논문

마이크로서비스 아키텍처의 빛과 그림자

마이크로서비스 아키텍처 적용 시 고려사항

Light and Shadow of Microservice Architecture

양인호(Yang, In-ho)*

* ㈜크루메이트 공공사업본부 차장, 한남대학교 일반대학원 기록관리학과 박사과정.
■ 투고일 : 2019년 03월 18일 ■ 최초심사일 : 2019년 04월 02일 ■ 게재확정일 : 2019년 04월 15일.
■ 기록학연구 60, 283-315, 2019, https://doi.org/10.20923/kjas.2019.60.283

〈초록〉

소프트웨어 산업은 새로운 비즈니스 모델의 등장으로 빠르고 유연하게 대처할 수 있는 기술로의 변화가 요구되고 있다. 이와 관련하여 차세대 기록관리시스템의 아키텍처로 거론되고 있는 마이크로서비스는 민첩성과 편리성을 지니는 아키텍처로 급부상했다. 마이크로서비스를 적용한 기록관리시스템을 개발한다면 혁신의 기반과 함께 민첩성과 확장성을 확보할 수 있다. 본 연구의 목적은 마이크로서비스를 적용한 기록관리시스템 구축 시 효율적인 방안을 제시하는 데 있다. 이를 위해 먼저 전통적 방식의 아키텍처를 살펴보고, 마이크로서비스의 주요 특징을 설명하였다. 또한 마이크로서비스 아키텍처를 적용한 시스템 구축 시 '마이크로서비스의 짧은 역사', '기술의 성숙도', '프로젝트팀의 기술 수준' 등 세 가지 측면을 검토해야 함을 정리하였다. 그리고 국내 환경에 맞춰 마이크로서비스를 적용한 기록관리시스템 구축 시 고려사항을 제시하였다.

주제어 : 마이크로서비스, 서비스지향 아키텍처, 모놀리스 아키텍처, 데브옵스, 차세대 기록관리시스템

〈Abstract〉

The emergence of new business models software industry is demanding a change to technology that can cope quickly and flexibly. In this regard, microservices, which is being addressed as the architecture of the next-generation record management system, has emerged as an agile and convenient architecture. If record-management system with micro-service is developed, agility and expandability with basement of innovation can be ensured. The purpose of study is to suggest efficient ways when record-management system with micro-service is built. For this, tranditional architecture has been checked and main features of micro-service have been explained. Also, it was summarized that three points : 'Short history of micro-service', 'Maturity

of technology' and 'Techical level of project team' have to be reviewed when record-management system with micro-service architecture is built. And we suggested some issues to consider when constructing the records management system applying microservices according to the domestic environment.

Keywords : Microservice, Service-Oriented Architecture, Monolith Architecture, DevOps, Next Generation Records Management System

1. 서론

오늘날 IT산업은 제4차 산업혁명을 이끌 만큼 폭발적으로 성장하고 있다. 산업이 발전하면서 소프트웨어는 늘어났고, 증가된 소프트웨어는 새로운 비즈니스 모델의 등장으로 빠르고 유연하게 대처할 수 있는 기술의 변화가 요구되고 있다.

"소프트웨어를 유지하는 것은 '이전처럼 동작하게 하는 것'이 아니다. '변화하는 세상에서도 항상 유용하게 만드는 것'이다." 원자 이론학자 제시카 커(Jessica Kerr)의 말을 인용한 SW 컨설팅 회사 피보탈(Pivotal) 필 웹(Phil Webb)의 발표는 오늘날 소프트웨어가 가지는 의미를 되새기게 한다. 이처럼 오늘날 웹 또는 모바일 기반의 서비스를 제공하는 대부분의 회사는 소프트웨어와 데이터를 다루는 능력, 그 자체가 경쟁력이다. 아이디어를 동작하는 소프트웨어로 구현하고, 이 소프트웨어를 가능한 한 빨리 고객에게 전달해서 그 가치를 검증하는 일을 반복하는 것은 사업의 성패에 매우 중요하다(Josh Long · Kenny Bastani 2018, 20).

이와 관련하여 주목할 만한 기술이 있다. 아마존 최고기술책임자(CTO) 베르너 보겔스(Werner Vogles)는 2006년 JAOO 컨퍼런스에서 오늘날 NoSQL의 기반이 되는 CAP이론(CAP theorem)에 대해 이야기 했다. 또한 자체 데이터

베이스를 갖고 서비스를 개발하고 운영하는 작은 팀에 대해서도 이야기했다. 오늘날 이러한 구조는 데브옵스(DevOps)로 불리며, 이와 같은 아키텍처(architecture)는 마이크로서비스(Microservices)로 알려져 있다(Eberhard Wolff 2016, 15). 마이크로서비스 아키텍처는 개별적으로 동작하는 작고(micro), 독립적인 서비스들의 결합을 통해 하나의 큰 애플리케이션(응용프로그램)을 구축할 수 있는 아키텍처로 기존 방식보다 애플리케이션을 보다 더 빠르게 구축할 수 있다.

최근 기록관리 분야에서도 차세대 기록관리시스템 설계와 관련하여 마이크로서비스에 대한 관심이 높아지고 있다. 안대진 · 임진희는 "국가기록원이 표준영구기록관리시스템을 만들어 배포하는 것이 불가피하다면 최소한 기관의 업무환경에 맞추어 기능을 커스터마이징할 수 있는 구조로 설계되어야 함"(안대진 · 임진희 2016)을 강조하였으며, 2017년 국가기록원 R&D사업 '차세대 기록관리 모델 재설계 연구'의 일환으로 수행된 연구들은 하나같이 "시스템 문제의 원인인 모놀리스(Monolithic) 설계에 대해 모듈화를 통한 유연하고 확장성 있는 마이크로서비스 아키텍처로 재설계가 필요함"을 주장하고 있다(주현미 · 임진희 2017a; 주현미 · 임진희 2017b; 안대진 · 임진희 2017; 김기정 · 신동수 2018; 오진관 · 임진희 2017).

이런 연구들의 영향으로 서울시는 2019년 서울기록원 건립을 앞두고 마이크로서비스 방식의 모듈화된 기록시스템을 개발하고 있으며(안대진 · 임진희 2017), 국가기록원도 단위별 변경이 가능한 '다기능 · 연계형 복합시스템'으로의 전환을 모색 중이다(이승억 2017).

하지만 기존 연구들은 마이크로서비스 아키텍처 도입 시 나타나는 이점에만 주목하는 경향이 있어 실제 서비스 구축 시 발생할 수 있는 문제점에 대한 논의는 부족한 실정이다. 사실 마이크로서비스는 아직까지 표준화된 구축 방법론도 없으며, 10년이 채 안 되는 짧은 기간 동안 발전해온 아키텍처로 아직은 기술적인 보완이 필요하다. 이와 관련하여 앞으로 소프트웨어 시스템을 구성하는 방안으로 "마이크로서비스가 미래를 주도할 수 있는가?"

라는 물음에 IT컨설턴트 마틴 파울러(Martin Fowler)는 유보적인 입장을 취했다. 그 이유는 '마이크로서비스의 짧은 역사', '기술적인 성숙도 부족', '프로젝트팀의 기술 수준' 으로 답변을 요약할 수 있다. 마틴 파울러는 분명 마이크로서비스는 해볼 만 한 것이지만, 마이크로서비스를 적용하고자 한다면 불명확한 몇 가지 사항들에 대해 판단하고 도전해야 할 것이라고 조언하고 있다(Martin Fowler 2014).

본 연구는 먼저 전통적 방식의 아키텍처를 먼저 살펴보고 마이크로서비스의 주요 특징과 마틴 파울러가 입장을 유보한 세 가지 이유에 대해 논의하고자 한다. 이러한 논의를 통해 마이크로서비스를 적용하여 기록관리시스템 구축 시 고려해야 할 주요 사항들을 제시하고자 한다. 마이크로서비스가 무엇인지 정확히 파악하고 대처할 수 있어야 실패에 대한 위험성을 낮출 수 있기 때문이다.

먼저 2장에서는 아키텍처의 개념과 함께 기존 전통적 방식인 모놀리스(Monolith) 아키텍처와 서비스 지향 아키텍처(Service Oriented Architecture, SOA)의 정의와 특징을 제시하였다. 3장에서는 마이크로서비스 아키텍처의 이점과 마틴 파울러가 제시한 불명확한 세 가지 사항들에 대해 논의한 후 4장에서 기록관리시스템 구축 시 고려해야 할 주요 사항들을 제시하였다.

연구방법은 문헌조사를 통한 사례연구로 주요 기업과 IT전문가들의 블로그 및 사이트를 분석하여 그들의 경험을 토대로 특징과 시사점을 정리했다. 이 연구의 한계는 마이크로서비스를 적용한 시스템 구축 시 기술적인 부분에 대한 명확한 방법론을 제시하지 못했다는 데 있다. 이는 마이크로서비스가 우리나라에서 이제 막 도입이 고려되고 있는 새로운 방식이며, 아직 기술적으로 완벽하게 정착하지 못한 외부 환경과도 연관된다.

2. 전통적 방식의 아키텍처

1) 소프트웨어 아키텍처(Software Architecture)

아키텍처(Architecture)의 개념은 인류 역사만큼 오래되었다. 인간이 만든 모든 창조물에는 아키텍처가 존재한다. 로마 시대의 아키텍트로 유명한 비트루비우스(Vitruvius)나 르네상스 시대의 레오나르도 다빈치(Leonardo da Vinci), 미켈란젤로(Michelangelo) 역시 건축물뿐 아니라 선박, 기계, 도시 설계, 시계와 같은 다양한 기술 분야에 아키텍처를 적용하였다(김치수 2015, 216). 이처럼 아키텍처는 건축 양식에 많이 사용되어 왔고, 건축물의 뼈대, 도시 전체로 확장하면 도시의 전체적인 구조 같은 구조물의 특성을 결정하는 기본 구조를 뜻한다.

