70042 Group Project Report: Memory Allocation for Kernel-Bypass Frameworks

Ethan Sim (ers19) Alvin Jonathan (aj22) Brendan Cheu (tcc22)
Christopher Adnel (ca222) Si Wei Liang (sl222)

Supervisor: Marios Kogias **Date**: 9th March 2022

1. Background

Caching systems store responses to frequently-made requests from users within main memory [1]. This reduces expensive resource usage (e.g., backend databases), which lowers latency [2]. Many popular web applications, such as Facebook and Wikipedia, use Memcached, a key-value store-based caching system termed "the gold standard of web caching" [3]. Despite its popularity, Memcached suffers from well-documented performance limitations. Notably, when Memcached is run on vanilla operating systems, packets are processed by these operating systems' network stacks, and significant delays are incurred due to system calls [4]. As the number of system calls made increases with the number of packets received [5], aggregating requests into fewer packets of greater size will reduce these delays, improving performance.

However, if request aggregation involves the kernel when processing packets, this overhead is not absorbed, only transferred. Kernel-bypassing frameworks, such as the Data Plane Development Kit (DPDK) [6], overcome this by manipulating packets entirely within userspace. Yet, modifying existing applications to use these frameworks may be costly, as the application must now implement and attain high performance on a dedicated network stack [2]. We therefore implement a software middlebox on top of DPDK, which absorbs the packet-processing overhead without requiring the modification of deployed applications.

2. Project Architecture

In the context of our project, *clients* are users seeking to retrieve values from Memcached, a key-value store. These clients issue `get` requests using User Datagram Protocol (UDP) in the format **"GET <key 1> <key 2>..."**. Requests contain at least one key, but, per the Memcached protocol, cannot exceed the length of a single UDP packet [7]. Upon receiving a get request, Memcached returns the corresponding value for each key in an individual UDP reply with the format **"VALUE <request id> <value length> <unique id> <actual value>"**, which may comprise multiple packets. Our *middlebox* sits between clients and Memcached: it will aggregate client requests before passing these aggregated requests on to Memcached. Upon receiving a reply from Memcached, it will steer the returned values from Memcached to the respective clients.

Firstly, the Middlebox receives the packet at the Network Interface Card (NIC). The Middlebox will bypass the kernel space to begin filtering and processing the packets at the user space. This filtering is conducted by looking at the headers of the packet. Only packets that use Internet Protocol (IP), communicated via UDP and containing expected source and destination IP addresses and ports are retained.

Secondly, the Memcached UDP protocol frame header (not to be confused with the UDP header of the transport layer) at the start of the payload segment of the packet will be processed. The frame header is 8 bytes long, and only the first 2 bytes, which represents the client's request id, will be used by the Middlebox.

Thirdly, the Middlebox will process the payload to extract the keys in the client's request and merge each key with a new request ID. The new request ID is generated by the Middlebox to identify each request made by the Middlebox to Memcached. The result of the merging will be used as separate keys for the hashmap. The hashmap stores this newly generated key and the client info to keep a record of the clients that expects a reply for each key in each Memcached request. Concurrently, a unique key list, that filters out requests for the same key, is being populated.

Finally, when the unique key list fully fills a packet, the keys will be sent as a single request to Memcached. Middlebox implements an algorithm to handle low loads when we do not have enough keys to fill a packet. The Middlebox will populate headers of the outgoing network packet as the packet exits.

When the Memcached server returns with the values for each key separately, the hashmap will then be used to look for the relevant clients to forward each reply to. Figure 1 summarises our approach.

