Rocinante

Mustafa Bayramov
CS244B project
mbayramo@stanford.edu

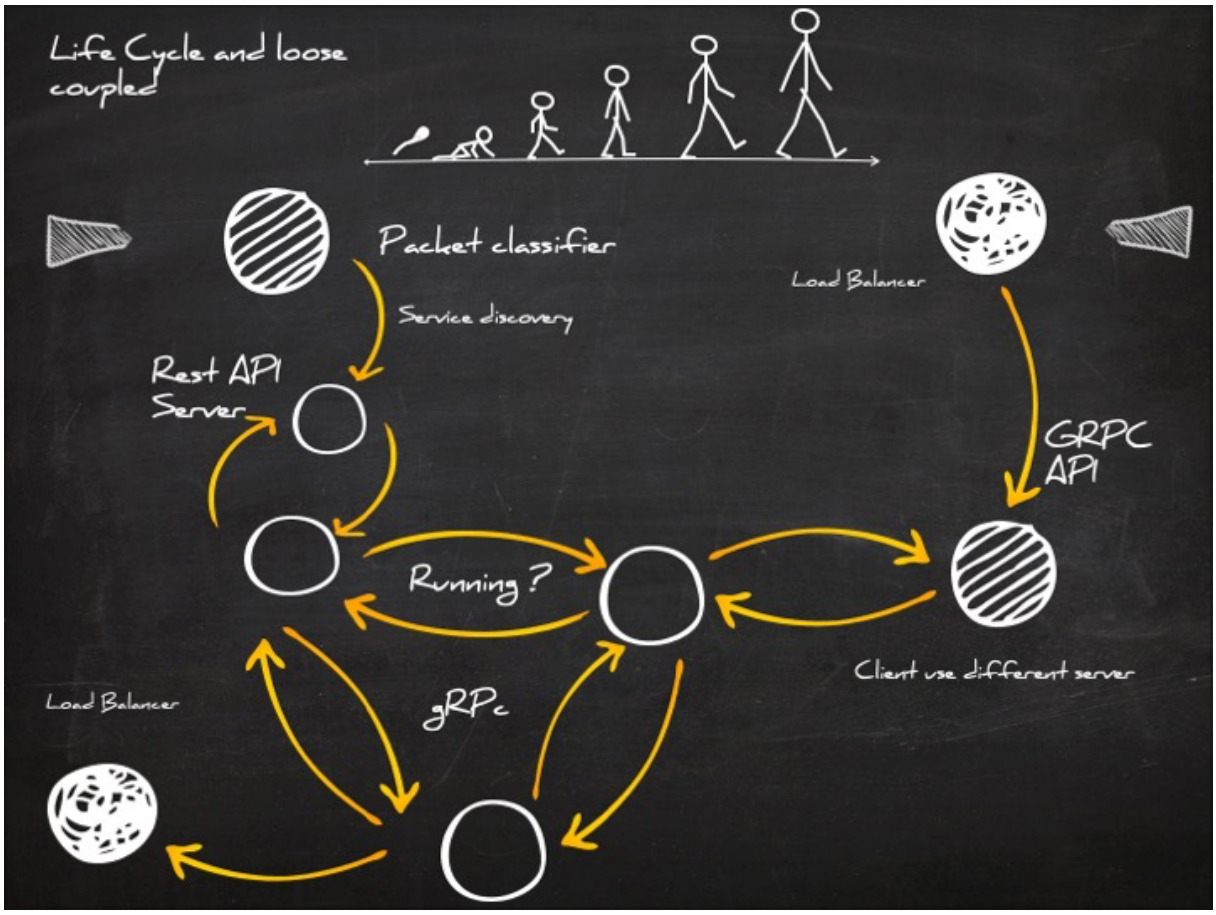
Initial proposal statement

The general idea and proposal have a much broader scope. For example, in many CNF deployment, one of the essential components of the Kubernetes system, ingress controller. There are several open-source projects such HA load balancer etc.) that sole role in figuring out a target worker node. Traditionally, load balancer decisions based on mapping public IPv4/IPv6 endpoint address (service) to address of worker node via a set of policies. That later translated by worker node to actually to address container runtime.

