Politecnico di Torino

Corso di Laurea in Ingegneria informatica

Tesi di Laurea

Protezione automatica basata su policy nel computing continuum

Relatori
prof. Fulvio Valenza
prof. Riccardo Sisto
dott. Francesco Pizzato
dott. Daniele Bringhenti

Candidato
Salvatore Tartaglione

Academic Year 2023-2024

Sommario

Il Cloud computing è una tecnologia sempre più popolare tra le aziende, in quanto consente loro di richiedere servizi remoti rapidamente, senza dover costruire o gestire l'infrastruttura IT necessaria, affidandosi invece a un provider che si occupa di tutto. In questo contesto, le risorse computazionali sono state raggruppate in grandi data center per facilitare la loro gestione. Tuttavia, con l'aumento dei dispositivi IoT e l'incremento dei dati generati, l'"Edge computing" è emerso come un nuovo paradigma in cui le risorse non sono più centralizzate, ma distribuite e collocate il più vicino possibile alle sorgenti di dati e agli utenti finali, migliorando la velocità di elaborazione e riducendo la latenza. L'ulteriore evoluzione di questo concetto è il "Liquid computing", un paradigma che consente una condivisione flessibile delle risorse creando un pool dinamico di risorse che possono essere scambiate secondo necessità. Il progetto FLUIDOS ha come obiettivo l'implementazione del Liquid computing per sfruttare la potenza inutilizzata nell' Edge, creando un sistema scalabile e dinamico che colma il divario tra Edge e Cloud computing. In questo ambiente, un nodo Provider offre servizi, mentre un nodo Consumer li richiede. Attraverso una transazione, il Consumer concorda quali servizi intende trasferire al Provider. Tuttavia, proteggere l' Edge, dove le risorse sono distribuite più vicino alla sorgente dei dati, presenta nuove sfide di sicurezza. A differenza degli ambienti Cloud centralizzati, i nodi dell' Edge sono esposti a una maggiore varietà di minacce, a causa della loro vicinanza agli utenti finali e ai dispositivi nella rete. Inoltre, i confini di ciascun cluster diventano dinamici, poiché possono espandersi su più provider. Questo comporta una maggiore complessità, poiché un singolo provider può servire più utenti, rendendo necessari robusti meccanismi di isolamento per gestire i vari carichi di lavoro. Questa esposizione richiede controlli rigorosi per garantire e gestire lo scambio di risorse e dati nel continuum. Il contributo di questa tesi è progettare un sistema automatizzato capace di applicare meccanismi di sicurezza che rendano sicure le connessioni tra i servizi trasferiti dal Consumer e i servizi del Provider. Per comprendere meglio il ruolo del sistema automatizzato all'interno di FLUIDOS, è utile riassumere il processo del progetto. Il processo FLUIDOS inizia quando il nodo Consumer identifica i provider idonei che soddisfano i requisiti di risorse e servizi. Una volta selezionato un provider, il Consumer e il Provider stabiliscono un accordo di condivisione delle risorse, formalizzato in un contratto che include le richieste di intento del Consumer e gli intenti di autorizzazione del Provider. Le richieste di intento del Consumer specificano le connessioni necessarie a supportare i suoi servizi trasferiti, mentre gli intenti di autorizzazione del Provider controllano e monitorano queste connessioni per garantire il rispetto delle politiche di sicurezza. Un processo automatizzato di armonizzazione allinea questi intenti, risolvendo eventuali discrepanze, mentre un processo di traduzione li converte in

politiche di rete Kubernetes che vengono applicate per rendere le comunicazioni fra i servizi sicure.

Ringraziamenti

Ringrazio il prof. Fulvio Valenza, il prof. Riccardo Sisto , il dott. Francesco Pizzato e il dott. Daniele Bringhenti per avermi dato l' opportunità di svolgere questa tesi che mi ha appassionato molto e per avermi guidato nello svolgimento e nella stesura. Un ringraziamento speciale va ai miei genitori, i miei zii ed i miei nonni che mi hanno sempre supportato sia nel mio percorso accademico che nel mio percorso di vita, rendendomi possibile il raggiungimento di quest' obbiettivo e aiutandomi nei vari problemi riscontrati nel percorso. Volevo ringraziare sopratutto Giovanni, che mi ha accompagnato in questa avventura da fuori sede, insieme abbiamo condiviso gioie e affrontato difficoltà sia di vita che di università. Volevo ringraziare i miei amici Giuseppe, Raffaele e Mario per esserci sempre stati e per le belle serate insieme. Volevo ringraziare tutti i miei amici conosciuti durante questo percorso, in particolare Salvatore, Gabriele, Yonas, Francesco, Nicola ,Chiara, Luana, Federico, Davide, Noemi, Dario, Sante, per i bei momenti trascorsi insieme.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Il **Cloud computing** è una tecnologia sempre più usata dalle aziende in quanto permette, in modo immediato, di richiedere servizi remoti, eliminando la necessità di costruire e gestire l' infrastruttura IT relativa ad essi, affidandosi ad un **provider** che li gestisce. In tale ambito le risorse computazionali sono centralizzate presso grandi data-center gestite dai provider. Tuttavia, con l' emergere di nuovi dispositivi IoT e con l' aumento del volume dei dati generati è nata una nuova tecnologia chiamata **Edge computing**. A differenza del Cloud, le risorse offerte non sono **centralizzate**, ma sono poste il più vicino possibile alla sorgente dei dati e all' utente finale, per migliorare la velocità di elaborazione dei dati e diminuire la latenza. L' affermazione di queste due tecnologie ha fatto emergere nuovi concetti legati ad esse, in particolare il **Liquid computing** risulta essere un paradigma che consente una condivisione **fluida** delle risorse tra diversi domini amministrativi, generando così un pool di risorse continuo che può essere preso in prestito o ceduto in modo dinamico. A tale tecnologia è associato il concetto di "liquido" , in quanto la computazione viene percepita come un qualcosa di "continuo" che si può muovere facilmente come un fluido, senza alcun vincolo, permettendo alle applicazioni di eseguire parte del calcolo dove ha più senso in termini di costi, latenza o efficienza. La computazione può quindi spostarsi da un nodo ad un altro in modo fluido e continuo, ad esempio una persona potrebbe eseguire un task su un determinato dispositivo, in seguito cambiare dispositivo e continuare l' esecuzione del task , che si troverà nello stato in cui era nel dispositivo precedente. Il progetto europeo **FLUIDOS** ha come obbiettivo l' implementazione del concetto del **Liquid computing** per cercare di sfruttare l' enorme quantità di calcolo inutilizzata nell' **edge**, cercando di integrare tali dispositivi che faticano ad integrarsi fra loro per formare un **continuum** di calcolo scalabile e sicuro , colmando il divario fra **Cloud** ed **Edge**. In un contesto dove diversi domini amministrativi condividono o ospitano risorse altrui, la sicurezza diventa una priorità per proteggere le risorse sia del dominio ospitante che quelle condivise. In particolare, garantire una comunicazione sicura tra i servizi richiede un'attenzione particolare per evitare che le risorse possano essere vulnerabili ad attacchi o accessi non autorizzati da parte dei domini esterni. Il contributo di questa tesi si concentra sul' implementazione **automatica** di questi meccanismi di sicurezza per offrire delle comunicazioni sicure nel contesto Fluidos.

1.1 Struttura della tesi

La tesi è suddivisa nei seguenti capitoli:

- Capitolo 2: in questo capitolo vengono introdotti i concetti di background. Verranno presentati brevemente il Cloud,l' Edge ed il Liquid Computing. Successivamente verrà descritto Kubernetes, parlando della sua architettura e della rete di Kubernetes ed infine verrà introdotto il progetto FLUIDOS.
- Capitolo 3: in questo capitolo verrà descritto l' obbiettivo della tesi e la soluzione da perseguire rispetto al problema presentato.
- Capitolo 4: in questo capitolo verrà descritto l' approccio adottato fornendo una soluzione di alto livello e concettuale.
- Capitolo 5: in questo capitolo verrà mostrata l' implementazione della soluzione descritta nel capitolo precedente.
- Capitolo 6: in questo capitolo verrà mostrato un caso d' uso rilevante per mostrare le funzionalità della soluzione adottata.
- Capitolo 7: in questo capitolo verranno esposte le conclusioni del lavoro svolto e delineate le prospettive future.

Capitolo 2

Background

2.1 Kubernetes

2.1.1 Introduzione al capitolo

In questo capitolo verrà delineato il contesto operativo del progetto FLUIDOS, con un'introduzione ai concetti di **Cloud computing**, **Edge computing** e **Liquid computing**. Successivamente, verrà esaminato l'orchestratore **Kubernetes** e il suo ruolo all'interno del progetto. Infine, si procederà con una presentazione dettagliata delle risorse e delle caratteristiche principali di Fluidos.

2.1.2 Cloud Computing

In passato le aziende gestivano le richieste per i propri servizi online tramite propri server, tuttavia con la rapida espansione di Internet e il conseguente aumento del numero di richieste da gestire, comprare più server poteva sembrare la soluzione più ovvia al problema. Si osservò tuttavia che , mediamente , il carico di lavoro di ogni server non veniva sfruttato appieno, comportando uno spreco di energia per ogni server in funzione. Tali aziende pensarono di fornire come servizio quel carico di lavoro inutilizzato ad aziende più piccole, che non potevano permettersi la gestione e l' acquisto dei server. In questo contesto nasce l' idea del **Cloud Computing** come un modello innovativo per la fornitura di tre tipi di servizi:

- **Hardware as a service**: viene offerto lo spazio fisico dove il richiedente del servizio può posizionare i propri server.
- **Infrastracture as a service**: viene offerta l' infrastruttura hardware al richiedente.
- **Platform as a service**: viene offerta la piattaforma software dove il richiedente può fornire servizi.
- **Software as a service**: viene offerto un software gestito dal fornitore per il richiedente.

In tale ambito si fa largo uso del concetto di **virtualizzazione** per permettere ai diversi servizi dei richiedenti di essere in esecuzione sullo stesso server in modo isolato, come se fossero su server separati. La virtualizzazione consente un migliore utilizzo delle risorse hardware riducendone i costi e favorendo la **scalabilità** delle applicazioni.

2.1.3 Edge Computing e Liquid Computing

In una classica architettura di **Cloud computing**, i dati vengono archiviati ed elaborati in maniera centralizzata, all'interno di un **data center**. Questo approccio presenta delle limitazioni per le applicazioni che richiedono requisiti stringenti in termini di latenza e prestazioni in tempo reale. L'**Edge computing** mira a superare tali vincoli, spostando l'elaborazione dei dati verso l'**Edge**, ossia il più vicino possibile al punto in cui i dati vengono generati e utilizzati. In questo modo, le applicazioni time-sensitive possono funzionare in maniera ottimale, poiché riducendo la distanza tra l'unità di elaborazione e la fonte dei dati, si abbassa sensibilmente la latenza. I concetti di **Edge** e **Cloud** non si escludono a vicenda, sono stati sviluppati per scopi diversi, ma se usati insieme possono apportare vantaggi per le applicazioni che necessitano di entrambe le tecnologie. Il **Liquid computing** è un paradigma che mira a sfruttare i vantaggi offerti delle due tecnologie. Il suo obbiettivo è quello di creare un infrastruttura computazionalmente flessibile che permette di distribuire le risorse di elaborazione, archiviazione e di rete fra il cloud e l' edge in base alle esigenze specifiche dell' applicazione. Tale soluzione permette quindi alle applicazioni di sfruttare la potenza del Cloud per elaborazioni che richiedono molta potenza di calcolo, ma allo stesso tempo permette di sfruttare la vicinanza delle risorse dell' Edge computing se c'è bisogno di elaborazioni a bassa latenza.

2.1.4 Containerizzazione

Prima di introdurre **Kubernetes** risulta essenziale parlare di una tecnica di **virtualizzazione** che sarà usata in Kubernetes, ossia la **containerizzazione**. E' una forma di **virtualizzazione leggera** che permette a più applicazioni di essere eseguite sulla stessa istanza, garantendo l'isolamento dei processi e il controllo delle risorse. Attraverso i container, è possibile limitare e gestire in modo preciso l'accesso ai processi rispetto alla CPU, alla memoria e allo spazio su disco, assicurando un utilizzo efficiente e sicuro delle risorse condivise. Risulta essere un concetto importante nell' ambito Cloud in quanto permette di eseguire su un server del provider diverse applicazioni in modo isolato. Una delle tecnologie di containerizzazione più usate è **Docker**.

2.1.5 Panoramica di Kubernetes

Kubernetes [1] è un sistema open-source utilizzato per eseguire il **deployment**, **scalare** e gestire **applicazioni containerizzate** ovunque. Tale sistema permette quindi di adattare le applicazioni secondo le esigenze che sono in continuo cambiamento, facilitando la gestione delle applicazioni e gestendo il loro fallimento in

modo automatico, difatti è molto utilizzato in ambito cloud in quanto permette di **orchestrare** le applicazioni **cloud native**. In Kubernetes le **API** sono separate dall' implementazione, difatti possiamo creare un oggetto in K8s che poi potrà essere implementato a seconda del contesto dal cloud provider. K8s ha un approccio totalmente **dichiarativo** , in tale approccio si descrive la logica senza specificare il control flow. La struttura delle risorse in Kubernetes segue il suo approccio dichiarativo , nella quale bisogna specificare i seguenti componenti in un **manifesto**:

- **apiVersion** ed **item**: che identificano il **tipo** dell' oggetto.
- **spec**: specifica lo stato desiderato dell' oggetto.
- **status**: specifica lo stato corrente, spesso fornito al sistema.
- **labels** and **annotations** che fungono da metadati.

```yaml
apiVersion: VERSION # eg. v1
kind: TYPE # eg. Pod
metadata:
  name: NAME # eg. nginx
  annotations:
    # eg. k8s.v1.cni.cncf.io/networks: multus1
    test: true
  labels:
    color: blue # eg. release: stable
spec:
    # Expected status of the resource
status:
     # Current status of the resource
```

Figura 2.1. Risorsa Kubernetes

Kubernetes adotta un approccio basato su un **ciclo di controllo** (control-loop) che permette di far si che lo stato di un oggetto converga alla specifica desiderata della risorsa dichiarata nel manifesto. Per raggiungere tale obbiettivo, K8s usa un **controllore** che osserva lo stato condiviso del cluster ed effettua i cambiamenti per raggiungere lo stato **desiderato** della risorsa. L' **API server** è il cuore del piano di controllo in quanto espone le **API** per permettere gli utenti, parti differenti del cluster e componenti esterni di comunicare fra loro. Esso permette di fare query e manipolare lo stato degli oggetti in Kubernetes.

Pod e Deployments

Le applicazioni sono eseguite in uno o più **Pod**, il quale è un set di uno o più **container** che sono strettamente legati fra loro e rappresenta l' unità di esecuzione fondamentale per le applicazioni Kubernetes. Un Pod è un entità **effimera**, possono

essere aggiornati, eliminati e rischedulati in base al loro stato. Tuttavia quando si vuole eseguire un applicazione su un Cluster, non si definisce un solo Pod, ma si fa uso della risorsa **deployment** che permette di gestire il ciclo di vita di un applicazione specificando ad esempio l' immagine dell' applicazione ed il numero di pod sulla quale eseguire l' applicazione. Tale risorsa quindi permette di avere a disposizione più pod dove l' applicazione può essere eseguita. Questa risorsa garantisce che l'applicazione rimanga disponibile anche in caso di fallimento di un singolo Pod, avviando l'applicazione su altri Pod.

Namespace e RBAC

In un cluster Kubernetes possiamo dunque definire un set di risorse differenti, risulta quindi fondamentale avere una **divisione logica** di quest' ultime grazie all' utilizzo della risorsa **namespace**. In alcuni scenari risulta fondamentale vincolare l' accesso e la visibilità delle risorse interne ad un namespace tramite il controllo degli accessi basato su ruoli **RBAC**. Il controllo degli accessi basato su ruoli di K8s controlla quali oggetti possono eseguire determinate **azioni** su altri oggetti. Per ottenere ciò, si fa uso della risorsa **cluster role** che permette di specificare le azioni concesse su di una risorsa. Successivamente tramite un' altra risorsa detta **cluster role binding**, viene associato il **cluster role** ad un oggetto interno al cluster, che potrà quindi eseguire le azioni definite nel cluster role sull' oggetto definito in quest'ultimo.
Di solito quando si vuole concedere un' autorizzazione ad un oggetto o ad un' entità, si crea un identità univoca all' interno del cluster che viene associata a quest' ultimo.

CRD

Se le risorse standard di **Kubernetes** non soddisfano i requisiti, è possibile definire delle risorse costum chiamate **Custom Resource Definition (CRD)**, che permettono di definire risorse personalizzate per estendere le funzionalità native di Kubernetes. Tuttavia, chi crea una risorsa personalizzata deve anche fornire l'implementazione di un **Operatore**: un componente responsabile di monitorare i cambiamenti della risorsa e gestire la logica necessaria per eseguire le operazioni previste dalla risorsa stessa.

2.1.6 Architettura Kubernetes

L' architettura software di Kubernetes [2] mostrata in figura 2.2 mostra come i vari componenti interagiscono all' interno di un cluster. L' architettura risulta essere composta da: **componenti master** e **componenti dei nodi**.

