# Mem-o-Recall

A Memcached like caching service

## Time Spent

| **Activity** | **Time spent** |
|---|---|
| Online Reading | 1.5 hrs. |
| System design | 1 hr. |
| Prototyping | 1 hr. |
| Coding, with unit tests | 7 hrs. |
| Debugging | 1 hr. |
| Documentation and read-me | 1.5 hr. |
| **Total** | **13 hrs.** |

Online Reading included

- Familiarizing myself with Memcached in general, esp. how it differs from REDIS
- Memcached architecture and philosophy
- Memcached client functionality
- Memcached text protocol
- How to implement a TCP server/listener in .Net Core
- Network event loops in windows and .Net Core
- Best practices around asynchronous usage of network stream
- Can the cache be made thread safe in a lock-less way?
- Lock striping
- How .net runtime implements ‘MemoryCache’ (built in in-memory cache)
- Can ‘ConcurrentDictionary’ readily available in .net runtime be modified to implement eviction?

## Assumptions and simplifications

1. Clients will connect to the server using TCP only, on port 11211.
2. Clients will use Memcached text-protocol.
3. For text, UTF-16 encoding will be used.
4. Like Memcached, the server instances are independent and disjoint from each other. The clients have the logic to figure out which one of the many cache servers to talk to.
5. Each key can at most be 250 characters (Enforcing the limit is pending in the implementation).
6. A value corresponding to a key can at most be 2000 characters (Enforcing the limit is pending in the implementation).
7. The limitation on the overall memory utilization of the cache is used enforcing a limit on the total number of entries I the hash table. The number of bytes in memory used by the entire cache is not calculated.
8. Expiry on items is not enforced. Flags and no-reply is completely ignored.
9. Commands: Only the following commands are supported.
    a. GET
        i. Multiple keys are supported (Enforcing a limit is pending in the implementation).
        ii. The flags are not returned.
    b. SET
        i. Flags and expiry are accepted in the request but is not stored and enforced.
        ii. CAS variant is not supported.

10. Only the sever that runs on one cache service node has been implemented. This server is unaware of other cache servers in the cluster, if any. Also, the client-side logic of hashing the key to figure out the server to communicate to has been omitted for simplicity. The sample client provided can talk to exact one cache server and will use that server for all keys.

## Design

### System design

The design for this has drawn inspiration from Memcached. Some of the key design decisions are highlighted below.

**Eviction:**

LRU eviction policy has been chosen as this fits the 'time' locality principle driving caching in general. This is implemented over the built in Dictionary in .net runtime using a doubly linked list where the head represents the most recently used item and tail represents the least recently used item. For simplicity, I have chosen simple non-segmented LRU. A more sophisticated implementation would use segmented LRU where 'hot', 'warm' and 'cold' items are maintained and an item would transition between these stages before it gets evicted out.

**Thread safety**

The out of the box hash table (Dictionary) implementation in .Net core runtime is not thread safe. The thread safe 'ConcurrentDictionary' does not provide an easy way to manage the eviction engine without further locking. To achieve thread safety, any manipulation to the dictionary and/or the LRU eviction engine need to be serialized using a lock. A single lock typically works well for a few threads, but the performance typically degrades with a large number of threads (few hundred). Lock striping with having multiple dictionaries and multiple locks is one alternate option. I've also researched if it would be possible to go lock-less, but couldn't come up with a reasonably simple design. To keep this simple, I am using a single in-memory dictionary per node and a single lock guarding access to this dictionary. It is a conscious trade off between ease of implementation (in the context of this exercise) vs. possible non-linear degradation of performance at scale.

**Concurrency level**

Ideally, event loops are used to spawn thread/tasks to process requests. For simplicity here, I have one main server thread that is running the TCP listener. As soon as a client makes a TCP connection, the main server thread swaps off a new thread to process that TCP connection from to the client. At steady state, the number of threads will be one more than the number of active TCP connections to the server on the specified port. The thread handing requests from a specific connection processes the requests in the order they arrive, one after the other.

**Security**

There is no authentication between the clients and the caching server. The two main reasons for this are performance and simplicity. Typically, in a large-scale service, the 'clients' are other services running in the same datacenter. The caching servers are not discoverable and/or accessible from outside the subnet and it is a reasonable assumption to make that only legitimate 'clients' can talk to the caching server. Also, avoiding authentication and authorization checks on each connection would help reduce the latency, which is critical for a caching service.

