공학
트렌드

소프트웨어공학센터 경영지원TF팀

품질고도화를 위한 실용적인 소프트웨어 아키텍처 리뷰

Part 2 : 프랙티컬 아키텍처 리뷰의 검토 내용과 사례

2014. 6. 10. [제95호]

Contents

정보통신산업진흥원 부설
nipa 소프트웨어공학센터

Ⅰ. 아키텍처 리뷰 프로세스

아키텍처 리뷰의 내용을 바탕으로 아키텍처 리뷰를 하기 위한 프로세스의 내용을 설명하고자 한다. 사내에서 수많은 솔루션들이 생성되면서 아키텍처 리뷰를 해야 하고, 이를 관리하는 사람과 프로세스가 필요하게 되었다.

아키텍처 리뷰의 흐름을 통제하고 관련 담당자들이 모이는 회의실을 예약하는 아키텍처 리뷰 담당자가 필요하다. 아키텍처 리뷰 담당자로서 Coordination을 진행하고 피드백까지 아키텍처 리뷰의 모든 단계에 참가하고 최종적으로 피드백을 주는 역할도 한다.

아키텍처 리뷰 담당자는 〈그림 1〉과 같이 리뷰 대상을 공지하고, 관련 자료를 아키텍처를 설계한 개발자 또는 아키텍트로부터 자료를 전달받는다. 1편에서 소개한 5개의 자료 (시스템 아키텍처, 어플리케이션 스택 아키텍처, 시스템 구성 및 네트워크 아키텍처, 서비스 흐름도)를 바탕으로 관련 매니저와 아키텍처 전문가들에게 미팅을 요청한다. 그리고 리뷰 회의를 진행하고 관련 피드백 내용을 전달한다.

그림 1_아키텍처 리뷰 흐름

리뷰 담당자는 자료를 받는 일과 회의실을 예약하는 일에서부터 회의를 리딩/진행하는 다소 쉽지 않은 일도 맡게 된다. 때로는 설계자의 감정을 상하고 않고, 아키텍처가 한 리뷰의 의의를 충분히 이해시켜서 〈그림 2〉와 같이 회의에 참여시킬 수 있도록 독려하기도 한다.

그림 2_아키텍처 리뷰 모임

아키텍처 리뷰 회의는 설계자의 아키텍처를 보면서 이해관계자 및 매니저와 아키텍트들이 들어가기 때문에 수많은 질의/응답이 나오게 하고 설계자에게 피드백을 자연스럽게 주고받을 수 있도록 하는 것이 목적인 회의이다.

II. 아키텍처 품질

아키텍처 품질을 체크하는 방법을 정의할 필요가 있다. 기술에 대한 내용을 주관적인 판단으로 검토하는 것이 결코 쉽지 않기 때문에 객관적인 품질 지표를 바탕으로 언급하도록 가이드 해야 한다. 특히 리뷰 하는 동안에는 쓸 데 없는 농담이나 공신력 없는 소문이나 기술/본질과 관련 없는 내용들은 언급 하지 않도록 주의하는 것이 중요하다.

아키텍처 품질 지표는 〈그림 3〉과 같이 성능/확장성, 보안, 가용성 등 크게 3개로 나누었다. 필자는 성능/확장성을 솔루션/서비스의 가장 중요한 부분이라고 생각한다. 그 다음으로 가용성을 중요하게 여기고 있는 터라 중요하게 생각하는 우선순위 중심으로 설명하고자 한다.

그림 3_아키텍처 리뷰 품질 지표

필자는 Apache Http/Tomcat 기반으로 하는 웹 서비스와 웹을 기반으로 하는 솔루션/서비스 개발을 주로 해왔고 아키텍처 리뷰 기획도 그런 프레임에서 바라보고 있다. 또한 필자의 소속 회사 또한 웹 기반의 솔루션/서비스 분야라서 전체적인 내용과 설명이 웹 기반으로 설명/적용되어 있음을 먼저 밝혀둔다.

2.1 성능 / 확장성

성능/확장성은 성능과 확장성 두 가지를 묶었는데, 이 2개의 품질 지표의 궁극적인

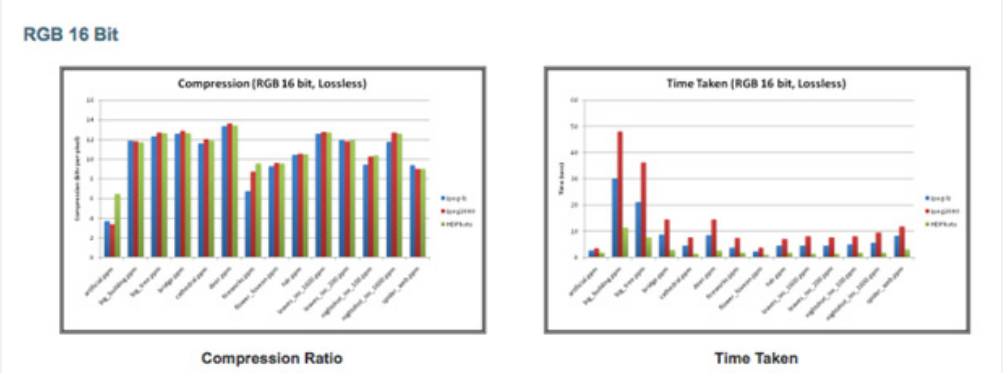
목표가 동일하기 때문이다. 이 궁극적인 목표에는 '요청량이 많아 질 때 어떻게 처리할 수 있느냐'라는 관점을 내포하고 있다. 좀 더 구체적으로 들어가보면-요청량이 많아질 때 성능적인 아키텍처를 고민할 수밖에 없고, 궁극적으로 요청량 처리(아키텍처)와 요청 저장량 (스토리지) 또는 최종 저장량을 어떻게 할 것인가에 대한 구체적인 내용을 작성할 수밖에 없다.

