분산 시스템에서의 부하 공유 기법 설계 및 성능 평가

김 구 수[†] · 엄 영 익[††]

요 약

본 논문에서는 분산 시스템에서 프로세서 자원들을 효율적으로 사용하기 위한 부하 공유 기법을 제안한다. 본 기법에서는 분산 시스템 내의 각 호스트들의 상태를 부하의 정도에 따라 서버와 소스로 구분하였으며, 각 호스트의 빈번한 상태 변화에 따른 오버헤드를 감소시키기 위하여 3개의 임계값을 사용하여 호스트들의 상태 변화 과정을 관리하도록 하였다. 또한 전체 호스트의 수 및 시스템 이용도에 따라 적절한 수의 중개자 호스트들을 두도록 하고, 중개자 호스트로 하여금 고부하 호스트인 소스로부터 저부하 호스트인 서버로의 작업 이주 과정을 중개하도록 하였으며, 중개자 호스트의 부하에 따라 중개자 역할의 전이도 가능하도록 하였다. 각종 시스템 인자들(임계값, 이용도, 전체 호스트의 수, 중개자 호스트의 수 등)이 제안 기법의 성능에 어떻게 영향을 미치는가를 알기 위하여 시뮬레이션으로 제안 기법을 평가하고 그 결과를 보였다.

Design of the Load Sharing Scheme and Performance Evaluation in Distributed Systems

Gu Su Kim[†] · Young Ik Eom[††]

ABSTRACT

In this paper, we propose a load sharing scheme in distributed systems. In the proposed scheme, the state of each host is classified as a server or a source by its current load and, to prevent excessive state changes of each host, we used three threshold values for identifying the current state of each host. Based on the threshold values, some hosts, called brokers, manage the servers registered to them. The brokers, whose number is determined by the system utilization factor and the total number of hosts, support task migration processes from overloaded sources to lightly loaded servers. Also they can hand over the broker's role to another host when it is overloaded. Simulation studies were performed for examining the sensitivity of each system parameters such as threshold values, utilization factor, the number of hosts, and the number of brokers to the system performance indices including mean response time, mean queue length.

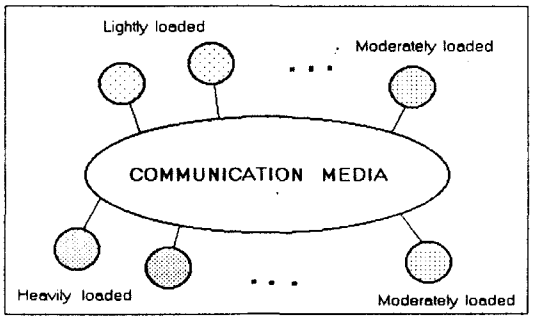
1. 서 론

분산 시스템(distributed system)은 자치적인(autonomous) 개별 호스트들이 네트워크를 통해 묶여져, 사용자에게 하나의 가상의 컴퓨터로 여겨지도록 서비스가 제공되는 시스템이다[1]. 분산 시스템은 자원의 공유가 가능하도록 해주고, 시스템의 신뢰성(reliability)과 가용성(availability)을 향상시키며, 전체 시스템의 성능을 증가시켜 준다. 하지만 분산 시스템에는 분산 프로세스들의 관리, 프로세스간 통신, 동기화 등과 같

† 비 회 원:(주)대교컴퓨터 사이에듀팀 연구원
†† 종신회원:성균관대학교 정보공학과 교수
논문접수:1996년 11월 11일, 심사완료:1997년 7월 23일

이 프로세스들과 관련된 문제점들과 네이밍(naming), 원격 자원에 대한 접근과 할당, 교착상태, 보호, 인증 서비스 등과 같이 자원과 주소에 관련된 문제점들이 있다[2]. 특히 시스템의 자원들 중에서 프로세서(processor)는 값비싼 자원에 속하며 이러한 프로세서 자원을 보다 효율적으로 사용하고자 하는 노력이 계속되고 있다.

분산 시스템상의 워크스테이션들은 일반적으로 항상 바쁘게 운영되지는 않는다. 실제로 Mutka 와 Linvy의 조사에 따르면 평상시 컴퓨터 능력의 30 퍼센트만이 이용된다고 한다[3]. 이러한 현실에서 컴퓨터 능력에 대한 사용자들의 요구가 아주 큰 경우 단일 컴퓨터에서 만족시키기 힘든 경우가 있고, 어떤 경우는 컴퓨터의 능력이 사용자들의 요구를 훨씬 능가할 때도 있다. 이러한 상황에서 분산 시스템에서는 몇몇 컴퓨터들이 과부하 상태에서 운영되고 있는 동안, 다른 컴퓨터들은 한가한(idle) 상태에서 운영되는 경우가 흔하게 나타날 수 있다(그림 1).



(그림 1) 부하의 분산이 없는 분산 시스템
(Fig. 1) Distributed system with no load sharing

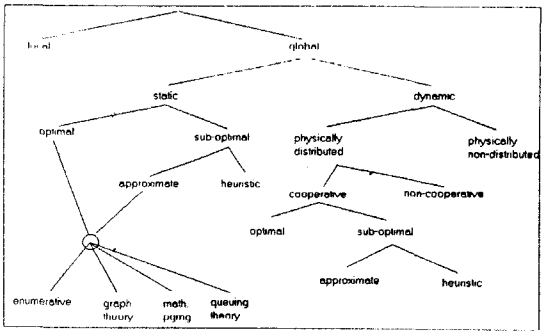
이러한 현상은 값비싼 프로세서 자원들을 비효율적으로 사용하게 되는 결과를 초래한다. 이러한 시스템들 사이에 발생하는 부하의 불균형을 해소하기 위해서는 과부하 상태의 호스트들의 부하를 덜어 주어야만 한다. 이처럼 과부하 상태의 호스트에서 작업의 일부를 저부하 상태의 호스트로 이주시켜 서비스 받게 하여 부하의 불균형을 해소시키고 전체적으로 작업들의 평균 응답 시간을 줄이되 사용자에게는 이러한 사실들이 은닉되게 하는 것을 부하 공유(load sharing) 기법이라 한다[4].

본 논문에서는 중개자(broker)라는 호스트들을 두어 저부하 상태의 호스트(서버)들을 관리하고 과부하 상태의 호스트(소스)들로부터의 프로세스 이주 요청에 대해 프로세스 이주의 목적지를 선정해 줌으로서 부하의 공유를 이루는 기법을 제안하고자 한다.

2. 부하 공유 기법의 분류 및 정책

2.1 부하 공유 기법의 분류

부하 공유 문제는 일종의 스케줄링 문제에 속한다. 이 문제를 해결하는 기법들을 구체적으로 각 기법들의 특징에 따라 구분하면 (그림 2)와 같이 분류할 수 있다[5].



(그림 2) 부하 공유 기법의 분류
(Fig. 2) Classification of load sharing schemes

(1) 지역적(local) 기법과 전역적(global) 기법

지역 스케줄링은 단일 프로세서의 타임 슬라이스(time slice)에 프로세스를 할당하는 기법을 말하며 전역 스케줄링은 프로세스를 시스템내의 여러 호스트들중 어느 호스트에서 실행시킬 것인지를 결정하는 기법을 말한다.

(2) 정적(static) 기법과 동적(global) 기법

전역 스케줄링 기법은 정적(static) 기법과 동적(dynamic) 기법으로 구분되어진다. 정적 기법은 시스템이 진행되기 전에 미리 스케줄링을 완료하여 이후에는 각 호스트의 부하에 관계없이 정해진 스케줄링 결과 대로 운영하는 기법이다. 반대로 동적 기법은 매 순간마다 그 당시의 시스템의 상태에 따라 스케줄

링을 수행하는 기법이다.

(3) 분산(distributed) 기법과 비분산(non-distributed) 기법

전역 동적 스케줄링이 어느 한 호스트에서만 이루어지는 경우에는 이를 비분산 기법이라 하며, 이 스케줄링에 여러 프로세서들이 같이 참여하는 경우를 분산된 기법이라 한다.

