08.01.2024

Warszawa

Pracownia Dyplomowa Magisterska

Autonomiczne pętle sterowania i wykorzystanie operatorów Kubernetes na potrzeby zarządzania cyklem życia usług skonteneryzowanych (CNF).

Andrzej Gawor

Opiekun: dr. Inż. Dariusz Bursztynowski

1. Cele pracowni

- Zarządzanie usługami komunikacyjnymi w środowisku operatorskim w świetle standaryzacji na przykładzie praci ETSI ENI, w szczególności koncepcja autonomicznej zamkniętej pętli sterowania wg ETSI GR ENI 017 V2.1.1 (2021-08), Experiential Network Intelligence (ENI); Overview of Prominent Control Loop Architectures

- Dokładne zapoznanie się z platformą Kubernetes w wybranym środowisku (wg własnej decyzji, np. minikube, Kind, inne). W ramach tego zadania mieści się opanowanie podstaw Kubernetes Operator Framework (koncepcja Custom Resource, pojęcia operatora, implementacja własnego operatora).

- Implementacja przykładowych operatorów wg własnego pomysłu.

- Próba określenia zasad modelowania pętli autonomicznej opartej o wzorzec CRD (Custom Resource Definition) oraz operatory Kubernetes, wzorowanej na ogólnym podejściu ETSI ENI wg dokuemntu z punktu 1.

2. Poczynione kroki

Lista wykonanych czynności znajduje się poniżej. W kolejnych podsekcjach każdy krok został opisany:

- Zapoznanie się z koncepcją zamkniętej pętli sterowania wg. ETSI ENI

- Zapoznanie się z platformą Kuberntes

- Utworzenie środowiska deweloperskiego

- Praktyczne zapoznanie się z konceptem CRD

- Zapoznanie się z operator pattern

- Zapoznanie się ze środowiskiem Kubebuilder

- PoC: Operator komunikujący się z zewnętrznym komponentem poprzez REST

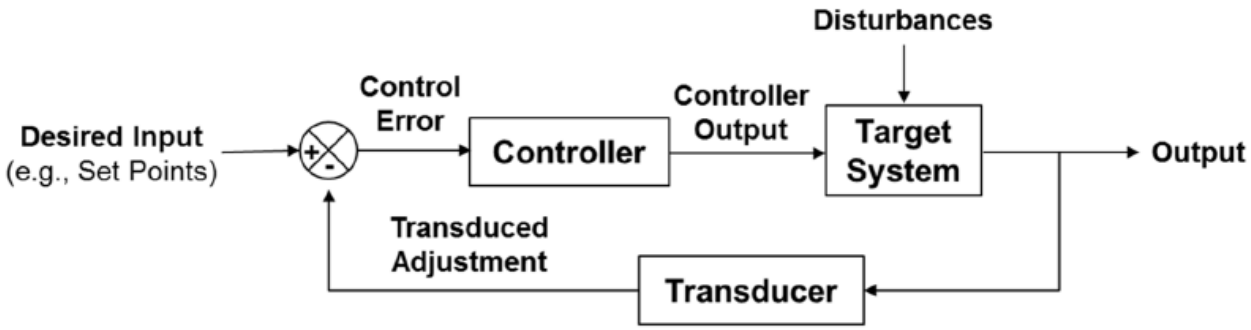
- Próba modelowania autonomicznych pętli sterowania

2.1 Zapoznanie się z koncepcja zamkniętej pętli sterowania wg ETSI ENI

Wprowadzenie do sieci takich technologii jak SDN, NFV czy Network Slicingu sprawia, ze są one dużo bardziej elastyczne i konfigurowalne. Następnym krokiem jest uczynienie je inteligentnymi, tak aby dopasowywały oferowane usługi bazując na zmianach w: potrzebach użytkowników, warunkach środowiskowych czy celach biznesowych operatora. Takie podejście jest fundamentem 6 generacji sieci komórkowych. Jednym z komitetów ETSI – European Telecommunications Standards Institute jest ENI – Experiential Network Intelligence. Misją ENI jest rozwój standardów dla kognitywnych systemów zarządzania siecią, wprowadzających metryki optymalizacji i dostosowywania się sieci bazując na doświadczeniu jej oraz innych sieci.

