CellCycle

A proposal for Elastic Distributed Shared Memory Application

Alessandro Fazio

+39 3339850025
fazio.alessandro@hotmail.com

Andrea Gennusa

+39 3475240799
andrea.gennusa@gmail.com

ABSTRACT

From the beginning of Computer Science the cache was the simplest and fastest way to speedup a system that use permanent data.
Today, most of services are going to web and cloud architecture, and cache needs to be updated to new way of use resources.
The most famous tool to do it is Memcached [2], written in C, it's a simple Client-Server application that stores Key-Value elements in RAM and exposes a TCP interface to get and set it. It was improved by the use of a hash function, on client side, to assign a specific server for every key from a pool of running applications.
Redis [3] is another famous tool, born from a developer hired from VMWare, often used as Key-Value persistent database, useful for in memory caching too.
We took inspiration from these applications to go beyond creating a Memcached-Like service to use at best the new cloud computation services.
Idea was to create something reliable and elastic, native of Cloud Computation Services, that uses a structured communication model, that offers a multi entrypoint, adapting to loading level and providing a Memory manager and a LRU replication policy that use at best memory capacity.

1. Target

- Memory - System is able to store size limited KeyValue objects, in volatile memory RAM. Memory capacity of each machine has to be used at the best, replacement of old values has to be done only when memory exceed the actual capacity. Idea is to convert the largest part possible of available ram in storage space for key values elements, and to keep them for the longest period.

- Distributed - System is composed by different "machines" called nodes or cells, where software is executing. Nodes work together and communicate to exchange service informations and data. Each node is an entrypoint of the service, internal complexity has to be hided to end user.

- Shared - All machines have their own ram. Memory available to all machines is used to offer memory storage service. Physical memory separation must be hided to end user.

- Elastic - System is able to scale up and scale down, using the numbers of active machines. Application must have a metric tool to measure actual load of memory operations. If the machine is underloaded, application have to schedule a scale down element, and shutdown a virtual machine, on the other case a machine must be created and integrated.

2. Needs

Needs are elicited from pdf presentation of April 16 2016 and from specific project specification of August 25 2016.

2.1 Not functional needs

1. You can choose the programming language
2. You can use support libraries and tools to develop your project
3. System/service with configurable parameters (no hard-coded!)– Through a configuration file/service
4. You must test all the functionalities of your developed system/service and present and discuss the testing results in the project report
5. System/service supports multiple, autonomous entities contending for shared resources
6. System/service supports real-time updates to some form of shared state
7. System/service state should be distributed across multiple client or server nodes
8. – The only allowed centralized service can be one that supports users logging on, adding or removing clients or servers, and other housekeeping tasks
9. System/service scalability and elasticity

10. System/service fault tolerance, in particular system/ service continues operation even if one of the participant nodes crashes (optionally, recovers the state of a crashed node so that it can resume operation)

2.2 Functional needs

1. Realize a distributed shared memory system that supports application scale-up and Scale-down
2. Design a totally elastic solution: add new nodes and remove existing nodes without restarting other nodes in the system
3. Scale-down can be challenging due to need to perform state reintegration
4. Handle node failover
5. ~~Support at least two consistency models~~ (deleted cause of new functional need in opposition: 13)
6. Some examples: memcached, Hazelcast

2.3 New Not Functional needs (Requirements of 25-08-2016)

11. Test application with various workload: only read operations workload, read and write operations workload. Tools like memaslap could be used.
12. Test main use cases and document them in relation.

2.4 New Functional needs (Requirements of 25-08-2016)

7. System must support operations: add a value, update a value, get a value, delete a value. Values are size limited. Every value has an unique key, users can operate on values using the appropriate key.
8. An API specification must be provided.
9. System must support at least one cache replacement policy, to avoid the fill up of RAM.
10. System must support at least one balancement policy to organize cache servers of the distributed application.
11. System must organize dynamically server instances during application execution. In particular, system must allocate and deallocate VM instances, providing a scale out and scale in policy, in a Cloud Environment as Amazon Web Services.
12. System must ensure reliability, through replication policy of stored objects among servers. A specification of supported failures must be provided.
13. System must support one consistency model. Choice of design must be discussed in relation.

3. CellCycle - Overview

Nodes of the system are structured as a circle, a cycle. Every element is linked before and after with another node. Nodes are numbered clockwise through float ids.

Each node splits his ram to manage his data (he's the Master of this data) and backup data of previous node (he's the Slave of this data). So, each node is a Master, while it's the slave of the previous node.

Every node is an entry point for the memory provider service.

On the side of circle, there are management messages. Cycle Operations maintain knowledge of nodes updated, communicating death or birth of nodes, and to challenge each other to lock parents.

On the other hand, requests of clients reach a node, that has to deliver real memory request to correct node, sending a "secant" message on another node. Exchange of request and response are Secant Operations.

The real memory management is done inside Memory Module.

Application is organized in modules, communicating each other mainly through network to guarantee independency, reliability and interchangeability.

4. Cycle Operations - Chain Module

4.1 Name Space of keys and nodes

To split name space of the keys, every node chooses a contiguous set of keys. Every node continues the name space set of the previous one.

Example:
Name space is an integer between 0 and 127.

- Node 1 has [0,31] set as Memory Master, and [96, 127] as Slave Memory.
- Node 2 has [32, 63] set as Memory Master, and [0,31] as Slave Memory.
- Node 3 has [64, 95] set as Memory Master, and [32, 63] as Slave Memory.
- Node 4 has [96, 127] set as Memory Master, and [64, 95] as Slave Memory.

4.2 Adding a node (cell duplication)

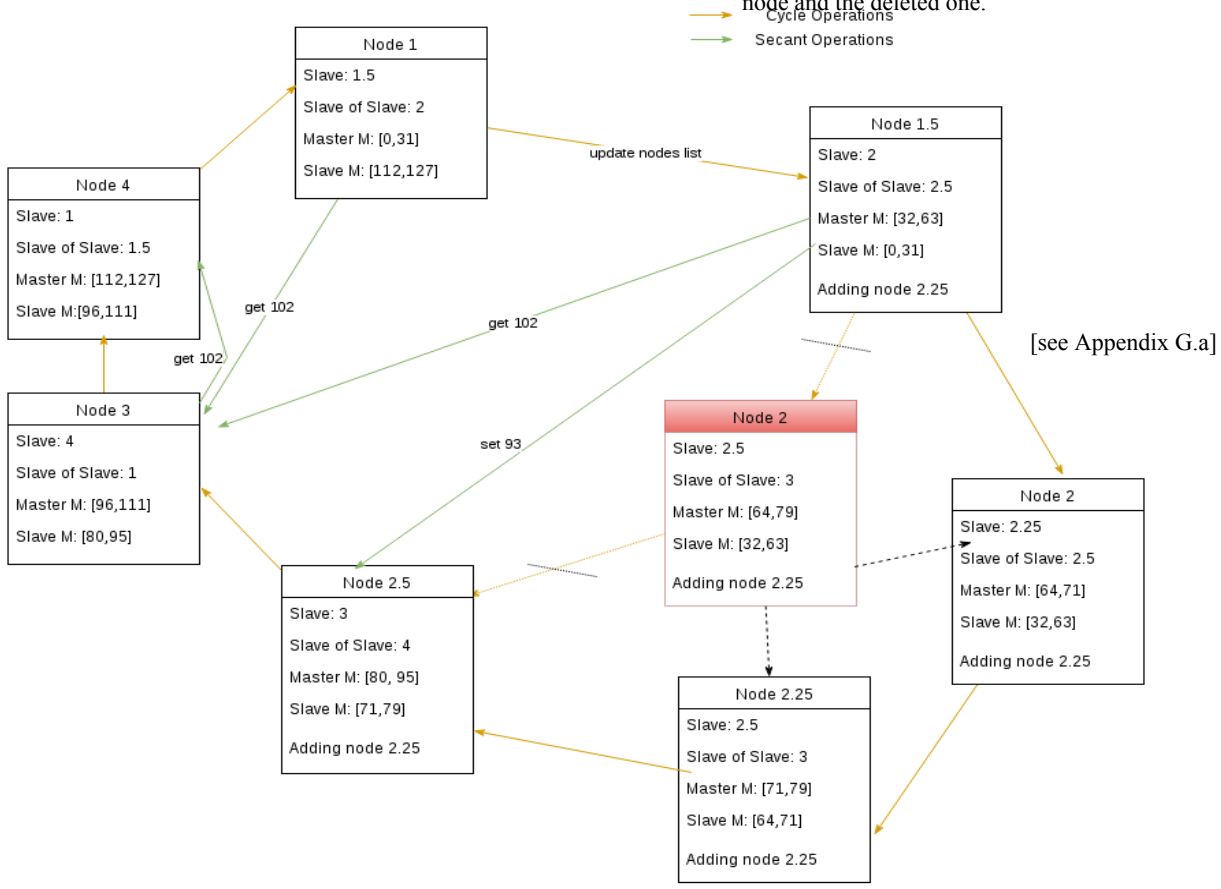
When a node runs off capacity, he can create a new Slave after him.
The name of the new node is a float number, between the name of the full node and the next one. When the new node boots up, Master memory of the first node is transferred to new one.
First node drops the second half of his Master keys, transferring to new one.
The next node now it's the slave of the new node, not the first anymore. So he has to drop first half of his Slave keys.
The new node takes Master keys from the first node and splits it in his Master keys (first half) and Slave keys (second half).
Responsibilities of the full node is now been splitted into two nodes, like a cell division.
Every node adding operation involves only 3 node: the full node, the next one and the new node.
The computational cost of the adding operation is the transferring of Master keys of full node (we can accept this cost, we assume that system runs on a high speed intranet).

