공학
트렌드

소프트웨어공학센터 경영지원TF팀

대용량 서비스를 지탱하는 분산 캐시 시스템
Part 3 : Memcached 분산 캐시 시스템의 소개

2015. 3. 3. [124호]

Contents

Ⅰ. Memcached 서버와 API 소개
Ⅱ. Memcached 운영

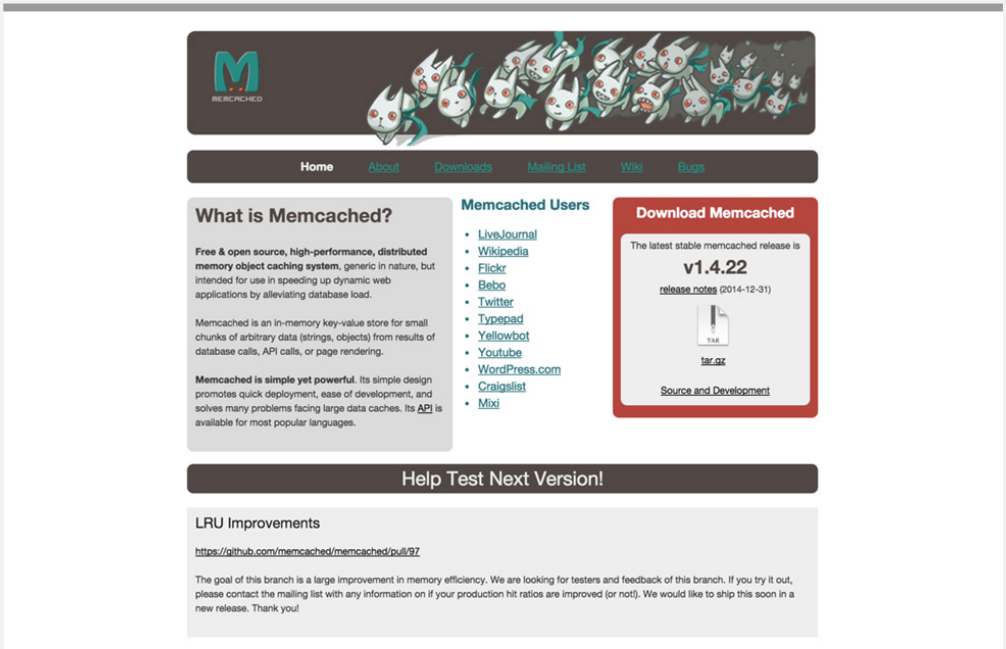
정보통신산업진흥원 부설
nipa 소프트웨어공학센터

Ⅰ. Memcached 서버와 API 소개

1.1 소개

Memcached는 메모리(memory)로 캐시하는 것(cached)의 합성어이다. Memcached는 Danga Interactive 사(社)가 운영하는 LiveJournal.com의 속도를 높이기 만든 분산 캐시 시스템이다. BSD 라이선스(Revised BSD License)를 가진 오픈소스이다. 2003년도에 Perl로 만들어졌다가 추후 C로 다시 개발이 되었다. libevent 라이브러리 의존성이 있어서 해당 라이브러리가 먼저 설치되어 있지 않으면 정상적으로 설치할 수 없다. 〈그림 1〉은 Memcached 홈 페이지로서, 문서와 다운로드, 위키, 버그 리포트로 이루어져 있다.

