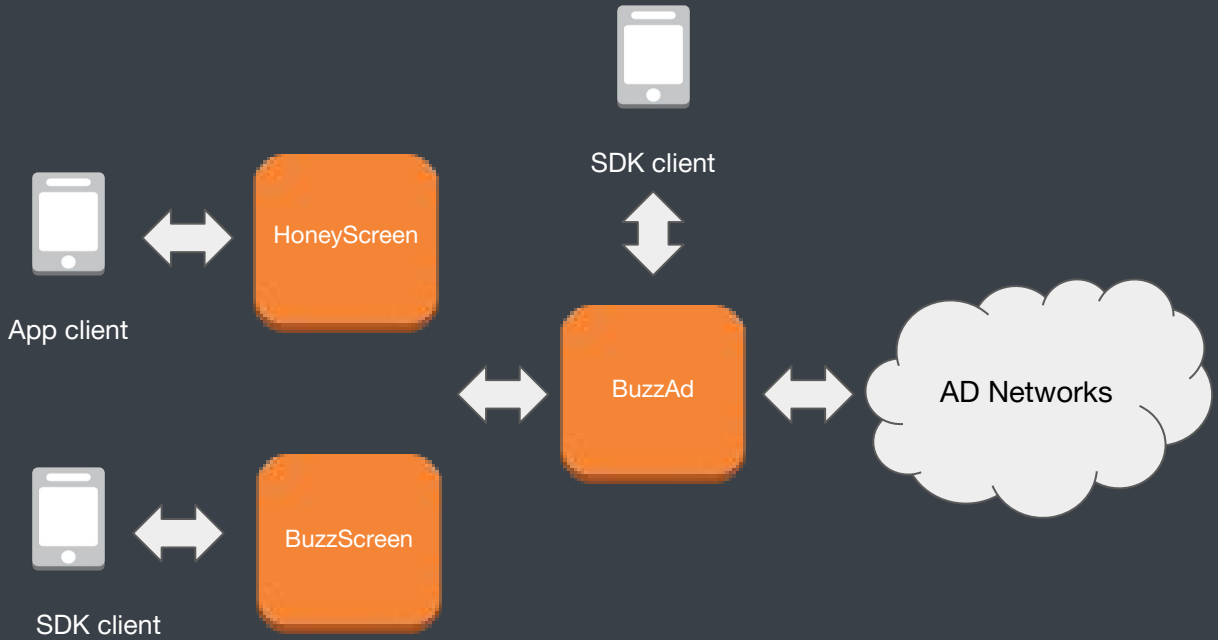
Django에서의 대용량 트래픽 처리 - 병목을 찾아라

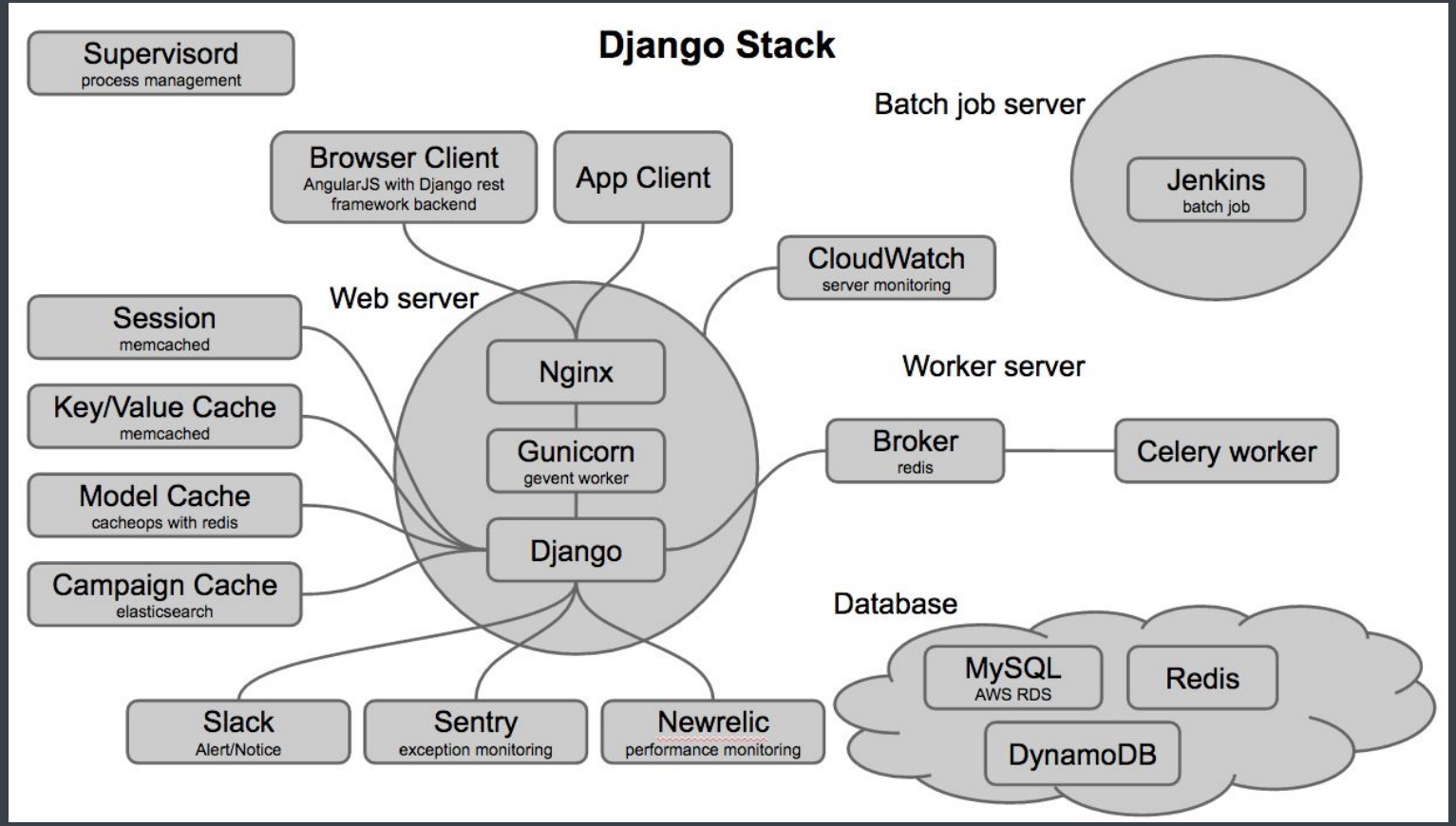
서주은 - zuneseo@buzzvil.com

버즈빌의 서비스

- 허니스크린(잠금화면 포인트앱), 버즈스크린(잠금화면 SDK), 버즈애드(광고 플랫폼)
- AD Network과의 통신이 빈번해 일반적인 Web Application보다 더 I/O intensive



Django를 활용한 서버 스택



어느날 갑자기 장애 발생

- 웹 서버 로그 체크
- 병목을 찾자
  - 성능 문제로 판단되는 경우
- 대부분의 병목은 데이터베이스 문제로 인해 발생
- **MySQL** 상태 확인
  - **Process list**
  - **CPU**
  - **Memory**
  - **Disk I/O**

전부 멀쩡하다

CPU Usage Saturation

예상 할 수 있는 문제들

- 웹 서버 프로세스의 개수 < 시피유 코어 개수
  - Python GIL로 인해 thread가 동시에 여러 코어에서 실행되지 못한다
  - 프로세스의 개수를 하드코딩 한 상태에서 더 많은 코어를 가진 인스턴스로 업그레이드를 하는 경우
- Gevent worker 개수가 충분하지 못함
  - 모든 worker가 I/O 응답 대기중
- Gevent가 제대로 동작하고 있지 않을 가능성

