Voltha Architecture in a clustered HA configuration

Sergio Slobodrian, Ciena

CORD Build Wed, November 7th, 2017

ciena
Experience. Outcomes.

CORD
Central Office Re-architected as a Datacenter

ciena
Experience. Outcomes.

Abstract

This talk will dive in on VOLTHA's clustered high availability architecture including load balancing. The talk will focus on each of the minimally required containers, their role, and interactions. This will include Consul, Kafka, Envoyd, Envoy, Fluentd, OfAgent, and last but not least the VOLTHA core. For completeness we'll also quickly skim over grafana, dashd, and shovel. We will also work through several example API calls through the VOLTHA stack and how they are handled by each of touch points along the way.

High Level Diagram

ONOS Apps

Northbound services

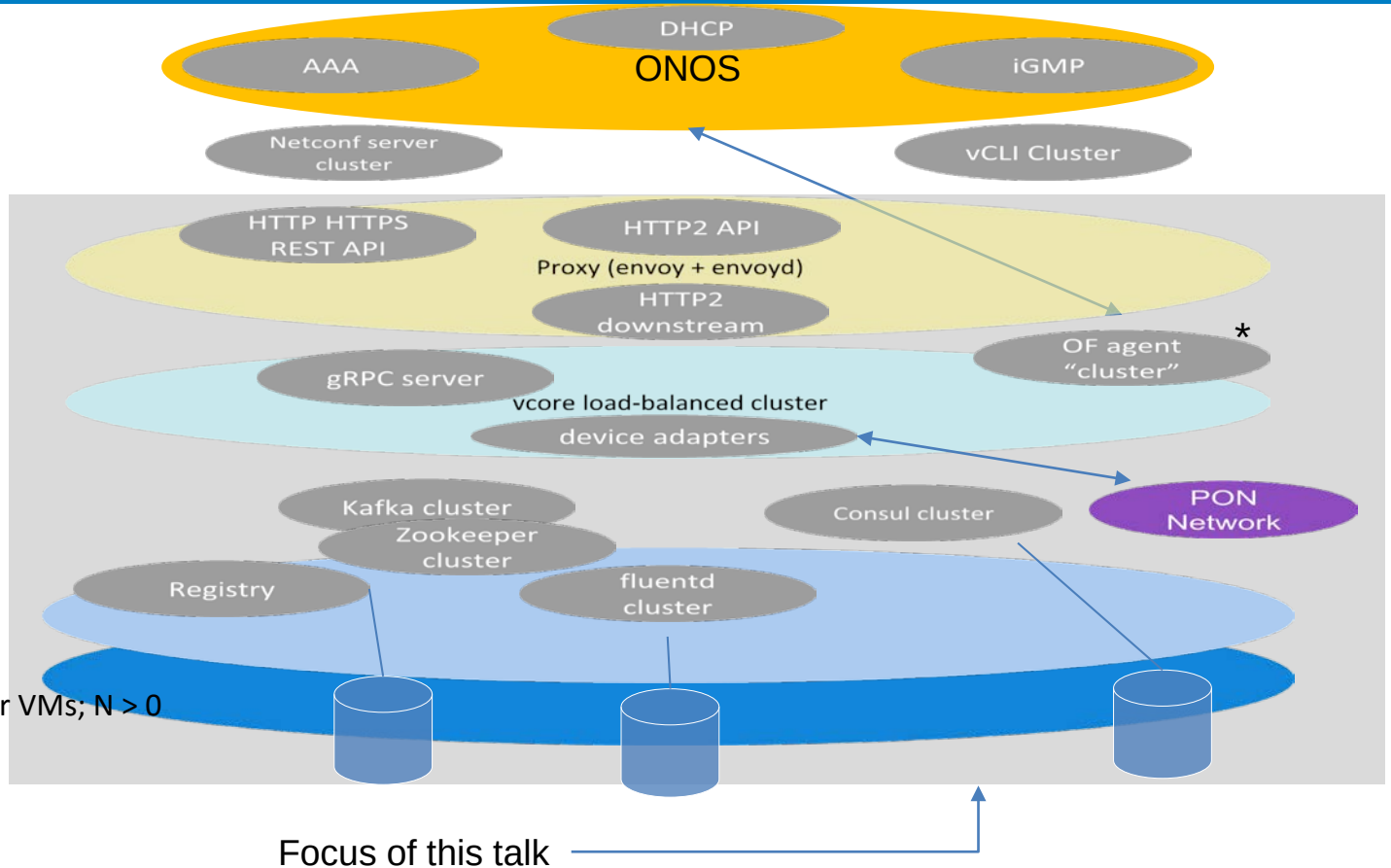
Load balancing proxy

vOLT-HA cluster

Support Services

Docker Swarm Mode

2N+1 Hardware Servers or VMs; N > 0



Focus of this talk

High Level Concepts

- Docker in swarm mode is used as the platform
  - 3 overlay networks are used to support the application
    - voltha_net, kafka_net, consul_net
- Each service is independently started and clustered (individual compose files)
  - Facilitates individual scaling to expected load
  - Allows for service specific optimizations
- All services are run in load-balancing clusters.
- Number of servers (or VMs) underlying the cluster is 2N+1; N>0.
- NOTE: Currently HA is more of a fast-failover but evolving to true HA with mSec failover.