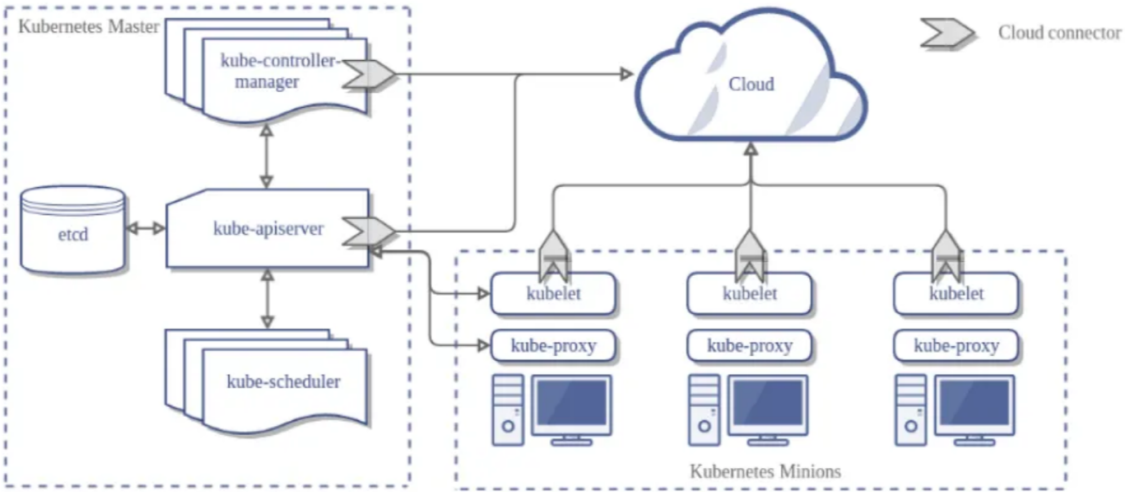


Figura 2.2. Architettura Kubernetes

Componenti master

Questi componenti, sono responsabili di tutte le decisioni globali sul cluster ed includono:

- **kube-apiserver**: **front-end** per il control-plane di Kubernetes, esso espone le API di K8s.
- **kube-scheduler**: monitora i nuovi pod generati e seleziona dei nodi sulla quale eseguire i pod.
- **etcd**: database distribuito di tipo **chiave-valore** che permette in modo affidabile di memorizzare i dati nel cluster.
- **kube-control manager**: controllore K8s che implementa un ciclo di controllo osservando lo stato condiviso del cluster tramite il **kube-apiserver** effettuando cambiamenti con lo scopo di convergere allo stato desiderato.

Componenti nodi

Questi componenti che sono eseguiti su ogni nodo includono:

- **kubelet**: un agente che mantiene sotto controllo lo stato di ogni pod verificandone la loro vitalità e sanità informando il **kube-apiserver**.
- **kube-proxy**: gestisce le regole di rete sull' host.

2.2 Rete Kubernetes

La rete in **Kubernetes** [3] gioca un ruolo fondamentale per garantire la comunicazione tra i vari componenti del cluster. La rete di Kubernetes crea un ambiente in cui i dati possono spostarsi liberamente ed in modo efficiente attraverso la rete definita dal software. Una caratteristica fondamentale della rete K8s è la sua struttura di **rete piatta**, ossia che tutti i componenti possono connettersi fra loro senza dipendere da un altro hardware, cioè un pod può comunicare con un altro pod anche se sono in esecuzione su due nodi differenti. Quindi Kubernetes permette di astrarre la complessità della rete, permettendo agli sviluppatori di concentrarsi sulla creazione e gestione delle applicazioni anziché sulla gestione della rete stessa. In K8s ci sono quattro tipi di comunicazione di base:

- **rete da container a container**: nelle configurazione di base i container interni ad uno stesso Pod possono comunicare fra loro usando il localhost in quanto appartenenti allo stesso namespace di rete.
- **rete pod-to-pod**: ogni Pod in Kubernetes ha il proprio indirizzo IP, consentendo la comunicazione diretta fra Pod indipendemente dal nodo in cui essi risiedono.
- **rete tramite servizio**: ogni Pod in Kubernetes ha un proprio indirizzo IP, ma essendo entità effimere, il loro indirizzo IP potrebbe cambiare se per esempio un Pod venisse riavviato o rischedulato su un altro nodo, per tale motivazione in K8s si fa uso dei **Service** che rappresentano un meccanismo che permette di esporre applicazioni che sono in esecuzione su uno o più Pod, su un indirizzo IP che non cambia, il Service quindi rappresenta un **punto di accesso stabile**. La figura 2.3 mostra come il service possa gestire del traffico in ingresso destinato ad un applicazione che è in esecuzione su più Pod.

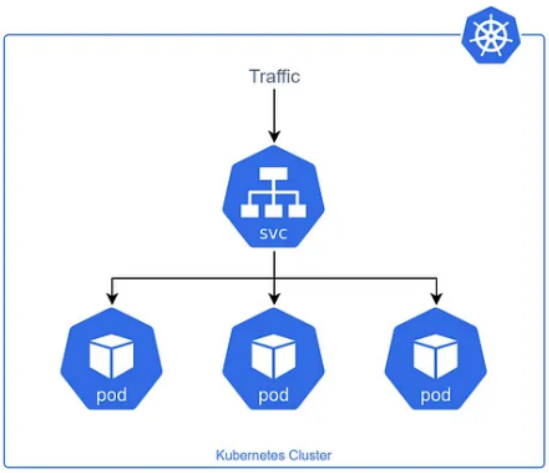


Figura 2.3. Service Kubernetes

Esistono diversi tipi di **Service** che si possono definire:

- **ClusterIP**: espone il servizio internamente al cluster.
- **NodePort**: espone il servizio all'esterno del cluster, mappando una porta statica su tutti i nodi del cluster, che reindirizza il traffico verso il servizio tramite un IP del nodo.
- **LoadBalancer**: espone il servizio esternamente facendo uso del load balancer del cloud provider.

2.2.1 Network policies

Le **Network policies** [4] implementano meccanismi di **firewall** che consento agli amministratori di rete di regolare l' accesso fra i differenti pod in K8s. Tuttavia, le Network Policies non sono fornite di default in Kubernetes: devono essere configurate e gestite dall'amministratore di rete del cluster. Per attivare queste politiche di rete, è possibile utilizzare un plugin **CNI** (Container Network Interface). La struttura di una K8s Network policy è composta da:

- **metadata**: campo in cui è possibile specificare il **nome** della network policy ed il **namespace** dove la network policy verrà applicata.
- **spec**: campo nella quale è possibile specificare tramite **podSelector** a quali pod la network policy sarà applicata.
- **policyTypes**: campo che può assumere solo due valori: **ingress** e/o **egress** a seconda della tipologia di network policy da applicare.

Le regole di **ingress** o **egress** possono essere applicate attraverso i rispettivi campi, utilizzando gli stessi selettori, come ad esempio:

- **ipBlock**: specifica un intervallo di indirizzi IP per cui il traffico è consentito.
- **namespaceSelector**: consente di indicare un namespace per il quale è possibile controllare il traffico.
- **ports**: definisce le **porte** e i **protocolli** per i quali è autorizzato il traffico.

Esempio network policy

Come mostrato in figura 2.4, questa network policy si applica ai pod con le label "app" che hanno valore "pod2" nel namespace "default". Tale esempio mostra una network policy sia di ingress che di egress, in particolare i pod selezionati potranno ricevere del traffico dai pod che hanno la label "app" con valore "pod1" ed inviare del traffico ai pod con la stessa coppia label-valore.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: pod2
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: pod1
  egress:
    - to:
      - podSelector:
          matchLabels:
            app: pod1
```

Figura 2.4. Esempio Kubernetes network policy

2.2.2 CNI

La **CNI** o **Container Network Interface** [5] è una specifica standardizzata e un insieme di API che definiscono come i plug-in di rete devono abilitare la rete dei container. Le CNI possono assegnare indirizzi IP, creare namespace di rete, configurare percorsi di rete e così via per abilitare la **connessione pod-to-pod**.

Calico

Calico [6] è una CNI che rappresenta una soluzione di networking di livello L3/L4 e sicurezza che consente alle applicazioni Kubernetes e alle applicazioni non Kubernetes/legacy di comunicare in modo fluido e sicuro. In Kubernetes di default tutti i pod all'interno di un cluster possono comunicare tra loro senza restrizioni. Con Calico, è possibile applicare Network Policies di default-deny per bloccare tutte le comunicazioni e poi definire ulteriori policy di rete per abilitare in modo dettagliato solo le comunicazioni desiderate.

2.3 Progetto Fluidos

Il progetto **FLUIDOS (Flexible, scaLable, secUre, and decentralIseD Operating System)** [7] ha l'obiettivo di sfruttare l'enorme capacità di elaborazione

inutilizzata presente ai margini della rete, distribuita tra dispositivi eterogenei che spesso faticano a integrarsi tra loro e a formare in modo coerente un **continuum** di elaborazione senza interruzioni colmando il divario fra **edge** e **cloud** computing. Il progetto si baserà su sistemi operativi consolidati e su un orchestratore open source ossia **Kubernetes** permettendo la condivisione delle risorse tramite procedure di accordo e aggregazione. Tale progetto farà uso di algoritmi di ottimizzazione per minimizzare i costi ed i consumi energetici nel continuum computazionale, usando efficientemente le risorse. FLUIDOS mira a creare un ecosistema che faciliti la collaborazione tra diversi attori nell'ambito dei servizi e delle applicazioni edge. L'obiettivo principale è quello di sviluppare un mercato aperto, dove servizi edge e app possano essere scambiati e utilizzati senza dipendere da specifici fornitori di cloud. Questo approccio non solo stimola l'innovazione, ma promuove anche l'autonomia digitale europea, incoraggiando la crescita di un panorama tecnologico più diversificato e competitivo.

Liqo

Per raggiungere i suoi obbiettivi, FLUIDOS si avvale di **Liqo** [8] in quanto quest'ultimo facilita l'integrazione tra dispositivi eterogenei, permettendo loro di collaborare formando un vero e proprio continuum di elaborazione distribuita. **Liqo** è un progetto open-source che permette di creare topologie **multi-cluster** Kubernetes in modo dinamico e senza interruzioni, supportando infrastrutture eterogenee on-premise, cloud ed edge. Il processo che connette due cluster è chiamato **peering**, mentre il processo di allocazione di una risorsa su un **cluster remoto** è detta **offloading**. In tale contesto si definisce **consumer** colui che usa le risorse del **cluster remoto**, mentre si definisce **provider** colui che le fornisce ai consumer. Liqo gestisce il peering tra due cluster configurando automaticamente le VPN necessarie per connetterli. Per raggiungere tale obbiettivo Liqo definisce una **CRD** chiamata **TunnelEndpoints** che in un contesto **multi-cluster** permette di automatizzare e gestire la connessione fra due cluster differenti. In particolare permette di stabilire dei **tunnel di rete** che consentono la comunicazione sicura fra risorse locali del cluster consumer e le risorse offlodate di tale cluster sull' ospitante. Tale risorsa è creata sia lato **provider** che lato **consumer**(vedi 2.5). Contiene informazioni rilevanti come:

- **ClusterId e Cluster Name:** l' Id ed il nome del cluster con la quale è avvenuto il **peering**. Quindi sul provider conterrà l' ID del consumer e viceversa.
- **Endpoint locali e remoti**: include indirizzi IP publici o privati su cui i tunnel devono essere stabiliti.