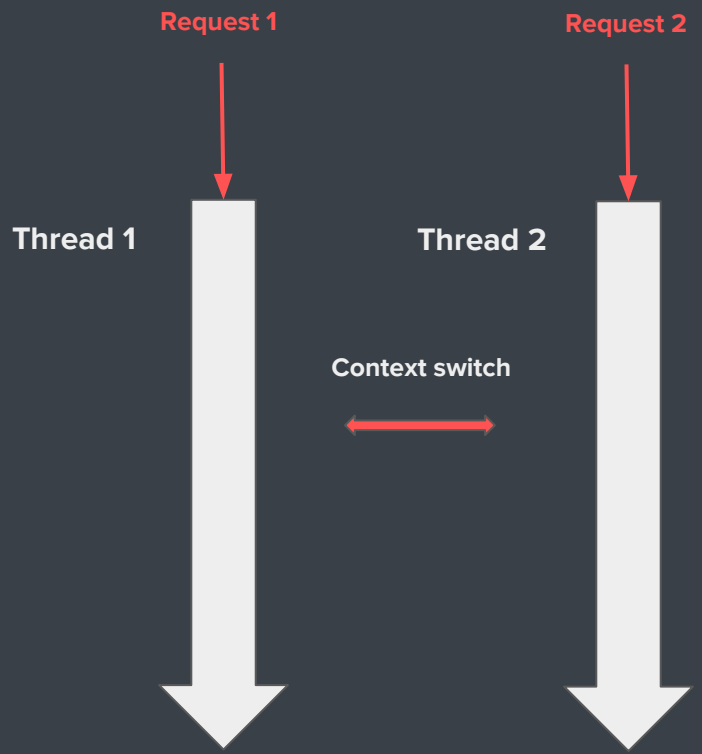
Gunicorn with gevent worker

Gevent is a framework for scalable asynchronous I/O with a fully synchronous programming model.
(http://mauveweb.co.uk/posts/2014/07/gevent-asynchronous-io-made-easy.html)

- **Gunicorn**
  - **Gunicorn 'Green Unicorn' is a Python WSGI HTTP Server for UNIX.**
  - **Gevent worker 지원**
- **Gevent**
  - **Coroutine-based Python networking library**
  - 동기방식 프로그래밍 모델로 짜여진 로직을 수정 없이 비동기 I/O로 동작

