Cell Cycle Specification

A proposal for Elastic Distributed Shared Memory Application

Andrea Gennusa 28-08-2016 - updated to new requirements

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

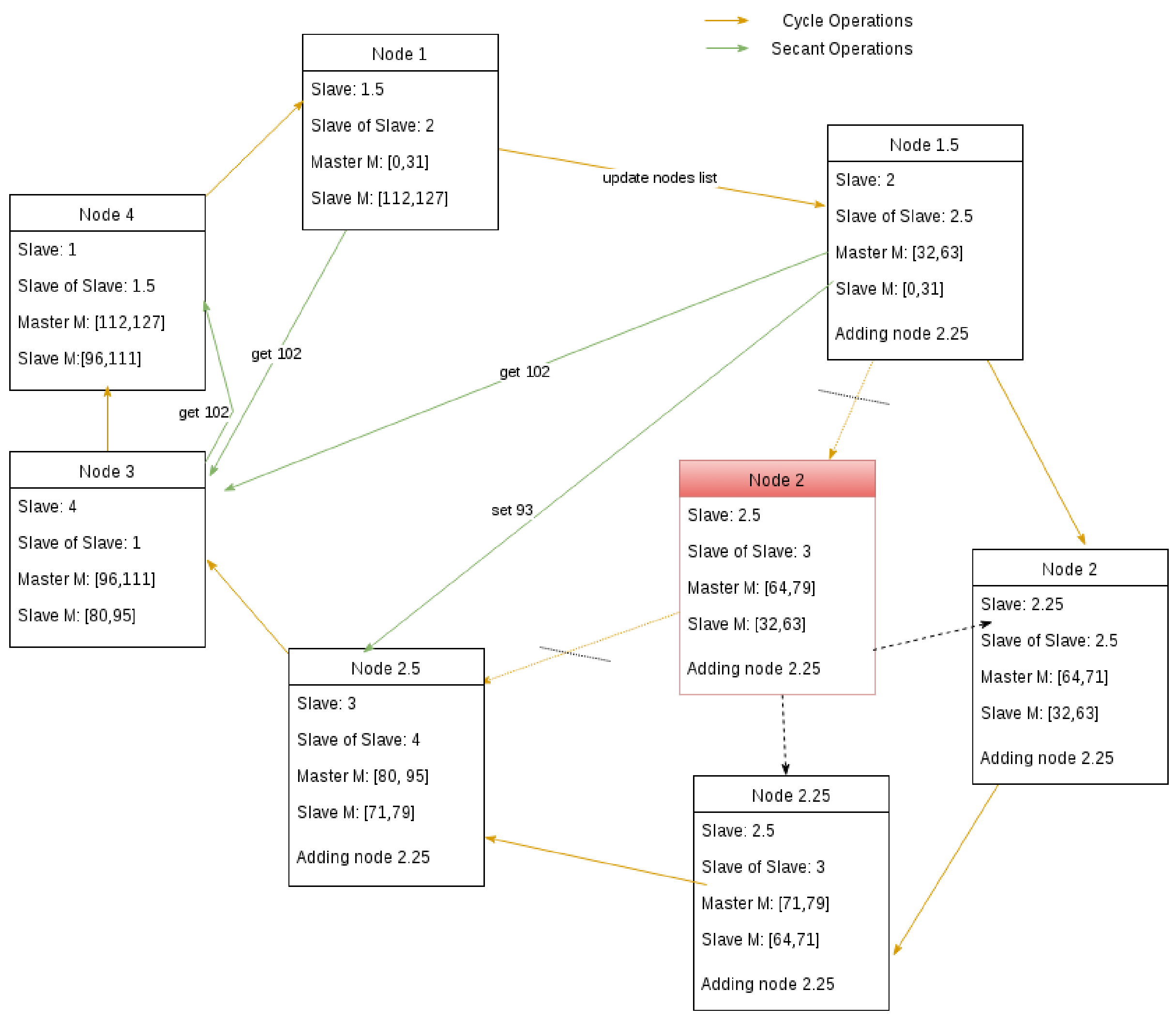
1	[0,31]	[112,127]
1.5	[32,63]	[0,31]
~~2~~	~~[64, 71]~~	~~[32,63]~~
2.25	~~[72, 79]~~ → [64, 79]	~~[64,71]~~ → [32, 63]
2.5	[80, 95]	~~[72, 79]~~ → [64, 79]
3	[96,111]	[80,95]
4	[112, 127]	[96,111]

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.

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.

Andrea Gennusa 28-08-2016 - updated to new requirements