AHU 서버의 병목현상을 해소하기 위한 부하분산 방법

안광호 [O1] 김성수 [1] 조위덕 [2]
[1] 아주대학교 정보통신전문대학원
{waterz[O], sskim}@ajou.ac.kr
[2]아주대학교 유비쿼터스 시스템 연구센터
chowd@ajou.ac.kr

A Load-Balancing Method to remove the Bottleneck of AHU Server

Kwangho An[O1], Sungsoo Kim[1] We-Duke Cho[2]
[1]Graduate School of Information and Communication, Ajou University
[2]Center of excellence for Ubiquitous System, Ajou University

요 약

AHU(Autonomic Healing Utility)는 결함발생요인을 관리하고 결함이 발생할 가능성이 있는 응용프로그램을 검출하기 위하여 결함예방서비스와 더불어 프로액티브한 결함복구서비스를 지원한다. Server-Client 구조를 갖는 AHU는 하나의 AHU 서버에 너무 많은 AHU 클라이언트들이 자가치유 서비스를 요청하는 경우 병목현상이 발생하여 AHU 서버의 처리용량을 넘게 되고 AHU가 제공하는 자가치유시간을 증가시켜 자가치유서비스의 질을 저하시키거나 AHU의 SPOF(Single Point of Failure)로 작용하여 AHU의 전반적인 결함을 발생시키는 원인이 된다. 따라서 본 논문에서는 AHU 시스템의 부하를 분산시켜서 AHU 서버에 생기는 병목현상을 해소하기 위한 방법을 제안하고 이를 구현하여 기존의 AHU와 AHU 서버들 사이의 부하분산 방법을 적용한 AHU를 비교하는 성능평가를 수행하였다.

1. 서론

최근 몇 년 동안 대두되고 있는 유비쿼터스(Ubiquitous)는 "언제, 어디서나"라는 뜻으로 사용자가 손쉽게 어디서나 컴퓨터를 사용할 수 있는 환경이다[1]. 이를 위해서는 소형 디바이스와 무선환경이 필요하다. 이러한 유비쿼터스 환경은 제한된 자원과 무선 네트워크로 인한 접속불량이라는 문제점을 가지고 있다. 이에 따라 잦은 결함이 발생하고 이것이 심하면 정상적인 서비스의 유지가 어렵게 된다. 이에 사용자가 신뢰하고 컴퓨팅 환경을 사용할 수 있는 고신뢰성 기술이 필요하게 되었고 오토노믹 컴퓨팅(Autonomic Computing)[2]이 결합되었다. 오토 노믹 컴퓨팅은 자율신경계 (Autonomous Nervous System)에서 유래된 용어로 주변 컴퓨팅 환경이 변화할 경우 시스템을 적절하게 변형시켜 적용시키는 것으로 이것을 자동화하는 기술이다.

오토노믹 컴퓨팅은 자가치유(Self-Healing), 자가최적화 (Self-Optimizing), 자가보호(Self-Protecting), 자가설정(Self-Configuring)의 기술들로 구성되어 있다. 연구하고자 하는 대상인 AHU (Autonomic Healing Utility)[3]은 자가치유기능을 제공해주는 기술을 구현한 소프트웨어 프로그램이다.

AHU는 결함발생요인을 관리하고 결함이 발생할 가능성이 있는 응용프로그램을 검출하기 위한 결함예방서비스와 더불어 프로액티브한 결함복구 서비스를 자동화함으로 자가치유기능을 제공한다[4]. AHU는 그림 1과 같은 Server-Client의 Two-Tier 구조를 갖는 소프트웨어 재활기술(Software

본 연구는 21세기 프론티어 연구개발사업의 일환으로 추진되고 있는 지식경제부의 유비쿼터스컴퓨팅및네트워크원천기반기술개발사업의 08B3-S2-10M 과제로 지원된 것임

Rejuvenation)[5]과 더불어 서비스 마이그레이션 (Service Migration)[6] 기반의 복구기술을 활용한다.

