The Hebrew University of Jerusalem
School of Engineering and Computer Science
Israel

LogMemcached
An RDMA based Continuous Cache Replication

LogMemcached
שכפול זכרון מטמון מבוסס RDMA

by
Samyon Ristov

Supervised by
Prof. Danny Dolev
and
Dr. Yaron Weinsberg
and
Dr. Tal Anker

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

April 2017
ניסן ה'תשע"ז

Abstract

One of the advantages of cloud computing is its ability to quickly scale out services to meet demand. A common technique to mitigate the increasing load in these services is to deploy a cache.

Although it seems natural that the caching layer would also deal with availability and fault tolerance, these issues are nevertheless often ignored, as the cache has only recently begun to be considered a critical system component. A cache may evict items at any moment, and so a failing cache node can simply be treated as if the set of items stored on that node have already been evicted. However, setting up a cache instance is a time-consuming operation that could inadvertently affect the service's operation.

This paper addresses this limitation by introducing cache replication at the server side by expanding Memcached (which currently provides availability via client side replication). This paper presents the design and implementation of LogMemcached, a modification of Memcached's internal data structures to include state replication via RDMA to provide an increased system availability, improved failure resilience and enhanced load balancing capabilities without compromising performance and with introducing a very low CPU load, while keeping the main principles of Memcached's Design Philosophy.

Abstract - Hebrew

אחד היתרונות של מחשוב ענן הינו היכולת להתרחב במהירות על מנת להתמודד עם הדרישות. גדילת קצב הבקשות מצד הלקוח מגדילה בתורה את העומס על שכבת המידע של השירות. הגישה המקובלת כיום להתמודדות עם העומס הינה פרישת זכרון מטמון.

זה אך טבעי ששכבת זכרון המטמון תתמודד גם עם נושאים כמו זמינות, סבילות מול תקלות ועקביות, אך בפועל נושאים אלה לרוב זוכים להתעלמות משום שרק בזמן האחרון זכתה שכבת זכרון המטמון להחשב כרכיב קריטי במערכת. משום שזכרון המטמון יכול למחוק כל פריט שמור בכל רגע נתון, ניתן להתייחס לכשל של כל מערכת זכרון המטמון פשוט בתור מערכת בה נמחקו כל הפריטים השמורים, ומשום שתמיד ניתן להוסיף מחדש כל פריט שנמחק, כשל במערכת נראה כאינו קריטי.

עם זאת, במקרים רבים יצירת זכרון מטמון חדש לוקחת זמן רב, מה שיכול לפגוע משמעותית בפעילות המערכת.

מסמך זה מציע גישה להתמודדות עם מגבלות אלה על ידי הצגת שכפול של זכרון המטמון בצד שרת. בעבודה זו אנו עורכים את Memcached, אחת ממערכות זכרון המטמון הנפוצות ביותר, אשר נכון לעכשיו מספקת זמינות על ידי ביצוע שכפול בצד הלקוח בלבד. עבודה זו מציגה את התכנון והמימוש של LogMemcached, התאמה של מבני הנתונים של Memcached על מנת לאפשר שכפול של מצב המערכת על ידי RDMA, ובכך לשפר את הזמינות והסבילות של המערכת, ללא התפשרות על ביצועים או עומס על המעבד, ותוך כדי שמירה על עקרונות התכנון של Memcached.

Acknowledgements

I would like to express deepest gratitude to Prof. Danny Dolev. For his helpful guidance, support, encouragement and endless patience throughout this research. I would also like to express my sincere thanks to Dr. Yaron Weinsberg and Dr. Tal Anker. This thesis would not have been possible without their help. The equipment for the thesis was provided by The Hebrew University of Jerusalem and Mellanox, making the research of the thesis' fields possible. Finally, I wish to thank my parents and friends for their unmeasurable support and love.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and Objectives

The use of key-value stores and caches is a common practice in scale-out deployments and large cloud-services. Examples of this include Facebook's use of Memcached [24], Amazon's Dynamo [13], Voldemort by LinkedIn [5] and Twitter's use of Redis [7].

Memcached, one of the most popular key-value web caches, is an open source, high-performance, distributed memory object caching system with a simple and widely adopted API. Systems that require caching tend to operate under heavy load [10, 20, 24] and so improving cache performance is very important. Many efforts have been made to decrease Memcached's latency by using Remote Direct Memory Access (RDMA) [18, 17, 29], but the main performance hit occurs when the cache is cold (e.g. after crash recovery), which, in some cases might cause a database breakdown [6]. In order to prevent a high miss-rate, a common practice is to use cache warm-up mechanisms [24, 37] or client-side replications [2].

Recent technological trends show that large cloud-services providers massively adopt kernel and CPU bypassing technologies, such as RDMA via InfiniBand, RoCE and iWARP network protocols [23]. Kernel and CPU bypassing is a natural choice for boosting Memcached performance, as well as for enabling server side state replication.

Server-side replication offers transparent-to-client failure-resilience, system upgrades and maintenance resets, and thus improves overall system availability. Replicated server states also allow for improved load balancing, which is often also achieved by client-side or proxy replication [24].

Memcached's internal design and data structures are heavily based on pointers, which are challenging to replicate. Therefore we present *LogMemcached* [9], a modified version of Memcached with an append-only cyclic log, which is more amenable to server-side-replication, and with RDMA to enable continuity of the server-side replication. LogMemcached supports Memcached's API and maintains the main principles of Memcached's Design Philosophy [3]:

- Simple Key-Value Store: Items are made up of a key, an expiration time, optional flags, and raw data.
- $O(1)$: All commands are implemented to be as fast and lock-friendly as possible.
- LRU Cache, a.k.a. "Forgetting is a Feature": Items expire after a specified amount of time.

LogMemcached has comparable performance to Memcached, but allows a simple master-slave replication scheme. Other than during initialization, the replication-master is not aware of the replication, and only the replicated-slave is responsible for the replication process. The slave machine is able to perform any required task - such as answering to Memcached's queries in read-only mode, hosting another instance of Memcached master for keys partitioning, writing backups to HDFS, or any other task - and therefore the replication process does not entail any wasted resources.

LogMemcached keeps the same API, and all its modifications are transparent to the client. All API functions are supported, and the client-server interface remains the same, based on TCP, which means that there is no need to modify client libraries in order to work with LogMemcached.

To the best of our knowledge, we are the first to study the data structures necessary to provide simple, RDMA-oriented, and continuous state replication mechanisms in key-value caches.

1.2 Thesis Outline

The following section briefly introduces RDMA and Memcached, focusing on Memcached's internal data structures. Section 3 discusses LogMemcached's design and its replication mechanism. In section 4 we evaluate our system and compare its performance, user space load and kernel space load to those of Memcached, and compare the relative advantages of server-side vs. client-side replication.

Chapter 2

Background

2.1 RDMA

Remote Direct Memory Access functionalities are implemented in InfiniBand, iWARP (internet Wide Area RDMA Protocol) and RoCE (RDMA over Converged Ethernet). Current hardware enables up to 40 Gbps of bandwidth in each direction for all three technologies (and up to 100 Gbps for InfiniBand and RoCE). RDMA provides high-throughput, low-latency networking, with zero-copy and kernel and CPU bypassing.

RDMA operates via the Verbs API. This API offers four transfer types, or "verbs": SEND, RECV, WRITE and READ. The SEND and RECV verbs allow direct data transfer between user space applications and the network interface, bypassing the kernel. When the SEND and RECV verbs are used (they are always used together), both sender and receiver must actively participate in all data transfers: before each SEND operation, the receiver must perform a RECV operation. In RDMA WRITE only the sender side is active, and the receiver side is not aware of the operation: the sender "pushes" the data to the registered receiver's memory, bypassing the kernel on both sides. RDMA READ operates in a symmetrical fashion, performed by the receiver only: the receiver "pulls" data from the registered sender's memory, bypassing the kernel on both sides; the sender is not aware of the performed READ operation.