그림 1_Memcached 홈페이지 메인 화면

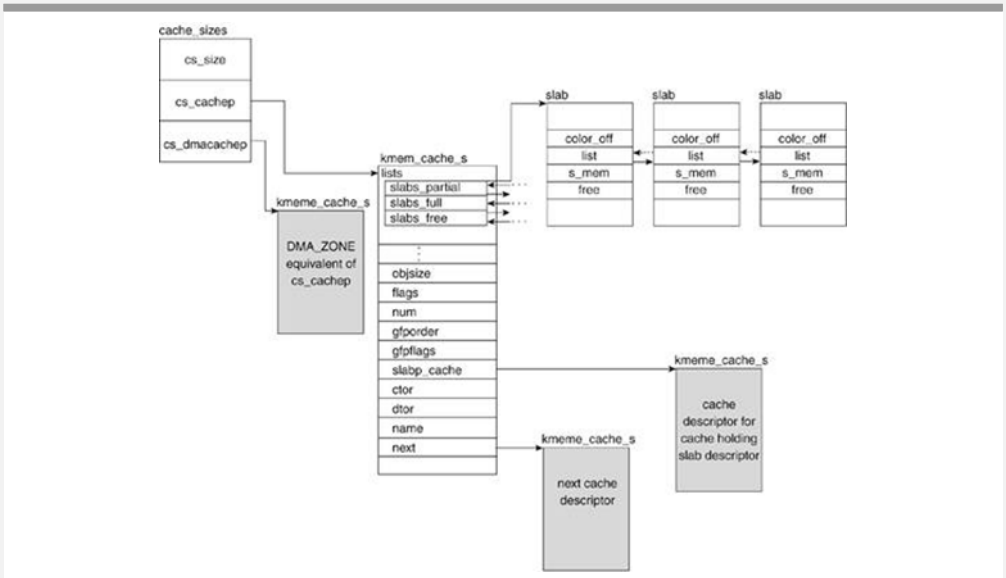


출처 : http://memcached.org/

Memcached는 LiveJournal 뿐 아니라, 위키피디어, 플리커, 트위터, 유튜브, 징가, 텀플러, 워드프레스, 페이스북에서 사용하고 있으며 구글 웹엔진, 믹시, 하테나, 아마존 등과 같은 클라우드 서비스에서 애플리케이션 서버의 캐시 시스템을 확장하고자 할 때 사용할 수 있는 컴포넌트(Component)로 쓰이고 있다. Memcached는 키-값(Key-Value) 구조

로서 메모리에 저장하는 해시 테이블(hash table)을 포함한다. 만약 테이블이 가득 차면, 오래된 데이터가 가장 먼저 사라지는 LRU(Least recently used) 방식을 사용한다. Memcached는 다른 분산 캐시와 달리 특이한 점을 가지고 있다. 슬랩(slab) 메모리 할당자를 두어 메모리 관리를 효율적으로 하고 있다는 점이 큰 장점이다. Uptime(업타임, 운영시간)이 오랫동안 지속되면 메모리 할당/해제로 인해서 메모리 단편화 현상이 일어나는 현상이 있다. 이를 위해서 보다 효율적인 슬랩 메모리 할당자를 이용하여 메모리 단편화를 최소화시킬 수 있다. 슬랩 메모리 할당자의 핵심을 간단히 언급한다. 슬랩은 고정된 크기로 묶여 있는 연속된 메모리의 블럭의 하나를 의미한다. 슬랩 할당자는 메모리를 상태로(다 사용, 부분적 사용, 비어 있는 것) 관리한다. 자주 사용하는 데이터를 미리 할당해 놓고 사용자 요구가 있으면 바로 반환할 수 있다. 자주 호출되는 캐시를 따로 만들어서 내부 파편화를 만들지 않게 한다. 따라서 메모리 공간을 효율적으로 할당하기 때문에 빠르다. 〈그림 2〉는 상기에서 설명한 슬랩 할당자 구현 설명이다. 슬랩 리스트에 partial, full, free 리스트를 관리하고 있음을 확인할 수 있다.

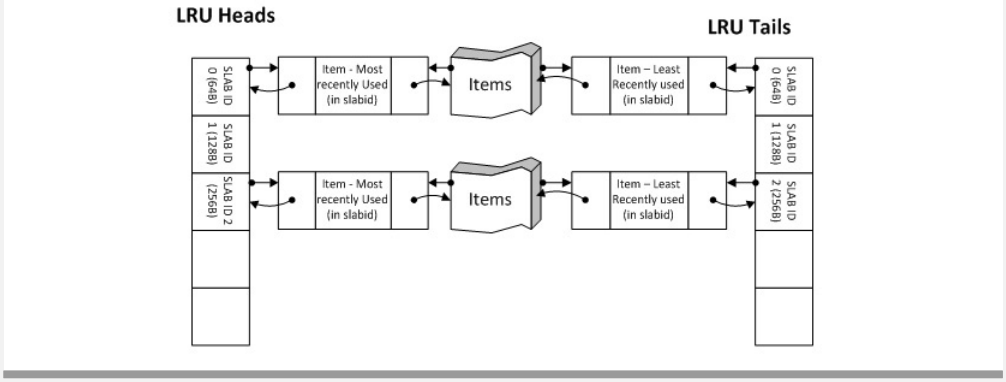
그림 2_슬랩 할당자 내부 구조



출처 : http://flylib.com/books/en/4.454.1.55/1/

〈그림 3〉은 Memcached가 슬랩 별로 LRU를 관리하는 구조이다. 자주 사용되는지 안되는지를 리스트로 관리하는 구조이다. 메모리가 부족하면 LRU Tails에서부터 검색해서 메모리를 확보하게 된다. 단점은 유효시간(expire)으로 인해서 이미 삭제가 되었으나, LRU Heads에 가까운 캐시 데이터들을 정리하지 못하는 부분이 있었는데, 이를 해결해 주는 LRU Crawler가 1.4.18 부터 추가되었다.