**Areas for improvement in this design:**

1. A more sophisticated locking mechanism using lock striping will alleviate any potential contention on a single lock at large throughput.
2. Network event loops can be used for true concurrently at the request level. Given the sequential nature of send-request and wait-for-response before sending the next request in the clients, we need to evaluate if further concurrency will have any benefit.
3. The diagnostics library can be improved to be more sophisticated and support asynchronous batch uploads to log server.
4. Using segmented LRU, having ‘hot’, ‘warm’ and ‘cold’ items gives us better control on tuning the eviction engine such that in steady state, the hit to miss ratio is maximized.

## Library / Class design

When designing the layout of the classes and libraries, the main factors considered were:

- Simplicity
- Separation of concerns
- Making the core components generic
- Testability (mocking dependencies out to control the underlying behavior)
- Injectability (to enable use of dependency injection at runtime)

At a high level, the entire project is divided into the following libraries:

### `MemRecall.Core`

This library contains the implementation of in-memory cache. The code that ensures thread safety and the LRU engine resides here. The implementation of the cache is exposed via a parametrized interface `IConcurrentCache`. This enables mocking for tests and dependency injection. Also, the cache has been implemented in a generic fashion to work on any key types (TKey) and value types (TValue). However, the current implementation needs to be slightly modified to accept an `IEqualityComparer` instance to override the `GetHashcode()` and `Equals()` behavior for non-string type keys.

### `MemRecall.Server`

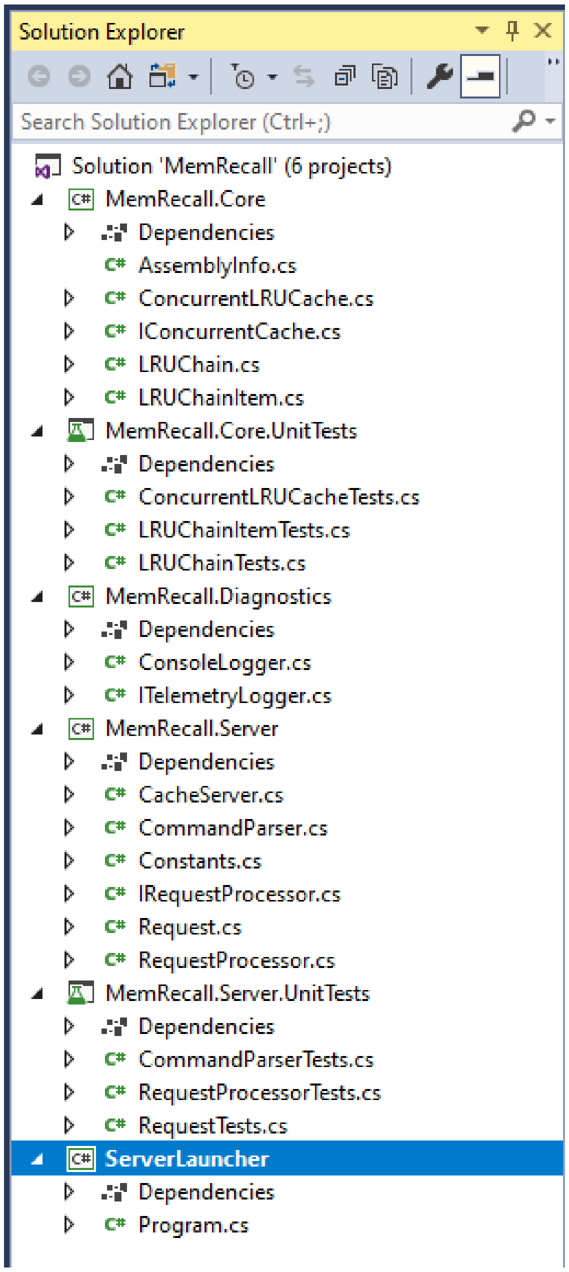
This library contains the implementation of the TCP listener, protocol handler/text parser, a request abstraction to store all the information related to the incoming request along with any result data once the request has been executed, a request-processor to satisfy the request against the in-memory cache. The functionality of the request processor is made available using an `IRequestProcessor` interface. The implemented request processor takes a dependency on `IConcurrentCache<TKey, TValue>` interfaces. The overall cache server itself take a dependency on `IRequestProcessor` and `ITelemetryLogger`.

