http://dx.doi.org/10.3745/KIPSTD.2012.19D.1.063

복잡도가 높은 모바일 어플리케이션 설계를 위한 아키텍처 패턴과 적용지침

장 정 란[†] · 라 현 정[††] · 김 수 동[†††]

요 약

안드로이드 OS, iOS 등 여러 모바일 디바이스 운영체제를 통해서 모바일 디바이스는 다양한 소프트웨어 어플리케이션을 설치·운영하는 모바일 컴퓨팅 기능을 제공하고 있다. 나아가 개인용 컴퓨팅 용도뿐 아니라 엔터프라이즈를 위한 어플리케이션 클라이언트 단말기로도 사용될 것으로 예상한다. 그러나 모바일 디바이스는 자원의 제한성, 모바일 무선 네트워크 지원 능력 등 이전의 피처폰이나 개인용 컴퓨터에는 없는 특징이 있다. 따라서, 모바일 어플리케이션을 개발하기 위해서는 어플리케이션의 설계 시에 모바일 디바이스가 가진 속성을 반영하는 방법이 필요하다. 아키텍처는 소프트웨어의 특성을 반영하는 비기능적 요구사항을 충족시켜 소프트웨어의 품질을 향상시키는 설계 방법이다. 아키텍처 설계를 하기 위해서 모바일 어플리케이션의 특징으로부터 아키텍처 비기능적 요구사항인 아키텍처 드라이버를 추출한다. 기존의 소프트웨어 아키텍처 설계는 여러 패턴 제공하고 있지만, 모바일 어플리케이션의 특징을 반영하지 않았기 때문에 적용하기가 어렵다. 본 논문에서는 모바일 어플리케이션의 주요 특징을 반영한 어플리케이션 개발을 위해, 모바일 어플리케이션에 적합한 아키텍처 패턴을 정의하고, 각 패턴을 설계하는 지침을 제안한다. 먼저, 모바일 디바이스 및 어플리케이션의 특징을 정의하고, 이 특징으로부터 아키텍처 드라이버를 추출한다. 그리고 모바일 어플리케이션 개발을 위한 아키텍처 패턴을 정적 뷰와 동적 뷰 관점으로 설명하고, 각 아키텍처 패턴을 적용하여 모바일 어플리케이션 아키텍처를 설계하기 위한 적용 지침을 제공한다. 그리고 제시된 아키텍처 패턴의 사례 연구를 통해 적용 가능성을 보여준다. 마지막으로 제시된 아키텍처 패턴을 아키텍처 드라이버 별로 평가하고 또한 기존 연구와의 비교를 통해 본 논문에서 제시하는 아키텍처 패턴을 평가한다.

키워드 : 아키텍처 패턴, 모바일 어플리케이션, 모바일 디바이스, 적용지침

Practical Architectural Patterns and Guidelines for Designing Complex Mobile Applications

Jang Jeong Ran[†] · Hyun Jung La[††] · Soo Dong Kim[†††]

ABSTRACT

Mobile devices with Android OS and iOS have been emerged as mobile computing devices where various software applications are deployed. Furthermore, they are anticipated to be used not only for traditional personal computing but also for enterprise computing. However, such mobile devices have their intrinsic characteristics such as limited resources and flexible network capabilities, which are not revealed in traditional computers. Hence, there is high demand for methods to develop mobile applications with reflecting their intrinsic characteristics. Since those characteristics belong to non-functional requirements, they should be reflected in architecture design while designing mobile applications. To design architecture, the architecture drivers that are architecture non-functional requirements are decided from mobile application characteristics. Conventional architecture design methods do not consider those characteristics so that the methods cannot be straightforwardly applied to mobile applications. In this paper, to efficiently develop mobile applications reflecting those characteristics, we propose a set of architecture patterns and define a guideline to apply those patterns. First, we define the characteristics of mobile applications distinguished and derive architectural drivers from them. Then, we propose architecture patterns in terms of static and dynamic views and define an architectural guideline to apply the patterns to designing architecture for mobile application. And, we perform case studies to verify the applicability of proposed patterns. Finally, we assess the proposed architecture patterns by proving how the patterns can fulfill identified architecture drivers and by comparing our approach with previous works.

Keywords : Architecture Patterns, Mobile Applications, Mobile Devices, Architectural Guideline

※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2009-0076392).
† 준 회 원 : 숭실대학교 컴퓨터학과 석사과정
†† 정 회 원 : 숭실대학교 모바일 서비스 소프트웨어공학센터 연구교수(교신저자)
††† 종신회원 : 숭실대학교 컴퓨터학부 교수

논문접수 : 2011년 8월 30일
수 정 일 : 1차 2011년 12월 12일
심사완료 : 2011년 12월 13일

1. 서 론

최근 들어 아이폰과 안드로이드폰 등의 다양한 스마트폰을 비롯하여 아이패드와 같은 태블릿 PC도 출시되면서, 다양한 형태의 모바일 디바이스가 주목을 받고 있다. 모바일 디바이스는 무선 인터넷 접속 기능을 제공하는 휴대 가능한 모든 종류의 장비들을 총칭하는 용어로서, 유연한 무선 인터넷 접속성과 높은 이동성을 제공한다는 장점을 가지고 있다. 이런 이유로, 미국뿐 아니라 한국에서도 스마트폰 이용자가 급증하고 있으며, 일부에서는 데스크톱 컴퓨터를 대체할 클라이언트 장비가 될 것이라고 예상하고 있다. 그러나 일반적으로 모바일 디바이스는 휴대를 목적으로 하여 크기가 작으므로 전통적인 컴퓨터에 비해 컴퓨팅 자원이 부족하다. 그 결과 자원 소비가 많고 높은 복잡도를 가진 어플리케이션은 모바일 디바이스에서 설치, 운영되기 어렵다. 하지만 모바일 디바이스는 풍부한 네트워크를 지원할 수 있는 능력 등의 장점이 있어 서비스 등의 외부 자원을 사용하기가 용이하다. 즉, 모바일 어플리케이션의 특징을 반영할 수 있는 모바일 어플리케이션만을 위한 개발 기법이 필요하다.

아키텍처는 시스템의 중요한 설계 결정의 집합일 뿐 아니라 시스템 구성요소들의 구조이다. 아키텍처는 전체적인 시스템의 구조를 표현하기 때문에 비기능적 요구사항인 성능, 보안 등의 품질 속성을 만족하게 할 수 있도록 설계 사항이 결정된다. 앞서 언급한 모바일 어플리케이션의 특징은 모바일 어플리케이션 개발 시 고려해야 하는 비기능적 요구사항이라 할 수 있다. 그러므로 모바일 어플리케이션의 특징의 반영은 모바일 어플리케이션의 아키텍처 설계 단계에서 이루어져야 한다. 현재 널리 사용되고 있는 아키텍처 설계 프로세스에서는 이러한 모바일 어플리케이션의 특징을 반영하지 않아 비효율적으로 모바일 어플리케이션 개발이 이루어지고 있다.

아키텍처 패턴은 소프트웨어 개발시 반복적으로 발생하는 설계 문제에 대한 아키텍처 설계 결정사항의 집합이다[1]. 따라서, 복잡한 기능성을 가지고 있는 모바일 어플리케이션의 아키텍처 설계시 모바일 어플리케이션이 가지고 있는 제약 사항을 해결하고 활용하기 위해서는 모바일 어플리케이션에 대한 아키텍처 패턴 및 설계 기법이 필요하다.

본 논문에서는 모바일 특징을 효과적으로 모바일 어플리케이션 설계에 반영할 수 있는 아키텍처 패턴을 제안하고 아키텍처 패턴을 설계하기 위한 설계 지침을 제안한다. 모바일 어플리케이션의 특징으로부터 모바일 어플리케이션의 아키텍처 드라이버를 식별한다. 식별된 아키텍처 드라이버를 반영하기 위한 모바일 어플리케이션을 위한 아키텍처 패턴을 제안한다. 이 아키텍처 패턴은 현재 널리 사용되는 모바일 어플리케이션의 특징 및 범용적인 모바일 어플리케이션의 아키텍처 드라이버와 앞으로 유망한 형태인 모바일 엔터프라이즈 컴퓨팅을 지원하는 고품질의 모바일 어플리케이션을 설계하기 위하여 정의된 것이다. 그리고, 제안된 아키텍처 패턴에 대한 구조와 동적 뷰를 통해 아키텍처의 장, 단점을 설명하고, 제안된 아키텍처 패턴을 실제 모바일 어플리케이션에 적용하여 아키텍처를 효과적으로 활용하기 위한 적용 지침을 제시한다. 제시된 아키텍처 패턴의 적용가능성을 보여주기 위해 사례연구를 실행하고 사례연구에서 얻은 결과 및 교훈을 설명한다. 또한 제시된 아키텍처 패턴이 모바일 어플리케이션을 위한 아키텍처 드라이브를 만족시키는지 알아보기 위하여 아키텍처 드라이버 별로 아키텍처 패턴을 평가하고 또한 기존 연구와의 비교를 통해 본 논문이 독창적임을 보여준다. 제시된 아키텍처 패턴을 활용하면 효과적으로 고품질의 모바일 어플리케이션을 개발할 수 있다.

논문의 구성은 다음과 같다. 3장에서 모바일 어플리케이션의 특징을 정의하고, 4장에서는 이를 반영한 아키텍처를 설계하기 위해 아키텍처 드라이버를 제시한다. 5장에서는 고품질의 모바일 어플리케이션 개발을 위한 아키텍처 패턴을 제안한다. 6장에서는 제시된 아키텍처 패턴의 적용가능성을 보여주기 위해 사례 연구를 실행하고, 7장에서는 아키텍처 패턴을 평가하고 또한 기존 연구와 비교한다.

2. 관련 연구

모바일 어플리케이션 아키텍처에 대한 관련 연구를 분석하기 위하여, 클라이언트/서버, 서비스 등 기능 분할을 이용하여 아키텍처 설계하는 연구에 대해 알아본다.

Lee의 연구에서는 클라이언트/서버 아키텍처와 클라이언트-서버 간의 연결성(connectivity)을 고려하여 모바일 어플리케이션 아키텍처의 종류를 나열하였다[2]. 이를 위해, 먼저 클라이언트는 씬 클라이언트(Thin Client), 팻 클라이언트(Fat Client), 웹 페이지 호스팅 클라이언트로 분류하였고, 서버는 1 티어, 2 티어, 3 티어로 분류하였다. 그리고 연결성은 '항상 연결', '부분적 연결', '연결 없음'으로 분류하였다. 그리고 이들을 다양하게 조합한 여러 종류의 아키텍처 패턴을 제안하였다. 이들의 연구는 다양한 종류의 모바일 어플리케이션 아키텍처를 제안하였지만, 각 아키텍처를 효과적으로 설계하는 방법은 다루지 않았다.

Gruhn의 연구에서는 웹 기반의 온라인 아키텍처, 리치 클라이언트 온라인 아키텍처, 리치 클라이언트 하이브리드(hybrid) 아키텍처, 팻 클라이언트 오프라인 아키텍처의 4가지 종류의 모바일 어플리케이션 아키텍처를 제안하였다[3]. 이 4가지의 아키텍처는 어플리케이션이 네트워크에 의존하는지에 대한 유무와 클라이언트 어플리케이션이 데이터베이스를 사용하는지에 대한 유/무를 기준으로 분류한 것이다. 그리고 서비스, 데이터, 소스 코드의 중복성, 소프트웨어 분포, 사용자 인터페이스 설계, 상호작용을 위한 기법, 보안성 등의 관점을 고려하여 가장 적절한 아키텍처를 제안하였다. 그러나 이런 비즈니스 요구사항 별로 적합한 아키텍처 종류 외에 이를 설계할 수 있는 방법이 보완되어야 한다.

Dagtas의 연구에서는 Lightweight Architecture (LAW)라는 모바일 어플리케이션을 위한 경량 (lightweight) 컴포넌트 기반 프레임워크를 제안하였다[4]. LAW는 컴포넌트 컨테이너, 레지스트리, 컨트롤러, 커뮤니케이션 매니저, 데이터 매니저, 로그 매니저로 구성된다. LAW에서 실행되는 컴포넌트는 OGSi 명세를 준수하여 설계되었다. 클라이언트 측에서 데이

터를 관리해야 하는 부담을 줄이기 위해, 이 연구는 서버 측에서 관리되는 비즈니스 객체의 일부에 대한 복사본인 경량 비즈니스 객체 모델을 제안하였다. 그리고 모바일 어플리케이션을 위한 커뮤니케이션 모델을 정의하였다. 이들의 연구는 프레임워크를 이용하여 모바일 어플리케이션을 설계할 수 있는 방법은 구체적으로 정의하지 않았다.

Natchetoi의 연구에서는 J2ME환경에서 실행 가능한 비즈니스 어플리케이션을 위한 경량 서비스 기반 아키텍처를 제안하였다[5]. 이 연구에서는 적은 량의 데이터 전송 및 저장, 능동적 데이터 적재, 보안을 위해 아키텍처 설계 방법을 제안하였다. Ennai의 연구에서는 모바일 어플리케이션을 위한 서비스지향 프레임워크를 제안하였다[6]. 이 프레임워크는 서비스기반 경량 어플리케이션의 배치와 사용자 컨텍스트에 맞는 적절한 서비스와 장치의 적용, 능동적 서비스 호출과 같은 핵심적인 구조적 고려사항을 만족하고 있다. 아키텍처를 구성하는 컴포넌트는 동적인 서비스 발견 및 바인딩, 상황인지 서비스 분배, 비동기적 푸시 서비스 호출을 위해 서로 상호작용을 지원하도록 설계된다. 위의 연구들은 모바일 디바이스의 특징을 고려하여 서비스를 기반으로 하는 아키텍처와 아키텍처 설계 고려사항을 식별하였다. 그리하여, 모바일 어플리케이션에서 서비스를 사용할 수 있는 기반을 제공하였지만, 모바일 어플리케이션 아키텍처 설계와 관련된 구체적이고 체계적인 지침이 추가될 필요가 있다.

Aaratee의 연구에서는 상황인지 모바일 어플리케이션을 위한 모바일 SOA 프레임워크를 제안하였다[7]. 모바일 디바이스는 저장공간과 컴퓨팅 능력에 제한이 있기 때문에, 엔터프라이즈를 위한 어플리케이션에는 기존의 방법으로는 적합하지 않다. 모바일 SOA의 미들웨어는 경량이고 유연한 CAMA를 구축하고 운영하여 실시간으로 변화하는 컨텍스트가 서비스 될 수 있도록 보장한다. 그러나 상황 인지만을 고려한 모바일 어플리케이션에 대해서 다루어, 모바일 어플리케이션을 위한 아키텍처가 구체적이고 실용적이지만, 범용적으로 사용되기는 한계가 있다.