```
Spec:
  Backend Type:  wireguard
  backend_config:
    Port:        31107
    Public Key:  R+7/AsrPTo46xZZ9DZp63kjM9sQueksEV674mG0aLSM=
  Cluster Identity:
    Cluster ID:              ec08ea63-de72-42d6-a0ad-d364371f4e94
    Cluster Name:            fluidos-provider
  Endpoint IP:               172.18.0.5
  Local External CIDR:       10.97.0.0/16
  Local NAT External CIDR:   10.99.0.0/16
  Local NAT Pod CIDR:        10.98.0.0/16
  Local Pod CIDR:            192.168.0.0/16
  Remote External CIDR:      10.97.0.0/16
  Remote NAT External CIDR:  10.99.0.0/16
  Remote NAT Pod CIDR:       10.98.0.0/16
  Remote Pod CIDR:           192.168.0.0/16
```

Figura 2.5. Esempio tunnelEndpoints

Protocollo REAR

Il progetto FLUIDOS fa uso del protocollo **REAR** (Resource Advertisement and Reservation) [9], per assicurare che le risorse computazionali siano pubblicizzate e rese disponibili in modo standardizzato e sicuro all'interno del continuum. In questo contesto, i nodi che offrono risorse si comportano come **provider**, mentre i nodi che le richiedono sono i **consumer**. REAR facilita la gestione dinamica e collaborativa delle risorse tra dispositivi eterogenei, favorendo un'integrazione efficiente tra edge e cloud. Tale protocollo è basato sul concetto di **Nodo**, che rappresenta un entità computazionale sotto il controllo della stesso dominio amministrativo. Un nodo può includere una o più macchine ed è modellato con un set di primitive che permette di nasconderne i dettagli interni consentendo di pubblicizzare le caratteristiche più rilevanti. Ogni nodo Rear è composto da un **importer** che scopre e filtra le risorse disponibili per un consumer e da un **exporter** che pubblicizza le risorse del nodo agli altri membri del continuum fungendo da provider per altri nodi. Inoltre un altro componente, chiamato **contract manager** terrà traccia delle risorse acquistate sotto forma di contratto una volta che la transazione è stata effettuata.
Il modello dei dati di REAR prevede la definizione di due termini: **Flavor** e **FlavorType**. Il primo definisce le diverse informazioni necessarie per descrivere diversi tipi di risorse, mentre il FlavourType rappresenta il tipo di risorsa che verrà acquistata nella transazione.

REAR workflow

Per comprendere al meglio come avviene una transazione fra un **provider** ed un **consumer** in FLUIDOS è utile ripercorrere il workflow del protocollo REAR. Come mostra la Figura 2.6 il protocollo definisce diversi messaggi:

- **List Flavor**: messaggio inviato dal consumer per ricercare dei flavors offerti dai provider. Una volta collezionati diversi flavors di differenti provider, il consumer ne sceglierà uno.

- **Reserve Flavor**: tale messaggio permette al consumer di prenotare le risorse presso il provider relativo al flavor scelto. Il provider verificherà se il flavor scelto è ancora disponibile e risponderà al consumer con un riassunto del processo di prenotazione inviando l' id della transazione e una deadline per acquistare tale flavor.

- **Purchase Flavor**: messaggio inviato dal consumer per finalizzare l' acquisto del flavor offerto. Il consumer invierà anche l' ID del flavor che sta acquistando. Se la transazione va a buon fine, una copia di un contratto sarà inviata al consumer. Il contratto conterrà tutte le informazioni della transazione.

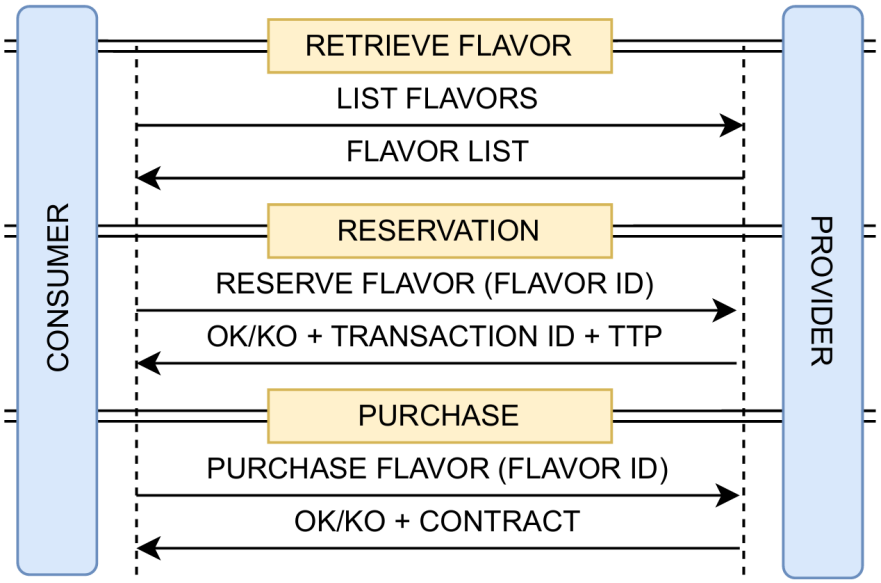


Figura 2.6. workflow del protocollo REAR

2.3.1 Transazione in Fluidos

La transazione in Fluidos segue il workflow del protocollo REAR illustrato in precedenza, implementando i vari componenti della transazione come CRD. Di seguito vengono presentati questi componenti, descrivendone il ruolo all'interno del flusso di una transazione.

Scelta dei candidati al peering

Ogni **provider** che può offrire delle risorse computazionali, nel contesto **Fluidos**, le offre tramite una **CRD** chiamata **Flavour**. Tale risorsa include campi come:

- **owner**: ossia il possessore di tale flavour, in questo campo è anche possibile specificare l' id del possessore.

- **price**: ossia prezzo di tale servizio.

- **characteristics**: caratteristiche di computazione, quali cpu,gpu etc. offerte.
- **networkAuthorizations**: autorizzazioni di rete concesse su quella porzione di servizio offerto.

Quando un consumer vuole definire delle risorse che intende offloadare presso un provider, fa uso di un **solver** per ricercare una soluzione disponibile per la propria richiesta. Qui entra in gioco il **REAR manager**, ossia un componente che riceve dal solver le soluzioni proposte e le trasforma in richieste di servizi e risorse. Il Rear Manager avvia una fase di **Discovery** per trovare dei candidati al peering che possano soddisfare le richieste del consumer. I candidati al peering sono definiti in FLUIDOS come **PeeringCandidates**, ossia una CRD che definisce gli stessi campi dei **flavour** che sono espressi nel provider. A tal punto, il Rear Manager interagendo con un processo di **verifica** sul **consumer** sceglierà uno dei candidati al peering scegliendo il candidato che offre dei servizi che siano conformi alle richieste di rete e di computazione del consumatore.

2.3.2 Prenotazione e allocazione delle risorse

Una vola scelto il candidato al peering, il **Rear manager** creerà una **reservation** che permette di **prenotare** le risorse offerte dal provider definite nel **peering candidate**. La **reservation** è una CRD definita in FLUIDOS che definisce i seguenti campi :

- **seller**: informazioni come l' Id , l' IP ed il dominio del nodo provider.
- **buyer**: informazioni come l' Id , l' IP ed il dominio del nodo consumer.
- **peeringCandidates**: candidato al peering scelto e namespace in cui esso si trova.
- **type**: tipo di risorsa da riservare.

Una volta prenotate le risorse, il consumer allocherà le risorse sul lato del provider completando la transazione, facendo uso della CRD **allocation** che contiene due campi fondamentali:

- **intentId**: id dell' intento, ossia della richieste che il consumer effettua verso il provider.
- **contractId**: id del **Contratto** creato dalla transazione fra consumer e provider.

Il **Contratto** è una risorsa fondamentale in FLUIDOS, è una **Costum Resource Definition K8s** che serve a definire e gestire la negoziazione e lo scambio di risorse fra nodi e permette inoltre di specificare delle richieste del consumer. I campi fondamentali di tale risorsa sono:

- **Buyer**: identità del nodo acquirente.

- **ExpirationTime**: Tempo di scadenza del contratto.
- **Seller**: Identità del nodo venditore.
- **TransactionID**: ID della transazione a cui appartiene il contratto.
- **NetworkRequests**: contiene uno o più identificativi di **ConfigMap** che conterraneo le richieste di rete , sotto forma di **intenti** [4.1.1], del consumer.

Una volta che le risorse saranno allocate, il **consumer** potrà eseguire l' offload delle proprie risorse, che verranno riflesse tramite **Liqo** sul provider. Il consumer dovrà definire le proprie **richieste di rete** tramite una **config-map** il cui nome sarà contenuto nel contratto che verrà riflesso nel provider per permettere a quest' ultimo di applicare le policy di rete sulle risorse offlodate.

Capitolo 3

Obbiettivo della tesi

3.1 Contesto e problema esistente

Nel contesto dell'infrastruttura FLUIDOS, che permette a più cluster di interagire e condividere risorse computazionali, la sicurezza delle comunicazioni tra i nodi diventa un aspetto critico. In un sistema dove i cluster (sia locali che remoti) collaborano per il bilanciamento delle risorse, la protezione delle informazioni e la limitazione degli accessi sono requisiti essenziali. Attualmente, molte implementazioni di reti distribuite non riescono a garantire il necessario isolamento di rete tra cluster e si affidano in larga misura a configurazioni manuali di network policies, con un elevato rischio di configurazioni errate. In particolare, l'interazione tra consumer e provider , in FLUIDOS, avviene attraverso il processo di offloading grazie a Liqo, in cui il cluster consumer invia risorse e servizi al cluster provider. Tuttavia, questa configurazione espone entrambi i cluster a rischi di sicurezza : il consumer potrebbe voler limitare l'accesso ai propri servizi eseguiti in remoto, mentre il provider deve garantire che le risorse condivise non compromettano il funzionamento del proprio sistema né siano esposte a traffico non autorizzato. Le configurazioni manuali richiedono un significativo impegno amministrativo e possono facilmente risultare complesse e soggette a errore, specie in ambienti multi-cluster dove le interazioni di rete devono essere coordinate in modo preciso. Questo scenario evidenzia la necessità di una soluzione automatizzata per armonizzare e applicare in modo efficiente le regole di rete , riducendo così sia i tempi di gestione che i rischi di configurazione inappropriata.

3.1.1 Obiettivo della soluzione

Questa tesi propone una soluzione automatizzata per garantire la sicurezza delle comunicazioni nel contesto multi-cluster di FLUIDOS, attraverso l'armonizzazione e la traduzione delle richieste di intenti espresse dai consumer e dagli intenti di autorizzazione definiti dal provider. L'obiettivo principale è sviluppare un sistema automatizzato che possa interpretare e conciliare automaticamente le diverse politiche di rete espresse da ciascun nodo, trasformandole in network policies di Kubernetes coerenti e sicure. Nello specifico, il sistema:

- **Verifica** sul lato del consumer quali candidati possono accogliere le richieste del consumer.
- **Armonizza** sul lato del provider automaticamente gli intenti definiti, risolvendo eventuali conflitti.
- **Traduce** sul lato del provider gli intenti armonizzati in Network Policies di Kubernetes, garantendo l'applicazione automatica delle regole di accesso appropriate tra i nodi.

Questa soluzione fornisce un linguaggio di alto livello per definire intenti di accesso e limitazioni di rete, il quale viene poi tradotto in modo automatico in regole di basso livello, eliminando così la necessità di interventi manuali e assicurando la coerenza e la sicurezza delle configurazioni di rete in un contesto FLUIDOS.

Capitolo 4

Approccio

4.1 Introduzione approccio

Durante la fase di offloading il Consumer invia delle richieste di rete al Provider, il quale potrà decidere se accettarle, rifiutarle o renderle conformi con le proprie regole di autorizzazione. La soluzione proposta prevede la definizione delle richieste e delle regole in linguaggio di alto livello, facendo uso di **intenti** [10] con una struttura definita.

4.1.1 Intenti

Per definire tutti i possibili scenari d' uso è possibile far uso di tre tipi di **intenti**:

- **intenti privati**: definiti dal **consumer** in quanto il suo obiettivo è quello di proteggere le comunicazioni interne al proprio cluster locale. Sono relativi alle comunicazioni che avvengono all' interno dello stesso **virtual cluster** e quindi non sono soggetti all' autorizzazione dell' host.

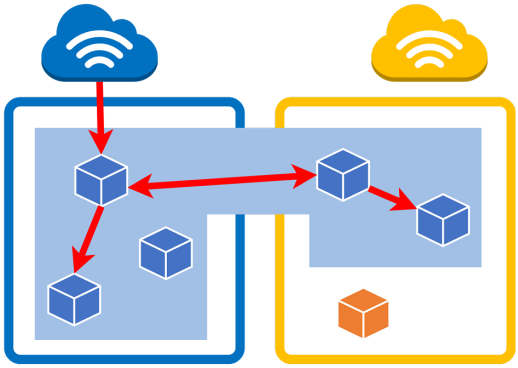


Figura 4.1. Intenti privati

- **intenti di richiesta**: definiti dal **consumer** per proteggere le comunicazioni fra le risorse offloadate sul cluster remoto. Sono relativi a comunicazioni

inter-virtual cluster e quindi sono soggetti al processo di armonizzazione in base agli intenti del cluster ospitante.

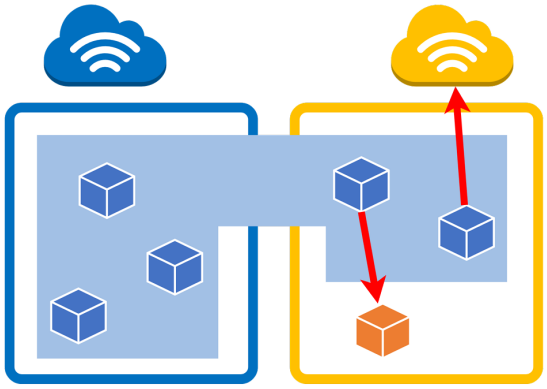


Figura 4.2. intenti di richiesta

- **intenti di autorizzazione**: definiti sia dal **provider** che dal **consumer** per proteggere le proprie risorse dai servizi dell' altro peer o dalla rete esterna. Si suddividono in
  - **Comunicazioni vietate**: comunicazioni che devono essere bloccate o filtrate da tutti gli host.
  - **Comunicazioni obbligatorie**: comunicazioni che vengono inserite dall' host, per esempio per ottenere i log di ogni applicazioni ospitata.

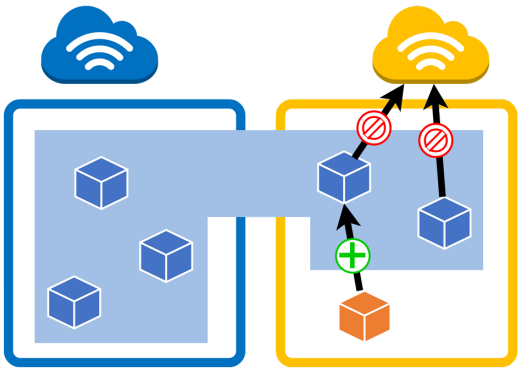


Figura 4.3. intenti di autorizzazione

Struttura intenti

La struttura di un intento è così definita:

from SRC to DST, protocol[:port[-endPort]]

- **SRC** e **DST** possono essere sia pod singoli, che gruppi di pod con la stessa etichetta e possono avere associati un namespace o un indirizzo o un gruppo di indirizzi espresso tramite la notazione CIDR. Il simbolo **"*"** può essere usato per simboleggiare tutti i pod/namespace aventi una certa etichetta o un certo valore.

- **protocol** può assumere il valore di qualsiasi protocollo di trasporto (TCP, UDP, SCTP etc.), inoltre il valore **"ALL"** rappresenta tutti i protocolli.

- **port** può essere una porta o un range di porte.

Inoltre è possibile specificare nel campo **isHostCluster** se la risorsa fa parte dell' **cluster host** (valore **true**) oppure del **cluster remoto** (valore **false**).

4.1.2 Workflow generale

Nel contesto **FLUIDOS**, un **consumer** vorrebbe richiedere ad un **provider** di ospitare alcuni suoi servizi. Il **provider** definisce degli **intenti di autorizzazione** che i servizi del consumer dovranno rispettare quando verranno ospitati sul provider. Tali intenti sono definiti sotto forma di **Flavors**, ossia delle **Costum resource K8s (CRD)** che definiscono configurazioni di rete e di risorse . Il consumer che vuole **offloadare** delle risorse fa uso di un componente chiamato **solver** (anch' esso è una CRD) che permette di analizzare le politiche di rete stabilite dai **provider** e trovare la soluzione che massimizzi l' efficienza, minimizzi i conflitti rispetto alle richieste del consumer e le regole imposte dal provider tramite i flavors. Il **solver** trova diversi **candidati al peering (peering candidates)**, che saranno analizzati da un componente chiamato **verifier** che andrà a scegliere tra i vari **peer** il primo che risulta avere delle autorizzazioni di rete che siano conformi con le richieste del consumer. Una volta trovato il candidato, il solver **prenota** le risorse di quest'ultimo e successivamente **alloca** le risorse concordate, effettuando l' **offloading** di uno o più namespace. Per definire le regole di rete o **intenti di richiesta**, il **consumer** definisce in uno dei namespace da offloadare una **config-map** che conterrà tutti gli intenti di richiesta che dovranno essere presi in considerazione dal provider. Sia dal lato del provider che del consumer, quando il **solver** crea l' **allocazione** delle risorse, si definisce un **contratto** che contiene tutte le informazioni necessarie e concordate fra **consumer** e **provider**. In tale risorsa, è presente un campo che specifica il nome della **config-map** dove si troveranno gli intenti di richiesta del consumer. L' **offloading** delle risorse sarà gestito da **Liqo**, che rifletterà i namespace e i servizi concordati dal consumer al provider. In quest' ultimo sarà in esecuzione un **controller** che , non appena si accorge che un namespace è stato offlodato, legge il contratto stabilito con il consumer, per ottenere gli intenti di richiesta definiti dal consumer. Tali intenti saranno **armonizzati** per risolvere differenti tipi di discordanze rispetto agli intenti di autorizzazione definiti sul provider, successivamente tali intenti saranno **tradotti** in **Kubernetes Network Policies** per abilitare le differenti connessioni. Nei sottoparagrafi successivi verranno mostrati con più dettaglio i componenti descritti in questa prima panoramica ripercorrendo in maggior dettaglio i passi del workflow.

4.1.3 Ricezione degli intenti di richiesta e armonizzazione

Una volta che le risorse vengono offloadate sul **provider**, un **controllore** in esecuzione su di esso, se ne accorge e legge il **contratto** definito durante la transazione fra provider e consumer. Il controllore cercherà nel contratto il campo **networkRequests** che conterrà il nome della config-map definita dal consumer in uno dei namespace offlodati. Tale config-map contiene gli **intenti di richiesta** del consumer, che verranno letti ed armonizzati dal controllore risolvendo i differenti tipi di **discordanze** fra intenti di richiesta e intenti di autorizzazione espressi dal provider. Una volta armonizzati gli intenti, il modulo di traduzione li tradurrà in **Kubernetes Network Policies** per poi applicarli sul cluster. Nelle sezioni successive saranno presentati più nel dettaglio il **controllore**, il **traduttore** e l' **armonizzatore**.

Armonizzatore

Gli intenti espressi dal consumer possono essere in disaccordo con gli intenti espressi dal provider e quindi un componente di **armonizzazione** (interno al **controllore**) si occuperà di armonizzare eventuali intenti discordanti. Ovviamente il cluster ospitante ha il potere decisionale, ossia può decidere quale **intento di richiesta** del consumer può essere applicato o meno, potendone forzare dei nuovi. Per mostrare un' esempio di armonizzazione illustriamo la figura 4.4 che mostra due cluster, Cluster 1 e Cluster 2. Cluster 1 richiede l' offload delle risorse bank_payment e order_placement sul cluster 2, il quale possiede le risorse resource_monitor, database, product_catalogue. I bordi tratteggiati blu simboleggiano namespace differenti, i bordi tratteggiati neri invece simboleggiano i pod che sono in esecuzione per il nome del deployment indicato. I bordi calcati sono invece i bordi dei cluster.

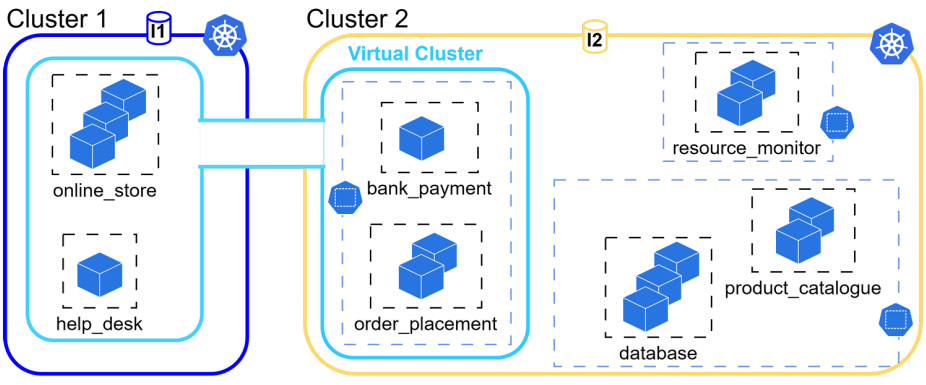


Figura 4.4. Scenario di esempio

Supponendo di avere i seguenti **Request intents**:

- [**R1**]: *from "app:order_placement" to "app:bank_payment", ALL*
- [**R2**]: *from "app:order_placement" to "app:product_catalogue", TCP:80*

- [**R3**]: *from "app:bank_payment" to I2, ALL*

ed il seguente **Authorization intent**:

- [**A1**] : *from any offloaded pods to "app:product_catalogue" , TCP:80*

Il componente di armonizzazione produrrà i seguenti intenti armonizzati:

- [**R1**]: *from "app:order_placement" to "app:bank_payment", ALL*
- [**R2**]: *from "app:order_placement" to "app:product_catalogue", TCP:80*
- [**AR1**]: *from "app:bank_payment" to "app:product_catalogue", TCP:80*

[**R3**] quindi sarà negato, mentre verrà aggiunto l' intento **AR1** armonizzato. La figura 4.5 mostra le comunicazioni , tramite delle frecce bidirezionali, consentite dopo il processo di armonizzazione.

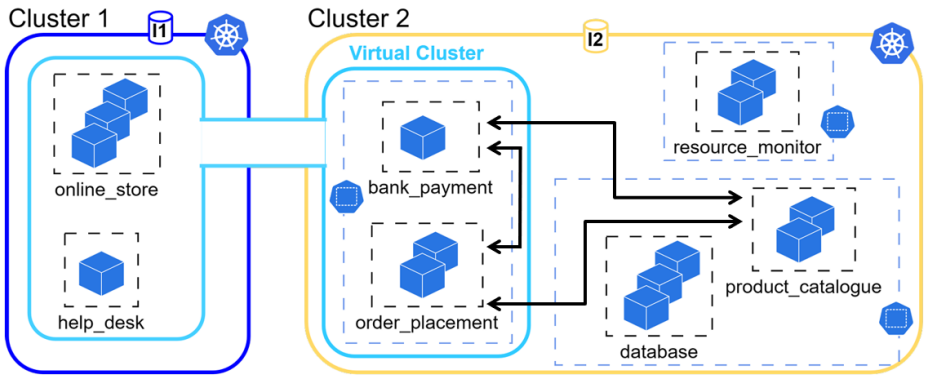


Figura 4.5. Scenario di esempio armonizzato

Traduttore

Il **traduttore** è un componente del controllore che prende in ingresso un set di intenti armonizzato e li traduce in **Kubernetes network policies** di **egress** e di **ingress**. Data l' alta espressività degli intenti (vedi 4.1.1), il traduttore deve riuscire a tradurre tutte le possibili casistiche:

- **From All Pods in a namespace to a specifc Pod in a specific namespace**: in tal caso si vogliono abilitare le comunicazioni di tutti i pod all' interno di un namespace verso un pod specifico in un namespace specifico. Il traduttore dovrà creare network policies di egress per ogni pod sorgente e una policy di ingress per il pod destinazione.

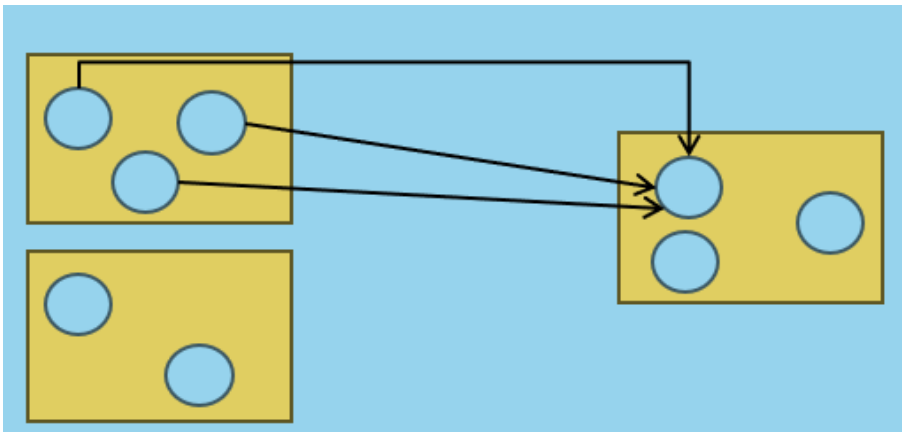


Figura 4.6. Caso 1

- **From a specific Pod in a namespace to All Pods in a specific namespace**: in tal caso si vogliono abilitare le comunicazioni di un pod all' interno di un namespace verso tutti i pod in un namespace specifico. Il traduttore dovrà creare una policy di egress per il pod sorgente e policies di ingress per ogni pod destinazione.

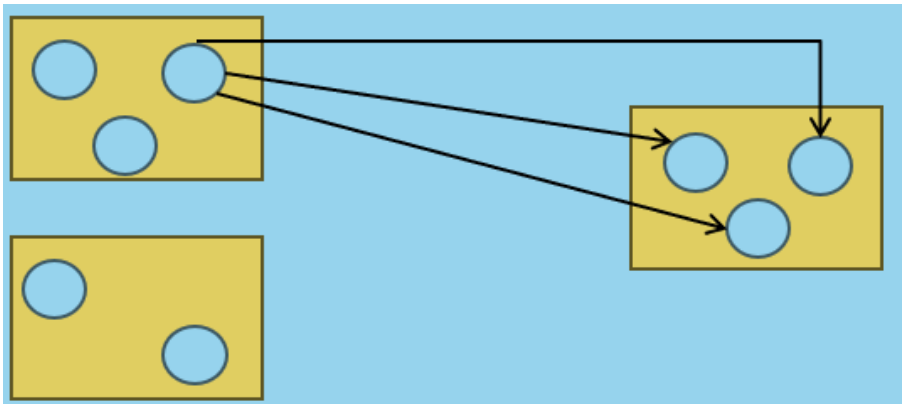


Figura 4.7. Caso 2

- **From All Pods in a namespace to a All Pods in a specific namespace**: in tal caso si vogliono abilitare le comunicazioni di tutti i pod all' interno di un namespace specifico verso tutti i pod in un namespace specifico. Il traduttore dovrà creare policies di egress per ogni pod sorgente e policies di ingress per ogni pod destinazione.

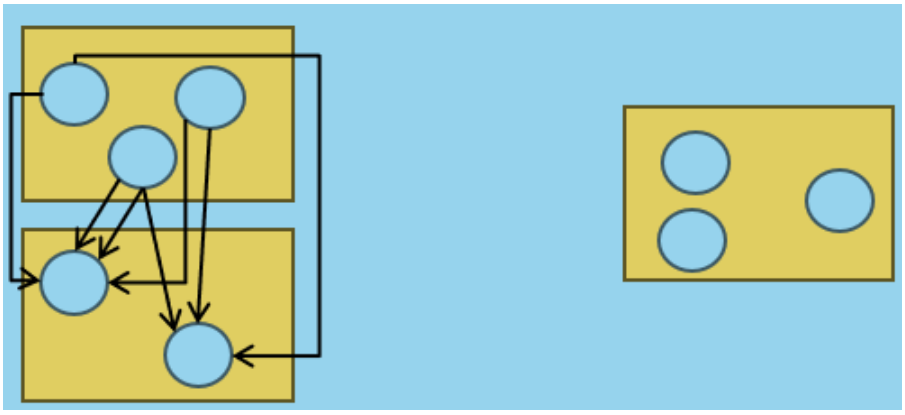


Figura 4.8. Caso 3

- **From All Pods in a namespace to CIDR address**: in tal caso si vogliono abilitare le comunicazioni di tutti i pod all' interno di un namespace specifico verso un indirizzo CIDR. Il traduttore dovrà creare policies di egress per ogni pod sorgente.

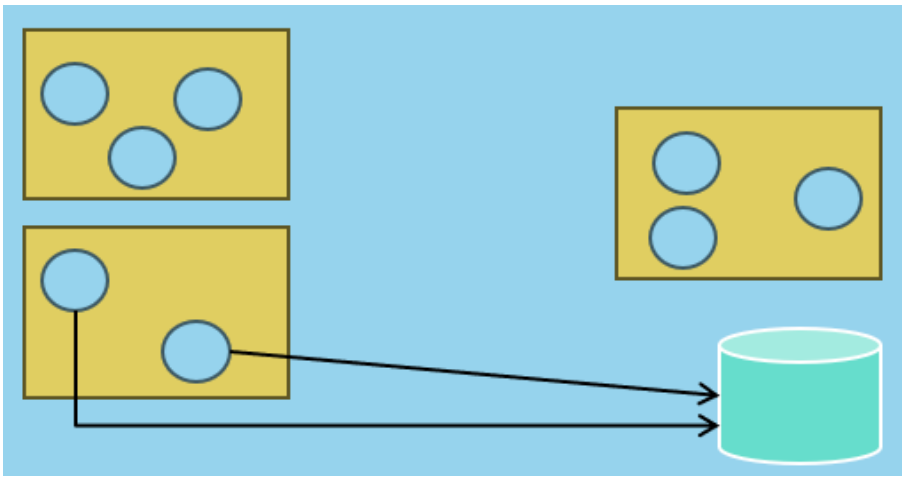


Figura 4.9. Caso 4

- **From Pods with a certain key in all namespace to a specific pod in a namespace**: in tal caso si vogliono abilitare le comunicazioni di tutti i pod con una certa chiave fra tutti i namespace disponibili verso un pod in un namespace specifico. Il traduttore dovrà creare policies di egress per ogni pod sorgente e una policy di ingress per il pod destinazione.

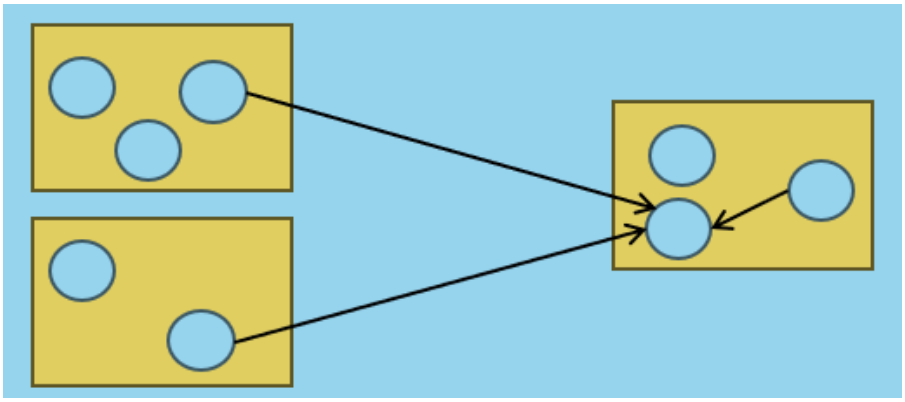


Figura 4.10. Caso 5

- **From a specific Pod to Pods with a certain key in all namespaces**: in tal caso si vogliono abilitare le comunicazioni di un pod specifico verso tutti i pod con una certa chiave fra tutti i namesapace. Il traduttore dovrà creare una policy di egress per il pod sorgente e una policy di ingress per ogni pod destinazione.

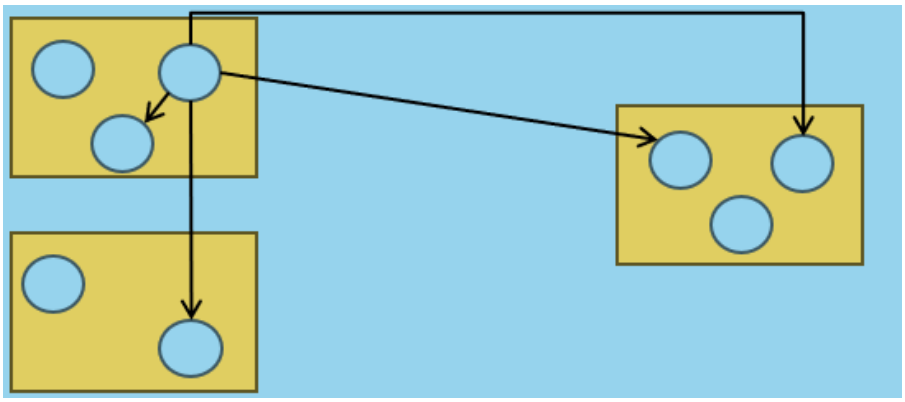


Figura 4.11. Caso 6

- **From Pods with a certain key in all namespace to Pods with a certain key in all namespaces**: in tal caso si vogliono abilitare le comunicazioni da tutti i pod con una certa chiave fra tutti i namespace verso tutti i pod con una certa chiave fra tutti i namesapace. Il traduttore dovrà creare policies di egress per ogni pod sorgente e una policy di ingress per ogni pod destinazione.

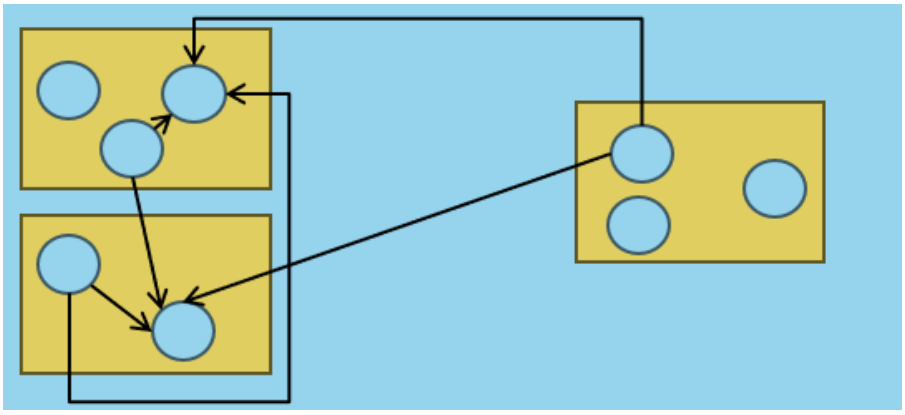


Figura 4.12. Caso 7

- **From all Pods in all namespaces to all Pods in all namespaces** : caso estremo in cui si vogliono abilitare tutte le comunicazioni fra i diversi pod. In questa casistica il traduttore deve creare policies di egress per ogni pod sorgente e policies di ingress per ogni pod destinazione.

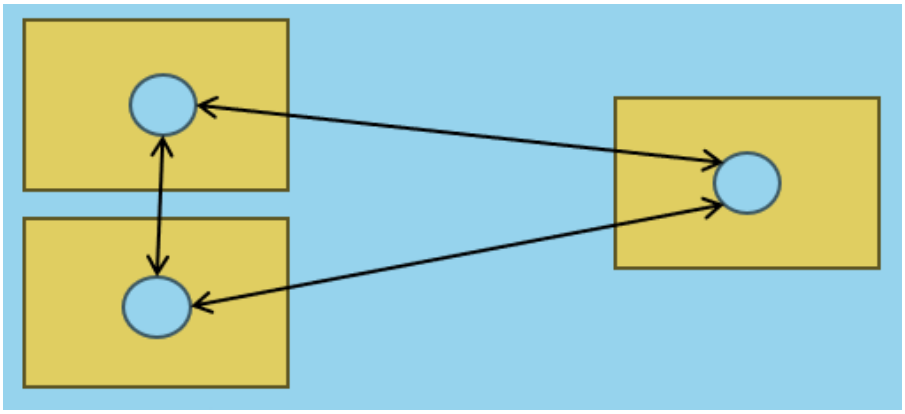


Figura 4.13. Caso 8

Le casistiche presentate offrono una panoramica generale delle capacità del traduttore, che, infatti, riesce a distinguere tra i **namespace remoti** e quelli **locali**, permettendo così di tradurre anche gli intenti che richiedono questa distinzione. Ritornando all' esempio della figura 4.5, il risultato della traduzione dell' intento armonizzato [**R1**] sarà:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: request_1_egress
  namespace: fluidos
spec:
  policyTypes:
  - Egress
  podSelector:
    matchLabels:
      app: order_placement
  egress:
    to:
    - namespaceSelector:
        matchLabels:
          name: default
      podSelector:
        matchLabels:
          app: product-catalogue
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: request_1_ingress
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: product_catalogue
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: fluidos
      podSelector:
        matchLabels:
          app: order_placement
    ports:
    - port: 80
      protocol: TCP
```

