Zeon: A Spatial KV Store

Darshan Kapashi (darshank@stanford.edu), Sagar Chordia (sagarc14@stanford.edu)

1. Introduction

Zeon is a spatial key-value store where the key is k-dimensional point, typically a location, with the property that nearby keys are mapped on the same node. As an object moves in space, its location changes frequently. To find nearby objects, we need to find all points close to the given point. This motivated the design of Zeon, to optimize it for frequent updates and heavy read queries.

A simple hash on the point might spread nearby keys to several nodes. Locality Sensitive Hashing might help to some extent, but because of the underlying probabilistic nature, we would still need to query (potentially) every node to answer a query, followed by an aggregation of the results. Content Addressable Networks solve problem of co-locating keys but due to lack of central intelligence, it does sub-optimal load balancing and replication. Zeon overcomes the shortcomings of CAN and DHT.

In Zeon the global key space is divided into set of disjoint rectangles and node is responsible for some set of nearby rectangles. If a hot spot is detected on some rectangle, Zeon will either transfer or split up the rectangle and assign it to a lesser-loaded node. It is suited better for an in-memory based workload (small values), but it could work for larger sized values too, with some tweaks. One can imagine applications such as a Taxi search (like UBER), a real time location service or nearby friends built using Zeon.

The paper is structured as follows. Section 2 talks about the API exposed by Zeon. Section 3 describes the Design of the various modules in the system. Section 4 exposes the details of the implementation. Section 5 will be evaluation. Section 6 shows how some example applications that can be built on top of Zeon. Section 7 lists some possible extensions.

2. API

The unit of data stored in KV is

```
struct Data { zeonid_t zid, Point point, Version version, string value}
```

The most frequently used methods (and the methods which are optimized for Zeon's design):

- `setData(data)`
  Take the Data object and store it in the system. The client may or may not wish to update the value associated with the key
- `getData(point)`
  Returns the values associated with this point.
- `getNearest(point)`
  The client can either ask for k nearest points, or ask for all points in a circle/square around this point

Some other infrequently used methods but still efficient to use:

- `createData(data)`
  Create an object with the given data. This must be called the first time that the object is created.
- `getPointsInRegion(region)`
  Return the points that lie in the given region

Methods which are supported but might be inefficient:

- `getData(zeonid_t)`
  Return the data corresponding to this id
- `removeData(zeonid_t)`
  Delete the data corresponding to this id

3. Design

Server

A server is a process running on a node. Each server is assigned a region (set of rectangles) by the leader and it acts as master for that region. Server also acts as a replica for region managed by other servers. Server processes the queries to zeon. There are 2 types of queries:

Reads

A server is eligible to receive a read request for any point. On receiving a read request, the server

- Check if server is the master or a replica for the rectangle containing the requested point, if not
  - it will send a SERVER_REDIRECT response to the client along with a list of servers it should contact next.
- If it is indeed responsible for the rectangle containing the point, or is replica for the region it will.
  - Read and fetch from local datastore
  - Contact servers containing nearby rectangles if needed or for consistency guarantees.

Such a design also enables us add caching when needed. Instead of the redirect, this server will itself fetch the data from the responsible servers and cache the response locally until some time to live (TTL). This way reads can be load balanced across many more servers than the master + replicas.

Writes

A write must go through the master of the rectangle that the point belongs to. (It will send a SERVER_REDIRECT if it is not the master). Furthermore, this master might be the new master associated with this zeonid_t, and some other server was previously responsible for this zeonid_t. On receiving a write, the server will do the following:

Case I: It was the master previously too	Case II: This zeonid_t had a different master
• Find the zeonid_t in its local datastore. • Write the value on reliable storage synchronously • Append the new point to reliable storage asynchronously • Update the in-memory data structures • Send asynchronous messages to the replicas, informing them of the new data • Acknowledge the write request	• Find the server that had the value previously • Fetch the value from that server • Do everything as in Case I • Send an invalidation message to the previous master and replicas. This essentially says, “delete this point from your data structures

Communication with the leader

A server periodically sends a heartbeat message to the leader. This message also contains its various statistics like memory usage, cpu usage, log file sizes, queries per second it is serving, etc. A server also awaits the leaders command to split or merge a region. That is, add or remove rectangles from its region (as a master or as a replica). It participates in a two phase commit with the master. On receiving a routing information update, it will:

- Figure out what data it is missing
- Use the old routing information and fetch this data from the servers that have this data
- Store this data in temporary data structures

When the server receives the commit message, it will then materialize the temporary data structures and start serving queries using these.

