모바일 게임 전용 BaaS/PaaS 구현 사례와 디자인 트레이드 오프

김진욱 (jinuk.kim@ifunfactory.com)
iFunFactory Inc.

Who am I

- 김진욱 (jinuk.kim@ifunfactory.com)
- 현재: 게임 서비스를 위한 하이브리드 클라우드 서비스 개발 중
- 2012~2013: 넥슨 코리아 신기술개발실; 게임 서비스 용 클라우드 개발
- 2007~2012: 엔씨소프트 서버플랫폼 팀; 게임 서버 개발

Contents

- Design of *Game Services*
- Design of *Game Server* : Tradeoffs
- Case Studies

Introduction

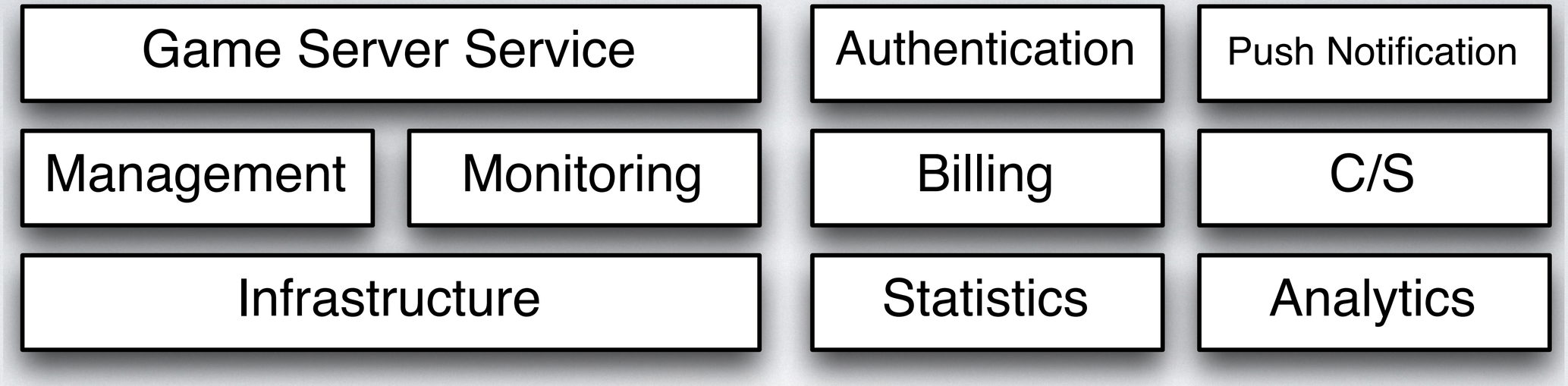
What is a BaaS or a PaaS ?

- Backend-as-a-Service: 게임 서비스를 위한 클라이언트 이외의 것들을 필요한 만큼 제공해주는 것
- Platform-as-a-Service: 소프트웨어 서비스를 만들기 위한 도구와 라이브러리, API 등을 필요한 만큼 제공해주는 것

Design & Tradeoffs

- 디자인: 무언가를 만들기 위한 제안이나 계획, 그리고 그 실행
- 트레이드오프: 디자인 단계마다 내린 결정에 따라 생기는 장/단점
- 게임 서비스 / 게임 서버를 만들 때 필요한 결정과 그 선택의 영향을 살펴보겠다.

Design of Game Services



게임 서비스 아키텍쳐

게임 서버 자체와 이를 운용하는 하드웨어와 서비스들
게임을 서비스하기 위해 필요한 각종 내/외부 서비스들

게임 서비스를 위한 BaaS: 디자인 요소

A. 어떤 컴퍼넌트를 제공해줄 것인가?

B. 멀티 플랫폼지원을 어떻게 도울까?

C. 게임 서버는 어떤 인터페이스를 보게 될까?
(=게임 서버는 어떤 제약을 받게 되는가?)

어떤 컴퍼넌트를 제공할까?

- BaaS가 모든 서비스를 제공하는 것은 아니다
- 필요한 서비스를 제공하고 / 일부는 외부 서비스를 연동하게 해준다
- 혹은, 거의 모든 서비스를 외부 서비스에 맡기고 이를 연동하는 역할만 할 수도 있다

어떤 컴퍼넌트를 제공할까?