There is another set of use cases that primary job provide load balancer capability for network service. There are several deployment scenarios that we can primary different as in-line and out of the box. Essentially, in the first case, load balance usually deployed in the direct path between an external pubic network such as the Internet and server farm. The load balancer maintains state information about each client and each connection to a server farm. Another deployment scenario network element such a router that is in direct path perform redirect function based on 5-tuple or 7-tuple. In many cases, that operation must be symmetrical since the server farm usually uses none routable IP blocks.

In all this case for a given flow information load balancer must choose same server from pool of servers.
Load balancer can't assume that actually server stateless, and in most of the cases server create a state information.

Last set of use cases, flow classification, for example, in many systems that perform deep packet inspect, one of the first steps in pipeline is TCP/UDP flow classification. For instance, if a service provider offers parental control service for mobile customers or internet service offers to a customer. In most basic form , for HTTP and HTTPS protocol packet, usually analyzed for content and consulted an internal database that categorized each domain by categories, for other protocol systems need to perform sophisticated heuristics. In many cases, classification needs sample N number of a packet to determine a protocol type, since some protocols, use encryption and b obfuscate itself. The most notorious and well known system is great Chinese firewall.



High level

The primary objective of the Rocinante project provides a distributed system and API abstraction on top to
support TCP/UDP flow state synchronization between different data centers and cluster members. In essence,
it allows the developer to leverage REST or GRPC API for an application status of the application.
There is many use case under that fall category, Firewall TCP/UDP state, DPI classification state,
load balancer server selection hash, heartbeat liveness monitoring for the server farm.

For example, if we have a set of load balancer that need synchronize that current state.
The system supports traditional distribute key-value storage, but the primary motivation provides an
abstract and API layer that can be easily consumed by a different application
that requires fast synchronization.

There are two application written on top show case capabilities.

- A load balancer that serialize source hash selection. For example if load balancer chosen server A from server farm pool for a given client. It will serialize decision to cluster any other load balancer will do a check and if given client already in cluster load balancer
  B and C will select same server from server farm pool.

- A packet sniffer that uses libpcap framework in order to capature raw stream and serialize to a cluster. Current implementation support standard pcap type of filters.

Overview.

Rocinante's system consists set of controllers node that forms a cluster. A cluster provider the capability
to store key-value a data that eventually consistent among all cluster members, and a REST interface (server and client)
client that can serialize data, via REST API interface. The system provide RERT and gRPC client that develop
can leverage.

- In order cluster maintain synchronization and consensus cluster implements RAFT protocol.

- The communication between cluster member done via gRPC interface and model via protobuf.

- The initial leader election protocol follow RAFT specification.

- Currently, system doesn't snapshot define in section 5.3 protocol spec, but it s something to be added

- Rocinante provide capability to serialize metric and instrumentation serialize to Prometheus
  system. At Moment, it serializes Vote Request,Respond , Append Request/Respond, client submit
  request, committed data , go routine and memory monitoring.

- All data pushed to Grafana for monitoring and instrumentation purpose.

Initial protocol.

The implementation follows a RAFT recommendation and split consensus model to set of sub
problems.

- Leader Election
- Log Replication

During a start up phase a system reads a configuration specification. The server specification
consists a spec for each node in the cluster. For example spec consists of IP and port number pair
for each node in the cluster, the REST api interface , that also shared with a build-in web server, that provide
simple UI dashboard interface, a metric server that uses separate
build in web server.

At the moment, both the REST API server and the metric server runs as separate goroutine of the main server.
Another approach and idea de-couple both from the main server and introduce different grpc semantic to
communicate internally. This case has one major drawback that it might potentially would require additional
synchronization on the server-side. All counter for metric and instrumentation are atomic, but since its single
client prometheus, it doesn't require synchronization.

A Rocinante will automatically allocate internal server id for each member node. The current scheme
uses hash(IP:PORT) and generates a 64-bit identifier for each node. For example if each server bind
on 0.0.0.0:xxxxx then hash is deterministic.

In current implementation the cluster communication between each member done via gRPC transport

protocol, and it supports generic binding via protobuf. In current todo list add C client and introduce
intel DPKD flow classifier app.

