코바 어플리케이션의 동적 부하 분산을 위한 실시간 모니터링 기법 및 메트릭스
(A Real-Time Monitoring Method and Dynamic Load-Balancing Metrics for CORBA Applications)

최 창 호 † 김 수 동 ††
(Chang Ho Choi) (Soo Dong Kim)

요 약 인터넷이 점차 보급되면서 오늘날의 대부분의 소프트웨어들이 인터넷을 기반으로 하는 분산 어플리케이션으로 변해가고 있다. 코바라는 미들웨어를 사용하여 개발하는 방식이 이러한 웹 기반 소프트웨어 개발을 쉽게 해줄 수는 있지만, 소프트웨어의 완성 단계에서 최적화된 소프트웨어의 분산을 도와줄 수 있는 성능검증 방법이 제시되지 않고 있다. 또한, 분산 시스템의 운영 단계에서 동적으로 부하를 조절하기위한 모니터링 기법이나 부하 분산을 위한 메트릭스가 제시되지 않고 있다. 본 논문에서는 코바 어플리케이션의 실행 시에 객체간의 메시지를 모니터링할 수 있는 기법과 부하 메트릭스, 부하분산을 위한 메트릭스를 제시한다. 부하를 계산하기위해 어플리케이션에서 발생하는 이벤트들과 그 이벤트들 간의 시간을 정의하여 부하와 관계 있는 데이타들을 추출한다. 추출된 데이타들로부터 부하를 계산하는 공식을 유도하고 계산된 부하들을 이용하여 부하 분산 메트릭스를 제시한다. 또한 구현사례를 통하여 제시된 모니터링 알고리즘 및 부하 메트릭스와 부하 분산 메트릭스의 적용성과 효율성을 살펴본다.

Abstract As Internet is being widely used as an infra of distributed applications, the most of today's softwares are changing into Internet-based distributed applications. The development methods using the middleware, like CORBA ORB, make the development of the web-based software easy. However, the performance verification method useful for an optimized software distribution is not provided at software development. Additionally, monitoring methods and metrics for dynamic load-balancing are not presented at run-time. This paper presents the method to monitor the message between objects, load metric, and metrics for load-balancing. To calculate a load of a node, we define events occurred between applications, time between the events, then extract the data related to a load. And we derive formula calculating the load from the extracted data. Then using the formula, we present the metrics for dynamic load-balancing. Moreover, we observe the utilization and efficiency of the monitoring algorithm, load metric, and load-balancing metrics.

1. 서 론

인터넷/인트라넷의 보급이 확산되면서 웹을 기반으로 한 어플리케이션의 개발이 확대되고 있다. 이들 어플리케이션은 기존의 클라이언트/서버 어플리케이션보다 다양한 형태로 분산되며, 웹상에서의 어플리케이션은 자바, C++, 코바[1][2], COM, DCOM, 자바 애플릿, 자바 스크립트, ActiveX, CGI, HTML등 다양한 기술을 요구한다[8].

위 기술들을 모두 사용하여 어플리케이션 개발을 할 필요는 없기 때문에, 우리는 어떤 기술들을 사용해야 할지를 선택해야 한다. 이 선택에서도 분산 미들웨어를 선택하는 것이 중요한데, 이에 따라서 개발 플랫폼 등이 달라지게 되기 때문이다. 여기에는 크게 두 가지의 선택이 있는데, 마이크로소프트의 DCOM과 OMG(Object Management Group)의 코바가 그것이다.

† 학생회원 : 숭실대학교 컴퓨터학부
chakhany@chollian.net
†† 종신회원 : 숭실대학교 컴퓨터학부 교수
sdkim@computing.soongsil.ac.kr
논문접수 : 1998년 11월 5일
심사완료 : 2000년 1월 19일

DCOM과 코바 모두 분산 어플리케이션의 미들웨어로 많이 사용되지만, 특징적으로 마이크로소프트의 윈도우와 개발툴에 익숙하고 C++을 위주로 개발하는 사람들은 DCOM을 사용하고, C++과 Java, 멀티플랫폼(Multi-Platform)을 위주로 개발하는 사람들은 주로 코바를 사용한다. Windows95,98,NT를 갖고 있는 마이크로소프트가 분산 미들웨어인 DCOM을 지원하므로 유리한 고지를 선점하고 있지만 이를 기반으로 개발된 분산 어플리케이션이 마이크로소프트라는 특정 벤더(Vendor)에 종속되는 반면에, 코바를 기반으로한 분산 어플리케이션은 거의 모든 플랫폼을 지원한다.

분산 어플리케이션에 있어서 이들 미들웨어가 기본적인 개발환경, 즉 통신 환경 및 컴파일 도구 등은 지원을 하지만, 이들이 적절히 분산되어 있는지 측정할 수 있는 도구 및 관련 연구가 부족하고 분산 어플리케이션을 관리할 수 있는 환경이 제대로 갖추어져 있지 않아서 개발자와 운영자에게 분산 어플리케이션 개발 및 운영에 큰 부담을 주고 있다.

그러므로, 본 논문에서는 코바 어플리케이션을 관심대상으로 이들을 모니터링할 수 있는 기법을 제시하고, 이를 이용하여 부하를 측정하고 측정된 부하를 이용하여 부하를 분산할 수 있는 메트릭스를 제시하려고 한다. 또한 코바 어플리케이션을 모니터링하여 부하를 측정하는 것을 자동으로 할 수 있는 도구를 구현하여 코바 어플리케이션의 성능 향상을 도모하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 성능 측정 및 동적 부하 분산에 관한 기존 연구를 살펴보고, 3장에서는 기존 인터셉터에 어떤 기능을 추가하여 CAM(CORBA Application Monitor)을 위한 인터셉터를 구현하는지를 설명한다. 4장에서는 모니터링 알고리즘을 기술하고, 5장에서는 인터셉터가 모니터링한 데이타와 동적 부하 메트릭스에 필요한 데이타를 정의하며, 이들 정의로부터 동적 부하 메트릭스와 부하 분산 메트릭스를 기술한다. 6장에서는 CAM을 평가하고, 7장에서 결론을 맺는다.

2. 기존 연구

객체 지향 시스템에서의 성능 측정에 관한 연구가 있어 왔다[9~14]. 기존의 연구에서는 클래스의 클러스터링을 위하여 객체간의 응집도와 결합도를 사용하는 기법을 사용하는데, 이는 시스템의 정적인 구조를 기반으로 하므로 성능 향상을 위하여 실시간에 그 객체를 분산하기 위한 방법은 아니다. 즉, 호출한 메시지 개수와 응답한 메시지 개수는 시간에 따라 변하고 어플리케이션을 사용하는 실제의 여러 가지 변수에 의해 달라질 수 있기 때문이다. 또한, 동적 응집도[12]와 결합도를 계산하기 위한 메트릭이 제시되어 있지만, 시스템의 성능향상을 위하여 객체의 응집도와 객체간의 결합도를 측정하여 객체들을 재배치하는 것이 하나의 방법이지만, 응집도와 결합도에서 고려되지 않는 성능 요소를 더 고려하여 실시간의 부하를 계산하고 분산하는 것이 타당하다.

또한, 병렬 처리 분야에서 동적 부하 분산에 관한 연구[15][16]가 있어 왔으나, 시스템, 부하분산 대상, 부하분산 범위와 방법이 본 논문에서 다루는 부하 분산과는 다르다. 기존 연구가 다중 프로세서 시스템에서 병렬 코드의 자동 생성[15] 등의 방법으로 프로세스간의 태스크를 효율적으로 분산 처리하는 데에 관한 연구라면, 본 논문은 여러 이기종 시스템에서 서버 객체를 이동 또는 복제, 그리고 대역폭을 조정하여 시스템의 부하를 분산시키는 데에 관한 연구이다.

