RDMA for Memcached 0.9.5 User Guide

HIGH-PERFORMANCE BIG DATA TEAM
http://hibd.cse.ohio-state.edu

NETWORK-BASED COMPUTING LABORATORY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
THE OHIO STATE UNIVERSITY

Last revised: November 10, 2016

Contents

1 Overview of the RDMA for Memcached Project

RDMA for Memcached is a high-performance design of Memcached over RDMA-enabled Interconnects. In addition to enabling low latencies on InfiniBand and 10/40GigE cluster, we introduce new features into the Memcached design including support for high-performance HDFS (HHH) burst-buffer mode and non-blocking Libmemcached API semantics. This version of RDMA for Memcached 0.9.5 is based on Memcached 1.4.24, and Libmemcached 1.0.18. This file is intended to guide users through the various steps involved in installing, configuring, and running RDMA for Memcached over InfiniBand. This guide provides examples to illustrate the use of both traditional blocking Set/Get APIs (`memcached_set`/ `memcached_get`) and newly introduced non-blocking Set/Get APIs (`memcached_iset`/`memcached_iget`/`memcached_bset`/`memcached_bget`). This guide also describes the micro-benchmarks defined for experimentation with the different advanced features in the RDMA for Memcached package.

If there are any questions, comments or feedbacks regarding this software package, please post them to rdma-memcached-discuss mailing list (rdma-memcached-discuss@cse.ohio-state.edu).

2 Features

High-level features of RDMA for Memcached 0.9.5 are listed below. New features and enhancements compared to 0.9.4 release are marked as (NEW).

- Based on Memcached 1.4.24
  - Compliant with the new Memcached's core LRU algorithm
- Based on libMemcached 1.0.18
- High performance design with native InfiniBand and RoCE support at the verbs level for Memcached Server and Client
- High performance design of SSD-assisted hybrid memory
  - Support for enabling and disabling direct I/O for SSD read/write
- Compliant with libMemcached 1.0.18 APIs and applications
- (NEW) Non-Blocking Libmemcached Set/Get API extensions
  - (NEW) APIs to issue non-blocking set/get requests to the RDMA-based Memcached servers
  - (NEW) APIs to support monitoring the progress of non-blocking requests issued in an asynchronous fashion
  - (NEW) Facilitating overlap of concurrent set/get requests
- (NEW) Support for burst-buffer mode in Lustre-integrated design of HDFS in RDMA for Apache Hadoop-2.x
- Support for both RDMA-enhanced and socket-based Memcached clients

- Easily configurable for native InfiniBand, RoCE, and the traditional sockets based support (Ethernet and InfiniBand with IPoIB)
- On-demand connection setup
- Tested with
  - (NEW) Native Verbs-level support with Mellanox InfiniBand adapters (DDR, QDR, FDR, and EDR)
  - RoCE support with Mellanox adapters
  - Various multi-core platforms
  - SATA-SSD, PCIe-SSD, and NVMe-SSD

3 Installation Instructions

3.1 Pre-requisites

In order to use the RDMA-based features provided with RDMA for Memcached, install the latest version of the OFED distribution that can be obtained from `http://www.openfabrics.org`. It also requires cyrus-sasl-devel package (eg: `yum install cyrus-sasl-devel`) and the libhugetlbfs package (`http://libhugetlbfs.sourceforge.net`).

3.2 Download

The RDMA for Memcached package consists of Memcached Server executable and Libmemcached Client libraries.

The rpm for the latest version of RDMA-Memcached package can be downloaded from:
`http://hibd.cse.ohio-state.edu/download/hibd/rdma-memcached-0.9.5-1.el6.x86_64.rpm`.

The latest version of RDMA-Memcached is also available as a tarball at:
`http://hibd.cse.ohio-state.edu/download/hibd/rdma-memcached-0.9.5.tar.gz`.

3.3 Installing from RHEL6 package

Running the following command script will install the software in /usr/local/rdma-memcached-0.9.5.

```
rpm -Uvh rdma-memcached-0.9.5-1.el6.x86_64.rpm
```

Alternatively, users can use `--prefix=PATH` to install in specific path.

