Cell Cycle Specification

A proposal for Elastic Distributed Shared Memory Application

Andrea Gennusa 5-09-2016 - added system delivery model

Target

Cells Cycle is a P2P architecture for a distributed shared memory in the cloud, elastic. Informations are stored like KeyValue objects, values are raw data.

Memory - System is able to store KeyValue objects, in this case volatile memory like DRAM
Distributed - System is composed by different machines where software is executing, all cooperating
Shared - All machines have their own ram. Memory available to all machines is used to offer memory storage service.
Elastic - System is able to scale up and scale down, using the numbers of active machines.

Needs

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)

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

New Not Functional needs (Requirements of 28-08-2016)

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.

New Functional needs (Requirements of 28-08-2016)

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.

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.

Values have an expiration time, after that the value can be deleted from the system (if it wasn't updated, in that case the expiration date is incremented).

Every node could be an entry point for the memory provider service.

Name Space of keys

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.

Values have an expiration time, after that the value can be deleted from the system (if it wasn't updated, in that case the expiration date is incremented).

Every node could be an entry point for the memory provider service.

Cycle Operations

Adding a node (cell duplication)

When a node runs off available ram, 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 ram, 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 ram 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]

Example (node 3 generates node 4)

id	Master Memory	Slave Memory
1	[0,63]	[112,127]
2	[64, 79]	[0,63]
2.5	[80, 95]	[64, 79]
3	[96,111]	[80,95]
4	[112, 127]	[96,111]

Example (node 1 generates node 1.5)

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

Example (node 2 generates node 2.25)

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]	[64,71]
2.5	[80, 95]	[72, 79]
3	[96,111]	[80,95]
4	[112, 127]	[96,111]

How to choose name of a new child:

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

This naming behaviour is needed to maintain the total order relationship between nodes, other structures as P2P Chord 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).

Deleting a node (cell absorption)

A node could have a free memory if stored values reach the expiration time. 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

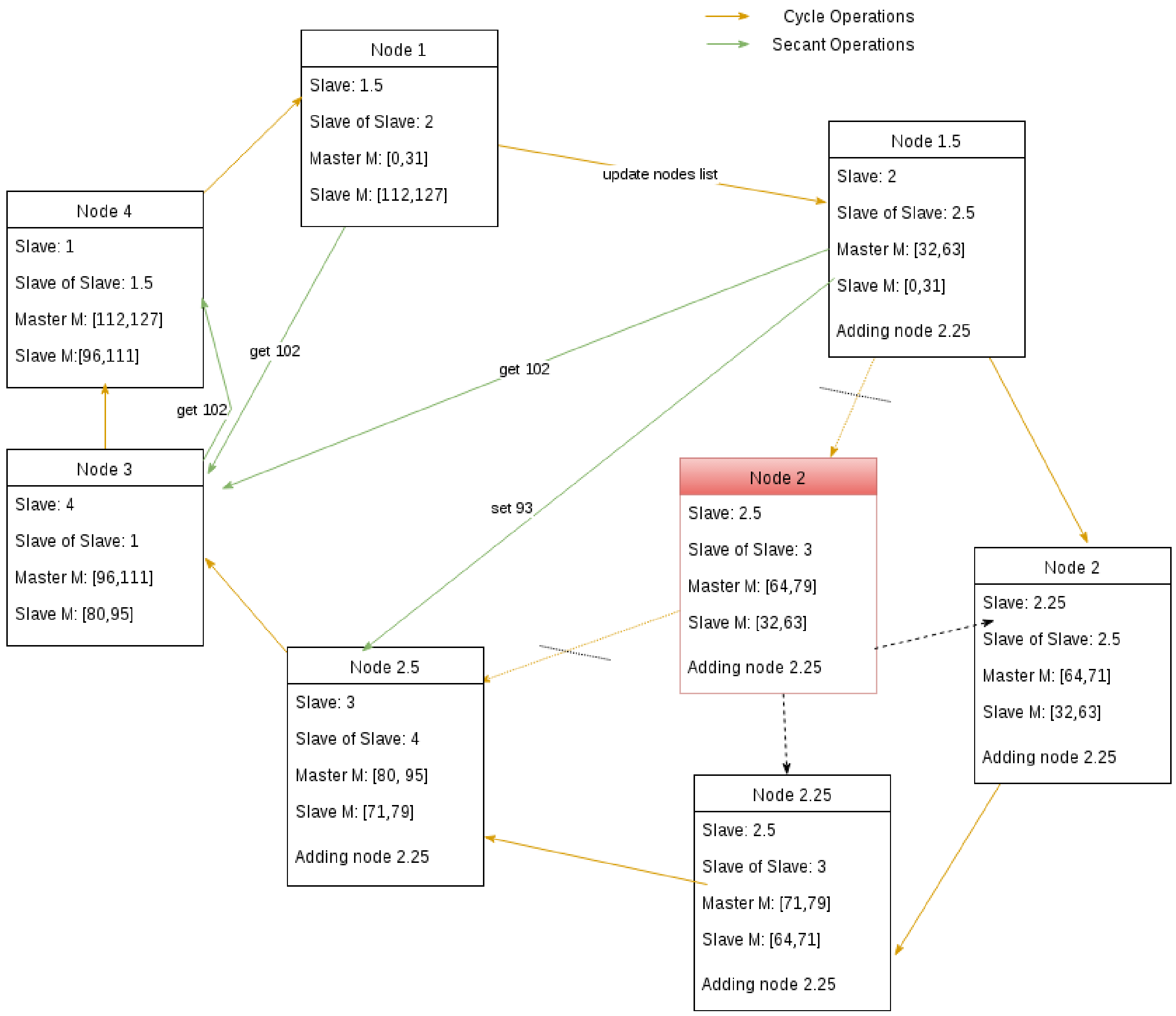
1	[0,31]	[112,127]
1.5	[32,63]	[0,31]
~~2~~	~~[64, 71]~~	~~[32,63]~~
2.25	~~[72, 79]~~ → [64, 79]	~~[64,71]~~ → [32, 63]
2.5	[80, 95]	~~[72, 79]~~ → [64, 79]
3	[96,111]	[80,95]
4	[112, 127]	[96,111]

Updating list of nodes (monitoring alive cells)

We assume there's a node at the beginning with the list of nodes.
The list contains for each row the id, the ip and additional information about a node.
At the boot of the system the node sends to his slave the list with version number 0.
When a node receives the list, it adds his informations to the list (if there aren't yet) and sends it to his Slave, keeping the same list version number. If his Slave is alive and acks the list, the cycle continues.
Otherwise his Slave has a timeout to ack the sent list, when the timeout reaches the end the Slave is declared as dead, it is deleted from the list and the version number of the list is updated. The Slave of the node is changed with the "Slave of the Slave", the next online node of the cycle. In this case, involved nodes state is flagged as "cell absorption" and the deleting node operation is started.