(4) 협력적(cooperative) 기법과 비협럽적(noncooperative) 기법

한 프로세서의 상태가 다른 프로세서의 스케줄링 결정에 영향을 미치는 경우를 협력적 기법이라 하며, 어느 한 프로세서의 상태가 다른 프로세서의 스케줄링 결정에 영향을 미치지 않고 프로세서 단위로 독립적 결정에 의존하는 경우를 비협력적 기법이라 한다.

(5) 최적(optimal) 기법과 준-최적(sub-optimal) 기법

최적 기법은 시스템에 관한 모든 정보, 즉 호스트의 상태, 프로세스에 필요한 자원 등이 모두 미리 알려져야 하고 이를 기반으로 하여 최적의 성능 향상을 이루도록 스케줄링 하는 기법이다. 준-최적 기법은 최적 기법은 아니지만 어느 한계선에서 성능 향상을 이루도록 스케줄링 하는 기법을 말한다.

(6) 개략적(approximate) 기법과 경험적(heuristic) 기법

개략적 기법이란 최적의 상태를 찾는 것이 아니라 어느 정도의 만족할 만한 성능 향상을 찾는 기법을 말한다. 반면에, 경험적 기법은 프로세스와 시스템 부하 특성에 관한 사전 정보에 대해 가장 현실적인 가정을 세우고 이를 바탕으로 스케줄링 정책을 수행하는 기법을 말한다.

본 논문에서는 시스템의 현재 상태를 정기적으로 조사하여 각 순간마다 상황에 맞는 스케줄링을 하는 동적이고 개략적인 기법을 제안하고자 한다.

2.2 부하 공유에 필요한 정책(policy)

부하 공유는 결국 분산 시스템의 성능을 향상시키기 위해 "어느 프로세스를 언제 어느 호스트로 옮겨야 하는가?"라는 문제를 야기하며 이를 결정하기 위해 고려해야 할 스케줄링 정책(policy)들이 몇 가지 있다[6]. 우선은 어느 호스트가 과부하 상태인지를 결정하고, 언제 그 호스트의 작업을 다른 호스트로 보내야 할 지를 결정해야하는 전송(transfer) 정책이 필요하다. 전송 정책이 송신자 호스트를 정했다면, 다음으로는 전송할 작업을 선정하는 선택(selection) 정책이 필요하다. 또 프로세스 이주의 목적지, 즉 수신자 호스트를 선정하기 위한 탐색(location) 정책이 필요하며, 마지막으로 시스템에 참여하는 다른 호스트들의 상태에 관한 정보를 언제 수집할 것인지, 어디에 수집을 해야 하는지, 어떠한 정보를 수집해야 하는지를 결정하는 정보(information) 정책이 필요하다.

3. 기존의 연구

기존의 부하 공유 기법들 중 가장 기본적이고 잘 알려져 있는 기법으로는 입찰(bidding) 기법이 있다[7]. 이 기법에서는 프로세스를 이주시키고자 하는 호스트에서 프로세스 이주 의도를 전 호스트로 방송(broadcast)하게 된다. 이 메시지를 받은 다른 호스트들은 자신의 상태와 제공할 수 있는 자원들의 여건들을 메시지에 담아 응답하게 된다. 프로세스를 이주시키려는 호스트는 일정 시간 동안만 응답 메시지를 수신하고, 수신된 메시지들을 분석하여 가장 최적의 호스트를 선택하며 선택된 호스트에게 선택되었음을 알리고, 이주를 시작한다. 그리고 다른 호스트들에게는 탈락되었음을 알린다. 이 기법은 알고리즘은 간단하나 이주시킬 때마다 방송(broadcast)을 해야 하는 오버헤드가 있다. 이로 인한 시간 지연 때문에 호스트의 상태가 변경되어 올바르지 못한 결정을 내릴 수가 있다.

Ni가 제안한 분산 Drafting 알고리즘은 입찰 방식과는 반대로 동작한다[8]. 이 기법에서 호스트들은 세 가지 상태 변화를 한다. 호스트의 상태에는 부하가 심하게 걸린 상태(heavy loaded), 보통의 수준인 상태(normal loaded), 부하가 적게 걸린 상태(light loaded)가 있다. 각 호스트들은 인접 호스트들의 상태를 기록하는 프로세서 부하 테이블을 가지고 있으며 어떤 호스트가 상태 변경을 하면 인접 호스트들에게 알린다. 이 메시지를 수신한 호스트는 해당 부하 테이블을 갱신시킨다. 각 호스트들은 정기적으로 자신의 상태를 조사하다가 자신의 상태가 저부하 상태(light load)

로 인지되면 자신의 프로세서 테이블에 있는 과부하 상태의 호스트들에게 프로세스를 자신에게 이주시킬 것(드래프트 요구)을 요청한다. 이 메시지를 받은 과부하 상태의 호스트들은 자신의 상태가 여전히 과부하 상태이면 자신의 드래프트-나이(draft-age)를 전달하며, 드래프트-나이를 받은 호스트는 수신한 드래프트-나이 중 가장 높은 값을 가진 호스트에게 자신에게 작업을 보낼 것은 제시하고, 이어서 프로세스 이주 단계로 들어간다. 이 기법에서는 호스트가 상태를 변경한 후 일정 기간 동안 그 상태가 유지된 경우에만 상태 변경 사실을 인접 호스트에게 알림으로서 급격한 상태 변경을 예방한다. 그러나 이 기법은 자신의 인접 호스트들에게만 드래프트 요청 및 상태 변경을 알리므로 네트워크 토폴로지에 영향을 많이 받게 되며 시스템 전체의 상태가 반영되지 않고 국부적으로 반영되는 단점을 가지고 있다.

Lin은 그래디언트(gradient) 모델에 기반한 부하 공유 기법을 제안하였다[9]. 각 호스트는 자신과 바로 연결된 호스트들과 정보를 교환한다. 여기서도 호스트들의 상태는 저부하(light), 보통(moderate), 과부하(heavy)의 세가지로 나누어진다. 각 호스트들은 저부하 상태 호스트로 부터의 최소 거리값을 가진다. 저부하 호스트는 0값을 가지고 인접 호스트가 저부하 상태가 아니면 인접 호스트의 거리값은 1을 더한다. 그리고 자신의 값을 다른 인접 호스트들에게 전파하고 이 전파를 받은 호스트는 자신이 저부하 상태가 아니면 그 값에 1을 더하고 인접 호스트에게 전파한다. 저부하 상태면 자신의 값을 0으로하고 모든 인접 호스트에게 전파한다. 이런 전파과정을 하면 자신의 거리값은 저부하 상태의 호스트로부터의 최소 거리값이 된다. 자신의 부하가 과부하(heavy) 상태가 되면 프로세스를 자신의 거리값보다 작은 호스트에게 전달하고 프로세스를 받은 호스트는 다시 인접 호스트 중 자신의 거리값보다 작은 호스트에게 전달하게 된다. 이를 반복하면 종국에 가서는 거리값이 0인 호스트, 즉 저부하 상태의 호스트에 도달하게 된다. 이주 프로세스는 이곳에서 실행된다. 거리값은 현 호스트로 부터 저부하 호스트까지의 최소 거리가 된다. 따라서 프로세스를 이주시킬 때 최단 경로로 이주시키게 된다. 이 기법은 인접 호스트들의 거리 값과 부하를 자신과 비교해야 하므로 부하가 변경될 때 인접 호스트들의 상태를 메시지로 가져와야 하고 계산 결과를 인접 호스트들에게 전파시켜야 한다. 이러한 행동이 모든 호스트들에서 발생하게 되므로 메시지의 양이 굉장히 늘어나게 된다. 그리고 이주하던 프로세스가 전파되는 도중 여러 호스트들이 상태 변경을 심하게 발생시키면 네트워크를 떠돌 가능성이 있다.