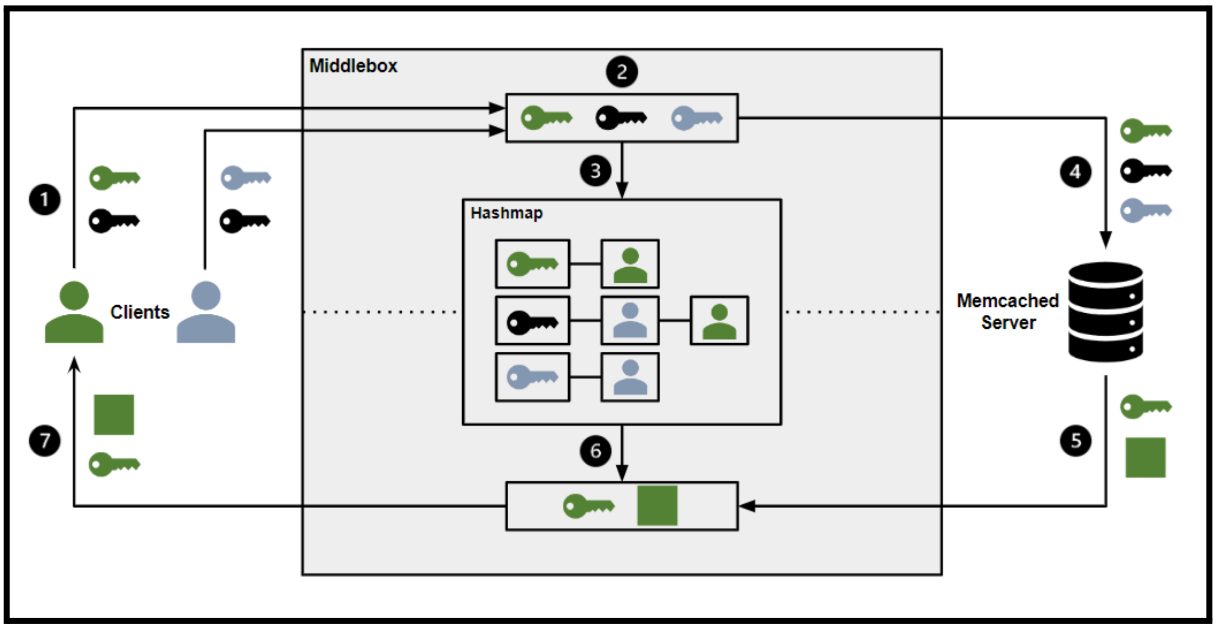


Figure 1: Summary of middlebox operation. Two clients each send a two-key get request to the middlebox. The black key appears in both clients' requests (1). Upon reaching the middlebox, the requested keys are extracted and combined into a single packet; duplicates are removed (2). Client information, such as the IP address, is concurrently extracted and associated with the requested keys using a hashmap (3). The combined packet is then sent to Memcached (4). Upon receiving a reply to a requested key containing the key and its associated value (5), the key is used to query the hashmap, identifying the requesting client (6), and the reply is forwarded accordingly (7).

3. Implementation

The development of the Middlebox took reference from the team supervisor's previous work to achieve the following functionalities [8]. Firstly the Middlebox, being a latency-critical component, achieves kernel-bypassing by using DPDK libraries as seen in figure 2.

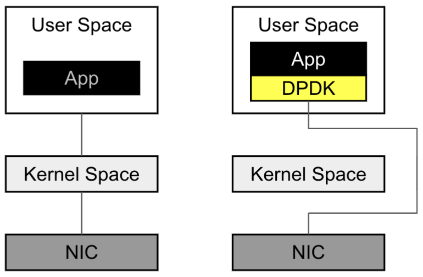


Figure 2: How applications communicate with NIC without (left) and with (right) DPDK

DPDK libraries allow the Middlebox to communicate with the NIC and obtain packets from the NIC's queue without using the kernel space, reducing overhead incurred due to system calls and context switches. Additionally, using DPDK avoids system calls when memory allocation is performed during runtime [6]. DPDK manages its own pool of memory which is allocated at the start of the program. Therefore, memory allocation can be performed using the memory pool without the need to use functions, such as malloc(), that involve system calls. Memory is also reused whenever possible.

At low loads, there will not be enough requests from clients to fully fill a packet. Thus, an algorithm, seen in figure 3, is implemented to avoid a scenario where the client is waiting for too long or the Memcached is left idling while client requests are being processed. This is achieved by ensuring that the outstanding number of packets at the server is not too low and the timing each request stays in the Middlebox is not too long using a configurable timeout.