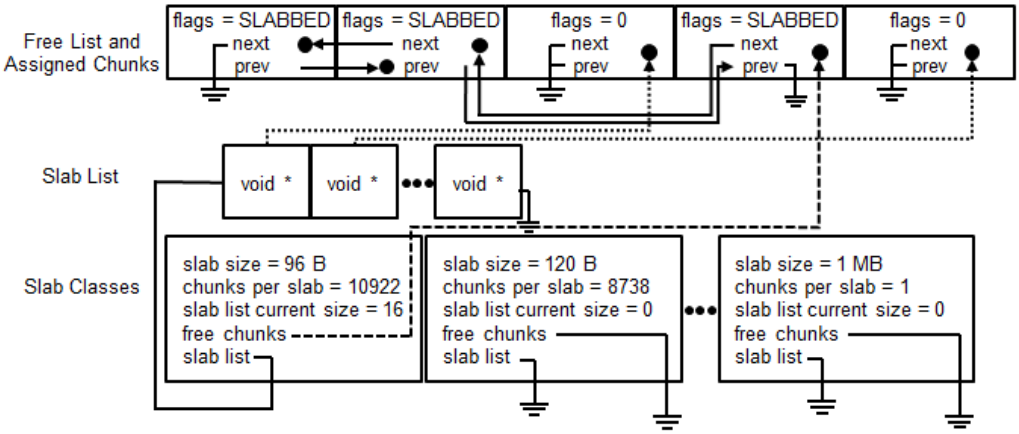

Figure 2.1: Memcached's slab allocation diagram. In this example two items are assigned to a slab class for items of size up to 96B.

2.2 Memcached

Memcached is an in-memory key-value cache, which basically operates as a remote hash-table. Memcached works with *items*, which are objects comprised of metadata, a key and a value. Items are managed using three data structures: a Slab Allocation for storage, an Association Array for quick access, and an LRU (Least Recently Used) List for eviction from the cache.

In order to store items and manage memory, Memcached implements Slab Allocation using three data structures: Slab Class Array, Slab Lists, and Free Lists (see Figure 2.1). A *Slab Class* is associated with a given size range, a given capacity (i.e. how many items it can store) and a tally of items already allocated to it. The *Slab List* is a dynamic array of pointers to assigned chunks (i.e. each entry points to a single chunk, which in turn contains a single item). Memcached allocates pages of 1MB, each associated with a given slab class; pages are further divided into chunks of size/quantity defined by the associated slab class. The *Free List* is a doubly-linked-list of available (i.e. unassigned) chunks.

To enable fast access, Memcached uses an association array (hash table) of items. Each item appears in a list of items with the same hash key, which may be located in different slabs, resulting in a cross-slab linked list.

Eviction is implemented using a Least Recently Used (LRU) doubly-linked-list. Memcached

maintains one LRU list per slab.

Memcached provides a simple API, consisting of the following primitives: Get, Gets (a Get which also returns an item's CAS value), Set, Add (which creates and sets a not-yet-existent item), Replace (which sets an existing item), Append, Prepend, Cas (Compare-and-swap), Delete, Increment, Decrement and Touch (which updates an item's expiration time).

Chapter 3

LogMemcached Design

3.1 Objectives

LogMemcached seeks to improve Memcached's system availability, failure resilience and load balancing without compromising performance and CPU load, and while keeping the main principles of Memcached's Design Philosophy.

System availability and failure resilience are improved by adding a continuously replicated cache and by providing a simple, fault-tolerant, master-slave replication mechanism. These prevent the cold-cache problem described in section 1 and and also simplify system setup and configuration. The simplified setup and configuration, in turn, allow for maintenance of a replicated backup and for handling of failure events. Load balancing is improved by the use of read-only replicas [24].

3.2 Architecture Design

State replication refers to the replication of the system's internal state and data, as opposed to *message replication*, which refers to the replication of messages which can then be replayed to recreate the state. State replication offers lower memory requirements and lower CPU

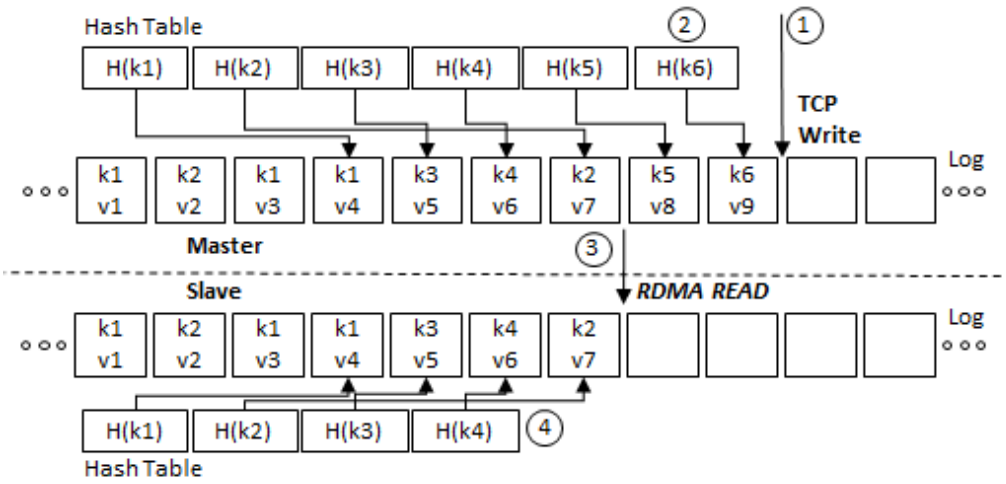

Figure 3.1: LogMemcached's storing and replication mechanism.

overhead: message replication is simpler to implement, but requires additional memory to store the replicated messages [26] and (if using TCP) to support the kernel's buffer. Message replication also requires additional CPU to manage the TCP stack and to replay the replicated messages.

RDMA was chosen for LogMemcached's replication mechanism in order to reduce both memory and CPU usage. Using RDMA and state replication together reduces memory usage by eliminating the need to store replicated messages, since the state is copied directly from the master's memory into the slave's memory. RDMA also eliminates the need to maintain the TCP stack or to perform kernel context switches, and thus reduces CPU usage. (Eliminating the kernel's context switch is essential: Leverich *et al.* [20] found that only 17% of Memcached's runtime is spent in user code vs. 83% in kernel code, and of that 83%, 37% is spent in Soft IRQ context).

RDMA provides four main verbs for data transmission: SEND, RECV, READ and WRITE. LogMemcached uses RDMA READ verbs performed by the slave. This lets each slave be responsible for its own state and speed. By contrast, SEND and RECV require careful communication management which involves the CPU for both master and slave and requires additional memory. Although WRITE offers a plausible solution, it is less scalable since it requires that the master maintain every slave's state and repeatedly validate that every slave is alive.

Memcached's data structures are not suited for state replication, since their heavy reliance on pointers requires gathering and modifying many memory areas with each replication. In order to support replication we use an append-only cycled log for item storage, instead of the original slab list and their associated slab classes. Replicating data from logs is easy, since all the items are arranged in one continuous segment.

Memcached's native hash table is kept unchanged. The free list mechanism that appears in Memcached is kept too, but we use it only to manage items' metadata allocations. Only one free list is used, because regardless of item's size, the metadata size is constant. Neither of these data structures is replicated, and their state is reconstructed after performing the replication.

Memcached's API is fully supported, and is easy to implement: Get and Gets are performed by a hash table lookup to reach the item in the log, and Gets also returns the item's CAS value which is now stored as part of the item. Set is performed by first allocating memory in the log - marking it as DIRTY - then by reading an item's value from the socket and writing the value to the allocated memory, adding the key to the hash table, and changing the item's flag from DIRTY to STORED. Add and Replace are implemented similarly to Set, but with an initial check to see if the key appears in the hash table. Append and Prepend are also similar, but also copy the item from the log to the log head, and then append/prepend the new value. Cas stores the CAS value in a new item. Increment and Decrement first look at the old value, then write the new modified value to the log head. Touch writes the existing item to the log head, with its expiration time modified.

Delete is the only command that is implemented differently: The item key is removed from the master's hash table, and a special item, marked as DELETED, is appended to the end of the log to signal that the replication slave should delete the respective item from its hash table as well. This is necessary since the hash table is not replicated.

Item eviction is performed as follows: a second thread constantly monitors the log's status, and when the log is close enough to full, the thread removes items from the log tail. When an item is removed, the thread checks if the item's key appears in the hash table. If it does, the thread then checks if it points to the removed item's place, and if so removes the key from the

hash table. (If the key appears in the hash table but points to an other location, that means that a newer instance of that item exists, and there is no need to remove the entry from the hash table). To release enough free space for incoming items of different sizes, eviction is done in batches.

The eviction algorithm removes items in FIFO order by default, but since Memcached uses LRU eviction, this option is available too. In LRU mode, The Get command returns the item to the client, but also writes it to the log head. In both Memcached and LogMemcached an item may be reordered because of a Get command only once every 60 seconds. This limitation prevents rapid writing to the log head after every Get command. In LogMemcached a special counter in the item's metadata counts how many Get commands requested the item. If LRU mode is configured, then any value above 0, once every 60 seconds, initiates the rewriting mechanism: setting the threshold a to higher value approximates Least Frequently Used (LFU) mode.

