Laurea Magistrale in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Orchestrazione ed armonizzazione di policy per la protezione nel computing continuum

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

Candidato
Luca Ruberto

Anno Accademico 2023-2024

Questo lavoro è soggetto a Licenza Creative Commons

Alla mia famiglia

Sommario

In un mondo sempre più connesso e caratterizzato da applicazioni sempre più complesse, emerge la necessità di un nuovo approccio nel panorama del cloud computing. Il liquid computing rappresenta una rivoluzione in quest'ambito, fornendo agli utenti l'accesso ad un pool di risorse senza confini, accessibile in qualsiasi momento e da qualsiasi posizione geografica. Favorendo un continuum di risorse tra diversi domini di amministrazione, si previene il sottoutilizzo delle risorse e si offre un deployment delle applicazioni maggiormente flessibile, scalabile e resiliente. Il liquid computing prevede un approccio decentralizzato, multiproprietario e intent-driven, garantendo un continuum di risorse e servizi. FLUIDOS (Flexible, scaLable, secUre and decentralIzeD Operating System) è un progetto europeo che nasce con l'obiettivo di implementare questa visione. Il progetto si concentra su tre proprietà fondamentali: trasparenza del deployment, trasparenza delle comunicazioni e trasparenza delle risorse disponibili. Questa tesi affronta nello specifico lo sviluppo di un orchestratore di sicurezza, con l'obiettivo di garantire un adeguato isolamento a livello di rete per i diversi workload eseguiti nel continuum. Una delle problematiche introdotte dal liquid computing è la protezione dei bordi. Il concetto di singolo server fisico non esiste più in questo caso, ma si può parlare di cluster virtuale, ossia l'insieme logico di tutte le risorse appartenenti ad uno stesso utente e distribuite potenzialmente su più cluster fisici. Il bordo del cluster virtuale diventa un bordo dinamico, capace di estendersi su più cluster fisici. Coinvolgendo cluster e dispositivi diversi, richiede nuove soluzioni per essere protetto. Nello specifico la protezione del cluster che mette a disposizione le proprie risorse da potenziali danni causati dalle applicazioni offloadate e la protezione delle stesse applicazioni da furto di dati e codice e interferenze da parte dell'host. L'idea proposta prevede l'integrazione con il processo di acquisizione delle risorse, in modo da implementare un corretto isolamento non appena il processo di peering viene completato. Il tutto si basa su degli intenti, policy ad alto livello in cui è possibile esprimere le richieste in termini di connessioni di rete concesse o negate. Il processo di condivisione delle risorse prevede due attori diversi: il Consumer e il Provider. Il Provider mette a disposizione le proprie risorse, il Consumer invece sfrutta le risorse messe a disposizione dal Provider. Ogni attore può dunque esprimere le proprie richieste sotto forma di intenti, producendo due set di intenti differenti. Il processo di offloading delle risorse prevede dapprima uno scambio delle richieste, in secondo luogo una verifica della compatibilità di queste richieste (Verify) ed infine un processo di armonizzazione (Harmonize) in caso di esito positivo. Il risultato dell'armonizzazione viene poi tradotto in primitive di isolamento (Kubernetes Network Policies) ed applicato all'interno del cluster. Dopo una prima fase di descrizione della struttura dell'orchestratore e del suo funzionamento complessivo, l'attenzione si concentrerà

sull'implementazione e sulla validazione di due moduli nello specifico: Verify e Harmonize.

Ringraziamenti

È stato un lungo (lunghissimo) viaggio, ma finalmente è giunto al termine.
Un ringraziamento ai miei relatori, al prof. Sisto e al prof.Valenza, per avermi dato l'opportunità di lavorare a questa tesi, al dott. Bringhenti e in particolar modo al dott. Pizzato, per avermi seguito e consigliato in questi mesi.
Un ringraziamento a tutta la mia famiglia.
Ai miei genitori, per avermi dato la possibilità di frequentare l'università nonostante tutto.
A mia madre, per essermi stata sempre vicino, per non aver mai mollato, è grazie a lei se sono qui a scrivere queste righe oggi.
A mio padre, per avermi consigliato e supportato soprattutto nei momenti di difficoltà durante questo percorso.
A mio fratello, per esserci sempre stato.
A mia nonna, per avermi cresciuto, per tutto l'amore e i sacrifici fatti con nonno per noi.
Alle mie zie, a Manuele, per il loro affetto e per i momenti passati insieme.
A miei nonni, a mia zia, a tutti quelli che non ci sono più
Agli amici di una vita, per tutte le avventure vissuti insieme.
A "Tavola Calda", per tutti gli aneddoti che ormai allietano le mie giornate.
Al gruppo del fantacalcio, per le risate e gli sfottò che accompagnano ogni stagione.
A me stesso, per avercela fatta.

Indice

Elenco delle figure 9

Listings 11

1 Introduzione 13

1.1 Descrizione della tesi 14

2 Liquid Computing 15

2.1 Cloud computing 15

2.2 Liquid computing 17

2.2.1 Principali caratteristiche 17

2.2.2 I pilastri del liquid computing 18

2.2.3 Scenari di deployment 20

3 Kubernetes e Liqo 22

3.1 Kubernetes 22

3.1.1 Architettura 23

3.1.2 Object 24

3.2 Framework e tool 27

3.3 Liqo 28

4 FLUIDOS 32

4.1 Introduzione 32

4.2 Obiettivi 32

4.3 FLUIDOS computing continuum 33

4.4 FLUIDOS Node 35

4.4.1 Componenti 36

5 Obiettivi della tesi 41

6 Approccio per la protezione dei bordi di rete 43
6.1 Il problema 43
6.2 Tipi di comunicazione 43
6.3 Gli Intenti 44
6.4 L'Integrazione 47
6.5 Il Workflow 47
6.6 Il Controller 49
6.7 Verifica della compatibilità 49
6.8 Armonizzazione degli Intenti 49
6.9 Traduzione degli Intenti armonizzati 50

7 Implementazione e validazione 51
7.1 Verify 51
7.2 Harmonize 54
7.3 Validazione 58

8 Conclusioni 73

Bibliografia 74

Elenco delle figure

2.1 L'evoluzione del deployment delle applicazioni. Singolo server, Virtualizzazione e Containerizzazione 17

3.1 I componenti di un cluster Kubernetes 22
3.2 Pod 25
3.3 Offloading in Liqo 30

4.1 Deployment delle applicazioni con approccio tradizionale vs FLUIDOS 33
4.2 Comunicazioni service-to-service vs FLUIDOS 34
4.3 Risorse disponibili con l'approccio tradizionale vs FLUIDOS 35
4.4 FLUIDOS Node 35

6.1 Tipi di comunicazione in FLUIDOS 44
6.2 Private Intent 45
6.3 Request Intent 45
6.4 Authorization Intent 46
6.5 Workflow-Integrazione 48

7.1 Verify 52
7.2 Harmonize 54
7.3 I due cluster prima del processo di offloading 58
7.4 Flavor 1 60
7.5 Flavor 2 61
7.6 Flavor 3 61
7.7 Installazione del FLUIDOS Node 62
7.8 Dati del Provider 63
7.9 Verify - Flavor 1 64
7.10 Verify - Flavor 2 65
7.11 Verify - Flavor 3 65

7.12 Reservation 67
7.13 Contratto 68
7.14 Stato del Peering 68
7.15 Discordanze Tipo 1 e 2 70
7.16 Discordanze Tipo 3 71
7.17 Risorse offloadate 72
7.18 Esempio di connessione del Consumer 72
7.19 Esempio di connessione del Provider 72

Listings

3.1 Pod.yaml 25
3.2 Service.yaml 26
3.3 Deployment.yaml 27
4.1 Flavor.yaml 36
4.2 FlavorType.yaml 37
6.1 Esempio File MSPL 46
7.1 Request Intent 59
7.2 Authorization Intent (1) 60
7.3 Authorization Intent (2) 60
7.4 Authorization Intent (3) 61
7.5 PeeringCandidate 64
7.6 Reservation 66
7.7 Allocation.yaml 67
7.8 Contract 68

Capitolo 1

Introduzione

Negli ultimi anni, il cloud computing ha radicalmente trasformato il modo in cui accediamo e utilizziamo le risorse informatiche distribuite a livello globale, così come il deployment delle applicazioni. La proliferazione dei dispositivi connessi e una crescente complessità delle applicazioni, ha reso necessario un nuovo approccio nell'accesso alle risorse: il liquid computing. Il liquid computing permette l'accesso ad un insieme unico di risorse, accessibile in qualsiasi momento e da qualsiasi luogo, abbattendo i confini geografici e amministrativi. Questo continuum di risorse consente di rimediare al problema del sottoutilizzo delle risorse, oltre ai problemi di scalabilità, flessibilità e resilienza nel deployment delle applicazioni. FLUIDOS (lexible, scaLable, secUre and decentralIzeD Operating System) è un progetto europeo che mette in pratica la visione del liquid computing. L'obiettivo è appunto quello di creare un continuum di risorse distribuite, sfruttando in modo efficiente le risorse inutilizzate ai bordi della rete, garantendo comunque la sicurezza e la trasparenza delle comunicazioni. Nel contesto del liquid computing e quindi di FLUIDOS, il concetto di un singolo server fisico non esiste più. Emerge invece il concetto di cluster virtuale, un insieme logico di risorse potenzialmente distribuite su più infrastrutture fisiche e appartenenti ad un singolo utente o ad un'applicazione. Il concetto stesso di bordo diventa dinamico, potendosi estendere su cluster fisici diversi. Questi cambiamenti richiedono necessariamente una protezione dei bordi e delle risorse. Da queste considerazioni, è nata la necessità lo sviluppo di un orchestratore di sicurezza, capace di garantire un isolamento adeguato a livello di rete per i vari attori e carichi di lavoro eseguiti nel continuum. In particolar modo è necessaria la protezione delle risorse del Provider che mette a disposizione e del Consumer che le utilizza. Introducendo il concetto di intento, ossia policy definite ad alto livello, è possibile definire per i vari attori in gioco le proprie richieste in termini di connessioni autorizzate o richieste. Nel lavoro di tesi si affronta il problema della richiesta di offloading delle risorse del Consumer nel cluster remoto del Provider. L'integrazione tra moduli già esistenti, come quelli del FLUIDOS Node, e moduli sviluppati nel corso di questa tesi e in tesi parallele, permette la gestione dello scambio di intenti, la validazione e l'armonizzazione, al fine di prenotare e acquisire le risorse per l'offloading e stabilire una connessione di peering tra Consumer e Provider.

1.1 Descrizione della tesi

- Nel Capitolo [2] viene introdotto e illustrato il concetto di Liquid Computing. Dopo una breve introduzione sul settore del cloud computing, l'evoluzione nel deployment delle applicazioni e le problematiche delle tecnologie attuali, si affronta il tema centrale del Liquid Computing e le sue caratteristiche.
- Il Capitolo [3] presenta tecnologie necessarie per lo sviluppo del progetto di tesi. Si trattano in dettaglio due framework come Kubernetes e Liqo, con la descrizione della loro architettura.
- Il Capitolo [4] presenta il progetto FLUIDOS illustrandone gli obiettivi e le caratteristiche distintive. Viene anche descritto il FLUIDOS Node, ambiente fondamentale all'intento di FLUIDOS.
- Nel Capitolo [5] viene descritto in maniera più approfondita il lavoro di tesi.
- Il Capitolo [6] affronta la problematica della protezione dei bordi di rete nel contesto del cloud computing. Dopo aver illustrato la problematica e alcuni concetti necessari, viene affrontato il problema dell'integrazione dei due moduli oggetto del lavoro di tesi, Verify e Harmonize, con un Controller, un Traduttore e le varie componenti del FLUIDOS Node.
- Il Capitolo [7] presenta l'implementazione dei vari moduli, descrivendone prima le caratteristiche e il design e in seguito illustrando un caso di studio per la validazione del lavoro.
- Il Capitolo [8] è il capitolo conclusivo della tesi, con la descrizione degli obiettivi raggiunti e possibili lavori futuri.

Capitolo 2

Liquid Computing

2.1 Cloud computing

Il settore del cloud computing ha visto una grande evoluzione nel corso degli ultimi anni, e con esso anche il concetto di deployment delle applicazioni. Il cloud computing ha permesso infatti ad aziende e utenti di accedere a risorse e servizi computazionali distribuiti globalmente ed accessibili in modo flessibile tramite internet. In passato, l'approccio utilizzato dalle aziende per il deployment delle applicazione era quello di "un server, un'applicazione", che prevedeva l'assegnazione di un server fisico per ogni applicazione. Le problematiche di tale approccio erano molteplici: scarsa scalabilità, sottoutilizzo delle risorse, costi elevati e complessità. La virtualizzazione può essere considerata una rivoluzione in questo contesto, correggendo alcune criticità del vecchio approccio. La virtualizzazione introduce quindi il concetto di astrazione delle risorse, permettendo di eseguire più Virtual Machine su un singolo nodo fisico, separando così l'hardware dal sistema operativo. Questa nuova tecnologia garantisce un maggiore isolamento, un aumento dell'affidabilità, oltre alla possibilità di personalizzazione dell'ambiente di lavoro. Nei contesti in cui il consumo eccessivo di risorse fisiche, come RAM e CPU, e del tempo non sono trascurabili, la virtualizzazione mostra i suoi limiti. La containerizzazione [1] nasce come risposta a queste esigenze, fornendo una soluzione più leggera per il packaging di applicazioni in un formato compatibile con sistemi diversi. I container introducono un concetto simile alla virtualizzazione, isolando però le applicazioni a livello del sistema operativo invece che a livello hardware, favorendo così la portabilità delle applicazioni, con un focus sul concetto di Platform-as-a-Service (PaaS) [1]. I container possono essere considerati come un'unica entità che contiene al suo interno codice, librerie, runtime e tutto il necessario per eseguire un'applicazione, con la possibilità di interagire anche tra di loro. La granularità dei container, intesa come il numero e il tipo di applicazioni al suo interno, può variare. In sostanza, la containerizzazione fornisce un ambiente di esecuzione più leggero e la possibilità di sviluppare, testare e distribuire applicazioni verso un gran

[1] Modello di cloud computing che permette agli sviluppatori di accedere ad un ambiente per lo sviluppo, la gestione e la creazione delle applicazioni concentrandosi solo sul codice, senza la necessità di gestire l'infrastruttura sottostante.

numero di server senza particolari configurazioni, con risultati indipendenti dal SO sottostante. L'elevato numero di container può portare però ad una maggiore complessità e difficoltà nella loro gestione, problematica che invece non si presentava con la virtualizzazione, essendo le macchine virtuali presenti in numero minore. La containerizzazione ha aperto la strada alla rivoluzione cloud native, con applicazioni sviluppate nativamente per lavorare in un ambiente cloud. A differenza delle applicazioni tradizionali, che spesso sono monolitiche, le applicazioni cloud native sono divise in microservizi. In questa architettura modulare, ad ogni microservizio può essere associato un container specifico. La gestione può essere affidata ad un orchestratore, e Kubernetes è la piattaforma maggiormente utilizzata in questi casi. L'orchestratore permette la gestione di interi data center, gestendo il ciclo di vita dei microservizi. La diffusione dell'Internet of Things (IoT) ha portato la necessità di cambiare di cambiare il modo in cui sono processati i dati. Le applicazioni IoT spesso richiedono di processare grandi quantità di dati, la gestione di dati privati e tempi di risposta brevi, da qui lo spostamento della gestione e del trattamento dei dati ai bordi della rete e la nascita della visione dell'Edge Computing [2]. Con il termine *edge* si intendono tutte le risorse informatiche e di rete nel percorso tra la sorgente dei dati e i data center. La trasformazione dell'infrastruttura in una schiera di silos[2] isolati e connessi, ha introdotto però il problema della frammentazione e del sottoutilizzo, impedendo la distribuzione seamless[3] di applicazioni tipica degli approcci distribuiti. Il problema della frammentazione interessa anche i data center, con molte compagnie che hanno visto diffondersi il fenomeno del cluster sprawl [4]. L'ottimizzazione dei costi, l'alta disponibilità e la distribuzione geografica, richiesti dagli approcci multi-cloud e hybrid-cloud e dall'esigenza di scalabilità ha portato a questo scenario. La frammentazione limita inoltre la flessibilità e il dinamismo nell'assegnazione dei carichi di lavoro, costringendo un'applicazione ad essere associata inizialmente ad un'infrastruttura specifica. Una gestione ottimizzata delle risorse e un deployment efficiente delle applicazioni risulta dunque difficile in questo contesto. L'esecuzione di applicazioni complesse composte da diversi microservizi, richiede invece l'interazione tra diverse infrastrutture. L'esigenza di gestire in modo dinamico più applicazioni complesse, indipendentemente dalla loro posizione geografica, oltre a quella di scalabilità, flessibilità e resilienza ha aperto la strada alla nascita di un nuovo approccio: il *liquid computing*.

[2]Insieme di dati isolati.

[3]Senza soluzione di continuità. Termine utilizzato per evidenziare un'interazione fluida e priva di interruzioni.

[4]Termine che si riferisce alla diffusione crescete del numero di cluster all'interno di un'organizzazione.

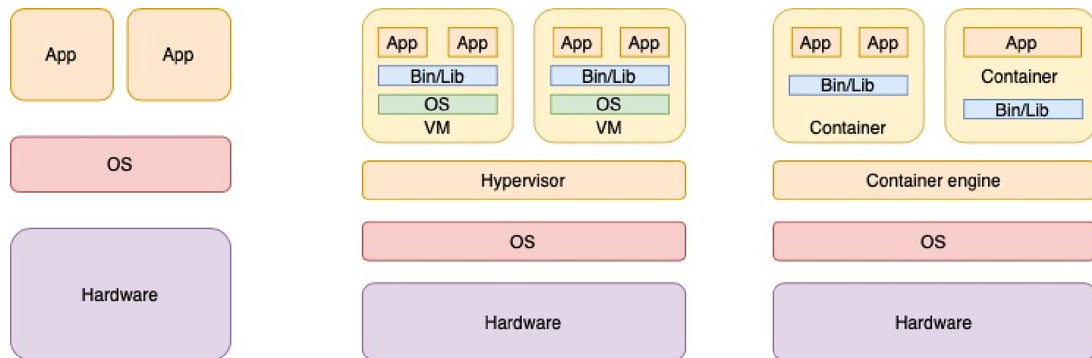

Figura 2.1. L'evoluzione del deployment delle applicazioni. Singolo server, Virtualizzazione e Containerizzazione

2.2 Liquid computing

Il liquid computing [5] [4] è un nuovo paradigma che mira a risolvere le problematiche appena illustrate. Partendo dai modelli tradizionali di edge e cloud computing, si propone di creare un ambiente altamente dinamico e fluido. Questo continuum informatico permette di avere un insieme unico di risorse, superando il concetto di confine tra i vari cluster che lo compongono.