코바 환경에서의 동적 부하 분산에 관한 연구는 현재 새로운 분야이다. [17]에서 객체의 이동을 통해 시스템의 성능을 향상시키려는 연구가 이루어지기는 했지만, 서버의 부하 측정을 위한 선행 작업이 필요하다. 서버의 부하가 단위 시간당 서버에 전달된 요청의 수와 크기([17]의 6.1.3참조)에 의해 계산되는데 부하 측정 전에 미리 클라이언트의 요청마다 그 크기가 계산되어 있어야 한다. 그러나, 본 논문에서는 부하 및 부하 분산 메트릭스 계산에 필요한 모든 항목을 실시간에 계산하기 위한 모니터링 방법 및 여기에서 도출한 데이타를 이용한 부하분산 메트릭스를 소개한다.

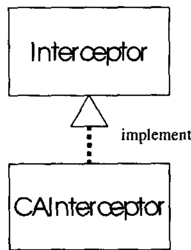
3. 인터셉터 구현 기법

3.1 ORB Core & ORB Services

코바는 OMG에서 분산 어플리케이션을 위한 미들웨어로 제안한 사양이며, ORB는 코바의 OMA구조에서 어플리케이션 간의 메시지 교환을 도와주는 중심적인 역할을 담당한다. ORB는 크게 코아 부분과 서비스 부분으로 나뉘는데, 코바 아키텍처에서 ORB Core는 객체 바인딩, 동적 호출, 정적 호출, 인터페이스 저장소 관리, ORB간 연동, 인터셉터[1] 등 객체의 기본적인 통신 기능을 제공하며, ORB Services는 이 ORB Core를 기반으로 Naming Service, Event Service, Persistent Object Service, Life Cycle Service, Concurency Control Service, Externalization Service, Relationship Service, Transation Service, Query Service, Security Service, Trading Object Service[2] 등이 만

들어진다. ORB Services는 ORB Core를 사용하는데 있어서, ORB Core의 클래스들을 상속 또는 구현하거나, 어그리게이션(Aggregation) 또는 컴포지션 (Composition) 등을 통하여 만든다.

본 논문에서 제시하는 모니터링 기법은 ORB Core의 부분인 인터셉터(Interceptor)라는 장치를 이용하여 코바의 클라이언트 어플리케이션과 서버 객체(Target Object)간의 메시지를 가로챈다. CAM은 코바 어플리케이션 간의 메시지를 가로채는 CAInterceptor를 만드는데 있어서 인터셉터의 인터페이스를 구현함으로써 만들어진다. 즉, CAInterceptor는 ORB Core를 이용하여 만든 일종의 ORB Service이다.



3.2 ORB 서비스들 & 코바 인터셉터

코바의 인터셉터는 그 인터페이스를 제공함으로써, ORB Services를 쉽게 구현할 수 있도록 해준다. 코바의 인터셉터를 이용하면 클라이언트와 서버 객체 사이의 호출 경로(Invocation Path)에 개입하여 메시지를 변경하거나 특정 이벤트가 발생할 때 특정 기능을 하도록 구현할 수 있다.

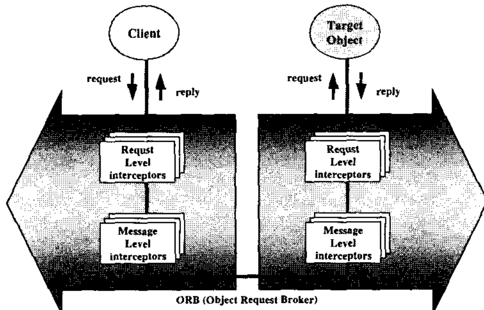


그림 1 인터셉터를 이용한 ORB Service

코바 사양에서는 요청-레벨(Request-Level) 인터셉터와 메시지-레벨(Message-Level) 인터셉터에 대한 인터페이스를 제공하는데, 요청-레벨 인터셉터는 구조화된 호출 메시지를 받아서 이를 변경할 수 있도록 하며 메시지-레벨 인터셉터는 스트림 형태의 메시지를 받아서 이를 변경할 수 있도록 한다.

그림 1은 인터셉터를 이용한 ORB 서비스들의 일반적인 메시지 흐름을 보이는데, 클라이언트와 서버 객체 사이에서 이들의 호출 경로 상에서 전달되는 요청과 응답에 개입함을 알 수 있다. 또한, 각 인터셉터들은 동등하게 클라이언트쪽과 서버 객체쪽에 위치하며, 클라이언트쪽의 인터셉터와 서버 객체쪽의 인터셉터 사이에서도 그 호출된 메시지가 전달된다.

3.3 코바 인터셉터 & 비지브로커 인터셉터

윗 절에서 우리는 코바의 인터셉터를 살펴보았다. 코바는 사양만을 갖고 그 구현이 없기 때문에 구현제품에 따라 인터페이스가 달라질 수 있다. 현재는 비지브로커에서 인터셉터 장치를 지원하고 있으므로, 이를 통하여 CAM을 구현하려 한다.

표 1 코바의 인터셉터 & 비지브로커의 인터셉터

CORBA interceptor's interfaces and methods	VisiBroker interceptor's interfaces and methods
<ClientRequestInterceptor> client_invoke client_response	<ClientInterceptor> prepare_request send_request send_request_succeeded send_request_failed receive_reply receive_reply_failed
<TargetRequestInterceptor> target_invoke target_response	<ServerInterceptor> locate locate_succeeded locate_failed locate_forwarded receive_request prepare_reply send_reply send_reply_failed request_completed shutdown
<MessageInterceptor> send_message receive_message send receive	<BindInterceptor> bind bind_succeeded bind_failed rebind rebind_succeeded rebind_failed

그러면, 비지브로커 인터셉터의 인터페이스를 살펴보자. 이 인터셉터는 코바의 인터셉터와 그 기능은 같지만 인터페이스는 조금 차이가 있다. 표 1에서와 같이 비지브로커 인터셉터는 BindInterceptor, ClientInterceptor, ServerInterceptor로 구성되며, 각각 인터셉트하는 메시지와 이벤트가 다르다. BindInterceptor는 클라이언트의 바인딩 메시지와 그 이벤트를 인터셉트하며, Client-Interceptor는 클라이언트가 요청할 때 발생하는 이벤트에 따라 그 메시지를 가로챌 수 있도록 하며, Server-Interceptor는 서버 객체가 응답할 때 발생하는 이벤트

에 맞추어 그 메시지를 인터셉트할 수 있도록 인터페이스를 정의하고 있다.

코바의 인터셉터와 비지브로커의 인터셉터는 구조적 메시지와 비구조적인 메시지 모두를 인터셉트할 수 있도록 인터페이스를 제공하는 점은 같지만, 비지브로커의 인터셉터는 클라이언트의 바인딩에 대해 별도의 인터페이스를 두고 있는데, 이는 매 호출마다 불려지는 ClientInterceptor와 ServerInterceptor와는 달리 첫 호출이 발생하기 이전에 한번 호출되는 BindInterceptor이다.

3.4 CAM을 위한 인터셉터 구현 기법

이상에서 일반적인 인터셉터의 정의 및 그 구현 방법에 대해 간단히 살펴보았다. 코바 어플리케이션을 모니터링하기 위한 인터셉터도 일반적인 인터셉터의 구현 방법과 유사하며, 특징적으로 CAInterceptor는 노드의 부하 및 노드간의 트래픽 부하를 계산할 수 있는 데이타를 추가로 인터셉트해야 하므로, 그림 2와 같이 노드의 인터넷 어드레스 및 그 이름을 얻는 부분과 호출상의 각 이벤트 간의 시간을 측정할 수 있도록 타이머를 추가해야 한다. 또한, 클라이언트와 서버 객체간의 호출 경로를 찾기 위해서 클라이언트는 자신이 실행되고 있는 노드뿐만 아니라 그 서버 객체가 실행되고 있는 노드의 정보도 함께 필요로하며, 서버 객체도 접근하는 클라이언트의 노드를 함께 인식해야 한다.