Example (node 2 is running off capacity, it creates node 3):

id	Master Memory	Slave Memory
1	[0,63]	[64,128]-->[96,127]
2	[64,127]-->[64,95]	[0,63]
3	[96,127]	[64,95]

Example (node 2 is running off capacity again, it creates node 2.5):

id	Master Memory	Slave Memory
1	[0,63]	[96,127]
2	[64,95]--> [64, 79]	[0,63]
2.5	[80, 95]	[64, 79]
3	[96,127]	[64,95]-->[80,95]



[see Appendix G.a]

4.3 Naming of new Cells

There are two standard behaviour for requester to name a new child, depends on name of the Slave node of the creator.

If the greater whole number of Slave id and Requester id are the same (e.g. 3.1 and 3.999 or 3.4 and 4):

Name new node with float (Requester id + (Slave id - Requester id)/2)

Else:
Name new node with the greater whole number of Requester Id
If Requester Id is already a whole number, it's taken Requester Id + 1.

This naming behaviour is needed to maintain the total order relationship between nodes, other structures as P2P Chord [5] use a PseudoRandom Generator to name new nodes, and generate a new random number between two ids, using high value numbers, hoping there won't be consecutive ids (this solution can be easily avoided with our method).

4.4 Deleting a node (cell absorption)

A node could be underutilized if stored values are not requested or updated. In that case a node could be deleted from the cycle. A node could be deleted due to a unexpected crash. Similar to adding operation, in deleting operation just three nodes, two after the crashed node and the deleted one.

The Slave (the node after) of a deleted node must merge Master Memory and Slave Memory into his new Master Memory. In fact, he has to add to his Master Memory the part of deleted node, that is his Slave Memory. When a Master dies, the related Slave becomes the new Master.

Now this node has just to ask to previous node his Master Memory to make the Slave Memory, because the node before the deleted node now has lost his Slave.
After that, the node that is after the deleted node replacement has to increase the Slave Memory with new data of his Master.
Example (node 2 crashed):

id	Master Memory	Slave Memory
1	[0,31]	[112,127]
1.5	[32,63]	[0,31]
2	[64, 71]	[32,63]
2.25	[72, 79] → [64, 79]	[64,71] → [32, 63]
2.5	[80, 95]	[72, 79] → [64, 79]
3	[96,111]	[80,95]
4	[112, 127]	[96,111]

4.5 List Operations

The whole system provides a list which contains several informations about each node. The list is supposed to update its informations in case of an added node or a dead node. At first boot each node has its own list, everybody knows each other. In case of scale-up and scale-down a specific message needs to be seen by every alive node in the whole system. Here's the node's structure:

[see Appendix G.b]

4.5.1 List Structure

The list is represented by a Python dictionary in which each elements is like id : ListValue, where the list value is an object that contains the target node with its master and slave. Furthermore, each node keeps stored a dead list, useful in the memory transfer process.

4.5.2 List Channel

In order to describe the receiver - sender structure we have to define a communication channel between nodes beforehand. We have only two possibilities: Internal Channel, External Channel.

4.5.2.1 Internal Channel

The main goal is to handle messages from/to:

- Memory Management: scale-up, scale-down requests, memory transfer notifications
- Slave node: at first boot the slave needs to notify its presence to the master
- Slave of slave node: when the slave dies, a new connection is established with the slave of slave
- New added node: when a new instance is created, the new node has to notify its presence to the master

4.5.2.2 External Channel

The main goal is to handle messages from/to:

- Master node: as we said, each node has a connection with the following one to propagate a message and complete the cycle

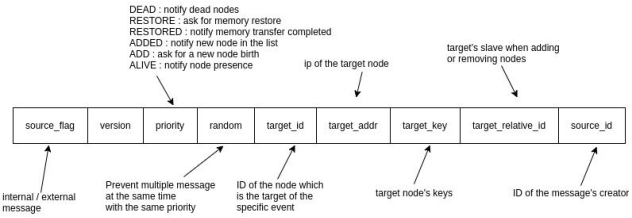
4.5.3 Message

A message is the base information exchanged in order to notify any kind of update to the system list. These are the basic requirements to create a successful message structure:

- It is not compulsory for a cycle to be fulfilled, in general
- If a node begins a cycle with message m and no other messages are generated during the message m propagation, that cycle must end
- If multiple messages are generated at the same time (it can be also a period), one of them must complete the cycle

To provide a consistent update of the list, a version is necessary, each version represents an accurate situation of the cycle at that moment (e.g. version = 5, it means that we are at the fifth list event of the system). A priority is given due to different events like adding or removing a node. Sometimes two nodes ask for adding/removing a node (message with the same priority) at the same time, the solution given is a random number to decide which message has to be forwarded. Each node has a version to send referring to the message and a last seen version to control the cycle flow. Furthermore, last seen priority and last seen random are provided for the same aim. These are the list version requirements:

- If a cycle ends with version v, each node has v as last seen version
- If a cycle ends with version v and a node wants to ask for a node addition or removal, the sending message version is v+1
- When a node creates and sends a message m, last seen priority and last seen random are immediately updated with m's priority and random
- When a node with id = x receives an external message m whose source_id is not x, last see version is updated to m's version



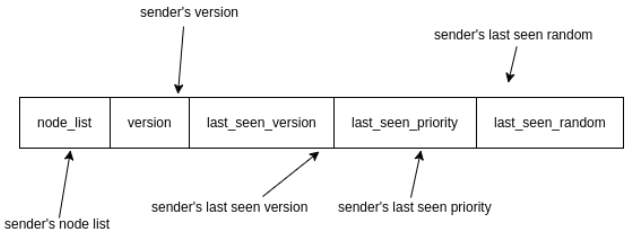
Message Structure

[see Appendix G.c]

4.5.3.1 Information Message

So far we discussed about a kind of message that is propagated through the list cycle. Let's talk about a particular internal message that is used by Receiver / Sender to exchange list informations when a

new node begins its life in the cycle. This message is used during a new node creation process. Here's the structure:



Information Message Structure

[see Appendix G.d]

4.5.3.2 Neutral Message

A neutral message is essential in these situations:

- Notify the presence of a new node in the list
- Notify that a node is being removed from the list

These are the main features:

- A neutral message doesn't have to respect the list flow rules, it's just to inform the rest of the list
- A neutral message doesn't have unuseful informations, each node has to act accordingly, nobody is able to ignore it