2.2.1 Principali caratteristiche

Intent-driven

Il Consumer può esprimere, mediante policy ad alto livello, le proprie richieste ed i vincoli desiderati, senza dover conoscere i dettagli dell'infrastruttura sottostante. L'approccio seamless, eliminando la necessità di definire i bordi del cluster in modo netto, non richiede più una configurazione e una scelta di un'infrastruttura specifica per la propria applicazione. Il sistema ha dunque il compito di decidere l'ambiente di esecuzione più adatto in base ai requisiti (geografici, di costo, ecc...) scelti dall'utente. In questo modo si evita il fenomeno del cluster sprawling, svincolandosi dall'esigenza di associare ogni cluster ad una diversa proprietà. Rispetto all'approccio a silos si aprono maggiori opportunità per un utilizzo più contestualizzato delle risorse, che consente ottimizzazioni e maggiore scalabilità.

Architettura decentralizzata

Ogni attore ha la possibilità di mettere a disposizione le proprie risorse con gli stessi privilegi degli altri membri. In base alle richieste viene effettuata una ricerca dei dei vari peer disponibili prima di effettuare una connessione. Solo risorse astratte

[5]Termine coniato nel 2014 da InfoWorld [3] per indicare un flusso di dati e attività da un dispositivo all'altro, in riferimento ad Handoff di Apple. In ambito cloud ci si riferisce alla creazione di un continuum di risorse composte da infrastrutture edge e cloud.

(RAM, CPU, ecc...) possono essere richieste, garantendo così la protezione dei dati. La topologia può evolvere in seguito alle richieste indipendenti di ciascun cluster.

Multiproprietà

L'insieme delle risorse messe in condivisione si estende attraverso diversi domini amministrativi, nonostante i singoli cluster appartengano ad un solo proprietario. Ciascun autore ha la possibilità di decidere quali risorse e servizi mettere a disposizione e con chi, prima di stabilire un nuovo peering. L'infrastruttura ha il compito di configurare il sistema in base alle richieste dei vari attori in gioco ed evitare problemi tra di loro. Misure di sicurezza sono richieste per proteggere ciascun attore dall'altro e per proteggere l'intera infrastruttura dagli attori che mettono a disposizione le risorse.

Topologia fluida

Ogni componente può decidere autonomamente quando unirsi o abbandonare. In questo scenario dinamico entrano in gioco dispositivi di diverse dimensioni, dai data center di grandi dimensioni ai dispositivi personali. Il sistema deve prevedere la continua evoluzione delle risorse, garantendo un'esperienza d'uso sempre priva di interruzioni.

2.2.2 I pilastri del liquid computing

Nei prossimi paragrafi sono illustrate i pilastri per mettere in pratica la visione del liquid continuum. Kubernetes, grazie alla sua larga diffusione e alla possibilità di utilizzare API personalizzate, viene considerata in questo caso come la piattaforma più adatta per fungere da substrato al continuum di risorse su cui è basato. In ogni caso i concetti sono generali e possono essere applicati anche ad altre piattaforme.

Scoperta Dinamica e Peering

Le parti in gioco vengono definite come Consumer e Provider, con il primo che sfrutta le risorse messe a disposizione dal secondo. L'approccio decentralizzato non richiede la presenza di un'entità centrale in grado di stabilire manualmente le connessioni tra le varie parti. Le connessioni possono essere stabilite e interrotte in qualsiasi momento grazie a questo dinamismo. Il peering è inteso come connessione unidirezionale, con il Consumer che utilizza risorse e servizi del cluster remoto resi disponibili dal Provider. Questo processo prevede quattro fasi principali:

- La *Scoperta* dei cluster candidati con cui connettersi, da larghi domini a dispositivi locali.
- L'*Autenticazione* dei candidati selezionati con la creazione di un canale di comunicazione sicuro.

- La *Negoziazione delle risorse* tra Consumer e Provider, con lo scambio di messaggi con richieste e offerte. Il processo è policy-driven, con ogni cluster che può decidere ad ogni step se continuare la negoziazione o no. Il fine della negoziazione è quello di arrivare all'adozione di un contratto smart [5], in cui entrano in gioco soldi e risorse.

- La *Finalizzazione del peering*, con lo scambio dei parametri necessari per il processo di offloading delle risorse e per la configurazione dei meccanismi di isolamento.

Continuum di risorse gerarchico

Il cluster locale ottiene l'accesso logico alle risorse remote in seguito al completamento delle relazioni di peering. L'astrazione del continuum permette di preparare queste risorse per l'offloading delle applicazioni, rispettando i vincoli della multiproprietà ed evitando di condividere informazioni riservate. I cluster connessi vengono rappresentati, differentemente dai nodi associati ad ogni server fisico dell'approccio tradizionale, da *nodi locali*, *virtuali* e *grandi*. *Locali* perché collegati direttamente al cluster di cui sfruttano le risorse; *virtuali* perché rappresentano risorse non direttamente correlate ad un hardware specifico; *grandi*, di dimensioni potenzialmente maggiori rispetto ai nodi tradizionali. Vengono individuati dunque due possibili modelli di cluster. I *cluster estesi* composti da nodi fisici tradizionali (lavoratori) e da nodi virtuali, utili per l'ottimizzazione delle risorse, e i *cluster virtuali* composti invece esclusivamente da nodi virtuali. Questo secondo modello permette un'organizzazione gerarchica delle risorse, resa possibile dalla distribuzione dei carichi di lavoro ai cluster remoti in modo trasparente da parte dell'astrazione del nodo virtuale. Lo scheduler si occupa di selezionare il nodo di esecuzione adeguato in seguito alla distribuzione di un nuovo carico di lavoro nel cluster locale. Nel caso in cui il nodo scelto sia virtuale, è necessario un ulteriore livello di scheduling per assegnarlo ad un cluster fisico per la sua esecuzione. Le decisioni di scheduling possono essere dunque prese a più livelli di astrazione, migliorando l'affidabilità generale e riducendo il numero delle opzioni tra cui valutare. Questo schema è facilitato dalle API di Kubernetes, anche se in alcuni contesti si deve ricorrere ad una personalizzazione della logica.

Riflessione di Risorse e Servizi

La gestione dei propri carichi di lavoro allocati è assegnata ad ogni nodo virtuale, il cluster remoto è invece il responsabile della loro esecuzione. Da qui l'esigenza di una *riflessione delle risorse*: gli oggetti sono duplicati, esistendo sia nella loro forma *nativa* (nel cluster locale) che nella loro forma *ombra* (nel cluster remoto). Anche le modifiche e i cambiamenti devono essere riflessi, con la sincronizzazione delle risorse, rispettando sempre la protezione delle informazione ove richiesto. I cambiamenti possono avvenire in seguito a modifiche locali, come nel caso delle *riflessioni in uscita* o in seguito a modifiche nel cluster remoto, come nel caso delle *riflessioni in entrata*. Robustezza e scalabilità sono quindi garantite dalla riflessione delle informazioni.

Continuum di Rete

In aggiunta al continuum di risorse, è necessaria anche l'implementazione di un continuum di rete, vista la necessità dei microservizi di comunicare tra loro in modo diretto. L'utilizzo di indirizzi IP privati per le comunicazioni interne e l'esigenza di gestire un ambiente dinamico, data la natura stessa del liquid computing, rendono l'utilizzo del solo continuum di risorse non sufficiente a garantire il corretto funzionamento. Prendendo in considerazione un cluster centrale C ed n cluster periferici, sono possibili due differenti topologie. La topologia *hub and spoke* comporta tutto il passaggio del traffico attraverso il cluster centrale, con n interconnessioni tra i nodi periferici ed esso. È adatto nei casi in cui non sia necessaria la distribuzione su più cluster remoti e nei casi in cui sia presente una topologia di rete a stella. Nel secondo modello, *a maglia opportunistica*, c'è una interconnessione tra tutti i cluster periferici, che posso all'evenienza fungere da nodo centrale, dando vita a topologie dinamiche.

Storage e Continuum dei Dati

Utilizzando una pratica comune nel contesto dei Big Data, la *data gravity*, si dà vita ad un continuum dei dati, consentendo alle applicazioni di accedervi anche nel caso questi siano distribuiti su più cluster remoti. Questa pratica prevede lo spostamento della potenza computazionale verso il luogo in cui sono archiviati, rendendo il processo più efficiente visti i grandi volumi di dati da elaborare. In caso di quantità limitate di dati, è consentita l'estensione dei carichi di lavoro statici all'interno dei cluster.

2.2.3 Scenari di deployment

Elastic cluster

Il paradigma del liquid computing permette in questo caso di aggregare risorse e renderle disponibili come un unico pool di risorse, capace di espandersi e contrarsi in modo dinamico. Questo permette di ridurre la frammentazione dei cluster sparsi e di bilanciare e assorbire i picchi di carico (*cloud bursting*). L'*elastic cluster* può essere utilizzato sia negli scenari edge-computing, dove le risorse sono spesso limitate, sia negli scenari cloud tradizionali. Le risorse possono essere condivise tra data center locali e pubblici, avvicinando le applicazioni che richiedono una latenza bassa all'utente finale e destinando le infrastrutture con maggiori capacità computazionali i task più complessi. Grazie all'approccio decentralizzato e alla procedura di peering, è possibile la condivisione di risorse tra cluster appartenenti ad organizzazioni diverse.

Super cluster

Il super cluster è applicabile nei contesti in cui ci sono addirittura migliaia di piccoli cluster spesso situati ai bordi della rete, come nel caso dei grandi operatori

di telecomunicazione. Il punto di accesso unico facilita la gestione e il deployment delle applicazioni e risulta utile nei casi c'è bisogno di migrare un'applicazione da un cluster ad un altro per esigenze di disaster recovery o interventi infrastrutturali. Inoltre, oltre a ridurre il carico amministrativo, permette all'operatore di replicare lo stesso servizio su diversi cluster. Pur essendo all'apparenza un approccio centralizzato, ogni cluster di livello 2 mantiene la propria logica di orchestrazione, mitigando eventuali problemi di sincronizzazione e intervenendo durante le interruzioni di rete. È possibile inoltre combinare gli *elastic cluster* con i *super cluster*, creando un ambiente ancora più dinamico.

Brokering cluster

Nei casi in cui le applicazioni sono complesse, il liquid computing prevede la creazione di una nuova Resource and Service Exchange Points (RXPs), in cui le terze parti fungono da intermediari tra Consumer e Provider. Il Consumer può così connettersi ad un RXP ed accedere subito ad un sete di risorse. Questo approccio riduce la complessità ed offre la possibilità anche ai piccoli attori di partecipare al mercato cloud, mettendo a disposizione le proprie risorse.

Capitolo 3

Kubernetes e Liqo

In questo capitolo verranno illustrati vari framework e tool necessari per l'installazione e per l'esecuzione dell'ambiente di sviluppo nell'ambito della realizzazione dell'orchestratore di sicurezza.

3.1 Kubernetes

Kubernetes (K8s) è una piattaforma open-source utilizzato per il deployment, l'esecuzione e la scalabilità di applicazioni containerizzate (divise in moduli più piccoli). Il suo ruolo è quello di orchestratore dei container, aumentando l'affidabilità e le performance. Creato inizialmente da alcuni dipendenti di Google nel 2014, è ora mantenuto dalla CNCF (Cloud Native Computing Foundation) [6][7].

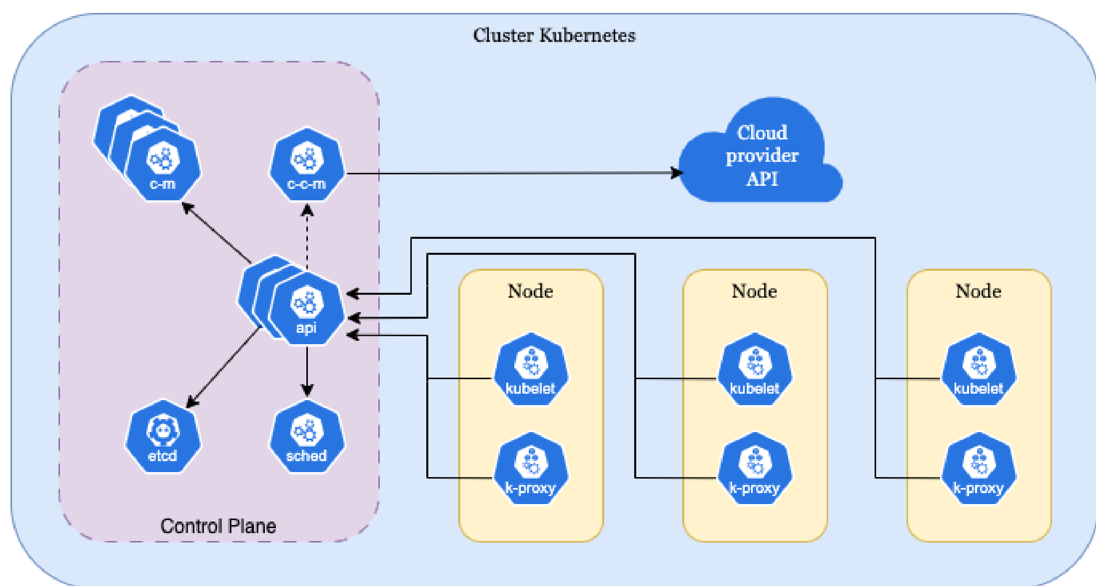

Figura 3.1. I componenti di un cluster Kubernetes

3.1.1 Architettura

Un cluster di Kubernetes è costituito da un Control Plane e da uno o più Worker Node, che eseguono le applicazioni containerizzate. I Worker Node ospitano i Pod che sono componenti del carico di lavoro delle applicazioni. Il Control Plane si occupa di gestire i Worker Node e i Pod nel cluster. Generalmente il Control Plane viene eseguito su più computer e il cluster di solito gestisce più nodi, fornendo tolleranza ai guasti e alta disponibilità.

Control Plane

Il Control Plane, conosciuto anche come Master Node è il gestore del cluster, controlla e coordina tutte le operazioni prendendo globali sul cluster e favorendo la comunicazione tra i vari componenti. Nonostante possa essere eseguito su qualsiasi macchina nel cluster, per semplicità si sceglie di eseguire tutti i componenti del Control Plane sulla stessa macchina impedendo l'esecuzione dei container degli utenti.

API Server L'API Server rappresenta il frontend del Kubernetes Control Plane, esponendo le API di Kubernetes che permettono di interagire con esso.

etcd Archivio di key-value per tutti i dati importanti per il funzionamento di Kubernetes, come dati di configurazione, metadati o stati.

Scheduler Lo Scheduler si occupa di assegnare i Pod appena creati ad un Worker Node, tenendo in considerazione i requisiti delle risorse, i vincoli, le regole di affinità per ottimizzare al meglio il processo.

Controller Manager Questo componente è il responsabile dell'esecuzione dei processi del controller e del monitoraggio dei vari stati degli oggetti. In base ai cambiamenti di stato rilevati prende le proprie decisioni per eventualmente ripristinare lo stato desiderato. Esistono diverse tipologie di controller:

- **Node Controller**: Gestisce i casi in cui un nodo smetta di funzionare.
- **Job Controller**: Osserva gli oggetti Job generati per una singola esecuzione e crea i Pod necessari a completarla
- **EndpointSlice Controller**: Popola gli oggetti EndpointSlice
- **ServiceAccount Controller**: Genera ServiceAccount di default per i nuovi namespace.

Cloud Controller Manager Nei casi in cui Kubernetes non è in esecuzione in un ambiente di sviluppo locale, è richiesto un componente che si occupa di gestire la logica per l'integrazione con il Provider cloud. Il Cloud Controller Manager si occupa soltanto dei componenti che interagiscono con la piattaforma cloud e non dei componenti che interagiscono esclusivamente con il cluster. I controller che possono dipendere dal cloud Provider sono:

- **Node Controller**: Verifica se un nodo è stato eliminato nel cloud dopo che ha smesso di rispondere.
- **Route Controller**: Per configurare i percorsi(route) nell'infrastruttura cloud sottostante.
- **Service Controller**: Gestisce la creazione, l'aggiornamento e l'eliminazione dei load balancers del cloud Provider.

Node

Un Nodo (Node)[8] può essere una macchina fisica o virtuale a seconda del cluster. Esso contiene i servizi per gestire i Pod ed è a sua volta gestito dal Control Plane. Sono previsti più nodi in un cluster, ad esclusione di ambienti di apprendimento o con risorse limitate.

Kubelet

Agente responsabile dell'implementazione dei Pod e dei Node, agendo al livello di esecuzione dei container. Si occupa dell'esecuzione e del mantenimento dello stato desiderato.

Kube Proxy

Proxy di rete eseguito su ciascun nodo del cluster, in grado di raggruppare Pod in base a politiche di accesso comuni.

Container runtime

Responsabile dell'esecuzione dei container e del loro ciclo di vita all'interno dell'ambiente di Kubernetes.

3.1.2 Object

Un Object in Kubernetes è un'entità persistente. Viene utilizzato per rappresentare uno stato e può essere espresso nel formato .yaml.

- **apiVersion**: Specifica la versione dell'API Kubernetes da utilizzare.