### `MemRecall.Diagnostics`

This library exposes the `ITelemetryLogger` interface. Typically, the methods exposed via this interface are detailed enough to accept a ‘context’, correlation identifier, optional performance metrics, the log level (verbose vs. informational vs. warning etc.) amongst other things. These logs are usually buffered in memory and periodically pushed to the log server in an asynchronous fashion. But for the simplicity of this project, the interface is very simple, and the current implementation simply writes the logs to the default console.

`MemRecall.Core.UnitTests` and `MemRecall.Server.Unit` tests are the unit tests project for the respective libraries. Wherever possible, MoQ mocking framework has been used to mock dependent interfaces to keep tests focused to the specific area.

`ServerLauncher` is a simple console application that can be used to run the server.



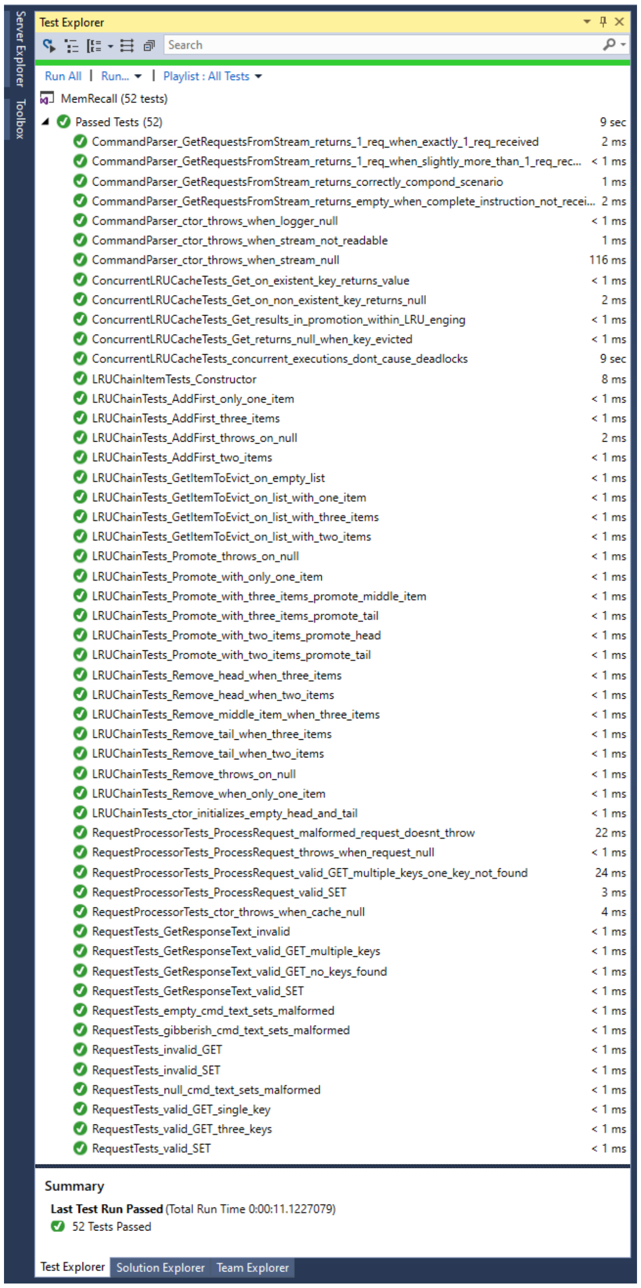
```
Solution 'MemRecall' (6 projects)
  MemRecall.Core
    Dependencies
    AssemblyInfo.cs
    ConcurrentLRUCache.cs
    IConcurrentCache.cs
    LRUChain.cs
    LRUChainItem.cs
  MemRecall.Core.UnitTests
    Dependencies
    ConcurrentLRUCacheTests.cs
    LRUChainItemTests.cs
    LRUChainTests.cs
  MemRecall.Diagnostics
    Dependencies
    ConsoleLogger.cs
    ITelemetryLogger.cs
  MemRecall.Server
    Dependencies
    CacheServer.cs
    CommandParser.cs
    Constants.cs
    IRequestProcessor.cs
    Request.cs
    RequestProcessor.cs
  MemRecall.Server.UnitTests
    Dependencies
    CommandParserTests.cs
    RequestProcessorTests.cs
    RequestTests.cs
  ServerLauncher
    Dependencies
    Program.cs
```

## Code borrowed/copied

None. The implementation of the TCP listener closely follows the pattern and guidance described in the Microsoft documentation for the respective classes.