아키텍처의 개념을 소프트웨어로 한정한다면 “소프트웨어 시스템의 구조를 비롯한 시스템 개발에 중요한 영향을 미치는 결정들로, 소프트웨어 시스템 개발에서 특정 시스템에 대하여 요구되는 기능과 품질을 확보하고, 소프트웨어 시스템의 구축 및 지속적인 개선이 용이하도록 하는 구조(강성원 2010, 131)”라고 정의할 수 있다. 소프트웨어 아키텍처를 통해 소프트웨어 전체 구조를 한눈에 파악 할 수 있으며, 각 요소 간의 관계 및 품질 요구사항을 예측할 수 있는 도구인 것이다.

그러나 이전에는 개발자들이 각자의 경험 또는 정형화되지 않은 스타일대로 소프트웨어를 개발하고 유지하였다. 이런 방식은 관련 이해관계자들의 기술 논의 및 의사소통의 어려움과 개발에 참여하지 않은 사람이 유지보수 시 시스템 구조를 이해하기 어려운 문제점이 발생하였고, 이러한 불편함을 해소하고자 소프트웨어 아키텍처 스타일(software architecture style)이 적용되기 시작했다.

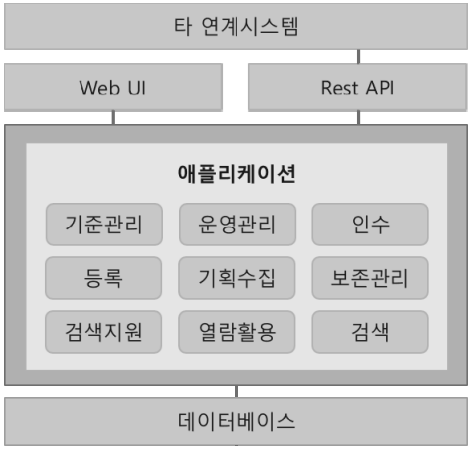
기록물에도 일반문서, 도면, 사진, 정부간행물 등의 다양한 스타일(유형)이 존재하는 것처럼 아키텍처도 다양한 스타일이 존재한다. 여기서 주목해야 할 것은, 아키텍처 스타일은 그 자체로서 주어진 요구사항을 충족시키는

해법을 제시하지는 않는다는 것이다. 단지 그러한 종류의 요구사항을 충족시키기 위하여 이런 스타일의 아키텍처가 효과적일 것으로 예견하기 때문에 그 스타일을 선택하게 되는 것이다(강성원 2010, 136). 본 연구의 주제인 마이크로서비스 아키텍처도 기존 모놀리스 아키텍처가 지닌 문제점을 효과적으로 접근하기 위한 하나의 스타일이라 할 수 있다. 따라서 아키텍처 스타일을 선택할 때는 각 스타일이 지닌 특징을 이해하고 그 스타일이 지닌 장점을 잘 활용할 수 있는 방안을 함께 고려해야 한다.

2) 모놀리스 아키텍처(Monolith Architecture)

모놀리스 아키텍처는 널리 활용되고 있는 전통적인 아키텍처로 모든 업무 로직이 하나의 애플리케이션 형태로 패키지 되어 서비스 되는 방식이다(박성훈 2018, 6).

〈그림 1〉 모놀리스 아키텍처



※출처 : 2018년 기록물관리정보시스템 통합 유지보수사업 제안요청서(국가기록원, 2018)를 바탕으로 재구성

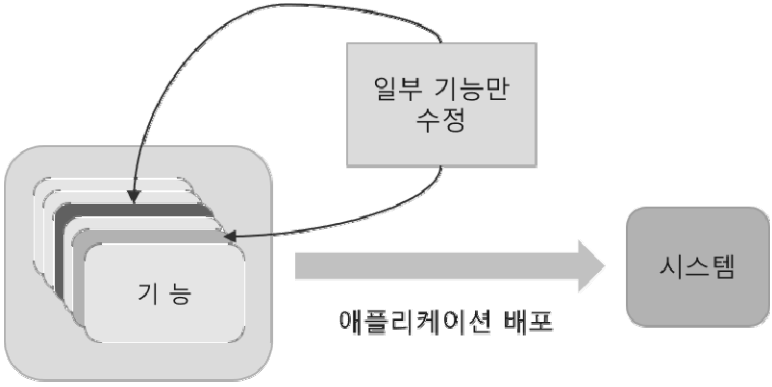
일반적으로 〈그림 1〉과 같이 비즈니스 계층을 담당하는 애플리케이션이 존재하고, 애플리케이션은 하나의 데이터베이스를 참조하는 구조를 취한다. 또한 사용자에게 인터페이스와 시스템 연계를 위한 RESTful API를 갖는다. 이렇게 만들어진 애플리케이션은 하나의 패키지 형태로 배포된다. 자바(Java)의 경우 웹 애플리케이션이라면 위와 같이 WAR 파일로 빌드 되어 톰캣(Tomcat)과 같은 WAS에 배포되는 구조를 갖는다.

모놀리스 아키텍처의 특징으로 첫째, 앞으로 설명할 서비스 지향 아키텍처나 마이크로서비스 아키텍처에 비해 쉽고, 빠르게 구축할 수 있는 장점이 있다. 또한 하나의 단일체 구조이기 때문에 관리가 수월하며, 테스트 및 배포가 간편하고, 서비스 장애 발생 시 장애 파악이 용이하다(임근원 외 2018).

하지만 단일체 구조의 접근방식은 점차 서비스가 확장되고 규모가 커지면서 몇 가지 한계가 나타난다. 대부분의 개발자는 전체 시스템의 구조를 알지 못하기 때문에 재활용 가능한 모듈을 방치한 채 중복된 개발로 코드를 계속적으로 추가하게 되며, 이로 인해 사용하지 않는 코드는 계속 증가하고, 파악하기 힘든 모듈 간의 연계성으로 시스템은 점점 더 복잡해진다. 이로 인해 개발자는 버그 수정 및 새로운 기능구현에 어려움을 겪을 수 있으며, 수정을 하더라도 본인의 의도치 않은 또 다른 버그를 만들어낼 가능성이 존재한다.

둘째, 어플리케이션의 빌드 및 배포시간, 서버의 기동에 많은 시간이 소요된다. 〈그림 2〉와 같이 단일 어플리케이션 중 일부 프로그램만 수정하려고 해도 관련 없는 기능들까지 단일 애플리케이션이 빌드 되어 다시 배포되어야 한다. 또한 변경 배포에 따른 영향도를 파악하기 위해 직접적으로 관련 없는 많은 사람의 노력과 시간을 할애해야 하며, 서비스의 연속성을 위해 2중화, 3중화 되어 있는 장비에 순차적으로 배포하고 재기동하는 시간[1)]이 수십 분에서 수 시간 정도 소요된다(박성훈 2018, 7).

〈그림 2〉 단일 어플리케이션 배포



※출처 : 자바 기반의 마이크로서비스 이해와 아키텍처 구축하기(박성훈, 2018)

셋째, 어느 한 기능에서 장애 발생 시 전체 시스템에 영향을 줄 수 있다. 잘못된 코드를 배포하거나 트래픽 증가로 인하여 서비스 성능에 문제가 있을 때, 서비스 전체의 장애로 확대되는 경우가 발생할 수 있다(쿠팡기술블로그 2018).

넷째, 프로젝트 팀 간의 경계에는 방치되는 영역이 존재할 수 있으며, 이것은 프로젝트의 리스크로 작용 할 수 있다. 모놀리스 형태의 프로젝트에서는 일반적으로 아키텍처팀, 공통팀, 프론트 개발팀, 백엔드 개발팀, 디자인팀, 기획팀 등으로 구성된다. 이들 사이의 경계에는 애매한 업무 영역이 존재하게 되는데 이 업무를 어느 팀에서 맡을 것인가의 문제로 각 팀 간의 감정싸움이 일어날 수 있으며, 이로 인해 무의미한 시간이 소요된다.

다섯째, 대부분의 프로젝트는 개발팀과 운영팀이 다르다. 실제 개발팀에

1) 소셜 커머스 사이트 쿠팡은 배포와 관련하여 기존 모놀리스 아키텍처하의 배포 프로세스는 100명 미만의 개발자로 구성된 조직 안에서는 잘 동작하였지만, 이후 조직이 성장하고, 많은 개발자들이 추가 투입되면서 약 5분정도 수정한 코드를 배포하기 위해 2~3일 정도 대기하는 경우도 종종 발생하였음을 밝혔다(쿠팡기술블로그 2018).

서는 프로젝트 종료 후 운영팀에 업무를 인계하게 되는데 버그 및 기능 수정사항 발생 시 실제 개발에 참여하지 않은 운영팀 개발자가 업무를 진행하는데 어려움이 발생할 수 있다.

이런 단점에도 불구하고 모놀리스 아키텍처는 널리 활용되어 왔으며, 당시에는 최고의 아키텍처로 인식되었다. 국가기록원의 중앙영구기록관리시스템, 표준기록관리시스템도 모놀리스 아키텍처를 기반으로 개발되어 현재까지 활용되고 있다.