- **kind**: Indica il tipo di oggetto: Pod, Deployment, Service.
- **metadata**: Racchiude informazioni per identificare l'oggetto come name, namespace e label.
- **spec**: Definisce lo stato desiderato per un oggetto.

Esistono vari tipi di Object in Kubernetes e verranno ora illustrati i principali.

Pod

I Pod rappresentano la più piccola unità di deployment in Kubernetes. Un Pod incapsula uno o più container che condividono risorse di storage e di rete, un indirizzo IP univoco e informazioni sulla loro esecuzione. I container all'interno di un Pod condividono lo stesso ciclo di vita.

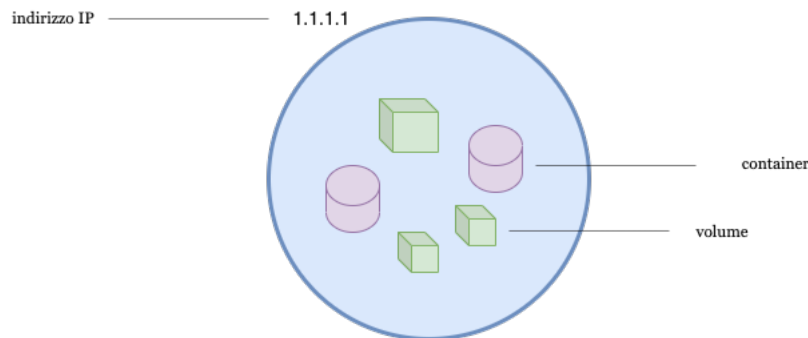

Figura 3.2. Pod

Listing 3.1. Pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
 name: my-pod
spec:
 containers:
 - name: my-container
   image: nginx:latest
   ports:
   - containerPort: 80
```

Namespace

I Namespace forniscono un meccanismo per isolare le risorse all'interno di un cluster. Il nome delle risorse deve essere univoco all'interno dello stesso namespace ma non tra tutti i namespace. È possibile dividere le risorse di un cluster tra più utenti. I namespace iniziali di Kubernetes sono:

- **default**: Ti dà la possibilità di iniziare senza scegliere un particolare namespace.
- **kube-node-lease**: Utilizzato per memorizzare lo stato dei nodi in modo che il control plane possa rilevare malfunzionamenti.
- **kube-public**: Utilizzato per informazioni che devono essere condivise nel cluster.
- **kube-system**: Riservato per le risorse di sistema.

Creare un nuovo namespace:

```
kubectl create namespace new-namespace
```

Elencare i namespace:

```
kubectl get namespaces
```

Service

I Service sono un'astrazione che permettono di raggruppare più Pod in una rete, definendo una policy mediante la quale accedervi. Ad ogni Service viene assegnato un indirizzo IP virtuale, differente da quello dei Pod, aggirando la loro natura effimera. Tipi di Service:

- **ClusterIP**: Tipo predefino, espone il Service su un indirizzo IP interno al cluster.
- **NodePort**: Espone il Service su una porta specifica del nodo.
- **LoadBalancer**: Crea automaticamente un load balancer fornito dal Provider cloud per esporre esternamente il Service.
- **ExternalName**: Utilizzato per reindirizzare il traffico su un alias DNS esterno al cluster.

Creare un Service:

```
kubectl apply -f myservice.yaml
```

Elencare i Service:

```
kubectl get services
```

Listing 3.2. Service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
```

```
    myApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Deployment

Il Deployment fornisce un'astrazione per gestire i Pod. Garantisce che il numero di repliche di un Pod sia sempre in esecuzione. Il Deployment fornisce meccanismi per gestire lo scaling, l'aggiornamento e il rollback delle applicazioni.
Creare un Deployment:

```
kubectl apply -f mydeployment.yaml
```

Elencare i Deployment:

```
kubectl get deployments
```

Listing 3.3. Deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

3.2 Framework e tool

K¡ind

KIND (Kubernetes IN Docker) [9] è un tool utile per eseguire cluster Kubernetes in locale utilizzando container Docker come ”nodi” del cluster.

Per installare KIND è dunque necessario installare prima Docker. Per creare un cluster:

```
kind create cluster
```

Per interagire con il cluster:

```
kubectl cluster-info --context kind-kind
```

Per cancellare un cluster:

```
kind delete cluster
```

È anche possibile caricare immagini Docker in un nodo:

```
kind load docker-image my-custom-image-0 my-custom-image-1
```

Per fare una build:

```
kind build node-image
```

Risultato dunque uno strumento utile per creare e configurare rapidamente cluster Kubernetes per essere testati in locale.

Helm

Helm [10] è un gestore open-source di pacchetti Kubernetes.

Calico

Calico [11][12] è un progetto open-source che fornisce networking e sicurezza per container Kubernetes. Offre funzionalità di networking di livello 3 (L3) e livello 4 (L4) OSI[1] ed utilizza Border Gateway Protocol (BGP) per costuire tabelle di routing e facilitare la connessione tra nodi. Nel nostro caso è utilizzato con Kubernetes e di fatto implementa il plugin Kubernetes Container Network Interface (CNI). Per assegnare un indirizzo IP ad ogni Pod e creare così una rete di livello 3 (L3), una rete CIDR più grande viene divisa in tanti piccoli di indirizzi IP che possono essere assegnati. I Pod in un nodo possono comunicare con i Pod in ciascun altro nodo senza l'utilizzo di NAT.

3.3 Liqo

Liqo[2] è un progetto open-source che consente di creare topologie multi-cluster di Kubernetes in modo dinamico e con approccio seamless. Supporta in questo modo la visione del concetto di liquid computing illustrato prima. Per supportare infrastrutture eterogenee, liqo non apporta modifiche alle API standard di Kubernetes [4][13].

[1] OSI: Open Systems Interconnection o Modello OSI/ISO, standard architetturale per definire i protocolli di rete.

[2] Codice sorgente disponibile all'indirizzo https://github.com/liqotech/liqot

Scoperta e Peering dei cluster remoti

La prima fase è quella dell'individuazione dei possibili cluster remoti con qui effettuare la connessione. L'output del processo di scoperta è una Custom Resource (CR) *ForeignCluster*, costituita da un endpoint per il cluster remoto e dello stato di peering desiderato, oltre ad eventuali parametri aggiuntivi. Con *peering*, in liqo, si intende una relazione di consumo delle risorse unidirezionale tra due cluster Kubernetes. In questa relazione, un cluster, definito come Consumer, ha la possibilità di offloadare le proprie risorse su un cluster remoto, il Provider, ma non viceversa. Il Consumer instaura così un *outgoing peering* (peering in uscita) verso il Provider e il Provider a sua volta riceve un *incoming peering* (peering in entrata) dal Consumer. Lo stesso cluster può svolgere sia il ruolo del Consumer che il ruolo del Provider, consentendo di ottenere peering bidirezionali grazie alle varie combinazioni, in una configurazione asimmetrica.
L'instaurazione della relazione di *peering* tra due cluster si compone di 4 fasi principali:

- **Autenticazione**: Una volta individuato il cluster remoto B come proprio obiettivo, il cluster A genera una chiave privata locale ed invia una *Certificate Signing Request* (CSR) e un pre-shared token al cluster B. Se il token è valido, il cluster B firma la CSR con il proprio certificato (CA), condendo al cluster A soltanto i permessi nella fase di peering.

- **Negoziazione delle risorse**: Il cluster Consumer A crea una nuova CR *ResourceRequest*, indicando la volontà di richiedere risorse computazionali o servizi del cluster B. Il *CRDReplicator*, responsabile dell'interazione tra cluster, replica la *ResourceRequest* sul cluster B. Una volta ricevuta la richiesta, B scopre automaticamente il cluster A e, se configurato per accettare la richiesta, esegue l'autenticazione simmetrica. Intanto, la *ResourceRequest* viene processata da una logica personalizzata e l'esito viene ritrasmesso attraverso un aggiornamento del suo Status, che può esser utilizzato per rifiutare la richiesta di peering. Una *ResourceOffer* viene creata in seguito all'accettazione da parte di B, specificando la quantità di risorse e servizi disposto a mettere in condivisione. Il *CRDReplicator* si occupa di replicare la richiesta sul cluster A. Una volta che l'offerta viene accettata da A, B concede ad esso i permessi necessari per utilizzare le proprie risorse, configurando i meccanismi di isolamento per la rete, la sicurezza e l'utilizzo delle risorse.

- **Setup del virtual node**: Il Consumer crea un nuovo *virtual node* per astrarre le risorse condivise da ciascun cluster remoto. L'implementazione avviene grazie a una modifica del Virtual Kubelet (VK) fornito da Kubernetes. Il VK si occupa di gestire il ciclo di vita del virtual node, dalla sua creazione, passando per l'allineamento dello stato del nodo e per il controllo della salute del cluster. Il VK ha il compito di mappare ogni operazione su un Pod *gemello* nel cluster remoto per permettere l'esecuzione. Questo componente ha dunque il compito di preparare l'ambiente per il processo di offloading.

- **Setup del tessuto di rete**: I due cluster configurano il tessuto di rete stabilendo un tunnel VPN in base ai parametri stabiliti nella fase precedente.

Le comunicazioni tra Pod del cluster locale e quelli del cluster remoto sono così abilitate.

Offloading

Il processo di offloading è reso possibile dal virtual node e permette di espandere il local cluster su un remote cluster, replicando le operazioni dei pod locali sui corrispondenti pod remoti. Ogni qualvolta un namespace viene selezionato per l'offloading, Liqo crea un namespace gemello nel cluster remoto. Esso ospita i pod offloadati e le risorse necessarie per la loro esecuzione. In seguito allo scheduling del pod, viene creato un pod gemello nel cluster remoto per consentire l'esecuzione. Sono supportati sia i pod stateless che i pod stateful. Per garantire la resilienza del sistema, vengono creati dei *ShadowPod*, una CR che mantiene lo stato desiderato dei pod anche in caso di interruzione del servizio. L'ultima fase è quella della riflessione delle risorse che consiste nel propagare e sincronizzare le informazioni del control plane necessarie per l'esecuzione dei workload offloadati. I vari servizi (rete, storage, configurazione, ...) appartenenti al namespace selezionato per l'offloading, vengono automaticamente propagati nel cluster remoto per permettere il loro riallineamento con la shadow copy.

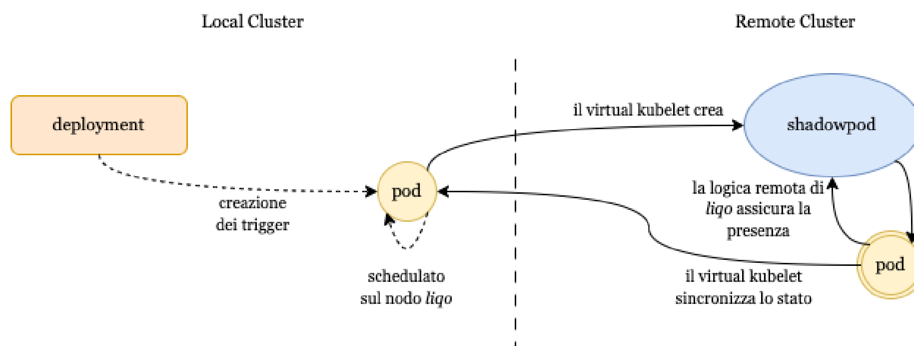

Figura 3.3. Offloading in Liqo

Tessuto di rete

Il tessuto di rete è un sottosistema di *liqo* che permette ai Pod offloadati di comunicare tra loro come se fossero eseguiti localmente, estendendo il modello di rete di Kubernetes in maniera trasparente. Tutti i Pod in un cluster possono comunicare con i Pod connessi del cluster remoto, con o senza traduzione NAT. Un meccanismo di traduzione degli indirizzi è necessario quando c'è una sovrapposizione tra indirizzi IP, dato che sono possibili diverse configurazioni (diverse CNI).

Tessuto di storage

Il tessuto di storage è il sottosistema responsabile dell'offloading dei workload stateful nel cluster remoto creando un continuum di storage. Ciò è reso possibile da

due approcci: data gravity e rinvio del binding dello storage. Il data gravity si basa su policy automatiche che permettono di selezionare il cluster adeguato per i vari pod, garantendo che i pod siano associati ai cluster dove risiedono fisicamente le risorse richieste. Il secondo approccio permette di attendere l'assegnazione di un utente a un determinato cluster prima di effettuare il binding, in modo che i pool di storage risiedano nella stessa posizione dei workload programmati.

Capitolo 4

FLUIDOS

4.1 Introduzione

FLUIDOS (Flexible, scaLable, secUre and decentralIzeD Operating System)[14][15] è un progetto europeo che nasce con l'obiettivo di creare un continuum di elaborazione seamless, sfruttando le risorse inutilizzate ai bordi della rete e distribuite su dispositivi eterogenei che faticano a integrarsi tra loro. Basandosi su sistemi operativi consolidati e su orchestratori diffusi come Kubernetes, punta a introdurre un nuovo livello di funzionalità.

4.2 Obiettivi

- Rendere fluidi i bordi ed unificarli con il cloud creando un continuum decentralizzato di risorse, sfruttando la scoperta e l'integrazione automatica delle risorse.
- Decentralizzare la gestione delle risorse, creando un software open-source consentendo l'interoperabilità e la condivisione tra utenti e Provider differenti.
- Utilizzo di algoritmi di AI e training per la mobilità e la previsione del traffico, grazie all'orchestrazione di un continuum di servizi e applicazioni su dispositivi e domini diversi.

- Implementazione del paradigma Zero-Trust[1] per garantire la sicurezza e l'accesso a risorse geograficamente distribuite.
- Favorire la nascita di un mercato di app, servizi e stakeholder diversi, svincolati da una precisa infrastruttura cloud per assicurare l'indipendenza economica dell'Europa.

4.3 FLUIDOS computing continuum

I paragrafi seguenti descrivono le tre caratteristiche distintive di FLUIDOS, che non possiamo trovare in soluzioni già esistenti.

Trasparenza del deployment

Tradizionalmente, il deployment di un'applicazione composta da più microservizi prevede che ogni componente venga assegnato a un target specifico. Una volta che è stata decisa la posizione di ciascun componente, non c'è possibilità di modifica se non ricominciando il processo da capo. Ne consegue che qualsiasi ottimizzazione dinamica non è applicabile in questo contesto, richiedendo il deployment di quei componenti per cui è stata trovata una nuova posizione ottimale. FLUIDOS invece, prevede un'interfaccia intent-based che garantisce che ogni microservizio sia assegnato alla posizione ideale all'inizio del processo e la possibilità di ottimizzazioni dinamiche in caso di necessità. Tutte le operazioni risultano semplificate, avendo gli sviluppatori accesso a un unico punto di controllo e di deployment, in quanto il lavoro di FLUIDOS avviene a un livello superiore rispetto ai servizi.

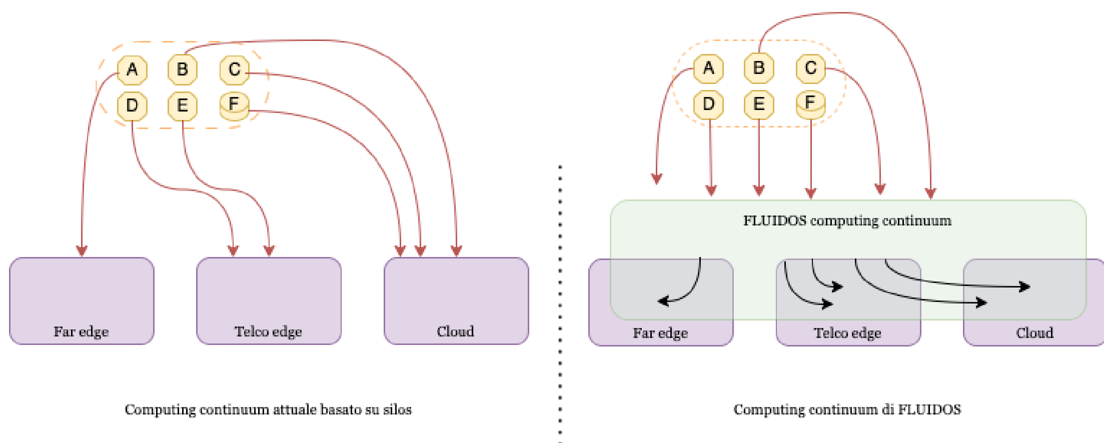

Figura 4.1. Deployment delle applicazioni con approccio tradizionale vs FLUIDOS

[1]Zero Trust è un modello di sicurezza utilizzato per proteggere un'organizzazione partendo dal presupposto che nessuna persona o nessun dispositivo debba essere considerato attendibile a prescindere, anche se si trova già all'interno della rete di un'organizzazione[16]

Trasparenza delle comunicazioni

La comunicazione tra microservizi può essere complessa da gestire. Partendo dal presupposto che le comunicazioni variano a seconda dello spazio virtuale in cui si trovano, sono normalmente consentite tutte le comunicazioni all'interno del cluster e negate tutte le comunicazioni verso l'esterno. Ad accrescere questa complessità c'è il fatto che le primitive richieste dal cluster per comunicazioni all'interno (Kubernetes ClusterIP service) sono diverse da quelle richieste per le comunicazioni verso l'esterno (Kubernetes NodePort o LoadBalancer services). In aggiunta a ciò deve esserci una configurazione manuale per permettere queste comunicazioni. Questa limitazione può in realtà essere aggirata con l'utilizzo dei message broker, soltanto nei casi in cui i microservizi utilizzino queste primitive. FLUIDOS invece permette tutte le comunicazioni tra microservizi, grazie allo spazio virtuale creato tra essi. Di conseguenza, i microservizi possono comunicare a prescindere dalla propria posizione, evitando le configurazioni manuali che possono portare a malfunzionamenti.

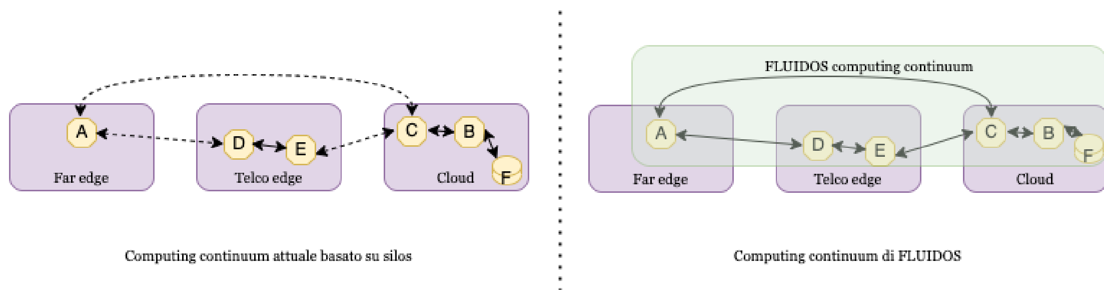

Figura 4.2. Comunicazioni service-to-service vs FLUIDOS

Trasparenza delle risorse disponibili

Un problema che si riscontra soprattutto nei casi in cui ci sono poche risorse disponibili ai bordi, con pochi server disponibili, è quello di non poter acquisire e sfruttare nuove risorse messe a disposizione nel continuum. Questo si verifica perché con l'architettura tradizionale i microservizi possono sfruttare solamente le risorse (RAM, CPU, ecc...) all'interno del proprio cluster. Ciò porta a problemi sia durante le normali operazioni che durante gli aggiornamenti, e quindi ad un'interruzione del servizio per mancanza di risorse disponibili, anche se sarebbero disponibili nelle vicinanze o in altre zone del continuum. Con la creazione di uno spazio virtuale di computazione che si estende su più domini fisici, FLUIDOS risolve queste limitazioni, rendendo disponibili le risorse a tutti i dispositivi presenti nello spazio virtuale, a prescindere dalla propria posizione fisica.

Il FLUIDOS Node [14][15][17] è un ambiente informatico unico appartenente ad un singolo dominio amministrativo e costituito da una o più macchine. Con l'obiettivo di nascondere i dettagli sottostanti, le macchine sono modellate con lo stesso set di primitive, consentendo comunque di esportare le funzionalità più importanti. Il FLUIDOS Node, non avendo particolari requisiti di omogeneità, corrisponde ad

un cluster Kubernetes, ed è per questo che viene orchestrato dal Kubernetes control plane. In realtà, può essere esso stesso considerato un orchestratore, capace di accettare richieste di carichi di lavoro, eseguire job specifici nel proprio dominio e presentare un set di policy omogeneo nell'interazione con altri nodi. Questo avviene anche grazie al fatto che al suo interno siano presenti servizi software e set di risorse che possono essere utilizzati localmente o condivisi con altri nodi. Può essere composto da un singolo set di dispositivi o da più set. Il Node presenta tutte le caratteristiche del Node Kubernetes descritte nel paragrafo [3.1.1]. Le comunicazioni interne sono gestite dal Kubernetes Controller, mentre le comunicazioni esterne tramite API RESTful. Il FLUIDOS Node è stato sviluppato in linguaggio Go, sfruttando l'ambiente di Kubernetes per gli oggetti e i pattern. Nella Figura [4.4] è possibile osservare i vari componenti presenti all'interno di FLUIDOS Node e che saranno descritti nei paragrafi seguenti.

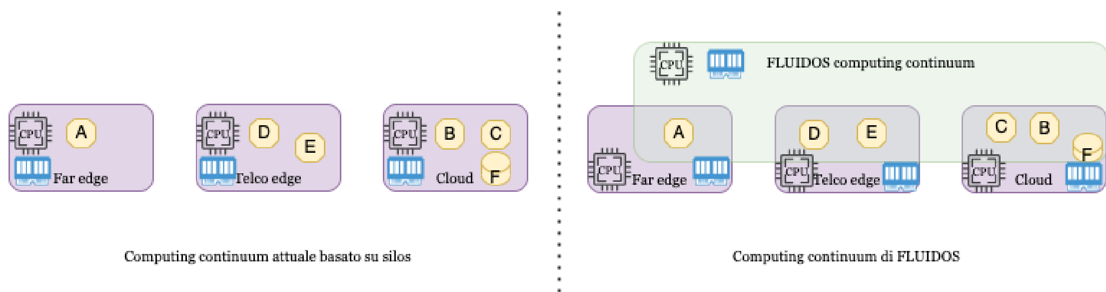

Figura 4.3. Risorse disponibili con l'approccio tradizionale vs FLUIDOS

4.4 FLUIDOS Node

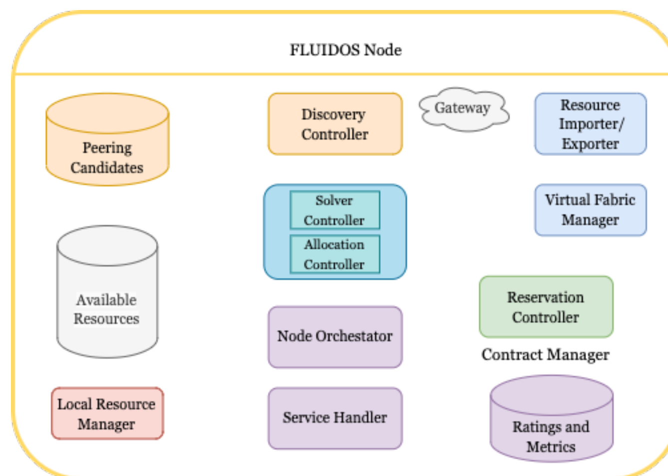

Figura 4.4. FLUIDOS Node

4.4.1 Componenti

Flavor

Il **Flavor** (Listing [4.1]), introdotto dal protocollo REAR [18][19], è il modello di dati utilizzato all'interno del FLUIDOS Node per la descrizione delle risorse. Un **Flavor** fornisce un modello strutturato per la rappresentazione e la gestione delle risorse computazionali. Risulta indipendente dal **FlavorType**, un puntatore ad una struttura che contiene le caratteristiche di ogni risorsa.

Listing 4.1. Flavor.yaml