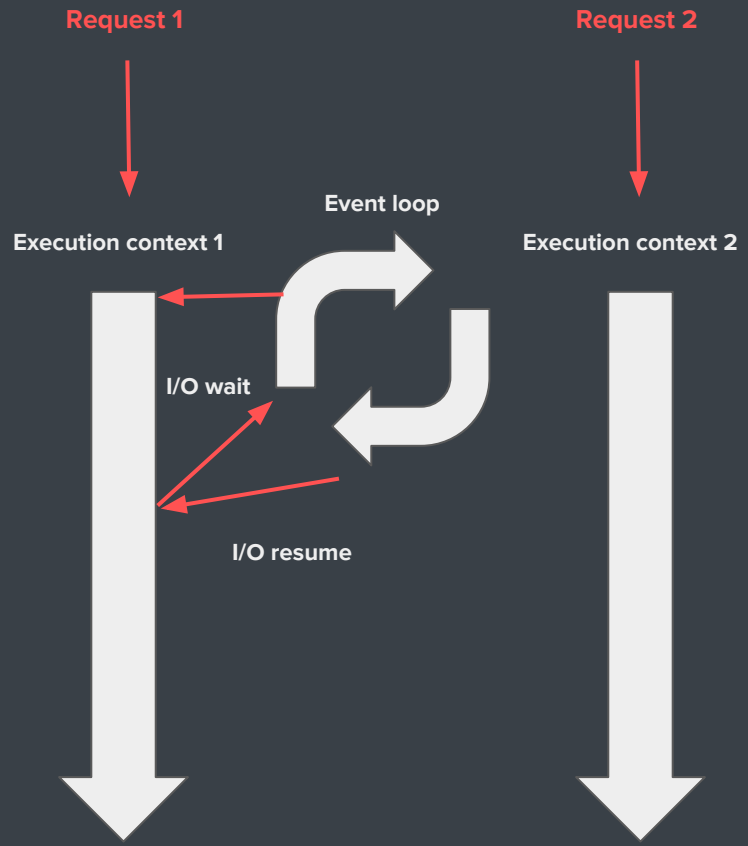
Thread per request

- HTTP 요청마다 별도의 Thread를 사용
- 메모리등 시스템 자원을 더 많이 소모
- Context switch
  - 오버헤드가 있다
  - OS-level context switch
  - Blocking I/O 호출 또는 인터럽트 등에 의해서 발생



Multiple requests with single thread

- I/O가 많은 경우 성능상 이점
- Non-blocking I/O를 사용해야함
- 코드가 복잡해짐
  - 하나의 이벤트루프에서 I/O wait을 하는 형태로 코드가 구성됨
- 여러가지 프로그래밍 모델
  - Callback 기반
  - Co-routine 기반
- 일반적으로는 프로그래밍 스타일에 변화가 필요



Multiple requests with gevent

- 기존의 Non-blocking 코드를 그대로 활용
- Context-switch
  - Blocking I/O 호출 시
  - 명시적으로 yield를 하는 경우
- Monkey-patch 필요
  - Blocking 함수를 이벤트 루프 기반의 Non-blocking 함수로 바꾸고 다른 context로 넘어갈 수 있도록 함

Patch된 socket recv 함수

```
def recv(self, *args):
    sock = self._sock  # keeping the reference so that fd is not
    while True:
        try:
            return sock.recv(*args)
        except error as ex:
            if ex.args[0] != EWOULDBLOCK or self.timeout == 0.0:
                raise
            # QQQ without clearing exc_info test__refcount.test_c
            sys.exc_clear()
        self._wait(self._read_event)
```

Patch된 recv가 호출되는 stack trace

```
/python2.7/httplib.py", line 1132, in getresponse\n    response.begin()\n',
/python2.7/httplib.py", line 453, in begin\n    version, status, reason = self._read_status()\n',
/python2.7/httplib.py", line 409, in _read_status\n    line = self.fp.readline(_MAXLINE + 1)\n',
/python2.7/socket.py", line 480, in readline\n    data = self._sock.recv(self._rbufsize)\n',
ges/gevent/_socket2.py", line 280, in recv\n    self._wait(self._read_event)\n',
ges/gevent/_socket2.py", line 179, in _wait\n    self.hub.wait(watcher)\n',
ges/gevent/hub.py", line 627, in wait\n    result = waiter.get()\n',
ges/gevent/hub.py", line 875, in get\n    return self.hub.switch()\n',
ges/gevent/hub.py", line 606, in switch\n    return greenlet.switch(self)\n',
```

예상 할 수 있는 문제들

- 웹 서버 프로세스의 개수 < 시피유 코어 개수
  - Python GIL로 인해 thread가 동시에 여러 코어에서 실행되지 못한다
  - 프로세스의 개수를 하드코딩 한 상태에서 더 많은 코어를 가진 인스턴스로 업그레이드를 하는 경우
- Gevent worker 개수가 충분하지 못함
  - 모든 worker가 I/O 응답 대기중
- Gevent가 제대로 동작하고 있지 않을 가능성

Monkey Patch!

Gevent가 실제로 동작하게 만들기

- I/O가 필요한 라이브러리는 순수하게 Python으로만 구현된 것을 사용
- C-based python I/O libraries
  - MySQLdb
  - PyLibMCCache
- Pure python I/O libraries
  - PyMySQL
  - python-memcached

자신이 사용중인 패키지를 체크해봅시다

- 발병 할 때 까지 무증상
  - Gevent 호환이 안 된다 != 라이브러리가 동작하지 않는다
  - 다만 C 코드 안에서 blocking I/O가 호출 될 시에 실제로 blocking이 될 뿐
  - 웹 서버/데이터베이스 등의 상태가 정상이기 때문에 방심하기 쉽다
- 발병 시 원인 찾기가 어려움
  - Python 서버는 여유롭게 blocking된 상태로 기다리는 중
  - 애꿎은 nginx에서만 python 서버 응답을 기다리다가 에러

