http://dx.doi.org/10.3745/KTSDE.2013.2.8.507

Methods for Stabilizing QoS in Mobile Cloud Computing

Hyun Jung La† · Soo Dong Kim††

ABSTRACT

Mobile devices have limited computing power and resources. Since mobile devices are equipped with rich network connectivity, an approach to subscribe cloud services can effectively remedy the problem, which is called Mobile Cloud Computing (MCC). Most works on MCC depend on a method to offload functional components at runtime. However, these works only consider the limited verion of offloading to a pre-defined, designated node. Moveover, there is the limitation of managing services subscribed by applications. To provide a comprehensive and practical solution for MCC, in this paper, we propose a self-stabilizing process and its management-related methods. The proposed process is based on an autonomic computing paradigm and works with diverse quality remedy actions such as migration or replicating services. And, we devise a pratical offloading mechanism which is still in an initial stage of the study. The proposed offloading mechanism is based on our proposed MCC meta-model. By adopting the self-stabilization process for MCC, many of the technical issues are effectively resolved, and mobile cloud environments can maintain consistent levels of quality in autonomous manner.

Keywords : Mobile Cloud Computing, QoS, Autonomous Management, Self-stabilization, Offloading

모바일 클라우드 컴퓨팅을 위한 QoS 안정화 기법

라 현 정† · 김 수 동††

요 약

모바일 디바이스는 크기가 작기 때문에 PC에 비해 컴퓨팅 자원이 부족하여, 높은 복잡도를 가진 애플리케이션을 설치 및 운영하기 어렵다. 그러나, 모바일 디바이스는 풍부한 네트워크 연결 능력이 있어 외부 자원을 사용하기가 용이하므로, 모바일 클라우드 컴퓨팅 (Mobile Cloud Computing, MCC) 연구가 활발히 진행되고 있다. MCC에서는 주로 기능 컴포넌트를 다른 노드로 오프로딩 (Offloading) 시킴으로써, 모바일 노드의 자원 문제를 해결하는 접근법을 적용한다. 그러나, 현재 진행되고 있는 MCC에 대한 연구는 사전에 결정된 노드로 오프로딩하는 연구만 위주로 진행되고 있으며, 모바일 디바이스가 구독하는 서비스 문제를 해결할 수 없는 한계점이 있다. 본 논문에서는 자율 안정화할 수 있는 MCC를 구현하기 위한 실용적인 프로세스와 품질 안정화 기법을 제안한다. 먼저, 효과적으로 품질을 관리하기 위한 MCC 메타모델을 제시하고, 이를 기반으로 8개의 활동으로 구성된 품질 관리 프로세스를 제안하며, 핵심 활동에 대한 상세 기법을 정의한다. 그리고, 실용적 수준으로 연구가 많이 진행되지 않은 오프로딩 기법을 MCC 메타모델에 제시된 여러 요소들과 상호작용하여 QoS 문제를 효과적으로 해결할 수 있도록 설계한다. 마지막으로 실험을 통해 품질 자율 관리 프로세스의 적절성을 증명한다. MCC의 품질 자율 안정 관리 프로세스와 품질 향상 기법들을 채택함으로써, 클라우드 서비스를 구독하는 모바일 애플리케이션의 품질을 관리하는데 여러 기술적 이슈를 효과적으로 해결할 수 있다. 그리고, MCC에 속한 여러 애플리케이션과 서비스들은 관리자의 개입없이 자율적으로 일정 수준의 품질을 유지할 수 있게 된다.

키워드 : 모바일 클라우드 컴퓨팅, QoS, 자율 관리 프로세스, 자율 안정화, 오프로딩

1. 서 론

스마트폰, 태블릿 PC와 같은 모바일 디바이스는 휴대폰 기능뿐 아니라 컴퓨팅 능력까지 겸비하고 있고, 유연한 무선 인터넷 접속성과 높은 이동성(Mobility)을 제공한다는 장점을 가지고 있다. 그러나, 모바일 디바이스는 PC에 비해 메모리, CPU, 베터리 등 컴퓨팅 자원이 부족하므로[1][2], 자원 소비가 많고 높은 복잡도를 가진 애플리케이을 설치 및 운영하기 어렵다. 자원 부족 등의 문제를 효과적으로 해결하기 위한 방법으로 모바일 애플리케이션이 클라우드 서비스의 기능을 구독 및 사용하는 모바일 클라우드 컴퓨팅 (Mobile Cloud Computing, MCC)이 널리 연구되고 있다[3][4][5].

기존의 MCC 연구에서는 주로 기능 컴포넌트를 다른 노드로 오프로딩 (Offloading) 시킴으로써, 모바일 디바이스의

※ 이 논문은 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2012R1A6A3A01018389, 2012R1A1B3004130).
† 종신회원 : 숭실대학교 모바일 서비스 소프트웨어공학 센터 연구 교수
†† 종신회원 : 숭실대학교 컴퓨터학부 교수
논문접수 : 2012년 12월 24일
수 정 일 : 1차 2013년 2월 15일, 2차 2013년 4월 21일
심사완료 : 2013년 5월 17일
* Corresponding Author : Soo Dong Kim(sdkim777@gmail.com)

자원 문제를 해결하는 접근법을 주로 사용한다. 이는 사전에 결정된 노드로 제한적인 방법으로 오프로드가 수행되어 MCC의 동적으로 변화하는 환경을 반영하지 못하는 한계점을 가지고 있다. 그리고, MCC는 모바일 디바이스의 제한된 자원 문제 해결할 수 있다는 장점이 있지만, 일정 수준의 QoS를 효과적으로 관리하는데 다음과 같은 기술적인 어려움이 있다.

- 모바일 애플리케이션 사용자의 높은 이동성과 무선 네트워크 환경으로 인해 QoS가 불안정함.
- 서비스는 제한된 가시성과 제약적인 관리성의 특징을 가지고 있으므로, 구독한 서비스의 결함 및 낮은 QoS가 발견될 경우, 서비스 소비자 입장에서 이를 해결하는 방법이 제약적임.

본 논문에서는 자율 안정화할 수 있는 MCC를 구현하기 위한 실용적인 프로세스를 제안한다. 제안된 프로세스는 자율 컴퓨팅 패러다임에 기반을 하고 있고, 다양한 종류의 QoS 품질 향상 기법인 모바일 노드에서 서버 노드로 모바일 애플리케이션을 오프로딩하는 기법, 서버 노드에서 다른 서버 노드로 서비스를 이주 및 복제하는 기법 등과 상호연동된다.

논문의 구성은 다음과 같다. 2장에서는 오프로딩 위주의 연구를 정리하였으며, 3장에서 QoS 관리와 관련된 기술적인 문제를 해결할 수있는 MCC의 메타모델을 제시하며, QoS 관리 중앙집중 부하를 줄이기 위한 클러스터 개념을 제안한다. 4장에서는 MCC의 자율 안정화를 가능하게 하는 8가지 단계로 구성된 QoS 자율 관리 프로세스를 제안한다. 5장에서는 MCC의 QoS 문제 발생 시 실행할 수 있는 여러 품질 안정화 기법 중 실용적 수준으로 연구가 많이 진행되지 않은 오프로딩 기법을 상세히 설계한다. 제시된 기법들은 MCC 메타모델에 제시된 여러 요소들과 상호작용하여 QoS 문제를 효과적으로 해결할 수 있도록 초점을 맞추어 설계된다. 6장에서는 품질 자율 관리 프로세스의 적절성을 증명하기 위해 실험을 수행한다. 그리고, 그 결과를 정량적인 수치 데이터를 기반으로 요약함으로써 제시된 기법을 평가한다. 마지막으로, 7장에서는 본 연구의 결론 및 향후연구를 기술한다.

MCC의 품질 자율 안정 관리 프로세스와 품질 향상 기법들을 채택함으로써, 클라우드 서비스를 구독하는 모바일 애플리케이션의 품질을 관리하는데 여러 기술적 이슈를 효과적으로 해결할 수 있다. 그리고, 관리자의 개입없이 자율적으로 MCC 품질을 일정 수준 이상으로 유지할 수 있게 된다.