```
{
    "flavorId": "flavor-123",
    "ProviderId": "Provider-456",
    "timestamp": "2024-07-15T12:00:00Z",
    "location": {
        "latitude": "37.7749",
        "longitude": "-122.4194",
        "country": "USA",
        "city": "San Francisco",
        "additionalNotes": "Near the waterfront"
    },
    "networkPropertyType": "5G",
    "type": {
        "name": "k8slice",
        "data": {
            "$ref": "flavor-types/k8slice.json"
        }
    },
    "price": {
        "amount": "100",
        "currency": "USD",
        "period": "monthly"
    },
    "owner": {
        "domain": "example.com",
        "nodeId": "node-789",
        "ip": "192.168.1.1",
        "additionalInformation": {
            "LiqoID": "liqo-001"
        }
    },
    "availability": true
}
```

Dei vari modelli di **FlavorType** (Listing [4.2]) disponibili, si menziona quello che verrà utilizzato all'interno del progetto: il *K8Slice*. Definisce un cluster Kubernetes e comprende sia risorse hardware che policy legate al suo uso e al suo deployment all'interno di Kubernetes.

Listing 4.2. FlavorType.yaml

```
{
    {
    "name": "ExampleK8Slice",
    "characteristics": {
        "architecture": "x86",
        "cpu": "2",
        "pods": "5",
        "memory": "4Gi",
        "gpu": {
            "model": "NVIDIA Tesla K80",
            "cores": "1",
            "memory": "12Gi"
        },
        "storage": "100Gi"
    },
    "properties": {
        "latency": 50,
        "securityStandards": [
            "GDPR",
            "HIPAA"
        ],
        "carbonFootprint": {
            "embodied": 2000,
            "operational": []
        },
        "networkAuthorizations": {
            "deniedCommunications": [
                {
                    "$ref": "intents/network-intent.json"
                }
            ],
            "mandatoryCommunications": [
                {
                    "$ref": "intents/network-intent.json"
                }
            ]
        }
    },
    "policies": {
        "partitionability": {
            "cpuMin": "500m",
            "memoryMin": "1Gi",
            "podsMin": "1",
            "cpuStep": "250m",
            "memoryStep": "512Mi",
            "podsStep": "1"
        }
    }
}
}
```

Local ResourceManager

Ha il compito di monitorare le risorse interne dei cluster, ossia i singoli nodi all'interno di un FLUIDOS Node. Per ogni nodo produce un Flavor Custom Resource (CR), che viene conservato all'interno del cluster per utilizzo e gestione.

Available Resources

Available Resources è quel componente che si occupa del salvataggio dei Flavor. È costituito da due strutture dati principali:

- **Free Flavor**: Destinata al salvataggio dei Flavor così come definito da REAR.
- **Reserved Flavor**: Destinata al salvataggio di oggetti e documenti relativi a varie risorse, ciascuno rappresentato come Flavor.

L'**Available Resources** svolge la funzione di repository centralizzata a cui è possibile richiedere una lista di risorse, rappresentando la gestione delle risorse in FLUIDOS grazie ad un efficiente utilizzo e gestione delle risorse computazionali. Si integra con Kubernetes etcd con approccio seamless per garantire efficienza nel salvataggio e nel recupero dei dati.

Discovery Manager

Il **Discovery Manager** svolge la funzione di monitor dei Discovery Custom Resource (CR) generati dal Solver Controller. In particolare è possibile descrivere i tre obiettivi principali:

- Popolare la tabella dei Peering Candidate (Consumer): È il compito principale del **Discovery Manager** ed avviene identificando i Flavor disponibili basati sul messaggio iniziale scambiato come previsto dal protocollo REAR.
- Scoperta (Consumer): Invia un messaggio LIST-FLAVORS in broadcast a tutti i FLUIDOS Node conosciuti.
- Offrire i Flavor idonei (Provider): Fornisce i Flavor compatibili con le specifiche richieste.

Il **Discovery Controller** è il componente interno al **Discovery Manager** che si occupa di monitorare e gestire lo stato dell'oggetto *Discovery* per raggiungere la configurazione desiderata. Funziona così:

1. Quando viene creato un nuovo oggetto *Discovery*, inizia il processo di scoperta collegandosi con il *Gateway* per trovare un Flavor compatibile con il selettore *Discovery*.

2. Se non viene trovato alcun Flavor, il processo si considera fallito. Altrimenti, il primo Flavor trovato viene individuato come *Peering Candidate* e gli altri vengono scartati.
3. Aggiorna l'oggetto *Discovery* con il *Peering Candidate* appena trovato.
4. Il *Discovery* è risolto e il processo può considerarsi concluso.

Peering Candidate

È una lista di nodi con cui sarà possibile stabilire una connessione di peering. È costantemente aggiornata dal **Discovery Manager**. Internamente ciascun Peering Candidate è gestito come una Custom Resource.

REAR Manager

Gestisce le richieste di risoluzione traducendole in richieste di servizi o risorse, ricercando risorse disponibili esterne. Fa partire il processo di *Discovery* se non è disponibile nessun **Peering Candidate**, se invece un candidato viene trovato può far partire la fase di **Reservation**. Se il processo di ricerca è stato completato con l'allocazione delle risorse e il salvataggio dei contratti, può far partire la fase di **Peering**.

Solver Controller

Il **Solver Controller** si occupa di mantenere allineato lo stato dell'oggetto *Solver* con la configurazione desiderata, monitorandolo costantemente.

- Ricerca i PeeringCandidate, se disponibili ne seleziona uno, altrimenti inizia il processo di ricerca creando una *Discovery*. Se lo stato *findCandidate* è risolto, vuol dire che è stato trovato un candidato, altimenti il Solver ha fallito.
- Nel caso ci sia anche la fase *ReserveAndBuy*, inizia il processo di *Reservation* che si conclude con la creazione di un *Reservation* object. Se il processo si è concluso con successo, avviene la prenotazione e l'acquisto delle risorse.
- Nel caso in cui sia prevista anche la fase *EnstablishPeering*, fa partire il processo di peering.

Allocation Controller

L'**Allocation Controller** si occupa di mantenere allineato lo stato dell'oggetto *Allocation* con la configurazione desiderata, monitorandolo costantemente.

Contract Manager

Gestisce la prenotazione e l'acquisto di risorse. Si occupa in particolare della gestione e della negoziazione dei contratti tra i nodi.

Reservation Controller

Il **Reservation Controller** si occupa di mantenere allineato lo stato dell'oggetto *Reservation* con la configurazione desiderata, monitorandolo costantemente. Nel momento in cui il *Reserve* flag del nuovo *Reservation* object è impostato, viene avviato il processo di *Reserve*. Con il *FlavorID* estratto dal *PeeringCandidate* del *Reservation* object, viene avviato il processo di reservation attraverso il gateway. Un oggetto *Transaction* viene creato in caso di processo concluso con successo. Il processo di *Purchase* è previsto nel caso in cui il flag *Purchase* sia impostato. L'oggetto *Transaction* viene utilizzato per far partire il processo e, nel caso in cui questo si concluda con esito positivo, viene aggiornato lo stato della *Reservation* e viene salvato il *Contract*.

Capitolo 5

Obiettivi della tesi

FLUIDOS, progetto allo stato attuale ancora in fase di sviluppo, si caratterizza per un approccio modulare, che ha permesso di suddividere il lavoro in ambiti specifici, ciascuno focalizzato su componenti architetturali e tematiche diversi. Questa tesi si concentra sul tema della sicurezza e dell'isolamento delle reti. Nei primi capitoli è stato affrontato il contesto generale ed introdotto l'ambiente di FLUIDOS. Restano da affrontare le problematiche che si riscontrano con il nuovo approccio, introducendo il tema della protezione dei bordi di rete, in un contesto che ora presenta un cluster virtuale, costituito da dispositivi eterogenei e distribuiti. Il lavoro di tesi si concentra in un primo momento sulla descrizione di questa problematica, descrivendo anche le tipologie di comunicazioni interessate. Vengono poi illustrate le policy ad alto livello necessarie per poter descrivere le proprie richieste, ossia gli intenti. Una volta introdotto il contesto ci si può concentrare sulla parte centrale della tesi, ossia lo sviluppo di un orchestratore di sicurezza. Andando con ordine è possibile spiegare meglio gli obiettivi della tesi:

- La realizzazione dei moduli *Verify* e *Harmonize* è avvenuta quasi di pari passo, condividendo molti algoritmi ed avendo a che fare con gli stessi dati in input. In particolar modo è stato necessario presentare gli intenti in una forma fruibile per i due moduli. Essi sono stati definiti con il linguaggio MSPL e forniti ai due moduli come file xml. È stato necessario inoltre definire i cluster del Provider ed del Consumer, per poter testare il corretto funzionamento in una prima fase di esecuzione in locale, prima dell'integrazione nel modulo definitivo. Dopo il cambiamento di alcune specifiche, è stato necessario adeguare i due moduli in modo da lavorare con una visione parziale, limitata solo ai pod e ai namespace dei loro cluster. Di conseguenza c'è stato un adeguamento per il funzionamento della *Verify* lato Consumer e della *Harmonize* lato Provider. Riguardo le strutture interne ritroviamo nella *Verify* alcuni algoritmi per la verifica delle sovrapposizioni già implementati nella *Harmonize*, ma in questo caso in forma semplificata, non essendoci la necessità di correggere ma soltanto verificarne la compatibilità. Nel Capitolo [6] verrà spiegato il funzionamento e il ruolo dei due moduli, mentre nel Capitolo [7] verrà illustrata la loro implementazione e le caratteristiche interne.
- Una volta realizzati e testati in locale i due moduli, è stato necessario integrarli con i moduli del FLUIDOS Node e con altri moduli realizzati nel

corso di una tesi parallela. Si è proceduto quindi ad installare l'ambiente e a modificare il Controller per poter collaborare sia con FLUIDOS Node che con i moduli realizzati nei due lavori di tesi. È stato necessario adeguarsi alle nuove specifiche di progetto andando a modificare gli input dei moduli in modo da poter lavorare con i Flavor contenti gli intenti e non più con i file xml precedentemente descritti. Il workflow del processo sarà presentato nel Capitolo [6], con il Capitolo [7] dedicato all'implementazione.

- L'ultima fase è quella stata quella dei test dell'intero ambiente di sviluppo e la validazione dei risultati con un caso di studio presentato appositamente. Il caso di studio presenta una richiesta di offloading da parte del Consumer e la definizione delle rispettive policy di rete. Grazie all'utilizzo di strumenti come Kubernetes e Liqo, è stato possibile simulare il comportamento dell'orchestratore in un ambiente in grado di simularne il funzionamento anche in un cluster riproducibile in locale. Il caso di studio preso in esame e i risultati della sua implementazione verranno presentati nel Capitolo [7].

Capitolo 6

Approccio per la protezione dei bordi di rete

6.1 Il problema

Il concetto del liquid computing, illustrato nel Capitolo [2], permette di creare un continuum di risorse, anche appartenenti a tenant e domini di amministrazione diversi, per evitare che le risorse siano sottoutilizzate. Questo approccio introduce però un problema di sicurezza in quanto non esiste più il concetto di singolo nodo fisico da proteggere, ma questo può essere condiviso da risorse multiple ed eterogenee, essendo in un contesto multi-cloud e multi-tenant. Il singolo server fisico viene sostituito da un cluster virtuale, cioè l'insieme logico delle risorse appartenenti alla stessa entità e potenzialmente distribuite su più cluster fisici. Da un bordo della rete ben definito si passa ad un bordo dinamico, capace di estendersi su più cluster fisici. Questo cambiamento richiede un nuovo approccio per la protezione dei bordi e delle risorse [20].

6.2 Tipi di comunicazione

Per capire meglio il problema della protezione dei bordi, è utile illustrare le varie casistiche e i vari tipi di comunicazione che possono verificarsi in un ambiente cloud e nello specifico all'interno di FLUIDOS. Gli esempi illustrati in Figura [6.1] raffigurano due cluster fisici diversi (i rettangoli in blu e in giallo) ed un virtual cluster che si estende sui due cluster fisici. I pod sono rappresentati come cerchi pieni. Le comunicazioni sono rappresentate da frecce verdi. Il colore blu contraddistingue le risorse (pod, cluster), di un'entità che possiamo immaginare come il Consumer. Il colore giallo contraddistingue le risorse del Provider. Sono stati illustrati soltanto i casi d'uso per cui è stato affrontato il problema nello sviluppo del progetto.
Lo scenario (1) descrive una comunicazione tra pod appartenenti al Consumer e situati all'interno dello stesso cluster fisico (intra-cluster) e dello stesso cluster virtuale (intra-virtual cluster), posseduti entrambi dal Consumer.
Lo scenario (2) illustra la comunicazione tra due pod appartenenti al Consumer, situati all'interno dello stesso cluster virtuale (intra-virtual cluster) posseduto dal

Consumer ma nel cluster fisico remoto (inter-cluster) del Provider (hosting cluster). Gli scenari (3) e (4) affrontano due casi simili. La comunicazione tra pod (o tra pod e internet) appartenenti a diversi virtual cluster (inter-virtual cluster) ma situati all'interno dello stesso cluster fisico remoto (intra-cluster). Coinvolgendo sia il Consumer che il Provider, questo scenario richiede una soluzione più complessa, in quanto potrebbero verificarsi conflitti. Questi due scenari nello specifico saranno l'oggetto del lavoro che verrà affrontato nei prossimi capitoli.

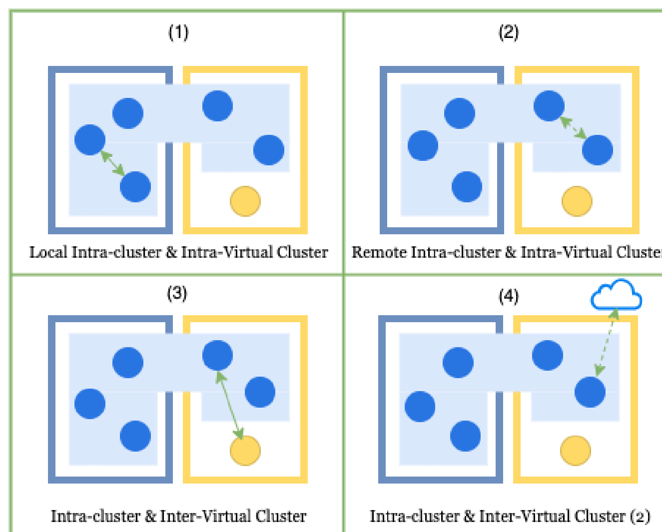

Figura 6.1. Tipi di comunicazione in FLUIDOS

6.3 Gli Intenti

Per rappresentare i vari scenari delle comunicazioni e le richieste dei vari attori, vengono utilizzati gli intenti. Gli intenti sono un'astrazione con cui è possibile stabilire un set di regole per descrivere le proprie richieste in termini di comunicazioni consentite o negate. Il processo prevede che sia il Consumer che il Provider debbano esprimere le proprie richieste sotto forma di intenti, di conseguenza ci sarà bisogno di tipologie diverse di intenti per descrivere adeguatamente le varie richieste. Nello specifico, l'esigenza del Consumer è quella di proteggere le risorse offloadate da eventuali furti di codice o dati o da interference da parte dell'host. Il Provider d'altra parte, vuole assicurarsi che le proprie risorse non siano danneggiate dalle applicazioni offloadate da un altro utente. Prima di illustrare le varie tipologie di intenti, va fatta una doverosa precisazione. Il Consumer e il Provider sono dotati di una visione parziale, ossia possono conoscere la composizione e la struttura soltanto del proprio cluster e delle proprie risorse. Nella definizione degli intenti, non è possibile dunque specificare la volontà di accedere ad uno specifico servizio, non potendo conoscere i pod o i namespace all'interno del cluster.
Sono stati definiti quattro tipi di intenti:

- **Private Intent**: I Private Intent (Figura [6.2]) riguardano le comunicazioni tra pod all'interno del virtual cluster (intra-virtual cluster), che possono

coinvolgere risorse locali o remote. Per queste comunicazioni non c'è bisogno dell'autorizzazione dell'host.

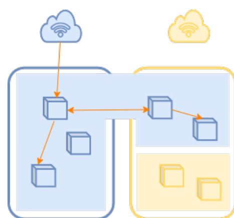

Figura 6.2. Private Intent

- **Request Intent**: I Request Intent (Figura [6.3]) descrivono le comunicazioni inter-virtual cluster, superando i bordi virtuali del cluster esteso. Possono riguardare comunicazioni verso l'esterno (internet), specificando un indirizzo CIDR o un URL, oppure verso i servizi dell'hosting cluster.

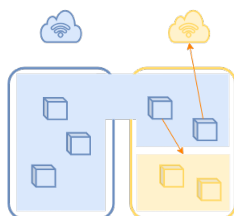

Figura 6.3. Request Intent

- **Authorization Intent**: Gli Authorization Intent (Figura [6.4]) sono definiti esclusivamente dal Provider per autorizzare comunicazioni inter-virtual cluster. Riguardano quindi le comunicazioni tra risorse offloadate dal Consumer e quelle del Provider stesso. Esistono due tipologie di Authorization Intent:
  - *ForbiddenCommunications*: Il Provider esprime le comunicazioni che vuole bloccare, siano queste richieste di accesso ai servizi o di comunicazione verso l'esterno (internet).
  - *MandatoryCommunications*: Il Provider definisce quelle comunicazioni che vuole imporre al Consumer. Un esempio può essere il *monitoring* delle risorse offloadate per raccogliere dati di log.
- **Setup Intents**: Comunicazioni richieste da FLUIDOS ed utilizzate per scopi di configurazione. Non riguarderanno questo lavoro nello specifico.

Per quanto riguarda il formato, la struttura è simile a quella delle Kubernetes Network Policy:

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

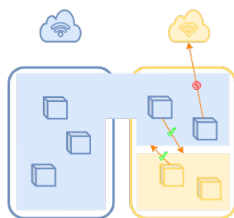

Figura 6.4. Authorization Intent

- **SRC** e **DST** possono essere definiti come un pod o un gruppo di pod con stessa etichetta e stesso namespace, oppure come un indirizzo CIDR. La wildcard "*" può essere utilizzata per indicare l'intero cluster. Ricordando la questione parziale, va fatta una dovuta precisazione in questo senso. Prendiamo il caso in cui la sorgente sia il Consumer e la destinazione sia il Provider (l'esempio vale anche per il caso contrario):
  - La sorgente può essere definita liberamente, avendo il Consumer piena conoscenza delle proprie risorse.
  - Per definire la destinazione invece, nel caso in cui si voglia accedere ai servizi o alle risorse del Provider e non effettuare una connessione verso l'esterno (internet), è obbligatorio utilizzare la wildcard, che comprende i servizi/risorse del Provider nel loro complesso, non potendo conoscere i dettagli su pod e namespace interni.
- protocol può essere qualsiasi protocollo di trasporto. Le opzioni disponibili sono "TCP", "UDP", "SCTP" e "ALL".
- **port** indica una porta specifica o un range di porte nel caso di **endPort**, attraverso il quale effettuare la comunicazione.

I Request, i Setup e i Private Intent possono essere espressi soltanto in whitelisting (indicando le connessioni desiderate), in linea con il comportamento delle Kubernetes Network Policy, mentre gli Authorization intent possono essere espressi sia in whitelisting che in blacklisting (indicando le connessioni bloccate).

Per esprimere gli intenti in un primo momento era stato scelto linguaggio MSPL (Medium-level Security Policy Language), ampiamente usato e validato in vari contesti, in un secondo mento si è scelto invece di integrarli all'interno dei Flavor. Risulta comunque utile illustrare questa implementazione per comprendere la struttura degli intenti.

Listing 6.1. Esempio File MSPL

