UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

MACROAREA DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN
INGEGNERIA INFORMATICA

TESI DI LAUREA MAGISTRALE

Distribuzione delle Richieste in Sistemi Function-as-a-Service Multi-Architettura

Relatori:
Prof.ssa Valeria Cardellini
Dott. Gabriele Russo Russo

Laureando:
Filippo Muscherà
Matricola: 0338276

Anno Accademico 2024/2025

Abstract

Il paradigma Function-as-a-Service (FaaS) si sta affermando come modello di riferimento per la computazione serverless sia per il Cloud Computing che per l'Edge Computing, coprendo l'intero spettro del cosiddetto Edge-Cloud Continuum. Il FaaS consente di eseguire funzioni *on demand*, delegando al provider la gestione e il setup dell'infrastruttura sottostante. Questo modello è quindi particolarmente adatto anche per l'Edge Computing, dove è necessario elaborare i dati in prossimità dell'utente o della sorgente dei dati, per garantire una risposta a bassa latenza e ridurre il traffico di rete e il carico sui data center cloud.

Nell'Edge, i nodi di calcolo presentano spesso architetture hardware differenti, con una forte presenza di processori ARM, caratterizzati da un'elevata efficienza energetica, accanto ai più tradizionali processori x86. In letteratura, diversi studi hanno analizzato l'impatto di questa eterogeneità nel contesto serverless, evidenziando i vantaggi e gli svantaggi offerti dalle diverse architetture a seconda dello specifico carico di lavoro. Questi studi mostrano infatti come l'eterogeneità possa essere sfruttata per ottimizzare i tempi di risposta e i consumi energetici, a fronte però di una maggior complessità a livello di orchestrazione e gestione delle richieste. Tuttavia, nonostante queste evidenze, la maggior parte dei framework FaaS esistenti non offre meccanismi dinamici e politiche adattive per sfruttare appieno architetture multiple. Questa mancanza rappresenta un ostacolo per operare in maniera ottimale nell'Edge-Cloud Continuum, dove è fondamentale utilizzare al meglio tutte le risorse disponibili per garantire la massima adattabilità e scalabilità del sistema.

Partendo da queste premesse, questo lavoro di tesi propone e implementa un framework per l'esecuzione e il bilanciamento multi-architettura intelligente, adottando come caso di studio e base di sviluppo la piattaforma Serverledge, un framework open-source per il FaaS nel continuum Edge-Cloud. Il lavoro si articola in diverse fasi. Innanzitutto, estende l'infrastruttura ridefinendo il protocollo di registrazione dei nodi e le strutture dati interne per includere informazioni sull'architettura hardware, adeguando parallelamente le immagini dei container e i runtime per garantirne la compatibilità con ARM. Risolto il problema di rendere il sistema *compatibile* con architetture eterogenee, il focus si è spostato sul rendere il sistema *intelligente* rispetto alla scelta del nodo di esecuzione, con l'obiettivo di massimizzare le prestazioni complessive del cluster.

Per realizzare quest'ultima fase, è stato fondamentale lo sviluppo di un load balancer *architecture-aware* capace di distribuire le richieste d'invocazione in modo mirato su un cluster ibrido. A differenza delle politiche statiche tradizionali, il sistema proposto utilizza algoritmi di *reinforcement learning*, nello specifico *multi-armed bandit* (MAB), per apprendere dinamicamente quale architettura offre le prestazioni migliori per ogni specifica funzione.

Questi algoritmi sono pensati per cercare un equilibrio tra l'esplorazione di nuove opzioni e lo sfruttamento delle conoscenze acquisite, fino al raggiungimento di una convergenza verso l'opzione migliore. Da un lato, quindi, il sistema può inviare una richiesta a un nodo ARM per valutarne le prestazioni; dall'altro, può continuare a indirizzare le richieste verso nodi x86 qualora abbia appreso che questi offrono risultati migliori per quella specifica operazione. Le decisioni del load balancer sono quindi guidate da un processo di apprendimento continuo che permette al sistema di adattarsi in tempo reale alle condizioni operative del cluster, tenendo conto di fattori come l'utilizzazione della memoria e le prestazioni storiche delle funzioni su ciascuna architettura.

L'implementazione include sia politiche context-free come UCB1, sia MAB contestuali come LinUCB, che sfruttano le metriche relative all'utilizzo della memoria in tempo reale per affinare il loro processo decisionale.

La logica decisionale è stata integrata con un meccanismo di consistent hashing per distribuire uniformemente il carico tra i nodi fisici. Si utilizza un anello di hash distinto per ogni architettura, con ogni nodo replicato più volte all'interno del proprio anello. In questo modo, quando una richiesta arriva, il load balancer può scegliere l'anello più appropriato in base alla politica di reinforcement learning e selezionare il nodo specifico tramite consistent hashing. Questo approccio permette di indirizzare le richieste verso l'architettura più performante per ciascuna funzione, massimizzando l'affinità tra richieste e nodi e favorendo il riutilizzo di ambienti di esecuzione già attivi.

La validazione è stata condotta in maniera sperimentale su cluster ibridi composti da nodi x86 e ARM, confrontando le performance del sistema proposto con quelle di un approccio statico. I risultati evidenziano come la soluzione implementata non solo renda possibile l'esecuzione del framework su infrastrutture eterogenee, ma migliori le prestazioni complessive del sistema grazie alla capacità di adattarsi autonomamente alle dinamiche del cluster. In particolare, il load balancer, integrato con meccanismi di consistent hashing e politiche di reinforcement learning, riesce a ottimizzare la distribuzione del carico tra architetture diverse e a massimizzare il riutilizzo di ambienti di esecuzione già attivi, riducendo la latenza e il numero di cold start. Questi risultati confermano l'efficacia dell'approccio proposto nel garantire sia flessibilità che performance in scenari Edge eterogenei e dinamici.

Indice

Capitolo 1

Introduzione

1.1 Function-as-a-Service e Edge Cloud

In questa prima sezione si presenta il contesto di riferimento oggetto di studio, partendo dall'introduzione al paradigma del Serverless e del Function-as-a-Service (FaaS), e successivamente analizzando come il Cloud Computing stia evolvendo anche verso l'Edge Computing. Questo comporta una crescente eterogeneità nell'infrastruttura hardware sottostante, con la presenza di cluster di nodi ibridi, basati sia su architetture x86 che ARM [1].

1.1.1 Il FaaS

Il Serverless Computing rappresenta un'evoluzione del paradigma di Cloud Computing che astrae la gestione dell'infrastruttura sottostante, permettendo agli sviluppatori di concentrarsi esclusivamente sulla logica applicativa. All'interno di questo modello si colloca il Function-as-a-Service (FaaS), un paradigma che permette di eseguire singole funzioni in risposta a eventi specifici. Infatti, a differenza di quanto succede con i modelli tradizionali come IaaS[1] o PaaS[2], il FaaS introduce nuovi modelli di costo basati sul consumo, consentendo agli utenti di godere di scalabilità automatica, senza dover gestire direttamente l'infrastruttura sottostante [2].

[1]Infrastructure-as-a-Service: fornisce accesso on demand a risorse come server, spazio di archiviazione, networking e virtualizzazione.

[2]Platform-as-a-Service: offre una piattaforma cloud flessibile e scalabile per sviluppare, eseguire il deployment, eseguire e gestire app.

Il FaaS rappresenta quindi un'evoluzione rispetto ai modelli di servizio classici del Cloud Computing, perché sposta verso il Cloud provider tutte queste responsabilità di gestione dell'infrastruttura: la configurazione e il dimensionamento delle virtual machine (VM), il bilanciamento delle richieste in arrivo e l'implementazione dei meccanismi di sicurezza. In questo modello, le funzioni tendono a essere *stateless* e vengono eseguite *on-demand* in risposta a determinati eventi, come per esempio una richiesta HTTP o un messaggio da un nodo IoT[3]. Si tratta di funzioni serverless, ovvero unità di codice eseguite nel cloud senza la necessità di gestire direttamente server o infrastruttura, poiché l'ambiente di esecuzione viene avviato e scalato automaticamente dal provider. Quando questo si verifica, le funzioni vengono eseguite per la durata necessaria al calcolo e al loro completamento tutte le risorse utilizzate verranno distrutte o allocate per altre funzioni. Si configura quindi un vero e proprio modello di *pay-per-execution*, dove i costi sono calcolati con precisione al millisecondo solo durante l'effettiva esecuzione del codice, eliminando tutti quei costi legati alle risorse inattive tipici delle architetture basate su virtual machine o container sempre attivi [3].

I cloud provider, per eseguire funzioni on-demand e in ambienti isolati, utilizzano diverse tecniche. La più classica è quella dell'utilizzo di container *Docker*, in cui si garantisce isolamento tramite i *namespaces* e *cgroups* di Linux, pur condividendo tutti lo stesso kernel. Altre soluzioni più innovative sono per esempio le MicroVM di Amazon Web Services (AWS) Lambda [4], o i *sandboxed container* di Google [5]. Le prime consentono di avere kernel dedicati mantenendo le virtual machine estremamente leggere, mentre i secondi intercettano le *syscall* per creare isolamento senza richiedere un *hypervisor* completo.

[3]Internet of Things: una rete di oggetti ("things") dotati di sensori, software e tecnologie che si connettono e scambiano dati con altri dispositivi e sistemi tramite Internet.

Tuttavia, l'allocazione e deallocazione così rapida e aggressiva di risorse, se da un lato consente l'utilizzo di questi modelli di costo, dall'altro introduce una nuova problematica con cui scontrarsi: i *cold start*.

Infatti, se tutte le risorse allocate per una funzione vengono eliminate dal cloud provider, la prima richiesta in arrivo dopo un periodo di inattività richiede l'inizializzazione da zero dell'ambiente di esecuzione. Questo può voler dire, per esempio, dover creare il container per la funzione, avviare il runtime e copiare il codice al suo interno.

Si possono avere quindi due casi ben distinti, come illustrato in Figura 1:

- **Cold Start**: in cui si ha una latenza introdotta dalla creazione dell'intero ambiente di esecuzione. In ambienti Edge con risorse limitate, questo tempo può anche superare il tempo di esecuzione effettivo della funzione [6], [7].
- **Warm Start**: in cui si ha l'esecuzione su un'istanza *idle* o già attiva, dove la latenza è ridotta alla sola elaborazione della richiesta.

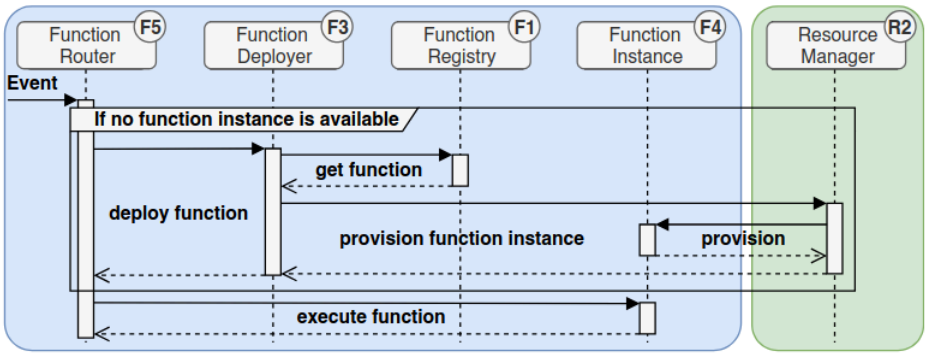

Figura 1: Operazioni da compiere in caso di cold e warm start all'interno di un sistema FaaS [8]

Tecniche come il *pre-warming* sono spesso impiegate per mitigare questo problema, andando a preparare un ambiente di esecuzione per la funzione prima che questo sia effettivamente richiesto. Questo rappresenta comunque un *trade-off* tra diminuzione della latenza e aumento di utilizzo delle risorse.

Nonostante queste sfide, oggi, grazie alle sue caratteristiche, il paradigma FaaS ha un'adozione diffusa in moltissimi settori. Per esempio, viene utilizzato per

applicazioni web, analisi di flussi di dati, Machine Learning, monitoraggio di sistemi e pipeline DevOps, IoT e calcolo scientifico [9].

1.1.2 Edge Computing

Per Edge Computing si intende quel paradigma che sposta l'elaborazione e l'archiviazione dei dati verso il bordo (*edge*) della rete, spostando la computazione dai data center centralizzati verso la "periferia" della rete, in prossimità delle sorgenti dei dati e delle richieste [10]. È importante notare come l'Edge Computing non sostituisca il Cloud Computing, ma lo estenda. Si crea così il Cloud-Edge *Continuum* (o semplicemente Cloud Continuum), ovvero un'integrazione fluida tra questi due paradigmi infrastrutturali: si va a creare un ecosistema continuo in cui le applicazioni e i servizi possono operare [11]. Infatti, in base alle proprie esigenze di latenza e potenza di calcolo, ogni applicazione potrà eseguire più vicina all'edge o al data center.

Eseguire vicino all'Edge della rete è utile in tutti quegli scenari *latency-sensitive* in cui è più importante avere latenze molto basse piuttosto che capacità computazionali molto elevate. Infatti, l'idea è quella di avere nell'Edge nodi meno potenti ma più numerosi rispetto a quelli che si hanno a disposizione nei data center. Uno scenario tipico può essere, per esempio, quello dell'IoT. In questo contesto si possono avere moltissimi dispositivi che producono un'elevata quantità di dati. Inviarli ed elaborarli tutti su dei data center centrali introdurrebbe una latenza elevata, oltre a richiedere un'elevata capacità di trasmissione di dati sulla rete. Per questo è necessario avere nodi in grado di elaborare questi dati quanto più vicino possibile alla sorgente dei dati stessi. Con la rapidissima diffusione dell'IA, si sta cercando di utilizzare lo stesso approccio anche per l'inferenza dei modelli, introducendo nell'Edge della rete dispositivi con hardware dedicato a questi scopi. Allo stesso tempo esistono comunque moltissimi scenari in cui è invece necessaria molta potenza di calcolo, che solo i data center centrali possono garantire, come nel caso di *Machine Learning* intensivo o di *Big Data Analysis*. Per questo, Cloud ed Edge

non rappresentano due entità separate, ma componenti di un ecosistema unico che bilancia potenza, latenza e vicinanza ai dati.

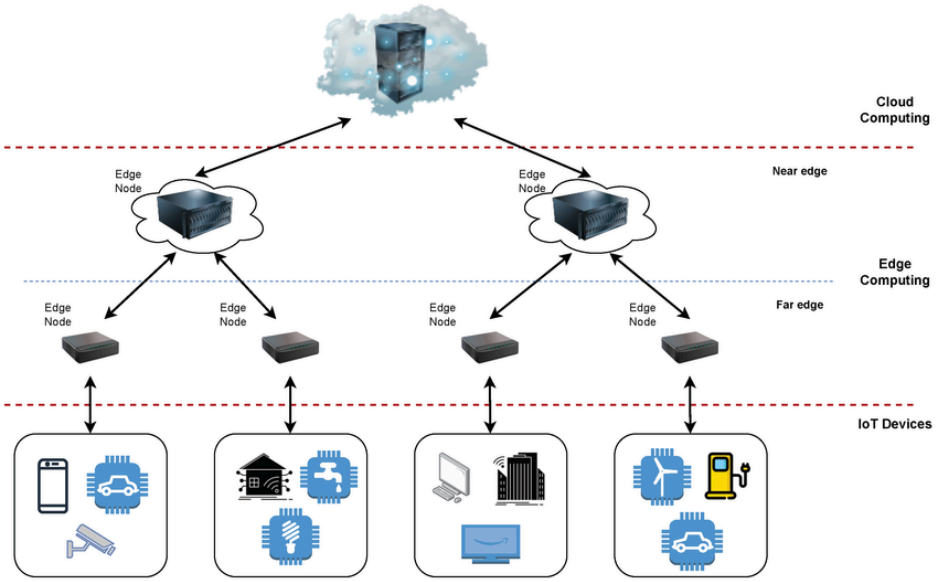

Figura 2: Il Cloud-Edge Continuum [12]

1.2 Eterogeneità hardware

Questa spinta nello spostare parte della computazione dal Cloud all'Edge introduce anche una nuova sfida a livello infrastrutturale: l'eterogeneità dell'hardware. Infatti, mentre nei data center Cloud le architetture dei processori sono spesso omogenee e dominate dall'architettura x86-64[4], nell'Edge si ha un'ampia gamma di dispositivi diversi [1]. Qui i vincoli stringenti su consumi energetici e dissipazione termica favoriscono l'adozione di processori con architetture RISC[5] basati prevalentemente su ARM. Per esempio, si possono trovare dispositivi come Raspberry Pi, NVIDIA Jetson o Google Coral affiancati a dispositivi equipaggiati con CPU x86 più classiche.

Di conseguenza, l'infrastruttura hardware nell'Edge è inevitabilmente ibrida, con nodi ad alte prestazioni affiancati a nodi ad alta efficienza. In un contesto

[4]Processori Intel e AMD. A volte ci si riferisce a questa architettura più semplicemente come "x86" o come "amd64"

[5]Reduced Instruction Set Computer. In contrasto con il CISC, il Complex Instruction Set che caratterizza le CPU x86-64.

di questo tipo l'eterogeneità dell'hardware non è più un'eccezione da dover gestire, ma una caratteristica intrinseca del sistema, da tenere in considerazione e da sfruttare. I framework FaaS, soprattutto se pensati per essere utilizzati nell'Edge-Cloud Continuum, hanno quindi la necessità di poter operare su diverse architetture. Questo è necessario sia per poter essere compatibili con questo tipo di ecosistemi, sia per trarne un vantaggio in termini di prestazioni, consumi energetici e costi.

1.2.1 Affinità Architetturali

Nell'ambito di un framework FaaS pensato per lavorare nel Cloud Continuum, destinato a operare con *cluster* di nodi ibridi (x86 e ARM), bisogna tenere conto anche delle affinità architetturali delle funzioni. Infatti, l'efficienza di un'architettura rispetto a una funzione non dipende solamente dalla potenza di calcolo grezza. Processori con architetture diverse hanno un instruction set diverso, micro-architettura diversa, approcci diversi al *multi-threading* e ottimizzazioni diverse. Tutto questo porta a quella che si può definire come affinità architetturale: le prestazioni offerte da processori con architetture diverse variano significativamente in base alla natura delle istruzioni eseguite. Questo significa che, potenzialmente, il tempo di esecuzione di una funzione varierà in maniera non trascurabile in base al tipo di architettura su cui questa è eseguita. È importante notare che funzioni diverse avranno affinità con architetture diverse, e predire a priori l'architettura ideale per una funzione risulta estremamente complesso [13], [14].

Quindi, per un framework FaaS sarà anche importante andare a scegliere in maniera intelligente su quale architettura eseguire ogni funzione al fine di massimizzare le prestazioni. Bisogna anche tenere conto che l'algoritmo responsabile di queste scelte potrebbe essere eseguito sull'Edge della rete, e quindi da nodi con capacità computazionale limitata. Per evitare latenze che ne annullerebbero i benefici, è necessario usare algoritmi e modelli abbastanza leggeri da non introdurre ritardi percepibili nell'esecuzione delle funzioni.

1.3 Compatibilità Multi-Architettura

Come accennato in precedenza, prima di poter passare alla fase in cui si *ottimizza* il sistema FaaS rispetto alle varie architetture supportate, bisogna rendere il sistema stesso *compatibile* con queste architetture.

Rendere il codice del framework FaaS compatibile con due architetture, come x86 e ARM, non costituisce la sfida principale. Molti linguaggi di programmazione moderni supportano infatti la compilazione cross-architettura, e la loro tool-chain facilita la generazione di due binari distinti del framework da distribuire sui nodi corrispondenti. Inoltre, non sarà necessario decidere a runtime quale architettura utilizzare: basterà eseguire il binario appropriato sul nodo su cui viene effettuato il deploy del framework.

La sfida principale, invece, riguarda la compatibilità degli ambienti di esecuzione dedicati alle singole funzioni. In questo caso non si conosce a priori il codice della funzione né la sua affinità con le diverse architetture. È quindi necessario progettare gli ambienti in modo da poter eseguire qualsiasi funzione correttamente, sia su nodi x86 sia su nodi ARM, senza interventi manuali o adattamenti complessi da fare a runtime.

1.3.1 Docker

Una delle scelte più comuni da parte delle piattaforme FaaS per "impacchettare" le funzioni e garantirne l'isolamento è l'utilizzo di tool per la **containerizzazione**, come Docker.

L'immagine di un container Docker è un pacchetto software leggero, autonomo ed eseguibile che include tutto il necessario per eseguire un'applicazione: codice, runtime, strumenti di sistema, librerie di sistema e impostazioni. Queste immagini diventano effettivamente container in fase di esecuzione e, nel caso dei container Docker, le immagini diventano container quando vengono eseguite dal Docker Engine [15].

È possibile eseguire più container sulla stessa macchina e condividere il kernel del sistema operativo con altri container, ciascuno dei quali viene eseguito come processo isolato nello spazio utente [15].

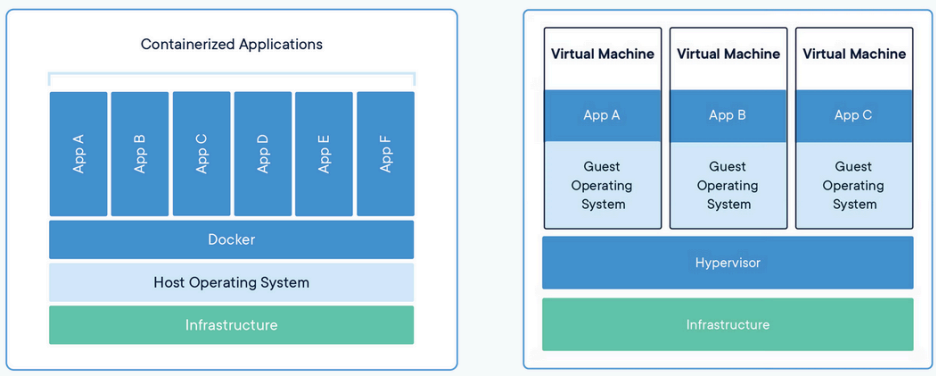

Figura 3: Differenza tra isolamento tramite container e VM [15]

Mentre una VM astrae l'intero sistema operativo, ogni container crea un ambiente di esecuzione isolato all'interno del sistema operativo stesso. Questo però significa che l'immagine di un container costruita, per esempio, per un'architettura x86 non potrà essere eseguita su un processore basato su ARM. Questo perché i binari e le librerie contenuti nel container stesso saranno compilati per un'architettura diversa. In un contesto eterogeneo come quello descritto fino a questo momento, quindi, non basta più inviare il container a un nodo pronto all'esecuzione, ma è necessario garantire che l'immagine sia stata compilata per l'architettura specifica di quel nodo. Anche la funzione inserita all'interno del container, se compilata, deve avere come target l'architettura adatta.

Per un framework FaaS che vuole supportare cluster ibridi, questo significa o dover gestire immagini multi-architettura o mantenere versioni parallele degli ambienti di runtime per ogni architettura supportata. Solo risolvendo questo problema di compatibilità a livello infrastrutturale diventa poi possibile sfruttare algoritmi di scheduling intelligenti per distribuire il carico in modo efficiente.

1.3.2 Limiti delle soluzioni attuali

Nonostante i benefici potenziali dell'eterogeneità hardware e la crescente rilevanza del Cloud-Edge Continuum, la maggior parte dei framework FaaS è ancora progettata attorno a un modello pensato per data center omogenei. L'approccio dominante nell'industria oggi si limita a garantire la compatibilità multi-architettura di base: le piattaforme riescono a eseguire container su nodi con architetture diverse senza andare in errore, ma le strategie di scheduling restano statiche e ignare dell'hardware sottostante, utilizzando algoritmi come Round Robin o simili.

In sostanza, l'eterogeneità viene tollerata, non sfruttata.

Eppure, la letteratura ha mostrato chiaramente come le affinità architetturali possano influire in modo significativo sui tempi di esecuzione a seconda del tipo di workload. Quello che manca ancora nei framework odierni è un meccanismo nativo e dinamico capace di mappare le richieste in ingresso verso l'architettura più adatta, tenendo conto del carico reale del sistema in quel momento. È proprio questa assenza di "intelligenza" nello scheduling il gap principale da colmare per operare efficacemente nell'Edge: l'obiettivo non è più solo far girare una funzione senza errori su un cluster ibrido, ma fare in modo che l'eterogeneità hardware smetta di essere un problema da gestire e diventi invece una risorsa sfruttabile per incrementare le performance.

1.4 Obiettivi della Tesi

L'obiettivo di questa tesi è affrontare le problematiche descritte finora, sviluppando un sistema FaaS in grado di supportare e sfruttare appieno l'eterogeneità dell'hardware. A questo scopo, si utilizza Serverledge come piattaforma di riferimento su cui progettare, implementare e valutare le soluzioni proposte. Serverledge è un framework Function-as-a-Service progettato per lavorare nell'Edge-Cloud Continuum. Consente agli utenti di definire ed eseguire funzioni su nodi distribuiti situati nell'Edge della rete o in data center cloud [16].

Il lavoro di questa tesi si articola in più fasi. Inizialmente l'obiettivo sarà quello di rendere Serverledge **compatibile** anche con le architetture ARM. Per farlo sarà necessario innanzitutto rendere il framework stesso compilabile ed eseguibile su ARM, dopodiché bisognerà rendere compatibili anche i runtime offerti da Serverledge con entrambe le architetture x86 e ARM. In questa fase si proporrà anche un'estensione dei runtime di default disponibili per gli utenti.

Per sfruttare l'eterogeneità hardware, sarà necessario gestire le richieste in modo intelligente. L'obiettivo successivo sarà dunque quello di realizzare un meccanismo di load balancing *Architecture-Aware*. Serverledge dovrà quindi essere consapevole di avere a disposizione nodi di architetture diverse, e dovrà essere in grado di selezionare per ogni funzione un runtime compatibile per la sua esecuzione. È importante notare che Serverledge, oltre a offrire dei runtime di default per le funzioni, consente anche agli utenti di utilizzare dei propri runtime *custom*. Questo rappresenta un'ulteriore sfida in fase di progettazione e implementazione, perché aggiunge una nuova fonte di incertezza: non si potrà sapere a priori se i runtime custom degli utenti saranno compatibili con entrambe le architetture supportate da Serverledge.