위의 연구 대부분은 서비스 기반 아키텍처 또는 경량 컴포넌트 기반 아키텍처와 같이 모바일 어플리케이션에 적용될 수 있는 한 개의 특정 아키텍처를 다루었고, 아키텍처 기술이 구조적 관점에서 상당히 추상적 수준에 머물고 있다. 그리고, 기존의 연구에서는 구조적 측면의 설명을 포함하고 있으나, 본 논문에서 기술한 정도의 설계 지침이 매우 미흡하게 정의되었다.

3. 모바일 디바이스 및 어플리케이션의 특징

모바일 디바이스는 전통적인 컴퓨터와는 구별되는 특징이 있으므로 그 특징들은 고품질의 모바일 어플리케이션을 개발하는데 반영되어야 한다. 3장에서는 모바일 컴퓨팅 및 모바일 어플리케이션에 관련된 대표적인 문헌[2][6][8]을 분석하여, 모바일 어플리케이션이 가지는 특징을 알아본다.

높은 이동성 (High Mobility): 모바일 어플리케이션은 높은 이동성을 가진다. 사용자는 이동 중에도 모바일 어플리케이션의 기능을 받을 수 있다. 즉, 모바일 디바이스를 소유하면, 자유롭게 어느 장소에서도 기능을 사용할 수 있게 된다. 높은 이동성은 사용자의 환경 및 상황 정보에 대한 변화를 가져온다. 이것은 풍부한 상황 정보를 활용한 어플리케이션의 개발할 수 있음을 의미하지만, 기존에는 존재하지 않았던 네트워크 끊김, 기기 이상 현상, 잡음 등이 문제가 발생할 수 있다.

자원 제약성: 모바일 디바이스는 휴대하기 편리한 작은 크기로 만들어져 있다. 이것으로 모바일 어플리케이션은 저성능의 모바일용 CPU, 저용량 메모리, 제한된 배터리 등의 한정된 자원 때문에 기능의 제약이 있다. 그러므로 모바일 어플리케이션은 한정된 자원을 이용하여 실행되어야 한다. 또한, 배터리 사용을 줄이기 위하여 어플리케이션 실행시간도 고려해야 한다.

무선 네트워크 사용: 모바일 디바이스의 정의에서 의미하듯이, 모바일 디바이스는 무선 인터넷 접속 기능을 제공한다. 대부분의 모바일 디바이스는 와이파이(Wi-Fi), 또는 통신사에서 제공하는 3G 네트워크를 이용하여 인터넷에 접근한다. 무선 네트워크 특징상 유선 네트워크보다 불안정하므로, 모바일 어플리케이션 기능 수행에 문제를 유발할 수 있다.

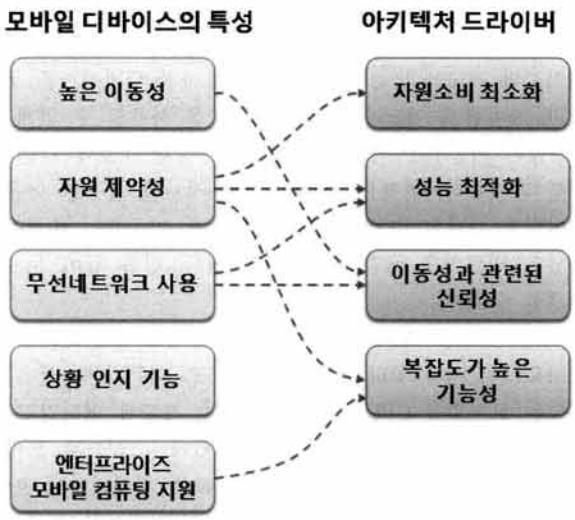
상황인지(Context Aware) 기능: 모바일 디바이스는 가속도, 중력, 자기장, 압력, 온도 등의 여러 센서를 가지고 있다. 이를 활용하여 모바일 어플리케이션은 여러 상황 정보를 감지한다. 예를 들면, 위성 위치확인 시스템(GPS)나 가속도계(Accelerometer) 등을 활용하여 위치정보와 속도정보를 알아낸 수 있다.

엔터프라이즈 모바일 컴퓨팅 지원: 모바일 어플리케이션이 널리 사용될 것으로 전망되면서 간단한 기능뿐 아니라, 엔터프라이즈 어플리케이션의 복잡한 기능까지 제공하는 모바일 어플리케이션의 필요성이 대두되고 있다. 이는 국방분야에서 모바일 컴퓨팅 개념을 이용하여 어플리케이션을 개발하려는 시도와 모바일 전자정부로의 움직임에서 유추할 수 있다. 그러므로, 일정 관리, 소셜 네트워킹과 같은 비교적 간단한 기능 외에 복잡한 기능도 제공할 수 있는 어플리케이션이 기대되고 있다.

모바일 디바이스 및 어플리케이션이 가지는 높은 이동성, 자원 제약성, 상황인지 등의 특징은 기존의 아키텍처에서 고려하지 않은 부분이다. 또한, 무선 네트워크나 엔터프라이즈 컴퓨팅 지원은 기존에도 고려된 상황이지만 모바일이라는 특수한 환경이 더해지게 되면서 네트워크 불안정성과 모바일 디바이스에서의 복잡한 기능 수행 등의 문제를 해결해야 한다.

4. 차세대 모바일 어플리케이션 아키텍처 드라이버

본 장에서는 모바일 어플리케이션의 아키텍처에 모바일 어플리케이션의 특징을 반영하기 위한 아키텍처 드라이버를 추출한다. (그림 1)과 같이 3장에서 식별한 5가지의 모바일



(그림 1) 모바일 디바이스의 특징과 아키텍처와의 유도 관계

디바이스 및 모바일 어플리케이션의 특징으로부터 4가지의 아키텍처 드라이버가 식별되었다.

자원소비 최소화: 모바일 어플리케이션 아키텍처는 모바일 디바이스의 자원 제약성에 따라 자원 소비를 최소화하도록 설계되어야 한다. 어플리케이션 실행 시에 필요한 자원이 부족할 때 성능 등의 품질 저하 문제나 실행이 멈추는 심각한 문제를 일으킬 수 있다. 예를 들면, 애니메이션 효과와 음향 효과 등 메모리 소비가 많은 자원이 필요한 동화 어플리케이션을 개발하였을 경우, 메모리를 2GB 이상을 지원하는 전통적인 컴퓨터에서 실행하는 데 있어 메모리 문제가 거의 없지만, 256MB 메모리가 탑재된 아이패드(iPad) 디바이스에서는 고용량의 데이터, 그래픽, 오디오 등의 자원을 처리하는데 메모리 누수 등 여러 형태의 문제가 발생 되었다.

성능 최적화: 모바일 어플리케이션 아키텍처는 모바일 어플리케이션이 최적의 성능을 낼 수 있도록 설계되어야 한다. 모바일 디바이스는 자원 제약성이 있기 때문에 복잡한 기능을 가진 어플리케이션은 모바일 디바이스에서 동작할 때 최적의 성능을 기대하기 어렵다. 또한, 무선 네트워크를 사용하여 원격에서 기능을 사용할 때에는 네트워크 비용 등의 성능 저하가 발생한다. 그러므로 이를 고려하여 아키텍처가 설계되어야 한다.

이동성과 관련된 신뢰성 보장: 모바일 디바이스는 사용자가 이동 중에 네트워크 사용이 필요한 어플리케이션을 지속적으로 실행할 수 있는 높은 이동성을 제공한다. 그러나 이동성 때문에 사용 중인 무선 통신 연결이 일시적으로 끊어지는 문제나 다른 통신 방식으로 변환 때문에 일시적인 차단 문제가 발생할 수 있다. 예를 들면, 사용자가 건물 내로 들어가면 사용 중인 3G 기반 데이터 네트워크는 끊어질 수 있고 이 때문에 사용 중인 어플리케이션 실행이 정상적으로 수행될 수 없다. 또한, GPS로 위치정보를 수신하여 사용하는 어플리케이션의 경우, 건물 실내 진입으로 인해, 위치 정보 수신이 불가능해지는 문제가 발생한다. 현재 이동성으로 인한 문제가 발생할 경우, 대부분 어플리케이션이 네트워크 연결의 문제를 메시지로 보여주고 실행을 정지하는 방식을 택하고 있다.

복잡도가 높은 기능 수행: 모바일 어플리케이션 아키텍처는 복잡한 비즈니스 로직을 가지거나 계산이 복잡한 기능을 수행할 수 있도록 설계되어야 한다. 모바일 디바이스의 자원 제약성과 엔터프라이즈 어플리케이션의 복잡한 기능까지도 고려하여 아키텍처가 설계되어야 한다.

5. 모바일 어플리케이션 아키텍처 패턴

본 장에서는 4장에서 식별한 네 가지의 아키텍처 드라이버를 기반으로, 효율적이며 실용적인 아키텍처 패턴을 정의 제안한다. 제안된 아키텍처 패턴은 모든 가능한 종류의 모바일 어플리케이션들에 모든 경우에 적용될 수 있는 것은 아니지만, 대부분의 모바일 어플리케이션 개발에 적용 가능하도록 범용성을 고려하여 제안된다.

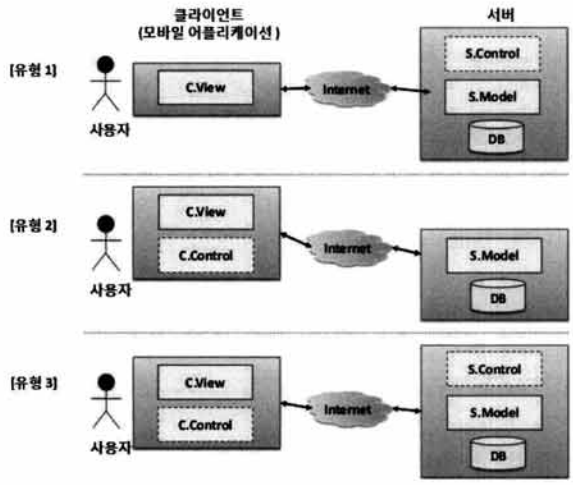
모바일 어플리케이션을 위한 아키텍처 패턴으로 Client-Server 아키텍처, Balanced MVC(Model-View-Control) 아키텍처, 서비스 기반 아키텍처, 그리고 하이브리드 아키텍처의 패턴을 제시하고 있다. 아키텍처 패턴의 정의와 더불어 제안된 패턴을 이용하여 효과적으로 모바일 어플리케이션의 아키텍처를 설계하기 위한 설계 지침을 제안한다. 각 설계 지침은 아키텍처 설계 이전의 산출물인 유즈케이스 모델 및 구조적 모델을 이용하여 아키텍처를 구성하는 컴포넌트와 인터페이스를 도출하도록 한다.

5.1 Client-Server 아키텍처 패턴

개요: Client-Server 아키텍처는 네트워크를 통해서 하나 이상의 자원을 제공하는 서버와 그 자원을 사용하는 클라이언트를 연결하는 아키텍처 스타일이다[9]. 모바일 어플리케이션을 위한 Client-Server 아키텍처 스타일은 기존의 Client-Server 스타일을 수용하며 무선 네트워크 지원과 자원의 제약성을 갖는 모바일 환경에 알맞게 최소한의 자원을 사용하는 아키텍처를 제안한다. 이 아키텍처의 특징은 기본적으로 클라이언트는 모바일 디바이스로 어플리케이션의 유저인터페이스(User Interface, UI) 역할을 담당하고, 서버는 데이터를 담당함으로써, 어플리케이션의 기능에 따라 효율적으로 어플리케이션을 수행할 수 있도록 개발하는 경우에 사용한다. 이 아키텍처는 다음 절에서 설명되는 Balanced MVC의 한 형태로서, Thin-Client Computing 방식을 지원하기 위해 별도의 아키텍처로 제안된다. 즉, 모바일 디바이스의 자원이 부족하거나 기능이 복잡한 경우, 데이터베이스나 복잡한 비지니스 로직을 서버에 배치하고, 클라이언트는 뷰 계층이나 단순한 비지니스 로직만을 가지는 구조이다.

구조적 뷰: 모바일 어플리케이션을 위한 Client-Server 아키텍처 패턴의 내부 컴포넌트는 모바일 개발에 적합한 MVC 패턴을 활용한다. 기존의 Client-Server 아키텍처 패턴에 따라 사용자와 인터페이스 역할을 하는 뷰 계층은 클라이언트에 있고 데이터베이스를 관리하는 모델 계층은 서

버에 있다. 즉, 뷰는 모바일 디바이스를 통해 사용자에게 화면을 보여주고, 모델은 여러 사용자의 모든 데이터를 별도의 저장소에 영속적으로 저장하여 관리한다. 뷰 계층과 모델 계층을 연결하는 컨트롤 계층은 개발할 모바일 어플리케이션의 특징에 따라 클라이언트나 서버에서 활용할 수 있게 한다. 따라서 컨트롤 계층의 위치에 따라서 Client-Server 패턴은 (그림 2)와 같이 3가지 유형으로 분류된다.



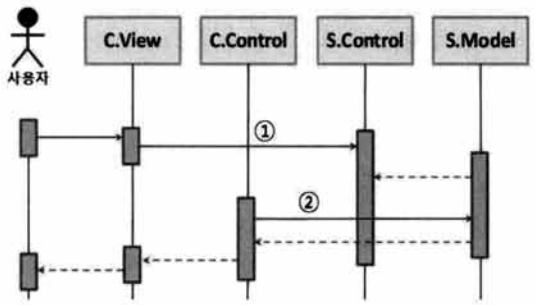
(그림 2) Client - Server 아키텍처의 3가지 유형

유형 1에는 클라이언트에 뷰 계층(*C.View*)만 존재하며, 서버에는 컨트롤 계층(*S.Control*), 모델 계층(*S.Model*), 그리고 데이터베이스가 존재한다. 클라이언트에 있는 뷰 계층은 사용자와의 인터페이스를 담당하고 네트워크를 통해서 서버와 통신한다. 서버에서 컨트롤 계층은 기능을 수행하고 모델 계층을 통해 데이터를 관리한다. 이 유형은 모바일 디바이스 상에서 운영하기에는 복잡한 기능을 서버로 위임하여 수행하게 하는 어플리케이션에 적용한다. 또한, 기능을 수행하는 데 있어서 데이터에 자주 접근해야 하는 경우에 적합하다.