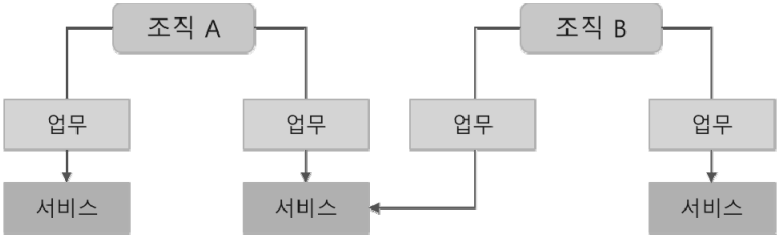
3) 서비스 지향 아키텍처(Service Oriented Architecture)

마이크로서비스 아키텍처(이하 MSA)가 주목받기 이전부터 기업 환경에서는 중복되는 프로세스나 업무들을 하나의 서비스 단위로 개발하여 각 서비스는 호출 가능한 상태로 개발하자는 노력이 계속되어 왔다. 이는 서비스의 생성과 활용을 높여서 비즈니스 환경 변화와 업무 변화에 민첩하게 대응할 수 있는 아키텍처를 갖추기 위함이다(박성훈 2018, 15).

이러한 요구에 대응하기 위한 방안 중에 하나가 서비스 지향 아키텍처(이하 SOA) 이다. SOA란 기존의 애플리케이션들의 기능들을 비즈니스적인 의미를 지니는 기능 단위로 묶고, 표준화된 호출 인터페이스를 통해서 서비스(소프트웨어 컴포넌트) 단위로 재조합한 후, 이 서비스들을 다시 서로 조합(Orchestration)하여 애플리케이션을 만들어내는 소프트웨어 아키텍처이다[2](Jammes, F. · H. Smit 2005, 62-64). 즉 애플리케이션의 개발의 측면보다 서비스를 조합하거나 분리하는 데 중점을 두는 방식이다.

2) SOA는 초기에 통합중심의 '기초적(Fundamental) SOA' 모델에서 '네트워크 SOA'로 발전하고 이후 '프로세스(Process-Enabled) 중심의 SOA'로 발전한다.

〈그림 3〉 서비스 지향



※출처 : 자바 기반의 마이크로서비스 이해와 아키텍처 구축하기(박성훈, 2018)

이처럼 SOA는 전통적인 프로그램 중심의 설계·개발 방식에서 벗어나 비즈니스 프로세스 관점에서 재활용 가능한 단위로 서비스를 설계·개발함으로써 특정 프로세스나 서비스 변경 또는 내부·외부 시스템과의 비즈니스 통합 시 효율적이고 빠른 대응이 가능하다는 점에서 그 의미가 깊다(이상효·양해술 2009, 1576).

SOA는 느슨한 결합관계(Loosely Coupled)를 지향한다. 느슨한 결합관계로 서비스들은 의존관계를 최소화하여 유연성 갖으며, 독립성이 보장된다.

데이터의 교환 방식은 공개표준에 따라 결정된다. 하나의 웹서비스가 다른 웹서비스로 메시지를 전송하는 방식은 전 세계적으로 표준화되어 있고, 이러한 표준화에 준하는 일련의 프로토콜을 통해 전송된다(Thomas Erl 2006, 43). 서비스 사용자는 표준화된 서비스 발견 방식에 따라 어디서든지 원하는 서비스를 찾아 사용이 가능하다(이상효·양해술 2009, 1576).

여러 서비스를 연결하기 위해서 ESB (Enterprise Service Bus)와 같은 메시지 기반 미들웨어인 MOM(Message Oriented Middleware)과 같은 솔루션을 활용하여 프로세스에 따라 묶여진 서비스의 집합인 복합 컴포넌트(Composite Component)를 통해 구축되는데(임철홍 2006) 이로 인해 서비스 간 조합 및 통합이 용이하다. SOA의 가장 큰 매력 중 하나는 이전에 통합되지 않았던 환경을 통합된 체계로 인도한다는 것이다. 기존 애플리케이션과 신규 애플

리케이션을 캡슐화하여 커뮤니케이션 프레임워크에 표준화된 형태의 통합을 할 수 있다(Thomas Erl 2006, 45-46).

이외에도 SOA는 조직의 기민성(시장의 빠른 대처)과 기존 자산을 이용할 수 있다는 점으로 인한 비용절감 효과, 프로세스 중심의 아키텍처 등의 여러 장점으로 많은 기업의 관심을 받아왔다. 행정안전부의 온나라시스템도 우리나라 공공부문에서는 최초로 SOA 사상에 근거하여 행정업무의 개별 기능들을 서비스화 하여 다른 시스템에서도 쉽게 재활용할 수 있도록 구축했다(보안뉴스 2009).

IBM, BEA시스템즈, 마이크로소프트(MS) 등 내로라하는 IT기업들이 SOA 관련 제품을 출시하며 관심이 높아지던 무렵 시장 조사 업체 가트너(Gartner)가 SOA 개념을 선보인지 10년이 지난 시점에 SOA를 도입한 다수의 기업들에게서 나타난 문제점과 함께 부정적인 이미지가 급속도로 확산되었다. 이런 상황에서 세계적인 시장조사기관인 버튼그룹의 부사장 토마스 메인(Thomas Manes)이 쓴 'SOA는 죽었다'(SOA is dead, Long Live Services)란 도발적인 제목의 글은 해외에서 커다란 반향을 불러일으켰다(ZDNet Korea 2009). 토마스 메인에 따르면 SOA를 도입한 기업에서 많은 비용을 투자했지만 IT시스템은 예전과 크게 달라진 바가 없으며, 오히려 비용은 더 많이 들고 프로젝트 기간은 더 오래 소요되며, 많은 취약점을 지닌 시스템이라는 것이다. 결과적으로 SOA는 약속된 이익을 제공하지 못했다(Thomas Manes, n.d.). 또한 기업들이 SOA를 도입하려는 근본적인 이유는 '서비스' 관점에 있는데, IT업체들은 여전히 '기술'과 '아키텍처'만의 접근 방식을 논했기 때문이라는 견해도 있다(ZDNet Korea 2009).

이와 더불어 ESB를 여러 개의 서비스를 조합하는 로직에 무겁게 사용함으로 오버헤드[3]가 발생했으며(조대협 2014), 중앙집중식 데이터 관리방식을 계속 유지했던 점(정도현 2014), SOA는 조직의 거버넌스 입장에서 서비

3) 특정한 목표를 달성하기 위해 간접적 혹은 추가적으로 요구되는 시간, 메모리, 대역폭 혹은 다른 컴퓨터 자원을 말한다(네이버 백과사전 n.d.).

스를 고려해야 하는데 이러한 원칙을 반영한 최적화된 표준 방법론이 없다는 점이 한계로 꼽힌다.

토마스 메인의 말처럼 SOA는 죽었지만 '서비스 지향' 사상은 계속적으로 진화하고 있다. 이는 마이크로서비스로 이어진다.

3. 마이크로서비스 아키텍처(Microservice Architecture)

1) 마이크로서비스의 주요 특징

마이크로서비스는 지난 몇 년 동안 이전에는 거의 불가능했던 민첩성과 편리성을 지니는 아키텍처로 급부상했다. Nginx의 통계에 따르면 대기업의 36%가 현재 마이크로서비스를 이용 중이며, 26%는 도입을 검토하고 있음을 확인할 수 있다(NGINX n.d.).

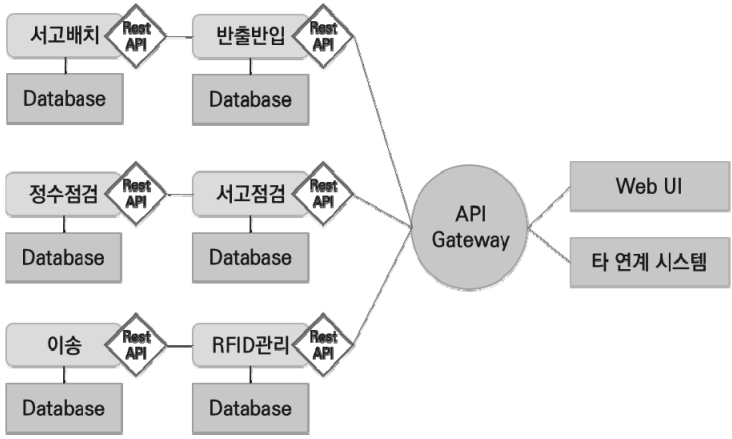
마틴 파울러(Martin Fowler)에 따르면 MSA는 "단일 애플리케이션을 작은 서비스 모음으로 개발하는 접근 방식으로서 작은 서비스는 자체적으로 실행할 수 있고, 경량 메커니즘을 통해 서로 통신한다. 이런 서비스는 비즈니스 기능을 기반으로 구축되며, 독립적으로 자동화하여 배포될 수 있다. 서로 다른 프로그래밍 언어로 작성되고 다른 데이터 저장 기술을 사용할 수 있기 때문에 이 서비스의 중앙 집중식 관리는 거의 필요하지 않다(Martin Fowler 2014)."고 설명했다.

마이크로서비스는 첫째, 이름에서도 알 수 있듯이 '작은 서비스'를 지향한다. 작은 서비스들이 모여 비즈니스 단위를 이루며 작게 분할된 서비스는 독립적인 팀 조직으로 연결된다. 예를 들면 기존 모놀리스 방식에서는 서고관리라는 하나의 대메뉴 안에 포함되었던 서브 기능들이 서고배치, 반출반입, 정수점검 등 각기 독립적으로 모듈화 되며, 이 서비스들은 각각의 독립적인 팀 조직이기도 한 것이다. 이렇게 규모가 작은 서비스는 더 이상

기능이 필요가 없어지거나 시간이 지나 레거시 시스템으로 바뀔 때 타 서비스로의 교체 및 삭제가 수월하며, 개발 및 운영자는 유지보수가 수월하다.