Figure 3.1 illustrates LogMemcached's storing and replication mechanism. The initialization process registers the log to the network interface and sends its address to the slave machine. Following this, the normal operation sequence begins when the master machine receives a Set request via standard Memcached TCP communication (1), appends the received item to its log head and adds a reference to the hash table (2). The slave machine then performs an RDMA READ operation, copying the item from the master's log to its own log (3), independent of the master's CPU. Finally, the slave machine validates that the received item is marked with the STORED flag and not the DIRTY flag, and adds a reference to the item to its hash table (4). If the item is marked as DIRTY that means that the item has not yet been stored (on the master), which in turn means that the slave machine will replicate that memory segment again. The slave machine does not know when items arrive at the master machine, and so it must constantly pull items from the master's log and place them at the head of its own log, advancing the head only when it copies items marked as 'stored' and overwriting 'dirty' items.

LogMemcached relaxes consistency in favor of availability. The approach is similar to Redis and Pileus [30], providing eventual consistency to achieve better performances both in replication and query responses.

3.3 Pseudocode

This section describes the implementation of the API in Memcached and LogMemcached, presented in a form of a pseudocode. The provided pseudocode is an abstraction of the real code, therefore, for readability, some details were omitted, as Memcached's flags; the way LogMemcached handles full buffer and more. First, we define the pseudocode's primitives, followed by the API implementation description.

Algorithm 3.1: Memcached Primitives

```
hash(val1) - receives a value and returns its hash value.
item_lock(val1) - 'item' lock.
item_unlock(val1) - unlocks 'item' lock.
association_map_find(val1, val2) - finds an item in the association map with key 'val1' and
hash value 'val2'.
association_map_insert(val1, val2) - interests an item into the association map with key '
val1' and hash value 'val2'.
association_map_delete(val1, val2) - deletes an item from the association map with key 'val1
' and hash value 'val2'.
lru_lock(val1) - LRU lock.
lru_unlock(val1) - unlocks LRU lock.
lru_list_link(val1) - puts item given in 'val1' to the head of the LRU list.
lru_list_unlink(val1) - removes item given in 'val1' from the LRU list.
slabs_global_lock - 'slabs' lock. No input needed.
slabs_global_unlock - unlocks 'slabs' lock. No input needed.
slabs_allocate(val1, val2) - allocates an item from slab id 'val1' and with size 'val2'.
slabs_free(val1) - removes 'val1' from slabs.
calculate_slab_id(val1) - calculates the minimal slab size to fit item with size 'val1'.
copy_value(val1, val2, val3, val4) - copies 'val1' and 'val2' into 'val3'. 'val4' is either
APPEND or PREPEND, and defines the copy order.
lru_pull_tail(val1, val2) - pulls item from the LRU lists. If 'val2' is NULL, the function
pulls only unlocked and expired items from the list of items sized 'val1'. If 'val2' is
EVICT, the function pulls an LRU item from the LRU list of items sized 'val1'.
```

Algorithm 3.2: LogMemcached Primitives

```
hash(val1) - receives a value and returns its hash value.
item_lock(val1) - 'item' lock.
item_unlock(val1) - unlocks 'item' lock.
association_map_find(val1, val2) - finds an item_metadata in the association map with key '
val1' and hash value 'val2'.
association_map_insert(val1, val2) - interests an item_metadata into the association map
with key 'val1' and hash value 'val2'.
association_map_delete(val1, val2) - deletes an item_metadata from the association map with
key 'val1' and hash value 'val2'.
freelist_global_lock - 'freelist' lock. No input needed.
freelist_global_unlock - unlocks 'freelist' lock. No input needed.
freelist_allocate() - allocated an item_metadata from freelist.
freelist_free(val1) - removes 'val1' from freelist.
copy_value(val1, val2, val3, val4) - copies 'val1' and 'val2' into 'val3'. 'val4' is either
APPEND or PREPEND, and defines the copy order.
lru_mode_get_count - a setting defining how many get queries needed before an LRU reordering
 is performed.
item_metadata - metadata that is handled by the association map and the freelist.
item_metadata is not replicated directly. item_metadta.item points to the item in the log.
item_data - the item itself, with its flags, key and value.
memlog_global_lock - 'memlog' lock. No input needed.
memlog_global_unlock - unlocks 'memlog' lock. No input needed.
memlog_allocate(val1) - allocated item_data of size 'val1' from the log.
```

Algorithm 3.3: Memcached item_allocate

```
metadata.slab_id = calculate_slab_id(metadata.size)
index = 0
item = NULL
lru_pull_tail(metadata.size, NULL) // Try to reclaim memory first
while item not allocated and index < 10
    slabs_global_lock
    item = slabs_allocate(metadata.slab_id, metadata.size)
    slabs_global_unlock
    index = index + 1
    if item not allocated:
        lru_pull_tail(metadata.size, EVICT)
if item allocated:
    item.metadata = metadata
return item
```

Algorithm 3.4: LogMemcached item_allocate

```
index = 0
item_data = NULL
item_metadata = NULL
while item_data not allocated and index < 3
    memlog_global_lock
    it_data = memlog_allocate(metadata.size)
    memlog_global_unlock
    index = index + 1
    if item_data not allocated:
        sleep 1 millisecond
if item_data is allocated:
    freelist_global_lock
    item_metadata = freelist_allocate()
    freelist_global_unlock
    // if there was a place for it_data, there is also a place for item's metadata
    item_metadata.item = item_data
    add ITEM_DIRTY to item_metadata.item.flags
return item_metadata
```

Algorithm 3.5: Memcached Get and Gets

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
item = association_map_find(key, hash_value)
if item found and expired:
    association_map_delete(item, hash_value)
    lru_lock(item.slab_id)
    lru_list_unlink(item)
    lru_unlock(item.slab_id)
    slabs_global_lock
    slabs_free(item)
    slabs_global_unlock
if item found and not expired and time since last updated ≥ 60 seconds:
    lru_lock(item.slab_id)
    lru_list_unlink(item)
    lru_list_link(item)
    lru_unlock(item.slab_id)
item_unlock(hash_value)
return item
// in "Gets" use: return item, item.cas
```

Algorithm 3.6: LogMemcached Get and Gets

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
item_metadata = association_map_find(key, hash_value)
if item_metadata found and expired:
    association_map_delete(item_metadata, hash_value)
    freelist_global_lock
    freelist_free(item_metadata)
    freelist_global_unlock
if item_metadata found and not expired and time since last updated ≥ 60 seconds:
    if LRU mode enabled and item_metadata.get_count ≥ lru_mode_get_count:
        new_item_metadata = item_allocate(key, metadata)
        association_map_delete(item_metadata, hash_value)
        freelist_global_lock
        freelist_free(item_metadata)
        freelist_global_unlock
        association_map_insert(new_item_metadata, hash_value)
        item_metadata = new_item_metadata
        item_metadata.get_count = 0
        remove ITEM_DIRTY from item_metadata.flags
        add ITEM_STORED to item_metadata.flags
    else:
        item_metadata.get_count = item_metadata.get_count + 1
item_unlock(hash_value)
return item_metadata.item
// in "Gets" use: return item, item.cas
```

Algorithm 3.7: Memcached Set

```
key, metadata = read from TCP socket
item = item_allocate(key, metadata)
item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and expired:
    association_map_delete(old_item, hash_value)
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_unlock(old_item.slab_id)
    slabs_global_lock
    slabs_free(old_item)
    slabs_global_unlock
if old_item found and not expired:
    association_map_delete(old_item, hash_value)
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_unlock(old_item.slab_id)
association_map_insert(item, hash_value)
lru_lock(item.slab_id)
lru_list_link(item)
lru_unlock(item.slab_id)
item_unlock(hash_value)
```

Algorithm 3.8: LogMemcached Set

```
key, metadata = read from TCP socket
item_metadata = item_allocate(key, metadata)
item_metadata.item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item_metadata = association_map_find(key, hash_value)
if old_item_metadata found and expired:
    association_map_delete(old_item_metadata, hash_value)
    freelist_global_lock
    freelist_free(old_item_metadata)
    freelist_global_unlock
if old_item_metadata found and not expired:
    association_map_delete(old_item_metadata, hash_value)
association_map_insert(item_metadata, hash_value)
remove ITEM_DIRTY from item_metadata.flags
add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.9: Memcached Add

```
key, metadata = read from TCP socket
item = item_allocate(key, metadata)
item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and expired:
    association_map_delete(old_item, hash_value)
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_unlock(old_item.slab_id)
    slabs_global_lock
    slabs_free(old_item)
    slabs_global_unlock
else if old_item found and not expired and time since last updated ≥ 60 seconds:
    //add only adds a nonexistent item, but promote to head of LRU
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_list_link(old_item)
    lru_unlock(old_item.slab_id)
else if old_item not found:
    association_map_insert(item, hash_value)
    lru_lock(item.slab_id)
    lru_list_link(item)
    lru_unlock(item.slab_id)
item_unlock(hash_value)
```

Algorithm 3.10: LogMemcached Add