- 외부 서비스를 쓰면 사용자가 알아야하는 서비스 내 API가 줄어든다
- 외부 서비스가 많을 수록 전체 서비스의 통일성이 저하

멀티플랫폼 지원

- 서버 수준의 API로 제공? (eg. RESTful API)
- 플랫폼 별 클라이언트 프레임워크 / 라이브러리 제공?

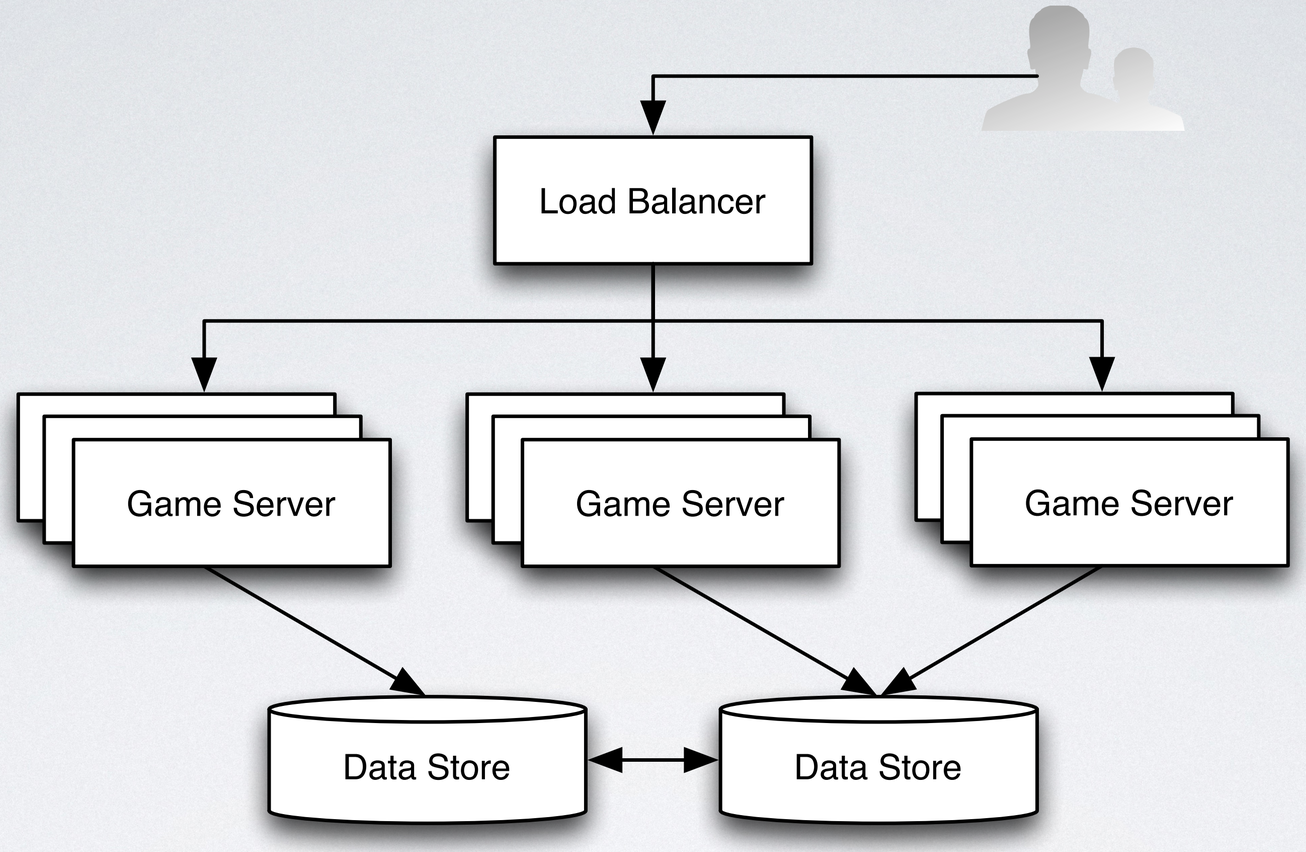
게임서버 인터페이스

- 서버 하드웨어를 빌려준 수준의 자유도를 줄 것인가? (eg. Amazon EC2)
- 특정 프로그래밍 모델을 강제할 것인가? (eg. Google AppEngine)
- 특정 클라이언트 API/모델을 강제할 것인가? (eg. Photon Realtime)

Design Of Game Servers:
Tradeoffs

History: 두 가지 선택지

- Web 서비스 류의 구현
- 전통적인 PC Online 게임 류의 구현



게임 서버 구성

사용자와 통신하고 / 이를 처리하고 / 저장하는 일련의 기능들

Design: *Goals*

- 게임이 요구하는 게임 플레이 특성 (eg. 실시간대전 / 턴 방식 보드게임)
- 확장성이 얼마나 필요한가?
- 개발 시간 (vs. 게임 안정성)
- 운영 비용

무엇이 선택을 좌우할까?