## Testing

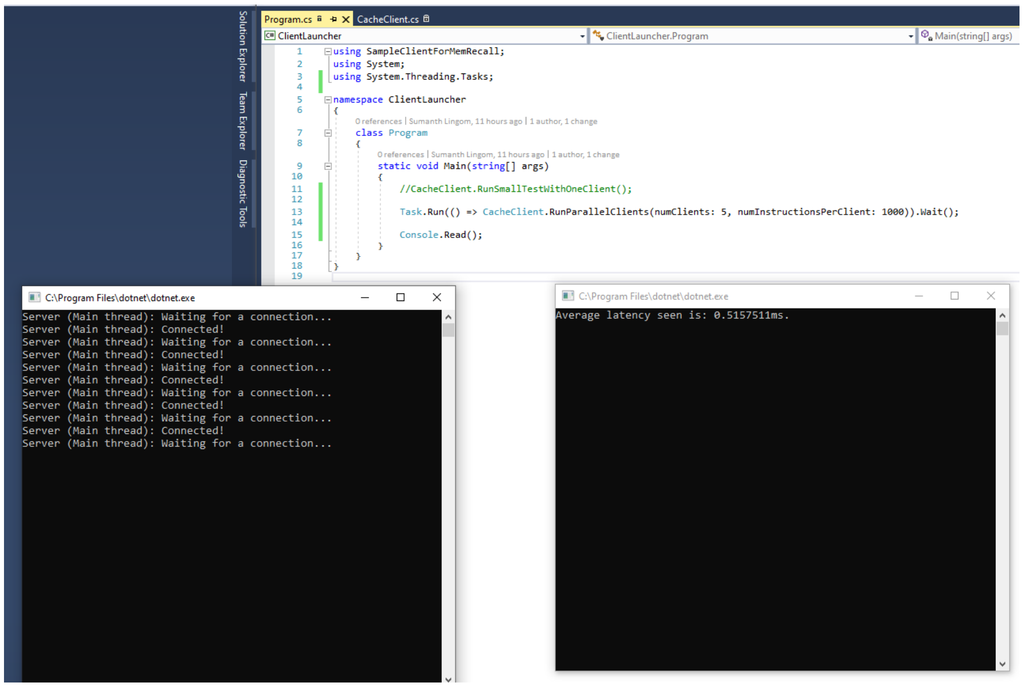
Wherever reasonably possible, I have tried to achieve full code coverage. The caching related classes, LRU engine, parser, request, and request processor have all been unit tested by mocking their dependencies. I also have a sample client that exercises the main line success scenarios end to end. A screenshot is available at the end of the document.



```
MemRecall (52 tests)
Passed Tests (52)                                                                        9 sec
  CommandParser_GetRequestsFromStream_returns_1_req_when_exactly_1_req_received           2 ms
  CommandParser_GetRequestsFromStream_returns_1_req_when_slightly_more_than_1_req_rec...  < 1 ms
  CommandParser_GetRequestsFromStream_returns_correctly_compond_scenario                  1 ms
  CommandParser_GetRequestsFromStream_returns_empty_when_complete_instruction_not_recei... 2 ms
  CommandParser_ctor_throws_when_logger_null                                            < 1 ms
  CommandParser_ctor_throws_when_stream_not_readable                                      1 ms
  CommandParser_ctor_throws_when_stream_null                                            116 ms
  ConcurrentLRUCacheTests_Get_on_existent_key_returns_value                             < 1 ms
  ConcurrentLRUCacheTests_Get_on_non_existent_key_returns_null                            2 ms
  ConcurrentLRUCacheTests_Get_results_in_promotion_within_LRU_enging                    < 1 ms
  ConcurrentLRUCacheTests_Get_returns_null_when_key_evicted                             < 1 ms
  ConcurrentLRUCacheTests_concurrent_executions_dont_cause_deadlocks                      9 sec
  LRUChainItemTests_Constructor                                                           8 ms
  LRUChainTests_AddFirst_only_one_item                                                  < 1 ms
  LRUChainTests_AddFirst_three_items                                                    < 1 ms
  LRUChainTests_AddFirst_throws_on_null                                                   2 ms
  LRUChainTests_AddFirst_two_items                                                      < 1 ms
  LRUChainTests_GetItemToEvict_on_empty_list                                            < 1 ms
  LRUChainTests_GetItemToEvict_on_list_with_one_item                                    < 1 ms
  LRUChainTests_GetItemToEvict_on_list_with_three_items                                 < 1 ms
  LRUChainTests_GetItemToEvict_on_list_with_two_items                                   < 1 ms
  LRUChainTests_Promote_throws_on_null                                                  < 1 ms
  LRUChainTests_Promote_with_only_one_item                                              < 1 ms
  LRUChainTests_Promote_with_three_items_promote_middle_item                            < 1 ms