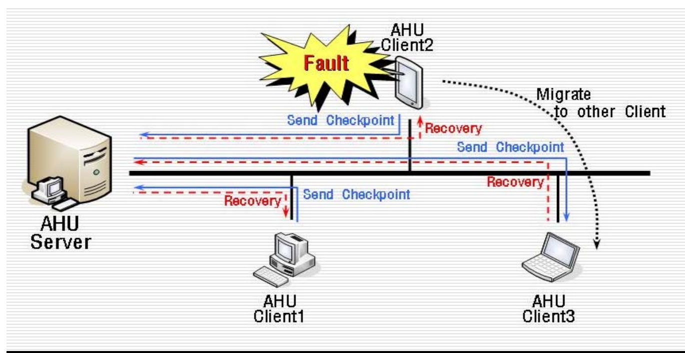


그림 1. AHU의 계층구조

AHU 시스템에서 AHU 클라이언트는 자가치유 서비스를 제공받기 위해 AHU 서버에 자가치유 서비스를 받고자 하는 응용프로그램을 AHU 서버에 등록한다. 그리고 등록된 응용프로그램의 CPU 사용량, 메모리 사용량, 네트워크 상태, 남은 배터리 상태 등의 체크포인트 데이터(Checkpoint Data)를 주기적으로 보낸다. AHU 서버는 이 체크포인트 데이터를 분석해 자가치유 서비스를 제공받는 응용프로그램의 상태를 판단한다. 이때 자가치유 서비스를 제공하는 여러 AHU 서버 중 하나의 AHU 서버에 너무 많은 AHU 클라이언트들이 자가치유 서비스를 요청하게 되는 경우가 발생할 수 있다. 이는 AHU 서버의 처리 용량을 넘어 병목현상을 발생시키고 AHU의 SPOF(Single Point of Failure)를 일으키는 원인이 된다. AHU 서버의 성능에 영향을 미치게 된다는 것은 자가치유 시간(Recovery Time)을 증가시켜 제공되는 자가치유 서비스의 질을 저하시키는 것이다.

기존의 AHU에 대한 연구에서는 AHU 서버의 부하에 대한 영향이나 AHU 서버들 사이의 부하정보 공유 및 부하분산에 대한 것은 고려되어 있지 않다[3]. 따라서 AHU 서버가 자가치유 서비스의 질을 저하시키는 과부하 상태가 되지 않도록 안정적인 서비스를 위한 연구가 필요하다. 본 논문에서는 AHU 시스템의 부하를 분산시킴으로써 AHU 서버에 생기는 병목현상을 해소하기 위한 방법을 제안하고 이를 구현하여 그 성능을 평가한다.

2. 관련 연구

2.1 Competition based Load-balancing[7]

Abed A.K.등은 각 클러스터(Cluster) 내의 컴퓨팅 자원을 부하분산(Load-balancing)하는 알고리즘을 제안했다. 제안된 알고리즘은 서로 협력하며 실제적으로 컴퓨팅을 수행하는 워크스테이션 (Workstation)들이 모여 클러스터를 구성하고 각 클러스터마다 부하(Workload)가 고루 분배되도록 하는 역할의 워크스테이션을 Cluster Controller(CC)로써 하나를 설정한다. 각각의 워크스테이션에 에이전트 (Agent)를 두어 부하를 관리하고 그 정보를 교환할 수 있도록 한다. CC의 에이전트는 자신이 속한 클러스터의 남은 컴퓨팅 자원량을 종합하고 다른 클러스터의 CC의 에이전트로부터 타 클러스터 내의 컴퓨팅 자원 정보를 받아 모자라는 컴퓨팅 자원을 공유할 수 있도록 한다.