```
key, metadata = read from TCP socket
item_metadata = item_allocate(key, metadata)
item_metadata.item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item_metadata = association_map_find(key, hash_value)
if old_item_metadata found and expired:
    association_map_delete(old_item_metadata, hash_value)
    freelist_global_lock
    freelist_free(old_item_metadata)
    freelist_global_unlock
if old_item_metadata found and not expired and time since last updated ≥ 60 seconds:
    //add only adds a nonexistent item, but promote to head of LRU
    if LRU mode enabled and old_item_metadata.get_count ≥ lru_mode_get_count:
        new_item_metadata = item_allocate(key, metadata)
        association_map_delete(old_item_metadata, hash_value)
        freelist_global_lock
        freelist_free(old_item_metadata)
        freelist_global_unlock
        association_map_insert(new_item_metadata, hash_value)
        item_metadata = new_item_metadata
        item_metadata.get_count = 0
        remove ITEM_DIRTY from item_metadata.flags
        add ITEM_STORED to item_metadata.flags
    else:
        old_item_metadata.get_count = item_metadata.get_count + 1
else if old_item_metadata not found:
    association_map_insert(item_metadata, hash_value)
remove ITEM_DIRTY from item_metadata.flags
add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.11: Memcached Replace

```
key, metadata = read from TCP socket
item = item_allocate(key, metadata)
item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and not expired:
    association_map_delete(old_item, hash_value)
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_unlock(old_item.slab_id)
    association_map_insert(item, hash_value)
    lru_lock(item.slab_id)
    lru_list_link(item)
    lru_unlock(item.slab_id)
item_unlock(hash_value)
```

Algorithm 3.12: LogMemcached Replace

```
key, metadata = read from TCP socket
item_metadata = item_allocate(key, metadata)
item_metadata.item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item_metadata = association_map_find(key, hash_value)
if old_item_metadata found and not expired:
    association_map_delete(old_item_metadata, hash_value)
    association_map_insert(item_metadata, hash_value)
remove ITEM_DIRTY from item_metadata.flags
add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.13: Memcached Append and Prepend

```
key, metadata = read from TCP socket
item = item_allocate(key, metadata)
item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and not expired:
    new_metadata.size = item.metadata.size + old_item.metadata.size
    new_item = item_allocate(key, new_metadata)
    copy_value(old_item, item, new_item, APPEND/PREPEND)
    association_map_delete(old_item, hash_value)
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_unlock(old_item.slab_id)
    association_map_insert(new_item, hash_value)
    lru_lock(new_item.slab_id)
    lru_list_link(new_item)
    lru_unlock(new_item.slab_id)
item_unlock(hash_value)
```

Algorithm 3.14: LogMemcached Append and Prepend

```
key, metadata = read from TCP socket
item_metadata = item_allocate(key, metadata)
item_metadata.item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item_metadata = association_map_find(key, hash_value)
if old_item_metadata found and not expired:
    new_metadata.size = item_metadata.size + old_item_metadata.size
    new_item_metadata = item_allocate(key, new_metadata)
    copy_value(old_item_metadata, item_metadata, new_item_metadata, APPEND/PREPEND)
    association_map_delete(old_item_metadata, hash_value)
    association_map_insert(new_item_metadata, hash_value)
item_metadata = new_item_metadata
remove ITEM_DIRTY from item_metadata.flags
add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.15: Memcached Incr and Decr

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and not expired:
    old_item.value = old_item.value + metadata.delta
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_list_link(old_item)
    lru_unlock(old_item.slab_id)
item_unlock(hash_value)
```

Algorithm 3.16: LogMemcached Incr and Decr

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_metadata = association_map_find(key, hash_value)
if old_metadata found and not expired:
    item_metadata = item_allocate(key, metadata)
    item_metadata.item.value = old_metadata.item.value + metadata.delta
    association_map_delete(old_item_metadata, hash_value)
    association_map_insert(item_metadata, hash_value)
    remove ITEM_DIRTY from item_metadata.flags
    add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.17: Memcached Cas

```
key, metadata = read from TCP socket
item = item_allocate(key, metadata)
item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and not expired and
 old_item.cas_value = item.cas_value:
    association_map_delete(old_item, hash_value)
    lru_lock(old_item.slab_id)
    lru_list_unlink(old_item)
    lru_unlock(old_item.slab_id)
    association_map_insert(new_item, hash_value)
    lru_lock(new_item.slab_id)
    lru_list_link(new_item)
    lru_unlock(new_item.slab_id)
item_unlock(hash_value)
```

Algorithm 3.18: LogMemcached Cas

```
key, metadata = read from TCP socket
item_metadata = item_allocate(key, metadata)
item_metadata.item.value = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item_metadata = association_map_find(key, hash_value)
if old_item_metadata found and not expired and
 old_item_metadata.cas_value = itemitem_metadatacas_value:
    association_map_delete(old_item_metadata, hash_value)
    association_map_insert(item_metadata, hash_value)
remove ITEM_DIRTY from item_metadata.flags
add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.19: Memcached Touch

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item = association_map_find(key, hash_value)
if old_item found and not expired:
    old_item.expire_time = metadata.expire_time
item_unlock(hash_value)
```

Algorithm 3.20: LogMemcached Touch

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
old_item_metadata = association_map_find(key, hash_value)
if old_item_metadata found and not expired:
    item_metadata = item_allocate(key, metadata)
    item_metadata.item.value = old_item_metadata.item.value
    item_metadata.expire_time = metadata.expire_time
    association_map_delete(old_item_metadata, hash_value)
    association_map_insert(item_metadata, hash_value)
    remove ITEM_DIRTY from item_metadata.flags
    add ITEM_STORED to item_metadata.flags
item_unlock(hash_value)
```

Algorithm 3.21: Memcached Delete

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
item = association_map_find(key, hash_value)
if item found:
    association_map_delete(item, hash_value)
    lru_lock(item.slab_id)
    lru_list_unlink(item)
    lru_unlock(item.slab_id)
    slabs_global_lock
    slabs_free(item)
    slabs_global_unlock
item_unlock(hash_value)
```

Algorithm 3.22: LogMemcached Delete