Una volta ultimata la fase dedicata a rendere il framework compatibile con ARM, si entrerà nella seconda parte di questo lavoro: ottimizzare Serverledge per lavorare con entrambe le architetture supportate, per fare in modo che possa **sfruttare** a suo vantaggio l'eterogeneità dei cluster di nodi su cui esegue. In questa fase l'obiettivo non si limiterà più a far sì che il framework lavori correttamente con entrambe le architetture, ma si cercherà di rendere il meccanismo di load balancing Architecture-Aware intelligente e adattivo, per fare in modo che riesca a scegliere l'architettura migliore per ogni funzione in ogni momento. Per farlo, si ricorrerà ad algoritmi di Reinforcement Learning, e in particolare ai Multi-Armed Bandit (MAB).

Infine, si valideranno e si confronteranno le diverse soluzioni individuate tramite esperimenti realizzati su cluster eterogenei reali. Questi esperimenti

consentiranno di osservarne il comportamento in differenti scenari, così da valutare in modo accurato l'efficacia delle implementazioni proposte.

1.5 Organizzazione della Tesi

Il presente lavoro di tesi è organizzato in sette capitoli. Il Capitolo 2 analizza lo stato dell'arte, concentrandosi sulle differenze prestazionali tra architetture x86 e ARM, sulle tecnologie di isolamento tramite container multi-architettura e sulle strategie attuali di load balancing nel Cloud-Edge Continuum. Nel Capitolo 3 viene descritta l'integrazione del supporto multi-architettura all'interno di Serverledge, con l'estensione del modello dati, l'adeguamento dei runtime Docker per linguaggi interpretati e compilati e l'aggiornamento delle policy di offloading. Il Capitolo 4 tratta la riprogettazione del load balancer, introducendo l'uso del consistent hashing con nodi virtuali per gestire in modo stabile e bilanciato i diversi anelli di risorse hardware. Nel Capitolo 5 sono illustrati gli algoritmi di Reinforcement Learning, con l'implementazione dei modelli Multi-Armed Bandit per la selezione intelligente dell'architettura ottimale basata su reward storici e contesto operativo. Il Capitolo 6 presenta i risultati sperimentali ottenuti su cluster ibridi in ambiente Google Cloud, confrontando l'efficacia delle politiche adattive rispetto alle soluzioni tradizionali. Infine, il Capitolo 7 riassume le conclusioni del lavoro e propone possibili sviluppi futuri.

Capitolo 2

Analisi dello stato dell'arte

2.1 Architetture nei FaaS

In questo capitolo si esamina l'utilizzo delle architetture x86 e ARM nei principali sistemi Function-as-a-Service (FaaS), analizzando le differenze e i vantaggi che queste offrono, con l'obiettivo di offrire una panoramica sull'attuale stato dell'arte.

L'evoluzione delle piattaforme FaaS ha visto, negli ultimi anni, una transizione da data center basati esclusivamente su processori x86 a un ecosistema in cui i processori ARM sono sempre più diffusi. Questa diversificazione non sta avvenendo solamente per motivi commerciali, ma per ottimizzare il rapporto tra prestazioni e consumi nelle infrastrutture Cloud ed Edge [17].

Come già accennato nella Sezione 1.2.1, la differenza principale tra queste due architetture risiede nell'Instruction Set Architecture (ISA)[6]. L'architettura x86 è basata su un CISC (Complex Instruction Set Computer), caratterizzato da istruzioni complesse, di lunghezza variabile, in grado di eseguire operazioni multi-step in un singolo ciclo di clock. Architetture di questo tipo, generalmente, favoriscono carichi di lavoro che richiedono un alto throughput in single-thread. Processori basati su questo tipo di istruzioni solitamente hanno anche un overhead maggiore per la decodifica delle istruzioni, un consumo

[6]Insieme delle istruzioni macchina e dei formati dati che definiscono l'interfaccia tra software e hardware di un processore.

di energia maggiore e di conseguenza una maggior produzione di calore. Al contrario, i processori ARM sono basati su RISC (Reduced Instruction Set Computer), un set di istruzioni più semplici e di lunghezza uniforme, orientate all'efficienza e al parallelismo. In un contesto serverless, l'efficienza termica ed energetica di ARM spesso si traduce in costi minori per le operazioni da parte del cloud provider, che consente anche agli utenti finali di avere un risparmio. In generale, infatti, i cloud provider offrono istanze ARM a costi inferiori anche del 30-40% rispetto a quelle x86 [18], [19].

ARM e x86 gestiscono in maniera molto diversa anche il SMT (Simultaneous Multi Threading[7]), una tecnica che permette a più thread indipendenti in esecuzione di utilizzare, in parallelo, le risorse offerte da un singolo processore. Su x86 questa tecnica è presente su praticamente tutti i processori moderni, e utilizzata in modo molto aggressivo. Su ARM, al contrario, è quasi sempre assente, o comunque molto più conservativa rispetto alla controparte x86. L'utilizzo del SMT, se da un lato garantisce performance più spinte, dall'altro significa anche maggiore contesa sulle risorse. Questo aspetto è estremamente interessante per quanto riguarda i sistemi FaaS. Infatti, i nodi di un framework FaaS si ritrovano continuamente a eseguire funzioni diverse per utenti diversi, e quando il carico sulle CPU è alto, questo può portare a una contesa sulle risorse maggiore rispetto a quella che avverrebbe su un nodo con processore ARM. Questo alla fine comporta tempi di esecuzione fino a quasi 3 volte più variabili su x86 rispetto ad ARM per quanto riguarda l'esecuzione di funzioni tramite piattaforme FaaS come AWS [13], [20].

I Cold Start, introdotti nella Sezione 1.2.1 e illustrati schematicamente nella Figura 1, sono un altro aspetto critico dei sistemi FaaS. Alcuni benchmark suggeriscono come i cold start su ARM possano essere fino al 20% più rapidi per quanto riguarda l'hardware usato da AWS Lambda [21].

Infine, come accennato nella Sezione 1.2.1, in base al tipo di funzione che si esegue, questa potrebbe performare meglio su una specifica architettura

[7]Più noto come *Hyper-Threading* sulle CPU Intel

piuttosto che su un'altra. La differenza sui tempi di esecuzione della stessa funzione su architetture diverse, può arrivare fino al 50% nei casi limite, come mostrato da Chen et al. [13]. Nei loro esperimenti hanno considerato un set di funzioni che stressano diverse caratteristiche del sistema, come CPU, I/O e memoria, eseguendo queste funzioni su CPU di architetture diverse (x86 e ARM) con prestazioni comparabili. In questo scenario, alcune funzioni sono risultate in media più veloci su ARM, altre più rapide su x86, mentre altre ancora hanno mostrato prestazioni simili su entrambe le architetture.

Inoltre, va considerato anche che è estremamente complesso predire a priori se e su quale architettura una funzione eseguirà più velocemente. Spesso è necessario ricorrere a soluzioni dinamiche, che prevedono l'esecuzione e la profilazione delle funzioni per raccogliere diverse metriche. Questi dati vengono poi utilizzati in modelli predittivi per stimare come una funzione si comporterebbe su un'altra architettura [14].

Diventa quindi chiaro che, nelle piattaforme FaaS, non esista un'architettura superiore all'altra in termini assoluti, ma che tutto dipenda dal contesto di utilizzo e dalle specifiche esigenze.

2.2 Isolamento delle funzioni

Come anticipato nella Sezione 1.3, i container Linux sono una delle soluzioni principali per isolare l'esecuzione di funzioni. Un container è un'unità standard di software che raggruppa il codice e tutte le sue dipendenze, in modo che l'applicazione funzioni in modo rapido e affidabile in ambienti diversi [15]. Nel contesto dei FaaS un container contiene il codice della funzione e tutte le librerie necessarie per la sua corretta esecuzione. Il principale strumento per il deployment di container è sicuramente Docker, introdotto nella Sezione 1.3.1.

2.2.1 Container Docker multi-architettura

Docker supporta nativamente l'esecuzione su architetture eterogenee, come x86 e ARM, purché le immagini e i container siano progettati e realizzati in

modo appropriato. Questo è possibile grazie a due componenti complementari: l'OCI[8] *Image Specification* e Docker buildx.

Nel modello OCI, un'immagine non è un singolo oggetto opaco, ma una struttura composta da più elementi: il manifest, l'index, un set di layers e una configuration [22].

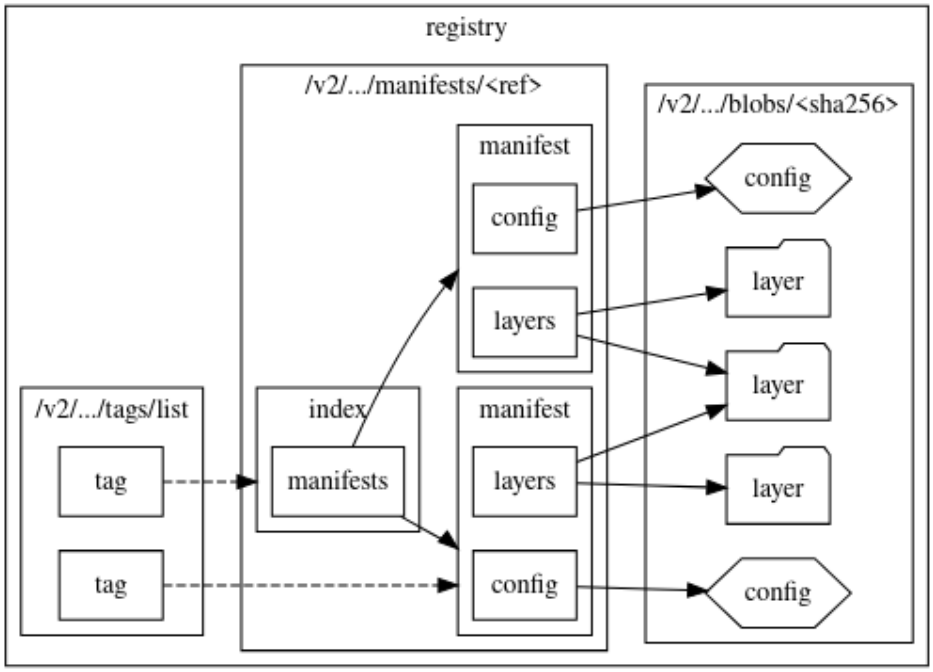

Figura 4: Esempio di immagine con più varianti organizzate in un unico index

L'index, in particolare, è l'elemento che permette un utilizzo trasparente dei container anche in un contesto multi-architettura. Nello specifico, quando un nodo richiede l'immagine Docker a uno specifico tag, invierà la richiesta al registry di Docker. Qui, tramite l'index, verrà selezionata la variante opportuna per la combinazione OS-Architettura del nodo che ha fatto la richiesta, come si può vedere nella Figura 4. Per esempio, se un nodo con processore ARM e uno con processore x86 richiedessero l'immagine relativa allo stesso tag, uno otterrebbe la variante `linux/arm64` mentre l'altro quella `linux/amd64`. In

[8]Open Container Initiative: progetto della Linux Foundation per creare degli standard per i container.

questo modo la gestione delle immagini multi-architettura è trasparente perché entrambe le varianti sono legate allo stesso tag.

Per sfruttare questa funzionalità, è necessario che la build delle immagini sia eseguita specificando come target tutte le architetture sulle quali dovranno essere eseguite. Di default, infatti, quando si definisce un `Dockerfile`, la build di quella immagine viene fatta solamente per l'architettura corrente.

Per poter costruire immagini Docker per più architetture, è necessario usare Docker buildx, un frontend per BuildKit che consente di generare immagini multi-architettura a partire da un singolo `Dockerfile`. Tramite il comando

```
docker buildx build --platform linux/amd64,linux/arm64
```

è possibile costruire, in un unico passaggio, più varianti dell'immagine (una per ogni architettura) [23]. Le immagini saranno poi pubblicate associandole allo stesso tag, tramite l'image index OCI descritto in precedenza. Se l'ambiente di build non dispone fisicamente di tutte le architetture richieste, buildx può usare QEMU[9], che permette di eseguire binari non nativi in emulazione durante la fase di build. In questo modo quindi si può eseguire la build di un'immagine per una data architettura, senza doverne disporre fisicamente. In questa situazione c'è un trade-off tra usabilità ed efficienza: eseguire una build tramite emulazione può essere sensibilmente più lento che farlo su hardware reale.

In conclusione, questi meccanismi rappresentano lo stato attuale della tecnologia per supportare, a livello di container, workload multi-architettura che devono essere schedulati dinamicamente tra risorse x86 e ARM.

2.3 Load Balancing Architecture-Aware in ambito FaaS

Lo stato dell'arte sul load balancing architecture-aware nelle piattaforme FaaS è ancora relativamente recente e in rapida evoluzione. Tuttavia, esistono già

[9]Emulatore free e open source che utilizza la traduzione binaria dinamica per emulare il processore di un computer.

diversi lavori che affrontano questo problema e mostrano come si possa trarre grande beneficio dalla conoscenza dell'architettura del nodo che si sceglie per l'esecuzione.

2.3.1 Architecture-Awareness in piattaforme Serverless

Un contributo importante è sicuramente quello di Arys et al. sull'*Energy-Aware Scheduling of a Serverless Workload in an ISA-Heterogeneous Cluster* [24]. In questo studio gli autori considerano un cluster misto con architetture x86 e ARM, e mostrano come il modello serverless si presti in modo naturale a sfruttare l'esecuzione su ISA diversi, anche grazie al fatto che in questo modello si può assumere che le funzioni da eseguire siano stateless. In questo lavoro gli autori costruiscono offline un modello che mette in relazione prestazioni ed energia utilizzata da ogni funzione rispetto alle due architetture. Questo modello, costruito offline, viene poi utilizzato da uno scheduler che decide, per ogni invocazione di una funzione, su quale architettura eseguirla. I risultati sperimentali dimostrano come, a parità di workload, si riesca a ridurre il consumo energetico fino al 15.2%. Tale approccio dimostra come uno scheduling architecture-aware non sia solo possibile, ma anche vantaggioso. Questo lavoro si avvicina molto ai meccanismi che si vogliono realizzare in questa tesi, sebbene qui il focus sia sul minimizzare i consumi energetici piuttosto che sul minimizzare il tempo di risposta.

Un altro lavoro utile per analizzare lo stato dell'arte, è quello di Du et al. [25], che introducono *Molecule*, una piattaforma di *«serverless computing on heterogeneous computers»* che estende il paradigma FaaS a sistemi che includono CPU general-purpose e acceleratori (DPU, FPGA, GPU[10]). Pur non concentrandosi specificatamente su architetture x86 e ARM, questo studio si concentra sul fornire un'astrazione dell'hardware sottostante, e discute alcuni dei problemi che ne derivano, come il bilanciamento tra distribuzione del

[10]GPU (Graphics Processing Unit), FPGA (Field-Programmable Gate Array) e DPU (Data Processing Unit) sono acceleratori hardware specializzati progettati rispettivamente per calcolo parallelo massivo, logica riconfigurabile e offload di funzioni di rete e sistema.

carico e sfruttamento degli acceleratori. Questo lavoro contribuisce a rafforzare l'idea che i sistemi FaaS si trovano sempre più spesso a operare con hardware eterogeneo, e che questo sia un qualcosa che possa essere sfruttato per massimizzare le prestazioni, anche se in questo caso si tratta di acceleratori hardware piuttosto che di architettura delle CPU.

Sempre all'interno del contesto serverless, e nello specifico nell'edge della rete, Aslanpour et al. [26] propongono un framework di *«load balancing for heterogeneous serverless edge computing»* chiamato *Hedgi*. In questo caso, l'eterogeneità riguarda sia l'hardware sia lo stack software dei nodi, ma l'idea di fondo è simile a quanto analizzato fino a questo momento: gli autori classificano i nodi in termini di throughput, latenza e consumi, e poi realizzano delle policy per il load balancing basate su pesi che riflettono le caratteristiche di ciascun nodo. Hedgi supporta esplicitamente la configurazione di policy per il load balancing consapevoli dell'eterogeneità dell'hardware, che cercano di massimizzare le performance. Lo studio dimostra che diverse scelte per la policy utilizzata (ad esempio favorire latenza o consumo energetico) producano effetti misurabili nel sistema. Pur non parlando direttamente di x86 e ARM, questo lavoro è un ulteriore esempio di load balancing hardware-aware in un contesto serverless.

2.3.2 Scheduling per sistemi eterogenei

Ci sono anche altri studi che analizzano la problematica della schedulazione di funzioni o unità di lavoro su hardware eterogeneo, seppur non nell'ambito del serverless. Tuttavia, sono comunque concettualmente vicini al lavoro svolto in questa tesi.

Per esempio, HEATS [27] introduce un orchestratore *«heterogeneity and energy-aware»* per l'esecuzione di workload all'interno di container. Questo orchestratore apprende le performance e la quantità di energia utilizzata dai nodi fisici e migra di conseguenza i task per sfruttare le differenze dell'hardware dei nodi su cui opera. Allo stesso modo, scheduler per cluster per il Deep

Learning come *Sia* [28] assegnano job a risorse eterogenee (GPU/CPU con caratteristiche diverse) massimizzando il goodput[11] del cluster.

Questi lavori confermano che il load balancing di un sistema moderno debba tener conto delle caratteristiche hardware dei nodi, e non possa limitarsi a schemi tipo round-robin o a policy statiche che ignorano queste caratteristiche.

Infine, si può analizzare lo studio di Diwan et al. [29], che introduce MIST (*Many-ISA Scheduling Technique for Heterogeneous-ISA Architectures*), un algoritmo di scheduling che opera in sistemi multi-ISA, e che utilizza alberi decisionali e contatori di performance per migrare dinamicamente le fasi di un programma verso l'ISA più affine a quella specifica fase. Gli autori mostrano che scegliere sempre l'ISA affine può portare a esecuzioni più veloci fino al 24,7% rispetto al miglior core single-ISA, mentre scegliere sempre un processore con ISA non affine può degradare le performance fino al 22,6%. Anche se questo lavoro opera a livello di core e non di funzioni FaaS, fornisce un forte supporto all'idea che uno scheduler architecture-aware possa ottenere benefici significativi rispetto a un semplice scheduler che ignora l'esistenza di architetture diverse nei cluster. Inoltre, questo studio rafforza l'idea che la scelta dell'architettura più adatta per l'esecuzione non possa essere facilmente prevista senza meccanismi di apprendimento e osservazione.

Nel complesso, lo stato dell'arte del load balancing architecture-aware nei sistemi FaaS mostra una tendenza sempre più marcata a considerare esplicitamente le caratteristiche hardware in fase di scheduling. Sfruttando le capacità delle diverse architetture e l'affinità tra funzioni e ISA, è infatti possibile raggiungere obiettivi come la riduzione dei consumi energetici, la diminuzione della latenza o il miglioramento dei tempi di risposta.

[11] Metrica di efficienza nel deep learning che unisce la velocità con cui vengono elaborati i campioni a quanto questi contribuiscano realmente all'apprendimento, indicando quanto rapidamente il modello stia facendo progressi durante l'addestramento.

2.4 Sistema FaaS di riferimento: Serverledge

In questa sezione viene descritto il sistema FaaS sul quale è stato eseguito l'intero lavoro di questa tesi.

Come già introdotto nella Sezione 1.1.1, l'integrazione del paradigma FaaS nel contesto dell'Edge-Cloud continuum nasce dall'esigenza di unire alla semplicità di sviluppo e astrazione dell'infrastruttura sottostante offerta dal FaaS, la bassa latenza, la vicinanza alle fonti di dati e la geo-distribuzione caratteristiche dell'Edge Computing. In scenari di applicazioni pervasive come l'IoT industriale, le smart cities, applicazioni real-time o l'Edge AI, delegare l'elaborazione esclusivamente ai data-center Cloud introduce ritardi di rete e congestione della banda, che possono portare a violazioni dei vincoli del QoS[12]. Da qui nasce la spinta per estendere il FaaS anche verso l'Edge della rete, sviluppando framework progettati espressamente per operare nell'Edge-Cloud Continuum. In questo contesto si colloca Serverledge [16], un framework FaaS che nasce proprio con queste premesse.

Serverledge è una piattaforma decentralizzata di Function-as-a-Service implementata in Go, progettata per operare nell'Edge-Cloud Continuum, che consente agli utenti di gestire l'esecuzione delle proprie funzioni direttamente all'Edge della rete o di effettuare l'offloading verso le regioni centrali del Cloud quando necessario. L'architettura di Serverledge organizza logicamente i nodi in zone geografiche. Una zona può rappresentare, ad esempio, un'area Edge, che raggruppa nodi situati vicino alle sorgenti dei dati (come all'interno della stessa città o area metropolitana), oppure una regione Cloud, che comprende nodi ospitati in data center centralizzati. Sono possibili anche configurazioni intermedie, che combinano caratteristiche di entrambe le tipologie.

[12]Quality of Service. Indica l'insieme di parametri e meccanismi che garantiscono prestazioni prevedibili e affidabili di un servizio, come latenza, throughput e disponibilità.

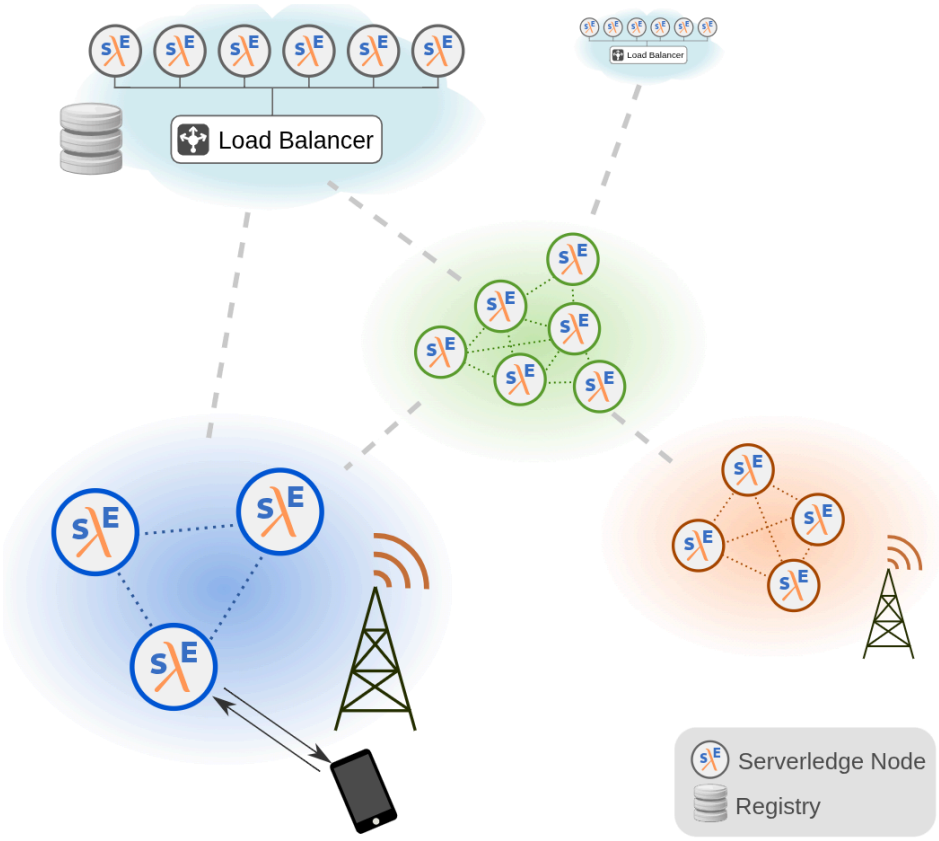

Figura 5: Overview di Serverledge [16]

Ogni nodo di Serverledge, a prescindere dalla zona o regione di appartenenza, è in grado di ricevere richieste per l'invocazione di funzioni, schedularle ed eseguirle localmente, senza dover necessariamente passare da un gateway centralizzato. Questo consente di ridurre la latenza per l'esecuzione, dato che non c'è un punto di ingresso centralizzato o uno scheduler globale.

Come si può osservare dalla Figura 5, in Serverledge sono presenti anche un global registry e dei load balancer. Il global registry mantiene le informazioni sui nodi, sulla loro collocazione, e sulle funzioni registrate nel sistema. Sebbene questo registry rappresenti una singola entità logica nell'architettura, può essere associato a più repliche per garantire scalabilità e tolleranza ai guasti. È realizzato tramite *etcd*[13], uno store chiave-valore distribuito e affidabile. Infine,

[13]https://etcd.io/

si può avere un load balancer per ogni zona o regione di Serverledge, per distribuire le richieste in arrivo tra i nodi di quel cluster.

Serverledge fa uso dei container Docker descritti nella Sezione 1.3.1 per eseguire le funzioni in maniera isolata. Adotta una politica di creazione di container on-demand, effettuandone il deployment solamente quando arriva una richiesta di esecuzione di una funzione. In quel caso, allora, se il nodo dispone di sufficienti risorse (CPU e memoria) andrà ad allocare un nuovo container relativo al runtime della funzione da eseguire. Per ridurre il numero di cold start per ogni funzione, un container allocato su un nodo non viene deallocato immediatamente al termine dell'esecuzione. Invece, viene mantenuto in un pool di container warm per un periodo di tempo prestabilito. Se arriva una nuova richiesta per una funzione con un container disponibile nel pool, quest'ultimo viene riutilizzato, evitando così la creazione da zero di un nuovo container e riducendo la latenza.

Può succedere però che il nodo a cui arriva la richiesta non abbia risorse sufficienti per la creazione del relativo container, e quindi non possa eseguire la funzione richiesta. Serverledge, in questo caso, supporta l'*offloading*, ovvero un meccanismo grazie a cui i nodi possono inoltrare le richieste verso altri nodi nella stessa zona, o nodi in zone o regioni differenti. In particolare, si hanno due tipologie di offloading:

- Offloading **verticale**: le richieste possono essere inoltrate da un nodo nell'edge verso uno in una regione cloud, in modo da sfruttare la maggior capacità di calcolo a sua disposizione.
- Offloading **orizzontale**: le richieste di invocazione possono essere inoltrate da un nodo Edge che non ha sufficienti risorse in un dato momento, a un altro nodo Edge. In questo modo l'esecuzione può avvenire senza attendere che si liberino risorse sul nodo originale, e senza dover incorrere in latenze maggiori date dal dover inoltrare una richiesta verso le regioni Cloud.

Quando un nodo esegue l'offloading, funge anche da proxy: inoltra la richiesta al nodo selezionato, attende l'esito dell'esecuzione e poi restituisce il risultato all'utente che l'ha inviata.

Un nodo Serverledge, come illustrato dalla Figura 6, ha al suo interno diversi componenti: un'API server, uno scheduler, un local registry, un offloader e un container pool.