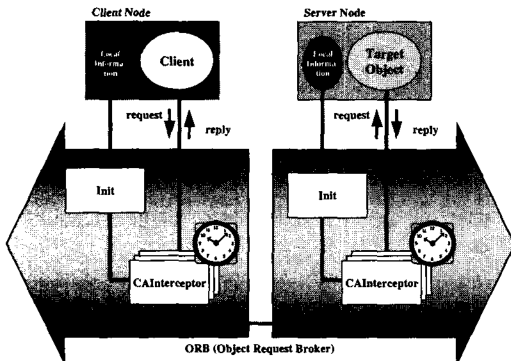


그림 2 CAM을 위한 인터셉터의 작동

그림 2를 통해 CAM을 위한 인터셉터의 작동에 대해 자세히 살펴보자. 우선 ORB가 떠있는 상태에서 서버 어플리케이션이 수행되며, 서버 어플리케이션이 서버 객체를 ORB에 등록하면, 클라이언트 객체가 서버 객체를 사용할 수 있다. 클라이언트나 서버가 실행되면서 CAInterceptor Service Init가 실행되는데, 수행하는 노드는 어플리케이션이 수행되는 동안 변동이 없다는 가정 하에 한번 만 인식하도록 하기 위해 인터셉터를 초기화하면서 자신의 노드를 인식해서 각 인터셉터에 정보를 넘겨준다. 또한, 인터셉터가 초기화되면서 BindInterceptor, ClientInterceptor, ServerInterceptor 객체를 생성한다. 생성된 각 객체는 클라이언트와 서버 객체 간에 메시지가 전달될 때 이 메시지를 가로채며 호출되는 이벤트마다 시간을 측정한다.

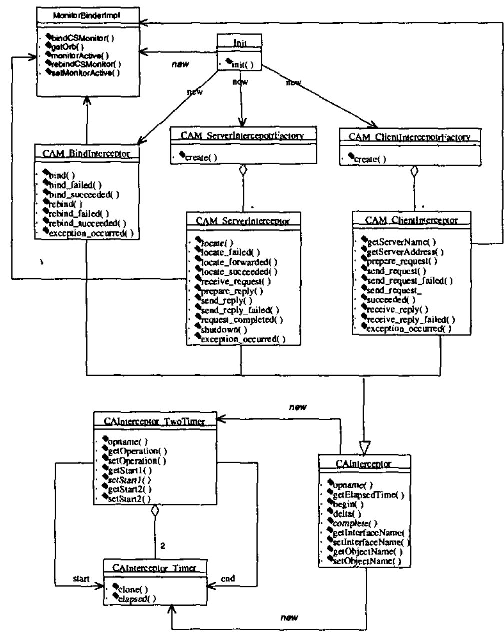


그림 3 CAInterceptor 클래스 다이어그램[5,6,7]

이제는 CAM을 위한 인터셉터를 구현하기 위한 클래스를 찾아보자. 그림 5의 클래스들은 비지브로커에서 제공하는 인터페이스를 구현하여 만든다. 비지브로커는 BindInterceptor, ClientInterceptor, ServerInterceptor 인터페이스 외에도 이들을 초기화할 때 필요한 Init 인터페이스와 ClientInterceptor의 인스턴스를 생성하는 ClientInterceptorFactory 인터페이스, ServerInterceptor의 인스턴스를 생성하는 ServerInterceptorFactory 인터페이스를 정의하고 있다.

비지브로커의 Init 인터페이스를 구현하여 Init 클래스를 구현하며, BindIntercpetor 인터페이스를 구현하여 CAM_BindInterceptor 클래스를, ClientInterceptor 인터페이스를 구현하여 CAM_ClientInterceptor 클래스를, ServerInterceptor 인터페이스를 구현하여 CAM_

ServerInterceptor 클래스를, ClientInterceptorFactory 인터페이스를 구현하여 CAM_ClientInterceptorFactory 클래스를, ServerInterceptorFactory 인터페이스를 구현하여 CAM_ServerInterceptorFactory 클래스를 각각 만든다.

위에서는 인터셉터가 구현해야 할 최소한의 클래스들에 대해서 살펴보았고, 이들 클래스 외에 부가적으로 필요한 기능에 대해 다른 클래스를 정의하여 사용할 수 있으며 공통 기능을 묶어 상위 클래스를 정의할 수도 있다. CAM_BindInterceptor, CAM_ServerInterceptor, CAM_ClientInterceptor의 공통 기능을 묶어 이들의 슈퍼클래스인 CAInterceptor클래스를 정의하였다. 그리고, 각 클래스에 전달되는 이벤트 간의 시간을 측정하기 위해 CAInterceptor_Timer와 CAInterceptor_TwoTimer 클래스를 정의했다. 또한, 인터셉터가 CAM이 동작하는지를 판단해서 그 가로챈 메시지와 이벤트를 전달해야 하므로 이를 담당하는 MonitorBinderImpl을 정의하였다.

4. 코바 어플리케이션 모니터링 알고리즘

코바 어플리케이션의 부하를 계산하기 위해서는 그림 4에서와 같이 호출 시에 발생하는 이벤트를 시각에 따라서 정의하고, 이들 이벤트들 간의 시간 중에서 의미있는 시간들을 기반으로 그 값을 계산해야 한다.

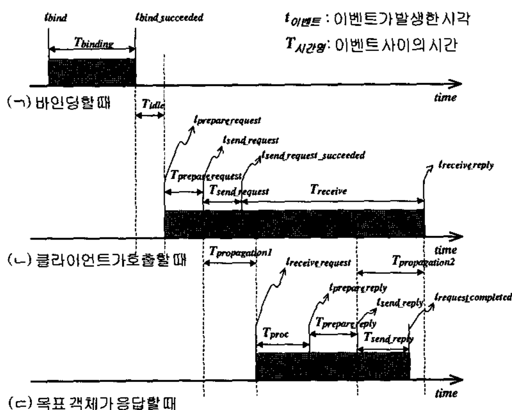


그림 4 호출시 발생하는 이벤트의 시간 정의

클라이언트가 서버 객체의 메쏘드를 호출하기 위해서 바인딩을 하게 되는데, 바인딩 시간 $T_{binding}$은 다음과 같다.

$$T_{binding} = t_{bind_succeeded} - t_{bind}$$

($t_{bind_succeeded}$: 바인딩이 성공한 시각, t_{bind}: 바인딩을 시작한 시각)

바인딩이 성공한 후에 클라이언트는 서버 객체의 레퍼런스를 얻어서 그 메쏘드를 요청하는데, 요청 준비 시간 $T_{prepare_request}$, 요청을 보내는 시간 $T_{send_request}$, 응답을 받는 시간 $T_{receive}$ 는 다음과 같다.

$$T_{prepare_request} = t_{send_request} - t_{prepare_request}$$

($t_{send_request}$:요청을 보내는 시각, $t_{prepare_request}$: 요청을 준비하는 시각)

$$T_{send_request} = t_{send_request_succeeded} - t_{send_request}$$

($t_{send_request_succeeded}$: 요청을 보내는데 성공한 시각)

$$T_{receive} = t_{receive_reply} - t_{send_request_succeeded}$$

($t_{receive_reply}$: 보낸 요청에 대해 응답을 얻는 시각)