그림 3_Memcached의 LRU 자료 구조



출처 : https://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0

Memcached의 슬랩 메모리 할당자와 LRU 데이터 관리 기능이 탁월하다. 데이터가 많고, 빈번하게 캐시 데이터가 삭제가 일어나는 경우에 Memcached는 메모리 부족 현상이 Redis에서 비해서 적다고 할 수 있다. 좀 더 자세한 Memcached 내부 구조를 살펴보기 위해서는 "Memcached Internals" [1]을 참조한다. Memcached는 최적화 하려는 시스템이기 때문에 최대값이 있다. 키의 최대 사이즈 250 byte(250 chars) 이고 값은 1MB를 넘어설 수 없다. 슬랩 메모리(Slab Chunk)는 디폴트로 최대 1MB로 되어 있다. Memcached 실행 시 -I 옵션을 아규먼트(argument)로 주어 1KB에서 128M가지 슬랩 청크 크기를 지정할 수 있다. 또한 최대 힙 메모리(heap memory)의 디폴트 크기는 64MB이다. Memcached 실행 시 -m 옵션을 주어 최대 힙 메모리 크기를 변경할 수 있다. Memcached 클라이언트는 Memcached[2]에서 공개했다. C/C++에서는 libmemcached 라이브러리가, Java에서는 spymemcached 라이브러리가 유명하다. Memcached가 이식성이 좋아서 다양한 제품군으로 쓰이기 한다. Mysql(Innodb) 앞에 두어 Memcached를 활용[3]하여 성능을 극대화할 수 있다. 비슷하게 웹 서버 오픈 소스인 Nginx에 Memcached를 붙여 웹 캐시 서버로 활용할 수 있다.[4] 또한, CouchDB[5]에 Memcached의 소스를 하나로 합쳐서 Couchbase[6]라는 제품이 나와 있다. Memcached는 캐시 데이터를 저장할 수 없다. 이를 보완한 제품이라 할 수 있다. 한편, Memcache의 가장 큰 약점은 클러스터링, 데이터 구조의 한계가 있는데, 이를 보완하기 위해서 naver에서는 zookeeper와 memcached를 활용한 Arcus[7]를 오픈소스화 했다.

1) https://docs.google.com/presentation/d/1H2ekfqdtAYmQjhgmQ4ll1f2lYLSsfU1S5w_FQOWzpe0/edit#slide=id.p
2) https://code.google.com/p/memcached/wiki/Clients
3) https://blogs.oracle.com/mysqlinnodb/entry/get_started_with_innodb_memcached
4) http://nginx.org/en/docs/http/ngx_http_memcached_module.html
5) http://couchdb.apache.org/
6) http://www.couchbase.com/

1.2 설치

Memcached 설치를 설명한다. 먼저 memcached의 의존성 라이브러리인 libevent를 설치한다.

```
$ wget --no-check-certificate https://github.com/downloads/libevent/libevent/libevent-2.0.20-stable.tar.gz
$ tar xvfz libevent-2.0.20-stable.tar.gz
$ ./configure
$ make
$ sudo make install
```

그리고, memcached 최신 버전은 1.4.22를 설치한다.

```
$ wget http://memcached.org/files/memcached-1.4.22.tar.gz
$ tar -zxvf memcached-1.x.x.tar.gz
$ cd memcached-1.4.22
$ ./configure
$ make
$ make test
$ sudo make install
```

Memcached의 실행 파일은 memcached는 /usr/local/bin/ 디렉토리에 생성된다. memcached 를 실행하면 memcached 가 실행한다. 특별히 로그가 보이지 않아도 실행 중이다.

```
$ memcached
```

디폴트로 Memcached는 64MB 메모리를 차지한다. 그리고 11211 포트를 기본 포트 사용하고 동시 1024 클라이언트를 처리한다. 캐시 데이터를 간단하게 저장하고 얻어오는 예제를 소개한다. telnet으로 기본 포트로 접속한다.

```
$ telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Memcached에 key라고 하는 데이터를 저장한다. 먼저 set key 명령어와 flag, 유효 시간, 데이터 크기를 입력하고 엔터를 입력 후, data 라는 값을 입력하고 엔터를 입력하면 "STORED" 라는 응답을 얻을 수 있다.

```
set key 0 900 4
data
STORED
```

7) http://naver.github.io/arcus/

Memcached에 key라는 데이터를 읽어오려면 get key 를 입력한다. 응답으로 flag, 길이를 얻고 저장 값인 data 값을 출력한다. 그리고 마지막으로 "END" 를 출력한다.