Every node is programmed to accept only the last list, so when it receives a list with updated version number, every list with lower version number is discarded and the node who sent the list is warned about it. That node is out of the system cause of some error, so he shuts down immediately.

When a node is over his ram acceptance level he can add to involved nodes state the flag of "cell duplication", to start the operation to add a node to system. A node can start this operation only if there's no "cell absorption" operation in involved nodes.



Secant Operations

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.
If the receiver is busy, he can forward the request to his Slave, that can answer using his Slave Memory. This is possible only if final consistency is chosen for a key.

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 5 messages are required (4 if Master is not busy): 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 SHA-n (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.

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.

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.

Memory Management

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.

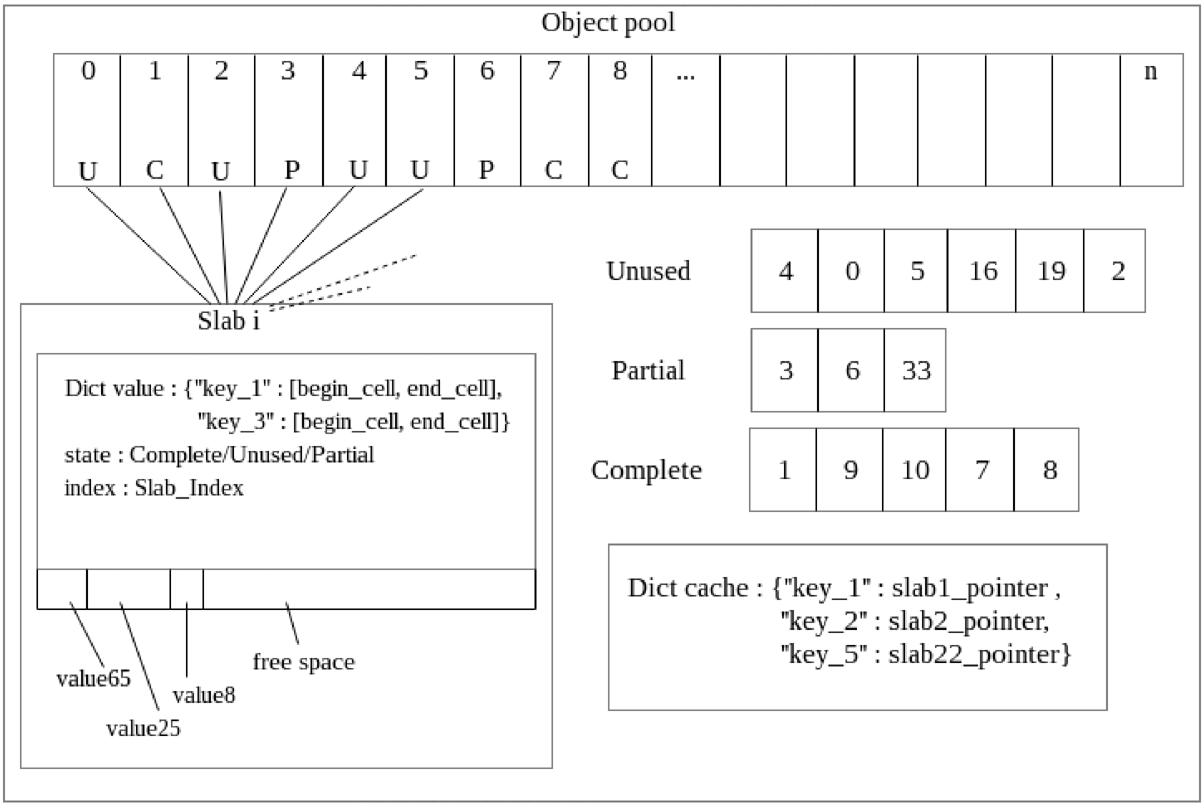
We focused on maximize throughput of get and set operations and minimize memory overhead needed by the application to work.
First design choice consists of storing values in single memory objects, resizable if needed, or building a centralized memory pool, preallocated, to sequentially insert data.

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.

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[1] allocator, the default allocator of linux kernel 2.4+.


Object pool
0
1
2
3
4
5
6
7
8
...
n
U
C
U
P
U
U
P
C
C
Unused
4
0
5
16
19
2
Partial
3
6
33
Complete
1
9
10
7
8
Slab i
Dict value : {"key_1" : [begin_cell, end_cell],
"key_3" : [begin_cell, end_cell]}
state : Complete/Unused/Partial
index : Slab_Index
value65
value25
value8
free space
Dict cache : {"key_1" : slab1_pointer ,
"key_2" : slab2_pointer,
"key_5" : slab22_pointer}

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.

[1] SLUB allocator original article: https://lwn.net/Articles/229984/

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

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.

