LINE Platform Development Chronicle

최흥배

https://github.com/jacking75/choiHeungbae

LINE DEVELOPER DAY_2015 Tokyo "LINE Platform Development Chronicle"

http://dev.classmethod.jp/event/linedevday2015-a4/

LINE 메시징 기반 역사

초기 아키텍처 개선: long polling

LINE은 2011년 6월에 발매.
스마트 폰에서 사용하기 쉬운 채팅 앱이 스마트 폰의 주류가 되어 가는 중이라서 어떻게든 빠르게 해서 2개월 정도로 해서 출시.

그래서 최초기의 아키텍처는 매우 심플.
리버스 프록시(Apache), 애플리케이션 서버(Tomcat) Back에 Redis와 MySQL.
애플리케이션은 Java와 Spring(이 부분은 지금도 마찬가지).

메시지 처리는 보통의 polling 이었다.
클라이언트가 서버에 fetch 하고,
데이터가 없으면 n초 중단하다가 다시 요청.
메시지를 보내면 최대 n초 지연되어 버리므로, n초 기다리는 동안에 push 통지를 받으면 요청하도록 하였다.

Polling+Push통지의 문제점

push의 지연
push 구조가 외부에 있어서 조절할 수 없어서 메시지 앱에서 메시지를 지연하는 것과 그 부분을 통제할 수 없는 것이 치명적이었다.

쓸데없는 request-response
polling 할 때마다 쓸데없는 request-response이 발생. **밧데리를 소비한다**.
그래서 Push 통보로 바뀌고 long polling을 구현하여 해결했다.

Push 통지에 의한 Long polling

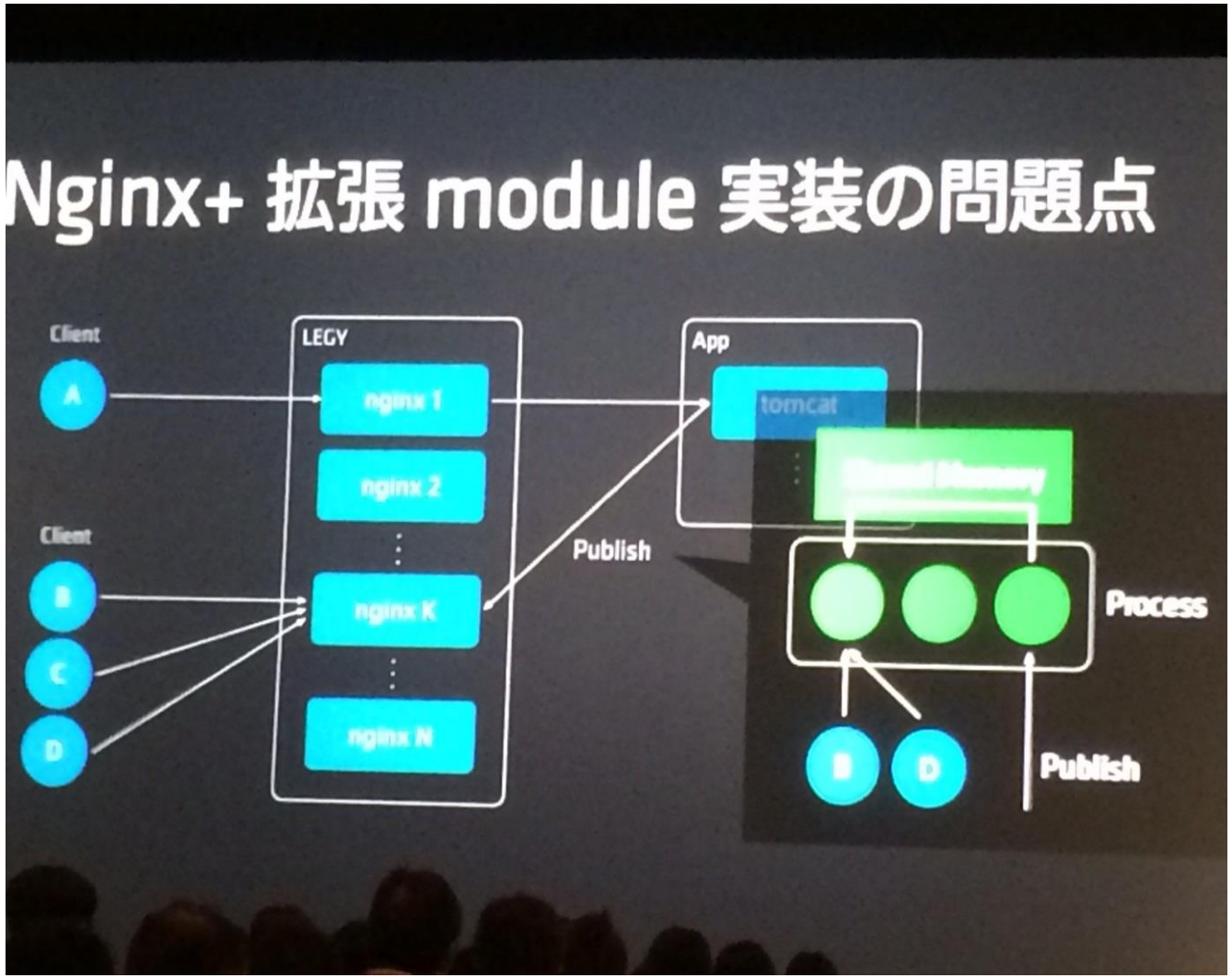
클라이언트와 서버 사이에 Gateway 층을 두고, 클라이언트는 Gateway에 메시지가 있는지 문의한다.

