Memcached Java API

Simon Johnston

September 10, 2008

This document outlines a Java API for `memcached`[1] that tries to faithfully reflect the API and protocol[2] of `memcached` itself while providing a set of additional capabilities that provide value to the user. The package is fairly low-level, as in [3] rather than try and provide any additional level of abstraction such as the package by Sallings [4].

Introduction

The `memcached` package is described in [1] as follows:

> `memcached` is a high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load.

The API package described in this document provides a relatively faithful Java API over the `memcached` protocol. It should be noted that the `memcached` developers do not provide any language bindings or API themselves, they publish the protocol between client and server and allow others to develop such bindings. For Java a number of bindings exist ([4, 3]), however for different reasons it was decided that th Jazz Foundation Services needed a low-level API on which to build, but that had the minimum possible pre-requisites and also partitioned non-protocol capabilities into a set of *extensions* thus making them both configurable and optional.

This document describes the design, implementation and use of the package such that clients can be developed that use `memcached` from Java (specifically the Jazz Foundation - see [5]) and additional extensions could be developed to provide functionality beyond that provided by `memcached` itself. The Jazz Foundation uses this API to provide a common cache service for multiple storage services and thus providing distributed, load balancing access to repository resources. The cache services within the Jazz Foundation provide their own interface and abstraction allowing different distributed cache implementations to be used - including IBM WebSphere eXtreme Scale[6]. It was decided that rather than building the `memcached` interface directly into the corresponding Jazz Foundation cache service these were separated allowing this Java API to be used in other places if required.

Design and Implementation

The design of the Java API was guided by two goals, one at least has been mentioned above:

1. Provide an API that faithfully reflects the underlying `memcached` protocol rather than provide any new abstraction over the protocol.
2. Make the API very Java-friendly, use Java collections, interfaces and common patterns in the development of the API to make it easy to understand for the Java developer.

The design therefore started with a Java interface, `IMemcachedConnection` that represented the connection to one or more `memcached` process(es). The interface has methods that correspond almost one to one with the capabilities of the protocol, set, add, update, delete, etc. The idea of providing this as an interface allows the distinct implementation of the ASCII (think telnet protocol) and binary specifications of the `memcached` protocol. Now there is a requirement for a client to be able to get an instance of this interface and so a factory class is provided that can provide the client an instance of either protocol. This convention would also allow the provision of other implementations, so if the protocol changes in an incompatible manner the new protocol could be supported with a new object with the same API.

The API as specified allows a number of methods to take a list of parameters, as demonstrated in the snippet from the protocol specification below.

```
<command name> <key> <flags> <exptime> <bytes> [noreply]\r\n
```

This might suggest the Java API should take a similar list of parameters, however the protocol when retrieving items a similar list of values is returned (see listing below). This implies that the API will need to package the response data into an object and this is exactly what we do with the `MemcachedItem` class. Once we had used this as the return type from the `get()` method it actually became easier and more symmetrical to have the update commands take an instance of this class as a parameter rather than the list of individual parameters suggested above. Note that while the flags property is available on the `MemcachedItem` class clients should beware of using this as it is used as a bit field by the implementation; clients should use the flags field in the same way using bit values above the `SAFE_BIT_MIN` property value specified.

```
VALUE <key> <flags> <bytes> [<cas unique>]\r\n
<data block>\r\n
```

The upshot of this is that the corresponding update/retrieval methods on the API are specified in the following manner:

```
1 public MemcachedItemStatus set(MemcachedItem item, long expiration)
2         throws MemcachedException;
3 public List<MemcachedItem> get(List<String> keys, boolean cas)
4         throws MemcachedException;
```

Protocol command name	Java api method name
add, append, get, prepend, replace, set, stats, version	*the same*
cas	checkAndSet
decr	decrement
flush_all	flush
gets	*folded into the get method*
incr	increment

Table 1: Protocol commands to API methods

In terms of the mapping from the protocol to API mapping the decision was to try and retain all the names from the protocol itself, however some were made more readable. Table 1 maps the names from the protocol specification to the Java API method names. The only interesting exception where the API was changed somewhat is the separation on the protocol of get and gets commands. In the protocol gets returns CAS values whereas get does not. In the Java API the decision was to provide a single `get()` method that has a boolean flag to denote whether CAS values are to be provided in the response.

Again, coming back to the goals above you'll see that the Java API provides nothing more than just the protocol client, additional capabilities such as serialization of Java objects, compression and security of cache data and so on are all provided as extensions to this API.

Key/Cache Hashing

One unique aspect of the `memcached` design is that the protocol does not implement everything required to build an application client, one key piece is left to the client in all cases - the mapping of keys to cache process(es).

The implementation is again based on an interface, `ICacheHashFunction`, that hashes a key and returns the actual socket connection to the correct cache process. Currently the API has a single implementation of this interface that uses the following hash function where C is the set of connections and k is the provided key. Also note that the current implementation uses the CRC-32 function whereas a number of other known cyclic redundency or checksum functions (Adler-32, Fletcher-32) are either faster or more reliable the CRC-32 algorithm is reliable enough and provided by the Java SDK.