```
get key
VALUE key 0 4
data
END
```

1.3 커맨드 종류

Memcached는 키-값만 저장하는 구조이기 때문에 커맨드는 단순하다.

1.3.1 검색 커맨드

데이터를 가져오는 커맨드로서 get, gets가 있다. get과 gets 커맨드의 차이는 cas (check and set) 토큰을 얻어 올지를 분류하는 차이밖에 없다.

get 커맨드는 데이터가 저장되는 키의 값을 얻어온다.

```
get key
```

get 커맨드의 응답은 다음과 같다. VAULE의 요청 포맷은 "VALUE 〈key〉 〈bytes〉WrWn 〈data〉" 이다. 자세한 내용은 아래 저장 커맨드에서 설명한다. 저장된 정보를 그대로 보여주고 END 출력 후 완료한다.

```
VALUE key 0 2
11
END
```

gets 커맨드는 데이터가 저장되는 키의 값과 cas 토큰 값을 함께 보여준다. cas 토큰에 대한 자세한 내용은 cas 커맨드에서 볼 것이다.

```
gets key
```

gets 커맨드의 결과는 get 커맨드의 결과에 cas 토큰이 추가되었다.

```
VALUE key 0 2 7
11
END
```

1.3.2 저장 커맨드

저장 커맨드는 set, add, replace, append, prepend, cas가 있다.

파라미터는 다음의 설명을 따른다.

- key : 데이터가 저장되는 키
- flags : Memcached에서 저장하는 32비트 unsigned integer
- exptime : 유효기간, 단위는 초
- bytes : 데이터 블럭의 바이트 수

리턴되는 값은 다음과 같다.

- STORED : 저장을 알림
- NOT_STORED : 저장되지 않음을 알림
- EXIT : cas 커맨드 사용 시 이미 존재하고 있음을 의미
- NOT_FOUND : 값이 존재하지 않거나 이미 삭제된 것을 의미

set 커맨드 요청은 다음과 같다. 'set 〈key〉 〈flags〉 〈exptime〉 〈bytes〉₩r₩n〈value〉₩r₩n'의 포맷을 따른다.

```
set key 0 900 4
data
```

set 커맨드의 결과 값은 다음과 같다.

```
STORED
```

add 커맨드는 set과 동일한 요청 포맷을 가진다. 중요한 점은 키가 아직 값을 가지고 있을 때 정상적으로 동작한다. 만약 키에 값을 가지고 있는 상태에서 add 커맨드를 실행하면 NOT_STORED 응답을 받을 것이다.

```
add key1 0 900 2
10
```

replace 커맨드는 저장되어 있는 키의 값을 주어진 값의 내용을 변경한다. 예제는 10으로 변경한다.

```
replace key 0 900 2
10
```

append 커맨드는 저장되어 있는 키의 값에 뒤에 주어진 값을 덧붙인다. 예제는 값의 뒤에 0을 더한다. 키의 값이 1이었다면 10이 된다.

```
append key 0 900 1
0
```

prepend 커맨드는 저장되어 있는 키의 값에 앞에 주어진 값을 덧붙인다. 예제를 값의 앞에 9를 더한다. 키의 값이 1이었다면 91이 된다.

```
prepend key 0 900 1
9
```

cas 커맨드는 gets 커맨드로 얻어온 cas 토큰 값으로 값을 저장할 수 있다.

아래 예제에 대한 설명이다. gets 커맨드로 키의 cas 토큰을 얻어오고 cas 커맨드의 끝에 gets 커맨드로 얻어온 cas 토큰을 입력해야 저장이 된다.

```
gets key
VALUE key 0 2 12
20
END
cas key 0 900 2 12
30
```

만약 cas 커맨드의 cas 토큰이 입력된 cas 토큰과 다르면, EXISTS 응답을 리턴받고 값을 저장할 수 없다.

```
cas key 0 900 2 30
40
EXISTS
```

1.3.3 삭제 커맨드

```
delete key
```

delete 커맨드로 삭제를 할 수 있다. delete 〈key〉 의 형식을 가진다.

```
DELETED
```

정상적으로 삭제가 일어나면, 응답 값은 다음과 같다.

1.3.4 증분/감분 커맨드

값이 존재한 상태에서 증분/감분을 incr,decr 커맨드를 이용할 수 있다.

key라는 키에 1이라는 값이 저장되어 있는 상태에서 incr과 decr를 사용한다.

```
incr key 1
```

응답 값은 key 에 1 이 저장된 값이 출력한다.

