KubeCon | CloudNativeCon

North America 2017

Building an Edge Computing Platform for Network Services Using Cloud Native Technologies

Stephen Wong, Software Architect, *FutureWei Technologies, Inc.*

Table of Content

- Content
  - Content
    - Content

NFV in the Context of Cloud Native Computing

- Who are we?
  - OPNFV contributors, looking at problem domain of cloud native computing's relevancy in NFV use cases
- NFV —- VNF
  - High-level definition of NFV: idea that majority of network functions could be provided by software running on top of COTS servers
  - High-level definition of VNF (virtual network function) —- as the term "VNF" will be used throughout the presentation
- The problem we are looking at: can we build VNFs as cloud native applications
  - Scalability
  - Resiliency / Fault Tolerance
  - Composability and reusability
  - Ease of development and deployment

Edge Network Services Applicability

- Edge use cases are suitable for cloud native VNFs due to:
  1. Resource constraints: therefore micro-service and containerization provide more optimal resource utilizations
  2. High Cost of Edge Maintenance: resiliency and fault tolerant nature of cloud native applications would be beneficial
  3. Ease of Deployment: suitable for remote push of new images of components, allows rollback if new component fails locally
  4. Replicable: entire deployment can be replicated to different sites

Suitable VNF Type for Cloud Native Application

- Not all VNFs are suitable for being implemented as cloud native applications
- Web backend application is transactional and event-driven by nature, VNFs that follow the same pattern would be natural fit
  - Network functions such as firewall
  - A subset of these network functions is of class transparent proxy —- need a unified way to direct traffic to VNF entrance point. This is known as the “service insertion” problem
  - Loadable / composable data path

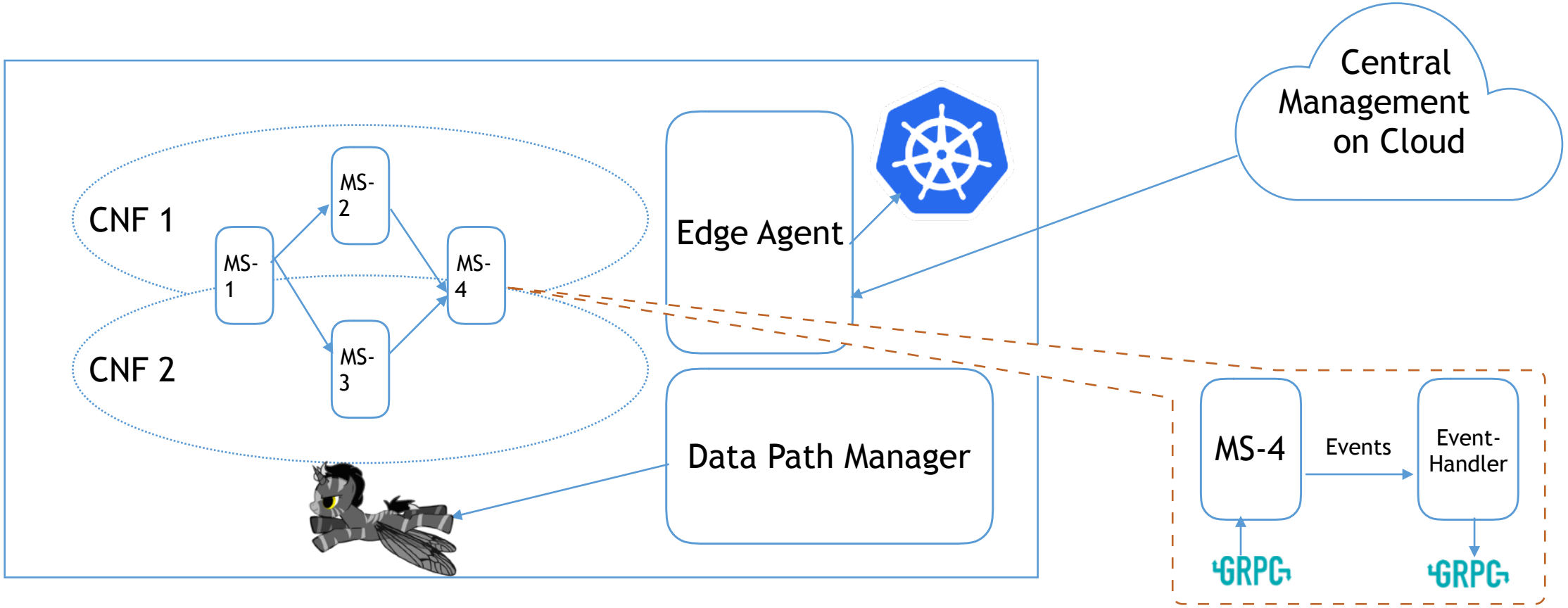
VNFFG (VNF Forwarding Graph)

- An important NFV orchestration feature is VNFFG
- Cloud native VNF forwarding graph should provide more programmability
- As cloud native application, VNF components communicate with each other via REST APIs / gRPC type mechanism; however, how VNFs are chained is user defined
  - Should everything be API driven?