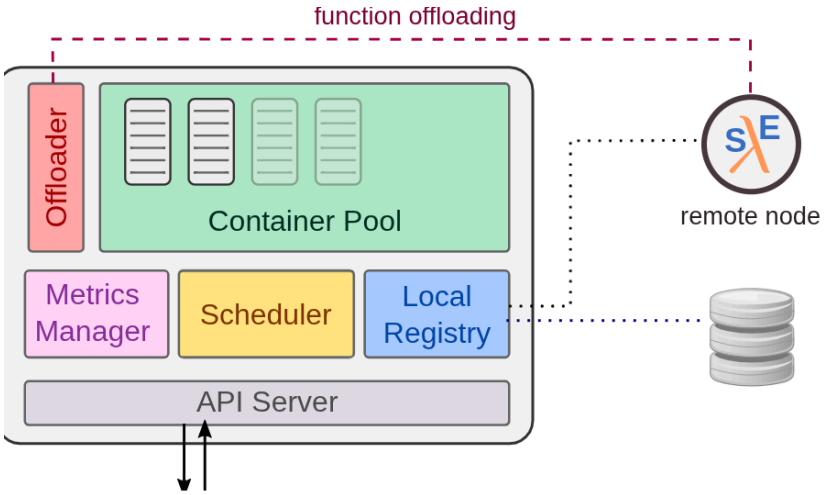

Figura 6: Architettura di un nodo Serverledge [16]

Il server API espone le operazioni per la gestione delle funzioni (creazione, invocazione, update o cancellazione, per citarne alcune). Queste funzionalità possono essere utilizzate tramite richieste HTTP sia dagli utenti di Serverledge, sia da altri nodi, per esempio in fase di offloading. Questo componente è realizzato tramite la libreria *Echo*[14], un web framework per Go ad alte prestazioni. Il local registry ha invece un doppio ruolo. In primo luogo agisce da cache per i dati ottenuti dal global registry. In questo modo le informazioni non devono essere reperite ogni volta dal registry globale, diminuendo la latenza. Le informazioni in cache hanno poi un *time-to-live*, in modo che possano essere periodicamente aggiornate. Inoltre, il local registry è anche responsabile di mantenere tutte quelle informazioni che sono di interesse per il nodo locale, e che non devono essere propagate a livello globale. Nello specifico, i nodi

[14]https://echo.labstack.com/

eseguono l'algoritmo *Vivaldi* [30] per costruire e aggiornare uno spazio virtuale di coordinate, che consente ai vari nodi di stimare la latenza di rete tra di essi. Vivaldi richiede che siano scambiati messaggi in maniera periodica, e Serverledge sfrutta questi messaggi per propagare tra i vari nodi anche informazioni riguardo lo stato delle risorse (CPU e memoria) di ogni nodo, e uno snapshot sullo stato del pool di container warm. Queste informazioni favoriscono poi un offloading efficace. In Serverledge, quando un nodo riceve una richiesta di esecuzione di una funzione, recupera tutte le informazioni relative dal local registry, se queste sono in cache, altrimenti contatterà il global registry. A questo punto, lo scheduler presente su ogni nodo, si dovrà occupare di stabilire dove e quando eseguire la funzione. Sulla base delle informazioni in suo possesso, lo scheduler può decidere di eseguire una funzione localmente, di fare offloading o di scartare la richiesta. Nel caso di esecuzione locale, poi si occupa di capire se è possibile utilizzare un container presente nel pool di container warm, o se è necessario eseguire il deploy di un nuovo container. Infatti, in base al tipo di funzione, sarà necessario uno specifico tipo di container: se una funzione è scritta, per esempio, in Python avrà bisogno di un runtime, e quindi di un container, su cui è effettivamente presente l'interprete Python. Se il nodo locale non dovesse avere abbastanza risorse per l'esecuzione della funzione, allora lo scheduler può decidere di fare offloading orizzontale o verticale per quella richiesta, oppure di scartarla, in base al tipo di policy con cui è configurato. Il container pool di Serverledge è realizzato per non essere legato a una specifica piattaforma di containerizzazione. Nella sua versione attuale, è presente l'integrazione con Docker.

Terminata l'esecuzione della funzione, il risultato viene restituito in formato JSON all'API server del client, che si occuperà poi di inoltrarlo all'utente che ha inviato la richiesta. La Figura 7 illustra il life-cycle di una funzione in Serverledge, e l'interazione tra i vari componenti descritti in questa sezione.

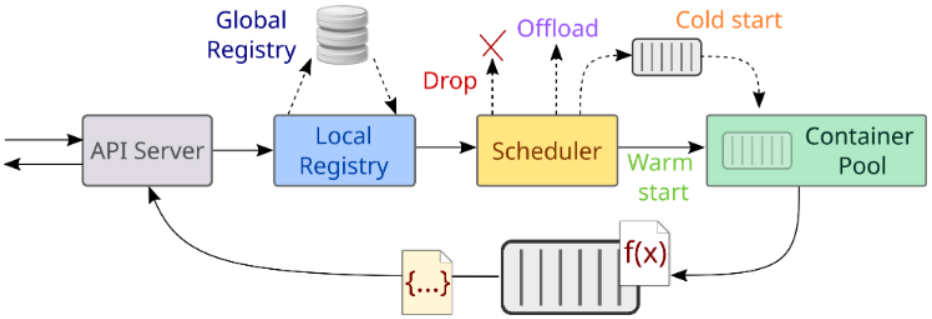

Figura 7: Interazione tra i componenti di un nodo Serverledge per l'esecuzione di una funzione

Le valutazioni sperimentali evidenziano come Serverledge superi Apache OpenWhisk in scenari *Edge-like* e mostri prestazioni competitive rispetto ai framework all'avanguardia ottimizzati per l'Edge, con il vantaggio aggiuntivo di supportare nativamente il vertical e l'horizontal offloading.

Nonostante questi vantaggi, nella sua versione originale Serverledge assumeva un cluster con hardware omogeneo. In particolare, non era previsto un supporto esplicito per architetture multiple: di conseguenza, l'esecuzione risultava limitata ad ambienti x86, non consentendo l'utilizzo di nodi basati su architetture diverse, come ARM.

Capitolo 3

Supporto multi-architettura in Serverledge

In questa sezione si descrive la soluzione proposta per rendere Serverledge compatibile con molteplici architetture. Vengono quindi analizzate le modifiche apportate alle strutture dati e agli algoritmi del framework, insieme agli interventi necessari sui runtime Docker offerti dalla piattaforma.

L'adattamento del control-plane di Serverledge non ha richiesto modifiche al codice sorgente. Grazie al fatto che l'intero framework è sviluppato in Go, questa fase si è rivelata estremamente rapida, poiché Go supporta compilazioni multi-architettura ed è quindi compatibile nativamente anche con ARM. Serverledge, inoltre, non fa uso di nessuna libreria specifica per architettura x86 o che non possiede una versione anche per ARM. Non sono state quindi necessarie modifiche nemmeno al processo di compilazione. Quando compilato per ARM, Go produce un binario compatibile anche con questa architettura.

3.1 Estensione del modello dati e del protocollo di discovery

Le prime vere modifiche sono state necessarie per introdurre la nozione di architettura all'interno di Serverledge. Infatti, sebbene il framework possa già eseguire su ARM, non è a conoscenza del fatto che stia eseguendo su

un'architettura piuttosto che su un'altra. Questa informazione sarà necessaria al framework per poter prendere decisioni riguardo allo scheduling di funzioni senza commettere errori. Se non venissero apportate delle modifiche, Serverledge potrebbe richiedere a un nodo ARM di eseguire una funzione che è compatibile solo con architetture x86, e viceversa. Un comportamento di questo tipo, chiaramente, porterebbe al fallimento dell'esecuzione delle funzioni.

3.1.1 Introduzione dell'architettura nei metadati

In primo luogo, si è reso necessario dotare ogni nodo Serverledge della consapevolezza dell'architettura su cui è in esecuzione, in modo che questa potesse essere condivisa con il resto del sistema. Questa informazione è stata quindi integrata ai metadati che ciascun nodo invia verso il global registry Etcd. Così facendo, si ottiene una visione globale del cluster, potendo conoscere in qualsiasi momento su quale hardware sta operando ogni singolo componente. A livello implementativo, questo risultato è stato ottenuto intervenendo sulle informazioni di stato locale, aggiungendo esplicitamente il campo `Arch` all'interno della struct `NodeID`.

Infine, si sono ristrutturate le informazioni mantenute sui runtime. Come introdotto nella Sezione 2.4, in Serverledge ogni funzione è eseguita all'interno di un runtime. L'implementazione attuale prevede che ogni runtime corrisponda a un'immagine di un container Docker. Serverledge offre sia dei runtime di default pronti all'utilizzo, sia la possibilità per l'utente di utilizzare un suo runtime custom. Mentre prima questo meccanismo non aveva bisogno di informazioni aggiuntive, ora è invece necessario mantenere, per ogni runtime, l'informazione su quali architetture questo supporta. Altrimenti si correrebbe il rischio di chiedere a un nodo di utilizzare un container Docker che non ha un'immagine definita per la sua architettura.

Queste informazioni sono mantenute localmente da ogni nodo nelle strutture dati `RuntimeToInfo` e `CustomRuntimeToInfo` di `internal/container/runtimes.go`. La prima è una struttura dati statica che contiene i nomi dei runtime di default e le informazioni sulle architetture da essi supportate. Come

si vedrà nella Sezione 3.3, tutti i runtime di Serverledge sono stati adattati per supportare sia x86 che ARM. La seconda struttura dati, invece, viene popolata dinamicamente quando un utente registra una nuova funzione che utilizza un runtime custom. L'informazione sulle architetture supportate è stata aggiunta anche alle funzioni stesse. Infatti, quando viene creata una funzione, viene inserita questa informazione anche nella struct `Function`. Si è fatta questa scelta perché questa struttura dati viene salvata sul global registry quando una funzione viene creata. In questo modo, quando un nodo deve effettuare l'offloading di una funzione o quando il load balancer deve selezionare il nodo target, può accedere a questa struttura dati. Ciò consente di prendere una decisione *architecture-aware*, tenendo conto sia dell'architettura degli altri nodi sia di quelle supportate dal runtime della funzione.

3.1.2 Architetture supportate dai runtime

Se per i runtime di default offerti da Serverledge, come anticipato, le architetture supportate sono note a priori, la stessa cosa non è vera per i runtime custom. In questo caso erano possibili due soluzioni. La prima consisteva nel richiedere all'utente, al momento della registrazione di una funzione con runtime custom in Serverledge, di specificare anche le architetture supportate dal runtime fornito. Questa soluzione tuttavia avrebbe introdotto un onere aggiuntivo per l'utente, oltre a non poter essere considerata del tutto affidabile, poiché l'utente potrebbe fornire informazioni scorrette o non aggiornate. Si è scelto dunque di optare per una seconda soluzione: fare in modo che Serverledge stesso fosse in grado di reperire le informazioni sulle architetture supportate da un qualsiasi runtime fornito dagli utenti. Per farlo è stata creata la funzione `GetImageArchitectures` in `internal/container/docker.go`, di cui si presenta un estratto nel Algoritmo 1. Questa funzione ha lo scopo di reperire i metadati dell'immagine Docker che riceve come parametro, identificare le architetture supportate e poi salvare le informazioni sul global registry. È strutturata in tre fasi principali:

1. Contattare il global registry per controllare se qualche altro nodo ha già eseguito la stessa operazione per questo runtime, ed ha quindi salvato lì il risultato. Se questo è il caso, la latenza è notevolmente ridotta rispetto al caso in cui bisogna andare a contattare il registry remoto di Docker (di default Docker Hub[15]).
2. Se l'informazione non è presente sul global registry di Serverledge, allora si procede a contattare il registry Docker remoto. Come spiegato nella sezione Sezione 2.2.1, qui bisogna distinguere tra il caso in cui l'immagine abbia un index oppure no. Nel primo caso si è in presenza di un'immagine multi-architettura ed è quindi necessario effettuare il *parsing* del manifest per estrarre le architetture supportate rilevanti per Serverledge. Se invece l'oggetto restituito dal registry remoto è un'immagine singola, significa che è supportata una sola architettura e l'informazione deve essere ricavata dal relativo file config. Come illustrato nella Figura 4, questa struttura per i dati non è specifica di Docker, ma di tutti quei servizi di containerizzazione che aderiscono al formato OCI. L'approccio utilizzato da questa soluzione, quindi, non è limitato a Docker, ma in futuro potrebbe essere utilizzato anche con altre piattaforme.
3. Infine, una volta recuperate, queste informazioni vengono inserite nel global registry di Serverledge, per poter essere riutilizzate con latenza minore in futuro.

[15]https://hub.docker.com/

Input: I (Nome dell'immagine Docker)
Output: A (Lista delle architetture supportate)
$A_c \leftarrow \text{GetFromCache}(I)$
if $A_c \neq \emptyset$ **then**
  return A_c
end
$M \leftarrow \text{FetchManifest}(I)$
$A \leftarrow \emptyset$
if $\text{IsMultiPlatform}(M)$ **then**
  for each platform $\in M$ **do**
    if platform.arch $\in \{\text{x86}, \text{ARM}\}$ **then**
      $A \leftarrow A \cup \{\text{platform.arch}\}$
    end
  end
else
  if M.arch $\in \{\text{x86}, \text{ARM}\}$ **then**
    $A \leftarrow A \cup \{M.\text{arch}\}$
  end
end
if $A \neq \emptyset$ **then**
  $\text{SaveToCache}(I, A)$
end
return A

Algoritmo 1: Pseudocodice della funzione `GetImageArchitectures` introdotta in Serverledge

3.2 Immagini Docker multi-architettura

Come anticipato, anche le immagini Docker offerte da Serverledge per i runtime di default hanno richiesto delle modifiche. Infatti, le versioni originariamente offerte erano tutte immagini compatibili solamente con x86. Al momento dello sviluppo, Serverledge offriva nativamente il supporto per gli ambienti di esecuzione Python 3.10 e Node.js 17. Per questi runtime è stata quindi utilizzata

una pipeline di build per la generazione di immagini Docker multi-architettura, compatibili sia con processori x86 che ARM. Per questi due linguaggi non sono state necessarie modifiche particolari, essendo entrambi interpretati (con Node.js che utilizza anche compilazione JIT[16] parziale).

In questi linguaggi, infatti, il codice sorgente non viene compilato in anticipo con istruzioni macchina specifiche per una determinata CPU. Di conseguenza la stessa identica funzione fornita dall'utente può essere eseguita indifferentemente su un nodo x86 o ARM. Con questi linguaggi, quindi, ciò che l'utente finale deve fornire a Serverledge non è altro che il codice stesso della funzione in formato testuale, e non degli eseguibili per le architetture da supportare.

La gestione delle differenze architetturali è delegata all'immagine del container, che dovrà ovviamente includere l'eseguibile dell'interprete o del motore JIT per quella specifica architettura. Sfruttando strumenti standard come Docker Buildx, introdotto nella Sezione 2.2.1, è sufficiente specificare che il processo di build costruisca l'immagine per entrambe le architetture. Un esempio di questo comando è riportato nel Listato 1. Sarà poi il demone Docker sul nodo di destinazione a scaricare la versione dell'immagine compatibile con la propria architettura hardware.

```
docker buildx build \
--no-cache \
--platform linux/amd64,linux/arm64 \
-t fmuschera/serverledge-python314:latest \
-f ./images/python310/Dockerfile \
--push .
```

Listato 1: Esempio di comando `docker buildx` usato per la build di immagine cross-architettura

[16]Just-In-Time: tecnica di compilazione che traduce dinamicamente il codice in codice macchina durante l'esecuzione, con l'obiettivo di migliorarne le prestazioni rispetto all'interpretazione pura.

Come si può evincere anche dal comando nel Listato 1, dovendo ricreare le immagini per Serverledge, si è anche aggiornata la versione di Python del runtime, passando dalla versione 3.10 alla 3.14. Infine, si è utilizzata la pipeline per la build appena descritta anche per l'immagine `base` di Serverledge. Questa immagine non rappresenta un runtime direttamente utilizzabile dagli utenti, ma è un componente che possono utilizzare per costruire in maniera semplice i propri runtime custom. Come mostrato nella Figura 8, in Serverledge ogni container ha al suo interno anche un `Executor`. Questo executor si occupa di ascoltare le richieste HTTP di *invoke* che arrivano da Serverledge stesso, prepara l'ambiente per l'esecuzione (per esempio settando i parametri che la funzione si aspetta di ricevere), esegue il comando per invocare la funzione, ne cattura il risultato e poi lo inoltra al nodo Serverledge. La sua funzionalità quindi è quella di fare da tramite tra il nodo e la funzione dell'utente all'interno del container Docker.

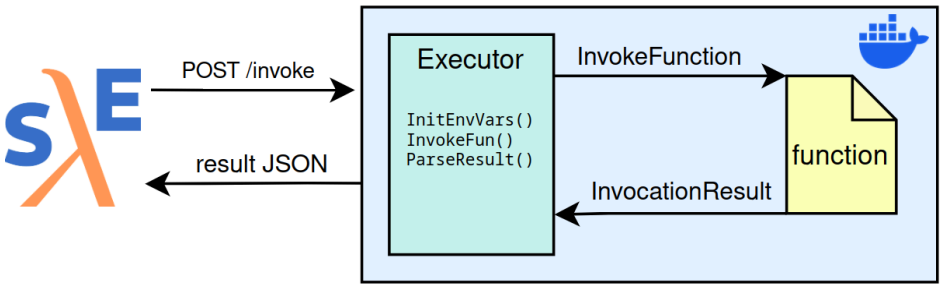

Figura 8: Struttura dei container usati in Serverledge

Nei runtime di default offerti da Serverledge, l'executor è implementato nello stesso linguaggio della funzione da eseguire, così da garantire la massima efficienza. Per gli utenti che necessitano di un runtime custom, l'immagine `base` di Serverledge fornisce un executor scritto in Go, integrabile all'interno delle proprie immagini. Dal punto di vista prestazionale, la soluzione più efficiente sarebbe quella di costruire un'immagine completamente dedicata, adattando la funzione allo schema di esecuzione previsto da Serverledge ed evitando così l'overhead introdotto dall'executor generico. L'utilizzo dell'immagine base

rappresenta quindi un compromesso per semplificare lo sviluppo, a fronte di un lieve costo in termini di overhead.

Queste operazioni consentono ora di rendere tutti i runtime offerti da Serverledge compatibili con entrambe le architetture x86 e ARM. Inoltre, la funzione `GetImageArchitectures` introdotta, permette di determinare in modo affidabile le architetture supportate anche dai runtime custom definiti dagli utenti. Tutto questo, unito all'integrazione dell'informazione sull'architettura nei nodi Serverledge e nel global registry rappresenta un primo passo fondamentale verso il supporto di scenari multi-architettura. Pur non rendendo ancora il framework pienamente operativo in questi contesti, queste modifiche creano le condizioni necessarie affinché il sistema possa iniziare a considerare l'eterogeneità hardware.

3.3 Estensione dei runtime Docker

Prima di proseguire ulteriormente con l'adattamento di Serverledge al supporto di scenari multi-architettura, questa sottosezione descrive l'estensione dei runtime di default offerti dal framework. In particolare, oltre a quelli già esistenti, sono stati introdotti anche runtime per linguaggi compilati, strutturalmente diversi rispetto a quelli per linguaggi interpretati analizzati finora. Questa differenza li rende particolarmente rilevanti nel contesto del supporto multi-architettura. Complessivamente sono stati aggiunti tre nuovi runtime: uno basato su Python e dedicato al Machine Learning, uno per Java 21 e infine un runtime per funzioni scritte in Go.

Python 3.12 ML Per questo runtime è stata scelta una versione di Python leggermente meno recente sulla quale tutte le principali librerie di Machine Learning fossero mature e ampiamente testate. Concettualmente l'immagine è simile a quella offerta dal runtime Python base, ma include librerie come `numpy`, `pandas` e `sklearn`. Viene installata anche la libreria di `tensorflow`, facendo attenzione all'architettura target dell'immagine, e utilizzando l'opportuna versione di questa libreria, che ha nomi diversi in base all'architettura per

cui è compilata. Sebbene lo stack di sviluppo per il Machine Learning e le relative librerie risultino più maturi su architetture x86, estendere il runtime al supporto di ARM consente a Serverledge una maggiore flessibilità nella distribuzione delle richieste.

Java 21 In questo caso è stato necessario scrivere da zero un nuovo `Executor`, il componente introdotto nella Sezione 3.2 e illustrato nella Figura 8. Questo componente è stato scritto in Java per integrarsi al meglio con le funzioni da eseguire all'interno del container stesso. Per la build di questa immagine, è stato utilizzato un approccio multi stage, come si può osservare nel Listato 2. Nella prima fase viene compilato l'Executor con le relative dipendenze, in un'immagine che include l'intero Java Development Kit (JDK). Nella seconda fase, il byte-code ottenuto viene copiato in un'immagine contenente solamente il Java Runtime Environment (JRE). Il vantaggio di operare in questo modo è quello di ottenere un'immagine molto più leggera, che contiene solamente il codice eseguibile dell'Executor e il necessario per *eseguire* codice Java, non per compilarlo. Un'immagine Docker più piccola ha anche il vantaggio di ridurre i tempi di cold start quando il nodo deve scaricare l'immagine dal repository remoto di Docker.

```
FROM eclipse-temurin:21-jdk-alpine AS builder
WORKDIR /build
RUN wget -O gson.jar https://repo1.maven.org/maven2/com/google/code/gson/
gson/2.13.2/gson-2.13.2.jar
COPY Executor.java .
RUN javac -cp .:gson.jar Executor.java

FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
RUN mkdir -p /app/
COPY --from=builder /build/*.class /app/
COPY --from=builder /build/gson.jar /app/
CMD ["java", "-cp", ".:gson.jar", "Executor"]
```

Listato 2: Dockerfile per l'immagine del runtime Java 21

Questo runtime si aspetta di ricevere dall'utente del codice Java già compilato nel formato `jar` o `fat-jar`. Il primo è un archivio Java che contiene il codice compilato ed eventuali risorse come i file di configurazione. Un `fat-jar`, invece, è un tipo di archivio che contiene al suo interno le classi della funzione e tutte le relative dipendenze. In questo modo si lascia più libertà e flessibilità agli utenti finali per la scrittura delle proprie funzioni. Si è scelto di non andare a richiedere all'utente direttamente il codice Java come si fa con Python, per esempio, per una serie di motivi. In primo luogo, eseguire la compilazione all'interno di Serverledge sarebbe estremamente complesso per via della necessità di gestire le possibili dipendenze esterne nel codice delle funzioni. Inoltre, il risultato di questa compilazione andrebbe memorizzato in cache per non dover ripetere questo processo a ogni invocazione, aggiungendo ulteriore complessità. Bisogna poi considerare che il processo di compilazione potrebbe essere un'operazione pesante, che aggiungerebbe notevole latenza al tempo di risposta, soprattutto in un framework progettato per lavorare anche nell'Edge della rete come Serverledge. Qui, infatti, i nodi possono essere in esecuzione anche su dispositivi meno prestazionali. La scelta di richiedere una versione già compilata del codice, è anche in linea con quanto avviene in altre piattaforme FaaS commerciali come AWS Lambda, Google Cloud Functions o Azure Functions. In questo modo ci si allinea allo standard *de facto* e si rende anche più agevole per gli utenti passare da una piattaforma all'altra.

Go Il runtime Go è senz'altro quello che presenta le sfide maggiori, essendo questo un linguaggio puramente compilato. In questo caso, quindi, non si potrà fare affidamento su nessun layer intermedio per la risoluzione delle differenze tra il binario prodotto per architetture diverse. Come fatto per il runtime di Java, anche qui per gli stessi motivi, si delega la compilazione del codice all'utente. In questo caso ci si aspetta dall'utente un file in formato `.zip`, contenente due eseguibili: uno compilato con target x86 e uno con target ARM. Per semplificare le operazioni dell'utente finale, l'intero processo è descritto in un'apposita sezione della documentazione di Serverledge e accompagnato da

un `Makefile` che permette di generare sia i due eseguibili sia l'archivio compresso. Questo `Makefile` specifica anche il parametro `CGO_ENABLED=0` durante la fase di build degli eseguibili. Questo parametro indica al compilatore Go che non può fare affidamento sulle librerie C del sistema operativo in uso. Questo consente all'eseguibile prodotto di essere completamente statico, e di non dipendere dalle librerie di sistema, che potrebbero essere diverse tra distribuzioni Linux diverse.

Ad esempio, i container basati su Alpine Linux, come quello utilizzato da questo runtime, impiegano la libreria C `musl`, un'implementazione più snella e minimale pensata per ambienti leggeri come i container [31]. Distribuzioni ampiamente diffuse tra cui Debian, Ubuntu o Fedora utilizzano invece la più comune `glibc`, che rappresenta lo standard *de facto* nel mondo Linux.

Ne consegue che un eseguibile compilato facendo affidamento sulla libreria C presente nel sistema di build potrebbe non essere compatibile con l'ambiente di destinazione. In particolare, un binario costruito facendo affidamento su `glibc` potrebbe non funzionare correttamente (o non avviarsi affatto) all'interno di un container basato su `musl`, generando errori a runtime. L'utilizzo del parametro `CGO_ENABLED=0` durante la fase di compilazione evita che si verifichino proprio questi scenari.

Il runtime Go presenta anche altre differenze rispetto a tutti quelli progettati fino a questo momento. Quando il container viene creato, al suo interno viene lanciato uno shell script. Questo si occupa di estrarre il codice dall'archivio ed eseguire il binario supportato dall'architettura del nodo corrente. Un frammento di questo script è disponibile nel Listato 3.

```
    for f in ./*.zip; do
        [ -e "$f" ] || continue
        unzip -o -- "$f"
    done

    ARCH=$(uname -m)
```

```
if [ "$ARCH" = "x86_64" ]; then
    BINARY="./handler_amd64"
elif [ "$ARCH" = "aarch64" ]; then
    BINARY="./handler_arm64"
else
    echo "Error: Unsupported architecture '$ARCH', or wrong names for
executables"
fi

if [ -f "$BINARY" ]; then
    chmod +x "$BINARY"
    exec "$BINARY"
else
    echo "Error: Binary '$BINARY' not found in the uploaded package."
    exit 1
fi
```

Listato 3: Frammento dello script `entrypoint.sh` per il runtime Go

Grazie a questo meccanismo, l'immagine può essere costruita con la pipeline descritta in precedenza e funzionare senza problemi su entrambe le architetture. Per quanto riguarda il componente Executor del runtime, è stata adottata una soluzione leggermente diversa rispetto ai casi precedenti: è stato creato un nuovo package all'interno del progetto, denominato `serverledge`, contenente un file Go con il codice dell'Executor. In questo file è stata definita anche la *signature* dell'handler che l'utente deve fornire.