이번엔 워커 장애 발생

- 메세지 브로커로 사용중인 redis의 큐에 메세지가 쌓기이 시작함
- CPU saturation 현상 재발생
- I/O와 관련된 부분 체크
  - Memcached 서버
  - 외부 서버로의 http request 요청

역시 전부 멀쩡하다

Strace

- 의심 가는 부분은 I/O 인데 디버깅을 위한 힌트는 어디에서 얻을까
- I/O요청 -> System call 발생이므로 strace를 이용해 워커의 동작을 모니터링
- > sudo strace -p 30588

Strace

- 의심 가는 부분은 I/O 인데 디버깅을 위한 힌트는 어디에서 얻을까
- I/O요청 -> System call 발생이므로 strace를 이용해 워커의 동작을 모니터링
- > sudo strace -p 30588

```
getppid()                                = 22255
clock_gettime(CLOCK_MONOTONIC, {17305924, 483709534}) = 0
clock_gettime(CLOCK_MONOTONIC, {17305924, 483762337}) = 0
epoll_wait(8, {{EPOLLIN, {u32=24, u64=103079215128}}}, 64, 999) = 1
clock_gettime(CLOCK_MONOTONIC, {17305924, 484295365}) = 0
recvfrom(24, "\7\0\0\1\0\0\0\0\0\0", 8192, 0, NULL, NULL) = 11
sendto(24, "\23\0\0\0\3SET AUTOCOMMIT = 1", 23, 0, NULL, 0) = 23
recvfrom(24, 0x43fb2a0, 8192, 0, 0, 0)  = -1 EAGAIN (Resource temporarily unavailable)
clock_gettime(CLOCK_MONOTONIC, {17305924, 484609362}) = 0
epoll_wait(8, {{EPOLLIN, {u32=24, u64=103079215128}}}, 64, 999) = 1
clock_gettime(CLOCK_MONOTONIC, {17305924, 486664220}) = 0
```

자네, 내가 언제 디비 접속하라고 한 적 있는가?

Celery

- **Message broker**
  - 메세지 전달을 위한 브로커 설정
  - **RabbitMQ/Redis/MySQL** 등을 지원함
  - **BROKER_URL**설정을 통해 **redis**를 사용하도록 명시
- 혹시나 워커 로직에서 **MySQL**접속을 하는 곳이 있는지 다시한번 체크
- **Result backend**
  - 실행한 태스크의 결과를 저장하는 기능
  - **MySQL**로 **result backend**가 설정되어 있었음
  - 일반적인 경우 필요없음
  - **CELERY_IGNORE_RESULT = True** 로 설정하자

교훈

- 문서를 자세히 읽자
- 병목을 찾기 위한 노력
  - CPU가 놀고 있다 -> 어딘가 병목이 있다
  - Gevent에 대한 이해 필요
- 힌트는 여러군데 있었음
  - “제약을 넘어: Gevent” - 2014 PyCon Korea
  - “Celery의 빛과 그림자" - 2015 PyCon Korea

저는 눈 뜬 장님이었습니다

Django ORM 캐싱

- 데이터베이스의 쿼리 결과를 캐싱하여 성능 향상
- 최대한 거저 먹고 싶다
  - 별도의 추가 코드 없이 transparent하게 사용 가능한 라이브러리 조사
  - Transparent => 이용하기 쉽다 & 의존성이 없다 => 나중에 갈아타기 쉽다
- 후보군
  - Johnny cache - 쓰다가 개발이 중단되어서 버림
  - Cacheops - 현재까지 유지보수가 잘 진행되고 있어 잘 사용중