```
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ITResourceOrchestration >
    <!-- Authorization Intents -->
    <ITResource>
        <configuration xsi:type="AuthorizationIntents">
            <capability>
                <Name>KubernetesNetworkFiltering</Name>
            </capability>
            <forbiddenConnectionList>
                <configurationRuleAction xsi:type="KubernetesNetworkFilteringAction">
                    <KubernetesNetworkFilteringActionType>DENY</KubernetesNetworkFilteringActionType>
                </configurationRuleAction>
```

```
                <configurationCondition xsi:type="KubernetesNetworkFilteringCondition">
                    <isCNF>false</isCNF>
                    <source xsi:type="PodNamespaceSelector">
                        <isHostCluster>true</isHostCluster>
                        <pod>
                            <key>*</key>
                            <value>*</value>
                        </pod>
                        <namespace>
                            <key>*</key>
                            <value>*</value>
                        </namespace>
                    </source>
                    <destination xsi:type="PodNamespaceSelector">
                        <isHostCluster>false</isHostCluster>
                        <pod>
                            <key>*</key>
                            <value>*</value>
                        </pod>
                        <namespace>
                            <key>name</key>
                            <value>handle-payments</value>
                        </namespace>
                    </destination>
                    <destinationPort>*</destinationPort>
                    <protocolType>SCTP</protocolType>
                </configurationCondition>
                <externalData xsi:type="Priority">
                    <value>60000</value>
                </externalData>
                <Name>AuthorizationDeny_2</Name>
                <isCNF>false</isCNF>
            </forbiddenConnectionList>
        </configuration>
    </ITResource>
</ITResourceOrchestration>
```


6.4 L'Integrazione

In questo capitolo verrà affrontata l'integrazione dei moduli sviluppati nel lavoro di tesi con i moduli sviluppati da un altro tesista, oltre che con FLUIDOS Node. Nello specifico l'integrazione della **Verify** e della **Harmonize**, oggetti del lavoro di tesi, con il **Controller** e con il **Translator**, oggetto di lavoro di un altro tesista, e con i vari moduli del FLUIDOS Node necessari per la prenotazione e l'acquisto delle risorse. Verranno quindi illustrati il workflow e il comportamento dei moduli in sintesi, con i dettagli tecnici lasciati per il prossimo capitolo.

6.5 Il Workflow

Consumer: Il cluster che si comporterà come Consumer delle risorse di FLUIDOS. Tramite degli intenti definisce l'offloading di alcune risorse nel cluster Provider e le connessioni desiderate.
Provider: Il cluster che si comporterà come Provider delle risorse di FLUIDOS. Tramite dei Flavor,metterà a disposizione alcune risorse.
Un **Controller** (non visibile nella Figura [6.5]) osserva le varie operazioni ed è il responsabile delle chiamate ai vari moduli per l'applicazione delle network policy, avendo visuale e controllo completo del cluster in cui risiede. È presente di conseguenza sia nel cluster Consumer che nel cluster Provider.
Il processo inizia con la richiesta da parte del Consumer di accedere a determinate risorse, descritta tramite intenti. Il **Node (Meta) Orchestrator** è responsabile dell'allocazione dinamica delle risorse, utilizzando algoritmi come ILP, Gredy, ecc. per trovare la soluzione ottimale. Nel caso non ci siano risorse disponibili, viene

contattato il **REAR Manager**. Una volta trovate le nuove risorse, esegue il peering e l'offloading nel cluster remoto.
Il protocollo **REAR** è utilizzato per l'annuncio, la prenotazione e il consumo delle risorse o dei servizi. In questo caso permette al Provider di condividere i propri Flavor con la descrizione delle risorse che si vogliono mettere a disposizione in modo che possano essere ricevuti dal Consumer.
Il **Solver** ha il compito di ricercare i PeeringCandidate, scegliendo in base a criteri come prezzi, risorse (CPU, RAM, ecc.), policy e in seguito alla superamento del processo di validazione per valutarne la compatibilità con i Flavor del Provider. Nel nostro caso è stato considerato un solo Provider che mette a disposizione 4 Flavor.
Una volta ricevuti i Flavor, il Solver può contattare il **Controller** che avvia il modulo **Verify** per verificare appunto la compatibilità tra intenti del Consumer e intenti del Provider contenuti nei Flavor. Dopo aver individuato il Flavor compatibile (il primo che ritorna true) e quindi il PeeringCandidate, è possibile proseguire con le fasi di applicazione della *Reservation* e della *Allocation* delle risorse. Viene dunque generato un *Contract* tra Consumer e Provider, contenente anche gli intenti definiti dal Consumer per le risorse da offloadare.
Tramite **Liqo** avviene l'offloading delle risorse del Consumer nel cluster remoto, contestualmente alla chiamata della **Harmonize** da parte del **Controller** lato Provider, per eseguire l'armonizzazione degli intenti.
L'ultima fase del processo è la chiamata del modulo **Translator** per l'enforcement degli intenti armonizzati contenuti nel contratto.

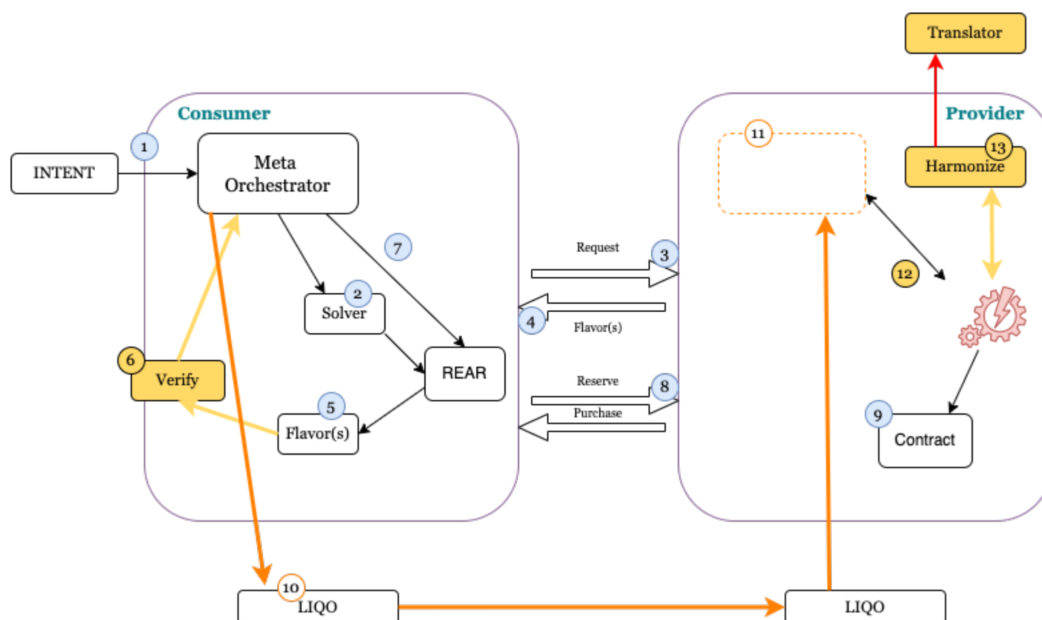

Figura 6.5. Workflow-Integrazione

6.6 Il Controller

Il **Controller** monitora attivamente i namespace, i pod e i Peering Candidate all'interno del cluster Kubernetes. In base a eventi come la creazione di nuovi namespace o l'aggiunta di Peering Candidate, il codice crea e applica diverse tipologie di Network Policy per controllare il traffico di rete tra i pod. Gestisce inoltre le risorse come gli Authorization Intent o i Request Intent per poter essere forniti con il formato corretto alle varie funzioni. Il **Controller** interagisce con altri moduli del sistema, come la Verifye la Harmonize, per verificare la conformità delle regole di autorizzazione e prendere decisioni sulla configurazione delle Network Policy.

6.7 Verifica della compatibilità

Il modulo **Verify** ha il compito di verificare la compatibilità tra gli intenti definiti dal Consumer e gli intenti definiti dal Provider, prima di procedere con la fase successiva. La **Verify** viene eseguita lato Consumer e restituisce true o false, a seconda della compatibilità o meno con le autorizzazioni concesse dal Provider. Il confronto viene effettuato tra i Request Intent del Consumer e gli Authorization Intent del Provider.

6.8 Armonizzazione degli Intenti

Il modulo **Harmonize** è composto da un algoritmo che rileva eventuali discordanze tra i Request Intent del Consumer e gli Authorization Intent del Provider. La **Harmonize** viene eseguita lato Provider ed emette un nuovo set di intenti armonizzati. È necessario segnalare che la versione realizzata era stata pensata inizialmente per una visione globale, quindi con piena conoscenza delle risorse dell'altro cluster. Il Provider ha pieno potere decisionale su quali Request Intent autorizzare, ma può intervenire soltanto sulle comunicazioni inter-cluster. Molti passaggi risultano essere quindi superflui per una visione parziale, ma per completezza viene illustrata comunque questa versione. La **Harmonize** può rilevare delle discordanze se gli intenti definiti dal Consumer non sono autorizzati o coerenti con gli intenti definiti dal Provider. Sono stati definiti tre tipi di discordanze:

- **Discordanze di Tipo 1**: Quando i Request Intent del Consumer non sono autorizzati (parzialmente o totalmente) dagli Authorization Intent del Provider.
- **Discordanze di Tipo 2**: Quando le MandatoryConnection definite dal Provider non sono presenti nei Request Intent definiti dal Consumer.
- **Discordanze di Tipo 3**: Quando i Request Intent del Consumer sono già stati autorizzati ma non hanno un intento speculare nei Request Intent del Provider.

6.9 Traduzione degli Intenti armonizzati

Il modulo *Translator* riceve in input gli intenti armonizzati dalla **Harmonize**, scritti in un linguaggio ad alto livello e ignorando l'attuale implementazione, ed effettua la traduzione nella configurazione a basso livello delle Kubernetes Network Policies. Il modulo può variare in base alla CNI utilizzata. Risulta fondamentale per il supporto ad ambienti multi-cluster, permettendo l'interoperabilità tra di loro. Dopo la traduzione viene effettuato l'enforcement delle Kubernetes Network Policies comunicando con il server API del cluster che ospita le risorse.

Capitolo 7

Implementazione e validazione

Il seguente capitolo prevede la descrizione dell'implementazione dei moduli oggetto del lavoro di tesi ed in seguito la validazione tramite l'applicazione di un caso di studio. I moduli oggetto del lavoro di tesi sono in questo caso la Verify e la Harmonize. La Validazione prevede prima l'illustrazione del caso di studio con i risultati dei processi di validazione e armonizzazione ed in secondo luogo la reale implementazione nell'ambiente di test secondo il workflow illustrato nel Capitolo [6]. I vari moduli qui illustrati sono stati sviluppati in linguaggio Java.

7.1 Verify

La funzione viene chiamata dal **Controller** ogni volta che viene ricevuto un nuovo Flavor contenente gli Authorization Intent del Provider per verificarne la compatibilità. Il modulo contenente la Verify e la Harmonize è stato strutturato seguendo in parte il pattern MVC (Model-View-Controller), con un Model che in questo caso non interagisce con un database. Come è possibile osservare dalla Figura [7.1], l'**HarmonizationController** ha il ruolo del Controller, l'**HarmonizationService** ha il ruolo della View e l'**HarmonizationData** ha, almeno in parte, il ruolo del Model. Il modulo si presenta lo stesso sia nel caso della **Verify** che nel caso della **Harmonize**, con la differenza che nel primo caso viene chiamato dal Consumer e nel secondo caso dal Provider.
La Verify viene chiamata dal Controller per ogni Flavor dello stesso Provider presente nella lista, fornendo come input gli **Authorization Intent** estratti dal Flavor e l'elenco dei pod e namespace del Consumer sotto forma di **Cluster**.

HarmonizationController

L'**HarmonizationController** richiama dunque l'**HarmonizationService** fornendogli gli stessi dati come input.

HarmonizationService

L'**HarmonizationService** si occupa di preparare e raccogliere tutte le risorse da fornire alla funzione *verify()* dell'**HarmonizationData** per poi eseguire il processo di verifica della compatibilità. I **Request Intent** vengono recuperati, almeno in questa prima fase, da un file xml intento al Consumer. Il **Cluster** viene organizzato come una HashMap multi-livello che mostra l'organizzazione dei pod nei vari namespace. La *verify()* viene chiamata una volta che è stato appurato che il Consumer abbia accettato il monitoring delle proprie risorse nei **Request Intent**.

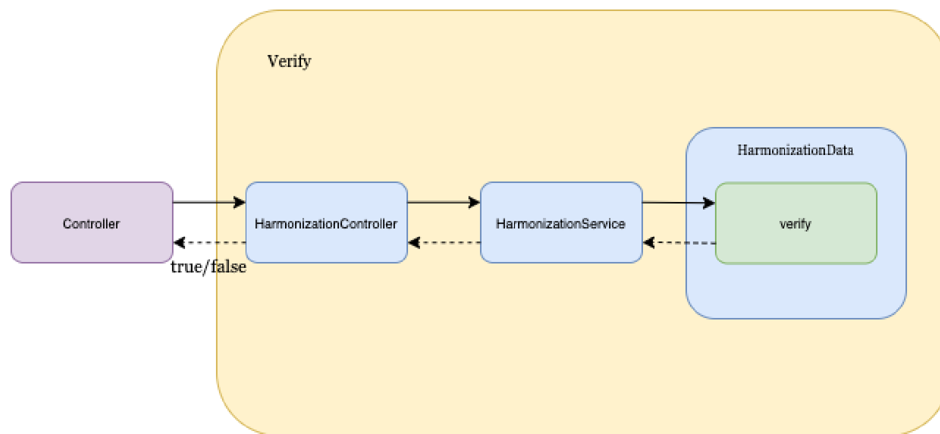

Figura 7.1. Verify

verify

La *verify* cicla sulle *ConfigurationRule* dei **Request Intent** chiamando ogni volta la *verifyConfigurationRule* per la verifica della compatibilità tra la **ConfigurationRule** corrente e tutte le *forbiddenConnection* degli **Authorization Intent**. Il ciclo si interrompe anticipatamente nel caso che la *verifyConfigurationRule* restituisca *false*, in quanto non può più esserci compatibilità.

verifyConfigurationRule

La compatibilità tra **Request Intent** e **Authorization Intent** si ha quando non viene riscontrata nessuna sovrapposizione tra le *ConfigurationRule* dei **Request Intent** e degli **Authorization Intent**. La sovrapposizione si ha quando tutti i campi delle *ConfigurationRule* hanno una sovrapposizione. I campi in questione sono *ProtocolType*, *Port*, *Src* e *Dst*. Per la verifica dei *ProtocolType* viene chiamata la funzione *computeVerifyProtocolType*. Per la verifica delle *Port* viene chiamata la funzione *computeVerifyPortRange*. Per la funzione di *Src* e *Dst* viene chiamata la *computeVerifyResourceSelector*.

computeVerifyProtocolType

La sovrapposizione tra protocollo di rete viene verificata controllando se i valori (TCP, UDP, SCTP) sono uguali oppure se è presente il valore ”ALL” che li racchiude tutti. In caso di sovrapposizione restituisce *true*, altrimenti *false*.

computeVerifyPortRange

Questa funzione controlla se i due intervalli di porte si sovrappongono. I due intervalli potrebbero in realtà essere una singola porta oppure nel caso sia presente la wildcard ”*” tutte le porte. La funzione restituisce *true* se c'è sovrapposizione, ossia quando:

$$x_0 \leq y_1 \quad e \quad x_1 \geq y_0$$

In caso contrario restituisce false.

computeVerifyResourceSelector

La *computeVerifyResourceSelector()* effettua un confronto tra **ResourceSelector**. Sono supportati due tipi di **ResourceSelector**:

- **CIDRSelector**: sovrapposizione tra due selettori CIDR.
- **PodNamespaceSelector**: sovrapposizione nei pod e nei namespace.

Viene prima effettuato un controllo sulla tipologia di **ResourceSelector** e poi in base al selettore si fa la verifica vera e propria della sovrapposizione:

- Entrambi **CIDRSelector**: viene calcolata la differenza tra i due range di indirizzi per verificare se c'è sovrapposizione.