2. 관련 연구

Chun의 연구에서는 애플리케이션의 작업량을 동적으로 분할하여 모바일 디바이스에서도 복잡한 기능을 수행하는 애플리케이션을 실행할 수 있는 접근법을 제안하였다[6]. 그리고, 가상 머신 환경에서 동적 오프로딩을 지원하는 CloneCloud라는 시스템을 제안하였다[7]. 이 시스템은 실시간에 현재 모바일 디바이스 자원 상태에 맞게 오프로드 가능한 컴포넌트를 선정하여 오프로딩을 수행한다. 이 연구는 애플리케이션 기능을 분할하는 기법 위주로 제시하고 있지만, 모바일 디바이스 외에 오프로드 목표 노드, 네트워크 등의 MCC 상태를 고려한 최적의 오프로딩 수행은 고려하고 있지 않다.

Satyanarayanan의 연구에서는 가상 머신 (Virtual Machine, VM) 기법들을 이용하여 기능성이 복잡한 기능을 사용자 주변에 위치한 풍부한 자원을 가진 컴퓨터 클러스터인 Cloudlet으로 오프로딩하는 기법을 제안하였다[8]. 이 연구에서는 VM 통합(synthesis) 기법을 이용하여 오프로딩 기법을 구체화하였지만, 현재 상황을 고려한 자율적인 오프로딩 기법을 제시할 필요가 있다.

Cuervo의 연구에서는 에너지 소모량을 고려하여 메소드 단위로 오프로드시키는 MAUI라는 시스템을 제안하였다[9]. MAUI는 모바일 디바이스의 자원 상태와 네트워크 상태를 고려하여 오프로딩으로 인한 이득이 큰 메소드를 이주시킨다. 이 연구는 .NET 프레임워크에서 제공하는 리플랙션을 이용하여 오프로딩 기법을 다소 실용적인 수준으로 제시하고 있지만, 핵심 알고리즘이 개념 위주로 설명되어 있다.

이 외에 아키텍처 접근 방법으로 모바일 클라우드 컴퓨팅 관련 문제를 해결하려는 다양한 연구가 진행되고 있다. Malek의 연구에서는 런타임에 모바일 소프트웨어를 이주시키기 위한 아키텍처 기반 프레임워크를 개략적으로 제안하였다[10]. Han는 런타임에 컴포넌트를 동적으로 재구성하는 적응적인(adaptive) 소프트웨어 아키텍처를 위한 커넥터를 상세 구현 지침없이 개략적으로 제안하였다[11].

이 외에도 Li 의 연구 [12], Yang의 연구 [13], Ye의 연구 [14], Hung의 연구 [15] 등 런타임 오프로딩을 지원하는 미들웨어 또는 프레임워크를 제안하는 여러 연구가 진행되었다. 그리고, Flinn의 연구 [16], Balan의 연구 [17], Kalasapur의 연구 [18]와 같이 유비쿼터스 환경에서 오프로딩을 지원하는 Cyber Foraging 기법[19]을 이용한 다양한 연구 또한 진행되고 있다.

현재까지 진행된 연구들을 분석하여 다음의 한계점을 도출한다.

- 오프로딩 기법만을 주로 고려하기 때문에, 서버 측에서 발생하는 문제를 해결하지 못함.
- 대부분 오프로딩을 위한 목표노드로 사전에 선정된 하나의 서버 또는 데스크톱 PC만 고려하기 때문에, 현재 품질 상태를 고려하여 여러 후보 중 최적의 노드로 오프로딩하는 것은 불가능함.
- 애플리케이션이 설치된 노드의 자원 이용가능성, 성능향상 유무 등 제한적인 기준으로 오프로드 여부가 결정됨.
- 오프로딩은 노드에 설치된 에이전트(agent)에 의해 수행되고, 모든 애플리케이션과의 오프로딩을 하나의 서버측 에이전트에 의해 관리되므로, 서버측 에이전트에 관리 부하가 치중될 수 있음.

3. 모바일 클라우드 컴퓨팅 메타 모델

2장에서 언급한 현재 연구의 제약적 및 기술적인 이슈들을 해결하기 위해서, 자율 컴퓨팅 개념을 적용하여 기존에 연구된 모바일 클라우드 컴퓨팅 환경을 확장한다. Fig. 1은 확장된 모바일 클라우드 컴퓨팅 환경의 메타모델을 보여준다. 확장된 요소들은 모서리가 둥근 사각형으로 표현한다.

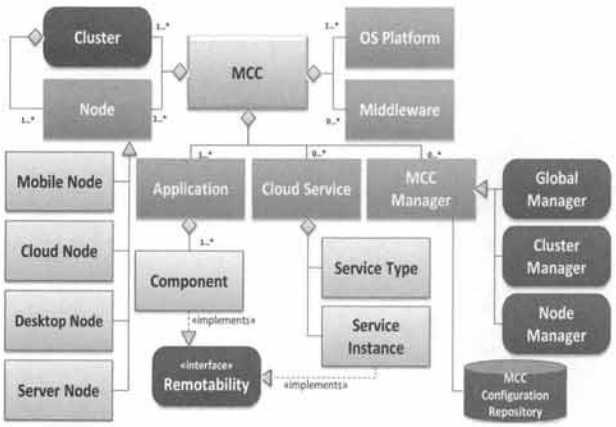

Fig. 1. A Meta Model of Extended MCC

확장된 MCC 환경은 노드(Node), 클러스터 (Cluster), OS 플랫폼, 미들웨어, 애플리케이션, 클라우드 서비스, MCC 매니저를 포함한 7가지 요소로 구성된다.

노드는 애플리케이션 또는 서비스를 설치하여 운영하는 물리적인 컴퓨팅 디바이스로서, 모바일 노드, 클라우드 노드, 데스크톱 노드, 서버 노드로 분류된다. 노드는 일반적으로 OS 플랫폼 상에서 애플리케이션을 운영하며, 애플리케이션이 필요로 하는 시스템 기능을 제공하는 미들웨어를 가지고 있다.

애플리케이션은 대부분 모바일 애플리케이션을 지칭하며, 클라우드 서비스는 여러 서비스 소비자에 의해 재사용될 수 있는 기능성을 의미한다[20]. 본 논문에서는 여러 클라우드 서비스 중 소비자가 필요로 하는 단위 기능을 제공하는 Component-as-a-Service (CaaS)만 고려한다. 각 서비스는 인터페이스 역할을 하는 서비스 타입(Service Type)과 서비스 인스턴스로 구성된다. 한 서비스 타입은 여러 개의 서비스 인스턴스(Service Instance)로 구현될 수 있다[20]. MCC에서 재사용 가능하며, 복잡한 계산을 수행하는 서비스를 사용함으로써 모바일 디바이스의 자원 제약성 문제를 해결할 수 있다.

클러스터는 확장된 MCC에서 새롭게 추가된 요소로서, 의존성이 높은 노드들을 그룹핑하여 관리하는 개념적인 단위이다. MCC를 구성하는 다양한 요소의 품질을 효과적으로 관리하기 위해 모니터링, 문제 상황 발견, 문제 상황을 해결하기 위한 품질 향상 계획 수립 등 관리 태스크들을 자율적으로 수행해야 한다. 만약, 노드 수가 급증한다면, 한 매니저가 모든 관리 태스크들을 실행하므로 이로 인한 오버헤드가 높아진다. 그러므로, 집중 포화될 수 있는 관리 태스크를 분산시키기 위해 MCC 전체를 클러스터를 나누고, 각 클러스터 별로 관리 태스크를 실행시키고자 한다. 클러스터의 구성에 따라 MCC의 전체 품질 안정화에 영향을 주게 되므로, 클러스터링을 최적의 상태로 결정하는 것이 매우 중요하다. MCC에서는 대표적인 클러스터링 알고리즘인 K-Means 알고리즘 [21][22], Hirarchical 알고리즘 [21][22]을 적용하여 클러스터링을 수행한다. 두 알고리즘은 두 노드 사이의 거리 절대값을 이용하지만, MCC에서는 상호작용 빈도수를 의미하는 논리적인 근접도 (Logical Closeness)를 고려한다. MCC에서 사용하는 논리적인 근접도를 계산하는데 고려된 기준은 다음과 같으며, 이 외의 다른 기준들이 추가될 수 있다.