  LRUChainTests_Promote_with_three_items_promote_tail                                   < 1 ms
  LRUChainTests_Promote_with_two_items_promote_head                                     < 1 ms
  LRUChainTests_Promote_with_two_items_promote_tail                                     < 1 ms
  LRUChainTests_Remove_head_when_three_items                                            < 1 ms
  LRUChainTests_Remove_head_when_two_items                                              < 1 ms
  LRUChainTests_Remove_middle_item_when_three_items                                     < 1 ms
  LRUChainTests_Remove_tail_when_three_items                                            < 1 ms
  LRUChainTests_Remove_tail_when_two_items                                              < 1 ms
  LRUChainTests_Remove_throws_on_null                                                   < 1 ms
  LRUChainTests_Remove_when_only_one_item                                               < 1 ms
  LRUChainTests_ctor_initializes_empty_head_and_tail                                    < 1 ms
  RequestProcessorTests_ProcessRequest_malformed_request_doesnt_throw                    22 ms
  RequestProcessorTests_ProcessRequest_throws_when_request_null                         < 1 ms
  RequestProcessorTests_ProcessRequest_valid_GET_multiple_keys_one_key_not_found         24 ms
  RequestProcessorTests_ProcessRequest_valid_SET                                          3 ms
  RequestProcessorTests_ctor_throws_when_cache_null                                       4 ms
  RequestTests_GetResponseText_invalid                                                  < 1 ms
  RequestTests_GetResponseText_valid_GET_multiple_keys                                  < 1 ms
  RequestTests_GetResponseText_valid_GET_no_keys_found                                  < 1 ms
  RequestTests_GetResponseText_valid_SET                                                < 1 ms
  RequestTests_empty_cmd_text_sets_malformed                                            < 1 ms
  RequestTests_gibberish_cmd_text_sets_malformed                                        < 1 ms
  RequestTests_invalid_GET                                                              < 1 ms
  RequestTests_invalid_SET                                                              < 1 ms
  RequestTests_null_cmd_text_sets_malformed                                             < 1 ms
  RequestTests_valid_GET_single_key                                                     < 1 ms
  RequestTests_valid_GET_three_keys                                                     < 1 ms
  RequestTests_valid_SET                                                                < 1 ms

Summary
  Last Test Run Passed (Total Run Time 0:00:11.1227079)
  52 Tests Passed
```