Top Down Review of the Architecture

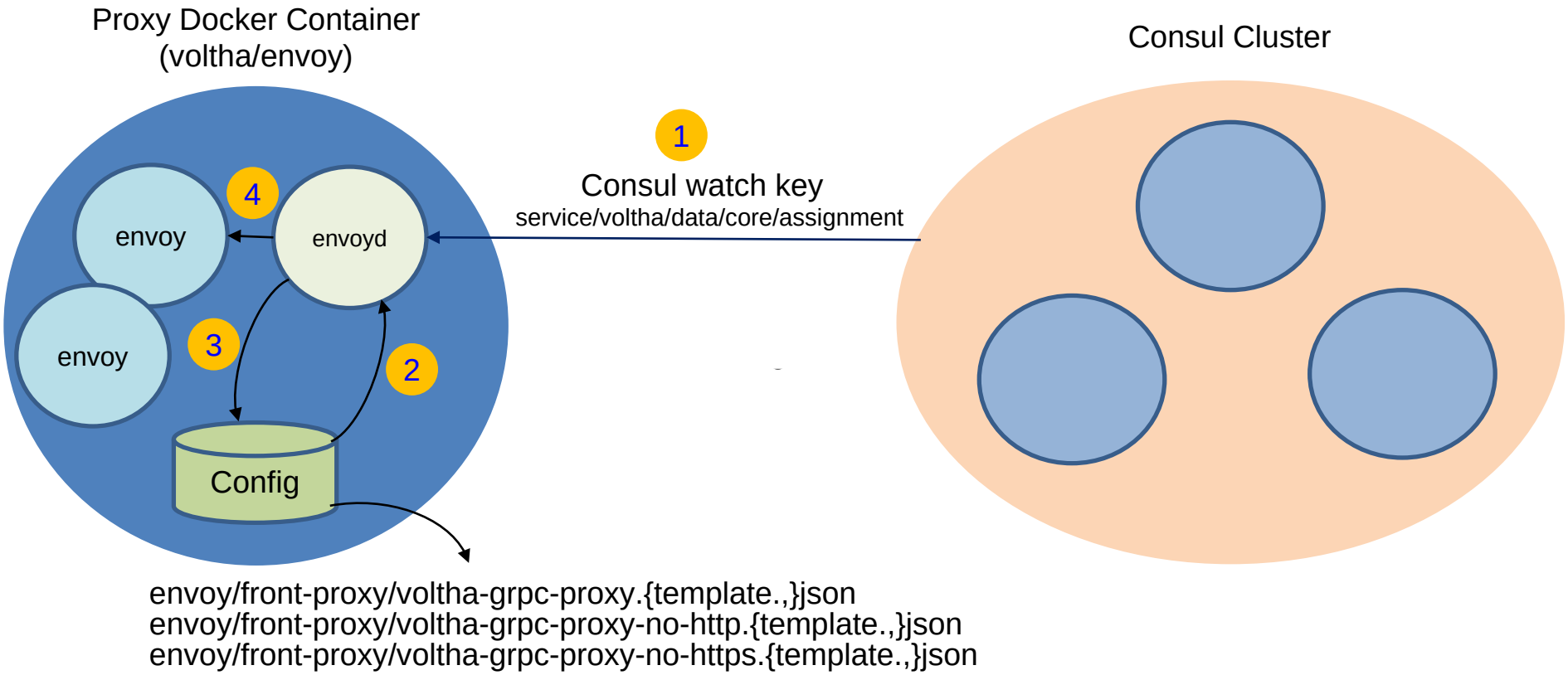
The envoy proxy

The Envoy Proxy

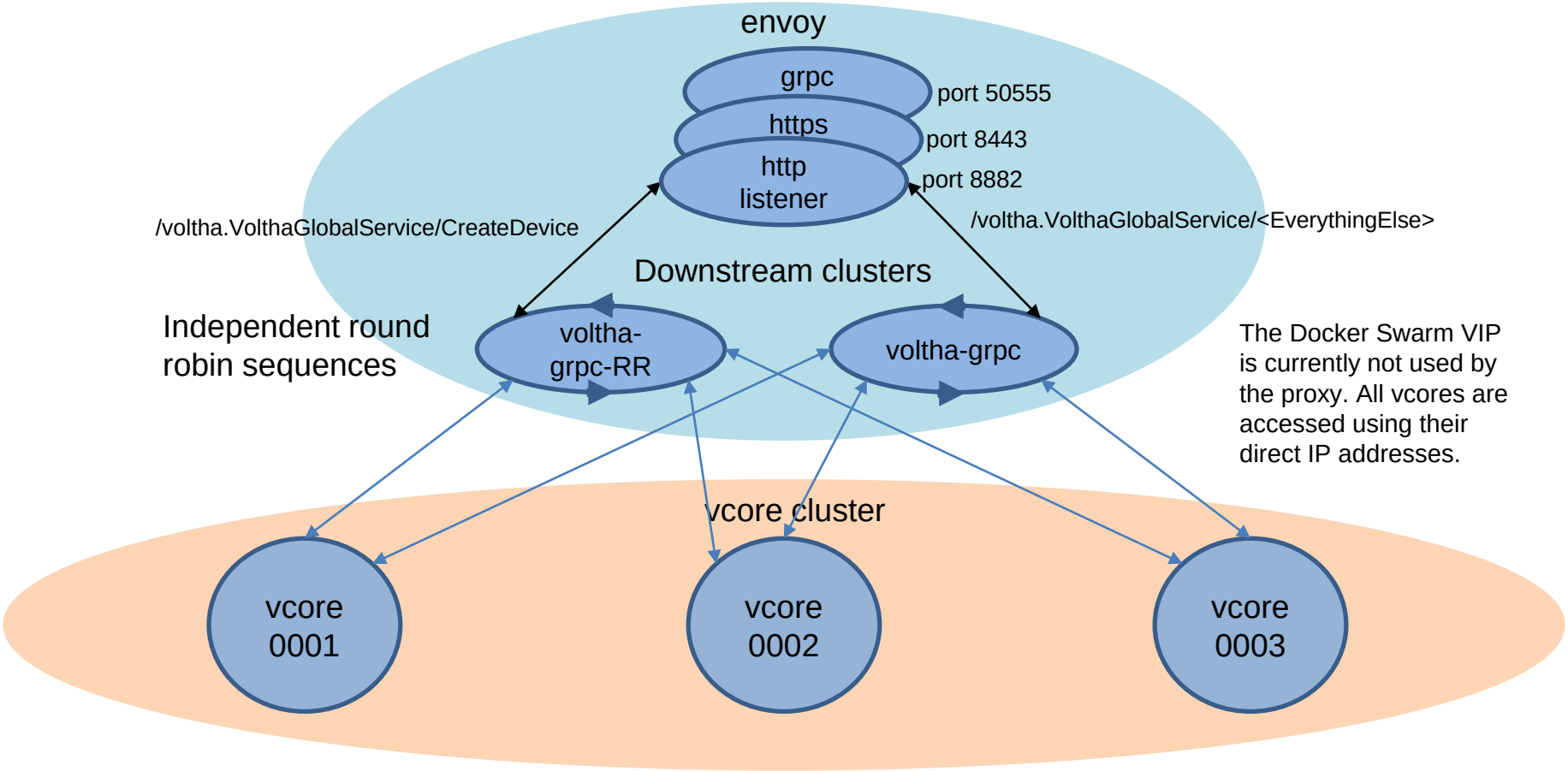
- The proxy itself was originally built at Lyft and is named envoy
  - https://www.envoyproxy.io/
- envoy supports hitless reconfiguration
- envoy supports HTTP2/HTTP to HTTP2 forwarding
- envoy uses compiled protobuf definitions to present both an HTTP and an HTTPS REST API and convert them to HTTP2 downstream requests
- envoyd is a golang daemon that configures, starts, and re-starts the proxy as necessary
- envoyd configures envoy with 2 round robin sequences to provide deterministic load balancing.
- No clustering, only a single instance of the proxy runs which swarm restarts on failure
- The proxy connects only to voltha_net

Proxy Architecture & Interactions

Experience. Outcomes.



Load Balancing



Top Down Review of the Architecture

The OFagent

OFagent

- OFagent is the only service that doesn't connect to vcore through the proxy.
- The OF controller (ONOS) requires each logical device to maintain a distinct connection.
- The OF agent maintains a connection between ONOS and the vcore handling the logical device.
- OFAgent uses "ListLogicalDevices" API to retrieve all the logical devices in a given core and for each logical device creates a connection to ONOS.
- The 1:1 nailed up-connection ensures each logical device has a well defined connection to the controller (just like direct hardware connectivity of an openflow device).
- OFagent connects only to voltha_net