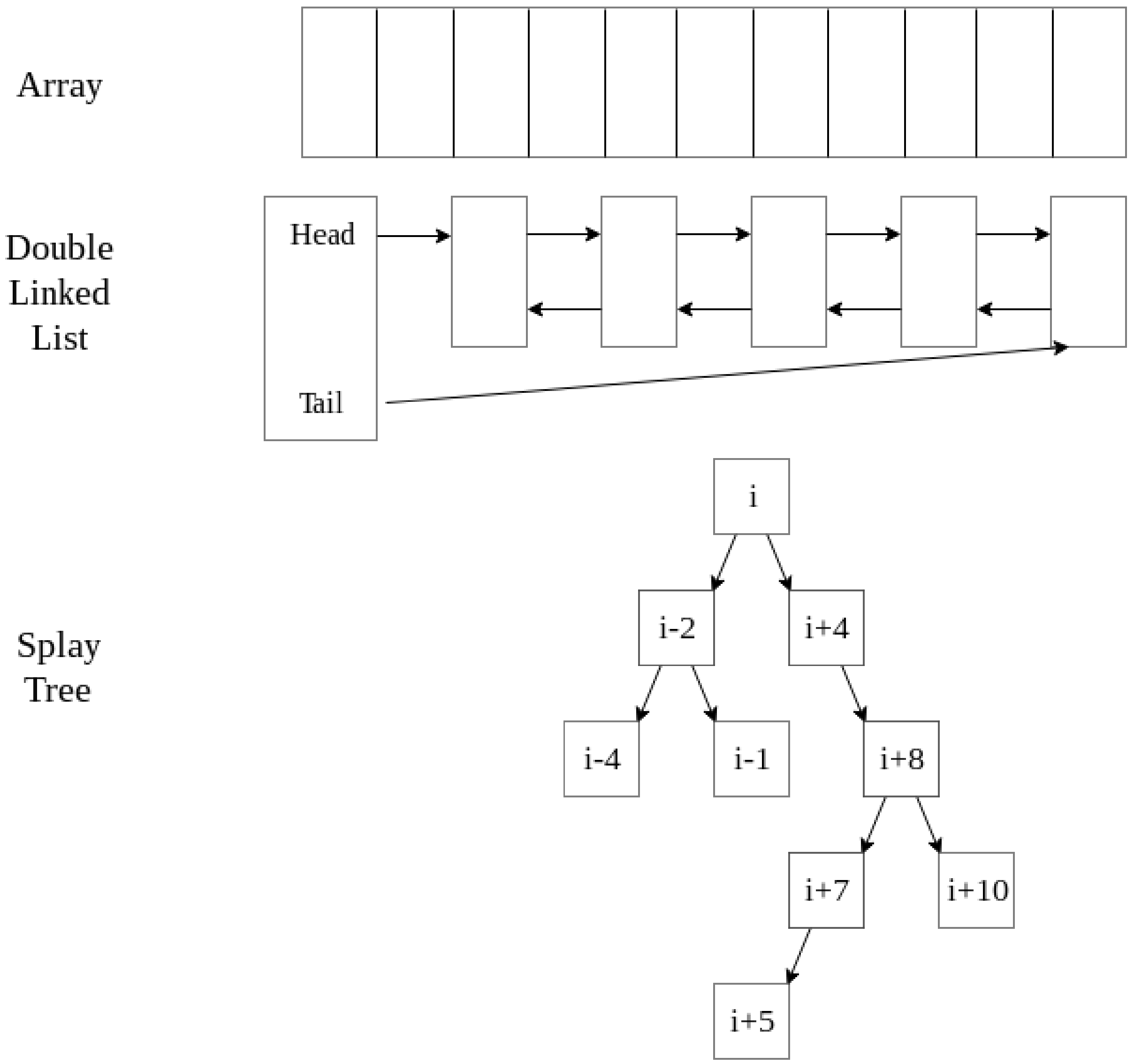
Supporting Data Structures

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

Supporting Data Structures for Slab cache

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) 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).

Supporting 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[2] data structure (SplayTree). 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).

[2] Splay Tree specification: http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf

Benchmark of Supporting 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[3] 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:

Test id	Operations	get/set ratio	values(Byte)	Allocated ram(MegaByte)	Slab Size (KiloByte)	Slabs purged
1	100.000	5	300	10	100	0
2	500.000	5	300	10	100	144
3	1.000.000	5	300	10	100	367
4	100.000	5	300	0.1	10	432

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

5	500.000	5	300	0.1	10	2173
6	1.000.000	5	300	0.1	10	4321
7	5.000.000	5	300	0.1	10	21674
8	100.000	5	900	1	10	1258
9	500.000	5	900	1	10	6492
10	1.000.000	5	900	1	10	13001
11	5.000.000	5	900	1	10	64971
12	100.000	5	900	10	10	507
13	500.000	5	900	10	10	5964
14	1.000.000	5	900	10	10	12597
15	5.000.000	5	900	10	10	64808
16	100.000	1	300	10	100	49
17	500.000	1	300	100	100	521
18	1.000.000	1	300	100	100	1055
19	5.000.000	1	300	100	100	5140
20	100.000	5	300	100	1.000	0
21	500.000	5	300	1.000	1.000	0
22	1.000.000	5	300	1.000	1.000	0
23	5.000.000	5	300	1.000	1.000	0

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.

Benchmark: Environment

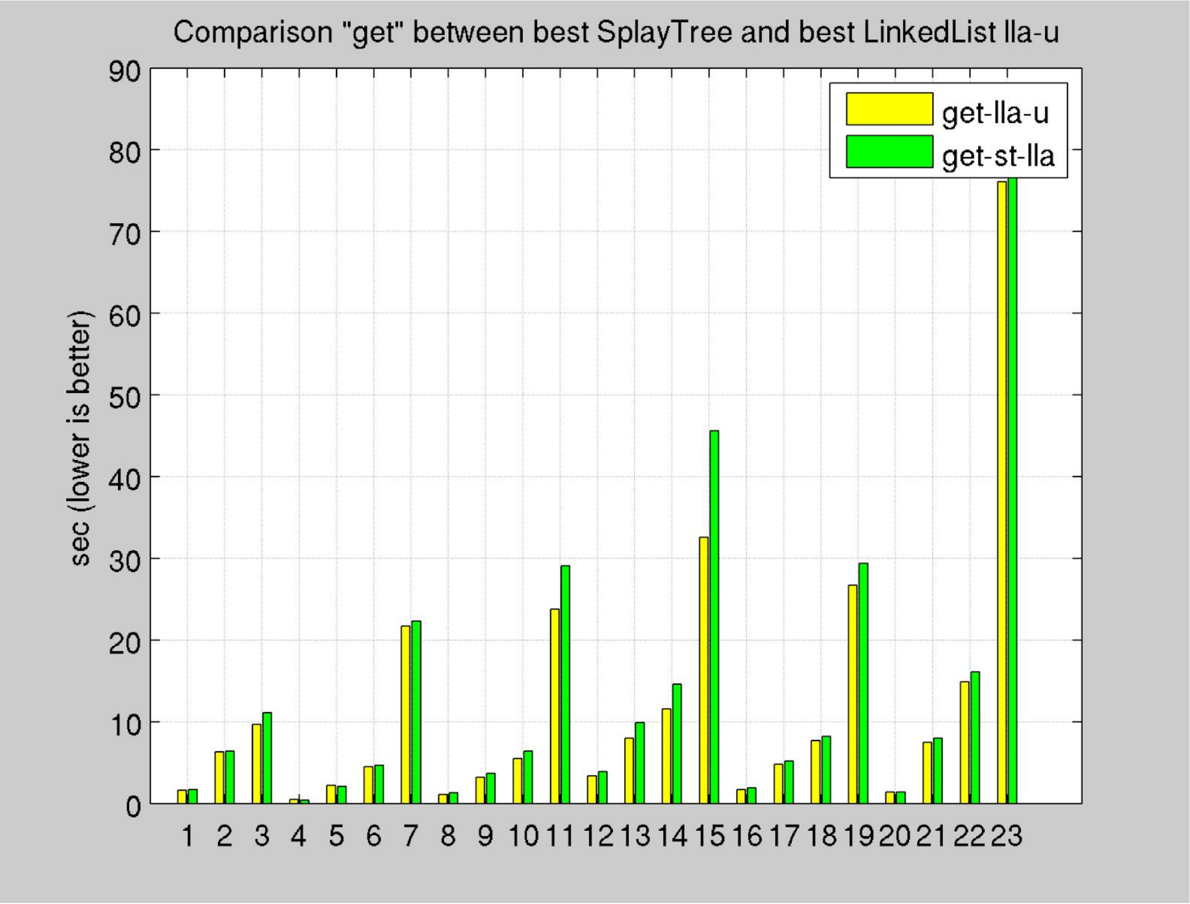
Hardware: HP EliteBook 9470m, CPU Intel I7 2.6 Ghz (with Turbo Boost at 3.2 Ghz), 16 GB ddr3 ram, m-sata ssd 128 GB Samsung.
Software: Linux Debian Stretch (Alpha version) 64 bit, Python 2.7 , stock kernel.
Test: Python memory_profiler module for memory tests and Python cProfile for time profiler.