```
incr key 1
```

1.3.5 통계 커맨드

```
stats
STAT pid 73409
STAT uptime 7171
STAT time 1424064562
STAT version 1.4.17
STAT libevent 2.0.20-stable
STAT pointer_size 64
...
```

stats 명령어는 업타임, 버전, 메모리 정보, 사용자 쓰레드 개수 등을 보여준다.

캐시 히트율(Hit rate)을 확인하려면 stats 명령어의 get_hits와 cmd_get을 살펴본다.

히트율을 얻는 공식은 get_hits를 cmd_get으로 나눈 값이다. 만약 다음과 같이 통계 정보가 나왔다고 가정해보자. 히트율은 34%이다.

```
STAT cmd_get 723212
STAT get_hits 2102124
```

stats items는 아이템 리스트를 출력한다. items 뒤에 나오는 번호가 slab id 이다.

```
stats items
STAT items:1:number 1
STAT items:1:age 337
STAT items:1:evicted 0
STAT items:1:evicted_nonzero 0
STAT items:1:evicted_time 0
STAT items:1:outofmemory 0
STAT items:1:tailrepairs 0
STAT items:1:reclaimed 2
STAT items:1:expired_unfetched 0
STAT items:1:evicted_unfetched 0
END
```

statis cachedump 커맨드는 slab id의 값을 이용하여 키의 일부 값을 저장할 수 있다. 첫 번째 파라미터는 slab id이고, 두 번째 파라미터는 개수를 의미한다. slab id에 저장된 타입 정보를 볼 수 있다.

```
stats cachedump 1 1
ITEM key [2 b; 1424065178 s]
END
```

stats slabs 커맨드는 slab에 대한 정보를 출력한다.

```
stats slabs
STAT 1:chunk_size 96
STAT 1:chunks_per_page 10922
STAT 1:total_pages 1
STAT 1:total_chunks 10922
STAT 1:used_chunks 1
STAT 1:free_chunks 10921
STAT 1:free_chunks_end 0
..
STAT active_slabs 1
STAT total_malloced 1048512
END
```

statis sizes 커맨드는 slab 개수를 출력한다.

```
stats    sizes
STAT    96 1
END
```

stats settings 커맨드는 현재 Memcached의 설정을 보여준다.

```
stats    settings
STAT    maxbytes 33554432
STAT    maxconns 1024
STAT    tcpport 11211
...
```

stats reset 커맨드는 현재 상태를 모두 초기화한다. flush_all 커맨드와 함께 쓰일 수 있다.

```
stats reset
RESET
```

1.3.6 기타 커맨드

모든 키를 삭제한다. redis와 달리 성능에 영향을 주지 않으며, 빠른 속도로 실행이 완료된다.