- 두 노드 간의 물리적인 거리 (km)
- 두 노드 간의 선언된 네트워크 대역폭 (bit/sec) : 두 노드 간 네트워크에 대한 정보로부터 유도됨.
- 특정 시간 동안 측정된 네트워크 대역폭 (bit/sec) : 특정 시간 동안 발생한 데이터 전송 건의 데이터 전송 속도(= 전송된 데이터 크기/전송된 시간)의 평균치
- 특정 시간 동안 측정된 두 노드 간의 상호작용 횟수 : 메세지 호출 수로부터 유도됨.
- 특정 시간 동안 측정된 애플리케이션의 응답시간 (Response Time)

MCC 관련 연구에서 언급한 일정한 수준의 QoS를 유지하고, 런타임 결함 문제를 해결하기 위한 효과적인 방법은 컴포넌트 또는 서비스를 동적으로 오프로딩하거나 배치하는 것이다[3]. 그러나, MCC는 다양한 OS 플랫폼 및 미들웨어를 가진 노드들이 포함되어 있으므로, 이질성을 고려하여 오프로딩 기능을 구현하는게 어렵다. 그러므로, 동적 오프로딩를 지원하기 위해 필요한 메소드를 정의한 인터페이스인 Remotability를 통해, 오프로딩 가능한 애플리케이션들은 이 인터페이스를 구현하도록 한다.

MCC 매니저는 해당 노드의 자원 및 애플리케이션 또는 서비스의 품질을 측정하고 품질 향상 활동을 수행하는 등의 관리 태스크를 담당하는 노드 매니저, 클러스터 단위에서 품질 저하 여부를 판단하며 향상 계획을 수립하는 등의 관리 태스크를 수행하는 클러스터 매니저, 전체 MCC의 관리 태스크를 담당하는 글로벌 매니저로 구분된다. 클러스터 매니저와 글로벌 매니저는 각각 형상 관련 저장소를 통해, 현재 MCC에 속한 노드, 애플리케이션, 서비스 등의 형상 정보 및 각각의 QoS 정보를 관리한다.

4. 모바일 클라우드 컴퓨팅 품질 자율 관리 프로세스

일정 수준 이상의 QoS를 유지하기 위해, 확장된 MCC 메타모델 외에 이를 위한 자율 안정화 품질 관리 프로세스가 필요하다. Fig. 2는 8개의 활동(Activity)로 구성된 자율 안정화 품질 관리 프로세스를 보여준다. 이 프로세스는 메타모델에 언급한 3가지 종류의 매니저에 의해서 운영된다.

노드 매니저가 해당 노드의 여러 품질을 측정함으로써 자율 안정화 품질 관리 프로세스가 시작된다 현재 측정된 품질과 이전에 측정된 품질의 차이가 사전에 정의된 임계치보다 큰 경우, 노드 매니저는 담당 클러스터 매니저에게 품질 데이터를 전송한다 (활동 ❶). 각 클러스터에 할당되어 있는 클러스터 매니저는 현재 품질이 안정한 수준에 있는지 아닌지를 평가하고(활동 ❷), 현재 품질이 불안정한 상태인 경우 클러스터 수준의 품질 향상 계획을 수립한다 (활동 ❸). 계획에 따라 클러스터 수준의 품질을 안정화시키는 활동을 수행하며 (활동 ❹), 수정된 형상 정보를 클러스터 형상 관리 저장소에 업데이트한다 (활동 ❺). 클러스터 형상 관리 저장소 관리하는 정보는 해당 클러스터에 포함된 노드에 대한 스펙 정보, 노드들에 설치된 애플리케이션 및 서비스 기능성 정보, 품질 데이터 히스토리, 기능 호출 히스토리 정보 등이 있다.

활동 ❸에서 현재 발생된 문제를 해결하기 위한 클러스터 수준의 품질 향상 계획을 수립하지 못하면, 이 문제는 글로벌 수준으로 위임된다. 이 후의 활동은 클러스터 수준의 품질 향상 활동과 동일하다.

활동 ❸의 지침은 한 클러스터가 아니라 여러 클러스터 간이라는 적용 범위를 제외하고 활동 ❻에 적용될 수 있다. 활동 ❹와 활동 ❼과 관련된 내용은 5장에서 상세히 다룰 것이고, 활동 ❺과 활동 ❽ 지침은 데이터베이스에 정보 업데이트를 수행하기 때문에 다소 간단하다. 그러므로, 본 장에서는 8가지 활동 중 자율 관리 프로세스에서 핵심 활동인 활동 ❶부터 활동 ❸까지의 상세 지침 및 기법을 정의한다.

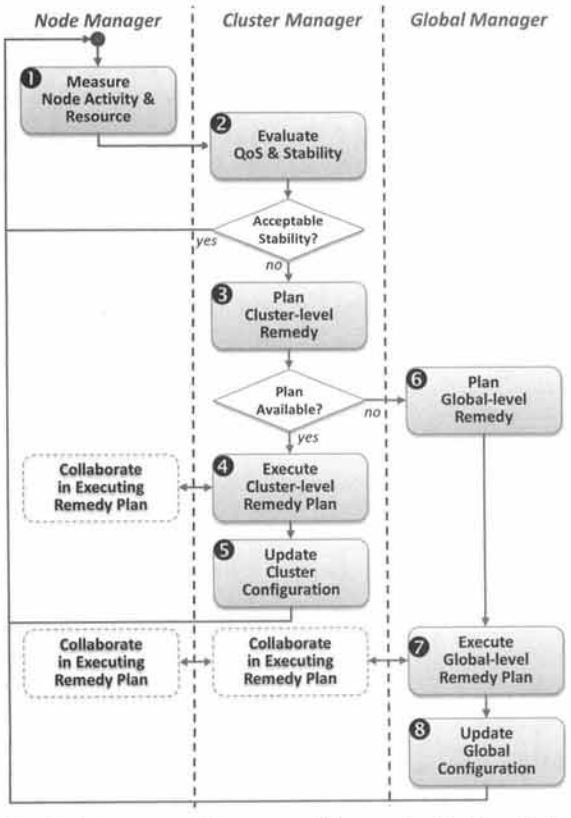

Fig. 2. Autonomous Management Process to Stabilize QoS

4.1 활동 1. 노드 자원 및 노드 활동 품질 측정에 대한 지침

활동 ❶은 각 노드에 설치된 노드 매니저가 QoS 관리에 필요한 품질 정보를 획득하는 단계이다. 이 활동을 통해 획득되는 정보는 Table 1과 같다.

Table 1. Measurable Data for Monitoring Targets

Monitoring Target	Measured Data
Application	Turnaround Time, Amount of Used Resources
Service	Response Time, Throughput, Amount of Used Resources
Node	· Resource-related Information (Available Amount of CPU, Memory, and/or Battery, etc) · Applicaton/Service-related Information (# of Executing Applications/Services, Ammount of Used Resources, etc)

노드 매니저는 주기적으로 또는 애플리케이션이 서비스를 호출하는 등의 이벤트가 발생할 때마다 활동 1을 수행한다. 이 때, 플랫폼에서 제공하는 API를 이용하여 노드의 자원을 측정할 수 있으며, 노드에 설치된 애플리케이션 및 서비스는 로그 정보를 이용하여 측정할 수 있다. 예를 들어, JVM이 설치된 노드인 경우에는 Runtime 클래스에서는 제공하는 freeMemory(), maxMemory() 등의 메소드를 이용하여 이용 가능한 메모리양, 이용 가능한 최대 메모리양 등의 정보를 획득할 수 있다.