Figura 4.14. esempio traduzione intento armonizzato

Traduttore ed **armonizzatore** sono parte della soluzione proposta che prevede l' uso di un **controllore** che è in esecuzione sul **provider** e che non appena il **consumer** effettua l' offload delle risorse, legge gli intenti dal **contratto**, li passa all' **armonizzatore** che restituisce il set di intenti armonizzati al controllore il quale li traduce e successivamente li applica in modo automatico. Il contributo di questo lavoro prevede l' implementazione del controllore e del processo di traduzione in esso integrato.

4.2 Controllore

Il **controllore** è in esecuzione sia sul **consumer** che sul **provider**. In entrambi i casi il **controllore** effettua l' **autenticazione** con l' **Api-Server** del cluster in cui esso esegue, ciò è necessario affinché il controller riceva tutte le informazioni necessarie per attuare i processi di **verifica** (lato consumer), **armonizzazione e traduzione** (lato provider).

Workflow controllore lato consumer

E' possibile descrivere il workflow del controllore sul lato del consumer in due macro-step. Nel primo macro-step, il controllore si **autentica** all' **Api-Server** del cluster consumer per ricevere da esso la **lista dei pod e dei namespace disponibili** e per poter applicare le **network policy di default** che consentono l' applicazione delle policy di **default deny** e consentono i namespace del consumer di comunicare con gli eventuali namespace che verranno offlodati. (vedi figura 4.15)

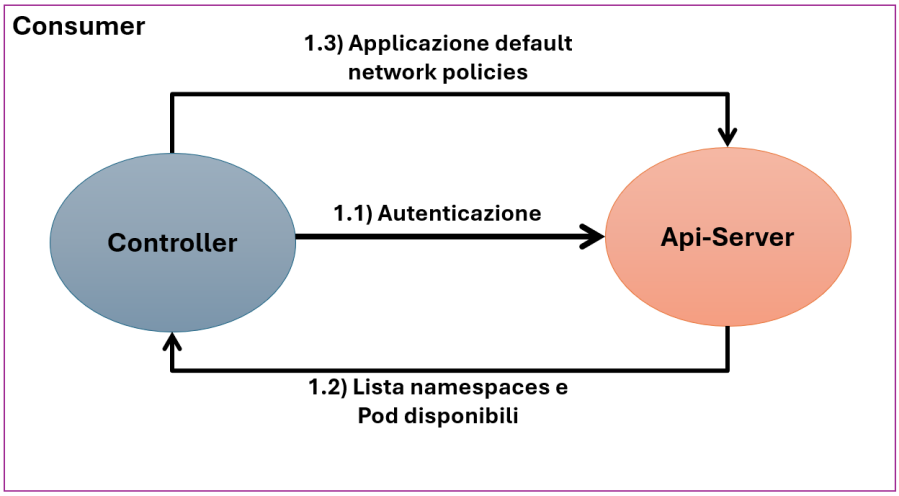


Figura 4.15. Controllore workflow 1 lato consumer

Successivamente nel secondo macro-step, non appena vengono trovati dei **candidati al peering**, il controllore legge questi ultimi e li sottopone ad un processo di **verifica** restituendo il primo **peering candidate** che soddisfa i requisiti del processo. (vedi Figura 4.16)

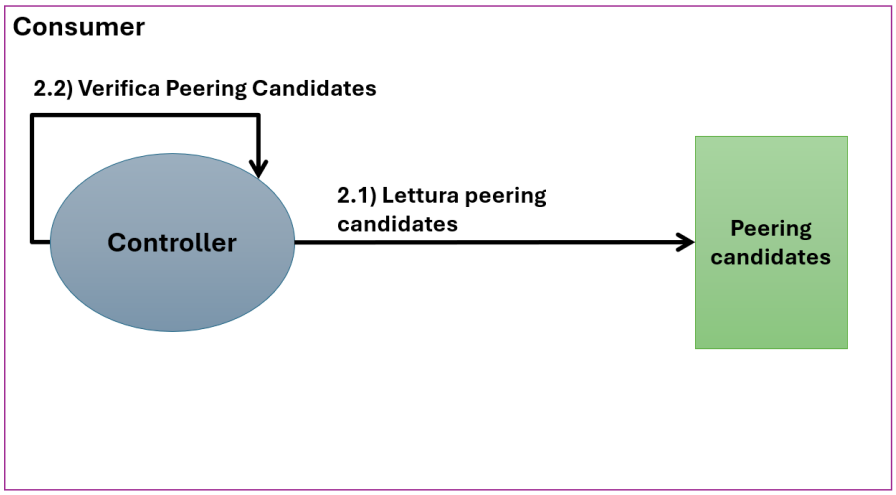


Figura 4.16. Controllore workflow 2 lato consumer

Tale peering candidate scelto sarà il provider che offrirà i servizi richiesti dal consumer, ospitando le risorse che quest' ultimo offloderà sul provider. La transazione sarà finalizzata grazie ad un **contratto** stabilito fra le due entità.