유형 2에는 클라이언트에 뷰 계층과 컨트롤 계층(*C.Control*)이 존재하고, 서버에는 모델 계층과 데이터베이스가 존재한다. 클라이언트의 뷰 계층은 유형 1의 역할과 동일하며, 클라이언트에 존재하는 컨트롤 계층은 뷰 계층에서 요구하는 기능을 직접 받아 프로세스를 실행한다. 이 유형에서는 뷰 계층에서 컨트롤 계층으로 유형 1과 반대로 네트워크를 거치지 않고 직접 접근이 가능하므로 처리 속도가 빠르다. 하지만, 모든 기능을 모바일 디바이스 상에서 처리하기 때문에 많은 자원을 필요로 하는 복잡한 기능을 수행할 수 없다는 점을 고려해야 한다. 또한, 네트워크의 비용을 고려하여 컨트롤 계층과 모델 계층의 상호작용이 비교적 드문 경우, 즉 컨트롤 계층과 모델 계층 간의 데이터 이동이 적은 경우에 잘 적용된다.

유형 3에는 클라이언트에 뷰 계층이 존재하고 서버에 모델 계층과 데이터베이스가 존재한다. 그리고컨트롤 계층이 클라이언트와 서버 양쪽에 모두 존재하는 방식으로 실행됨을 의미한다. 유형 1과 유형 2와 다르게 컨트롤 계층이 뷰 계층과 모델 계층 모두에 빈번한 상호작용이 있는 기능이 필요한 경우, 이 유형을 적용한다. 클라이언트 컨트롤 계층에서는 뷰와 상호작용하는 기능을 실행하게 하고, 서버 컨트롤 계층에서는 모델 계층과 상호작용이 필요한 기능은 실행하여 기능을 동시에 수행할 수 있도록 한다. 이는 높은 병렬 처리 효과와 더불어 네트워크 오버헤드로 인한 성능의 문제를 효과적으로 해결한다.

동적 뷰: Client-Server 아키텍처의 모든 경로는 MVC 방식으로 View에서 Control을 통해 Model 순으로 진행된다. 동적 뷰는 클라이언트와 네트워크가 어떠한 경로로 데이터를 전송하는지를 초점을 두고 있다. (그림 3)은 Client-Server 아키텍처의 두 가지 경로를 보여주고 있다.



(그림 3) Client - Server 아키텍처 패턴의 상호작용 경로

경로 1은 *C.View*에서 네트워크를 통해 사용자가 요청한 기능을 *S.Control*로 전달한다. *S.Control*은 *C.View*로부터 요청받은 기능을 수행하여 그 결과를 다시 *C.View*로 보낸다. 또한, 데이터가 필요한 경우 *S.Model*에 접속하여 데이터를 받아온다. *S.Model*은 *S.Control*에서 요청한 데이터를 데이터베이스가 처리하게 한다. 경로 1은 웹사이트를 예로 들 수 있다. 사용자가 웹 브라우저를 통해 정보를 얻고, 사용자가 요청하는 정보는 웹 서버 측에서 처리하여 결과만 다시 브라우저에 보여준다.

경로 2는 *C.Control*에서 *S.Control*로 위치가 이동한다. 이 외에는 경로 1과 같다. 경로 2는 퍼즐 게임에서 예를 들 수 있다. 퍼즐의 이미지들은 서버에 저장되어 있고, 사용자가 해당 어떠한 레벨의 퍼즐을 선택하였을 때, 게임을 하는 데 필요한 이미지 데이터는 서버에서 가져오고, 이미지를 섞고, 위치시키고, 퍼즐을 맞추는 기능은 클라이언트마다 독립적으로 수행한다.

적용 지침: Client-Server 아키텍처 패턴에서 설계 시 고려할 사항은 어플리케이션의 기능을 수행하는 컨트롤 컴포넌트의 위치에 있다. 그리고 클라이언트에 모델 컴포넌트가 필요한지의 여부이다. 클라이언트에 모델 컴포넌트가 필요할 경우에는 5.2 Balanced MVC아키텍처 패턴을 사용한다. 먼저, 기본적으로 요구사항 명세서와 유즈케이스 모델의 작성을 통해 기능을 분석한다.

스텝 1. 패턴 유형 선택

스텝 2. 아키텍처 패턴 유형에 따른 컴포넌트의 배치

스텝 3. 컴포넌트의 인터페이스 설계

스텝 1: Client-Server 아키텍처 패턴의 3가지 유형 중 어플리케이션에 알맞은 유형을 선택하기 위해서는 요구사항 명세서에서 데이터와 기능을 분석해야 한다. 유즈케이스 기능 복잡도, 사용자-시스템 간의 메시지 교환 빈도, 교환되는 데이터 양을 이용하여 유형을 판단한다.

유즈케이스의 기능 복잡도는 기능 점수를 이용하여 파악할 수 있다. 기능 점수는 사용자에게 제공되는 기능성의 양을 측정하는 대표적인 기법[10]이므로, 본 논문에서도 이 기법을 이용하여 유즈케이스 별 기능 복잡도를 유도한다. 이 결과로 상대적으로 복잡도가 높은 유즈케이스에 대한 기능은 서버 측에 위치하도록 하고, 낮은 복잡도를 유즈케이스는 클라이언트 측에 배치하도록 한다.

사용자-시스템 간 메시지 교환 빈도는 유즈케이스 명세서로부터 도출할 수 있다. 유즈케이스 명세서에는 해당 유즈케이스를 수행하기 위한 워크플로우를 기술하며, 액터와 시스템과의 상호작용 빈도를 이용하여 메시지 교환 빈도를 유추할 수 있다. 이 결과로, 사용자와 밀접하게 연관되는 흐름은 클라이언트 측에 배치를 하고, 사용자와 빈도수가 낮은 흐름은 서버 측에 위치하는 후보가 된다.

교환되는 데이터 양 역시, 유즈케이스 명세서의 워크플로우로부터 유추할 수 있다. 각 메시지 교환에는 데이터가 매개변수 형태로 전달되는데, 이 매개변수로 전달되는 데이터 타입와 데이터 개수를 이용하여 교환되는 데이터 양을 유추한다. 이 결과로, 많은 양의 데이터가 교환되는 기능의 경우는 클라이언트 측에 위치하도록 한다.

이렇게, 유즈케이스 기능 복잡도, 사용자-시스템 간의 메시지 교환 빈도, 교환되는 데이터 양의 값을 예측하여, 서버에 위치하는 기능 및 클라이언트에 위치하는 기능, 각 기능의 역할(Model, View, Controller)를 결정함으로써, 적절한 유형을 선택한다. 이 때, 모바일 어플리케이션의 특징 및 사용자 요구사항을 반영하여, 각 기준 요소 별로 다른 가중치를 두어 유형을 결정한다. 예를 들어, 기능 복잡도가 중간이며, 사용자-시스템 간의 메시지 교환 빈도 잦으며, 교환되는 데이터 양이 낮은 기능인 경우에는 해당 기능의 위치가 명확히 결정하기 위하여, 사용자가 더 중요하게 생각하는 기준에 높은 가중치를 두어 적절한 유형을 선택한다.

스텝 2: 유즈케이스 모델을 이용하여 각 계층에 포함되는 컴포넌트를 도출한다. 뷰에 포함되는 컴포넌트는 유즈케이스 명세서에 기술된 액터와 시스템 간의 상호작용을 참조하여 도출한다. 모델에 포함되는 컴포넌트는 데이터 저장을 목적으로 하는 시스템 기능들을 분석하여 도출한다. 컨트롤에 포함되는 컴포넌트는 유즈케이스 모델에 명시된 기능 그룹을 이용하여 도출한다.

스텝 3: 이 단계에서는 유즈케이스 명세서를 기반으로 *C.Control*과 *S.Control*에 속한 컴포넌트들의 인터페이스를 정의한다. *C.Control*과 *S.Control*에 포함된 클래스들과 매핑되는 기능 그룹을 먼저 결정한다. 그리고 기능 그룹에 포함되는 유즈케이스를 확인한다. 유즈케이스 명세서의 액터와 클라이언트 시스템 간의 상호작용, 클라이언트 시스템과 서버 시스템 간의 상호작용을 분석함으로써, *C.Control*과 *S.Control*의 컴포넌트에 대한 인터페이스와 주요 기능을 정의할 수 있다.

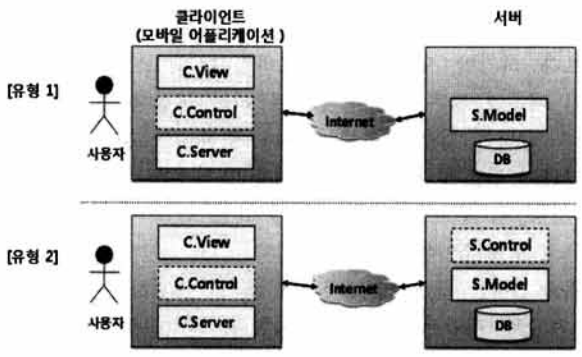
5.2 Balanced MVC 아키텍처 패턴

개요: MVC 아키텍처는 객체 지향 시스템에서 널리 사용되는 아키텍처 설계 방식이다[11]. 3장에서 언급한 모바일 어플리케이션의 특징으로는 기존의 MVC 아키텍처가 그대로 적용되기 어렵다. 그러므로 모바일 어플리케이션의 특징을 고려하여 안드로이드 모바일 아키텍처[12]의 내용을 기반으로 Balanced MVC 아키텍처를 제안한다. 5.1에서 설명한 Client-Server 아키텍처는 자원의 최소화로 클라이언트에 모델 계층이 없는 반면, Balanced MVC는 작은 크기의 데이터를 클라이언트에서도 저장하여 사용할 수 있도록 설계하여 클라이언트와 서버 양측에 MVC 구조를 적용한 아키텍처 패턴이다. Balanced MVC 아키텍처에서 허용하는 클라이언트가 가질 수 있는 모델 계층, 즉 클라이언트용 데이터베이스는 자주 쓰이는 정보나 개인적인 정보를 사용자의 디바이스에 직접 저장 관리할 수 있으므로, 성능이나 보안의 관점에서 Client-Server 아키텍처가 제공하지 못하는 아키텍처 관점을 가지고 있다.

구조적 뷰: Balanced MVC 아키텍처는 (그림 4)와 같이 모바일 디바이스 측의 클라이언트 어플리케이션과 서버 시스템에 각각 분리된 MVC 형태를 보이고 있다. Balanced MVC 아키텍처 패턴은 설계 시점부터 모바일 어플리케이션에도 데이터 저장소를 두어 모바일 디바이스와 서버에 각각 데이터를 저장한다는 차이점이 있다. 모델, 뷰, 컨트롤 계층에 속한 각 모듈은 기존 MVC의 모듈 기능과 동일하다.

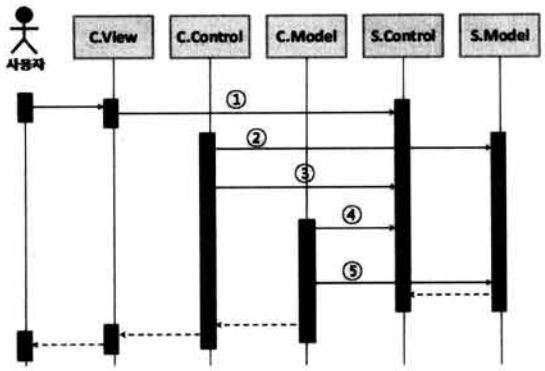
Balanced MVC 아키텍처를 설계하는데 주요 결정사항은 클라이언트 어플리케이션과 서버 시스템 부분에 컨트롤과 모델 계층을 분리하는 방법에 있다. 즉, 기능성과 데이터들을 최적화하여 분리해야 한다는 점이다. 이를 위해, Balanced MVC 아키텍처를 (그림 4)와 같이 두 가지 유형으로 분리한다. 두 가지 유형을 결정하는 주요 요소는 컨트롤 계층을 *C.Control*과 *S.Control*로 분리하는 방법이다.

유형 1에는 *C.Control*이 존재하고 *S.Control*은 존재하지 않는다. 이는 *C.Control*이 *S.Model*의 일부분과 밀접하게 상호작용을 하므로 *S.Model*의 데이터를 *C.Model*로 복제해야



(그림 4) 유형별 Balanced MVC 아키텍처의 2가지 유형

하는 경우에 적용될 수 있다. 즉, *C.Model*과 *S.Model* 간의 높은 병렬처리 효과를 기대할 수 있다. 유형 2에는 *C.Control*과 *S.Control*이 존재한다. 이는 의존도가 높은 *Control*과 *Model*을 각각 클라이언트와 서버 시스템에 위치시켜야 하는 경우에 적합하다.



(그림 5) Balanced MVC 아키텍처 패턴의 상호작용 경로

동적 뷰: Balanced MVC 아키텍처에서는 (그림 5)와 같이 유형 1과 유형 2를 위한 5가지의 상호작용 경로가 존재한다. 경로는 어플리케이션의 특징에 맞게 설정한다. (그림 5)에서 보이는 5가지 경로 중 경로 1, 경로 2, 경로 3은 Client-Server 아키텍처에서도 언급된 내용이므로 경로 4와 경로 5에 대해서 설명한다. 동적 뷰는 클라이언트와 네트워크가 어떠한 경로로 데이터를 전송하는지에 초점을 두고 있다.

경로 4는 *C.Model*과 *S.Control* 간의 상호작용을 나타낸다. 클라이언트에 저장된 데이터가 서버 측 비즈니스 로직에서 필요할 때 이 경로를 사용할 수 있다. 예를 들어, 사용자의 개인 정보는 서버에 있는 모델에 저장하는 것보다는 클라이언트의 데이터베이스에 저장하는 것이 보안 측면에서 좋다. 이 경우에 데이터가 필요할 때에만 서버 측에 데이터를 전송하게 하는 방식을 쓰게 하는 어플리케이션은 경로 4를 적용 할 수 있다.