4.2 활동 2. 품질 안정화 정도 평가에 대한 지침

활동 ❷는 여러 노드 매니저에서 전달된 측정치를 이용하여, 클러스터 수준의 안정화 상태를 결정하는 단계이다. 이를 위해, Fig. 3과 같이 3가지 수준의 안정화 상태를 정의한다.

Fig. 3. Three Stability Levels

- 정상 상태 (Normal): 정상 안정화 상태에 있으며, 어떤 품질 안정화 활동도 수행될 필요가 없다.
- 품질 저하 상태 (Poor): 현재 품질의 안정화 정도가 낮은 상태이다. 저하된 품질을 정상 상태로 바꾸기 위한 품질 안정화 활동이 필요하다.
- 품질 초과 상태 (Exceeding): 현재 품질 안정화 정도가 필요 이상으로 높다. 이는 상대적으로 많은 자원이 소비되어 현재 품질이 과다하게 높음을 암시한다.

위의 안정화 상태를 결정하기 위해서 특정 품질 x에 대한 임계 상한치인 $Threshold_{high}(x)$와 임계 하한치인 $Threshold_{low}(x)$가 필요하다. 만약 x가 애플리케이션 또는 노드에 대한 품질 측정치일 경우에, 품질 관리자 또는 사용자

가 경험 및 도메인 지식을 기반으로 기준치 값을 결정한다. 그리고, x가 서비스의 품질 측정치일 경우에는 Service Level Agreement (SLA)에 기술된 값을 이용하여 결정된다. 예를 들어 WSLA [23]를 준수한 SLA의 경우, <Predicate xsi:type="Less">와 <Predicate xsi:type="Greater">에 엘리먼트에 기술된 내용을 기반으로 두 개의 임계치를 결정할 수 있다.

클러스터 매니저는 현재 측정된 품질 측정치와 위의 기준치를 비교하여, 클러스터 수준의 품질 안정화 상태를 결정한다.

4.3 활동 3 및 활동 6. 품질 향상 계획 수립에 대한 지침

활동 ❸와 활동 ❻은 품질 측면에서 비정상 상태를 정상 상태로 복구하기 위한 품질 향상 계획을 클러스터 수준 또는 글로벌 수준에서 수립하는 단계이다. 이를 위해, Fig. 4와 같이 품질 향상 계획을 수립하기 위한 2가지 측면을 고려하며, 이는 품질 향상 계획을 수립하는데 중요한 역할을 한다.

Fig. 4. Two Aspects in Making Remedy Plan

품질 저하 상태에 있는 경우, 현재 품질을 높이기 위한 품질 향상 계획이 수립되어야 한다. 반대로, 품질 초과 상태에 있는 경우, 낭비되는 자원을 회수하여 현재 품질을 하향 평준화(Equalizing Stability)하여 정상 상태로 되돌리도록 품질 향상 계획이 수립되어야 한다. 즉, 현재 품질 안정화 상태에 따라 품질 향상 계획의 목적이 달라진다. 이것이 품질 향상 계획의 첫번째 측면인 "목적(Goal)"이다.

품질 향상 계획은 현재 품질 문제의 심각도에 따라 클러스터 수준에서 해결될 수 있거나 글로벌 수준에서 해결될 수 있다. 클러스터 수준의 품질 향상 계획은 해당 클러스터 내(Intra-cluster)에 있는 애플리케이션 또는 서비스 복제 및 이주 등의 품질 향상 활동을 포함하지만, 글로벌 수준의 품질 향상 계획은 클러스터 간(Inter-Clusters)의 애플리케이션 또는 서비스 복제 및 이주 등의 품질 향상 활동을 포함한다. 이것이 품질 향상 계획의 두번째 측면인 "커버리지(Coverage)"이다.

위의 2가지 측면에 고려하면, 품질 향상 계획 유형은 1) 클러스터의 품질을 향상시키는 계획, 2) 글로벌 수준의 품질을 향상시키는 계획, 3) 클러스터의 품질을 하향 평준화하는 계획, 4) 글로벌 수준의 품질을 하향 평준화하는 계획으로 분류된다.

4가지 유형의 품질 향상 계획을 수립하기 위해서 1) 해당 문제에 대한 원인 식별, 2) 원인을 해결하기 위한 최적의 품질 향상 활동 선택, 3) 선택된 품질 향상 활동에 대한 상세 계획 수립을 수행한다.

태스크 1은 품질 안정화 평가 결과를 상세히 분석함으로써 근본적인 원인을 식별한다. 예를 들어, 응답시간이 정해진 임계치 기준에 못미쳐 품질 안정화에 문제가 있다고 판단되면, 클러스터 매니저는 응답시간 관련 측정치와 자원 관련 측정치를 이용해서 모바일 애플리케이션, 네트워크, 서비스 중 어느 부분에 문제가 있는지를 확인할 수 있다.

태스크 2는 현재 식별된 원인을 해결하기 위한 가장 적절한 품질 향상 활동을 선정한다. Fig. 5는 MCC에서 고려하고 있는 6가지 후보 품질 향상 활동을 보여준다.

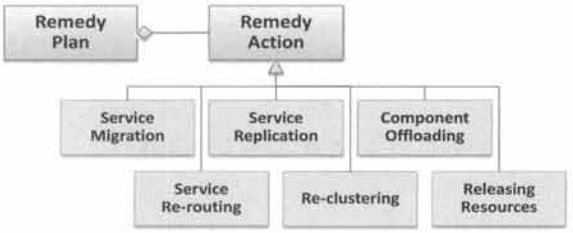

Fig. 5. Six Types of Remedy Actions

서비스 리라우팅(re-routing)은 한 서비스 인스턴스로의 호출을 동일한 서비스 타입을 구현한 다른 서비스 인스턴스로의 호출로 변경시키는 기법이다. 재클러스터링(Re-clustering)은 현재 클러스터들의 형상 정보를 변경하는 것이다. 서비스 이주(Service Migration)는 문제가 되는 서버 노드에 설치된 서비스 인스턴스를 다른 노드로 이동시키고, 원래 노드에 있는 서비스 인스턴스는 삭제하는 기법이다. 서비스 복제(Service Replication)는 서비스 이주와 유사하지만, 원래 노드에 있는 서비스 인스턴스를 삭제하지 않는다. 컴포넌트 오프로딩은 모바일 노드에 설치된 애플리케이션의 일부 또는 전체 기능을 다른 노드로 복사하는 기법이며, 이에 대한 상세 내용은 5장에서 다룬다.

가장 적절한 품질 향상 활동은 근본적인 원인에 따라 결정된다. MCC에서는 원인-품질 향상 활동 간의 의존성 관계를 규칙 형태로 정의하여 관리하며, 클러스터 매니저 또는 글로벌 매니저는 최적의 품질 향상 활동을 선택하기 위해 규칙을 이용한다. Table 2는 품질 향상 계획 유형 1인 클러스터 수준의 품질 향상 계획을 수립하기 위한 규칙 일부이다. 하나의 원인에 대해 여러 개의 품질 향상 활동이 적용 가능하다면, 클러스터 매니저는 해당 품질 향상 활동을 수행함으로써 얻을 수 있는 효과에 대한 비용 함수 (Cost Function)를 이용하여, 높은 품질 이익을 산출하는 활동을 선택한다.

