Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Dipartimento di Informatica-Scienza e Ingegneria (DISI)
Corso di Laurea Magistrale in Ingegneria Informatica

Sviluppo e Valutazione di uno Scheduler Latency-Aware per Kubernetes

Candidato:

Lorenzo Ziosi

Relatore:

Prof. Paolo Bellavista

Correlatore:

Dott. Alberto Schiassi

Anno Accademico 2023/2024

Sessione II

Ringraziamenti

Desidero esprimere la mia più profonda gratitudine alla mia famiglia, in particolare a mia **mamma**, mio **papà** e mio fratello **Nicolò**. Grazie a voi ho potuto intraprendere questo percorso, che mi ha portato fino a questo importante traguardo. Avete sacrificato tanto per me, donandomi amore, passione e l'ispirazione necessaria per arrivare fino a questo punto.

Un sincero ringraziamento va anche ai miei amici più cari: **Alessandro**, **Giorgio**, **Matteo**, **Riccardo** e **Simone**. Nonostante le differenze e le difficoltà che abbiamo affrontato nel corso degli anni, siete stati una presenza costante e avete condiviso con me momenti indimenticabili che porterò sempre nel cuore.

Un pensiero speciale va a **Chiara**, **Giulietta** e **Sergio**. Grazie per avermi accolto a braccia aperte in ogni occasione, facendomi sentire non solo il benvenuto, ma anche parte integrante della vostra famiglia. Il vostro affetto e la vostra generosità mi hanno sempre fatto sentire a casa.

Infine, il mio ringraziamento più profondo va ad **Alice**, la persona per me più importante. Sei stata la mia costante, la mia guida e la mia forza motrice. Grazie per aver creduto in me anche nei momenti in cui io stesso faticavo a farlo. Sei la ragione per cui continuo a spingermi oltre i miei limiti: il tuo amore e il tuo sostegno mi danno ancora oggi la forza per superare le sfide più difficili. Ti devo più di quanto riesca a esprimere con le sole parole.

Questo traguardo non è soltanto il risultato del mio impegno, ma anche il frutto dell'amore, del sostegno e della vicinanza di tutte le persone che mi hanno accompagnato lungo il cammino. Grazie di cuore a tutti voi.

Indice

Introduzione

0.1 Contesto e Motivazioni

L'**Internet of Things (IoT)** e l'**Industrial Internet of Things (I-IoT)** stanno trasformando rapidamente numerosi settori industriali, dall'automazione delle fabbriche alla gestione delle infrastrutture critiche. In questi contesti, l'**edge computing** gioca un ruolo fondamentale, portando la potenza di calcolo più vicina ai dispositivi e ai sensori, riducendo così la **latenza** e migliorando la reattività dei sistemi. Tuttavia, l'eterogeneità delle risorse, la variabilità delle condizioni di rete e l'urgente necessità di rispondere rapidamente agli eventi richiedono nuove soluzioni per la gestione delle risorse computazionali.

Kubernetes, piattaforma open-source per l'orchestrazione di container, ha guadagnato un'importanza crescente nel facilitare la gestione di applicazioni containerizzate anche in ambienti edge. Tuttavia, lo scheduler nativo di Kubernetes, progettato per ambienti cloud tradizionali, non è ottimizzato per gestire le specifiche esigenze di contesti edge I-IoT, dove la latenza e la posizione geografica delle risorse giocano un ruolo cruciale. La necessità di un sistema di schedulazione che tenga conto della latenza e delle peculiarità di un ambiente edge è quindi evidente.

Il presente lavoro si propone di affrontare questa sfida sviluppando uno **scheduler custom** per Kubernetes, concepito specificamente per contesti edge I-IoT. Lo scheduler sarà **latency-aware**, ossia in grado di ottimizzare la schedulazione dei carichi di lavoro in funzione della latenza di rete e della posizione delle risorse edge, migliorando così le prestazioni complessive del sistema.

0.2 Obiettivi della Tesi

L'obiettivo principale di questa tesi è progettare, implementare e valutare uno scheduler custom per Kubernetes, che sia in grado di ottimizzare la distribuzione dei carichi di lavoro in ambienti edge I-IoT tenendo conto della latenza di rete e delle risorse disponibili. Gli obiettivi specifici includono:

- **Analisi dello Stato dell'Arte:** studiare le soluzioni esistenti per la schedulazione in ambienti Kubernetes e identificare le lacune specifiche per il contesto edge I-IoT.
- **Progettazione dello Scheduler:** definire un'architettura di schedulazione latency-aware, che possa essere integrata nel sistema di orchestrazione Kubernetes.
- **Implementazione:** sviluppare e integrarare lo scheduler in un ambiente Kubernetes.
- **Valutazione Sperimentale:** testare lo scheduler in scenari edge simulati, confrontandone le prestazioni con lo scheduler nativo di Kubernetes e con altre soluzioni latency-aware.
- **Analisi dei Risultati:** analizzare i risultati ottenuti per verificare se e in che misura lo scheduler soddisfa gli obiettivi prefissati.

0.3 Struttura della Tesi

La tesi è organizzata come segue:

- Capitolo 1 - Contesto e obiettivo della tesi: viene presentato il contesto generale della schedulazione in Kubernetes, con particolare enfasi sugli ambienti edge e sulle problematiche legate alla latenza.
- Capitolo 2 - Analisi del Paper di Riferimento: viene discusso il paper di riferimento su cui si basa lo scheduler sviluppato in questa tesi, analizzando i suoi contributi, i limiti e le aree di miglioramento.
- Capitolo 3 - Progettazione dello Scheduler: in questo capitolo viene descritta l'architettura proposta per lo scheduler, i requisiti e le specifiche, e vengono giustificate le scelte tecnologiche.
- Capitolo 4 - Implementazione dello Scheduler: questo capitolo illustra i dettagli dell'implementazione dello scheduler, il setup dell'ambiente di sviluppo, il codice principale, e le metodologie di testing.

- Capitolo 5 - Valutazione e sperimentazione: viene presentata la valutazione sperimentale dello scheduler, con un'analisi dettagliata delle prestazioni e una discussione dei risultati ottenuti.

Capitolo 1

Contesto e obiettivo della tesi

Il campo della schedulazione in Kubernetes è ampiamente studiato, specialmente in contesti cloud tradizionali dove la scalabilità, la disponibilità e la gestione delle risorse sono le principali preoccupazioni. Tuttavia, con l'emergere dell'edge computing, e in particolare del contesto **I-IoT (Industrial Internet of Things)**, nuove sfide stanno emergendo, tra cui la gestione della **latenza**, l'**eterogeneità delle risorse** e la necessità di decisioni di schedulazione consapevoli della **posizione geografica**. In questo capitolo, viene fornita una panoramica delle soluzioni esistenti per la schedulazione in Kubernetes, con un focus sulle **implementazioni custom** e sui lavori di ricerca correlati che affrontano le sfide specifiche del contesto edge I-IoT.

1.1 Kubernetes

Kubernetes è una piattaforma **open-source** sviluppata da Google, progettata per automatizzare la gestione, il deployment e lo scaling di applicazioni containerizzate. Negli ultimi anni, Kubernetes è diventato lo **standard de facto** per l'orchestrazione dei container, grazie alla sua capacità di gestire applicazioni in ambienti distribuiti con grande efficienza e flessibilità. Kubernetes facilita l'esecuzione di applicazioni su **cluster** di macchine fisiche o virtuali, offrendo strumenti avanzati per la gestione delle risorse, il bilanciamento del carico, la scalabilità automatica e la riparazione automatica delle applicazioni in caso di **failure** [7].

Gli scopi principali di Kubernetes includono:

- **Orchestrazione dei Container:** Kubernetes coordina l'esecuzione di container su un cluster di nodi, gestendo il ciclo di vita delle applicazioni e garantendo che siano sempre disponibili e funzionanti.
- **Gestione delle Risorse:** Kubernetes monitora e ottimizza l'utilizzo delle risorse del cluster, assicurando che i carichi di lavoro siano distribuiti in modo efficiente.

- **Scalabilità:** Kubernetes permette di scalare facilmente le applicazioni in base alla domanda, aggiungendo o rimuovendo container automaticamente.
- **Portabilità:** grazie alla sua architettura basata su container, Kubernetes rende le applicazioni altamente portabili tra diversi ambienti di esecuzione, come cloud pubblici, privati o on-premises.

1.1.1 Architettura di Kubernetes

Kubernetes è progettato con un'architettura **modulare** e **scalabile**, composta da diversi componenti che collaborano per fornire funzionalità avanzate di orchestrazione e gestione delle risorse [4]. Alcuni dei principali componenti di Kubernetes includono:

- **Master Node:** il Master Node è il cuore di un cluster Kubernetes, responsabile della gestione e del controllo del cluster. Include vari componenti chiave come il kube-apiserver, il kube-controller-manager, il kube-scheduler e il etcd.
- **Worker Node:** i Worker Node sono i nodi del cluster che eseguono i container e le applicazioni. Ogni Worker Node include il kubelet, il kube-proxy e il container runtime (come Docker o containerd).
- **Pod:** il Pod è l'unità minima di esecuzione in Kubernetes, contenente uno o più container che condividono risorse e spazio di rete. I Pod sono schedulati su nodi del cluster e possono essere scalati in base alle esigenze.
- **Service:** i Service sono un'astrazione che definisce un set di Pod e una politica di accesso a tali Pod. I Service consentono di esporre le applicazioni in modo affidabile e scalabile all'interno del cluster o all'esterno di esso.
- **Deployment:** i Deployment sono un controller che gestisce la distribuzione e l'aggiornamento di applicazioni in Kubernetes. Consentono di definire lo stato desiderato delle applicazioni e di garantire che venga mantenuto in modo automatico.

1.1.2 API

Le API di Kubernetes rappresentano il cuore della piattaforma, permettendo di **interagire** con il cluster per gestire risorse, operazioni di controllo e monitoraggio di applicazioni containerizzate. In questo capitolo, viene fornita una panoramica delle API di Kubernetes, focalizzandosi sui concetti chiave e sulle librerie Go che facilitano lo sviluppo e l'interazione con queste API.

Panoramica delle API di Kubernetes

Kubernetes espone un'ampia serie di API che consentono agli sviluppatori e agli amministratori di orchestrare container, gestire risorse di sistema e automatizzare task operativi. L'interazione con le API può essere effettuata tramite diversi strumenti, come `kubectl`, il client CLI di Kubernetes, oppure direttamente tramite richieste HTTP RESTful [3].

Le API di Kubernetes sono state progettate per essere **estendibili** e **flessibili**. Le principali caratteristiche delle API includono:

- **Modelli di risorse dichiarative:** le risorse vengono gestite tramite manifest YAML o JSON che descrivono lo stato desiderato del sistema.
- **Supporto per oggetti di alto livello:** Kubernetes fornisce oggetti astratti per gestire la complessità del sistema, come Pod, Service, Deployment, ConfigMap e molti altri.
- **Controller e operatori:** le API lavorano con controller e operatori che osservano lo stato corrente delle risorse e fanno convergere il sistema verso lo stato desiderato.
- **Versioning delle API:** Kubernetes supporta diverse versioni delle sue API (v1, beta, alpha), per garantire compatibilità retroattiva e stabilità.

Interazione con le API tramite il Client Go

Per lo sviluppo del progetto descritto nei capitoli successivi è stato scelto il linguaggio **Go**, grazie alla disponibilità di numerose librerie offerte da Kubernetes, tra cui **client-go**. Quest'ultima libreria consente di creare strumenti personalizzati in grado di interagire direttamente con le API di Kubernetes.

Ecco alcune caratteristiche chiave di client-go:

- **Client e Configurazione:** client-go consente di creare facilmente client per comunicare con il server API di Kubernetes. La configurazione può essere generata automaticamente usando il contesto kubeconfig, che permette di autenticarsi e interagire con il cluster.

```
import (
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/tools/clientcmd"
)

func createClient() (*kubernetes.Clientset, error) {
  config, err := clientcmd.BuildConfigFromFlags("",
  ↪  clientcmd.RecommendedHomeFile)
  if err != nil {
    return nil, err
  }
  return kubernetes.NewForConfig(config)
}
```

Codice 1.1: *Esempio di creazione del client*

- **Gestione delle risorse Kubernetes:** la libreria fornisce metodi per creare, leggere, aggiornare e cancellare (CRUD) le risorse Kubernetes come Pod, Service, Deployment e molte altre.

```
pods, err := clientset.CoreV1().Pods(namespace).List(context.TODO(),
↪  metav1.ListOptions{})
if err != nil {
    // handle error
}
```

Codice 1.2: *Esempio di elencazione dei pod*

- **Informer e Cache:** client-go include un sistema di informer e cache che permette di monitorare i cambiamenti delle risorse Kubernetes in modo efficiente, riducendo il carico di richieste API ripetute. Gli informer sono utili per lo sviluppo di operatori e controller.

```
podInformer := informerFactory.Core().V1().Pods().Informer()
podInformer.AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc:    func(obj interface{}) { /* handle add */ },
    UpdateFunc: func(oldObj, newObj interface{}) { /* handle update */ },
    DeleteFunc: func(obj interface{}) { /* handle delete */ },
})
```

Codice 1.3: *Esempio di utilizzo di un informer*

1.1.3 Lo scheduler di Kubernetes

Uno dei componenti chiave di Kubernetes è lo **scheduler**, il modulo responsabile per la decisione su quale nodo del cluster eseguirà un determinato pod (l'unità minima di esecuzione in Kubernetes). Lo scheduler nativo di Kubernetes utilizza un processo in due fasi:

1. **Fase di Filtering (Predicates):** in questa fase, lo scheduler esclude i nodi che non soddisfano i requisiti di base del pod, come le risorse disponibili (CPU, memoria), le affinità di rete, e le politiche di tolleranza ai taint.

2. **Fase di Scoring (Priorities):** dopo il filtraggio, lo scheduler assegna un punteggio ai nodi rimanenti in base a vari criteri, come la disponibilità delle risorse, la prossimità geografica, e altre metriche configurabili. Il nodo con il punteggio più alto viene scelto per eseguire il pod.

Il processo di schedulazione in Kubernetes è progettato per essere estendibile, permettendo agli sviluppatori di creare scheduler custom o di modificare il comportamento dello scheduler esistente tramite configurazioni e plugin. Tuttavia, lo scheduler nativo è principalmente ottimizzato per ambienti cloud **centralizzati** e non considera esplicitamente aspetti critici in contesti edge, come la latenza di rete e la variabilità delle risorse [2].

1.1.4 Scheduler Custom in Kubernetes

Kubernetes offre una notevole flessibilità per quanto riguarda lo sviluppo di **scheduler custom**, permettendo di adattare il processo di schedulazione alle esigenze specifiche di applicazioni o ambienti particolari, come i contesti edge I-IoT. Ci sono diverse strade

percorribili per sviluppare uno scheduler custom in Kubernetes, ognuna delle quali offre diversi gradi di personalizzazione e complessità:

1. **Configurazione Avanzata dello Scheduler Nativo:** Kubernetes permette di configurare il suo scheduler nativo attraverso la personalizzazione di politiche di scheduling esistenti. Utilizzando le policy di schedulazione e i parametri di configurazione, è possibile influenzare le decisioni di schedulazione senza dover scrivere codice. Questa strada è utile per implementazioni relativamente semplici, dove si desidera modificare il comportamento dello scheduler con minimi sforzi di sviluppo.

2. **Sviluppo di uno Scheduler Custom Completamente Separato:** per esigenze estremamente specifiche e complesse, è possibile sviluppare uno scheduler custom completamente separato dallo scheduler nativo di Kubernetes. Questo approccio offre la massima flessibilità, permettendo di definire completamente da zero le logiche di schedulazione. Tuttavia, comporta anche una maggiore complessità e richiede un approfondito lavoro di sviluppo e integrazione. Un esempio potrebbe essere uno scheduler che utilizza metriche non convenzionali o che si integra con sistemi esterni per prendere decisioni di schedulazione.

3. **Utilizzo di Plugins con lo Scheduler Framework:** lo scheduler framework di Kubernetes è un'architettura modulare che permette di estendere o sostituire componenti dello scheduler attraverso plugin custom. Questo approccio consente di sviluppare plugin per le varie fasi del ciclo di schedulazione, come FilterPlugin, ScorePlugin, ReservePlugin, PreBindPlugin, e BindPlugin. Ogni plugin può essere sviluppato in Go e integrato nel processo di schedulazione per soddisfare **esigenze specifiche**, come la gestione della latenza, l'ottimizzazione delle risorse, o l'implementazione di politiche di posizionamento avanzate.

1.1.5 Scheduler framework

Lo **scheduler framework** di Kubernetes è un'architettura avanzata progettata per estendere e personalizzare il comportamento dello scheduler nativo, consentendo agli sviluppatori di creare soluzioni di scheduling su misura per rispondere a esigenze specifiche. Questo framework rappresenta un'evoluzione significativa rispetto alla configurazione tradizionale dello scheduler, offrendo una modularità e flessibilità che lo rendono adatto a

una vasta gamma di scenari, inclusi quelli che richiedono la gestione avanzata delle risorse, come i contesti edge I-IoT [6].