2.2 Migration based Load-balancing[8]

Di Wu등은 DHT(Distributed Hash Table)를 사용하는 P2P(Peer-to-Peer) 시스템의 마이그레이션 기반의 부하분산 기법인 RDS (Rendezvous Directory Strategy)와 ISS (Independent Searching Strategy)를 비교 분석하고 두 가지 방법의 장점을 사용한 GBS(Gossip Based Strategy)를 제안했다. [8]에서 비교 분석한 내용을 보면 RDS의 경우 각 피어(Peer)는 랑데부 디렉토리 내의 다른 피어들에게 주기적으로 부하정보를 보낸다. ISS는 오직 부하정보에 대한 요청이 있을 때만 요청한 피어에게 부하정보를 제공한다. RDS는 대부분의 경우 ISS보다 효율적이다. 하지만 ISS의 경우 확장성과 강인성에서 RDS보다 더 나은 효율을 보인다. GBS는 전체 시스템을 그룹으로 묶고 그 그룹 내에서는 가쉽 프로토콜(Gossip Protocol)을 사용하여 부하정보를 공유하여 그룹 내에 부하분산 상태를 이룬다. 시스템이 여러 그룹으로 구성되는 경우 각 그룹 내에서 부하분산 상태를 만든 후 다른 그룹의 부하정보를 받아 부하분산 상태를 만든다.

AHU 서버의 과부하를 분산시키기 위해 AHU 서버의 몇 가지 요소를 모니터링하고 부하분산을 위해 결정해야 할 것들이 있다. 먼저 현재 AHU 서버에 대해 얼마나 많은 부하가 가해지고 있는지 CPU 사용량, 메모리 사용량 등을 지속적으로 모니터링 해야 한다. 그리고 AHU 시스템의 자가치유 시간을 증가시키는 수준으로 과부하가 가해지는 경우 어떤 AHU 서버로 부하를 옮겨 AHU 시스템 전반적으로 부하가 균형을 이루게 할 것인지 결정한 후 부하를 옮기는 작업을 수행하여 부하분산을 완성한다. 이를 위해 AHU 서버의 부하상태를 모니터링하고 AHU 서버의 부하정보를 공유할 수 있는 방법이 필요하다.

3. 본론

본 논문에서는 [7]을 바탕으로 그림 2와 같이 AHU 서버에 에이전트를 추가하여 AHU 서버의 부하상태를 모니터링하며 서로 다른 AHU의 에이전트와 부하정보(Load Information)을 공유할 수 있도록 한다. 이때 부하정보를 어떤 방식으로 다른 AHU 서버의 에이전트와 공유하는가 하는 것이 문제가 된다. 계속해서 AHU 서버들 사이의 부하분산 상태를 유지

하는 것이 아니라 한 AHU 서버가 과부하 상태가 될 때 그 상태를 벗어나도록 부하를 분산시킬 필요가 있다.

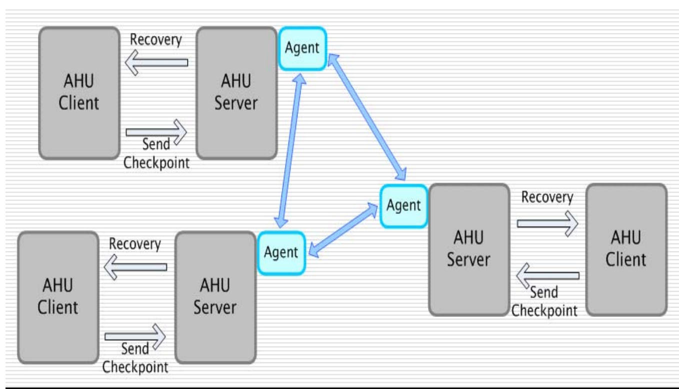


그림 2. 에이전트를 추가한 AHU의 구조도

따라서 부하정보를 자주 주고 받는 것은 불필요한 커뮤니케이션 오버헤드를 발생시킨다[9]. 그러므로 AHU 서버를 모니터링하는 에이전트는 주기적으로 부하정보를 보내야 하는 RDS방식이 아니라 필요할 때만 ISS 방식을 사용하여 부하정보를 주고 받도록 한다. 이렇게 ISS 방식을 사용하는 이유는 이때 에이전트는 AHU 서버에 추가적으로 동작하게 되므로 AHU 서버의 성능에 영향을 미치지 않기 위해 사용하는 시스템 자원량이 작을수록 좋기 때문이다.