위에 언급한 기법들은 모두 분산 된 기법을 사용한다. 이들과는 달리 Theimer는 프로세스 이주 결정을 내리는 특수한 전담 호스트를 설정하여 부하 공유를 하는 집중화된(centralized) 기법을 제안하였다[10]. 이 전담 호스트를 제외한 다른 호스트들은 자신의 상태 변경을 이 전담 호스트에게 수시로 보고하게 되고, 과부하 상태의 호스트는 이 전담 호스트에게 이주 요청을 보내게 된다. 전담 호스트가 이주 요청을 받으면 호스트들의 상태 정보를 조사하여 가장 작은 부하의 호스트의 주소를 이주 요청을 한 호스트에게 보내주게 된다. 전담 호스트는 고정되어 있으므로 전담 호스트에서 메시지의 병목 현상으로 인하여 올바르지 못한 결정을 내릴 수 있다. 그리고 결함 허용(fault tolerancy)에도 적절히 대응하지 못한다.

이 외에도 [11]에서는 순찰(visitor) 메시지를 사용하고 있는데, 이 메시지는 분산 시스템의 평균 부하값과 최소 부하 노드, 최대 부하 노드에 대한 정보를 수집하며 논리적으로 구성된 링을 순환하게 된다. 순찰 메시지를 받은 호스트는 자신의 부하값과 메시지의 내용을 분석, 비교하여 부하의 분산 정책을 수행하게 된다.

트리 구조상에서의 부하의 분산 기법도 제시되어 있다[12]. 이 기법 역시 수학적인 이론을 바탕으로 분산 정책을 세운 정적인 기법에 속한다. 이 기법에서는 작업의 할당 문제를 네트워크 플로우(network flow)의 문제로 변환하였으며 분할 트리(cut tree)를 이용하여 최적의 할당을 구하고 있다. 이 기법의 복잡도는 네트워크 플로우(network flow) 알고리즘의 복잡도 및 이의 적용 횟수에 의존한다. 또한 스타형 컴퓨터 네트워크를 기반으로 하는 정적 부하 균형 기법도 제시되어 있는데[13] 이 기법에서는 스타형 네트워크의 특성인 중앙 노드에 대해 별도로 인식하지 않고 과부하 노드로부터 저부하 노드로 작업을 이주시키는 방법을 제시하였다.

부하 공유 기법에서는 노드들의 상태 정보를 얻기

위한 메시지의 양이 성능에 영향을 미치게 되는데, 이 메시지의 양을 줄이기 위해 각 프로세서에게 송신 집합을 부여하고 이 집합에 속해 있는 프로세서들에게만 부하정보를 보내는 기법도 제안되어 있다[14]. 이 기법은 프로세서의 수가 N일 때 메시지의 수가 $\sqrt{N}$ 이 되도록 하고 있다.

과부하 호스트와 저부하 호스트 모두가 모니터 호스트를 통하여 작업의 이주를 요청할 수 있는 하이브리드 알고리즘도 제시되어 있는데[15], 이 기법에서 상태 정보의 교환은 각 그룹간에만 이루어지며 모니터 호스트가 이를 감시하게 된다. 수신자 그룹의 모니터 호스트는 작업의 이주를 받게 되며 이를 수행할 적절한 수신자가 없다면 자신이 이를 처리하고 그 결과를 송신자에게 반환하게 된다.

하이퍼큐브 형태의 시스템에서 노드의 고장을 고려한 부하 공유 기법[16]과 결함 내성 다중 컴퓨터 시스템에서의 프로세스 할당 기법[17, 18] 등도 제안되어 있으며, 확률 분포를 이용하여 분산 시스템에서의 부하 공유에 대한 잠재성(potential)을 제안한 결과가 발표되어 있다[19].

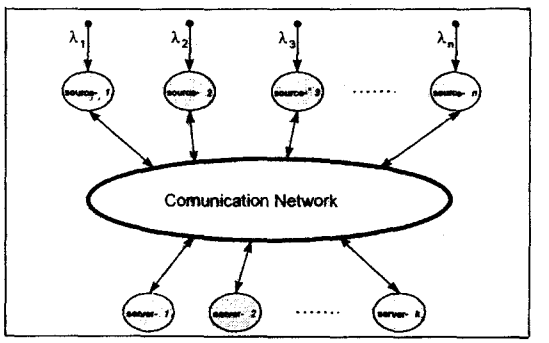
본 논문에서와 같이 여러개의 임계값(threshold value)들을 사용하여 각 노드의 부하를 구분하고 이를 이용하여 부하공유를 이루도록 하는 기법도 제안되어 있으며[20], 다만 이는 분산 실시간 시스템을 대상으로 하여 전체 시스템이 각 태스크들의 제한시간을 맞추는 것을 목적으로 하고 있다. 이 기법은 각 노드의 부하 상태를 해당 노드의 큐 길이에 따라 저부하(underloaded), 중부하(medium-loaded), 만부하(fully-loaded), 고부하(overloaded) 등으로 나누어 부하 이동의 기준으로 삼고 있다.

4. 제안 기법

4.1 시스템 모델

네트워크에 연결되어 있는 호스트들은 자신의 부하 값에 따라 서버와 소스로 구분되어진다(그림 3). 소스는 부하가 많이 걸려 자신의 작업을 다른 호스트로 이주시켜 실행시켜야 하는 호스트이다. 서버는 부하가 적게 걸린 저부하 상태의 호스트로서 프로세스를 이주시킬 때 목적지로 선정될 수 있다[21].

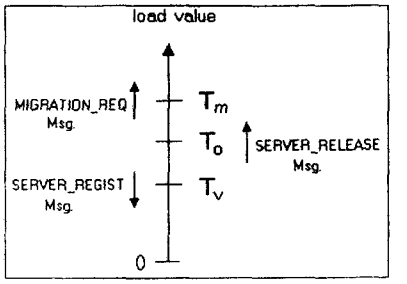
본 연구에서는 각 호스트들의 상태 변화에 관한 정보를 수집하고 프로세스 이주가 필요한 경우 그 목적지를 선정해 주는 중개자 호스트들을 네트워크에 추가한다. 중개자 역할을 수행하게 될 호스트들은 시스템이 초기화될 때 임의의 규칙에 따라 선정되고, 그 이후 중개자 호스트들은 시스템의 상태에 따라 동적으로 그들의 역할을 인계할 수 있게 되며, 시스템 전체적으로는 부하에 따라 소스, 서버, 중개자들이 서로 협력하여 동작하게 된다. 본 제안 기법에서는 네트워크상에서 모든 메시지들이 분실되지 않고 반드시 전송되는 것으로 가정한다.



(그림 3) 네트워크 구성
(Fig. 3) Network Configuration

(1) 서버와 소스를 구분하는 임계값과 그 특징

각 호스트들은 정기적으로 자신의 부하(load) 값을 조사하게 되는데, 자신의 부하를 미리 정해진 임계값(threshold)과 비교하여 상태 변화를 하게 되고, 자신의 상태에 맞는 행동을 취하게 된다. (본 논문에서의 부하는 각 호스트의 작업 큐의 길이로 설정한다.) 상태 변화의 기준이 되는 임계값은 다음과 같이 세가지로 나누어진다(그림 4).



(그림 4) 3가지 임계값과 메시지
(Fig. 4) Three threshold values and messages

T_v: 호스트의 부하가 이 값 보다 낮아지면 서버가 된다.
T_o: 호스트의 부하가 이 값을 넘어서면 소스가 된다.
T_m: 호스트의 부하가 이 값을 넘어서면 프로세스를 이주시킬 수 있다.