클라이언트가 요청을 보낸 이후에 서버 객체는 그 요청을 받아서 처리하고 그 응답을 클라이언트에게 돌려주는데, 요청 처리 시간 T_{proc}, 응답을 준비하는 시간 $T_{prepare_reply}$, 응답을 보내는 시간 T_{send_reply}는 다음과 같다.

$$T_{proc} = t_{prepare_reply} - t_{receive_request}$$

($t_{prepare_reply}$: 응답을 준비하는 시각, $t_{receive_request}$: 요청을 받는 시각)

$$T_{prepare_reply} = t_{send_reply} - t_{prepare_request}$$

(t_{send_reply}: 응답을 보내는 시각)

$$T_{send_reply} = t_{request_completed} - t_{send_reply}$$

($t_{request_completed}$: 응답을 보내고 호출을 마치는 시각)

그리고, 그림 4에서 $T_{propagation1}$은 바인딩이 성공한 후에 클라이언트가 요청 메시지를 준비하는데 걸린 시간이며, $T_{propagation2}$는 클라이언트의 요청 메시지가 서버 객체에 전달되는 시간이고, $T_{propagation3}$는 서버 객체의 응답이 클라이언트에 전달되는 시간을 말한다. 원격 호출일 경우 $T_{propagation2}$와$T_{propagation3}$가 시스템의 부하에 많은 영향을 줄 수 있으며, $T_{receive}$가 매우 커지게 된다. 클라이언트는 요청 후에 응답을 받기까지의 기다리는 시간 T_{wait}에 민감하게 반응하게 되는데, T_{wait}는 요청을 보내서 응답을 받는데 까지 걸린 시간이므로 다음과 같다.

$$T_{wait} = t_{receive_reply} - t_{send_request} = T_{send_request} + T_{receive_reply}$$

그림 5는 3.4절에서 제시한 인터셉터 구현 기법과 그림 4에서 보여주고 있는 이벤트의 시간 정의를 기반으로 의사 코드 (Psedudo Code) 형태로 작성한 코바 어플리케이션 모니터링 알고리즘이다.

클라이언트의 요청은 ORB에서 정의하는 시간 순서에 의해 인터셉터를 통해 서버로 전달되게 되므로, 인터셉터에서는 이들의 호출 순서를 임의적으로 바꿀 수 없으며 관심 대상은 호출시 발생하는 메시지의 형태에 따

라서 그 시간을 잴 수 있는 메커니즘과 노드의 로컬 정보를 인식하는 데에 있다.

코바 어플리케이션 모니터링은 모니터링한 데이타를 보여주는 모니터(Monitor)의 레퍼런스와 노드의 아이디를 얻어서 모니터에 노드의 아이디를 알림으로써 초기화된다. 이후에는 들러오고 나가는 메시지의 타입을 확인하여 메시지의 타입에 따라서 각 시간을 측정한다. 그리고, 이렇게 측정된 시간을 노드의 아이디와 합께 모니터에 보낸다. 모니터에서는 전체 시스템에서 측정된 시간 및 노드 정보를 시스템 관리자에게 보여주게 된다.

```
/* monitor: 인터셉트한 메세지를 통합 관리하는 모니터의 레퍼런스 */
/* nodeId: 로컬 호스트의 인터넷 어드레스 */
/* message: 코바 어플리케이션 간에 주고 받는 메시지 */
/* messageType: message의 포멧에 따라 형태를 스트링으로 정의 */
/* bindingTime: 클라이런트가 target object에 연결하는 시간 */
/* prepareTime: 보낼 메세지를 준비하는 시간 */
/* sendTime: 메세지를 보내는 시간 */
/* receiveTime: 보낸 메세지에 대해 응답을 받는데 걸린 시간 */
/* processingTime: 클라이언트의 요청을 프로세싱하는데 걸린 시간 */
/* operationName: 클라이언트가 요청하는 서버의 operation 이름*/

Algorithm CAM
Begin
  Monitor := get_monitorReference(monitorName);
  If (monitor = not null) then begin
    nodeId := get_localHostAddress();
    monitor->locate_node(nodeId);
    while (is_application_alive) begin
      messageType := get_message_type();
      message := get_messageReference();

      /* 바인딩 메시지 모니터링 */
      if (messageType = "bind") then
        bindingTime := get_SystemTime();
      else if (messageType = "bind_failed") then
        monitor->bind_failed();
      else if (messageType = "bind_succeeded") then begin
        bindingTime := bindingTime - emTime();
        get_tatgetObject();
      end;

      /* 클라이언트 어플리케이션 메시지 모니터링 */
      else if (messageType = "prepare_request") then
        prepareTime := get_SystemTime();
      else if (messageType = "send_request") then begin
        operationName := get_operationName(message);
        sendTime := get_SystemTime();
        prepareTime := prepareTime - sendTime;
        atach nodeId to message;
      end;
      else if (messageType = "send_request_succeeded")
      then
      begin
        sendTime := get_SystemTime() - sendTime;
        receiveTime := get_SystemTime();
      end;
      else if (messageType = "receive_reply") then begin
        serverNodeId := extract_nodeId(message);
        receiveTime := get_SystemTime() - receiveTime;
        monitor->client_request_info(nodeId, serverNodeId,
            prepareTime, sendTime, receiveTime);
      end;

      /* 서버 어플리케이션 메시지 모니터링 */
      else if (messageType = "receive_request") then begin
        clientNodeId := extract_nodeId(message);
        operationName := get_operationName(message);
        processingTime := get_SystemTime();
      end;
      else if (messageType = "prepare_reply") then begin
        prepareTime := get_SystemTime();
        processingTime := get_SystemTime() -
            processingTime;
      end;
      else if (messageType = "send_reply") then begin
        sendTime := get_SystemTime();
        prepare = sendTime - prepareTime;
        atach nodeid to message;
      else if (messageType = "request_completed") then
      begin
        sendTime = get_SystemTime() - sendTime;
        monitor->server_reply_info(nodeId, clientNodeId,
            processingTime, prepareTime, sendTime);
      end;
    end;
  end;
end;
```

그림 5 코바 어플리케이션 모니터링 알고리즘

5. 동적 부하 분산을 위한 메트릭스

5.1 부하 측정을 위한 데이타

CAM을 위한 인터셉터가 모니터링할 수 있는 데이타를 나열하고, 이 중에서 코바 어플리케이션의 부하를 계산하는데 필요한 데이타를 도출해보자.

5.1.1 모니터링 데이타

표 2는 CAM을 위한 인터셉터가 인터셉트할 수 있는 데이타들이다. 이 데이타들을 파악함으로써 코바 어플리케이션의 움직임을 시각적으로 볼 수 있도록 할 수 있으며, 시스템의 부하를 판단할 수 있는 기본적인 데이타를 CAM에게 제공할 수 있다.

5.1.2 부하분산을 위해 도출한 데이타

표 2에 나열된 데이타 중에서 시스템의 부하와 관련된 데이타들을 도출하면 다음과 같다.