Leader

Leader acts as coordinator among various servers and maintains all the metadata. It keeps track of server to region mapping. It maintains and acts as ground truth for routing information. It collects heartbeat message, detects failure of node and assigns new master or decide to replicate more. It detects hotspot regions and does load balancing. Incase a busy node and free nodes are found then some region managed by busy node is split and merged into free node

Region split operation

A region may need to be split because of several reasons like uneven query load on different regions, server queuing the requests, failure of some servers etc. To accomplish this, the leader does the following:

- The leader periodically checks and finds a busy node and a free node based on thresholds.
- A region can be split in 2 ways. Either, the rectangles of a region can be transferred from the busy server to the free server, or a rectangle of the busy server can be broken into smaller rectangles. This decision is made based on the current load and such that the load will even out after the split.
- A 2 Phase Commit Protocol is used to transfer the ownership of the rectangles. In the prepare phase, the leader sends the new routing information to both the free server and the busy server.
    - Prepare (free server): The free server fetches all the points in the transferred region from the original server that was responsible for it.
    - Prepare (busy server): The busy server prepares to release ownership of the transferred region
    - Commit (free server): Once both the servers have acknowledged to prepare, the leader sends a commit message to the free server first.
    - Commit (busy server): After the commit to the free server is successful, then the leader can commit this transaction on the busy server.
- The leader updates the global routing table and broadcasts it to all other servers.

This order is chosen so that no region is unassigned to a node at any point of time. There is a possibility of a region being managed by more than one server, because of failures in the 2 Phase Commit. But, it should be transient since the leader will rollback the changes in a short time.

A periodic consistency check at the leader ensures that only 1 server is the master for a region at any time and in case of a conflict, the leader decides who will be the master based on its history of splits and merges.

Fault tolerance

An application will typically store the value once (the blob of data associated with the key) and update it infrequently (e.g. user profile), whereas the location is updated frequently. The system guarantees that the "*value*" part of the data will be written reliably and not lost across server crashes, but some points might be lost (although it is straightforward to make both parts of the data reliably stored if needed).

The leader uses the heartbeat messages from the servers to detect if a server has failed. Once a server is deemed to be dead, leader chooses an existing replica as new master server. The leader also checks the number of replicas alive for each region and in case it's less than a critical threshold then, then it will create more replicas. Replica servers can be chosen based on several factors such as load balance criteria, proximity to the master server, etc.

Client

The client library is a simple wrapper over the client-server API. It needs to take care of the following:

- For every zeonid_t, client cache the previous point that was used for the write request and send it along with every write request. We can omit the previous point, but it reduces the required inter-server communication when an object changes masters. By providing the previous point, we have a good guess of its previous master and can move the data in 1 request, as compared to several requests to first find its previous master and then transfer the data.
- On receiving a SERVER_REDIRECT, use the list of nodes returned by the server and attempt to re-send the request to these nodes

4. Implementation

Datastore

- This is storage system on each machine that the server module uses.
- It has a reliable storage medium (disk) which can be written to synchronously (flushed, reliably written) as well as asynchronously (no guarantees in case of failure)
- It also has some volatile storage (memory), which offers fast operations

We have use Apache Thrift for Remote Procedure Calls (RPCs). There are 3 programs that make this system: leader, server, and client. There are 3 interfaces in play: client to server, server/leader to server and server to leader.