시스템이 초기화될 당시에는 각 호스트들에 작업이 없으므로 모든 호스트들을 서버로 운영된다. 각 호스트들은 자신이 서버임을 인지하면 먼저 자신이 서버임을 중개자 호스트에게 알리는 중개자 등록 절차를 수행한다. 각 호스트들은 현재 시스템에 어떠한 호스트들이 중개자인지를 저장하는 중개자 리스트를 가지고 있다. 그리고 현재 그 중개자에 등록된 서버가 있는지 없는지를 알 수 있는 플래그도 가진다. 서버 등록 절차에서 각 호스트들은 먼저 이 중개자 리스트를 보고 자신의 중개자를 하나 선정하게 된다. 중개자를 선정할 때 서버가 등록되어 있지 않은 중개자를 우선 선택 대상으로 삼는다. 만약 모든 중개자에게 서버들이 등록되어 있다면 그들 중 임의의 중개자를 하나 선택하여 자신의 중개자로 선정한다. 중개자를 선정하였다면 SERVER_REGIST 메시지를 자신의 중개자에게 전송하게 된다. 이후 자신의 중개자는 프로세스 이주의 목적지로 이 서버를 선택할 수 있게 된다.

사용자의 이용이 증가하여 호스트의 부하가 증가하여 임계값 T_o를 넘어서면 서버에서 소스로 상태를 변화하게 된다. 자신이 소스가 된 것을 인지하면 이 사실을 자신의 중개자에게 알리는 서버 해제 절차를 수행하게 된다. 서버 해제 절차에서 소스는 SEVER_RELEASE 메시지를 자신의 중개자에게 전송하게 된다.

소스의 부하가 계속해서 증가하여 임계값 T_m을 넘어서면 자신의 작업을 다른 호스트로 이주시켜 수행할 수 있게 된다. 이때 임의의 중개자를 선정하여 MIGRATION_REQ 메시지를 전송하여 자신의 프로세스 이주 의도를 알리고 프로세스 이주 절차를 수행하게 된다.

시간이 지나 사용자의 이용이 줄어들어 호스트의 부하가 T_v 보다 낮아지면 서버로 상태 변화를 하게 되고 서버 등록 절차를 수행하게 된다. 각 호스트에서의 이러한 서버 등록, 해제 및 프로세스 이주와 관련한 과정을 다음의 (알고리즘 1)에서 보인다.

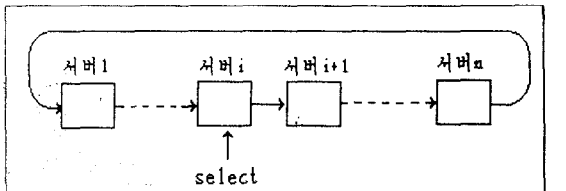
(알고리즘 1) 상태변화에 따른 호스트의 동작

```
void Node::E_check(void)
{
   if (play == SERVER) {
      if (load > Tsource) { // 임계값 To
         E_release(); // SEVER_RELEASE 전송
      }
   }
   else {
      if (load > Tmigration) { // 임계값 Tm
         E_request_migration(); // MIGRATION_REQ 전송
      }
      else if (load < Tserver) { // 임계값 Tv
         E_regist(); // SERVER_REGIST 전송
      }
   }
}
```

위에서 본 것과 같이 서버에서 소스로, 소스에서 서버로의 상태 변화는 두 가지 임계값 T_o와 T_v에 의해서 결정된다. 임계값 T_o와 T_v를 통하여 부하 변동의 완충 역할을 하게 하였다. 만약 단일 임계값을 사용한다면 그 임계값 부근에서 부하가 심하게 변동하는 경우 상태 변화도 빈번하게 일어나는 스래싱(thrashing)이 발생하게 된다. 본 기법에서는 임계값 T_o와 T_v를 두어 빈번한 상태 변화에 대비한 완충 작용을 하면서 스래싱을 방지한다.

(2) 중개자의 설정과 특징

중개자에서는 자신에게 등록된 서버들을 관리한다. 이를 위해서 자신에게 등록된 서버들을 서버 환형 연결 리스트로 관리한다(그림 5). 임의의 호스트로부터 SERVERR_EGIST 메시지를 받으면 그 호스트를 자신이 관리해야 할 서버로 간주하고 자신의 서버 환형 연결 리스트에 추가하며 SEVER_RELEASE 메시지를 받으면 해당 호스트를 서버 환형 연결 리스트에서 제거한다. 임의의 소스에서 부하가 T_m을 넘어서서 중개자에게 MIGRATION_REQ 보내게 되면 이



(그림 5) 서버 환형 연결 리스트
(Fig. 5) Circular linked list of servers

러한 프로세스 이주 요청을 받은 중개자는 자신의 서버 환형 연결 리스트에서 순환적으로 서버를 선정하게 된다.

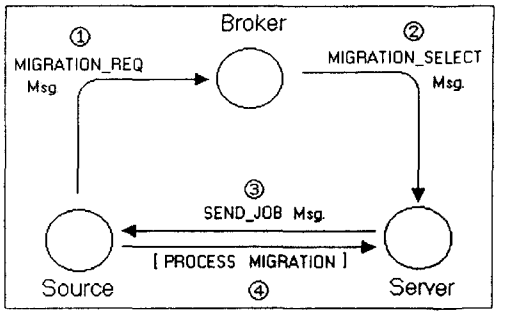
중개자는 일반 호스트와 같이 모든 서비스를 제공하면서, 소스와 서버를 연결해 주는 역할도 하게 된다. 등록되어 있던 서버들이 모두 해제되어 서버 환형 연결 리스트가 비게되면 자신에게 등록된 서버가 없다는 것을 알리기 위해 SEVER_EMPTY 메시지를 모든 호스트들에게 방송(broadcast)하게 된다. 이렇게 함으로써 자신에게 프로세스 이주 요청이 오는 것을 막고, 서버가 등록할 때 자신에게 우선적으로 등록하도록 한다. SEVER_EMPTY 메시지를 받은 호스트들은 자신이 가지고 있던 중개자 리스트에 이를 반영하고 나중에 서버로 등록할 때 이들 중개자를 우선적으로 선정한다. 등록된 서버가 없다가 서버 등록을 받게 되면 SEVER_NOTEMPTY 메시지를 방송하여 등록된 서버가 있음을 알리고, 이후부터 소스들은 이 중개자에게 프로세스 이주 요청 보낼 수 있게 된다. 중개자 노드들이 수신된 메시지를 처리하는 방법은 (알고리즘 2)에 나타나 있다.

(알고리즘 2) 중개자에게 수신된 메시지에 대한 처리 알고리즘

```
void Node::B_receive(void)
{
   switch(rmsg.request) {
      case SERVER_REGIST :
         B_regist();
         break;
      case SERVER_RELEASE :
         B_release();
         break;
      case MIGRATION_REQ :
         id = B_select();
         Packmsg(MSG, MIGRATION_SELECT, id, 10);
         break;
      case BROKER_SWITCH :
         broker->Change(oldbroker.id, newbroker.id);
         broker->Setstate(newbroker.id,(SEMPTY)newbroker.state);
         break;
      case SERVER_EMPTY :
         broker->Setstate(rmsg.senderid, EMPTY);
         break;
      case SERVER_NOTEMPTY :
         broker->Setstate(rmsg.senderid, NOTEMPTY);
         break;
   }
}
```

4.2 프로세스 이주 절차

소스 호스트가 자신의 부하를 조사하였을 때 부하가 T_m을 넘어서면 프로세스 이주 요청 절차에 들어가게 된다. 먼저 자신이 가지고 있는 중개자 리스트를 조사하여 등록된 서버가 있는 중개자들 중에서 임의의 중개자를 하나 선정하며 그 이후 다음의 절차를 거쳐 프로세스를 이주시키게 된다(그림 6).



(그림 6) 프로세스 이주 절차
(Fig. 6) Process migration procedure

① 선정된 중개자에게 MIGRATION_REQ 메시지를 보낸다. 이 메시지를 받은 중개자는 자신의 서버 환형 연결 리스트에서 다음 선택 대상을 선정한다.
② 선택된 서버에게 MIGRATION_SELECT 메시지를 보냄으로써 이주의 목적지로 선정되었음을 알리고 소스의 ID도 알려준다.
③ 서버는 소스에게 SEND_JOB 메시지를 전달한다.
④ 소스는 자신의 큐에서 이주시킬 프로세스를 선정하고 이를 제거한 후 서버로 이주시킨다.