3.1. 에이전트 구조 및 기능

AHU 서버의 부하분산을 위한 에이전트의 구조는 그림 3과 같이 모니터링 모듈(Monitoring Module), 판단 모듈(Decision Module), 통신 모듈(Communication Module)로 구성되어 있다.

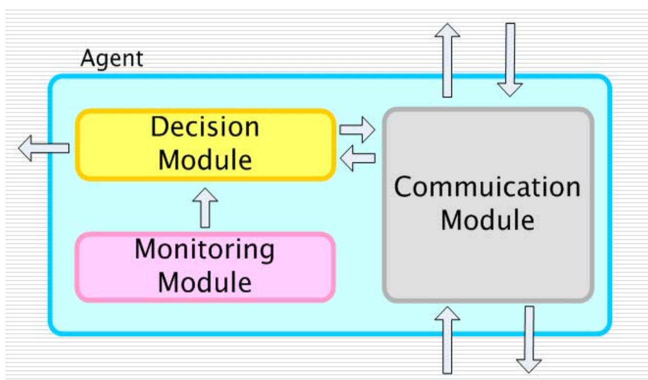


그림 3. 에이전트의 구조도

3.1.1 모니터링 모듈

모니터링 모듈은 에이전트가 속한 하나의 AHU 서버의 CPU 사용량, 메모리 사용량 등을 주기적으로 수집하여 AHU 서버에 걸리는 부하 정도를 모니터링 한다. 수집된 부하정보는 판단모듈로 전달한다.

3.1.2 통신 모듈

통신모듈은 부하정보를 공유할 수 있도록 서로 다른 에이전트와 통신하는 역할을 수행한다. 이를 위해 각 AHU 서버의 에이전트들은 서로의 주소 리스트를 미리 가지고 있어야 한다.

3.1.3 판단 모듈

판단 모듈은 모니터링 모듈을 통해 수집된 AHU 서버의 부하정보를 바탕으로 현재 AHU 서버의 상태가 너무 많은 AHU 클라이언트의 응용프로그램에 자가치유 서비스를 제공하는 과부하(Overloaded) 상태인지, 적정한 수준의 AHU 클라이언트의 응용프로그램에 자가치유 서비스를 제공하는 정상(Normal) 상태인지 판단한다.

표 1은 AHU 서버를 모니터링 할 에이전트의 각 모듈의 특징을 정리한 표이다.

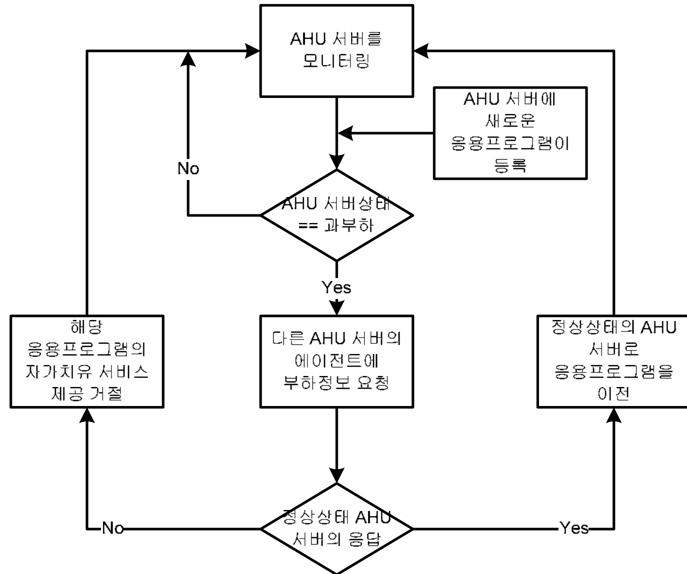
표1. 모듈별 기능

모 듈	기 능
모니터링 모듈	• AHU 서버를 모니터링 하며 자원상태 데이터 수집
판단 모듈	• 자원상태 데이터를 기반으로 AHU 서버의 상태를 판단
통신 모듈	• 다른 AHU 서버의 에이전트와 통신