Questa organizzazione semplifica notevolmente l'uso per l'utente finale, rendendo l'integrazione di una funzione in Serverledge estremamente lineare: basta una singola chiamata all'API `serverledge.Start(funcName)`, come mostrato nel Listato 4.

```
import (
  // Import the Serverledge Go SDK
  "github.com/serverledge-faas/serverledge/serverledge"
```

```
)
// Arguments:
//   - params: A map[string]interface{} containing the input parameters
provided by the caller.
//
// Returns:
//   - interface{}: The result payload (struct, map, string, etc.) to be
returned as JSON.
//   - error: An error object if the execution failed.
func myHandler(params map[string]interface{}) (interface{}, error) {

  // ... [User's code] ...
}

func main() {
  // Start the Serverledge runtime
  serverledge.Start(myHandler)
}
```

Listato 4: Esempio di funzione utilizzabile con il runtime Go di Serverledge

Garantire che le immagini Docker dei runtime fossero multi-architettura ha rappresentato un passaggio imprescindibile per consentire all'intero sistema FaaS di operare concretamente in un cluster eterogeneo. Sebbene l'estensione della piattaforma con nuovi runtime non fosse un requisito stretto per questa transizione, l'integrazione di ambienti per linguaggi compilati come Go ha permesso di affrontare le reali complessità legate all'incompatibilità dei file binari. A differenza dei linguaggi interpretati, dove l'adattamento multi-architettura si riduce essenzialmente alla configurazione delle pipeline di build multi-target, i linguaggi compilati presentano le sfide più complesse, richiedendo la gestione esplicita e il corretto utilizzo di binari distinti in base alla piattaforma hardware di destinazione.

Avendo ora dotato Serverledge della capacità di riconoscere ed eseguire le funzioni in modo trasparente sia su nodi x86 che ARM, si ha la necessità di gestire questa eterogeneità anche a livello di orchestrazione distribuita. Il

passo logico successivo consiste quindi nell'abilitare il sistema a far rispettare questi vincoli architetturali durante la fase di offloading. Per farlo, è necessario adattare le relative policy affinché una richiesta non venga mai inoltrata verso un nodo incompatibile con l'ambiente di esecuzione richiesto dalla funzione.

3.4 Integrazione dei vincoli architetturali nelle policy di offloading

Come introdotto nella Sezione 2.4, in Serverledge ogni nodo può delegare l'esecuzione di una funzione a un altro nodo dell'Edge oppure a un nodo situato in un data center cloud. Nella versione originale del framework, questa situazione si verificava principalmente in due casi: quando la policy imponeva di non eseguire funzioni in locale, oppure quando il nodo non disponeva di risorse sufficienti.

Con l'estensione al supporto multi-architettura emerge ora un ulteriore scenario. Può accadere che il nodo corrente sia abilitato all'esecuzione delle funzioni e disponga delle risorse necessarie, ma non supporti l'architettura richiesta dal runtime della funzione stessa. Anche in questo caso diventa necessario ricorrere all'offloading, così da garantire comunque l'esecuzione.

Di conseguenza, tutte le policy di offloading devono essere adattate per tenere conto dei vincoli architetturali dei runtime, assicurando che una funzione venga inoltrata esclusivamente verso nodi compatibili con la piattaforma richiesta.

3.4.1 Offloading policy di Serverledge

Serverledge presenta quattro policy di base, che possono poi essere estese per deploy più specifici.

- **Default Policy**: esegue la funzione localmente, se non ha abbastanza risorse, rifiuta la richiesta. Può essere utilizzata in deploy con nodo singolo, oppure in cluster con un load balancer, dove si vuole delegare solo al load balancer stesso la responsabilità di scegliere a quali nodi inviare le richieste.

- **Cloud-Only Policy**: con questa policy, l'esecuzione in locale viene disabilitata, e rende i nodi Serverledge nell'edge che la utilizzano dei semplici proxy verso i nodi nel cloud.
- **Cloud-Edge Policy**: abilita sia l'esecuzione in locale sul nodo, sia l'offloading, che però può essere eseguito solo verso nodi nel cloud, ricadendo nel caso di offloading verticale descritto in precedenza.
- **Edge-Only Policy**: Abilita sia l'esecuzione in locale, sia l'offloading orizzontale, dando però precedenza proprio all'esecuzione su un altro nodo nell'Edge.

3.4.2 Adattamento delle policy

Default Policy Nel caso della policy di default, è necessario introdurre un controllo preliminare che verifichi la compatibilità tra l'architettura del nodo corrente e quella richiesta dal runtime della funzione da eseguire. Per farlo, si sfruttano i metadati aggiunti alla struttura dati di ogni funzione descritti nella Sezione 3.1.1. In questo modo si controlla se nella lista di architetture supportate dal runtime della funzione c'è anche l'architettura del nodo corrente.

Cloud-Only Policy Nel caso della policy Cloud-Only, non è necessario modificare direttamente la policy stessa, quanto la funzione `handleCloudOffload`. Qui infatti è dove si trova la logica per l'identificazione del `RemoteOffloadingTarget` e il conseguente inoltro della funzione. In Serverledge ogni nodo può configurare un target per l'offloading verticale. In questo caso, tuttavia, non è sufficiente applicare lo stesso controllo previsto dalla policy di default, limitandosi a verificare la compatibilità tra l'architettura del nodo remoto e quella richiesta dalla funzione.

Il motivo è che il target remoto può assumere due forme differenti: può essere un nodo Serverledge "classico", oppure un load balancer. Nel primo caso è corretto effettuare il controllo architetturale, poiché sarà quel nodo a eseguire direttamente la funzione. Nel secondo caso, invece, tale verifica non avrebbe senso: un load balancer non esegue la funzione localmente, ma si limita a instradare la richiesta.

L'offloading verso un load balancer può quindi essere sempre effettuato. Sarà poi responsabilità del load balancer individuare, all'interno del cluster di riferimento, un nodo compatibile con l'architettura richiesta o, in assenza di alternative valide, segnalare l'impossibilità di eseguire la funzione.

Cloud-Edge Policy La policy Cloud-Edge può essere ora adattata semplicemente riutilizzando i meccanismi introdotti nella policy precedente. In primo luogo andrà a tentare l'esecuzione locale, che potrà avvenire solo se il controllo sulla compatibilità del runtime della funzione con l'architettura del nodo è superato. Se questo controllo non è soddisfatto, o se lo è ma il nodo non ha risorse sufficienti, bisogna optare per l'offloading verso il Cloud. Qui viene allora invocata la funzione `handleCloudOffload` adattata in precedenza, che ora, prima di inoltrare una richiesta verso il cloud, applicherà i controlli sui vincoli architetturali appena implementati.

Edge-Only Policy Infine, è stata adattata la policy Edge-Only. In questo caso si è intervenuti sulla funzione che seleziona un nodo dalla *Edge Zone* per l'esecuzione, `pickEdgeNodeForOffloading`. Nella versione originale, non essendo presenti vincoli architetturali, la funzione si limitava a scegliere il primo nodo disponibile tra i *neighbors*. Per poter operare correttamente in un contesto multi-architettura, è stata quindi estesa in modo da considerare esplicitamente l'architettura richiesta dal runtime.

In primo luogo, la funzione recupera la lista dei nodi appartenenti alla stessa zona del nodo corrente, utilizzando le informazioni progressivamente raccolte tramite l'algoritmo Vivaldi, introdotto nella Sezione 2.4. Successivamente, l'insieme viene filtrato mantenendo esclusivamente i nodi compatibili con il runtime della funzione richiesta. Tra questi viene selezionato quello con maggiore memoria libera, così da favorire un bilanciamento del carico più efficace durante l'offloading orizzontale.

Per rendere questa fase più efficiente, è stata inoltre introdotta una cache associata al nodo scelto per ciascuna funzione. In questo modo, se un nodo Serverledge deve effettuare più volte l'offloading orizzontale della stessa fun-

zione in un intervallo temporale ristretto, può riutilizzare la decisione già presa senza dover iterare nuovamente su tutti i neighbors. Questo non solo riduce il tempo di esecuzione di `pickEdgeNodeForOffloading`, ma favorisce anche i warm start sul nodo di destinazione, contribuendo ad abbassare i tempi di risposta percepiti dall'utente. La durata del *time-to-live* della cache è configurabile dall'utente; il valore di default è pari a 60 secondi, in linea con il tempo per cui un container warm rimane attivo prima di essere rimosso dal *janitor thread*.

Infine, per questa policy è stata introdotta la possibilità di effettuare un *fallback* sull'esecuzione locale qualora nessun neighbor sia disponibile a prendere in carico la richiesta. In tal caso, previa verifica della compatibilità architetturale, si procede con l'esecuzione sul nodo corrente. Poiché l'esecuzione locale non era prevista nella versione originale della policy, questa opzione è stata resa attivabile su esplicita indicazione dell'utente tramite l'apposito file di configurazione di Serverledge.

Con le modifiche apportate alle policy di offloading, i singoli nodi di Serverledge sono ora in grado di operare in totale sicurezza all'interno di un cluster eterogeneo, inoltrando le richieste senza il rischio di generare fallimenti dovuti a incompatibilità architetturali. Tuttavia, in scenari di deploy reali su larga scala, la distribuzione del traffico non è affidata esclusivamente al meccanismo *peer-to-peer* dell'offloading, ma è tipicamente gestita da un load balancer che funge da punto di ingresso e possiede una visione globale del cluster.

Il prossimo passo fondamentale consiste quindi nell'estendere questa *architecture-awareness* anche al load balancer. Come vedremo nelle prossime sezioni, l'intervento su questo componente richiede due fasi distinte: in primo luogo, sarà necessario riprogettarne la struttura interna per garantire il rispetto dei vincoli di compatibilità architetturale. Poi, una volta assicurata la correttezza dell'esecuzione, si procederà a dotarlo di una logica decisionale intelligente. L'obiettivo finale non sarà più soltanto determinare dove una funzione *può* essere eseguita, ma sfruttare attivamente l'eterogeneità hardware per decidere su quale architettura *conviene* eseguirla al fine di massimizzare le prestazioni.

Capitolo 4

Distribuzione delle Richieste Architecture-Aware

Nel capitolo precedente si è discusso di come dotare i singoli nodi della capacità di eseguire funzioni in un'infrastruttura eterogenea, intervenendo sui runtime e sulle policy di offloading per garantire la compatibilità architetturale. Tuttavia, rendere l'esecuzione di una funzione compatibile a livello di singolo nodo rappresenta solo una soluzione parziale in un'architettura distribuita. Affidare la gestione dell'eterogeneità esclusivamente ai meccanismi di rifiuto e re-inoltro (offloading) dei singoli nodi worker introdurrebbe notevoli inefficienze, generando traffico di rete superfluo e latenze maggiori.

Infatti, dal punto di vista tecnico sarebbe possibile configurare un cluster di nodi worker con offloading abilitato e associarvi un load balancer Round Robin[17]. Nello stato attuale dello sviluppo, assumendo un cluster eterogeneo, non si verificherebbero errori: i nodi utilizzerebbero l'offloading per inoltrarsi le richieste ed eseguirle su macchine con architetture compatibili.

Tuttavia, questa configurazione comporterebbe un aumento del traffico di rete, un numero maggiore di *hop* per ogni richiesta utente e, in generale, un carico

[17]Strategia di bilanciamento del carico che assegna le richieste in modo ciclico e sequenziale ai nodi disponibili, distribuendole equamente senza considerare il loro stato interno.

computazionale più elevato sull'intero cluster. Le funzioni verrebbero quindi eseguite correttamente, ma in uno scenario chiaramente subottimale.

Va inoltre osservato che questo ragionamento vale solo con offloading attivo. Se la policy dei nodi non lo prevedesse, un load balancer Round Robin potrebbe causare un *failure-rate* sensibilmente più alto. Un nodo worker selezionato in modo inconsapevole dal load balancer, se non compatibile con il runtime richiesto, non avrebbe altra scelta che scartare la richiesta, anche nel caso in cui nel cluster fossero presenti nodi pienamente idonei dal punto di vista architetturale.

Questo capitolo affronta il problema del routing delle richieste attraverso la riprogettazione del load balancer centrale di Serverledge. L'obiettivo di questa fase è costruire un'infrastruttura di smistamento architecture-aware, in grado di separare logicamente pool di risorse eterogenee e di instradare il traffico già a monte, assicurando il rispetto dei vincoli hardware prima che la richiesta raggiunga i nodi periferici del cluster.

4.1 Limiti del load balancing tradizionale in cluster eterogenei

La prima difficoltà nella gestione di un cluster multi-architettura nasce dal fatto che i meccanismi di load balancing tradizionali non siano pensati per questo scenario. Nelle architetture FaaS classiche, tipiche dei data center cloud, il load balancer lavora assumendo che tutte le risorse siano tra loro equivalenti. Con questa ipotesi di omogeneità, algoritmi classici come il Round Robin o il Least-Connections funzionano molto bene: ogni nodo worker è considerato equivalente dal punto di vista computazionale e compatibile con tutte le funzioni. Di conseguenza, la logica di smistamento può restare "agnostica" rispetto all'hardware e concentrarsi solo sulla distribuzione del traffico.

In un ambiente ibrido Edge-Cloud, però, questa ipotesi non vale più. Applicare una logica agnostica a un cluster eterogeneo rende il load balancer una fonte di inefficienza. Se ignora l'architettura dei nodi, può selezionare destinazioni

incapaci di eseguire il runtime richiesto, ad esempio inviando un container x86 a un nodo ARM.

Questo porta a continui fallback o a re-routing orizzontali costosi, riducendo i benefici dell'Edge Computing e aumentando inutilmente il traffico di rete, come visto nell'esempio precedente. Diventa quindi necessario ripensare la logica del load balancer, passando da una visione uniforme del cluster a una gestione partizionata e consapevole dell'hardware.

4.1.1 Load balancer originale di Serverledge

Il load balancer integrato nel framework Serverledge era proprio basato sull'algoritmo Round Robin. In particolare, l'implementazione presente in `internal/lb/lb.go` utilizza la libreria Echo per fornire un `ProxyBalancer`.

Un `ProxyBalancer` è un componente che funge da proxy tra il client e i nodi del cluster. Riceve le richieste in ingresso e le inoltra ai nodi disponibili secondo la logica di bilanciamento configurata (come illustrato dalla Figura 9). Oltre alla distribuzione del carico, può occuparsi di aspetti quali l'health check dei nodi, la gestione delle connessioni e la trasparenza verso il client, che non deve conoscere direttamente quale nodo eseguirà la funzione. In sostanza, il ProxyBalancer unisce il comportamento di un proxy HTTP a quello di un load balancer tradizionale.

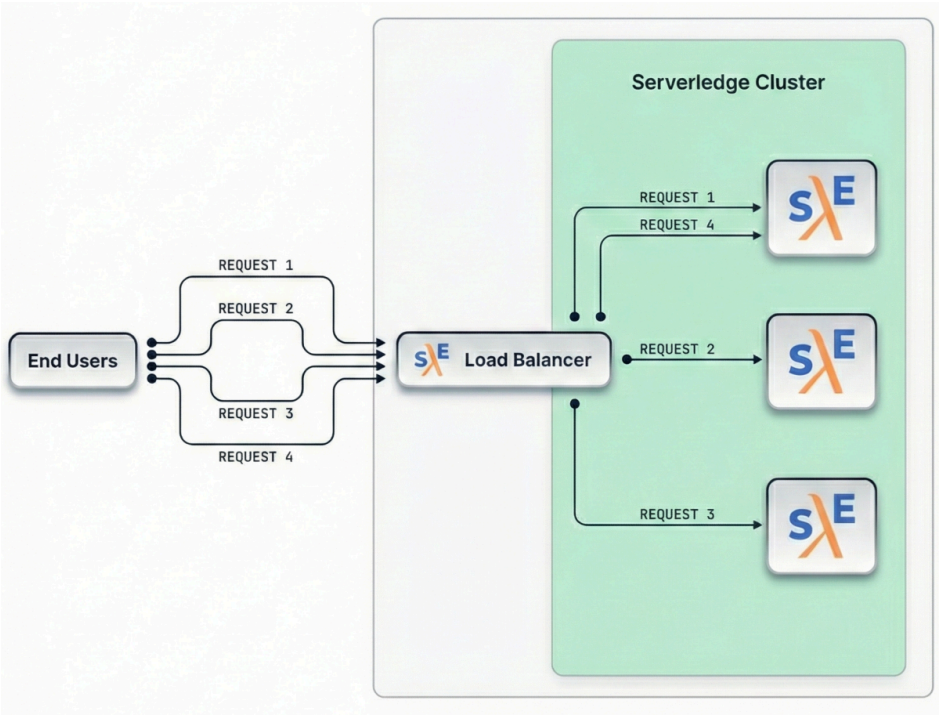

Figura 9: Load balancer Round Robin originale di Serverledge

In questa implementazione, il load balancer viene inizializzato con una lista unica di target, che include tutti i nodi del cluster a cui è associato. Una volta completata l'inizializzazione, il load balancer inizia a gestire le richieste dei client e monitora periodicamente, ogni 30 secondi, lo stato dei nodi. Questo permette di aggiungere automaticamente nuovi nodi che entrano a far parte del cluster e di rimuovere quelli che vengono eliminati o che subiscono un guasto.

4.2 Load balancer Architecture-Aware

Il primo aspetto su cui è necessario intervenire per realizzare un load balancer architecture-aware riguarda la gestione dei target. Fino a questo momento, il load balancer operava, come detto, su un'unica lista di target, dalla quale selezionava ciclicamente il nodo a cui inoltrare la richiesta. Questo approccio rappresenta un limite per il nuovo modello che si intende progettare. È infatti necessario suddividere i nodi in base all'architettura, così che il load balancer possa prima determinare l'architettura più adatta all'esecuzione della funzione

e, successivamente, selezionare il nodo migliore tra quelli appartenenti a quella categoria. Il tutto deve essere supportato da meccanismi e strutture dati che permettano al load balancer di scalare in modo efficiente, gestendo un numero elevato di richieste concorrenti senza introdurre un overhead significativo.

A tale scopo, si è deciso di utilizzare il **consistent hashing**, associando ogni architettura a uno specifico ring.

4.2.1 Consistent Hashing

In un sistema come Serverledge, dove i nodi possono entrare e uscire dalla rete (per esempio per guasti, oppure per accensioni o spegnimenti legati alla disponibilità dei dispositivi), il meccanismo con cui le richieste vengono instradate verso i nodi ha un ruolo fondamentale. L'approccio più semplice al bilanciamento del carico basato su funzioni di hash prevede l'uso dell'operazione di modulo: data una chiave k, che può essere per esempio il nome di una funzione, e un numero N di nodi disponibili, il nodo assegnato viene calcolato come $\text{hash}(k) \bmod N$.

Tuttavia, questo approccio presenta un limite importante in ambienti FaaS: se il numero di nodi N cambia (ad esempio passando da N a $N+1$ per l'aggiunta di un nuovo server), la maggior parte delle chiavi verrà riassegnata a nodi diversi, causando un fenomeno noto come *rehashing* globale. Questo implica che le associazioni tra funzioni e nodi, fino a quel momento stabili, vengono modificate. Il risultato è il rischio di invalidare inutilmente molti dei container warm già presenti sui nodi, poiché per ciascuna funzione si finirebbe per selezionare un nodo diverso rispetto al passato. Ne deriverebbe un'ondata di cold start che causerebbe picchi di latenza nel sistema.

Per risolvere questo problema, si è scelto quindi di utilizzare il consistent hashing, una tecnica introdotta originariamente da Karger et al. nel 1997 [32] per il caching web distribuito e utilizzata ancora oggi anche da sistemi su larga scala come Amazon Dynamo [33]. In ambito FaaS, questo stesso approccio è adottato con successo anche da framework consolidati come Apache Open-

Whisk per gestire la distribuzione delle invocazioni delle funzioni all'interno del cluster.

4.2.1.1 Hash Ring

L'idea fondamentale del consistent hashing è quella di mappare sia i nodi del sistema sia gli identificatori delle funzioni (in questo caso, i nomi) all'interno dello stesso spazio di indirizzamento circolare, chiamato appunto Hash Ring.

Supponiamo che la funzione di hash scelta produca valori in un intervallo $[0, 2^{32} - 1]$. Questo spazio viene trattato come un anello in cui l'ultimo valore è logicamente contiguo al primo (il valore $2^{32} - 1$ è adiacente allo 0), come illustrato dalla Figura 10.

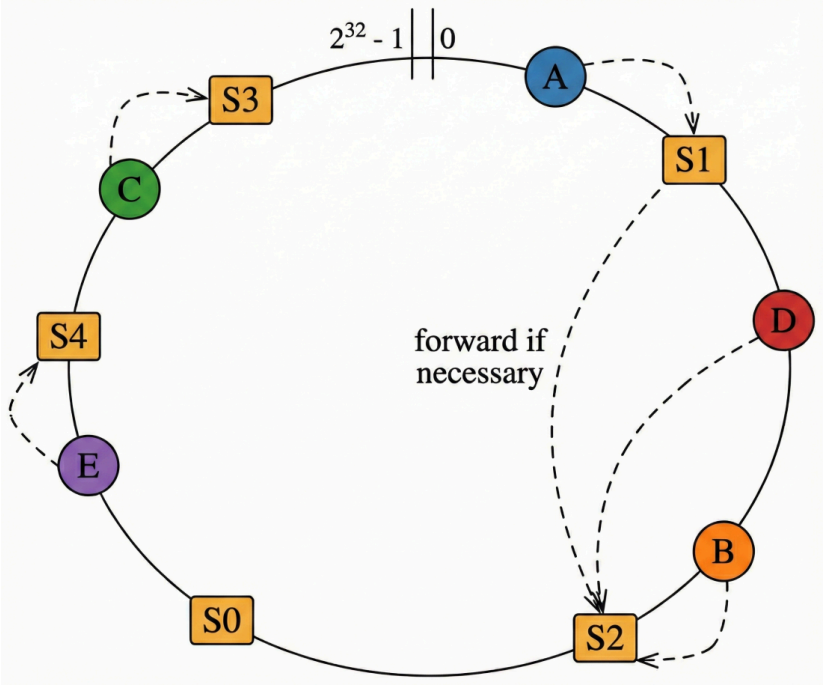

Figura 10: Schema di funzionamento di un hash ring. Gli elementi circolari rappresentano le funzioni, mentre quelli rettangolari i nodi/server

Il processo di assegnazione si scompone in due fasi:

- **Posizionamento dei nodi**: quando un nodo si unisce al cluster, il suo identificativo univoco (che in Serverledge è ottenuto dall'unione di uno `shortuuid` e il timestamp Unix ottenuto al momento della registrazione

del nodo) viene passato alla funzione di hash, ottenendo un valore che rappresenta la posizione fisica del nodo sull'anello.

- **Assegnazione delle richieste**: quando arriva una richiesta per una determinata funzione, il nome della funzione viene anch'esso *hashato*, ottenendo una posizione sull'anello. Per stabilire quale nodo debba prendere in carico la richiesta, l'algoritmo percorre l'anello in senso orario a partire dalla posizione della chiave, fermandosi non appena incontra il primo nodo disponibile.

Nell'implementazione Serverledge di questo hash ring, prima di selezionare questo nodo come target per l'esecuzione, si eseguono dei controlli sulle risorse. Se il nodo trovato ha sufficiente memoria e CPU disponibili, allora viene effettivamente selezionato, altrimenti si continua a scorrere l'anello in senso orario per trovare il prossimo nodo candidato, come si può vedere dalla Figura 10. Se terminato un giro completo dell'anello nessun nodo ha sufficienti risorse per l'esecuzione, si informa il load balancer che potrà così agire di conseguenza. È importante osservare che i controlli effettuati in questa fase riguardano esclusivamente le risorse disponibili e non l'architettura. Si assume infatti che la verifica della compatibilità tra funzione e architettura sia già stata eseguita *prima* di avviare la ricerca di un nodo candidato all'interno del relativo hash ring.

Il vantaggio principale di questo meccanismo sono la sua resistenza ai cambiamenti della topologia del cluster e la massimizzazione dei warm start. Se un nuovo nodo viene aggiunto all'anello, assumerà la responsabilità solo di quelle chiavi che si trovano tra la sua posizione e quella del nodo che lo precede in senso antiorario. Tutte le altre chiavi nel sistema rimarranno assegnate ai loro nodi originali. In maniera simile, se un nodo subisce un guasto o viene rimosso, solo le chiavi di sua competenza verranno riassegnate al nodo successivo in senso orario. In media, se il sistema è composto da N nodi, l'aggiunta o la rimozione di un nodo comporterà lo spostamento di sole $\frac{1}{N}$ delle chiavi, preservando la maggior parte dei container warm esistenti. L'utilizzo del consistent

hashing fa sì che, per ogni funzione, l'ordine con cui i nodi vengono considerati per l'esecuzione sia deterministico ma specifico: diverso tra funzioni differenti e invariato per la stessa funzione nel tempo.

Questo comportamento favorisce il riutilizzo dei container ancora warm derivanti da esecuzioni precedenti. A un livello più basso, contribuisce anche a sfruttare meglio la località dei dati e l'efficacia della cache CPU, migliorando ulteriormente l'efficienza del sistema.

4.2.1.2 Nodi Virtuali (Repliche) e Distribuzione del Carico

Pur risolvendo il problema del rehashing, l'algoritmo di base del consistent hashing presenta comunque alcune criticità. In primo luogo, posizionando i nodi in modo casuale sull'anello, è molto probabile che la loro distribuzione non sia perfettamente uniforme. Questo porta alla formazione di intervalli di anello, e quindi porzioni di chiavi, molto disomogenei, sbilanciando il carico di lavoro: un nodo a cui è toccato una porzione di anello molto ampio riceverà molte più richieste rispetto a un nodo con una porzione più stretta. In secondo luogo, l'algoritmo base non tiene conto delle differenze tra le risorse dei nodi: un dispositivo a basse prestazioni riceverebbe statisticamente la stessa quantità di richieste di un nodo sensibilmente più prestazionale, portando facilmente alla saturazione del primo.