```
flush_all
OK
```

참고로 Memcached 서버에 저장된 키를 살펴볼 수 있는 방법은 아쉽게도 없다.

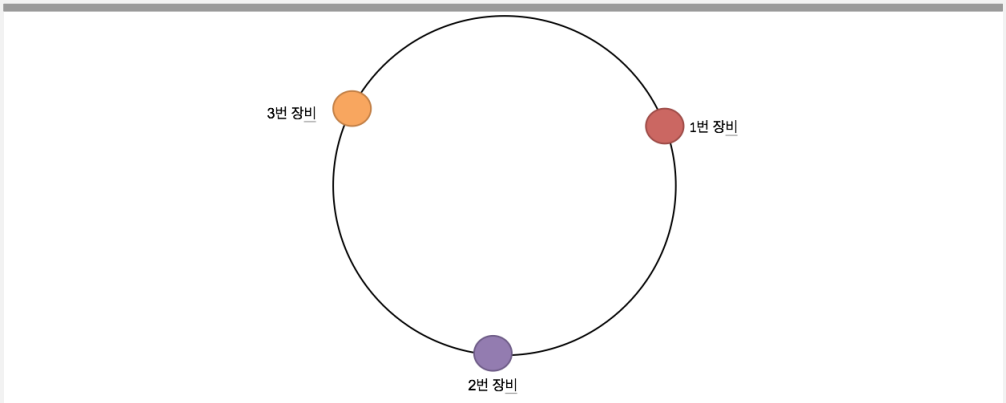
II. Memcached 운영

2.1 부하 분배 (Load Balance)

Memcached 서버는 캐시 데이터를 직접 분배할 수 있는 방법이 전혀 없다. 그러나 Memcached 클라이언트에서 캐시 데이터를 분배하는 알고리즘을 사용하여 캐시 데이터를 분배하고 있으며, 이를 통해서 부하를 분배하고 캐시 데이터를 많이 저장할 수 있다.

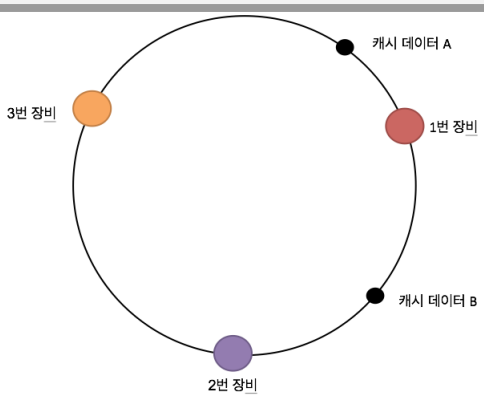
Memcached가 사용하고 있는 Consistent Hash이라는 분배 알고리즘을 설명한다. 위에서 설명한 일반적인 해쉬 함수를 기반으로 하고 있지만, Memcached가 추가되거나 장애 발생 시 모든 Memcached 서버에 대해서 재할당을 다시 실행하지 않고 일부 데이터에 대해서만 재할당이 발생한다. 즉 변경된 서버에서만 재할당을 이루게 한다는 특징을 가지고 있다. 따라서 캐시 히트가 조금 영향을 줄 수 있겠지만 전체 성능을 좌우하지 않는다. Consistent Hash 알고리즘에 대한 설명을 다음과 같다. 〈그림 4〉는 Memcached 3대에서 키를 나누는 현황을 보여주고 있다. 1번 장비, 2번 장비, 3번 장비가 모든 키를 3등분하고 있다.

그림 4_Memcached 서버 장비 셋팅



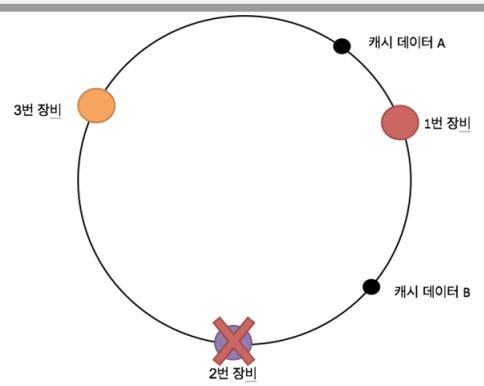
〈그림 5〉는 〈그림 4〉에서 캐시 데이터 A와 B가 추가되었을 때, Consistent Hash 알고리즘에 의해 데이터가 추가된 것을 설명한 것이다. 캐시 데이터 A는 1번 장비로 저장하고, 캐시 데이터 B는 2번 장비에 저장한다.

그림 5_Memcached 서버 장비 셋팅 후 캐시 데이터 2개가 들어온 상황



〈그림 6〉처럼 만약 2번 장비가 장애가 발생하여 Memcached 클라이언트가 접근할 수 없다고 가정한다면, 캐시 데이터 B는 데이터 유실이 된다. 애플리케이션 서버에서 2번 장비가 없다고 판단하고 있는 상태에서 캐시 데이터 B를 저장하려고 한다면 Memcached 3번 장비에 저장한다. 그러나 캐시 데이터 A는 1번 장비에 그대로 저장되어 있다.

그림 6_Memcached 서버 장비 중 한 대의 장비가 장애가 발생한 경우



최근에 바뀌었던 캐시 데이터 B가 3번 장비에 저장된 상태에서 오래된 캐시 데이터 B를 포함한 2번 장비가 다시 서비스에 투입되었다고 가정해 본다. 그렇다면 〈그림 5〉와 같은 상태가 된다. 일반적으로 문제가 발생하지는 않지만, 특수하게 일어날 수 있다. 애플리케이션 서버에서 2번 장비의 오래된 캐시 데이터 B 대신 3번 장비의 새로운 캐시 데이터 B를 바라보지 못하는 부분이 발생하여 데이터가 일치하지 않는 현상이 나타날 수 있다. 데이터 정합성이 매우 중요한 상황이라면 장애 이후 다시 원래대로 두기 전에 2번 장비의 캐시 데이터를 모두 삭제하는 것이 좋다. 그리고 유효기간을 주어 데이터가 자연스럽게 삭제되도록 정책을 잘 세워두는 것도 좋을 수 있다. Memcached 자바 클라이언트인 Spymemcached를 사용할 때, Memcached 서버에 Consistent Hash 알고리즘을 쓰겠다는 설정을 다음과 같이 할 수 있다.

```
ConnectionFactory connFactory = new ConnectionFactoryBuilder()
        .setLocatorType(Locator.CONSISTENT).build();
MemcachedClient client =
        new MemcachedClient(connFactory, AddrUtil.getAddresses(serverString));
```

2.2 설정 주의 사항

Memcached 서버를 소스 컴파일 해서 설치 시 특별한 설정 파일이 없기 때문에 파라미터로 실행해야 한다. (RPM 설치 시에는 설정 파일이 있다.)자세한 도움말은 -h를 아규먼트로 넣으면 확인할 수 있다.