만약 작은 규모의 요청량만 들어왔다고 가정한다면 성능이나 확장성 보다는 다른 품질 지표들이 더 중요할 수 있다.

성능을 높이기 위한 많은 노력들이 있다. 알고리즘을 잘 사용하여 성능을 높일 수 있고, 속도가 빠른 네트워크나 솔루션/서비스를 이용해서 처리하는 경우도 있다. 필자가 속해 있는 웹 기반에서는 다양한 방법들을 제시할 수 있다.

만약, 비 손실 이미지 압축 알고리즘을 써야 하는 솔루션/서비스라면 소요되는 시간 대비 압축률이 제일 좋은 HDPhoto방식을 〈그림 4〉와 같이 테스트를 통해서 고민하고 최선의 선택을 선택할 수도 있다.

그림 4_비손실 압축알고리즘 테스트 자료



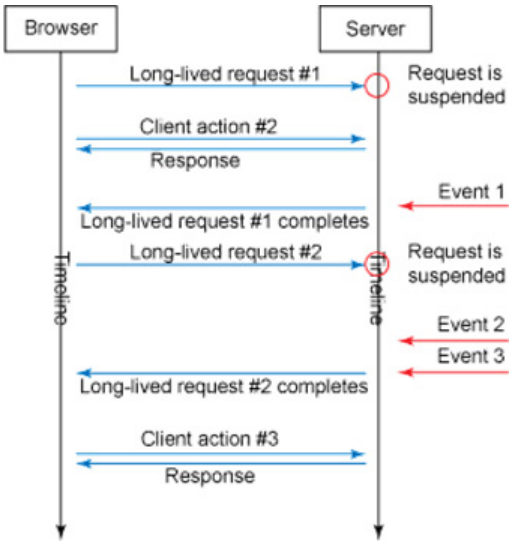
출처: http://www.imagecompression.info/lossless/

Web/WAS의 경우, 다음의 예를 들 수 있다.

한 대의 WAS 서버에서 요청을 처리하는 아키텍처가 있다고 하자. 한 대의 서버에서 감당할 수 있는 수준 이상의 요청량이 들어오면, 처리하지 못한다. 이때는 L4/L7 스위치를 맨 앞에 물리적으로 위치하게 하고, 여러 대의 WAS가 처리하도록 하고, Nginx나 Apache Http 서버를 WAS 앞에 두어 Load Balacing을 두도록 가이드 할 수 있다.

그러나 연결 지향적인 요청이 많다면 KeepAlive 설정을 On으로 처리하는 것도 좋은 방법이다. 다음 〈그림 5〉의 Polling과 같은 Event 단위의 처리가 많다면 NonBlocking 또는 Comet이나 Piggyback, Long/short polling 같은 아키텍처를 구성할 수도 있을 것이다.

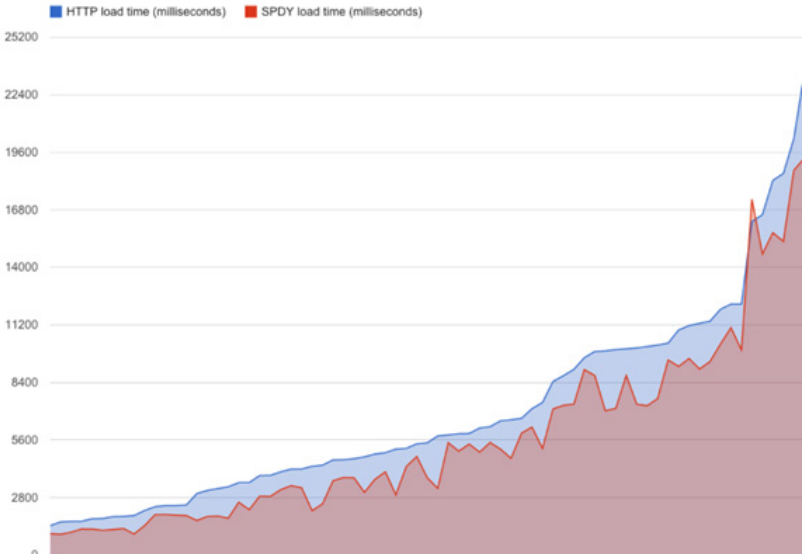
그림 5_Comet Architecture



출처: http://www.ibm.com/developerworks/library/wa-reverseajax1/

또는 〈그림 6〉와 같이 http 대신 spdy를 써서 모바일 환경에서 성능을 높일 수 있는 방법을 같이 고민할 수 있다.

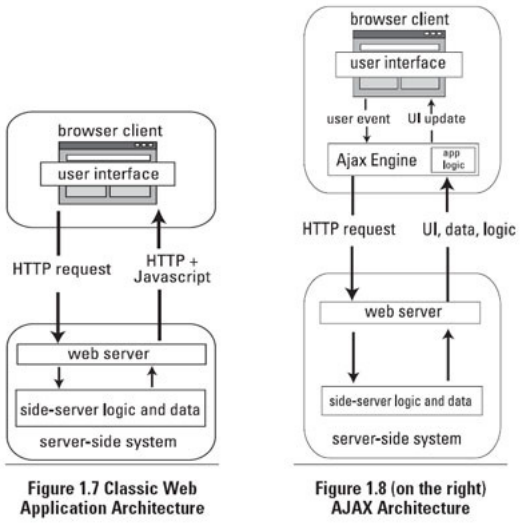
그림 6_모바일 환경에서 SPDY가 Http보다 속도가 빠르게 나온다는 벤치마크 정보



출처: https://developers.google.com/speed/articles/spdy-for-mobile

또는 〈그림 7〉처럼 웹 UI의 성능을 위해서 AJAX 을 사용할 수 있다.