Architettura e Componenti del Kubernetes Scheduler Framework

Il Kubernetes scheduler framework è costituito da diversi **plugin point** o punti di estensione, ognuno dei quali permette di inserire logica custom in vari momenti del ciclo di schedulazione. Questi punti di estensione consentono agli sviluppatori di intervenire durante le diverse fasi della schedulazione di un pod, dalla selezione dei nodi candidati alla decisione finale di binding. Ecco una panoramica dettagliata dei principali plugin point:

1. **QueueSort**

   - **Descrizione:** questo plugin determina l'ordine in cui i pod vengono selezionati per la schedulazione. È il primo passo nel ciclo di schedulazione e può essere utilizzato per modificare la priorità dei pod in coda.
   - **Utilizzo:** un plugin QueueSort personalizzato può essere utilizzato per garantire che pod con priorità più alta (ad esempio, applicazioni critiche) vengano schedulati prima di altri pod meno prioritari.

2. **PreFilter**

   - **Descrizione:** il PreFilter Plugin esegue controlli preliminari su un pod prima che venga processato dalle altre fasi dello scheduler. Questa fase può essere utilizzata per verificare la disponibilità delle risorse o per raccogliere informazioni necessarie per le fasi successive.
   - **Utilizzo:** un PreFilter Plugin potrebbe verificare se i requisiti del pod, come la disponibilità di volumi di storage specifici o configurazioni di rete, sono soddisfatti prima di procedere con la schedulazione.

3. **Filter**

   - **Descrizione:** durante la fase di filtering, lo scheduler esclude i nodi che non soddisfano i requisiti minimi del pod. Il Filter Plugin consente di definire regole custom per questo processo.

- **Utilizzo:** un Filter Plugin può essere utilizzato per escludere nodi che non soddisfano criteri specifici come la disponibilità di risorse di rete, l'allocazione di risorse richieste come CPU o memoria, il rispetto di vincoli di sicurezza, oppure **per filtrare i nodi con latenza elevata**, garantendo che solo i nodi con bassa latenza vengano considerati per la schedulazione del pod.

4. **PostFilter**

   - **Descrizione:** se nessun nodo passa la fase di filtering, il PostFilter Plugin viene attivato. Questo plugin consente di implementare logiche di recupero, come rilassare alcune condizioni o tentare di schedulare su nodi meno ottimali.
   - **Utilizzo:** potrebbe essere utilizzato per rilassare temporaneamente alcuni requisiti del pod (ad esempio, riducendo le risorse richieste) o per riprovare la schedulazione su nodi che inizialmente non soddisfacevano tutti i criteri.

5. **Score**

   - **Descrizione:** il Score Plugin assegna un punteggio ai nodi che hanno superato la fase di filtering, ordinandoli in base a criteri personalizzati.
   - **Utilizzo:** un Score Plugin potrebbe assegnare punteggi ai nodi in base alla loro capacità residua di risorse, alla loro vicinanza ai requisiti di rete o alla loro attuale carica di lavoro, ottimizzando così la distribuzione dei pod nel cluster.

6. **Reserve**

   - **Descrizione:** una volta che un nodo è stato selezionato per un pod, il Reserve Plugin riserva le risorse necessarie su quel nodo.
   - **Utilizzo:** questo plugin può essere utilizzato per garantire che le risorse richieste dal pod siano riservate e non disponibili per altri carichi di lavoro, prevenendo conflitti o overcommitment. I plugin che mantengono lo stato a runtime (chiamati anche "stateful plugins") dovrebbero utilizzare questa fase **per essere notificati dallo scheduler quando le risorse su un nodo vengono riservate e quando vengono liberate per un determinato pod**.

7. **Permit**

- **Descrizione:** il Permit Plugin consente di eseguire controlli aggiuntivi prima che un pod venga legato al nodo. Può bloccare o ritardare la schedulazione in attesa di ulteriori verifiche o condizioni.
- **Utilizzo:** potrebbe essere utilizzato per introdurre una fase di approvazione manuale o automatizzata basata su politiche aziendali prima che il pod venga effettivamente schedulato.

8. **PreBind**

   - **Descrizione:** questo plugin esegue azioni specifiche prima che un pod venga legato a un nodo, come l'aggiornamento di configurazioni o l'esecuzione di script custom.
   - **Utilizzo:** potrebbe essere utilizzato per preparare l'ambiente di esecuzione del nodo, come configurare specifici parametri di rete o configurare volumi di storage necessari per il pod.

9. **Bind**

   - **Descrizione:** il Bind Plugin esegue la fase di binding, cioè associa effettivamente il pod al nodo selezionato.
   - **Utilizzo:** questo plugin consente di personalizzare il processo di binding, ad esempio integrandolo con sistemi di terze parti o monitorando in modo dettagliato l'operazione di binding per garantire la conformità con politiche aziendali.

10. **PostBind**

    - **Descrizione:** questo plugin viene eseguito dopo che il binding è stato completato e può essere utilizzato per eseguire azioni post-schedulazione, come la notifica di sistemi esterni o l'avvio di processi di monitoraggio.
    - **Utilizzo:** potrebbe essere utilizzato per attivare notifiche o avviare servizi correlati, come il monitoraggio delle prestazioni del pod appena schedulato, oppure **per ottenere informazioni su quale nodo ha effettivamente schedulato il pod**, registrando questi dati per analisi o auditing successivi.

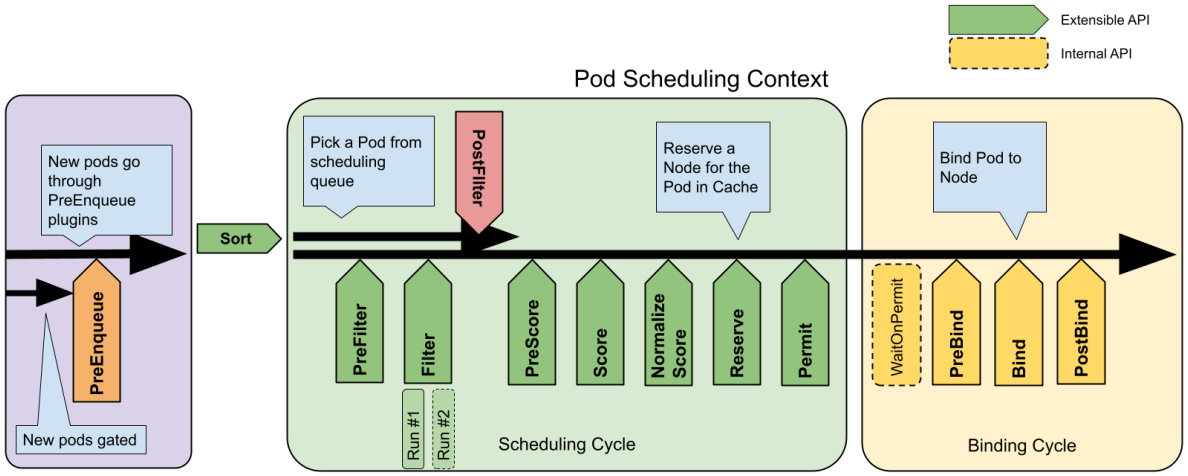


Figura 1.4: *Punti di estensione dello scheduling framework*

Plugin già Sviluppati per lo Scheduler Framwork di Kubernetes

Nel tempo, sono stati sviluppati numerosi plugin che coprono una vasta gamma di funzionalità, migliorando la gestione delle risorse e ottimizzando le decisioni di schedulazione [5]. Di seguito sono elencati alcuni dei plugin più noti e utilizzati:

- **NodeResourcesFit:** questo plugin filtra i nodi in base alla disponibilità delle risorse come CPU e memoria. Esclude i nodi che non hanno abbastanza risorse per soddisfare le richieste del pod.
- **TaintToleration:** il plugin TaintToleration consente ai pod di essere programmati su nodi che hanno "taint", solo se il pod ha una "toleration" corrispondente.
- **PodAffinity:** questo plugin valuta l'affinità o l'anti-affinità dei pod con altri pod nel cluster, determinando se un pod deve essere schedulato vicino a (o lontano da) un altro pod.
- **NodeAffinity:** simile al PodAffinity, ma applicato ai nodi, questo plugin valuta le preferenze o le regole di affinità dei nodi per determinare dove un pod dovrebbe essere schedulato.
- **VolumeBinding:** questo plugin garantisce che i volumi di storage richiesti da un pod siano disponibili e possano essere collegati al nodo scelto.

Nonostante l'esistenza di numerosi plugin che migliorano vari aspetti della schedulazione in Kubernetes, è importante sottolineare che non esistono plugin già sviluppati che

offrano una funzionalità di analisi della latenza dei nodi. I plugin esistenti si concentrano principalmente sulla gestione delle risorse, sulle preferenze di affinità, e sulle necessità di storage, ma non tengono conto della latenza di rete come un fattore determinante nella decisione di schedulazione.

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta2
kind: KubeSchedulerConfiguration
profiles:
- schedulerName: multipoint-scheduler
    plugins:

    # Disable the default QueueSort plugin
    queueSort:
        enabled:
        - name: 'CustomQueueSort'
        disabled:
        - name: 'DefaultQueueSort'

    # Enable custom Filter plugins
    filter:
        enabled:
        - name: 'CustomPlugin1'
        - name: 'CustomPlugin2'
        - name: 'DefaultPlugin2'
        disabled:
        - name: 'DefaultPlugin1'