The above command will upgrade any prior version of RDMA-Memcached that may be present.

3.4 Installing from tarball

To install using tarball,

- Extract memcached and libmemcached binaries from the tarball downloaded:
  `tar -xf rdma-memcached-0.9.5-bin.tar.gz`
- Change directory to find RDMA-based memcached binary and libmemcached libraries:
  `cd rdma-memcached-0.9.5`

Please email us at `rdma-memcached-discuss@cse.ohio-state.edu` if your distro does not appear on the list or if you experience any trouble installing the package on your system.

4 Runtime Parameters

Some advanced features in RDMA for Memcached 0.9.5 can be manually enabled by users. Runtime parameters in RDMA for Memcached 0.9.5 include:

4.1 Common Runtime Parameters

We describe the runtime parameters that are common to both Memcached Server and Client in this section.

4.1.1 MEMCACHED_USE_ROCE

- Class: Run time
- Default: 0 (Disabled)

This parameter needs to be set to 1, for running RDMA-Memcached on RoCE clusters.

4.2 Memcached Server Parameters

We describe the runtime parameters that are specific to Memcached Server in this section.

-p <**num**>: RDMA port number to listen on (default: 11211)

-E <**num**>: TCP port number to listen on (default: 11212). This is not enabled unless -E option is specified

-l <**addr**>: Interface to listen on (default: INADDR_ANY, all addresses) <addr> may be specified as host:port. If you don't specify a port number, the value you specified with -p or -U is used. You may specify multiple addresses separated by comma or by using -l multiple times

-d: Run as a daemon

-m <**num**>**:** Max memory to use for keys/index in megabytes (default: 64 MB)

-h: Print this help and exit

-i: Print memcached and libevent license

-t <**num**>**:** Number of RDMA threads to use (default: 4)

-H <**num**>**:** Number of TCP threads to use (default: 0)

-T <**SSD-path**>**:** Enable hybrid mode with RDMA Memcached (Specify full path to file on SSD. If the SSD directory does not exist, memcached will exit with error)

-z: Enable direct I/O for SSD read/write (O_DIRECT flag)

-e <**num**>**:** Max memory to use for values in megabytes (default: 64 MB)

-g <**num**>**:** Max limit of SSD file size for hybrid mode in megabytes (default: 4 GB)

-o burst_buffer=<**SSD-dir-path**>**:** Enable burst-buffer mode for HHH and provide SSD directory path for persisting data.

5 RDMA-Memcached Server

Memcached Server can be setup and run similar to the sockets-based Memcached server. For a list of arguments, please run `<RDMA-MEMCACHED_INSTALL_PATH>/memcached/bin/memcached -h`.

5.1 Server Runtime Modes

Our package can supports two primary modes: in-memory and hybrid-memory modes. In addition to this, we support the Memcached-based burst-buffer mode for high-performance HDFS available in the RDMA for Hadoop 1.0.0.

- In-memory mode: This mode adheres to the default design of memory management in Memcached. When memory is limited, the memcached server either evicts older key/value pairs to make place for new key/value pairs, or returns Out-Of-Memory error.
- Hybrid-memory mode: This mode extends the memory management schemes in Memcached to use SSD's assistance when memory is limited, rather that discarding or erroring out, in order to accommodate more key/value pairs in Memcached.
- Burst-buffer mode for HHH: This mode enables high-performance RDMA-based HDFS (HHH) to use Memcached server as a burst-buffer mode (HHH-L-BB). This mode uses Memcached server cluster to alleviate the load on the Lustre parallel file system while using the HHH-L mode and takes advantage of high-speed SSDs local to the Memcached servers for fault-tolerance.

5.2 Running RDMA-Memcached Server with In-Memory Mode

For default in-memory mode, please run

```
$$> <RDMA-MEMCACHED_INSTALL_PATH>/memcached/bin/memcached -t
    <num-threads> -m <max-memory-in-megabytes>
```

5.3 Running RDMA-Memcached Server with Hybrid-Memory Mode

For hybrid-memory mode, please run

```
$$> <RDMA-MEMCACHED_INSTALL_PATH>/memcached/bin/memcached -t
    <num-threads> -T <path-to-file-on-SSD>
    -m <max-memory-for-keys-in-megabytes>
    -e <max-memory-for-values-in-megabytes>
    -g <max-SSD-file-size-limit-in-megabytes>
```

Normal Cached I/O is used to access SSD in the hybrid mode. If desired, direct I/O can be enabled using `-z` runtime parameter.