경로 5는 *C.Model*과 *S.Model* 간의 상호작용을 나타낸다. *C.Model*과 *S.Model*과의 상호작용은 비즈니스 로직이 필요하지 않은 상태에서 클라이언트에 저장된 데이터를 서버에 있는 데이터에 저장해야 할 필요가 있는 어플리케이션을 만들 때 필요하다. Google 주소록을 예로 들 수 있다. 사용자가 안드로이드폰에 전화번호를 저장하면, 그 번호는 Google 주소록에 자동으로 저장하게 되는데, 이런 유사한 기능이 필요한 어플리케이션에 경로 5를 적용할 수 있다.

적용 지침: Balanced MVC 아키텍처는 클라이언트 어플리케이션과 서버 시스템이 다른 위치에 존재하기 때문에, 네트워크 오버헤드가 항상 발생하며 이는 전체 성능에 악영향을 미칠 수 있다. Balanced MVC 아키텍처의 다양한 상호작용에 따라 네트워크 통신 횟수가 다르므로, 적절한 상호작용 경로를 선택하여, 성능 저하를 최소화시키도록 해야 한다. 효과적으로 Balanced MVC 아키텍처가 설계되기 위해서는, [12]에서 제시한 것처럼 요구사항 명세서와 유즈케이스 모델은 필수적으로 작성되어야 하며, 개념적 객체 모델이 선택적이다. 기존의 산출물을 이용하여 Balanced MVC 아키텍처를 설계하기 위해 3단계 스텝의 설계 지침을 제안한다.

스텝 1. 적절한 패턴 유형 선택
스텝 2. 각 계층에 포함되는 컴포넌트를 유도하여 선택한 패턴 유형 구체화
스텝 3. 각 컴포넌트의 인터페이스 정의

스텝 1: 개발하고자 하는 대상 어플리케이션에 적절한 패턴 유형을 선택하기 위해서는 요구사항 명세서, 객체 모델이 필요하며, 클라이언트 어플리케이션과 서버 시스템 간의 통신 비용(Communication Cost)을 줄이는 것을 주 목적으로 한다.

유형 1, 유형 2를 분별하는 가장 큰 차이점이 *S.Control*의 유/무이므로, 서버 측을 복잡한 기능을 수행하는 역할인지, 단지 데이터 저장하는 역할인지에 따라서 유형 선택 프로세스를 시작한다. 계층 간의 의존성은 *Depends(A, B)*라는 함수를 이용하여 결정할 수 있으며, 이 함수는 A와 B 간의 예측 가능한 의존성 정도를 반환한다. 이 값은 유즈케이스 명세서에 기술된 액터, 클라이언트 시스템, 서버 시스템, 외부 서비스 간의 메시지 교환 횟수 또는 데이터 교환 횟수를 참고하여 결정된다. 이에 대한 상세한 프로세스는[12]에 기술되어 있다.

스텝 2: 이전 산출물인 유즈케이스 모델을 이용하여 각 계층에 포함되는 컴포넌트를 도출한다. *C.View*에 포함되는 컴포넌트는 유즈케이스 명세서에 기술된 액터와 클라이언트 시스템 간의 상호작용을 참조하여 도출하도록 한다. *C.View*에 속하는 컴포넌트들의 상세한 구조는 추후 사용자 인터페이스 설계 과정에서 구체적으로 결정된다. *C.Model*과 *S.Model*에 포함되는 컴포넌트는 유즈케이스 모델에서 데이터 저장을 목적으로 하는 시스템 기능들을 분석하여, 유즈케이스 명세서 중 서버 시스템에서 작동하는 기능 중 데이터 저장을 목적으로 하는 액션(action)을 기반으로 도출한다. *C.Control*과 *S.Control*에 포함되는 컴포넌트는 유즈케이스 모델에 명시된 기능 그룹을 이용하여 도출한다. 즉, 유즈케이스 모델에 포함된 모든 기능은 목표 시스템에서 제공하는 기능을 유즈케이스로 나열하고 있으며, 각 유즈케이스들은 서브 시스템 별로 그룹화되어 관리된다. 예를 들어, 회원 관리 서브시스템에는 회원 가입, 회원 정보 수정, 회원 정보 삭제, 회원 정보 검색의 유즈케이스가 포함되며, 이를 위한 *C.Control*과 *S.Control*에는 회원 관리 컴포넌트를 도출할 수 있게 된다.

스텝 3: 이 단계에서는 유즈케이스 명세서를 기반으로 *C.Control*과 *S.Control*에 속한 컴포넌트들의 인터페이스를 정의한다. *C.Contrl*과 *S.Control*에 포함된 클래스들과 매핑되는 기능 그룹을 먼저 결정한다. 그리고, 기능 그룹에 포함되는 유즈케이스들을 확인한다. 유즈케이스 명세서의 액터와 클라이언트 시스템 간의 상호작용, 클라이언트 시스템과 서버 시스템 간의 상호작용을 분석함으로써, *C.Control*과 *S.Control*의 컴포넌트에 대한 인터페이스와 주요 오퍼레이션을 정의할 수 있다.

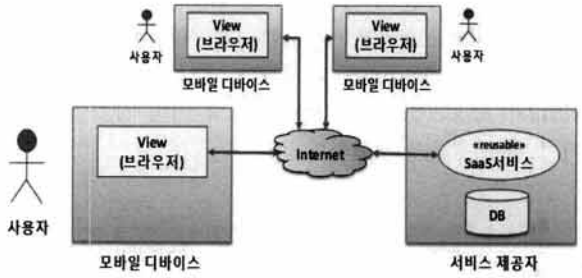
5.3 서비스 기반 아키텍처 패턴

개요: 서비스 기반 아키텍처는 한 어플리케이션이 재사용 가능한 서비스 기능을 구독 및 호출함으로써 어플리케이션의 기능을 수행하는 방식의 아키텍처 스타일이다[13]. 서비스는 내부 구현을 외부로 노출하지 않고, 서비스 사용자가 네트워크를 통한 인터페이스만을 이용하여 접근할 수 있는 느슨한 결합방식을 사용한 기능 단위이다. 서비스 제공자는 여러 사용자가 공통으로 사용할 수 있는 기능을 제공하고, 서비스 사용자는 일반 PC 또는 모바일 디바이스에 상관없이 기능을 호출할 수 있다. 그리고 느슨한 결합방식을 기반으로 하고 있으므로, 서비스 사용자는 하나의 인터페이스를 통하여 동일한 인터페이스를 가진 여러 서비스를 실시간에 동적으로 호출할 수 있다. 그리고 필요로 하는 기능성이 서비스로 제공되어 개발 비용의 절감 및 시기 적절성(Time-to-Market)의 이점을 이용할 수 있다.

구조적 뷰: 서비스 기반 아키텍처는 모바일 디바이스 측의 클라이언트 어플리케이션과 서비스로 각각 분리된 형태를 보이고 있다. 클라이언트는 모바일 디바이스에 위치하며 잘 정의된 인터페이스를 통해 서비스를 호출한다. 서비스는 클라이언트가 요청하는 작업을 수행하고 결과를 클라이언트에 보낸다.

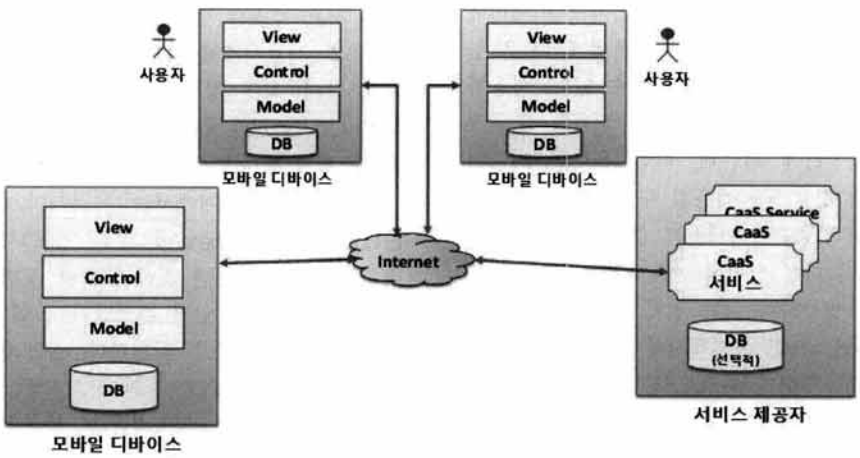
서비스 기반 아키텍처는 잠재적인 사용자를 위해 모바일 어플리케이션에서 제공해야 하는 모든 기능을 제공해주는 SaaS (Software as a Service)와 모바일 어플리케이션의 일부 기능을 단위 서비스로 CaaS(Component as a Service) 종류를 고려하여, 두 가지로 서비스 기반 아키텍처 패턴을 제안한다.

SaaS 기반 아키텍처는 모바일 어플리케이션에서 필요로 하는 모든 기능을 SaaS를 통해서 만족하므로, (그림 6)과 같이 뷰와 관련된 모듈, 특히 웹 브라우저만 모바일 디바이스에 위치하며, 나머지 컨트롤과 모델 계층에 속한 기능은 SaaS 서비스 형태로 서비스 제공자의 서버에 배포시키는 구조를 가진다. SaaS는 특정 도메인에서 재사용 가능한 전체 소프트웨어 기능을 제공할 수 있는 범용적이며, 재사용 가능한 서비스를 일컫는다. SaaS는 모바일 디바이스에서 서비스를 테일러링 할 수 없는 구조이므로 서비스 제공자는 SaaS 서비스를 모바일 환경에 맞게 테일러링을 하여 제공하여야 한다.



(그림 6) SaaS 기반 아키텍처의 구조적 뷰

CaaS 기반 아키텍처는 모바일 어플리케이션에서 필요로 하는 일부 기능을 위해 CaaS를 사용하므로, (그림 7)과 같이 모바일 디바이스에 뷰 계층, 컨트롤 계층, 모델 계층을 가지는 클라이언트 어플리케이션이 존재하고, 클라이언트 어플리케이션에서는 필요에 따라 서비스 제공자에 의해 제공되는 CaaS 를 구독하는 구조를 가진다. 클라이언트 어플리케이션에서는 하나 이상의 CaaS를 조합하고 모바일 어플리케이션에 국한된 기능은 추가로 구현함으로써, 사용자에게 기능을 제공하는 것이다. CaaS 서비스는 서비스의 성격과 어플리케이션의 목적에 따라 세 가지 형태가 가능하다. 즉, CaaS 서비스는 모바일 디바이스의 뷰 계층, 컨트롤 계층, 모델 계층에서 각각 호출할 수 있다. 뷰 계층에서 서비스를 호출하는 경우는 제공되는 CaaS 서비스의 데이터가 바로 사용자에게 보여주기를 원하는 상황에 사용한다. 컨트롤 계층에서 서비스를 호출하는 경우는 모바일 디바이스로 비즈니스 프로세스 혹은 비즈니스 프로세스의 일부를 동작시킬 때 사용한다. 컨트롤 계층에서는 뷰 계층에서 사용자의 명령을 받아 처리하고 결과를 모델 계층을 통해 데이터베이스에 저장한다. 모델 계층에서 서비스를 호출하는 경우는 사용자의 정책에 따라 CaaS 서비스에서 제공되는 데이터 일부를 데이터베이스로 저장하는 경우 사용한다.



(그림 7) CaaS 기반 아키텍처의 구조적 뷰

서비스 기반 아키텍처는 독립형 아키텍처 패턴을 제외한 나머지 아키텍처와 마찬가지로, 제한된 자원을 가지는 모바일 디바이스에서 복잡한 기능의 어플리케이션을 사용할 수 있으며, 서비스를 재사용함으로써 빠른 시기 적절성을 보장할 수 있고, 개발 비용이 적게 들며, 유지보수의 어려움도 없는 장점이 있다. 그러나 기능 중 일부는 서비스 측에서 실행되기 때문에, 네트워크 안정성이 보장되어야만 사용자가 기능 호출에 대한 응답을 받을 수 있다. 그리고 제3자에서 개발된 서비스를 사용하는 것이기 때문에, 품질 관리에 어려움이 있고 서비스 구독 비용이 필요하게 된다. 또한, 모든 기능 호출은 반드시 네트워크 통신을 필요로 하므로 독립형 모바일 어플리케이션만큼 빠른 시간 내에 응답을 받지는 못한다.

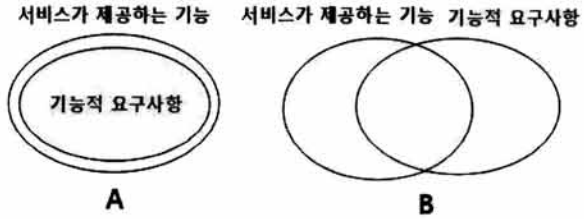
동적 뷰: SaaS 기반 아키텍처에서 사용자는 모바일 디바이스의 뷰 계층에서 기능을 호출한다. 이는 인터넷을 통해 SaaS 서비스에 전달된다. SaaS 서비스는 사용자가 요청한 기능을 수행하며 필요한 경우 SaaS 서비스가 가지고 있는 데이터베이스를 사용한다. 그리고 SaaS 서비스는 수행한 결과를 모바일 디바이스에 넘겨준다. 결과 값은 일반적으로 뷰 계층의 브라우저에 출력된다.

CaaS 기반 아키텍처는 모바일 디바이스의 컨트롤 계층에서 전체 비즈니스 프로세스 로직을 관리하며, 일부 기능을 서비스 제공자로부터 제공되는 CaaS의 기능을 호출함으로써 수행한다. 그리고 모든 정보는 모바일 디바이스의 데이터베이스에 저장하게 된다.

적용 지침: 유즈케이스 모델과 같은 기존의 산출물을 이용하여 서비스 기반 아키텍처를 설계하기 위해, 4단계 스텝의 설계 지침을 제안한다.

스텝 1. 기능적 요구사항으로 이용 가능한 서비스 검색
스텝 2. 사용할 서비스를 결정하고 이를 서비스 기반 아키텍처에 반영
스텝 3. 결정된 아키텍처를 기반으로 각 컴포넌트 설계
스텝 4. 각 컴포넌트의 주요 인터페이스 정의

스텝 1: 요구사항 명세서에 기술된 기능적 요구사항을 기반으로 이용 가능한 서비스 검색한다. (그림 8)은 기능적 요구사항과 서비스가 제공하는 기능 간의 관계를 보여준다. A는 서비스가 제공하는 기능이 기능적 요구사항을 모두 지원하는 경우이고 B는 일부의 기능적 요구사항을 서비스에서 제공하는 경우이다. 만약 어떠한 기능적 요구사항도 서비스가 제공하는 기능이 없는 경우에는 서비스 기반 아키텍처 패턴으로 설계하지 않는다.