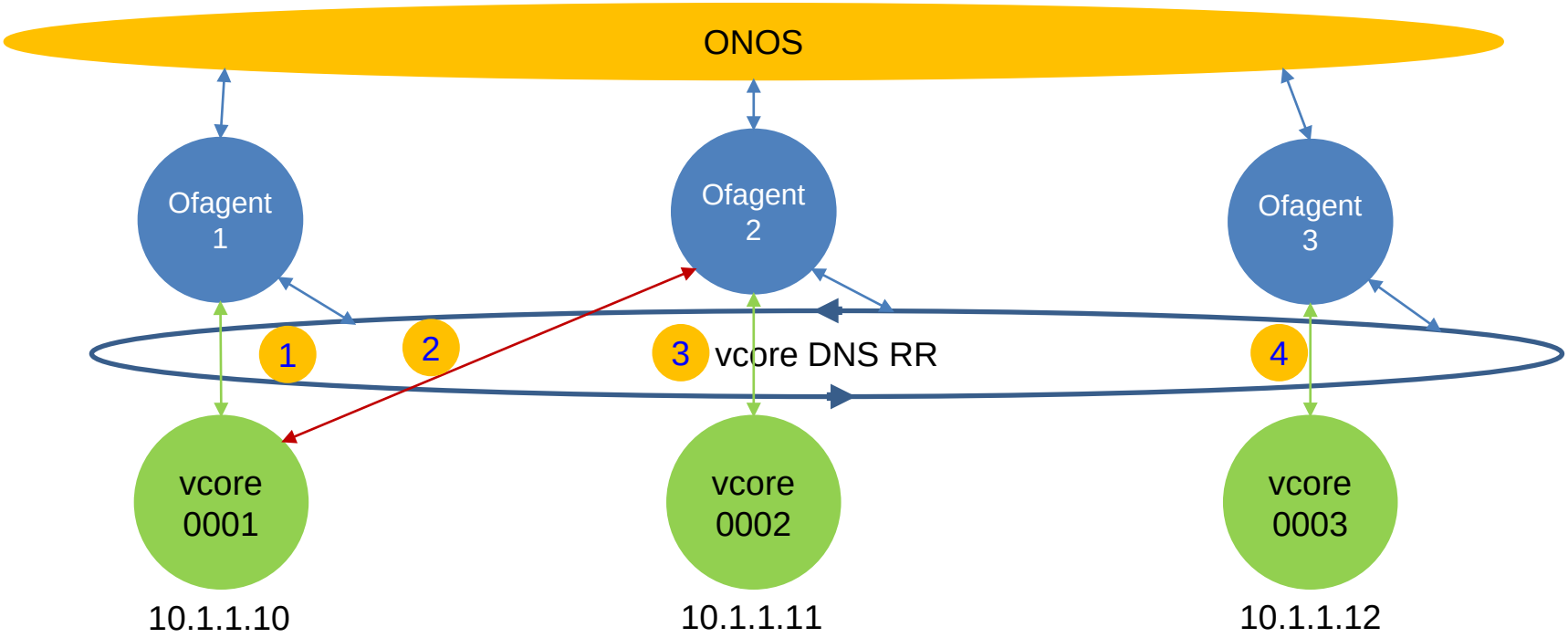
OFagent startup

- On startup each agent attempts to connect to a core using the vcore domain name.
- This will be rejected if an agent is already connected to that core
- The agent will attempt the connection again leveraging the fact that the domain name resolution in docker swarm will round robin through all the containers serving the domain name.
- Eventually each of the OFagents will connect to one and only one core and each core will have one and only one ofagent.

OFagent restart

- When an OFAgent is down, it's corresponding vcore detects the failure (a heartbeat mechanism) and deregisters that OFAgent from itself. The vcore is now ready to accept a new OFAgent connection.
- When an OFAgent is restarted (by docker), that OFAgent follows the same startup mechanism to bind itself to a vcore.
- If only 1 OFAgent was down then the restarted OFAgent will bind to the same vcore as the failed one.
- A similar mechanism is used if the vcore that goes down instead of the OFAgent.