Terminology

1. CNF: **C**loud-native **N**etwork **F**unction —- defined by *operator*, made up of a list of *microservices*
2. Client: initiates traffic which would pass through the edge node
3. Event-handler: a sidecar on a pod. Event-handler is per-*µsrv* (see definition below), can dynamically load *operator* defined handlers, and implements the SDK stubs
4. µsrv (microservice): a unit of compute service which exposes a set of APIs, the “lego pieces” that would make up a CNF
5. Operator: defines and deploys CNFs

Architecture of Our Approach



Technical Descriptions

I. Event-driven Model
   1) Event handler running as sidecar in the same pod as microservices
   2) Microservices expose a set of events, which operators can define and dynamically loaded
   3) An SDK would be provided by the framework. Operators invoke SDK calls in their custom event handlers, then internally those SDK calls would become gRPC / REST API calls

II. Programmable Datapath
   1) Takes network policy rules and program data path engine accordingly
   2) Rule format : { <ip> port : {redirect | copy} => [list of labels]}
   3) Utilizes eBPF (IOVisor BCC) to load BPF code

Demo Description

1. Operator uses the portal to define a CNF (gateway) which includes multiple µsrvs (policy-mgr, content-inspect), each µsrv publishes a set of events, and operator can choose to implement these event-handlers and customized them via a set of Python SDK methods, and deploy the CNF to a target edge node
2. Previously non running µsrvs (policy-mgr, content-inspect) now running on edge node, with the new event-handler (per µsrv) built/pushed running alongside the core µsrv (running as a container)
3. Client sends an HTTP request —- the transport traffic is being redirected to the specified pod associated with the datapath policy's 'redirect' action's label; and subsequently the call sequence matches what the operator specified
4. Operator updates portal to add a µsrv (anti-virus) with updating a previous SDK call (virus_scan SDK call) from an existing event handler (scan_done event from content-inspect)
5. After CNF (re)deploy, the call flow now goes to new µsrv (anti-virus) with the correct RPC call
6. Operator wants to narrow what type of traffic should get virus scan (as opposed to everything), thus updating the event_handlers (remove virus_scan from content-inspect's scan_done, add it as a condition under policy_check from policy-mgr)

Demo

- [To-Be Added: demo diagram]

Roadmap (1): Data Collection

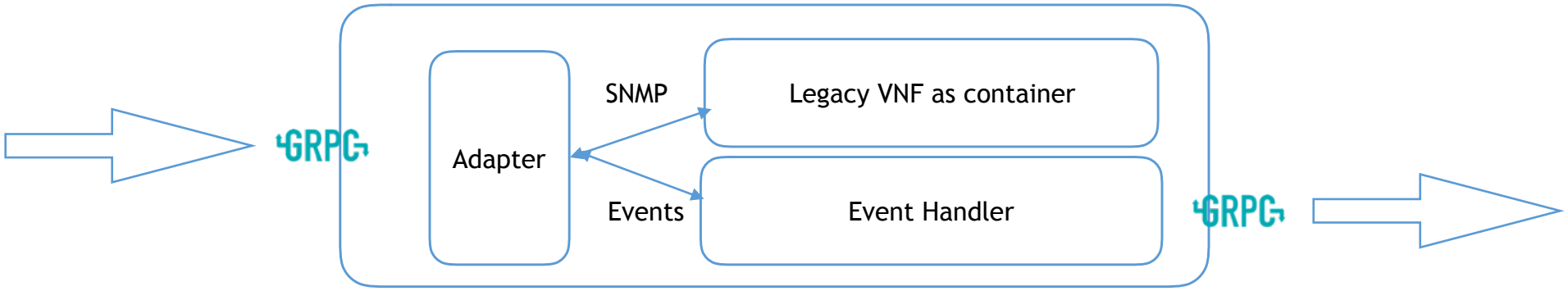
- For edge use cases, it is important to support rich set of data to be collected, also, allowing software-defined filtering of the data being sent to centralized management
- As networking solution, network tracing is important
  - An important reason why eBPF is chosen for datapath engine
  - Particularly IOVisor BCC comes with a rich set of kernel tracing code
- Data correlation: everything throughout the stack (OpenTracing calls, eBPF map, redis objects…etc) all have exact same key (the **cookie** parameter being passed throughout the events / gRPC calls), thus allowing us to bundle data from various sources to be related