Workflow controllore lato provider

E' possibile descrivere il workflow del controllore sul lato del provider in tre macro-step che verrano discussi nel dettaglio in seguito. Nel primo macro-step, il controllore si **autentica** all' API-Server del provider, riceve da esso la **lista dei pod e dei namespace** disponibili sul cluster ospitante ed applica delle **network policy di default**(vedi 4.17). Le **network policy di default** sono necessarie per permettere di applicare la politica deafault deny e per permette di far comunicare i namespace offlodati sul provider con le risorse del consumer.

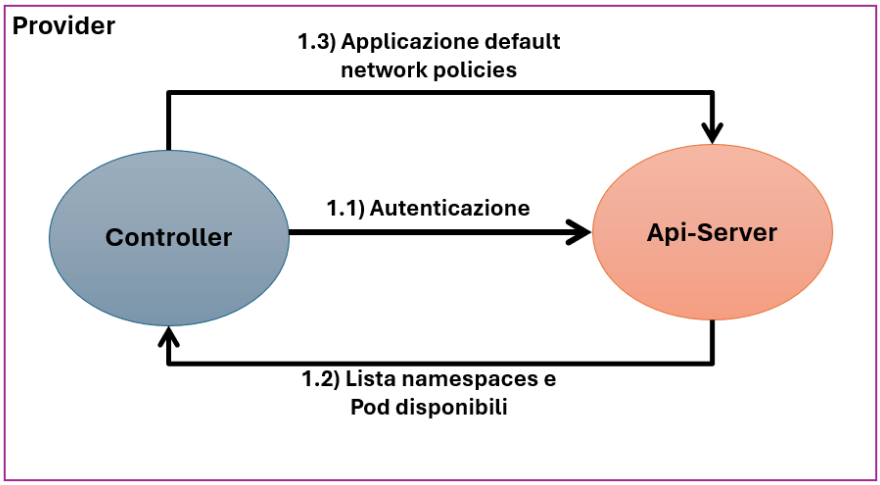


Figura 4.17. Controllore workflow 1 lato provider

Dopo la fase di autenticazione il controllore rimane in esecuzione e verrà notificato dall' Api-Server quando uno o più namespace verranno **offloadati**. Se ciò accade si attiva il secondo macro-step , in cui il **Controller** interagisce con l' **armonizzatore**, in particolare il controllore ottiene la lista degli **intenti di richiesta** contenuti nella **config map**, in uno dei namespace offlodati, definita nel **contratto**, successivamente li passa all' **armonizzatore** che li armonizza e restituisce al **controller** gli intenti armonizzati (vedi 4.18)

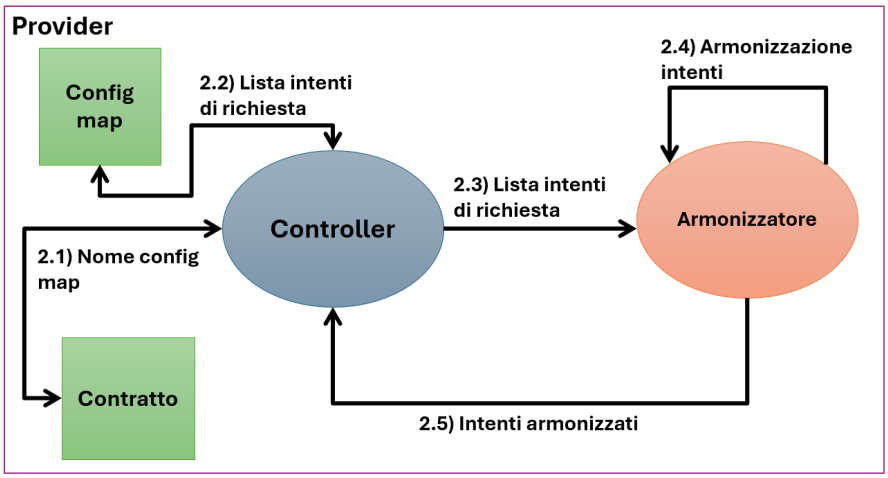


Figura 4.18. Controllore workflow 2 lato provider

Nel terzo macro-step, il controllore effettua la **traduzione** degli intenti in **Kubernetes Network Policies** e successivamente le **applica** interagendo con l' **Api-Server** (vedi 4.19).

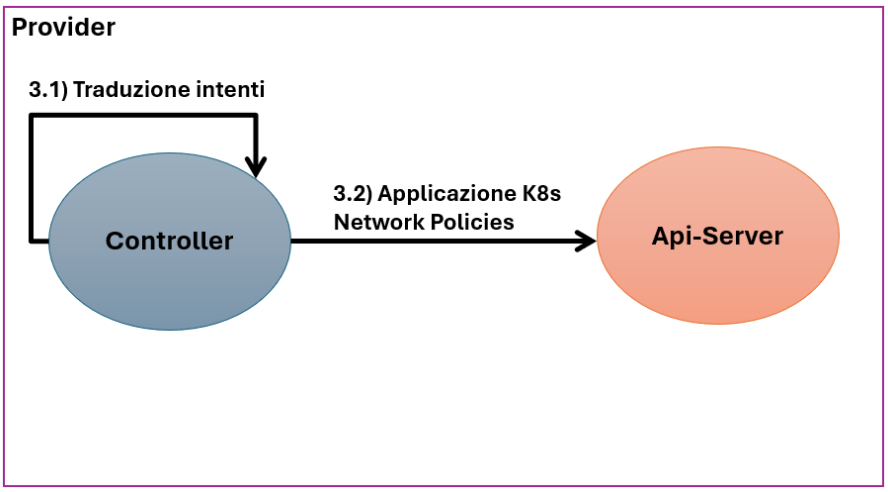


Figura 4.19. Controllore workflow 3 lato provider

Autenticazione

Affinché il **controllore** possa interagire con il cluster ospitante, è importante che esso si **autentichi** all' **API-Server**. L' autenticazione di un componente è fondamentale per garantire la **sicurezza**, il **controllo** e **tracciabilità** delle operazioni nel cluster. L' autenticazione è solo un primo passo, difatti una volta che il controllore è autenticato, ad esso verranno assegnati dei permessi che permetteranno al componente di effettuare le azioni desiderate descritte nel **workflow**.

Capitolo 5

Implementazione

Il **Controllore** è stato implementato usando il linguaggio di programmazione **Java**, successivamente ne è stata creata l' **immagine**, la quale è stata caricata in una repository pubblica **Docker** e successivamente è stato creato il **Deployment** che è stato applicato sul **Cluster provider**. Il Deployment viene applicato nel namespace **FLUIDOS** ed è associato al Service Account **costum-controller**.

Implementazione autenticazione

Per implementare il processo di **autenticazione** si è fatto uso del **service account** per creare un **identità univoca** all' interno del cluster alla quale poi associare diversi ruoli. Il service account è stato chiamato **"costum-controller"** ed è stato creato nel **cluster ospitante** sfruttando il comando:

```
kubectl create serviceaccount costum-controller
```

Successivamente, è stato generato un **secret** nel namespace **default**, associato al **service account custom-controller**. Il secret contiene un **token di autenticazione** generato da Kubernetes, che funge da credenziale. Il token permette al service account di autenticarsi in modo sicuro presso l'**API-Server** del cluster. Quando un pod è associato ad un service account, Kubernetes inietta il **token di autenticazione** in un determinato path del Pod. Il Controllore quindi accede a tale path per estrapolare il token di autenticazione per poi autenticarsi presso l' **Api-Server** del cluster in modo automatico.

RBAC e assegnazioni ruoli

Affinché il controllore possa eseguire tutte le azioni necessarie, deve ottenere i **permessi** per compiere le operazioni richieste. A tal fine, sono stati definiti dei **ruoli** per esprimere le **azioni** consentite su specifiche risorse all'interno del cluster. Successivamente è stato fatto il **binding** fra tali ruoli e il service account associato al controller per garantirne il funzionamento.

Il **controllore** interagisce con la seguenti **risorse**:

- **peeringcandidates**: l' interazione con tale risorsa consente al controllore in esecuzione nel consumer di scegliere il candidato al peering.
- **flavors**: l' interazione con tale risorsa consente al controllore in esecuzione nel provider di estrapolare gli intenti di autorizzazione da considerare nel processo di armonizzazione.
- **contratto**: l' interazione con tale risorsa è fondamentale per il provider in quanto in questo modo può ottenere il nome della **config-map** nella quale può leggere gli **intenti di richiesta** del consumer da considerare nel processo di armonizzazione e di traduzione.
- **config-map**: l' interazione con tale risorsa è fondamentale in quanto una volta letto il nome della config-map dal **contratto**, bisogna accedere alla config-map in uno dei namespace offlodati dal consumer per leggere gli **intenti di richiesta** di quest' ultimo.
- **reservation**: tale risorsa è creata dal consumer dopo che esso ha scelto il candidato al peering. In tale risorsa è presente l' id del **buyer** e del **seller**. L' interazione con tale risorsa è necessaria per permettere al controllore di capire se esso è in esecuzione nel cluster consumer o nel cluster provider.
- **namespace e pod**: l' interazione con tali risorse è fondamentale per ottenere le informazioni su pod e namespace del cluster in cui il controller è in esecuzione.
- **network policies**: l' interazione con tale risorsa è necessaria per permettere al controllore di creare le policy che verranno tradotte.

Cluster role e Cluster role binding per interagire con le risorse

Per consentire al controllore di interagire con le varie risorse definite in **FLUIDOS**, è necessario fornire dei permessi al controllore espressi tramite ruoli. I ruoli sono stati definiti tramite **cluster role**, consentendo le seguenti azioni sulle risorse elencate precedentemente:

- **get**: per ottenere una specifica risorsa.
- **list**: per ottenere tutte le risorse specifiche in un determinato momento.
- **watch**: per permettere al controllore di monitorare quella risorsa specifica quando viene aggiunta.

Inoltre per le **network policies** è stato anche abilitato il permesso **create** per permettere di applicare nel cluster. Successivamente il binding fra **cluster-role** e **service-account** è stato realizzato usando la risorsa **cluster-role-binding**.

L' esempio in figura 5.1, mostra la creazione di un cluster role per i peering candidates, consentendo su quest' ultima le operazioni di get, list e watch. Mentre la figura 5.2 mostra il binding che permette di associare il ruolo definito nel cluster role al service account associato al controllore.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: "peeringcandidates-cluster-role"
  namespace: fluidos
rules:
 - apiGroups: ["advertisement.fluidos.eu"]
   resources: ["peeringcandidates"]
   verbs: ["get", "list", "watch"]
```

Figura 5.1. Cluster role per i peeringcandidates

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: peeringcandidates-binding
subjects:
- kind: ServiceAccount
  name: "costum-controller"
  namespace: fluidos
roleRef:
  kind: ClusterRole
  name: "peeringcandidates-cluster-role"
  apiGroup: rbac.authorization.k8s.io
```

Figura 5.2. Cluster role binding per i peeringcandidates

5.1 Implementazione del controllore

Una volta che il controllore si è autenticato e si è connesso al cluster, esso avvia **cinque thread**:

- **thread per monitorare i pod**: necessario per capire se un pod è stato aggiunto o eliminato.
- **thread per monitorare i namespace**: necessario per capire se un namespace è stato aggiunto, eliminato o offlodato.
- **thread per monitorare la risorsa tunnelendpoint**: necessaria in quanto una volta aggiunta tale risorsa, il controller può configurare le regole di rete necessarie per consentire la comunicazione fra i servizi offlodati e quelli locali.
- **thread che applica le network policy di default**: tale thread applica diverse network policy di default:
  - policies di deny che vietano ad un servizio di comunicare liberamente con qualsiasi altro servizio, in quanto di default in Kubernetes qualsiasi servizio può comunicare liberamente con chiunque.
  - policies che abilitano la comunicazione con il DNS di Kubernetes nel caso in cui ci siano ulteriori policy che permettano ad un pod di raggiungere un determinato hostname esterno.
  - policies che permettono la comunicazione fra risorse offlodate nel cluster remoto e risorse non offlodate nel cluster locale.
- **thread che monitora i vari candidati al peering**: tale thread monitora gli eventuali candidati al peering per poi sceglierne uno, dopo un processo di **verifica**.

In generale questi thread fanno uso di un **watch** sulla risorsa da monitorare. Questo componente notifica il controllore ogni qualvolta la risorsa monitorata subisce una modifica o viene aggiunta. Il controllore è composto da tre funzioni:

- **funzione di verifica**: chiamata se il controllore è in esecuzione sul consumer per scegliere quali tra i candidati al peering possano soddisfare le proprie richieste.
- **funzione di armonizzazione**: chiamata quando il controllore è in esecuzione sul provider per armonizzare gli intenti di richiesta del consumer e gli intenti di autorizzazione del provider, per risolvere le discordanze.
- **funzione di traduzione**: chiamata quando il controllore è in esecuzione sul provider per tradurre gli intenti armonizzati in Kubernetes Network policies.

Traduzione intenti

Il traduttore è una funzione che viene chiamata dal controllore ogni qualvolta una risorsa viene offlodata. Il traduttore affinché possa tradurre correttamente gli intenti armonizzati avrà bisogno di reperire le informazioni dei **namespace locali**, ossia i namespace del cluster ospitante e dei **namespace remoti**, ossia i namespace offlodati dal consumer. Tali informazioni li ottiene dai thread del **controller** che monitorano i namespace e riesce a distinguere un namespace locale da uno remoto grazie ad un **annotazione** che viene aggiunta da **Liqo** nel caso in cui il namespace sia stato offlodato. Il traduttore non andrà a considerare, i namespace di sistema e i namespace della CNI Calico nel processo di traduzione.
Dato un intento, il traduttore dapprima estrapola ogni suo campo, successivamente chiamerà diverse funzioni che permetteranno di creare i diversi campi delle Kubernetes network policies [vedi struttura 2.2.1]. In generale il traduttore ,per ogni intento, crea almeno una network policy di **Egress** per garantire il traffico in uscita dalla sorgente alla destinazione e di **Ingress**, per garantire il traffico in entrata dalla sorgente alla destinazione, tuttavia se l' intento specifica come destinazione un range di indirizzi CIDR, a quel punto il traduttore creerà soltanto policies di **Egress**. Dato che un intento può selezionare piu pod/namespace tramite il selettore di chiave o di valore, il traduttore tramite dei cicli può ricreare le diverse network policies per ognuno di questi pod/namespace sfruttando anche le informazioni interne al cluster. Per esempio un intento potrebbe voler applicare una policy a tutti i pod di un determinato namespace, il traduttore creerà network policy differenti per ogni pod di questo namespace, facendo uso delle informazioni su pod e namespace ottenute dal controller. E' stata usata la classe java **V1NetworkPolicy** della libreria ufficiale **Kubernetes** per permettere la creazione delle network policies, che poi saranno automaticamente applicate sul cluster dal **controllore**.

Capitolo 6

Validazione

6.1 Installazione nodi FLUIDOS

Tale capitolo mostrerà i risultati ottenuti definendo un esempio di utilizzo reale. La demo presentata è stata realizzata utilizzando il sottosistema **WSL** Linux insieme a **Docker Desktop**, con l'installazione di due nodi **FLUIDOS**. Sui due nodi è stata configurata la CNI Calico, e infine è stato installato Liqo per effettuare il peering quando necessario.

6.2 Esempio d' uso

Supponiamo di avere un **consumer** ed un **provider**. Il consumer possiede i seguenti namespace:

- **payments**: namespace in cui sono in esecuzione applicazioni che si occupano di effettuare pagamenti come: **app-payment-1**, **app-payment-2**.
- **products**: namespace in cui sono in esecuzione applicazioni che si occupano di tenere traccia di diversi tipi di prodotti come: **mobile-products** e **desktop-products**.
- **users**: namespace in cui sono in esecuzione applicazioni che si occupano di gestire gli utenti come: **list-users** e **list-credentials**.