ENI zauważa, iż do automatyzacji zarządzania siecią można wykorzystać istniejące już w innych dziedzinach zamknięte pętle sterowania. Ogólny schemat/architektura takiej pętli jawi się na rysunku

poniżej:



Akcje sterowania są zależne od „feedback'u" jaki przyniesie sterowany system po zrealizowanych poprzednich akcjach sterownia. W ten sposób pętla może uczyć się jak poszczególne działania wpływają na system i dopasywować je na przyszłość.

W ramach pracowni zapoznano się z raportem ENI [1]. Dokument przedstawia kilka pętli: OODA, MAPE-K, FOCALE, GANA, COMPA oraz FOCALE v3.

2.2 Zapoznanie się z platformą Kubernetes

Z racji zerowej wiedzy w temacie Kubernetes w ramach pracowni wybrany został kurs dający podstawową wiedzę w szerokiej gamie aspektów tejże platformy. Kurs realizowany na platformie KodeKloud, która oprócz treści wykładowych oferuje również dostęp do praktycznych ćwiczeń laboratoryjnych. Ukończony kurs pozwala uzyskać certyfikat wydawany przez CNCF (Cloud Native Computing Foundation), fundacje będącą właścicielem platformy Kuberntes [2].

Wiedza nabyta podczas kursu dała podstawy, aby móc zapoznać się z konceptami Kubernetes dotyczących operatorów bezpośrednio z oficjalnej dokumentacji.

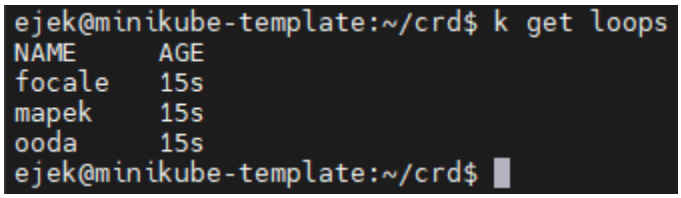
2.3 Utworzenie środowiska deweloperskiego

W ramach pracowni przygotowano maszynę wirtualną z systemem ubuntu-server, a na niej stworzono „sztuczny" klaster Kubernetes za pomocą Minikube. Minikube pozwala postawić lokalny kluster Kubernetes jako pojedynczy kontener w Dockerze. Kontener ten udostępnia port do komunikacji z komponentem kube-api-server, a z racji, że ten jest architektonicznym pojedynczym brokerem wymiany wiadomości w klastrze, to dla operatorów dewelopowanych zewnętrzenie względem kontenera Minikube, tak utworzony klaster jest transparentnty. Innymi słowy w obrębie prac niniejszej pracowni magisterskiej bez różnicy jest czy użyty zostanie prawdziwy klastrer Kubernetes, czy emulacja w postaci Minikube. Różnica leży w łatwości oraz czasie „postawienia" klastra, dlatego zdecydowane się na takie rozwiązanie.

Całość procesu została udokumentowana: [3].

2.4 Praktyczne zapoznanie się z konceptem CRD

Custom Resource Definition (CRD) jest to mechanizm Kubernetes pozwalający na rozszerzenie jego możliwosci o rejestracje nowych typów (ang. „kind") obiektów API. Obiektów, które są niestandardowe, to znaczy – definiowane przez użytkownika klastra. W ramach pracowni przygotowano przykładowe CRD, tak aby móc zarejestrować własny niestandardowy typ (ang. „kind") obiektu API, a następnie utworzyć kilka obiektów tego typu.



Rys. 1Prezentacja stworzonych niestandardowych obiektów "loops"

Dokumentacja: [5].