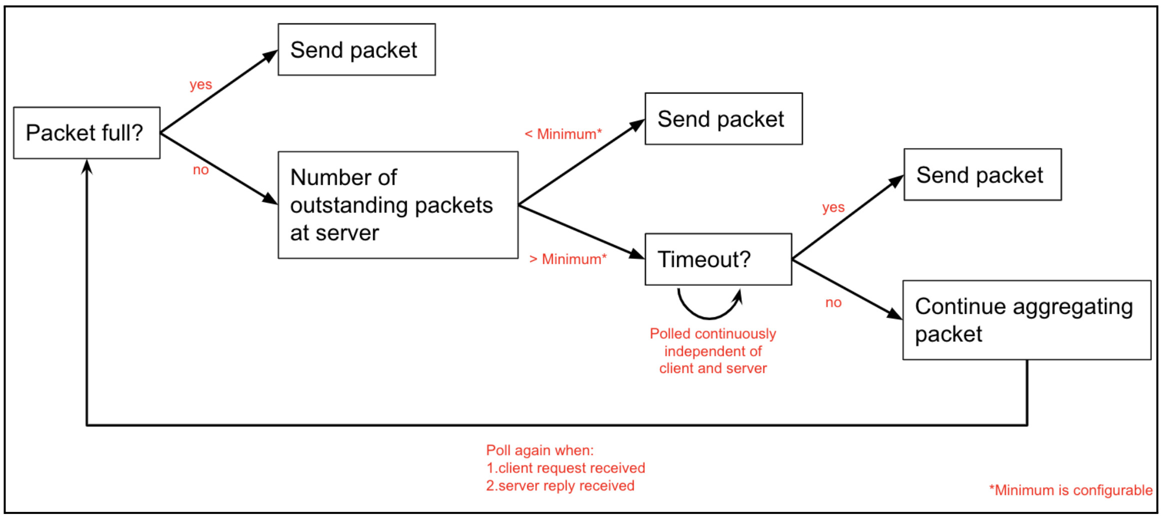
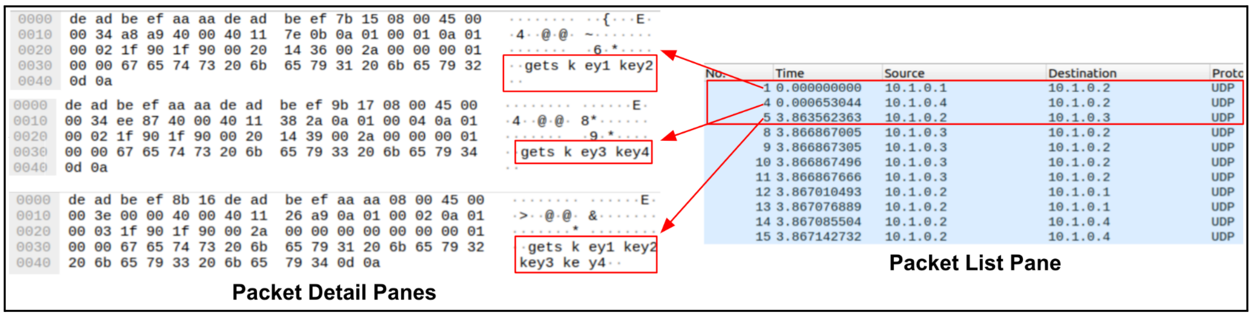


Figure 3: Algorithm implemented in Middlebox

4. **Evaluation**

This section describes the tests performed to ensure that the expected functionalities of the Middlebox are met. Screenshots of Wireshark's "Packet Detail Panes" and "Packet List Panes" show each packet's content, and source and destination I.P respectively, in the same sequence, for all tests. The tests involves client 1 (10.1.0.1) and client 2 (10.1.0.4) communicating with Memcached (10.1.0.3) through the Middlebox (10.1.0.2).

The first test conducted ensured that the middlebox was correctly merging packets from multiple clients and follows the Algorithm in section 3. This can be observed in the wireshark screenshots in figure 4.



```
0000  de ad be ef aa aa de ad  be ef 7b 15 08 00 45 00   ········ ··{···E·
0010  00 34 a8 a9 40 00 40 11  7e 0b 0a 01 00 01 0a 01   ·4··@·@· ~·······
0020  00 02 1f 90 1f 90 00 20  14 36 00 2a 00 00 00 01   ········ ·6·*····
0030  00 00 67 65 74 73 20 6b  65 79 31 20 6b 65 79 32   ··gets k ey1 key2
0040  0d 0a                                              ··

0000  de ad be ef aa aa de ad  be ef 9b 17 08 00 45 00   ········ ······E·
0010  00 34 ee 87 40 00 40 11  38 2a 0a 01 00 04 0a 01   ·4··@·@· 8*······
0020  00 02 1f 90 1f 90 00 20  14 39 00 2a 00 00 00 01   ········ ·9·*····
0030  00 00 67 65 74 73 20 6b  65 79 33 20 6b 65 79 34   ··gets k ey3 key4
0040  0d 0a                                              ··

0000  de ad be ef 8b 16 de ad  be ef aa aa 08 00 45 00   ········ ······E·
0010  00 3e 00 00 40 00 40 11  26 a9 0a 01 00 02 0a 01   ·>··@·@· &·······
0020  00 03 1f 90 1f 90 00 2a  00 00 00 00 00 00 00 01   ·······* ········
0030  00 00 67 65 74 73 20 6b  65 79 31 20 6b 65 79 32   ··gets k ey1 key2
0040  20 6b 65 79 33 20 6b 65  79 34 0d 0a               key3 ke y4··
```