```
$ memcached -h
```

중요한 파라미터 설명은 다음과 같다.

- -d : 데몬으로 실행
- -p : TCP 포트 번호
- -u <username> : 전환할 사용자 이름
- -m <num> : 최대 메모리 (MB 단위, 기본 값은 64)
- -t <num> : 사용자 쓰레드 (기본 값은 4)
- -v : 로그 출력
- -vv : 좀 더 자세히 로그 출력 (클라이언트 command를 출력할 수 있음)
- -vvv : 아주 많이 로그 출력
- -c : 동시 접속자 수
- -m : 메모리 크기 •-U : udp 포트 지정. 안쓰려면 0을 설정
- -b : 백로그 큐 크기(backlog queue limit)를 지정

예를 들어, 데몬으로 실행하여 쓰레드를 8개로 돌리고 최대 메모리를 10G로 하고 싶다면, memcached는 다음과 같이 실행한다.

```
$ memcached -d -m 10480 -t 8
```

사용자 쓰레드는 전체 쓰레드 개수가 아닌 병렬 처리를 담당하는 워커 쓰레드(Worker Thread)의 개수를 의미한다. 내부적으로 실행되는 쓰레드는 여기서 지정할 수 없다.
디폴트 실행 시 동시 접속자 수는 1024이다. -c 옵션을 주어 동시 접속자 수를 높인다.
동시 접속자 수를 10240 으로 하고 싶다면 다음과 같다.

```
$ memcached –c 10240
```

그리고, 메모리 크기는 디폴트로 64MB이므로 메모리 크기를 살핀 후 적당하게 변경하면 좋다. 다만 물리 메모리 이상으로 잡아서 Swap 메모리를 쓰지 않게 해야 한다. Swap 메모리를 사용하면 성능이 매우 좋아지지 않는다. 9GB의 메모리를 설정하려면 다음과 같다.

```
$ memcached –m 9216
```

Memcached 서버의 백로그 큐 개수는 1024이다. 이를 여유 있게 다음과 설정하여 큐잉(queuing)을 잘 할 수 있도록 한다.