2.5 Zapoznanie się z operator pattern

Po pierwsze należy wyjaśnieć pojęcie controller pattern. Kubernetes dla swoich domyślnych obiektów API stosuje właśnie ten wzorzec. Kontroler w Kubernetesie jest komponentem, który realizuje pętle sterowania. Resource definition sporządzony w formacie yaml definiuje tzw. „desired state" danego zasobu. Gdy w kube-api-server dojdzie do utworzenia/modyfikacji/usunięcia obiektu, Kubernetes wywoła dla tego obiektu kontroller danego typu. Kontroller dokonuje wtedy tzw. rekoncyliacji (ang. „reconcile"), czyli pogodzenia/pojednania stanu „desired" ze stanem obserwowanym w klastrze. Przykładowo: W klastrze Kubernetes istnieje domyśle Kind „ReplicaSet". Użytkownik tworzy obiektu tego typu i w resource defnition file specyfikuje liczbę podów o wartości 3. Kiedy kubectl przy tworzeniu obiektu wysyła odpowiedni request to kube-api-server, ten wywołuje kontroller typu „ReplicaSet" odpowiednim wydarzeniem. Kontroller tworzy odpowiednią liczbę podów. Następnie podczas runtime, jeśli użytkownik zechce usunąć jeden z podów, to kube-api-server, który otrzyma żadanie usunięcia znowu wywoła kontroler powiązanego z tym podem obiektu ReplicaSet. Kontroler porówna stan akutalny (liczba podów 2) z desired state (liczba podów 3) i zadecyduje o wysłaniu żadania na dostawienie nowego poda.

Logika kontrolerów dla domyślnych obiektów Kubernetes została zaimplementowana przez programistów Kubernetes. Jest to ważne spostrzeżenie z punktu widzenia operatorów, ponieważ operator Kubernetes to nic innego jak kontroller dla typów niestandardowych wprowadź pojęcie. Różnica w tym, że implementacja logiki tych kontrolerów należy do autorów CRD.

Cała ścieżka nauki została udokumentowana pod linkiem: [4].

2.6 Zapoznanie się ze środowiskiem Kubebuilder

Maszynie wirtualnej przygotowanej w kroku 2.3 zainstalowano kubebuilder i rozpoczęto prace nad przykładowymi operatorami. Początkowo wybrano framework OperatorSDK, który jest nakładką na kubebuilder i ma jeszcze bardziej przyspieszyć wdrożenie prostych operatorów. Jednakże porzucono ten pomysł na rzecz lepszego zrozumienie podstaw wytwarzania operatorów. Dzięki użyciu kubebuilder zaimplementowano CronJob controller z poradnika znajdującego się na oficjalnej stronie kubebuilder. Podczas tego kroku w szczególności dobrze dało się poznać architekturę biblioteki controller-runtime potrzebnej do implementacji operatorów Kubernetes.

2.7 PoC: Operator komunikujący się z zewnętrznym komponentem poprzez REST

Jednym z zagadnień modelowania autonomicznych zakmniętych pętli sterowania jest zbadanie możliwość wyekstrachowania logiki ich bloków funkcjonalnych poza klaster Kubernetes. Pierwszym pomysłem jaki nasuwa się w tej kwestii jest komunikacja operatora z zewnętrznym serwerem HTTP. W tym celu zaimplementowano prosty serwer HTTP odpowiadający na każde żądanie ciągiem znaków „Hello world",