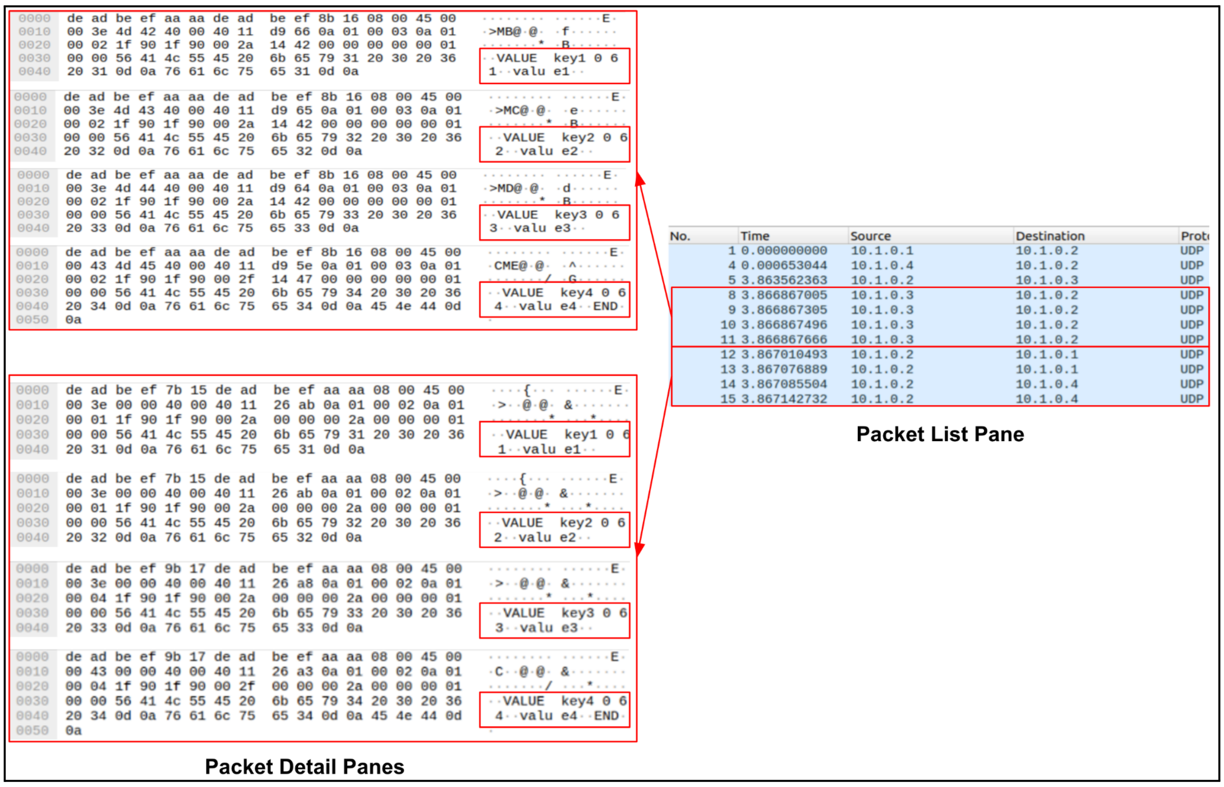
Packet Detail Panes

```
No.   Time          Source      Destination  Protc
   1 0.000000000   10.1.0.1    10.1.0.2     UDP
   4 0.000653044   10.1.0.4    10.1.0.2     UDP
   5 3.863562363   10.1.0.2    10.1.0.3     UDP
   8 3.866867005   10.1.0.3    10.1.0.2     UDP
   9 3.866867305   10.1.0.3    10.1.0.2     UDP
  10 3.866867496   10.1.0.3    10.1.0.2     UDP
  11 3.866867666   10.1.0.3    10.1.0.2     UDP
  12 3.867010493   10.1.0.2    10.1.0.1     UDP
  13 3.867076889   10.1.0.2    10.1.0.1     UDP
  14 3.867085504   10.1.0.2    10.1.0.4     UDP
  15 3.867142732   10.1.0.2    10.1.0.4     UDP
```

Packet List Pane

Figure 4: Client 1 sends requests for key1 and key 2 and client 2 sends requests for key 3 and key4. Middlebox successfully merges the packets and forwards one request ("gets key1 key2 key3 key4") to server. Also, Middlebox follows the algorithm and sends to Memcached even though it is not fully filled as we are below the pre-set number of outstanding requests at Memcached. Middlebox is expected to produce the same result as above in a timeout as well.