소스가 이주시킬 프로세스를 선정할 때에는 작업 큐에 있는 모든 프로세스들을 검사하며 대기 시간이 프로세스 이주에 드는 오버헤드 보다 큰 프로세스를 선정한다. 이와 관련한 프로세스 선택 정책의 구체적인 방법은 다양하게 나타날 수 있을 것이다.

4.3 중개자 역할 인계 방법

중개자 호스트 자신도 자신의 작업들을 서비스해 가며 서버와 소스에게 프로세스 이주 중개 서비스를 해주게 되며, 따라서 중개자 자신도 부하가 증가하여 과부하 상태로 변경될 수 있을 것이다. 중개자가 과부하 상태에서 프로세스 이주 중개 서비스를 한다는 것은 부적당하며, 이 경우 중개자 자신 또한 프로세스를 다른 호스트로 이주시키는 것이 바람직할 것이다. 따라서 이때는 자신의 중개자 역할을 다른 호스트가 넘겨받도록 해야 하며 이를 위해서 중개자 자신도 정기적으로 자신의 부하를 조사하여 상태 변화를 인지해야 한다. 자신의 부하가 T_o를 넘어서면, 즉 소스로 상태 변화가 일어나면 더 이상 중개자로서 활동

하지 않는다. 이 경우 중개자는 서버 환형 연결 리스트에서 자신의 역할을 인계할 서버를 선정한다. 중개자 역할을 인계할 서버가 선정되면, 중개자는 BROKER_SWITCH 메시지를 모든 호스트들에게 방송하여 중개자 역할 인계가 발생했음을 알린다. 그리고 자신이 가지고 있던 서버 환형 연결 리스트와 프로세스 이주 요청과 관련된 메시지 큐를 새로운 중개자에게 전달하며 자신은 소스 상태가 된다. BROKER_SWITCH 메시지를 새로운 중개자가 아닌 이전 중개자가 방송하는 것은 중개자가 변경된 사실을 좀 더 빨리 알리기 위해서이다. BROKER_SWITCH 방송을 받은 호스트들은 자신의 중개자 리스트를 갱신하되, 이전 중개자를 자신의 중개자로 등록한 서버들은 새로운 중개자를 자신의 중개자로 바꾸게 된다. 중개자 역할 인계 과정은 (알고리즘 3)에서 보이고 있으며, 중개자 역할 인계 사실을 전달받은 각 호스트에서의 처리 과정은 (알고리즘 4)에서 보인다.

(알고리즘 3) 중개자 역할 인계 알고리즘

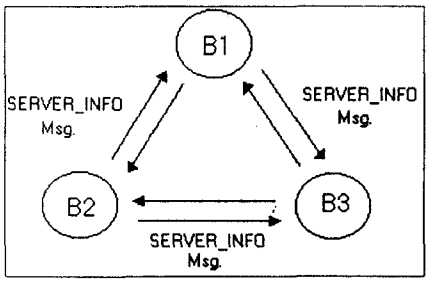
```
void Node :: B_check(void)
{
   if (load > Tsource) {
      svt->Select();
      Packmsg(BROADCAST, BROKER_SWITCH, newbroker.id, 10);
      svt->Release(newbroker.id);
      broker->Change(host_id, newbroker.id);
      broker->Setstate(newbroke.id, newborker.state);
      play = SOURCE;
      load_check = 0;
   }
}
```

(알고리즘 4) BROKER_SWITCH 방송 수신 처리 알고리즘

```
void Node::E_switch(void)
{
   broker->Change(oldbroker.id, newbroker.id);
   broker->Setstate(newbroker.id, newbroker.state);
   if (play == SERVER && mybroker == oldbroker.id) {
      mybroker = rmsg.receiveid;
   }
   ...
}
```

4.4 중개자들 간의 통신

분산 시스템은 결함(fault)을 허용하면서 시스템의 안전성을 보장해 주어야 한다. 이를 위해 중개자 호스트가 고장을 일으킨 경우 적절히 대처하여야 한다. 결함 허용(fault tolerancy)을 보장하기 위해 중개자 호스트들은 정기적으로 상호간 통신을 한다. 메시지에는 자신이 관리하고 있는 서버 환형 연결 리스트를 담아 다른 중개자에게 SEVER_INFO 메시지를 보내게 된다(그림 7).



(그림 7) 중개자들 간의 통신
(Fig. 7) Communications between brokers

일단 다른 중개자가 고장을 일으킨 것을 감지하면 우선 순위가 높은 중개자가 고장이 발생한 중개자에서 관리하던 서버 환형 연결 리스트로부터 서버를 하나 선정하여 중개자 역할을 대행하게 하고, 서버 환형 연결 리스트를 넘겨준다.

4.5 메시지 분실에 대한 해결 기법

모든 메시지는 분실됨 없이 전송됨을 가정한다. 그러나 중개자 역할 인계 절차를 통해 중개자가 바뀌는 과정에서 중개자에게 가는 메시지가 뜻하지 않은 결과를 초래할 수 있다. 앞에서 언급했듯이 중개자의 부하가 T_o를 넘어서면, 중개자 역할 인계 절차를 수행하게 된다. 중개자 역할 인계 과정에서 이 사실을 모르는 호스트들은 이전 중개자에게 계속해서 메시지를 보내게 된다. 따라서 중개자 역할 인계 과정이 시작된 이후부터 BROKER_SWITCH 방송을 수신할 때까지, 이 시간 간격 동안 호스트들로부터 이전 중개자에게 전송된 메시지들에 대한 처리가 필요하게 된다.

통신 중이던 객체(object)의 이동으로 인한 메시지 분실에 대한 처리는 일반적으로 세 가지 기법이 사용된다[22].

(1) 메시지 방향 재설정(message redirection)

이동 전의 호스트에서 이동 후의 호스트 주소를 기억하고 있다가 수신되는 메시지를 이동후의 호스트에게 전달 해주며, 다른 호스트들은 이동한 사실을 알지 못하고 계속해서 이전 호스트로 메시지를 보내게 된다.

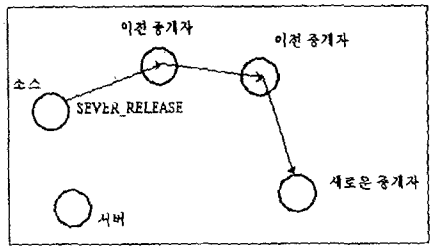
(2) 메시지 분실 방지(message loss prevention)

이동 전 호스트에서 수신된 메시지를 버퍼에 저장하였다가 이동 후의 호스트로 버퍼를 전달해 주게 되고 이동이 끝난 경우 다른 호스트에게 주소 변경을 알리는 방법이며 다른 호스트에서는 중개자 이동 사실을 인지한 후 메시지를 이동 후의 호스트로 전송하게 된다.

(3) 메시지 분실 복구(message loss recovery)

이동 과정에서 전달된 메시지를 모두 무시해 버린다. 그리고 이동이 완료된 후 새로운 주소를 다른 호스트로 알려 준다. 새로운 주소를 받은 호스트들에서 무시된 메시지를 새로운 주소로 다시 전송한다.

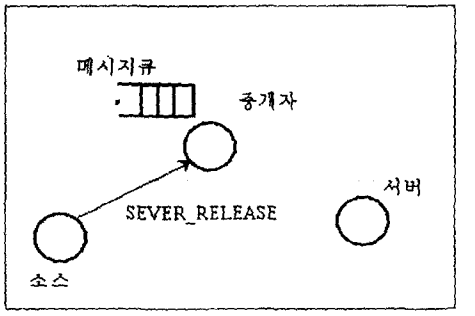
메시지 방향 재설정을 본 연구에 적용하면 메시지는 중개자 역할 인계 경로를 따라 전달되야 한다(그림 8). 하지만 이 기법은 이동의 횟수가 많을 수록 전달 경로가 길어지게 되므로 메시지 전송과 처리가 늦어지므로 부적당하다.