마지막 태스크 3은 선택된 품질 향상 활동에 대한 상세한 계획을 수립하는 단계이다. 예를 들어, 서비스 리라우팅의 경우 문제 서비스 인스턴스를 설치한 노드와 인접하고, 동일한 서비스 유형을 구현한 서비스 인스턴스를 가지고 있는 노드를 선택해야 하며, 기능 단위 이동과 관련된 품질 향상 활동의 경우에는 이동시킬 기능 컴포넌트, 목표 노드 등을 선택해야 한다. 품질 향상 활동의 종류에 상관없이 문제가 있는 서비스 또는 애플리케이션의 역할을 위임할 목표 노드

Table 2. Rules for Remedy Plan (Type 1)

Faults	Causes	Applicable Remedy Actions
Low QoS on Applicatoin	Low Resource on Node	Component Offloading
	Low QoS of Service Invoked by the Application	Service Rerouting, Service Replication
Low QoS on Service	Low Resource on Node	Service Migration, Service Re-routing
	Unstable Network, Large Number of Invocatgions	Service Replication, Service Re-routing

가 선정되어야 하며, 이는 최단 경로를 찾는 대표적인 매커니즘인 Dijkstra 알고리즘 [24]을 이용하여 결정된다.

5. QoS 안정화을 위한 오프로딩 기법

본 장에서는 4장의 Fig. 2에서 제안된 안정화 프로세스 중 활동 ❻과 ❽에서 사용되는 오프로딩 기법에 대한 실용적인 설계와 구현 방법을 제시한다. 서비스 이주, 서비스 복제, 서비스 리라우팅에 대한 상세한 기법들은 SOC 분야에서 많은 연구가 진행되었고, MCC에는 [25][26]의 연구 기법을 적용한다. 재클러스터링은 클러스터링 알고리즘인 K-Means, Hirarchical 알고리즘 [21][22], 자원 회수는 Garbage Collector를 호출함으로써 구현될 수 있으므로, 기술적 어려움이 크게 없어 본 장에서 다루지 않는다. 그러므로, 본 장에서는 모바일 노드 관련 문제를 해결하기 위한 주요 기법인 오프로딩을 상세히 다룬다. MCC에서는 효과적인 오프로딩을 위해서, 오프로딩 시기에 따라 정적 오프로딩과 동적 오프로딩으로 분류하며, MCC 메타모델의 요소들과 프로세스와 일관성 있게 기술적으로 상세화한다.

5.1 정적 오프로딩 (Static Offloading) 기법

1) 정적 오프로딩 구조

QoS 문제가 발생하기 이전에 소스 노드에 설치된 애플리케이션을 미리 목표 노드에 오프로드시키고, 문제가 발생했을 시에는 목표 노드에게 기능 실행을 위임하는 것을 정적 오프로딩이라고 한다. Fig. 6은 정적 오프로딩의 정적 배치 상태를 보여준다. QoS가 정상 상태이라도 하더라도, 하나 이상의 목표 노드에 오프로드 가능한 전체 애플리케이션을 구성하는 n개의 컴포넌트 ($COMP_1 \cdots COMP_n$)가 설치되어 있다.

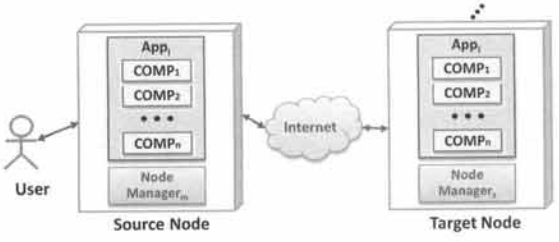

Fig. 6. Structural View of Static Offloading

2) 정적 오프로딩 런타임 절차

MCC에서 해당 애플리케이션의 오프로딩 여부, 오프로딩의 목표 노드 등은 활동 ❺또는 활동 ❼에서 클러스터 매니저에 의해 이미 결정된다. 노드 매니저는 이 정보를 이용하여, Fig. 7에 명시된 정적 오프로딩 절차를 수행한다. 이 그림은 사용자가 소스 노드에 있는 애플리케이션 (Application_n)의 opA()를 요청하면, 이는 내부적으로 opB()…opZ()를 실행하여 결과값을 반환한다고 가정하고 작성된 것이다.

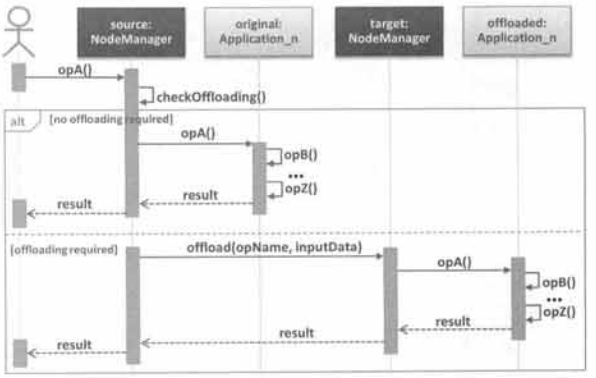

Fig. 7. Behavioral View of Static Offloading

사용자는 애플리케이션 Application_n의 기능인 opA( )를 호출하지만, 내부적으로는 *NodeManager*가 호출된다. *NodeManager*는 checkOffloading( ) 메소드 호출을 통해, 현재 사용자의 요청에 대한 오프로딩 실행 여부를 확인한다. 그리고, 품질 문제가 없으면, *original:Application_n*의 opA( )를 호출한다. 그렇지 않은 경우에, 클러스터 매니저가 결정한 목표 노드의 노드 매니저에 실행시킬 오퍼레이션 이름(opName)과 사용자의 입력값 (inputData)을 offload( ) 메소드를 통해 전달한다. 이 후, 목표 노드의 노드 매니저인 *target:NodeManager*는 사전에 오프로드된 *offloaded: Application_n*의 opA()를 실행시킨다.

정적 오프로딩 설계에서는 사용자에게 투명한 방법으로 오프로딩을 수행하기 위해, 애플리케이션을 호출하는 메소드를 노드매니저가 가로채어야 하며, 이를 위해 Proxy 패턴 [27]을 적용한다. Fig. 8은 이를 위해 동적 프록시를 지원하는 java.lang.reflect.Proxy를 이용한 *NodeManager* 설계모델이다. NodeManager는 java.lang.reflect.InvocationHandler를 구현한 Node_InvocationHandler에 구현된 invoke( ) 메소드를 통해 오프로딩 여부에 따라 *Applicatin_n*의 메소드를 실행한다.

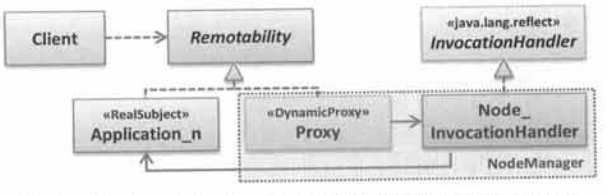

Fig. 8. Design of NodeManager with java.lang.reflect.proxy

3) 수식을 통한 정적 오프로딩의 이점 증명

정적 오프로딩으로 인한 QoS 이점을 증명하기 위해 필요한 용어를 다음과 같이 정의한다.

- 애플리케이션이 수행할 전체 작업량 (Workload): W_{Total}
- 오프로딩 여부를 확인하고, 오프로딩을 위한 데이터를 전송하는 등 소스 노드에 설치된 노드매니저의 오프로딩 관련 작업량: W_{SNM}
- 오프로드된 컴포넌트의 메소드 실행을 요청하고, 결과값을 소스 노드로 전달하는 등 목표 노드에 설치된 노드매니저의 오프로딩 관련 작업량: W_{TNM}
- 모바일 노드의 프로세서 속도(Speed): S_M
- 목표 노드 (서버 노드, 클라우드 노드, 데스크탑 노드)의 프로세서 속도: S_T
- 모바일 노드에서 작업처리시 소모되는 에너지량: P_M
- 모바일 노드에서 목표노드로 데이터 전송시 소모되는 에너지량: P_{Tr}
- 목표 노드에서 작업처리시 대기하면서 모바일 노드에서 소모되는 에너지량: P_{Idle}
- 두 노드(M,T) 간 네트워크 대역폭 : $BW_{M,T}$ 또는 $BW_{T,M}$
- 오프로딩된 컴포넌트 수행에 필요한 입력 데이터 크기: $SIZE_{In}$
- 오프로드된 컴포넌트의 실행 후 반환값 크기: $SIZE_{RV}$

오프로딩이 실행되지 않은 경우 실행시간($RT_{withoutOffloading}$)과 자원소모량($EC_{wihtoutOffloading}$)은 다음과 같다.

$$RT_{withoutOffloading} = \frac{(W_{Total} + W_{SNM})}{S_M} \quad (1)$$

$$EC_{withoutOffloading} = \frac{(W_{Total} + W_{SNM})}{S_M} \times P_M \quad (2)$$

실행 시간은 처리되는 양을 프로세서 속도를 나눈 값이며, 자원 소모량은 실행 시간과 소모되는 에너지량의 곱[28]이다. 오프로딩이 일어나지 않더라도 노드매니저는 오프로딩 여부를 확인해야 하므로, 분모는 $W_{Total}+W_{NM}$가 된다.