메시지가 없으면 Gateway는 대기, 메시지가 오면 서버에서 Gateway에게 그것을 Publish한다. Gateway에서 서버로 메시지를 fetch 한다.

클라이언트가 기다린다는 것은 변하지 않았지만 이렇게 실시간성을 확보했다.

발매 후 3개월 정도로 Apache를 사용하던 층을 Nginx+확장 모듈로 long polling 하는 층으로 바꾸었다.

segmentation fault 개선: Erlang제 Gateway 도입



2012년 1월에 1000만 유저를 넘어섰을 무렵에 segmentation fault이 빈발.

프록시가 죽어서 대처해야 했음.

문제는 공유 메모리.
클라이언트 A가 클라이언트 B에 메시지를 보낼 때 서버는 클라이언트 B의 접속을 가지고 있는 프로세스에 Publish 하는데 그 부분에서 공유 메모리를 사용해야 했다.

공유 메모리에 문제가 있어서 에러가 발생.

여기에 Erlang 도입.
포인트는 병행성, 분산, 고급, 무 정지 배포.

Erlang은 경량 프로세스를 많이 실행하고 메세지 패싱으로 동시성을 확보하기 때문에 문제가 되는 부분의 처리가 잘 되었다. 게다가, 병행성 처리를 아키텍처 깊은 곳에 견고하게 은폐할 수 있었다.

분산 처리도 하기 쉬웠다. C 언어에 비해서 추상도가 높고, 고 레이어 프로그래밍을 할 수 있었다.

VM을 재 기동하지 않고도 코드를 갱신할 수 있는 hot swap이 있었다. 서비스 정지를 할 수 없는 메시지 서비스에서 무 정지 배포를 할 수 있다는 것은 큰 이점이었다.

그래서 Erlang제 LINE Event Delivery Gateway(LINE 자체는 **LEGY**라고 부른다)을 도입했다. 메시지 이외에도 읽은 상태 통보, 친구가 프로필을 갱신했는지 통지 등의 이벤트를 메시지처럼 딜리버리 하므로 '이벤트 딜리버리'라고 부른다.

Erlang으로 다시 만들어서 로직을 컨트롤할 수 있는 형태로 구현 할 수 있도록 되어서 행복하였다.

connection이 너무 많은 문제를 개선: SPDY 도입

2012년 7월 클라이언트와 서버 간에 connection이 너무 많은 문제가 발생.
문제는 여러 있었지만 이번에는 그 중 프로토콜 레벨에서 connection 수를 줄이는 개선을 한 것을 이야기 한다.

connection이 이렇게 늘어난 것은 HTTP로 long polling connection과 기타 API 용으로 언제나 2개 이상의 connection을 걸고 있었다.

이에 대해서 HTTP(S)에서 SPDY로 프로토콜을 변경했다. SPDY는 당시 Google에서 제안한 draft 프로토콜이었다. 그러나 LINE이 직면했던 과제를 거의 다 해결한 프로토콜이어서 채용하였고 LEGY와 클라이언트 애플리케이션에 구현했다.