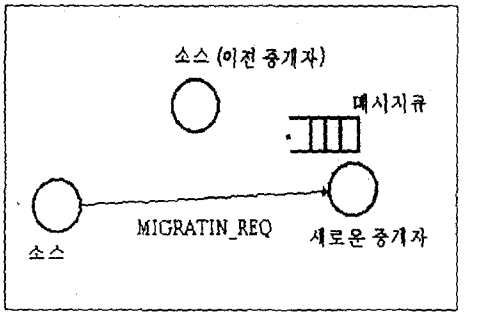
(그림 8) 메시지 방향 재설정을 사용한 예
(Fig. 8) Examples of message redirection

본 연구에서는 메시지 분실 방지와 메시지 분실 복구 두 가지 기법을 같이 사용한다. 중개자 역할 인계가 시작되기전 이전 중개자에게 온 메시지중 처리되지 않은 메시지는 모두 메시지 큐에 저장하여 새로운 중개자에게 전달해 주게 되고, 중개자 역할 인계가 완료된 후 BROKER_SWITCH 방송을 받은 호스트들은 앞으로 새로운 중개자의 주소로 메시지를 보내게 된다. 그리고 그 메시지들은 새로운 중개자에 의해 처리된다(그림 9).

(그림 9-a)는 중개자 역할 인계 시작 전에 서버에서 소스로 변경된 어느 한 호스트에서 자신의 중개자에게 SERVER_RELEASE 메시지를 보내는 경우를 보



(a) 중개자 역할 인계 시작 전



(b) 중개자 역할 인계 완료 후

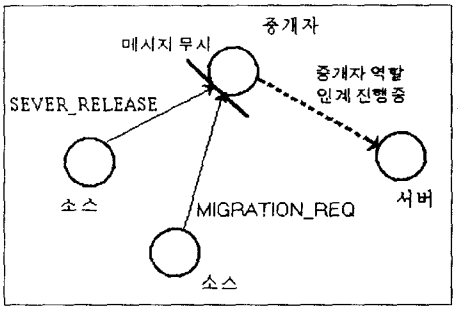
(그림 9) 메시지 분실 방지를 사용한 예
(Fig. 9) Examples of message loss prevention

여준다. 중개자는 이 메시지를 메시지 큐에 추가하고 앞서 도착한 메시지를 처리한다. (그림 9-b)는 중개자 역할 인계 후 메시지 큐가 전달되고 소스는 새로운 중개자에게 MIGRATION_REQ메시지를 전송하는 경우를 보여준다.

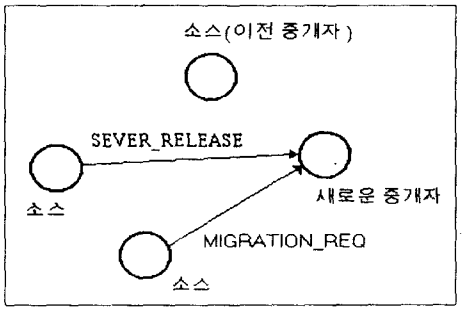
그러나 중개자 역할 인계가 시작된 후부터 다른 호스트들이 중개자 역할 인계 방송을 받기 전까지의 과정에서 이전 중개자에게 보낸 메시지는 모두 무시해 버린다. 이 경우 각 호스트에서는 중개자 역할 인계가 끝나고 BROKER_SWITCH 방송을 받은 후 새로운 중개자에게 메시지를 다시 보낸다(그림 10).

(그림 10-a)는 중개자 역할 인계 과정 중에 이전 중개자로 두개의 소스가 각각 SERVER_RELEASE 메시지와 MIGRATION_REQ 메시지를 전송하였으나 모두 무시됨을 보여주고 있으며, (그림 10-b)는 중개자 역할 인계가 끝나고 호스트들이 새로운 중개자 주소를 받게되는 경우 새로운 중개자로 메시지를 다시

전송하고 새로운 중개자가 이 메시지들을 처리하는 상황을 보여준다.



(a) 중개자 역할 인계 진행중



(b) 중개자 역할 인계 후

(그림 10) 메시지 분실 복구를 사용한 예
(Fig. 10) Examples of message loss recovery

5. 성능 평가

시뮬레이션 모델에서 각 호스트들에 대한 큐잉 모델은 M/M/1 모델로 하며 따라서 스케줄링 기법은 FIFO 스케줄링을 가정한다. 일반적인 메시지 전송과 수신에 드는 시간을 각각 10ms로 가정하고, 프로세스를 이주하는 데 있어서 소스 측에서 프로세스를 보내는데 100ms, 프로세스를 수신하여 이주 전의 프로세스 상태로 만들어 주는데 100ms의 시간을 소요하는 것으로 가정한다.

시스템 활용도(utilization) ρ는 아래와 같다.

$\rho = \lambda S$

where λ = "interarrival rate"

S = mean service time

평균 도착 시간 간격을 1 sec로 두고 S를 변경하여 $0.1 \le \rho \le .95$ 구간에서의 평균 응답 시간(단위:ms)과 큐의 길이로 본 제안 기법의 성능을 평가하였다. 이후 그림이나 도표에 M/M/1으로 나타난 경우는 같은 환경에서 부하 공유 기법을 사용하지 않은 경우를 의미한다.

5.1 임계값 설정

임계값의 설정을 위하여 부하 공유 기법을 사용하지 않은 경우(M/M/1)의 평균 큐 길이를 계산해 본다. 본 모델을 기반으로 한 시뮬레이션 결과 부하 공유 기법을 사용하지 않은 경우(M/M/1)의 평균 큐 길이는 (표 1)과 같이 나타났다.

(표 1) 평균 큐 길이
(Table 1) Mean queue length

ρ	M/M/1	ρ	M/M/1
0.1	1.11	0.7	3.26
0.2	1.24	0.75	4.13
0.3	1.45	0.8	4.86
0.4	1.63	0.85	6.93
0.5	2.01	0.9	10.13
0.6	2.34	0.95	16.3
0.65	2.84		

본 기법에서 사용하는 임계값은 앞에서 설명한 바와 같이 T_o, T_v, T_m의 세가지가 있으며 부하 공유 기법을 사용하지 않은 경우의 호스트의 평균 큐 길이를 기반으로 하여 다음과 같이 임계값을 설정한다.

(1) $0.1 \le \rho < 0.5$

$0.1 \le \rho < 0.5$ 구간에서 평균 큐의 길이는 2 이하이다. 이 구간에서 큐의 길이가 1이면 소스가 되고 2가 되면 프로세스를 이주시키며 0인 경우에만 서버로서 활동하도록 한다.

(2) $0.5 \le \rho < 0.65$

$0.5 \le \rho < 0.65$ 구간에서는 평균 큐의 길이가 3 이

하이다. 큐 길이가 2가 되면 소스가 되고 3이상이면 프로세스를 이주시키도록 하며 1이하가 되면 서버로서 활동하도록 한다.

(3) $\rho = 0.7$

$\rho = 0.7$인 경우 호스트의 수가 15개인 경우에는 (표 2)에서 보이듯이 임계값 $T_o/T_v/T_m$를 1/2/3으로 하는 것이 가장 좋은 성능을 나타냄을 알 수 있다.

(표 2) $\rho = 0.7$, 노두 수 15개인 경우 임계값과 평균 큐 길이
(Table 2) Threshold values and mean queue lengths in case of $\rho = 0.7$ and 15 nodes

임계값	중개자1	중개자2	중개자3	중개자4
0.5/1/1.5	4.42	3.14	3.26	3.30
1/2/3	2.71	2.93	3.01	3.15
2/3/4	2.91	2.93	3.02	3.15

(4) $\rho = 0.8$

$\rho = 0.8$인 경우에는 호스트의 수가 40개인 경우 (표 3)에서와 같이 임계값 $T_o/T_v/T_m$를 6/8/10으로 하는 것이 가장 좋은 성능향상을 나타냄을 알 수 있다.