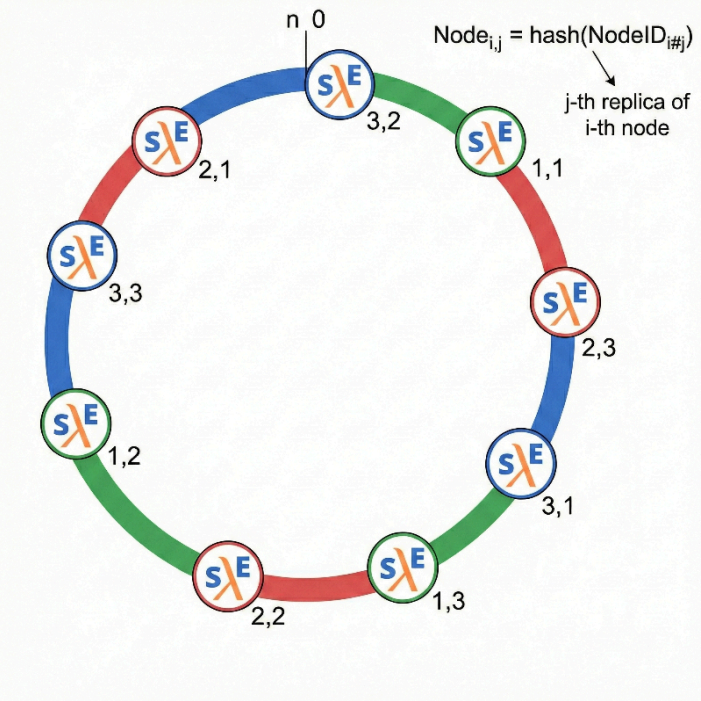

Figura 11: Schema di un hash ring con virtual nodes e 3 repliche per nodo

Per superare queste limitazioni, l'implementazione di questo meccanismo in Serverledge utilizza la tecnica dei nodi virtuali (o repliche), un'estensione già descritta nel lavoro originale sul consistent hashing [32] e successivamente adottata anche in sistemi reali, come Amazon Dynamo [33] e Apache Cassandra [34]. Invece di mappare un nodo fisico a un singolo punto sull'anello, a ogni nodo vengono associati punti multipli (i nodi virtuali), come mostrato dalla Figura 11. Questo risultato si ottiene applicando la funzione di hash a combinazioni dell'ID del nodo e di un indice incrementale, come nell'esempio del Listato 5.

```
for i := 0; i < r.replicas; i++ {
  key := fmt.Sprintf("%s#%d", node.ID, i)
  h := hash(key)
  r.ring = append(r.ring, h)
}
```

Listato 5: Esempio di generazione di `replicas` nodi virtuali per un singolo nodo fisico all'interno di un hash ring

L'impiego dei nodi virtuali offre due grandi vantaggi per il bilanciamento del carico all'interno del ring:

- **Uniformità**: Aumentando il numero di repliche per ogni nodo, i server fisici si distribuiscono sull'anello in modo comunque pseudo-casuale, ma molto più omogeneo. La varianza nella dimensione delle porzioni di chiavi assegnate a ciascun nodo fisico diminuisce notevolmente, garantendo una ripartizione del carico più bilanciata.
- **Proporzionalità**: Il numero di nodi virtuali assegnati a un server fisico può potenzialmente essere modulato in base alle sue capacità computazionali. Un nodo più potente può ricevere un numero elevato di repliche virtuali, occupando di fatto una porzione maggiore dell'anello complessivo e attirando proporzionalmente più richieste. Questo significa che si possono estendere gli hash ring per poter essere anche *capacity-aware*, ripartendo in maniera automatica il carico rispetto alle dimensioni dei nodi del ring.

4.2.2 Integrazione Multi-Architettura nel load balancer

Nel contesto dell'estensione multi-architettura realizzata per Serverledge, il meccanismo del consistent hashing è stato integrato nel load balancer per prendere parte al processo di selezione del nodo per l'esecuzione. La struttura che si va a creare è quindi quella descritta dalla Figura 12.

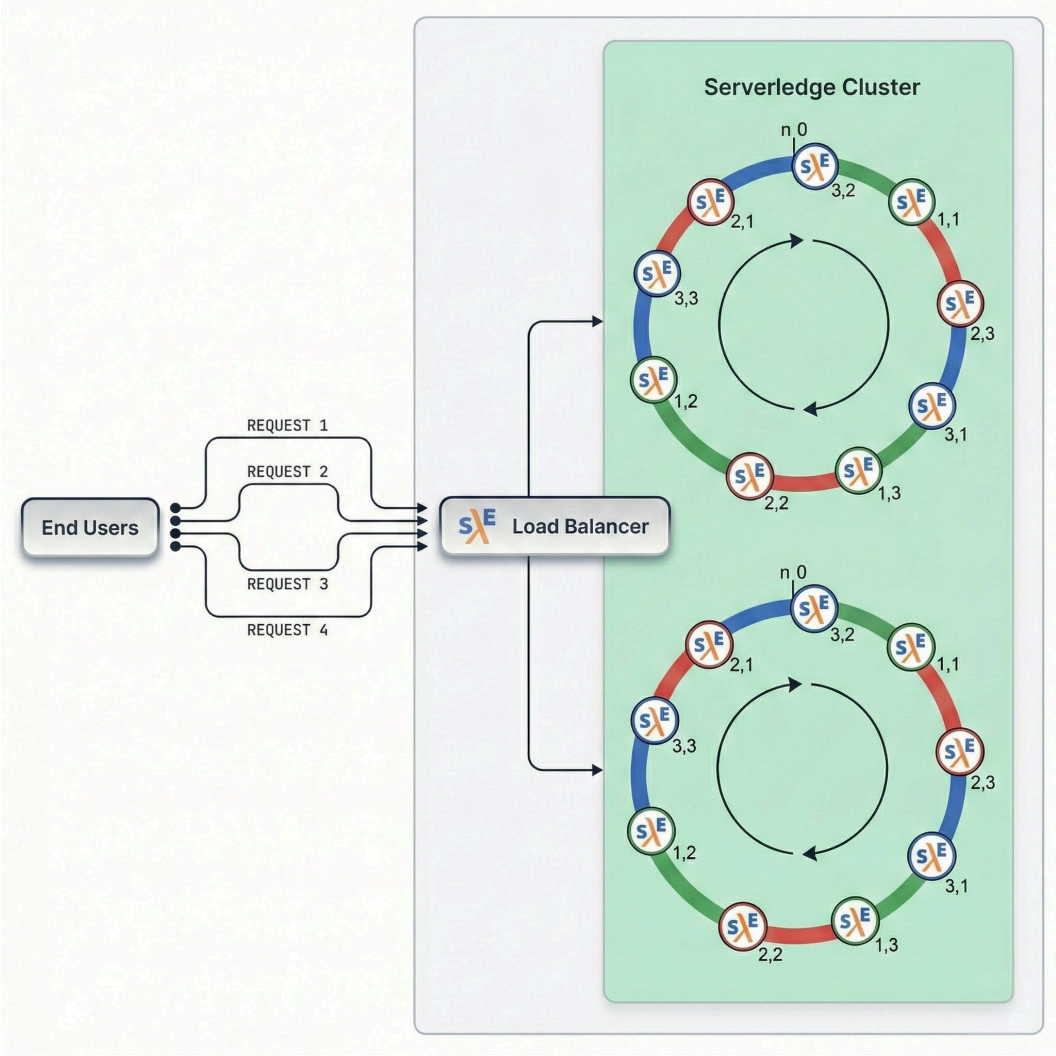

Figura 12: Load balancer Architecture-Aware realizzato con l'ausilio del consistent hashing

Come anticipato in precedenza, nel nuovo load balancer architecture-aware, la selezione del nodo su cui eseguire si suddivide in due fasi:

1. Il load balancer seleziona l'*architettura* su cui eseguire la funzione, in base alla policy su cui è stato impostato: inizialmente sono state integrate, come *baseline*, le modalità Round-Robin e Random. Nelle prossime sezioni verranno integrate anche soluzioni basate sul Reinforcement Learning e, in particolare, sui Multi-Armed Bandit. Da notare come ora il Round-Robin non sarà più effettuato a livello di singolo nodo, ma a livello di architettura. In questa fase si seleziona solo l'architettura su cui eseguire, e quindi l'hash

ring da cui selezionare il nodo, e non il nodo stesso. Durante questo processo, se il runtime della funzione dovesse supportare una sola architettura, questa verrebbe ovviamente scelta a priori, indipendentemente dalla modalità in cui è impostato il load balancer, per garantire una corretta esecuzione della funzione.

2. Per ogni architettura supportata da Serverledge, viene creato un hash ring. Tutti i nodi di un cluster vengono poi opportunamente suddivisi tra questi anelli, in base alle loro caratteristiche. In questa fase si ricerca, all'interno dell'hash ring relativo all'architettura selezionata al punto 1, un nodo su cui eseguire la funzione, seguendo i meccanismi descritti nella Sezione 4.2.1. Se nessun nodo dovesse avere sufficienti risorse per garantire l'esecuzione della funzione, il load balancer può estendere la ricerca agli altri hash ring, se compatibili con la funzione. In questo modo si massimizzano le probabilità di eseguire con successo una funzione, anche se questa non dovesse essere eseguita sull'architettura selezionata nella prima fase. Se nemmeno così è possibile trovare un nodo con risorse sufficienti all'esecuzione, non rimane altra scelta che quella di scartare la richiesta.

Questo procedimento è descritto in maniera schematica dall'Algoritmo 2, che illustra i vari step eseguiti e gli eventuali fallback utilizzati dal load balancer.

Input: F (Funzione da eseguire), P (Policy di bilanciamento), H (Lista degli Hash Ring)
Output: N (Nodo selezionato) oppure **Errore**
$A_{\text{supp}} \leftarrow \text{GetSupportedArchitectures}(F)$
if $|A_{\text{supp}}| = 1$ **then**
  $A_{\text{sel}} \leftarrow A_{\text{supp}[0]}$ ▷ *Fast-path: architettura obbligata*
else
  $A_{\text{sel}} \leftarrow \text{ApplyPolicy}(P, A_{\text{supp}})$ ▷ *Scelta architettura in base alla policy*
end
$N \leftarrow \text{SearchRing}(H[A_{\text{sel}}], F)$ ▷ *Ricerca in architettura selezionata*
if $N \neq \emptyset$ **then**
  return N
end
$A_{\text{alt}} \leftarrow A_{\text{supp}} \setminus \{A_{\text{sel}}\}$ ▷ *Fallback: estendere ricerca ad altri ring*
for each $A \in A_{\text{alt}}$ **do**
  $N \leftarrow \text{SearchRing}(H[A], F)$
  if $N \neq \emptyset$ **then**
    return N
  end
end
return DropRequest()

Algoritmo 2: Algoritmo di load balancing architecture-aware a due fasi

Il precedente load balancer, come spiegato nella Sezione 4.1.1, era realizzato tramite il `ProxyBalancer` della libreria `middleware` di Go. Per fare in modo che il nuovo load balancer potesse rimanere intercambiabile con la sua versione originale, si è deciso di strutturare la versione architecture-aware in modo da rimanere conforme all'interfaccia del `ProxyBalancer`. Per farlo, sono stati implementati gli stessi metodi offerti dai `ProxyBalancer` (`AddTarget`, `RemoveTarget` e `Next`, la funzione per selezionare il target per la prossima esecuzione). I primi due metodi sono stati realizzati in modo da funzionare in maniera trasparente con gli hash ring di ogni architettura, andando ad aggiungere e rimuovere

un nodo dal relativo hash ring assieme a tutte le sue eventuali repliche. In Serverledge, infatti, il meccanismo dei nodi virtuali è stato reso configurabile: chi esegue il deploy può decidere, tramite il file di configurazione, quante repliche avere per ogni nodo fisico negli hash ring. Impostando questo valore a 1, il meccanismo dei nodi virtuali viene disattivato. Il valore di default per il numero di nodi virtuali è impostato a 128. È stata presa questa scelta dopo aver valutato le *best practice* di altri framework che fanno uso degli stessi meccanismi. Generalmente si considera il numero ideale di repliche tra le 100 e le 256 per diminuire significativamente la varianza nella distribuzione del carico, garantendo così un'assegnazione delle chiavi, e quindi delle richieste, molto più uniforme ed equa tra i nodi fisici del cluster.

Il metodo `Next` implementa proprio le due fasi descritte in precedenza. Nella fase di ricerca di un nodo idoneo all'esecuzione della funzione, vengono controllate la memoria disponibile e il numero di CPU disponibili in quel momento per quel nodo.

Inoltre, anche la funzione per istanziare questo nuovo load balancer ha la stessa signature di quella precedente, prendendo come input un'unica lista di target. È poi la funzione di *init* del load balancer architecture-aware che individua l'architettura di ogni nodo tramite il tag associato, e lo inserisce nell'hash ring corretto.

4.2.2.1 Monitoraggio delle risorse del cluster

Per poter prendere le decisioni descritte in merito alle risorse disponibili sui nodi del cluster, il load balancer deve avere accesso a tutte queste informazioni nel momento stesso in cui è chiamato a selezionare il nodo target. Sarebbe infatti improponibile andare a pingare i singoli nodi *durante* la fase di scelta del target, perché le latenze di rete aggiungerebbero un overhead molto pesante all'intera fase decisionale. Queste informazioni devono quindi essere già note al load balancer, e devono essere mantenute aggiornate nel tempo. Si è scelto dunque di adottare un approccio misto per realizzare questo meccanismo di monitoraggio delle risorse del cluster:

- In fase di inizializzazione, quando il load balancer va a fare discovery di tutti i nodi presenti nel cluster, richiede anche ad ognuno di essi lo stato delle sue risorse. Qui può quindi ottenere informazioni sia su metriche dinamiche (come la memoria attualmente disponibile), sia su metriche statiche (come la memoria totale di un nodo). Qui è accettabile mandare una richiesta ad ogni nodo, dato che siamo in fase di inizializzazione del cluster, e dunque non si hanno vincoli stringenti sulla latenza.
- Queste informazioni devono essere mantenute aggiornate nel tempo, per questo sono stati previsti diversi punti in cui queste metriche vengono ricalcolate. Il load balancer esegue un monitoraggio periodico del cluster per verificare l'eventuale ingresso di nuovi nodi o l'uscita di nodi già presenti. Questo monitoraggio viene effettuato su una goroutine[18] secondaria che esegue a intervalli regolari, definiti ora tramite il file di configurazione. Pertanto, quando il load balancer contatta uno a uno tutti i nodi del cluster, viene fatta inviare anche una richiesta alle API per ottenere informazioni sullo stato delle risorse del nodo, in modo da poter mantenere aggiornate le informazioni che possiede. In precedenza il meccanismo per il monitoraggio usava una comunicazione *best effort* tramite protocollo UDP. Questa parte di codice è stata ristrutturata per usare TCP, garantendo una comunicazione affidabile. Per farlo si è riutilizzata l'API `/status` offerta da tutti i nodi Serverledge.
- Inoltre, queste informazioni vengono aggiornate ogni volta che un nodo viene selezionato come target. Supponiamo infatti che il load balancer abbia selezionato il nodo `NodeA` per eseguire la funzione `func1`. Una volta che la scelta è stata presa, il load balancer sa quante risorse il nodo dovrà allocare per quella funzione. Può allora andare a sottrarre dalle informazioni che mantiene in memoria sul nodo `NodeA` la quantità di memoria e il numero di CPU che `func1` utilizzerà per eseguire. In questo modo, se dovesse arrivare un'altra funzione da eseguire prima che `func1` abbia terminato la sua

[18]Una goroutine è un thread molto leggero gestito dal runtime di Go.

esecuzione su `NodeA`, le informazioni in possesso del load balancer sarebbero comunque coerenti con lo stato attuale del cluster.

- Infine, si sfrutta il fatto che il load balancer si comporti anche da proxy per i client, come spiegato nella Sezione 4.1.1. Questo significa che il risultato dell'esecuzione di una funzione su un nodo, passerà dal load balancer prima di essere inoltrato verso chi ha inviato la richiesta. Si è scelto allora di sfruttare il messaggio HTTP inviato dal nodo target al load balancer per aggiornare nuovamente le informazioni sulle risorse disponibili. Il modo meno invasivo per farlo, è quello di sfruttare gli header HTTP di questo messaggio. Il nodo target, quindi, una volta completata l'esecuzione della funzione, aggiunge al messaggio da inviare al load balancer anche una serie di header contenenti il numero di CPU disponibili in quel momento, la quantità di memoria libera, l'architettura e l'identificativo del nodo.

Quest'ultimo header è necessario affinché, alla ricezione della richiesta, il load balancer possa capire a quale nodo far corrispondere questo aggiornamento delle risorse. Per evitare ogni minimo spreco di banda, il load balancer procede a rimuovere questi header dopo averli utilizzati, per assicurarsi che non vengano inoltrati anche al client. Tutte queste operazioni sono possibili grazie all'utilizzo di un `ProxyConfig` personalizzato. Questo è sostanzialmente un costrutto della libreria `middleware` che consente di definire delle callback da eseguire quando il `ProxyBalancer` sta eseguendo la fase di proxy.

Tutte le informazioni sulle risorse dei nodi del cluster sono mantenute in una struttura dati appositamente realizzata, denominata `NodeMetricCache`. Questa struttura dati è stata realizzata avendo in mente un suo utilizzo in scenari di alta concorrenza, per assicurarsi che non finisca per essere un *bottleneck* per l'intero load balancer. Il load balancer infatti è progettato per servire ogni richiesta con una goroutine separata. La `NodeMetricCache` è stata quindi dotata di un mutex `RWMutex`, che consente di richiedere il *lock* specificando se si intende leggere o scrivere i dati. Questo consente di servire tutte le richieste di sola lettura in parallelo, andando quindi a garantire un utilizzo concorrente

delle informazioni sulle risorse. Solamente quando una goroutine deve effettuare operazioni di scrittura, è allora necessario garantire l'accesso esclusivo ai dati, consentendo un solo flusso di esecuzione per volta. È inoltre necessario considerare che gli aggiornamenti alla struttura dati delle risorse possono provenire da più goroutine per motivi diversi: ad esempio, può verificarsi quasi in contemporanea un aggiornamento dovuto al monitoraggio periodico e un altro generato dalla ricezione della risposta HTTP dallo stesso nodo.

In questo scenario, il mutex impedisce accessi in scrittura concorrenti, ma non è sufficiente a garantire la correttezza semantica degli aggiornamenti. È infatti necessario assicurarsi che un aggiornamento venga applicato solo se contiene informazioni più recenti rispetto a quelle già memorizzate.

Può accadere, ad esempio, che un aggiornamento più vecchio arrivi in ritardo a causa di latenze di rete, dopo che uno più recente è già stato elaborato. In assenza di ulteriori controlli, si rischierebbe quindi di sovrascrivere dati aggiornati con informazioni obsolete.

Per evitare questo problema è stato introdotto un timestamp associato a ogni aggiornamento. In questo modo è possibile ordinarli temporalmente e accettare esclusivamente quelli più recenti.

In questa sezione si sono discusse tutte le modifiche effettuate al componente del load balancer per renderlo architecture-aware, ovvero per consentirgli di funzionare in maniera efficace e senza generare errori non necessari in uno scenario multi-architettura, adattando le sue strutture dati e i suoi algoritmi. Nel capitolo successivo, si procederà invece ad applicare al load balancer tutte quelle modifiche che gli possono permettere di *sfruttare* al meglio il fatto di essere associato a un cluster di nodi eterogenei.

Capitolo 5

Integrazione dei Multi-Armed Bandit nel Load Balancer

In questa sezione si introducono i Multi-Armed Bandit (MAB), che sono stati integrati all'interno del load balancer architecture-aware per permettergli di sfruttare il più possibile le affinità architetturali delle funzioni da eseguire, allo scopo di migliorare i tempi di risposta per gli utenti, utilizzare in maniera efficiente le risorse e bilanciare comunque il carico in maniera equilibrata.

Si valutano quindi le possibili opzioni, si introducono a livello teorico quelle selezionate per l'implementazione e, infine, si descrive la loro applicazione all'interno di Serverledge.

5.1 Requisiti della soluzione

Prima di scendere nei dettagli dei modelli selezionati, bisogna capire a fondo qual è il problema che si sta cercando di risolvere, e quali requisiti deve rispettare la soluzione proposta.

Come discusso nella Sezione 1.2.1 e nella Sezione 2.3, le funzioni possono presentare una specifica affinità architetturale. In base al tipo di workload che eseguono, alle *system call* utilizzate e, più in generale, alla loro natura, alcune funzioni possono risultare più efficienti su una determinata architettura rispetto a un'altra.

Il load balancer architecture-aware progettato fino a questo momento risolve il problema di rendere il sistema compatibile con cluster eterogenei, con nodi suddivisi in hash ring separati per architettura. Tuttavia, nello scenario in cui una funzione possa essere eseguita sia su x86 che su ARM, non si pone il problema di scegliere quale architettura sia migliore per l'esecuzione di quella funzione. Si assicura solamente che il nodo scelto sia compatibile e abbia spazio sufficiente per allocare il runtime della funzione senza dover restituire un errore.

Il problema che invece ci si pone ora è proprio quello di rispondere alla domanda: potendo eseguire una funzione su entrambe le architetture, su quale *conviene* eseguirla, per minimizzare il tempo di esecuzione?

Come anticipato nella Sezione 1.2.1, prendere questa decisione a priori è estremamente complesso, e porta a scelte spesso poco accurate. Infatti, avendo a disposizione solamente il codice della funzione senza nessuna informazione aggiuntiva non è possibile prevedere se e con quale architettura quella funzione avrà un'affinità maggiore. Inoltre, avendo introdotto nella Sezione 3.3 anche runtime per linguaggi compilati come Go, ci sarebbero casi in cui non si avrebbe a disposizione nemmeno il codice sorgente della funzione, ma solo la sua versione binaria. Questa è una situazione che comunque si sarebbe potuta verificare anche con i runtime custom utilizzati dagli utenti.

Risulta quindi chiaro che l'algoritmo per la selezione dell'architettura ottimale non possa fare un'analisi di tipo statico, ma necessiti di raccogliere dati a runtime. Inoltre, la decisione sull'architettura su cui eseguire una funzione viene presa per ogni funzione a ogni invocazione. È quindi evidente che il processo decisionale deve aggiungere un overhead trascurabile, altrimenti finirebbe per rallentare ogni singola richiesta nel sistema, andando a vanificare tutto il guadagno che si otterrebbe eseguendo le funzioni sull'architettura a loro più affine. Il load balancer poi non è un nodo a cui è delegata la reale esecuzione delle funzioni, ed è quindi lecito immaginare che in molti scenari non sia eseguito su un nodo con particolari capacità computazionali. Questa è

un'ulteriore ragione per ricercare una soluzione che faccia uso di un algoritmo leggero ed efficiente.

5.1.1 Possibili soluzioni

Durante la progettazione del load balancer sono state considerate diverse alternative algoritmiche. In una prima fase si è valutato l'impiego di modelli di *supervised learning*, come la Regressione Lineare o le Random Forest. Questo tipo di approccio richiede però un addestramento offline su dataset storici di grandi dimensioni. Studi recenti, come quello di Chen et al. [14], hanno mostrato che è possibile predire il tempo di esecuzione su architettura ARM a partire da profilazioni effettuate su x86, raggiungendo accuratezze superiori al 90%. Tuttavia, tali risultati sono stati ottenuti su un insieme limitato di sole 18 funzioni. In un contesto FaaS reale, caratterizzato da un numero potenzialmente molto elevato di funzioni, replicare questo approccio risulterebbe problematico: dovendo operare con un numero di funzioni diverse estremamente ampio e destinato a crescere nel tempo, si rischia di rendere il dataset di addestramento troppo limitato e obsoleto, adatto solamente ad alcune tipologie di funzioni e con un'alta varianza nell'accuratezza delle predizioni.

Per superare i limiti dell'addestramento offline, si è quindi preso in considerazione il **Reinforcement Learning** (RL), che consente di apprendere direttamente dall'interazione con il sistema.

In particolare, la scelta è ricaduta sui **Multi-Armed Bandit** (MAB). Questa classe di algoritmi consente un apprendimento interamente online, senza necessità di dataset per l'addestramento, e bilancia in modo automatico l'esplorazione di nuove architetture con lo sfruttamento di quelle già note. Inoltre, la loro complessità computazionale contenuta garantisce tempi di decisione estremamente rapidi, rendendoli ottimali per l'utilizzo che si intende farne.

5.2 Multi-Armed Bandit

I Multi-Armed Bandit sono una classe di algoritmi che rappresentano uno degli approcci fondamentali nel campo del Reinforcement Learning.

L'idea alla base dei Multi-Armed Bandit deriva da una classica analogia del mondo del gioco d'azzardo: immaginiamo un giocatore di fronte a una fila di slot machine (tradizionalmente chiamate *«one-armed bandits»*). Ognuna di queste macchine ha una probabilità di vincita sconosciuta al giocatore. L'obiettivo del giocatore è massimizzare il proprio guadagno totale in un determinato numero di giocate, tirando una sola leva alla volta.

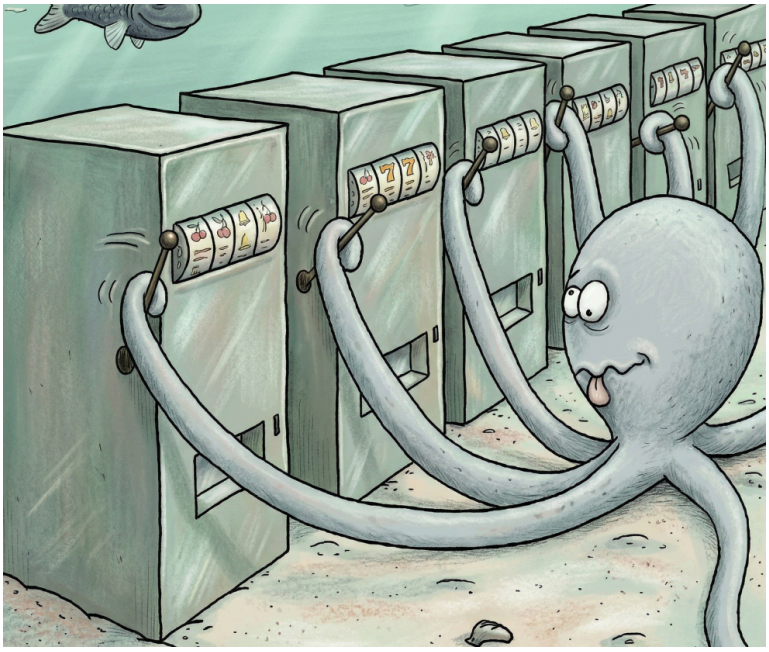

Figura 13: L'analogia da cui prendono il nome i Multi-Armed Bandit

Per raggiungere questo scopo, il giocatore affronta un problema fondamentale noto come il dilemma tra esplorazione e sfruttamento (*exploration-exploitation dilemma*):

- **Esplorazione** (Exploration): Il giocatore deve provare diverse slot machine per raccogliere informazioni sulle loro probabilità di vincita. Sebbene questo possa portare a giocate perdenti a breve termine, è l'unico modo per scoprire quale macchina sia la più redditizia.