- 노드의 서버 객체 개수(ObjectCountNi)

서버 객체는 클라이언트로부터 받은 요청에 대해 연산을 하거나 그 결과를 데이타베이스에 저장을 하므로

표 2 인터셉터가 모니터링한 데이타

데이타 분류	데이타
노드의 이름 정보	노드의 인터넷 이름(Internet Host Name), 노드의 인터넷 주소(Internet IP Address)
바인딩 정보	객체 이름, 인터페이스 이름, 이벤트 (bind, bind_succeeded, bind_failed, rebind, rebind_succeeded, rebind_failed), Tbinding
클라이언트 정보	요청하는 메쏘드 이름, 서버 객체 이름, 이벤트 (prepare_request, send_request, send_requested_succeeded, send_request_failed, receive_reply, receive_reply_failed), $T_{prepare\ request}$, $T_{send_request}$, $T_{receive}$
서버 객체 정보	응답하는 메쏘드 이름, 서버 객체 이름, 이벤트 (locate, locate_succeeded, locate_failed, locate_forwarded, receive_request, prepare_reply, send_reply, send_reply_failed, request_completed, shutdown), T_{proc}, $T_{prepare\ reply}$, $T_{send\ reply}$
메시지 흐름	클라이언트 노드 이름과 연결하고 있는 서버 노드 이름의 쌍

시스템의 CPU 및 디스크 자원을 사용하게 된다. 그러므로, ObjectCountNi가 커지면 노드의 부하가 커진다고 말할 수 있으므로, 시스템 부하와 관련이 있다.

(정의1) $ObjectCount_{Ni} = Total\ \#\ of\ Objects\ located\ on\ N_i (N_i : ith\ Node)$

- 객체의 응답 개수(ReplyCountOi)

서버 객체는 그 응답 개수에 따라서 시스템의 자원을 많이 또는 적게 사용할 수 있으므로 ReplyCountOi를 파악하는 것은 객체의 부하를 계산하는데 필요하다.

(정의2) $ReplyCount_{Oi} = Total\ \#\ of\ Replies\ of\ O_i (O_i : ith\ Object)$

- 노드의 응답 개수(ReplyCountNi)

위와 같은 이유로 노드의 응답 개수는 노드의 부하를 계산하는데 필요하다.

(정의3) $ReplyCount_{Ni} = Total\ \#\ of\ Replies\ of\ Objects\ located\ on\ N_i$

- 객체의 프로세싱 시간(ProcessTimeOi)

프로세싱하는 데 걸린 시간은 시스템의 자원인 CPU의 사용률과 밀접한 관련이 있으므로 이는 노드의 부하와 관련이 있다.

(정의4) $ProcessTime_{Oi} = \sum_{k=1}^{ReplyCount\ O_i} T_{proc_{M_k}},\ where\ O_i\ processes\ M_k\ (M_k : kth\ Message)$

- 노드의 프로세싱 시간(ProcessTimeNi)

프로세싱하는 데 걸린 시간은 시스템의 자원인 CPU의 사용률과 밀접한 관련이 있으므로 이는 노드의 부하와 관련이 있다.

(정의5) $ProcessTime_{Ni} = \sum_{k=1}^{ObjectCount\ N_i} ProcessTime_{Ok},\ where\ O_k\ is\ located\ on\ N_i$

- 객체의 요청 개수(RequestCountOi)

객체의 요청은 요청을 준비하고 이를 보내기위한 처리를 하기위해 CPU나 메모리, 디스크 자원을 사용하므로, RequestCountOi가 객체의 부하에 영향을 준다.

(정의6) $RequestCount_{Oi} = Total\ \#\ of\ Requests\ of\ O_i$

- 노드의 요청 개수(RequestCountNi)

클라이언트 객체의 요청은 요청한 클라이언트 노드의 부하에도 영향을 준다.

(정의7) $RequestCount_{Ni} = Total\ \#\ of\ Requests\ of\ Objects\ located\ on\ N_i$

- 노드간의 원격 요청 개수(RemoteRequestCountNi,Nj)

원격 호출일 경우에는 네트웍의 트래픽을 가중시킨다. 한 클라이언트가 원격지의 서버 객체로 요청을 빈번하게 한다면 이들 사이의 전송 경로 상의 트래픽이 가중되게 된다.

(정의8) $RemoteRequestCount_{Ni,Nj} = Total\ \#\ of\ the\ pair\ (Q_i, P_j),\ Q_i: the\ Requests\ on\ N_i,\ P_j: the\ Replies\ of\ the\ Objects\ on\ N_j$

- 노드간의 평균 전송 시간(AveragePropagationTimeNi,Nj)

평균 전송 시간은 이 노드와 응답하는 노드간의 전송 시간을 의미하므로 이의 네트웍 속도와 연관이 있으며 다른 노드와의 트래픽과도 밀접한 관련이 있다. 한 노드에서 다른 노드로의 평균 전송 시간이 길다면 그 전송 경로의 네트웍 부하가 크다고 말할 수 있다.

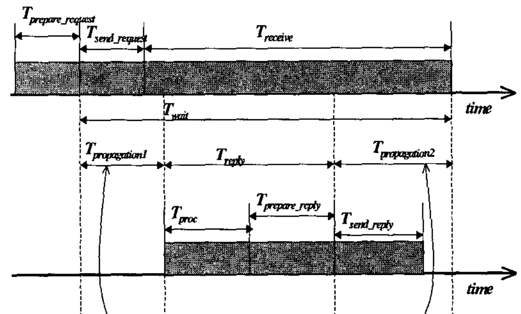


그림 6 네트웍 전송 시간(T_{prop})의 정의

그림 6에서 Treply를 서버 객체가 요청을 받아서 응답을 보내는 시간으로 정의할 경우, AveragePropagationTimeNi,Nj은 다음과 같이 구할 수 있다.

(정의9) $T_{wait_{Ni,Nj}} = T_{send_{request\ Nj}} + T_{receive_{Nj}}$,
$T_{send_{request\ Nj}}$: $T_{send_{request}}$ *of a Request, which is requested from a client* N_i

(정의10) $PropagationTime_{Mi,Mj} = T_{wait_{Mi,Mj}} - T_{proc_{Mj}} - T_{prepare-reply_{Mj}}$, *where* $PropagationTime_{Mi,Mj}$ *is* T_{prop} *of the one request from* N_i *to* N_j

(정의11) $PropagationTime_{Ni,Nj} = \sum_{k=1}^{n} PropagationTime_{Mi,Mj}$, *where n is total request # from* N_i *to* N_j

(정의12) $AveragePropagationTime_{Ni,Nj} = \frac{PropagationTime_{Ni,Nj}}{n}$

5.2 동적 부하와 관련된 데이타

5.1.2절에서 부하 측정을 위한 데이타들을 제시하였는데, 본 절에서는 (정의1~12)에서 정의된 데이타들이 얼마나 부하와 관련이 있는지를 살펴보고 이에 따라 가중치를 줌으로써, 동적 부하를 계산할 수 있는 메트릭과 이를 통한 동적 부하 분산 메트릭을 제시하려 한다.

5.2.1 한 노드에서의 부하 측정시 고려해야 할 데이타

노드의 부하를 고려할 때에 $ObjectCount_{Ni}$와 $ReplyCount_{Ni}$가 고려되어야 한다. 서버 객체는 노드의 자원(CPU, 주메모리, 디스크, 네트웍 등)을 사용하여 클라이언트로부터 받은 요청에 응답을 한다. 이 과정에서 서버 객체는 연산을 하거나 데이타베이스에 저장을 하거나 또 다른 서버 객체에 요청을 하기도 한다. 서버 객체가 많다는 것은 그만큼 그 노드의 주메모리를 차지하는 것은 분명하다. 그러나, 서버 객체의 수가 많더라도, 클라이언트로부터 요청이 적다면 CPU나 디스크를 적게 쓸 수 있으므로 서버 객체가 응답한 개수도 함께 고려가 되야 한다.