(그림 8) 기능적 요구사항과 서비스가 제공하는 기능과의 관계

스텝 2: 사용할 서비스를 결정하고 이를 서비스 기반 아키텍처에 반영한다. 서비스는 A의 경우에 SaaS 기반 아키텍처 혹은 CaaS 기반 아키텍처로의 설계가 모두 가능하다. 아키텍처 드라이버의 중요도에 따라 다음과 같이 선택할 수 있다. 자원 소비를 최소화할 경우에는 SaaS 기반 아키텍처를 선택하는 것이 좋다. 앞에서 설명한 대로 SaaS 기반 아키텍처는 모바일 디바이스에 뷰 계층만 존재하면 되기 때문에 자원 소비를 최소화할 수 있다. 높은 성능을 원할 때에는 기능적 요구사항을 분류하고 잦은 호출 및 데이터 접근이 많은 기능은 디바이스에 배치하는 것이 좋다. 신뢰성이 중요한 기능적 요구사항은 디바이스 측의 배치를 고려한다. 복잡도가 높은 기능성이 요구사항에 있는 경우 SaaS 기반 아키텍처로 구현하거나 해당 요구사항을 CaaS 서비스로 이용한다.

B는 CaaS 기반 아키텍처로 설계가 가능하다. 기능적 요구사항은 CaaS 기반 아키텍처에서는 디바이스와 CaaS로 나누어진다. 자원소비가 많은 기능에 대해서는 디바이스에 배치하지 않는 것이 좋다. 성능의 최적화 측면에서는 디바이스에서 직접 서비스를 호출하는 것이 좋으나 이는 CaaS의 신뢰도가 떨어지는 경우 전체적인 어플리케이션의 신뢰도가 하락한다. CaaS 서비스가 항상 요구사항과 정확히 일치하는 것은 아니다. CaaS 서비스가 요구하는 기능성과 거의 일치하지만, 정확히 매칭되지 않는 것을 부분 매칭이라 한다. 이는 다양한 형태의 어댑터를 통해 해결 가능하다[14]. 예를 들면, CaaS로 원하는 대부분 만족하게 하지만 완벽하게 만족하지 하지 못하는 경우 Decorator 패턴을 사용하여 서비스 클라이언트에서 필요한 기능을 추가한다. 따라서, 클라이언트 측에는 가상 서비스가 운영되고 있는 것처럼 보이며, 가상 서비스에는 추가된 기능성을 위한 새로운 인터페이스를 정의하고 이를 아키텍처에 반영한다.

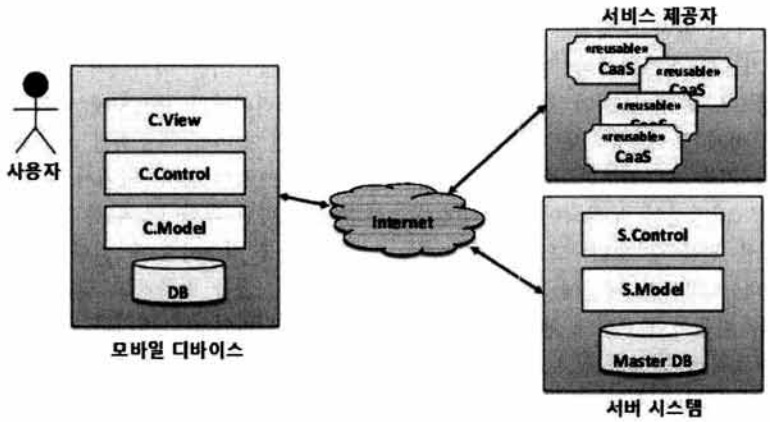
스텝 3: 결정된 아키텍처를 기반으로 각 컴포넌트를 설계한다. 유즈케이스 모델과 객체 모델을 이용하여 기능성 컴포넌트를 도출한다. 신뢰성을 높이기 위해서는 오프라인 모드 컴포넌트를 도출한다. 두 가지 경우의 아키텍처에 대해 다음과 같이 계층을 나누고 도출한 컴포넌트를 배치한다.

- SaaS 기반 아키텍처의 경우: 디바이스에 웹 브라우저만 존재한다.
- CaaS 기반 아키텍처의 경우: 디바이스의 뷰 계층, 컨트롤 계층, 모델 계층이 있고 서비스로 도출된 CaaS 가 있다. 그러므로, 뷰, 컨트롤, 모델 계층에 속하는 컴포넌트를 도출한다.

스텝 4: 이 단계에서는 유즈케이스 명세서를 기반으로 각 컴포넌트들의 인터페이스를 정의한다. SaaS는 클라이언트 어플리케이션에 뷰 계층만 존재하므로, CaaS를 적용한 어플리케이션에 대해서만 이 스텝을 수행한다. 인터페이스를 도출하는 방법은 앞서 설명한 방법과 동일하다.

5.4 하이브리드 아키텍처 패턴

개요: 지금까지 알아본 Client-Service 아키텍처 패턴, Balanced MVC 아키텍처 패턴, 서비스 기반 아키텍처 패턴은 각각 장/단점을 가지고 있다. Client-Server 아키텍처 패



(그림 9) Balanced MVC, 서비스 기반 아키텍처가 융합된 하이브리드 아키텍처 패턴의 구조적 뷰

턴은 서버 쪽에만 데이터베이스를 위치시킴으로써 적은 양의 데이터라도 반드시 서버 측에 요청해야 하는 오버헤드가 있으며, Balanced MVC 아키텍처 패턴에서는 재사용 가능한 서비스를 고려하지 않았으며, 서비스 기반 아키텍처 패턴은 재사용 가능한 서비스가 배포되어 있는 상태에서만 적용할 수 있다는 단점이 있다. 이들의 단점을 보완하고, 장점을 극대화하기 위해서 이 세 가지 패턴을 다양하게 조합한 하이브리드 아키텍처 패턴을 제안한다.

구조적 뷰: 하이브리드 아키텍처는 독립형 아키텍처 패턴을 제외한 모든 아키텍처 패턴들이 서로 다양하게 조합을 이루는 형태이다. 그 중 하나의 예로, (그림 9)는 Balanced MVC, 서비스 기반 아키텍처 패턴이 조합되어 일부 기능은 서버 시스템에서 실행되고 다른 일부 기능은 CaaS 서비스로 실행되는 구조를 보여준다. 즉, Balanced MVC 아키텍처를 적용하여 비교적 적은 자원을 필요로 하는 간단한 기능은 클라이언트 측에서 실행되며, 복잡한 계산 및 데이터 조작을 요구하는 기능은 서버 측에서 실행된다. 서비스 기반 아키텍처 패턴을 적용하여 일부 기능은 공통적이고 재사용 가능한 서비스 형태로 제공되어 모바일 디바이스의 클라이언트 어플리케이션에서 호출한다.

하이브리드 아키텍처는 앞서 언급된 각 아키텍처 패턴의 장점을 모두 활용할 수 있지만, 아키텍처 구조가 다소 복잡하여 적용하기 어려운 단점을 가지고 있다.

동적 뷰: 하이브리드 아키텍처의 동적 뷰 역시 융합된 아키텍처 종류에 따라 다양한 형태를 보인다. Balanced MVC와 서비스 기반 아키텍처 패턴을 융합한 하이브리드 아키텍처 패턴의 경우, 모바일 디바이스의 컨트롤 계층에서 전체 비즈니스 프로세스 로직을 관리하며 일부 기능을 서버 시스템의 *S.Control* 또는 서비스 제공자로부터 제공되는 CaaS의 기능을 호출함으로써 수행한다. 그리고 사용자에 특화된 정보는 모바일 디바이스의 데이터베이스에 저장하며 여러 사용자 간의 상호작용에 관련된 정보는 서버 시스템의 데이터베이스에 저장되도록 프로세스가 진행된다. 주목할 점은 하이브리드 아키텍처는 모바일 어플리케이션의 전체 기능을 외부에 위치시키는 SaaS 기반 서비스 아키텍처 패턴이 포함되지 않는다. 그리고 클라이언트 측 데이터베이스 위치 여부가 확연히 다른 Client-Server 아키텍처와 Balanced MVC 아키텍처는 같이 조합되지 않는다.

적용 지침: 하이브리드 아키텍처 패턴의 설계 시에는 먼저 융합될 아키텍처를 선정하는 것이 중요하다. 아키텍처가 선정된 후에는 *C.Control, S.Control, C.Model, S.Model,* CaaS를 적절히 선택하여 설계해야 한다. 각 계층에 기능을 분할할 때에는 자원 활용의 최소화, 성능 최대화를 중점을 둔다. 즉, 자원 활용을 최소화하는 동시에 성능을 최대화시킬 수 있도록 적절히 기능을 분할해야 한다. 그러므로 다음과 같은 3가지 스텝을 거쳐 하이브리드 아키텍처 패턴을 설계한다.

스텝 1. 융합될 아키텍처 선정

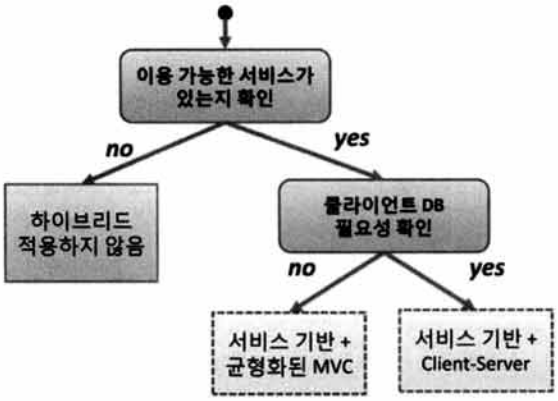
스텝 2. (서비스 기반 아키텍처 패턴 적용 시) 가장 최적의 CaaS 선정 및 컴포넌트 설계

스텝 3. (Balanced MVC 아키텍처 패턴 적용 시) CaaS에 의해서 제공되는 기능을 제외하고 적절한 Balanced MVC 아키텍처 유형 선택 후, 기능 분할 설계

스텝 4. (Client-Server 아키텍처 패턴 적용 시) CaaS에 의해서 제공되는 기능을 제외하고 적절한 Client-Server아키텍처 유형 선택 후 기능 분할 설계

스텝 1: 요구사항을 분석하여 Client-Server, Balanced MVC, 서비스 기반 아키텍처 중 하이브리드 아키텍처로 융합될 아키텍처를 선정한다. (그림 10)은 하이브리드 아키텍처 결정 프로세스를 보여준다.

하이브리드 아키텍처 선정에 먼저 분석된 요구사항을 분석하여 이용 가능한 서비스가 있는지를 확인한다. 즉, 유즈케이스 모델에 명세 된 기능 개요, 사전 조건, 사후 조건, 입력 데이터, 출력 값 등의 정보를 이용하여 동일하거나 유사한 기능을 제공하는 서비스가 있는지를 확인한다. 유사한 기능을 제공하는 서비스만 발견되면, 비즈니스 분석을 통해 서비스를 이용한 어플리케이션 개발 비용과 서비스를 이용하지 않은 어플리케이션 개발 비용을 비교하여 서비스 이용 유/무를 결정한다.



(그림 10) 하이브리드 아키텍처 패턴을 위한 아키텍처 선정 프로세스

둘째, 서비스로 제공되는 기능 외에 대하여 기능복잡도 등을 고려하여 모바일 디바이스에 별도의 클라이언트 데이터베이스가 관리되어야 하는지를 결정한다. 즉, 클라이언트 어플리케이션에 별도의 데이터베이스를 관리하지 않으면, Balanced MVC 아키텍처를 적용하지 않고 Client-Server 아키텍처를 적용한다.

스텝 2 ~ 스텝 4: 각 아키텍처 패턴 설계 지침을 따라 수행한다.

6. 사례 연구

본 장에서는 아키텍처 패턴의 적용 가능성을 보이기 위하여, Client - Server, Balanced MVC, 서비스 기반 아키텍처 패턴, 하이브리드 아키텍처 패턴에 대한 사례 연구를 수행한다.

6.1 Client-Server 아키텍처 패턴의 적용

Publication Manager는 논문 출판과 관련된 정보 관리를 위한 안드로이드 기반의 어플리케이션이다. 이 어플리케이션은 컨퍼런스나 저널의 정보를 관리하는 기능과 연구 논문을 출판하기까지 학생의 라이프 사이클을 관리하는 기능을 제공한다. Publication Manager는 클라이언트 측과 서버 측의 기능 분할에 대한 요구사항이 포함되어 컨퍼런스나 저널 정보, 논문별 현재 진행 상태 정보가 서버 측에 관리되어야 하므로, Client-Server 아키텍처 패턴이 적절한 패턴으로 선정되었다.