*Below example of configuration if we want run 3 instances on localhost.

```
artifact:
  cleanupOnFailure: true
  cluster:
    name: test
    controllers:
      - address: 127.0.0.1
        port: 35001
        rest: 8001
        metric: :2112
        wwwroot: /Users/spyroot/go/src/github.com/spyroot/rocinante/pkg/
template/
      - address: 127.0.0.1
        port: 35002
        rest: 8002
        metric: :2113
        wwwroot: /Users/spyroot/go/src/github.com/spyroot/rocinante/pkg/
template/
      - address: 127.0.0.1
        port: 35003
        rest: 8003
        metric: :2113
        wwwroot: /Users/spyroot/go/src/github.com/spyroot/rocinante/pkg/
template/
```

prometheus.yml

```
scrape_configs:
  - job_name: rocinante
    scrape_interval: 10s
    static_configs:
```

```
    - targets:
        - localhost:2112
        - localhost:2113
        - localhost:2134
```

After reading a configuration and deciding what port to listen, For example, if we want to run a Rocinante
on the same server for debug purpose. Rocinante will automatically check port allocation and bind each server
instance to a respected TCP/UDP port.

After server finish initial configuration it will move itself to Follower state based on protocol specification.
If cluster already stable state than a node will remain in same Follower state and will receive messages
from a leader.

RAFT specification indicates that at any given time, each server must be in one of three states: leader, follower,
or candidate. Rocinante adds additional state to a protocol. The primary purpose signal to a network that server
is not ready, the motivation behind that two use cases when we want gracefully shut down a server or
shutdown a GRPC or any other external interfaces.

During Shutdown or init state server will not accept any messages. Note we need to differentiate a shutdown state
for gRPC interface and overall entire server shutdown. In the first case, shutdown gRPC provides a capability
to shutdown gRPC only meanwhile respond on all REST API call that doesn't require to store data. For example,
monitoring, or simulate partition. For example, one unit test cover case if we Shutdown a leader's gRPC interface
that triggers a new leader election that leads to a situation that new leader elected and host that has Shutdown
gRPC state remain in leader state. Later we enable gRPC back, and that creates a split-brain scenario and
partition cluster unit test.

In the second use case, if the server needs to perform initial IO operation that require a significant amount of time,
for example during initial boot - start up time, it can't accept can't accept communication.

Rocinante supports variable timers and generally observation with the semantics described in the original paper,
related explicitly to heartbeat timer. If a network provides stable connectivity, then the same node will remain
a leader for a very long time, since there is no need to re-elect a leader during a stable state.

Rocinante adds added additional options to randomly and none deterministically change a leader by delaying heartbeat
messages for sufficiently long time so other node force re-elect a new leader.

As it mentioned, each server communicates using remote procedure calls and implement RAFT spec that defines
two types of RPCs. RequestVote RPCs are initiated by candidates during elections and Append-Entries
RPCs are initiated by leaders to replicate log entries and to provide a form of a heartbeat.

Below protocol, spec defined for GRPC. Note that in Rocinante log entry and command modeled as key-value pair.
Since the go map data structure and interface provides idempotent, we can replay the log for the same key,
value pair, and that will guarantee the most recent update applied and serialized to stable storage.
Note that guarantees eventual consistency. If one of the servers behind it will eventually catch up.

```
message LogEntry {
    uint64 Term  = 1;
    KeyValuePair command = 2;
}

message KeyValuePair {
    string key = 1;
```

```
    bytes value = 2;
}
```

```
message PingMessage {
    string name = 1;
}

message PongReply {
    string message = 1;
}

message RequestVote {
    uint64   term = 1;
    uint64  candidateId = 2;
    uint64   LastLogIndex = 3;
    uint64   LastLogTerm = 4;
}

message RequestVoteReply {
    uint64 Term = 1;
    bool  VoteGranted = 2;
}

message AppendEntries {
    uint64    Term = 1;
    uint64    LeaderId = 2;
    uint64    PrevLogIndex = 3;
    uint64    PrevLogTerm  = 4;
    repeated  LogEntry entries = 5;
    uint64    LeaderCommit = 6;
}

//https://stackoverflow.com/questions/43167762/how-to-return-an-array-in-
protobuf-service-rpc
message AppendEntriesReply {
    uint64 Term  = 1;
```

```
    bool  Success = 2;
}

message SubmitEntry {
    KeyValuePair command = 1;
}

message SubmitReply {
    uint64 LeaderId = 1;
    uint64 NodeId = 2;
    bool   Success = 3;
    string Address = 4;
}

message LogEntry {
    uint64 Term  = 1;
    KeyValuePair command = 2;
}

message KeyValuePair {
    string key = 1;
    bytes value = 2;
}

message CommitEntry {
    bytes command = 1;
    uint64 Index = 2;
    uint64 Term = 3;
}
```

Rocinante gRPC interface uses none blocking semantics to communicate between each node in a cluster,
and between client servers, Rocinante also leverage separate goroutine and concurrency for heartbeat channel,
internal and external communication to external clients, internal communication for commit channels.