- 게임이 특정 선택을 강제 (실시간 대전, …)
- 팀을 만들 수 있는가 (해당 기술을 아는 프로그래머의 수)
- 빨리 만들어야하나? 부하를 견디는게 중요한가?
- 예측한 서버 수, 서비스 커질 때 확장 비용

게임 서버 아키텍쳐: 디자인 선택지

- Transport Layer: HTTP vs. Custom Transport
- Server Logic: Stateless vs. Stateful
- Data Store:
  - Write / Cache Policy
  - Consistency / Isolation for horizontal scaling
  - Wirte format, …

Transport Layer

게임 클라이언트와
게임 서버는
어떤 방법으로
메시지를 주고 받을까?

HTTP vs. Custom Transport

- HTTP : 웹 그리고 모바일 시대의 공용어 (*Lingua Franca*)
- Custom Transport: PC 온라인 시절과 웹 시대의 커스텀 프로토콜

HTTP ?

- 많은 수의 모바일 게임이 HTTP로 통신한다
- 게임 서버는 웹 애플리케이션으로 구현
- HTTP 위에 JSON, XML 혹은 protobuf 등의 메시지를 실어 보낸다

HTTP: *Pros*

- 잘 정의된 프로토콜이라 트랜스포트 자체를 신뢰할 수 있다
- 서버 프레임웍과 클라이언트 라이브러리가 많다
- 로드밸런서를 이용해서 서버 수를 쉽게 확장(IaaS 사용에 용이)
- PC 온라인 게임 서버 직군 이외의 개발자가 많다

HTTP: *Cons*

- 양방향 통신을 할 수 없다
- HTTP 프로토콜 자체의 오버헤드 (헤더, 요청 압축, …)
- 하나의 연결에 대해 동시에 하나의 요청만 보낼 수 있다
- 리버스 엔지니어링 하기 쉽다

Custom Transport?

- 꽤 많은 수의 모바일 게임이 사용 중
- PC 온라인 게임에서 부터 쓰던 방식:TCP 등을 사용한 통신을 구현
- C++/C# 뿐만 아니라 node.js 등을 활용
- 커스텀 메시지 혹은 JSON, protobuf, thrift 메시지를 전송한다

Custom Transport: *Pros*

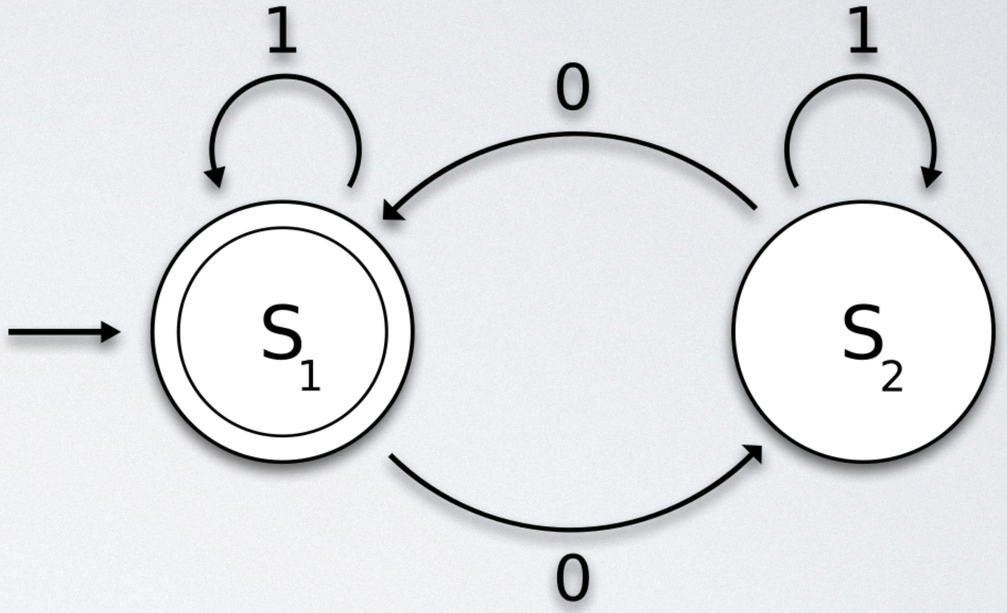
- 실시간 / 양방향 전송이 쉽다
- 특정 게임에 특화시켜서 메시지 구성이나 압축을 효율화하기 좋다
- 리버스 엔지니어링하기 좀 더 복잡하다 (외부에 노출된게 적다)