    # Enable and reorder custom score plugins
    score:
        enabled:
        - name: 'DefaultPlugin2'
        weight: 1
        - name: 'DefaultPlugin1'
        weight: 3
```

Codice 1.5: *Configurazione di esempio per "kube-scheduler"*

1.1.6 Scheduler Custom per Contesti Edge

In ambienti edge, le limitazioni di banda, la latenza di rete e l'eterogeneità dei dispositivi richiedono soluzioni di schedulazione più sofisticate rispetto a quelle utilizzate nei data center tradizionali. Alcuni approcci per la schedulazione in contesti edge includono:

- **Schedulazione Basata su Geolocalizzazione:** alcuni scheduler custom tengono conto della posizione geografica dei nodi edge per ridurre la latenza tra il dispositivo

e il nodo su cui è eseguito il pod. Questo approccio è particolarmente utile in applicazioni I-IoT dove la reattività è critica.

- **Schedulazione Basata sulla Latenza:** altri scheduler custom monitorano la latenza di rete in tempo reale e utilizzano queste informazioni per decidere dove posizionare i pod. Questo è fondamentale per applicazioni che richiedono bassa latenza, come il controllo in tempo reale dei processi industriali.

- **Schedulazione Basata sulle Risorse Disponibili:** gli ambienti edge spesso presentano risorse limitate e variabili. Alcuni scheduler sono progettati per tenere conto dell'eterogeneità dei nodi edge, assegnando carichi di lavoro in base alle risorse attualmente disponibili e alle capacità specifiche di ciascun nodo.

1.1.7 Modelli Esistenti

Negli ultimi anni, sono stati proposti vari approcci per migliorare la schedulazione in contesti distribuiti con un focus sulla latenza. Queste soluzioni sono particolarmente rilevanti per applicazioni edge I-IoT, dove la latenza può avere un impatto diretto sulla qualità del servizio e sulla sicurezza.

Alcuni studi hanno proposto estensioni allo scheduler nativo di Kubernetes per renderlo latency-aware. Questi approcci generalmente includono:

- **Modelli di Predizione della Latenza:** alcuni approcci utilizzano modelli di machine learning per prevedere la latenza basata su dati storici e attuali, migliorando l'accuratezza delle decisioni di schedulazione.

- **Monitoraggio in Tempo Reale della Latenza:** includono meccanismi per monitorare continuamente la latenza tra i nodi e tra i dispositivi IoT e i nodi edge, utilizzando questi dati per guidare le decisioni di schedulazione e di spostamento dei pod tra i nodi.

1.2 Conclusioni

Il panorama attuale della schedulazione in Kubernetes per ambienti edge e I-IoT è in continua evoluzione, con numerosi sforzi di ricerca che mirano a rendere lo scheduling più consapevole della latenza e più adatto a contesti distribuiti e eterogenei.

Questo lavoro di tesi si inserisce in questo contesto, proponendo lo sviluppo di uno scheduler-plugin custom per Kubernetes che affronti specificamente le esigenze di latenza e le sfide di ambienti edge I-IoT. Nei capitoli successivi, verranno esplorati in dettaglio il design, l'implementazione e la valutazione di questa soluzione, con l'obiettivo di colmare alcune delle lacune identificate in questo stato dell'arte.

Capitolo 2

Analisi del Paper di Riferimento

Il paper "Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge" [1] propone un approccio innovativo per la schedulazione in ambienti Kubernetes focalizzato sulla **riduzione della latenza** end-to-end percepita dagli **utenti finali**. Questo lavoro affronta una delle principali sfide emergenti nel contesto dell'edge computing, dove la vicinanza delle risorse ai dispositivi degli utenti è cruciale per garantire prestazioni elevate e reattività delle applicazioni. Al fine di prendere decisioni di schedulazione, il paper propone una soluzione che integra **misurazioni della latenza** direttamente a livello applicativo, offrendo così un miglioramento significativo rispetto agli approcci tradizionali che si basano su metriche raccolte a livello di rete.

2.1 Obiettivi

Il principale obiettivo del paper è sviluppare un meccanismo di schedulazione latency-aware per Kubernetes che sia in grado di ridurre la latenza end-to-end (E2E) percepita dagli utenti. Per raggiungere questo obiettivo, gli autori propongono un'architettura che raccoglie metriche di latenza a livello applicativo piuttosto che a livello di rete, e utilizza queste informazioni per prendere **decisioni di schedulazione dinamiche**, garantendo che i pod vengano posizionati sui nodi del cluster che offrono la latenza più bassa possibile.

2.2 Contributi Principali del Paper

Il paper offre diversi contributi innovativi che possono essere riassunti nei seguenti punti:

1. **Raccolta di Metriche di Latenza a Livello Applicativo:** invece di basarsi su servizi esterni per monitorare la latenza di rete, il metodo proposto nel paper raccoglie le misurazioni della latenza direttamente dall'applicazione orchestra-

ta, permettendo di ottenere una visione più accurata della latenza percepita dagli utenti.

2. **Schedulazione Iterativa Basata su Scoperta:** gli autori propongono una strategia di scoperta iterativa, in cui repliche temporanee dell'applicazione (chiamate "sentinel replica") vengono distribuite su diversi nodi del cluster per misurare la latenza effettiva. Questa strategia consente al sistema di individuare dinamicamente il nodo con la latenza più bassa per ospitare il pod principale.

3. **Integrazione di un Meccanismo di De-Schedulazione:** il paper introduce anche un meccanismo di de-schedulazione che rimuove automaticamente le repliche meno performanti, consentendo di mantenere solo le istanze con la latenza più bassa, ottimizzando ulteriormente l'allocazione delle risorse.

4. **Adattamento Dinamico alle Condizioni Variabili:** un ulteriore contributo del paper è la capacità del sistema di adattarsi dinamicamente a cambiamenti nelle condizioni di rete o di carico computazionale, garantendo che le decisioni di schedulazione siano sempre ottimizzate in base alla latenza percepita.

2.3 Architettura Proposta

L'architettura proposta dagli autori è composta da diverse componenti chiave:

- **Scheduler Latency-Aware:** questo componente è responsabile di distribuire le repliche dell'applicazione sui nodi del cluster e di raccogliere le metriche di latenza dai nodi per determinare quale replica offre le migliori prestazioni in termini di latenza.

- **De-Scheduler:** questo componente lavora in stretta collaborazione con lo scheduler per eliminare le repliche che offrono prestazioni inferiori, mantenendo solo quelle con la latenza più bassa.

- **Applicazione di Misurazione della Latenza:** composta da un client e un server, questa applicazione misura la latenza round-trip (RTT) tra l'utente finale e i vari nodi del cluster, inviando queste informazioni allo scheduler per supportare le decisioni di schedulazione.

- **MQTT Broker:** utilizzato per facilitare la comunicazione tra le diverse componenti del sistema, specialmente tra l'applicazione di misurazione della latenza e il de-scheduler.

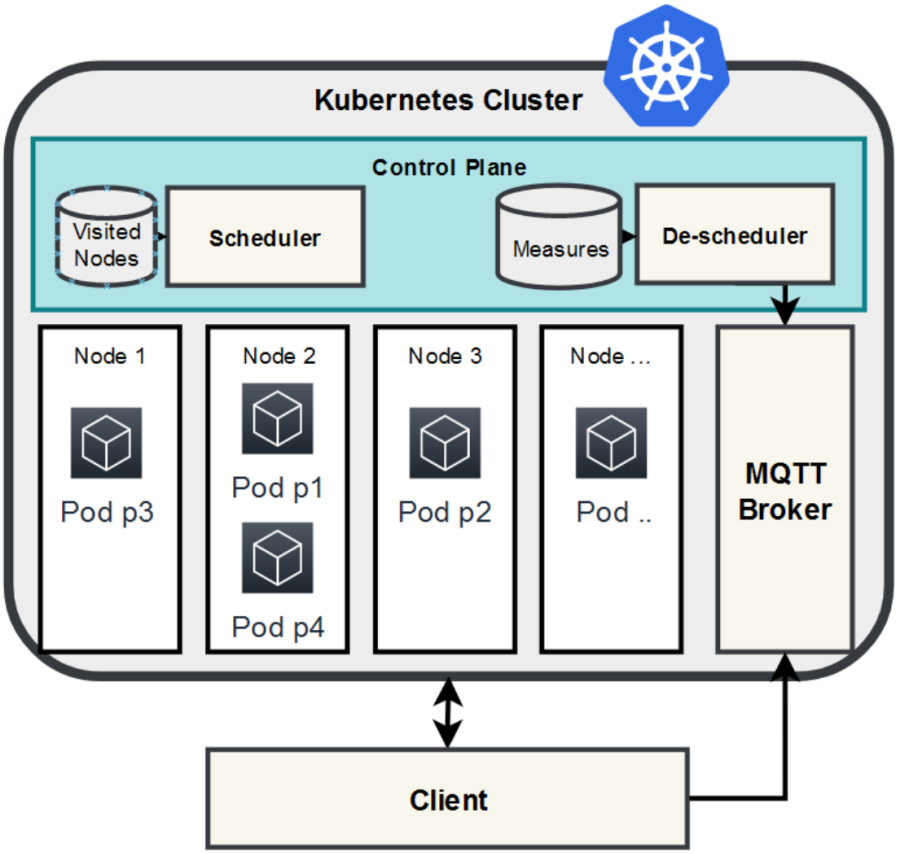


Figura 2.1: *Architettura Proposta in [1].*

L'intero sistema è progettato per essere trasparente all'utente finale, che percepisce solo un miglioramento della latenza man mano che il sistema converge verso la configurazione ottimale.

2.4 Algoritmo di Descheduling

Il de-scheduling flow rappresenta il processo mediante il quale il sistema **identifica** e **rimuove** i pod meno performanti, garantendo che il servizio continui a funzionare sui nodi con la latenza più bassa. Gli autori non adottano un approccio che identifica immediatamente il nodo con la latenza più bassa per l'esecuzione di un pod. Al contrario, il nodo ottimale viene identificato attraverso un **processo iterativo e sperimentale**, in cui il sistema tenta diverse allocazioni e raccoglie dati per determinare la configurazione

ottimale. Questo approccio è definito come strategia di scoperta iterativa, in cui la latenza end-to-end (E2E) viene progressivamente ottimizzata attraverso una serie di tentativi controllati.

Il processo in questione è composto dai seguenti passaggi:

1. **Sottoscrizione al Topic MQTT:** il de-scheduler si sottoscrive a un topic specifico del broker MQTT, ricevendo le misurazioni di latenza (RTT) inviate dai client dell'applicazione.

2. **Ricezione delle Misurazioni di Latenza:** quando un client dell'applicazione comunica con un pod, misura la latenza RTT e invia questi dati, insieme all'identificativo del pod e a un timestamp, al broker MQTT. Il de-scheduler riceve queste informazioni dal topic MQTT.

3. **Archiviazione delle Misurazioni:** il de-scheduler archivia queste misurazioni in una struttura dati. Ogni misurazione è associata a un pod specifico, consentendo al sistema di confrontare le prestazioni dei vari pod.

4. **Soglia di De-Schedulazione:** il de-scheduler verifica se è stata raggiunta la soglia di de-schedulazione, definita come il numero minimo di misurazioni richieste per prendere una decisione di de-schedulazione. Se questa soglia viene superata, il de-scheduler attiva la logica di de-schedulazione (punti 5-7).

5. **Identificazione del Pod da Rimuovere:** il de-scheduler analizza le misurazioni di latenza per determinare quale pod ha le peggiori prestazioni (cioè il RTT più alto). In caso di parità, il pod con il timestamp più vecchio viene scelto per la rimozione.

6. **De-Schedulazione del Pod:** il pod identificato come meno performante viene de-schedulato, ovvero rimosso dal cluster. Questo provoca uno squilibrio nello stato desiderato del cluster, che induce Kubernetes a creare una nuova istanza del pod.

7. **Ri-Schedulazione del Pod:** una volta che un pod viene de-schedulato, Kubernetes avvia una nuova istanza del pod, e il processo di schedulazione riprende per posizionare il nuovo pod su un altro nodo del cluster non ancora visitato o con prestazioni migliori.

8. **Ripetizione del Processo:** questo processo continua iterativamente fino a quando tutti i nodi del cluster sono stati valutati e il sistema ha identificato il miglior set di nodi su cui mantenere i pod, garantendo la latenza minima possibile per l'utente finale.

Il de-scheduling flow assicura che il servizio venga eseguito sui nodi con la latenza più bassa, migliorando l'esperienza dell'utente finale e ottimizzando l'uso delle risorse del cluster.

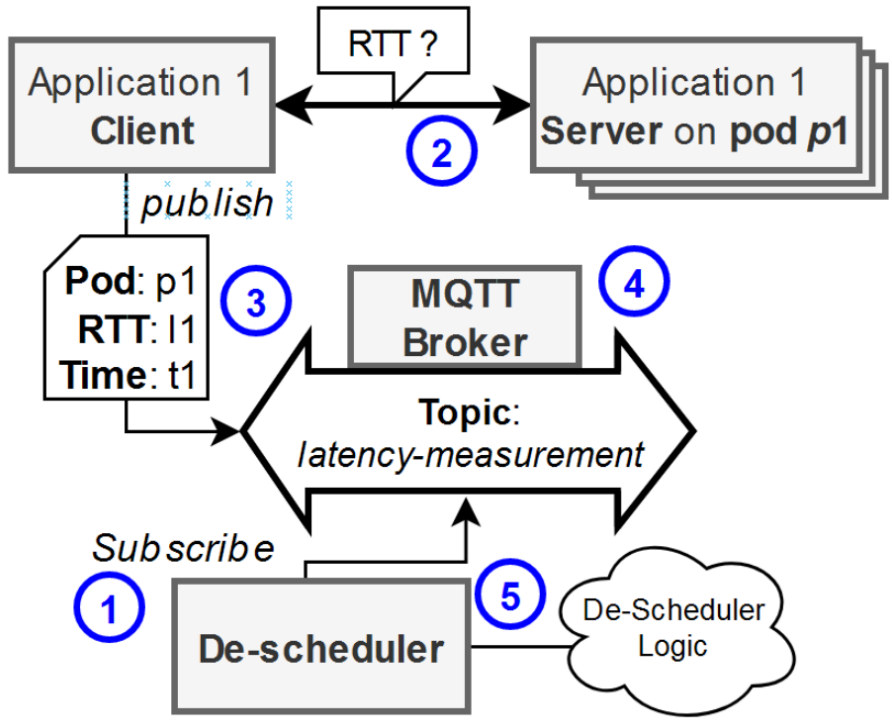


Figura 2.2: *Descheduling flow proposto in [1].*

2.5 Valutazione delle Prestazioni

Gli autori hanno implementato la loro soluzione in un cluster Kubernetes composto da 10 nodi virtuali, ciascuno con una latenza di rete simulata per emulare condizioni realistiche di edge computing. La valutazione sperimentale ha dimostrato che il loro approccio permette di ridurre significativamente la latenza percepita rispetto allo scheduler di default di Kubernetes.

In particolare, il sistema è stato in grado di:

- **Ridurre la Latenza E2E:** la latenza è stata ridotta a un valore minimo di 1ms dopo aver esplorato tutti i nodi del cluster, un miglioramento notevole rispetto alla latenza iniziale di 200ms misurata su uno dei nodi.
- **Convergenza nel Tempo:** il sistema converge verso la configurazione ottimale in un tempo ragionevole, determinato principalmente dal numero di nodi nel cluster e dal tempo necessario per osservare e misurare la latenza su ciascun nodo.
- **Adattabilità:** il meccanismo di schedulazione proposto si è dimostrato capace di adattarsi a variazioni nelle condizioni di rete e di carico, garantendo che la latenza rimanesse minima anche in presenza di cambiamenti dinamici.

2.6 Limiti e Sfide Aperte

Nonostante i risultati promettenti, il paper evidenzia alcune sfide e limiti dell'approccio proposto:

- **Rischio di Comportamenti Oscillatori:** il processo iterativo di scoperta del nodo ottimale, combinato con la possibilità di reset periodici della struttura dei nodi visitati (insieme V), potrebbe portare a comportamenti oscillatori, dove il sistema continua a passare da un nodo all'altro senza stabilizzarsi realmente.
- **Dipendenza da Condizioni di Partenza:** la necessità di distribuire e gestire repliche multiple dell'applicazione per misurare la latenza potrebbe introdurre un overhead significativo, specialmente in ambienti con risorse limitate.
- **Gestione dei Failure e Resilienza:** l'approccio proposto non è stato testato in combinazione con altri sistemi di monitoraggio e gestione delle risorse, il che potrebbe rappresentare una sfida in scenari reali complessi.

Molti dei punti appena descritti derivano dal fatto che la soluzione proposta non è implementata come un plugin per il **Kubernetes Scheduler Framework**. Essa limita la **manutenibilità**, la **portabilità**, la **compatibilità** e l'**adottabilità** della soluzione. Integrare la logica di scheduling proposta nel framework esistente di Kubernetes potrebbe

non solo mitigare questi problemi (lasciando spazio di esecuzione anche ad altri plugin), ma anche ampliare l'adozione della soluzione all'interno della comunità Kubernetes, rendendola più sostenibile e utile in un contesto di produzione reale.

Tuttavia, questa analisi del paper fornisce una base solida per lo sviluppo del plugin di schedulazione custom descritto in questa tesi, che mira a implementare e potenzialmente migliorare l'approccio proposto, esplorando ulteriori ottimizzazioni e adattamenti necessari per specifici casi d'uso nel contesto edge I-IoT.

Capitolo 3

Progettazione dello Scheduler

La progettazione dello scheduler-plugin latency-aware per Kubernetes richiede una comprensione approfondita delle esigenze specifiche degli ambienti edge I-IoT, dove la latenza e l'eterogeneità delle risorse giocano un ruolo cruciale. Lo scopo principale di questo scheduler è ottimizzare la latenza end-to-end (E2E) percepita dagli utenti finali, garantendo al contempo un utilizzo efficiente delle risorse del cluster. Questo capitolo descrive in dettaglio i **requisiti** e le **specifiche**, l'**architettura** proposta, gli **algoritmi** utilizzati, e le **scelte tecnologiche** adottate per la progettazione e implementazione dello scheduler.

3.1 Requisiti e Specifiche

1. **Raccolta Dati in Tempo Reale:** il sistema deve raccogliere continuamente dati di latenza dai nodi del cluster, misurando la latenza E2E percepita dall'utente.

2. **Schedulazione Iterativa e Dinamica:** lo scheduler deve adottare un approccio iterativo, testando diversi nodi del cluster per trovare la configurazione con la latenza minima.

3. **De-Schedulazione Automatica:** il sistema deve essere in grado di rimuovere automaticamente le repliche dei pod con prestazioni inferiori in termini di latenza, mantenendo attivi solo i pod ottimali.

4. **Integrazione con lo scheduler-framework:** lo scheduler deve integrarsi con lo scheduler-framework Kubernetes, rispettando le API messe a disposizione, aumentando la flessibilità e l'interoperabilità del sistema.

5. **Adattabilità a Condizioni Variabili:** lo scheduler deve essere in grado di adattarsi dinamicamente a cambiamenti nelle condizioni di rete e carico computazionale, avviando nuove iterazioni di scoperta quando necessario.

3.2 Architettura

L'architettura è progettata per essere modulare e flessibile, consentendo di adattarsi ai diversi scenari di edge computing. L'architettura si compone di diversi componenti chiave, ciascuno con responsabilità specifiche.

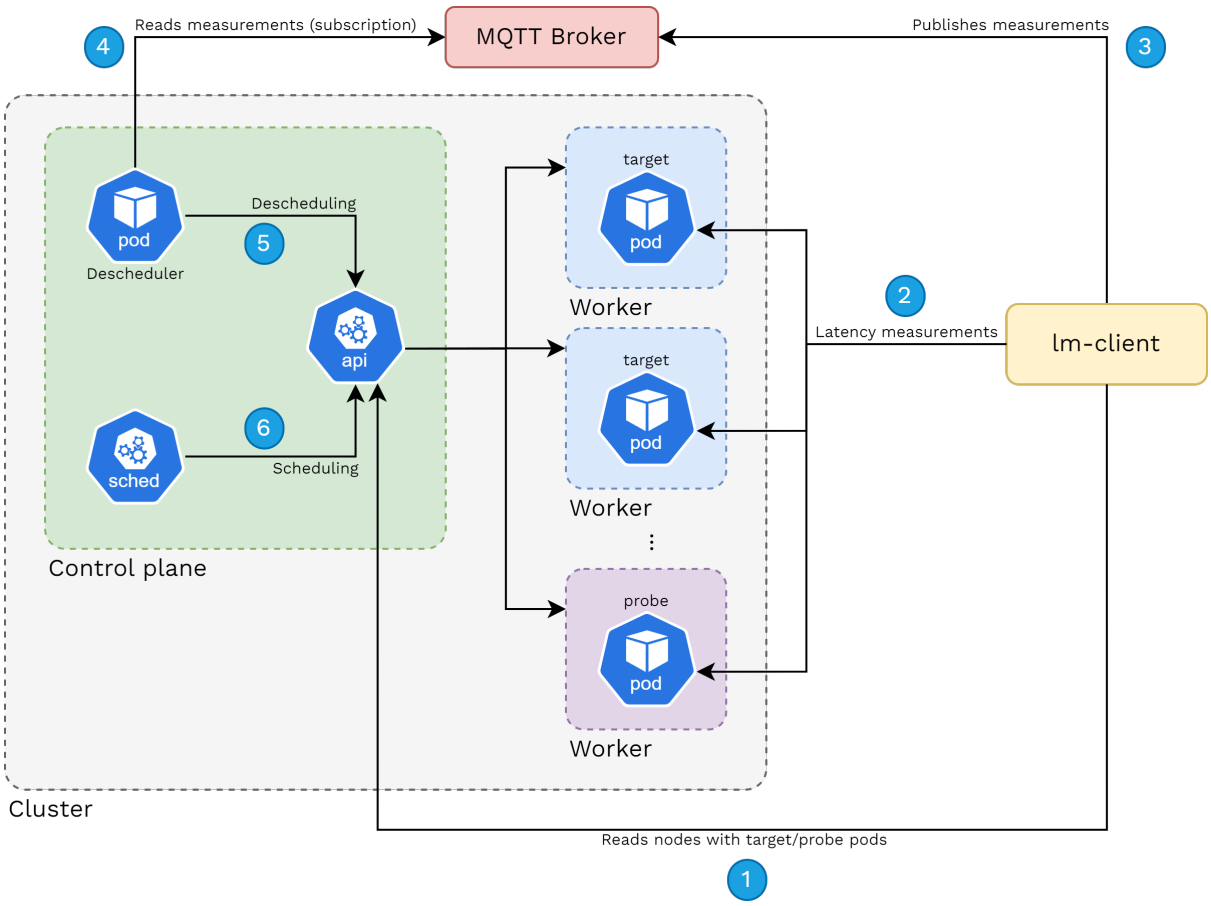


Figura 3.1: *Architettura dello Scheduler-Plugin*

3.2.1 Latency-Meter

Il Latency-Meter è un modulo che raccoglie e aggiorna continuamente le **misurazioni** della latenza tra i client e i pod distribuiti sui nodi del cluster. Per farlo, utilizza un'applicazione client-server distribuita su ciascun nodo, con lo scopo di misurare in tempo reale la latenza di **round-trip (RTT)**. Questo sistema è integrato con un broker MQTT, che permette al modulo di comunicare le misurazioni della latenza e di distribuirle ai vari moduli interessati all'interno del sistema.

Il client, situato all'esterno del cluster, è l'entità che desidera la diminuzione della latenza percepita.

3.2.2 Scheduler

Lo Scheduler gestisce il processo di allocazione dei pod sui nodi, distribuendo i container applicativi e sentinella sui nodi **non ancora visitati**. È sviluppato utilizzando lo Scheduler Framework, il che permette di estenderne le funzionalità con l'aggiunta di altri plugin. Inoltre, si interfaccia con l'API di Kubernetes per orchestrare la creazione e la gestione dei pod sui nodi selezionati, seguendo le logiche di priorità e affinità definite all'interno del cluster.

3.2.3 Descheduler

Il Descheduler ha la funzione di identificare e rimuovere i pod che presentano le **peggiori prestazioni** in termini di latenza. Analizza i dati raccolti dal client, trasmessi attraverso il broker MQTT, per valutare le prestazioni dei vari pod e decidere quali di essi devono essere deschedulati per migliorare l'efficienza del sistema. L'algoritmo di descheduling implementa una logica decisionale basata su **due tipologie di pod**, con l'obiettivo di garantire un miglioramento costante della latenza percepita dal client, ottimizzando la distribuzione delle risorse e migliorando le prestazioni complessive del cluster rispetto al client.

3.2.4 Pod Target

Il pod Target rappresenta l'unità operativa chiave all'interno del cluster, contenente sia il **container applicativo** da esporre ai servizi esterni, sia il **container lm-server** dedicato a rispondere alle richieste del client per la misurazione della latenza. Questa doppia funzione permette al pod target di gestire contemporaneamente l'esecuzione delle applicazioni richieste e di fornire in tempo reale i dati di latenza necessari per ottimizzare le prestazioni del sistema.

3.2.5 Pod Probe

Il pod Probe è un componente essenziale per garantire il raggiungimento della condizione di latenza minima. Questo pod contiene solamente un **container lm-server** dedicato alla misurazione della latenza tra il client e il nodo in cui è eseguito.

3.2.6 Flussi di Comunicazione e Coordinazione

La figura 3.1 contiene anche la sequenza delle operazioni svolte dai vari componenti durante il processo di schedulazione e deschedulazione. I flussi di comunicazione e coordinazione tra i moduli sono essenziali per garantire un funzionamento efficace e coordinato del sistema.

1. **Recupero dei nodi:** il client invia una richiesta al control-plane del cluster per ottenere l'elenco dei nodi coinvolti nel processo, ossia quelli che ospitano un pod target o un pod probe.

2. **Misurazione delle latenze:** il client invia direttamente richieste ai nodi che ospitano i pod target e probe, minimizzando il numero di intermediari, per misurare il tempo di andata e ritorno (RTT) tra il client e i nodi del cluster.

   Viene inoltre disabilitato il load balancing interno di Kubernetes, garantendo che le richieste vengano inviate direttamente ai nodi specifici senza l'intervento di meccanismi di bilanciamento del carico, così da ottenere misurazioni del tempo di andata e ritorno (RTT) più precise.

3. **Pubblicazione delle misurazioni:** il client pubblica le misurazioni di latenza sul broker MQTT, che le distribuisce al descheduler, per prendere decisioni informate sulla deschedulazione dei pod.

4. **Lettura delle misurazioni:** il descheduler legge le misurazioni di latenza dal broker MQTT e valuta le prestazioni dei pod attivi, decidendo quali pod deschedulare per migliorare le prestazioni del sistema.

5. **Deschedulazione del pod:** il descheduler rimuove il pod con le peggiori prestazioni in termini di latenza, liberando risorse e migliorando la distribuzione dei pod sul cluster. Questo genera una discrepanza tra i pod richiesti e quelli effettivamente in esecuzione, che verrà risolta dallo scheduler nella fase successiva.

6. **Schedulazione del pod:** lo scheduler rileva la rimozione del pod e avvia una nuova iterazione del processo di schedulazione per riallineare la distribuzione dei pod sul cluster, garantendo che i nodi già visitati non vengano selezionati nuovamente in quanto non ottimali.

È importante sottolineare che questo processo di schedulazione e deschedulazione iterativo è progettato per garantire un miglioramento **continuo** delle prestazioni del sistema, adattandosi dinamicamente ai cambiamenti nelle condizioni di rete e carico computazionale.

Il processo entra in esecuzione dopo una fase di inizializzazione, in cui vengono distribuiti i pod target e probe sui nodi del cluster in modo del tutto casuale in quanto non si dispone di informazioni sulla latenza tra i nodi.

3.2.7 L'importanza del pod Probe

Come accennato in precedenza, il pod Probe è un componente essenziale per garantire il raggiungimento della condizione di latenza minima. Data la natura **iterativa** dell'algoritmo, ogni qualvolta viene effettuato un descheduling la scelta del nuovo nodo su cui allocare il pod è **aleatoria** in quanto non si dispone di informazioni sulla latenza dei nodi ancora non visitati. Questo potrebbe portare a una scelta subottimale del nodo, con conseguente aumento della latenza percepita dal client.

Di conseguenza, ciò implica che, anche se la condizione attuale è ottimale, il confronto tra i pod potrebbe identificare come "peggiore" un nodo che, tuttavia, risulta comunque migliore rispetto agli altri nodi non ancora visitati. Questo fenomeno si verifica quando viene raggiunta una condizione ottimale, ma restano nodi non ancora visitati.

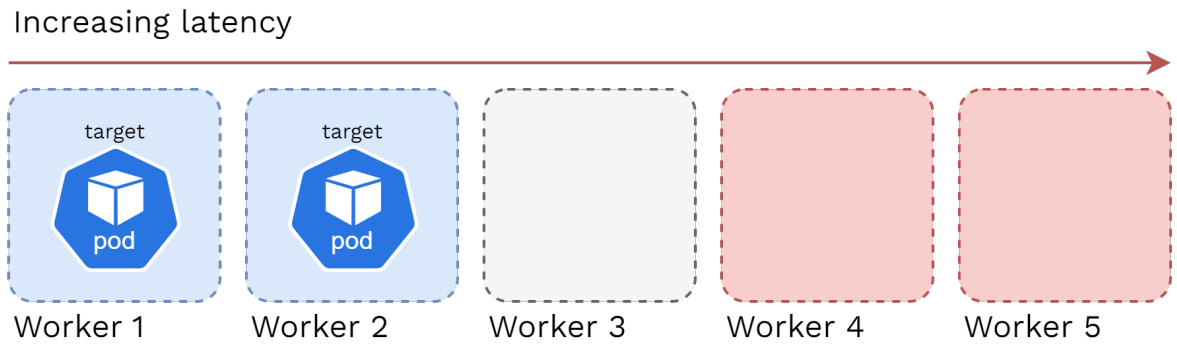


Figura 3.2: *Condizione di partenza per scheduling subottimale*

Nella figura 3.2 si è verificata esattamente la condizione appena descritta: i due pod Target sono stati allocati su nodi con latenza minima (nodi 1 e 2), i nodi con latenza massima (nodi 4 e 5) sono già stati visitati e quindi esclusi dalle valutazioni future. Il

nodo 3, che presenta una latenza maggiore rispetto ai nodi 1 e 2, non è stato ancora visitato.

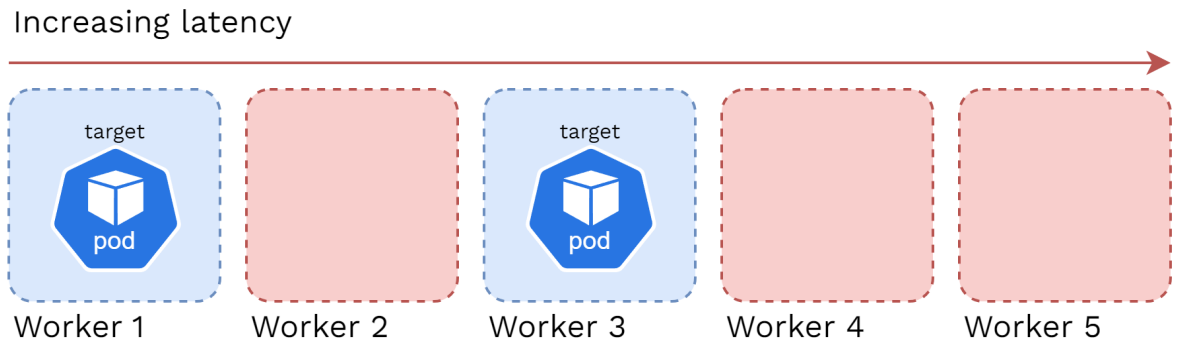


Figura 3.3: *Scheduling subottimale*

Nella figura 3.3 si può vedere come il confronto tra i pod Target identifica il pod nel nodo 2 come "peggiore", rendendolo quindi soggetto a deschedulazione. Di conseguenza, il pod viene rimosso dal nodo 2 e riallocato su un nodo non ancora visitato, in questo caso il nodo 3, nonostante quest'ultimo presenti una latenza maggiore rispetto ai nodi 1 e 2. Questo spostamento comporta un **aumento della latenza** percepita dal client.

L'introduzione del pod Probe con i due seguenti accorgimenti permettono di evitare il problema appena descritto:

1. Se il descheduler identifica come "peggiore" un pod Probe, quest'ultimo viene deschedulato senza influire sulla latenza percepita dal client relativamente al servizio applicativo esposto.

2. Se il descheduler individua come "peggiore" un pod Target, questo viene deschedulato e **riallocato su un nodo in cui è presente un pod Probe**. Questo approccio garantisce un **miglioramento continuo** delle prestazioni del servizio applicativo esposto (ospitato nel pod Target), anche se in maniera graduale, evitando di assegnare i pod Target a nodi non ancora visitati, che potrebbero avere latenze maggiori. Tale scelta è giustificata dal fatto che, se un pod Target viene classificato come "peggiore", significa che **esiste almeno un pod Probe con una latenza inferiore**.

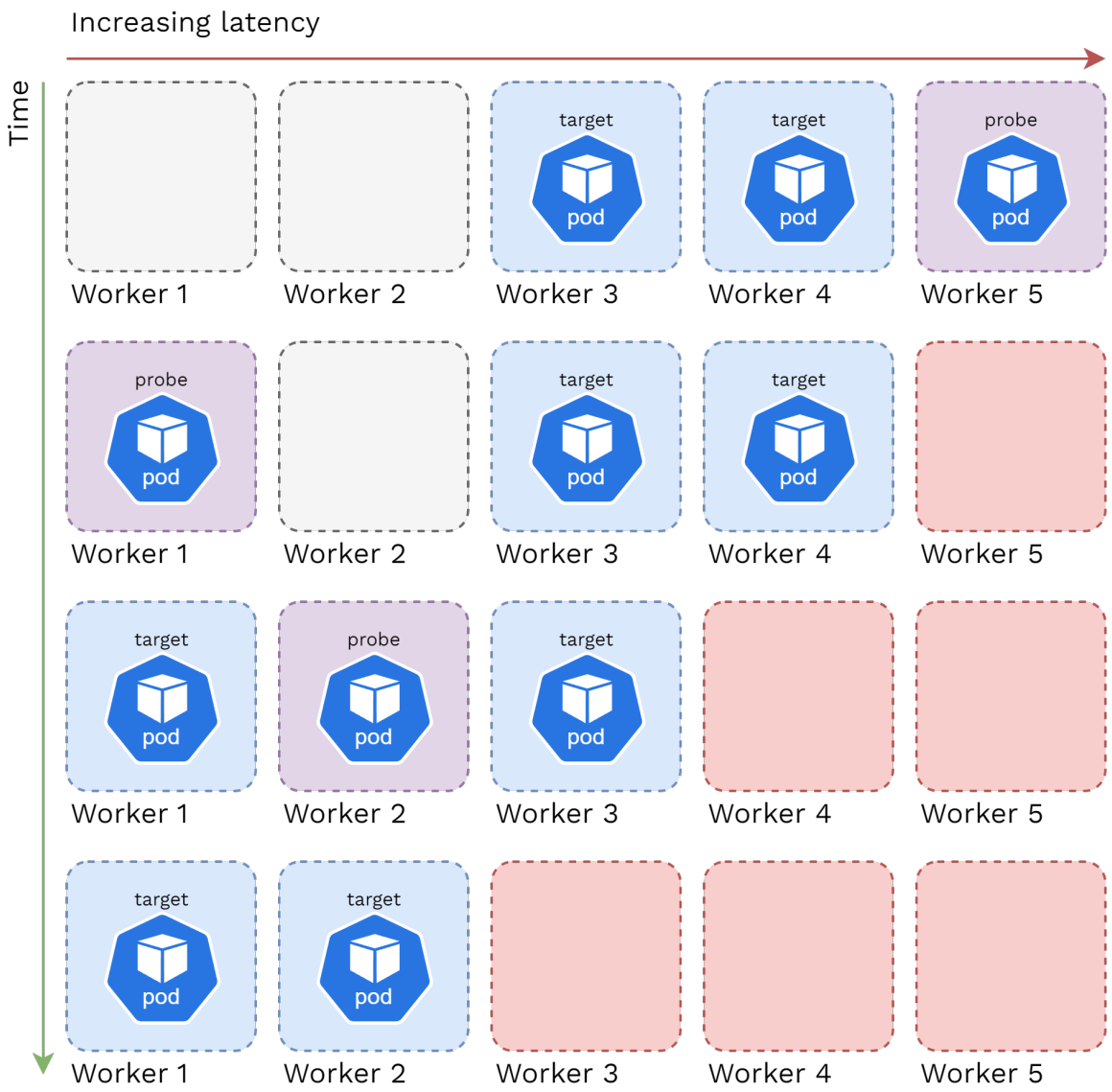


Figura 3.4: *Esempio di scheduling ottimale con il pod Probe*

3.3 Scelte Tecnologiche

La scelta delle tecnologie utilizzate per lo sviluppo del plugin latency-aware è stata guidata da diversi fattori, tra cui la necessità di garantire elevate prestazioni, un'integrazione fluida con Kubernetes e un'elevata **efficienza** nei contesti distribuiti tipici degli ambienti edge. Ogni tecnologia è stata selezionata con l'obiettivo di rispondere alle esigenze specifiche del progetto, migliorando la comunicazione, l'affidabilità e la scalabilità dell'intero sistema.

3.3.1 Linguaggio di Programmazione

Il plugin è stato sviluppato utilizzando **Go**, non solo per i vantaggi in termini di prestazioni e gestione delle risorse, ma anche perché lo **scheduler framework di Kubernetes** è interamente scritto in Go. L'utilizzo di Go permette una perfetta compatibilità con l'ecosistema Kubernetes e consente l'integrazione nativa con le sue API e componenti. Essendo un linguaggio compilato, Go produce eseguibili ad alte prestazioni, ottimizzati per lavorare con carichi ridotti di memoria e CPU, un aspetto cruciale per lo sviluppo di un plugin di scheduling che deve operare in tempo reale senza aggiungere overhead significativo al sistema.

Inoltre, Go offre eccellenti capacità di **gestione della concorrenza** tramite le goroutine, permettendo al plugin di eseguire operazioni simultanee, come il monitoraggio continuo dello stato dei nodi e dei pod, garantendo decisioni rapide ed efficienti. Questa capacità di gestire contemporaneamente più attività in modo efficiente è essenziale in un sistema distribuito come Kubernetes, dove le operazioni devono essere gestite senza ritardi o blocchi.

Infine, Go include numerosi strumenti e librerie che facilitano l'integrazione con le **API di Kubernetes**, semplificando la gestione delle risorse, come la creazione, migrazione e rimozione dei pod all'interno del cluster.

3.3.2 Utilizzo di MQTT per la Comunicazione

Il protocollo **MQTT** è stato scelto come meccanismo di comunicazione tra i diversi moduli del sistema per la sua **leggerezza** e **affidabilità**. MQTT è particolarmente adatto in ambienti distribuiti e con risorse limitate, come gli scenari edge o IoT, dove la rete può essere intermittente e la larghezza di banda limitata.

- **Efficienza nella trasmissione dei messaggi:** il design leggero di MQTT consente una trasmissione rapida dei messaggi con un consumo minimo di risorse, assicurando una bassa latenza nella comunicazione tra i moduli dello scheduler e il descheduler, e mantenendo una comunicazione affidabile tra il client e i nodi del cluster.
- **Supporto per connessioni instabili:** MQTT gestisce in modo efficiente situazioni di connessione instabile, fornendo funzionalità di persistenza dei messaggi

per garantire che le informazioni cruciali, come le misurazioni della latenza, siano sempre recapitate anche in condizioni di rete difficili.

- **Modello publish/subscribe:** il modello di comunicazione **publish/subscribe** di MQTT permette ai vari componenti del sistema di comunicare in modo asincrono, senza richiedere connessioni dirette tra di loro. Questo migliora la scalabilità e la modularità del sistema, consentendo al descheduler di ricevere dati da più fonti senza essere vincolato a una topologia rigida di comunicazione.

3.3.3 Integrazione con Kubernetes

In Kubernetes, per **integrare un plugin di scheduling**, non è possibile semplicemente "aggiungere" il plugin come in altri sistemi. È necessario **ricompilare l'intero scheduler**, il che comporta un processo più complesso rispetto all'aggiunta di un semplice modulo estendibile. Questo significa che, nonostante vengano chiamati "plugin", devono essere integrati direttamente nel codice sorgente dello scheduler e poi ricompilati per essere eseguiti. Questo approccio, pur non essendo flessibile come l'aggiunta di plugin dinamici, garantisce un'integrazione stretta e ottimizzata con le logiche di Kubernetes.

L'integrazione nativa del plugin consente di sfruttare appieno le caratteristiche avanzate di Kubernetes, come il **controller di replica**, le **policy di sicurezza** e le **logiche di priorità**, assicurando che le decisioni di scheduling e descheduling rispettino le policy configurate per il cluster. Inoltre, questa stretta integrazione garantisce che il plugin possa agire in tempo reale su informazioni aggiornate sullo stato dei nodi e dei pod, ottimizzando continuamente l'allocazione delle risorse per ridurre la latenza percepita dagli utenti.

3.4 Considerazioni Finali sulla Progettazione

La progettazione del plugin scheduler latency-aware è stata concepita per rispondere alle esigenze specifiche degli ambienti **edge I-IoT**, dove l'ottimizzazione delle risorse e la riduzione della latenza sono fondamentali. L'adozione di una **architettura modulare** e l'uso di **algoritmi iterativi e dinamici** consentono al sistema di migliorare significativamente le prestazioni del cluster, riducendo la latenza end-to-end (E2E) percepita dagli utenti finali.

Il sistema non solo garantisce efficienza e adattabilità, ma tiene anche conto delle **sfide e delle limitazioni** proprie degli ambienti edge, come la variabilità della rete e le risorse limitate. Per affrontare queste sfide, il plugin include meccanismi di **ottimizzazione continua**, che gli permettono di adattarsi dinamicamente alle condizioni mutevoli dell'ambiente.

Capitolo 4

Implementazione dello Scheduler

In questo capitolo verrà affrontata nel dettaglio l'implementazione dello scheduler-plugin custom per Kubernetes, analizzando i singoli componenti che costituiscono l'architettura del sistema. Ogni componente sarà descritto evidenziando le tecnologie utilizzate e le principali sfide tecniche affrontate durante lo sviluppo.

4.1 Latency-Meter

Il Latency-Meter è strutturato come un'applicazione client-server scritta in Go. Il client invia richieste tramite il protocollo UDP al server, che risponde con un messaggio di ritorno, consentendo al client di calcolare il tempo necessario per processare la richiesta (round-trip time - RTT). Questo calcolo della latenza è essenziale per valutare le performance di rete tra i nodi del cluster Kubernetes.

Per garantire che il client invii richieste solo ai nodi che eseguono un pod Target o Probe, è stato implementato un meccanismo di **discovery dei nodi** A.1. Questo è stato realizzato utilizzando la libreria **client-go** di Kubernetes, che consente di interrogare il cluster e ottenere informazioni in tempo reale sui nodi attivi e sui pod in esecuzione.

Il client necessita di autenticarsi e autorizzarsi con il cluster Kubernetes per poter accedere a queste informazioni. A tal fine, viene utilizzato un file di configurazione che contiene le credenziali e i parametri di accesso al cluster. Questo file di configurazione è stato generato tramite il comando: `kubectl config view --minify --flatten`.

Il risultato di questo comando è salvato in un file che viene utilizzato dal client per gestire correttamente l'accesso sicuro al cluster Kubernetes. Questo approccio garantisce che il client possa identificare e monitorare solo i nodi pertinenti, ottimizzando così la raccolta di dati di latenza senza sovraccaricare il sistema con richieste superflue.

4.1.1 Parametri di configurazione

I componenti del Latency-Meter sono configurabili tramite argomenti inseriti nella riga di comando. Questi argomenti consentono di personalizzare il comportamento del client e del server:

- **Client**
  1. `--kubeconfig`: percorso del file di configurazione Kubernetes del client.
  2. `--broker`: indirizzo IP del broker MQTT.
  3. `--topic`: topic MQTT su cui pubblicare i dati di latenza.
  4. `--clientID`: identificativo univoco del client MQTT.
  5. `--lmport`: porta del server Latency-Meter.
  6. `--label`: selettore di label per individuare tutti pod Target e Probe.
  7. `--namespace`: namespace in cui cercare i pod Target e Probe.
  8. `--measurements`: numero di misurazioni che il client del Latency-Meter deve eseguire per ogni nodo, al fine di ottenere una media accurata del tempo di latenza (RTT).
- **Server**
  1. `--lmport`: porta su cui il server Latency-Meter deve ascoltare le richieste UDP.

4.1.2 Deployment

Il client Latency-Meter può essere eseguito **esternamente** al cluster Kubernetes come container Docker. Questo approccio semplifica notevolmente il processo di esecuzione del client, poiché non richiede l'installazione di alcun software aggiuntivo sulla macchina che lo esegue, a patto che essa abbia accesso alla rete del cluster Kubernetes.

Utilizzando Docker, il client può essere avviato facilmente su qualsiasi macchina, garantendo una maggiore **portabilità** e **flessibilità** operativa. Per eseguire il client Latency-Meter, è sufficiente lanciare un container Docker con i parametri di configurazione desiderati, inclusi il file di configurazione Kubernetes (kubeconfig) e la porta da utilizzare per la comunicazione.

Ecco un esempio del comando Docker da eseguire:

```
docker run --rm -v ./kubeconfig:/root/kubeconfig lm-client
↪  -kubeconfig=/root/kubeconfig -lmport=30007
```

Codice 4.1: *Comando per l'avvio del container client*

In questo comando:

- `--rm`: rimuove il container dopo l'esecuzione.
- `-v ./kubeconfig:/root/kubeconfig`: monta il file di configurazione Kubernetes (kubeconfig) nella directory `/root/kubeconfig` del container.
- `lm-client`: nome dell'immagine Docker del client Latency-Meter.
- `-kubeconfig=/root/kubeconfig`: specifica il percorso del file di configurazione Kubernetes all'interno del container.
- `-lmport=30007`: specifica la porta su cui il client Latency-Meter deve ascoltare le richieste UDP.

Il server Latency-Meter, invece, deve essere eseguito all'interno dei nodi del cluster Kubernetes come pod. I dettagli implementativi di questo aspetto sono trattati nel prossimo paragrafo.

4.2 Pod Target e Probe

Nel capitolo precedente sono stati introdotti i concetti di pod Target e Probe, evidenziando il loro ruolo cruciale nel sistema. In questo paragrafo, verranno descritti nel dettaglio i deployment e i servizi associati che costituiscono i pod Target e Probe, illustrando come vengono configurati e distribuiti all'interno del cluster Kubernetes.

4.2.1 Deployment

Di seguito viene riportato un file di deployment che definisce entrambi i pod. È importante notare che l'uso di nginx nel deployment è solo a scopo dimostrativo, per rappresentare

un'applicazione containerizzata comune che gestisce traffico web. Il file di deployment utilizza anche un container per il Latency-Meter all'interno dei pod Target, che raccoglie dati di latenza utili per il sistema di schedulazione.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:
    feature: latency-aware-deployment
    app: nginx
spec:
    replicas: 3
    selector:
    matchLabels:
        feature:
        ↪ latency-aware-deployment
        app: nginx
    template:
    metadata:
        labels:
        feature:
        ↪ latency-aware-deployment
        app: nginx
    spec:
        containers:
        - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
        - name: lm-server-container
        image: devrols/lm-server:3
        ports:
        - containerPort: 8080
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: lm-server-deployment
    labels:
    feature: latency-aware-deployment
    app: lm-server
spec:
    replicas: 1
    selector:
    matchLabels:
        feature:
        ↪ latency-aware-deployment
        app: lm-server
    template:
    metadata:
        labels:
        feature:
        ↪ latency-aware-deployment
        app: lm-server
    spec:
        containers:
        - name: lm-server-container
        image: devrols/lm-server:3
        ports:
        - containerPort: 8080
```