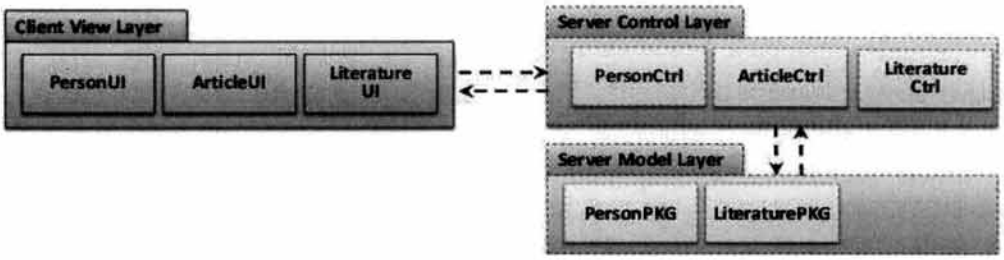
〈표 1〉 Publication Manage 어플리케이션의 기능 복잡도를 위한 가중치

요인	설명	값	가중치
데이터 타입	단순 데이터	0.05	0.2
	복합 데이터	0.2	
데이터 개수	3개 미만	0.05	0.2
	3개에서 7개	0.15	
	7개 이상	0.2	
알고리즘 복잡도	$f(x) \in \{O(1), O(\log_2 n)\}$	0.1	0.6
	$f(x) \in \{O(n), O(n\log_2 n), O(n^2)\}$	0.3	
	$f(x) \in \{O(n^m), O(2^n)\}$	0.6	

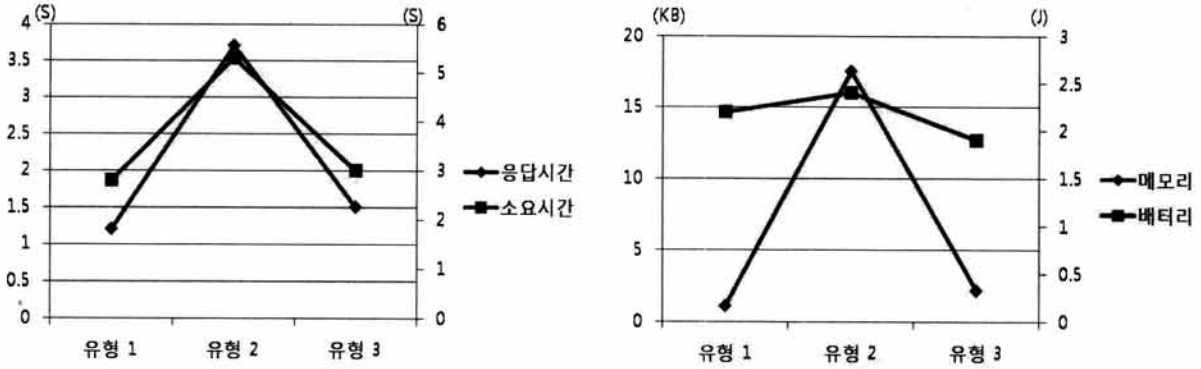
적용 지침에 따라서 먼저, 어플리케이션의 아키텍처 패턴 유형을 결정한다. 기능 복잡도를 구하기 위해서 기능 점수 기법을 확장하여, <표 1>과 같이 알고리즘 복잡도, 데이터 개수, 데이터 타입 순으로 각 요인에 대한 가중치를 부여하였다.

유즈케이스로 나온 회원 정보 보기 기능과 출판 기록 보기 기능에 대해 유즈케이스 명세서를 통해 각각의 기능 복잡도를 구한 결과는 다음과 같다. 회원 정보보기 기능은 회원이름, 최종 학위, 최종 학위 연도, 컨퍼런스 논문 수, 저널 논문 수, 이메일의 6개의 데이터가 있고 각각은 단순 데이터 타입이다. 반복문에 대한 내용이 없으므로 알고리즘 복잡도는 O(1)이다. 출판 기록보기 기능은 컨퍼런스 이름, 개최일, 출판 상태, 논문 제목, 공동 저자 등의 컨퍼런스 정보 집합과 저널 이름, 출판사, 논문 제목, 공동 저자 등의 저널 정보 집합으로 복합데이터가 필요하고, 기록에 대한 정보는 증가하는 특성이 있으므로, 데이터 개수를 무한으로 정한다. 하나 이상의 논문을 정렬해서 보여주어야 하므로 기본 알고리즘 복잡도는 $O(n^2)$이다. 기능 복잡도의 수가 작다면 유형 1에, 크다면 유형 2가 적합하고, 유형 3으로 기능이 수행되는 위치를 분리할 수 있다. Publication Manage 어플리케이션은 출판 기록보기 기능과 같은 복잡한 기능이 다수 있기 때문에, 유형 1과 유형 3이 후보가 되었다.

Publication Manager 어플리케이션은 정보를 입력하거나, 목록을 출력하는 생성, 업데이트, 수정, 삭제 기능이 많다. 정보 입력에서는 V-C-M 모두 같은 양의 데이터가 이동하여 고려 대상이 아니지만, 입력의 수가 빈번할수록, 데이터베이스에 저장되는 데이터가 출력되는 양은 늘어난다. 이



(그림 11) Client-Server 아키텍처 패턴 적용 사례



(그림 12) 유형별 Publication Manager의 효율성 비교

경우에는 데이터베이스를 관리하는 모델 계층과 비즈니스 로직을 담당하는 컨트롤 계층 간의 데이터 이동이 많아지게 되어, 유형 1이 유력한 후보가 되었다. 마지막으로 상호작용의 빈도는 특정 부분에서 높은 빈도가 나타나지 않아 유형 선택에 거의 영향을 미치지 않았다. 따라서 Publication Manager 어플리케이션의 아키텍처 유형 후보는 유형 1과 유형 3이다. 이 중 데이터 이동량을 고려했을 때, 가장 적합한 유형 1이 되고, (그림 11)과 같이 아키텍처를 설계하였다.

유형 1을 적용하여 설계된 아키텍처가 해당 요구사항에 가장 적합한 것인지를 평가하기 위하여, 모든 3가지 유형을 반영하여 Publication Manger를 실제 구현한 결과를 바탕으로 ISO/IEC 9126[15]의 효율성(Efficiency) 품질 속성을 측정하였다. Client-Server 아키텍처 패턴은 모바일 디바이스의 제한된 자원으로 복잡한 어플리케이션을 효과적으로 수행하기 위하여 제시된 것이므로, 효율적으로 어플리케이션이 실행되었는지를 평가하기 위하여 효율성을 사용한 것이다. (그림 12)와 같이 효율성을 평가하기 위하여 응답시간, 소요시간(Turnaround Time), 메모리 사용량, 배터리 사용량을 측정하였고, 유형 1이 가장 적은 값이 나옴을 확인할 수 있었다.

본 사례 연구를 통하여 Client - Server 아키텍처를 분석하면 다음과 같은 결과를 도출할 수 있다.

• Client - Server 아키텍처 구조는 Client 내에 저장 공간이 없이 사용자와의 UI나 간단한 기능을 수행하는 정도의 역할로 한정되기 때문에, 서버에 있는 데이터베이스에는 모든 정보의 열람이 허용 가능한 어플리케이션에 적합하다는 것을 알 수 있다.

• Client - Server 아키텍처 구조에서는 가장 적은 네트워크 사용량을 고려해서 세가지 유형 중에 선택할 수 있다. 네트워크 사용량에 따라 자원 소비량에 영향을 미친다.

6.2 Balanced MVC 아키텍처 패턴의 적용

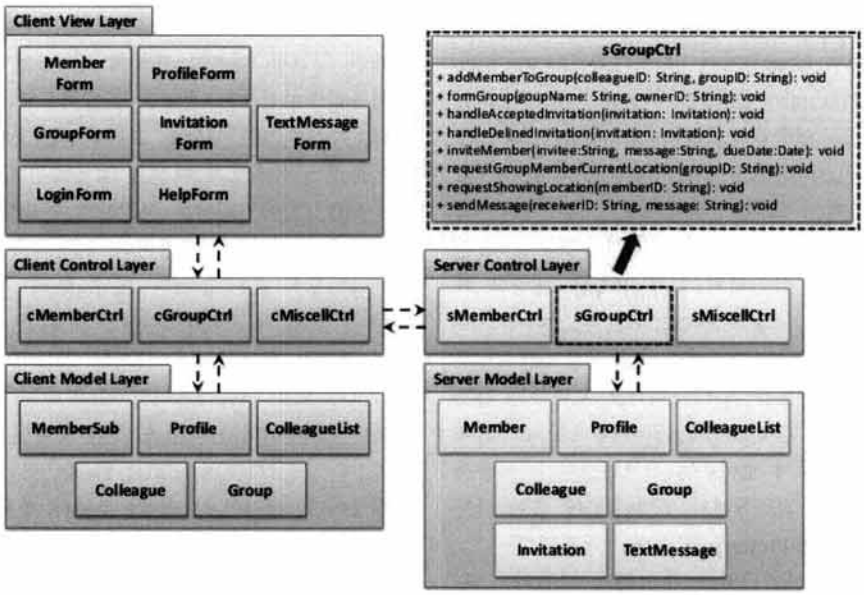
Mobile Mate Service(MMS)는 모바일 디바이스을 보유한 사용자의 위치정보를 이용하여 소셜 네트워킹 기능을 지원하는 어플리케이션으로, 회원 관리, 동료 관리, 동료 초대, 초대 수락 및 거절, 동료 위치 확인 등의 기능을 제공한다. MMS 요구사항 명세서에 클라이언트 측과 서버 측의 기능 분할에 대한 요구사항이 포함되어 있으므로, 객체 모델링 시에 클라이언트 측을 위한 객체 모델과 서버 측을 위한 객체 모델을 별도로 작성하는 Balanced MVC 아키텍처 패턴을 적용한다. Balanced MVC 아키텍처를 설계하기 위해서는 먼저 패턴 유형을 선택한다.

유즈케이스 명세서 상의 기술된 클라이언트 시스템과 서버 시스템 간의 메시지 상호작용과 데이터 상호전달 횟수를 분석하여, *C.Control*와 *S.Model*간의 의존도가 높은 것으로 판단되었다. 그러므로, *C.Control*과 *S.Model*이 직접적으로 상호작용하는 횟수를 줄이기 위하여, *C.Control*과 의존도가 낮은 부분을 *S.Control*로 분리하고, *S.Control*이 *S.Model*과 주로 상호작용을 하도록 유형 2를 MMS의 Balanced MVC 아키텍처로 선정하였다. (그림 13)은 유형 2를 적용하여 설계된 MMS의 Balanced MVC 아키텍처이다.

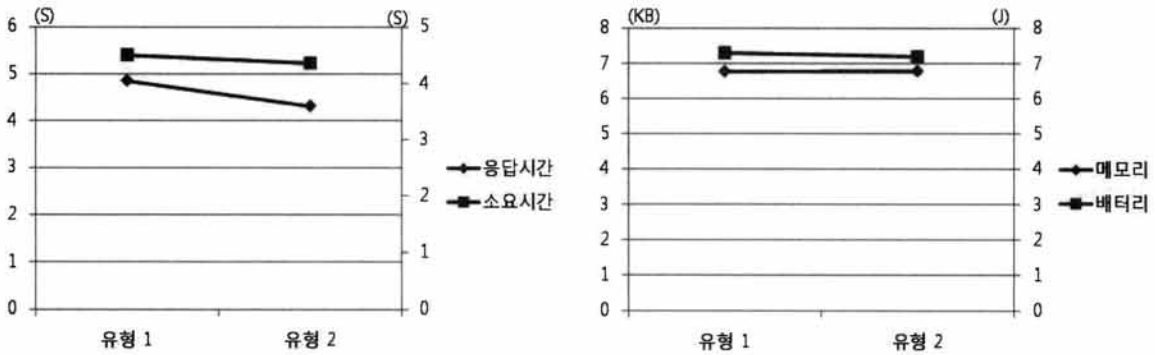
MMS에 적절한 경로를 선택한 후에, *C.Control*과 *S.Control*에 속할 클래스를 도출한다. 유즈케이스 다이어그램 작성 시에 4개의 기능 그룹을 이용하였지만, 그 그룹 중 멤버 관리 그룹과 프로파일 관리 그룹은 상당히 밀접한 관계가 있기 때문에 하나의 그룹으로 재정의하였다. 그리하여, 멤버 관리를 위한 컨트롤러 클래스*(cMemberCtrl과 sMemberCtrl)*, 그룹 관리를 위한 컨트롤러 클래스*(cGroupCtrl과 sGroupCtrl)*, 기타 기능들을 위한 컨트롤러 클래스*(cMiscellCtrl과 sMiscellCtrl)*를 클라이언트 측과 서버 측에 각각 배치하였다. 마지막으로, 각 컨트롤러 클래스에 포함되는 오퍼레이션을 유즈케이스 명세서를 이용하여 도출한다. (그림 13)을 보면, *sGroupCtrl*을 위한 오퍼레이션 목록들이 나열되어 있으며, 이는 *sGroupCtrl*과 관련된 그룹 관리 유즈케이스에 대한 명세서를 분석하여 도출하였다.

Client-Server 아키텍처 적용 사례와 유사하게, 유형 2를 적용하여 설계된 아키텍처가 해당 요구사항에 가장 적합한 것인지를 평가하기 위하여, 모든 2가지 유형을 반영하여 MMS를 실제 구현한 결과를 바탕으로 효율성을 측정하였다. (그림 14)는 응답시간, 소요시간, 메모리 사용량, 배터리 사용량을 측정한 결과이며, 유형 2가 유형 1보다 좋은 효율성을 보임을 확인할 수 있다.

본 사례 연구를 통하여 Balanced MVC 아키텍처를 분석하면 다음과 같은 결과를 도출할 수 있다.



(그림 13) Balanced MVC 아키텍처 적용 결과



(그림 14) 유형별 MMS의 효율성 비교

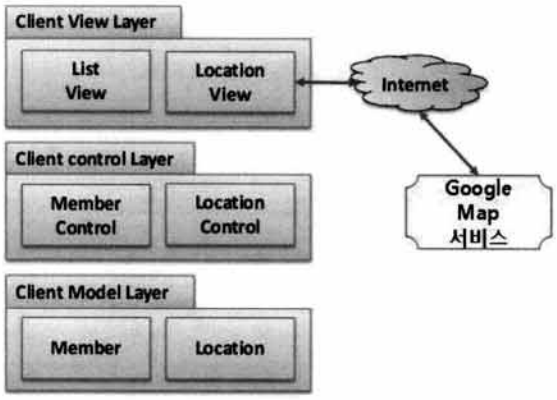
• 서버 측에는 모바일 디바이스를 통해 MMS에 접근하는 모든 사용자의 정보가 저장되어 있으며, 다수 사용자 간의 소셜 네트워킹을 위한 기능을 제공한다. 그리고 여러 사용자의 위치 정보를 검색하는 등의 다소 복잡한 계산을 해야 하는 기능을 수행한다. 이 기능들은 *sGroupCtrl*의 오퍼레이션으로 설계된다.

• 모바일 디바이스에 배포된 어플리케이션에는 특정 개인과 관련된 정보와 간단한 계산을 해야 하는 기능만 가지고 있어 많은 자원을 필요로 하지 않는다. 그러므로 모바일 디바이스 상에서 복잡한 기능을 많은 자원을 소모하지 않고 사용할 수 있게 된다.

• 클라이언트 측과 서버 측의 통신 비용을 절감하기 위하여, *C.Control*의 클래스가 *S.Model*에 속한 Profile 클래스에 직접 접근하는 것을 허용하였다.

6.3 서비스 기반 아키텍처 패턴의 적용

Personalized Map 어플리케이션은 CaaS 기반 모바일 어플리케이션으로, 지역 정보 그룹 관리, 사용자 관심 관리 등의 기능을 제공하며, 사용자가 목적에 맞게 여러 위치를 그룹화하여 저장할 수 있다. 예를 들어, 사용자가 "나만의 맛집"이라는 그룹을 만들고, 자신이 좋아하는 맛집에 대한 위치를 저장하여, 나중에 지도 상에 그 위치가 출력해준다.