```
key, metadata = read from TCP socket
hash_value = hash(key)
item_lock(hash_value)
item_metadata = association_map_find(key, hash_value)
if item_metadata found:
    add ITEM_DELETED to metadata.flags
    new_item_metadata = item_allocate(key, metadata)
    association_map_delete(item_metadata, hash_value)
    slabs_global_lock
    slabs_free(item_metadata)
    slabs_global_unlock
    remove ITEM_DIRTY from new_item_metadata.flags
    add ITEM_STORED to new_item_metadata.flags
item_unlock(hash_value)
```

Chapter 4

Evaluation

4.1 Environment

To evaluate LogMemcached we used Netgear's ProSafe XS708E Switch, Intel's Ethernet Controller 10-Gigabit X540-AT2 and Mellanox's ConnectX®-3 Pro for RoCE communication. Each server used eight Intel's Core i7-4790 CPU at 3.60 GHz processors, two Conexant's (Rockwell) DIMM DDR3 Synchronous 1600 MHz System's Memory, Intel's 8 Series/C220 Series Chipset Family PCI Express PCI bridge connected to X540-AT2 and Intel's Xeon E3-1200 v3/4th Gen Core Processor PCI Express x16 Controller connected to ConnectX®-3. The servers ran under Ubuntu 14.10, kernel 3.16.0-43 with OFED version 3.18, Memcached version 1.4.24 and libMemcached version 1.0.18.

We strove to evaluate the following four benchmarks: 1) throughput in LogMemcached vs. Memcached, 2) server-side replication throughput vs. client-side replication throughput, 3) LRU eviction affect on throughput and 4) the impact of replication on the master machine's CPU load.

We simulated Memcached's client using the Memaslap benchmarking tool, based on libMemcached [2]. Tests that used concurrency on the client side were configured with 8 concurrent connections performed by 4 threads. Memcached was modified to support 1GB sized pages,

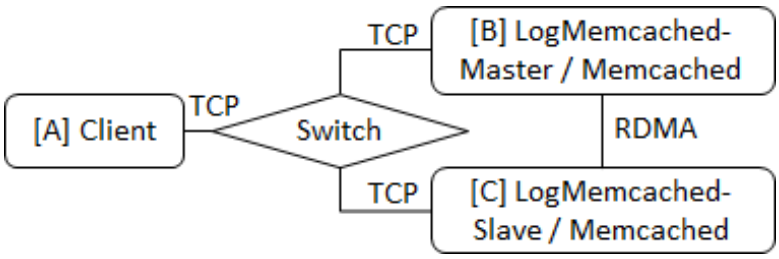

Figure 4.1: Evaluation setup.

	32B	64B	128B	256B	512B	1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB
1:	0	0	0	0	0	0	0	0	0.02	0	0.02	-0.13	-0.06	0	-0.02	-0.01
2:	0.1	-0.12	0.03	-0.11	0	-0.04	0.01	0	-0.01	0.01	0.01	0.02	0	-	-	-
3:	0.01	-0.01	0.03	-0.01	0	0	0.15	-0.03	-0.12	0	0.03	-0.02	-0.02	-0.06	-0.08	-0.08
4:	-0.05	-0.02	-0.04	-0.01	-0.02	-0.01	-0.04	0	0.03	0.01	-0.1	-0.11	-0.1	-0.1	-0.08	-0.07
5:	-0.09	-0.07	-0.11	-0.14	-0.15	-0.3	-0.15	-0.18	-0.17	-0.18	-0.17	-0.02	-0.26	-0.27	-0.35	-0.4
6:	-0.09	-0.09	-0.11	-0.15	-0.17	-0.29	-0.1	-0.19	-0.17	-0.55	-0.72	-0.77	-0.87	-0.9	-0.89	-0.9
7:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8:	1	1	1	1	1	1	1	1	1	1	1.06	1.24	1.33	1.49	1.89	1.3

Table 4.1: Results for tests (1)-(6), showing ΔQ, ΔQP_M and ΔQP_L as appropriate. The results of tests (7) and (8) show the ratio between the throughput of the Set operation vs. the throughput of the replication operation. Test (2) was limited to items sized up to 256KB because of Memaslap limitations.

rather than using a hard-coded 1MB limitation, so as to effectively evaluate the behavior of large items. Memaslap ran on machine A, while LogMemcached and Memcached ran on machines B and C, as described in Figure 4.1.

4.2 Test Cases

The following tests were performed:

- **Get throughput (ops/s) without and with concurrency, tests (1) and (2), respectively:** The client first set the maximal number of items possible without causing eviction on Memcached and on LogMemcached separately. The client then performed Get operations on these items. No misses occurred. See Table 4.1 and Figure 4.2.

- **Set throughput (ops/s) without and with concurrency, tests (3) and (4), respectively:** The client performed Set operations on Memcached and on LogMemcached. See Table 4.1 and Figure 4.3.

- **Client side replication throughput (ops/s) without concurrency:** The client performed Set operations on a software proxy. The proxy replicated the requests to two Memcached instances (test (5)) and to two LogMemcached instances (test (6)), and returned an answer to the client after receiving answers from both Memcached/LogMemcached servers. To neutralize the proxy's effect on the throughput, we compare the throughputs in this case to the throughput of Set operations without replication, performed on the same proxy. See Table 4.1 and Figure 4.4.

- **Server side replication throughput (ops/s) without and with concurrency, tests (7) and (8), respectively:** The client performed Set operations on LogMemcached Master, and a LogMemcached slave replicated the stored items. We then compare the throughput of the Set operation with the throughput of the replication operation. See Table 4.1 and Figure 4.4.

- **LRU eviction affect on throughput with a client Get/Set ratio of 9:1:** The client performed Get and Set operation, using 9:1 ratio (reflective of real life workloads [10]). The Client attempted to get the last set items within a predefined window (the window size is the number of last set items by a single client). We performed four different tests: 1) Without concurrency with 1K window (9), 2) Without concurrency with 10K window (10), 3) With concurrency with 1K window (11), and 4) With concurrency with 10K window (12). See Figures 4.5 and 4.6

- **Time spent in running user processes (without and with concurrency) and kernel processes (without and with concurrency), tests (13), (14), (15) and (16), respectively:** The client performs only Set operations on four different servers: 1) A Memcached server, 2) A LogMemcached master with no replication, 3) A LogMemcached replication master, and 4) A LogMemcached replication slave. We show the percentage of time spent by the CPU on these operations relative to the overall time spent by the

CPU. See Figures 4.7 and 4.8.

4.3 Performance Analysis

Denote the number of served queries by LogMemcached and Memcached as L_Q and M_Q respectively. We define $\Delta Q = \frac{L_Q - M_Q}{M_Q}$ to represent the overhead of LogMemcached relative to Memcached. We use ΔQ in tests (1)-(4) above. Positive ΔQ values indicate an advantage of LogMemcached over Memcached, while negative values indicate the opposite. For tests (5) and (6) we denote M_{RQ} and L_{RQ} to be the number of queries served by Memcached and LogMemcached respectively with client side replication, and define $\Delta QP_M = \frac{M_{RQ} - M_Q}{M_Q}$ and $\Delta QP_L = \frac{L_{RQ} - L_Q}{L_Q}$.

The results of tests (1)-(4) indicate that LogMemcached's performance is comparable to that of Memcached for both Get and Set operations. The results were consistent and repetitive, thus we assume that the oscillation around zero is a result of the difference between memory allocations in LogMemcached and Memcached.

Results for tests (5)-(6) show a noticeable advantage to LogMemcached owing to the high cost of client side replication.

The results of tests (7) and (8) show that for small items LogMemcached's replication is fast enough to catch up with the Set operations, but a backlog is accumulated for large items and under moderate load. Since Memcached is designed mainly for small item sizes, which dominate real life loads [10], the slowdown for larger items is not significant. Moreover, since real workloads operate mainly under Get-intensive loads [10], in real-life scenarios the replication mechanism will have time to catch up with the pace of the Set mechanism. Additional solution is to use a scatter-gather mechanism, which is natively supported by RDMA libraries, that allows replication of multiple items via a single replication.

The results of (9)-(11) evaluations showed that although LogMemcached's LRU implementation is comparable to Memcached's, LogMemcached nevertheless shows an increase in Get Misses