그림 7_AJAX Architecture

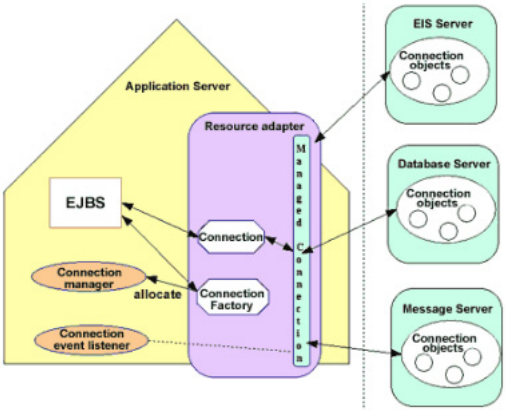


출처: https://ajax.sys-con.com/node/338111

UI 컴포넌트들이 많은 웹 서비스의 경우라면 빠른 AJAX를 활용하거나, Http Caching 정책 (예, Etag)을 잘 사용하여 국내가 아닌 해외 각지에서 빠르게 웹 화면을 보여줄 수 있도록 한다.

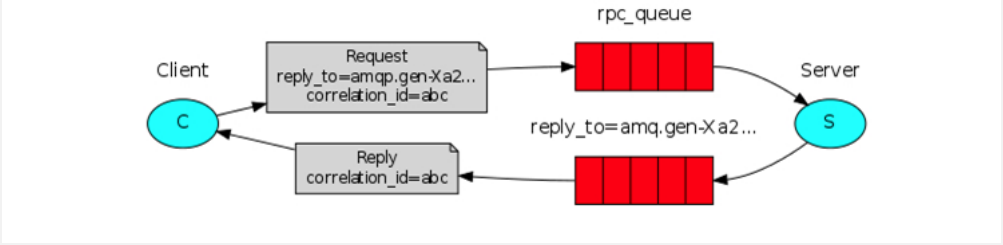
〈그림 8〉과 같이 WAS 와 DB간의 연결(Connection)을 완충하는 Connection Pool을 쓸지, 〈그림 9〉처럼 Message Queue 중 어떤 솔루션을 사용하고 어떤 방식을 써서 대용량 요청을 처리하는 것을 좋을지 살펴볼 수 있다.

그림 8_Connection Pool 정보



출처: http://www.javaworld.com/article/2076221/jndi/dive-into-connection-pooling-with-j2ee.html

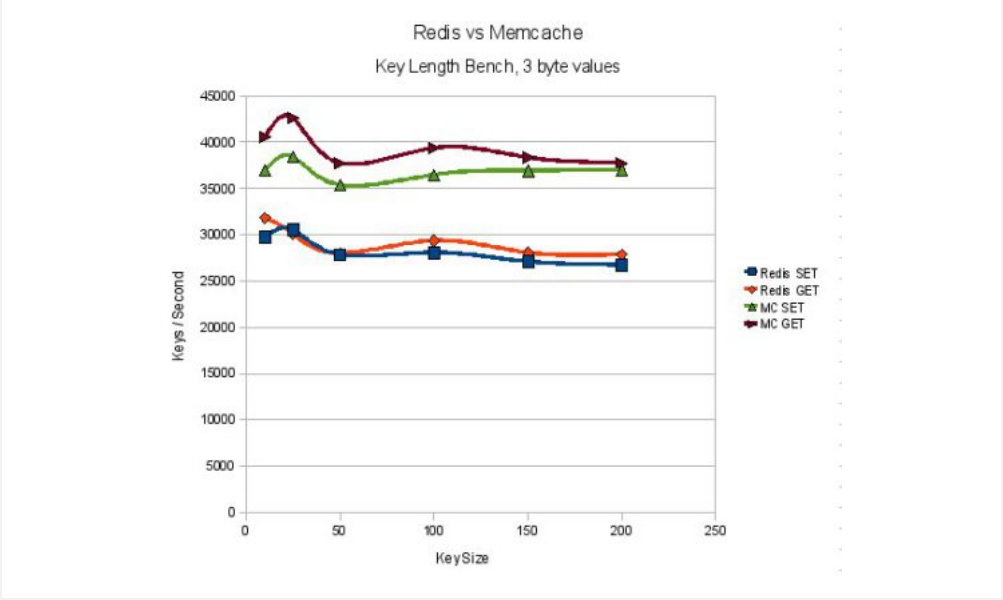
그림 9_Rabbit MQ 아키텍처



출처: http://www.rabbitmq.com/tutorials/tutorial-six-java.html

대용량 캐쉬 시스템인 Redis와 Memcached은 널리 알려져 있지만, 성능이 잘 나오는 특징이 있다. 〈그림 8〉의 정보처럼 이 둘 시스템은 Key size에 따라 혹은 get/set 호출 빈도에 따라 성능이 달라질 수 있다. 아키텍트는 캐쉬를 고민하는 솔루션/서비스에 이런 정보를 주어 확인할 수 있도록 할 것이다.