(표 3) $\rho = 0.8$, 노드수 40개인 경우 임계값과 큐 길이
(Table 3) Threshold values and mean queue lengths in case of $\rho = 0.8$ and 40 nodes

임계값	중개자1	중개자2	중개자3	중개자4
1/2/3	7.29	3.88	4.09	4.27
2/3/4	5.94	3.73	3.89	3.84
4/5/6	6.2	4.0	4.04	4.0

(5) $\rho = 0.9$

$\rho = 0.9$인 경우에는 호스트수가 80개인 경우 (표 4)에서와 같이 임계값 $T_o/T_v/T_m$를 6/8/10으로 하는 것이 가장 좋은 성능 향상을 나타냄을 알 수 있다.

(표 4) $\rho = 0.9$, 노드수 80개인 경우 임계값과 큐 길이
(Table 4) Threshold values and mean queue lengths in case of $\rho = 0.9$ and 80 nodes

임계값	중개자1	중개자2	중개자3	중개자4
5/7/9	8.24	7.2	6.72	6.61
6/8/10	6.56	6.38	6.47	6.52
7/9/11	7.12	6.78	6.94	6.79

위에서 보듯이 부하 공유 기법을 사용하지 않았을 때의 평균 큐 길이 값과 비슷한 값을 T_o 값으로 주는 것이 성능을 좋게 함을 알 수 있으며, ρ의 값이 0.9, 0.95인 경우는 그 보다는 다소 낮은 값을 사용하는 것이 성능을 최적화시킴을 알 수 있다. 위의 결과를 바탕으로 하여 시뮬레이션에 사용된 임계값($T_o/T_v/T_m$)은 (표 5)에서 보이는 바와 같이 설정하였다.

(표 5) 각 ρ값에 대한 임계값
(Table 5) Threshold values for each ρ values

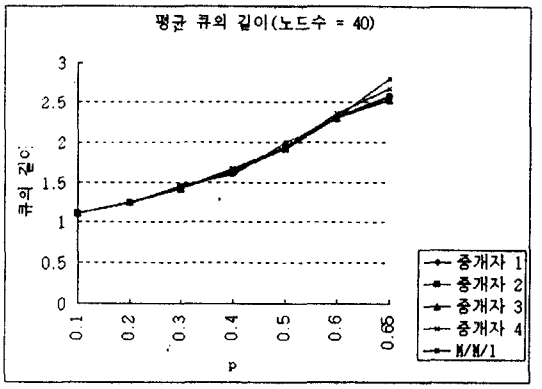
ρ	임계값
0.1~0.5	0.5/1/1.5
0.6~0.75	1/2/3
0.8	2/3/4
0.85	4/5/6
0.9	6/8/10
0.95	8/10/12

5.2 저부하 상태에서의 성능 평가

본 제안 기법이 저부하 상태에서 어느 정도의 성능을 보이는가를 판단하기 위해 ρ의 값이 0.1 부터 0.65까지의 값을 갖는 구간에서 시뮬레이션을 수행하였으며 그 결과를 (그림 11)에서 보이고 있다. 시스템의 호스트의 수를 40개로 설정하였으며, 중개자의 수가 1/2/3/4인 경우 각각에 대하여 ρ의 값이 변화함에 따라 평균 큐의 길이가 어떻게 변화하는지를 보였다. (그림 11)에서 M/M/1으로 표시된 경우는 부하 공유를 하지 않은 경우를 의미한다.

비교의 대상이 되는 부하 공유를 하지 않는 경우의 평균 큐의 길이가 1부터 3까지의 값을 가짐을 알 수 있으며, 이와 같이 큐의 평균 길이가 작은 경우에는

큐의 길이로 임계값을 결정하는 본 제안 기법에서는 성능 향상이 크게 나타나지 않음을 알 수 있다. 즉, 본 기법을 사용하는 경우 시스템의 부하가 작은 경우에는 큰 성능 향상을 나타나지 않으며 이는 일반적으로 합당한 결과라 할 수 있을 것이다.

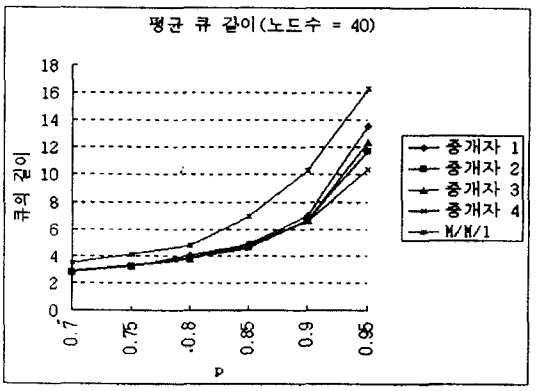


(그림 11) 저부하 상태인 경우의 평균 큐의 길이 (호스트의 수 = 40)

(Fig. 11) Mean queue length of lightly loaded case (40 hosts)

5.3 과부하 상태에서의 성능 평가

본 제안 기법이 과부하 상태에서는 어느 정도의 성능을 보이는가를 판단하기 위해 ρ의 값이 0.7부터 0.95까지의 값을 갖는 구간에서 시뮬레이션을 수행해 보았으며 그 결과를 (그림 12)에서 보이고 있다. 이 경우에도 시스템의 호스트 수를 40개로 설정하였으며, 중개자의 수가 1/2/3/4인 경우 각각에 대하여 ρ의 값이 변화함에 따라 평균 큐의 길이가 어떻게 변화하는지를 보였다. (그림 12)에서도 M/M/1으로 표시된 경우는 부하 공유를 하지 않은 경우를 의미한다.



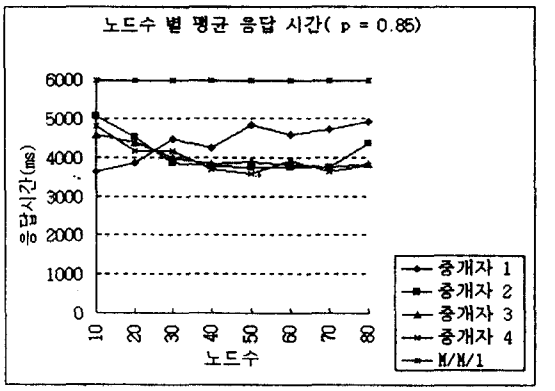
(그림 12) 과부하 상태인 경우의 평균 큐의 길이 (호스트의 수 = 40)

(Fig. 12) Mean queue length of heavily loaded case (40 hosts)

이 경우에는 ρ의 값이 상대적으로 작은 (0.7, 0.8)의 구간에서는 약 10% 정도의 성능 향상이 나타나고 있으며, ρ의 값이 0.85 이상으로 큰 구간에서는 작게는 25%에서부터 크게는 38% 까지의 성능 향상이 나타나고 있음을 알 수 있다.

5.4 호스트 수의 증가에 따른 평균 응답 시간

호스트 수의 증가에 따라 시스템의 평균 응답 시간이 어떻게 변화하는지를 파악하기 위해 ρ의 값이 0.85인 경우에 대해 시뮬레이션을 수행하였으며 그 결과를 (그림 13)에서 보이고 있다



(그림 13) ρ = 0.85인 경우 호스트 수와 평균 응답 시간

(Fig. 13) Mean response time vs number of hosts (ρ = 85)

호스트의 수가 적은 경우에는, 중개자의 수가 적은 경우 약 38% 정도의 성능 향상을 보이고 있으며 중개자의 수가 많은 경우 약 18% 정도의 성능 향상을 보이고 있다. 또한 호스트의 수가 많은 경우에는, 중개자의 수가 적을 경우 약 17% 정도의 성능 향상이 나타나고 있으며 중개자의 수가 많은 경우 최대 38% 까지의 성능 향상이 나타나고 있다. 결국, 호스트의 수가 적은 경우에는 중개자의 수가 적을 수록 성능 향상이 좋게 나타나고, 호스트의 수가 증가하면 중개

자의 수도 어느정도 늘어나는 것이 성능에 좋은 영향을 미치고 있음을 알 수 있다.