for large items, relative to Memcached. This occurs because LogMemcached evicts batches of items from the memory, and not single items, as Memcached does. In large windows, the client attempts to reach items that were already evicted. The gain in Get operation performance appears because it's faster to serve a Miss response than to return an item.

The results of tests (13) and (15) show that without concurrency LogMemcached master used on average (averaged over all item sizes) 0.78% of the CPU without replication, and 13.22% with replication. With concurrency it used on average 2.8% of the CPU without replication and 15.58% with replication. In both cases the replication increased the CPU use by about 12.5%. The results of tests (14) and (16) show that, on average, the replication does not require any additional CPU at kernel mode.

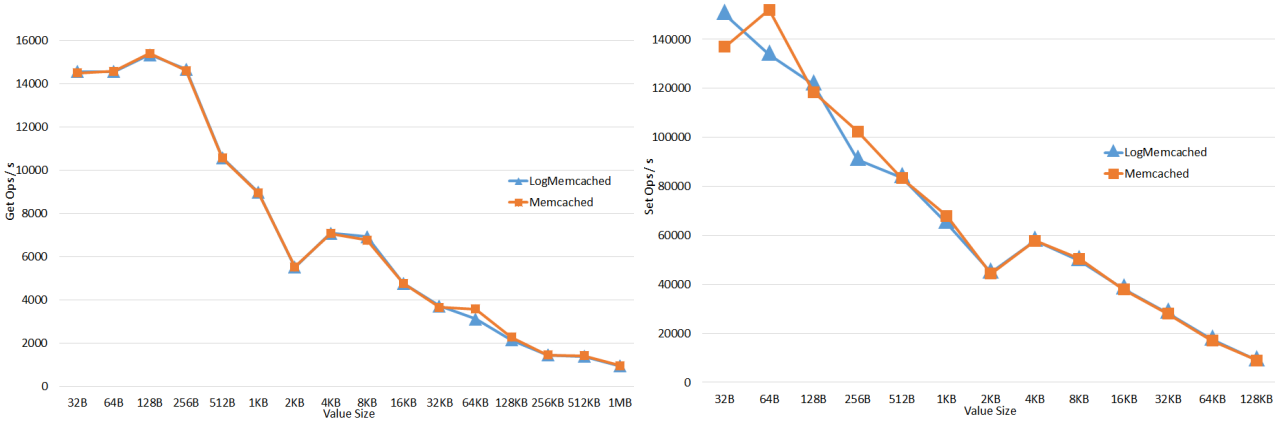

Figure 4.2: From left to right: results for tests (1) and (2).

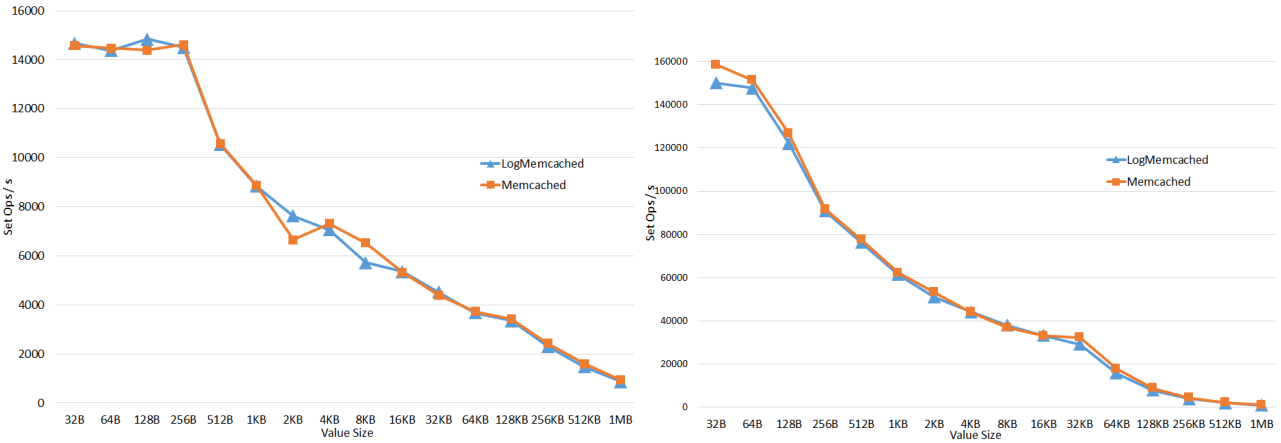

Figure 4.3: From left to right: results for tests (3) and (4).

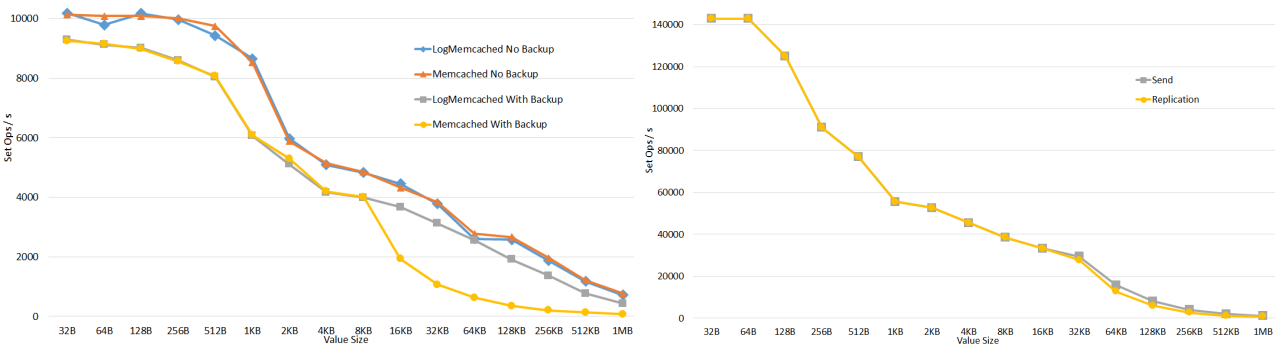

Figure 4.4: From left to right: results for tests (5), (6) (left graph) and (8).

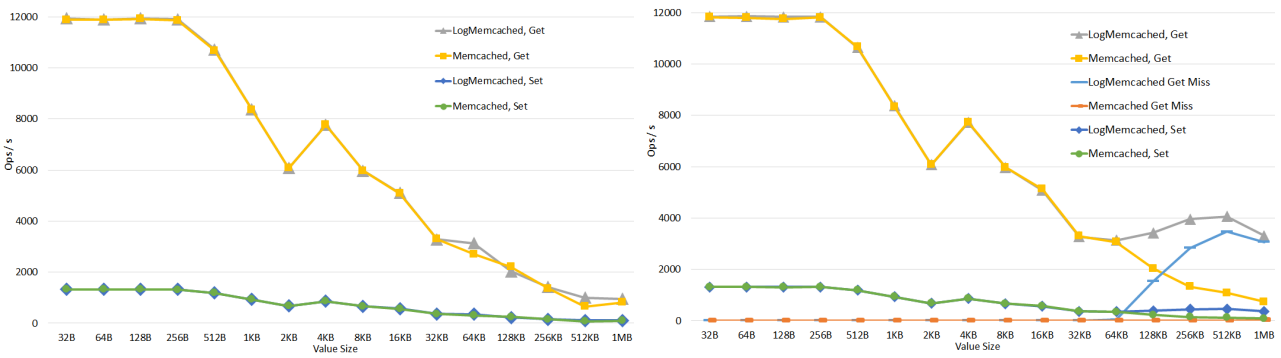

Figure 4.5: From left to right: results for tests (9) and (10). Test (9) does not presents the miss rate, since it was 0 for all item sizes smaller than 1MB. LogMemcached had missed 38% of all the Get requests for 1MB items while Memcached had 0 misses for 1MB items.

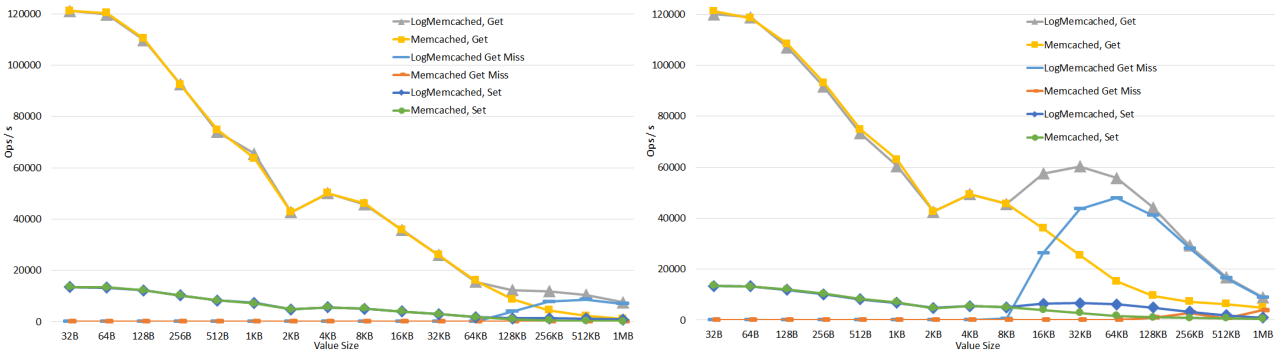

Figure 4.6: From left to right: results for tests (11) and (12).

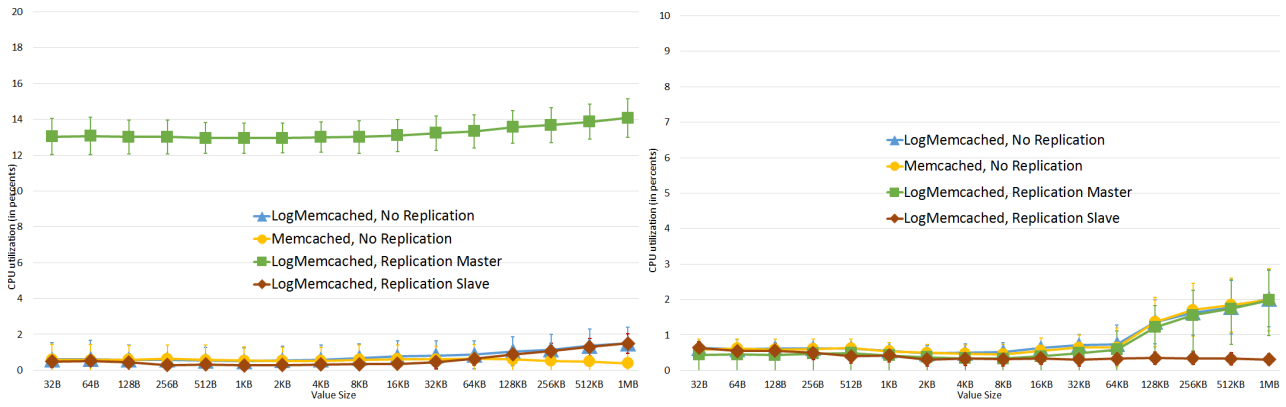

Figure 4.7: From left to right: results for tests (13) and (15), time spent running user processes and time spent running kernel processes. The tests were performed *without concurrency*, and presented in percentages of CPU utilization of the entire machine. Notice that the difference in vertical scales.

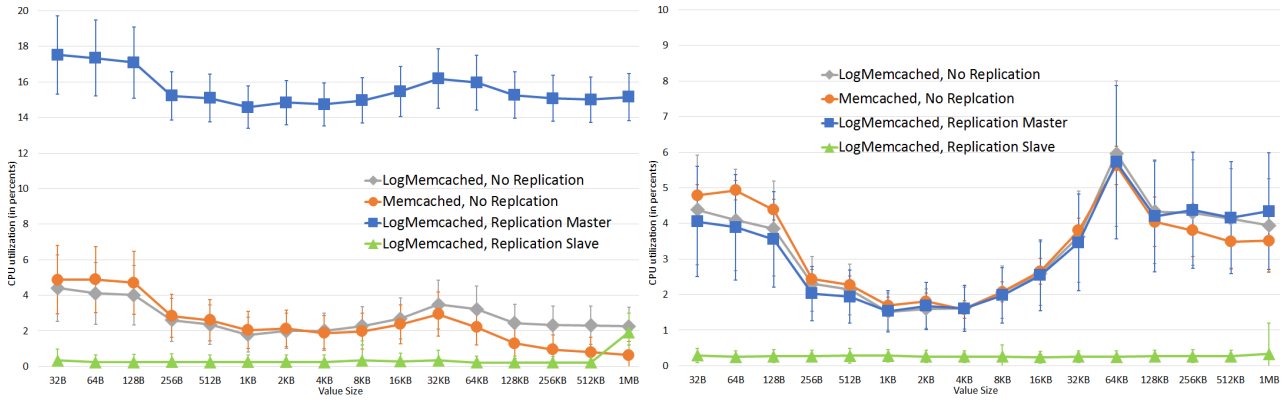

Figure 4.8: From left to right: results for tests (14) and (16), time spent running user processes and time spent running kernel processes. The tests were performed *with concurrency*, and presented in percentages of CPU utilization of the entire machine. Notice that the difference in vertical scales.

Chapter 5

Related Work

Enhancing Memcached by reducing its latency appears in several works. Jose *et al.* [17, 18] extends the existing open-source Memcached software and makes it RDMA capable - Both in libmemcached (client) and Memcached (server). In the first work they use RDMA USR (Unified Communication Runtime) RC (Reliable Connection) over Infiniband. The main claim is that since sockets offers byte-stream oriented semantics, using them causes a conversion between Memcached's memory-object semantics and Socket's byte-stream semantics, imposing an overhead. This is in addition to any extra memory copies in the Sockets implementation within the OS. This overhead can be reduced by using RDMA, because, in addition to its high raw performance, its memory based semantics fits well with Memcached's memory-object model. The communication model use RDMA Queue Pair (SEND/RECV), rather than native RDMA verbs as READ and WRITE. In their following work, Jose *et al* modifies the communication model, since it has been shown that RC transport imposes scalability issues due to high memory consumption per connection (standard configuration parameters requires around 64 KB of memory per connection). Such a characteristic is not favorable for middlewares like Memcached, where the server is required to serve thousands of clients. The Unreliable Datagram (UD) transport offers higher scalability, but introduce other challenges, such as how to efficiently handle the reliability, flow control and large message communication latency problems associated with InfiniBand's UD transport. These issues are addressed in the paper. The

work introduces a hybrid transport model which uses both RC and UD to deliver scalability and performance higher than that of a single transport.

Stuedi *et al* target clusters built from low-power CPUs, and propose a modified Memcached architecture (named "Memcached/RDMA") to leverage the one-side semantics of RDMA [29]. With those modifications in place, Memcached/RDMA uses fewer CPU cycles than the unmodified Memcached due to copy avoidance, less context switching and removal of serverside request parsing. As a result, this allows for 20% more operations to be handled by Memcached per second. While RDMA is a network technology typically associated with specialized hardware, the proposed solution uses soft-RDMA which runs on standard Ethernet and does not require special hardware. The main idea in Memcached/RDMA is to allow chunks to be read remotely without involving the Memcached user-level process. For this, Memcached/RDMA registers every newly allocated chunk with the soft-RDMA provider. The SET operation transmits, as part of the response message, the stag stored inside the chosen chunk. Clients maintain a table (stag table) matching keys with stags for a later use. Before issuing a GET operation, the client checks whether an entry for the given key exists in the stag table. If no entry is found, the GET operation is initiated using the TCP-based protocol. The server, however, rather than responding on the reverse channel of the TCP connection, will use a one-sided RDMA WRITE operation to transmit the requested key/value pair to the client using additional RDMA information that was inserted into the client request. Besides the key/value pair, the respond message from the server also includes the stag associated with the chunk storing the item at the server; this stag is inserted into the local stag table by the client. If a stag entry is found at the client before the GET request is issued, Memcached/RDMA will use a one-sided RDMA READ operation to directly read from the server the chunk storing the requested key/value pair.

Many works, such as Pilaf [23], HERD [19] and FaRM [14] propose designing new key-value stores, so as to incorporate RDMA in their fundamental design.

In Pilaf [23], clients directly read from the server's memory via RDMA to perform gets, which commonly dominate key-value store workloads. By contrast, put operations are serviced by the

server to simplify the task of synchronizing memory accesses. In the basic design, a client looks up a key in the hash table array using linear probing. Each probe involves two RDMA reads. The first read fetches the hash table entry corresponding to the key. If the entry is currently filled, the client initiates a second RDMA read to fetch the actual key and value strings from the extents region according to the address information stored in the corresponding hash table entry. The client checks whether the fetched key string matches the requested key. The Pilaf server handles all put operations. In a more advanced design, to achieve good memory efficiency with fewer probes than with linear probing (when the hash table is 60% full, the maximum number of probes required can be as high as 70), Pilaf uses 3-way Cuckoo hashing (at a fill ratio of 75%, the average and maximum number of probes in 3-way Cuckoo hashing is 1.6 and 3, compared to 2.5 and 213 respectively for linear probing).