In the current list of must do, I have the plan to add an ingress buffer channel to absorb a

small amount
of gRPC messages. Essentially, during transmit and receive routine, server doesn't hold the lock to process
RPC message as quickly as it can, but as soon as the server start processing, it must hold a lock.
So one idea creates a buffered channel to sink RPC message and marginalize blocking during event
processing.

During initial handshake, voting procedure or delayed or partial communication that triggers the election process,
server declares itself a candidate begin a concurrent communication to other peers.

##Log and Storage

At current state, system support in-memory storage that provides fast O(1) access to key value pair
or persistent storage interface. The current semantics doesn't use an optimized IO layer and
leverage gob library to serialize data to persistent storage.

I've tested gob and serialization to stable storage but doesn't provide adequate performance and pretty meaningless
without optimization. One idea introduces the LSM type of data structure for a permanent log.

Rocinante adjusted the original a log format and replaced original command with key-value pair instead.
In original RAF semantics in The leader appends the command to its log as a new entry, then issues
AppendEntries RPCs in parallel to each of the other servers replicate the entry. When the entry has been safely
replicated (as described below), the leader applies the entry to its state machine and returns the result of that
execution to the client, from here we can observe that we can apply the same semantic for any tree-based data
structure since the entire process is deterministic and same key and value can be applied simultaneously on N

number of the server since the entire point of consensus to have common agreement on value.

API and load balancer.

*Each node in cluster provider REST API and gRPC interface.

List of Rest call

- /leader provide capability to discover a current leader
- /shutdownGrpc shutdown grpc interface
- /shutdownNode/nodeid - shutdown entire server
- /log - responds with entire log as json, additional value size can be passed to get portion of a log
  last 10 etc
- /commited - responds with list of commited record to stable storage. note that log/ commited can be requested
  from any server and it useful property for unit testing where we
  can simulate different fail condition and compare log and commited data.
- /get provides capability to get a value for a given key
- /flows/{size}/{id:[0-9]+}" provide capabiloty to get given hash flow, hash can be 5 or 7 tuple serialized
  by a client.
- /peer/list responds with list of all peer for a given server and status of gRPC connection, it also serialize
  all server spec. REST end point / GRPC. This call used by client to auto discover cluster.
- /flows - responds with entire list of flows
- /role - responds with current role of server.

- /size size of log

Cluster discovery

Each node in the cluster responds to a subset of REST API that doesn't require a cluster leader role. For example,
since all cluster members form a full mesh of communication, we can observer the status of socket communication
from any node in the cluster. Note GPRC also provides a semantic where the client side of GRPC will automatically
reconnect. That way, Rocinante never remove client peer from the list of all peers.

So the server always knows a total number of peers in cluster and the number of peers with a stable ready state
connection. For example in steady-state if we check each node in a cluster, we will see that all nodes connected
in full mesh.

The same if two out of five nodes will disappear and partition a cluster, we will see that two clients connected
to two other peers are disconnected as well. We can use this property due to the nature of bi-directional
communication. We can also use heuristic on client side and detect partition case. For example if client see
that two out of five peers connected.

*Each node regularly updates a leader cache upon arrival RPC message. Note that leader ID consulted with the
state itself. Rocinante provides a rest API client that encapsulates an API interaction; during initial communication,
the client might not necessarily know about all peers in cluster nor assume about a current leadership role.

During REST API client object creation, the API rest-client uses Node Discovery, to update or retrieve the current
leader node endpoint. It issues the REST API call to discover who is a leader of the cluster, rest api server end point.

As part of the initial handshake, the client determines a REST API endpoint that requires API communication based
on the initial sequence. As part of discovery, the client also gets the full status of all peers. So in the case of a partial communication inside a cluster, the client can observe disconnected nodes.

The primary motivation is to minimize and reduce the client-side configuration required for each client.
It should also be sufficient to re-point a client to any IP address of a node in thee cluster.