(그림 15) 서비스를 뷰 계층의 부분으로 활용하는 어플리케이션

지도를 작성하는 기능은 많은 자원이 소모되어 모바일 디바이스의 자원만으로 구현하기 어렵고, 지도 상의 위치를 출력해주는 기능은 이미 Google에서 잘 정의한 Google Map 서비스가 있으므로, 이 CaaS 서비스를 사용하도록 결정하였다. 그리고 안드로이드 어플리케이션 구현 방법에 따르면, Google Map API를 Activity오류! 참조 원본을 찾을 수 없습니다.에서 호출하게 되어 있으므로, 뷰 계층의 LocationView에서 Google Map 서비스를 호출하도록 결정하였다. (그림 15)는 이런 아키텍처 결정 사항을 반영한 아키텍처의 구조적 뷰이다.

뷰 계층은 *LocaitonView*와 *ListView*로 구성되어 있다. *ListView*는 저장된 위치 정보를 리스트 형태로 보여주는 컴포넌트이고, *LocaitonView*는 Google Map을 기반으로 그 위에 저장된 위치 정보를 추가하여 보여주는 컴포넌트이다. 사용자 정보 및 위치 정보는 각각 컨트롤러와 모델 계층에 속한 컴포넌트를 통해 단말기 상에서 Preference 형태로 저장된다.

본 사례 연구에 대한 아키텍처를 분석하면 다음과 같은 결과를 도출할 수 있다.

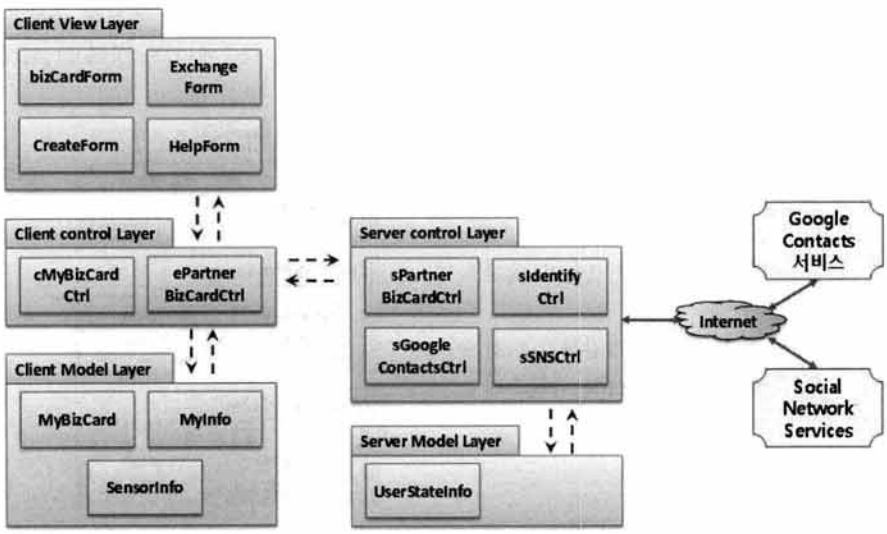
- 모바일 디바이스에는 위치 정보만 저장되어 있으며 사용자가 위치에 대한 정보를 계속 저장한다. 이는 특정 개인과 관련된 정보이고 또한 모바일 디바이스는 크지 않은 데이터를 저장하는 기능만을 가지고 있기 때문에 많은 자원을 소모하지 않는다.
- 지도의 방대한 데이터에서 특정 부분을 제공해야 하므로 복잡한 계산이 필요하고 제공되는 콘텐츠 또한 개인이 구축하기 어렵다. 그러므로 Google 지도 서비스를 CaaS 서비스로 활용하였다.

6.4 하이브리드 아키텍처 패턴의 적용

Mobile Business Card(MBC)는 모바일 디바이스를 사용하여 사용자들의 전자 명함을 교환하기 위한 안드로이드 기반의 어플리케이션이다. MBC는 사용자가 자신의 모바일 명함 관리하는 기능, 상대방과 모바일 명함을 교환하는 기능, 교환된 명함을 관리하는 기능을 제공한다. MBC는 명함은 개인 정보를 다루기 때문에 정보 보안이 필요하고, 모바일 어플리케이션으로 정보를 효율적으로 관리하는 것이 중요하다. 이러한 요구사항을 고려한다면, MBC는 (그림 16)의 하이브리드 아키텍처 패턴을 적용하는 것이 적합하다.

MBC의 아키텍처를 선정하기 위해서 융합될 아키텍처들을 선정하는 것이 선 과정이다. 사용자가 정보 입력 및 수정이 용이하도록 *C.Control*과 *C.Model*의 상호작용을 유지하도록 한다. MBC가 모바일 명함을 교환할 때는 모바일 디바이스의 센서를 활용하는데, 모바일 기기마다 센서의 기능과 범위가 다르므로 어플리케이션 이 원하는 결과 값의 범위에서 벗어난 경우 서버가 대안을 찾아 모바일 명함 교환의 실행률을 높인다. 지금까지의 구조는 Balanced MVC 아키텍처가 적용 대상이다. 하지만, 상대방으로부터 받은 모바일 명함을 관리하기에는 모바일 디바이스에는 자원이 부족하고, 서버에는 저장 비용이 많이 필요로 한다. 따라서 MBC는 서비스인 Google Contacts과 Social Network Service를 사용하여 정보를 저장하기도 하고 호출한다. Google Contacts은 상대방의 모바일 명함을 저장하고 호출하는 기능과 더불어 재사용이 가능하다는 장점을 가지고 있기 때문이다. 따라서 MBC는 Balanced MVC와 서비스 기반의 아키텍처를 융합한 하이브리드 아키텍처 패턴이 적용된다.

다음은 선정된 아키텍처를 바탕으로 Balanced MVC 아키텍처 유형을 선택한다. 클라이언트와 서버 측 모두 데이터 저장이 필요하므로 유형 2를 적용한다. 클라이언트와 서버 간의 네트워크 통신은 모바일 명함 교환 시에 일어나게 되므로, 클라이언트에서는 센서 기능을 처리하는 ePartenr CartCtrl 클래스가 있는 *C.Control* 계층과 서버에서는 카드 교환 시 디바이스의 상태를 수집하고 분석하는 sPartnerBiz Card 클래스가 있는 *S.Control* 계층 간의 상호작용이 일어나게 된다. 따라서 경로 3의 형태로 설계된다.



(그림 16) MBC 어플리케이션의 하이브리드 아키텍처 패턴 적용

본 사례 연구를 통하여 하이브리드 아키텍처를 분석하면 다음과 같은 결과를 도출할 수 있다.

• 클라이언트 측에는 사용자만 사용하는 정보를 저장하여, 정보 보호를 위한 구조로 설계된다. 사용자 정보는 *C.View, C.Contrl, C.Model*의 상호작용을 하며, 데이터 입력, 수정 및 전송 등의 간단한 기능을 수행한다. 클라이언트 측에 저장되는 데이터 양이 적고, 기능의 복잡도도 낮아 모바일 디바이스에서 자원 소모를 줄일 수 있다.

• 서비스에 데이터를 저장하는 방식을 사용하여 Balanced MVC의 서버가 가지는 비용을 줄 일 수 있다. *S.Control*는 서비스를 요청하고 서비스로부터 받은 데이터를 정제하는 부분을 담당함으로써, 기능을 분산하여 기능 요청에 대해 빠른 응답이 가능하다.

7. 평가

7.1 아키텍처 드라이버 별 아키텍처 패턴 평가

본 장에서는 본 논문의 4장에서 제안된 네 가지의 아키텍처 패턴이 4장에서 언급된 네 가지의 아키텍처 드라이버를 어떻게 만족하는지를 평가한다. 보다 논리적으로 평가하기 위하여, 각 아키텍처 드라이버를 결정할 수 있는 주요 특징을 세분화하여 평가한다.

7.1.1 자원 소비 최소화 관점

자원 서비스 최소화 아키텍처 드라이버는 모바일 디바이스의 고유한 특징으로 인해 유도된 것으로, 모바일 어플리케이션이 자원 소비의 효율성을 고려하여 설계하였는지를 평가함으로써 지원 유무를 판단할 수 있다. 자원 소비는 크게 메모리와 CPU, 네트워크 관점에서 기술할 수 있다[16].

메모리와 CPU 관점에서 평가하기 위하여 1) 모바일 디바이스가 아닌 다른 위치에서 기능을 수행하는지에 대한 여부와 2) 모바일 디바이스에서 수행되는 기능의 복잡도를 고려하고, 네트워크 관점에서 평가하기 위해, 3) 네트워크 사용 빈도를 고려한다.

Client-Server 아키텍처 패턴은 1) 모바일 어플리케이션의 일부 기능이 서버에서 실행되도록 설계되므로, 모바일 디바이스가 아닌 다른 위치에서 기능을 수행을 허용한다. 2) 복잡도가 높은 기능은 주로 서버 측에서 실행되고, 어플리케이션에서 사용하는 모든 데이터는 서버에서 관리되므로, 모바일 디바이스에서 수행되는 기능 복잡도가 낮아진다. 3) 서버 기능을 실행시키기 위해 네트워크를 통한 기능 호출이 생기므로, 네트워크 사용 빈도는 다소 높아진다.

Balanced MVC 아키텍처 패턴은 1) 일부 기능이 서버에서 실행되도록 설계되므로, 모바일 디바이스가 아닌 다른 위치에서 기능을 수행을 허용한다. 2) 복잡도가 높은 기능이 주로 서버 측에서 실행되므로, 모바일 디바이스에서 수행되는 기능 복잡도는 감소하지만, 모바일 디바이스에서 데이터 일부가 관리됨에 따라 Client-Server 아키텍처 패턴에서 모바일 디바이스에서 수행되는 기능성 복잡도보다는 다소 높다. 3) 서버 기능을 실행시키기 위해 네트워크를 통한 기능 호출이 생기므로, 네트워크 사용 빈도는 다소 높아진다.

서비스 기반 아키텍처 패턴은 1) 일부 기능이 서비스 제공자의 서버에서 실행되므로, 모바일 디바이스가 아닌 다른 위치에서 기능을 수행을 허용한다. 2) 서비스는 재사용성을 기준으로 설계된 기능 단위이므로, 서비스에서 복잡도가 높은 기능을 수행하게 될 경우에만 모바일 디바이스의 기능 복잡도는 낮아지게 된다. 반대로 서비스에서 복잡도가 낮은 기능을 수행할 경우에는 모바일 디바이스의 기능의 복잡도가 높을 수도 있다. 3) 서비스를 호출하기 위해 네트워크 통신비용이 발생한다.

하이브리드 아키텍처 패턴은 역시 1) 일부 기능이 서버 또는 서비스 제공자의 서버에서 실행되므로, 모바일 디바이스가 아닌 다른 위치에서 기능을 수행을 허용한다. 2) 복잡도가 높은 기능이 외부 서버에서 실행되거나 서비스를 통해 일부 기능을 제공받으므로, 모바일 디바이스에서 수행되는 기능성의 복잡도는 낮다. 그러나, 모바일 디바이스에 일부 데이터베이스를 관리함으로 이에 따라 복잡도가 약간 증가한다. 3) 외부 서버 기능 및 서비스를 호출하기 위하여 네트워크 통신 비용이 발생한다.

7.1.2 높은 성능 관점

성능은 시간을 측정하는 것으로 대표되는 특징이다[15]. 성능은 반응성과 확장성을 포함하는 개념으로 시스템 설계에 따라 많은 영향을 받는 품질 속성이다[17]. 따라서 성능 관점을 평가하기 위하여 1) 반응성과 2) 확장성을 고려한다.

Client-Server 아키텍처 패턴은 1) 일부 기능은 모바일 디바이스에 있으므로, 이 기능에 대해서는 빠른 응답성을 보장할 수 있지만, 서버에 있는 기능에 대해서는 네트워크 오버헤드가 발생 비용 (즉, 네트워크 통신비용)에 따라 응답성이 좌우된다. 대부분은 모바일 디바이스보다 많이 풍부한 컴퓨팅 파워를 가지고 기능을 빠르게 실행하므로, 네트워크 오버헤드로 인한 부작용이 상대적으로 작다. 2) 별도의 서버를 추가하거나 서버의 용량을 늘림으로써, 모바일 디바이스가 감당할 수 없는 작업량을 위임할 수 있으므로 확장성을 보장할 수 있다.

Balanced MVC 아키텍처 패턴은 1) Client-Server 아키텍처 패턴과 같은 이유로 빠른 응답성을 복잡할 수 있다. 그리고 일부 데이터가 모바일 디바이스에 관리되므로, 모든 데이터를 서버에 의존하는 Client-Server 아키텍처 패턴보다는 네트워크 통신 비용이 줄어들어 보다 빠른 응답성을 보장한다. 2) 별도의 서버를 추가하거나 서버의 용량을 늘림으로써, 모바일 디바이스가 감당할 수 없는 작업량을 위임할 수 있으므로 확장성을 보장할 수 있다.

서비스 기반 아키텍처 패턴은 1) 일부 기능은 모바일 디바이스에 있으므로, 이 기능에 대해서는 빠른 응답성을 보장할 수 있지만, 서비스로 제공되는 기능에 대해서는 네트워크 오버헤드가 발생 비용 (즉, 네트워크 통신비용)에 따라 응답성이 좌우된다. 2) 별도의 서버를 추가하거나 서버의 용량을 늘림으로써, 모바일 디바이스가 감당할 수 없는 작업

량을 위임할 수 있으므로 확장성을 보장할 수 있지만, 이는 서비스 제공자의 결정에 좌우된다.

하이브리드 아키텍처 패턴 역시 1) 모바일 디바이스에는 복잡도가 적은 기능을 처리하므로 빠른 응답을 보장하며, 일부 기능은 서비스로 실행되고, 복잡도가 높은 기능은 풍부한 컴퓨팅 파워를 가지는 서버 측에서 실행되므로 빠른 응답성을 보장한다. 서버 또는 서비스 실행에 따른 네트워크 통신 비용이 발생하지만, 이는 서버 기능 실행으로 인한 이점에 비해 적은 양이다 [18]. 2) 별도의 서버를 추가하거나 서버의 용량을 늘림으로써, 모바일 디바이스가 감당할 수 없는 작업량을 위임할 수 있으므로 확장성을 보장할 수 있다.