5.4 Running RDMA-Memcached Server with Burst-buffer Mode

For burst-buffer mode, please run

```
$$> <RDMA-MEMCACHED_INSTALL_PATH>/memcached/bin/memcached -t
    <num-threads> -m <max-memory-for-keys-in-megabytes>
    -o burst_buffer=<ssd-dir-path-to-persist-hdfs-data>
```

This runtime mode is introduced to support the burst-buffer mode of HHH design in RDMA for Hadoop 1.0.0 that leverages both local storage and Lustre for HDFS I/O. Please see here for more information on the RDMA for Hadoop package.

6 RDMA-Memcached Client

The RDMA for Memcached package supports APIs in the default libmemcached library for the Memcached binary protocol. In addition to these, we introduce RDMA-enabled non-blocking set and get APIs in RDMA for Memcached 0.9.5. These APIs enable the users to issue set or get requests without needing to block on the response. and the response of individual requests can be monitored in an asynchronous fashion.

6.1 Libmemcached Blocking API

6.1.1 Description

We support the default APIs with the blocking behaviour, including `memcached_set`, `memcached_get`, `memcached_mget`, `memcached_increment`, `memcached_decrement` and their variations. Their implementations are designed to be compatible with existing API semantics.

6.1.2 Client Example

Memcached client programs need to be compiled with RDMA for Memcached Client library. A simple Memcached client program is provided along with the package, and it gets installed in `<RDMA-MEMCACHED-_INSTALL_PATH>/libmemcached/share/example` folder. This example program just issues a Set operation to the server, and then does a Get operation to retrieve the value from the server. The status of operations are output. The Memcached server node and port can be specified as command line argument.

Users can compile Memcached Client applications with RDMA for Memcached Client library.

```
$$> gcc memcached_example.c -o memcached_example
    -I<RDMA-MEMCACHED_INSTALL_PATH>/libmemcached/include
    -L<RDMA-MEMCACHED_INSTALL_PATH>/libmemcached/lib
    -lmemcached -lpthread
```

The client application can be run be executed, and any environmental parameters can be specified at runtime.

```
$$> EXAMPLE_ENV_PARAM=VALUE ./memcached_example node123:11211
$$> Key stored successfully
$$> Key retrieved successfully. Key = KEY, Value = VALUE123
```

In this example, the client program does a Memcached Set operation to the server running on node123 at port 11211, followed by a Get operation from the same server.

6.2 Libmemcached Non-Blocking API for RDMA-Memcached

We introduce novel and high-performance non-blocking key/value store API semantics in RDMA for Memcached 0.9.5, that can co-exist with default blocking APIs. These non-blocking API extensions allow the user to separate the request issue and completion phases. The proposed APIs can issue set/get requests and return to the user application as soon as the underlying RDMA communication engine has communicated the request to the Memcached servers; without blocking on the response. Supporting progress APIs are available to check the progress of each operation in either a blocking or non-blocking fashion, respectively. We benefit from the inherent one-sided characteristics of the underlying RDMA communication engine, while ensuring the completion of every read or write request. By exploiting these non-blocking semantics to issue I/O requests, the client-side waits and server-side memory management and/or I/O can be overlapped with other data accesses or computations. For more information, please find our publication here.

6.2.1 Description

Non-blocking Request Structures: To facilitate the proposed non-blocking API semantics, we introduce a new structure called `memcached_request_st` that contains: (1) completion flag that the user can test or wait on for operation completion, (2) pointer to the buffer where the server response will be available, and (3) pointers to user's value buffer. For each non-blocking request, a unique `memcached_request_st` is created and is used to monitor the receipt of response from the server in an asynchronous fashion. APIs for creating and destroying these request items are as follows:

```
memcached_request_st *memcached_request_create(memcached_st *ptr,
                      void *value, size_t value_length, uint32_t flags);
```

This function is used to create a non-blocking request structure that will then be used by other non-blocking RDMA-enabled Libmemcached functions to communicate with the server. It takes `value` buffer pointer, value length and any flags to be used for the request. It returns a pointer to the `memcached_request_st` created or NULL on errors.

```
void memcached_request_free(memcached_st *ptr, memcached_request_st *req);
```

This function is used to clean up memory associated with a `memcached_request_st` structure. You should use this when you have received a response.