1. $f(C, k) = crc(k) \bmod |C|$

What this provides is a relatively random and even spread of keys to servers, however as with most such $k \rightarrow c$ hash functions it is important that each cache client sharing the set of connections must have the same set of connections and the set be ordered in the same way.

Client Usage

The first step in using the API is to get an instance of the connection interface, this interface is the main method by which a client interacts with the `memcached` process(es). To get such an instance the API provides a connection factory class with two static methods, one retrieves a default implementation, the other allows the client to pick either an implementation of the `memcached` ASCII or binary protocol[1]. The following listing demonstrates how the client can use the factory methods to retrieve an instance of a connection.

```
1 IMemcachedConnection connection = null;
2 connection= MemcachedConnectionFactory.getDefaultConnection();
3 connection = MemcachedConnectionFactory.getConnection(MemcachedProtocol.ASCII);
```

The next step is to connect to the list of `memcached` server process(es). This idea of a server list is important and so in effect the connection object is really a connection pool but rather than the traditional connection pool that is used to serve

```
1 List<ServerAddress> addresses = new ArrayList<ServerAddress>();
2 addresses.add(new ServerAddress("127.0.0.1"));
3 connection.connect(addresses);
```

Now, to use the API to cache data you need to use at least the `set()` and `get()` methods. The listing below shows how to create an item for storage, it needs only really a key and a data object (the constructor for `MemcachedItem` can take a string or byte array) although other options are available for more complex operations. Then you can call the API and set the item providing a cache timeout value, note that the API uses the memcached convention for specifying cache expiration times (see [2]). The protocol specifies a number of update commands (add, cas, replace, set, append, prepend) these are all reflected in the Java API but not all are demonstrated here.

The corresponding retrieval method, `get()`, takes a list of keys and returns a list of retrieved items. Note that any item in the original list that was not found on the server is not reported, so if three keys were provided but the server only found one item then the list contains only a single item.

```
1 MemcachedItem item = new MemcachedItem("key", "My Data Item");
2 MemcachedItemStatus status = connection.set(item, 2);
3
4 List<String> keys = new ArrayList<String>();
5 keys.add("key");
6 List<MemcachedItem> retrieved = connection.get(keys, false);
```

The `MemcachedItemStatus` response is an enumeration that once more maps relevant protocol responses to Java values and these are used in a number of the API methods.

Of course it's important to consider error handling when calling the API and the API provides three distinct exception classes, as demonstrated in the listing below. It is also possible for Java `IllegalArgumentException` or `IllegalStateException` exceptions to be thrown by the extensions, although the main API should map errors to `MemcachedClientException`. The JavaDoc for the API interface documents common cases for these exceptions.

[1] As of this time only the ASCII protocol is supported by the API.

```
try {
    String version = connection.version();
} catch (MemcachedClientException e) {
    ; // a badly formed request, or out-of-bounds arguments
} catch (MemcachedServerException e) {
    ; // an error in communicating with the server, or returned by the server
} catch (MemcachedException e) {
    ; // an invalid or unknown command or response from the server
}
```

Now we have managed to add items to the cache and retrieve them we may also need to explicitly remove something from the cache. The `delete()` method on the Java API allows for removal of items from the cache, and allows for the cache to not allow reuse of the key for a period of time (see the protocol specification). The API also provides a `flush()` method that removes all items from the cache. The time parameter in this case denotes a period for the server after which the flush occurs; the protocol specification recommends not having all cache process(es) flush at the same time, so the Java API has a parameter which if true will stagger the delay between servers.

```
connection.delete("key1", IMemcachedConnection.IMMEDIATE);

retrieved = connection.get(keys, false);
// retrieved.size() == 0

connection.flush(IMemcachedConnection.IMMEDIATE, false);
```

One capability built into the protocol is a special use of cache items to provide distributed counters, these are cache items that are treated as integer values and the protocol allows for increment/decrement operators to operate on these counters. These operations are also provided on the Java API in an identical fashion.

```
MemcachedItem item = new MemcachedItem("counter1", "0");
connection.set(item, 60);

Long value = connection.increment("counter1", 2);
// value == 2

value = connection.increment("counter1", 2);
// value == 4

value = connection.decrement("counter1", 1);
// value == 3
```

Finally, to gather statistics from the connected cache process(es) the `stats()` method returns a map with statistics from all servers as a simple map of (key, value) string pairs.

```
Map<ServerAddress, Map<String, String>> stats = connection.stats();

for (Iterator<ServerAddress> serverIterator = stats.keySet().iterator();
     serverIterator.hasNext();) {
    ServerAddress serverKey = (ServerAddress) serverIterator.next();
    System.out.println("server->" + serverKey.toString());
    for (Iterator<String> statIterator = stats.get(serverKey).keySet().iterator();
         statIterator.hasNext();) {
        String statKey = (String) statIterator.next();
        System.out.println("+ " + statKey + "->" + stats.get(serverKey).get(statKey));
    }
}
```