오프로딩은 1) 모바일 노드의 노드매니저가 W_{SNM} 처리, 2) 오프로딩 관련 데이터 전송, 3) 목표 노드의 노드매니저가 오프로드된 컴포넌트와 W_{TNM} 처리, 4) 실행 결과값 전송 단계를 거쳐 실행된다. 그러므로, 총 실행시간($RT_{staticOffloading}$)과 자원 소모량($EC_{staticOffloading}$)은 이를 각각 고려하여 다음과 같이 정의한다.

일반적으로 $S_M \ll S_T$, $W_{TNM} \ll W_{Total}$이기 때문에 $W_{Total}/S_M \gg (W_{Total}+W_{TNM})/S_T$ 라고 할 수 있다. 여기에, $SIZE_{Input}$와 $SIZE_{RV}$의 값이 작다면, 네트워크 오버헤드가 감소하여 오프로딩으로 인한 $RT_{withoutOffloading} \gg RT_{staticOffloading}$가 되며, $P_{Idle} \ll P_M < P_{Tr}$ [28]이기 때문에 $EC_{withoutOffloading} \gg EC_{staticOffloading}$가 됨을 증명할 수 있다.

5.2 동적 오프로딩 (Dynamic Offloading) 기법

1)동적 오프로딩 구조

문제가 발생했을 때 문제가 있는 모바일 애플리케이션을 런타임시에 목표 노드 쪽으로 오프로드시키는 것을 동적 오프로딩이라고 한다. Fig. 9는 동적 오프로딩의 정적 배치 상태를 보여준다. 정적 오프로딩과는 달리, QoS 문제가 발생하면 목표 노드에 애플리케이션을 런타임에 오프로드한다.

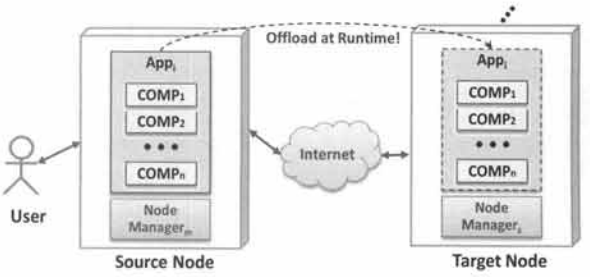

Fig. 9. Structural View of Dynamic Offloading

2) 동적 오프로딩 런타임 절차

정적 오프로딩과 마찬가지로, 오프로딩 여부, 오프로딩의 목표 노드 등은 활동 ❺또는 활동 ❼에서 클러스터 매니저에 의해 이미 결정된다. 그러므로, 노드매니저는 이 정보를 이용하여, Fig. 10에 명시된 동적 오프로딩 절차를 수행한다.

*source:NodeManager*는 오프로딩 수행 시, 오퍼레이션 이름(opName)과 사용자의 입력값 (inputData)외에 목표 노드의 노드 매니저에 실행시킬 애플리케이션 패키지 파일(app)을 offload( )의 매개변수로 전달하는 점 외에 정적 오프로딩과 동일하다. 이 후, *target:NodeManager*는 전달 받은 패키지 파일을 이용하여 *offloaded:Application_n*을 설치하고, *offloaded:Application_n*의 opA( )를 호출하여 결과값을 반환받는다.

동적 오프로딩 설계에서는 런타임에 동적으로 애플리케이션을 설치하고 메소드를 호출하여 오프로딩을 실행하게 하는 것이 핵심이며, 이를 위해 *target:NodeManager*는 *java.lang.reflect.ClassLoader* 이용하여 설계된다.

$$RT_{staticOffloading} = \frac{W_{SNM}}{S_M} + \frac{SIZE_{Input}}{BW_{M,T}} + \frac{(W_{TNM} + W_{Total})}{S_T} + \frac{SIZE_{RV}}{BW_{T,M}} \quad (3)$$

$$EC_{staticOffloading} = \frac{W_{SNM}}{S_M} \times P_C + \frac{SIZE_{Input}}{BW_{M,T}} \times P_{Tr} + \frac{(W_{TNM} + W_{Total})}{S_T} \times P_{Idle} + \frac{SIZE_{RV}}{BW_{T,M}} \times P_{Tr} \quad (4)$$

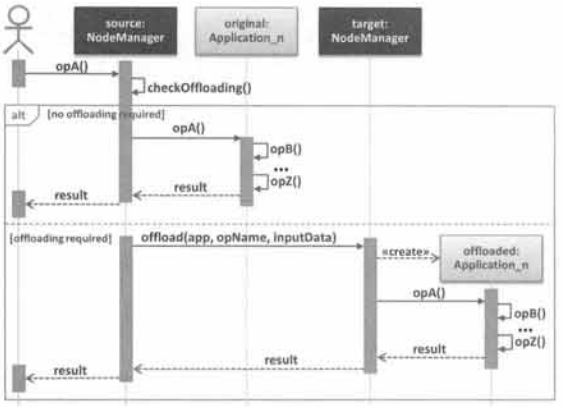

Fig. 10. Behavioral View of Dynamic Offloading

3) 수식을 통한 동적 오프로딩의 이점 증명

동적 오프로딩으로 인한 QoS 이점을 증명하기 위해 추가적으로 다음 용어를 정의한다.

• 오프로드될 애플리케이션 패키지: *Pkg*

모바일 노드에서의 애플리케이션 실행 시간은 정적 오프로딩과 동일하다. 동적 오프로딩이 실행되는 경우, 실행시간($RT_{DynOffloading}$)과 자원소모량($EC_{DynOffloading}$)은 다음과 같다.

여기서 유의할 점은 $RT_{StaticOffloading}$의 경우와 W_{TNM} 동일한 용어를 사용하고 있지만, 정적 오프로딩의 W_{TNM}의 값이 동적 오프로딩의 W_{TNM} 값보다 작다. 동적 오프로딩인 경우에 W_{TNM}은 오프로드될 애플리케이션 설치 작업과 해당 오퍼레이션을 호출하는 작업을 추가적으로 포함하기 때문이다.

일반적으로 $S_M \ll S_T$, $W_{TNM} \ll W_{Total}$이기 때문에 $W_{Total}/S_M \gg (W_{Total}+W_{TNM})/S_T$ 라고 할 수 있다. 여기에, $SIZE_{Pkg}$, $SIZE_{Input}$, $SIZE_{RV}$의 값이 크지 않다면, 네트워크 오버헤드가 감소하여 오프로딩으로 인한 $RT_{withoutOffloading} \gg RT_{DynOffloading}$가 되며, $P_{Idle} \ll P_M < P_{Tr}$이기 때문에 $EC_{withoutOffloading} \gg EC_{DynOffloading}$가 된다는 것을 증명할 수 있다. 동적 오프로딩은 정적 오프로딩에 비해 *Pkg* 전송과 W_{TNM} 처리하는 시간이 소요되므로, 정적 오프로딩의 성능이 더 좋다. 그러므로, MCC에서는 동적 오프로딩 후, 정적 오프로딩으로 전환될 수 있도록 노드의 자원에 크게 영향을 미치지 않으면 오프로드된 애플리케이션을 삭제하지 않는다.