3.2 부하분산 과정

부하분산 과정은 순서도 1과 같이 진행된다. 에이전트의 모니터링 모듈이 AHU 서버의 부하상태를 모니터링하는 것으로 시작한다. 자가치유 서비스를 제공받기 위해 AHU 서버에 새로운 응용프로그램이 등록을 하거나, AHU 서버의 다른 작업으로 인해 AHU 서버가 과부하 상태가 된다. AHU 서버가 과부하 상태로 판단되면 에이전트는 통신 모듈을 통해 다른 AHU 서버의 에이전트에게 부하정보를 요청한다. 부하정보 요청을 받은 에이전트들은 자신이 모니터링 하고 있는 AHU 서버의 상태가 과부하 상태인지 정상상태인지를 부하정보로 응답한다. 부하정보를 요청한 에이전트는 부하정보 요청에 대한 응답을 받은 후 가장 젊은 응용프로그램을 정상상태로 응답한 에이전트의 AHU 서버에게 자가치유 서비스를 받도록 한다. 가장 젊은 응용프로그램이란 실행시간이 가장 짧은 응용프로그램을 말한다. 응용프로그램들은 실행시간이 길어지면 메모리 누수, 버그 등으로 노후(Process Aging)되기 때문이다[5]. 응용프로그램이 노후된다는 것은 그만큼 결함이 발생하기 쉽기 때문에 자가치유 서비스를 받을 확률이 높아진다는 것이다. 그러므로 실행시간이 가장 짧은

응용프로그램은 결함이 발생하여 자가치유 서비스를 시작해야 할 확률이 상대적으로 가장 낮기 때문에 다른 AHU 서버로 옮기는 동안에도 문제가 될 확률이 가장 낮다.



순서도 1. 부하분산 과정

부하정보 요청에 대한 응답을 받을 때 동시에 여러 개의 에이전트에서 정상상태로 응답이 오게 되는 경우 제일 먼저 도착한 응답을 보낸 에이전트 쪽으로 AHU 클라이언트의 응용프로그램이 자가치유서비스를 받을 수 있도록 한다. 만일 정상 상태의 AHU 서버로 AHU 클라이언트의 응용프로그램을 넘기는 사이 그 정상 상태의 AHU 서버가 과부하 상태로 변하게 되는 경우 해당 AHU 서버에서 부하분산 과정을 처음부터 시작하게 된다.

만일 정상상태라고 응답한 에이전트가 없는 경우 다시 다른 에이전트에 부하정보를 요청한다. 다시 정상상태라고 응답하는 에이전트가 없을 경우 추후에 자가치유 서비스를 받기 위해 등록하고자 하는 응용프로그램의 등록을 거절한다.

4. 성능분석

4.1 성능분석 환경

본 절에서는 AHU 서버의 과부하를 얼마나 효과적으로 분산시킬 수 있는지 기존의 AHU와 본 논문에서 제안한 부하분산 기법의 성능을 비교분석 한다. 성능을 측정하기 위한 환경으로 전체 AHU 시스템을 AHU 서버 2대와 AHU 클라이언트 5대로 구성하였다. AHU 서버는 데스크톱 PC Pentium® 4 CPU 2.80GHz를 512MB RAM으로 2대를 구성하였고 AHU 클라이언트는 데스크톱 PC Pentium® 4 CPU 3.0GHz를 2.0GB RAM으로 1대, 데스크톱 PC Pentium® 4 CPU 2.80GHz를 512MB RAM으로 1대, 데스크톱 PC Pentium® 3 1.0GHz를 256MB RAM으로 2대, 노트북 PC Pentium® Mobile 1.7GHz를 1GB RAM으로 1대를 구성하였다.

각 AHU 클라이언트에서는 동일한 크기의 데이터(100KB)를 가진 Java기반의 NotePad 프로그램을 AHU 서버에 등록하여 자가치유서비스를 받도록 했다. Java 기반의 NotePad만 자가치유서비스를 받게 하는 이유는 NotePad 상에 가지고 있는 데이터를 AHU 클라이언트에서 체크포인트 데이터로 AHU 서버에게 주기적으로 보내게 되므로 체크포인트 데이터가 커질수록 AHU 서버에는 더 큰 부하가 걸리게 된다. 그렇기 때문에 좀 더 눈에 띄는 결과를 관찰하기 위하여 100KB의 데이터를 가진 NotePad를 사용했다.