Custom Transport: *Cons*

- 더 복잡하다 = 프로그래머 생산성이 떨어진다
- 사용자가 늘어날 때 로드밸런싱이 복잡해진다
- 유지보수 할 코드의 양이 많다

게임 서버 로직

게임 서버가 유저 상태를
얼마나 많이
얼마나 오래 유지하면서
게임 로직을 처리할까?

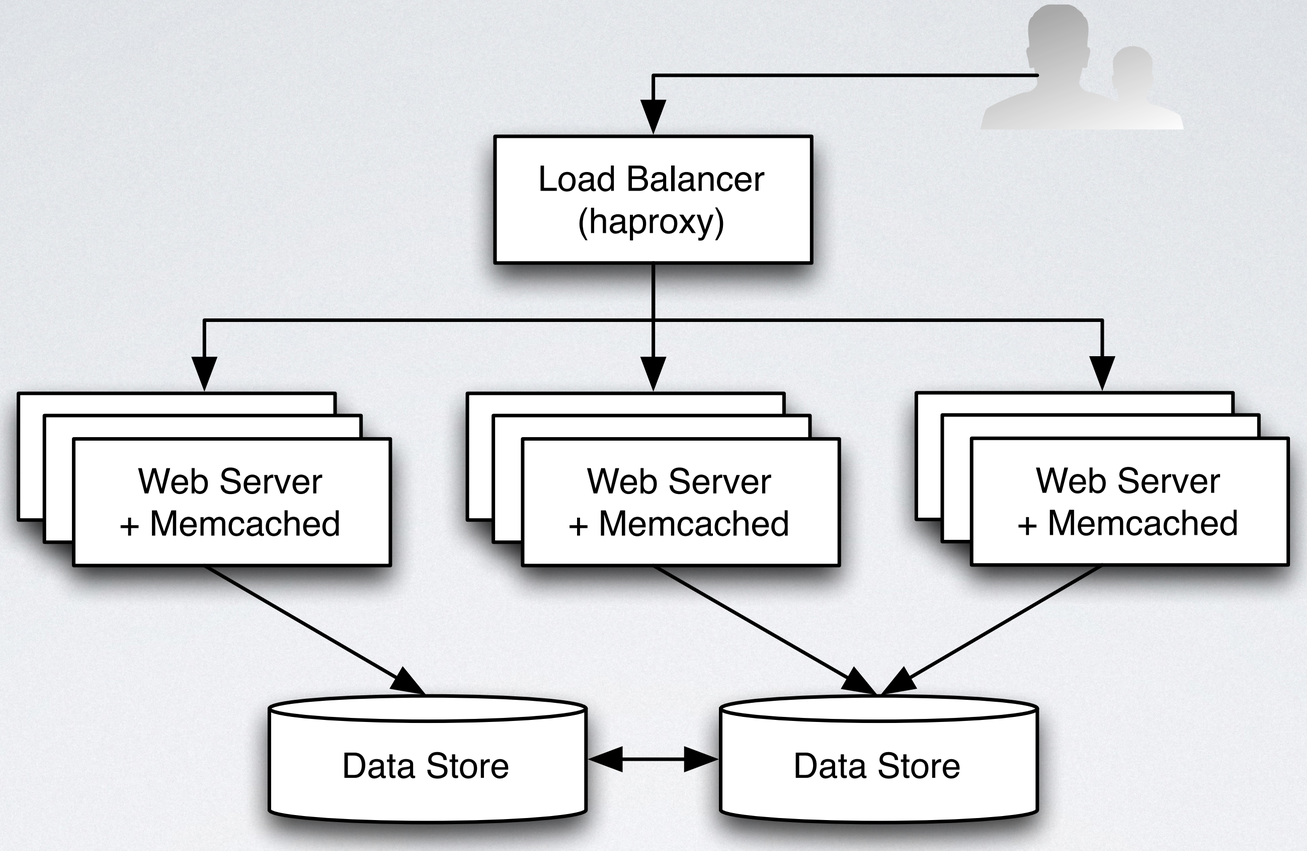


Stateless vs. Stateful

- Stateless: 웹 서버처럼 서버에 상태를 유지하지 않는 구조
- Stateful: PC 온라인게임 서버처럼 서버에 유저 상태를 유지하는 구조

Stateless Game Server