Non-blocking Set APIs: We introduce non-blocking APIs for set requests with and without buffer re-use guarantees. With buffer re-use guarantee, the `value` buffer used to create the `memcached_request_st` can be re-used or returned once the request has successfully completed. On the other hand, without buffer re-use guarantee, the `value` buffer cannot be re-used until a response has been received (via progress APIs).

```
memcached_return_t memcached_iset(memcached_st *ptr, const char *key,
size_t key_length, memcached_request_st *req,
time_t expiration, uint32_t flags);
```

This function issues a set request without buffer re-use guarantees. It takes a key and its length, along with the `memcached_request_st` created. On success the value will be MEMCACHED_SUCCESS. Use `memcached_strerror()` to translate this value to a printable string.

```
memcached_return_t memcached_bset(memcached_st *ptr, const char *key,
size_t key_length, memcached_request_st *req,
time_t expiration, uint32_t flags);
```

This function issues a set request with buffer re-use guarantees. It takes a key and its length, along with the `memcached_request_st` created. On success the value will be MEMCACHED_SUCCESS. Use `memcached_strerror()` to translate this value to a printable string.

Non-blocking Get APIs: Complimentary to set, we introduce non-blocking APIs for get requests with and without buffer re-use guarantees.

```
memcached_return_t memcached_iget(memcached_st *ptr, const char *key,
size_t key_length, memcached_request_st *req);
```

This function issues a get request without buffer re-use guarantees. It takes a key and its length, along with the `memcached_request_st` created to receive the response and the value from the Memcached server. On success the value will be MEMCACHED_SUCCESS. Use `memcached_strerror()` to translate this value to a printable string.

```
memcached_return_t memcached_bget(memcached_st *ptr, const char *key,
size_t key_length, memcached_request_st *req);
```

This function issues a get request with buffer re-use guarantees. It takes a key and its length, along with the `memcached_request_st` created to receive the response and the value from the Memcached server. On success the value will be MEMCACHED_SUCCESS. Use `memcached_strerror()` to translate this value to a printable string.

Non-blocking Progress APIs: Non-blocking progress APIs check the progress of the operation in either a blocking or non-blocking fashion. The APIs are as follows:

```
void memcached_test(memcached_st *memc, memcached_request_st *req);
```

This function is used to monitor progress of a non-blocking set or get request in a non-blocking fashion. If response has been received it returns MEMCACHED_SUCCESS on success or MEMCACHED_FAILURE. If the response has not yet been received it returns MEMCACHED_IN_PROGRESS.

```
void memcached_wait(memcached_st *memc, memcached_request_st *req);
```

This function is used to monitor progress of a non-blocking set or get request asynchronously but in a blocking fashion. It returns MEMCACHED_SUCCESS on success completion of request or MEMCACHED_FAILURE.

6.2.2 Client Example

Memcached client programs using non-blocking APIs need to be compiled with RDMA for Memcached Client library. An example that illustrates how the iset/bset/iget/bget operations is used is provided along with the package, and it gets installed in `<RDMA-MEMCACHED-_INSTALL_PATH>/libmemcached/share/example` folder. This example program (memcached_nb_example.c) can be compiled similar to the example described in Section 6.1.2.

6.3 Libmemcached Support for RDMA-based HDFS (HHH) Burst-Buffer Mode

We introduce configurable parameters to enabled the burst-buffer mode for Hadoop users running applications with HHH and underlying file system with Lustre integrated support. This feature can be enabled in `hdfs-site.xml` by setting `hadoop.bb.enabled` to `true` and `memcached.server.list` with a comma-separated list of Memcached server hostnames.