성능분석 기준은 AHU 서버 1과 AHU 서버 2 중 서버 1에만 AHU 클라이언트들의 응용프로그램들을 계속해서 등록시켜 자가치유서비스를 받도록 한다. 이때 AHU 시스템 전체적으로 얼마나 많은 수의 응용프로그램에 자가치유 서비스를 제공 가능한지 비교한다.

4.2 성능분석 결과

4.2.1 기존의 AHU 서버

기존의 AHU는 그림 4와 같은 결과를 보인다. 그림 4와 그림 5의 가로축은 2개의 AHU 서버로 구성된 AHU 전체에서 자가치유 서비스를 제공받고 있는 응용프로그램의 수를 나타내고 세로축은 AHU 서버의 CPU 사용량을 나타낸다. 세로축의 CPU 사용량은 AHU 서버에 자가치유 서비스를 받을 하나의 응용프로그램을 등록 시킨 후 일정시간 동안 다른 작업을 하지 않은 상태의 CPU의 사용량을 측정한 평균값을 나타낸다.

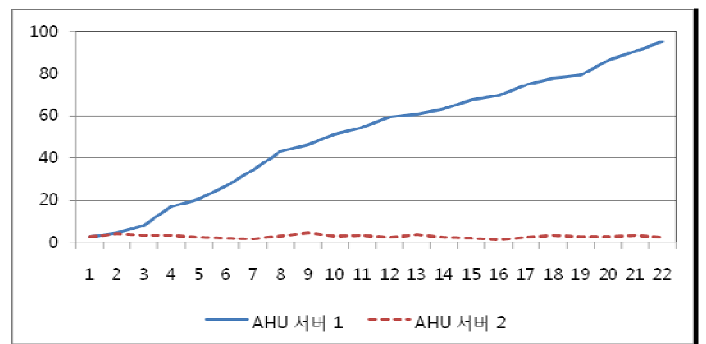


그림 4. 기존의 AHU의 측정 결과

기존의 AHU 측정 결과 계속해서 AHU 서버 1에 자가치유 서비스를 받도록 응용프로그램을 등록 시키므로 AHU 서버 1의 CPU 사용량은 계속해서 증가하게 된다. AHU 서버 2는 응용프로그램을 등록시키지 않으므로 CPU 사용량이 거의 변화없다.

그리고 AHU 서버 사이에 부하분산 과정이 없으므로 AHU 서버 1의 CPU 사용량이 계속 증가하여 과부하 상태가 된다. AHU 서버 1은 과부하 상태에서도 계속 새로운 응용프로그램을 등록시켜 자가치유 서비스를 받게 하지만 AHU 서버 2는 계속해서 정상상태로 있게 된다.

4.2.2 부하분산 방법이 적용된 AHU 서버

본 논문에서 제안한 AHU 서버들 사이의 부하분산 과정이 있는 경우는 그림 5와 같은 결과를 보인다. 4.1의 실험 방식과 동일하게 AHU 서버 1에만 자가치유 서비스를 받을 응용프로그램을 등록하게 되므로 AHU 서버 1의 CPU 사용량만 증가하게 된다. 그리고 AHU 서버 1에만 응용프로그램을 등록하게 되므로 AHU 서버 1은 과부하 상태가 되고 AHU 서버 2는 정상상태를 유지한다.