4.5.4 Transition Table

To build a stable system, which can support node creations and removals easily, we need a state machine. Each node in the list has the same basic behavior, but if it's necessary to add or remove nodes, they need to act accordingly. These are the states that represent the node's behavior.
We have to introduce a few assumptions in order to describe the states:

- Each message that comes to the node respects the list flow rules
- The interested node has id n
- When we talk about relatives or relatives of relatives, we assume that myself is included

These are the states:

- **Free** : nothing to do, can accept everything from the previous node
- **BusyAddPS** : n is waiting for a message containing a new node (let's say t) added, the creator of t is n's relative. Until the process is finished, n is not allowed to accept other scale out requests from other relatives.
- **BusyAddPL** : n is waiting for a message containing a new node (let's say t) added, the creator of t is n's relative of relative. Until the process is finished, n is not allowed to accept other scale out requests from other relatives.
- **BusyDeadPS** : n is waiting for a message containing an old node's memory (let's say t) restored, the creator of t is n's relative. Until the process is finished, n is not allowed to accept other scale in/out requests from other relatives.
- **BusyDeadPL** : n is waiting for a message containing an old node's memory (let's say t) restored, the creator of t is n's relative of relative. Until the process is finished, n is not allowed to accept other scale in/out requests from other relatives.

In the following transition matrix we will talk about messages that respect the list flow rules.

from \ to	Free	BusyAddPS	BusyAddPL	BusyDeadPL	BusyDeadPS
Free	X	pas	pal	pdl	pds
BusyAddPS	added or pa	paa and ps	paa and pl	pad and pl	pad and ps
BusyAddPL	added or pa	paa and ps	paa and pl	pad and pl	pad and ps
BusyDeadPL	restored or pa	X	X	pad and pl	pad and ps
BusyDeadPS	restored or pa	X	X	pad and pl	pad and ps

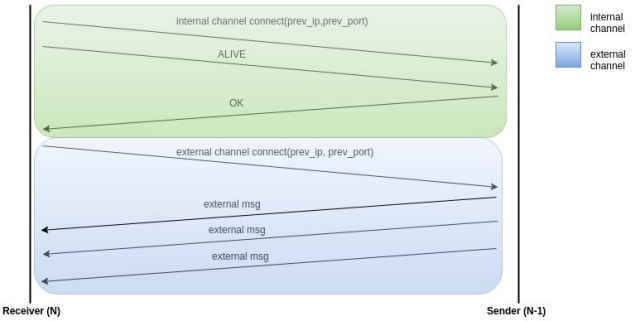
Legend

pas = new add message sent by a relative
pal = new add message sent by a relative of relative
paa and ps = new ADD message sent by a relative that passes the previous ADD message in the same cycle with version *v*
paa and pl = new ADD message sent by a relative of relative that passes the previous ADD message in the same cycle with version *v*
pad and ps = new RESTORE message sent by a relative that passes the previous RESTORE message in the same cycle with version *v*
pad and pl = new RESTORE message sent by a relative of relative that passes the previous RESTORE message in the same cycle with version *v*
added or pa =
1) new ADDED message whose target is a relative or relative of relative
or
2) new ADD message sent by a node that is not relative or relative of relative that passes the previous ADD message in the same cycle with version *v*
restored or pa =
1) new RESTORED message whose target is a relative or relative of relative
or
2) new RESTORE message sent by a node that is not relative or relative of relative that passes the previous RESTORE message in the same cycle with version *v*

[see Appendix G.e]

4.5.5 Receiver

This thread handles incoming messages, typically from an external channel and puts the message into the queue. The internal channel is only used by the receiver to attach itself to the master. The main function is to provide a sort of alarm just in case of dead nodes. If it doesn't receive any message within a socket timeout, a new dead node cycle is started.
At first boot, the reader starts a synchronization process with the previous node. This phase is essential in order to initialize the internal and external channel. The receiver thread sends an ALIVE message to notify its presence to the previous one and waits for a reply, after that the channel is considered ready to receive external messages.



[see Appendix G.f]

4.5.5.1 Dead Node

The receiver thread creates a DEAD message and puts it into the receiver / sender queue. This kind of behavior guarantees decoupling between receiver and sender. In this case the connection initialization explained before restarts. The only difference is that this time the

receiver needs to know its master of master (remember that master is dead), it's always possible to know this information by getting the node id from the list.

4.5.5.2 Added Node

The receiver thread simply checks for the target_id field of the message to understand if it is the new master or master of master or just a new node without any relationship. The new node is stored in the list and if we are talking about a new master, a connection initialization is restarted, just as if there were a dead node or at first boot.

4.5.5.3 New Birth Connection

When a new node is created the procedure is:

- Notify the memory management section about a new memory transfer
- Synchronize with the master (just like for dead nodes or at first boot)
- Make and send an ADDED message to the master to start a cycle for new added nodes

4.5.6 Sender

This thread receives messages either from an internal channel or from the queue whose only producer is the receiver thread. The main activity of this thread is to evaluate and forward messages to the next node provided that it respects some list flow rules., here's the pseudocode:

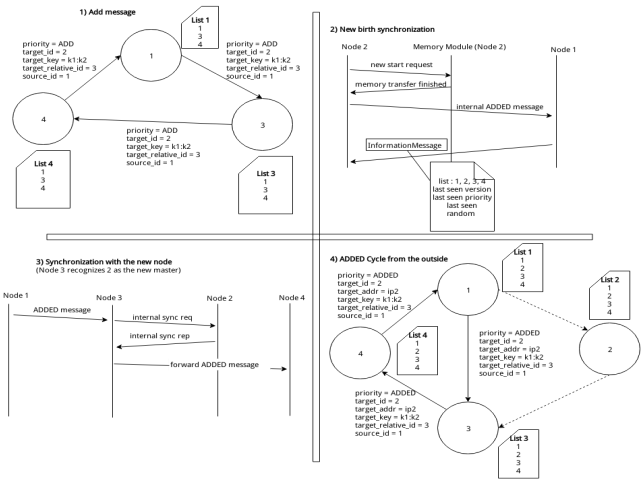
```
if msg_version >= last_seen_version:
    if last_seen_priority < msg_priority:
        consider_message(msg)
    elif last_seen_priority == msg_priority:
        if last_seen_random < msg_random:
            consider_message(msg)
```

This code is only for non neutral messages, because they have to send a "request" to trigger a specific event, so they need a common agreement. If we are talking about neutral messages, we have only to check if the message contains a version that respects the list flow rules.

4.5.6.1 Scale out

The sender thread receives a scale out request from the memory module, then it sends an ADD message to the following node and waits for a new message. Now these are the possibilities when the sender s receives a message from the queue (let's say m with version v):

- The message m respects the list flow rules but is not equal to the last add message, somebody won the cycle with version v, if the current state allows s to send a new scale out message, s sends a new message m with version v+1.
- The message m respects the list flow rules and is equal to the last add message. The node that sent the message creates a new EC2 instance. After a brief synchronization, a new added cycle starts to notify the rest of the list.