Mentre il provider possiede i seguenti namespace:

- **handle-payments**: namespace creato per gestire eventuali pagamenti con applicazioni come: **bank-1**, **bank-2** e **bank-3**.
- **monitoring**: namespace creato per monitorare le risorse nel cluster tramite applicazioni come **resource-monitor**.

Il tutto è mostrato nella figura 6.1

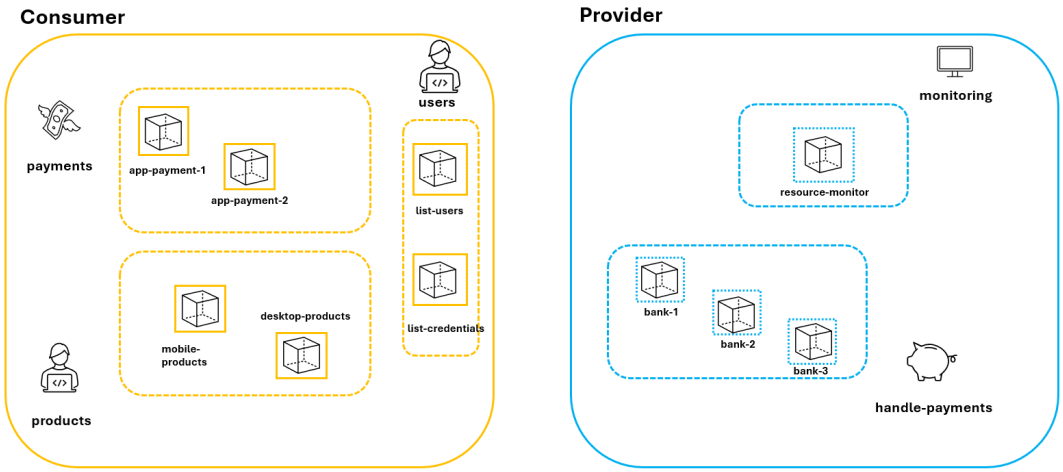


Figura 6.1. caso d' uso

Supponiamo che il **consumer** voglia offlodare le risorse dei namespace **payments** e **products** nel **provider**, cosicché quest' ultimo possa gestire i pagamenti dei vari prodotti del consumer. [vedi figura 6.2]

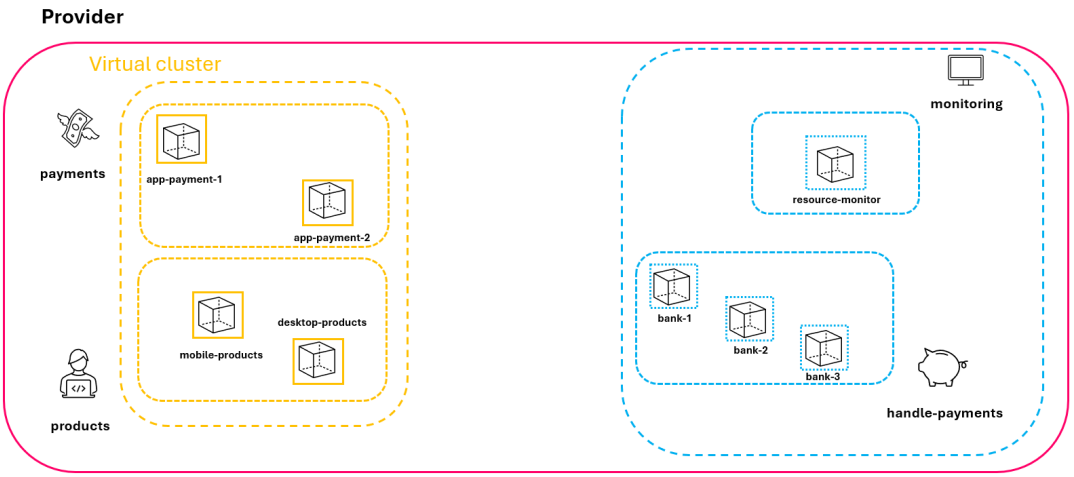


Figura 6.2. Namespace del consumer offlodati sul provider

Il **consumer** durante la transazione con il provider per l' **offloading** definisce degli intenti di richiesta.

Intenti di richiesta

- (*) **RequestIntent_1** - Src: [app:app-payment-1 - name:payments], Dst: [*:* - *:*], DstPort: [80], ProtocolType: [TCP] false true

Descrizione: Il consumer richiede al provider di permettere la comunicazione del pod "app:app-payment-1" con qualsiasi pod dei namespace del provider sulla porta 80 con il protocollo TCP.

- (*) **RequestIntent_2** - Src: [app:app-payment-2 - name:payments], Dst: [*:* - *:*], DstPort: [90], ProtocolType: [TCP] false true

  Descrizione: Il consumer richiede al provider di permettere la comunicazione del pod "app:app-payment-2" con qualsiasi pod dei namespace del provider sulla porta 90 con il protocollo TCP.

- (*) **RequestIntent_3** - Src: [app:mobile-products - name:products], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL] false true

  Descrizione: Il consumer richiede al provider di permettere la comunicazione del pod "app:mobile-products" con un set di indirizzi CIDR (142.250.0.0/15).

- (*) **RequestIntent_4** - Src: [app:desktop-products - name:products], Dst: [*:* - *:*], DstPort: [85], ProtocolType: [UDP] false true

  Descrizione: Il consumer richiede al provider di permettere la comunicazione del pod "app:desktop-products" con tutti i pod del provider usando il protocollo UDP sulla porta 85.

E' possibile mostrare le richieste di intento del consumer con la Figura 6.3.

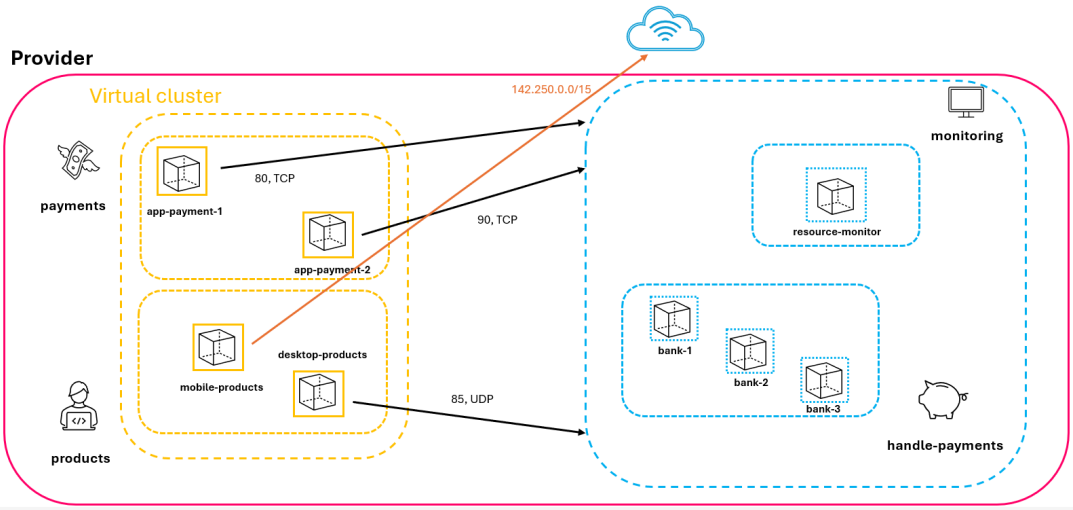


Figura 6.3. Richieste di intento

Tuttavia il provider definisce tre **flavour** ossia la lista degli **intenti di autorizzazione** e degli **intenti obbligatori**.

Flavour 1

- (*) **AuthorizationDeny_1** - Src: [*:* - *:*], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e l'intervallo di indirizzi IP 142.250.0.0/15 è negata per tutti i protocolli.

- (*) **AuthorizationDeny_2** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [*], ProtocolType: [SCTP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo SCTP.

- (*) **AuthorizationDeny_3** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [*], ProtocolType: [UDP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo UDP.

- (*) **AuthorizationDeny_4** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [0-79], ProtocolType: [TCP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo TCP sulle porte da 0 a 79.

- (*) **AuthorizationMandatory_1** - Src: [app:resource-monitor - name:monitoring], Dst: [*:* - *:*], DstPort: [43], ProtocolType: [TCP] false true

  Descrizione: È obbligatoria la comunicazione tra il pod "app:resource-monitor" nel namespace "monitoring" e qualsiasi pod destinazione sulla porta 43, utilizzando il protocollo TCP.

Flavour 2

- (*) **AuthorizationDeny_1** - Src: [*:* - *:*], Dst: [142.250.0.1/15], DstPort: [*], ProtocolType: [ALL] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e l'intervallo di indirizzi IP 142.250.0.1/15 è negata per tutti i protocolli.

- (*) **AuthorizationDeny_2** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [*], ProtocolType: [SCTP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo SCTP.

- (*) **AuthorizationDeny_3** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [82-120], ProtocolType: [TCP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo TCP sulle porte da 82 a 120.

- (*) **AuthorizationMandatory_1** - Src: [app:resource-monitor - name:monitoring], Dst: [*:* - *:*], DstPort: [43], ProtocolType: [ALL] false true

  Descrizione: È obbligatoria la comunicazione tra il pod "app:resource-monitor" nel namespace "monitoring" e qualsiasi pod destinazione sulla porta 43, utilizzando qualsiasi protocollo.

Flavour 3

- (*) **AuthorizationDeny_2** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [*], ProtocolType: [SCTP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo SCTP.

- (*) **AuthorizationDeny_3** - Src: [*:* - *:*], Dst: [*:* - name:handle-payments], DstPort: [0-79], ProtocolType: [TCP] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e il pod con nome "handle-payments" è negata per il protocollo TCP sulle porte da 0 a 79.

- (*) **AuthorizationDeny_4** - Src: [*:* - *:*], Dst: [1.1.1.1/20], DstPort: [*], ProtocolType: [ALL] true false

  Descrizione: La comunicazione tra qualsiasi pod sorgente e l'intervallo di indirizzi IP 1.1.1.1/20 è negata per tutti i protocolli.

- (*) **AuthorizationMandatory_1** - Src: [app:resource-monitor - name:monitoring], Dst: [*:* - *:*], DstPort: [43], ProtocolType: [ALL] false true

  Descrizione: È obbligatoria la comunicazione tra il pod "app:resource-monitor" nel namespace "monitoring" e qualsiasi pod destinazione sulla porta 43, utilizzando qualsiasi protocollo.

Tali **flavour** verranno mostrati dal **solver** al consumer sottoforma di **peeringcandidates**. Il consumer avvierà un processo di **verifica** che consente di scegliere il candidato che sia **compatibile** con le proprie richieste di intento 6.2.

6.3 Processo di verifica

Il processo di **verifica** viene avviato sul **consumer** non appena il **solver** durante la fase di discovery trova dei candidati al peering che possono offrire dei servizi al consumer. Tale processo ritornerà **true** al primo **peering candidate** compatibile con le richieste del consumer.

PeeringCandidate 1

Il **verifier** restituisce **false** per il peering candidate definito dal provider tramite il Flavour 1 [6.2].

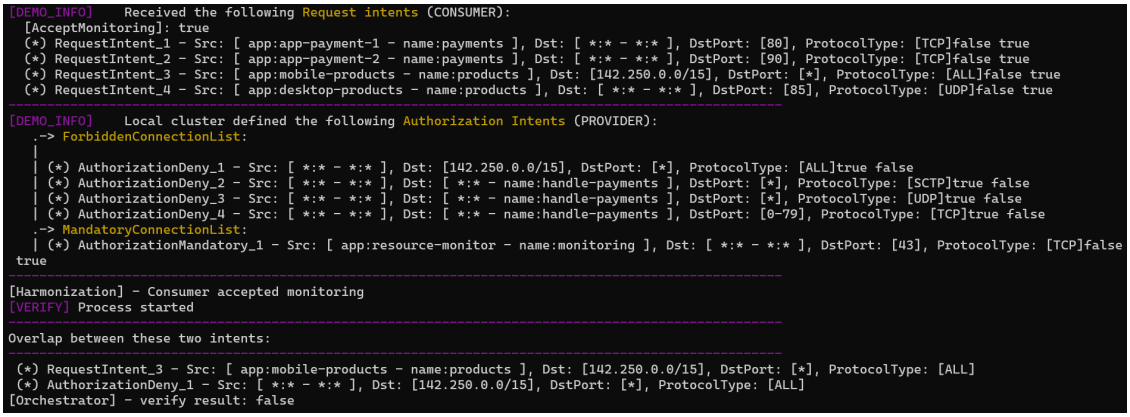


Figura 6.4. Risultati verifier per il primo peering candidate

La figura 6.4 mostra che il verifier ha trovato che **RequestIntent_1** e **AuthorizationDeny_1** non sono compatibili e dunque il processo ritorna **false**. Questo perché il consumer con la prima richiesta di intento vorrebbe connettersi all' indirizzo CIDR 142.250.0.0/15, ma ciò viene negato da questo candidato al peering, come mostra la Figura [6.5].

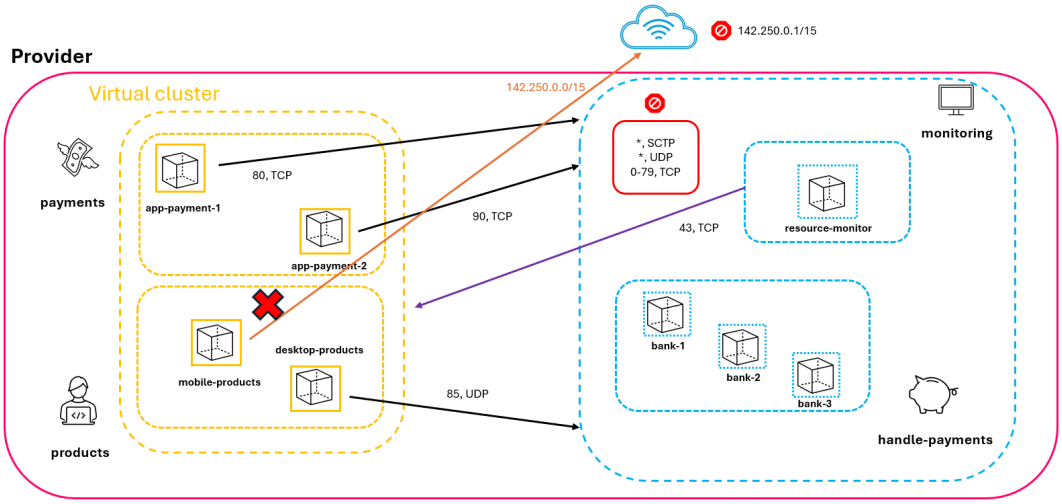


Figura 6.5. Risultati verifier per il primo peering candidate

PeeringCandidate 2

Il **verifier** restituisce **false** per il peering candidate definito dal provider tramite il Flavour 2 [6.2].

```
[DEMO_INFO]    Received the following Request intents (CONSUMER):
  [AcceptMonitoring]: true
  (*) RequestIntent_1 - Src: [ app:app-payment-1 - name:payments ], Dst: [ *:* - *:* ], DstPort: [80], ProtocolType: [TCP]false true
  (*) RequestIntent_2 - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]false true
  (*) RequestIntent_3 - Src: [ app:mobile-products - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]false true
  (*) RequestIntent_4 - Src: [ app:desktop-products - name:products ], Dst: [ *:* - *:* ], DstPort: [85], ProtocolType: [UDP]false true
------------------------------------------------------------------------------------------------------------------------------------

[DEMO_INFO]    Local cluster defined the following Authorization Intents (PROVIDER):
   .-> ForbiddenConnectionList:
   |
   | (*) AuthorizationDeny_1 - Src: [ *:* - *:* ], Dst: [142.250.0.1/15], DstPort: [*], ProtocolType: [ALL]true false
   | (*) AuthorizationDeny_2 - Src: [ *:* - *:* ], Dst: [ *:* - name:handle-payments ], DstPort: [*], ProtocolType: [SCTP]true false
   | (*) AuthorizationDeny_3 - Src: [ *:* - *:* ], Dst: [ *:* - name:handle-payments ], DstPort: [82-120], ProtocolType: [TCP]true false
   .-> MandatoryConnectionList:
   | (*) AuthorizationMandatory_1 - Src: [ app:resource-monitor - name:monitoring ], Dst: [ *:* - *:* ], DstPort: [43], ProtocolType: [ALL]false
 true
------------------------------------------------------------------------------------------------------------------------------------

[Harmonization] - Consumer accepted monitoring
[VERIFY] Process started
------------------------------------------------------------------------------------------------------------------------------------

Overlap between these two intents:
------------------------------------------------------------------------------------------------------------------------------------

 (*) RequestIntent_2 - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]
 (*) AuthorizationDeny_3 - Src: [ *:* - *:* ], Dst: [ *:* - name:handle-payments ], DstPort: [82-120], ProtocolType: [TCP]
[Orchestrator] - verify result: false
------------------------------------------------------------------------------------------------------------------------------------
```