The return path of replies from Memcached in the first test allowed us to ensure that the Middlebox is able to route the replies from Memcached to the right clients using the Hashmap. This can be observed in figure 5, which shows the values returned by Memcached to the multikey `get` request made in figure 4.



```
0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  00 3e 4d 42 40 00 40 11  d9 66 0a 01 00 03 0a 01   ·>MB@·@· ·f······
0020  00 02 1f 90 1f 90 00 2a  14 42 00 00 00 00 00 01   ·······* ·B······
0030  00 00 56 41 4c 55 45 20  6b 65 79 31 20 30 20 36   ··VALUE  key1 0 6
0040  20 31 0d 0a 76 61 6c 75  65 31 0d 0a                1··valu e1··

0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  00 3e 4d 43 40 00 40 11  d9 65 0a 01 00 03 0a 01   ·>MC@·@· ·e······
0020  00 02 1f 90 1f 90 00 2a  14 42 00 00 00 00 00 01   ·······* ·B······
0030  00 00 56 41 4c 55 45 20  6b 65 79 32 20 30 20 36   ··VALUE  key2 0 6
0040  20 32 0d 0a 76 61 6c 75  65 32 0d 0a                2··valu e2··

0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  00 3e 4d 44 40 00 40 11  d9 64 0a 01 00 03 0a 01   ·>MD@·@· ·d······
0020  00 02 1f 90 1f 90 00 2a  14 42 00 00 00 00 00 01   ·······* ·B······
0030  00 00 56 41 4c 55 45 20  6b 65 79 33 20 30 20 36   ··VALUE  key3 0 6
0040  20 33 0d 0a 76 61 6c 75  65 33 0d 0a                3··valu e3··

0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  00 43 4d 45 40 00 40 11  d9 5e 0a 01 00 03 0a 01   ·CME@·@· ·^······
0020  00 02 1f 90 1f 90 00 2f  14 47 00 00 00 00 00 01   ·······/ ·G······
0030  00 00 56 41 4c 55 45 20  6b 65 79 34 20 30 20 36   ··VALUE  key4 0 6
0040  20 34 0d 0a 76 61 6c 75  65 34 0d 0a 45 4e 44 0d    4··valu e4··END·
0050  0a                                                 ·

0000  de ad be ef 7b 15 de ad  be ef aa aa 08 00 45 00   ····{··· ······E·
0010  00 3e 00 00 40 00 40 11  26 ab 0a 01 00 02 0a 01   ·>··@·@· &·······
0020  00 01 1f 90 1f 90 00 2a  00 00 00 2a 00 00 00 01   ·······* ···*····
0030  00 00 56 41 4c 55 45 20  6b 65 79 31 20 30 20 36   ··VALUE  key1 0 6
0040  20 31 0d 0a 76 61 6c 75  65 31 0d 0a                1··valu e1··

0000  de ad be ef 7b 15 de ad  be ef aa aa 08 00 45 00   ····{··· ······E·
0010  00 3e 00 00 40 00 40 11  26 ab 0a 01 00 02 0a 01   ·>··@·@· &·······
0020  00 01 1f 90 1f 90 00 2a  00 00 00 2a 00 00 00 01   ·······* ···*····
0030  00 00 56 41 4c 55 45 20  6b 65 79 32 20 30 20 36   ··VALUE  key2 0 6
0040  20 32 0d 0a 76 61 6c 75  65 32 0d 0a                2··valu e2··

0000  de ad be ef 9b 17 de ad  be ef aa aa 08 00 45 00   ········ ······E·
0010  00 3e 00 00 40 00 40 11  26 a8 0a 01 00 02 0a 01   ·>··@·@· &·······
0020  00 04 1f 90 1f 90 00 2a  00 00 00 2a 00 00 00 01   ·······* ···*····
0030  00 00 56 41 4c 55 45 20  6b 65 79 33 20 30 20 36   ··VALUE  key3 0 6
0040  20 33 0d 0a 76 61 6c 75  65 33 0d 0a                3··valu e3··

0000  de ad be ef 9b 17 de ad  be ef aa aa 08 00 45 00   ········ ······E·
0010  00 43 00 00 40 00 40 11  26 a3 0a 01 00 02 0a 01   ·C··@·@· &·······
0020  00 04 1f 90 1f 90 00 2f  00 00 00 2a 00 00 00 01   ·······/ ···*····
0030  00 00 56 41 4c 55 45 20  6b 65 79 34 20 30 20 36   ··VALUE  key4 0 6
0040  20 34 0d 0a 76 61 6c 75  65 34 0d 0a 45 4e 44 0d    4··valu e4··END·
0050  0a                                                 ·
```