OFagent interactions



Top Down Review of the Architecture

The VOLTHA core

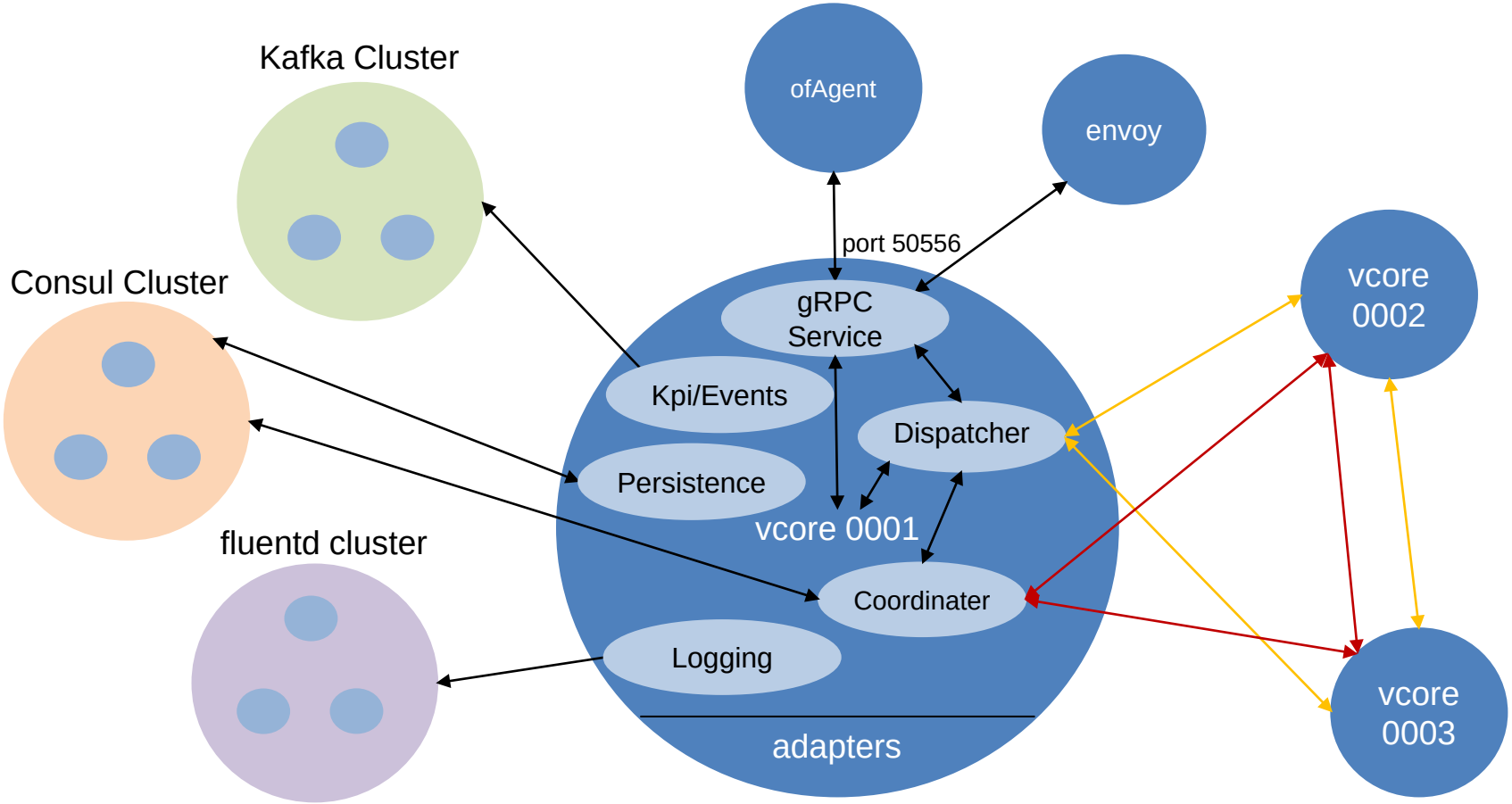
The VOLTHA core (vcore)

- The cord/voltha container runs in a cluster as the vcore service
- There are 3 vcore functions to support HA
  - The dispatcher
  - The coordinator
  - Persistence
- All requests to vcore are made through the gRPC API to the VolthaGlobalService
- All global gRPC requests (except “CreateDevice”) to the vcore are first processed by the dispatcher.
  - As we saw earlier, “Creation” requests are load balanced by the proxy to ensure homogeneous distribution of devices across all running vcore instances.
  - A request targeted locally will be processed by the local instance
  - A request aimed for a remote vcore will be dispatched via gRPC to that vcore
  - A global query request (e.g. listdevices) will be broadcasted to all vcores and the responses combined by the vcore where the request first landed.

The VOLTHA core (vcore)

- The coordinator performs three functions.
  - Vcore “leadership election” – always 1 leader
  - Manages the list of available vcore members as well as creating the host ip to vcore id mapping.
  - Assigning work to new instances (either during vcore scaling or when a vcore crashes)
- Persistence is achieved through the consul K/V store
  - Used by the coordinator to keep the current cluster state
  - Stores the vcore data model
  - One convenient backup and restore point
- The voltha core only connects to the voltha_net network