7 Memcached Micro-benchmarks

The OHB Micro-benchmarks support stand-alone evaluations of Memcached, Hadoop Distributed File System (HDFS), HBase and Spark (See here). OSU HiBD-Benchmarks (version 0.9.2) for Memcached consist of Get and Set latency micro-benchmarks.

7.1 Building OHB Memcached Micro-benchmarks

The source code can be downloaded from `osu-hibd-benchmarks-0.9.2.tar.gz`. The source can be compiled with the help of the Maven (version 3.3.0 or higher) as follows:

1. Ensure that libmemcached.home property is set to the fully-qualified RDMA-Memcached-0.9.5 install path in `$OHB_INSTALL_PATH/memcached/pom.xml`.

```
<properties>
  <libmemcached.home>${RDMA_MEMCACHED_INSTALL_DIR}/libmemcached
  </libmemcached.home>
</properties>
```

2. Run Maven to build the OHB Micro-benchmark for Memcached

```
$ mvn clean package
```

All micro-benchmarks will be installed in `$OHB_INSTALL_PATH/memcached/target`. More details on building and running the micro-benchmarks are provided in the README.memcached.txt.

This micro-benchmark suite is also available with RDMA-Memcached package in the following directory: `<RDMA-MEMCACHED_INSTALL_PATH>/libmemcached/bin`. A brief description of the benchmark in the following sections.

7.2 Memcached Latency Micro-benchmark (ohb_memlat)

This micro-benchmark measures latency of a memcached operation for different data sizes. The test can run in three modes:

1. GET - The OHB Get Micro-benchmark measures the average latency of memcached get operation.
2. SET - The OHB Set Micro-benchmark measures the average latency of memcached set operation.
3. MIX - The OHB Mix Micro-benchmark measures the average latency per operation with a get/set mix of 90:10.

In all three micro-benchmarks, the memcached operations are repeated for a fixed number of iterations, for different data sizes (1B to 512KB). The average latency per iteration is reported, ignoring any overheads due to start-up.

The test mode can be selected using runtime arguments `--benchmark=<GET|SET|MIX|ALL>`. Similarly, Memcached server can be specified using the runtime argument `--servers=<SERVER[:PORT]>`. Below is a list of all runtime parameters,

`--servers=<SERVER[:PORT]>`
List which servers you wish to connect to.

`--benchmark=<SET|GET|MIX|ALL>`
Specify OHB Micro-benchmark type. Supported micro-benchmarks:

`SET` - OHB Set Micro-benchmark.
Micro-benchmark for memcached set operations.

`GET` - OHB Get Micro-benchmark.
Micro-benchmark for memcached get operations.

`MIX` - OHB Mix Micro-benchmark.
Micro-benchmark for memcached set/get mix.

`ALL` - Run all three OHB Micro-benchmarks.

`--version`
Display the version of the application and then exit.

`--help`
Display help and then exit.

An example of running this micro-benchmark is as follows:

```
$$> <LIBMEMCACHED_INSTALL_PATH>/bin/ohb_memlat --servers=storage01:11211
    --benchmark=MIX
```

7.3 Memcached Hybrid Micro-benchmark (ohb_memhybrid)

This micro-benchmark measures average latency and success rate of memcached gets for different data sizes, for a specified penalty for a miss in the Memcached server. The micro-benchmark first populates Memcached server with more keys that what can probably fit in memory, and accesses these keys at random in one of the two modes:

1. UNIFORM - All keys are selected uniformly at random.
2. NORMAL - Some keys are accessed more frequently than others.

For any of these modes, tests can be run with different spill factors, maximum memory available to Memcached server, and value size of key/value pair. The size of key is fixed at 16 Bytes. Below is a list of all runtime parameters:

`--servers=<SERVER[:PORT]>`
List which servers you wish to connect to.

--spillfactor=<value>
Spill factor (value greater than 1.0 preferably to simulate over-flow). Default is 1.33.

--maxmemory=<value in MB>
Max Server Memory (in MB). Default is 64 MB.

--valsize=<value in Bytes>
Value Size of Key/Value Pairs (in Bytes). Default is 4KB. Key size fixed to 16 Bytes.

--scanmode=<UNIFORM | NORMAL>
Pattern for memcached_get. Default is UNIFORM. Supported modes:

UNIFORM - All keys are selected at random with equal probability.