Codice 4.2: *Deployment dei pod Target e Probe*

Il primo deployment definisce un pod Target che include un container `nginx` e un `lm-server` (Latency-Meter server) per raccogliere le misurazioni della latenza. Il secondo deployment rappresenta un pod Probe con un unico container `lm-server`, utilizzato per monitorare e inviare dati sulla latenza.

4.2.2 Servizi

Mentre il deployment si occupa di definire i pod Target e Probe, i **servizi** permettono di esporre questi pod all'interno del cluster Kubernetes. La tipologia di servizio utilizzata

per il container applicativo dipende dal tipo di accesso richiesto e varia in base alle necessità specifiche del caso d'uso.

Al contrario, il servizio che espone i container `lm-server` deve essere gestito direttamente dal **configuratore** del plugin di scheduling, poiché è una componente essenziale dell'architettura dello scheduler. Questo servizio deve essere configurato con attenzione per evitare che il bilanciamento del carico interno di Kubernetes influisca negativamente sulle misurazioni di latenza. Sebbene il bilanciamento del carico sia utile per garantire scalabilità e affidabilità, esso potrebbe **deviare le richieste** del Latency-Meter verso nodi diversi, compromettendo la coerenza delle misurazioni.

Per questo motivo, il servizio utilizzato è di tipo `NodePort` e viene configurato con la proprietà `externalTrafficPolicy` impostata su `Local`. Questa configurazione assicura che le richieste in arrivo su un nodo vengano gestite dal pod residente sullo stesso nodo (se disponibile), garantendo misurazioni di latenza precise e coerenti, senza interferenze dovute al bilanciamento del carico.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: lm-server-service
  labels:
    feature: latency-aware-deployment
spec:
  type: NodePort
  # from docs: "avoids a second hop for LoadBalancer and NodePort type
  ↪  Services, but risks potentially imbalanced traffic spreading."
  externalTrafficPolicy: Local
  selector:
    feature: latency-aware-deployment
  ports:
    - protocol: UDP
      port: 8080
      targetPort: 8080
      nodePort: 30007
```

Codice 4.3: *Servizio per lm-server*

Questo servizio espone il pod `lm-server` su una porta specifica (8080) e assegna un `nodePort` (30007) per consentire l'accesso al pod da un nodo esterno.

4.3 Scheduler Plugin

Il plugin di scheduling, sviluppato in Go a partire dal codice sorgente dello scheduler framework di Kubernetes, sfrutta diversi **punti di estensione** per implementare una logica di scheduling personalizzata. Di seguito sono descritti i principali punti di estensione utilizzati:

- **Filter A.2:** in questa fase, vengono filtrati i nodi che non soddisfano i requisiti del pod da schedulare. La logica di filtraggio specifica è la seguente:

    1. Se il pod in questione non è né un Target né un Probe, non avviene alcun filtraggio specifico per questo pod.
    2. Se il pod è di tipo Target e ci sono nodi che già eseguono un pod Probe, il pod Target deve sostituire quel pod Probe.
    3. Se un nodo non è già stato visitato in precedenza, viene contrassegnato come valido per il pod in schedulazione.
    4. Infine, se il nodo non rispetta i criteri descritti, viene considerato non idoneo per il pod, poiché è già stato visitato o non risponde ai requisiti.

- **Reserve A.3:** questo punto di estensione viene utilizzato per aggiornare lo stato interno del plugin. È particolarmente utile perché, in caso di fallimento nelle fasi successive, viene invocata la controparte Unreserve per ripristinare lo stato precedente e liberare le risorse. I principali comportamenti del plugin in questa fase includono:

    1. Se il nodo non è già stato visitato, viene aggiunto alla lista dei nodi visitati. Se il pod è di tipo Probe, il nodo viene aggiunto anche alla lista dei nodi che ospitano solo pod Probe.
    2. Se il pod è di tipo Target e il nodo era precedentemente segnato come un nodo contenente solo un pod Probe, questo viene rimosso dalla lista dei nodi contenenti esclusivamente pod Probe.

- **Unserve A.4:** sebbene non sia una vera e propria fase di scheduling, l'Unreserve è strettamente collegata alla fase Reserve ed è utilizzata per ripristinare lo stato del

plugin in caso di fallimento durante il processo di scheduling. Le principali funzioni di questa fase includono:

1. I pod che non sono di tipo Target o Probe vengono semplicemente ignorati.
2. Il nodo viene rimosso dalla lista dei nodi visitati, se presente. Inoltre, se il pod è di tipo Probe o Target ed era stato programmato per sostituire un nodo che ospita solo un pod Probe, il nodo viene reinserito nella lista dei nodi "Probe-only".

- **PostBind A.5:** questo punto di estensione viene utilizzato per eseguire azioni successive al binding del pod al nodo come l'eliminazione di un pod Probe quando viene sostituito da un pod Target. Questa azione viene eseguita in fase di PostBind, poiché è solo a questo punto che si ha la certezza che il binding tra il pod e il nodo sia avvenuto con successo. Se l'eliminazione venisse eseguita in fasi precedenti, potrebbe causare comportamenti indesiderati o incoerenze nel cluster.

4.3.1 Configurazione

Il plugin di scheduling può essere configurato utilizzando il file `scheduler-config.yaml`, che contiene le informazioni riguardanti lo scheduler (o i scheduler, se utilizzati più profili) e i **plugin da utilizzare**. Questo file è caricato all'avvio del scheduler e consente di specificare i plugin da utilizzare e le relative configurazioni.

Un esempio di configurazione per utilizzare lo scheduler-plugin Latency-Aware è la seguente:

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
  leaderElect: false
clientConnection:
  kubeconfig: "REPLACE_ME_WITH_KUBE_CONFIG_PATH"
profiles:
- schedulerName: default-scheduler
  plugins:
    filter:
      enabled:
      - name: LatencyAware
    reserve:
      enabled:
      - name: LatencyAware
    postBind:
      enabled:
      - name: LatencyAware
      disabled:
      - name: "*"
  pluginConfig:
  - name: LatencyAware
    args:
      probeAppLabel: "REPLACE_ME_WITH_PROBE_APP_LABEL"
      targetAppLabel: "REPLACE_ME_WITH_TARGET_APP_LABEL"
      taintToleration: "REPLACE_ME_WITH_TAINT_TOLERATION"
```

Codice 4.4: *Configurazione di esempio per abilitare lo scheduler-plugin*

Mentre non è obbligatorio inserire solamente il plugin Latency-Aware, è importante notare che il plugin deve essere abilitato per le fasi di Filter, Reserve e PostBind. Inoltre, è necessario specificare le label dei pod Target e Probe (nell'esempio 4.2 rispettivamente `nginx` e `lm-server`), che vengono utilizzate per identificare i pod pertinenti durante il processo di scheduling. Infine, è possibile abilitare la compatibilità col plugin **Taint Toleration** per garantire che i pod Target possano essere schedulati su nodi con taint specifici (ad esempio ignorando i nodi control plane).

4.4 Descheduler

L'ultimo componente dell'architettura è il Descheduler, un'applicazione progettata per rimuovere i pod Target o Probe con le latenze più elevate. Il Descheduler è implementato

come un pod che comunica con il broker MQTT per ricevere i dati relativi alla latenza, ed è eseguito sul control plane poiché fa parte dei componenti fondamentali del sistema.

Affinché il Descheduler possa rimuovere i pod Target o Probe con latenze elevate, è indispensabile definire una **politica di rimozione** basata sui dati di latenza raccolti. La politica implementata prevede l'eliminazione del pod con la latenza più alta solo se il numero di misurazioni ricevute dai vari nodi supera una **soglia minima**, configurabile come parametro.

4.4.1 Parametri di configurazione

Il Descheduler è configurabile tramite argomenti inseriti nella riga di comando proprio come il Latency-Meter. Gli argomenti supportati dal Descheduler sono:

1. `--kubeconfig`: percorso del file di configurazione Kubernetes del client.
2. `--broker`: indirizzo IP del broker MQTT.
3. `--topic`: topic MQTT su cui pubblicare i dati di latenza.
4. `--clientID`: identificativo univoco del client MQTT.
5. `--measurements`: Numero di misurazioni che il descheduler deve ricevere affinché possa prendere una decisione di rimozione.
6. `--target`: selettore di label per i pod Target.
7. `--probe`: selettore di label per i pod Probe.
8. `--label`: selettore di label per individuare tutti pod Target e Probe.

Il numero di misurazioni (parametro `--measurements`) è fondamentale per assicurare che il Descheduler prenda decisioni basate su una quantità sufficiente di dati. Se questo parametro è troppo basso, il Descheduler potrebbe rimuovere pod Target o Probe con latenze elevate in modo prematuro o anche dopo aver raggiunto la condizione ottimale, compromettendo la stabilità del sistema.

Ad esempio, in un cluster con 5 nodi, 2 pod Target e 1 pod Probe, il processo di scheduling seguirebbe uno schema simile a quello illustrato in figura 3.4. Se il parametro `--measurements` è impostato a 1, il Descheduler potrebbe scegliere di rimuovere il pod

Target sul nodo 2 a causa della latenza più elevata rispetto a quella del pod Target sul nodo 1, anche se il sistema si trova già in una condizione ottimale.

Per questo motivo, è essenziale che il numero di misurazioni sia **superiore al numero di pod Target da schedulare**, garantendo così un processo decisionale accurato.

4.4.2 Moduli

Il Descheduler è composto da due moduli principali: il **client MQTT** e il **controller**. Il client MQTT è responsabile della comunicazione con il broker MQTT per ricevere i dati di latenza, mentre il controller si occupa di analizzare i dati e prendere decisioni di rimozione dei pod Target o Probe.