〈그림 4〉는 영구기록관리시스템의 서고관리 기능에 대한 마이크로서비스 아키텍처의 개략적인 구성 예시이다. 각 서비스는 REST API를 통해 통신하며, 클라이언트(Web UI)는 API Gateway를 통해 각 서비스들의 데이터를 전달받아 화면에 출력한다. 각각의 마이크로서비스들은 자체 데이터베이스를 보유하며 크기는 작지만 각각의 서비스는 하나의 애플리케이션 형태로 작은 모놀리스 형태와 유사한 구조를 갖는다. 마틴 파울러의 말처럼 하나의 큰 대형 애플리케이션을 작은 서비스로 분리한 형태의 구조를 취한다.

〈그림 4〉 '서고관리' 기능에 대한 마이크로서비스 아키텍처(예시)



둘째, 강력하고 효율적인 모듈화 개념을 제공한다. 기존 모놀리스 환경에서는 관련 없는 기능들 간의 의존성으로 한 가지 기능에 대한 개선작업이라 하더라도 다른 여러 기능에 발생 가능한 영향도를 체크해야만 했다. 반면 마이크로서비스는 분산 통신을 통해서만 다른 마이크로서비스를 호출

할 수 있기 때문에 마이크로서비스 사이의 의도하지 않은 의존성이 발생할 가능성이 적다. 또한 특정 서비스에 장애발생 시 모듈(서비스)간의 독립성으로 인해 해당 서비스에만 제한적으로 영향을 미친다.

셋째, 각 서비스는 독립성을 지닌다. 각 서비스는 자체적인 데이터베이스(개별 또는 공유된 데이터베이스에서 분리된 스키마)를 가지며, 다른 서비스와의 의존성이 적다. 하나의 서비스는 기능적으로 응집되며, 단 한 가지 목적에 집중된 서비스이므로 코드는 단순 명확해지고, 오류발생 확률도 최소화된다(박성훈 2018, 59). 기존 모놀리스 시스템에서는 타 기능과의 의존성으로 추가 개발이 힘들고, 배포 시에도 전체시스템을 대상으로 해야만 했다. 그러나 마이크로서비스에서는 변경이 필요한 서비스만 독립적으로 기능 개선이 가능하며, 필요한 서비스만 배포가 가능하다. 또한 부하가 집중되는 특정 기능 때문에 전체 애플리케이션을 대상으로 스케일 아웃(Scale out)[4]할 필요가 없으며 각 서비스의 특성에 맞게 자원을 할당할 수 있어 효율적인 자원 활용이 가능하다.

넷째, 각 마이크로서비스는 서로 다른 기술로 구현할 수 있는 유연성을 제공한다. 마이크로서비스는 자유롭게 언어 선택이 가능하며 다양한 플랫폼에서 구현 가능하다. 예를 들어 서고배치 서비스는 자바(java)를 기반으로, 반출반입 서비스는 파이썬(python)을 기반으로 구축이 가능하다. 또한 데이터베이스도 각 서비스가 가지는 특성을 고려(NoSQL, RDBMS 등)하여 자유롭게 선택할 수 있다.

다섯째, 마이크로서비스는 네트워크를 통해 통신한다. 이를 위해 마이크로서비스는 REST나 메시징(messaging) 같은 느슨한 결합(Loosely Coupled)을 지원하는 프로토콜을 사용한다(Eberhard Wolff 2016, 17).

---

4) 스케일 아웃이란 접속된 서버의 대수를 늘려 처리 능력을 향상시키는 것이다. 수평 스케일로 불리기도 한다. 전형적으로는 웹 서버 펌으로서 사용되고 있는 랙 마운트 서버군에 서버를 추가하거나 브레이드 서버에 브레이드를 추가하는 것 등이다(ZDNet Korea 2006).

이러한 기술적인 특징 외에 주목해야할 사항은 MSA가 해당 조직 및 비지니스적인 측면의 변화를 수반한다는 것이다. 앞서 언급한 바와 같이 대규모 프로젝트팀(모놀리스 방식)의 경우 시스템만큼 복잡한 팀 구조를 수반하며, 각 팀 사이의 소통에 오버헤드가 발생할 수 있고, 프로젝트 팀 간의 경계에는 방치되는 영역이 발생할 수 있다. 마이크로서비스는 이러한 문제를 해결하기 위한 아키텍처 수준의 접근방법이다. 마이크로서비스의 작은 팀[5]들은 독립적으로 프로젝트를 진행할 수 있어 의사결정이 빠르며, 독립적인 테스트 진행으로 서비스 품질이 증가한다. 한 팀의 속도가 느려지거나 방해물을 만나는 경우에도 다른 팀에게 거의 영향을 주지 않아 프로젝트 위험이 감소된다(Eberhard Wolff 2016, 95).

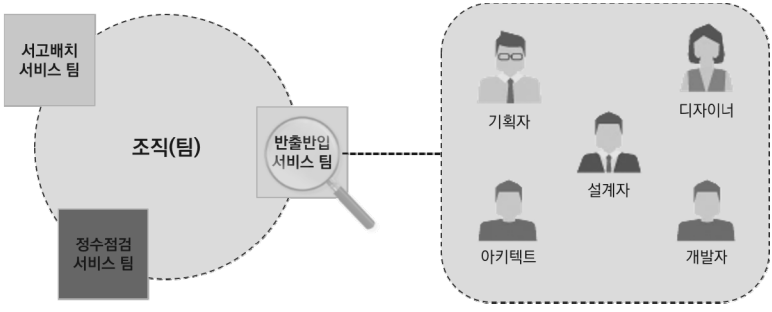
또한 이러한 팀 구조는 데브옵스(DevOps) 사상과도 연결된다. 데브옵스란 말 그대로 개발(Development)과 운영(Operation)의 합성어로서, 소프트웨어 개발자와 정보기술 전문가 간의 소통, 협업 및 통합을 강조하는 개발 환경이나 문화를 말하며 제품과 서비스를 단시간에 개발 및 배포하는 것을 목적으로 한다(조지훈 2018, 43).

서비스를 만들어 배포하고 운영하기 위한 팀 구성은 조직의 크기, 업무의 성격, 비용 등의 이유로 다양한 형태로 구성된다. 데브옵스 측면에서는 개발에서 운영까지를 하나의 파이프라인으로 형성하여 소스 코드의 배포가 필요할 때 즉시 반영되는게 목표이다. 이를 위해서 가장 이상적인 팀 구성 모델은 서비스의 기획, 설계, 개발, 배포 및 운영까지 서비스 생명주기(lifecycle)가 한 팀으로 수행되는 것이다. 이렇게 된다면 의사소통과 의사결정을 위한 노력과 비용을 절감할 수 있고, 원하는 시점에 즉시 서비스를 배포할 수 있어 비즈니스 변화에 빠르게 대응할 수 있는 이점이 있다(박성훈 2018, 40).

---

5) 스크럼 가이드에 따르면 3~9명의 팀 크기를 권장하고 있다(scrumguides n.d.).

〈그림 5〉 데브옵스의 팀 구성(예시)



실제 아콘소프트의 자료에 따르면 컨테이너 기술과 데브옵스 운영방식을 이용했을 때, 개발에서 테스트/검증, 운영까지 애플리케이션 배포속도는 160% 향상하며, 서버 자원 사용량은 80% 감소하고, 개발과 테스트, 운영 담당자 간 커뮤니케이션 비용은 약 70%가 감소하는 것으로 나타났다(Acornsoft 2018, 10).

〈표 1〉 컨테이너 기반 데브옵스의 이점

구분	효과
· 응용 프로그램 배포 속도 (개발 → 테스트 / 검증 → 운영)	· 배치 속도 160% 향상(5일 → 2일)
· 서버 리소스 사용 (클러스터 리소스 통합관리)	· 서버 자원 사용량 80% 감소
· 서버 운영관리	· 관리 작업 90% 감소
· 롤백/복구 속도	· 300% 빠른 회복(30분 → 10분 미만)
· 개발, 테스트, 운영담당자 커뮤니케이션 비용	· 비용 70% 절감

※출처 : Cocktail Cloud Use Cases White Paper (Acornsoft, 2018)

이처럼 마이크로서비스 측면에서 데브옵스의 팀 구성은 '자율적 조직'을 지향하고 있으며 각 팀은 서비스 출시 및 배포, 기술의 선택에 있어 독립성

과 민첩성을 가질 수 있어 효율적인 팀 구조를 확보할 수 있다.