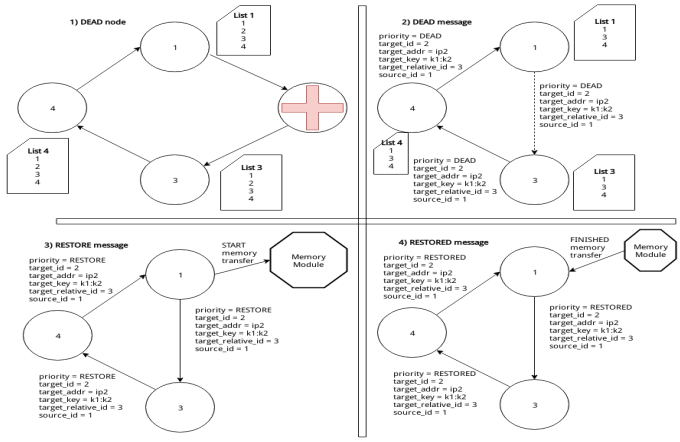
[see Appendix G.g]

4.5.6.2 Scale in

The sender thread receives a scale in request from the memory module, then it sends an ADD message to the following node and waits for a new message. Now these are the possibilities when the sender s receives a message from the queue (let's say m with version v):

- The message m respects the list flow rules but is not equal to the last add message, somebody won the cycle with version v, if the current state allows s to send a new scale in message, s sends a new message m with version v+1.
- The message m respects the list flow rules and is equal to the last add message. The node that sent the message stops its usual behavior and exits. The master recognize the following node as dead:
    1. A new DEAD message is sent, the master waits for the DEAD cycle to be completed
    2. A new RESTORE message is sent, the master waits for the RESTORE cycle to be completed
    3. Suppose that the cycle is completed (the message could be lost due to other RESTORE message with higher random), a new request to the memory module is sent.
    4. After the memory transfer is completed, a new RESTORED cycle is sent by the master. This part is essential because we have to notify that a dead node is completely restored. Now previous and the following node keep the dead's keys, and the list is consistent.

In the following scheme we omit all the list flow fields (not necessary in this example) and represent a list made of only ids (to make it simple). We suppose that node 2 is dead.



[see Appendix G.h]

4.6 Hash Function

Application needs to convert string keys in integers keys, to correctly find the assigned node for every key. The conversion of the key is an highly used operation, for every request it's executed and usually it's not smart to use an lru cache for this values due to variability.

According to a benchmark [12] on hash function performance and collisions, it was choosen to use CRC32 [6] as hash function, thanks to good performance, implementation code availability and a low collision probability.

5. Secant Operations - ExtraCycle Interface

5.1 Get a value

Every node of the system is an entry point for a client communication. Received requests for value by key are forwarded to Master node for that key. It's possible to know the Master node thanks to list of user, that contains for each node the set of maintained keys.
A request of get is sended to Master of a key, from a node to another one, cutting across the cycle of nodes.

The requests are balanced between Master of a key and its Slave node.

In this way, for each key there are always (if there are no crashes) two nodes that can provide the value. For each request just 4 messages are required : two from client to a node, and three secant communication cutting the cycle from a node to another one.

Due to similarity of hash function like MD5 and CRC [6](to associate the name of a key to real used key) and pseudorandom generators the load is statistically distributed on system machines. Similar key names, due to avalanche effect, with high probability are maintained by different nodes. Moreover if a node is overloaded, he can split his load into two nodes with cell duplicating operation.

5.2 Set a value

A set request is sended to a node of the system. As for "Get a value" operation, he can consult the list of nodes to find the correct Master of that key. The request is sended to Master node that updates the value. After that the Master sends to his Slave the new value to update his Slave Memory.
If a key is not in the system, it is created, otherwise is updated.

5.3 Delete a value

A delete request is sended to a node of the system. As for "Get a value" operation, he can consult list of the nodes to find the correct Master of that key and forward the request. Master of that key will delete from memory the object.

5.4 Access point of application

Clients need ip of at least one node of the system to send requests. Service could be registered at a Local Authoritative DNS Server, that periodically receives the complete list of nodes.
Client asks to DNS Server an Entry Point Ip for the system, and the server sends a random ip of the system with low lease time (to prevent obsolescence of data).

5.5 ExtraCycle interface

To guarantee compatibility with Memcached client libraries, available on every platform, it was decided to accept the same interface offered by Memcached Servers. On a configurable port, a client can connect to the system using an existent Memcached client implementation (client standalone, telnet, libraries…) and he cannot see differences between a Memcached server and ExtraCycle CellCycle entry point.
This interface is offered in multithread way, with option to configure the number of thread available to end users. At the moment we can manage Memcached operations: "GET , SET, ADD, DELETE".
The main difference between a pure Memcached server and CellCycle server is that the actual node where the operation is executed is calculated server side, using CRC32 [6] on the key of the value. Using che nodes list of cycle system, request is sent to the correct node transparently, users have not to worry about what is the correct node of a certain key.
Cause of common high ratio between GET operations and SET operations the system balances GET requests between Master node of the requested key and Slave node. SET operations and not readonly ones, must be executed directly on Master node to guarantee sequential consistency for SET operations.

5.5.1 ExtraCycle Interface Commands

see Appendix C.

5.5.2 ExtraCycle: Receiver thread

This thread is only a listener that accept incoming connection over tcp on a specified port. When it receives a request for service, it adds the request to a shared queue and waits for another request.

5.5.3 ExtraCycle: Service thread

Service thread number can be configured. The role is to accomplish every request in shared queue and send response to requester.

5.6 Consistency Model

To guarantee performance, final consistency was the target from beginning. According to CAP theorem, the only way to provide a fast service, without network latences to propagate operations to Slaves, was choose Availability and Partition Tolerance.

Clients can see a different througput for set and get operations. Get operations are parallelized between master and slave copies, set operation are executed serially on master node. Clients has a monotonic-write consistency model using system. In fact, analyzing our Master-Slave system, we can associate to it the primary backup model, of type remote write.

6. Memory Management Module

6.1 Memory Structure: Object Pool

The essential requirement of the application is to store and make available informations dynamically. Our target is to provide this service, taking advantage of all unused ram on cache servers.

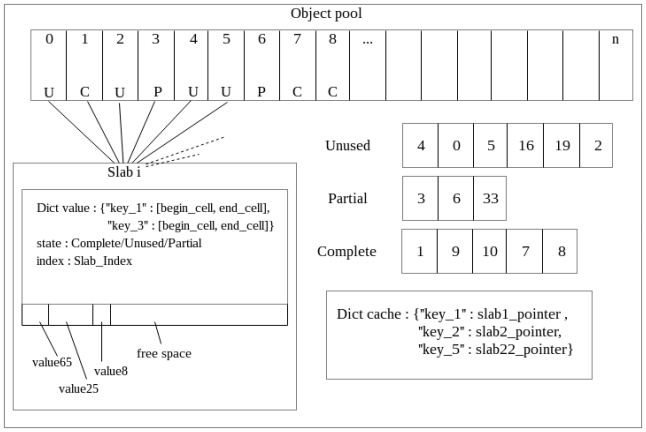
We focused on maximize throughput of get and set operations and minimize memory overhead needed by the application to work.
First design choice consists of storing values in single memory objects, resizable if needed, or building a centralized memory pool, preallocated, to sequentially insert data.

Single objects model is the simplest structure possible, but implies a large overhead, due to the large number of entities stored in memory.

With a preallocated memory pool we can guarantee a minor overhead and great performance, in this way there's no need to ask the system for allocate new pages of memory, all memory is allocated when the program starts, set operations has only to use that space and write pool portions.
To guarantee performances we operated with a low level data structure.

6.2 Memory Allocation: SLUB Allocator

To avoid fragmentation, a typical problem of this kind of architecture, we took inspiration from linux world, using a slab allocation mechanism, in particular SLUB [10] allocator, the default allocator of Linux kernel 2.4+.



[see Appendix G.i]

We preallocate an entire, contiguous, byte array filled with a default value, then we logically split this array in slabs: contiguous portions of bytes of a fixed size.
Size of array and slabs are fixed and configurable through setting file.

To insert new elements on pool, key it's associated with slab pointer in a dictionary of the cache, after that a single key is linked with a slab. In the selected slab there's an association between key and a tuple of two main array cells, begin point of value and end point. Using the tuple is possible to access to value stored on array, in particular, on portion of array reserved by determined slab.

Using slabs there's no way to save a value with length greater than slab size, it's important to choose a predetermined and compatible fixed slab size.
Every value that's it's not greater than slab size could be stored in that slab if it has enough available space. In this way every slab it's filled up with different size values until available space reaches zero.

To maintain memory, there're saved informations about unused, partial and complete slabs into lists.

6.3 Memory Replacement Policy: LRU

Instead of a replacement policy based on timeout, to increase the cache hit we took inspiration from Least Recently Used replacement policy.