```
$ memcached –b 8192
```

UDP 포트를 사용하지 않으려면 '-U 0' 을 사용한다.

위의 나온 모든 내용을 적용한다고 가정하면, 다음과 같을 것이다.

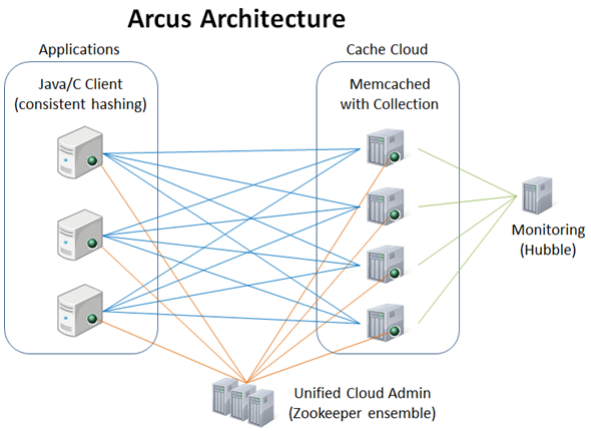
```
$ memcached –d –m 10480 –t 8 –c 10240 –m 9216 –b 8192 –U 0
```

2.3 Memcached 관련 오픈 소스 사용하기

순수하게 Memcached를 사용할 수 있으나, 제약들이 존재하기 때문에 Memcached 기반 오픈 소스를 활용하는 방식을 소개한다. Memcached 서버가 클러스터링, 복제, 장애 자동 복구 등을 지원하지 않기 때문에 이를 지원하는 오픈소스 제품들이 한 때 있었다. 예를 들어, repcached(2011년 마지막 패치), moxi(2010년 마지막 패치)와 같은 유명한 오픈 소스는 더 이상 지원하지 않기 때문에 사용에 유의할 필요는 있다. Naver에서 오픈 소스로 공개된 Arcus[8]를 소개하고자 한다. Arcus는 Memcached 기반으로 만들어진 오픈 소스이다. 특히 Naver에서 운영하면서 좋은 기능이 추가되었다. Zookeeper를 사용함으로서, 캐시 노드 리스트를 관리하고 자동 장애 복구를 할 수 있다. 그리고, List, Set, B+tree와 같은 Collection을 이용할 수 있는 API를 제공하고, 이를 위한 Small Memory Allocator를 적용하였다. 특정 prefix를 주면, 삭제가 가능한 API를 제공한다. 또한 모니터링 솔루션인 hubble도 제공하고 있다. 〈그림 7〉은 Arcus 구조에 대한 설명이다.

8) http://naver.github.io/arcus/

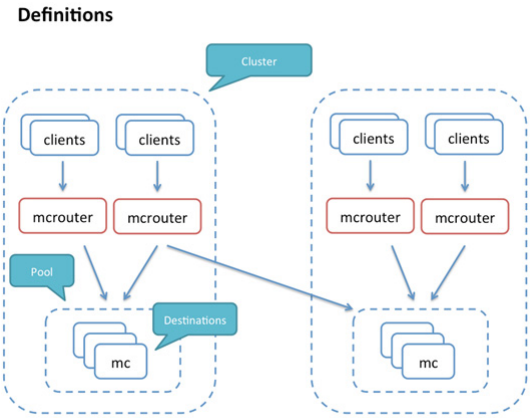
그림 7_Arcus 구조



출처 : https://github.com/naver/arcus

Facebook은 Mcrouter[9]라는 클러스터링, 복제가 가능한 오픈소스를 공개했다. Memcached 클라이언트는 Mcrouter에 접속해서 API를 저장할 때, Mcrouter는 여러 IDC에 분산된 Memcached 서버에 데이터를 분산 저장한다. 이를 위해서 Mcrouter는 Consistent Hash 알고리즘으로 분배할 수 있으며, prefix별로 라우팅이 되게 하고, Connection Pool을 유지한다. 특히 모니터링을 하여 자동 장애 복구 기능을 한다. 그 외에 Memcached 서버가 막 추가되어 캐시 데이터가 전혀 없는 경우, 캐시 데이터를 채우는 기능, 브로드캐스팅(broadcasting) API, SSL API를 제공한다. 〈그림 8〉은 Mcrouter 구조를 설명한 그림이다.

그림 8_Mcrouter 구조



출처 : https://code.facebook.com

9) https://github.com/facebook/mcrouter

참고로 Memcached를 개발한 개발자(Brad Fitzpatick)가 Groupcache[10]을 개발했는데, 이는 Memcached를 대체하려고 만든 오픈소스이다. 현재는 그다지 활발하지는 않지만, 아이디어 차원에서 공유하고자 한다. 지금까지 설명한 Memcached 서버는 재시작 또는 서버 장애로 새로운 장비를 도입했을 때 다시 데이터를 채워질 때까지는 계속 애플리케이션 서버가 캐시 데이터를 생성해야 하는 부분이 존재한다. 캐시 데이터가 채워지기까지는 부하가 일시적으로 발생한다. 그래서, Groupcache는 캐시 데이터를 채우는 Cache Filling Mechanism을 적용하고 있다. 또한 데이터를 변경하지 못하도록 하며 자주 불리는 캐시 데이터를 자동으로 다른 노드로 복제해서 캐시 시스템의 부담을 줄이는 방식을 이용한다.

Ⅲ. 결론

애플리케이션의 서버와 스토리지의 부하를 낮추게 하는 Memcached는 아주 간결한 기능을 가지고 있지만 캐시 시스템의 의도에 맞게 개발된 오픈 소스이다. Memcached에서 사용 가능한 API를 다루었고 Memcached 서버가 문제가 되더라도 Memcached 클라이언트의 Consistent Hash 알고리즘에 의해서 정상적으로 동작할 수 있는 부분을 설명했다. 그리고, Memcached를 기반으로 하고 있는 Arcus와 Mcrouter를 소개하여 클러스터링과 복제가 가능한 오픈 소스를 소개했다.

지금까지 양대 분산 캐시인 Redis와 Memcached를 설명했다. 분산 캐시는 대용량 트래픽을 처리할 수 있음으로서, 스토리지의 부하를 줄여 많은 처리를 할 수 있게 한다. 또한 데이터 공유용으로도 사용할 수 있다. Memcached는 Redis보다 일찍 세상에 나왔고 간단하고 확장성이 높은 탓에 유명한 회사의 개발자들이 많이 사용하고 있다. 한편 Redis는 쓰기 편하고 다양한 API가 있으며 복제 기능을 제공하기에 최근에 많은 개발자들이 많이 활용하고 있다. 독자들 중에 휘발성이 높지만 확장성과 높은 성능을 제공하는데 필요한 아키텍처를 구성한다면, 분산 캐시를 고려하면 좋을 것 같다.

10) https://github.com/golang/groupcache

참고 자료

1. Memcached 커맨드 https://code.google.com/p/memcached/wiki/NewCommands
2. 슬랩 메모리 할당자 설명 http://flylib.com/books/en/4.454.1.55/1/
3. Facebook에서 Memcached를 이용한 사례 https://www.usenix.org/sites/default/files/conference/protected-files/nishtala_nsdi13_slides.pdf
4. Facebook에서 Memcached를 이용한 사례 https://code.facebook.com/posts/296442737213493/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/