- **Sfruttamento** (Exploitation): Una volta individuata una slot machine che sembra restituire buone vincite, il giocatore vorrà giocarci il più spesso possibile per massimizzare il profitto basandosi sulla conoscenza accumulata fino a quel momento.

Se il giocatore esplora troppo a lungo, sprecherà giocate su macchine scadenti. Se invece sfrutta troppo presto la prima macchina che restituisce una buona vincita, rischia di ignorare altre macchine potenzialmente molto più ricche, rimanendo bloccato in un ottimo locale e ignorando l'ottimo assoluto. Gli algoritmi della famiglia MAB forniscono strategie matematicamente rigorose per bilanciare dinamicamente l'esplorazione e lo sfruttamento nel tempo, con l'obiettivo di minimizzare il *regret* (il «rimpianto»), ovvero la differenza tra il guadagno effettivamente ottenuto e il guadagno massimo che si sarebbe potuto ottenere conoscendo fin dall'inizio la macchina migliore [35], [36].

5.2.0.1 Applicazioni pratiche e load balancing

Nonostante siano nati in ambito puramente teorico, i MAB sono oggi ampiamente utilizzati nell'industria per risolvere problemi di «scelte sequenziali sotto incertezza» [36]. Sono alla base di molti sistemi di raccomandazione (come quelli di Netflix o Amazon), sistemi di online advertising per la scelta del banner pubblicitario con il più alto click-through rate[19], e protocolli di routing nelle reti wireless. [37]

Più recentemente, il paradigma MAB è stato applicato con grande successo anche al campo dei sistemi distribuiti e, in particolare, al load balancing [38], [39]. In un sistema dinamico come il Cloud o l'Edge Computing, le latenze di rete, il carico sui server e le prestazioni dell'hardware possono variare imprevedibilmente. Trattare i server (o, nel nostro caso, le architetture hardware) come i «bracci» di un Multi-Armed Bandit permette al load balancer di misurare costantemente i tempi di risposta: la funzione inoltrata su un server rappre-

[19] Il click-through rate è il rapporto tra il numero di click su un contenuto e il numero di volte in cui esso è stato mostrato agli utenti.

senta la giocata, e il tempo di risposta misurato[20] costituisce la ricompensa (*reward*). Diversi lavori in letteratura dimostrano che l'uso di politiche MAB all'interno dei load balancer consente di ottenere distribuzioni del carico capaci di adattarsi spontaneamente a colli di bottiglia di rete o a prestazioni degradate da parte dei nodi [38], riducendo significativamente i ritardi di elaborazione e superando le tradizionali politiche statiche come il Round Robin. Ad esempio, nel contesto dei sistemi FaaS, è stato esplorato l'uso di MAB per selezionare adattivamente la migliore politica di instradamento in base agli obiettivi di Quality of Service (QoS) imposti dagli utenti [39].

5.2.1 MAB tradizionali e Contextual Bandit

Nelle implementazioni classiche, note come MAB senza contesto (o *context-free*), l'algoritmo valuta la bontà di un braccio basandosi esclusivamente sulla storia delle ricompense ottenute in precedenza. Un esempio celebre di questa categoria è l'algoritmo UCB1 (Upper Confidence Bound). In un MAB senza contesto, se un server X si è dimostrato mediamente veloce in passato, l'algoritmo tenderà a inviargli il traffico, assumendo che tale performance sia costante e valida in senso assoluto.

Tuttavia, in scenari complessi come l'instradamento di richieste FaaS, questa semplificazione può risultare insufficiente. La velocità di risposta di un server non è una proprietà assoluta, ma può dipendere anche da fattori aggiuntivi. Per esempio, una funzione che ha affinità con architettura ARM tende a eseguire più rapidamente su un nodo di tale architettura. Tuttavia, questo potrebbe non essere vero nel caso in cui quel nodo stia smaltendo molte richieste in parallelo, e ci sia un'alta contesa sulle sue risorse.

Per risolvere questo problema si introducono i *Contextual* MAB (MAB Contestuali). In questa variante, prima di effettuare una scelta, l'algoritmo riceve un'informazione aggiuntiva sull'ambiente o sulla richiesta, denominata appunto contesto, o stato. L'algoritmo impara così a mappare i risultati non

[20] O un suo inverso, in realtà, come vedremo più avanti dopo aver introdotto il modello matematico dei MAB.

solo sul braccio scelto, ma sulla combinazione contesto-braccio. Nel contesto della nostra implementazione, l'apprendimento contestuale, realizzato tramite l'algoritmo LinUCB [40], permette al load balancer di imparare regole sull'affinità tra funzioni e architetture, ma anche a gestire eventuali eccezioni e situazioni differenti. Questo approccio consente di sfruttare appieno l'affinità architetturale eterogenea del sistema FaaS, trasformando il load balancer in un orchestratore intelligente capace di generalizzare l'esperienza passata e di sfruttare queste conoscenze per minimizzare i tempi di risposta.

5.2.2 UCB1: Context-free MAB

L'algoritmo UCB1 (Upper Confidence Bound), introdotto da Auer et al. nel 2002 [41], rappresenta una delle più note soluzioni al problema dei Multi-Armed Bandit senza contesto. L'idea fondamentale di UCB1 è il principio dell'«ottimismo di fronte all'incertezza»: quando l'algoritmo non possiede abbastanza informazioni su una certa opzione (un braccio del bandit, o nel caso del load balancer di Serverledge, un'architettura hardware), assume ottimisticamente che questa possa essere la migliore possibile, incoraggiandone così l'esplorazione.

Il meccanismo di base è relativamente semplice, e si fonda su un'unica formula. A ogni istante di tempo (o interazione) t, l'algoritmo deve scegliere un'azione a tra un insieme di azioni possibili. Per farlo, calcola per ogni braccio un indice, chiamato Upper Confidence Bound, e seleziona il braccio che massimizza tale valore.

L'equazione che governa la scelta del braccio a al tempo t è la seguente:

$$a_t = \text{argmax}_a \left(\underbrace{Q_a}_{\text{exploitation}} + \underbrace{c\sqrt{\frac{\ln t}{n_a}}}_{\text{exploration}} \right)$$

Questa formula si scompone in due termini ben distinti, che mostrano proprio come l'algoritmo cerchi di coniugare *exploration* ed *exploitation*:

- Il termine di **Sfruttamento** (Q_a): questo è il valore atteso empirico, ovvero la media delle ricompense ottenute finora scegliendo l'azione a. Nel contesto del load balancing, la ricompensa è spesso modellata come l'inverso del tempo di esecuzione (più il tempo è basso, più la ricompensa è alta). Se un'architettura ha risposto velocemente in passato, il suo Q_a sarà elevato. Questo termine spinge l'algoritmo a sfruttare i nodi che si sono già dimostrati performanti.
- Il termine di **Esplorazione** ($c\sqrt{\frac{\ln t}{n_a}}$): questo termine rappresenta il margine di incertezza (l'intervallo di confidenza superiore). È composto da:
  - t è il numero totale di scelte effettuate finora nel sistema (considerando tutti i bracci). Al crescere di t, il logaritmo naturale $\ln(t)$ cresce molto lentamente.
  - n_a è il numero di volte in cui il braccio specifico a è stato scelto. Se un braccio è stato scelto pochissime volte, n_a sarà piccolo, e trovandosi al denominatore, farà "esplodere" il valore del termine di esplorazione.
  - c è una costante di bilanciamento (spesso posta teoricamente a $\sqrt{2}$). Maggiore è il valore di c, più l'algoritmo favorirà l'esplorazione. Più è piccolo, più l'algoritmo diventerà greedy[21], favorendo lo sfruttamento.

Come anticipato, la caratteristica di UCB1 che lo rende così utilizzato, è la sua capacità di potersi auto-bilanciare. Quando il sistema parte, l'incertezza è altissima e il termine di esplorazione domina, spingendo il bandit (e quindi, in Serverledge, il load balancer) a testare tutte le architetture disponibili. Man mano che un'architettura viene selezionata frequentemente (aumenta il suo n_a), la frazione $\frac{\ln t}{n_a}$ diventa sempre più piccola, riducendo l'incertezza e facendo diminuire il bonus di esplorazione. In questa fase, a guidare la scelta sarà quasi esclusivamente la media storica Q_a. Tuttavia, poiché il numeratore contiene $\ln t$, che continua a crescere a ogni singola richiesta, anche i bracci peggiori, non utilizzati da tempo, vedranno lentamente rialzarsi il loro termine di incertezza.

[21] Letteralmente: «avido» o «ingordo». Qui indica la tendenza a privilegiare il reward immediato, trascurando possibili benefici a lungo termine.

Quando questo termine supererà le prestazioni dell'architettura attualmente preferita, l'algoritmo darà al braccio perdente una "seconda opportunità". Questo consente al sistema di non rimanere bloccato per sempre su un ottimo locale (ad esempio a causa di un momentaneo sovraccarico dell'architettura migliore in quella situazione, o di un'esecuzione "fortunata" da parte dell'architettura meno affine). Si può vedere poi come questi meccanismi collaborino alla scelta dell'architettura nello pseudocodice dell'Algoritmo 3.

Input: A (Architetture supportate), c (Parametro di esplorazione)
State: n_a (Contatore selezioni per a), Q_a (Stima reward per a), t (Contatore totale)
for each $a \in A$ **do**
  if $n_a = 0$ **then**
    return a ▷ *Forza l'esplorazione di architetture mai provate*
  end
end
$a_t \leftarrow \text{argmax}_{a \in A}\left(Q_a + c\sqrt{\frac{\ln t}{n_a}}\right)$ ▷ *Bilancia exploitation ed exploration*
return a_t
procedure UpdateReward(a_t, r_t)
  $n_{a_t} \leftarrow n_{a_t} + 1$ ▷ *Incrementa contatore per l'architettura selezionata*
  $Q_{a_t} \leftarrow Q_{a_t} + \frac{r_t - Q_{a_t}}{n_{a_t}}$ ▷ *Aggiorna la media mobile della reward*
  $t \leftarrow t + 1$ ▷ *Incrementa il contatore temporale globale*
end

Algoritmo 3: Algoritmo UCB1 per la selezione dell'architettura

Auer et al. hanno anche dimostrato matematicamente che UCB1 garantisce un limite superiore logaritmico al regret atteso, il che significa che il suo tasso di errore si riduce asintoticamente in modo ottimale [41]. Tuttavia, il limite di UCB1 nel contesto FaaS eterogeneo è rappresentato dal fatto che non possiede un contesto. Infatti, si potrebbero avere situazioni in cui un'architettura fino a quel momento ottimale, si ritrovi in uno stato per cui non è più in grado di garantire esecuzioni mediamente più veloci. Per esempio, potrebbe ritrovarsi

ad avere un carico pesante da smaltire, che tecnicamente consente comunque di eseguire la funzione, ma non con l'efficienza che avrebbe con un sistema più scarico. Per superare questa limitazione, è necessario introdurre il concetto di contesto.

5.2.3 LinUCB: Contextual MAB lineare

Si introduce quindi la famiglia dei Contextual MAB. In questo modello, prima di effettuare la scelta, l'ambiente fornisce all'algoritmo un vettore di caratteristiche (features) osservabili, chiamato contesto. Nel load balancer di Serverledge si utilizza il contesto per far conoscere al MAB lo stato delle risorse dei nodi del cluster. In particolare, come si spiegherà in dettaglio più avanti, si darà al MAB una visione d'insieme sulle risorse dei nodi, aggregati a livello di architettura. Questo perché, ricordiamo, il MAB sceglie l'architettura, e non il nodo specifico su cui andare a eseguire, che è invece selezionato tramite il consistent hashing. Far avere al MAB informazioni a livello dei singoli nodi sarebbe quindi inutile e potenzialmente fuorviante.

L'algoritmo LinUCB (Linear Upper Confidence Bound), reso popolare da Li et al. nel 2010 [40] per l'ottimizzazione del click-through rate in *Yahoo!*, rappresenta una delle implementazioni più note di questa tipologia di MAB. L'assunzione di base del LinUCB è che esista una relazione *lineare* (da cui deriva il nome «Linear») tra il contesto fornito e la ricompensa attesa per un dato braccio.

In LinUCB, per ogni architettura hardware a, l'algoritmo mantiene e aggiorna un proprio vettore di pesi $\hat{\theta}_a$. Quando arriva una nuova richiesta caratterizzata dal vettore del contesto x_t (di dimensione d), la ricompensa attesa per l'architettura a viene stimata come il prodotto scalare tra i pesi dell'architettura e le features della funzione: $\hat{\mu}(t, a) = x_t^T \hat{\theta}_a$.

Esattamente come in UCB1, LinUCB non si affida solo alla stima attuale, ma aggiunge un intervallo di confidenza per garantire l'esplorazione. La formula per la selezione dell'azione al tempo t diventa:

$$a_t = \text{argmax}_a \left(\underbrace{x_t^T \hat{\theta}_a}_{\text{exploitation}} + \underbrace{\alpha \sqrt{x_t^T A_a^{-1} x_t}}_{\text{exploration}} \right)$$

I componenti chiave di questa equazione matematica sono:

- Il termine di **Sfruttamento Lineare** ($x_t^T \hat{\theta}_a$): Questo termine calcola la previsione della ricompensa basandosi su ciò che l'algoritmo ha imparato. Il vettore $\hat{\theta}_a$ viene stimato usando la *Ridge Regression*[22], minimizzando l'errore tra le ricompense passate e le previsioni.
- La Matrice (A_a): Per calcolare in modo efficiente la regressione e l'incertezza, l'algoritmo mantiene per ogni braccio una matrice quadrata A_a di dimensione $d \times d$ (dove d è la dimensione del vettore contesto x_t). Questa matrice, inizializzata solitamente come la matrice identità I_d, accumula le informazioni storiche sui contesti visitati per quel braccio ($A_a = A_a + x_t x_t^T$). L'inversa di questa matrice, A_a^{-1}, rappresenta l'incertezza, indicando in quali direzioni dello spazio delle features l'algoritmo ha scarsa esperienza.
- Il termine di **Esplorazione Contestuale** ($\sqrt{x_t^T A_a^{-1} x_t}$): Questa espressione matematica rappresenta la larghezza dell'intervallo di confidenza. Se il nuovo contesto x_t contiene features in direzioni poco esplorate in passato per l'architettura a, il prodotto con la matrice A_a^{-1} restituirà un valore elevato, incoraggiando LinUCB a esplorare l'architettura con questo nuovo tipo di stato.
- Il *parametro* di esplorazione (α): Analogamente al parametro c di UCB1, α è un valore che regola l'equilibrio tra sfruttamento ed esplorazione. Valori elevati di α forzano l'algoritmo a esplorare maggiormente i vari bracci, mentre un α vicino allo zero lo trasforma in un algoritmo puramente greedy che esegue solo le previsioni lineari.

[22]La Ridge Regression, nota anche come regolarizzazione L2, è uno dei diversi tipi di regolarizzazione per i modelli di regressione lineare. Corregge l'overfitting dei dati di addestramento, penalizzando pesi troppo grandi [42].

Tutti questi componenti, messi insieme, vanno a comporre l'algoritmo del MAB LinUCB, illustrato tramite lo pseudocodice dell'Algoritmo 4.

Input: A (Architetture supportate), x_t (Vettore di contesto), α (Parametro di esplorazione)
State: A_a (Matrice $d \times d$), b_a (Vettore delle reward $d \times 1$)
$p_{\max} \leftarrow -\infty$
$a_t \leftarrow \emptyset$
for each $a \in A$ **do**
  if A_a non è inizializzata **then**
    $A_a \leftarrow I_d$ ▷ *Inizializza a Matrice Identità*
    $b_a \leftarrow 0_{d\times 1}$ ▷ *Inizializza a Vettore Nullo*
  end
  $\hat{\theta}_a \leftarrow A_a^{-1} b_a$ ▷ *Calcolo coefficienti tramite Ridge Regression*
  $p_a \leftarrow x_t^T \hat{\theta}_a + \alpha\sqrt{x_t^T A_a^{-1} x_t}$ ▷ *Stima ricompensa + Incertezza*
  if $p_a > p_{\max}$ **then**
    $p_{\max} \leftarrow p_a$
    $a_t \leftarrow a$
  end
end
return a_t
procedure UpdateReward(a_t, x_t, r_t)
  $A_{a_t} \leftarrow A_{a_t} + x_t x_t^T$ ▷ *Aggiorna la matrice con le features del nuovo stato*
  $b_{a_t} \leftarrow b_{a_t} + r_t x_t$ ▷ *Aggiorna il vettore con la reward pesata sul contesto*
end

Algoritmo 4: Algoritmo LinUCB per la selezione dell'architettura basata sul contesto

5.3 Integrazione in Serverledge

Avendo definito matematicamente sia UCB1 che LinUCB, il passo successivo consiste nell'implementare questi modelli nel load balancer architecture-aware di Serverledge. Sebbene LinUCB si presenti come la soluzione migliore per ge-

stire al meglio un cluster eterogeneo, l'implementazione di UCB1 mantiene un ruolo cruciale all'interno del sistema, fungendo sia da potenziale meccanismo di fallback, sia da solida baseline comparativa priva di overhead contestuale. In questo modo, infatti, si potrà verificare che i meccanismi aggiuntivi per la gestione del contesto di LinUCB non aggiungano un overhead significativo.

La Figura 14 illustra ora come si integrano, concettualmente, i modelli MAB all'interno del load balancer architecture-aware di Serverledge.

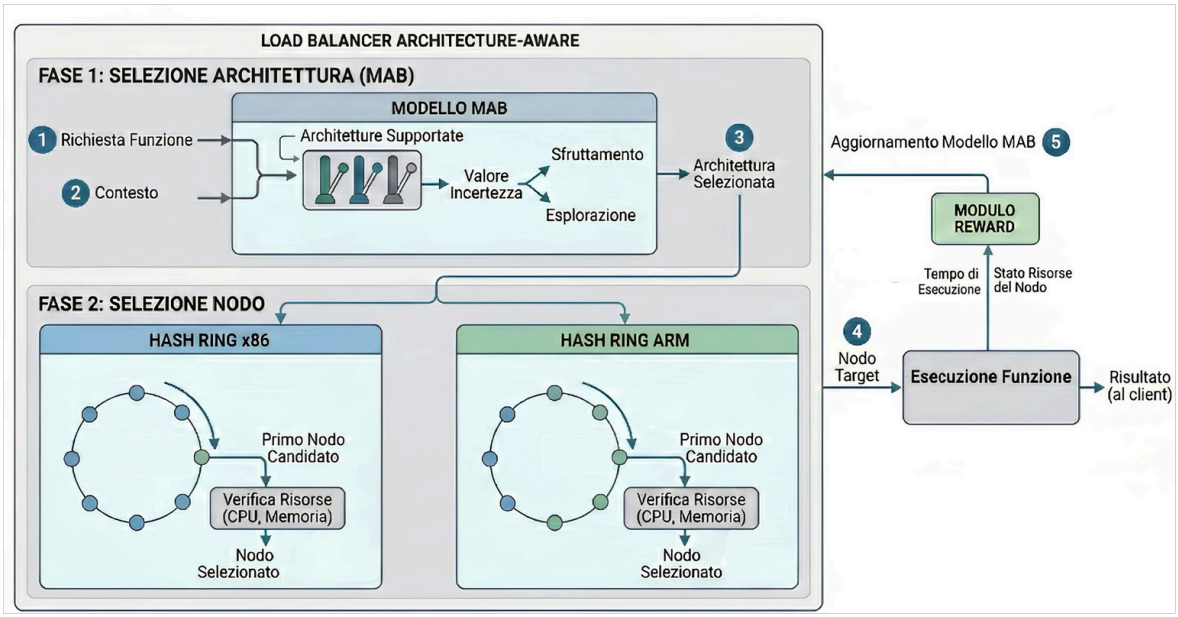

Figura 14: Schema del LB di Serverledge con integrazione dei MAB

Questa figura riassume quindi tutto il processo decisionale del load balancer:

1. Una richiesta di esecuzione di una funzione arriva al load balancer, che recupera il MAB associato alla funzione (o lo crea se è la prima invocazione di questa funzione).
2. Al MAB, se necessario, viene fornito anche il contesto. Come detto, infatti, solo LinUCB ne fa uso, mentre UCB1 è un MAB context-free.
3. Il MAB, sulla base dei dati a sua disposizione, e decidendo se fare *esplo-ration* o *exploitation*, seleziona un braccio e quindi un'architettura.
4. A questo punto, iterando sull'hash ring dell'architettura scelta, si seleziona il nodo per l'esecuzione. Nella Figura 14, per semplicità, non è rappre-

sentato il meccanismo di fallback sul secondo hash ring introdotto nella Sezione 4.2.2, che comunque è presente.

5. Al termine dell'esecuzione sul nodo selezionato, il load balancer raccoglie le metriche necessarie per aggiornare il reward del MAB relativo al braccio scelto. Parallelamente, riceve anche un aggiornamento sullo stato delle risorse del nodo, per poter mantenere una visione coerente e aggiornata dell'intero cluster.

Prima di poter procedere all'implementazione, è necessario definire analiticamente i parametri dei MAB: la funzione di reward per entrambi gli algoritmi e il vettore di contesto specifico per LinUCB.

5.3.1 Definizione analitica dei parametri

La prima scelta concettuale è stata quella di considerare come reward esclusivamente il tempo di esecuzione effettivo della funzione sul nodo, escludendo sia i tempi di cold start sia la latenza di rete per l'invio e la ricezione della richiesta. L'esclusione dei cold start è motivata dal fatto che il compito del MAB è individuare le affinità tra funzioni e architetture; includere questi eventi, rari e molto variabili, avrebbe introdotto un "rumore" poco rappresentativo. Per quanto riguarda la latenza di rete, si è assunto che, trattandosi di nodi all'interno dello stesso cluster, le differenze fossero trascurabili e quindi non fornissero informazioni utili sulle affinità architetturali.

5.3.1.1 Modellazione di UCB1

Per la definizione del reward di UCB1 inizialmente si era considerata una formulazione del tipo $\frac{1}{T_{\text{esecuzione}}}$, ma alla fine si è optato per un reward definito come:

$$\text{Reward}_{\text{UCB1}} = -\ln(T_{\text{esecuzione}})$$

È stata presa questa scelta con lo scopo di normalizzare in maniera implicita i dati per il reward. Il problema, infatti, è il peso del termine di exploitation Q_a nella formula del bandit UCB1, definita nella Sezione 5.2.2. Se i tempi di esecuzione sono molto piccoli o molto grandi, quel termine ha un peso molto

forte o molto debole rispetto al termine di exploration. Questo porta quindi il MAB, in base al tipo di funzione, a essere troppo *greedy* o troppo propenso all'exploration, finendo per imitare il comportamento dell'algoritmo Round Robin.

Con l'utilizzo del logaritmo, invece, il confronto tra i possibili reward delle due architetture diventa:

$$\Delta \text{ Reward} = (-\ln(T_{\text{ARM}})) - (-\ln(T_{\text{AMD}})) = \ln(T_{\text{AMD}}) - \ln(T_{\text{ARM}}) = \ln\left(\frac{T_{\text{AMD}}}{T_{\text{ARM}}}\right)$$

In questo modo, riducendo il confronto a un rapporto, il peso del termine Q_a è normalizzato, e il suo valore assoluto non dipende più da quanto è rapida o lunga l'esecuzione di una funzione. Questo permette al bandit UCB1 di lavorare correttamente sia con funzioni molto leggere e rapide, sia con funzioni più pesanti con tempi di esecuzione lunghi.

In particolare si è scelto il logaritmo naturale *negativo* perché in questo modo non bisogna cambiare l'obiettivo del MAB, che rimarrà quello di *massimizzare* il reward. Infatti, con l'aggiunta del segno meno, i reward saranno generalmente negativi, ma reward più piccoli (vicini allo zero) corrisponderanno a tempi di esecuzione migliori, e verranno quindi preferiti dal MAB.

Scenario	**Architettura**	$T_{\text{esecuzione}}$	**Reward** $\frac{1}{T}$	**Reward** $-\ln(T)$
Funz. veloce	ARM	1 ms	1.0	0.0
Funz. veloce	AMD	2 ms	0.5	-0.69
Confronto	Δ Reward	**-**	**0.5**	**0.69**
Funz. lenta	ARM	100 ms	0.010	-4.60
Funz. lenta	AMD	200 ms	0.005	-5.29
Confronto	Δ Reward	**-**	**0.005**	**0.69**

Tabella 1: Esempio di confronto tra diverse formulazioni per il reward

La Tabella 1 mostra un esempio di questa dinamica. Se una funzione impiega il doppio del tempo su un'architettura rispetto a un'altra, si nota che con il reward basato sul logaritmo la differenza rimane proporzionale, mentre usando

l'inverso del tempo di esecuzione le differenze diventano molto più evidenti. In altre parole, il logaritmo garantisce una stima più bilanciata del braccio da scegliere, mentre l'inverso può portare a un'eccessiva exploitation per la funzione veloce e a un'eccessiva exploration per quella lenta.

5.3.1.2 Modellazione di LinUCB

Per la modellazione del MAB LinUCB, si è partiti definendo il contesto, scegliendo di rappresentarlo basandosi sull'occupazione di memoria dei nodi, aggregata a livello di architettura. Un motivo fondamentale che ha guidato questa scelta è legato al modello di allocazione delle risorse del framework Serverledge. Nel sistema, quando un utente registra una funzione, il parametro relativo alla memoria viene sempre impostato dall'utente o, se non specificato, direttamente dalla `serverledge-cli` a un valore di default. Questo garantisce che l'aggregazione di questo dato rifletta in maniera lineare e coerente la quantità di istanze in esecuzione e l'effettivo ingombro sul nodo.