Freeing a single value it's not suggestible, due to fragmentation inside slabs deleting single objects. Also this will complicate slab management, storing free holes locations other than last used cell.

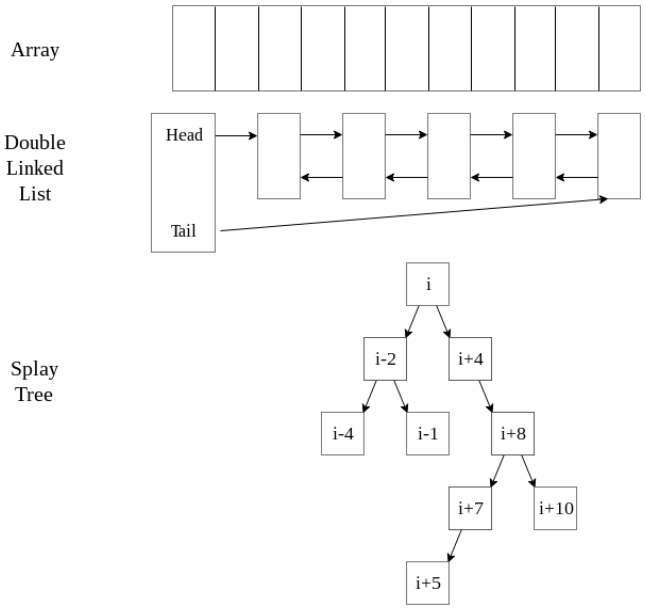
It was decided to have slab replacement policy instead of value replacement policy. When all slabs are partial or complete, then the less used slab is cleared, and the slab is marked as unused to be filled up again. Only slab containing less frequently used values are cleared, with a lazy acting, when it's required to store a new value.

6.4 Data Structures

6.4.1 Data Structures for Slab Cache

To maintain cache services, informations about unused, partial and complete slabs are needed with utilization indexes needed by LRU mechanism.

As described in SLUB definition, there must be lists of partial, complete and unused slabs. Get operations don't involve these data structure, on the other hand set operations initially require a value to insert size compatible partial slab, then an unused slab in lack of right partial, or finally trigger the LRU replacement function that returns a cleared unused slab.



Data structures for allocator must fastly support set operations, that require to move slab from a list to another one (from partial to complete, from unused to partial, from complete or partial to unused) and to find a suitable partial slab (available space must be greater than new value size to insert).
Traditional arrays are not the best strategy, due to heavy impact of remove operations of items (it's required a new array instance and a copy of all elements to new one, or a shift of middle elements number in the average case).
To reach target of performance it was developed a custom double linked list library (LinkedList [13]) that permits adding, removing, moving operations weight in O(1).

Instead of create slab object and double linked list nodes with pointer to slab objects, to speed up the process and to minimize used ram (number of slab could be great), it was decided to merge slab objects and list nodes. Pointers to next element and previous elements (required by double linked list for every element) are stored in slab as attributes.
To maximize the possible parallelism of these three lists (unused, partial and complete) there are next pointer and prev pointer for each list in each nodes (in this way it's easier to maintain lists in multi threaded environment). To support this feature, we created custom libraries LinkedListDictionary and LinkedListArrays.

LinkedListDictionary it's a wrapper of LinkedList, that uses a dictionary (Python Dictionary) in every node to maintain next, prev values and a dictionary in main cache to maintain every head and tail pointer for each list. E.g. the next elements of a node in unused list is stored as "unused-next" in node dictionary. Python Dictionary are documented to permits access in O(1), they are implemented with a hash table function.

LinkedListArrays it's another wrapper of our library LinkedList, developed to be more efficient than LinkdedListDictionary. Instead of Python dictionary there's an array for next pointers, prev pointers in every node and arrays of tails and heads in main cache object. Every index is pre-associated with a list (e.g. 0 for unused, 1 for complete), access to informations it's always O(1).

6.4.2 Data Structures for LRU Mechanism

When LRU Mechanism is triggered, a complete or a partial slab must be selected, cleared and moved to unused list. Fundamental it's to maintain informations about utilization of slabs. A poor used slab must be cleared if needed always before than most used slabs, to increase cache hits.

First idea was to take idea from operative systems and use an array of integers, to increase with every operation, but this will cost O(slabNumber) at every iteration (our system is general purpose intended, it can't use custom hardware to do this).

After that we built LRU list as a simple array, in which every element has a pointer to specified slab object. After a get or set operation the list node associated with involved slab it's shifted up in lru list. At every LRU clear an object must be moved to first index of list, involving the shift of all elements before that. This solution has another trouble, to find the correct node associated to a slab it's needed to read all elements of list. Almost all operations will cost O(slabNumber).

Due to unsatisfiable performance we developed a library using Splay Tree [8] data structure (SplayTree [14]). Splay Tree is a binary tree, each node is a slab. It's called splay for the operation of tree rotation, when a node is splayed it becomes the new root of the tree. This operation costs only O(log(slabNumber)) in the average case. To find the deepest leaf of the LRU tree, corresponding to least recent used node (after a get or a set the slab is splayed, and it becomes the root), we must explore all the three, and it costs O(slabNumber).

Even if slabs number is limited compared to values number, we integrate LinkedListDictionary and LinkedListArrays with LRU list, reducing cost of every operation (shift to first element, shift one element, access to an element) to O(1).

6.4.3 Benchmark of Data Structures

Tests are ideated to push to limits the single threaded throughput. After a theoretical analysis of data structures we ideated several tests to try data structures and their implementation in real simulations. Due to elevated computationally cost of simple array solutions, only Splay Tree and Linked List (and derived lists) was tested.

Simulations are intended to reproduce real world usage of the system. After an empiric study of hypothetical requirements of a real applications, we took inspiration from a Facebook article [7] in which

it's analyzed workload of largest Memcached deployment in the world, capturing "284 billion requests from five different Memcached use cases over a period of 58 days".

We simulated APP and ETC pools discussed in article, that represent most usual and generic usage, using "get/set ratio" with a value of 5 (according to article, surprendly "get/set ratio" it's higher, almost 30 in most cases, higher than assumed in literature. 5 it's a compatible value with some article pools that permits to adequately test set speed). We chose value size of 300 Byte or 900 Byte, choosing values according to chapter 3.2 "Request Sizes" of article: "Except for ETC, 90% of all cache space is allocated to values of less than 500 Byte".

Slab Size/Allocated Ram ratio value is determined to be stable over all parameter changes. Last group test is intended to represent real instances, with a lot of ram used and consequently a low number of slabs purged (to fill a large value of pool, several million of sets need to be executed), other one are intended to represent system in "low resources" environments. Application is intended to be used 24h, over a long time period, in this way the last tests are significant to real resources environment startup, others are significant to reproduce behaviour when all the ram is filled up, forcing system to clear slabs.

Tests are grouped by input settings: [see Appendix A.a]

Architectures under tests are:

- lld : partial, complete, unused and lru list are implemented with Double Linked List, using a dictionary to store lists informations inside slabs and main cache object.
- lla_nu : partial, complete, unused and lru list are implemented with Double Linked List, using arrays to store lists informations inside slabs and main cache object.
- lla_u : similar to lla_nu, except for push method. Objects are inserted in partial list at the begin of the list, instead of the end, in this way new set operations check always the most recently cleared slabs before.
- st_a : partial, complete and unused lists are implemented with simple arrays. Lru is implemented with a Splay Tree.
- st_lla : partial, complete and unused lists are implemented with Double Linked Lists with Arrays. Lru is implemented with a Splay Tree.