Each server is listens on 2 ports, 1 for the client requests and 1 for requests from another server or the leader. It uses a threaded server, where each request uses it own thread. Several other threads are spawned to accomplish periodic tasks, like ping the leader, collect system statistics, or to enable the request to complete faster and queue up some operations to be executed later. The Datastore maintains read-write locks on each id. This way, write operations on 1 id, need not halt read operations on other ids. If the number of ids becomes large enough that the overhead of a lock per id becomes significant, these locks can be on id ranges instead of 1 id each.

get_value (zeonid_t)	When the master associated with an id changes, a server calls this on the previous master and transfers the data to itself
getDataForRectangle (rectangle)	When a server receives new routing information and has to handle new rectangles, it will call this method on the servers that were previously responsible for this rectangle and ask for all the data that server has
invalidate (zeonid_t)	When the master associated with this id changes, the new master will copy over the data and then ask the previous master to remove it from its datastore.
replicate (data)	When a server receives this command, it stores this data in its datastore
getNearestKByPoint (point, k)	When a client asks for nearby points, the master or the replica server scans its datastore, finds local points and may need to query the nearby regions for additional points. It uses this method to ask for points in nearby rectangles.
receiveRoutingInfo (routingInfo)	When the leader decides to split or merge a region, it involves the servers that are part of some data movement in a 2 Phase Commit. If that transaction succeeds, then remaining nodes need to have new routing information which is transferred via this API
prepareRecvNodeInfo (routingInfo)	When a server is a part of a split or merge, it needs to transfer over some data or fetch data from other servers. To do this reliably and reversibly in case some server fails to complete the data transfer, the leader engages these servers in a 2 Phase Commit. Upon receiving this call, the server fetches any data it needs to, stores it in temporary data structures and awaits the commit message. The server can continue serving queries meanwhile
commitRecvNodeInfo (routingInfo)	When the leader decides to split or merge a region, it involves the servers that are part of some data movement in a 2 Phase Commit. If that transaction succeeds, then remaining nodes need to have new routing information which is transferred via this API

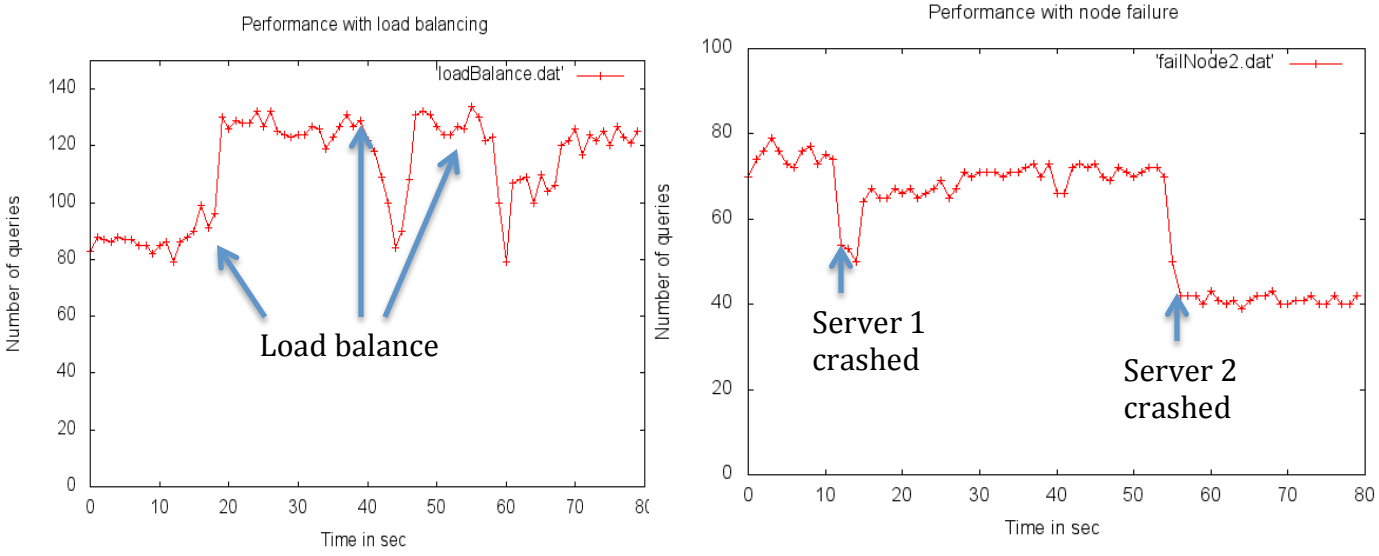
The interface to the leader is as follows:

initializeConfig (config)	A system administrator will manually call this method and provide the list of machines and the configuration of the system. Upon receiving this, the leader will partition the global id-space into regions and map servers to a region. This is a one-time call, which will initialize Zeon.
ping(nodeInfo)	Every server will periodically collect system statistics and sends heartbeat message to the leader.
getRoutingInfo()	On starting up, each server will use this method to receive the global routing information and start serving queries.