2) 마이크로서비스 아키텍처 구축 시 고려사항

우리 조직에 마이크로서비스가 적합할까? 수잔 파울러(Susan Fowler)는 인포큐와의 인터뷰에서 “대부분의 중소기업은 마이크로서비스 아키텍처를 채택하는 게 반드시 도움이 되지는 않을 것이며, 대부분의 회사는 기존 시스템을 마이크로서비스로 전환하지 않아야 한다.”는 자신의 견해를 전했다. 그녀에 따르면 마이크로서비스가 정말 잘 작동하는 몇 가지 사례는 시스템이 다소 복잡하지만 많은 기능 간에 매우 명확한 경계 구분이 가능한 시스템이거나, 응용 프로그램이 더 이상 확장성이 없는 한계 지점에 도달하고, 그 한계로 인해 심각한 성능 및 안정성 문제가 발생하여 응용 프로그램 및 개발자의 생산 속도가 더는 효율적이지 않을 때이다(InfoQ 2017). 그녀의 견해는 “마이크로서비스는 복잡한 시스템에서만 유용하다.”(Martin Fowler 2014)는 마틴 파울러의 견해와 “해당 분야를 제대로 이해하지 못해 적절한 경계를 찾는 것이 어렵다면 마이크로서비스를 사용하지 말아야 한다.”는 샘 뉴먼과의 견해와도 일치한다(Sam Newman 2017, 327-328). 다시 말해 아직 한계에 도달하지 않았고, 시스템의 명확한 경계를 찾을 수 없다면, 마이크로서비스로의 전환은 필요 없다는 것이다.

이처럼 먼저 기관은 MSA가 자신의 기관에 적합한지 여부를 먼저 판단할 필요가 있다. 특히 새롭게 만들어지는 시스템의 경우 처음부터 MSA를 적용하는 것은 경계해야 한다. 마틴 파울러는 MSA를 잘 사용하는 조직에 있어 공통적인 패턴을 발견했는데 첫 번째는 MSA 적용이 성공한 거의 모든 조직의 시작은 모놀리스 아키텍처였다는 점, 두 번째 특징은 처음부터 MSA를 기반으로 구축한 조직의 시스템들은 거의 모든 경우 심각한 문제가 발생했다는 점이다. 마틴 파울러는 이 패턴 때문에 응용 프로그램을 충분히 가치

있게 만들 수 있다고 확신하더라도 MSA로 새 프로젝트를 시작해서는 안 된다는 동료들의 입장을 전했다(Martin Fowler 2015).

이처럼 단지 신기술이라는 이유로 마이크로서비스를 도입하는 것은 매우 위험한 시도일 수 있다. 복잡한 애플리케이션을 소규모의 간단한 서비스로 분해해도 서비스를 실행하고 관리하는데 필요한 아키텍처는 몇 가지 큰 문제에 부딪힐 수 있다. 서비스는 모두 함께 동작해야 애플리케이션의 전체 기능을 제공할 수 있기 때문이다(Boris Scholl 외 2016, 45).

이번 장에서는 서론에서 언급한 "마이크로서비스가 미래를 주도할 수 있는가?" 라는 물음에 마틴파울러가 유보적인 입장을 취했던 세 가지 측면에 대해 논의하고자 한다.

(1) MSA의 짧은 역사

아마존 최고기술책임자(CTO) 베르너 보겔스(Werner Vogels)는 아마존의 성장을 이끌어 낸 기술 선택의 빠른 진화에 대해 2006년 6월 컴퓨터 잡지인 〈ACM Queue〉와의 인터뷰에서 다음과 같이 말했다. "아마존닷컴 기술 진화의 많은 부분은 끊임 없는 성장을 통해, 가용성과 성능을 유지하면서도 초확장성(ultra scalability)을 확보할 수 있는 방향으로 추진되었습니다." 보겔스는 아마존이 초확장성을 확보하려면 소프트웨어 아키텍처 스타일을 다르게 가져가야 할 필요가 있었다고 말한다. 아마존닷컴은 모놀리스 애플리케이션 스타일로 시작했지만, 시간이 지남에 따라 운영에 점점 더 많은 팀이 관여하게 되었으며, 이로 인해 코드베이스의 책임 한계와 소유권이 불투명해지기 시작했다. 그리고 "개발팀은 자신이 빌드한 것을 소유해야 한다."는 생각을 전했다. "만들면 운영까지" 이것이 아마존의 모델이다(Josh Long · Kenny Bastani 2018, 42). 오늘날 보겔스의 이러한 생각은 '데브옵스(DevOps)'로 불리며, 이와 같은 아키텍처는 '마이크로서비스'의 사상과 일치한다.

이처럼 일부 기업에서는 MSA라는 용어가 정착되기 이전부터 비슷한 방법론을 사용하고 있었다. 이후 2011년 5월 베니스에서 개최된 소프트웨어 아키텍처의 워크숍에서 “마이크로서비스”라는 용어를 최초로 사용하였다. 제임스 루이스는 2012년 3월, 33회 Degree in Kraków in Microservices - Java, the Unix Way에서 사례 연구로서 이 아이디어 가운데 일부를 발표했으며(위키백과 n.d.), 2014년 제임스 루이스와 마틴 파울러가 공동으로 발표한 칼럼에 의해 본격적으로 알려지기 시작했다.

MSA는 종종 SOA와 비교 되는데 큰 의미에서 보자면 MSA는, SOA에서 정의한 서비스 중에서 ‘fine grained’ 서비스로 정의되는 하나의 종류이며, API Gateway역시 SOA에서 정의한 ESB의 하나의 구현방식에 불과하다(조대협 2014). 이처럼 MSA는 SOA에서 발전된 SOA의 최신버전으로 보는 시각(Adam Bertram 2017, Maria Korolov 2017, 조대협 2014, Boris Scholl 외 2016, 재인용)과, SOA와 다를 게 없는 마케팅 용어에 불과하다고 폄하하는 시각(NGINX 2015)이 존재한다. 그렇다면 왜 SOA가 아닌 다른 이름으로 불리고 있는가? 이에 대해 빅토르 파르시트(Viktor Farcic)는 ESB 제품의 출현으로 SOA는 잘못된 방향으로 도달했으며, 마이크로서비스 운동은 어떤 점에서 SOA에 대한 오해와 모두가 시작했던 곳으로 되돌아가려는 의도에 대한 반응이라 말한다(Viktor Farcic 2017, 55).

어쨌든 분명한 것은 그 시작점이 SOA라 하더라도 MSA는 10년이 채 안되는 짧은 역사를 가지고 있고, 이제 막 부상하기 시작한 아키텍처라는 점이다. 비록 현대 조직에 적합한 아키텍처이며, 지금까지는 모놀리스 아키텍처의 근본적인 문제점들을 해결하는데 긍정적인 아키텍처 스타일로 평가받고 있다 하더라도 아직까지 충분한 시간이 지나지 않았다는 점 또한 인지하여야 한다. 아키텍처는 구축이 완료되고, 몇 년 후에나 진정한 평가가 가능하기 때문이다(Martin Fowler 2014).

(2) 기술의 성숙도

MSA는 분명 이점도 많지만 아직 기술적으로 해결해야 할 문제도 산적해 있다. MSA로 시스템 구축 시 고려해야 할 기술적인 사항을 살펴보면 첫째, 마이크로서비스는 분산 시스템(distributes system)이라는 것이다. 마이크로서비스 사이의 호출은 네트워크를 통해 이루어지기 때문에 마이크로서비스가 많으면 많을수록 지연 시간에 영향을 주며, 응답시간은 빠르지 않다(Eberhard Wolff 2016, 99). 이것은 단순히 통신 속도를 개선하기 위한 개발이 추가되어야 하며, 네트워크 최적화가 반드시 선결되어야 한다는 뜻이다.

기존 모놀리스 아키텍처에서는 주로 3계층(3-tier) 아키텍처를 사용한다. 시스템 성능이나 개발, 유지 보수의 효율 개선을 위해 클라이언트/서버의 응용 프로그램 구조를 표현(presentation), 응용(application), 데이터(data) 등 3개의 논리적 기능 모듈로 나눈 형태(네이버 백과사전 n.d.)로 클라이언트가 특정 요청을 전송하면 애플리케이션 영역은 로직을 실행하여 데이터베이스에서 요청한 데이터를 불러와서 UI를 통해 사용자에게 출력하는 단순한 구조를 지닌다.

그러나 마이크로서비스 아키텍처에서는 상황이 다르다. 〈그림 4〉와 같이 모든 클라이언트의 요청은 API Gateway를 통해서 처리된다[6]. 따라서 클라이언트는 API Gateway로 요청하고, API Gateway는 받은 요청을 필요한 마이크로서비스에 요청하여 결과 값을 받아 다시 클라이언트로 보내게 된다. 그러나 이런 호출이 단순하게만 이뤄지지 않는다는 점에서 문제가

6) API Gateway란 모든 클라이언트 요청에 대한 end point를 통합하는 서버로서 인증 및 권한, 모니터링, logging 등의 기능이 있다(우아한 형제들 기술블로그 2017). MSA에서 API Gateway를 도입하는 것은 선택사항일 수 있지만 본 연구에서는 API Gateway를 도입하였다는 가정 하에 서술하였다. 하지만 API Gateway를 도입한다는 것은 유지보수해야 할 장비가 하나 더 추가되었다는 의미이며, API Gateway에 장애 발생시 MSA의 모든 서비스의 장애로 이어질 수 있으며, 각 서비스의 API를 수정하면 API Gateway 또한 수정해야 하는 단점이 있다.

발생한다.

예를 들어, 서고배치 UI화면에서 해당 사용자가 소속된 기록관 서고의 정보들을 자동으로 출력하려면 클라이언트는 API Gateway로 관련 내용을 요청하고 API Gateway는 '사용자관리' 서비스에 해당 사용자의 소속기관이 어디인지 관련 정보를 요청한 후 받은 결과 값을 기반으로 다시 '서고배치' 서비스에 해당 소속기관의 정보를 요청해서 받아야 한다. 모놀리스 아키텍처에서는 하나의 요청이 마이크로서비스에서는 여러 서비스에 다수의 요청이 될 수 있으며[7], 이런 요청관계가 많아질수록 속도는 지연되며, 시스템은 복잡해진다.