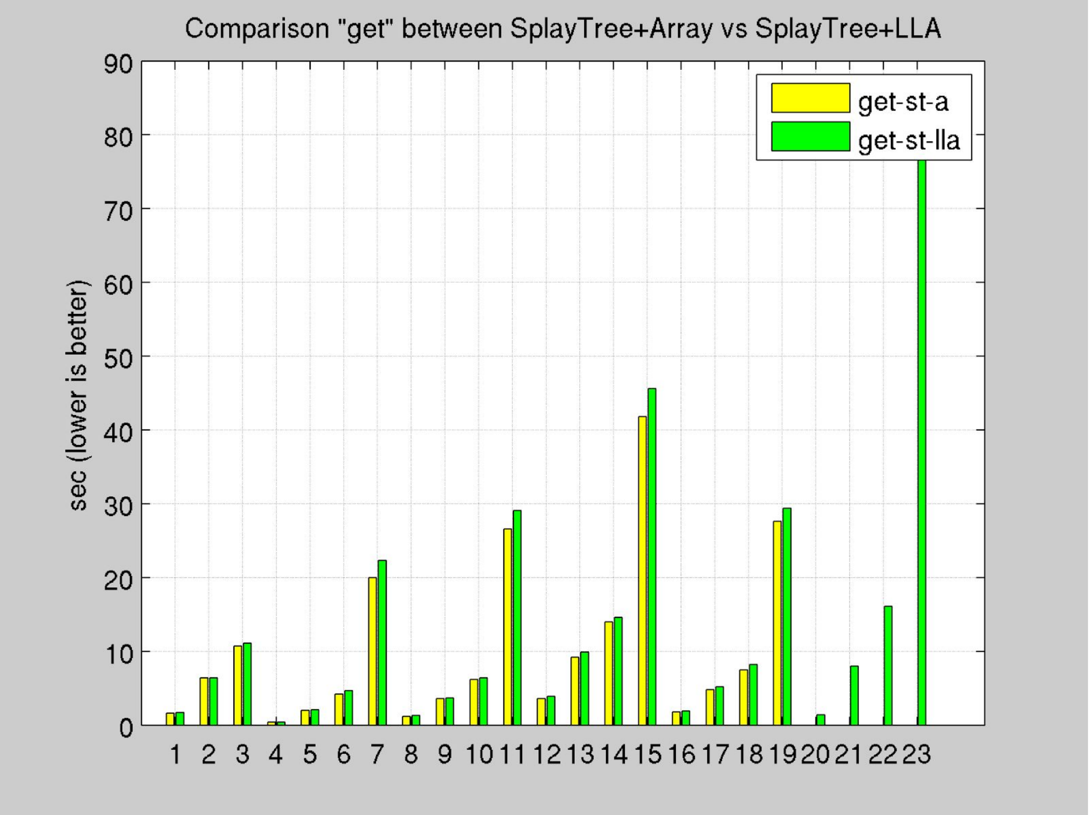
Benchmark: Get operation

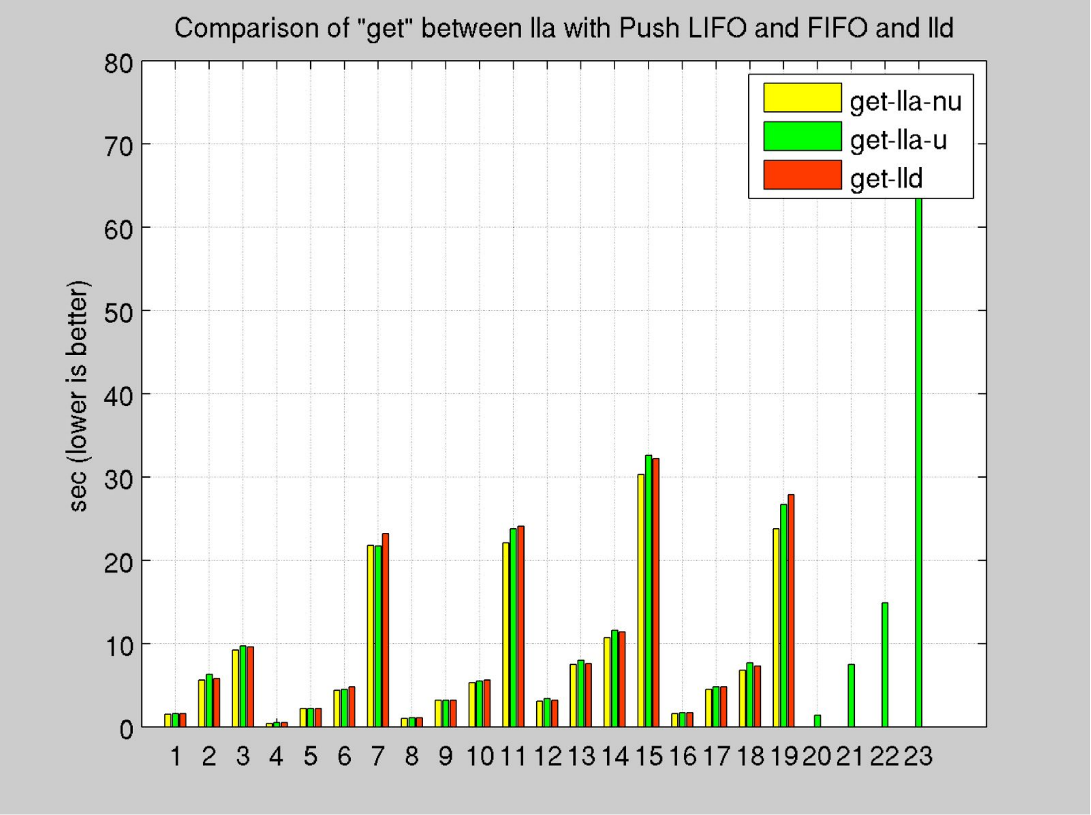
Test results:

Test id	Cumulative get for lld (s)	Cumulative get for lla_nu (s)	Cumulative get for lla_u (s)	Cumulative get for st_a (s)	Cumulative get for st_lla (s)
1	1.6	1.5	1.6	1.6	1.7
2	5.8	5.6	6.3	6.4	6.4
3	9.6	9.2	9.7	10.7	11.1
4	0.5	0.4	0.5	0.4	0.4
5	2.2	2.2	2.2	2.0	2.1
6	4.8	4.4	4.5	4.2	4.7
7	23.2	21.8	21.7	20.0	22.3
8	1.1	1.0	1.1	1.2	1.3
9	3.2	3.2	3.2	3.6	3.7
10	5.6	5.3	5.5	6.2	6.4
11	24.1	22.1	23.8	26.6	29.1
12	3.2	3.1	3.4	3.6	3.9
13	7.6	7.5	8.0	9.2	9.9
14	11.4	10.7	11.6	14.0	14.6
15	32.2	30.3	32.6	41.8	45.6
16	1.7	1.6	1.7	1.8	1.9
17	4.8	4.5	4.8	4.8	5.2
18	7.3	6.8	7.7	7.5	8.2
19	27.9	23.8	26.7	27.6	29.4

20	NA	NA	1.4	NA	1.4
21	NA	NA	7.7	NA	8.0
22	NA	NA	14.9	NA	16.1
23	NA	NA	76.1	NA	86.8



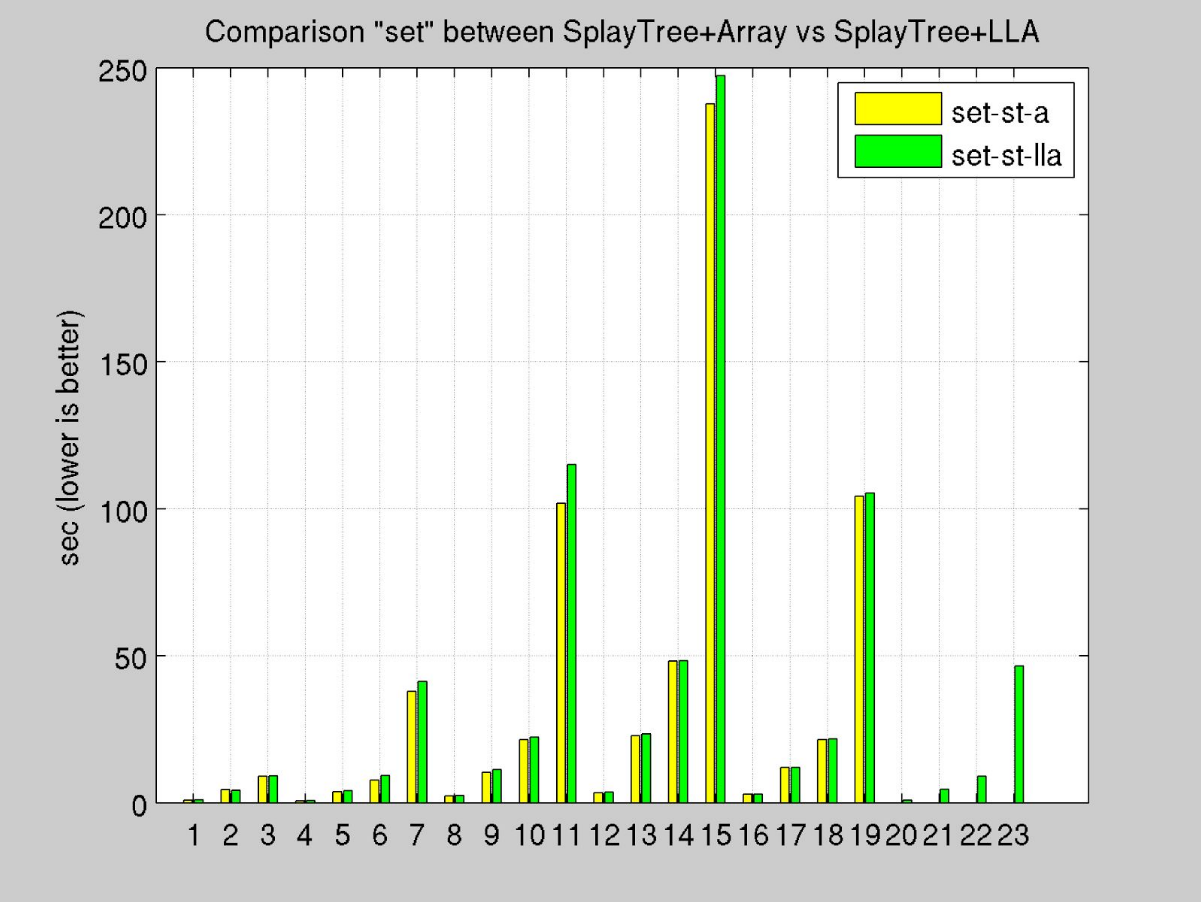

Comparison "get" between SplayTree+Array vs SplayTree+LLA
sec (lower is better)
0
10
20
30
40
50
60
70
80
90
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
get-st-a
get-st-lla