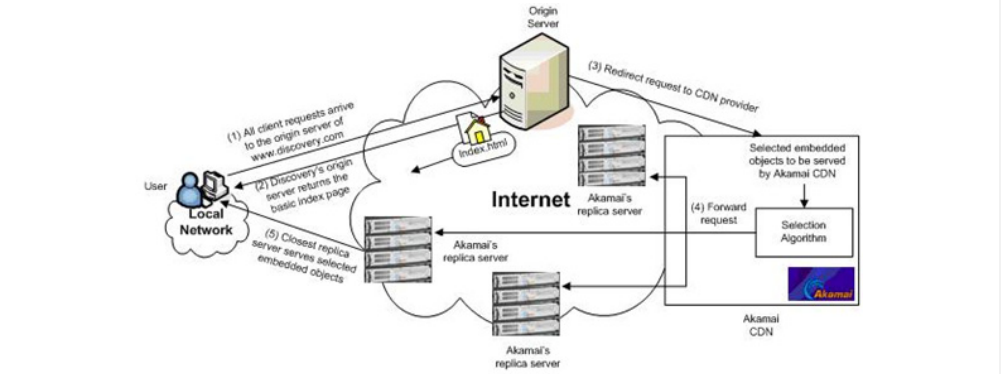
그림 10_Redis와 Memcache 비교



출처: http://systoilet.wordpress.com/2010/08/09/redis-vs-memcached/

정적 리소스(png/jpg와 같은 이미지 파일, css, xml)들을 WAS에서 처리하도록 하는 경우가 있는데, 이를 자주 사용하면 Apache Http 서버에서 처리하도록 할 수 있다. 내부 캐쉬를 사용하는 Nginx를 사용하면 보다 높은 수준의 속도를 보장받을 수 있다. 〈그림 11〉과 같이 트래픽이 순간적으로 늘 수 있으나, 서버 운영비용을 줄이기 위해서 CDN를 활용하는 방식을 사용할 수도 있다.

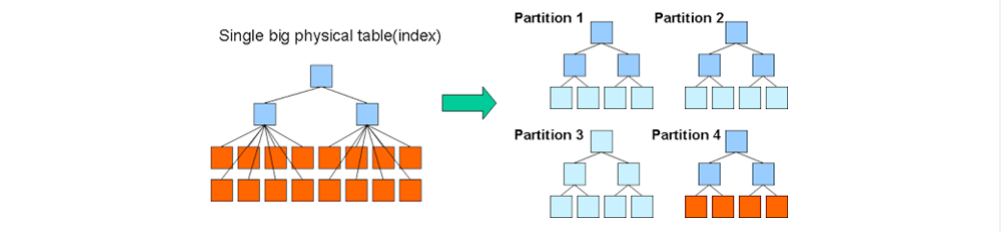
그림 11_CDN(Akamai) 적용 사례



출처: http://www.cloudbus.org/cdn/RD/CDNs.html

의외로 많은 개발자들이 CDN을 활용하지 않고 있으며, 특히 Tomcat과 같은 WAS 에서 정적 리소스를 돌리는 상황이 많이 발견했다. 이렇게 성능 관점보다는 단순한 구현에만 치중되는 개발 사례가 의외로 많이 존재했다. Nginx의 캐쉬 기능과 Web 서버의 캐쉬를 잘 활용할 수 있는 방법을 제시할 수 있다. 스토리지의 경우는 mysql을 사용하여 multi-master부터 partitioning, sharding 을 고민할 수 있고, mongo, hbase나 cassandra와 같은 nosql제품을 고민하는 경우가 있다. nosql을 쓰고자 하는 합당한 논리가 있다면 사용할 수 있을 것이다. 다음 〈그림 12〉는 하나의 큰 테이블을 내부적으로 여러 개의 테이블로 나누어 성능을 최적화 하는 방법을 사용한 예이다.

그림 12_Partitioning 정책 적용 사례



출처: http://yoshinorimatsunobu.blogspot.kr/2011/05/proper-handling-of-insert-mostly-select.html

만약 웹 기반의 솔루션/서비스라면 다음과 같이 Layer 별로 가용성을 높일 수 방법을 아키텍처 리뷰에서 집중적으로 얘기할 수 있을 것이다.

Network Layer (json/xml/plain-text, Http/SPDY)

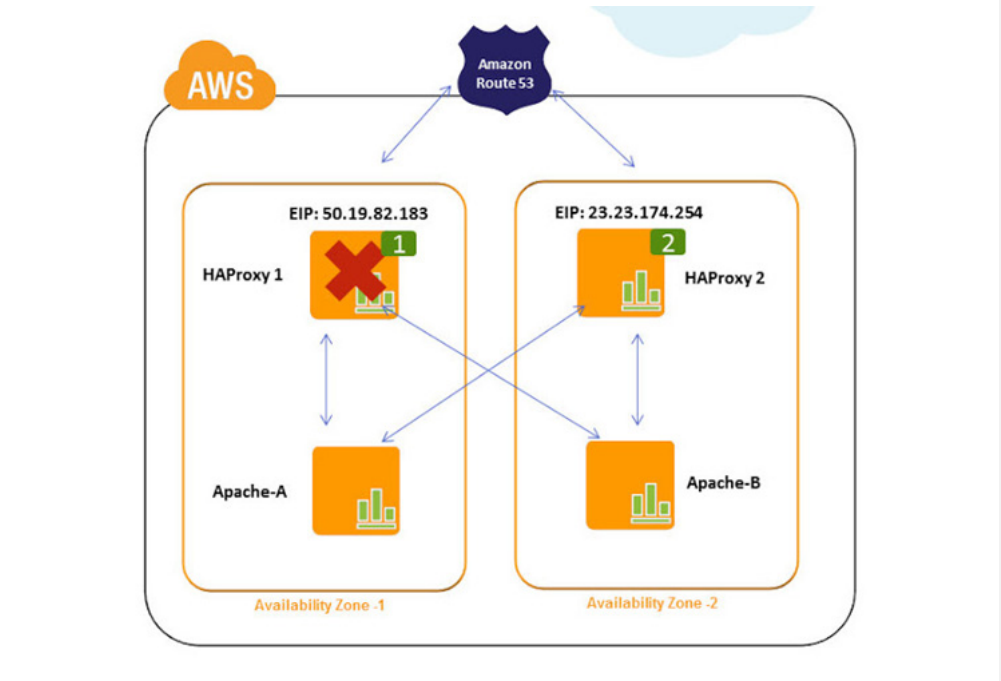
- Web/App Layer (Ajax, Comet, Tomcat, Nginx/Aapche Http/Vanish/CDN, Connection Pooling, 알고리즘, 데이터 구조)
- Store Layer (일반 DB, Nosql-redis/memcached, Sharding-mysql,cubrid)

2.2 가용성

가용성(Availability)이란 장애 없이 정상적으로 사용한 시간(Uptime)대비 장애 포함한 전체 사용 시간(Uptime+Downtime)으로 나눈 값을 말한다. 가용성 값이 높은것을 고가용성(HA, High Availability)이라고 하는데, 아키텍처 리뷰에서 말하는 가용성은 장애를 최대한 예방할 수 있는 시스템으로 개발되어 있는지 확인하는 품질 지표로 쓰인다.