- 대부분의 웹 애플리케이션 서버가 여기에 해당
- 게임 메시지를 처리하기 위해서 데이터 저장소 계층에 접근해야 함
- 유저 간 상호작용 처리도 항상 데이터 저장소 계층을 이용



Configuration Example

Web 서비스에서 사용하던 컴퍼넌트들을 활용해서 작성한다

Stateless Game Server: *Pros*

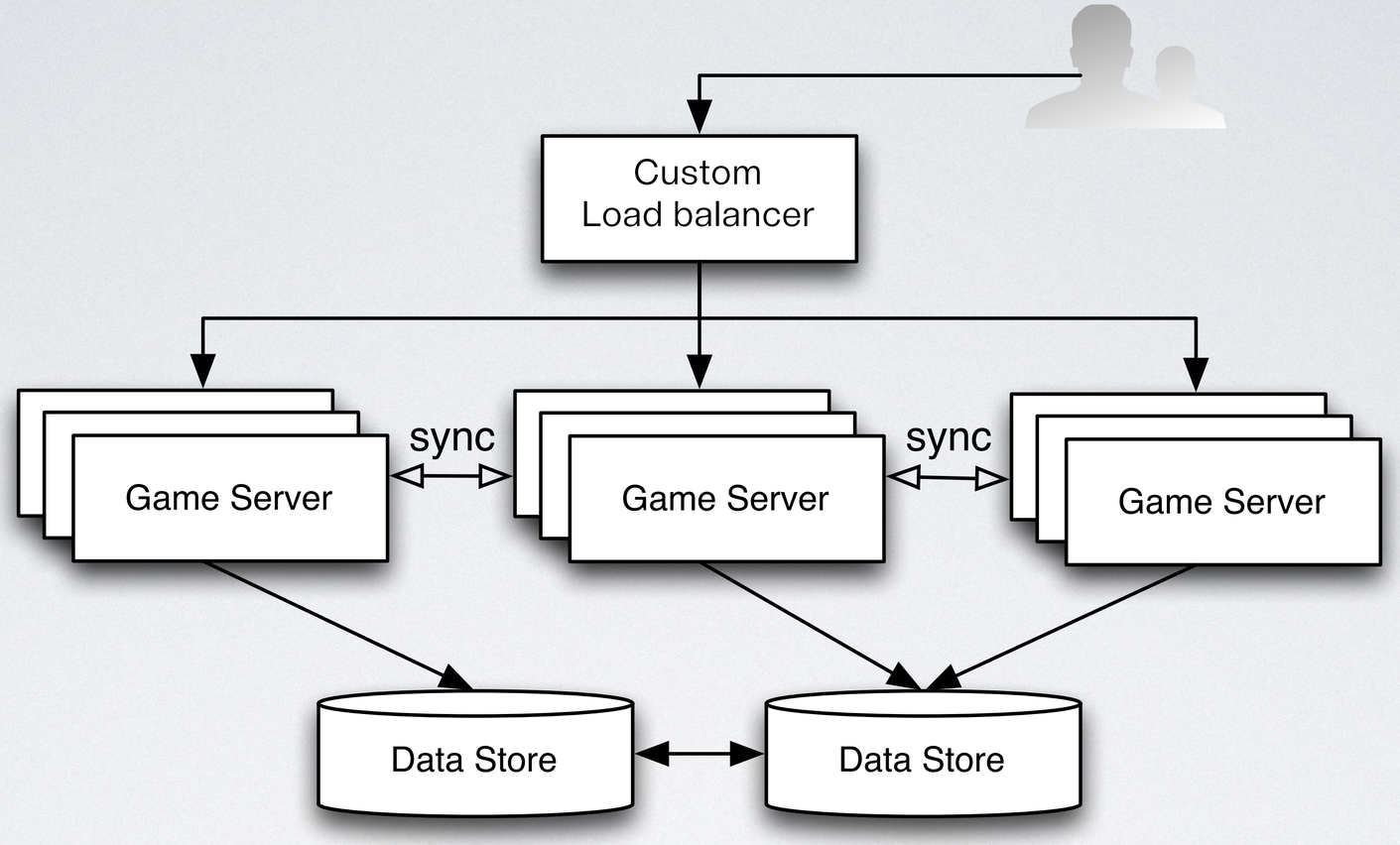
- 게임 서버 크래시에 안전한 편
- 게임 서버를 수평적으로 확장하기 쉽다
- 웹 서비스에서 활용하는 각종 컴퍼넌트를 활용하기 쉽다:
  (load-balancer, memcached, redis, …)