6. 품질 자율 관리 프로세스를 적용한 QoS 안정화 실험

본 장에서는 4장과 5장에서 제안된 품질 자율 관리 프로세스의 적절성과 적용가능성을 평가하기 위해 실험을 수행하고, 이에 대한 결과를 평가한다.

6.1 실험 환경

본 실험을 위해 Fig. 11과 같은 환경을 구성하였다. 실험을 위한 초기 설정은 애플리케이션과 서비스 간의 호출 빈도수와 노드 간의 근접도를 기반으로 2개의 클러스터로 구성되었다. 즉, App_1과 App_2는 S_1^1을 호출하고 *DNode1*과 *SNode2*는 내부 네트워크로 연결되어 있으므로 Cluster1에 속하며, App_3과 App_4는 S_2^1를 호출하기 때문에 Cluster2에 속한다.

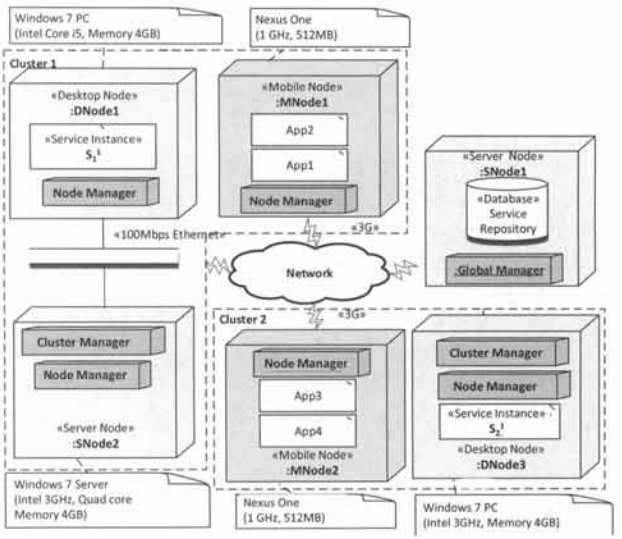

Fig. 11. MCC Deployment for Experiment

6.2 실험 결과 및 평가

본 실험을 위해, 푸아송 분포(Poission Distribution)을 이용하여 애플리케이션과 서비스 인스턴스 간의 호출하였으며, Fig. 12는 이 과정에서 측정된 애플리케이션과 서비스의 품질 변화를 보여준다. 본 실험을 위해 애플리케이션과 서비스 인스턴스의 응답시간을 측정하였고, $Threshold_{High}$ 값에 대한 상대값(0…1)을 QoA과 QoS로 이용하였다. 그림의 윗부분은 4개의 애플리케이션 품질을 나타내며, 아랫 부분은 2개의 서비스 인스턴스의 품질을 나타낸다. 여기서 x축은 실험 경과 시간을 의미한다.

*Cluster1*의 클러스터 매니저가 *MNode1*, *DNode1* 에 설치된 노드매너저로부터 애플리케이션과 서비스 인스턴스의 QoA와 QoS 값을 획득하고, 현재 품질 안정화를 평가한다.

$$RT_{DynOffloading} = \frac{W_{SNM}}{S_M} + \frac{SIZE_{Input} + SIZE_{Pkg}}{BW_{M,T}} + \frac{(W_{TNM} + W_{Total})}{S_T} + \frac{SIZE_{RV}}{BW_{T,M}} \tag{5}$$

$$EC_{DynOffloading} = \frac{W_{SNM}}{S_M} \times P_C + \frac{SIZE_{Input} + SIZE_{Pkg}}{BW_{M,T}} \times P_{Tr} + \frac{(W_{TNM} + W_{Total})}{S_T} \times P_{Idle} + \frac{SIZE_{RV}}{BW_{T,M}} \times P_{Tr} \tag{6}$$

Fig. 12에서 나타나듯이, 50초 부근에 App_1과 App_2에 대한 QoA 뿐 아니라 S_1^1의 QoS가 $Threshold_{low}$보다 낮은 값을 가지기 때문에, Cluster1의 클러스터 매니저는 "품질 저하" 문제가 있다고 판단하였다. 곧바로 Cluster1의 클러스터 매니저는 App_1, App_2, S_1^1이 설치된 노드의 자원과 S_1^1 호출에 대한 로그를 파악하여, 단시간 내에 집중 요청된 S_1^1이 원인임을 확인하였다. 그리고 Table 2의 규칙을 이용하여, 클러스터 매니저는 S_1^1을 물리적으로 가까운 곳에 위치한 $SNode_2$에 복제하기로 결정하였다. 그 결과, 53초 부근에서 App_1과 App_2의 QoA가 정상 범위로 향상되었고, S_1^2가 추가되었음을 확인할 수 있다.

50초 부근에 발생한 문제를 해결한 후, Cluster1의 클러스터 매니저는 지속적으로 QoA와 QoS 값을 획득하여 품질 안정화 정도를 평가하였고, 211초 부근에서 App_3의 QoA가 $Threshold_{low}$보다 낮아 다시 한번 문제를 식별하였다. 이전 경우와 유사하게 클러스터 매지너는 App_3 이 설치된 모바일 노드에 자원이 부족하여 품질이 감소하였음을 확인하였다. 그리고, 클러스터 매니저는 $SNode_2$에 동적 오프로딩을 수행하도록 결정하였고, 이 후 QoA가 다시 향상됨을 확인할 수 있었다. App_3의 모든 기능이 모바일 노드에서 실행된 경우 (211초 부근) 이미 해당 요청을 10초 소요하였기 때문에, 수식 (1)의 $(W_{Total}+W_{SNM}/S_M)$의 값이 10임을 알 수 있다. 동적 오프로딩을 수행한 경우의 오프로딩 관련 데이터 전달 시간, 오프로드된 기능 수행 시간, 결과값 반환 시간을 측정한 결과 각각 0.4 sec, 2.5sec, 0.1 sec 이었으며, 수식 (5)의 첫번째 피연산자인 W_{SNM}/S_M을 제외한 연산자에 해당되는 값임을 알 수 있다. 본 실험의 결과는 수식에 대한 해석과 동일함을 확인할 수 있었다.

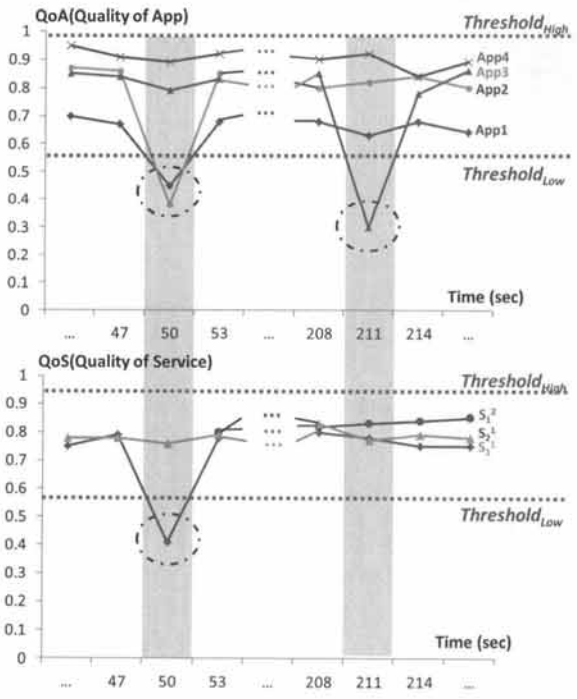

Fig. 12. Quality of Applications and Services

7. 결 론

본 논문에서는 자율 안정화할 수 있는 MCC를 구현하기 위한 실용적인 프로세스와 MCC에 적용될 수 있는 여러 품질 안정화 기법 중 오프로딩을 상세한 수준으로 설계하였다. 먼저 MCC에 대한 메타모델을 정의하였고, QoS 관리로 인한 병목현상 등의 오버헤드를 줄이기 위해 클러스터 개념을 도입하였다. 그리고, 자율 컴퓨팅 개념을 이용하여, 총 8단계로 구성된 품질 자율 관리 프로세스 정의하였다. 제안된 프로세스는 컴포넌트 오프로딩 외에 서비스 이주 및 복제, 서비스 리라우팅, 서비스 재클러스터링 등 다양한 종류의 QoS 품질 향상 기법과 상호연동된다. 이 중, 실용적 수준으로 연구가 많이 진행되지 않은 오프로딩 기법을 MCC 메타모델에 제시된 여러 요소들과 상호작용하여 QoS 문제를 효과적으로 해결할 수 있도록 초점을 맞추어 설계하였다. 마지막으로, 품질 자율 관리 프로세스의 적절성을 증명하기 위해 실험을 수행하여, 제한된 자원을 가지는 모바일 디바이스의 QoS 불안정화 문제를 포함한 클라우드 서비스를 구독하는 모바일 애플리케이션의 품질을 관리하는데 여러 기술적 이슈를 효과적으로 해결할 수 있음을 확인할 수 있었다.