a do operatora z poprzedniego kroku dodano w funkcji Reconcile kawałek kodu, który wykonuje zapytanie do wspomnianego serwera HTTP.

```go
func (r *CronJobReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    log := log.FromContext(ctx)

    resp, err := http.Get("http://localhost:14555")
    if err != nil {
        // Handle error
        return ctrl.Result{}, err
    }
    defer resp.Body.Close()

    // Read the response body
    bodyBytes, err := io.ReadAll(resp.Body)
    if err != nil {
        // Handle error
        return ctrl.Result{}, err
    }

    // Convert body to string
    bodyString := string(bodyBytes)

    log.V(1).Info(fmt.Sprintf("=====================================: %s", bodyString))
```

```
2024-01-08T02:23:05Z    INFO    setup   starting manager
2024-01-08T02:23:05Z    INFO    controller-runtime.metrics      Starting metrics server
2024-01-08T02:23:05Z    INFO    starting server {"kind": "health probe", "addr": "[::]:8081"}
2024-01-08T02:23:05Z    INFO    controller-runtime.metrics      Serving metrics server  {"bindAddress":
2024-01-08T02:23:05Z    INFO    Starting EventSource    {"controller": "cronjob", "controllerGroup": "ba
2024-01-08T02:23:05Z    INFO    Starting EventSource    {"controller": "cronjob", "controllerGroup": "ba
2024-01-08T02:23:05Z    INFO    Starting Controller     {"controller": "cronjob", "controllerGroup": "ba
2024-01-08T02:23:05Z    INFO    Starting workers        {"controller": "cronjob", "controllerGroup": "ba
2024-01-08T02:23:05Z    DEBUG   =====================================: Hello world!
        {"controller": "cronjob", "controllerGroup": "batch.tutorial.kubebuilder.io", "controllerKind":
ob-sample", "reconcileID": "eb94b36a-a096-4d8d-b766-e99f465abea3"}
2024-01-08T02:23:05Z    INFO    KubeAPIWarningLogger    unknown field "spec.jobTemplate.metadata.creatio
2024-01-08T02:23:05Z    INFO    KubeAPIWarningLogger    unknown field "spec.jobTemplate.spec.template.me
2024-01-08T02:23:05Z    DEBUG   created Job for CronJob run     {"controller": "cronjob", "controllerGro
pace":"default"}, "namespace": "default", "name": "cronjob-sample", "reconcileID": "eb94b36a-a096-4d8d-b
-08T02:23:00Z", "job": {"namespace": "default", "name": "cronjob-sample-1704680580"}}
2024-01-08T02:23:05Z    DEBUG   =====================================: Hello world!
        {"controller": "cronjob", "controllerGroup": "batch.tutorial.kubebuilder.io", "controllerKind":
ob-sample", "reconcileID": "8d02ae54-aac8-404f-89b0-0bf5f3f24e24"}
2024-01-08T02:23:05Z    DEBUG   no upcoming scheduled times, sleeping until next        {"controller": "
:"cronjob-sample","namespace":"default"}, "namespace": "default", "name": "cronjob-sample", "reconcileID
2024-01-08T02:23:05Z    DEBUG   =====================================: Hello world!
        {"controller": "cronjob", "controllerGroup": "batch.tutorial.kubebuilder.io", "controllerKind":
ob-sample", "reconcileID": "e3d2ca33-64ab-4e45-b902-f5755e80a136"}
```

2.8 Próba modelowania autonomicznych pętli sterowania

2.8.1 Wstęp

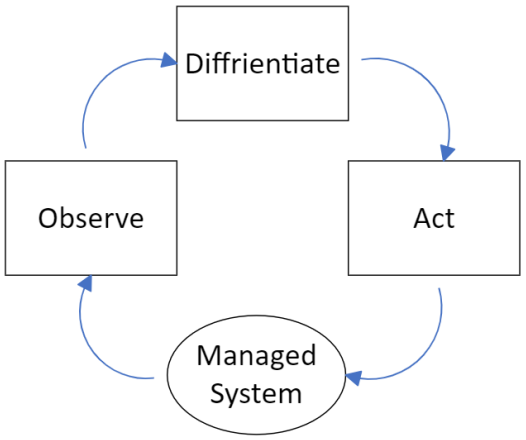
Podstawowy Controller Pattern w Kubernetes jest wystarczający dla domyślnych zasobów Kubernetes. Pozwala on na implementacje prostej zamkniętej pętli sterowania, której cała logika znajduje się w kodzie źródłowym kontrolera. Pętla ta zazwyczaj polega na:

- Obserwacji zmian, które zaszły w obiekcie, czyli rejestracja jego aktualnego stanu,