HERD [19] focuses its design on reducing network round trips and uses a single round trip while taking two unconventional decisions: First, it does not use RDMA reads, despite the allure of operations that bypass the remote CPU entirely. Second, it uses a mix of RDMA and messaging verbs, despite that the messaging primitives are considered slow. Because of lack of richness of RDMA operations - an RDMA operation can only read or write a remote memory location - it is not possible to do more sophisticated operations such as dereferencing and following a pointer in remote memory, other works has focused on using RDMA reads to traverse remote data structures, similar to what would have been done had the structure been in local memory. This approach requires multiple round trips across the network. HERD implement a key-value cache, in which clients transmit their request to the server's memory using RDMA writes. The server's CPU polls its memory for incoming requests. On receiving a new request, it executes the GET or PUT operation in its local data structures and sends the response back to the client. As RDMA write performance does not scale with the number of outbound connections, the response is sent as a SEND message over a datagram connection. For small key-value items, HERD's throughput is similar to native RDMA read throughput.

FaRM [14] exposes the memory of machines in the cluster as a shared address space. Applications can use transactions to allocate, read, write, and free objects in the address space with location transparency. FaRM uses one-sided RDMA reads to access data directly and it uses

RDMA writes to implement a fast message passing primitive. This primitive uses a circular buffer to implement a unidirectional channel. The buffer is stored on the receiver, and there is one buffer for each sender/receiver pair. The receiver periodically polls the word at the "Head" position to detect new messages. The sender uses RDMA to write messages to the buffer tail and it advances the tail pointer on every send. The receiver makes processed space available to the sender lazily by writing the current value of the head to the sender's copy using RDMA. FaRM's shared address space consists of many shared memory regions. To access an object, FaRM uses a form of consistent hashing to map the region identifier to the machine that stores the object. If the region is stored locally, FaRM obtains the base address for the region and uses local memory accesses. Otherwise, FaRM contacts the remote machine to obtain a capability for the region, and then uses the capability, the offset in the address and the object size to build an RDMA request. Transactions use optimistic concurrency control with an optimized two-phase commit protocol that uses RDMA.

MICA [21] also builds a new key-value store, rather than relying on Memcached's, and also uses an append-only log similar to ours as one of its core data structures, but does not support build in replication. MICA implements an unconventional choices in aspects of request handling, including parallel data access, network request handling, and data structure design. Concurrent access is used by most key-value systems, where multiple CPU cores can access the shared data. The integrity of the data structure must be maintained using mutexes, optimistic locking, or lock-free data structures. MICA uses Exclusive access, By partitioning the data ("sharding"), each core exclusively accesses its own partition in parallel without inter-core communication. MICA can fall back to concurrent reads if the load is extremely skewed, but avoids concurrent writes. Regarding network, MICA uses direct NIC access. By targeting only small key-value items, it needs fewer transport layer features. Clients are responsible for retransmitting packets if needed. MICA uses Intel's DPDK (Data Plane Development Kit) instead of standard socket I/O. This allows the user-level server software to control NICs and transfer packet data with low overhead. MICA uses NIC multi-queue support to allocate a dedicated RX and TX queue to each core. Cores exclusively access their own queues without synchronization. MICA avoids packet data copy throughout RX/TX and request processing. Upon receiving a request, MICA

avoids memory allocation and copying by reusing the request packet to construct a response. In cache mode it uses circular logs to manage memory for key-value items and lossy concurrent hash indexes to index the stored items. Each MICA partition consists of a single circular log and lossy concurrent hash index.

Several other works propose a modification of Memcached's internal data structures and enhancing of specific characteristics [15, 16], the use of proxies [24, 27, 28], modification of the underlying network [31, 35, 36], specialized OSes [11, 25] or FPGA [12, 22] to boost Memcached's (and other caches') performance.

The work on mcrouter by Facebook [24] describes how Facebook leverages memcached as a building block to construct and scale a distributed key-value store that supports the world's largest social network. According the the paper, a social network's infrastructure needs to allow near realtime communication, aggregate content on-the-fly from multiple sources, be able to access and update very popular shared content, and scale to process millions of user requests per second. Facebook relies on memcache to lighten the read load on the databases. The client logic is provided as two components: a library that can be embedded into applications or as a standalone proxy named mcrouter. This proxy presents a memcached server interface and routes the requests/replies to/from other servers. mcrouter is a core component of the cache infrastructure at Facebook, and provides multiple functionalities, as connection pooling, replicated pools, cold cache warm up, prefix routing and more.

Within some pools, Facebook use replication to improve the latency and efficiency of memcached servers. The replication solves the problem provided in the following example: a memcached server holds 100 items and capable of responding to 500k requests per second. Each request asks for 100 keys. The difference in memcached overhead for retrieving 100 keys per request instead of 1 key is small. To scale the system to process 1M requests/sec, supposedly they add a second server and split the key space equally between the two. Clients now need to split each request for 100 keys into two parallel requests for $\sim$50 keys. Consequently, both servers still have to process 1M requests per second. However, if they replicate all 100 keys to multiple servers, a client's request for 100 keys can be sent to any replica. This reduces the load per

server to 500k requests per second. Each client chooses replicas based on its own IP address. This approach requires delivering invalidations to all replicas to maintain consistency.

When they bring a new cluster online, an existing one fails, or perform scheduled maintenance the caches will have very poor hit rates diminishing the ability to insulate backend services. A system called Cold Cluster Warmup mitigates this by allowing clients in the "cold cluster" (i.e. the frontend cluster that has an empty cache) to retrieve data from the "warm cluster" (i.e. a cluster that has caches with normal hit rates) rather than the persistent storage. This takes advantage of the aforementioned data replication that happens across frontend clusters. With this system cold clusters can be brought back to full capacity in a few hours instead of a few days.