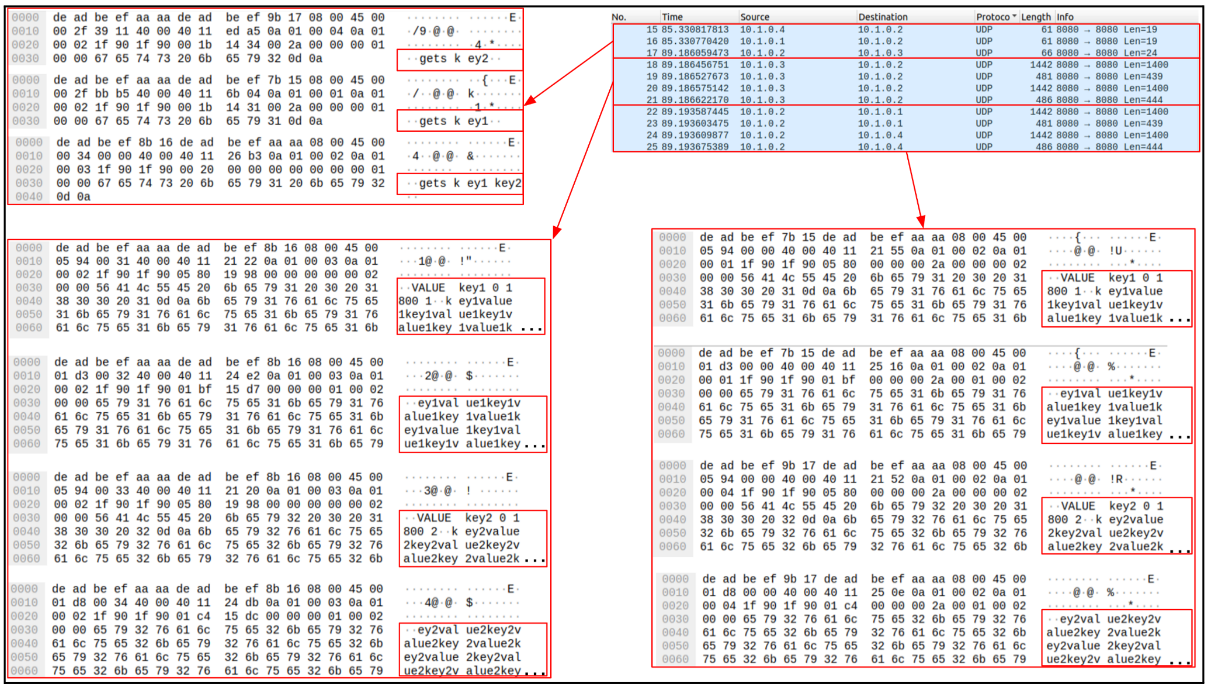
Packet Detail Panes

```
No.  Time         Source    Destination  Proto
 1   0.000000000  10.1.0.1  10.1.0.2     UDP
 4   0.000653044  10.1.0.4  10.1.0.2     UDP
 5   3.863562363  10.1.0.2  10.1.0.3     UDP
 8   3.866867005  10.1.0.3  10.1.0.2     UDP
 9   3.866867305  10.1.0.3  10.1.0.2     UDP
10   3.866867496  10.1.0.3  10.1.0.2     UDP
11   3.866867666  10.1.0.3  10.1.0.2     UDP
12   3.867010493  10.1.0.2  10.1.0.1     UDP
13   3.867076889  10.1.0.2  10.1.0.1     UDP
14   3.867085504  10.1.0.2  10.1.0.4     UDP
15   3.867142732  10.1.0.2  10.1.0.4     UDP
```

Packet List Pane

Figure 5: The top left image shows the values returned by Memcached for all 4 keys (in figure 4) separately. From the packet list pane, it can be observed that each reply in the bottom left image is being routed correctly to client 1 and 2.