- Entrambi **PodNamespaceSelector**: se pod e namespace del primo **ResourceSelector** risultano essere uguali a quelli del secondo selettore, c'è sovrapposizione.
- Selettori differenti: non può esserci sovrapposizione.

La funzione restituisce *true* se c'è sovrapposizione, *false* in caso contrario.

Verify

Se almeno una delle tre funzioni appena illustrate restituisce *false* non può esserci sovrapposizione. Riepilogando la **Verify** restituisce *true* alla fine del processo se non si verifica alcuna sovrapposizione. Alla prima sovrapposizione trovata, il processo si interrompe e viene restituito *false*. Il Controllore esamina i vari Flavor e la ricerca del *peeringCandidate* si interrompe con successo al primo Flavor che risulta compatibile con i **Request Intent** e quindi dopo la prima volta che la **Verify restituisce true**.

7.2 Harmonize

La Harmonize (Figura [7.2]) viene chiamata dal **Controller** per verificare nuovamente la compatibilità tra gli intenti, effettuando anche il controllo sulle richieste del Provider specificate nelle *mandatoryConnection*. La versione qui analizzata è stata realizzata in linea con una visione globale, quindi con la conoscenza di tutte le informazioni e le risorse da parte di Consumer e Provider. Il passaggio ad una visione parziale prevede alcune piccole modifiche nel codice ma sostanzialmente la versione utilizzata è uguale a quella precedente. La differenza sta nel fatto che alcuni controlli effettuati dall'algoritmo risultano superflui in questo caso e quindi la **Harmonize** si comporta quasi come la **Verify**. In questo caso il **Controller** viene eseguito lato Provider e quindi la conoscenza delle risorse è limitata a quest'ultimo.

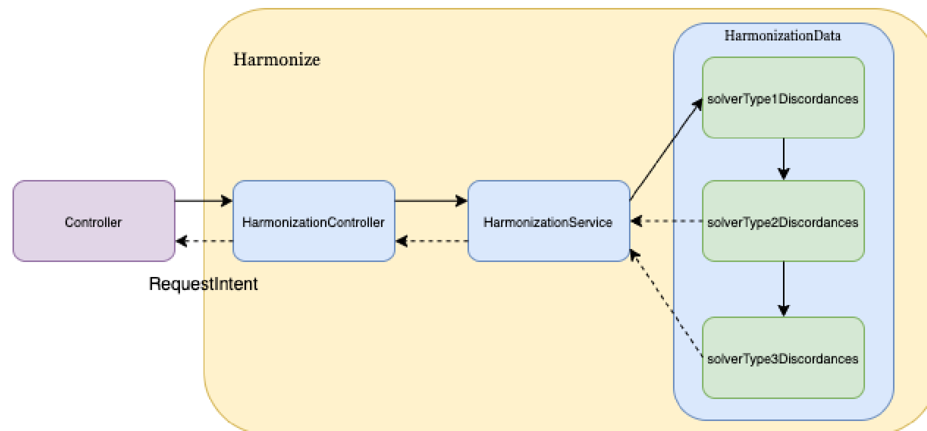

Figura 7.2. Harmonize

HarmonizationController

La **Harmonize** viene chiamata mediante l'**HarmonizationController** che riceve in input il **Cluster** Provider, i **Request Intent** e gli **Authorization Intent** definiti nel contratto. I parametri vengono trasmessi anche alla *harmonize* dell'**HarmonizationService**.

HarmonizationService

All'interno della funzione viene organizzato il **Cluster** come HashMap multi-livello, vengono recuperati i **Request Intent** del Provider da file xml e viene effettuata la chiamata alle tre funzioni per la risoluzione delle discordanze ricevendo i **Request Intent** del Consumer armonizzati dopo le prime due chiamate e i **Request Intent** del Provider dopo la chiamata alla terza funzione. Questi vengono poi restituiti al chiamante.

solverType1Discordances

Le discordanze di Tipo-1 si hanno quando i **Request Intent** del Consumer non sono pienamente autorizzati dal Provider. In sostanza per ogni richiesta del Consumer viene effettuata questa operazione:
Consumer.Request - Provider.AuthorizationIntent.forbiddenConnection.
La funzione esegue un ciclo esterno sui vari **Request Intent** del Consumer, invocando per ognuno di essi la harmonizeConfigurationRule che riceve come input la singola richiesta del Consumer e la lista delle connessioni negate dal Provider. Viene restituita al chiamante la lista degli intenti armonizzati.

solverType2Discordances

Le discordanze di Tipo-2 si verificano quando le *mandatoryConnection* del Provider non sono totalmente soddisfatte dal Consumer. In questo caso le connessioni vanno aggiunte alla lista di intenti armonizzati. intentiArmonizzati = intentiArmonizzati + (mandatoryConnectionList - intentiArmonizzati).
Il ciclo for viene eseguito sulla lista delle *mandatoryConnection* e ad ogni iterazione viene chiamata la harmonizeConfigurationRule che riceve in input la singola *mandatoryConnection* e la lista degli intenti armonizzati sino ad ora.

solverType3Discordances

Le discordanze di Tipo-3 si verificano quando gli intenti già armonizzati dal Consumer non hanno una regola speculare nell'hosting cluster. Se ciò si verifica un nuovo intento armonizzato viene aggiunto alla lista degli intenti armonizzati del Provider. Viene eseguito un ciclo sugli intenti armonizzati del Consumer per verificare se la connessione è già aperta lato Provider. La harmonizeConfigurationRule riceve in input ad ogni chiamata la singola regola del Consumer e la lista di **Request Intent** del Provider.

harmonizeConfigurationRule

È una funzione utilizzata per armonizzare la regola corrente con una lista di altre regole. Rappresenta il ciclo interno delle funzioni di risoluzione delle discordanze. Il confronto avviene tra la regola corrente e una regola alla volta della lista di regole. Chiameremo la regola corrente *conn*, la lista di regole *connList* e la singola regola all'interno della lista *confRule*. Le varie casistiche esaminate internamente al ciclo:

- Step 1.1: si verifica se c'è sovrapposizione tra i *ProtocolType*. Viene chiamata la computeHarmonizedProtocolType e se il valore ritornato corrisponde a quello della *conn*, si passa alla prossima iterazione in quanto non c'è sovrapposizione, in caso contrario si prosegue con il prossimo step.
- Step 1.2: si considerano in questo caso i due range di porte (*DestinationPort*). La funzione utilizzata per rilevare la sovrapposizione è la computeHarmonizedPortRange che in caso di valore ritornato uguale a quello della

DestinationPort della *conn* si prosegue con l'iterazione seguente non essendoci sovrapposizione, altrimenti si passa al prossimo step.

- Step 1.3: vengono valutate sorgente e destinazione. La funzione di supporto utilizzata è la computeHarmonizedResourceSelector. Il risultato ritornato viene confrontato con la *conn* tramite la compareResourceSelector e se la funzione restituisce true non c'è sovrapposizione tra le regole attuali e si passa alla prossima iterazione. Il processo viene ripetuto prima per la sorgente e poi per la destinazione se l'esecuzione non s'è giù interrotta.

- Step 2: Arrivati a questo punto vuol dire che c'è una sovrapposizione tra *ProtocolType*, *DestinationPort*, *Source* e *Destination*.

- Step 2.1: se c'è una sovrapposizione parziale tra gli intervalli di porte, si crea una nuova regola per l'intervallo di porte, che può anche essere diviso in due intervalli, grazie alla addHarmonizedRules. Viene poi chiamata ricorsivamente la harmonizeConfigurationRule

- Step 2.2: lo stesso processo dello step precedente viene ripetuto in questo caso con la sovrapposizione tra protocolli di rete.

- Step 2.3: se c'è una sovrapposizione tra i selettori di sorgente o destinazione, viene creata una nuova regola modificata con la addHarmonizedRules e chiamata ricorsivamente la harmonizeConfigurationRule.

- Step 3: può esserci una sovrapposizione completa o parziale con le regole aggiunte ricorsivamente, ma in ogni caso la regola deve essere scartata.

Alla fine del ciclo viene ritornata la lista con gli intenti armonizzati.

computeHarmonizedProtocolType

Funzione che calcola la differenza tra due set di protocolli di rete. Dati i due set (o singoli) protocolli, ritorna la differenza tra il primo e il secondo. I protocolli previsti sono: "TCP", "UDP", "SCTP" e "ALL".
Possono verificarsi vare casistiche:

- Due valori uguali: viene ritornata una stringa vuota.

- Il secondo valore è "ALL": viene ritornata una stringa vuota.

- Il primo valore è "ALL": ritorna la differenza escludendo il secondo valore dal set.

- Nessuno dei casi precedenti: sono due set disgiunti, viene ritornato il primo valore.

computeHarmonizedPortRange

Calcola la differenza tra due intervalli di porte (**portRangeX** e **portRangeY**) restituiendo la differenza tra il primo e il secondo set. Vengono gestite le varie casistiche, come i casi in cui uno dei due valori sia un intervallo e l'altro una singola porta oppure entrambi siano intervalli o singole porte. Se le porte coincidono o se la porta/intervallo di porte è incluso nel secondo, viene ritornata una stringa vuota, altrimenti **portRangeX**. Se si tratta di due intervalli di porte differenti, viene calcolata la differenza. La funzione restituisce dunque il nuovo intervallo di porte calcolato (o una stringa vuota) oppure **portRangeX** nel caso non ci sia sovrapposizione.

computeHarmonizedResourceSelector

Questa funzione calcola la differenza tra due **ResourceSelector** ritornando un selettore armonizzato contente le risorse comprese in *sel-1* e non in *sel-2*. I selettori possono essere di due tipi: **CIDRSelector** e **PodNamespaceSelector**. Se sono diversi non viene calcolata la differenza e la funzione ritorna. Possiamo esaminare le varie casistiche:

- Selettori diversi: non viene calcolata la differenza e la funzione ritorna al chiamante.
- **CIDRSelector** e **CIDRSelector**: Viene calcolata e ritornata la differenza tra i due range di indirizzi.
- **PodNamespaceSelector** e **PodNamespaceSelector**:
  - Verifica se i selettori appartengono allo stesso cluster, ritornando al chiamante nel caso si riferiscano a due cluster diversi.
  - Viene effettuata la differenza tra i due **PodNamespaceSelector**.
    Chiameremo i due selettori *pns1* e *pns2*.
    Se *pns2* comprende tutti i pod e i namespace, la differenza è 0 e viene ritornata una lista vuota. Per verificare se c'è sovrapposizione o nulla, vengono selezionati tutti i pod grazie alla funzione *selectPods()* e vengono rimossi dalla prima lista i pod della seconda lista, il risultato viene poi ritornato.

selectPods

La funzione riceve come input il *pns* e la hashmap del cluster selezionato.

- Caso 1: Vengono selezionati tutti i pod del cluster se namespace e pod vengono definiti come "*".
- Caso 2: Vengono selezionati tutti i pod di un certo namespace se il namespace è indicato e il pod è "*".

- Caso 3: Vengono selezionati tutti i pod namespace con una specifica etichetta sui pod se il namespace è "*" e il pod è specifico.
- Caso 4: Vengono selezionati specifici pod in un namespace se entrambi i parametri sono indicati.

compareResourceSelector

La funzione calcola la differenza stretta tra due **ResourceSelector** e ritorna *true* se sono strettamente uguali e *false* altrimenti.

addHarmonizedRules

Funzione che genera una nuova regola in base al tipo di sovrapposizione rilevata sostituendo il vecchio valore con il nuovo valore fornito

7.3 Validazione

Il caso di studio usato per la validazione presenta due entità diverse, il cluster e il Consumer che vogliono interagire tra di loro nel processo di peering e offloading. Il Consumer vuole effettuare l'offloading di alcune risorse nel cluster del Provider che mette a disposizione il proprio cluster. Le comunicazioni richieste dal Consumer tramite i Request Intent, dovranno essere validate dalla Verify, verificando la compatibilità con le scelte effettuate dal Provider. Nella figura [7.3] viene illustrata la situazione dei due cluster prima del peering e dell'offloading. I rettangoli esterni rappresentano i due cluster, mentre i rettangoli interni raggruppano i pod (raffigurati come cubi) contraddistinti dallo stesso namespace (in nero, accompagnato da un simbolo). Il colore giallo indica gli elementi del Consumer, il colore arancione indica gli elementi del Provider.

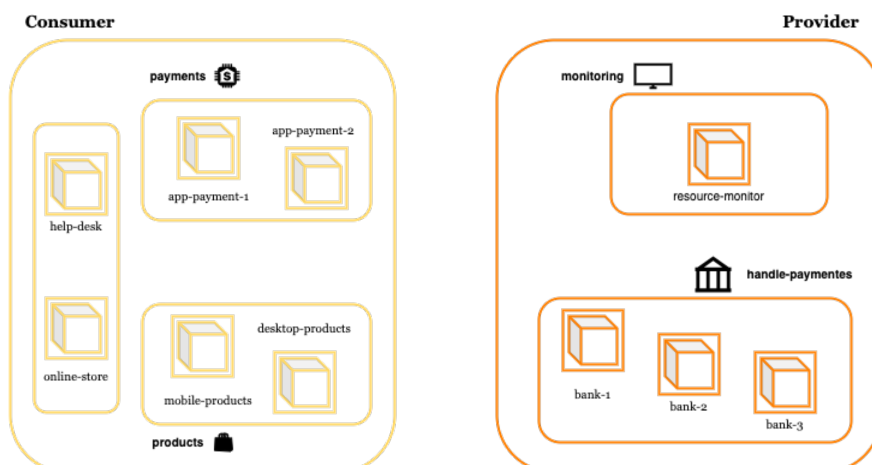

Figura 7.3. I due cluster prima del processo di offloading

Le figure successive descrivono tre diversi Flavor corrispondenti a tre diversi set di Authorization Intent definiti dal Provider. Per quanto riguarda il Consumer i RequestIntent restano in tutti e tre i casi gli stessi. Il processo di verifica prevede quindi il confronto tra le richieste del Consumer e le decisioni prese dal Provider in merito alle proprie risorse, e si interrompe al primo flavor che restituisce il risultato true, ossia la compatibilità tra i due set di intenti. Il Consumer vuol richiede la connessione da pod specifici verso tutte le risorse del Provider o verso l'esterno, indicando eventualmente una porta (o un set di porte) e/o un protocollo di trasporto. Il Provider viceversa, può impedire l'accesso a tutte le risorse per le connessioni che prevedono determinate porte (o set di porte) e protocolli. Definisce inoltre, la connessione da un suo pod, verso il Consumer, per una funzione di monitoring.

- Il rettangolo viola rappresenta il cluster fisico del Provider.
- Il rettangolo giallo più esterno rappresenta il virtual cluster del Consumer. Per motivi di spazio non viene mostrata l'altra parte di virtual cluster che si appoggia sul cluster fisico del Consumer.
- I rettangolo interni rappresentano i namespace e i cubi al loro interno i pod.
- Le frecce nere rappresentano le connessioni che il Consumer vuole effettuare. La freccia parte da un pod specifico ed arriva sul rettangolo arancione esterno che include tutte le risorse del Provider, non potendo il Consumer conoscere la struttura interna di questo cluster.
- La freccia blu rappresenta la connessione del Consumer verso l'esterno (internet).
- La freccia viola rappresenta la connessione imposta dal Provider nei confronti del Consumer. Anche in questo caso la freccia arriva sul rettangolo esterno che racchiude tutte le risorse del Consumer.
- I divieti rossi servono ad indicare le connessioni che sono state negate dal Consumer in base alle proprie decisioni.
- Il quadrato rosso accompagnato dal simbolo di divieto racchiude tutte le connessioni bloccate da parte del Provider.

Lista dei Request Intent definiti dal Consumer:

Listing 7.1. Request Intent