Replication is normally performed via proxies or client-side libraries, as we see with mcrouter by Facebook [24], NetKV [33], Zhang *et al.* [34], or libMemcached [2]. Repcached [8] modified Memcached to support message-oriented server-side replication without RDMA; Cocytus [32] modified Memcached to support replication and applied erasure coding to achieve memory efficiency. Other server-side replication solutions, such as MemcacheDB [4] and Couchbase [1], choose to support Memcached's protocol as part of a persistent store, which enables a wide range of database techniques for replication, but in doing so can no longer be considered cache stores.

DARE [26] implements consistent RDMA-based replication of Replicated State Machines (RSMs), replicating the state indirectly via the message log. Since large-scale datacenter architectures are more susceptible to faults of single components, DARE offer consistent services on such unreliable systems to be a replicated state machines (RSMs). The work on DARE propose a new set of protocols, Direct Access REplication, based on RDMA primitives, and evaluate them with a strongly consistent key-value store; They show that RDMA introduces various new options, such as log access management. In total, the work on DARE provide a complete RDMA RSM protocol, an RDMA performance model of DARE in a failure-free scenario, a failure model for RDMA systems in which they analyze both the availability and reliability of DARE, and a demonstration of how DARE can be used to implement a strongly-consistent

key-value store.

Redis [7] - one of the most popular Memcached competitors in key-value cache systems - allows message-oriented server-side replication without the use of RDMA. Redis uses asynchronous replication. However, slaves periodically acknowledge the amount of data processed from the replication stream. Aside from connecting a number of slaves to the same master, slaves can also be connected to other slaves in a cascading-like structure. Redis replication is non-blocking on the master side. This means that the master will continue to handle queries when one or more slaves perform the initial synchronization. Replication is also non-blocking on the slave side. While the slave is performing the initial synchronization, it can handle queries using the old version of the dataset. However, after the initial sync, the old dataset must be deleted and the new one must be loaded. The slave will block incoming connections during this window (that can be as long as many seconds for very large datasets). Slaves support a read-only mode, in which slaves will reject all write commands, so that it is not possible to write to a slave because of a mistake. It is possible to configure a Redis master to accept write queries only if at least N slaves are currently connected to the master. However, because Redis uses asynchronous replication it is not possible to ensure the slave actually received a given write, so there is always a window for data loss. Redis slaves ping the master every second, acknowledging the amount of replication stream processed. Redis masters will remember the last time it received a ping from every slave. The user can configure a minimum number of slaves that have a lag not greater than a maximum number of seconds, If there are at least N slaves, with a lag less than M seconds, then the write will be accepted.

Chapter 6

Conclusion

As caches become integral components of web services, system designers can no longer ignore cache node failures as the overhead of warming the cache can greatly impact the service. High-availability and fault tolerance of caches have thus become a key design consideration for any caching system. Our work provides an innovative solution that implements these using continuous state replication via RDMA. The replication frequency can be easily adjusted to adapt to load and reliability requirements. Our system extends and uses the same client API as Memcached and thus can be transparently plugged into existing cloud services. Future work may explore ways to optimize log management via log compaction algorithms and splitting the log into buckets for different sized items; Provide customized consistency requirements via a modified replication scheme, and to evaluate scalability via the RDMA READ and WRITE verbs.

Bibliography

[1] Couchbase: Nosql database. https://www.couchbase.com/.

[2] libmemcached. http://libmemcached.org/.

[3] Memcached. https://github.com/memcached/memcached/wiki/Overview.

[4] Memcachedb. http://memcachedb.org/.

[5] Project voldemort. http://www.project-voldemort.com.

[6] reddit's may 2010 "state of the servers" report. https://redditblog.com/2010/05/11/reddits-may-2010-state-of-the-servers-report/.

[7] Redis. https://redis.io/.

[8] Repached. http://repcached.lab.klab.org.

[9] Logmemcacged. 2017. https://github.com/HUJI-DANSS/LogMemcached.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, CO, 2014. USENIX Association.

[12] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA, 2013. USENIX.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[15] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[16] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, 2015. USENIX Association.

[17] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 236–243, Washington, DC, USA, 2012. IEEE Computer Society.

[18] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[20] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[21] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[22] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[23] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, 2013. USENIX.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook.

In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[25] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, CO, 2014. USENIX Association.

[26] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, New York, NY, USA, 2015. ACM.

[27] Manju Rajashekhar. Twemproxy: A fast, light-weight proxy for memcached. 2012. https://blog.twitter.com/2012/twemproxy.

[28] Manju Rajashekhar and Yao Yue. Caching with twemcache. 2012. https://blog.twitter.com/2012/caching-with-twemcache.

[29] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 347–353, Boston, MA, 2012. USENIX.

[30] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[31] A. F. R. Trajano and M. P. Fernandez. Two-phase load balancing of in-memory key-value storages through nfv and sdn. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 409–414, July 2015.

[32] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and available in-memory kv-store with hybrid erasure coding and replication. In *FAST*, pages 167–180, 2016.

[33] W. Zhang, T. Wood, and J. Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 5–14, July 2016.

[34] Wei Zhang, Jinho Hwang, Timothy Wood, K.K. Ramakrishnan, and Howie Huang. Load balancing of heterogeneous workloads in memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*, Philadelphia, PA, 2014. USENIX Association.

[35] Wei Zhang, Guyue Liu, Ali Mohammadkhan, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Sdnfv: Flexible and dynamic software defined control of an application- and flow-aware data plane. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 2:1–2:12, New York, NY, USA, 2016. ACM.

[36] Wei Zhang, Timothy Wood, K.K. Ramakrishnan, and Jinho Hwang. Smartswitch: Blurring the line between network infrastructure & cloud applications. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, 2014. USENIX Association.

[37] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 59–72, San Jose, CA, 2013. USENIX.