Cacheops ORM 캐싱

- 기존 코드 변경 없이 **Transparent**하게 사용 가능
  - Article.objects.filter(tag=2)
- **cache()** 함수를 이용해 수동으로 캐싱 선택 가능
  - Article.objects.filter(tag=2).cache()
- 캐싱 **timeout** 지정
- 오퍼레이션 별**(get, fetch, count, exists)**로 캐싱 여부 지정 가능

```
CACHEOPS_DEFAULTS = {
    'timeout': 60*60
}
CACHEOPS = {
    'auth.user': {'ops': 'get', 'timeout': 60*15},
    'auth.*': {'ops': ('fetch', 'get')},
    'auth.permission': {'ops': 'all'},
    '*.*': {},
}
```

Cache Invalidation

- **Join사용시 invalidation**
  - 해당 join query에 연관된 row가 변경된 경우 모두 추적하여 invalidation 필요
  - 케이스가 다양하므로 invalidation 로직이 복잡해진다
- **Network등의 문제로 DB에는 write가 성공하였으나 캐시 invalidate이 실패한 경우**
- **캐시업데이트가 된 이후 transaction rollback 발생하는 경우**
  - row-1 update -> cache invalidate -> row-1 select -> cache update -> unexpected error -> transaction rollback
- **Django ORM을 통하지 않고 직접 데이터베이스를 변경하는 경우**
- **완벽하게 데이터베이스와 캐시의 일관성을 유지하는 것은 쉽지 않다**

Caching & Invalidation 전략

어차피 캐시 일관성을 보장하는게 쉽지 않다면 일관성이 100% 보장되지 않더라도 괜찮은 로직에만 캐싱을 활용한다. 그리고 캐시 expire시간을 1분 정도로 짧게 가져가 invalidation이 실패하더라도 최대 1분안에는 일관성이 성립되도록 한다. 이로인해 캐싱 가능한 데이터의 범위가 줄어들 수는 있지만 일관성 보장에 대한 걱정을 할 필요가 없다.

- 직접 캐싱 로직을 관리하지 않고 Cacheops라는 라이브러리 사용
- Invalidation이 어느정도까지 되는지 파악하기가 쉽지 않다
  - Join, Transaction 처리
- 라이브러리의 Invalidation 로직에 버그가 있을 가능성이 높다
  - 버그가 있다면 매우 발견하기 힘듦
- 쉽게 쓰려고 도입했는데 최대한 쉽게 쓸수 있는 전략을 선택
  - Expire time을 짧게 설정
  - 최종 일관성만 보장되면 괜찮은 로직에만 캐싱을 도입

Cacheops Function 캐싱

- Decorator를 이용해 쉽게 함수의 리턴값을 캐싱
- Model dependent한 함수를 위해 model 변경 시 invalidation 하도록 지정 가능
- @cached_as 인자와 함수의 인자값에 의해 key가 결정됨

```
@cached_as(Article, timeout=120)
def article_stats():
    return {
        'tags': list(Article.objects.values('tag').annotate(Count('id')))
        'categories': list(Article.objects.values('category').annotate(Count('id')))
    }
```

Cacheops 기타 기능

- local_get
  - 캐싱 결과를 프로세스 메모리에 저장
  - 매우 빠르다
  - Invalidation을 안하므로 극단적으로 바뀔 일이 없는 데이터에 대해서만 적용
- View Caching
- File Cache
  - Cache backend로 파일 사용
- Django template integration

요약

- 성능 문제 발생 시 병목을 찾는 것이 중요하다
  - CPU, Memory, Disk IO, Network 등을 모니터링 하고 상황 파악
  - 단순히 시스템 자원의 성능문제가 아닐수도 있다
- Gevent를 worker로 사용하는 경우 Blocking I/O를 하는 C 라이브러리를 사용중인지 꼭 체크
- Celery의 result backend의 존재에 대해 알고 있어야 함
- Django ORM 캐싱으로 cacheops를 추천
- 라이브러리 사용시 문서를 꼼꼼히 읽자

감사합니다

Q&A

WE ARE HIRING!