vcore Architecture and Interactions



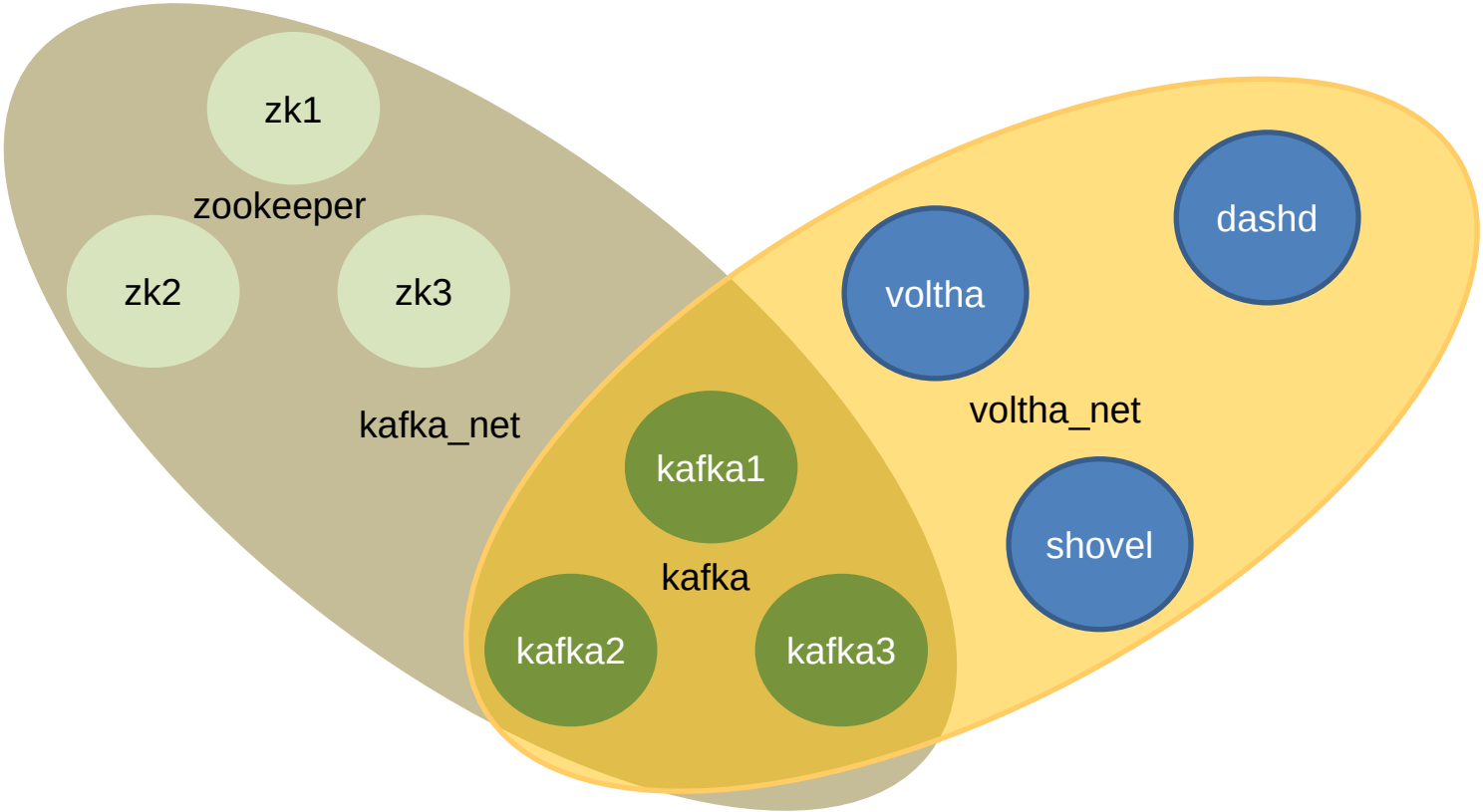
Top Down Review of the Architecture

Kafka and Zookeeper

Kafka and Zookeeper

- Kafka is the primary mechanism for distributing KPIs/events to upstream systems.
- A special overlay network is created for Zookeeper/Kafka communication. (kafka_net)
- All zookeepers need to know about each other.
  - Domain names are used to identify individual instances of zookeeper each running in their own container.
  - Zookeeper only connects to kafka_net
- Kafka needs to know about all the zookeeper instances
  - Zookeeper's domain names are used to locate them.
  - Kafka connects to both kafka_net and voltha_net