Meanwhile, Rocinante leaves to the implementer of application logic optimization related to the number
of interactions to a server. One example, the client might cache the existing leader id that will minimize
the number of calls to a server and round trip, and the client can reset the leader ID only when the node
responds that is not a leader anymore or node status changed, or election term changed.

Load balancer App

Currently, it implements two algorithms—standard round robbin mainly used for testing purpose and source hash selection.

The source hash algorithm used to serialize hash to the Rocinante cluster,
during a request, load balancer hashes client requests and use as a key entry for server selection. The hash value
serialized, other load balancers upon a subsequent request first check request hash in a cluster.
For example, if N number of the load balancer used as anycast address, all load balancers will deterministically
forward traffic to the same server for the same remote client. The value for a hash resolve to a target server
allows all load balancers to choose the same server from the pool for the same client and provides stickiness.

Below is a configuration that the load balancer will read.

There is are three sections. The first section describes what VIP ports to listen, timeout, etc. and other global
property for a load balancer. The server section defines the entire server farm that the load balancer will use for a
given VIP, and it will continuously monitor and probe each server.

API section describes the Rocinante cluster. Note that the rest-client automatically discovers a leader and requires
only one server. But in case if the server disconnected and the client still needs to know all cluster members,
it requires a partial list of other members.

For example:
If, during the initial setup client successfully discovered a leader and all members of the cluster.
Later, if the leader disconnected due to partition, etc., the rest-client will find a new leader because it has an
entire list of all members.

If we indicate that only one server and that server never responds to the client's initial request, the client will
not be able to find a leader—that way. We need to mention a majority of servers so we can discover all other members.

Note that the client doesn't check the current leader on each request. It was done as an optimization to
reduce the round trip over a network. It will discover a cluster only server will reject a request if the target
server not a leader of a cluster.

```
pool:
    name: test
    bind: 0.0.0.0:9001
    # we indicate only one ip and let rest client discover cluster leader
    api:
      - address: 192.168.254.48
        rest: 8001
        grpc: 35001
```

```
      - address: 192.168.254.48
        rest: 8002
        grpc: 35002
      - address: 192.168.254.48
        rest: 8003
        grpc: 35003
    servers:
      - address: 192.168.254.48
        port: 8887
      - address: 192.168.254.48
        port: 8888
      - address: 192.168.254.48
        port: 8889
```

Example of entire configuration

```
artifact:
  cleanupOnFailure: true
  cluster:
    name: test
    controllers:
      - address: 127.0.0.1
        port: 35001
        rest: 8001
        wwwroot: /Users/spyroot/go/src/github.com/spyroot/rocinante/pkg/
template/
      - address: 127.0.0.1
        port: 35002
        rest: 8002
        wwwroot: /Users/spyroot/go/src/github.com/spyroot/rocinante/pkg/
template/
      - address: 127.0.0.1
        port: 35003
        rest: 8003
        wwwroot: /Users/spyroot/go/src/github.com/spyroot/rocinante/pkg/
template/
```

```
  pool:
    name: test
    # we indicate only one ip and let rest client discover cluster leader
    api:
      - address: 127.0.0.1
        rest: 8001
        grpc: 35001
    servers:
      - address: 127.0.0.1
        port: 8887
      - address: 127.0.0.1
        port: 8888
      - address: 127.0.0.1
        port: 8889
  global:
```

##Usags

- First we start prometheus server.