I due moduli sono implementati come **goroutine separate** all'interno del processo principale del Descheduler: questo approccio consente di gestire in modo efficiente la comunicazione con il broker MQTT e l'elaborazione dei dati di latenza, garantendo un funzionamento stabile e affidabile del sistema. È necessario però prestare attenzione alla **concorrenza** e alla **sincronizzazione** dei dati tra i due moduli per evitare comportamenti indesiderati o inconsistenze nel sistema: per ovviare a questo problema, è stato utilizzato il package `sync` di Go per garantire la corretta sincronizzazione dei dati tra le goroutine. Le due goroutine prevedono tre funzioni di callback che agiscono tutte su una struttura dati condivisa, chiamata `measurements`, controllata da un mutex, `measurementsMutex`. In particolare, le funzioni di callback sono:

- **Modulo MQTT** A.6:
  1. `messageHandler`: funzione che gestisce i messaggi ricevuti dal broker MQTT. Viene utilizzata per aggiornare la struttura dati `measurements`, aggiungendo le nuove misurazioni di latenza o sostituendo quelle esistenti.
- **Modulo Controller** A.7:
  1. `descheduleIntervalFunction`: funzione che controlla periodicamente i dati di latenza e decide se rimuovere un pod Target o Probe. Se il numero delle misurazioni supera la soglia configurata e la condizione non è ancora quella ottimale (funzione `IsStable`), il controller prende una decisione di rimozione in base ai dati raccolti.

2. `podDescheduledFunction`: funzione che viene invocata ogni qualvolta viene individuato un evento di rimozione di un pod Target o Probe. Questa funzione si occupa di aggiornare la struttura dati `measurements` rimuovendo i dati relativi al pod rimosso.

Il listener `podDescheduledFunction` sembrerebbe ridondante in quanto la struttura dati potrebbe essere aggiornata direttamente nel callback `descheduleIntervalFunction` se venisse individuato un pod da deschedulare, tuttavia la sua presenza è importante per due motivi:

- Garantisce che la struttura dati venga aggiornata in seguito a un effetivo evento di descheduling, evitando che il pod venga rimosso dalla struttura dati prima che la rimozione sia effettivamente avvenuta (ad esempio nel caso si verificasse un errore).
- Lo scheduler potrebbe decidere di rimuovere un pod Probe sostituendolo con un pod Target. Il Descheduler dovrebbe essere in grado di rilevare questa modifica e aggiornare i dati di latenza di conseguenza, garantendo che le decisioni di rimozione siano basate sui dati più recenti disponibili.

4.4.3 Configurazione dei permessi

Per garantire che il Descheduler possa leggere i dati del cluster ma anche rimuovere i pod Target o Probe, è necessario configurare correttamente i **permessi** all'interno del cluster Kubernetes: il seguente file di configurazione definisce le risorse fondamentali per abilitare il corretto funzionamento del Descheduler all'interno di un cluster Kubernetes.

In particolare, viene creata un `ServiceAccount` posizionata all'interno del namespace `kube-system`. Questo `ServiceAccount` è essenziale per **autenticare** e **autorizzare** il Descheduler nell'interazione con il cluster.

Successivamente, viene definito un `ClusterRole` che specifica i permessi necessari. Questo ruolo include i seguenti privilegi:

1. Accesso ai **pod** per eseguire operazioni di ottenimento, elencazione, monitoraggio, aggiornamento e cancellazione;
2. Accesso ai **nodi** per ottenere informazioni, elencarli e monitorarne lo stato;

3. Accesso agli **eventi** per crearli, modificarli e aggiornarli;

4. Accesso ai **deployments** per monitorare e gestire i deployment applicativi.

Infine, attraverso un `ClusterRoleBinding` il `ServiceAccount` viene associata al `ClusterRole`, permettendo al Descheduler di eseguire tutte le operazioni previste per il processo di descheduling.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-descheduler-serviceaccount
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: custom-descheduler-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch", "update", "delete"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["events"]
  verbs: ["create", "patch", "update"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "update"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: custom-descheduler-clusterrolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: custom-descheduler-clusterrole
subjects:
- kind: ServiceAccount
  name: custom-descheduler-serviceaccount
  namespace: kube-system
```

Codice 4.5: *File di configurazione dei permessi per il Descheduler*

Capitolo 5

Valutazione e sperimentazione

In questo capitolo si presentano i risultati sperimentali ottenuti con l'implementazione dello scheduler latency-aware per Kubernetes. L'obiettivo principale delle sperimentazioni è verificare l'**efficacia** dello scheduler nel ridurre la latenza end-to-end (E2E) nelle applicazioni distribuite in ambienti eterogenei, confrontando le sue prestazioni con lo **scheduler nativo** di Kubernetes. Attraverso una serie di test in scenari realistici, è stato possibile dimostrare come il nostro scheduler migliori la distribuzione dei carichi di lavoro in base alla latenza di rete, ottimizzando il posizionamento dei pod nei nodi del cluster e garantendo **prestazioni superiori** in contesti con requisiti stringenti di latenza.

5.1 Scenario di test

Per valutare le prestazioni dello scheduler latency-aware, è stato allestito un ambiente di test su un cluster Kubernetes multi-nodo utilizzando **Kind** (Kubernetes in Docker) [8]. Il cluster è costituito da **9 nodi**, di cui **1 nodo control plane** e **8 nodi worker**. I nodi worker sono stati configurati con valori di **latenza di rete differenti**, simulando un ambiente eterogeneo in cui alcuni nodi presentano latenze più elevate rispetto ad altri. In dettaglio, la configurazione del cluster è la seguente:

Nodo	**Tipo**	**Latenza**
mn-control-plane	control-plane	0ms
mn-worker	worker	100ms
mn-worker2	worker	200ms
mn-worker3	worker	300ms
mn-worker4	worker	400ms
mn-worker5	worker	500ms
mn-worker6	worker	600ms
mn-worker7	worker	700ms
mn-worker8	worker	800ms

Tabella 5.1: Configurazione dei nodi e valori di latenza

Per simulare la latenza di rete tra i nodi del cluster, è stato utilizzato il modulo `tc` (traffic control) di Linux per introdurre un **ritardo nei pacchetti** in entrata e in uscita da ciascun nodo worker. Questo ha permesso di ricreare un ambiente di test realistico in cui i nodi presentano latenze differenti, simulando una rete eterogenea con nodi distribuiti in diverse aree geografiche.

La configurazione Kind utilizzata per creare il cluster è la seguente:

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
# One control plane node and eight "workers".
nodes:
- role: control-plane
- role: worker
- role: worker
- role: worker
- role: worker
- role: worker
- role: worker
- role: worker
```

Codice 5.1: *Configurazione Kind*

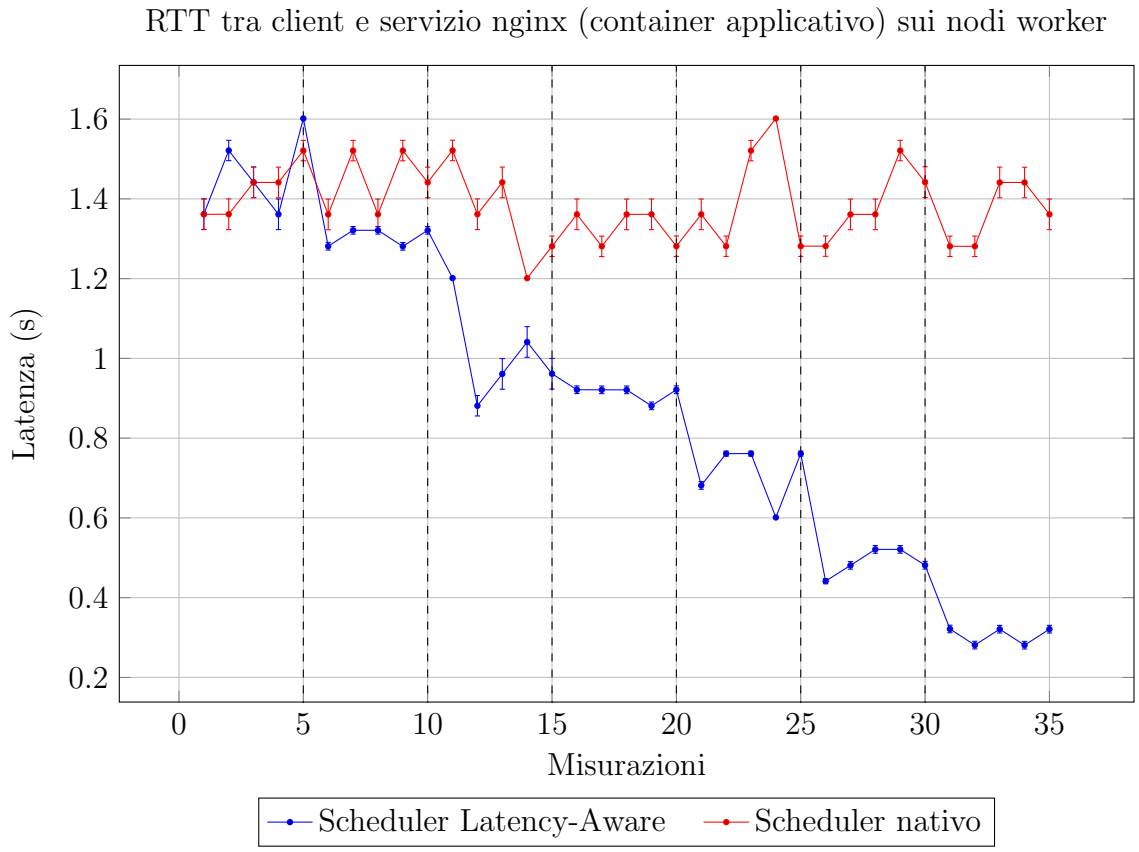
La configurazione dei pod utilizzati nei test è la stessa indicata nel codice 4.2: una sola replica del pod Probe e tre repliche del pod Target, il cui container applicativo è un server nginx.

5.2 Risultati sperimentali

Per valutare le prestazioni dello scheduler latency-aware, sono state condotte **diverse misurazioni consecutive**, durante le quali sono stati rilevati i **tempi di risposta (RTT)** tra il client e il servizio nginx (container applicativo) eseguito sui nodi worker del cluster. I valori registrati risultano approssimativamente il doppio delle latenze impostate, poiché il ritardo viene applicato sia ai pacchetti **in entrata** che a quelli **in uscita**. Durante ciascuna iterazione di descheduling e rescheduling dei pod (rappresentate dalle linee tratteggiate verticali nei grafici sottostanti), sono state effettuate esattamente **5 misurazioni**, ciascuna delle quali rappresenta la media di 5 richieste.

5.2.1 Primo test: scenario peggiore

Il primo test simula lo scenario peggiore, in cui i pod sono collocati nei nodi con la latenza più elevata: i pod Target sono distribuiti sui nodi `mn-worker6` e `mn-worker8`, mentre il pod Probe si trova sul nodo `mn-worker7`. Il test è stato condotto in due fasi: nella prima fase è stato impiegato lo scheduler latency-aware, mentre nella seconda è stato utilizzato lo scheduler nativo di Kubernetes, mantenendo le **stesse condizioni iniziali**. I risultati ottenuti sono illustrati nel grafico seguente:



Il grafico mostra chiaramente che lo scheduler latency-aware riesce a **ridurre la latenza** tra il client e il servizio nginx rispetto allo scheduler nativo di Kubernetes. In questo scenario, il pod Probe viene **progressivamente spostato** su nodi con latenze inferiori, anche se non raggiunge immediatamente quelli con le latenze migliori, arrivandovi solo verso la fine del test. Questo comportamento comporta un numero elevato di iterazioni prima di raggiungere la configurazione ottimale.

L'oscillazione dei valori di latenza visibile nel grafico è causata dal **bilanciamento interno** delle richieste gestito da Kubernetes. Questo comportamento deriva dal mecca-

nismo di distribuzione delle richieste, che considera diversi fattori per bilanciare il carico sui nodi del cluster, ma non assicura sempre che la risposta provenga dal nodo con la latenza più bassa.

Nell'immagine seguente è illustrata l'evoluzione nel tempo dello scheduling dei singoli pod sui nodi worker del cluster:

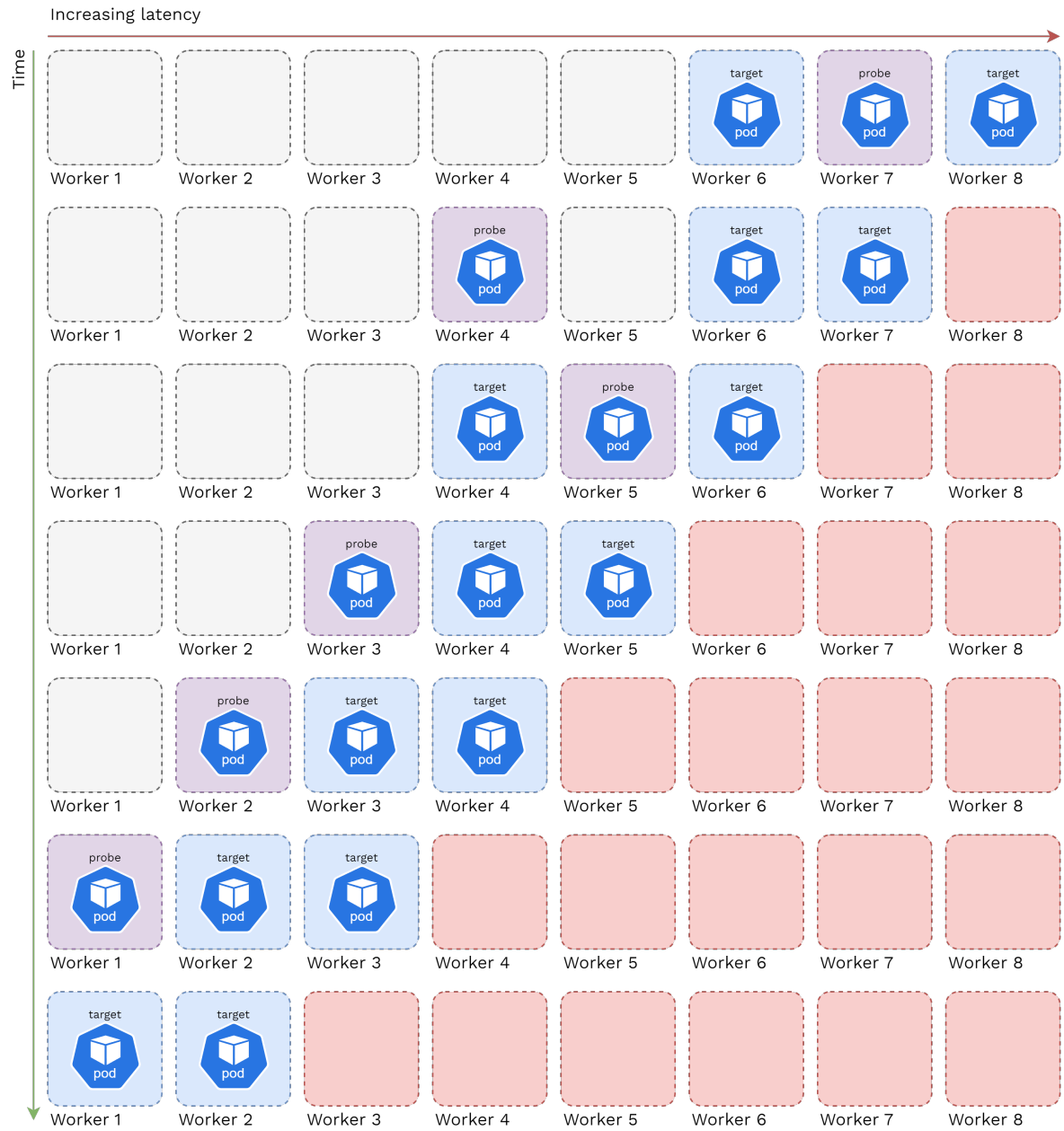
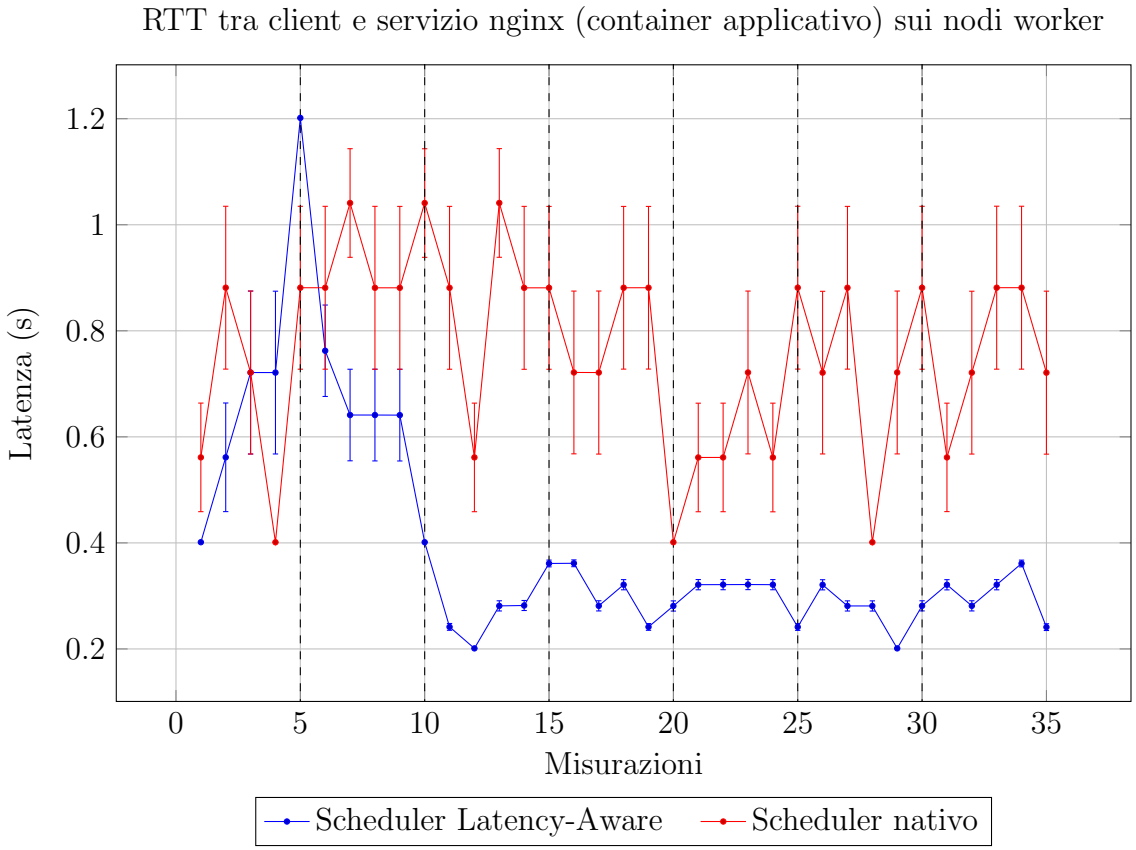