Kafka Zookeeper Interactions



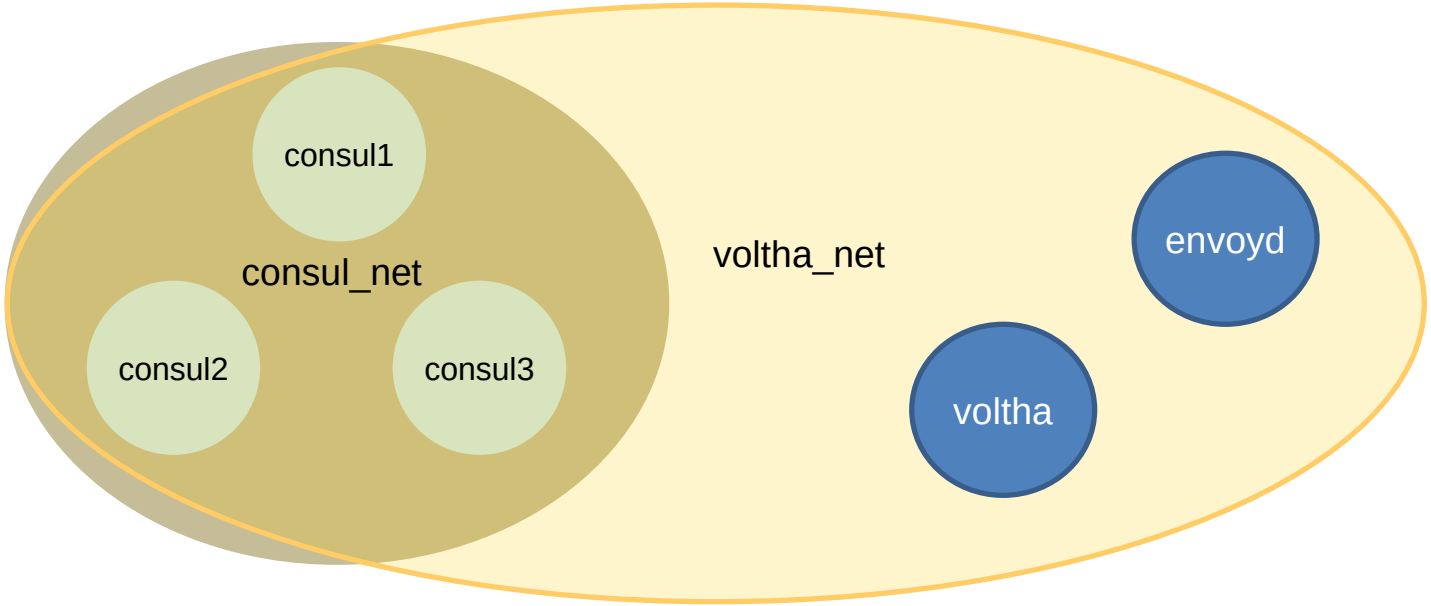
Top Down Review of the Architecture

Consul

Consul

- consul is the primary key/value store for VOLTHA
- consul is deployed in global mode
  - Only one consul instance per server
  - In event of server failure only 2 consul instances continue to run
  - This is the only service deployed this way, all others use replicas=X
- consul uses consul_net for instance to instance communication
  - The consul_net network has a very small IP address space (5)
    - This allows all addresses to be exhaustively listed on consul's command line (retry-join) allowing them to find each other.
- consul connects to both consul_net and voltha_net
- consul mounts a filesystem from the host to persist its data (/cord/incubator/voltha/consul), an external SAN could also be used.
- Though consul appears in most compose files, this is a legacy use of consul as a name server, swarm fulfills that role now.

Consul Interactions



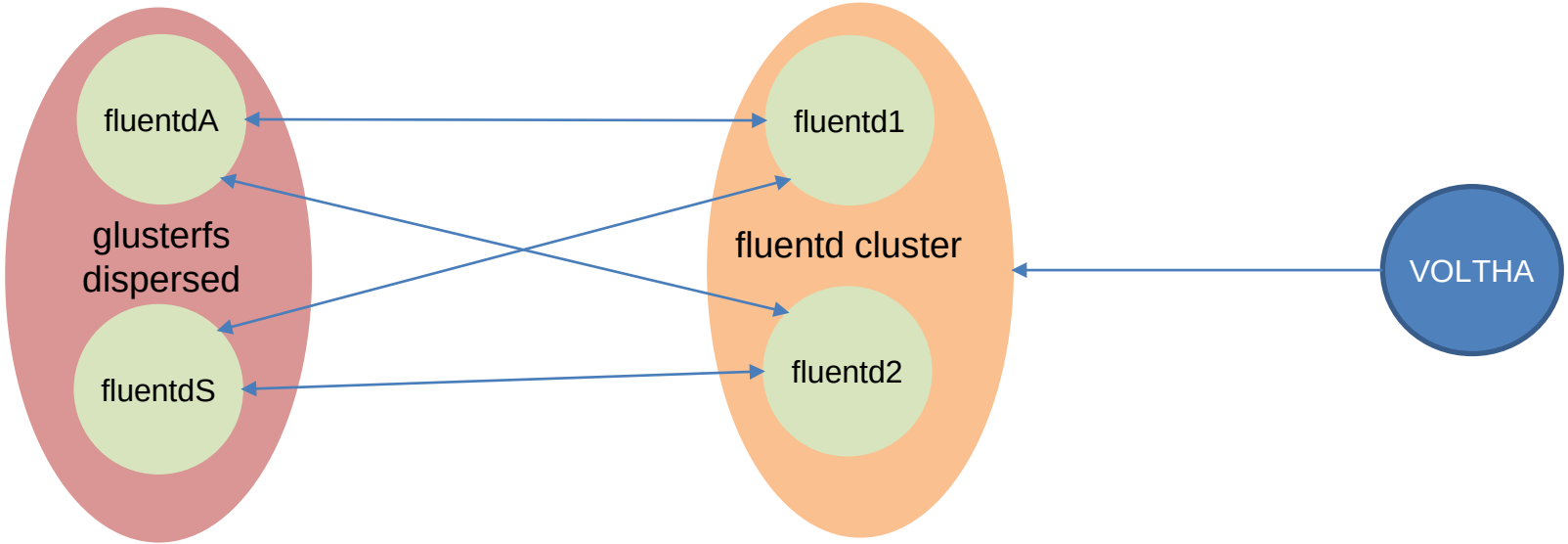
Top Down Review of the Architecture

Fluentd

fluentd

- fluentd is used as a log intake and aggregator
- Not all containers currently use fluentd for logging
  - vcore does
- fluentd uses a glusterfs replicated filesystem to store the logs
  - Currently the filesystem for each glusterfs brick is stored in a loop mounted file rather than a partition on disk or external storage.
    - Logging space is limited to the file size limiting the exposure and possibility that all available disk space is consumed by the logs.
  - The replication mode used is dispersed (similar to raid 5) 2 bricks for data one for parity. The loss of any one brick doesn't impact data integrity.
- There are 2 intakes under one vIP (domain) and 2 aggregators (active and standby)
- fluentd only uses voltha_net.

fluentd interactions



Top Down Review of the Architecture

Registry

Registry

- Implemented as an insecure registry on port 5001
- Single instance restarted by docker on failure
- The registry uses a glusterfs replicated filesystem to store the images
  - Currently the filesystem for each glusterfs brick is stored in a loop mounted file rather than a partition on disk or external storage.
  - Image storage space is limited to the file size.
  - The replication mode used is replicated (similar to raid 1) 1 brick for data on each host. The loss of any 2 bricks doesn’t impact data integrity.