표 3 모니터링 데이타와 노드의 동적 부하 관련도

Monitoring Data \ Load	Load in N_i	Traffic-Load between N_i, N_j
Object Count ($ObjectCount_{Ni}$)	○	
Reply Count ($ReplyCount_{Ni}$)	○	
Processing Time ($ProcessTime_{Ni}$)	◎	
Request Count ($RequestCount_{Ni}$)	△	
Remote Request Count ($RemoteCount_{Ni,Nj}$)		△
Average Propagation Time ($AveragePropagationTime_{Ni,Nj}$)		◎

(△: loosely related ○: related ◎: tightly related)

또한 노드의 부하를 고려할 때에 $ProcessTime_{Ni}$가 고려되어야 하며, 노드의 부하와 관련이 크다. 프로세싱 하는 데 걸린 시간은 시스템의 자원인 CPU의 사용률과 밀접한 관련이 있으므로 이는 노드의 부하와 관련이 있다. $ProcessTime_{Ni}$가 큰 노드i의 CPU사용율은 확실히 $ProcessTime_{Nj}$가 작은 노드j의 부하보다는 크다고 말할 수 있으므로, $ObjectCount_{Ni}$와 $ReplyCount_{Ni}$보다 부하를 측정하는데 보다 정확한 자료이다. 그러므로, 부하를 계산하는데 있어 $ObjectCount_{Ni}$와 $ReplyCount_{Ni}$의 가중치보다는 크게 주어야 한다.

노드의 부하를 고려할 때에 $RequestCount_{Ni}$가 고려될 수 있으며, 노드의 부하와 어느 정도 관련이 있다. 클라이언트의 요청은 요청을 하는 클라이언트 객체의 수행을 의미하며 이는 노드의 자원을 사용할 것이다. 그러나, $RequestCount_{Ni}$가 크더라도 클라이언트 객체에 따라서 요청에 따른 메시지 크기가 다를 수 있으므로 노드의 부하를 구하는데 가중치를 크지 않게 준다.

5.2.2 한 객체의 부하 측정시 고려해야 할 데이타

객체의 부하에 대해 고려해야 할 데이타는 표 4와 같다. 노드의 부하를 고려할 때와 마찬가지로 객체의 부하를 구할 수 있다. 노드의 부하를 구할 때 고려해야 할 데이타 중에서 $ObjectCount_{Ni}$는 하나의 객체에 대한 부하를 구하는데 의미가 없으므로 제외하고, $ReplyCount_{Ni}$, $RequestCount_{Ni}$, $ProcessTime_{Ni}$를 고려했듯이 객체에 대해서도 $ReplyCount_{Oi}$, $RequestCount_{Oi}$, $ProcessTime_{Oi}$를 고려한다.

표 4 모니터링 데이타와 객체의 동적 부하 관련도

Monitoring Data \ Load	Load of O_p
Reply Count ($ReplyCount_{Op}$)	○
Processing Time ($ProcessTime_{Op}$)	◎
Request Count ($RequestCount_{Op}$)	△

(△: loosely related ○: related ◎: tightly related)

5.2.3 두 노드 간의 트래픽 부하 측정시 고려해야 할 데이타

또한, 원격 호출일 경우에는 그 경로상의 네트웍 트래픽을 가중시키므로, $RemoteRequestCount_{Ni,Nj}$가 고려되어야 한다. $RemoteRequestCount_{Ni,Nj}$는 원격 요청의 수를 말하며, 한 클라이언트가 원격지의 서버 객체로 요청을 빈번하게 한다면 이들 사이의 전송 경로 상의 트래픽이 가중되게 된다.

$RemoteRequestCount_{Ni,Nj}$가 크더라도 요청의 메시지 크기나 네트웍 속도에 따라서 트래픽 부하가 달라질 수

있으므로 네트웍 부하와 정확하게 비례하지는 않으므로, $AveragePropagationTime_{Ni,Nj}$도 함께 고려되어야 한다. $AveragePropagationTime_{Ni,Nj}$은 네트웍을 통해 전송되는 시간을 말하며, 클라이언트가 요청을 보내고 기다린 시간에서 서버 객체가 요청을 처리하고 응답을 준비하는 시간을 뺀 시간이다.

5.3 동적 부하 메트릭스

윗 절에서 찾아낸 데이타와 동적 부하와의 관계에서 가중치를 부여하고 그 관계가 비례 또는 반비례인지를 나타내면 표 5, 표 6과 같다. 그리고 그 데이타가 부하와 관련이 많고 적음에 따라 가중치를 부여하면, 그 가중치는 $Wl < Wr < Wt$임을 알 수 있다.

도출된 데이타들은 그 값들의 단위가 다르므로 이들을 일관성 있게 정량화 할 필요가 있다. 5.1.2절에서 제시한 집중률은 각 데이타가 어느 노드에 집중되는지를 알아보는데 좋지만, 전체 노드의 수가 커지면 상대적으로 집중률이 낮아져 일관성이 없어진다. 그러므로, 각 데이타 중에서 최대 값을 1로 하여 상대적인 값으로 계산하여 전체 노드의 크기에 관계없이 다른 데이타들과 비교할 수 있는 값을 사용하여 부하를 구한다. 도출한 데이타는 다음과 같은 범위를 갖는다.

표 5 노드의 부하 측정에 대한 가중치 및 비례/반비례 관계

Monitoring Data \ Load	Load in Ni	Traffic-Load between $N_i N_j$
Object Count ($ObjectCount_{Ni}$)	W_r, 비례	
Reply Count ($ReplyCount_{Ni}$)	W_r, 비례	
Processing Time ($ProcessTime_{Ni}$)	W_t, 비례	
Request Count ($RequestCount_{Ni}$)	W_l, 비례	
Remote Request Count ($RemoteRequestCount_{Ni,Nj}$)		W_l, 비례
Average Propagation Time ($AveragePropagationTime_{Ni,Nj}$)		W_t, 비례

(weight: $W_l < W_r < W_t$)

표 6 객체의 부하 측정에 대한 가중치 및 비례/반비례 관계

Monitoring Data \ Load	Load of O_p
Reply Count ($ReplyCount_{Op}$)	W_r, 비례
Processing Time ($ProcessTime_{Op}$)	W_t, 비례
Request Count ($RequestCount_{Op}$)	W_l, 비례

(weight: $W_l < W_r < W_t$)

시스템에 노드가 $N_1, N_2, \cdots, N_i, \cdots N_j, \cdots, N_{n-1}, N_n$과 같이 있고 임의의 Ni에 객체 $O_1, O_2, \cdots, O_p, \cdots, O_{ObjectCountNi-1}, O_{ObjectCountNi}$이 존재한다고 가정한다.

$O=\{ObjectCount_{N1},\cdots, ObjectCount_{Nn}\}$, $Max_ObjectCount = Max(O)$이면, $0 \le ObjectCount_{Ni} / Max(O) \le 1$,

$P=\{ProcessTime_{N1},\cdots,ProcessTime_{Nn}\}$, $Max_ProcessTime = Max(P)$이면, $0 \le ProcessTime_{Ni} / Max(P) \le 1$,

$Q=\{RequestCount_{N1},\cdots,RequestCount_{Nn}\}$, $Max_RequestCount=Max(Q)$이면, $0 \le RequestCount_{Ni}/Max(Q) \le 1$,

$R=\{ReplyCount_{N1},\cdots, ReplyCount_{Nn}\}$, $Max_ReplyCount = Max(R)$이면, $0 \le ReplyCount_{Ni} / Max(R) \le 1$,

$M=\{RemoteRequestCount_{Ni,N1},\cdots,RemoteRequestCount_{Ni,Nj},\cdots,RemoteRequestCount_{Ni,Nn}\}$, $Max_RemoteRequestCount = Max(M)$, $i \ne j$이면, $0 \le RemoteRequestCount_{Ni,Nj} / Max(M) \le 1$,

$A=\{AveragePropagationTime_{Ni,N1},\cdots,AveragePropagationTime_{Ni,Nj},\cdots,AveragePropagationTime_{Ni,Nn}\}$, $Max_AveragePropagationTime = Max(A)$, $i \ne j$이면, $0 \le AveragePropagationTime_{Ni,Nj} / Max(A) \le 1$이다.