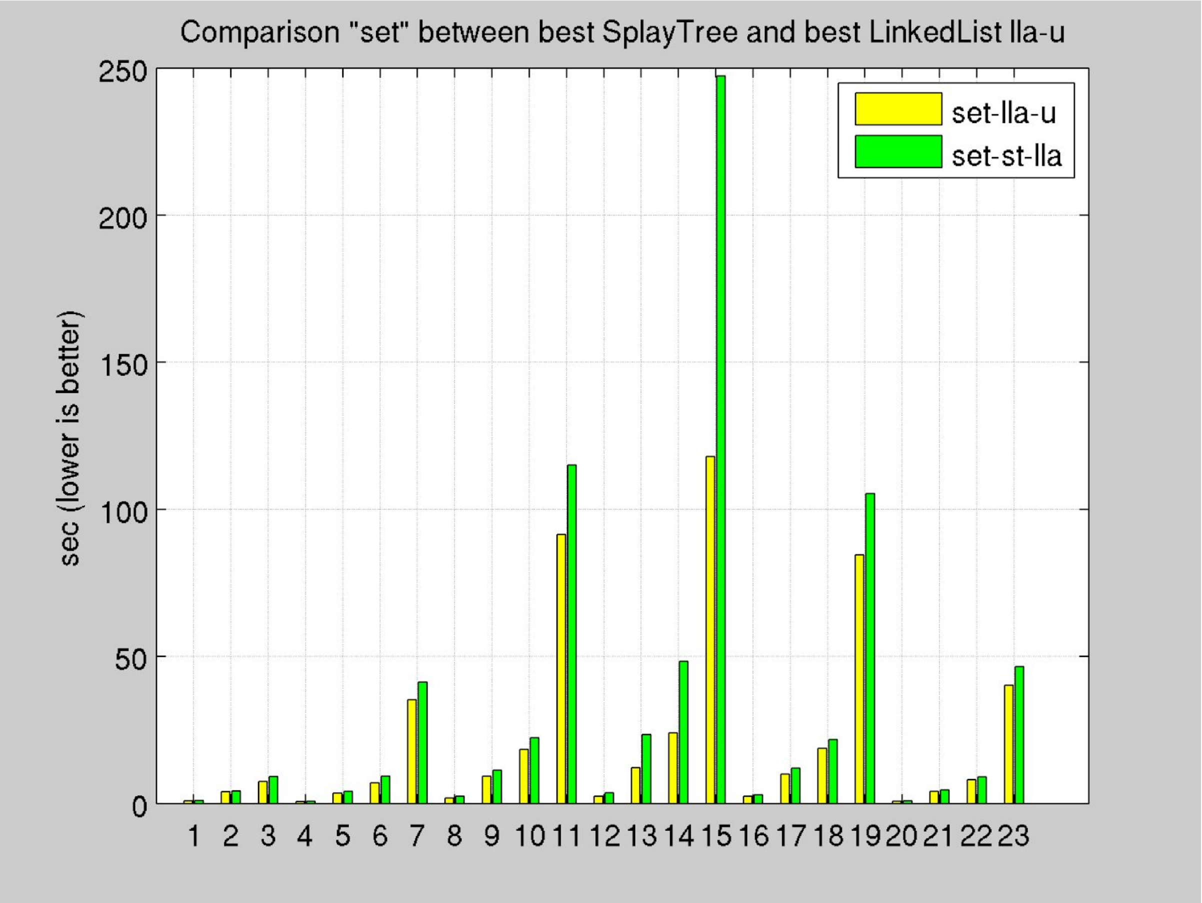

Comparison of "get" between lla with Push LIFO and FIFO and lld
sec (lower is better)
0
10
20
30
40
50
60
70
80
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
get-lla-nu
get-lla-u
get-lld