둘째, 마이크로서비스 간의 리팩토링(Refactoring)[8]은 어려움이 따른다. 하나의 마이크로서비스는 그 크기가 작으므로 리팩토링이 간단하다. 그러나 마이크로서비스 사이에서는 상황이 다르다. 한 마이크로서비스에서 다른 마이크로서비스로 기능을 이전하는 것은 매우 복잡한 작업이며, 심지어 마이크로서비스간에는 기술 및 프로그래밍 언어도 다를 수 있다. 이에 따라 리팩토링의 경우, 기능은 새로운 마이크로서비스로 이동되어야 하며, 어떤 경우에는 반드시 다른 마이크로서비스의 기술로 새로 구현되어야 하며 그 이후에 해당 마이크로서비스로 이전되어야 한다(Eberhard Wolff 2016, 106).

셋째, 마이크로서비스간의 통신은 네트워크를 통해 이루어지기 때문에 신뢰할 수 없다. 장애가 발생할 가능성을 염두한 시나리오를 준비해야 하며, 준비된 시나리오에 따라 장애를 신속하게 감지하고 대응할 수 있어야 한다. 예를 들어 RestTemplate으로 통신하는 서버가 통신장애로 응답하지 못하는 상황이 발생하면 이를 타임아웃 기능[9]을 적용하여 경로를 변경하는

7) 통신 방식은 호출하는 쪽과 호출을 받는 쪽의 수로 구분할 수 있으며 이에 따라 일대일(one-to-one), 일대다(one-to-many)로 나뉘며, 동기(요청을 보내고 기다리는 방식), 비동기(요청을 보내고 다음 프로세스 실행)방식으로 구분할 수 있다.

8) 리팩토링은 코드의 동작이나 의도는 유지하면서 코드의 구조, 재사용성, 가독성을 수정하여 전체 디자인을 개선하는 방법을 말한다(Martin Fowler 2012).

등의 조치가 필요하다.

넷째, MSA와 같은 분산시스템에서 모니터링 시스템 구축은 어려움이 따른다. 모놀리스 시스템에서는 장애가 발생한 경우분석을 시작할 수 있는 명확한 출발점이 있다. 또한 시스템의 로그를 분석하여 각종 에러에 대응할 수 있다. 하지만 MSA에서는 상호의존성이 있으므로 때로는 매우 어려운 작업이 될 수 있다.

마이크로서비스는 일반적으로 경량급 JSON, REST API를 통해 서로 통신하는데 경량급 API는 더 유연하고 확장하기 쉽지만, 모니터링이 필요한 새로운 인터페이스를 추가했을 때 문제를 일으키거나 버그를 유발하기도 한다(Lucas Carlson 2017, 5).

결과적으로 모든 마이크로서비스를 포함해서 더욱 세심하게 모니터링 해야 한다. 운영체제로부터의 일반적인 정보뿐 아니라 하드디스크와 네트워크에 대한 I/O도 분석해야 하며, 애플리케이션 메트릭에 기반한 애플리케이션 관련 정보도 볼 수 있어야 한다(Eberhard Wolff 2016, 108).

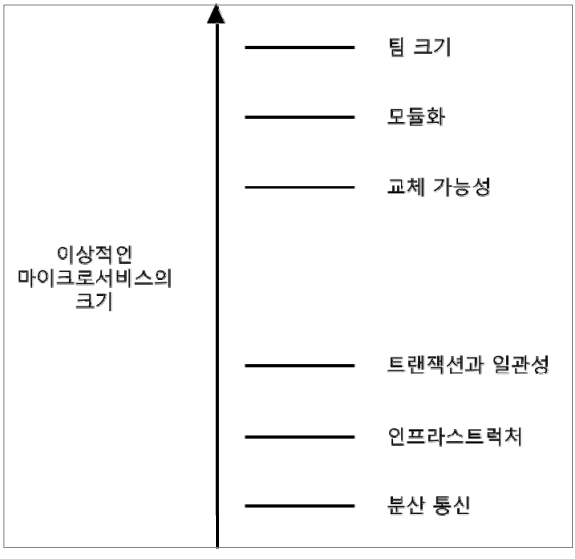
다섯째, MSA는 작은 서비스를 지향하고 있음을 살펴본 바 있다. 그렇다면 얼마나 작아야 하는지에 대한 의문이 따른다. 이에 대해 아마존 CEO 제프 베조스(Jeffrey Bezos)는 피자 두판의 법칙[10]을, 존 이브스(Jon Eaves)는 '2주 안에 재작성될 수 있는 것', 샘 뉴먼(Sam Newman)은 '충분히 작아서 더 이상 작아질 수 없는 크기' 라고 말하고 있다(Sam Newman 2017, 29). 결론적으로, 마이크로서비스의 크기의 상한에는 정답이 없으며, 일반적으로 다음 요소들로부터 영향을 받는다(Eberhard Wolff 2016, 60-61).

---

9) 일정시간 동안 서비스 요청에 대한 반응이 없을 경우 기존 요청 경로를 차단하고, 다른 경로로 요청 경로를 변경하는 기능(박성훈 2018, 16)

10) 팀원의 수가 피자 두 판으로 식사를 마칠 수 있는 규모 이상이 되어서는 안된다는 것(FASTCOMPANY 2015).

〈그림 6〉 마이크로서비스의 크기에 영향을 주는 요소들



※출처 : 마이크로서비스-유연하고 확장 가능한 소프트웨어 아키텍처(Eberhard Wolff, 2016)

1. 팀의 크기가 상한을 결정한다. 마이크로서비스는 결코 여러 팀의 작업이 필요할 만큼 커져서는 안된다.
2. 더 많은 모듈화는 마이크로서비스의 크기를 제한한다. 마이크로서비스는 개발자가 이해하고 더 개발하는 것을 허용하는 크기가 바람직하다.
3. 교체 가능성은 마이크로서비스의 크기를 감소시킨다. 따라서 교체 가능성은 마이크로서비스의 크기의 상한에 영향을 준다.
4. 마이크로서비스 크기의 하한은 인프라스트럭처에 의해 설정된다. 마이크로서비스에 필요한 인프라스트럭처를 제공하기가 힘든 경우 마이크로서비스의 개수는 더 적은 수로 유지돼야 한다.
5. 분산통신은 마이크로서비스의 개수에 따라 증가한다. 이러한 이유로, 마이크로서비스의 크기는 너무 작게 설정되어서는 안된다.

6. 데이터의 일관성과 트랜잭션은 오직 하나의 마이크로서비스 내에서만 보장된다. 따라서 마이크로서비스는 일관성과 트랜잭션에 여러 마이크로서비스가 관련될 수 있을 정도로 너무 작아서는 안 된다.

이러한 요소들을 고려하여 마이크로서비스의 크기를 설정해야 하며, 마이크로서비스의 고유한 이점을 해치지 않는 범위 안에서 고려해야 한다. 사실 서비스 경계를 정의하는 일은 MSA에서 가장 도전적인 작업 중 하나일 수 있고, 비즈니스 도메인을 잘 이해해야 하는 일이다(Boris Scholl 외 2016, 36). 이와 관련하여 샘 뉴먼은 자신의 저서에서 너무 성급하게 마이크로서비스로 분리해 어려움을 겪었던 경험을 언급했다. “프로젝트팀이 내린 서비스 경계에 대한 초기 견해가 맞지 않아 결국 팀은 서비스를 다시 단일 모놀리스로 합친 후, 1년 뒤 해당 모놀리스 시스템을 다시 마이크로서비스로 분리하였고, 이로 인해 서비스 간의 수많은 변경과 그에 따른 높은 비용 소모를 초래했다”고 밝혔다(Sam Newman 2017, 69).

이런 상황을 방지하기 위해선 프로젝트팀은 실제 서비스 설계단계에서 마이크로서비스의 크기 및 서비스 식별전략에 대한 구체적인 계획을 세운 후 접근해야 한다. 최대한 의존관계를 잘 분리할 수 있도록 단순하게 설계해야 함과 동시에 사용자 경험(User Experience)을 함께 고려해야 한다.

여섯째, 트랜잭션(transaction)의 정합성을 맞추기 위한 노력을 해야 한다. 데이터베이스는 다수의 사용자가 동시에 사용하더라도 항상 정확한 데이터를 유지해야 한다. 이를 위해 데이터베이스 관리 시스템(DBMS)은 트랜잭션을 관리함으로써 일관된 상태를 유지할 수 있도록 한다.

그러나 MSA에서는 서비스별로 각각의 DB를 사용하며 특성에 따라 DB 종류도 다를 수 있다. 게다가 API로 통신하는 여러 마이크로서비스에서 트랜잭션의 정합성을 맞추는 것은 어려운 작업이 될 수 있다. 이런 상황에선 각 서비스별로 중복되는 데이터도 생길 수 있으며, 한 마이크로서비

스에서는 변경이 일어났지만 다른 마이크로서비스에서는 변경이 되지 않는 경우도 생길 수 있다. 이러한 문제를 해결하기 위해 보상 트랜잭션(compensating transaction)을, 분산된 데이터 조회를 위해서는 이벤트 주도 아키텍처(event-driven architecture)를 적용 하는 등의 방법(NGINX 2015)으로 해결책을 모색해야 한다.

이외에도 MSA는 테스트가 힘들다는 점, 버전관리, 배포 자동화 구현의 어려움 등의 문제가 발생할 수 있다. 결과적으로 MSA의 기술적인 문제점은 기존 단일체 구조(모놀리스 아키텍처)를 작은 단위(마이크로서비스)로 나눠서 발생한다. 추가로 기술의 다양성을 추구할 수 있다는 장점이 오히려 관리의 어려움을 증가시킨다.