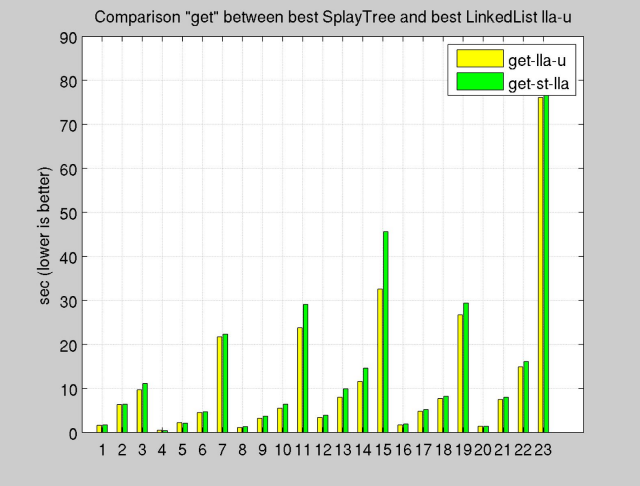
6.4.3.1 Benchmark: Environment

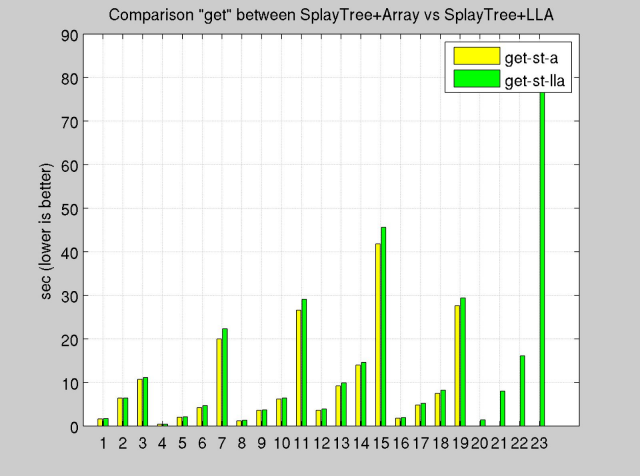
Hardware: HP EliteBook 9470m, CPU Intel I7 2.6 Ghz (with Turbo Boost at 3.2 Ghz), 16 GB ddr3 ram, m-sata ssd 128 GB Samsung.
Software: Linux Debian Stretch (Alpha version) 64 bit, Python 2.7 , stock kernel.
Test: Python memory_profiler module for memory tests and Python cProfile for time profiler.

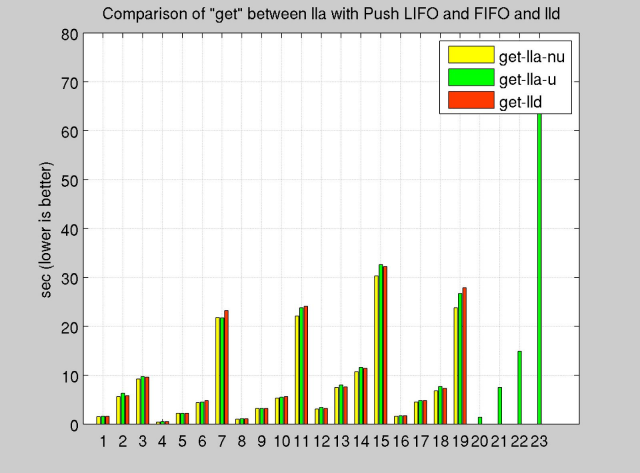
Each result is the average of 3 run.

6.4.3.2 Benchmark: Get Operation

Test results: [see Appendix A.b]

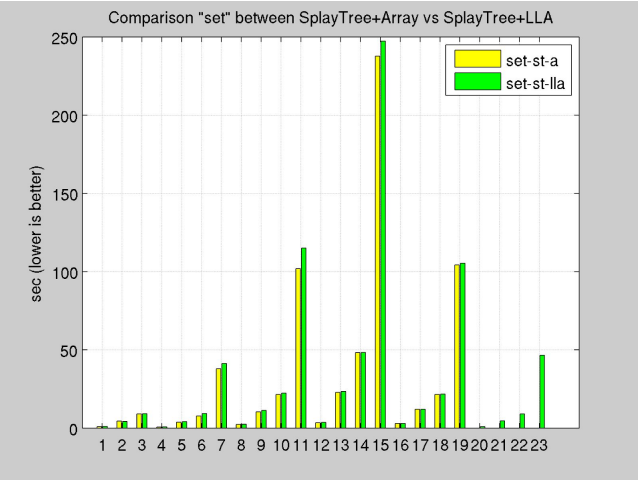


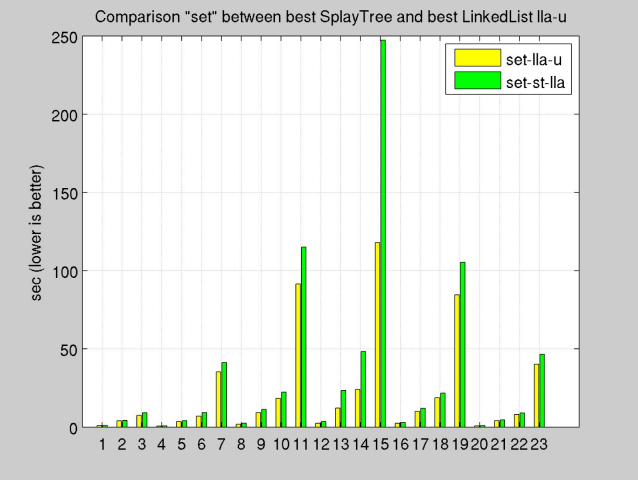


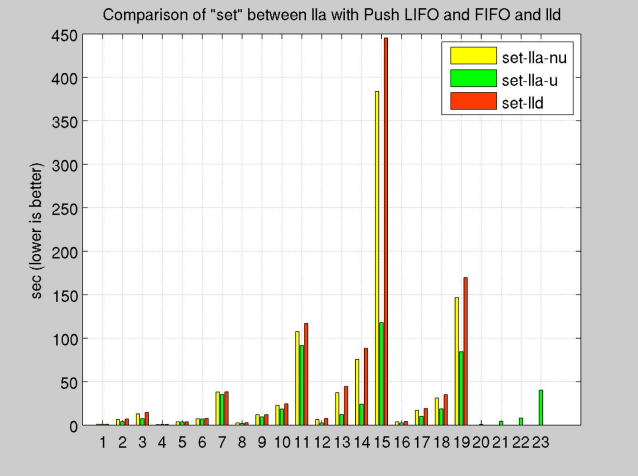


6.4.3.3 Benchmark: Set Operation

Test results: [see Appendix A.c]







6.4.3.4 Benchmark: Memory profile of lla_u solution

System is developed to make the memory overhead as light as possible. Merging linked lists with slab objects (Double Linked List with Dictionary and Arrays) help to reduce memory overhead. After millions of requests, allocated ram doesn't change too much.
[see Appendix A.d]

To store a value a pointer to slab is needed in main cache object dictionary (64bit at least) and begin-end cell information inside slab (key size is 6 byte in average, 30 byte for cell indexes in average). With 5.000.000 of iterations, overhead value is at least: (8+36 byte)*5.000.000 / 5 (get/set ratio in last test case) = 44.000.000 Byte = 44 MB.
This is the reason why overhead is not linked to Object Pool size, overhead just depends on number of stored values. With less allocated ram, objects are replaced, clearing space, explaining the little difference between test with high value(last group) of ram and low level of ram(test 15, 11, 7) with same operations number.
There's no evidence of memory leak, overhead of application is directly linked to necessary informations and it's not linked to Object Pool size (except for fact that a greater pool can contain an higher number of values before LRU mechanisms).

6.4.3.5 Benchmark conclusions

Tests confirm the computation cost study explained before, double linked lists results the best architecture to maximize throughput, in fact every operation costs O(1).
The modification of push function of Double linked list with Arrays has a large impact on set time, in fact compatible partial slabs can be find before inspect all list.
At the same time it's evident that Python Dictionary is not as fast as simple arrays to get elements, this implementation was discarded.

Splay Tree offers good performance, but O(slabNumber) required by lru operations (to find the deepest leaf of the tree) and O(log(slabNumber)) to splay a node (push it up to top of tree after every get or set operation) penalizes overall throughput.

Double Linked List with Arrays, updated with new policy for push function it's the chosen implementation thanks to offered overall performance.