Stateless Game Server: *Cons*

- 서버 쪽에서 생성하는 이벤트 처리가 복잡하다
- 게임 서버 간 동기화를 데이터 저장소 계층에서 처리하는 오버헤드
- 유저 간 상호작용을 대부분 데이터 저장소 수준의 동기화로 처리해야
- 데이터 저장소 계층이 복잡하다

Stateful Game Server

- 전통적인 PC 온라인 게임 서버들이 여기에 해당
- 게임 상태 / 유저 상태를 개별 게임 서버의 메모리 상에 유지
- 게임 서버 = 게임 로직 처리 + 데이터 저장소 제어



Configuration Example

게임 서버 사이에 명시적으로 동기화하는 프로토콜이 들어간다

Stateful Game Server: *Pros*

- 데이터 저장소에 저장할 방법을 제어하기 쉽다
- 실시간 이벤트나 서버 생성 이벤트 처리가 쉽다
- 유저간 상호 작용은 개별 게임 서버 수준에서 (대부분) 처리한다
- 서버 당 동접 처리 수가 높다

Stateful Game Server: *Cons*

- 게임 서버를 수평적으로 확장하기 힘들다
- 직접 제작해야하는 코드 양이 많다 (동기화, 로드밸런싱, …)
- 서버 크래시 처리가 복잡하다

데이터 저장소

쓰기 정책
캐시 구성
확장성
가용성
…

Facts

- 게임 서버의 데이터 저장소는 "쓰기 위주" (*write-intensive*)
- (거의 변하지 않는) 약간의 읽기 위주의 데이터들도 있다
- 데이터 업데이트가 모두 디스크로 가지는 않는다

Policy

- 데이터 쓰기는 어떻게 처리할까? (*write back* vs. *write through*)
- 분산 서버 문제: *Horizontal Scaling*
- 데이터 저장 포맷

데이터 쓰기

- In-memory write-back cache는 성능을 극대화할 수 있지만 서버 크래시에 매우 취약하다
- 데이터 저장소에 진행 중인 쓰기가 다른 유저에게 얼마나 빨리 보여야 할까? (consistency 문제 + isolation 문제)
- 분산된 데이터 저장소에서 쓰기 종료를 어떻게 보장할까?

데이터 쓰기: Cache

- 캐시 레이어 구성은 어떻게?
- 분산된 캐시 서버간의 coherency 처리는?
- 캐시 서버 - 게임 서버 간 인터페이스? (transparent cache?)

Datastore Scaling

- 게임은 쓰기 위주: master에 쓰기 + slave에서 읽기로는 최적화하기 쉽지 않다
- 데이터의 key에 따라 서로 다른 저장소 서버에 저장하는 방법을 쓴다
- 문제: 사용하려는 도구가 지원하는 수준? 자동화? 필요한 기능?

Datastore Scaling (2)

- DB 수준에서 auto-sharding? or
- 응용 프로그램에서 Consistent hashing?
- 분산 트랜잭션은 어떻게?
- 프로그래밍 모델에 제약?