NORMAL - Some keys queried more frequently that others (similar to normal distribution).

--misspenalty=<value in ms>
Additional latency (e.g. database access latency) to fetch key/value pair,
when it is a miss in memcached (in ms). Default is 1 ms.

An example of running this micro-benchmark where the client populates the Memcached server with 50% more key/values pairs (with value size 4KB) than what can fit into a server running with 64MB of RAM, assuming that the additional latency on a miss at the Memcached layer is 1.5 ms, with data being accessed in a normal pattern is as follows:

```
$$> <LIBMEMCACHED_INSTALL_PATH>/bin/ohb_memhybrid --spillfactor=1.5
    --servers=storage01:11211 --maxmemory=64 --misspenalty=1.5
    --scanmode=NORMAL --valsize=4096
```

7.4 Memcached Latency Micro-benchmark for Non-Blocking APIs (ohb_memlat_nb)

This micro-benchmark measures average latency of the newly introduced non-blocking set and get operations, i.e., iset, iget, bset, bget etc. The micro-benchmark issues a batch of first non-blocking set/get requests and monitors their completion using either wait or test APIs. This parameter is referred to as request threshold and behaves as a barrier to ensure progress and completion of all ongoing requests. The micro-benchmark behaves as follows:

1. For micro-benchmark runs with iset or bset, the number of set requests issued is equal to the <max-memory>/<value-size>.
2. For micro-benchmark runs with iget and bget, the Memcached servers are populated with <max-memory>/<value-size> key/value pairs. These pairs can be read using either at random or using a zipf pattern and the number of such requests can be controlled during runtime.

For any of these modes, tests can be run with different aggregated maximum server memory, value size of key/value pair, progress API of choice and the threshold of number of ongoing requests. The size of key is fixed at 16 bytes. Below is a list of all runtime parameters:

--servers=<SERVER[:PORT]>
List which servers you wish to connect to.

--reqtype=<API short name>
iset/bset/iget/bget.

--progresstype=<non-blocking progress API>
wait/test.

--pattern=<pattern for iget/bget>
Pattern for memcached_get/iget/bget. Default is ”random”. random: All keys are selected at random with equal probability. zipf: Keys selected using zipf distribution.

--maxmemory=<value in MB>
Max Server Memory (in MB). Default is 64 MB.

--numgets=<number of iterations>
Total number of key/value pairs to fetch from Memcached servers.

--valsize=<value in Bytes>
Value size of key/value pairs (in bytes). Default is 4KB. Key size fixed to 16 Bytes.

--reqthresh=<number of requests>
Number of pending non-blocking requests allowed.
Must be smaller than <max-memory>/<value-size>.

--verbose
Display more verbose output for micro-benchmark.

--version
Display the version of the application and then exit.

--help
Display help and then exit.

An example of running this micro-benchmark where the client populates the Memcached server with value size 4 KB and maximum memory of 1 GB, i.e., 256 K key/value pairs using iset API (`memcached_iset`) and wait API (`memcached_wait`) with a request threshold of 32 ongoing requests is as follows:

```
$$> <LIBMEMCACHED_INSTALL_PATH>/bin/ohb_memlat_nb --maxmemory=1024
     --valsize=4096 --servers=storage01:11211 --reqthresh=32
     --reqtype=iset --progresstype=wait
```

An example of running this micro-benchmark where the client reads 1000 key/value pairs from the Memcached server of value size 256 KB and maximum memory of 1 GB, i.e., from 4 K key/value pairs using bget API (`memcached_bget`) and wait API (`memcached_test`) with a request threshold of 16 ongoing requests is as follows:

```
$$> <LIBMEMCACHED_INSTALL_PATH>/bin/ohb_memlat_nb --maxmemory=1024
     --valsize=262144 --servers=storage01:11211 --numgets=1000
     --reqtype=bget --progresstype=test --reqthresh=16
```

8 Troubleshooting with RDMA-Memcached

If you are experiencing any problems with RDMA-Memcached package, please feel free to contact us by sending an email to rdma-memcached-discuss@cse.ohio-state.edu.