- Porównanie go z "desired state" i wyłapaniu różnic, a następnie

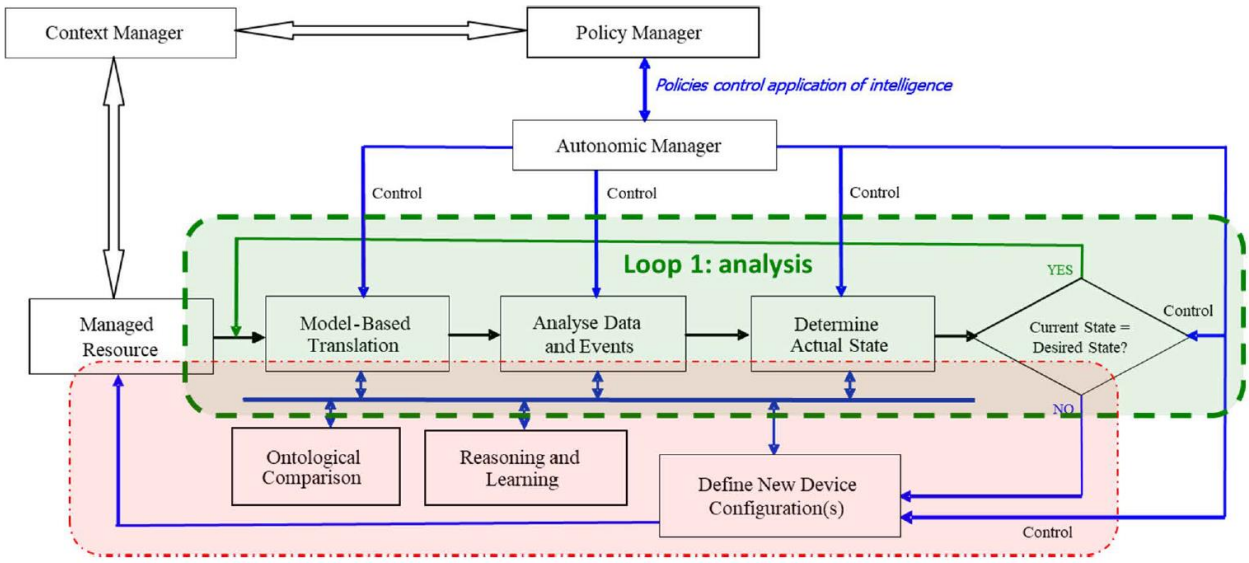
- Podjęciu odpowiednich akcji, które zbliżą stan aktualny do stanu "desired".

Schemat tej prostej pętli można przedstawić na rysunku poniżej:



Celem ostatecznym pracowni jest wypracowanie zasad modelowania bardziej złożonych zamkniętych pętli sterowania. Większy poziom ich złożoności polega na większej ilości bloków funkcjonalnych (które to mogą np. przechowywać dane historyczne, używać uczenia maszynowego itp.)

Weźmy dla przykładu pętle Focale:

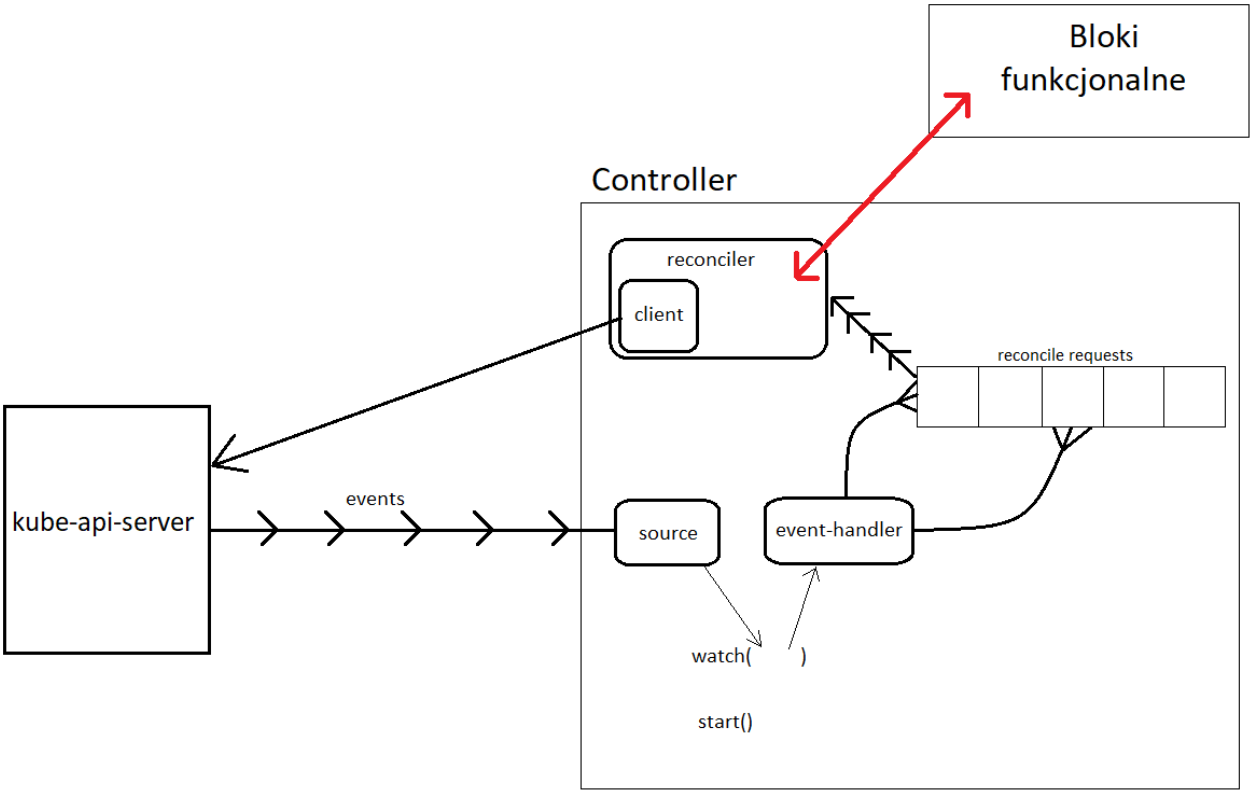


Po lewej stronie widzimy zarządzany system („managed resource”). Cała reszta podpisanych prostokątów na tym rysunku to bloki funkcyjne tej pętli.

Podstawowym pytaniem jest gdzie miały by się znajdować te bloki funkcjonalne. Wyróżniono dwa podejścia. Pierwsze zakłada umiejscowienie bloków funkcjonalnych pętli poza klastrem Kubernetes, wykorzystanie CRD do zamodelowania zarządzanego systemu, a operator tego CRD odwoływałby się do zewnętrznych bloków funkcjonalnych. Podejście to nazwano "external". Drugi model zaklada umieszczenie bloków funkcjonalnych w klastrze Kubernetes jako pody. To podejście pozwala też użyć mechanizmów Kubernetes do zamodelowania pętli (wyboru jej konkretnej wersji/implementacji). Jedno

główne CRD modelowało by zarządzany system, a drugie wybraną pętle sterowania. Podejście o nazwie "internal".