웹 서비스를 WEB/WAS 한 대로 쓰는 상황에서 하드디스크 에러가 발생하면 해당 서비스는 유용하지 않다. L4/L7 스위치를 써서 이중화를 하던지, Zookeeper나 HAProxy 와 같은 오픈 소스를 활용해야 한다. 〈그림 13〉의 예시처럼 가용성을 높이기 위해서 HAProxy를 2 대를 사용해서 한 대의 서버가 장애가 발생하더라도 다른 서버에는 장애가 나지 않도록 설계할 수 있다.

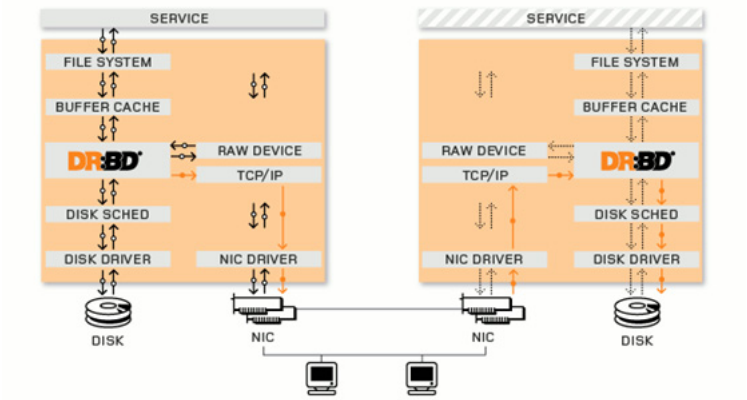
그림 13_AWS의 HA 사용 사례



출처: http://harish11g.blogspot.kr/2012/10/high-availability-haproxy-amazon-ec2.html

데이터 백업(미러링)을 통해서 장애를 최소화할 수 있는 〈그림 14〉의 DRDB를 사용하면 물리 서버의 가용성을 높일 수 있다.

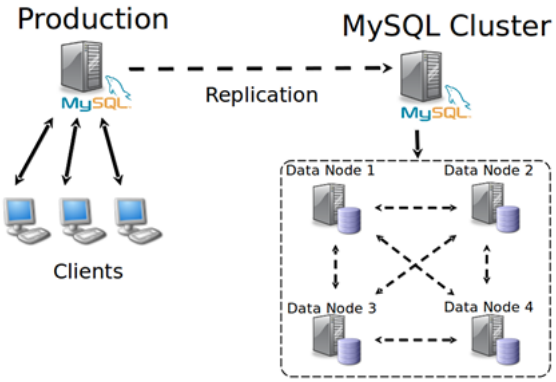
그림 14_DRBD 적용 사례



출처: http://www.drbd.org/

Mysql 을 이용할 경우 master-slave 의 2 copy 를 최소한으로 권장하고 대부분의 서비스는 3 copy로 쓸 수 있도록 한다. 〈그림 15〉의 예처럼 Replication을 두어 Mysql 장애에 잘 대응시킬 수 있고 성능 이슈까지 해결할 수 있다.

그림 15_Mysql clustering 사례



출처: http://mysql-nordic.blogspot.kr/2013/01/using-replication-to-make-move-from.html

가용성 확보를 위해서 Mysql은 어떠한 방법(예, MMM, MHA)를 사용할지? Oracle이라면 RAC를 적용할지를 작성할 수 있을 것이다. 특히 DB에SAN 스토리지와 같은 특별한 장비를 사용하여 가용성을 높일 계획이 있으면 아키텍처 문서에 포함시킨다.

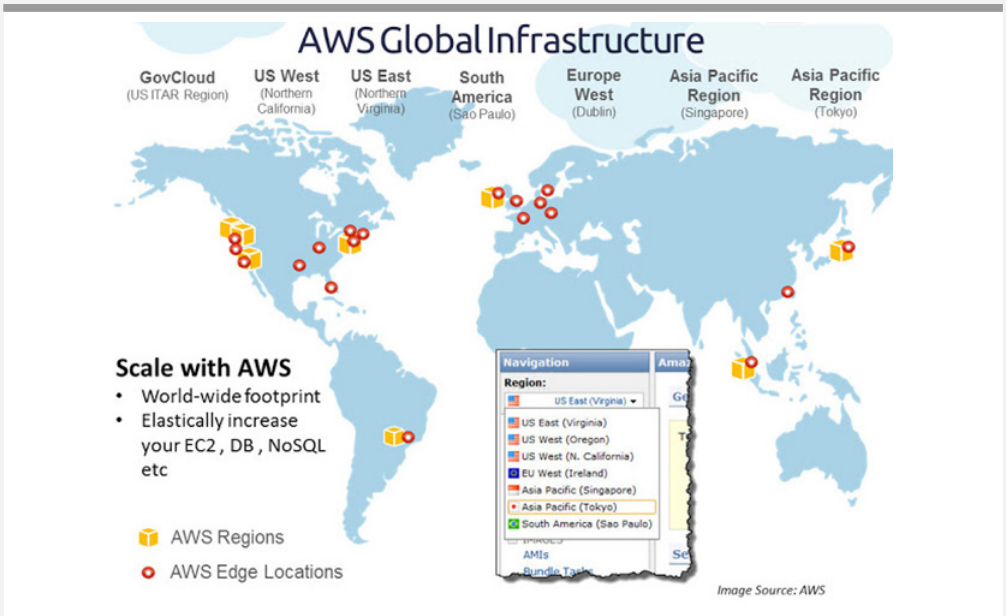
웹 서비스의 경우는 IDC에 나누어 서버들을 두어 가용성을 최대로 한다. 예를 들어 www.naver.com 의 경우는 GSLB(Global Service Load Balancing) 를 사용하여 IDC 이중화 및 무정지 서비스를 하고 있다.

```
$ nslookup www.naver.com
Server:         10.40.29.172
Address:10.40.29.172#53


Non-authoritative answer:
www.naver.com  canonical name = www.naver.com.nheos.com.
Name:   www.naver.com.nheos.com
Address: 202.131.30.12
Name:   www.naver.com.nheos.com
Address: 125.209.222.142
```