위와 같이 가정하고 다음을 구해보자.

- 노드 i의 부하 ($LOAD_{Ni}$) 메트릭

(정의13) $LOAD_{Ni} = W_l*RequestCount_{Ni} / Max_RequestCount + W_r*(ObjectCount_{Ni} / Max_ObjectCount + ReplyCount_{Ni} / Max_ReplyCount) + W_t* ProcessTime_{Ni} / Max_ProcessTime$

$0 \le LOAD_{Ni} \le W_l+2W_r+W_t$, 단, $W_l<W_r<W_t$

- 노드i와 노드j 간의 트래픽 부하 ($LOAD_{Ni,Nj}$) 메트릭

(정의14) $LOAD_{Ni,Nj} = W_l*RemoteRequestCount_{Ni,Nj} / Max_RemoteRequestCount + W_t*AveragePropagationTime_{Ni,Nj} / Max_AveragePropagationTime$

$0 \le LOAD_{Ni,Nj} \le W_l+W_t$, 단, $W_l<<W_t$

- 객체p의 부하 ($LOAD_{Op}$) 메트릭

(정의15) $LOAD_{Op} = W_l*RequestCount_{Op} + W_r*ReplyCount_{Op} + W_t* ProcessTime_{Op}$

5.4 동적 부하 분산 메트릭스

코바 어플리케이션이 기능적으로 분산되어 있지만,

성능적으로 잘 분산이 되어 있는 지는 모른다고 가정한다. 또한, 노드가 N_1, N_2, … N_{i-1}, N_i, N_{i+1}, … N_{n-1}, N_n과 같이 분포되어 있으며, N_i에는 서버 객체가 O_1, O_2, … O_p, …, $O_{ObjectCountNi-1}$, $O_{ObjectCountNi}$과 같이 분포되어 있다고 가정한다. ($1<i<n$, n=전체 노드 수일 때, $1<p<ObjectCount_{Ni}$가 된다.)

- 객체 이동(Object Migration) 메트릭

(정의16) $LOAD_{Ni}$가 가장 큰 N_i의 서버 객체중 $LOAD_{Op}$가 가장 작은 서버 객체 O_P를 $(LOAD_{Nj})2+LOAD_{Ni,Nj}$가 가장 작은 N_j로 옮긴다.

즉, 서버 객체 O_p를 노드의 부하가 작은 노드 중에서 그 노드로의 트래픽 부하가 적은 노드로 옮기는 것이다. 그런데, 여기서 노드i의 부하를 더 고려한 이유는 객체의 이동은 한번이지만 옮겨진 객체가 서비스하는 횟수는 여러 번이기 때문이다.

- 객체 복제(Object Replication) 메트릭

(정의 17) $LOAD_{Ni}$가 가장 큰 N_i의 서버 객체중 $LOAD_{Op}$가 가장 큰 서버 객체 O_p를 $LOAD_{Nj}+(LOAD_{Ni,Nj})^2$가 가장 작은 N_j에 복제한다.

즉, 서버 객체 O_p를 그 노드로의 트래픽 부하가 적은 노드 중에서 노드의 부하가 작은 노드로 옮기는 것이다. 그런데, 복제 시에 고려할 사항은 객체들의 상태를 일치시키는 문제이다. 객체들의 상태를 일치시키기 위해 복제된 객체간에 메시지 전달이 발생하는데, 두 노드 간의 네트웍부하가 이들의 상태를 일치시키기 위해 소비되는 시간을 결정하므로 옮겨질 노드의 부하보다는 두 노드 간의 네트웍부가가 보다 중요하게 고려되어야 한다.

- 네트웍 대역폭 확장 메트릭

(정의18) $Load_{Ni,Nj}$가 가장 큰 N_i와 N_j 사이의 네트웍 대역폭을 높인다.

5.5 동적 부하분산 예제

앞 절에서 설명된 동적 부하측정 메트릭에 의해 가능한 예제에 대해 각 부하를 구해보고, 정의된 부하 분산 메트릭스가 적합한지를 알아보자.

그림 7에서 (정의13)에 의해 각 노드의 부하를 구하면, 5.5, 6.5, 5.3이다. 노드2의 부하가 가장 크다는 것을 알 수 있으며, 노드2와 노드3사이의 트래픽 부하가 노드1과의 트래픽 부하보다 훨씬 크다. 여기서, 노드2의 부하를 객체 이동을 통하여 줄일 것인지 객체를 복제하여 줄일 것인지의 문제가 발생한다. 먼저 객체 이동에 대해 생각해보면 5.4에서 정의한 (정의16)에 의해 노드1로 옮기는 것이 좋으며, (정의17)에 의해서는 노드1로 복제해야 한다. 그러나 노드2로의 이동 비용에 대한 노드1로의 이동 비용 비율이 노드2로의 복제 비용에 대한 노드1로의 복제 비용 비율이 훨씬 적으므로, 노드1로의 복제가 좋다.

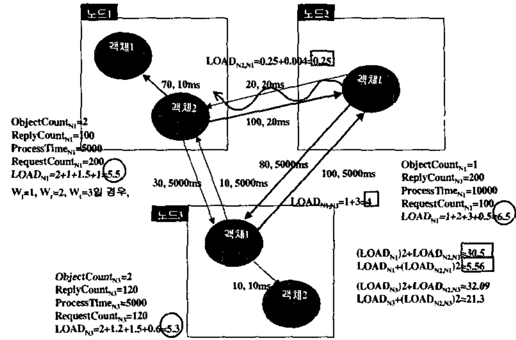


그림 7 동적 부하 분산 예제

6. CAM의 부하 평가

CAM의 부하 평가는 CAM이 코바 어플리케이션의 수행 속도에 미치는 영향을 알아보기 위한 평가이다. 이는 CAM이 코바 어플리케이션의 수행에 큰 영향을 미쳐 전체 성능을 떨어뜨린다면, 동적인 부하 분산의 의미를 퇴색하게 할 수 있으므로 중요하다. CAM의 부하를 평가하기위한 테스트 환경은 표 7과 같다.

표 7 S/W와 H/W 테스트 환경

No	OS	Test Applica-tion	Number of Client App. / Number of Server App.	CPU (Mhz)	Main Memory (Mbyte)	Application Distribution
1	Windows NT			200	128	Remote
2	Windows NT			200	32	Remote
3	Windows NT			200	128	Local
4	Windows NT			200	32	Local
5	Windows NT			120	128	Remote
6	Windows NT	Bank	N/ 1 (N=1,3,5,10,15,20)	120	32	Remote
7	Windows NT			120	128	Local
8	Windows NT			120	32	Local
9	Solaris			167	128	Remote
10	Solaris			(Sun Ultra1)	(Sun Ultra1)	Local

테스트할 코바 어플리케이션으로는 Bank어플리케이션을 사용하였으며, 테스트의 목적에 맞게 여기에서는 어플리케이션들만의 성능과 인터셉터를 사용했을 경우와 CAM을 같이 사용했을 경우의 성능의 차이를 알아봄으로써 CAM의 부하를 테스트하고자 한다. 표 8의 각 테스트 케이스에서 어플리케이션 간의 메시지 교환 횟수에 따라서 변화하는 수행 속도를 테스트하기 위해 클라이언트의 요청 횟수를 초당 6에서 120까지 변화하였다.