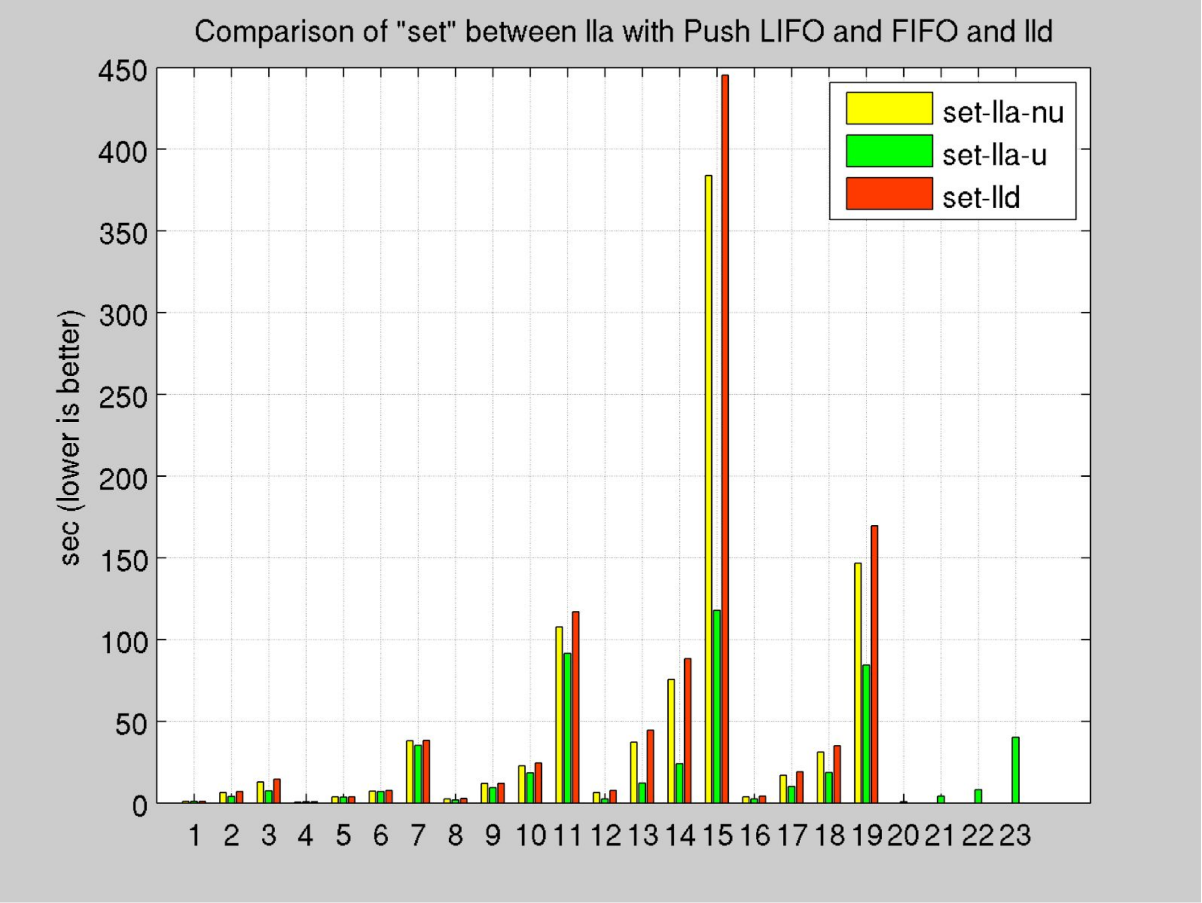
Benchmark: Set operation

Test id	Cumulative set for lld (s)	Cumulative set for lla_nu (s)	Cumulative set for lla_u (s)	Cumulative set for st_a (s)	Cumulative set for st_lla (s)
1	1.0	0.9	0.9	0.9	1.0
2	6.9	6.3	4.0	4.5	4.3
3	14.5	12.8	7.5	9.1	9.2
4	0.7	0.6	0.7	0.7	0.8
5	3.7	3.7	3.5	3.8	4.1
6	7.7	7.3	7.0	7.7	9.3
7	38.2	38.1	35.3	37.9	41.3
8	2.6	2.5	1.9	2.3	2.5
9	11.9	12.0	9.3	10.4	11.3
10	24.5	22.7	18.4	21.5	22.4
11	116.9	107.6	91.5	101.9	115.0
12	7.7	6.3	2.5	3.4	3.6
13	44.6	37.2	12.2	22.8	23.4
14	88.3	75.6	24.0	48.2	48.3
15	445.4	384.0	117.9	237.7	247.3
16	4.1	3.7	2.4	2.9	2.9
17	19.0	16.9	10.0	12.0	12.0
18	35.1	31.2	18.7	21.5	21.7
19	169.6	146.7	84.5	104.3	105.4
20	NA	NA	0.8	NA	0.9
21	NA	NA	4.1	NA	4.6
22	NA	NA	8.1	NA	9.1
23	NA	NA	40.2	NA	46.5

Andrea Gennusa 5-09-2016 - added system delivery model


Comparison "set" between SplayTree+Array vs SplayTree+LLA
sec (lower is better)
0
50
100
150
200
250
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
set-st-a
set-st-lla


Comparison "set" between best SplayTree and best LinkedList lla-u
sec (lower is better)
0
50
100
150
200
250
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
set-lla-u
set-st-lla



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.

Test id	Object Pool (MB)	Actual preallocated RAM (MB)	Allocated RAM after workload (MB)
1	10	10.0	1.0
2	10	10.0	4.9
3	10	10.0	8.9
4	0.1	0.1	1.2
5	0.1	0.1	4.0
6	0.1	0.1	6.8
7	0.1	0.1	32.2
8	1	1.4	1.1

9	1	1.4	4.4
10	1	1.4	6.7
11	1	1.4	31.9
12	10	10.1	1.1
13	10	10.1	5.3
14	10	10.1	8.7
15	10	10.1	33.9
16	100	10.1	1.6
17	100	100.2	7.6
18	100	100.2	13.7
19	100	100.2	41.9
20	1.000	954.9	1.2
21	1.000	954.9	5.4
22	1.000	954.9	9.6
23	1.000	954.9	46.4

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).

Benchmark conclusions

Tests confirm the computation cost study explained before, double linked lists results the best architecture to maximize throughput, in fact every operation cost 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.

Service Delivery Model

Memory Module was tested on single task - single thread environment, where variables were generated and consumed by the same thread.
Building a network server, characterized by a large amount of short requests, requires an architecture with multiple execution flows.
First idea was to use a separate threads to manage connections and memory management. To increase performance, memory module was developed in multithread way.

Internal Model

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, 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.

Intercommunication Model

Using interpreted language as Python, it's not possible to use a simple multithread architecture for overall application, due to GIL presence.
Structuring the memory module internally as parallel threads, organized by role, permits to gain advantage of I/O parallel requests, supported by GIL.
On the other hand, memory module cannot share the same interpreter as other modules, because of parallel operations will become slow.
Memory module is developed as an independent task, guaranteeing a great level of fault tolerance of other modules.
To communicate, memory module uses different pipes, organized by role, to communicate with setter and getters.

Andrea Gennusa 5-09-2016 - added system delivery model

Pipes

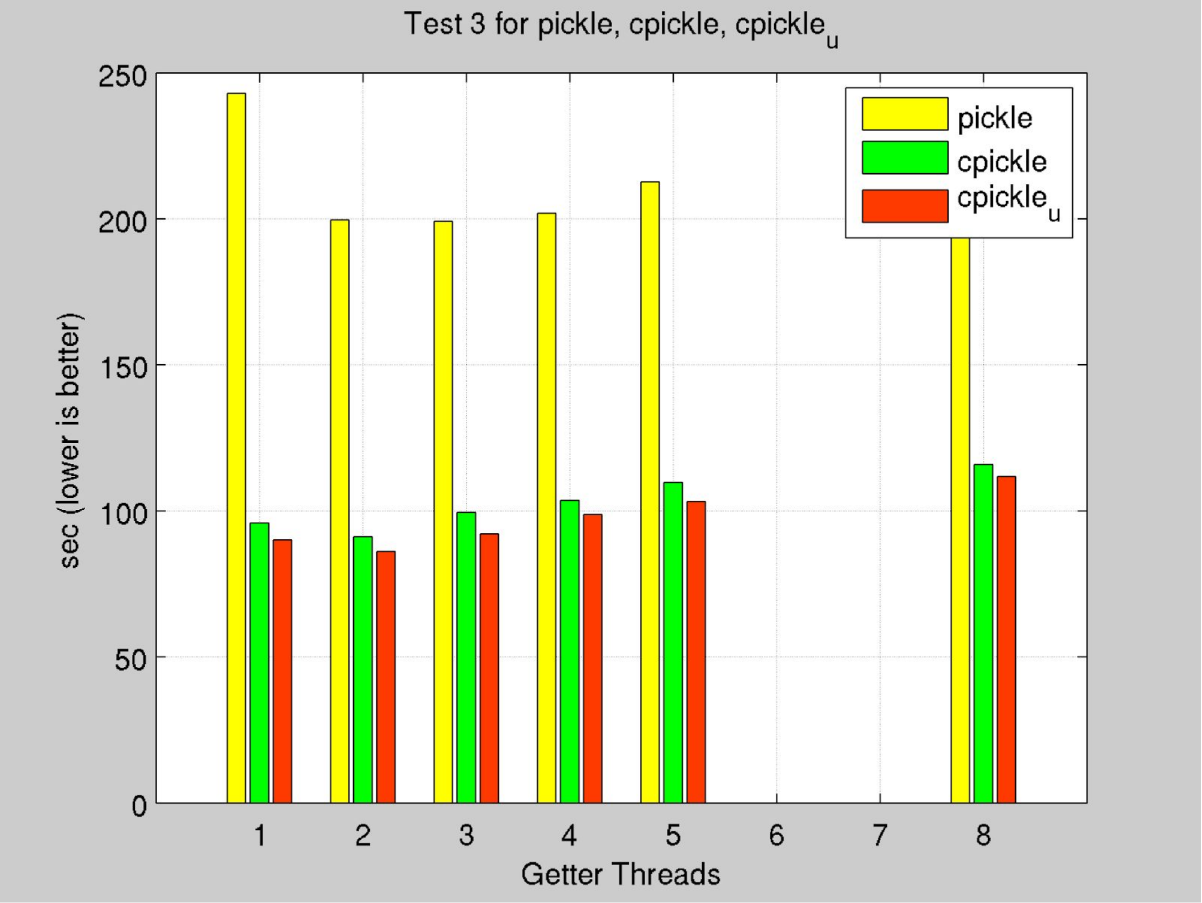
Communicating between tasks needs using shared memory or communication tools like pipes or queues.
Due to the presence of a large number of little size requests, shared memory lacks fast synchronization tools. Pipes are high speed double communication lines, perfect for this requirements.
Standard python pipes 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 that we reorganized imports in source code to gain the maximum throughput.