종합해 볼 때 ρ의 값이 $0.7 \leq \rho \leq 9.5$인 구간에서는 시스템의 각 호스트에 가해지는 부하가 커지므로 평균 큐의 길이가 길어지게 되며, 이 경우에는 큐의 길이를 임계값으로 이용하는 본 기법이 평균 30% 정도 성능을 향상시키고 있음을 알 수 있다.

6. 결 론

본 논문에서는 다중 중개자를 기반으로 한 동적 부하 공유 기법을 설계하였다. 본 기법의 특징은 소스와 서버를 구분하기 위해 세 개의 임계값을 사용하여 상태 변화 스래싱을 방지하는 완충 작용을 하도록 한 것이다. 그리고 서버들에 관한 정보를 수집하고, 소스의 프로세스 이주 요청에 대한 목적지를 선정하기 위해 중개자라는 호스트를 두었으며, 중개자가 언제나 저부하 상태인 경우에서만 동작하도록 하기 위해 자신의 부하가 높아지면 저부하 상태의 서버에게 중개자 역할을 인계하여 중개자 호스트의 부담을 덜어 주도록 하였다.

시뮬레이션을 통한 성능 평가에서 볼 수 있듯이 본 기법은 ρ의 값이 작은, 즉 시스템의 부하가 작은 경우에는 큰 효과를 보여주지 않지만, ρ의 값이 증가하여 부하가 큰 경우에는 약 25%에서 38% 정도까지의 성능 향상을 보여주고 있다.

또한 ρ 값을 고정시킨 상황에서 전체 호스트의 수가 적을 때, 중개자의 수가 많은 경우에는 성능의 향상이 약 17% 정도에 불과한 반면, 중개자의 수를 적게 하는 경우에는 약 38% 정도의 성능 향상을 보이고 있다. 그리고 전체 호스트의 수가 상대적으로 많을 때, 중개자의 수를 적게 하는 경우에는 약 18% 정도의 성능 향상을 보이는데 반해, 중개자의 수를 늘려주는 경우에는 약 35% 정도의 성능 향상을 보이고 있다. 이는 ρ 값을 고정시킨 상황에서 전체적인 호스트의 수가 적을 때는 중개자 수가 많으면 통신량이 증가하게 되고 이것이 오히려 오버헤드로 작용하므로 중개자 수를 적게 하는 것이 평균 큐의 길이를 줄어들게 하며, 호스트의 수가 많은 경우에는 중개자의 수가 적으면 중개자로 오는 메시지가 많을 경우 제때 처리하지 못하므로 중개자 수를 늘려주는 것이 평균 큐의 길이를 줄어들게 함을 의미하는 것으로 분석된다.

본 기법에서는 중개자 역할을 할 호스트들은 시스템이 운영되는 동안 변화하나, 중개자의 수와 임계값들은 초기에 설정되면 시스템이 운영되는 동안 변화하지 않는다. 시스템의 운영중에도 부하에 따라 중개자의 수와 임계값들이 적절히 변화할 수 있도록 하여 최적의 성능 향상을 이룰 수 있도록 부하공유 기법을 설계하는 연구가 앞으로 이루어져야 할 것으로 보인다.

참 고 문 헌

[1] G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems: Concepts and Design, 2-ed., A-W, 1994.

[2] A. S. Tanenbaum and R. V. Renesse, "Distributed Operating Systems," Computing Surveys, Vol. 17, No. 4, Dec. 1985.

[3] M. W. Mutka and M. Linvy, "Profiling Workstaion's Available Capacity for Remote Execution, Performance '87,1 In Proceedings of the 12th IFIPWG 7. 3 Symp. on Computer Performance, 1987.

[4] D. L. Eager, E. D. Lazowska and J. Zahorijan, "Adaptive Load Sharing in Homogeneous Distributed Systems," IEEE Tr. on Soft. Eng., Vol. SE-12, NO. 5, MAY. 1986.

[5] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheudling in General-Purpose Distributed Computing Systems," IEEE Tr. on Soft. Eng., Vol. SE-14, No. 2, Feb. 1988.

[6] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," IEEE Comp., Dec. 1992.

[7] D. J. Farber, "The Distributed Computer System," in Proc. 7th Annu. IEEE Comp. Soc. Int.Conf., Feb. 1973.

[8] L. M. Ni, C. W. Xu and T. B. Gendreau, "A Distributed Drafitng Algorithm for Load Balancing," IEEE Tr. of Soft. Eng., Vol. SE-11, No. 10, Oct. 1985.

[9] F. C. H. Lin and R. M. Keller , "The Gradient Model Load Balancing Method," IEEE Tr. on Soft. Eng., Vol. SE-13, No. 1, Jau. 1987.

[10] M. M. Theimer and K. A. Lantz, "Finding Idle Machines in a Workstation-based Distributed System," IEEE Tr. on Soft. Eng., Vol. SE-15, No. 11, Nov. 1989.

[11] 황준, 김영시, 김영찬, "단일 메시지 토큰의 순환에 의한 부하 균등화 알고리즘," 한국정보과학회 논문지, 제 18권, 제 6호, 1991.

[12] 서경룡, 이철훈, 박규호, "동질의 트리 구조에서의 효율적인 태스크 할당 알고리즘," 한국정보과학회 논문지, 제 20권, 제 2호, 1993.

[13] 김종근, "스타형 컴퓨터 네트워크의 정적 부하 균형," 한국정보과학회 논문지, 제20권, 제6호, 1993.

[14] 류재철, "분산 시스템에서의 부하 균형을 위한 알고리즘," 한국정보과학회 논문지, 제20권, 제3호, 1993.

[15] 박경우, 김병기, "분산 시스템에서 동적인 혼합 부하 균형 알고리즘," 한국정보과학회 논문지, 제21권, 제4호, 1994.

[16] Y. Chang, and K. G. Shin, "Load Sharing in Hypercube-Connected Multicomputers in the Presence of Node Failures," IEEE Tr. on Comp., Vol. 45, No. 10, Oct. 1996.

[17] J. Kim, H. Lee, and S. Lee, "Replicated Process Allocation for Load Distributed in Fault-Tolerant Multicomputers," IEEE Tr. on Comp., Vol. 46, No. 4, Apr. 1997.

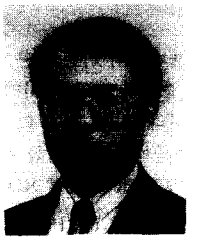
[18] S. Petri, and H. Langendorfer, "Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes," Operating Systems Review, Vol. 29, No. 4, Oct. 1995.

[19] M. G. Sriram and M. Singhal, "Measures of the Potential for Load Sharing in Distributed Computing Systems," IEEE Tr. on Soft. Eng., Vol. 21, No. 5, May 1995.

[20] K. G. Shin and Y. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," IEEE Tr. on Comp., Vol. 38, No. 8, Aug. 1989.

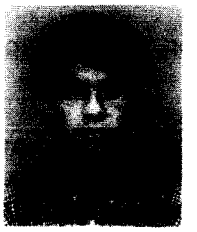
[21] Y. T. Wang and R. J. T. Morris, "Load Sharing in Distributed Systems," IEEE Tr. on Comp., Vol. C-34, Mar. 1985.

[22] A. Goscinski, Distributed Operating Systems: The Logical Design, A-W, 1991.



엄 영 익

1983년 서울대학교 계산통계학과 졸업(이학사)
1985년 서울대학교 대학원 전산과학전공(이학석사)
1991년 서울대학교 대학원 전산과학전공(이학박사)
1986년~1992년 단국대학교 전자계산학과 부교수
1993년~현재 성균관대학교 정보공학과 교수
관심분야:분산 시스템, 이동 컴퓨팅 시스템, 운영체제



김 구 수

1994년 성균관대학교 정보공학과 졸업(공학사)
1996년 성균관대학교 대학원 정보공학과 졸업(공학석사)
1996년~현재 (주)대교컴퓨터 사이에듀팀 연구원
관심분야:분산 시스템, 멀티미디어, 원격교육