Figura 5.2: *Evoluzione temporale dello scheduling dei pod sui nodi worker del cluster durante il primo test*

5.2.2 Secondo test: scenario intermedio

Il secondo test simula una condizione intermedia, in cui i pod sono distribuiti su nodi con latenze differenti: i pod Target sono collocati sui nodi `mn-worker2` e `mn-worker6`, mentre il pod Probe si trova su `mn-worker5`. Anche in questo caso, il test è stato eseguito in due fasi, come nel test precedente. I risultati ottenuti sono riportati nel grafico seguente:



Anche in questo scenario, lo scheduler latency-aware dimostra di ridurre la latenza tra il client e il servizio nginx rispetto allo scheduler nativo di Kubernetes. In questa situazione intermedia, il pod Probe viene schedulato già **dopo la prima iterazione** sul nodo con la latenza più bassa, `mn-worker`, consentendo al pod Target su `mn-worker5` di prenderne il posto nella **seconda iterazione**. Grazie alla posizione iniziale favorevole del pod Target su `mn-worker2`, la condizione ottimale viene raggiunta in poche iterazioni.

Nell'immagine seguente è mostrata l'evoluzione temporale dello scheduling dei singoli pod sui nodi worker del cluster:

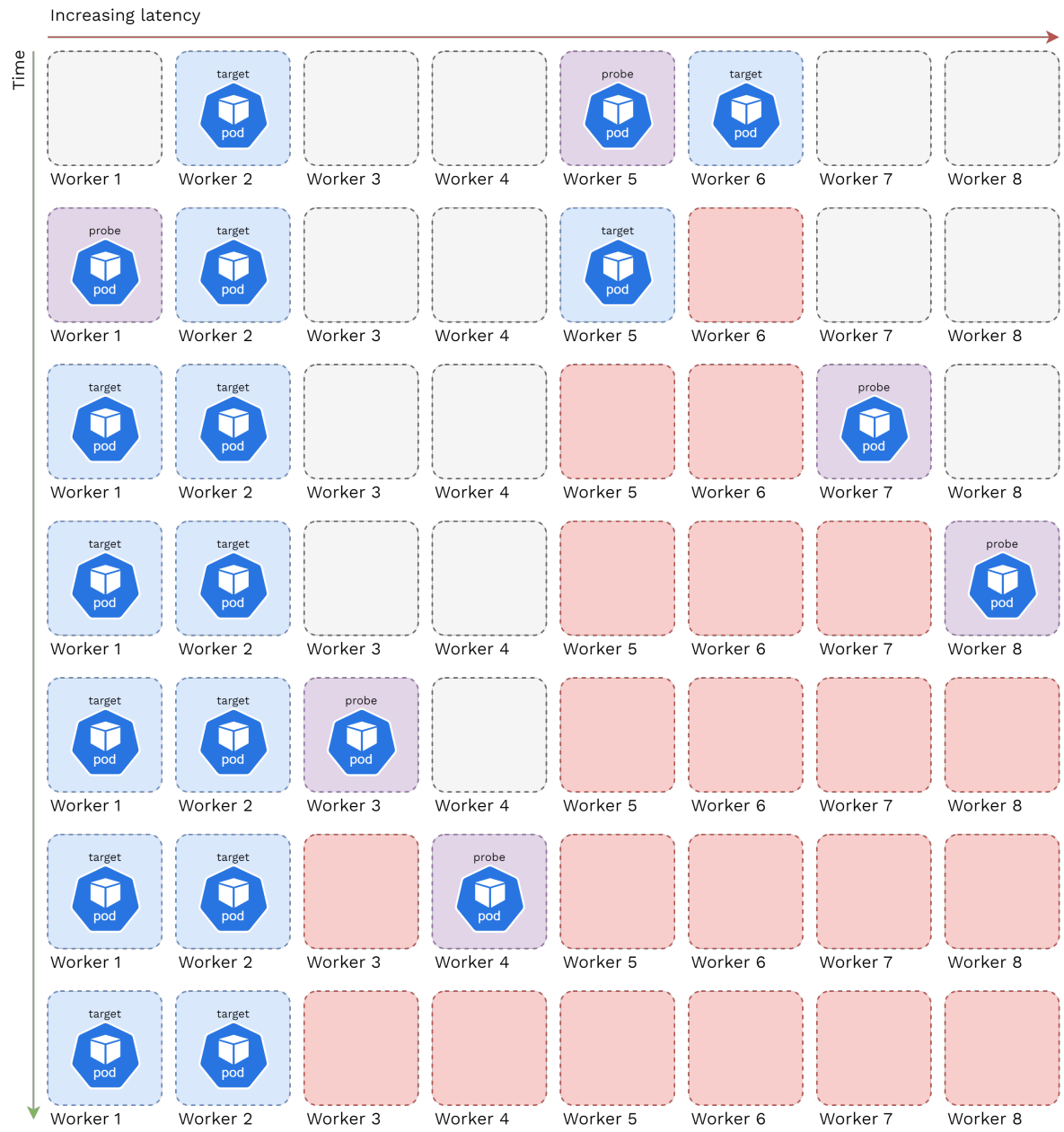


Figura 5.3: *Evoluzione temporale dello scheduling dei pod sui nodi worker del cluster durante il secondo test*

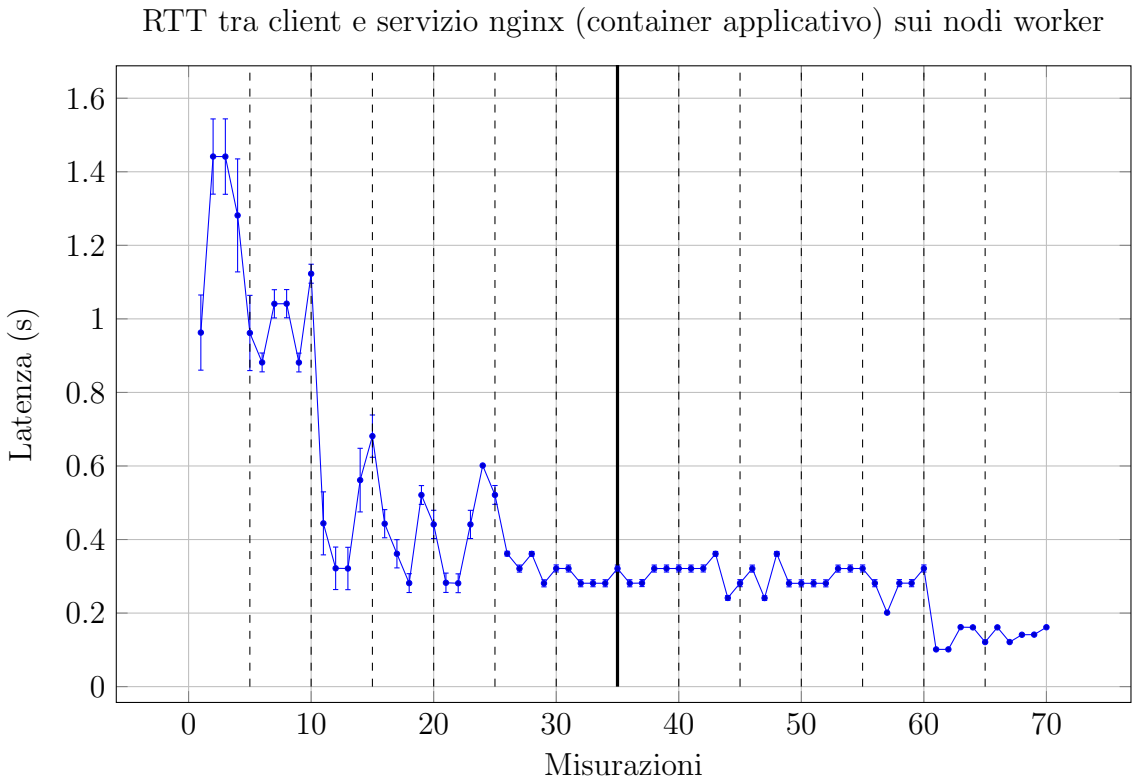
5.2.3 Terzo test: adattamento dinamico

Il terzo test prevede un adattamento dinamico della collocazione dei pod nei nodi del cluster in base alla **variazione della latenza** dei nodi. Questo test è più complesso rispetto ai precedenti e si sviluppa in diverse fasi:

- Inizialmente, i pod Target sono collocati sui nodi `mn-worker4` e `mn-worker8`, mentre il pod Probe è posizionato su `mn-worker6`.

- Viene eseguito un ciclo completo di descheduling e rescheduling dei pod per individuare i nodi con la latenza più bassa, `mn-worker` (100ms) e `mn-worker2` (200ms).
- Dopo un intervallo di 30 secondi dal raggiungimento della condizione ottimale, lo scheduler resetta lo stato per avviare un nuovo ciclo. Contemporaneamente, la latenza del nodo `mn-worker8` viene ridotta manualmente a 50ms per simulare un cambiamento dinamico.
- Un secondo ciclo di descheduling e rescheduling viene eseguito per adattare la collocazione dei pod in base alla nuova latenza del nodo `mn-worker8`.
- Il test si conclude con successo, identificando correttamente i nodi con latenza inferiore: `mn-worker` (100ms) e `mn-worker8` (50ms).

I risultati ottenuti sono riportati nel grafico seguente:



Dal grafico si può notare che, dopo la 35ª misurazione, quando avviene il reset dello scheduler, i tempi di latenza non aumentano, ma anzi **diminuiscono ulteriormente**. Questo comportamento è dovuto alla capacità dello scheduler di adattare dinamicamente

la collocazione dei pod in funzione della latenza dei nodi del cluster. In particolare, il pod Probe viene schedulato e deschedulato iterativamente sui nodi non occupati (`mn-worker` e `mn-worker2`) senza influire sui pod Target, fino a quando viene analizzato il nodo `mn-worker8`, la cui latenza è stata ridotta manualmente a 50ms. In questo caso, il pod Target su `mn-worker2` viene deschedulato e sostituito dal pod Probe su `mn-worker8`, ottimizzando così ulteriormente la latenza complessiva.

Nell'immagine seguente è mostrata l'evoluzione temporale dello scheduling dei singoli pod sui nodi worker del cluster:

Figura 5.4: *Evoluzione temporale dello scheduling dei pod sui nodi worker del cluster durante tutte le fasi del terzo test*

5.3 Valutazioni aggiuntive

L'implementazione dello scheduler custom latency-aware ha dimostrato **risultati positivi**, garantendo un'esperienza fluida e ininterrotta per l'utente finale, senza alcun **disservizio** durante il processo di miglioramento della latenza. Grazie ad una misurazione costante e al meccanismo di descheduling, lo scheduler **ottimizza progressivamente** le prestazioni del sistema, riducendo la latenza end-to-end senza compromettere la qualità del servizio percepita.

Uno degli aspetti chiave dello scheduler proposto è proprio la capacità di **monitorare in tempo reale** la latenza di rete tra i nodi e i dispositivi utente. Questa caratteristica permette di prendere decisioni di schedulazione basate su informazioni aggiornate, migliorando significativamente le **prestazioni** complessive del sistema rispetto a soluzioni che si basano su dati storici o statici.

Nella letteratura, molti scheduler non includono un meccanismo di monitoraggio continuo della latenza. Ad esempio, lo scheduler nativo di Kubernetes utilizza un sistema di filtraggio e punteggio che tiene conto solo delle risorse hardware disponibili, senza considerare in modo dinamico le **condizioni di latenza** della rete. Gli approcci che utilizzano il Kubernetes **FilterPlugin** spesso escludono la latenza dalle metriche di valutazione, affidandosi a metriche come CPU, memoria e preferenze di affinità, ma non tengono conto dell'impatto che la latenza di rete può avere sulle performance delle applicazioni distribuite, specialmente in contesti edge.

Al contrario, lo scheduler proposto non solo raccoglie **metriche in tempo reale**, ma le utilizza anche per valutare e ricalibrare **dinamicamente** l'allocazione dei pod, garantendo che le istanze con le latenze più elevate vengano identificate e deschedulate. Questa capacità di adattamento in tempo reale è fondamentale per ambienti dove la latenza ha un impatto diretto sulla qualità del servizio, come il controllo di processi industriali o l'orchestrazione di microservizi distribuiti in ambienti IoT.

Un altro importante vantaggio riguarda il **ridotto overhead** introdotto sui nodi worker. L'unica componente aggiuntiva che i nodi devono ospitare è il container `lm-server`, che viene utilizzato per misurare la latenza di rete. Questo container è leggero e non impatta significativamente sulle risorse del nodo (CPU, memoria o rete). Ciò permette di mantenere elevata l'**efficienza** del sistema anche su nodi con risorse limitate, rendendo

questa soluzione adatta anche per contesti edge e ambienti distribuiti a risorse ridotte.

Un possibile lavoro futuro potrebbe riguardare l'introduzione di un sistema di misurazione della latenza direttamente sulle richieste che arrivano al servizio, piuttosto che utilizzare richieste apposite inviate dal client di misurazione. Questo approccio consentirebbe di ottenere **dati ancora più realistici** sulla latenza percepita dagli utenti finali, migliorando ulteriormente la precisione delle decisioni di scheduling e riducendo l'overhead aggiuntivo legato al processo di monitoraggio attuale.

Infine, lo scheduler potrebbe essere esteso per gestire in modo più sofisticato scenari **multi-cluster**, consentendo una distribuzione ancora più efficiente delle risorse in ambienti ibridi e geograficamente distribuiti.

In conclusione, lo scheduler sviluppato rappresenta un avanzamento significativo nella gestione delle risorse per ambienti edge, offrendo una **soluzione pratica**, **scalabile** ed **efficiente** che può essere ulteriormente ottimizzata per soddisfare le future esigenze di gestione dei cluster Kubernetes.

Conclusione

La tesi presentata ha dimostrato con successo come sia possibile **migliorare le prestazioni di Kubernetes** in contesti edge mediante lo sviluppo di uno scheduler custom latency-aware. Questo lavoro ha risposto alle sfide poste dall'edge computing, dove la latenza e la disponibilità di risorse limitate rendono inefficace l'uso di scheduler tradizionali, sviluppati principalmente per ambienti cloud centralizzati. Grazie all'implementazione di un plugin che tiene conto della latenza di rete nella distribuzione dei pod, è stato possibile ottimizzare il posizionamento dei carichi di lavoro sui nodi del cluster, **riducendo i tempi di risposta** e migliorando la qualità del servizio percepita dagli utenti finali.

Uno degli aspetti chiave di questa implementazione è stato il meccanismo di descheduling, che ha permesso di **migliorare gradualmente** la latenza end-to-end senza introdurre disservizi per l'utente finale. Durante il ciclo iterativo di miglioramento, l'utente non ha subito alcun tipo di interruzione del servizio, in quanto il sistema ha continuato a funzionare regolarmente, ottimizzando la distribuzione dei pod in background. Questo risultato è stato reso possibile grazie a un'**architettura modulare** e **flessibile** che permette di raccogliere in tempo reale i dati di latenza e di adattarsi dinamicamente ai cambiamenti nelle condizioni di rete e carico del sistema.