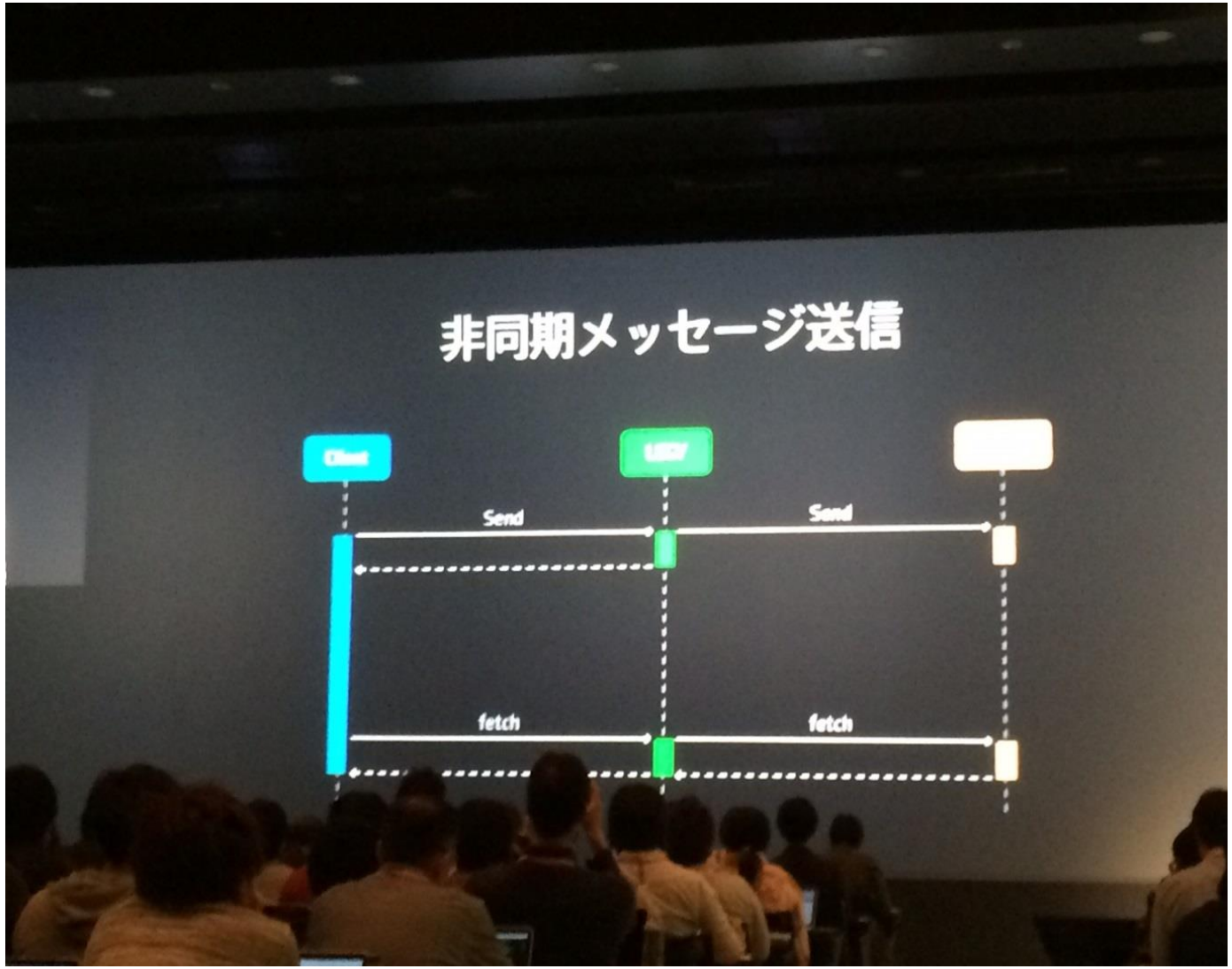
2015년에 HTTP2라는 이름으로 널리 알려지게 된 것처럼 하나의 connection 중에서 복수의 connection을 다룰 수 있다.
또, 헤더 압축 기능이 있어서 통신량을 줄일 수 있다.
LINE은 모바일 통신량이 적은 환경에서도 이용되고 있으므로 이것은 유저 체험 향상에 효과가 나왔다.
결과적으로 connection수는 절반~3분의 1로 삭감할 수 있었다.

해외에서 성능 추구: 비동기 메시지 송신 도입

2012년 10월 해외에서의 성능을 개선키로 했다.
LINE의 대부분의 서버가 메인 데이터 센터에 존재한다. 먼 곳에 있는 유저는 네트워크 레이턴시나 대역의 영향을 직접 받고 있었다.

소프트웨어 논리로는 해결하기 어려운 것으로 글로벌 거점에 데이터 센터(POP 이라고 부른다)를 갖추고 로드 밸런서와 LEGY를 설치했다.
클라이언트는 근처의 LEGY를 발견하고 통신한다.
LEGY와 메인 거점 사이는 전용선으로 접속했다.

그러나 전용선이라고 말하지만 거리가 멀면 레이턴시는 역시 커진다.
그래서 비동기 메시지 송신을 도입했다.



1. LEGY는 백엔드에 Send 한 후 응답을 기다리지 않고 클라이언트에 응답을 돌려준다.

2. UI을 경신해서 사용자가 다음 액션을 취하도록 했다. 이것에 의해 UX을 향상시켰다.

3. Backend에서 통신 에러가 났을 때 복구할 수 없으므로 LEGY는 아까의 백엔드에 대한 Send의 결과를 받아 클라이언트에 돌려줘서 에러는 클라이언트 측에서 커버한다.

메시지 기반은 이렇게 진화해 왔다. 현재도 MySQL이 HBase로 바뀐 것 외에 초기 아키텍처 상태이다. 세계 규모로 메시징을 처리하고 있지만 그림으로 하면 단순하다.