Auto Sharding: Example

- **Gizzard** from Twitter (https://github.com/twitter/gizzard); *retired*
- 프레임웍 수준에서 auto-sharding을 지원한다
- 네트워크 연결 단절 처리: 재시도 + 최신 연산만 실행
- 프로그래밍 모델: 모든 쓰기가 *idempotent & commutative*

Data Format

- Storage format vs. wire format ?
- 프로그래밍 편의성 (생산성) vs. 디스크 공간 + 저장소 CPU
- 사용할 수 있는 쿼리 방법 차이

Case Studies

Amazon Web Services

- Infrastructure-as-a-Service
- 일반적인 웹 서비스 구성에 필요한 요소들을 모두 가지고 있다
- 게임적인 요소는 모두 개발자 손에
- 꼭 웹 기반이 아니어도 괜찮다
  (제공하는 건 가상머신과 몇 가지 서비스)

AWS as a Backend Service

- 게임 서비스를 만드는데 쓸 수 있는 여러 빌딩 블럭을 제공한다
- 빌딩 블럭을 사용해서 각각의 게임 서비스를 제작해야 한다
- Write-intensive한 처리를 dynamo-db에 떠넘길 수도 있다
- Load-balancer 서비스 제공: 급격한 로드 상승엔 부적절할 수도 있다

AWS: Decisions

- Transport Layer: *ANY*
- Server Logic: *ANY*
- Data Store: *ANY*
- **HTTP + Stateless + Scalable Data Store (DynamoDB) 선호**

Heroku

- Platform-as-a-service
- 웹 서비스를 위한 플랫폼
- 웹 서버 + HTTP + RDB 기반의 게임 서비스를 만들기 좋다

Heroku as a Backend Service

- AWS 위에서 동작, 단 웹 서비스 기반의 서비스를 지원
- 많은 수의 애드온: nosql, notification, monitoring, …
- 지역 제한: US + EU
  (see https://devcenter.heroku.com/articles/regions)

Heroku: Decisions

- Transport: HTTP
- Server Logic: Stateless (web application)
- Data Store: Relational DB + In memory caches (memcached, redis)

Google AppEngine

- Platform-as-a-Service, *but* …
- *Managed-VM* 을 허용함 *(GCE)*
- 기존 *Google Game Platform* 과 연동
- 강력한 *analytics tools*

Google AppEngine as a Backend Service

- 웹 서비스 기반의 게임 서버를 지원
- 장시간 동작하는 비동기 작업도 가능
- 추가로, managed VM 을 통해서 AWS EC2 같은 일도 할 수 있다
- RDB가 아닌 데이터 저장소 계층 제공

Google AppEngine as a Backend Service (2)

- AppEngine이 제공하는 언어 제한: Python, Go, PHP, Java
- 프로그래밍 모델의 제한: 응답 방식, 라이브러리, 응답 시간 제약
- 데이터 저장소 모델 제한:
- 지역 제한: **중국**에서는 어떻게?

Google AppEngine: Decisions

- Transport Layer: HTTP
- Server Logic: Stateless + Managed VM + Asynchronous Jobs
- Data Store: non-relational DB (GQL) + memcached

Google Play Game Services

- Room 에 기반한 게임을 지원 (realtime)
- 혹은, 턴 방식 게임 지원
- 메시지 기반 (Android, iOS, Web)
- *Socket interface*는 *Android* 전용

Google Play Game Services: Decisions

- 기본적으로 AppEngine과 같지만,
- Transport Layer: Android는 HTTP 이외도 지원
- Server Logic: Google Compute Engine 위에 stateful 서버를 지원

Photon Realtime

- Software-as-a-Service
- Unity3D 전용 (플러그인)
- 제약이 심한 프로그래밍 모델
- 클라이언트 간 상태 복제 기반
- 제한적인 RPC 지원

Photon Realtime: Decisions

- Transport Layer: Custom Protocol
- Server Logic: Stateful (클라이언트 측의 상태 복사)
- Data Store: 별도 지원 없음

선택의 문제

Summary

- 게임 서비스에선 (A) 제공할 컴퍼넌트, (B) 멀티 플랫폼 지원 방식, (C) 게임 서버 인터페이스를 결정해야 한다
- 게임 서버 수준에선, (i) transport, (ii) server logic, (iii) data store 의 종류/형태를 결정해야 한다
- 이를 바탕으로 AWS, AppEngine, Heroku, Photon 등을 살펴봤다

Questions?