(3) 프로젝트 팀의 기술 수준

마틴 파울러는 “기술적인 숙련도가 부족한 팀은 항상 낮은 수준의 제품만 만들며, 이러한 팀이 마이크로서비스를 적용했을 때의 결과는 더 안 좋을 수 있다.”고 말한다. IT 컨설턴트 조대협도 “충분한 능력을 가지지 못한 팀이 MSA로 시스템을 개발할 경우에는 많은 시행착오를 겪을 수 있다”고 경고한다(조대협 2014).

이처럼 MSA를 적용하고자 한다면 프로젝트 팀의 기량과 경험을 고려해야 한다. 이제 막 마이크로서비스가 적용되기 시작한 우리나라에서는 MSA에 대한 이해도나 경험이 부족하기 때문에 더욱 경계해야 할 문제이다. 우리나라에선 아직까지 공공영역에서 마이크로서비스를 적용한 사례는 극히 드물며 민간 기업에서도 쿠팡, 11번가, 삼성 IOT 서비스, 우아한 형제들 등 일부 대기업 중심으로 서서히 변화가 일어나고 있는 중이다.

그렇다면 유독 공공부문에서만 마이크로서비스의 도입이 늦어지고 있는 이유는 무엇일까? 신기술에 대한 검증이 끝나지 않은 상태여서 보수적인 대응방식의 문제도 있겠지만 공공부문의 발주방식도 MSA의 적용을 어렵게

한다. 특히 프로젝트의 '짧은 기간'은 기술의 다양성 및 신기술의 적용을 더욱 힘들게 하는 요인이다. 마이크로서비스는 처음부터 꼼꼼한 설계가 필수인데 짧은 기간에 프로젝트를 진행하다 보니 제대로 된 설계 기간을 확보하기 힘들고, 결국 단기간에 성과를 낼 수 있는 모놀리스 방식을 떠밀리듯 채택할 수밖에 없는 것이다(장민 2017, 3).

4. 결론 : 기록관리시스템 MSA 적용 시 고려사항

국가기록원은 2017년에 진행한 '차세대 기록관리모델 재설계 연구' 사업과 2018년 '전자기록관리 고도화 BPR/ISP 사업'을 통해 차세대 기록관리모델 기반을 마련하였으며, 이를 바탕으로 현재의 방대한 기능을 세분화(모듈화)하여 단위별 변경이 가능한 '다기능·연계형 복합시스템'으로의 전환을 모색 중이다(이승억 2017).

'차세대 기록관리모델 재설계 연구' 사업의 결과에 따르면 차세대 기록관리시스템에 기록관리 본연의 기능만을 위한 보존측면의 '보존층'(Preservation Layer)과 실제 기록관 업무에 초점을 맞춘 활용측면의 '업무층'(Business Layer)으로 구분하였다. 이러한 구분으로 기관 특성에 맞게 차세대 기록관리시스템을 맞춤형으로 구성이 가능하다(국가기록원 2017a, 210-211). 이를 통해 중앙집중형 개발과 배포의 한계를 벗어나 다양성이 존재하는 기록 생태계를 구축하는 모델을 지향하고 있다(국가기록원 2017b, 152).

그러나 기술의 다양성 및 독립성·자율성의 장점은 관리나 운영의 오버헤드를 증가시킬 수 있다. 기존 기록관리시스템이 가지는 단점을 극복하고자 선정한 방식이 오히려 더 큰 문제를 일으키지 않도록 세밀한 준비를 해야 한다. 따라서 기록관리시스템을 마이크로서비스 적용 시 고려해야 할 몇 가지 사항[11]을 제시하며 결론을 대신하고자 한다.

첫째, 시스템 구축 이전부터 유지보수 및 운영의 효율성을 고려할 필요

가 있다. 마이크로서비스는 다양한 타입의 인터페이스와 상호의존성을 갖으며, 마이크로서비스의 수만큼 배포작업이 필요하기 때문에 운영의 오버헤드가 발생할 수 있다. 초기 설계 단계에서 운영의 효율성을 고려한 설계로 서비스를 사용하는 기관들의 운영에 대한 부담이 높아지지 않도록 유념해야 한다. 특히 각 마이크로서비스를 어떻게 배포(Deploy)할 것인가와 전체 서비스를 자동으로 모니터링 할 수 있는 툴의 선택은 신중을 기해야 한다.

둘째, 시스템 구축 후 조직구성에 대한 가이드를 제시할 필요가 있다. 마이크로서비스 아키텍처는 하나의 서비스가 곧 조직(팀)으로 연결되기 때문에 조직에 직접적으로 영향을 줄 수 있다(Eberhard Wolff 2016, 361). IT시스템에 따라 새로운 팀이 생성될 수 있으며, 이런 경우 시스템 구조는 조직의 부서 구조에 직·간접적으로 영향을 미친다. 그러나 국가기록원이 중앙에서 개발하여 배포하는 방식은 각 기관의 업무 프로세스와 조직구성을 모두 수반할 수 없다. 따라서 시스템 개발이 완료되면 그 결과에 맞춰 효율적인 조직구성에 대한 가이드를 제시할 필요가 있다. 물론 기관의 규모와 형태, 특성에 따라 달라질 수 있겠지만 기존 조직이나 새롭게 조직되는 기관에서는 큰 도움이 될 수 있다.

셋째, 공공사업의 경우 개발팀과 운영팀이 다를 수 있다는 것을 유념해야 한다. '차세대 기록관리모델 재설계 연구' 사업에서도 데브옵스 기반의 애자일 프랙티스의 이점을 제시하였으며(국가기록원 2017a) 많은 연구에서도 마이크로서비스 아키텍처는 데브옵스 기반의 운영방식을 함께 해야 함을 언급하고 있다.

하지만 대부분의 국내 기록관리 기관에서는 IT 관련 직군(기획, 개발, 운영 등)은 존재하지 않아 데브옵스 조직 구성은 현실적으로 힘들다. 유지보수 사업자로 조직을 구성하려 해도 민간기업과 달리 공공사업은 입찰 결과

11) 본 연구의 제안 사항은 소규모 기록관이 아닌 영구기록물관리기관을 대상으로 하였다.

에 따라 사업자가 선정되고, 예산확보의 어려움으로 개발 및 운영에 최적화된 데브옵스 조직 구성은 힘들 수 있다. 최대한 데브옵스 방식의 이점을 살릴 수 있는 방안 모색이 필요하다. 하나의 안으로 공무원 조직과 유지보수 업체의 인력을 활용하여 데브옵스 조직을 구성할 수 있다. 그동안은 IT부서를 중심으로 각 부서와의 협의를 통해 시스템 유지보수 및 운영을 진행했었다면, 그 오너쉽을 각 담당부서가 가져가고 IT부서의 협조를 얻는 체계로 바꾸는 것이다.

각 팀이 맡은 업무를 중심으로 하나(혹은 여러개)의 마이크로서비스를 담당(기획)하고, 유지보수 업체의 인력(개발 및 운영)을 팀에 포함시켜 운영조직을 꾸리는 것이다. 이렇게 구성된 운영팀은 자신의 서비스에 책임감을 갖고 방법론에 따라 효율적인 방안을 계속적으로 모색할 수 있다. 또한 이러한 방식은 큰 조직 내의 한명의 일원으로서 수동적인 역할이 아니라 자신이 주체가 되어 업무를 주도적으로 이끌어 나갈 수 있는 원동력을 제공할 수 있다.

넷째, MSA에서는 각 서비스의 특성에 맞는 언어 및 기술, Database의 종류도 다르게 선택할 수 있다는 점을 이미 언급한 바 있다. 그러나 이런 이점은 전체 시스템 관리의 어려움이 증가함을 의미한다. 언어별로 개발환경이나 배포체계도 달라져야 하며, 코드 중복의 가능성, 모니터링을 위한 시스템 구성에도 어려움을 겪을 수 있다. 또한 운영 및 유지보수 시 해당 언어에 대한 담당자 부재 혹은 이탈도 고려해야 한다. 그렇다면 이런 상황에서 마이크로서비스의 특성에 맞게 정말 '자유롭게' 선택이 가능할까? 현실적으로는 쉽지 않은 도전이 될 수 있다.

프로젝트팀은 개발 초기 단계에 프로젝트 개발 환경 및 향후 운영정책 등을 고려하고 현실적으로 폴리그랏 프로그래밍(Polyglot Programming)[12] 이 위에서 언급한 단점을 보완할 정도의 이점이 있는지 득실여부를 판단한

12) 2006년 닐 포드(Neal Ford)가 언급한 아이디어로 각 서비스의 목적과 특성에 맞게 언어와 기술을 자유롭게 사용하는 것을 말한다.

후 이에 대한 가이드라인을 설정해야 할 것이다. 물론 각 팀의 자율성을 해치지 않는 범위에서 설정되어야 하며, 이로써 무분별한 언어 선택에 대한 남용을 방지할 수 있다.

다섯째, 국내 개발업체의 기량과 경험을 고려해야 하며, SW 사업 발주 시 주관기관은 최대한 프로젝트 기간을 길게 확보할 필요가 있다. 마이크로서비스는 구축이 완료되더라도 실제 운영하면서 많은 문제가 발생할 수 있으므로, 안정화기간까지 고려하여 프로젝트 구축이 끝나면 구축업체에서 실제 운영까지 진행할 수 있어야 한다. 이를테면, '구축+운영'을 합친 계약을 검토할 필요가 있다.