```
ReqInt—1—Src: [app:app—payment—1, name:payments], Dst:[∗:∗, ∗,∗], DstPort: [80], Protocol:[TCP]
ReqInt—2—Src: [app:app—payment—2, name:payments], Dst:[∗:∗, ∗,∗], DstPort: [90], Protocol:[TCP]
ReqInt—3—Src: [app:desktop—products, name:products], Dst:[∗:∗, ∗,∗], DstPort: [85], Protocol:[UDP]
ReqInt—4—Src: [app:mobile—products, name:products], Dst:[142.250.0.0/15], DstPort:[∗], Prot:[ALL]
```

Use-case 1

In questo primo esempio la Verify restituisce *false* perché si verificano delle sovrapposizioni tra i due set di intenti (Figura [7.4]).

- Sovrapposizione tra ReqInt-3 e AuthDeny-3: non è permesso l'accesso a tutte le risorse utilizzando il protocollo UDP.

- Sovrapposizione tra ReqInt-4 e AuthDeny-4: l'indirizzo CIDR risulta tra quelli bloccati.

Listing 7.2. Authorization Intent (1)

```
AuthDeny−1−Src: [∗:∗, ∗,∗], Dst:[∗:∗, name:handle−payments], DstPort: [∗], Protocol:[SCTP]
AuthDeny−2−Src: [∗:∗, ∗,∗], Dst:[∗:∗, name:handle−payments], DstPort: [∗], Protocol:[UDP]
AuthDeny−3−Src: [∗:∗, ∗,∗], Dst:[∗:∗, name:handle−payments], DstPort: [0−79], Protocol:[TCP]
AuthDeny−4−Src: [∗:∗, ∗,∗], Dst:[142.250.0.0/15], DstPort: [∗], Protocol:[ALL]
AuthMand−1−Src: [app:resource−monitor, name:monitoring], Dst:[∗:∗, ∗,∗], DstPort:[43], Protocol:[TCP]
```

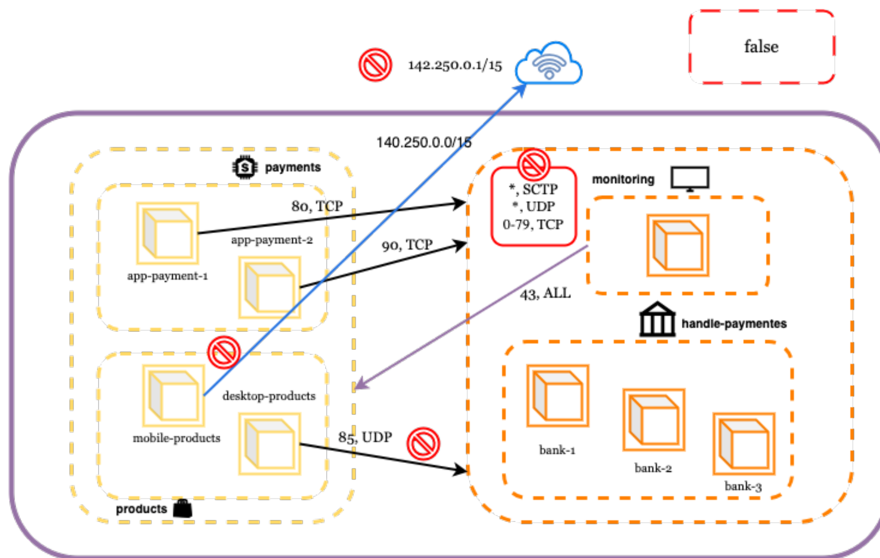

Figura 7.4. Flavor 1

Use-case 2

Anche in questo caso si verificano delle sovrapposizioni e quindi la Verify restituisce *false* (Figura [7.5]).

- Sovrapposizione tra ReqInt-2 e AuthDeny-2 poiché non è permesso l'accesso a tutte le risorse utilizzando la porta 90 e il protocollo TCP.

- Sovrapposizione tra ReqInt-4 e AuthDeny-4 poiché l'indirizzo CIDR risulta tra quelli bloccati.

Listing 7.3. Authorization Intent (2)

```
AuthDeny−1−Src: [∗:∗, ∗,∗], Dst:[∗:∗, name:handle−payments], DstPort: [∗], Protocol:[SCTP]
AuthDeny−3−Src: [∗:∗, ∗,∗], Dst:[∗:∗, name:handle−payments], DstPort: [82−120], Protocol:[TCP]
AuthDeny−4−Src: [∗:∗, ∗,∗], Dst:[192.168.0.1/15], DstPort: [∗], Protocol:[ALL]
AuthMand−1−Src: [app:resource−monitor, name:monitoring], Dst:[∗:∗, ∗,∗], DstPort:[43], Protocol:[TCP]
```

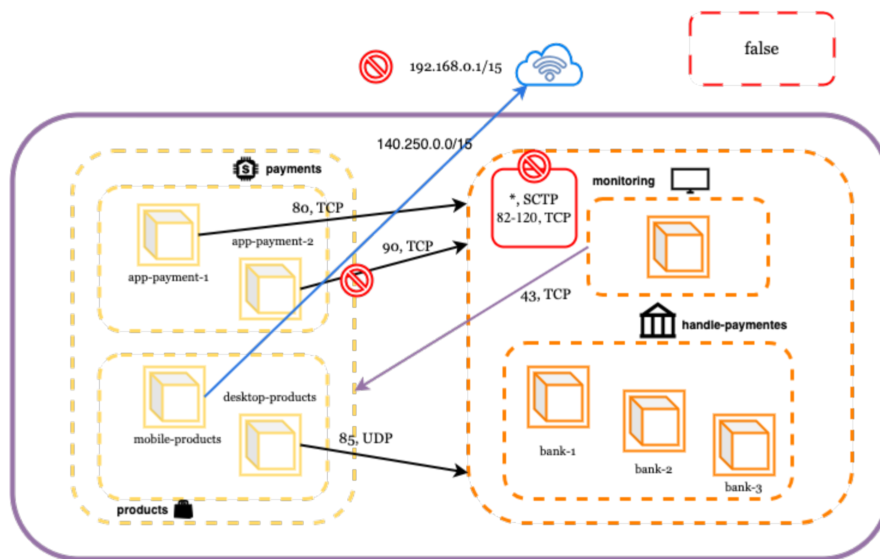

Figura 7.5. Flavor 2

Use-case 3

In questo caso non c'è nessuna sovrapposizione e quindi la Verify restituisce *true* interrompendo la ricerca del candidato (Figura [7.6]).

Listing 7.4. Authorization Intent (3)

```
AuthDeny—1—Src: [*:*, *,*], Dst:[*:*, name:handle—payments], DstPort: [*], Protocol:[SCTP]
AuthDeny—3—Src: [*:*, *,*], Dst:[*:*, name:handle—payments], DstPort: [0—79], Protocol:[TCP]
AuthMand—1—Src: [app:resource—monitor, name:monitoring], Dst:[*:*, *,*], DstPort:[43], Protocol:[TCP]
```

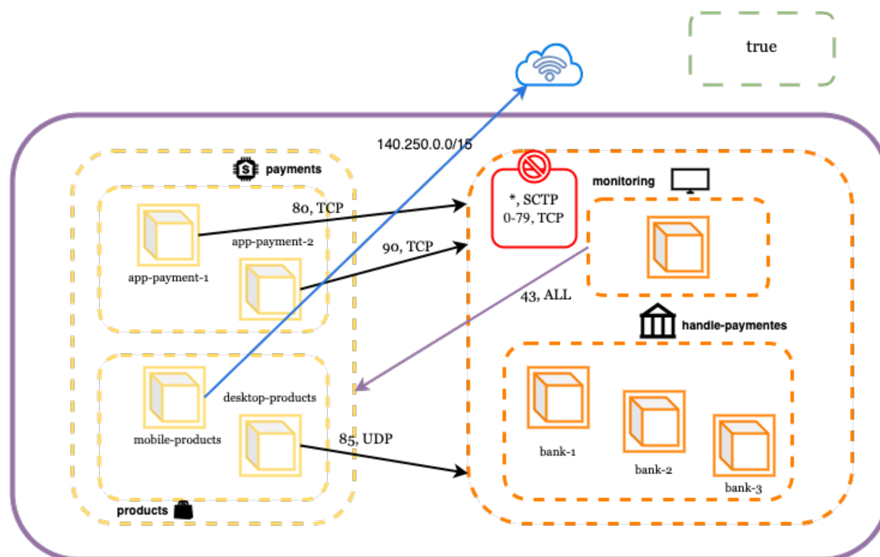

Figura 7.6. Flavor 3

La Demo prevede l'esecuzione dell'ambiente di FLUIDOS Node, unitamente alle altre componenti menzionate nel Capitolo [7]. Questa demo permette di applicare il caso di studio appena illustrato in un ambiente di test realizzabile in locale, grazie a Docker, Kubernetes e agli altri strumenti che verranno installati insieme

al FLUIDOS Node. In questa demo molti passaggi che nella versione finale dovranno essere effettuati in automatico, verranno effettuati manualmente, andando a modificare determinati file e configurazioni. Questa necessità nasce dal fatto che allo stato attuale non è ancora stato implementato un meccanismo per la totale automatizzazione del processo, essendo solo la prima versione del progetto di integrazione.

Installazione FLUIDOS Node

Il primo passo è l'installazione del FLUIDOS Node[1], già realizzato all'interno del progetto FLUIDOS, nei cluster Consumer e Provider. Il processo inizia semplicemente con l'esecuzione di uno script:

```
./setup.sh
```

In Figura [7.7] è possibile vedere le scelte iniziali fatte in termini di configurazione dell'ambiente d'uso: ambiente Demo Kind, un Consumer e un Provider, utilizzo di repository locali e l'abilitazione della ricerca automatica delle risorse.

```
-------------------------------------------------
      WELCOME TO THE FLUIDOS NODE INSTALLER
-------------------------------------------------
We'll now run the installation process for the FLUIDOS node.
What type of environment do you want to use? /
1. Use demo KIND environment (one consumer and one provider) /
2. Use a custom KIND environment with n consumer and m provides /
Please enter the number of the option you want to use:
 1
Do you want to use local repositories? [y/n] y
Do you want to enable resource auto discovery? [y/n] y
```

Figura 7.7. Installazione del FLUIDOS Node

I successivi passaggi nel processo di installazione:

- Installazione Calico CNI per il **Consumer** e per il **Provider**.
- Lo script chiama una funzione per verificare se sono installati tutti gli strumenti necessari e nel caso li installa: KIND, Docker, Kubectl, Helm, liqoctl, jq.
- Viene finalmente installato l'ambiente di lavoro e vengono creati il cluster **Consumer** e il cluster **Provider** .

[1]Codice sorgente disponibile sul sito https://github.com/fluidos-project/node

Modifica Flavor e Peering Candidates

Dopo aver installato FLUIDOS Node, è necessario preparare i Flavor offerti dal **Provider**. I Flavor saranno resi disponibili al Consumer tramite KIND.
I dati del Provider necessari vengono estratti dalla ConfigMap:

```
export KUBECONFIG=fluidos-Provider-1-config
kubectl get cm fluidos-network-manager-identity -n fluidos -o
    yaml
```

Con i dati appena estratti (**domain**, **id**, **nodeID**) [Figura 7.8] vengono modificati i Flavor e i PeeringCandidate.

```
domain: fluidos.eu
ip: 172.19.0.7:30001
nodeID: a8vder1o7a
```

Figura 7.8. Dati del Provider

Applicazione Flavor (Provider)

Una volta modificati i Flavor, è possibile cambiare configurazione passando al cluster Provider ed applicando i Flavor.

```
export KUBECONFIG=../fluidos-Provider-1-config
kubectl apply -f ../flavors/
```

Creazione Service e Solver

Nel cluster Consumer si procede con la creazione del **Service** e del **Solver**.

```
export KUBECONFIG=../fluidos-Consumer-1-config
kubectl create serviceaccount costum-controller -n fluidos
kubectl apply -f ../ControllerServiceAccount/
```

Creazione namespace (Consumer)

Vengono quindi creati i namespace per le risorse del Consumer da offloadare.

```
kubectl create namespace payments
kubectl create namespace products
kubectl create namespace users
```

Creazione namespace (Provider)

Anche i namespace del cluster Provider vengono creati.

```
kubectl create namespace monitoring
kubectl create namespace handle-payments
kubectl apply -f ../Final_demo/Deployments/Provider
```

Applicazione Solver

Viene applicato il Solver.

```
kubectl apply -f
    ../../deployments/node/samples/solver-custom.yaml
```

Patching PeeringCandidates

Questo è il primo passaggio fondamentale in cui si applica il Controller che avvierà il processo di verifica chiamando il modulo Verify. Il primo Flavor che supererà il processo di validazione sarà incluso nel PeeringCandidate (Listing 7.5) contenente i dati del Provider e le autorizzazioni di rete appena autorizzate. Nelle Figure 7.9, 7.10, 7.11 si possono vedere i risultati del processo di Verify applicato ai tre Flavor forniti dal Provider e illustrati precedentemente nello Use-case. Nello specifico i primi due Flavor falliscono la verifica, mentre per il terzo viene riscontrata una compatibilità e la Verify si interrompe ritornando true. Questo sarà il Flavor selezionato.