아마존의 경우, AWS서비스 가용성을 높이기 위해서 〈그림 16〉과 같이 여러 Region으로도 서비스를 할 수 있도록 했다. IDC 이중화 혹은 다중화의 서비스를 제공하고 있는 것이다.

그림 16_아마존 AWS 글로벌 Region 지원



출처: http://harish11g.blogspot.kr/2012/06/aws-high-availability-outage.html

가용성도 성능/확장성 관점처럼 다양하게 설계를 할 수 있다. Tier 별로 가용성을 어떻게 높일 수 있을 지 정리할 수 있고, 만약 웹 기반의 솔루션/서비스라면 Layer 별로 가용성을 높이는 방법을 아키텍처 리뷰에서 거론 할 수 있다.

- Network Layer (L4/L7 Switch, DNS, HAProxy, Zookeeper, DRBD)
- Web/App Layer (Web-Nginx/Apache-Http/Vanish, WAS-Tomcat/Netty/Jetty, 알고리즘, 데이터 구조)
- Database Layer (Mysql, Cubrid, Oracle-RAC, Altibase)

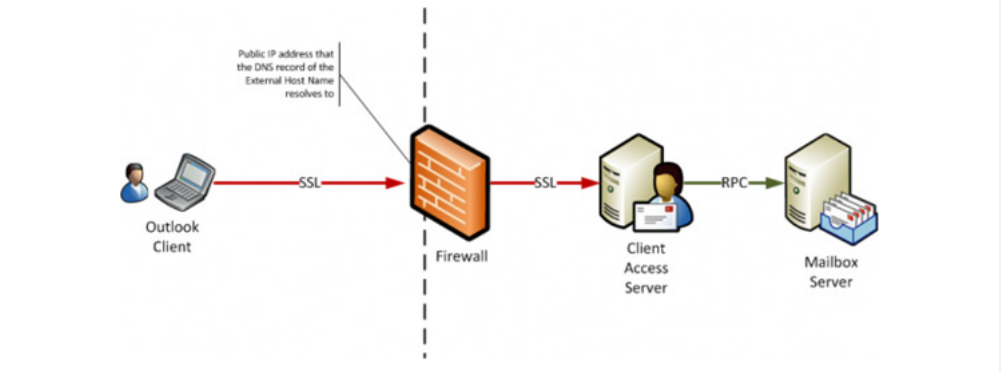
2.3 보안

필자는 보안 전문가가 아닌 개발자이다. 이에 개발과 운영을 진행하면서 중요하다고 생각했던 부분의 관점으로 접근하고자 한다.

수많은 은행들과 기업, 통신사, 개인 홈페이지까지 해킹으로 피해가 속출하는 요즈음, 보안은 중요한 품질 지표 중에 하나가 되었다. 때문에 스토리지에 암호화하여 저장하는 것뿐 아니라 통신 시 암호화하거나 외부에 오픈하는 웹 서비스 경우 방화벽을 설치하는 등 다양한 보안의 특성을 아키텍처 리뷰 문서에 포함하도록 하고 있다. 보안을 신경을 쓰고 있는지도 아키텍처 리뷰 문서에 반영하여 미리 보안 문제를 사전에 알릴 수 있도록 한다. 방화벽의 위치를 시스템 아키텍처에 포함시키거나, 공인 IP 또는 내부 IP 인지 구분하도록 하면 보다 효과적일 수 있다. 특히 통신 방법을 작성하면 그 효과는 더욱 극대화 된다.

다음의 〈그림 17〉의 Exchange 방화벽 적용 사례가 일명 베스트 프렉티스(Best Practice)라고 할 수 있겠다. 그림에서 알 수 있듯이 Client Access Server가 공인 IP지만 방화벽으로 노출을 최소화 했고, 통신을 SSL로 하고 그 중간에 보안 이슈가 없도록 했다. 내부 통신은 RPC를 이용해서 성능을 높이는 아키텍처로 디자인했다. 클라이언트와 서버, 서버와 서버끼리 통신 시 암호화를 어떻게 할지, 외부에서는 SSL로 내부에서는 HTTP나 TCP로 통신할지에 대한 정보를 아키텍처 리뷰에 작성하면 더욱 좋다.