Figura 6.6. Risultati verifier per il secondo peering candidate

La figura 6.4 mostra che il verifier ha trovato che **RequestIntent_2** e **AuthorizationDeny_3** non sono compatibili e dunque il processo ritorna **false**. Questo perché il consumer con la prima richiesta di intento vorrebbe far si che **app:app-payment-2** possa connettersi a tutti i pod del provider usando come porta di destinazione la porta 90, ma il provider definisce un' authorization deny che vieta ai qualsiasi pod del consumer di connettersi ai pod del namespace "handle-payments" nel range di porte destinazione che va da 82 a 120, rendendo la richiesta del consumer incompatibile.[vedi Figura 6.7]

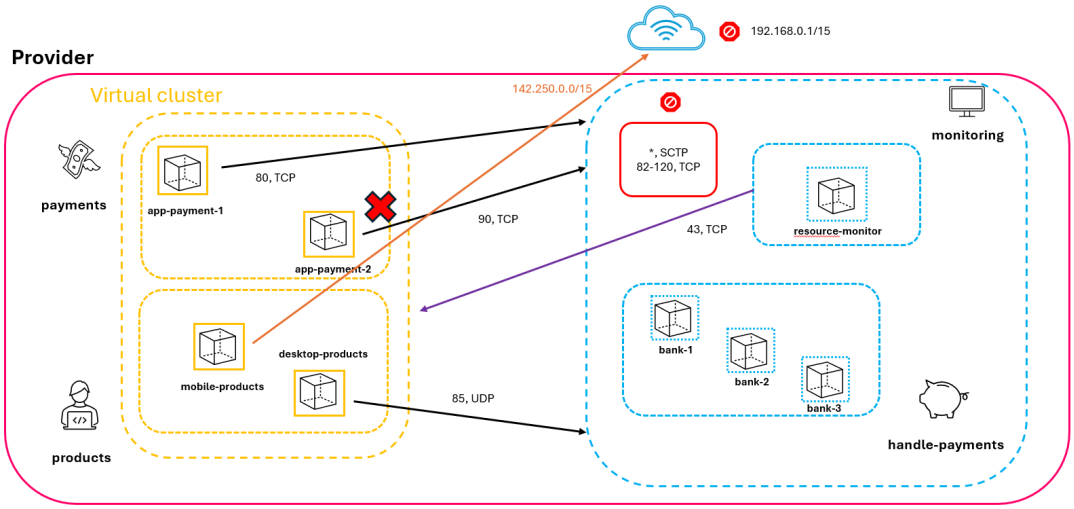


Figura 6.7. Risultati verifier per il secondo peering candidate

PeeringCandidate 3

Il **verifier** restituisce **true** per il peering candidate definito dal provider tramite il Flavour 3 [6.2].

```
[DEMO_INFO]    Received the following Request intents (CONSUMER):
  [AcceptMonitoring]: true
  (*) RequestIntent_1 - Src: [ app:app-payment-1 - name:payments ], Dst: [ *:* - *:* ], DstPort: [80], ProtocolType: [TCP]false true
  (*) RequestIntent_2 - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]false true
  (*) RequestIntent_3 - Src: [ app:mobile-products - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]false true
  (*) RequestIntent_4 - Src: [ app:desktop-products - name:products ], Dst: [ *:* - *:* ], DstPort: [85], ProtocolType: [UDP]false true
-------------------------------------------------------------------------------------------------------------------------------
[DEMO_INFO]    Local cluster defined the following Authorization Intents (PROVIDER):
   .-> ForbiddenConnectionList:
   |
   | (*) AuthorizationDeny_2 - Src: [ *:* - *:* ], Dst: [ *:* - name:handle-payments ], DstPort: [*], ProtocolType: [SCTP]true false
   | (*) AuthorizationDeny_3 - Src: [ *:* - *:* ], Dst: [ *:* - name:handle-payments ], DstPort: [0-79], ProtocolType: [TCP]true false
   | (*) AuthorizationDeny_4 - Src: [ *:* - *:* ], Dst: [1.1.1.1/20], DstPort: [*], ProtocolType: [ALL]true false
   .-> MandatoryConnectionList:
   | (*) AuthorizationMandatory_1 - Src: [ app:resource-monitor - name:monitoring ], Dst: [ *:* - *:* ], DstPort: [43], ProtocolType: [ALL]false
true
-------------------------------------------------------------------------------------------------------------------------------
[Harmonization] - Consumer accepted monitoring
[VERIFY] Process started
-------------------------------------------------------------------------------------------------------------------------------
[Orchestrator] - verify result: true
```

Figura 6.8. Risultati verifier per il terzo peering candidate

La figura 6.8 mostra che il verifier non ha trovato nessuna discordanza e quindi questo sarà il candidato al peering che il consumer sceglierà per eseguire l' offloading dei propri servizi.

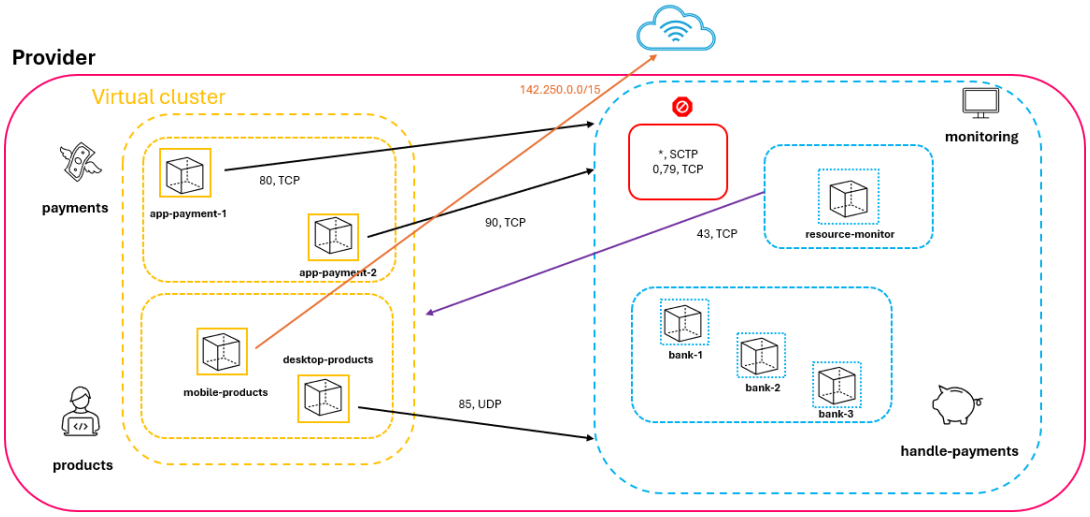


Figura 6.9. Risultati verifier per il terzo peering candidate

6.4 Prenotazione e allocazione delle risorse

Una volta trovato il peering candidate che sia compatibile con le proprie richieste di intento, il consumer **prenota** le risorse sul candidato scelto tramite la **CRD reservation**. Come mostra la figura 6.10, il consumer specifica le informazioni del **seller**, ossia il provider scelto ed il **buyer**, ossia le proprie informazioni.

```yaml
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Reservation
metadata:
  name: reservation-sample
  namespace: fluidos
spec:
  solverID: solver-sample
  # Set it as you want, following needs and requests in the solver.
  # Optional
  configuration:
    # Be sure to use the same type of the peeringCandidate
    type: K8Slice
    # Be sure to use values that are in the range of the peeringCandidate
    data:
      cpu: 1000m
      memory: 1Gi
      pods: "110"
  # Retrieve from PeeringCandidate chosen to reserve
  peeringCandidate:
    name: peeringcandidate-fluidos.eu-k8slice-mio-esempio3
    namespace: fluidos
  # Set it to reserve
  reserve: true
    # Set it to purchase after reservation is completed and you have a transaction
  purchase: true
  # Retrieve from PeeringCandidate Flavor Owner field
  seller:
    domain: fluidos.eu
    ip: 172.18.0.6:30001
    nodeID: 35bxvuekb9
  # Retrieve from configmap
  buyer:
    domain: fluidos.eu
    ip: 172.18.0.7:30000
    nodeID: 4ot9rtwow2
```

Figura 6.10. Prenotazione risorse sul candidato scelto

Successivamente applicherà tale risorsa sul cluster tramite il comando:

```
kubectl apply -f reservation.yaml
```

Una volta prenotate le risorse, verrà creato un contratto nel namespace FLUIDOS che descrive la transazione fra i due peer, il contratto è ottenibie con il comando:

```
kubectl get contracts -n FLUIDOS
```

La figura 6.11 mostra il nome del contratto e le informazioni sul seller e sul buyer immesse precedentemente nella **reservation**.

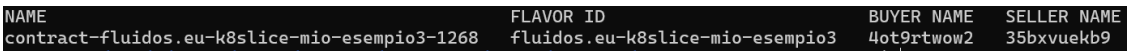


Figura 6.11. contratto

Per rendere la transazione effettiva il consumer dovrà applicare la CRD **allocation** tramite il comando:

```
kubectl apply -f allocation.yaml
```

specificando il nome del contratto stabilito fra i due peer come mostrato in figura 6.12.

```yaml
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Allocation
metadata:
  name: allocation-sample
  namespace: fluidos
spec:
  # Get it from the solver
  intentID: intent-sample
  # Retrieve information from the reservation and the contract bou d to it
  contract:
    name: contract-fluidos.eu-k8slice-mio-esempio3-1268
    namespace: fluidos
```

Figura 6.12. allocation

Una volta fatto ciò, il cluster consumer ed il cluster provider effettueranno il **peering** tramite **Liqo**. [vedi Figura 6.13]

```
┌─ Peered Cluster Information
│  fluidos-provider-1 - 8257b165-21dd-4e3b-9871-cae5131644c7
│      Type: OutOfBand
│      Direction
│          Outgoing: Established
│          Incoming: None
│      Authentication
│          Status: Established
│      Network
│          Status: Established
│          Network connection
│              Status:  Connected
│              Latency: 551µs
│              Gateway IPs
│                  Local:  172.18.0.5:32520
│                  Remote: 172.18.0.3:30086
│      API Server
│          Status: Established
│      Resources
│          Total acquired - resources offered by "fluidos-provider-1" to "fluidos-consumer-1"
│              cpu:               1000m
│              memory:            1.05GiB
│              pods:              110
│              ephemeral-storage: 0.00GiB
└
```

Figura 6.13. stato peering Liqo

6.5 Offloading delle risorse e applicazione delle network policies

Una volta che il peering è stato stabilito, il consumer potrà effettuare l' **offloading** dei namespace *products* e *payments*, sfruttando i seguenti comandi di Liqo:

```
liqoctl offload namespace payments --pod-offloading-strategy Remote

liqoctl offload namespace products --pod-offloading-strategy Remote
```

Non appena avviene l' offloading, il controller nel **provider** si accorge di ciò [Figura 6.14] e cerca nel **contratto** il nome della **config-map** dove si troveranno le richieste di intento del **consumer**. Tale config-map sarà offlodata in uno dei due namespace offlodati dal consumer, in questo caso nel namespace *payments* [vedi Figura 6.15].

```
Nuovo Namespace offloadato: payments-fluidos-consumer-1-3feed7
Nuovo Namespace offloadato: products-fluidos-consumer-1-3feed7
```

Figura 6.14. namespaces offlodati

```
ConfigMap trovata per il namespace: payments-fluidos-consumer-1-3feed7
Nessuna configMap trovata per il namespace: products-fluidos-consumer-1-3feed7
```

Figura 6.15. esempio controller config-map letta

Lette le richieste di intento queste verranno passate all' **armonizzatore** che armonizza le richieste di intento espresse dal consumer con gli intenti di autorizzazione del provider risolvendo le diverse discordanze. [vedi Figura 6.16]

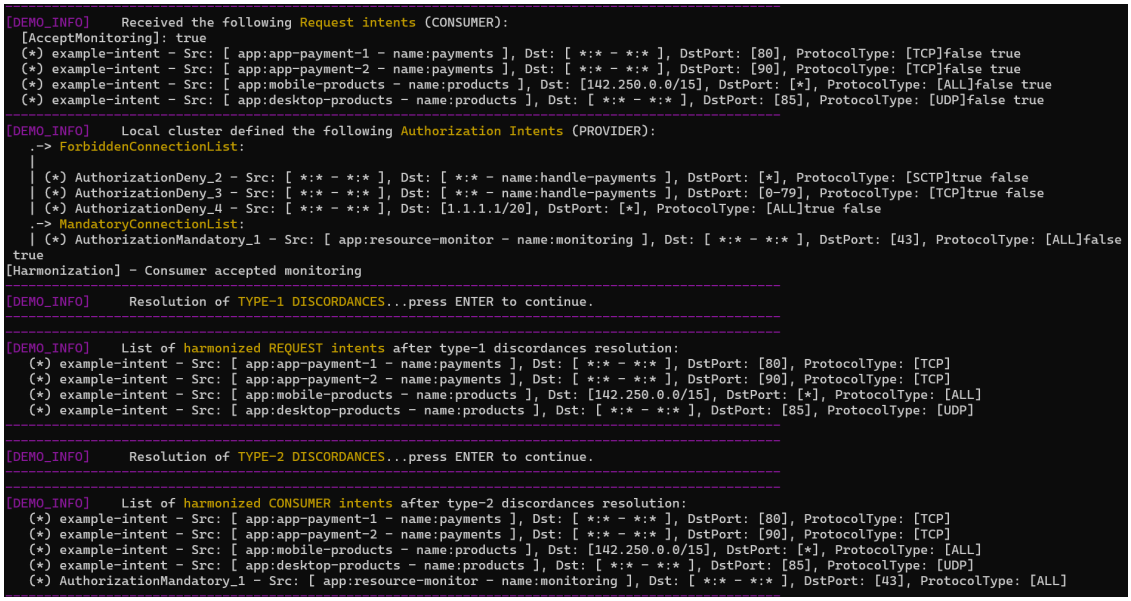


Figura 6.16. esempio risultato armonizzatore

Gli intenti armonizzati sono quindi:

- (*) **example-intent** - Src: [app:app-payment-1 - name:payments], Dst: [*:* - *:*], DstPort: [80], ProtocolType: [TCP]

  Descrizione: L'intento consente la comunicazione del pod "app:app-payment-1" con qualsiasi destinazione su qualsiasi namespace del provider, utilizzando la porta 80 e il protocollo TCP.

- (*) **example-intent** - Src: [app:app-payment-2 - name:payments], Dst: [*:* - *:*], DstPort: [90], ProtocolType: [TCP]

  Descrizione: L'intento consente la comunicazione del pod "app:app-payment-2" con qualsiasi destinazione su qualsiasi namespace del provider, utilizzando la porta 90 e il protocollo TCP.

- (*) **example-intent** - Src: [app:mobile-products - name:products], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]

  Descrizione: L'intento consente la comunicazione del pod "app:mobile-products" con un insieme di indirizzi IP definiti dal CIDR 142.250.0.0/15, senza restrizioni di porta o protocollo.

- (*) **example-intent** - Src: [app:desktop-products - name:products], Dst: [*:* - *:*], DstPort: [85], ProtocolType: [UDP]

Descrizione: L'intento consente la comunicazione del pod "app:desktop-products" con qualsiasi destinazione su qualsiasi namespace, utilizzando la porta 85 e il protocollo UDP.

- (*) **AuthorizationMandatory_1** - Src: [app:resource-monitor - name:monitoring], Dst: [*:* - *:*], DstPort: [43], ProtocolType: [ALL]

  Descrizione: L'intento obbliga la comunicazione del pod "app:resource-monitor" con qualsiasi destinazione su qualsiasi namespace del consumer, utilizzando la porta 43 e qualsiasi protocollo.

Tali intenti armonizzati sono quelli che saranno passati al **traduttore** che li tradurra in **Kubernetes network policies** e successivamente le applicherà sul cluster. [vedi Figura 6.17]

```
NetworkPolicy: exampleintentdefault78de91a4c0fde573818b1f4a11f75b705a70589d5d6cec2264f953003ff6f842 applicata per il namespace default
NetworkPolicy: exampleintentdefaultcdc77b79ba86111b9fc45b9577697a7f3d66ac53db54626990a25055beb8ab720 applicata per il namespace default
NetworkPolicy: exampleintentdefault17f4eb038e9158b361cd7aaaad1c53d3a15debafc3dd92f687cac65cc7f46ef94 applicata per il namespace default
NetworkPolicy: exampleintenthandle-paymentsccccbc9bb7696a4c27f1fd195777a2770d2918f8b54fa57784fb0b10c5ccea64 applicata per il namespace handle-payments
NetworkPolicy: exampleintenthandle-payments63b94b54e9a479fa40f67c079e1e445a9f795b38fbb79a1827edcaf1fef1c180 applicata per il namespace handle-payments
NetworkPolicy: exampleintenthandle-paymentsea24e4895796280742859653e467b2a866cdba1db27e3639b3340906a1a7fa69 applicata per il namespace handle-payments
NetworkPolicy: exampleintentmonitoringb7efc7e5dba36cef957775977e469e17c1c6be5873513169b7c309e87d49c39b2 applicata per il namespace monitoring
NetworkPolicy: exampleintentmonitoringdae3455ddc9a26e8018b4496ac296a69326773efcb0227125219ee73c607d048 applicata per il namespace monitoring
NetworkPolicy: authorizationmandatory1monitoringe4984f980af357767d11ffea9b203575b05bc0e30041a926dadff1971c49edf6 applicata per il namespace monitoring
NetworkPolicy: exampleintentmonitoringacba69788df5e412ea1de690052fc9efcd899fb121c5bbf0e252d310d0e3d212 applicata per il namespace monitoring
NetworkPolicy: authorizationmandatory1monitoringf7a2af71468e120e594a7ad2e8c2491422d9d547027a65420c681c92e7b7c1d3 applicata per il namespace monitoring
NetworkPolicy: exampleintentpayments-fluidos-consumer-1-3feed7ad42e3271f6d70064800a90b27ab0df92084cbc7051bd8bd98c40fd91e44aa58 applicata per il namespace payments-fluidos-consumer-1-3feed7
NetworkPolicy: exampleintentpayments-fluidos-consumer-1-3feed7f2c71cdd81a6f6b0ccbbcf940f799852920c53b93c62d1f651342adbaf7b64f1 applicata per il namespace payments-fluidos-consumer-1-3feed7
```

Figura 6.17. risultato applicazione network policy tradotte

Una volta applicate le network policies si otterà il risultato finale presentato in figura 6.18.