추후 제안된 프로세스를 복잡한 환경에 적용함으로써 각 활동에 적용된 기법을 정제 및 최적화하여 실제 MCC 환경에서 자율적으로 품질 문제를 해결할 수 있도록 한다.

참 고 문 헌

[1] B. König-Ries and F. Jena, "Challenges in Mobile Application Development," *it-Information Technology*, Vol.52, No.2, pp. 69-71, 2009.

[2] G.H. Forman and J. Zahorjan, "The Challenges of Mobile Computing," *Computer*, Vol.27, No.4, pp.38-47, 1994.

[3] M. Fernando, S.W. Loke, and W. Rahayu, "Mobile Cloud Computing: A Survey," *Future Generation Computer Systems*, Vol.29, pp.84-106, 2013.

[4] L. Guan, X. Ke, M. Song, and J. Song, "A Survey of Research on Mobile Cloud Computing," *Proc. 2011 10th IEEE/ACIS International Conf. on Computer and Information Science (ICIS 2011)*, pp.387-392, Dec., 2011.

[5] Y. Natchetoi, V. Kaufman, A. Shapiro, "Service-Oriented Architecture for Mobile Applications," *Proc. 1st Int'l Workshop on Software architectures and mobility (SAM 2008)*, pp.27-32, 2008.

[6] B.G. Chun and P. Maniatis, "Dynamically Partitioning Applications between Weak Devices and Clouds," *Proc. the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond (MCS 2010)*, Article No.7, 2010.

[7] B.G. Chun, S.H. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution between Mobile Device and Cloud," *Proc. 6th European Conf. on Computer Systems*

(EuroSys 2011), pp.301-314, 2011.

[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davis, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, Vol.8, No.4, pp.14-23, Oct., 2009.

[9] E.Cuervo, A. Balasubramanian, D.K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smarphones Last Longer with Code Offloaded," *Proc. 8th Annual Int'l Conf. Mobile Systems, Applications, and Services (Mobisys 2010)*, pp.49-62, 2010.

[10] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G.S. Sukhatme, "An Architecture-driven Software Mobility Framework," *The Journal of Systems and Software*, Vol.83, pp.972-989, 2010.

[11] S. Han, S. Zhang, Y. Zhaing, and C. Fan, "An Adaptable Software Architecture based on Mobile Components in Pervasive Computing," *Proc. the 6th International Conf. on Parallel and Distributed Computing, Applications, and Techniques (PDCAT 2005)*, pp.309-311, 2005.

[12] X. Li, H. Zhang, and Y. Zhang, "Deploying Mobile Computation in Cloud Service," *Proc. 1st Int'l Conf. on Cloud Computing (CloudCom 2009), Lecture Notes in Computer Science (LNCS) 5931*, pp.301-311, 2009.

[13] K. Yang, S. Ou, and H.H. Chen, "On Effective Offloading Services for Resource-Constrained Mobile Devices Running Heavier Mobile Internet Applications," *IEEE Communications Magazine*, Vol.46, No.1, pp.56-63, 2008.

[14] Y. Ye, N. Jain, L. Xia, S. Joshi, I. Yen, F. Bastani, K.L. Cureton, and M.K. Bowler, "A Framework for QoS and Power Management in a Service Cloud environment with Mobile Devices," *Proc. the 5th IEEE Int'l Symposium on Service Oriented System Engineering (SOSE 2010)*, pp. 236-343, 2010.

[15] S.H. Hung, C.S. Shih, J.P. Shieh, C.P. Lee, and Y.H. Huang, "Executing Mobile Applications on the Cloud: Framework and Issues," Computers and Mathematics with Applications, Vol.63, No.2, pp.573-587, Jan., 2012.

[16] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing Performance, Energy, and Quality in Pervasive Computing," *Proc. 22nd Int'l Conf. Distributed Computing Systems (ICDCS 2002)*, pp.217-226, 2002.

[17] P.K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb, "Simplifying Cyber Foraging for Mobile Devices," *Proc. 5th USENIX Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 2007)*, pp.272-285, 2007.

[18] S. Kalasapur and M. Kumar, "Scavenger: Transparent Development of Efficient Cyber Foraging Applications," *Proc. IEEE Int'l Conf. Pervasive Computing and Communications (PerCom 2010)*, pp.217-226, 2010.

[19] M. Sharifi, S. Kafaie, and O. Kashefi, "A Survey and Taxonomy of Cyber Foraging of Mobile Devices," *IEEE Communications Surveys & Tutorials*, preprint, 17 Nov., 2007.

[20] F. Gillett, "Future View: The New tech Ecosystems of Cloud, Cloud Services, And Cloud Computing," *Technical Report*, Forrester Research, August, 2008.

[21] Harrington, P., *Machine Learning in Action*, Manning Publication, 2012.

[22] Witten, I.H., Frank, E., and Hall, M.A., *Data Mining: Practical Machine Learning Tools and Techniques*, Third Edition, Morgan Kaufmann, 2011.

[23] H. Ludwig, A. Keller, A. Dan, R.P. King, and R. Franck, *Web Service Level Agreement (WSLA) Language Specification*, Version 1.0, IBM Corporation, January, 2003.

[24] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein C., Introduction to Algorithms, Third Edition, The MIT Press, 2009.

[25] Lee, J.Y., La, H.J., and Kim, S.D., "A Practical Framework for Self-Stabilization in Service-based Mobile Ecosystem," *Journal of Information Science and Engineering*, Institute of Information Science (To Appear).

[26] Lee, J.Y. and Kim, S.D., "Software Approaches to Assuring High Scalability in Cloud Computing," *In Proceedings of the 7th IEEE International Conference on e-Business Engineering (ICEBE 2010)*, pp.300-306, 2010.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995.

[28] K. Kumar and Y.H. Lu, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?," IEEE Computer, Vol.43, No.4, pp.51-56, 2010.

라 현 정

e-mail : hjla80@gmail.com

2003년 경희대학교 우주과학과(이학사)

2006년 숭실대학교 컴퓨터학과(공학석사)

2011년 숭실대학교 컴퓨터학과(공학박사)

2011년~현 재 숭실대학교 모바일 서비스 소프트웨어공학 센터 연구 교수

관심분야 : 서비스 지향 컴퓨팅 (Service Oriented Computing), 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스 (Mobile Service)

김 수 동

e-mail : sdkim777@gmail.com

1984년 Northeast Missouri State University 전산학(학사)

1988년/1991년 The University of Iowa 전산학(석사/박사)

1991년~1993년 한국통신 연구개발단 선임연구원

1994년~1995년 현대전자 소프트웨어연구소 책임연구원

1995년 9월~현 재 숭실대학교 컴퓨터학부 교수

관심분야 : 서비스 지향 아키텍처(SOA), 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스(Mobile Service), 객체지향 S/W공학, 컴포넌트 기반 개발 (CBD), 소프트웨어 아키텍처(Software Architecture)