Roadmap (2): Multiple Event Handlers per MicroService

- A major drawback of the demo implementation is that the µsrvs are non-sharable, i.e. how the event_handler is invoked per pod makes each µsrv only usable by one CNF
- Moving forward, to allow µsrv to be used by multiple CNFs, cookie filtering value can be used, and event channel can be pub/sub
- Request side needs to make a decision on which CNF a session belongs to (i.e., and most commonly, by client), such info needs to be propagated to **ALL** event_handler on all associated µsrv pods (i.e., like DPM, but for programming the event_handler to subscribe to a topic)
- Programmable session => CNF correlation?

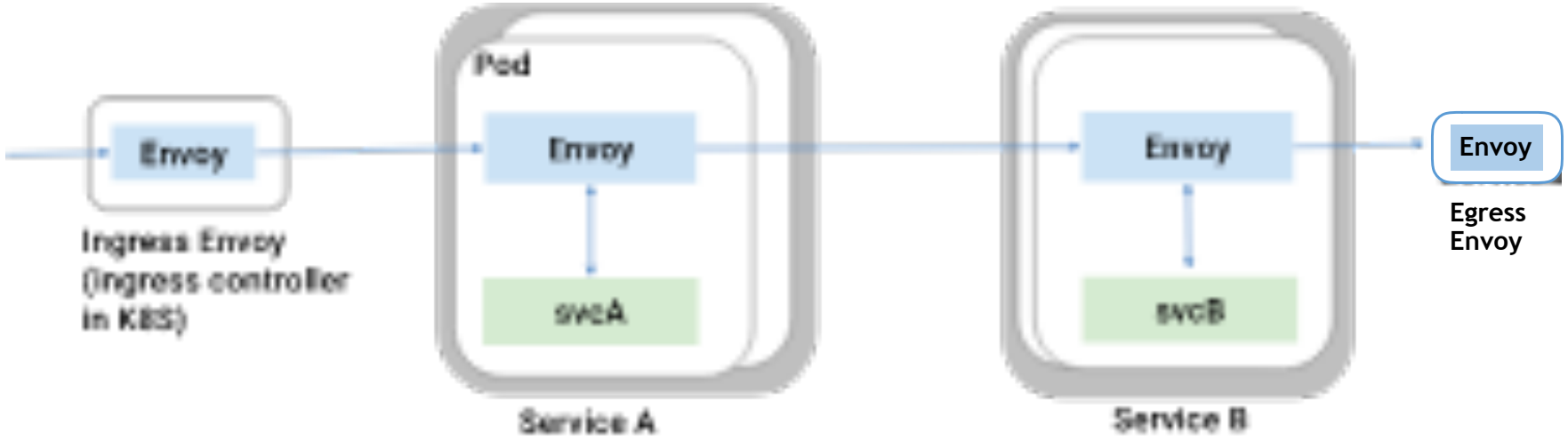
Roadmap (3): Legacy VNF Integration

- There exists a rich and widely deployed set VNFs used by various operators —- it is naive and irresponsible to believe VNFs needed to be rewritten to fit this cloud native model
- The two requirements for a µsrv in this framework:
    1. Event generation (and synchronous or asynchronous handlers for those events)
    2. gRPC request handling
- The above requirements do assume the VNFs have some degree of programmability
- One solution to incorporate legacy VNF in this framework would be having another sidecar / container per pod as an adapter to communicate with legacy VNF (as µsrv). The adapter would interface with the legacy VNF (ex: SNMP, CLI) whilst implementing a gRPC server with event generating logic



Roadmap (4): Service Mesh

- If scaling up is needed, this framework matches well with the current service mesh architecture (such as Istio)



Information

- Slack channel: TBD
- Project Clover under OPNFV (https://wiki.opnfv.org/display/CLOV)
  - Clover intends to investigate the issues and projects associated with building VNFs as cloud native applications
- Email Address: stephen.kf.wong@gmail.com