표 8 테스트 케이스

Test Case	Test Data
• **Bank:** Measure the client's average response time on the bank application when each server's received message count/sec are 6, 18, 30, 60, 90, 120 and use the result as a base value.	1 Bank Server & N Bank Client (N=1,3,5,10,15,20)
• **With CAInterceptor:** Measure the client's average response time on the bank application and CAInterceptor when each server's received message count/sec are 6, 18, 30, 60, 90, 120 and compare the result with the base value.	1 Bank Server with CAInterceptor & N Bank Client with CAInterceptor (N=1,3,5,10,15,20)
• **With CAInterceptor and CAMonitor:** Measure the client's average response time on the bank application CAInterceptor and CAMonitor when each server's received message count/sec are 6, 18, 30, 60, 90, 120 and compare the result with the base value.	1 Bank Server with CAInterceptor & N Bank Client with CAInterceptor & CAMonitor (N=1,3,5,10,15,20)

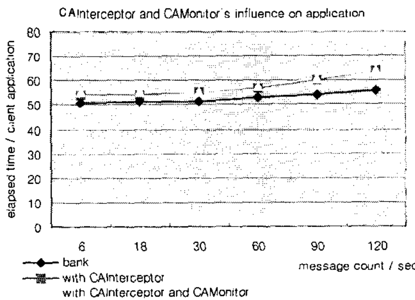


그림 8 1번 테스트 환경에서의 테스트 결과

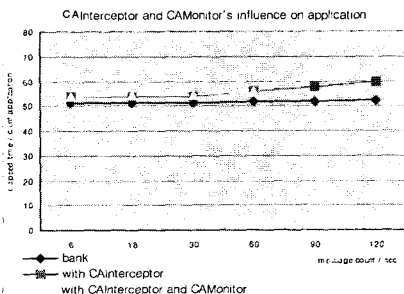


그림 9 3번 테스트 환경에서의 테스트 결과

그림 8과 그림 9를 비교해 보면 CAM과 코바 어플리케이션이 분산되어 있는지에 따라서 CAM이 어플리케이션의 수행속도에 미치는 영향이 달라짐을 알 수가 있다. 시스템이 분산 환경일 경우 보통의 경우는 1번 환경과 같이 되므로, CAM의 부하는 크지 않다고 할 수 있다.

3번 테스트 환경에서는 서버 어플리케이션으로 클라이언트 어플리케이션이 로컬 호출을 하므로 어플리케이션만 수행했을 경우는 1번의 경우보다 빠르지만, CAM과 같이 수행할 경우는 모니터의 부하까지 겹치게 되어 성능이 오히려 나빠짐을 알 수 있다.

7. 결론 및 향후 연구과제

이상에서 코바 어플리케이션의 동적 부하분산을 위한 모니터링 기법과 부하 분산 메트릭스를 제시하였다. 그 모니터링 기법은 코바 어플리케이션의 부하를 측정하기 위한 수단으로 사용하였으며, 이를 이용한 부하 분산 메트릭스는 코바 어플리케이션의 성능 향상에 도움을 줄 것이다. 본 논문에서는 노드의 부하, 객체의 부하, 노드간의 부하를 메트릭으로 제시하였으며, 이를 근거로 시스템에서의 과부하를 방지하도록 부하를 분산할 수 있는 메트릭스를 제시하고 있다. 또한, 실제로 코바라는 표준 미들웨어상에서 코바 어플리케이션을 모니터링하고 동적 부하를 구하는 방법을 제시하였으며 인터셉터와 CAM을 통해 이를 구현 하였다.

모니링에 대한 오버헤드 때문에 어플리케이션의 수

행 속도가 느려지게 되는데, 이는 모니터의 사용에 걸림돌이 되고 있다. 그러므로, 코바 어플리케이션 모니터링의 속도 향상에 대한 연구가 필요하다. CAM을 C++로 구현하거나, 인터셉터와 모니터간의 API를 단순화하고, 미들웨어라는 무거운 메시지 전송 수단보다는 소켓과 같은 가벼운 전송 수단을 선택하는 것도 하나의 방법일 수 있을 것이다. 표 5에서 제시한 가중치 W_l, W_r, W_t에 대한 통계적인 값이 제시되어 운영자가 쉽게 이를 적용할 수 있도록 해야 한다. 또한, 메트릭에 대한 보다 많은 산업계의 검증이 이루어져야 할 것이다. 그리고, 절차적 방식의 TP Monitor인 BEA' s TUXIDO, IBM' s CISC, IBM/Transarc' s Encina등이 동적 부하 분산에 소극적인 전략을 취하고 있다. 향후 네트웍 시스템의 보급으로 분산 환경이 필수적이며, 코바와 DCOM같은 분산환경에 적합한 동적 부하 분산 전략 및 관련 도구들에 대한 연구가 필요하다. 마지막으로, 본 논문에서 구현한 CAM이 시스템의 부하를 자동으로 측정하는 기능 뿐만 아니라, 시스템 관리자를 보조하거나 대신하여 시스템의 부하를 분산시킬 수 있도록 실시간 객체 복제 및 실시간 객체 이동 기능을 추가해야 하는 과제가 남아 있다.

참 고 문 헌

[1] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.2, Object Management Group, Framingham, Mass., February 1998.

[2] Object Management Group: CORBAservices: Common Object Services Specification, Object Management Group, Framingham, Mass, July 1997.

[3] Visigenic, Visibroker for Java Programmer's Guide Version 3.2, Visigenic Software, Inc. 1998.

[4] Visigenic, Visibroker for Java , Reference Manual Version 3.2, Visigenic Software, Inc. 1998.

[5] Martin Fowler, UML DISTILLED Applying the Standard Object Modeling Language , Addison-Wesley, 1997.

[6] J. Rumbaugh et. al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[7] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1994.

[8] R. Orfali, D. Harkey, J. Edward, *Client/Server Programming with JAVA and CORBA*, Jon Wiley&Sons: New York, NY, 1997.

[9] Simon Moser and Vojislav B. Misic, Measuring Class Coupling and Cohesion: A Formal Metamodel Approach , Asia Pacific Software Engineering Conference, Dec. 1997.

[10] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, PWS Publishing Company, 1997.

[11] Shyam R. Chidamber, Chris F. Kemerer, Towards a Metrics for object-oriented Design, In Proc. OOPSLA '91, pp.197-221, ACM 1991.

[12] 김수동, 김철진, "객체지향 클라이언트/서버 시스템 아키텍처", 정보처리학회지 특집 클라이언트/서버, Vol. 4, No. 6, pp.16-29, 1997.

[13] Martin Hitz and Behzad Montazeri, "Measuring Coupling in Object-Oriented Systems," Object Currents, Vol.1 No.4, April 1996.

[14] Teri Roberts, "Metrics for Object-Oriented Software Development, Workshop Report in Addendum to the Proceedings OOPSLA '92, pp.97-100, ACM 1992.

[15] M. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing for a NOW , 5th IEEE Intl. Symp. High-Performance Distributed Computing, also TR 602, U. Rochester, Aug. 1996.

[16] Herbert Kuchen, Andreas Wagener, Comparison of Dynamic Load Balancing Strategies , RWTH Aschen, West Germany, May 1990.

[17] Guenther Rackl, Load Distribution for CORBA Environments, Diploma Thesis, http://sunpaul9.informatik.tu-muenchen.de/projekte/dcw/da_rackl/, Jan. 1997.

최 창 호

1995년 숭실대학교 전자계산학과 학사. 1995년 ~ 1996년 데이콤 SI사업단 근무. 1999년 숭실대학교 전자계산학과 석사. 관심분야는 분산 객체 컴퓨팅, 객체지향 모델링, 객체지향 데이타 베이스, 전자상거래 시스템

김 수 동

정보과학회논문지 : 소프트웨어 및 응용 제 27 권 제 3 호 참조