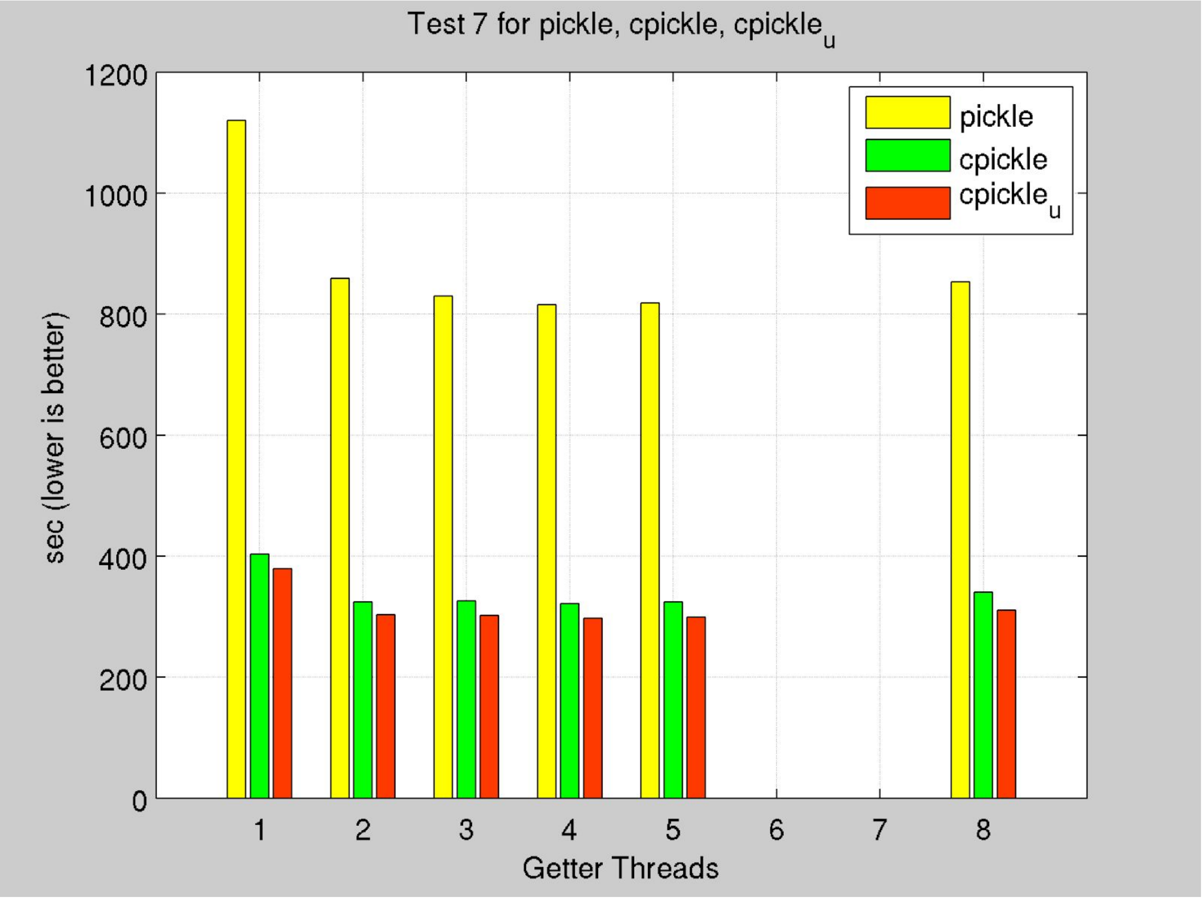
Benchmark

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.

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.


Test 3 for pickle, cpickle, cpickle$_u$
250
200
150
100
50
0
sec (lower is better)
1
2
3
4
5
6
7
8
Getter Threads
pickle
cpickle
cpickle$_u$


Test 7 for pickle, cpickle, cpickle$_u$
1200
1000
800
600
400
200
0
sec (lower is better)
1
2
3
4
5
6
7
8
Getter Threads
pickle
cpickle
cpickle$_u$

Client EntryPoint

Clients need ip of at least one node of the system to send requests. Service is 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).

Implementation

Communication Protocol

System is designed to be compatible with Memcached communication protocol (https://github.com/memcached/memcached/blob/master/doc/protocol.txt) for get and set operation.
Optionally a client could define the consistency level of a single value.
https://github.com/memcached/memcached/wiki/Commands

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, so OS will be Linux to improve performance.

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 increase 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.

External Libraries

We are thinking about using ZMQ library with Python to manage communication between nodes.

Another useful library is ntplib, to access ntp services directly from Python without make system calls.