LINE류 Microservice

LINE은 서비스가 성장하면서 스탬프나 타임 라인 등의 기능을 추가했다.
그것들을 어떻게 구현했는지 이야기 하겠다.

LINE은 급속도로 유저를 획득한 것도 있고, 사용자 요청이 많고, 새로운 기능에는 늘 스피드, 기능성, 품질 모든 것이 요구되는 추세다.

이에 대응하는 개발 스타일을 실천하고 있다.

LINE식 개발 스타일

MONOLITHIC 이 아닌 MICRO SERVICE.
MONOLITHIC 시스템에서는 우수한 사람이 개발에 참여하여도 우선 전체를 파악하지 못하면 새로운 서비스 개발에 착수할 수 없다. 서비스들을 API로 연결시킴으로써 신규 참여한 개발자도 금방 새 기능 개발에 착수할 수 있도록 했다.

구체적으로는 Talk-server(메시징 기반 어플리케이션 서버)에는 메시징과 소셜 그래프의 기능만 구현되어 있다. 기타 모든 기능은 다른 서비스로 동작하고 다른 팀이 개발하고 있다.

개발 프로세스나 배포 흐름도 서비스마다 독립되고 개발 언어와 백엔드의 미들웨어도 가지각색이다. 예를 들면 Talk-server는 Java지만 계정 관리는 Scala. 백 엔드의 핵심은 HBase와 Redis지만 서비스에 따라서는 Mongo와 Cassandra를 사용하고 있다.

가령 다음과 같은 각 서비스를 독립적으로 개발하고 API로 잇고 있다.
- 인증 관리
- abusing 검출
- 이벤트 처리
- push 통지 기반
- CMS

그래서 전체적으로는 간단하게 개발할 수 있다.

개발 연구

서비스를 나눠서 개발 하기 위해서 연구했던 3개를 소개한다.

1. 조직
팀을 독립시켜도 각각에게 재량을 주지 않으면 커뮤니케이션 비용이 늘거나, 담당이 누군지 모르는 일이 생겨서 개발 효율이 오르지 않는다.
그래서 조직도나 회사의 특별한 팀을 편성했다.
단기간으로 이테레이션을 돌리고 목적을 달성할 경우 축소 혹은 해산한다.

2. 프로토콜 관리
Apache Thrift, Protobuf, REST 채용
간단하게 하려고 wiki에 API을 써 놓고 그것을 기반으로 개발하려는 경향이 있다. 그것을 피한다.

상기 중 Thrift을 이번에 설명한다.
Thrift는 인터페이스 정의 언어(IDL). Thrift에서 서비스의 인터페이스를 기술한다. Thrift는 기술된 것에서 각종 언어의 Stub을 출력한다. 이에 의해서 인터페이스를 명확히 정의한다. IDL은 텍스트 기반이므로 API(서비스) 스펙에 관한 토론을 Github 상에서 Pull Request 베이스로 할 수 있게 된다. 미묘한 API를 난립시키면 전체적으로 찌그러져서, 이런 곳에서 서비스를 검토한다.

개발 연구

3. 설비(인프라)
개발 팀이 많이 생기므로 공통으로 필요한 것을 준비하지 않으면 모두가 똑같은 것을 만드는 일이 발생해서 전체적으로 개발 효율이 떨어진다.

공통 개발 흐름에 따른 설비를 준비한다.
Github Enterprise, Jenkins, 사내 배포 도구 PMC(브라우저 베이스로 GUI로 출시 할 수 있다), 사내 서비스 모니터링 도구 IMON(뒤에서 다양한 지표를 가시화할 수 있는)

이러한 개발 스타일을 시작할 즈음에는 아직 Microservice라는 말이 없어 일그러진 부분이 남아 있다. 그러나 Microservice의 이점으로 알려진 것은 누릴 수 있다.

조직이 커짐으로써 개발 효율도 올리는 것도 중요하다고 생각한다.

향후의 과제

멀티 데이터 센터
해외에는 프런트엔드 서버만 있다가 나중에는 아무래도 메인 데이터 센터에 연결할 필요가 있다.
앞으로는 해외 현지에서 완결된 서비스를 받도록 하고 싶다.

Microservices의 발전
Talk-Server가 꽤 거대하게 되어서 분할한다.
또 어떤 서비스가 떨어졌을 때 전체적으로 어떻게 되는지를 관리하는 장애물 설계를 앞으로 더 정비할 필요가 있다.