2.8.2 External



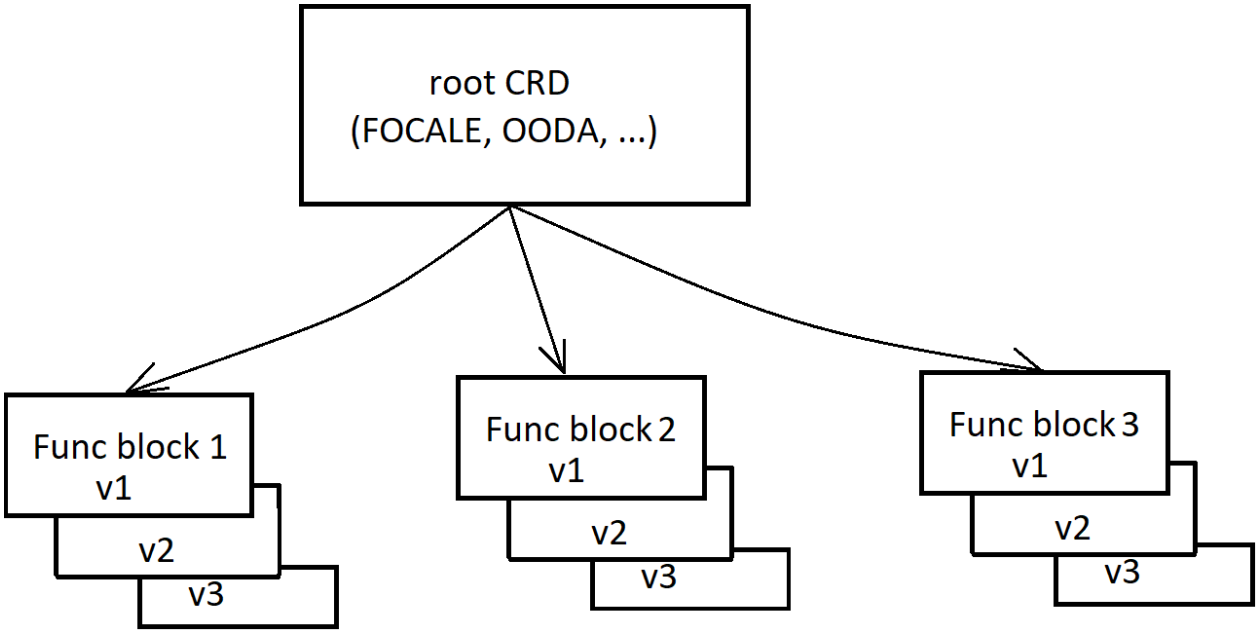
Główne CRD modeluje zarządzany system. Kontroler tego CRD komunikuje się z zewnętrznym systemem. Możliwy sposób komunikacji został przedstawiony w sekcji 2.7. Operator musi zakomunikować nowy stan systemu (do bloku funkcjonalnego odpowiadającego Observe w OODA), a następnie otrzymać instrukcje jakie akcje sterowania podjąć (od bloku Act w OODA). Jak widać operator jest tu tylko proxy, sam nie posiada żadnej logiki pętli sterowania. Posiada jedynie logikę translacji, normalizacji otrzymanych danych na język Kubernetes. Potrzebny jest zatem do zdefiniowania protokół do przekazywania tej logiki. Zarówno przykazywanie aktualnego stanu systemu jak i odbiór akcji sterowania. Dodatkowym aspektem do rozważenia jest asynchroniczność. Czy w jednej iteracji system zewnętrzny zdąży „wyczarować" odpowiedź? Wartym rozważenia jest tu użycie operatora tylko w jedną stronę, tzn. jedynie do informowania o stani systemu. Do wykonywania akcji system zewnętrzny może bezpośrednio używać kube-api-server. Rozwijając to podejście jednak, system zewnętrzny może sam obserwować wybrane obiekty przez kube-api-server.

2.8.3 Internal

Wszystkie rozważana z poprzedniego punktu pozostają takie same, gdyż po prostu system zewnętrzny to teraz pody wdrożone na tym samym klastrze co zarządzany system. Jedyną nową rzeczą jaka tu dochodzi jest zarządzanie pętlą sterowania w oparciu o pliki CRD w klastrze.

Skoro logika pętli znajduje się w klastrze, to też w klastrze zapada decyzja o jej konkretne implementacje, konfiguracje. Musi to znaleźć odzwierciedlenie w plikach CRD. Aby dać możliwość wyboru różnych

implementacji bloków funkcjonalnych pętli, zostanie wprowadzona dwuwarstwowa hierarchia plików CRD. Plik CRD znajdujący się na samej górze (tzw. "root CRD") o zadecyduje o wyborze pętli (FOCALE, OODA, MAPE-K, GANA). Pody z aplikacjami bloków funkcjonalnych mogą mieć różne wersje/implementacje. Także w pliku "root CRD" zostanie wskazana nazwa pliku CRD drugiego poziomu, który zawiera w sobie komponenty przedstawiające konkretną wersję/implementację.