## Performance

With **5 parallel clients**, **each client issuing 1000 requests** in succession with anywhere between 1 and 10ms gap between instructions, the average latency we see across all clients is **about 0.5ms.**

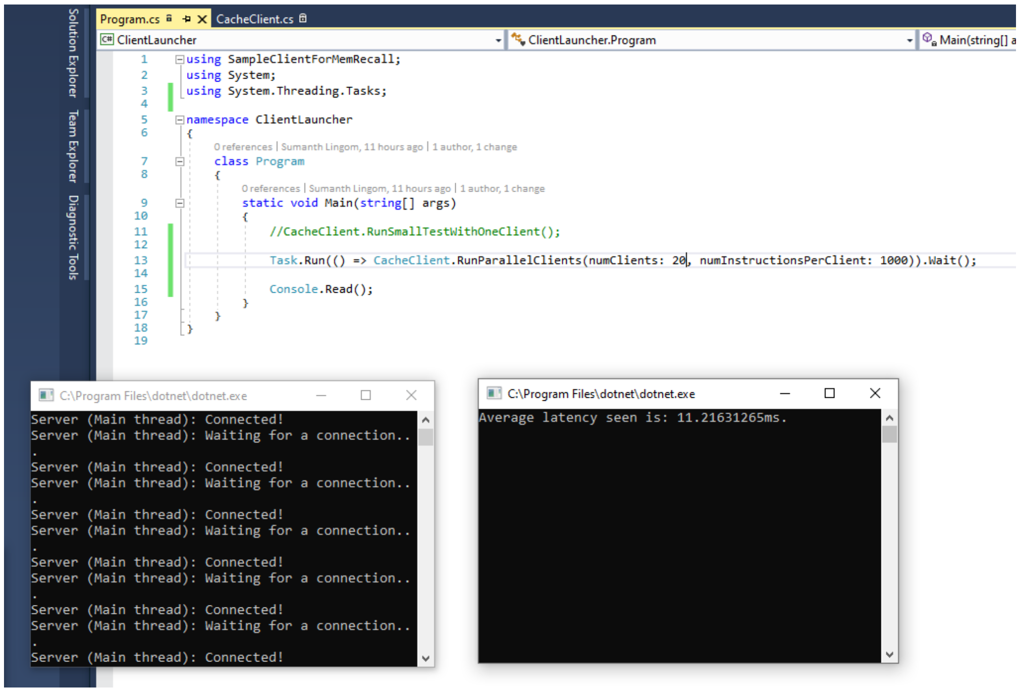


```
using SampleClientForMemRecall;
using System;
using System.Threading.Tasks;

namespace ClientLauncher
{
    0 references | Sumanth Lingom, 11 hours ago | 1 author, 1 change
    class Program
    {
        0 references | Sumanth Lingom, 11 hours ago | 1 author, 1 change
        static void Main(string[] args)
        {
            //CacheClient.RunSmallTestWithOneClient();

            Task.Run(() => CacheClient.RunParallelClients(numClients: 5, numInstructionsPerClient: 1000)).Wait();

            Console.Read();
        }
    }
}
```

```
Server (Main thread): Waiting for a connection...
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection...
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection...
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection...
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection...
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection...
```

```
Average latency seen is: 0.5157511ms.
```

With **20 parallel clients, each issuing 1000 requests**, the average latency goes up to **11.2ms**. The CPU pretty much shoots up to 100% (see perf problems below).



```
using SampleClientForMemRecall;
using System;
using System.Threading.Tasks;

namespace ClientLauncher
{
    0 references | Sumanth Lingom, 11 hours ago | 1 author, 1 change
    class Program
    {
        0 references | Sumanth Lingom, 11 hours ago | 1 author, 1 change
        static void Main(string[] args)
        {
            //CacheClient.RunSmallTestWithOneClient();

            Task.Run(() => CacheClient.RunParallelClients(numClients: 20, numInstructionsPerClient: 1000)).Wait();

            Console.Read();
        }
    }
}
```

```
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection..
.
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection..
.
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection..
.
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection..
.
Server (Main thread): Connected!
Server (Main thread): Waiting for a connection..
.
Server (Main thread): Connected!
```

```
Average latency seen is: 11.21631265ms.
```

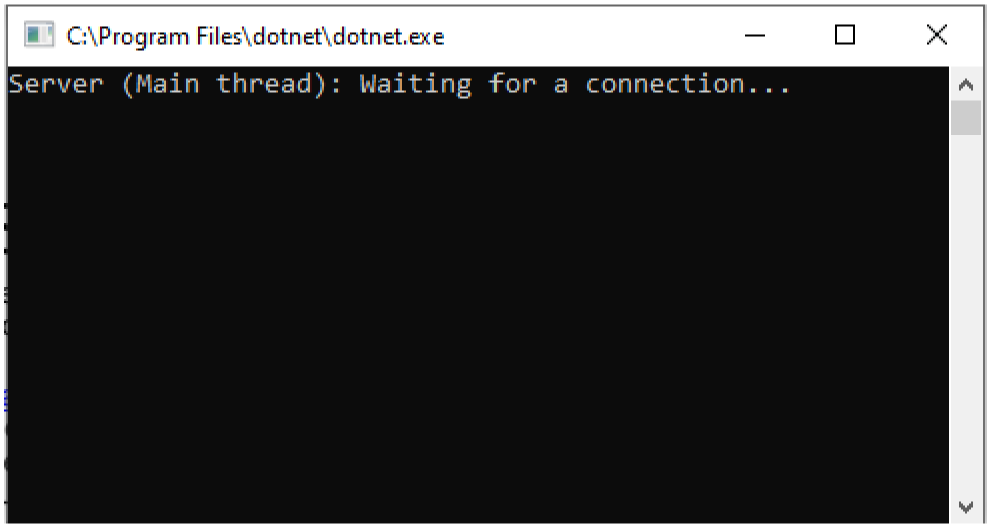
**Performance problems noticed**
As the number of parallel clients increases, I am noticing degrading performance. The following two are the likely reasons:

1. In the way TCP listener has been currently implemented, I have noticed that the waiting for data on the incoming network buffer seems to be a busy-wait operation, consuming CPU cycles. When multiple clients connect and the clients are not actively sending data but have the connections open, the CPU usage on the server seems to go high and stay there until the connections have been closed. Also, when a client explicitly closes the connection, it is taking a while for that to reflect on the server.
2. The single lock used by the cache also becomes a bottleneck as the number of concurrent calls to the lock() method increase.

## How to run the server?

Please make sure that you have .Net core 2.1 runtime along with Visual Studio 2017 installed on your machine.

1. Open the MemRecall.sln solution file, in the root folder, using Visual Studio 2017 IDE.
2. Right click on the solution and select 'Build solution'.
3. Make sure the 'ServerLauncher' project is set as the startup project. This is a simple console application that start the server.
4. Just hit 'F5' or run the solution using the green colored play button in visual studio. This should open up a console window and the console output should indicate that the server is now waiting for connections.



```
Server (Main thread): Waiting for a connection...
```

## Supported operations

- To connect to the server, please establish a TCP connection to the IP Address and 11211 port.
- Only memcahced text protocol is supported.
- UTF-16 LE encoding is used for communication.
- Currently, only GET and SET commands are supported.

### SET:

To set a single key, issue the command

```
SET <key> <flags> <expiry> [noreply]\r\n
<value_for_key>\r\n
```

And the response would be

```
STORED\r\n
```

Flags, expiry and noreply are ignored by the server. Only the value gets stored.

### GET:

To get a single key, issue the command

```
GET <key>\r\n
```

And the response would be

```
VALUE <key> <num_chars_in_value>\r\n
<value_for_key>\r\n
END\r\n
```

To get more than one key in the same request, issue the command

```
GET <key_1> <key_2> <key_3>\r\n
```

And the response would be

```
VALUE <key_1> <num_chars_in_value_for_key_1>\r\n
<value_for_key_1>\r\n
VALUE <key_2> <num_chars_in_value_for_key_2>\r\n
<value_for_key_2>\r\n
VALUE <key_3> <num_chars_in_value_for_key_3>\r\n
<value_for_key_3>\r\n
END\r\n
```

The flags value is not returned. Only the value for the key is returned.

For both commands, in case of error, the following is returned:

```
ERROR\r\n
```