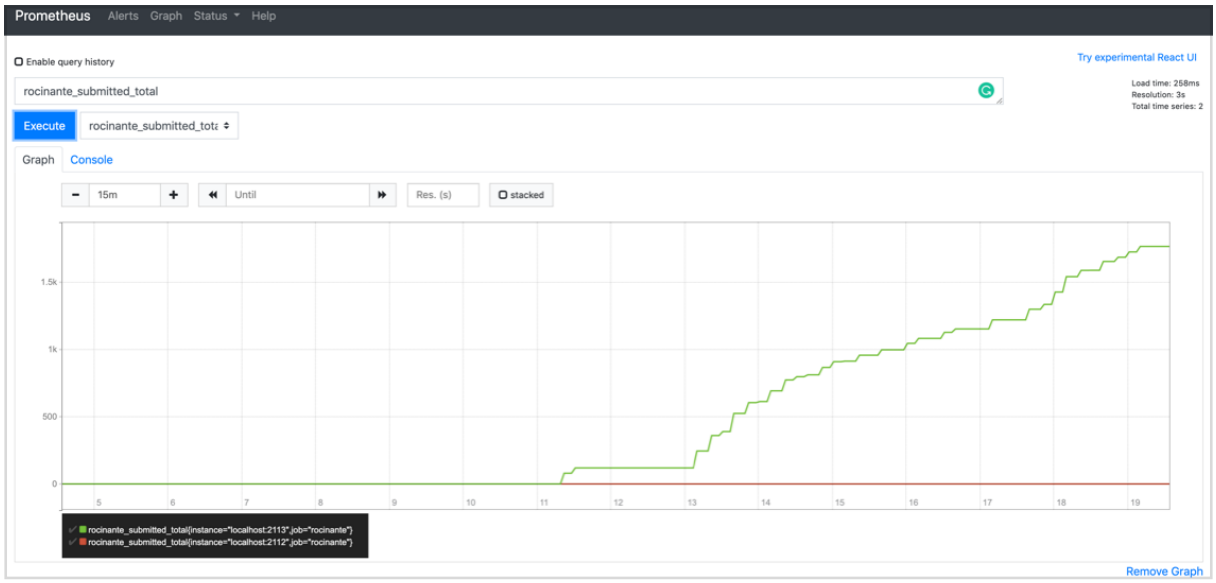
```
    #shell prometheus –config.file=$(GOPATH)/go/src/github.com/spyroot/
rocinante/prometheus.yml

    Than we can start all server.
   ```
    shell# ./rocinnante /Users/spyroot/go/src/github.com/spyroot/rocinante/
config.yaml
   ```
    Note you can start all server on same host but make sure you have different
ports.

    We can check metric directly.  This metric server used by prometheus
   ```
    #shell curl http://localhost:2112/metrics
   ```
    We can check build in web server open http://localhost:8001
```

Example metric

```
curl http://localhost:2112/metrics
rocinante_append_rx_total 901
# HELP rocinante_append_tx_total The total number of append tx events
# TYPE rocinante_append_tx_total counter
rocinante_append_tx_total 108
# HELP rocinante_committed_total The total number of vote tx events
# TYPE rocinante_committed_total counter
rocinante_committed_total 0
# HELP rocinante_submitted_total The total number of submit request events
# TYPE rocinante_submitted_total counter
rocinante_submitted_total 0
# HELP rocinante_vote_rx_total The total number of vote events
# TYPE rocinante_vote_rx_total counter
rocinante_vote_rx_total 27
# HELP rocinante_vote_tx_total The total number of vote tx events
# TYPE rocinante_vote_tx_total counter
rocinante_vote_tx_total 9
```

Metric pushed to prometheus

We can monitor number of RPC message (Append / Vote) monitor in real time convergence.



Sniffer usage.

Note you need root user. I've tested Mac OS and linux. Make sure libpcap installed.

```
 sudo ./sniffer --config /Users/spyroot/go/src/github.com/spyroot/rocinante/
config.yaml capture en0
```