7.1.3 신뢰성 관점

신뢰성은 시스템이 요구되는 기능을 명백히 규정된 조건에서 명세 된 시간 동안 제공하는 것이다[15]. 신뢰성은 위협에 따른 대응 능력을 통해 측정할 수 있으며, 모바일 어플리케이션에 위협을 줄 수 있는 요소, 즉, 1) 네트워크 불안정성과 2) 외부 자원 불안정성에 대한 대응 능력을 통해 각 아키텍처 패턴들을 평가한다.

본 논문에서 제시한 4가지 아키텍처 패턴은 1) 일부 기능을 서버에 위치시키거나 또는 서비스로 대체하여 수행하므로, 네트워크가 불안정하면 서버에 설치된 기능 및 서비스를 사용할 수 없게 되어 신뢰성에 악영향을 미친다. 이를 위해 네트워크가 불안정한 경우에 대비한 설계가 수반되어야 한다. 2) 일부 기능이 서버 또는 서비스 제공자 측에 실행되므로, 외부 자원이 불안정한 경우에 대응 능력을 보장해야 한다. Client-Server 아키텍처 패턴, Balanced MVC 아키텍처 패턴, 하이브리드 아키텍처 패턴은 모바일 어플리케이션을 위한 특정 서버에 기능을 설치하므로, 이 서버에 신뢰성을 보장하는 기법을 적용하여 아키텍처를 설계해야 한다[9]. 서비스 기반 아키텍처 패턴은 서비스 제공자 측에서 신뢰성을 얼마나 보장하는지에 따라 결정되므로, 이 경우에는 문제가 발생하였을 때 다른 서비스를 대체하도록 모바일 디바이스 측 어플리케이션에 설계가 되어야 한다.

7.1.4 복잡도가 높은 기능성 관점

복잡도가 높은 기능성 관점에서 아키텍처 패턴을 평가하는 것은 앞의 다른 아키텍처 드라이버와는 달리 비교적 간단하다. 이는 각 아키텍처 패턴이 복잡도가 높은 기능성을 수용할 수 있도록 설계 지침을 제공하는지를 기준으로 평가한다.

Client-Server 아키텍처와 Balanced MVC 아키텍처 패턴은 모바일 어플리케이션 설계부터 기능도가 높고 응집력이 높은 기능 단위를 서버로 배치시키는 구조이기 때문에, 복잡도가 높은 기능성을 가지는 어플리케이션을 이 아키텍처 패턴을 이용하여 개발할 수 있다. 단, Balanced MVC 아키텍처 패턴은 모바일 디바이스에도 별도의 데이터베이스를 관리하므로, 이에 따른 복잡도가 증가하게 된다. 그러므로, 모바일 디바이스에 관리되는 데이터가 기능 복잡도의 부하를 가중시키지 않도록 설계해야 한다.

서비스 기반 아키텍처는 일부 기능이 서비스 형태로 서비스 제공자에 의해 제공이 되지만, 서비스는 재사용성을 기준으로 설계된 단위이므로 기능성 복잡도와는 다소 무관하

〈표 2〉 기존 연구와의 비교 평가 (X: 언급하지 않음. △: 일부 언급함. ○: 언급함)

	모바일 특징 제시	모바일 아키텍처 스타일 정의	설계 지침 정의
Lee의 연구[2]	X (구체적으로 언급하지 않음)	△ (여러 형태의 클라이언트-서버 아키텍처 스타일 제안함)	X (구체적으로 언급하지 않음)
Gruhn 의 연구[3]	△ (이동성 지원, 풍부한 네트워크가 간접적으로 언급됨)	△ (네트워크 연결 지원 유/무에 따라 네 가지 종류의 아키텍처 스타일 제안함)	X (구체적으로 언급하지 않음)
Dagtas 의 연구[4]	△ (제한된 자원, 네트워크 불안정성이 간접적으로 언급됨)	△ (클라이언트-서버 기반으로 경량 아키텍처 제안하고, 주요 컴포넌트 도출함)	△ (일부 컴포넌트에 대한 설계 모델이 제안됨)
Natchetoi의 연구[5]	△ (제한된 자원, 네트워크 불안정성이 간접적으로 언급됨)	△ (SOA 서비스의 클라이언트로서의 아키텍처를 제안함)	X (설계 지침보다는 설계 이슈에 대해 대략 언급함)
Ennai의 연구[6]	△ (이동성 지원, 네트워크 불안정성이 간접적으로 언급됨)	△ (SOA 서비스의 클라이언트로서의 아키텍처를 제안함)	X (설계 지침보다는 SOA 사용 시 발생하는 이슈에 대해 설명함)
Aaratee의 연구[7]	○ (자원 제약성과 상황 인지 기능에 대하여 서론에 언급함.)	△ (컨텍스트 수집 및 추론에 대한 전반적인 아키텍처 구조를 설명하였지만, 특정 스타일을 언급한 것은 아님)	X (구체적으로 언급하지 않음)
본 연구	○ (3.1절에서 모바일 어플리케이션의 특징을 언급하고, 이로부터 아키텍처 드라이버도 도출함)	○ (4장에서 모바일 어플리케이션을 4개로 분류하여 아키텍처 패턴을 제안함)	○ (4장에서 제안한 아키텍처에 대한 구조 및 동적 뷰, 설계지침을 제공함. 또한, 5장에서는 사례연구를 통해 설계 지침의 적용 가능성을 증명함)

다. 즉, 서비스로 실행되는 기능이 복잡도가 높다는 것을 항상 보장할 수 없다. 그러므로, 서비스로 제공되는 기능이 복잡도가 높을 때만, 서비스 기반 아키텍처를 사용한 어플리케이션이 복잡도가 높은 기능성을 수용할 수 있게 된다.

하이브리드 아키텍처는 Balanced MVC, 서비스 기반 아키텍처 패턴 중의 일부를 융합한 형태이므로, 복잡도가 높은 기능성을 수용할 수 있다.

7.2 기존 연구와의 비교

본 절에서는 모바일 어플리케이션을 위한 아키텍처 설계에 대한 대표적인 연구와 비교 평가를 수행한다. 평가 기준은 널리 사용되는 아키텍처 설계 프로세스에 따라 모바일 어플리케이션을 위한 아키텍처 드라이버 도출을 위한 근거가 있는지, 모바일 어플리케이션 아키텍처를 위한 스타일이 제시되었는지, 해당 아키텍처를 설계하는 지침이 있는지를 기준으로 평가한다. 그러므로 모바일 특징 제시, 모바일 어플리케이션을 위한 아키텍처 스타일 정의, 설계 지침 정의를 포함하여 3가지를 평가 기준으로 하여, <표 2>과 같이 제안된 아키텍처 패턴을 평가한다.

본 평가에서 다룬 기존 연구들은 본 논문의 연구는 연구 범주와 깊이 2가지 측면에서 차이를 보인다. 먼저 연구 범주의 측면에서 보면, 기존 연구들은 서비스 기반 아키텍처 또는 오프로딩 아키텍처와 같이 모바일 어플리케이션에 적용될 수 있는 특정 아키텍처를 다루었다. 그리고 기존의 연구는 서버 측의 기능을 두고 클라이언트와 병행 실행하는 아키텍처만 제안하였고, 산업계에서 많이 사용되는 Client-Server, MVC, 서비스 기반과 같이 모바일 클라이언트와 서버에 기능을 적절히 배치하여 모바일 어플리케이션을 설계하는 여러 설계 선택사항을 정의하였다. 그리고 개별 아키텍처 패턴의 장점을 극대화하고, 단점을 최소화할 수 있는 하이브리드 개념의 아키텍처를 새로 정의하였다.

연구 깊이의 측면에서 보면, (1) 기존의 연구에서는 제시한 아키텍처에 대하여, 구조를 나타내는 다이어그램을 제시하고 있으나, 대부분의 아키텍처 기술이 구조적 관점에서 추상적 수준에 머물러 있다. (2) 본 논문에서는 최소한의 설계 순서와 적용 가능한 경우 등을 제시함으로써, 제안한 패턴을 적용하기 위한 설계 지침을 정의하여 고품질의 모바일 어플리케이션 아키텍처의 설계를 가능하게 하였다.

8. 결 론

모바일 디바이스의 특징으로 인해 모바일 디바이스에서는 자원 소비가 많고 높은 복잡도를 가진 어플리케이션은 모바일 디바이스에서 설치, 운영되기 어렵다. 그러나 모바일 디바이스는 풍부한 네트워크를 지원할 수 있는 능력 등의 장점이 있어 서비스 등의 외부 자원을 사용하기가 용이하다. 이런 특징들은 모바일 어플리케이션의 비기능적 요구사항에 속하므로, 이들은 아키텍처 설계 시에 고려되어야 한다. 즉, 복잡한 기능성을 가지고 있는 모바일 어플리케이션의 아키텍처 설계에서 모바일 어플리케이션이 가지고 있는 제약 사항을 빠르고 정확하게 해결하기 위해서는 모바일 어플리케이션에 대한 아키텍처 패턴 및 설계 기법이 필요하다.

그러므로 본 논문에서는 먼저 전통적인 시스템과는 다른 모바일 어플리케이션의 특징을 정의하고, 이를 반영한 아키텍처를 설계하기 위해 아키텍처 드라이버를 제시하였다. 고품질의 모바일 어플리케이션 개발을 위한 4가지 패턴인 Client-Server, Balanced MVC, 서비스 기반, 하이브리드 아키텍처 패턴을 제안하였다. 제안된 아키텍처 패턴에 대한 구조와 동적 뷰를 통해 아키텍처의 장, 단점을 설명하였고, 아키텍처 패턴을 효과적으로 설계하기 위한 설계지침을 제시하였다. 또한, 제시된 아키텍처 패턴의 적용 가능성을 보여주기 위해 사례 연구를 실행하였고 사례 연구를 하여 얻은 결과 및 교훈을 설명하였다. 마지막으로, 제시된 아키텍처 패턴이 제시된 아키텍처 드라이브를 만족하게 하는지 알아보기 위하여 아키텍처 드라이버 별로 아키텍처 패턴을 평가하였고 또한 기존 연구와의 비교를 통해 본 논문이 독창적임을 보여주었다.

제시된 아키텍처 패턴은 모바일 어플리케이션 아키텍처 설계 시 자주 나오는 방식으로 이를 활용하여 빠르고 정확한 고품질의 모바일 어플리케이션 개발을 예상한다. 본 논문에서는 언급된 아키텍처 패턴은 기능 분할을 중심인 모바일 어플리케이션이다. 향후 모바일의 센싱 기술, 융합 기술 등 새로운 기술에 대한 모바일 어플리케이션 아키텍처 패턴에 대한 지속적인 연구가 이루어질 것이다.

참 고 문 헌

[1] Taylor, R.N., Medvidovic, N., and Dashofy, E.M., *Software Architecture: Foundations, Theory, and Practice*, Wiley, January, 2009.

[2] Lee, V., Schneider, H., and Schell, R., *Mobile Applications: Architecture, Design, and Development*, Prentice Hall, April, 2004.

[3] Gruhn, V. and Kohler, A., "Aligning Software Architectures of Mobile Applications on Business Requirements," *In Proceedings of Workshop on Web Information Systems Modeling (WISM 2006) in conjunction with the 8th International Conference on Advanced Information Systems Engineering (CAISE 2006)*, pp.1113-1123, 2006.

[4] Dagtas, S. Natchetoi, Y., Wu, H., and Hamdi, L., "An Integrated Lightweight Software Architecture for Mobile Business Applications," *In Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pp.41-50, 2008.

[5] Natchetoi, Y., Kaufman, V., and Shapiro, A., "Service-Oriented Architecture for Mobile Applications," *In Proceedings of the 1st international workshop on Software architectures and mobility (SAM '08)*, pp.27-32, 2008.

[6] Ennai A and Bose S, "MobileSOA: A Service Oriented Web 2.0 Framework for Context-Aware, Lightweight and Flexible Mobile Applications," *In Proceedings of the 2009 12th Enterprise Distributed Object Computing Conference Workshop (EDOCW 2008)*, pp.348-382, 2008.

[7] Aaratee Chrestha, "MobileSOA framework for Context-Aware Mobile Applications," *In Proceedings of 11th Mobile Data Management (MDM 2010)*, pp.297-298, 2010

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

[9] Nick Rozanski and Eoin Woods, *Software Systems Architecture: working with stakeholders using viewpoints and perspectives*, Addison-Wesley, 2005, pp.146

[10] Pressman, R., *Software Engineering: A Practitioner's Approach*, 7th Edition, McGraw-Hill Science, 2009.

[11] Frank B, Regine M, Hans R, Peter S and Michael S, Pattern-Oriented Software Architecture, Wiley, 1996.

[12] 김수동, 라현정, "안드로이드 기반 모바일 서비스 어플리케이션의 아키텍처," *정보과학회지*, 제 28권, 제 6호, pp.25-34, 2010년 6월.

[13] Erl, T., *Service-Oriented Architecture (SOA): Concepts, Technology*, and Design, Prentice Hall, 2005.

[14] La, H.J. and Kim, S.D., "Adapter Patterns for Resolving Mismatches in Service Discovery," *In Proceedings of the 5th International Workshop on Engineering Service Oriented Applications (WESOA 2009)*, pp.498-508, Nov. 23, 2009.

[15] ISO/IEC 9126-1, Software Engineering-Product Quality-Part 1: Quality Model, ISO/IEC, June 15, 2001.

[16] Ryan, C. and Rossi, P., "Software, Performance, and Resource Utilization Metrics for Context-Aware Mobile Applications," *In Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, pp.12, 2005.

[17] Smith, C. U. and Williams, L., G., *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, 2002.

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

장 정 란

e-mail : jrjang10@gmail.com

2009년 숭실대학교 미디어학부(공학사)

2010년~현 재 숭실대학교 컴퓨터학과 석사과정

관심분야 : 소프트웨어 아키텍처(Software Architecture), 모바일 컴퓨팅(Mobile Computing)

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 소프트웨어공학센터 연구교수

관심분야 : 서비스 지향 컴퓨팅(Service-Oriented Computing), 소프트웨어 아키텍처(Software Architecture), 모바일 컴퓨팅(Mobile Computing)

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 : 객체지향S/W공학, 소프트웨어 아키텍처(Software Architecture), 클라우드 컴퓨팅(Cloud Computing), 모바일 컴퓨팅(Mobile Computing)