른 성능의 서버들로 구성된 서버 클러스터에 적용될 수 없다. 앞으로는, 각 서버 노드의 성능을 평가하기 위해 새로운 부하분산 매트릭스를 정의하고 그에 따른 구현방식에 대한 연구가 필요하다.

6. 참고 문헌

[1] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.3, Object Management Group, Framingham, Mass., June 1999.

[2] Object Management Group. CORBAservices: Common Object Services Specification, Object Management Group, Framingham, Mass., December 1998.

[3] Rackl, G., Load Distribution for Environ-ments, Diploma thesis, (http://wwwbode.Informa-tik.tumuenchen.de/.rackl/DA/da.html), University of Munich, 1997.

[4] Gebauer, C., Load Balancer LB .a CORBA component for load balancing, (http://www.vsb.cs.uni-frankfurt.de/), Diploma thesis, University of Frankfurt, 1997.

[5] T. Barth, G. Flender, B. Freisleben, F. Thilo, Load Distribution in a CORBA Environment, Proceedings of the International Symposium on Distributed Objects and Applications, 1998.

[6] Priya Narasimhan, Louise E. Moser, and P.M. Melliar-Smith, Using Interceptors to Enhance CORBA, Computer Magazine Vol. 32, p.62-68. No. 7, July 1999

[7] Inprise, Visibroker for Java Programmer's Guide Version 3.3, Inprise Corporation, 1998.

[8] Inprise, Visibroker for Java Reference, Inprise Corporation, 1998.

[9] Inprise, Visibroker for C++ Programmer's Guide Version 3.3, Inprise Corporation, 1998

[10] MSDN Library Visual Studio6.0, Windows32 API, Microsoft, 1999.

[11] MSDN Library Visual Studio6.0, Platform SDK, Microsoft, 1999.