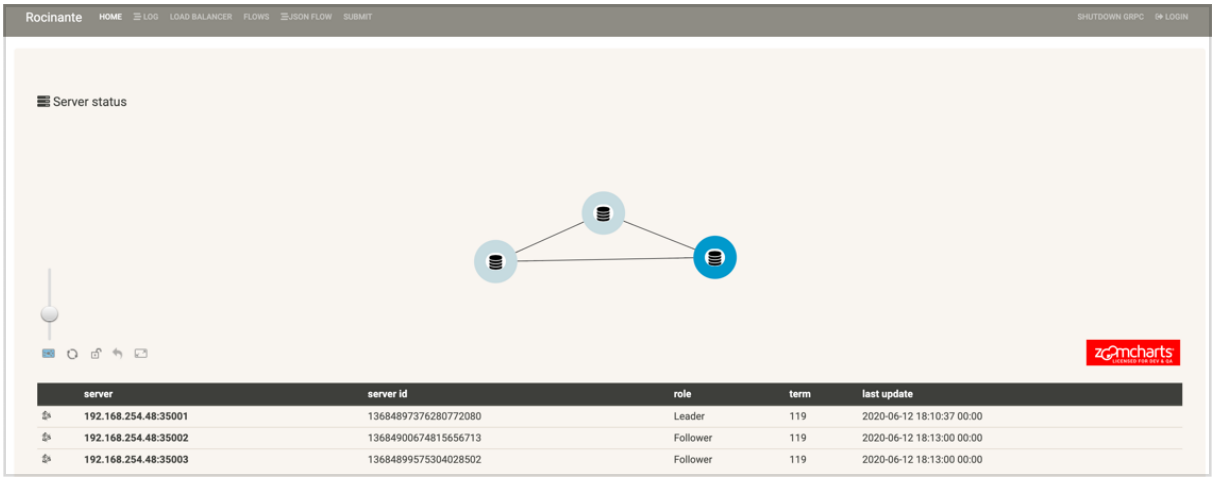
When you start a sniffer, you should see sniffer serialize everything a cluster.

```
I0610 15:51:29.069630    8174 restclient.go:230] Sending request
cluster req [http://192.168.254.48:8002/submit/MTEzNDczMTMxNjYwNTU4ODc2ODU=/Df-
DAgEC_4QAAf-CAABH_
4EDAQEDSGRyAf-
CAAEFAQdTcmNQb3J0AQwAAQdEc3RQb3J0AQwAAQVTcmNJcAEMAAEFRHN0SXABDAABBVByb3RvAQwAAA
Bq_4QAAgEFM
zk1NjABBDgwMDIBDjE3Mi4xNi4xNDkuMjIzAQ4xOTIuMTY4LjI1NC40OAEDVENQAAEEODAwMgEFMzk1
```

```
NjABDjE5Mi4xNjguMjU0LjQ4AQ4x
NzIuMTYuMTQ5LjIyMwEDVENQAA==] cluster leader [192.168.254.48:8002]
```

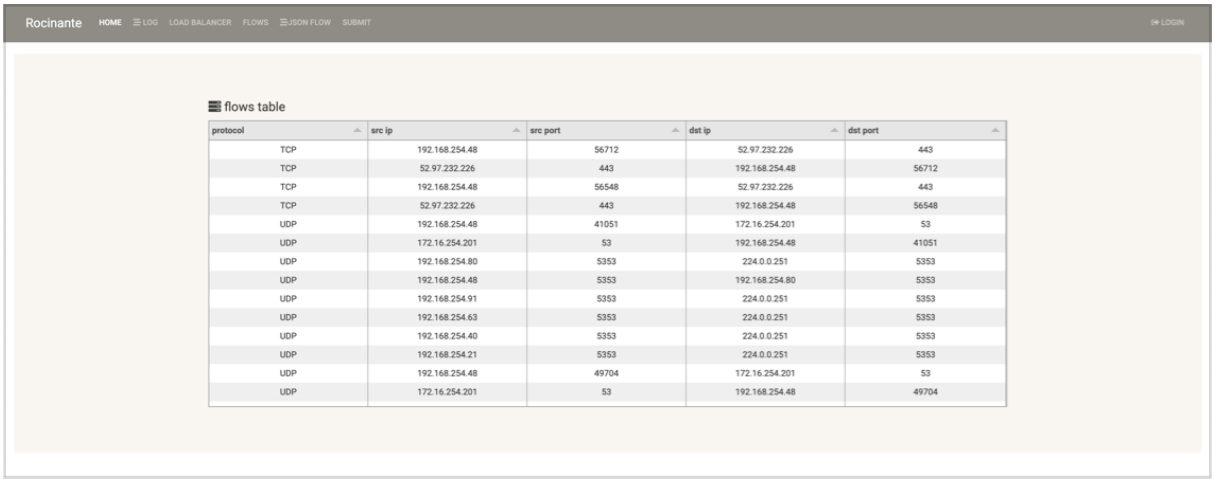
Screenshots

The web server automatically discovers all nodes. We can shutdown entire server or gRPC interface.



Classifier flows

All flow pushed from test sniffer app.



Roadmap

- Add integration with DPKD and Classifier and AF_XDP. Specifically port grpc client via C binding and create capability to leverage DPKD classifier

- gRPC stream interface. Specfcally create indirection that will represent a key as descriptor and allow client to write / read via IO interface.
- LSM for storage. Replace precision storage with LSM .

- Abstraction what we can store. i.e serialize radix etc tree as raw array.. For example a key is root tree, the value serialized radix tree

- be able to O(1), log time lookup for longest match

- Optimization submit RPC. (Create event driven approach)

- Evaluate pre-vote semantics. RAFT has two additional proposal evaluate option.