The second test ensures that when a request containing a single key has a value that requires a multipacket reply from Memcached to complete, the Middlebox is able to route the replies to the correct client, as seen in figure 6.



```
0000  de ad be ef aa aa de ad  be ef 9b 17 08 00 45 00   ········ ······E·
0010  00 2f 39 11 40 00 40 11  ed a5 0a 01 00 04 0a 01   ·/9·@·@· ········
0020  00 02 1f 90 1f 90 00 1b  14 34 00 2a 00 00 00 01   ········ ·4·*····
0030  00 00 67 65 74 73 20 6b  65 79 32 0d 0a            ··gets k ey2··

0000  de ad be ef aa aa de ad  be ef 7b 15 08 00 45 00   ········ ··{···E·
0010  00 2f bb b5 40 00 40 11  6b 04 0a 01 00 01 0a 01   ·/··@·@· k·······
0020  00 02 1f 90 1f 90 00 1b  14 31 00 2a 00 00 00 01   ········ ·1·*····
0030  00 00 67 65 74 73 20 6b  65 79 31 0d 0a            ··gets k ey1··

0000  de ad be ef 8b 16 de ad  be ef aa aa 08 00 45 00   ········ ······E·
0010  00 34 00 00 40 00 40 11  26 b3 0a 01 00 02 0a 01   ·4··@·@· &·······
0020  00 03 1f 90 1f 90 00 20  00 00 00 00 00 00 00 01   ········ ········
0030  00 00 67 65 74 73 20 6b  65 79 31 20 6b 65 79 32   ··gets k ey1 key2
0040  0d 0a                                              ··
```

```
0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  05 94 00 31 40 00 40 11  21 22 0a 01 00 03 0a 01   ···1@·@· !"······
0020  00 02 1f 90 1f 90 05 80  19 98 00 00 00 00 00 02   ········ ········
0030  00 00 56 41 4c 55 45 20  6b 65 79 31 20 30 20 31   ··VALUE  key1 0 1
0040  38 30 30 20 31 0d 0a 6b  65 79 31 76 61 6c 75 65   800 1··k ey1value
0050  31 6b 65 79 31 76 61 6c  75 65 31 6b 65 79 31 76   1key1val ue1key1v
0060  61 6c 75 65 31 6b 65 79  31 76 61 6c 75 65 31 6b   alue1key 1value1k ...

0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  01 d3 00 32 40 00 40 11  24 e2 0a 01 00 03 0a 01   ···2@·@· $·······
0020  00 02 1f 90 1f 90 01 bf  15 d7 00 00 00 01 00 02   ········ ········
0030  00 00 65 79 31 76 61 6c  75 65 31 6b 65 79 31 76   ··ey1val ue1key1v
0040  61 6c 75 65 31 6b 65 79  31 76 61 6c 75 65 31 6b   alue1key 1value1k
0050  65 79 31 76 61 6c 75 65  31 6b 65 79 31 76 61 6c   ey1value 1key1val
0060  75 65 31 6b 65 79 31 76  61 6c 75 65 31 6b 65 79   ue1key1v alue1key ...

0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  05 94 00 33 40 00 40 11  21 20 0a 01 00 03 0a 01   ···3@·@· ! ······
0020  00 02 1f 90 1f 90 05 80  19 98 00 00 00 00 00 02   ········ ········
0030  00 00 56 41 4c 55 45 20  6b 65 79 32 20 30 20 31   ··VALUE  key2 0 1
0040  38 30 30 20 32 0d 0a 6b  65 79 32 76 61 6c 75 65   800 2··k ey2value
0050  32 6b 65 79 32 76 61 6c  75 65 32 6b 65 79 32 76   2key2val ue2key2v
0060  61 6c 75 65 32 6b 65 79  32 76 61 6c 75 65 32 6b   alue2key 2value2k ...

0000  de ad be ef aa aa de ad  be ef 8b 16 08 00 45 00   ········ ······E·
0010  01 d8 00 34 40 00 40 11  24 db 0a 01 00 03 0a 01   ···4@·@· $·······
0020  00 02 1f 90 1f 90 01 c4  15 dc 00 00 00 01 00 02   ········ ········
0030  00 00 65 79 32 76 61 6c  75 65 32 6b 65 79 32 76   ··ey2val ue2key2v
0040  61 6c 75 65 32 6b 65 79  32 76 61 6c 75 65 32 6b   alue2key 2value2k
0050  65 79 32 76 61 6c 75 65  32 6b 65 79 32 76 61 6c   ey2value 2key2val
0060  75 65 32 6b 65 79 32 76  61 6c 75 65 32 6b 65 79   ue2key2v alue2key ...
```

```
No.  Time          Source    Destination  Protoco  Length  Info
15   85.330817813  10.1.0.4  10.1.0.2     UDP          61  8080 → 8080 Len=19
16   85.330770420  10.1.0.1  10.1.0.2     UDP          61  8080 → 8080 Len=19
17   89.186059473  10.1.0.2  10.1.0.3     UDP          66  8080 → 8080 Len=24
18   89.186456751  10.1.0.3  10.1.0.2     UDP        1442  8080 → 8080 Len=1400
19   89.186527673  10.1.0.3  10.1.0.2     UDP         481  8080 → 8080 Len=439
20   89.186575142  10.1.0.3  10.1.0.2     UDP        1442  8080 → 8080 Len=1400
21   89.186622170  10.1.0.3  10.1.0.2     UDP         486  8080 → 8080 Len=444
22   89.193587445  10.1.0.2  10.1.0.1     UDP        1442  8080 → 8080 Len=1400
23   89.193603475  10.1.0.2  10.1.0.1     UDP         481  8080 → 8080 Len=439
24   89.193609877  10.1.0.2  10.1.0.4     UDP        1442  8080 → 8080 Len=1400
25   89.193675389  10.1.0.2  10.1.0.4     UDP         486  8080 → 8080 Len=444
```

```
0000  de ad be ef 7b 15 de ad  be ef aa aa 08 00 45 00   ····{··· ······E·
0010  05 94 00 00 40 00 40 11  21 55 0a 01 00 02 0a 01   ····@·@· !U······
0020  00 01 1f 90 1f 90 05 80  00 00 00 2a 00 00 00 02   ········ ···*····
0030  00 00 56 41 4c 55 45 20  6b 65 79 31 20 30 20 31   ··VALUE  key1 0 1
0040  38 30 30 20 31 0d 0a 6b  65 79 31 76 61 6c 75 65   800 1··k ey1value
0050  31 6b 65 79 31 76 61 6c  75 65 31 6b 65 79 31 76   1key1val ue1key1v
0060  61 6c 75 65 31 6b 65 79  31 76 61 6c 75 65 31 6b   alue1key 1value1k ...

0000  de ad be ef 7b 15 de ad  be ef aa aa 08 00 45 00   ····{··· ······E·
0010  01 d3 00 00 40 00 40 11  25 16 0a 01 00 02 0a 01   ····@·@· %·······
0020  00 01 1f 90 1f 90 01 bf  00 00 00 2a 00 01 00 02   ········ ···*····
0030  00 00 65 79 31 76 61 6c  75 65 31 6b 65 79 31 76   ··ey1val ue1key1v
0040  61 6c 75 65 31 6b 65 79  31 76 61 6c 75 65 31 6b   alue1key 1value1k
0050  65 79 31 76 61 6c 75 65  31 6b 65 79 31 76 61 6c   ey1value 1key1val
0060  75 65 31 6b 65 79 31 76  61 6c 75 65 31 6b 65 79   ue1key1v alue1key ...

0000  de ad be ef 9b 17 de ad  be ef aa aa 08 00 45 00   ········ ······E·
0010  05 94 00 00 40 00 40 11  21 52 0a 01 00 02 0a 01   ····@·@· !R······
0020  00 04 1f 90 1f 90 05 80  00 00 00 2a 00 00 00 02   ········ ···*····
0030  00 00 56 41 4c 55 45 20  6b 65 79 32 20 30 20 31   ··VALUE  key2 0 1
0040  38 30 30 20 32 0d 0a 6b  65 79 32 76 61 6c 75 65   800 2··k ey2value
0050  32 6b 65 79 32 76 61 6c  75 65 32 6b 65 79 32 76   2key2val ue2key2v
0060  61 6c 75 65 32 6b 65 79  32 76 61 6c 75 65 32 6b   alue2key 2value2k ...

0000  de ad be ef 9b 17 de ad  be ef aa aa 08 00 45 00   ········ ······E·
0010  01 d8 00 00 40 00 40 11  25 0e 0a 01 00 02 0a 01   ····@·@· %·······
0020  00 04 1f 90 1f 90 01 c4  00 00 00 2a 00 01 00 02   ········ ···*····
0030  00 00 65 79 32 76 61 6c  75 65 32 6b 65 79 32 76   ··ey2val ue2key2v
0040  61 6c 75 65 32 6b 65 79  32 76 61 6c 75 65 32 6b   alue2key 2value2k
0050  65 79 32 76 61 6c 75 65  32 6b 65 79 32 76 61 6c   ey2value 2key2val
0060  75 65 32 6b 65 79 32 76  61 6c 75 65 32 6b 65 79   ue2key2v alue2key ...
```

