Dynamo

Contributors

1. Juraj Matuš
2. Ondrej Vaško
3. Martin Dulovič
4. Kristián Košťál

Usage

Prerequisites

To build and run this application, you need to have basic docker tooling installed: * maven * docker * docker-compose * docker-machine * weave

Build & run

Download the repository and create docker machines:

```
git clone https://github.com/jurajmatus/dps-dynamo.git
cd dps-dynamo/docker-machine-weave
```

The default configuration is for the application to be split into two machines: * Master * service discovery * logging and monitoring service * Slave * Dynamo nodes

To build and run the master, run the following commands:

```
docker-machine create -d virtualbox master
# routes ARP requests from docker-machine to docker containers
echo 'sudo -i; echo 1 > /proc/sys/net/ipv4/conf/all/proxy_arp' | docker-machine ssh
master
eval $(docker-machine env master)
weave launch
eval "$(weave env)"
docker-compose -f master.yml build

# Run - still from the same shell
docker-compose -f master.yml up
```

To build and run the slave, run the following commands:

```
docker-machine create -d virtualbox slave
# routes ARP requests from docker-machine to docker containers
echo 'sudo -i; echo 1 > /proc/sys/net/ipv4/conf/all/proxy_arp' | docker-machine ssh
master
eval $(docker-machine env slave)
```

```
weave launch $(docker-machine ip master)
eval "$(weave env)"
sh ../key-value-store/dropwizard/build.sh
docker-compose -f slave.yml build

# Run - still from the same shell
docker-compose -f slave.yml scale key-value-store=2
```

Cleaning Master

```
# In other shell than it was deployed in
docker-compose -f master.yml rm --all
weave stop
weave reset
eval "$(weave env --restore)"
```

Cleaning Slave

```
docker-compose -f slave.yml scale key-value-store=0
weave stop
weave reset
eval "$(weave env --restore)"
```

Setup Host and test

```
ip r add 10.32.0.0/12 dev vboxnet0  # adds route to Weave network from host computer
curl $(weave dns-lookup consul-server | head -n1):8500/v1/catalog/nodes | python -m json.tool
curl $(weave dns-lookup consul-server | head -n1):8500/v1/health/service/dynamo | python -m json.tool
curl $(weave dns-lookup haproxy | head -n1):8080/check_connectivity
#open logging in web browser: firefox http://$(weave dns-lookup logging-server | head -n1)/login, firefox http://$(weave dns-lookup logging-server | head -n1)/loganalyzer
```

API

Firstly you need to know an address of application's end-point. All addresses listed will be relative to this:

```
weave dns-lookup haproxy
```

Get

URL: /storage/{key}?minNumReads={minNumReads}

Method: GET

Parameters:

- *key* : String - BASE64 encoded byte array
- *minNumReads* : Integer - minimal number of replicas to acknowledge the request so that response could be sent

Response body:

- *key* : String - BASE64 encoded byte array
- *value* : Object
    - *version* : String - version string
    - *values* : Array[String] - BASE64 encoded byte arrays, one for each unresolved version

Put

URL: /storage/

Method: PUT

Content-type: application/json

Body:

- *key* : String - BASE64 encoded byte array
- *value* : String - BASE64 encoded byte array
- *fromVersion* : String - version string, exactly as received from the last get, or empty
- *minNumWrites* : Integer - minimal number of replicas to acknowledge the request so that write was considered complete

Response body:

- *success* : Boolean - success of operation. Will be false if old value of *fromVersion* is used, or if any other error occurs

Delete

There is no method to specifically delete an entry. To do so, Put method where value is empty string has to be issued.

Infrastructure

Service discovery

Proxy and load balancing

HaProxy with Consul Template

Logging

For distributed logging we use a central rsyslog daemon collecting the log entries from all hosts.

Each node then runs a local rsyslog that listens to application and pushes the logs to the central rsyslog over UDP in batches. That's achieved by using a queue. To handle temporary connection failures, daemon is configured to attempt retry if the send fails.

To view and filter messages we use web based front-end loganalyzer.

Metrics and monitoring

We use Graphite to collect metrics. Those are then viewed in Grafana.

Producing and sending metrics in an appropriate format is handled by Dropwizard metrics. It automatically generates various JVM performance metrics and allows to configure custom metrics in many representations, like counters, timers, histograms, etc.

Orchestration

Application

The main Dynamo application runs on top of Dropwizard framework.

Underlying systems

Http requests are handled by server Jetty and REST framework Jersey. All requests are handled asynchronously, using JAX-RS @Suspended annotation.

To store the data belonging to the node we use Berkeley DB.

For asynchronous messaging we use ActiveMQ. Every node runs an embedded instance.

Implementation

Partitioning

Values are partitioned based on their key. The key is hashed via **md5** function and mapped into consistent hash space of 64 bits.

Versioning

Each value has a version string associated with it, which is internally a **vector clock**. Vector clock contains up to 10 entries of node's ip address, version number and timestamp. Timestamp is not used in conflict resolution algortithm, it only serves to find oldest entries when trimming is performed.

Execution

Upon receiving of a request, the coordinator for the key is computed. If the receiving node is not responsible for it, the request is **redirected** to one of the responsible nodes via http protocol.

If the node is responsible, **request id** is generated for tracking and **state machine** is created for the request. All subsequent operations are then offloaded to **message queue** workers.

Nodes responsible for the key are computed and contacted via message queue to either replicate (PUT) or provide the value (GET). Version resolution is performed before writes and after collecting all reads. Based on the result of it, either the value is used as is, is merged or is rejected.

Internaly reads and writes are provided by BerkeleyDB. Reads are done non-transactionally. Writes are done **optimistically** - read is done, operations like version resolution are done, and then the value is written if change hasn't been done in the meantime. The last read-write is done in one transaction. If write fails, the whole sequence is repeated. This strategy was chosen to provide fast reads, but at the expense of possibility of failed writes.

Upon receiving the **acknowledgement** number *w* or *r*, response is sent back to the client. Timeout is checked. If it elapses, the topology is recomputed and new responsible nodes are contacted.

Upon receiving the acknowledgement number *n*, state machine is discarded from internal storage. If timeout is exceeded, http error code is sent back to the client.

Handling failure

TODO: temporary, permanent

Membership and failure detection

TODO

Adding and removing the nodes

TODO