Al contrario, il parametro relativo alla CPU è progettato per supportare esecuzioni di tipo best-effort: se l'utente non specifica un vincolo, il parametro viene impostato a 0, indicando al runtime (Docker) di non applicare limiti sull'uso della CPU, permettendo al container di sfruttare tutta la potenza di calcolo disponibile sul nodo. Sebbene questa scelta ottimizzi l'utilizzo delle risorse hardware da parte del framework, rende il parametro statico relativo alla CPU inadatto a fungere da feature per il MAB LinUCB. Aggregare i valori di CPU richiesti, infatti, restituirebbe solo la somma delle risorse esplicitamente "prenotate", mascherando il carico reale generato potenzialmente da decine di funzioni in esecuzione concorrente senza limiti dichiarati. Poiché l'algoritmo LinUCB presuppone una correlazione lineare tra il contesto e il reward, fornire un dato di contesto non correlato al reale carico del sistema avrebbe potuto compromettere la capacità del modello di convergere verso decisioni ottimali.

Oltre a questa incompatibilità strutturale, la scelta di escludere l'utilizzo dinamico della CPU dal calcolo del contesto in questa prima implementazione è

motivata da necessità metodologiche e di scoping del progetto. Essendo questa la prima integrazione di un algoritmo di Reinforcement Learning contestuale all'interno del sistema Serverledge, l'obiettivo primario era validare la correttezza del modello matematico e la solidità della nuova infrastruttura creata per implementarlo. Per questo motivo, si è ritenuto importante mantenere la dimensione del vettore delle feature ridotta, isolando un singolo parametro oltre al bias. L'occupazione di memoria presenta infatti un andamento legato al ciclo di vita dei container e risulta meno soggetta alla volatilità estrema e ai picchi transitori tipici del carico CPU in ambienti FaaS. Questa scelta ha permesso di valutare l'efficacia del MAB in un ambiente più controllato, riducendo il rumore statistico e rendendo più interpretabili i risultati sperimentali.

Tuttavia, l'esclusione della CPU dal contesto di LinUCB non implica che il sistema ignori completamente il carico computazionale. Il load balancer, infatti, valuta attivamente i requisiti statici di CPU (il numero di core assegnati alla funzione) durante la fase di selezione del singolo nodo all'interno dell'hash ring, garantendo comunque un primo livello di bilanciamento delle risorse, ed evitando che un nodo rifiuti una richiesta perché «Out of Resources».

L'architettura sviluppata pone quindi delle solide basi e si presta naturalmente a future espansioni. Una volta consolidata l'efficacia della memoria come feature di contesto, gli sviluppi futuri del progetto potranno espandere il vettore di LinUCB integrando metriche dinamiche relative alla CPU o alla rete, permettendo così di valutarne empiricamente il rapporto tra il potenziale guadagno prestazionale e l'aumento della complessità computazionale.

Tornando quindi all'occupazione di memoria, che costituisce l'attuale contesto, questo dato grezzo (espresso in percentuale, come ad esempio 35.5%) non viene utilizzato direttamente dall'algoritmo, ma subisce prima una trasformazione matematica.

Il vettore delle feature passato a LinUCB ha quindi la forma $[1, \sigma(u)]$, dove σ è una funzione **non lineare** calcolata come

$$\sigma(\text{memUsage}) = \frac{1}{1 - \text{memUsage} + \varepsilon}$$

Il termine ε è un numero molto piccolo che serve ad evitare una divisione per 0 nel caso di memoria completamente occupata, prevenendo un possibile crash del programma.

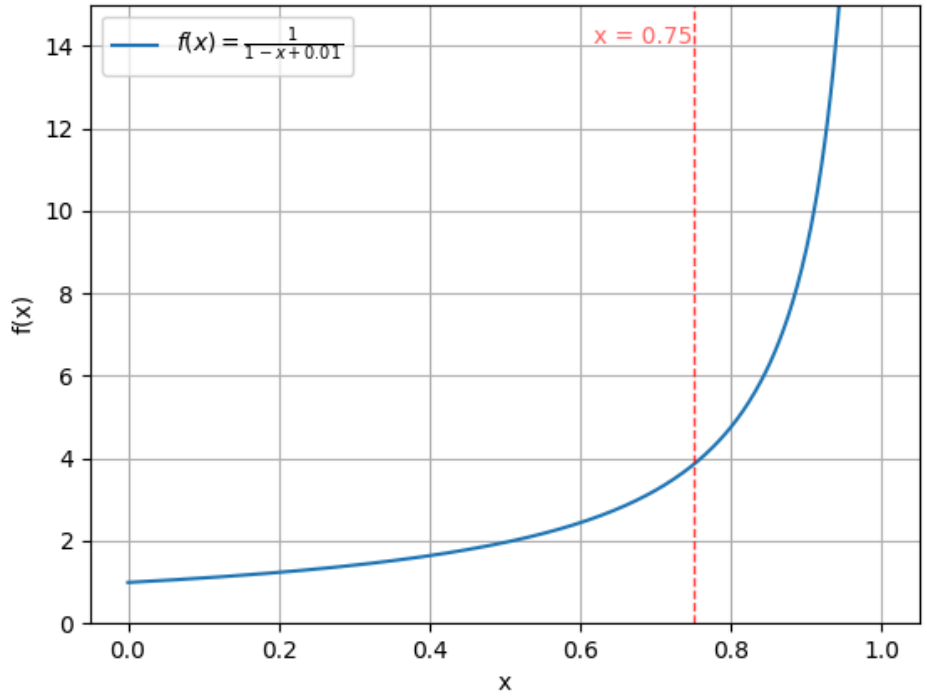

Figura 15: Grafico della funzione $\sigma(x)$ di LinUCB

Questa funzione non segue un andamento lineare, come evidente dalla Figura 15, ed è stata scelta appositamente. Infatti, in questo modo ha un valore relativamente basso fino a quando l'occupazione di memoria è circa al 75%, e poi cresce in maniera estremamente rapida oltre questa soglia. Così facendo, l'informazione che si dà al MAB è che avere una memoria piena al 95%, per esempio, non è solo *leggermente* peggio che averla piena al 75%. Infatti più la memoria dei nodi sarà satura più le loro prestazioni, verosimilmente, peggioreranno, e non lo faranno in maniera lineare.

Infine, si è definito il reward anche per questo MAB. Per gli stessi motivi illustrati in precedenza, anche la formula del reward di LinUCB si basa sul

logaritmo naturale negativo. Tuttavia, per questo secondo modello è stato introdotto un meccanismo aggiuntivo; il reward è quindi calcolato come:

$$\text{Reward}_{\text{LinUCB}} = -\ln(T_{\text{esecuzione}}) - \lambda \cdot \text{memPenalty}(\text{memUsage})$$

con la funzione memPenalty definita da:

$$\text{memPenalty}(\text{memUsage}) = \max\left(0.0, \frac{\text{memUsage} - 0.75}{1 - 0.75}\right)$$

Questa funzione, in maniera concettualmente simile a quanto fatto con la formula di $\sigma(\text{memUsage})$ utilizzata per le feature del contesto di questo MAB, cresce al crescere della memoria occupata.

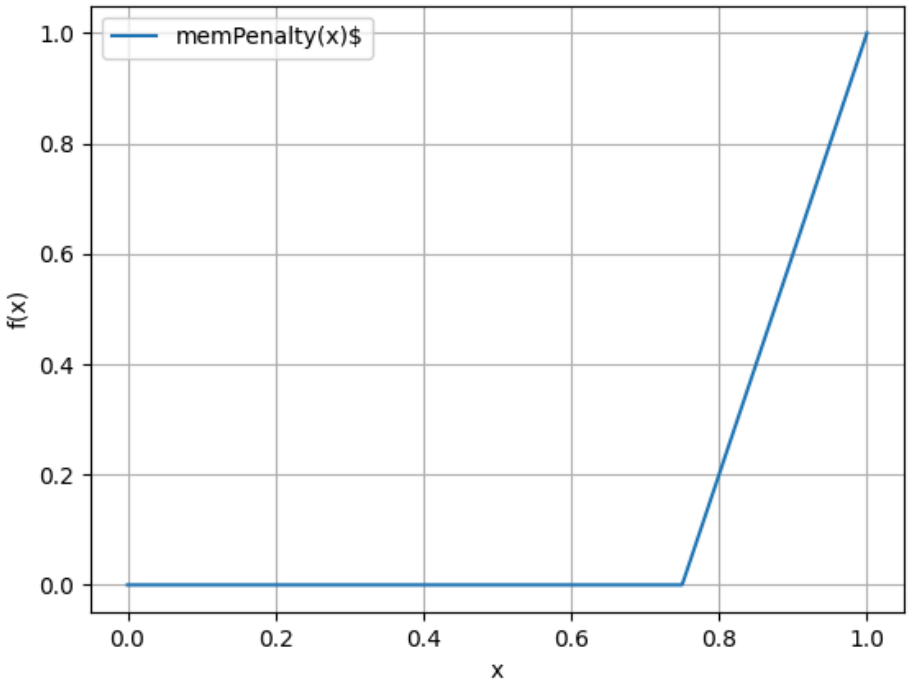

Figura 16: Funzione memPenalty per il reward del MAB LinUCB

Come mostra la Figura 16, questo grafico, a differenza di quello della Figura 15, cresce in maniera lineare a partire dalla soglia del 75%. Questa decisione è stata presa per introdurre un meccanismo di *soft-constraint* matematicamente prevedibile. Mentre la feature $\sigma(u)$ permette al modello di imparare il degrado esponenziale delle prestazioni, l'applicazione di una penalità lineare sul reward

serve a fornire un disincentivo graduale e controllato all'algoritmo. In questo modo si evitano fluttuazioni troppo estreme nella formula del reward che una penalità non lineare avrebbe potuto causare, destabilizzando il bilanciamento tra exploration ed exploitation.

Il peso di questa *memory penalty* è modellato per essere configurabile: il valore di λ verrà letto dal file di configurazione di Serverledge. Impostando $\lambda = 0$ si disattiva del tutto questa penalità, rendendo il $\text{Reward}_{\text{LinUCB}} = -\ln(T_{\text{esecuzione}})$, come nel caso di UCB1.

5.3.2 Implementazione dei MAB

Durante la fase di sviluppo del load balancer architecture-aware descritta nella Sezione 4, si era predisposto questo componente a poter lavorare in maniera intercambiabile con vari tipi di algoritmo. È stato quindi sufficiente aggiungere alla sezione che controlla il tipo di algoritmo da utilizzare, le voci relative ai MAB UCB1 e LinUCB.

5.3.3 Global Bandit Manager

Se il load balancer viene configurato in modalità `"MAB"`, a occuparsi della gestione di tutte le complessità legate ai bandit sarà il `GlobalBanditManager`. Questo componente funge da frontend per i MAB verso il load balancer stesso. In questo modo, dal load balancer si potranno utilizzare metodi generici come `GetBandit` e `bandit.SelectArm`, senza dover conoscere la tipologia di bandit in uso, e i dati di cui questo necessita (utilizzo o meno del contesto). Tutti i bandit implementano la stessa interfaccia `Policy`, in modo da poter essere utilizzati in maniera trasparente da parte del `GlobalBanditManager`. Per farlo, le signature delle funzioni di questa interfaccia prevedono la presenza di un contesto. Questo poi, a runtime, verrà effettivamente utilizzato da LinUCB, mentre sarà vuoto e ignorato se il MAB sta lavorando con l'algoritmo UCB1, essendo context-free. Ogni bandit è poi responsabile di implementare le funzioni per l'update del reward e per la selezione del braccio, secondo le caratteristiche del proprio algoritmo.

Il `GlobalBanditManager` si occupa anche di istanziare un nuovo bandit del tipo configurato ogni qual volta si riceve una richiesta per una funzione mai vista. Come detto, infatti, si avrà un bandit dedicato *per ogni* funzione.

5.3.4 UCB1

Il bandit UCB1 è implementato esattamente secondo quanto descritto dal suo modello matematico nella Sezione 5.2.2. La sua struct contiene solamente i seguenti campi:

```
  // Note: "ARM" is the architecture. "Arm", as in Multi-Armed Bandit, is a
possible architecture
  type ArmStats struct {
  Count      int64   // times we chose that arm/architecture
  SumRewards float64 // Sum of rewards
  AvgReward  float64 // Avg Reward (Q value in the formula)
}

type UCB1Bandit struct {
  TotalCounts int64                // number of total executions (t)
  Arms        map[string]*ArmStats // Map "amd64" -> Stats, "arm64" -> Stats
for each arm
  mu          sync.RWMutex         // Mutex for thread-safety
  c           float64              // Exploration parameter C
}
```

Listato 6: Strutture dati per l'implementazione di UCB1

L'occupazione di memoria dei dati presenti nel Listato 6 è riportata nella Tabella 2 (considerando un'architettura a 64-bit):

Tipo di dato / Struttura	**Dimensione approssimata**
`int64`	8 byte
`float64`	8 byte
`string` (header: pointer + len)	16 byte
`sync.RWMutex`	≈ 40 byte
`map` (header `hmap`)	≈ 48 byte
`bucket di map[string]*T (8 slot minimi)`	≈ 200 byte

Tabella 2: Occupazione di memoria approssimata delle strutture dati fondamentali in Go

La struttura `ArmStats` è composta da un `int64` e due `float64`, per un totale di 24 byte.

La struttura principale `UCB1Bandit` contiene anche:

- `TotalCounts` (8 byte),
- un puntatore alla `map` (8 byte),
- il parametro `c` (8 byte),
- un `sync.RWMutex` (≈ 40 byte).

Il totale della `struct` principale è quindi intorno a 64 byte.

Bisogna infine considerare la `map[string]*ArmStats`, di cui `UCB1Bandit` conserva un puntatore, e che viene allocata in memoria per ogni bandit UCB1. Senza scendere troppo in dettagli eccessivamente tecnici, la dimensione di una `map` di questo tipo è attorno ai 250 byte. Le due *key* `"amd64"` e `"arm64"`, che occupano 21 byte ciascuna, non vengono invece duplicate da Go per ogni struttura `UCB1Bandit`, e non serve quindi includerle nel conteggio. Queste stringhe infatti vengono conservate nella sezione dati read-only dell'eseguibile e riutilizzate per ogni allocazione.

Si ottiene una stima complessiva di circa 350 byte per istanza.

Il benchmark sperimentale effettuato tramite `runtime.MemStats` ha riportato un'allocazione media di circa 360 byte per istanza, valore coerente con questa analisi teorica, che non tiene conto di eventuali allineamenti in memoria e padding dei dati.

Numero di istanze (N)	**Memoria totale stimata**
1	≈ 360 byte
10	≈ 3 600 byte (≈ 3.5 KB)
100	≈ 36 000 byte (≈ 35 KB)
10 000	≈ 3 600 000 byte (≈ 3.4 MB)
1 000 000	≈ 360 000 000 byte (≈ 343 MB)

Tabella 3: Stima dell'occupazione di memoria in base al numero di istanze allocate

Assumendo un'occupazione media di circa 360 byte per istanza, la memoria totale cresce linearmente rispetto al numero di strutture dati allocate, come riportato nella Tabella 3.

Si può osservare che, anche in presenza di un numero molto elevato di funzioni uniche gestite dal load balancer, l'occupazione di memoria rimane complessivamente contenuta, consentendo al componente di scalare in modo efficiente senza introdurre criticità significative dal punto di vista delle risorse.

5.3.5 LinUCB

Anche per l'implementazione del MAB LinUCB si è seguito fedelmente il modello matematico definito nella Sezione 5.2.3. La formula di questo MAB prevede l'utilizzo di matrici, che comporta naturalmente un footprint in memoria maggiore e, anche per questo motivo, si è deciso di limitare il numero di feature, evitando di appesantire ulteriormente ogni istanza dell'algoritmo.

In particolare, ogni matrice ha dimensione $d \times d$, dove d è il numero di feature. Per come si è modellato il contesto, si ha allora $d = 2$, comprendendo la feature dell'occupazione di memoria e il termine di bias.

Svolgendo un benchmark simile a quello eseguito per UCB1 nella Sezione 5.3.4, si ottiene che l'occupazione di memoria di una singola istanza di LinUCB si attesta attorno ai 650 byte. Si tratta comunque di un valore contenuto, che consente al load balancer di scalare senza problemi anche su dispositivi con risorse limitate, pur risultando quasi il doppio rispetto a quello richiesto dal MAB UCB1.

Invece, per rendere i calcoli tra matrici il più veloci possibili si è fatto uso della libreria `gonum/mat`, che implementa queste operazioni in maniera efficiente.

5.3.5.1 Gestione del contesto

Per calcolare la metrica relativa alla memoria disponibile, i dati sull'occupazione della memoria dei singoli nodi sono stati aggregati per architettura, in modo da ottenere una visione complessiva dell'occupazione media su ciascun hash ring. Le informazioni provengono dalla struttura dati `NodeMetricCache` introdotta nella Sezione 4.2.2.1.

Il punto cruciale per l'implementazione di questo MAB, è stato proprio quello di dover gestire il contesto legato all'occupazione di memoria per il calcolo del reward. Il MAB, infatti, quando riceve una richiesta, valutando il contesto, l'exploration e l'exploitation seleziona un'architettura su cui eseguire. Quando poi si va a calcolare il reward (punto 5 della Figura 14), bisogna tenere conto anche qui del contesto con cui questa decisione è stata presa. L'occupazione di memoria viene quindi valutata in due momenti distinti, ma con una differenza sostanziale: quando viene considerata per scegliere l'architettura, si fa riferimento all'occupazione di memoria **attuale**. Quando invece viene utilizzata per l'aggiornamento del reward, bisogna utilizzare lo stato di occupazione della memoria che si aveva **quando la decisione è stata presa**. Utilizzando l'occupazione di memoria attuale anche in questo secondo caso, si falserebbe l'aggiornamento del reward, andando a penalizzare l'efficacia del MAB. Ad

esempio, se al momento della decisione l'architettura ARM ha il 50% di memoria libera e x86 il 70%, il MAB può scegliere ARM basandosi su queste percentuali. Quando si calcola il reward, bisogna usare proprio quei valori, e non l'occupazione attuale che magari nel frattempo è salita o scesa, altrimenti il reward rifletterebbe informazioni non corrispondenti al contesto in cui la scelta è stata fatta.

Per rendere tutto questo possibile, si è realizzato il seguente meccanismo:

1. Si è introdotta una struttura dati unica a livello di load balancer, chiamata `GlobalContextStorage`. Qui viene mantenuta una `sync.map` adatta per l'uso concorrente. Questa mappa chiave-valore utilizza una stringa come chiave e un `mab.Context` come valore. Quest'ultima struttura dati mantiene uno snapshot dello stato di occupazione della memoria degli hash ring.
2. Nel momento in cui il load balancer (in modalità MAB LinUCB) si trova a dover selezionare l'architettura su cui eseguire la richiesta corrente, esegue uno snapshot relativo alla memoria degli hash ring, e ne esegue lo `Store` nel `GlobalContextStorage`. La chiave per questa entry della mappa è generata tramite l'uso di uno `shortuuid`.
3. A questo punto il MAB LinUCB può prendere la decisione sull'architettura, e tramite consistent hashing si selezionerà un nodo per l'esecuzione. Alla richiesta HTTP inviata al nodo target, viene aggiunto anche un nuovo header `"Serverledge-MAB-Request-ID"`, il cui valore associato è proprio lo `shortuuid` generato al punto 2.
4. Il nodo target eseguirà localmente la funzione, e poi reinserirà nel pacchetto HTTP di risposta inviato al load balancer, lo stesso header `"Serverledge-MAB-Request-ID"` con lo stesso valore che aveva nella richiesta ricevuta.
5. A questo punto, quando sul load balancer si va a eseguire l'aggiornamento del reward per il MAB LinUCB, si usa questo `MAB-Request-ID` per andare a recuperare dal `GlobalContextStorage` l'entry corretta relativa allo snapshot di memoria. Con questi dati, LinUCB può effettuare un aggiornamento

del suo reward corretto e coerente con quanto osservato al momento della decisione presa.

La figura Figura 17 illustra schematicamente come interagiscano questi tre componenti per la realizzazione di questo meccanismo.

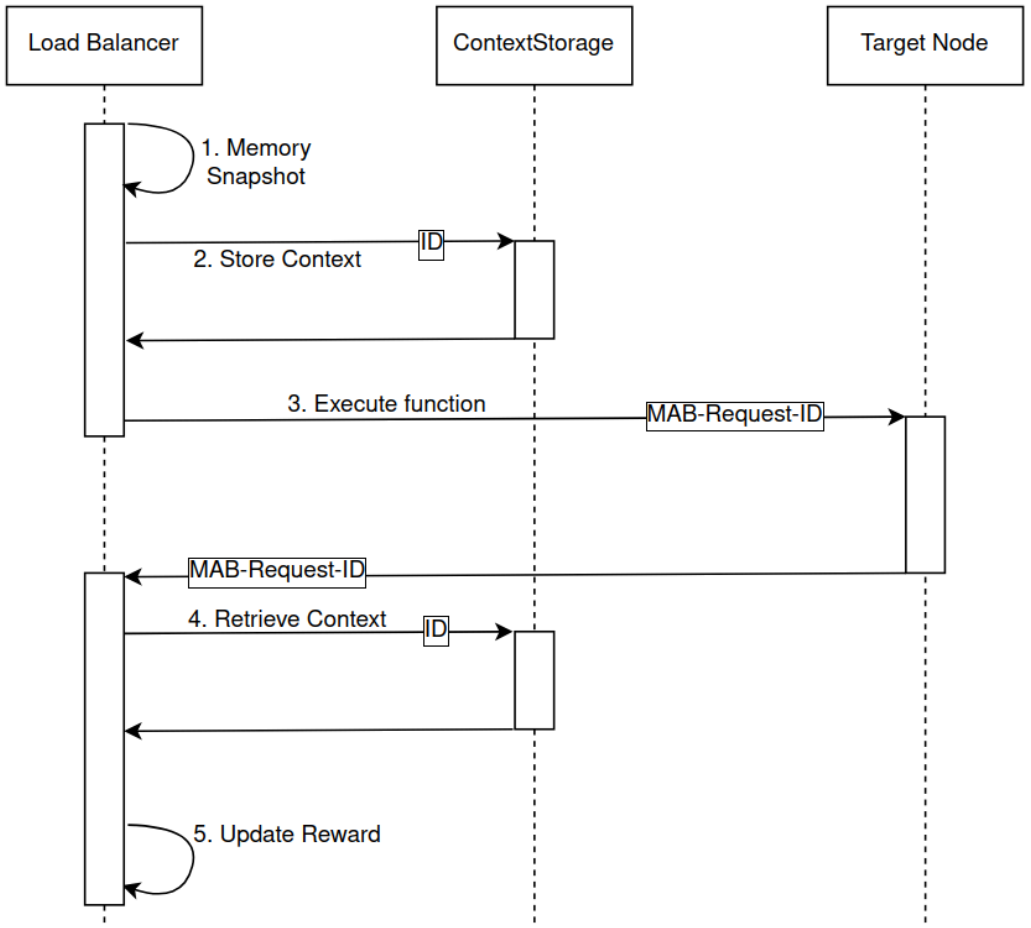

Figura 17: Sequence Diagram per illustrare l'utilizzo del MAB ID

La discontinuità nel blocco relativo al load balancer indica che l'instradamento della richiesta e l'elaborazione della risposta vengono gestiti da due goroutine separate.

Per rendere questo processo più efficiente:

- La funzione del `GlobalContextStorage` offerta per ottenere i dati dalla mappa, elimina la entry richiesta. In questo modo non è responsabilità del load balancer eliminare i dati letti da questa cache, mantenendo al minimo

l'occupazione di memoria di questa struttura dati, e prevenendo pericolosi *memory leak*.

- L'aggiornamento del MAB (sia questo UCB1 o LinUCB), viene lanciato durante la fase in cui il load balancer si comporta da proxy, con il meccanismo descritto nella Sezione 4.2.2.1. Questo aggiornamento viene eseguito tramite una goroutine, in modo da non bloccare ulteriormente la risposta che va inoltrata al client. Così facendo, non si aggiunge overhead per l'utente finale.

Con l'implementazione di questo meccanismo, si conclude la progettazione e l'integrazione dei MAB all'interno del load balancer architecture-aware. Avendo integrato sia UCB1 che LinUCB all'interno di Serverledge, il sistema è ora equipaggiato per gestire l'eterogeneità hardware, imparando a inviare le richieste verso l'architettura che minimizza i tempi di esecuzione.

Il capitolo successivo è quindi dedicato alla valutazione sperimentale della soluzione proposta.

Capitolo 6

Analisi dei risultati sperimentali

In questo capitolo si analizzeranno tutti gli esperimenti effettuati al fine di dimostrare il corretto funzionamento e l'efficacia delle soluzioni proposte. Nell'ordine, si esporrà prima il processo di verifica, eseguito parallelamente allo sviluppo del codice, e poi si introdurrà il setup sperimentale utilizzato per la fase di validazione. Infine, si analizzeranno i risultati degli esperimenti effettuati.

6.1 Verifica

Durante lo sviluppo del supporto multi-architettura di Serverledge sono state apportate modifiche significative al codice, introducendo nuovi algoritmi e componenti. Prima di procedere a una valutazione complessiva del sistema, è importante sottolineare che l'intero sviluppo è stato accompagnato da un'attività continua di testing dei singoli moduli. Infatti, sarebbe stato estremamente complesso individuare e isolare un errore in un singolo componente durante gli esperimenti finali, in cui si testa il comportamento e l'integrazione di tutti gli elementi del sistema.

Serverledge disponeva già di una propria *test suite*, che è stata eseguita regolarmente sia su architettura x86 sia su ARM, così da verificare il corretto funzionamento dell'intero framework in entrambi gli ambienti. La suite è stata inoltre estesa per verificare il corretto comportamento dei nuovi componenti

introdotti. In particolare, sono stati aggiunti test per i nuovi runtime, per la funzionalità di discovery delle architetture supportate dalle immagini dei container, e anche per la verifica del consistent hashing e del load balancer architecture-aware.

Oltre a essere eseguiti regolarmente in locale, tutti i test sono stati automatizzati tramite **GitHub Actions**. È stato infatti aggiunto un *workflow* dedicato che esegue l'intera suite di test a ogni *push* sul *repository* del progetto [43].

In questo modo è stato possibile garantire che ogni modifica introdotta non causasse regressioni e, qualora si verificassero problemi, che questi venissero individuati e corretti tempestivamente.