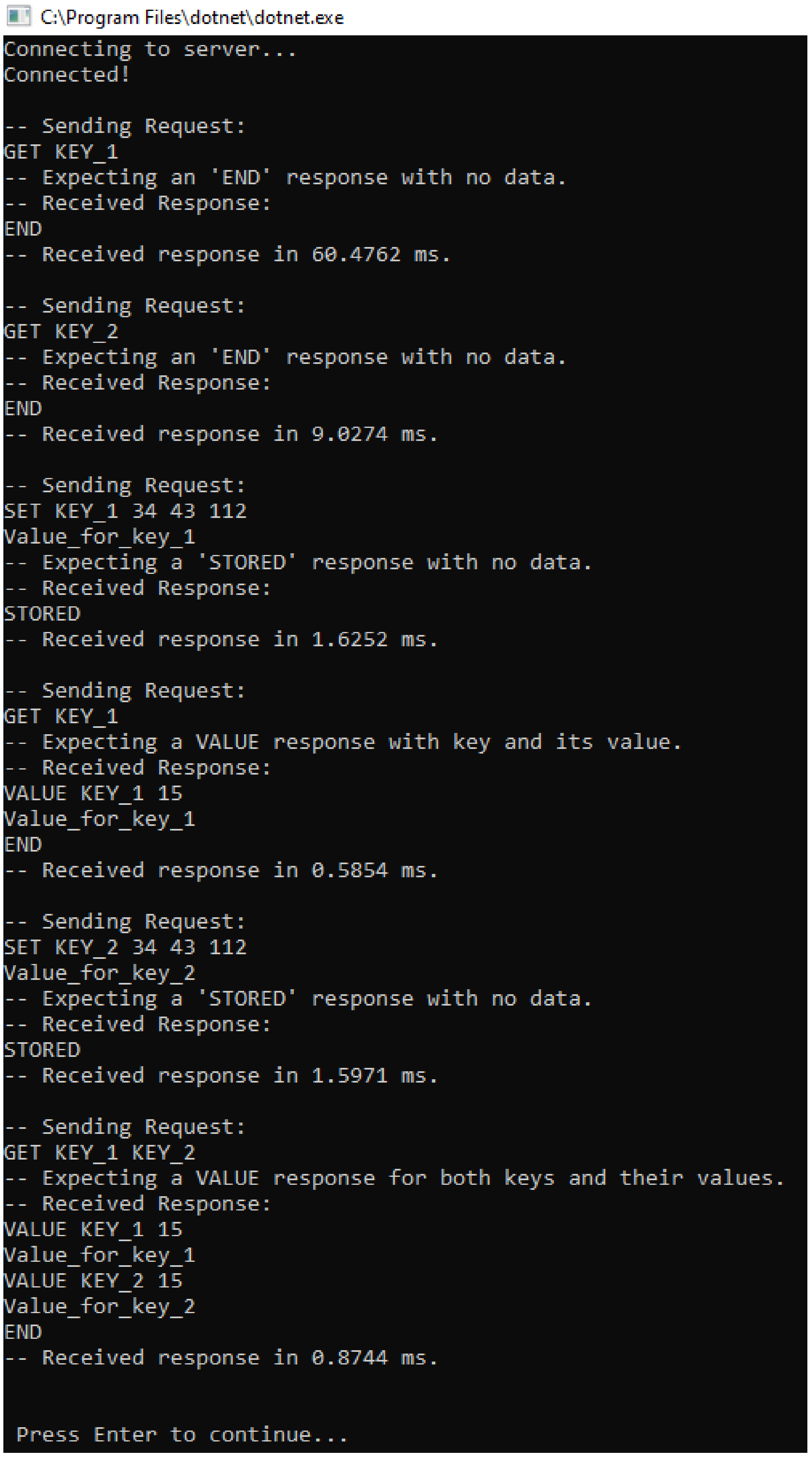
## How to run the client?

A sample client is available inside the ‘SampleClientForMemRecall’ folder.

1. Open the ‘SampleClientForMemRecall.sln’ solution file in the folder using Visual Studio 2017.
2. Right click on the solution and select ‘Build solution’.
3. Make sure the ‘ClientLauncher’ project is set as the startup project. This is a simple console application that start the client and cause it to connect to the server and make some sample requests.
4. Simply hit ‘F5’ or run the solution using the green colored play button in visual studio. This should open a console window and the console output should indicate that the client is connecting to the server.
5. Further console output will indicate the requests the client is making to the server, the expected output and the actual output.

Note: Please be sure to stop the start the service every time you want to re-run the client and if you want the expected result to match the actual result of the service. This is because of the order in which the sample client performs the operations. The test scenario programmed in the sample works only when the cache starts with a clean slate.

[Screen shot in the next page]



C:\Program Files\dotnet\dotnet.exe

```
Connecting to server...
Connected!

-- Sending Request:
GET KEY_1
-- Expecting an 'END' response with no data.
-- Received Response:
END
-- Received response in 60.4762 ms.

-- Sending Request:
GET KEY_2
-- Expecting an 'END' response with no data.
-- Received Response:
END
-- Received response in 9.0274 ms.

-- Sending Request:
SET KEY_1 34 43 112
Value_for_key_1
-- Expecting a 'STORED' response with no data.
-- Received Response:
STORED
-- Received response in 1.6252 ms.

-- Sending Request:
GET KEY_1
-- Expecting a VALUE response with key and its value.
-- Received Response:
VALUE KEY_1 15
Value_for_key_1
END
-- Received response in 0.5854 ms.

-- Sending Request:
SET KEY_2 34 43 112
Value_for_key_2
-- Expecting a 'STORED' response with no data.
-- Received Response:
STORED
-- Received response in 1.5971 ms.

-- Sending Request:
GET KEY_1 KEY_2
-- Expecting a VALUE response for both keys and their values.
-- Received Response:
VALUE KEY_1 15
Value_for_key_1
VALUE KEY_2 15
Value_for_key_2
END
-- Received response in 0.8744 ms.


 Press Enter to continue...
```