```
kubectl apply -f ../controller.yaml
kubectl apply -f ../peering_candidates_to_patch
```

Verify

```
[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: [142.250.0.0/15], 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: [*], ProtocolType: [UDP]true false
   | (*) AuthorizationDeny_4 - Src: [ *:* - *:* ], Dst: [ *:* - name:handle-payments ], DstPort: [0-79], ProtocolType: [TCP]true false
   .-> MandatoryConnectionList:
   | (*) AuthorizationMandatory_1 - Src: [ app:resource-monitor - name:monitoring ], Dst: [ *:* - *:* ], DstPort: [43], ProtocolType: [TCP]
----------------------------------------------------------------------------------------------------
[Harmonization] - Consumer accepted monitoring
[VERIFY] Process started
----------------------------------------------------------------------------------------------------
Overlap between these two intents:
----------------------------------------------------------------------------------------------------
 (*) RequestIntent_3 - Src: [ app:mobile-products - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]
 (*) AuthorizationDeny_1 - Src: [ *:* - *:* ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]
[Orchestrator] - verify result: false
```

Figura 7.9. Verify - Flavor 1

Per motivi di spazio si omettono le mandatoryCommunication e le deniedCommunication che è comunque possibile trovare negli Use Case precedentemente illustrati o nel risultato della fase di Verify.

Listing 7.5. PeeringCandidate

```
apiVersion: advertisement.fluidos.eu/v1alpha1
kind: PeeringCandidate
metadata:
  creationTimestamp: "2024-09-27T09:14:31Z"
  generation: 1
```

```
[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 7.10. Verify - Flavor 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]
--------------------------------------------------------------------------------------------------
[Harmonization] - Consumer accepted monitoring
[VERIFY] Process started
--------------------------------------------------------------------------------------------------
[Orchestrator] - verify result: true
--------------------------------------------------------------------------------------------------
```

Figura 7.11. Verify - Flavor 3

```yaml
  name: peeringcandidate-fluidos.eu-k8slice-mio-esempio3
  namespace: fluidos
  resourceVersion: "5242"
  uid: 22409d78-dc87-4237-b4a8-0e566a68ac27
spec:
  available: true
  flavor:
    metadata:
      name: fluidos.eu-k8slice-mio-esempio3
      namespace: fluidos
    spec:
      availability: true
      flavorType:
        typeData:
          characteristics:
            architecture: amd64
            cpu: 19940574197n
            gpu:
              cores: "0"
              memory: "0"
              model: ""
            memory: 7623252Ki
            pods: "110"
            storage: "0"
          policies:
            partitionability:
              cpuMin: "0"
              cpuStep: "1"
              gpuMin: "0"
              gpuStep: "0"
              memoryMin: "0"
```

```
              memoryStep: 100Mi
              podsMin: "0"
              podsStep: "0"
          properties:
            networkAuthorizations:
              mandatoryCommunications:
                ...
              deniedCommunications:
                ...
        typeIdentifier: K8Slice
      location:
        additionalNotes: None
        city: Turin
        country: Italy
        latitude: "10"
        longitude: "58"
      networkPropertyType: networkProperty
      owner:
        domain: fluidos.eu
        ip: 172.19.0.7:30001
        nodeID: a8vder1o7a
      price:
        amount: ""
        currency: ""
        period: ""
      ProviderID: a8vder1o7a
    status:
      creationTime: ""
      expirationTime: ""
      lastUpdateTime: ""
  solverID: solver-sample
status:
  creationTime: "2024-09-27T09:14:31Z"
  lastUpdateTime: ""
```

Reservation

A questo punto, avendo trovato un PeeringCandidate, è possibile prenotare le risorse applicando la Reservation (Listing 7.6) che conterrà sia i dati del Consumer (buyer) che i dati del Provider (seller) (Figura 7.12)

```
kubectl apply -f ../../deployments/node/samples/reservation.yaml
```

Listing 7.6. Reservation

```
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: 172.19.0.7:30001
    nodeID : a8vder1o7a
  # Retrieve from configmap
  buyer:
    domain: fluidos.eu
    ip: 172.19.0.5:30000
    nodeID: szyowcr38e
```

Figura 7.12. Reservation

Allocation

L'applicazione della Reservation prevede anche la generazione di un contatto (Figura 7.13). Il nome del contratto viene inserito all'interno della Allocation (Listing 7.7) che viene dunque modificata e applicata. È quindi possibile verificare il peering appena stabilito tra Consumer e Provider (Figura 7.14).

```
kubectl get contracts -n fluidos
kubectl apply -f ../../deployments/node/samples/allocation.yaml
liqoctl status peer
```

Listing 7.7. Allocation.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
contract:
  name: contract-fluidos.eu-k8slice-mio-esempio3-1268
  namespace: fluidos
```

NAME	SOLVER ID	RESERVE	PURCHASE	SELLER NAME	TRANSACTION ID	CONTRACT NAME	STATUS
reservation-sample	solver-sample	true	true	a8vder1o7a	6b19df731ef6d4f4043adfa25a9f50f9-1732185865781775730	contract-fluidos.eu-k8slice-mio-esempio3-1268	Solved

Figura 7.13. Contratto

Stato del Peering

```
lucaruberto@MacBookPro scripts % liqoctl status peer

┌─ Peered Cluster Information ──────────────────────────────────────────────
│ fluidos-provider-1 - dd6086bc-bf8a-47cc-83af-9ce40e808487
│     Type: OutOfBand
│     Direction
│         Outgoing: Established
│         Incoming: None
│     Authentication
│         Status: Established
│     Network
│         Status: Established
│         Network connection
│             Status:  Connected
│             Latency: 635µs
│             Gateway IPs
│                 Local:  172.19.0.3:30355
│                 Remote: 172.19.0.2:31352
│     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 7.14. Stato del Peering

Modifica Contratto

Il Contratto (Listing 7.8) viene modificato con l'aggiunta del nome della ConfigMap nel campo (networkRequest).

```
export KUBECONFIG=../fluidos−Provider−1−config
kubectl edit contracts contract−fluidos.eu−k8slice−mio−esempio3−1268 −n fluidos
```

Listing 7.8. Contract

```
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Contract
metadata:
  creationTimestamp: "2024-11-21T10:44:26Z"
```

```yaml
  generation: 10
  name: contract-fluidos.eu-k8slice-mio-esempio3-1268
  namespace: fluidos
  resourceVersion: "36415"
  uid: 7a0efcb0-90d2-455d-8799-01de210be36c
spec:
  buyer:
    domain: fluidos.eu
    ip: 172.19.0.5:30000
    nodeID: szyowcr38e
  buyerClusterID: 7271aa0f-75f4-4c75-bbb6-9cc7458a61b2
  configuration:
    data:
      cpu: "1"
      memory: 1Gi
      pods: "110"
    type: K8Slice
  expirationTime: "2025-11-21T10:44:26Z"
  flavor:
    apiVersion: nodecore.fluidos.eu/v1alpha1
    kind: Flavor
    metadata:
      annotations:
        kubectl.kubernetes.io/last-applied-configuration: |
          ...
      name: fluidos.eu-k8slice-mio-esempio3
      namespace: fluidos
    spec:
      availability: true
      flavorType:
        typeData:
          characteristics:
            action: allow
            architecture: amd64
            cpu: 19940574197n
            gpu:
              cores: "0"
              memory: "0"
              model: ""
            memory: 7623252Ki
            pods: "110"
            storage: "0"
          policies:
            partitionability:
              cpuMin: "0"
              cpuStep: "1"
              gpuMin: "0"
              gpuStep: "0"
              memoryMin: "0"
              memoryStep: 100Mi
              podsMin: "0"
              podsStep: "0"
          properties:
            networkAuthorizations:
                ...
        typeIdentifier: K8Slice
      location:
        additionalNotes: None
        city: Turin
        country: Italy
        latitude: "10"
        longitude: "58"
      networkPropertyType: networkProperty
      owner:
        domain: fluidos.eu
        ip: 172.19.0.7:30001
        nodeID: a8vder1o7a
      price:
        amount: ""
        currency: ""
        period: ""
      ProviderID: a8vder1o7a
```

```
  status:
    creationTime: ""
    expirationTime: ""
    lastUpdateTime: ""
networkRequests: esempio-config-map
peeringTargetCredentials:
  clusterID: dd6086bc-bf8a-47cc-83af-9ce40e808487
  clusterName: fluidos-Provider-1
  endpoint: https://172.19.0.7:30912
  token: b2cb7d8ec79c6596f763426fd38fec1d559873e317e773bf4045ddea428b1fdc567b
         ae30cc6f13e3e177b858cf1962154c76ab38e462b6cd3e5e4491b762f7e5
seller:
  domain: fluidos.eu
  ip: 172.19.0.7:30001
  nodeID: a8vder1o7a
transactionID: 6b19df731ef6d4f4043adfa25a9f50f9-173218586578177573
```

Creazione Controller e Service Account (Provider)

Il passaggio della creazione del Controller e del Service Account viene replicato anche per il Provider. Il Controller in questo caso chiama il modulo Harmonize per eseguire l'armonizzazione degli intenti.
L'armonizzatore procede con la risoluzione delle discordanze di Tipo 1 e 2 (Figura 7.15 e di Tipo 3 (Figura 7.16. La risoluzione delle discordanze di Tipo 3 risulta in realtà superflua in questo contesto, ma per poter dare una visione completa del processo è stata comunque eseguita e riportata. Il Controller avvierà poi il Traduttore per l'enforcement degli intenti armonizzati.

```
kubectl create serviceaccount costum-controller -n fluidos
kubectl apply -f ../ControllerServiceAccount/
kubectl apply -f ../controller.yaml
```

```
[DEMO_INFO]    Received the following Request intents (CONSUMER):
  [AcceptMonitoring]: true
  (*) example-intent - Src: [ app:app-payment-1 - name:payments ], Dst: [ *:* - *:* ], DstPort: [80], ProtocolType: [TCP]false true
  (*) example-intent - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]false true
  (*) example-intent - Src: [ app:mobile-payments - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]false true
  (*) example-intent - 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
-----------------------------------------------------------------------------------------------
[DEMO_INFO]     Resolution of TYPE-1 DISCORDANCES...press ENTER to continue.
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
[DEMO_INFO]    List of harmonized REQUEST intents after type-1 discordances resolution:
   (*) example-intent - Src: [ app:app-payment-1 - name:payments ], Dst: [ *:* - *:* ], DstPort: [80], ProtocolType: [TCP]
   (*) example-intent - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]
   (*) example-intent - Src: [ app:mobile-payments - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]
   (*) example-intent - Src: [ app:desktop-products - name:products ], Dst: [ *:* - *:* ], DstPort: [85], ProtocolType: [UDP]
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
[DEMO_INFO]     Resolution of TYPE-2 DISCORDANCES...press ENTER to continue.
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
[DEMO_INFO]    List of harmonized CONSUMER intents after type-2 discordances resolution:
   (*) example-intent - Src: [ app:app-payment-1 - name:payments ], Dst: [ *:* - *:* ], DstPort: [80], ProtocolType: [TCP]
   (*) example-intent - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]
   (*) example-intent - Src: [ app:mobile-payments - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]
   (*) example-intent - Src: [ app:desktop-products - name:products ], Dst: [ *:* - *:* ], DstPort: [85], ProtocolType: [UDP]
   (*) AuthorizationMandatory_1 - Src: [ app:resource-monitor - name:monitoring ], Dst: [ *:* - *:* ], DstPort: [43], ProtocolType: [ALL]
```

Figura 7.15. Discordanze Tipo 1 e 2

```
----------------------------------------------------------------------------------------------
[DEMO_INFO]    Resolution of TYPE-3 DISCORDANCES...press ENTER to continue.
----------------------------------------------------------------------------------------------
[DEMO_INFO]   List of harmonized PROVIDER intents after type-3 discordances resolution:
   (*) example-intent - Src: [ app:app-payment-1 - name:payments ], Dst: [ *:* - *:* ], DstPort: [80], ProtocolType: [TCP]
   (*) example-intent - Src: [ app:app-payment-2 - name:payments ], Dst: [ *:* - *:* ], DstPort: [90], ProtocolType: [TCP]
   (*) example-intent - Src: [ app:mobile-products - name:products ], Dst: [142.250.0.0/15], DstPort: [*], ProtocolType: [ALL]
   (*) example-intent - Src: [ app:desktop-products - name:products ], Dst: [ *:* - *:* ], DstPort: [85], ProtocolType: [UDP]
   (*) AuthorizationMandatory_1 - Src: [ app:resource-monitor - name:monitoring ], Dst: [ *:* - *:* ], DstPort: [43], ProtocolType: [ALL]
```

Figura 7.16. Discordanze Tipo 3

Offloading (Consumer)

Una volta superate le fasi di verifica ed armonizzazione, ed aver prenotato ed allocato le risorse, il Consumer può offloadare le proprie risorse.

```
export KUBECONFIG=../fluidos-Consumer-1-config
liqoctl offload namespace payments --pod-offloading-strategy
    Remote
liqoctl offload namespace products --pod-offloading-strategy
    Remote
kubectl apply -f ../Final_demo/Deployments/Consumer/
kubectl apply -f ../Final_demo/config_map/Esempio_configMap.yaml
```

Verifica

La Figura 7.17 mostra le risorse presenti nel cluster Provider. Oltre alle risorse allocate dalle varie componenti per il funzionamento dell'intero processo, è possibile notare i namespace delle risorse del Provider (handle-payments e monitoring) e quelli delle risorse offloadate dal Consumer(payments-fluidos-Consumer-1-235f85 e products-fluidos-Consumer-1-235f85). La Figura 7.18 w 7.19, mostrano rispettivamente una connessione di rete armonizzata del Consumer e del Provider. Comandi per eseguire il test di una connessione:

```
kubectl exec -it "nome_pod" -n "nome_namespace" -- /bin/sh
wget "indirizzo_ip_pod_da_raggiungere":"porta"
```

```
lucaruberto@MacBookPro scripts % kubectl get namespaces -n fluidos
NAME                                      STATUS   AGE
calico-apiserver                          Active   8h
calico-system                             Active   9h
cert-manager                              Active   8h
default                                   Active   9h
fluidos                                   Active   8h
handle-payments                           Active   6h15m
kube-node-lease                           Active   9h
kube-public                               Active   9h
kube-system                               Active   9h
liqo                                      Active   8h
liqo-storage                              Active   8h
liqo-tenant-fluidos-consumer-1-235f85     Active   6h46m
local-path-storage                        Active   9h
monitoring                                Active   6h15m
payments-fluidos-consumer-1-235f85        Active   6h15m
products-fluidos-consumer-1-235f85        Active   6h15m
tigera-operator                           Active   9h
```

Figura 7.17. Risorse offloadate

```
lucaruberto@MacBookPro scripts % kubectl describe networkpolicy exampleintentpayments-fluidos-consumer-1-235f852771aaca30e
70efdaa7c7e4cad4da920d0044d2d2622509a1615baf5c9b3a0f0 -n payments-fluidos-consumer-1-235f85
Name:         exampleintentpayments-fluidos-consumer-1-235f852771aaca30e70efdaa7c7e4cad4da920d0044d2d2622509a1615baf5c9b3a
0f0
Namespace:    payments-fluidos-consumer-1-235f85
Created on:   2024-11-21 13:54:11 +0100 CET
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     app=app-payment-1
  Not affecting ingress traffic
  Allowing egress traffic:
    To Port: 80/TCP
    To:
      NamespaceSelector: kubernetes.io/metadata.name=default
      PodSelector: <none>
  Policy Types: Egress
```

Figura 7.18. Esempio di connessione del Consumer

```
lucaruberto@MacBookPro scripts % kubectl describe networkpolicy authorizationmandatory1monitoring5ccca4bfed3f4720847d0
44f775ff3d69516283845830ed933006d2b3bc7423 -n monitoring
Name:         authorizationmandatory1monitoring5ccca4bfed3f4720847d044f775ff3d69516283845830ed933006d2b3bc7423a
Namespace:    monitoring
Created on:   2024-11-21 13:54:11 +0100 CET
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     app=resource-monitor
  Not affecting ingress traffic
  Allowing egress traffic:
    To Port: 43/TCP
    To:
      NamespaceSelector: kubernetes.io/metadata.name=payments-fluidos-consumer-1-235f85
      PodSelector: <none>
  Policy Types: Egress
```

Figura 7.19. Esempio di connessione del Provider

Capitolo 8

Conclusioni

Questa tesi ha contribuito allo sviluppo di una prima versione dell'orchestratore di sicurezza nell'ambito del progetto FLUIDOS. L'integrazione della Verify e della Harmonize con gli altri moduli ha prodotto risultati soddisfacenti. È stato possibile testare il comportamento dell'orchestratore anche in locale, riuscendo a replicare il funzionamento dei due cluster e lo scambio di richieste tra i due attori. L'intero processo, allo stato attuale, non è ancora del tutto automatizzato. Sono vari i passaggi da effettuare manualmente per modificare file o configurazioni. Nel caso della Verify e della Harmonize, attualmente alcuni dati vengono presi da file, non essendoci ancora un meccanismo capaci di fornirli direttamente al momento della chiamate delle funzioni. Nell'intero processo di offloading, prenotazione e acquisizione delle risorse e creazione di una connessione di peering, le modifiche di Allocation o del Contract sono effettuate manualmente e durante l'intero processo c'è l'esecuzione di 3 script da avviare dopo aver effettuato queste modifiche. L'orchestratore è stato in oltre testato per il funzionamento con due cluster riproducibili in locale, la versione definitiva dovrebbe poter funzionare con un ambiente molto più esteso e complesso. Una possibile estensione futura è dunque la completa automatizzazione del processo, che richie modifiche anche agli altri moduli già sviluppati, come quelli del FLUIDOS Node.

Bibliografia

[1] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, pp. 27–31, 2015.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[3] G. Gruman, "Welcome to the next tech revolution: Liquid computing," *https://www.infoworld.com/article/2180167/welcome-to-the-next-tech-revolution-liquid-computing.html*, Jul. 2014.

[4] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti, and A. Manzalini, "Computing without borders: The way towards liquid computing," *IEEE Transactions on Cloud Computing*, vol. 11, no. 03, pp. 2820–2838, 2023.

[5] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Gener. Comput. Syst.*, vol. 105, p. 475–491, Apr. 2020.

[6] G. Cloud, "Che cos'è kubernetes?" https://cloud.google.com/learn/what-is-kubernetes.

[7] Kubernetes, "Documentations," https://kubernetes.io/docs/home/.

[8] ——, "Kubernetes design and architecture," https://github.com/kubernetes/design-proposals-archive/blob/main/architecture/architecture.md#the-kubernetes-node.

[9] Kind, https://kind.sigs.k8s.io.

[10] Helm, https://helm.sh/docs/.

[11] Tigera, "About calico," https://docs.tigera.io/calico/latest/about/.

[12] IBM, https://www.ibm.com/docs/pl/cloud-private/3.1.2?topic=ins-calico.

[13] Liqo, https://liqo.io/.

[14] FLUIDOS, https://fluidos.eu/public-deliverables/.

[15] ——, "Fluidos documentation," https://github.com/fluidos-project/Docs.

[16] G. Cloud, "Che cos'è la sicurezza zero trust?" https://cloud.google.com/learn/what-is-zero-trust?hl=it.

[17] FLUIDOS, "Fluidos node documentation," https://github.com/fluidos-project/node/tree/main/docs.

[18] ——, "Rear project," https://github.com/fluidos-project/REAR.

[19] 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," *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, pp. 67–72, 2024.

[20] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "An intent-based solution for network isolation in kubernetes," in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, 2024, pp. 381–386.