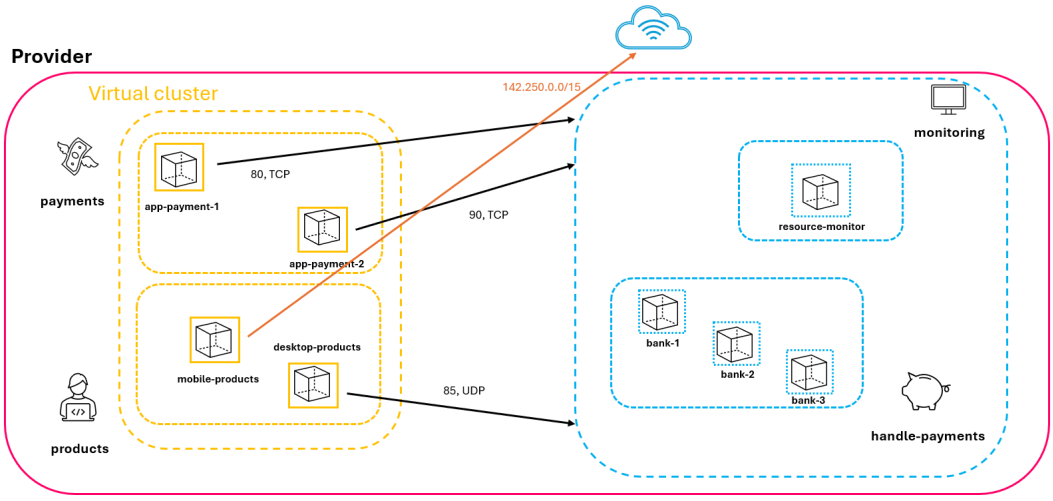


Figura 6.18. risultato finale

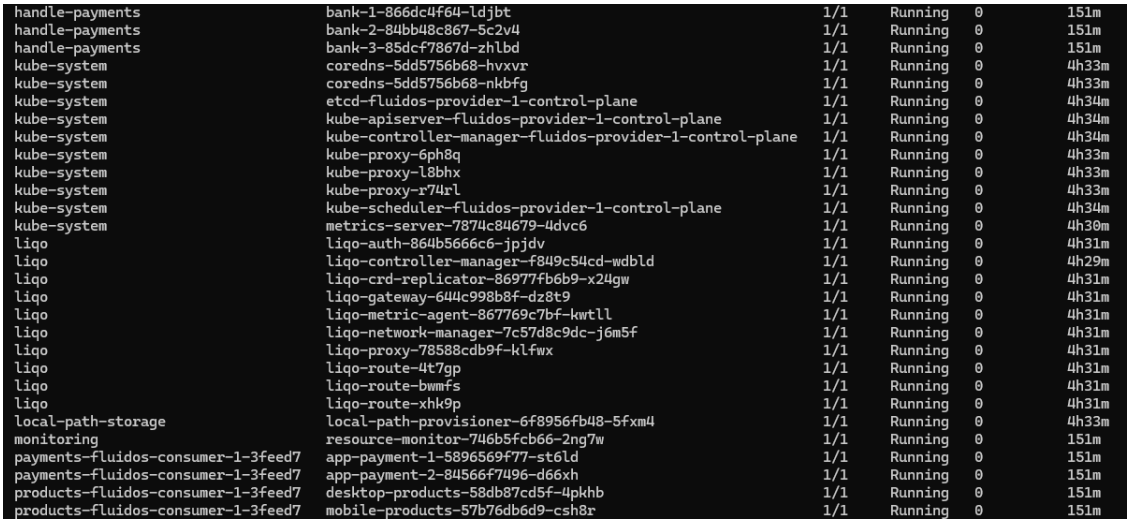
6.5.1 Test delle network policy

A titolo di esempio verrà mostrato il funzionamento di due network policy rilevanti. Il seguente intento armonizzato prevede che app-payment-1 del namespace payments possa comunicare con qualsiasi pod del provider:

Src: [app:app-payment-1 - name:payments], Dst: [*:* - *:*], DstPort: [80], ProtocolType: [TCP]

Per testare il corretto funzionamento della network policy, bisogna dapprima elencare tutti i pod interni al cluster Provider con il comando:

```
kubectl get pods --all-namespaces
```



handle-payments	bank-1-866dc4f64-ldjbt	1/1	Running	0	151m
handle-payments	bank-2-84bb48c867-5c2v4	1/1	Running	0	151m
handle-payments	bank-3-85dcf7867d-zhlbd	1/1	Running	0	151m
kube-system	coredns-5dd5756b68-hvxvr	1/1	Running	0	4h33m
kube-system	coredns-5dd5756b68-nkbfg	1/1	Running	0	4h33m
kube-system	etcd-fluidos-provider-1-control-plane	1/1	Running	0	4h34m
kube-system	kube-apiserver-fluidos-provider-1-control-plane	1/1	Running	0	4h34m
kube-system	kube-controller-manager-fluidos-provider-1-control-plane	1/1	Running	0	4h34m
kube-system	kube-proxy-6ph8q	1/1	Running	0	4h33m
kube-system	kube-proxy-l8bhx	1/1	Running	0	4h33m
kube-system	kube-proxy-r74rl	1/1	Running	0	4h33m
kube-system	kube-scheduler-fluidos-provider-1-control-plane	1/1	Running	0	4h34m
kube-system	metrics-server-7874c84679-4dvc6	1/1	Running	0	4h30m
liqo	liqo-auth-864b5666c6-jpjdv	1/1	Running	0	4h31m
liqo	liqo-controller-manager-f849c54cd-wdbld	1/1	Running	0	4h29m
liqo	liqo-crd-replicator-86977fb6b9-x24gw	1/1	Running	0	4h31m
liqo	liqo-gateway-644c998b8f-dz8t9	1/1	Running	0	4h31m
liqo	liqo-metric-agent-867769c7bf-kwtll	1/1	Running	0	4h31m
liqo	liqo-network-manager-7c57d8c9dc-j6m5f	1/1	Running	0	4h31m
liqo	liqo-proxy-78588cdb9f-klfwx	1/1	Running	0	4h31m
liqo	liqo-route-4t7gp	1/1	Running	0	4h31m
liqo	liqo-route-bwmfs	1/1	Running	0	4h31m
liqo	liqo-route-xhk9p	1/1	Running	0	4h31m
local-path-storage	local-path-provisioner-6f8956fb48-5fxm4	1/1	Running	0	4h33m
monitoring	resource-monitor-746b5fcb66-2ng7w	1/1	Running	0	151m
payments-fluidos-consumer-1-3feed7	app-payment-1-5896569f77-st6ld	1/1	Running	0	151m
payments-fluidos-consumer-1-3feed7	app-payment-2-84566f7496-d66xh	1/1	Running	0	151m
products-fluidos-consumer-1-3feed7	desktop-products-58db87cd5f-4pkhb	1/1	Running	0	151m
products-fluidos-consumer-1-3feed7	mobile-products-57b76db6d9-csh8r	1/1	Running	0	151m

Figura 6.19. risultato get pods

In particolare ci interesserà mostrare la connessione fra il pod *app-payment-1-5896569f77-st6ld* ed un pod qualsiasi che è interno ad un namespace del provider, per esempio: *bank-1-866dc4f64-ldjbt* del namespace handle-payment.
Per ottenere l' indirizzo IP di quest' ultimo pod, bisogna far uso del comando:

```
kubectl describe pods -n handle-payments
```

che mostrerà l' IP di tale pod.

```
Name:             bank-1-866dc4f64-ldjbt
Namespace:        handle-payments
Priority:         0
Service Account:  default
Node:             fluidos-provider-1-worker2/172.18.0.3
Start Time:       Mon, 21 Oct 2024 11:33:59 +0200
Labels:           app=bank-1
                  pod-template-hash=866dc4f64
Annotations:      cni.projectcalico.org/containerID: 622eb7c4b386501f059e517f5c7771bce056d4c85592b0c8b3d0ce2ce5de93ac
                  cni.projectcalico.org/podIP: 192.169.122.75/32
                  cni.projectcalico.org/podIPs: 192.169.122.75/32
Status:           Running
IP:               192.169.122.75
IPs:
  IP:           192.169.122.75
```

Figura 6.20. risultato describe pods

Successivamente bisognerà avviare una shell sul pod *app-payment-1-5896569f77-st6ld* con il comando:

```
kubectl exec -it app-payment-1-5896569f77-st6ld -n payments-
FLUIDOS-consumer-1-3feed7 -- /bin/sh
```

Su tale shell potremo testare la connessione sull' IP del pod bank-1. Come mostrato in figura 6.21 la network policy è stata applicata correttamente:

```
user@MSI:/mnt/c/Users/salva/Desktop/node-main/tools/scripts/Demo_da_presentare_last$ kubectl exec -it app-payment-1-5896569f
77-st6ld -n payments-fluidos-consumer-1-3feed7 -- /bin/sh
/ # wget 192.169.122.75
Connecting to 192.169.122.75 (192.169.122.75:80)
saving to 'index.html'
index.html           100% |********************************************************************************|   615  0:00:00 ETA
'index.html' saved
```

Figura 6.21. risultato connessione con bank-1

come contro esempio proviamo a connettere il pod app-payment-1 del namespace offlodato payments, con il pod mobile-products del namespace offlodato products. Dalla figura 6.22, otteniamo l' IP di quest' ultimo pod.

```
Name:             mobile-products-57b76db6d9-csh8r
Namespace:        products-fluidos-consumer-1-3feed7
Priority:         0
Service Account:  default
Node:             fluidos-provider-1-worker2/172.18.0.3
Start Time:       Mon, 21 Oct 2024 11:34:04 +0200
Labels:           app=mobile-products
                  liqo.io/managed-by=shadowpod
                  pod-template-hash=57b76db6d9
                  virtualkubelet.liqo.io/destination=8257b165-21dd-4e3b-9871-cae5131644c7
                  virtualkubelet.liqo.io/nodename=liqo-fluidos-provider-1
                  virtualkubelet.liqo.io/origin=eeab77f2-b6aa-4f6a-b1b1-981d5c649bb2
Annotations:      cni.projectcalico.org/containerID: d4f49faf8350f7e2d400c1b14a41bbc2d9fcf2b88d71bb6b18666a8a6652c755
                  cni.projectcalico.org/podIP: 192.169.122.83/32
                  cni.projectcalico.org/podIPs: 192.169.122.83/32
Status:           Running
IP:               192.169.122.83
IPs:
  IP:           192.169.122.83
```

Figura 6.22. mobile products IP

Se nel pod app-payment-1 provassimo a connetterci a tale IP [vedi Figura 6.23], la connessione fallirebbe in quanto non c'è nessuna network policy che ci consente tale connessione in quanto le network policy di **deny-default** vietano qualsiasi connessione che non sia esplicitamente consentita.

```
/ # wget 192.169.122.83
Connecting to 192.169.122.83 (192.169.122.83:80)
wget: can't connect to remote host (192.169.122.83): Connection timed out
```

Figura 6.23. connessione con mobile-products

Un altro esempio rilevante, riguarda l' intento armonizzato:

Src: [app:mobile-products - name:products], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]

ossia l' intento che vuole permettere il pod mobile-products offlodato di comunicare con un set di indirizzi CIDR proprietario di Google. Come mostrato in figura 6.24, eseguendo una shell sul pod mobile-products, quest' ultimo riesce a connettersi a Google grazie alla network policy che abilita tale connessione.

```
user@MSI:/mnt/c/Users/salva/Desktop/node-main/tools/scripts/Demo_da_presentare_last$ kubectl exec -it mobile-products-57b76d
b6d9-csh8r -n products-fluidos-consumer-1-3feed7 -- /bin/sh
/ # wget google.it
Connecting to google.it (142.250.180.163:80)
Connecting to www.google.it (142.250.180.131:80)
saving to 'index.html'
index.html           100% |********************************************************************************| 21149  0:00:00 ETA
'index.html' saved
```

Figura 6.24. connessione con IP di google

Se invece si prova a connettere app-payment-1 con Google.it [vedi Figura 6.25], la connessione fallirebbe in quanto non c'è alcuna network policy che esplicitamente abiliti tale connessione.

```
user@MSI:/mnt/c/Users/salva/Desktop/node-main/tools/scripts/Demo_da_presentare_last$ kubectl exec -it app-payment-1-5896569f
77-st6ld -n payments-fluidos-consumer-1-3feed7 -- /bin/sh
/ # wget google.it
Connecting to google.it (142.250.180.163:80)
wget: can't connect to remote host (142.250.180.163): Connection timed out
```

Figura 6.25. connessione con IP di google fallita

Capitolo 7

Conclusioni

Nel lavoro svolto in questa tesi è stata fornita l'implementazione di un meccanismo automatico per l' applicazione di policy di sicurezza nel contesto del cloud continuum. In particolare è stato affrontato il problema nel contesto del progetto europeo FLUIDOS, in cui diversi domini amministrativi richiedono o ospitano risorse di altri domini amministrativi. In tale contesto richiedente ed offerente possono enunciare delle regole ad alto livello , sotto forma di intenti , che possono essere discordanti fra loro. E' quindi risultato fondamentale l' implementazione di un meccanismo che in modo automatico possa conciliare eventuali intenti discordanti e successivamente tradurli ed applicarli in network policy. Nel corso della progettazione di tale meccanismo, sono state affrontate diverse sfide che possono essere prese da esempio per lavori futuri. In particolare, è stata trovata una soluzione peculiare per l' autenticazione, svolta tramite service account e token di autenticazione che ha permesso al controllore di autenticarsi e ricevere i permessi necessari per compiere le sue mansioni. Un ulteriore sfida è stata l' integrazione con il modulo di armonizzazione implementato da un altro tesista per il progetto FLUIDOS. Di rilevante importanza è stata l' implementazione del modulo di traduzione, in quanto , come mostrano i risultati ottenuti, il traduttore è in grado di coprire tutte le casistiche di traduzione esprimibili tramite gli intenti definiti in FLUIDOS. Il caso d' uso riportato, mette in luce le potenzialità del controllore il quale è in grado di ottenere dal cluster in cui è in esecuzione le informazioni necessarie per applicare le network policy tradotte e le network policy di default. Queste ultime sono necessarie in quanto permettono di limitare le comunicazioni a solo quelle esplicitamente consentite e di permettere la comunicazione con il DNS del cluster per raggiungere eventuali hostname esterni nel caso in cui ci fosse una policy che lo permetterebbe. Il lavoro svolto è stato sviluppato pensando al progetto FLUIDOS, che adotta Kubernetes come orchestratore. Tuttavia, in un futuro in cui il paradigma del liquid computing sarà sempre più centrale, con diversi domini amministrativi che collaborano per condividere risorse e servizi, risulterà essenziale disporre di meccanismi analoghi anche per orchestratori differenti. I risultati ottenuti da questa tesi possono quindi rappresentare un punto di partenza per l'implementazione di sistemi di automazione delle policy in un contesto più ampio, in grado di supportare altri orchestratori e rendere possibile una gestione sicura e flessibile delle risorse su scala globale.

Bibliografia

[1] K. Documentation. (2024) Che cos'è kubernetes? Accessed: 2024-11-13. [Online]. Available: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/

[2] Medium. (2024) Components of kubernetes architecture. [Online]. Available: https://gauravguptacloud.medium.com/components-of-kubernetes-architecture-6feea4d5c712

[3] I. C. Education. (2024) Kubernetes networking. Accessed: 2024-11-13. [Online]. Available: https://www.ibm.com/it-it/topics/kubernetes-networking

[4] J. Walker. (2023) Kubernetes network policy – guide with examples. [Online]. Available: https://spacelift.io/blog/kubernetes-network-policy

[5] Kubernetes. (2024) Network plugins. Accessed: 2024-11-13. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

[6] Tigera. (2024) About calico. Accessed: 2024-11-13. [Online]. Available: https://docs.tigera.io/calico/latest/about

[7] Fluidos. (2024) About fluidos. Accessed: 2024-11-13. [Online]. Available: https://fluidos.eu/about/

[8] Liqo. (2024) Liqo documentation. Accessed: 2024-11-13. [Online]. Available: https://docs.liqo.io/en/v1.0.0-rc.2/

[9] S. Galantino, E. Albanese, N. Asadov, S. Braghin, F. Cappa, A. Colli-Vignarelli, A. Y. Majid, E. Marin, J. Marino, L. Moro, L. Nedoshivina, F. Risso, D. Siracusa, A. Skarmeta, and L. Zuanazzi, "Building the cloud continuum with rear," pp. 67–72, 2024.

[10] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "An intent-based solution for network isolation in kubernetes," pp. 381–386, 2024.