Serializing Java Objects to the Cache

One common requirement will be to cache Java objects, not just byte arrays and so rather than have all clients work out how to do this themselves it's possible to use the `SerializationItem` class which extends `MemcachedItem` with the ability to get/set a Java object which gets serializer/deserialized into a byte form. In the following example you can see how we set a static serializer (the most common case) and are now able to use the item class in much the same way as we did in the examples above. The difference is that we pass an object into the item constructor instead of a string or byte array, and we use the `getObject()` method to retrieve the object from the item once it's retrieved.

```
1 SerializationItem.setSerializer(new JavaSerializationSerializer());
2 SerializationItem item = new SerializationItem("urn:ticker:ibm", new Customer());
3 connection.set(item, 60);
4
5 List<MemcachedItem> retrieved = connection.get(keys, false);
6 item = new SerializationItem(retrieved.get(0));
7 System.out.println(item.getObject().toString());
```

Importantly the client now has to know that the data returned is likely to have been serialized and the response from `get()` methods will be basic `MemcachedItem` objects and will have to be copied into a `SerializationItem` before it can be used.

Connection Extensions

One key method on the connection object is the `getExtension()` method that allows a client to retrieve an instance of an extension object. As mentioned above the API partitions extensions over and above the protocol implementation into extension objects that provide additional functionality that clients may frequently require. Currently the API provides two extension interfaces:

IConnectionCompression the capability to compress and decompress data as it is transmitted in and out of the `memcached` process(es). This reduces the amount of data transmitted and therefore improves performance, especially when communicating with processes not running on the local server. This extension is turned on by default.

IConnectionEncryption the capability to encrypt and descrypt the data as it is transmitted in and out of the `memcached` process(es). This provides a level of both privacy and tamper-proofing for resources as they become stored in processes outside of the Jazz Foundation repository itself. This extension is turned off by default as it must be supplied with an encryption key before it can be used.

Note that the use of these extensions does have an impact on the API, for example if either compression or encryption is being used the methods `append()` and `prepend()` are not available. This is because the concatenation of the data is performed by the `memcached` process(es) which are unaware of the fact that the original and the requested additional data are compressed. The result of appending one block of compressed data onto another block of compressed data would not be possible to then decompress.

The following listing demonstrates the use of the compression extension, as mentioned above the extension is on by default. Applications using this API should not turn on and off compression on the fly, the API does use the flags on an item to denote whether an item was compressed or encrypted a client could overwrite these flags or confuse the API in other ways. Note that the extension does have one configurable property, the minimum size of a resource before compression is used, so small resources will not be compressed as this is often inefficient.

```
IConnectionCompression compression = null;
compression = (IConnectionCompression)connection.getExtension(IConnectionCompression.class);
System.out.println("compressed? " + compression.isCompressed());
compression.setCompressionMinimumSize(4 * 1024);
```

The encryption extension is somewhat more complex as the capability cannot be enabled without first providing a key that is then used by the cryptographic process. These APIs do use Java exceptions to report particular errors in configuration, as shown in the listing below. Note that all clients sharing the `memcached` process(es) must be using the same key and so some mechanism for storing, distributing and configuring the key must be provided. Once the key has been set and the extension enabled all resources added to the cache will be encrypted. Note that the same comment above on compression applies, applications should not turn encryption on/off on the fly.

```
IConnectionEncryption encryption = null;
encryption = (IConnectionEncryption)connection.getExtension(IConnectionEncryption.class);
System.out.println("encrypted? " + encryption.isEncrypted());

KeyGenerator keygen = KeyGenerator.getInstance("DES");
SecretKey key = keygen.generateKey();

try {
    encryption.setCryptoKey(key);
} catch (IllegalArgumentException e) {
    ; // if the key contains invalid parameters, including cases where
      // key parameters clash with the cryptographic algoritm ones, set
      // with setCipherAlgorithm.
}

try {
    encryption.setEncrypted(true);
} catch (IllegalStateException e) {
    ; // thrown when set to true before the cryptographic key is set.
}
```

It is expected that client applications will configure and enable extensions during start up and the leave these settings alone as the applications run. In the case of the Jazz Foundation cache services these details are exposed as configuration properties that the cache service sets during initialization.

References

[1] [Online]. Available: http://www.danga.com/memcached/

[2] B. Fitzpatrick, "Memcached protocol," sixapart.com, Tech. Rep., 2008. [Online]. Available: http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt

[3] (2008, 3). [Online]. Available: http://whalin.com/memcached/

[4] (2008). [Online]. Available: http://code.google.com/p/spymemcached/

[5] J. Wiegand, "The ibm rational jazz strategy for collaborative application lifecycle management," IBM, Tech. Rep., 2008.

[6] [Online]. Available: http://www.ibm.com/software/webservers/appserv/extremescale/