5. Evaluation

We have focused on getting the distributed systems aspect of the project correct and efficient. The in-memory structures to index and retrieve the nearest points are not optimized and hence the queries per second in the charts are low.

We build a toy application using Zeon's API. On clicking in the canvas, it relays the request to the backend and adds a point. In the next webpage, you can click and query for the 3 nearest points. Watch the demo here - https://www.youtube.com/watch?v=5qmyEwJ6MeQ.



Load Balancing

The experimental setup is as follows. Zeon is setup on 3 servers. Each server has 1 replica. Each server manages 1 chunk of the region. We add 80000 points (80000 createData requests) initially. There are 20 clients sending get_nearest() requests to the system in parallel. All the requests are targeted at 1 region managed by 1 server. Initially the system is answering about 90 QPS and when the leader triggers a rebalance, there is a small dip followed by an increase in the QPS. The leader has split the hot region into 2 and the queries are split across more servers. However, subsequent rebalances are not as effective because, the amount of inter-server communication is still the same.

Server Failures

The experimental setup is the same as above, but the leader is disabled. In this case the requests are randomly spread across the entire global region. On the first server failure, the system is still able to deliver almost the same QPS as before since the server that went down is replicated on 1 other server. However, on the second failure, the QPS drops sharply since 1 region is offline now.

6. Applications

1. Taxi calling service like UBER

Location of driver and customers are the points in our system. Each city can be thought as one region managed by a server. GetNearsetKPoints API is called on user application when request for driver is made.

2. Nearby people (Realtime location)

Each person is assigned an id when they get added to the application. Every N seconds, the client device sends an update on the id to Zeon. The client device can be any device, a mobile phone or a cellphone tower that is tracking every phone. To find how many people are within a region around you, the client can query using the getNearest() API. To find out how many of your friends are around you, the client queries for nearby points and filters out non-friends.

3. Ride Sharing service

Zeon would need to be tweaked a little for supporting such a service. Every point will be assigned a TTL and you can have multiple active points corresponding to an id at any time. Each trip can be assigned an id. The trip's starting location is set and the trip's end point is set. When someone wants to find a way to go from A to B, the client issues a getNearest()query to find nearby starting points and nearby ending points in that time range.

4. Tracking service

Lets consider an application where traffic cameras can accurately identify vehicles. Each traffic camera identifies the vehicle and sends its location to either Zeon or an aggregator, which passes it on to Zeon. This way you can scalably track lots of vehicles at any point of time.

7. Extensions

Functionality

- N-dimensional points can be supported in this infrastructure by using N-dimensional cubes in N-dimensional space as splitting regions.

Optimizations:

- k-d tree or some better in-memory data structure can be used to fetch nearby points on a single node.
- Instead of every node storing the global routing information, intelligent routing can be used to avoid transfer of entire routing information on each node and reduce network bandwidth
- Intelligent allocation of replicated servers to reduce the RTT time incase master node for some region fails. Similarly intelligent mapping of regions to nodes to ensure that nearby regions are on nearby servers.

Robustness

- Incase leader fails, zeon can still serve the queries. But the load balancing and maintenance of replication is affected until leader is alive again. Potentially a quorum of leaders can be used with Paxos to solve this problem.
- Since we use optimistic two phase commit, some nodes may have values for regions they are not responsible for. Garbage collector can be implemented which periodically scans all the key-values and removes invalid keys or values based on timestamp or TTL.

8. Conclusion

Zeon provides a convenient, scalable and fault tolerant infrastructure which can be used by systems managing spatial points or regions. It borrows ideas from some standard systems but is optimized to serve queries on spatial points.

It uses ideas from The Google File System, where a single leader manages the metadata and chunk servers are responsible for some shard of data. It uses some ideas from Content Addressable Networks where they map their keys onto a virtual k-dimensional space and each node is responsible for some part of that space. It is inspired by Dynamo, which provides a high write throughput and low latency by sacrificing consistency to some extent.

9. References

1. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System.
2. Giuseppe DeCandia et. al. Dynamo: Amazon's Highly Available Key-value Store.
3. Sylvia Ratnasamy et al. A Scalable Content-Addressable Network