W pliku CRD (format YAML) w polu `schema.openAPIV3Schema.properties.spec.` będzie widoczna następująca struktura:

```yaml
type: object
properties:
funcBlock1:
  type: string
  enum:
    - v1
    - v2
    - v3
 funcBlock2:
  type: string
  enum:
    - v1
    - v2
    - v3
 funcBlock3:
  type: string
  enum:
    - v1
    - v2
    - v3
```

`enum` w yaml wylicza jakie wartość może przyjąć dane pole. Kontroler root CRD podczas tworzenia swojego obiektu będzie tworzył obiekt zdefiniowany przez CRD o nazwie wskazanej w polach `funcBlock*`. Te pliki CRD (dolnego poziomu) będą zawierały wszystkie potrzebne zasoby do uruchomienia odpowiednio skonfigurowanej instacji poda zawierającego logikę danego bloku funkcjonalnego we wskazanej wersji/implementacji.

Przykładowy „root CRD” plik dla OODA:

```yaml
---
# Example scaffold (incomplete) CRD for Focale closed loop architecture style
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
name: ooda.loops.example.org
spec:
  group: loops.example.org
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    plural: oodas
    singular: ooda
# Loopstyle in kind: Loopstyle will always take proper name of the specific style
of control loop, e.g. focale, mape-k, ooda, ...
  kind: Ooda
  shortNames:
  - od
# list of versions supported by this CustomResourceDefinition
versions:
  - name: v1
served: true
storage: true
schema:
  openAPIV3Schema:
    type: object
    properties:
      spec:
        type: object
        properties:
        status:
          type: string
        Observe:
          type: string
          enum:
            - ObserveV1
            - ObserveV2
        Orient:
```

```
        type: string
        enum:
          - OrientV1
          - OrientV2
      Decide:
        type: string
        enum:
          - DecideV1
          - DecideV2
      Act:
        type: string
        enum:
           - ActV1
           - ActV2
```

3. Wnioski i rezultaty

Podczas pierwszego semestru pracowni dyplomowej bardzo dobrze zapoznano się ze środowiskiem, ekosystemem, w którym prowadzone będą badania w ramach następnego semestru. Po rozpoznaniu możliwości środowiska (to jest: Operatory Kuberntes oraz Custom Resource Definitions) potrzebne jest ponowne określenie celów pracowni i przystąpienie do właściwych prac.

Kierunki rozwoju:

- opracowanie mechanizmów włączania w operatory złożonych polityk (czyli realizacji procesów decyzyjnych w oparciu o szeroki zbiór metryk oraz reguł/algorytmów decyzyjnych),

- włączenie do systemu bloków typu "learning" pozwalających zwiększać stopień "autonomiczności" systemu. UWAGA: włączanie bloków "learning" może okazać się analogiczne do innych systemów zewnętrznych i wtedy konwertuje się to w temat współpracy z aplikacjami zewnętrznymi.

4. Bibliografia

[1] ETSI GR ENI 017, Experiential Networked Intelligence (ENI); Overview of Prominent Control Loop Architectures; Dostępne online: https://www.etsi.org/deliver/etsi_gr/ENI/001_099/017/02.01.01_60/gr_ENI017v020101p.pdf

[2] Udemy, Certified Kubernetes Administrator with Practice Tests, Dostępne online: https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/

[3] Github, 0x41gawor; Dostępne online: https://github.com/0x41gawor/pdmgr/blob/master/minikube-dev-env.md

[4] Github, 0x41gawor; Dostępne online: https://github.com/0x41gawor/pdmgr/blob/master/learning-path.md

[5] Github, 0x41gawor; Dostępne online: https://github.com/0x41gawor/pdmgr/blob/master/crd-playground-lab.md