6.5 Parallelization

6.5.1 Architecture

Due to memory management complexity, making the system able to execute multiple set operations at the same time requires a large number of mutex and synchronization objects, that will decrease speed of overall operations. Thanks to Facebook Memcached article, discussed in previous chapter, was clear that set operations are almost irrelevant on "real world experience", with a get set ratio in average greater than 30:1.
Module is organized to have a thread for set operations and multiple threads for get operations, which every getter acquire a lock to access LRU functions, relegating the maintenance of data structures to setter thread.

6.5.2 Get Threads

According to Facebook article about Memcached typical usage, get operations require a great level of parallelization to avoid bottleneck

or lack of performances. Using ZMQ libraries, it was implemented a Request / Reply communication scheme, balanced on different threads using Router / Dealer sockets. In this way, synchronous requests are balanced on different threads using Round Robin scheduler.

6.5.3 Set Threads

Due to slab allocator, set operations requires more controls to be parallelized, mutexes are bottleneck of the system in this configuration, gain of throughput reached by parallelization is hided by mutexes lag. To improve speed of system, set is managed by one thread using a Push / Pull socket of ZMQ, structured as a asynchronous communication between client and server.

6.5.4 Set To Slave Thread

A Thread is dedicated to send to slave node new pairs key,value setted in memory. The communication model between set thread and set to slave thread is based on Python Queue, an asynchronous fifo data structures, thread safe, used to decouple real time set operations to set to slave memory node operation.

6.5.5 Proxy Get Thread

ZMQ library needs a thread to proxy from a single socket frontend to a backend interprocess socket, usable from multiple threads at the same time.

6.5.6 Metricator Thread

To measure usage of the system, and manage scale up and down requests from Master Memory Process of the application in each nodes, a reserved thread, with an interval setted by configuration file, calculates usage of set and get threads. Every threads save the working time in each period, metricator thread uses that informations to calculate actual throughput of master memory process. If utilization ratio is out of bounds setted by configuration file a new scale up or down request is sended to cycle operations module.

Instead of using memory capacity we decide to use as load metric the real throughput of memory module. In this way we preserve elements as long as possible, don't erasing values before it isn't necessary. Users can choose two different parameters of load measured as ratio of working time of get threads and set thread. In the end there's the possibility to choose the interval of time needed from an average load time and another one. When the calculated average is above or beyond limits a request of scale out or in is sended to chain module, and executed if possible, on the other hand if it's not possible it is demanded to a future request.

To avoid burst requests of nodes, scale period is rounded through a gaussian distribution. The real scale period is calculated real time iteratively by a normal distribution, of mean the configured parameter.

6.5.7 Socket Serialization

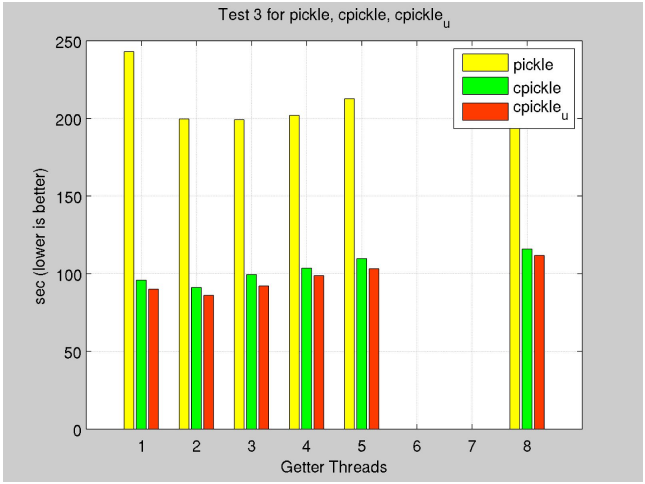
Standard Python ZMQ socket uses "pickle" to serialize object before sending on pipe. To reduce latency in communication it's used an optimized pickle,written in pure C, called 'cpickle'.
After first versions we reorganized imports in source code to gain the maximum throughput, in benchmarks this optimized version is visible with footer u (updated).

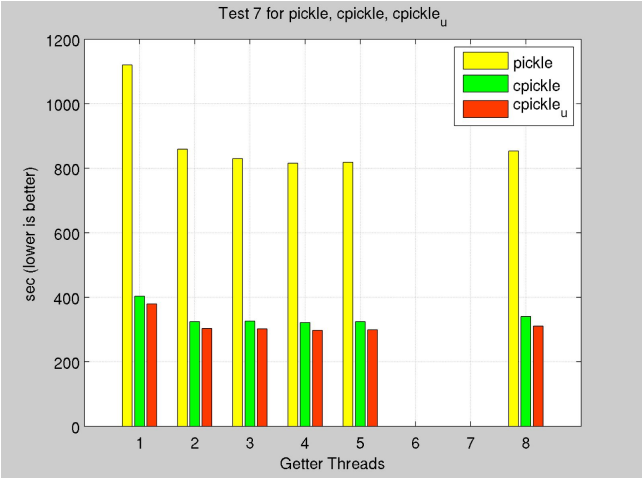
6.5.7.1 Benchmark

Benchmark between 'pickle', 'cpickle' and 'cpickle_u' versions use same tests environment discussed in previous chapter. In particular just test 3 and 7 was used to check performance of these serialization functions.
Tests were executed with different number of getter threads.
Due to multithread/task architecture, it's not possible to profile separately get and set operations, tests result times include initialization time, preparation of test input and side modules times.
In release version it was implemented cpickle_u.

6.5.7.2 Benchmark: Environment

Hardware: HP EliteBook 9470m, CPU Intel I7 2.6 Ghz (with Turbo Boost at 3.2 Ghz), 16 GB ddr3 ram, m-sata ssd 128 GB Samsung.
Software: Linux Debian Stretch (Test version) 64 bit, Python 2.7 , stock kernel.
Test: Python memory_profiler module for memory tests and Python cProfile for time profiler.
Each result is the average of 3 attempts.





7. Software Specification

7.1 Deployment

Application will run almost always on virtual machines, target is to deploy it to Amazon Web Services EC2 instances.
Application will run on VMs, on Linux OS.
Tests are made on T2 Nano and T2 Micro instances of AWS.

Application starts on custom AWS ami, based on ubuntu, where we basically setup git repository to clone source code and where we add our AWS credentials.

7.2 Programming Language

Application is based on Ram sharing, as a service, so it needs to easily access and manage in memory values.
Java (Hazelcast implementation) Memory management is demanded to JVM and Garbage Collector, in his 32 bit version supports a maximum of 2 GB of memory for each process. In 64 bit version this limit is not valid, but performance with a lot of used ram are poor.

C (Memcached implementation) is the lowest level programming language available, programmers can access directly on system calls to manage data but it needs some effort to communicate between nodes using the network.

Python is an interpreted language based on C. Performance are not comparable with Java and C, but increases the speed of coding, the readability of code (it is a project requirement). Python provides wrapper functions for malloc and free functions of C, offering to programmers full control on memory management.

Almost all latence of the distributed application depends on network delay, not execution time, so programming language execution speed is not relevant for system behaviour.

Moreover Python is compatible with a lot of third party libraries.

7.3 External Libraries

We used ZMQ library [10] with Python to manage communication between nodes. In particular, almost every socket are based on ZMQ socket. We used Push/Pull (for example for External Channel), Req/Rep (for example from ExtraCycle interface to Memory Module).
To manage external connections from clients on ExtraCycle interface, we used pure TCP Python Socket.
To manage communication with AWS tools, we used Boto3 for Python, official Amazon Web Services library.
Instead of use credential hard coded, we prefered to store personal credentials in ~/.aws files.

7.4 Source Code

Application is available at GitHub Repo [11].

7.5 Settings

Settings are defined in config.txt file, parameters are available at Appendix B. Settings used in tests are available at Appendix E.

7.6 Log

Application manages concurrent threads with Python Logger module. Standard Out, Error are saved on “logFile.txt” of the running machine.