Figure 6: Using the information in the Packet List Pane with each image: **Top left image**: client 1 and 2 made a request for key1 and key2 respectively, Middlebox sends the merged request to Memcached. **Bottom left image**: Memcached replies with a value that exceeds 1 packet length for each key (key1 with key1value1key1… and key2

with key2value2key2…). **Bottom right image:** key1 is forwarded to client 1 (10.1.0.1) and key2 is forwarded to client 2 (10.1.0.4), indicating forwarding done correctly.

* Details to set up, run and test the Middlebox with the Clients and Memcached can be found at: https://github.com/ajonw/group-project.

5. Limitations

The Middlebox does not guarantee that it is able to correctly pass to clients multipacket replies if in the same aggregated packet, 2 or more clients expect multipacket replies. When Memcached makes a multi packet reply, only the first packet contains information that the Middlebox uses to map back to the respective clients. Subsequent packets do not contain such information, thus the Middlebox is unable to identify which clients it should be forwarded to.

6. Future Work

First, by leveraging the existing Middlebox, future work could look into benchmarking the performance and quantifying the performance benefit of having the middlebox aggregate requests, by comparing it with the case where clients send requests directly to the server in the same environment. Lastly, future research may look into the possibility of enabling requests to be forwarded to multiple servers instead of just one. This will aid load balancing by spreading packets among multiple servers.

7. References

[1] Mertz J, Nunes I. A qualitative study of application-level caching. IEEE Transactions on Software Engineering. 2016 Dec 1;43(9):798-816.

[2] Ghigoff Y, Sopena J, Lazri K, Blin A, Muller G. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. InNSDI 2021 Apr 12 (Vol. 3, p. 4).

[3] Wiger N, Imalsina R. Performance at Scale with Amazon ElastiCache San Diego, California: Amazon Web Services; 2019 p. 3–4.

[4] Chidambaram V, Ramamurthi D. Performance analysis of memcached. unpublished Manuscript.[Online]. Available: http://citeseerx. ist. psu. edu/viewdoc/download. 2010.

[5] Hruby T, Crivat T, Bos H, Tanenbaum AS. On Sockets and System Calls: Minimizing Context Switches for the Socket {API}. In2014 Conference on Timely Results in Operating Systems ({TRIOS} 14) 2014.

[6] Memory in DPDK, part 1: General concepts [Internet]. DPDK. LF Projects; 2019 [cited 2023 Mar 9]. Available from: https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/

[7] Mate B. protocol.txt [Internet]. 2022. Available from: https://github.com/memcached/memcached/blob/master/doc/protocol.txt

[8] Marios K. netstack [Internet]. 2020. Available from: https://github.com/epfl-dcsl/r2p2/tree/master/netstack