Un altro elemento rilevante è stato l'**overhead minimo** introdotto sui nodi worker. L'unico componente aggiuntivo richiesto è stato l'lm-server, un container leggero che consente la raccolta delle misurazioni di latenza senza impattare significativamente sull'utilizzo delle risorse del nodo. Questo aspetto rende il sistema **adatto per ambienti a risorse limitate**, come spesso accade nel contesto edge, garantendo un equilibrio ottimale tra prestazioni e consumo di risorse.

Le simulazioni e i test sperimentali condotti hanno dimostrato l'**efficacia del sistema**, evidenziando come l'ottimizzazione iterativa della latenza migliori significativamente le prestazioni rispetto allo scheduler nativo di Kubernetes. Negli senari di testi lo scheduler latency-aware ha ridotto sensibilmente i tempi di risposta, offrendo un miglioramento tangibile della qualità del servizio.

Tuttavia, il lavoro ha aperto anche a nuove sfide e possibili **sviluppi futuri**. Un

aspetto che meriterebbe ulteriore approfondimento è la possibilità di misurare la latenza direttamente sulle richieste reali degli utenti finali, piuttosto che affidarsi a richieste apposite inviate dal client di misurazione. Questo permetterebbe di ottenere **dati ancora più accurati** e realistici, affinando ulteriormente il processo di ottimizzazione della latenza.

Un'altra direzione interessante per futuri sviluppi riguarda l'ottimizzazione del processo di schedulazione in ambienti **multi-cluster** o distribuiti geograficamente. In questi contesti, il posizionamento dei carichi di lavoro diventa ancora più critico, e la possibilità di integrare lo scheduler custom con soluzioni di orchestrazione multi-cluster potrebbe portare a risultati ancora più rilevanti in termini di riduzione della latenza.

In conclusione, questa tesi ha presentato una soluzione pratica e innovativa per la gestione della latenza in ambienti edge attraverso lo sviluppo di uno **scheduler latency-aware**. I risultati ottenuti dimostrano che è possibile migliorare sensibilmente le prestazioni di Kubernetes in **contesti distribuiti** e ad **alta sensibilità** alla latenza, offrendo una base solida per futuri sviluppi e ottimizzazioni.

Capitolo A

Appendice

```go
// GetNodeIPs recupera e restituisce i nomi dei nodi e i loro indirizzi IP
func GetNodeIPs(kubeconfig *string, labelSelector *string, namespace *string) ([]NodeInfo, error) {

	config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
	if err != nil {
		log.Fatalf("Error building kubeconfig: %s", err.Error())
	}

	// Crea un clientset Kubernetes
	clientset, err := kubernetes.NewForConfig(config)
	if err != nil {
		log.Fatalf("Error creating Kubernetes client: %s", err.Error())
	}

	// Ottieni i pod con l'etichetta specificata
	pods, err := clientset.CoreV1().Pods(*namespace).List(context.TODO(), metav1.ListOptions{
		LabelSelector: *labelSelector,
	})
	if err != nil {
		log.Fatalf("Error listing pods: %s", err.Error())
	}

	fmt.Printf("Found %d pods\n", len(pods.Items))

	// Mappa dei nodi per evitare duplicati
	var nodeInfos []NodeInfo
	for _, pod := range pods.Items {
		fmt.Printf("Pod: %s\n", pod.Name)

		nodeName := pod.Spec.NodeName
		// Ottieni i dettagli del nodo
		node, err := clientset.CoreV1().Nodes().Get(context.TODO(), nodeName,
		↪  metav1.GetOptions{})
		if err != nil {
			log.Printf("error getting node %s: %s", nodeName, err)
			continue
		}

		nodeInfo := NodeInfo{Name: nodeName, PodName: pod.Name, Namespace: pod.Namespace}
		for _, address := range node.Status.Addresses {
			if address.Type == corev1.NodeInternalIP {
				nodeInfo.InternalIP = address.Address
			} else if address.Type == corev1.NodeExternalIP {
				nodeInfo.ExternalIP = address.Address
			}
		}
		nodeInfos = append(nodeInfos, nodeInfo)
	}

	return nodeInfos, nil
}
```

Codice A.1: *Funzione per recuperare i nodi con pod Probe o Target in esecuzione*

```go
func (la *LatencyAware) Filter(ctx context.Context,
    cycleState *framework.CycleState,
    pod *corev1.Pod,
    nodeInfo *framework.NodeInfo) *framework.Status {

    appLabel := pod.Labels["app"]

    // If the pod is neither a probe nor a target, it can be scheduled anywhere
    if appLabel != ProbeAppLabel && appLabel != TargetAppLabel {
        return framework.NewStatus(framework.Success,
            fmt.Sprintf("Pod %v can be scheduled in %v", pod.Name, nodeInfo.Node().Name))
    }

    // If
    //  - the pod is a target
    //  - there are nodes with a probe and not a target
    // then the pod can be scheduled only in the nodes with a probe and not a target
    //                because they are (after the initial transient phase) better nodes than the one visited by the target pod descheduled
    if appLabel == TargetAppLabel && len(la.nodesWithProbeAndNotTarget.nodes) > 0 {
        if la.nodesWithProbeAndNotTarget.IsVisited(nodeInfo.Node().Name) {
            klog.Infof("[LatencyAware] Pod %v CAN be scheduled in \"probe\" only node: %v", pod.Name, nodeInfo.Node().Name)
            return framework.NewStatus(framework.Success,
                fmt.Sprintf("Pod %v can be scheduled in %v", pod.Name, nodeInfo.Node().Name))
        } else {
            klog.Infof("[LatencyAware] Pod %v CANNOT be scheduled in the node %v because there are nodes with only probe", pod.Name, nodeInfo.Node().Name)
            return framework.NewStatus(framework.Unschedulable,
                fmt.Sprintf("Pod %v cannot be scheduled in %v", pod.Name, nodeInfo.Node().Name))
        }
    }

    // If the node is not yet visited by the probe pod
    if !la.probeVisitedNodes.IsVisited(nodeInfo.Node().Name) {
        klog.Infof("[LatencyAware] Pod %v CAN be scheduled in: %v", pod.Name, nodeInfo.Node().Name)
        return framework.NewStatus(framework.Success,
            fmt.Sprintf("Pod %v can be scheduled in %v", pod.Name, nodeInfo.Node().Name))
    }

    // If the node is already visited by the probe pod it cannot be scheduled
    klog.Infof("[LatencyAware] Pod %v CANNOT be scheduled in: %v", pod.Name, nodeInfo.Node().Name)
    return framework.NewStatus(framework.Unschedulable,
        fmt.Sprintf("Pod %v cannot be scheduled in %v", pod.Name, nodeInfo.Node().Name))
}
```

Codice A.2: *Fase di Filter dello scheduling-plugin*

```go
func (la *LatencyAware) Reserve(ctx context.Context, state *framework.CycleState, pod *corev1.Pod,
    nodeName string) *framework.Status {
    appLabel := pod.Labels["app"]

    klog.Infof("[LatencyAware] RESERVE: Pod %v visited %v", pod.Name, nodeName)

    // If the pod is neither a probe nor a target, it can be scheduled anywhere
    if appLabel != ProbeAppLabel && appLabel != TargetAppLabel {
        return framework.NewStatus(framework.Success,
            fmt.Sprintf("Pod %v can be scheduled in %v", pod.Name, nodeName))
    }

    // If the node is not yet visited by the pod
    if la.probeVisitedNodes.SetVisitedIfNot(nodeName) {
        klog.Infof("[LatencyAware] Pod %v visited %v", pod.Name, nodeName)

        // If it's a probe only pod, the node is inserted in the list of nodes with a probe and not a
        target
        if appLabel == ProbeAppLabel {
            klog.Infof("[LatencyAware] Probe pod %v reserved %v", pod.Name, nodeName)
            la.nodesWithProbeAndNotTarget.SetVisited(nodeName)
        }

        return framework.NewStatus(framework.Success)
    }

    // If the pod is a target
    if appLabel == TargetAppLabel {
        // If the node was visited by a probe pod and not a target pod
        if la.nodesWithProbeAndNotTarget.SetUnvisited(nodeName) {
            // code here is executed only if the node was in the list of nodes with a probe and not a
            target

            // Prevents that two or more targets get scheduled in the node
            //   only one target can be scheduled in the node visited by the probe
            // If this is not done, the filter function will allow the scheduling of the target pod
            state.Write(targetOnProbeNodeStateKey, &TargetOnProbeNodeData{value: true})
        }
    }

    // If the node is already visited by the pod
    //    this happens when trying to schedule a target pod on a node already visited by a probe pod
    return framework.NewStatus(framework.Success)
}
```

Codice A.3: *Fase di Reserve dello scheduling-plugin*

```go
func (la *LatencyAware) Unreserve(ctx context.Context, state *framework.CycleState, pod *corev1.Pod,
    nodeName string) {
    appLabel := pod.Labels["app"]

    if appLabel != ProbeAppLabel && appLabel != TargetAppLabel {
        return
    }

    klog.Infof("[LatencyAware] UNRESERVE: Pod %v visited %v", pod.Name, nodeName)

    // Sets the node as unvisited
    la.probeVisitedNodes.SetUnvisited(nodeName)

    if appLabel == TargetAppLabel {
        // Retrieve the item from the CycleState
        raw, err := state.Read(targetOnProbeNodeStateKey)
        if err != nil {
            return
        }

        // Assert the type to your custom data type
        _, ok := raw.(*TargetOnProbeNodeData)
        if !ok {
            return
        }

        // If the node was replacing the probe node, resets the node as visited only by the probe
        la.nodesWithProbeAndNotTarget.SetVisited(nodeName)
    } else if appLabel == ProbeAppLabel {
        la.nodesWithProbeAndNotTarget.SetUnvisited(nodeName)
    }
}
```

Codice A.4: *Fase di Unreserve dello scheduling-plugin*

```go
func (la *LatencyAware) PostBind(ctx context.Context, state *framework.CycleState, pod *corev1.Pod, nodeName string) {
    klog.Infof("[LatencyAware] Pod %v has been scheduled in: %v (DT: %v)", pod.Name, nodeName, pod.DeletionTimestamp)

    appLabel := pod.Labels["app"]

    // If the pod is a target
    if appLabel == TargetAppLabel {
        // If the node was visited by a probe pod and not a target pod
        //        This can be done because the "Reserve" function is executed before the "PostBind" function
        //  the function inserts a state in the CycleState to signal that the target pod is replacing the probe pod

        // Retrieve the item from the CycleState
        raw, err := state.Read(targetOnProbeNodeStateKey)
        if err == nil {

            // Assert the type to your custom data type
            _, ok := raw.(*TargetOnProbeNodeData)
            if ok {
                // Find and deschedule the ProbeAppLabel pod on the same node

                state.Delete(targetOnProbeNodeStateKey)

                // Get the list of pods with the probe label (containing only probe)
                podList, err := la.handle.ClientSet().CoreV1().Pods(pod.Namespace).List(ctx, metav1.ListOptions{
                    LabelSelector: labels.SelectorFromSet(labels.Set{
                        "app": ProbeAppLabel,
                    }).String(),
                })
                if err != nil {
                    klog.Errorf("[LatencyAware] Error listing pods: %v", err)
                    return
                }

                // For each pod with the probe label
                for _, p := range podList.Items {
                    // If the pod is scheduled on the same node
                    if p.Spec.NodeName == nodeName {
                        klog.Infof("[LatencyAware] Descheduling ProbeAppLabel pod %v on node %v", p.Name, nodeName)
                        // Deschedule the probe pod on the same node
                        err := la.handle.ClientSet().CoreV1().Pods(p.Namespace).Delete(context.TODO(), p.Name, metav1.DeleteOptions{})
                        if err != nil {
                            klog.Errorf("[LatencyAware] Error deleting pod: %v", err)
                        }
                    }
                }
            }
        }
    }

    // Start the timeout to reset the visited nodes
    la.startTimeout(ctx)
}
```

Codice A.5: *Fase di PostBind dello scheduling-plugin*

```
mqttclient.ReadMessages(func(_ mqtt.Client, message mqtt.Message) {
    fmt.Printf("[MQTT] Received message: %s on topic: %s\n", message.Payload(), message.Topic())

    // Parse the message
    // The message should be in the format: "Pod: %s\Namespace: %s\nRTT: %s\nTimestamp: %s"
    messageStr := string(message.Payload())
    var podName, namespace, rtt, timestamp string
    _, err := fmt.Sscanf(messageStr, "Pod: %s\nNamespace: %s\nRTT: %s\nTimestamp: %s", &podName,
    ↪  &namespace, &rtt, &timestamp)
    if err != nil {
        fmt.Println("[MQTT] Error parsing message:", err)
        return
    }

    //convert rtt into time duration (format is 1.234567ms)
    rttDuration, err := time.ParseDuration(rtt)
    if err != nil {
        fmt.Println("[MQTT] Error parsing rtt:", err)
        return
    }

    // convert timestamp into time.Time (format is 2024-08-18T14:08:11Z)
    timestampTime, err := time.Parse(time.RFC3339, timestamp)
    if err != nil {
        fmt.Println("[MQTT] Error parsing timestamp:", err)
        return
    }

    // Create a new measurement
    measurement := Measurement{
        PodName:   podName,
        Namespace: namespace,
        RTT:       rttDuration,
        Timestamp: timestampTime,
    }

    measurementsMutex.Lock()
    defer measurementsMutex.Unlock()

    // Check if measurements already contains an entry with the same podname
    for i := 0; i < len(measurements); i++ {
        if measurements[i].PodName == podName {
            // Replace the existing entry with the new measurement
            measurements[i] = measurement
            return
        }
    }

    // Append the measurement to the measurements slice
    measurements = append(measurements, measurement)

    // Total measurements
    fmt.Println("[MQTT] Total measurements:", len(measurements))
})
```

Codice A.6: *Callback messageHandler del modulo MQTT del Descheduler*

```
descheduler.Run(func(d *Descheduler) {
    measurementsMutex.Lock()
    defer measurementsMutex.Unlock()

    if len(measurements) >= measurementsCount {
        if d.IsStable() {
            fmt.Println("[Descheduler] Deployment is currently stable")
            return
        }

        // get the measurement with the biggest RTT
        // Initialize the index of the measurement with the biggest RTT
        maxRTTIndex := 0

        // Iterate over the measurements slice
        for i := 1; i < len(measurements); i++ {
            // Check if the current measurement has a bigger RTT than the measurement with the biggest
            ↪  RTT
            if measurements[i].RTT > measurements[maxRTTIndex].RTT {
                // Update the index of the measurement with the biggest RTT
                maxRTTIndex = i
            } else if measurements[i].RTT == measurements[maxRTTIndex].RTT {
                // If the RTTs are equal, check the timestamps: the oldest measurement should be
                ↪  descheduled
                if measurements[i].Timestamp.Before(measurements[maxRTTIndex].Timestamp) {
                    // Update the index of the measurement with the biggest RTT
                    maxRTTIndex = i
                }
            }
        }

        // Get the measurement with the biggest RTT
        maxRTTMeasurement := measurements[maxRTTIndex]

        fmt.Println("[Descheduler] Descheduling pod:", maxRTTMeasurement.PodName)

        // Deschedule the podname
        //   if a target is deleted then also the measurement for the probe should be deleted (becuase the
        ↪  scheduler will deschedule it to replace it with a new target)
        err := d.DeschedulePod(maxRTTMeasurement.PodName, maxRTTMeasurement.Namespace)
        if err != nil {
            fmt.Println("[Descheduler] Error descheduling pod:", err)
        }
    }
}, func(d *Descheduler, pod *corev1.Pod) {
    measurementsMutex.Lock()
    defer measurementsMutex.Unlock()

    // Delete the entry from measurements
    for i := 0; i < len(measurements); i++ {
        if pod.Name == measurements[i].PodName && pod.Namespace == measurements[i].Namespace {
            fmt.Println("[Informer] Pod descheduled:", pod.Name)
            measurements = append(measurements[:i], measurements[i+1:]...)
            return
        }
    }
})
```

Codice A.7: *Callback descheduleIntervalFunction e podDescheduledFunction del modulo Controller del Descheduler*

Bibliografia

[1] Carlo Centofanti, W. Tiberti, A. Marotta, F. Graziosi e D. Cassioli. "Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge". In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023.

[2] Kubernetes. *Kube-Scheduler*. `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`.

[3] Kubernetes. *Kubernetes API*. `https://kubernetes.io/docs/concepts/overview/kubernetes-api/`.

[4] Kubernetes. *Kubernetes Components*. `https://kubernetes.io/docs/concepts/overview/components/`.

[5] Kubernetes. *Scheduler Configuration*. `https://kubernetes.io/docs/reference/scheduling/config/`.

[6] Kubernetes. *Scheduling Framework*. `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/`.

[7] Kubernetes. *What is Kubernetes?* `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`.

[8] The Kubernetes Authors. *Kind*. `https://kind.sigs.k8s.io/`.