7.7 Installation Manual

First, you need to create a Amazon AWS instance based on Linux with PyZMQ and ZMQ. In path /home/ubuntu/git/CellCycle/ clone a CellCycle Repo [11] Fork, where you can change settings file (you can skip this step using zip file of release version, unzipping it on /home/ubuntu/git/ folder on AMI. In this case you have to update config.txt file to use your ami id and credentials).
On AWS ami, in /home/ubuntu/.aws/config and /home/root/.aws/credentials you have to add AWS api configuration files, you can add different profiles.

Then you are ready to boot up the application.

On Non Amazon machine (you need to keep credentials in ~/.aws) you can launch application on Amazon instances with “python firstLaunchAWS <keyPairName> <profileName> <branchName>”.

Then Amazon instances will be created with your account.

8. Test and Performance

Running configuration is at [Appendix F].

8.1 Crash Test

We tested reliability of the system with a simple use case simulation. Test log is available at [Appendix D].

After system bootup, we write a key from a CellCycle interface (on node 1 in this case). We ask system which is the master node of that key, and the result is node 5. After a brief test on new value with a Get, through telnet we connect to node 5 simulating a Memcached-Like tcp communication.

With our custom command “CellCycle KillYourself Terminate” we destroyed the instance directly from Amazon EC2 Management.

After that, we asked on another instance for stored key, and the returned value is the one who we inserted at the beginning, so the system used Slave - Backup copy of that value to recover from node 5 “artificial” crash.

It's important to remark that "CellCycle KillYourself Terminate" is the same thing as an accidental, unpredictable crash of a machine, system notice the death only after that, when master data is lost.

[see Appendix D]

8.2 Real Load Simulation

8.2.1 Notes on Amazon EC2 Environment

Throughput, after some runs, appears to unstable to be used as a metric (due to cpu credit offered by Amazon EC2 T2 [15] and different workload on AWS datacenter during the day.

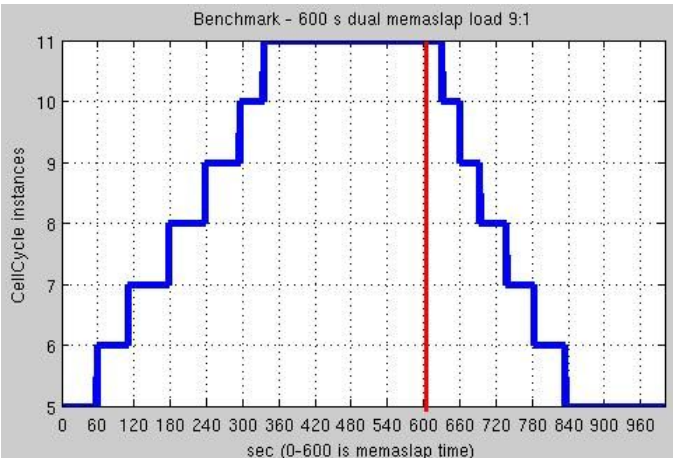
8.2.2 Memaslap

To execute benchmark we used Memaslap [16] of LibMemcached [17] as suggested in not functional needs. Due to compilation problems on recent Linux distributions like the one that we used on AWS EC2 (memaslap compilation flag in makefile is off by default due to common linking problem in compiling), we self compiled source code of version "1.0.18". To avoid network bottleneck during tests, we deployed Memaslap on Amazon EC2 T2 Micro instance, on the same VPC of CellCycle nodes.

As server of Memaslap parameters we use the first five nodes, their interface will use all nodes of the system, even the new ones after scale up events. .

8.2.3 Get:Set 9:1

According to [7] this benchmark represents a real use case, a possible kind of usage of this application. On average throughput is always around 4 MB/s (using more than 1 Memaslap clients on different instances, one is not necessary) that's a Memcached comparable value (10-12Mb/s on same configuration). This is a good result if we consider the difference between Python application performance and C application ones and that Memcached hasn't a multi entrypoint architecture (requests on node's ExtraCycle interface are redirected to other nodes).



Considering around 45 seconds to boot up a T2 Micro instance and start applications, we can see on above plot that application is able to scale up, against a solid, persistent load (2x Memaslap with 8 Threads, from second 0 to 600) with a speed only limited from EC2 T2 Micro instance boot up time. After a growth time, application stable itself to a load level on every node. In this way nodes are not overloaded by not bursty high loads.

After the load end, application has a rapid scale down time. In this phase, every node see a 0 load, so everyone is requesting to shut down his istance. ChainModule cooperates all these requests, and after the number of active nodes reaches the minimum level, all requests are discarded, and system is stable at min number of nodes. Memory is preserved during all phases (according to lru usage), as we can we in memaslap report, where cache misses are limited to 8000 on a very large number of get request [see Appendix E.a].

8.2.4 Get:Set 5:5

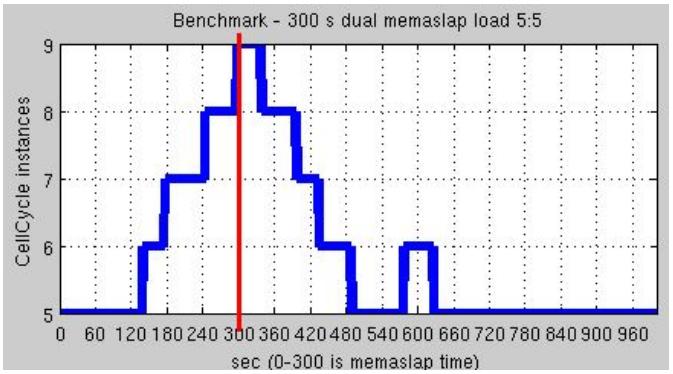
A test where there are as Set as Get is obviously a good way to see application bottleneck limits. Due to its nature, oriented to serve Get requests, application in general seems to have a lower throughput (real measures was not possible due to not stable conditions of environment). We decided to evaluate system behaviour on burst of Set requests instead of long periods (that are not too commons, according to [7]).

System scaled up incrementing instances number after a while (set operations are transmitted to memory module through Push/Pull ZMQ sockets, that make asynchronous the process. In this way it could be a enqueueing phase of initial jobs not immediately detected by metricator thread).. After the end of Memaslap effect (2x Memaslap with 5 Threads, from second 0 to 300), nodes began to request scale down.

System reaches the lowest level of nodes and then creates a new node. This event was largely studied during implementation phase and we understood that it is caused by large memory transfers.

To recover deleted node, nodes spent much effort on transferring slave memories to other ones. In this test, set values was drastically more than in other test, so nodes store more keys. When a node state has to be recovered, much more data have to be transferred, and that effort could be detected as application load, that can cause sporadic scale up events.

[see Appendix E.b]



9. REFERENCES

[1] Memaslap,2013. http://docs.libmemcached.org/bin/memaslap.html.

[2] Memcached, 2016. https://memcached.org.

[3] Redis, 2016. http://redis.io.

[4] Hazelcast, 2016. https://hazelcast.com

[5] Chord, 2016. https://pdos.lcs.mit.edu/chord

[6] Peterson, W. W.; Brown, D. T. (January 1961). "Cyclic Codes for Error Detection". Proceedings of the IRE. 49 (1): 228–235. doi:10.1109/JRPROC.1961.287814.

[7] Workload Analysis of a Large-Scale Key-Value Store - B.Atikoglu, Y.Xu, E.Frachtenberg, S.Jiang, M.Paleczny

[8] Splay Tree specification: http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf
[9] SLUB allocator original article: https://lwn.net/Articles/229984/
[10] ZMQ, 2016, www.zeromq.org
[11] CellCycle Repo on GitHub, 2016, https://github.com/AQuadroTeam/CellsCycle
[12] Benchmark is available at https://github.com/greenrobot/essentials/blob/master/web-resources/hash-functions-benchmark.pdf.
[13] GitHub Repo : https://github.com/Ablablab/PyDLLs
[14] GitHub Repo : https://github.com/Ablablab/pysplay
[15] Amazon EC2 T2, 2016, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html
[16] Memaslap tool, http://docs.libmemcached.org/bin/memaslap.html
[17] LibMemcached, http://libmemcached.org/libMemcached.html