그림 17_Exchange 방화벽 적용 사례

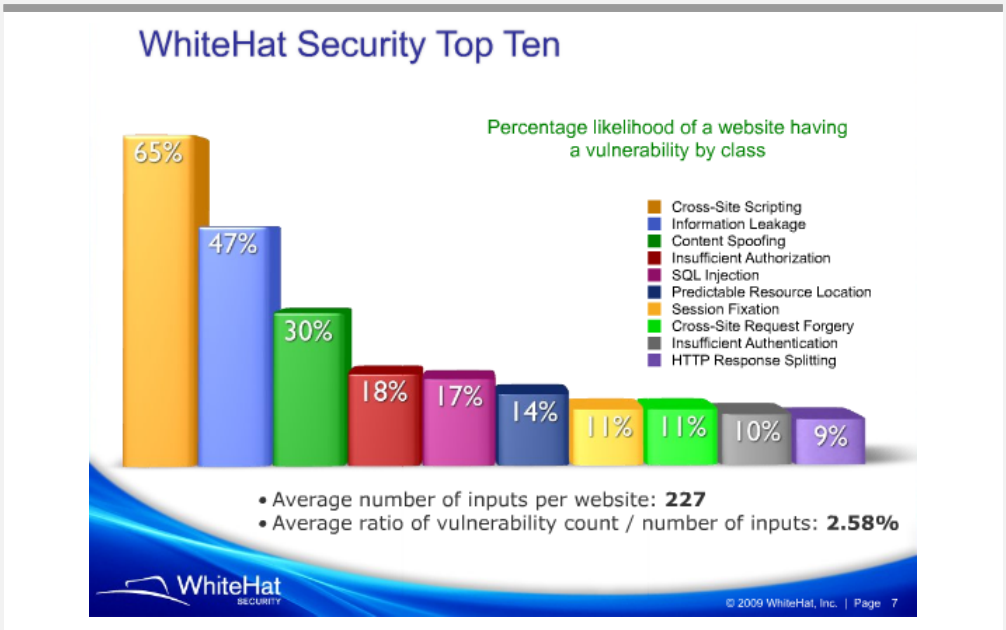


출처: http://exchangeserverpro.com/how-to-configure-exchange-server-2010-outlook-anywhere/

만약 방화벽 바깥에 오픈 API 서버가 있다면 모든 포트는 공격을 받을 수 있다. 단순히 80,443 포트만 오픈해야 하는 것들은 방화벽을 이용해서 공격을 최소화할 수 있도록 해야 한다.

〈그림 18〉은 WhiteHat에서 발표한 공격 패턴과 빈도율을 분석한 것이다. 다양한 방법으로 보안 공격이 수시로 들어오기 때문에 외부 오픈 서버들은 보안 공격에 뚫리지 않도록 각고의 노력이 필요하다. 우선, 공격에 대한 다양한 패턴을 방어할 수 있는 코드를 WAS에 적용 여부를 확인해야 한다. 개발 조직에 개발한 보안 모듈 혹은 자체적인 개발한 보안 모듈 사용 여부에 대한 내용은 아키텍처 리뷰 문서에 넣는 것이 좋다.

그림 18_보안 공격 사례 – WhiteHat 자료

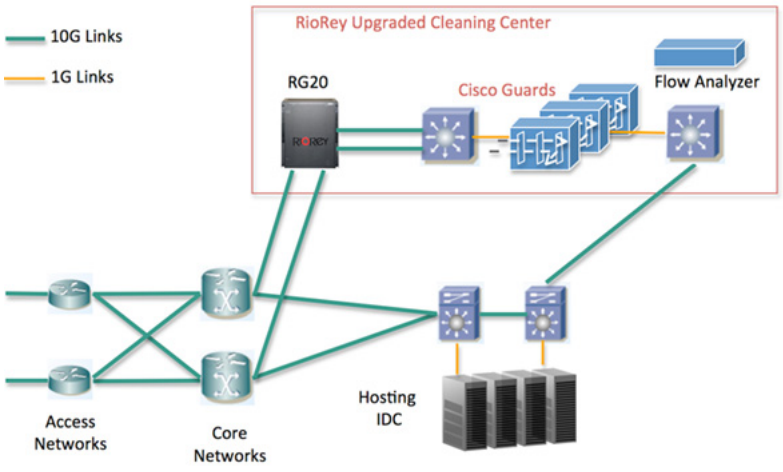


출처: http://www.slideshare.net/jeremiahgrossman/whitehat-security-website-security-statistics-report-q109

대형 포털의 경우 DDOS 공격과 같은 무차별 공격이 들어오는 경우가 많다. 경험 상 서버 장비를 늘려서 해결한 경우도 있지만, 워낙 필터링 해야 할 패킷이 많거나 패킷의 내용이 비정상적으로 유입되어 TCP 소켓을 계속 오픈시켜 자원을 더 이상 쓰지 못하게 하는 공격들이 있기 때문에 이를 대비하기 위해 보안 장비(Detector, Guard)를 이용하는 것을 추천한다.

〈그림 19〉처럼 외부로부터 들어오는 공격이 많은 웹 서비스라면, 보안 장비를 이용해서 공격을 대비하는 방안도 고민할 수 있다.

그림 19_RioRey DDOS 솔루션의 Cisco Guard 적용 사례



출처: http://www.riorey.com/products-cleaning-center.html

개인 정보를 다루는 고객 DB 쪽은 개인정보와 관련된 모든 정보를 어떤 식으로 알고리즘화 하는지, 저장하고 있는지 중요하다. 특히 회사 내부 보안 솔루션/ 외부 보안 솔루션을 사용하는 경우는 아키텍처 리뷰 문서에 포함시키도록 한다.

개인정보 및 중요 정보는 암호화 없이 단순히 Http로 전달하는 경우는 원칙을 정해 금지한다. 이를 최대한 https나 암호화된 알고리즘을 써서 개인정보 또는 중요 정보를 암호화 하도록 한다. 인하우스(In-house) 통신이더라도 Node간의 구간 통신을 Salt와발급한 Key를 이용하여 암호화/복호화하여 통신 전문을 공격자가 가로채더라도 복호화가 어렵도록 해야 한다. 또한 DB 에 저장하는 개인정보와 패스워드는 상황에 맞는 복잡한 알고리즘(AES128, SHA512, Salting 알고리즘, bcrypt) 등을 이용하여 최대한 공격을 막도록 해야 한다.

관련된 정보는 링크(http://helloworld.naver.com/helloworld/318732)를 참조한다.

인증 서버나 회원 서버의 경우는 성능보다는 보안을 가장 신경 써야 한다. 아키텍처 리뷰에서는 이를 감안하여 암호화/복호화하는 성능 저하보다는 보안 부분을 배려할 수 있어야 한다. 보안을 감안한다면 다양하게 설계를 할 수 있다. Tier 별로 보안 품질을 어떻게 높일 것인지 정리할 수 있다. 만약 웹 기반의 솔루션/서비스라면 다음과 같이 Layer 별로 보안 품질을 높이는 방법을 아키텍처 리뷰에서 거론할 수 있다.