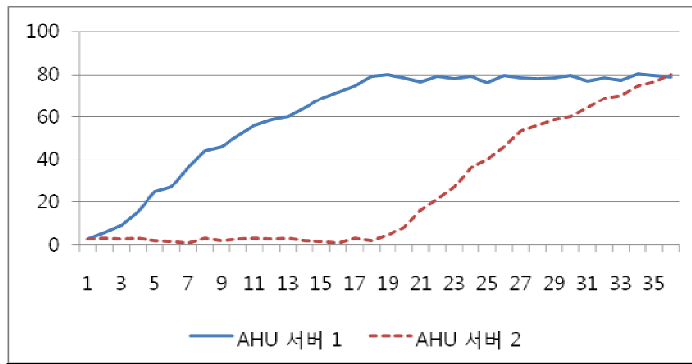


그림 5. 본 논문에서 제안된 AHU의 측정결과

하지만 AHU 서버들 사이의 부하분산 방법이 적용되어 있으므로 4.1의 결과와 다르게 AHU 서버 1이 CPU 사용량이 80%를 넘어 과부하 상태로 판단되는 19개째 응용프로그램에서 처음으로 AHU 서버 2로 응용프로그램들이 자가치유 서비스를 받게 된다. 그리고 계속해서 등록되는 응용프로그램들은 AHU 서버 2에서 자가치유 서비스를 받게 되어 그림 5와 같은 결과를 보이게 된다.

5. 결론

본 논문에서는 AHU가 가지는 Server-Client라는 구조적 특징 때문에 발생할 수 있는 하나의 AHU 서버에서의 병목현상을 해소하고 AHU 서버들 사이의 부하분산을 이루도록 하는 방법을 제안하였다.

AHU는 경량화되어 있지만 AHU 클라이언트로부터 계속해서 체크포인트 데이터를 받기 때문에 그 성능이 AHU 클라이언트의 체크포인트 데이터 크기에 많은 영향을 받는다. 체크포인트 데이터가 커지면 더 많은 부하가 걸리게 되고 체크포인트 데이터가 작아지면 상대적으로 더 적은 부하가 걸리게 되는 것이다. 향후에는 AHU 클라이언트의 체크포인트 데이터의 크기에 따른 AHU 서버의 병목현상 정도를 분석할 것이다.

AHU 서버의 병목현상을 해소하기 위한 부하분산 방법은 유비쿼터스 환경에서 더 많은 사용자에게 전반적인 AHU 시스템에 동시에 자가치유서비스를 제공할 수 있다. 유비쿼터스 환경이 점차 현대인의 생활 속에 자리잡아 감으로 AHU 서버의 부하분산 방법은 향후 많은 유익을 줄 수 있을 것으로 기대한다.

6. 참고문헌

[1] M. Brugnoli, et al., “User Expectations for Simple Mobile Ubiquitous Computing Environments,” Proceedings of 2nd Workshop on Mobile Commerce and Services, pp. 2-10, July 2005.

[2] C.L. Choi, “A Self-Management Autonomic System for High-Dependability,” Ph.D. Dissertation, Ajou University, 2007.

[3] 김성수, 조위덕, “Ubiquitous Smart Space,” 유비쿼터스 지능공간 백서, (재)유비쿼터스 컴퓨팅 사업단, 정보통신부, Nov. 2005.

[4] 최창열, 김성수, 조위덕, “고가용성 홈 서비스를 위한 오토노믹 자가관리 유틸리티,” 한국컴퓨터 종합학술대회 논문집 Vol 33, No. 1(A), pp. 136-138, June 2006.

[5] Y. Huang, et al., “Software Rejuvenation: Analysis, Module and Applications,” Proceedings of 25th International Symposium on Fault-Tolerant Computing, pp. 381-390, June 1995.

[6] J. Meehea, et al., “A Service Migration Case Study: Migrating the Condor Schedd,” Midwest Instruction and Computing Symposium, Apr. 2005.

[7] Abed A.K., et al., “Competition-Based Load Balancing for Distributed Systems,” Proceedings of 7th International Symposium on Computer Networks, pp. 230-235, June 2006.

[8] Di Wu, et al., “On the Effectiveness of Migration-based Load Balancing Strategies in DHT Systems,” Proceedings of 15th International Conference on Computer Communications and Networks, pp. 406-410, Oct. 2006.

[9] Shah R., et al., “On the Design of Adaptive and Decentralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments,” IEEE Transactions on Parallel and Distributed Systems, Vol. 18, No. 12, pp. 1675-1686, Dec. 2007.