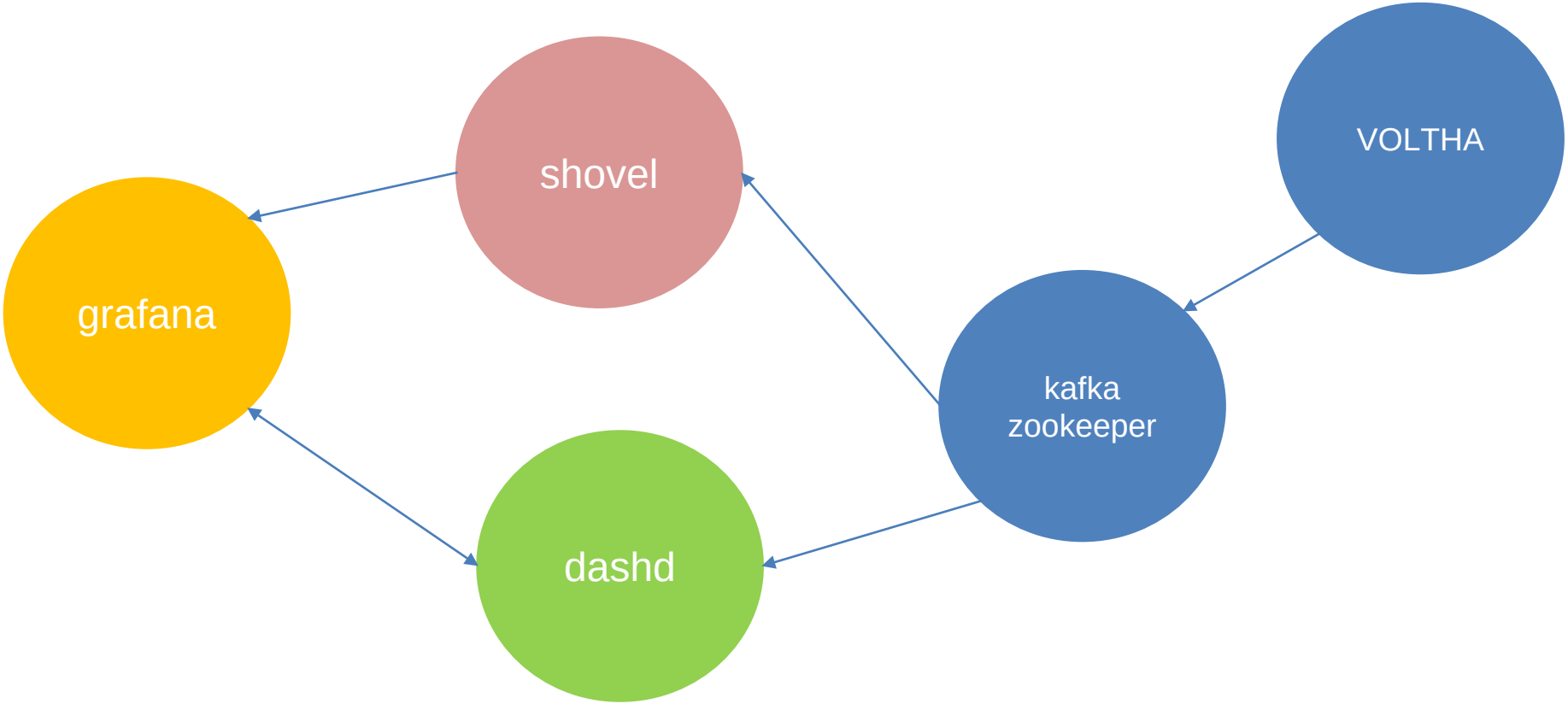
Top Down Review of the Architecture

Grafana KPI visualization

Grafana KPI visualization

- The KPI visualization is an upstream processing example
- Multiple containers are involved
  - grafana, the visualization container
    - Contains the carbon daemon, graphite, and grafana.
  - Shovel: the KPI forwarding engine
  - Dashd: the datasource setup, dashboard creation service
  - Kafka: The KPI source on the voltha.kpis topic
- dashd maintains a list of devices and dashboards
  - As new devices start publishing KPIs dashd adds new dashboards
  - On initial startup, dashd reconciles existing devices with existing dashboards
  - Currently, existing dashboards are never removed in case they've been manually created.
- All communications between the containers is through voltha_net.

Grafana Interactions



Thank You!

Questions?