마이크로서비스로 시스템을 전환하는 것은 어려운 작업이다. 사실 기술적인 성숙도나 국내의 현실로 보았을 때 아직은 시기상조일 수 있다. 그러나 보다 더 철저히 준비하고, 위험요소를 미리 파악하여 대처할 수 있다면 그렇게 어려운 일만도 아니다. 우리나라에서도 이제 막 MSA를 적용한 시스템에 대한 결과물들이 나오고 있다. 선행 사업들이 겪었던 경험과 결과물을 통해 위험요소 파악과 함께 보다 더 나은 서비스를 준비할 필요가 있다.

MSA를 적용한 기록관리시스템을 개발한다면 혁신의 기반과 함께 민첩성과 확장성을 확보할 수 있다. 데브옵스 조직까지 도입 될 수 있다면 서비스별 담당자가 명확해져 더 책임감을 갖으며 전반적인 서비스 품질 향상을 이룰 수 있다. 변화에는 언제나 리스크가 따르기 마련이다. 리스크 때문에 변화를 두려워한다면 어떤 것도 이룰 수 없다. 기존에는 당연시 여겼던 관례를 보다 더 유연하게 대응할 수 있는 방식을 적극 고려할 필요가 있다. 그런 면에서 서울기록원의 오픈소스 방식과 모듈화를 적용한 기록관리시스템 개발(안대진 · 임진희 2017)은 기존 중앙에서 개발과 배포를 통제하는 방식에서 벗어나 자율성과 유연성을 추구한다는 점에서 큰 의미가 있다. 이 연구가 MSA를 적용한 기록관리시스템을 개발하는데 도움이 되길 바란다.

〈참고문헌〉

강성원. 2010. 소프트웨어 아키텍처 설계의 근본 원리들. 『소프트웨어공학소사이어티 논문지』, 23(4), 125-139.

국가기록원. 2017a. 차세대 기록관리 모델 재설계 연구 결과보고서(2세부 연구과제).

국가기록원. 2017b. 차세대 기록관리 모델 재설계 연구 결과보고서(총괄, 1세부, 3세부 연구과제).

김기정, 신동수. 2018. 클라우드 컴퓨팅 환경 영구기록물관리 시스템 구축 방안 연구. 『한국기록관리학회지』, 18(3), 49-70.

김치수. 2015. 『쉽게 배우는 소프트웨어 공학』. 서울시 : 한빛아카데미.

네이버.n.d. 검색어 "3계층 구조", 검색일자 : 2019.4.15. https://terms.naver.com/entry.nhn?docId=862171&cid=42346&categoryId=42346

네이버.n.d. 검색어 "오버헤드", 검색일자 : 2019.4.15. https://terms.naver.com/entry.nhn?docId=2829829&cid=40942&categoryId=32828

박성훈. 2018. 『자바기반의 마이크로서비스 이해와 아키텍처 구축하기』. 경기도 : 제이펍.

보안뉴스. 2009. 정부, 온나라 시스템으로 행정효율 개선 가속할 것. 검색일자 2019.2.23. https://www.boannews.com/media/view.asp?idx=16279

아콘소프트. 2018. Cocktail Cloud Use Cases White Paper. 검색일자 : 2019.2.28. http://www.cocktailcloud.io/

안대진, 임진희. 2016. 디지털 아카이브 시스템 구축을 위한 공개 소프트웨어 활용방안 연구. 『정보관리학회지』, 33(3), 345-370.

안대진, 임진희. 2017. 기록시스템의 오픈소스화 전략 연구. 『기록학연구』, 52, 121-173.

오진관, 임진희. 2018. 차세대 기록관리시스템 재설계 모형 연구. 『한국기록관리학회지』, 18(2), 163-188.

우아한 형제들 기술 블로그. 2017. API GATEWAY－spring cloud zuul 적용기. 검색일자 : 2018.3.8. http://woowabros.github.io/r&d/2017/06/13/apigateway.html (2019.2.23.)

위키백과. n.d. 검색어 "마이크로서비스". 검색일자 : 2019.3.3. https://c11.kr/63vq

이상효, 양해술. 2009. SOA 소프트웨어의 사용성 평가 방법. 『한국산학기술학회논문지』, 10(7), 1575-1584.

이승억. 2017. 차세대 전자기록관리체계 재설계를 위한 방안 모색. 기록정책포럼 발표문.

임근원, 박해성, 김준희, 채병훈, 김주형, 이준범. 2018. DDD와 MSA로 쇼핑몰 구축하기. 『MICROSOFTWARE』. 394, 150-161.

장민. 2017. 마이크로서비스와 컨테이너로 완성하는 '클라우드 네이티브'애플리케이션 전략. ITWORLD IDG Summary, 1-8.

정도현. 2014. 마이크로서비스가 가져올 미래의 개발 패러다임. 검색일자 : 2019.2.24. http://www.moreagile.net/2014/10/microservices.html
조대협. 2014. 마이크로서비스 아키텍처(MSA의 이해). 검색일자 : 2019.2.24. https://bcho.tistory.com/search/msa의%20이해
조지훈. 2018. 더 웨더 컴퍼니의 데브옵스. 『MICROSOFTWARE』. 394, 42-49.
주현미, 임진희. 2017a. 차세대 전자기록관리 재설계 과제 연구. 『기록학연구』, 54, 151-178.
주현미, 임진희. 2017b. 차세대 전자기록관리 프로세스 재설계 연구. 『한국기록관리학회지』, 17(4), 201-223.
쿠팡기술블로그. 2018. 행복을 찾기 위한 우리의 여정, 쿠팡의 MSA-Part1. 검색일자 : 2018.02.22. https://c11.kr/63vt
Adam Bertram. 2017. 이제는 '마이크로서비스'가 대세. ITWORLD IDG Deep Dive(애플리케이션과 웹의 미래 마이크로서비스), 1-4.
Boris Scholl, Trent Swanson, Daniel Fernandez. 2016. 『Microservices with Docker on Microsoft Azure』, (김도균 역. 2017. 『Azure와 도커를 활용한 마이크로서비스 구현』, 서울 : 에이콘).
Eberhard Wolff. 2016. 『Microservices : flexible software architecture』, (김영기 역. 2016. 『마이크로서비스 : 유연하고 확장가능한 소프트웨어 아키텍처』, 서울 : 에이콘).
FASTCOMPANY. 2015. 3 Reasons Why Small Teams Make Better Tech Innovators. 검색일자 2019.3.5. https://c11.kr/63vr
InfoQ. 2017. Q&A with Susan Fowler on Production-Ready Microservices. 검색일자 : 2019.3.4. https://www.infoq.com/news/2017/01/production-ready-microservices
James Lewis, Martin Fowler. 2014. Microservices : a definition of this new architectural term. 검색일자 : 2018.02.12. https://martinfowler.com/articles/microservices.html
Jammes, F. and H. Smit. 2005. "Service Oriented Paradigms in Industrial Automation", 『Transactions on Industrial Informatics』, 1(1), 62-70.
Josh Long · Kenny Bastani. 2017. 『Cloud Native Java : Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry』, (정윤진, 오명운, 장현희 역. 2018. 『클라우드 네이티브 자바』, 경기도 : 책만).
Lucas Carlson. 2017. 마이크로서비스를 꼭 사용해야 하는 이유. ITWORLD IDG Deep Dive(애플리케이션과 웹의 미래 마이크로서비스), 5-11.
Maria Korolov. 2017. 매력적인 속도와 유연성엔 대가가 따른다. ITWORLD IDG Deep Dive(애플리케이션과 웹의 미래 마이크로서비스), 12-15.
Martin Fowler. 2015. MonolithFirst. 검색일자 : 2018.03.2. https://martinfowler.com/bliki/MonolithFirst.html

NGINX. 2015. Introduction to Microservices. 검색일자 : 2019.2.28. https://www.nginx.com/blog/introduction-to-microservices/

NGINX. n.d. The Future of Application Development and Delivery Is Now. 검색일자 : 2019.2.28. https://www.nginx.com/resources/library/app-dev-survey/

Sam Newman. 2015.『Building Microservices : Designing Fine-Grained Systems』, (정성권 역. 2017.『마이크로서비스 아키텍처 구축(대용량 시스템의 효율적인 분산 설계 기법』, 서울 : 한빛미디어).

SCRUM GUIDES. n.d. The Scrum Guide. 검색일자 : 2018.2.19. https://www.scrumguides.org/scrum-guide.html#team

Thomas Erl. 2005.『Service-Oriented Architecture : Concepts,Technology,and Design』, (장세영, 황상철, 이현정, 조문옥 역. 2006.『 SOA : 서비스 지향 아키텍처 개념에서 설계, 구현까지』, 서울 : 에이콘).

Thomas Manes. n.d. SOA is Dead; Long Live Services. 검색일자 : 2018.03.4. https://blog.naver.com/javava94/120064294751

Viktor Farcic. 2016.『The Devops 2.0 Toolkit : Automating the Continuous Deployment Pipeline with Containerized Microservices』, (전병선 역. 2017.『데브옵스 2.0 툴킷』, 서울 : 에이콘).

ZDNet Korea. 2006. 스케일 아웃과 스케일 업-서버 처리능력 향상의 길. 검색일자 : 2019.3.4. http://www.zdnet.co.kr/view/?no=00000039151294

ZDNet Korea. 2009. [SOA기획] 파티는 다시 시작되는가. 검색일자 : 2019.2.24. http://www.zdnet.co.kr/view/?no=20090824082522