- Network Layer (Guard, Node간 구간 암호화)
- Web/App Layer (Web-White/BlackList관리, SQL/XSS Injection방어)
- Database Layer (개인정보 및 패스워드 암호화)

ERD (Entity Relation Diagram)

아키텍처 리뷰 문서를 만들 당시에는 ERD는 상세하지 않을 뿐 더러 단순하고 핵심 개념만 있는 Entity-Relation 모델만 존재한다. 실제 핵심 개념을 설명할 수 있는 수준으로 진행하여 이해도를 높이도록 한다.

Ⅲ. 체크리스트

아키텍처 전문가들에게 〈표 1〉의 예제처럼 평가 매트릭스를 제공한다. 이 내용을 바탕으로 기본적인 리뷰를 진행할 수 있도록 한다.

표 1_평가 매트릭스 예제

성능/확장성	서비스 운영 시 성능 이슈 또는 병목 현상이 있을 만한 곳이 없는가?
	유저(User) 요청이 많아질 경우에 대한 성능적 아키텍처가 포함되었는가?
	정적 리소스(css, js)를 서비스하기 위해 CDN 또는 Web 서버를 잘 활용했는가?
	WAS 앞에 Web서버를 두고 적절히 Load Balancing과 FailOver를 이용했는가?
	Cache Lib/Cache Server를 적절히 이용했는가?
	부하를 잘 분산하고, 파일을 많이 사용하는 경우 분산파일시스템/NAS를 사용하도록 하였는가?
	...
보안	중요한 정보 또는 개인정보 전달 시 어떻게 전달하는가?
	외부 오픈 서버일 경우, 대용량 트래픽, DDOS, 해킹 공격에 대한 방어를 고민했는가?
	Security Zone 위치에 알맞게 서버가 구성되어 있는가? 중요 서버군은 방화벽 안쪽에 있는가?
	방화벽 외부 서버와 통신 인증 사용을 적절하게 반영했는가?
	...
가용성	Web/WAS 서버의 장애를 처리하는 내용이 들어가 있는가?
	스토리지 백업 및 요청 분산 및 장애에 대한 처리 방식이 들어가 있는가?
	네트워크 단절에 대한 처리 방식이 들어가 있는가?
	...

아키텍처 리뷰 회의에서의 토론 주제가 '좋은 아키텍처링 사례'와 '필수 수정 사항 & 추후 고려 사항'이 나올 수 있도록 한다. '좋은 아키텍처링 사례'는 아키텍처 리뷰를 통해서 나온 베스트 프랙티스로 추천될 만한 칭찬받을 내용이다. 그리고 '필수 수정사항'은 품질 속성 중 심각한 결함이 있을 만한 것으로 작성하여 수정을 유도하는 것이다. 그리고 '추후 고려 사항'은 2가지로 나누어지는데, 첫 번째는 당장은 하지 않아도 되지

만 아키텍처를 일부/부분적으로 권고 하는 내용이며, 두 번째는 실제 코드 구현/운영 환경에서 만날 수 있는 잠재적인 위험 또는 결함(Defect)을 예방할 수 있도록 내용을 담을 수 있다.

'리뷰'라는 특성 때문에 아키텍처를 설계하는 사람을 공격하는 것처럼 하지 않도록 하며 감정을 상하지 않도록 해야 한다. 따라서 리뷰의 결과가 '칭찬받아 마땅한 것(강점)'과 '고려해봐야 할 사항(약점)'으로 나오도록 한다. 아키텍처 리뷰 담당자는 애자일(Agile)의 회고와 비슷한 맥락으로 진행되도록 하는 것이 좋다.

아키텍처 리뷰 담당자는 회의 참석자들이 다양한 관점으로 아키텍처를 볼 수 있도록 회의를 진행한다. 그래서 장점과 단점들을 미리 파악할 수 있도록 사전에 자료를 공유하고, 생각할 부분을 사전에 알려주는 것도 좋은 방법이다. 리뷰 담당자는 회의에 나왔던 내용들을 정리하고 설계자에게 리뷰 때 나온 내용을 피드백 하도록 한다.

특히 관련 자료들은 모두 문서화(또는 위키에 작성)하여 다른 개발자나 아키텍트들이 열람할 수 있도록 공유하고, 같은 실수가 반복되지 않도록 한다. 좋은 사례 또한 공유하여 보고 배울 수 있도록 하는 것이 중요하다.

리뷰 담당자는 시간 배분에 주의해서 회의 시간이 1시간이 넘지 않도록 한다. 아키텍처가 리뷰 내용을 숙지하고, 이해하고 있다면 보통 30분 이내로 회의가 종료되지만 가끔씩은 1시간이 훌쩍 넘어갈 정도로 준비 안 된 경우가 있으니, 회의가 원활하게 진행되도록 회의 준비에 각별히 신경 쓰도록 한다.

Ⅳ. 결론

수 많은 솔루션들이 생성되는 만큼 이를 관리하는 사람과 프로세스가 필요해졌고, 이에 아키텍처 리뷰 담당자와 리뷰 프로세스, 리뷰 체크리스트 등의 필요/중요성이 대두되고 있음을 살펴보았다. 아키텍처 리뷰는 아키텍처 모듈의 중복을 최대한 줄이고 모호한 부분은 최대한 명확하게 잡아주는데 목적이 있다.(아키텍처의 품질지표인) 성능/확장성, 가용성, 보안 등이 잘못/누락되어 있거나 불분명한 설계로 나타날 수 있는 결함(defect)의 최소화하는 효과적인 방법이 아키텍처 리뷰임을 강조한 본 글이 보다 능률적인 업무 수행에 도움이 되길 바란다.