6.2 Setup sperimentale

Si descrive ora il setup sperimentale adottato per l'esecuzione degli esperimenti successivi. L'intera fase di valutazione è stata condotta su un cluster di macchine virtuali istanziate su Google Cloud Platform (GCP).

In totale sono state istanziate, per ogni esperimento, nove virtual machine, di cui si riportano i dettagli nella Tabella 4 sottostante:

Numero Istanze	**Ruolo**	**Architettura**	**vCPU**	**Memoria** (GB)
1	Generatore del Workload	x86	2	8
1	Load Balancer	x86	4	16
1	Global Registry	x86	2	8
3	Nodi x86	x86	4	16
3	Nodi ARM	ARM	4	16

Tabella 4: Virtual Machine utilizzate per gli esperimenti

Mentre per i «Nodi x86» e per i «Nodi ARM» l'architettura è fondamentale per l'esperimento, per quanto riguarda il generatore del workload, il load balancer e il registry è indifferente l'architettura scelta. Per questi componenti si sono

scelte VM x86, ma non sarebbe cambiato nulla utilizzando VM ARM. Queste virtual machine sono state organizzate come riportato nella Figura 18.

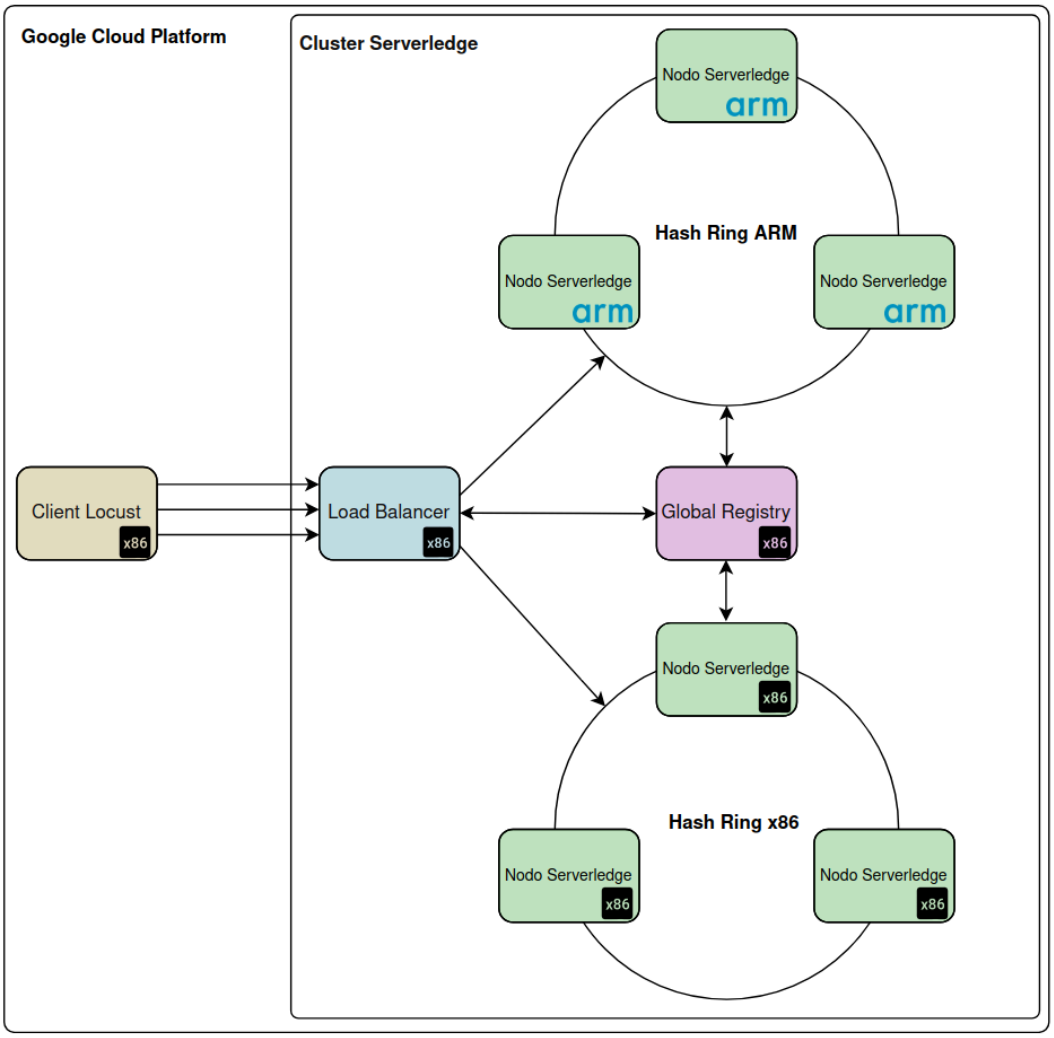

Figura 18: Organizzazione delle VM per gli esperimenti

L'obiettivo è configurare il cluster Serverledge con tre nodi per ogni architettura, gestiti da un load balancer impostato per operare con l'algoritmo da testare. I nodi Serverledge e il load balancer possono accedere al global registry per recuperare le informazioni necessarie al loro funzionamento.

All'esterno del cluster Serverledge è stata configurata una macchina virtuale, chiamata "Client Locust" nella Figura 18, incaricata di generare il workload.

Locust[23] è un tool open-source scritto in Python per test di carico di applicazioni e servizi. Permette di definire tramite codice il comportamento degli utenti simulati e di generare traffico concorrente per misurare prestazioni, latenza e capacità di scalabilità di un sistema. In questi esperimenti verrà utilizzato per simulare utenti che richiedono l'esecuzione di diverse funzioni.

Tutte le macchine virtuali sono state allocate nella stessa regione cloud di GCP, così da ridurre al minimo la latenza di rete ed evitare che potesse influenzare in modo significativo i risultati sperimentali. Per tutte le VM si è scelto come sistema operativo Ubuntu Minimal 24.04 LTS in versione `amd64` o `arm64` a seconda del caso.

Su ogni VM sono stati poi opportunamente configurati i vari componenti software, come spiegato nelle prossime sezioni.

6.2.1 Terraform e Ansible

Data la complessità della topologia di rete e il numero di macchine virtuali coinvolte, una configurazione manuale dell'ambiente di test sarebbe stata poco praticabile. Inoltre, per garantire misurazioni affidabili e non influenzate da stati o esecuzioni precedenti, si è scelto di distruggere e ricreare completamente il cluster prima della valutazione di ciascun algoritmo.

Per questo motivo, l'intero deployment dell'infrastruttura sperimentale è stato automatizzato tramite l'utilizzo combinato di Terraform e Ansible. Terraform è stato adottato secondo il paradigma *Infrastructure as Code* per il *provisioning* riproducibile delle virtual machine e delle risorse di rete su GCP. Ansible, invece, è stato impiegato per la configurazione delle macchine virtuali, automatizzando l'installazione delle dipendenze software, la predisposizione degli ambienti Docker e l'avvio dei nodi Serverledge.

Con Ansible sono stati definiti sia passaggi comuni a tutte le macchine virtuali, sia configurazioni specifiche per particolari nodi. Per rendere possibile questa distinzione, le VM create da Terraform sono state organizzate in gruppi

[23]https://locust.io/

(`Client`, `Load Balancer`, `Registry`, `Nodi x86` e `Nodi ARM`), così da applicare in modo selettivo i diversi task di configurazione.

Su tutte le VM si è proceduto a:

- installare le dipendenze di base (`git`, `make`, `python3`, ecc.);
- installare Go 1.25.3, scaricando il binario corrispondente all'architettura della VM;
- installare Docker e le sue dipendenze;
- clonare il repository con il progetto Serverledge multi-architettura;
- compilare il progetto tramite l'apposito `Makefile`.

Inoltre, sul nodo che ospita il client Locust, Ansible inserisce dinamicamente l'indirizzo IP della VM del load balancer nello script bash utilizzato per avviare l'esperimento. Questo passaggio si rende necessario perché, a ogni avvio delle VM, gli indirizzi IP vengono assegnati dinamicamente; di conseguenza, non è possibile specificare manualmente a priori gli indirizzi del registry o del load balancer nei file di configurazione.

Sulla virtual machine del registry, Ansible avvia invece un container Docker che esegue un'istanza di *Etcd*, fornendo così l'implementazione del global registry.

Sul load balancer, Ansible deve scrivere su tutti i file di configurazione utilizzati l'IP del registry. Per semplicità, infatti, si è creato un file di configurazione per ogni algoritmo del load balancer. In questo modo si può avviare questo componente di Serverledge con l'algoritmo scelto, semplicemente cambiando quale file di configurazione si passa come parametro.

Infine, sui nodi Serverledge, la procedura di deployment automatizzata:

- scrive sul file di configurazione l'IP del global registry;
- avvia il processo di Serverledge con il comando `"./bin/serverledge config_file.yaml"` all'interno di una sessione `tmux`, utile per recuperare i log durante eventuali operazioni di debugging;
- utilizza la `serverledge-cli`, esclusivamente sul primo nodo avviato, per registrare nel sistema tutte le funzioni impiegate negli esperimenti. Eseguire questo comando su tutte le macchine risulterebbe infatti superfluo e gene-

rerebbe traffico di rete inutile, oltre a messaggi di errore, poiché i nodi successivi riceverebbero il warning «function already exists».

L'unica fase della configurazione eseguita manualmente è quella in cui si va prima ad avviare il load balancer, e poi ad avviare lo script che lancia gli utenti Locust. Questo perché, in base all'esperimento che si sta svolgendo, si vuole poter scegliere in che modalità lanciare il load balancer. Solo a quel punto, è possibile lanciare l'esperimento.

6.2.2 Parametri

Nella Tabella 5 si riportano tutti i parametri configurabili per un esperimento, i loro possibili valori e quello di default.

Parametro	Default	Descrizione
`LOCUST_DURATION`	10m	Durata dell'esperimento.
`LB_POLICY`	RoundRobin	Il LB può essere impostato con: `RoundRobin`, `Random`, `UCB1`, `LinUCB`.
`USERS`	9	Numero totale di utenti Locust da creare. Ripartiti equamente tra i tipi di utente scelti per un esperimento.
`wait_time`	0.0	Espresso in secondi. Tempo che ogni utente Locust attende prima di inviare una nuova richiesta. Configurabile a livello di singola tipologia di utente.
lb.refresh_interval	30	Intervallo in secondi per il monitoraggio effettuato dal LB.
`lb.replicas`	128	Numero di virtual nodes per l'hash ring.
`mab.ucb1.c`	0.8	Parametro `c` di UCB1. Bilancia il peso dell'exploration rispetto all'exploitation.
`mab.linucb.alpha`	0.1	Parametro α di LinUCB. Bilancia il peso dell'exploration rispetto all'exploitation.
`mab.linucb.lambda`	0.7	Parametro λ di LinUCB. Determina il peso della funzione di memory-penalty per il reward.

Tabella 5: Parametri configurabili per gli esperimenti

6.2.2.1 Funzioni

In Locust si andranno a definire delle *tipologie* di utenti. A ciascuna tipologia di utente viene poi assegnata una funzione. Gli utenti di una determinata categoria, quindi, invieranno una richiesta per la funzione a essi associata, attenderanno il risultato, aspetteranno per `wait_time` secondi, e poi invieranno una nuova richiesta.

Per gli esperimenti sono state definite tre funzioni che generano carico sintetico, come segue:

- `amd_faster`: una funzione che esegue delle operazioni *cpu-bound* e che, in media, viene eseguita più velocemente su architettura x86 rispetto a quelle ARM. Utilizza il runtime Go implementato nella Sezione 3.3.
- `arm_faster`: come la precedente ma, come suggerisce il nome, esegue in maniera mediamente più rapida su ARM. Anche questa è una funzione che svolge operazioni *cpu-bound*.
- `amd_only`: questa funzione, anch'essa scritta in Go, fa tuttavia uso di un runtime custom, che ha un'immagine disponibile solo per architetture x86. Esegue un carico di lavoro più pesante, andando ad allocare della memoria, scrivendone il contenuto e poi calcolandone ricorsivamente l'hash `SHA256`. Può eseguire per un intervallo di tempo configurabile.

La progettazione di queste prime due funzioni nasce da una precisa necessità metodologica: creare le condizioni di eterogeneità prestazionale per fare in modo che gli algoritmi MAB possano dimostrare la loro utilità rispetto alle *baseline* (RoundRobin e Random). Un confronto ha senso solo se esiste la possibilità teorica di ottenere prestazioni migliori: se il comportamento ottimale coincide già con quello del Round Robin, non vi è alcuno spazio di miglioramento e quindi nessuna reale differenza da misurare.

In generale, un MAB dovrebbe comportarsi in modo equivalente o migliore rispetto al Round Robin. Se non esistono affinità tra funzioni e architetture, ovvero se i tempi di esecuzione risultano comparabili su entrambe, il MAB non può fare meglio: i reward associati ai diversi bracci tenderanno a essere simili e l'algoritmo finirà per alternarli, replicando di fatto il comportamento del Round Robin. In questo caso, il throughput complessivo sarà pressoché analogo indipendentemente dalla politica di instradamento.

Questo non rappresenta un limite dei MAB né un problema del framework, ma semplicemente uno scenario in cui non esiste margine di ottimizzazione. Situazioni di questo tipo sono state comunque considerate nella fase di verifica, per assicurarsi del corretto funzionamento dei componenti. Tuttavia, ai fini di un'analisi comparativa approfondita, risultano poco significative, poiché

in assenza di affinità architetturali non c'è spazio concreto per superare le baseline.

La funzione `amd_only`, invece, è stata progettata per analizzare il comportamento dei diversi algoritmi in scenari in cui una delle due architetture risulti temporaneamente più carica dell'altra. A questo scopo viene utilizzato un runtime custom che supporta esclusivamente una singola architettura. È stato scelto un runtime x86 per rendere lo scenario più realistico, data la maggiore diffusione di questa architettura. Tuttavia, dal punto di vista concettuale, non vi sarebbe alcuna differenza se `amd_only` fosse invece eseguibile solamente su ARM.

Il tempo di esecuzione della funzione è configurabile, così da poter generare scenari con livelli di carico differenti. Il carico complessivo può essere modulato gestendo la durata dell'esecuzione e l'intervallo tra due invocazioni successive, configurato tramite il parametro `wait_time` descritto nella Tabella 5.

6.3 Esperimento 1: scenario statico

L'obiettivo di questo primo esperimento è confrontare i quattro algoritmi (Random, RoundRobin, UCB1 e LinUCB) in uno scenario statico, valutandone le prestazioni quando il sistema riceve richieste per le funzioni `amd_faster` e `arm_faster`.

Per questo setup sono state adottate le condizioni descritte in precedenza, riducendo però il numero di utenti a 6. In questa fase non è stata utilizzata la funzione `amd_only`, poiché l'obiettivo era analizzare un contesto statico e bilanciato. Gli utenti Locust sono stati quindi distribuiti equamente, in modo da generare lo stesso carico per entrambe le funzioni considerate.

In questo scenario ci si aspetta che i MAB riescano progressivamente a identificare l'architettura più adatta per ciascuna funzione, indirizzando le richieste in modo coerente con le rispettive affinità. In particolare, rispetto a Random e RoundRobin, ci si attende una riduzione dei tempi medi di risposta e un throughput maggiore. In questa situazione non ci si aspetta che LinUCB possa

fare la differenza rispetto a UCB1, trovandosi in condizioni di utilizzazione statica.

6.3.1 Esperimento 1: Risultati

Si comincia l'analisi dei risultati con i grafici relativi al numero di richieste completate.

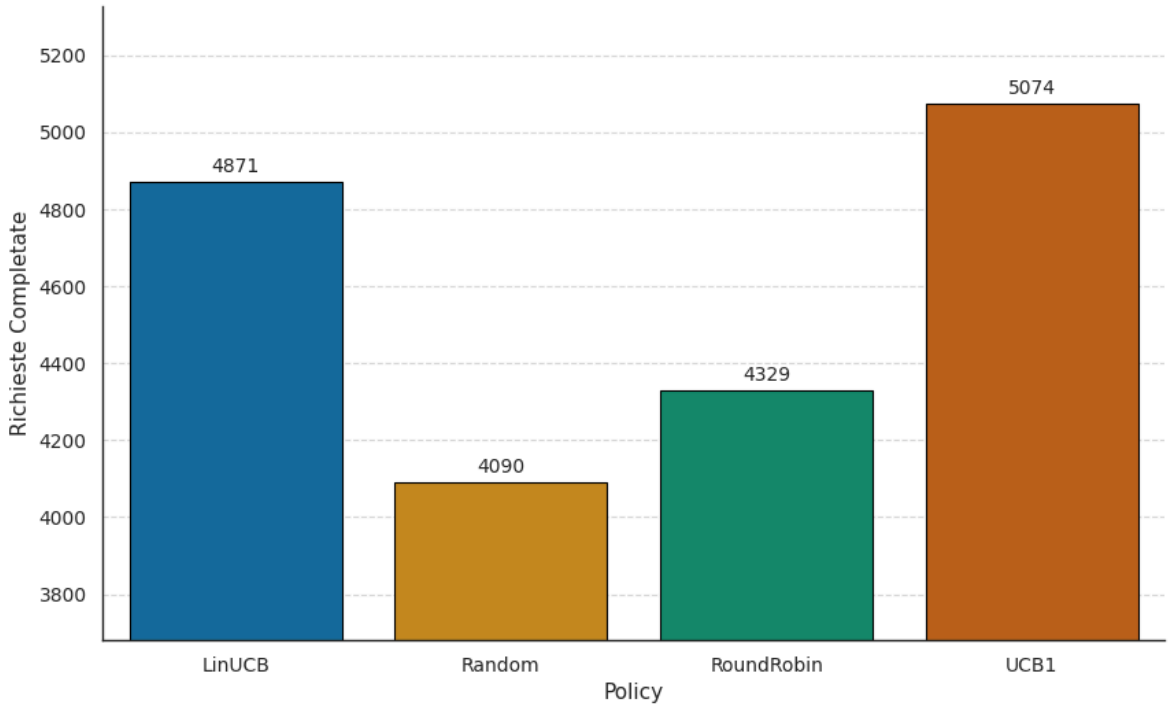

Figura 19: Grafico a barre delle richieste completate per l'esperimento con scenario statico

Analizzando la Figura 19, è subito evidente come i MAB, già in questo scenario statico, si siano comportati nettamente meglio rispetto alle baseline, riuscendo a completare un numero di richieste significativamente più alto.

In questo scenario, UCB1 ottiene addirittura risultati leggermente migliori di LinUCB. Si tratta di un comportamento coerente con il contesto dell'esperimento: l'informazione aggiuntiva fornita dal contesto sulla memoria non offre un vantaggio significativo, poiché il sistema si trova in una condizione statica e ideale.

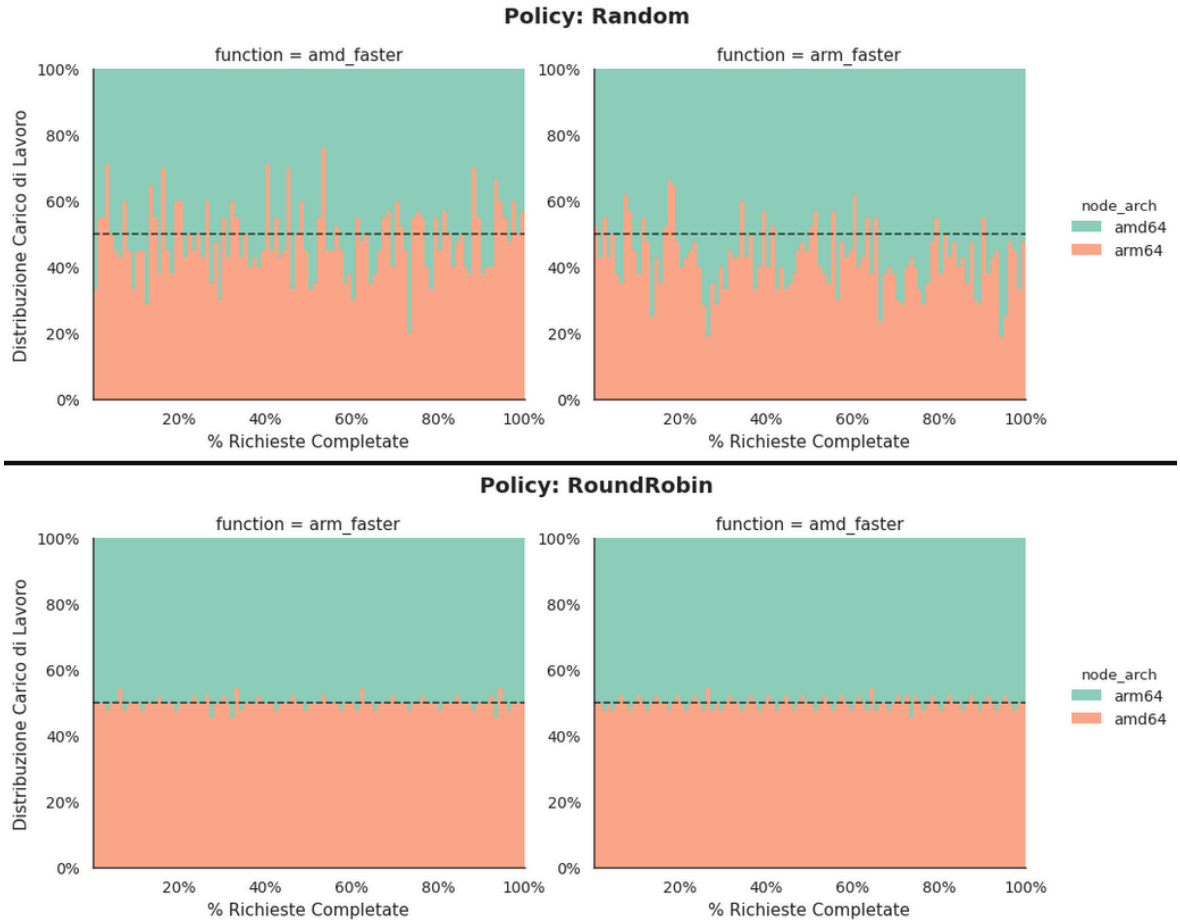

Figura 20: Distribuzione delle richieste sulle due architetture per le policy Random e RoundRobin

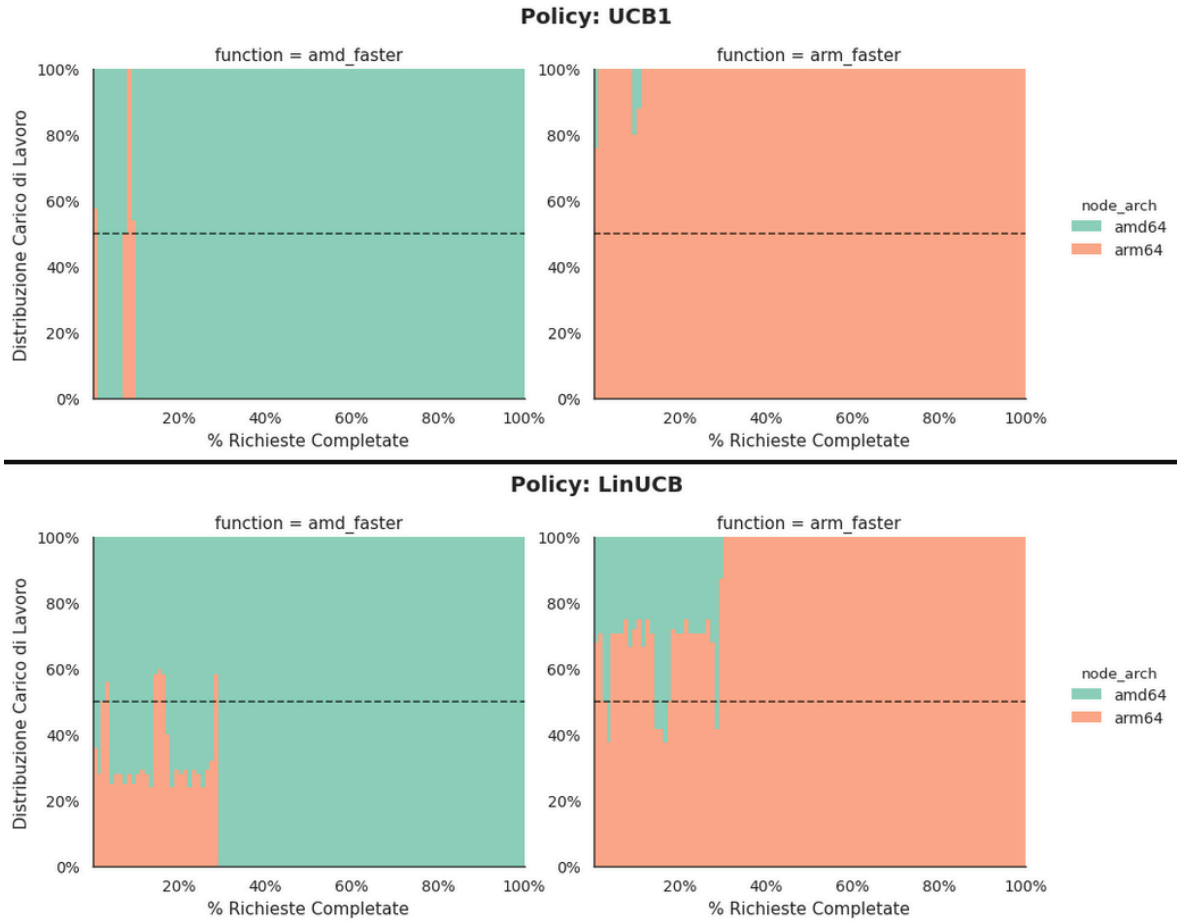

Figura 21: Distribuzione delle richieste sulle due architetture per le policy UCB1 e LinUCB

I grafici della Figura 20 e della Figura 21 servono a rappresentare come le varie policy abbiano selezionato le architetture su cui eseguire le richieste per ogni funzione **nel tempo**. Questo aspetto è particolarmente interessante da analizzare, poiché consente di osservare se e in che misura i MAB riescano ad apprendere le affinità di ciascuna funzione rispetto alle diverse architetture. Nel grafico, l'asse x rappresenta la percentuale di richieste completate per l'intera durata dell'esperimento. Si è scelto di esprimere questa quantità in percentuale poiché le policy hanno completato un numero di richieste differenti nell'arco dei 10 minuti della durata dell'esperimento. In questo modo il confronto risulta più equo. Sull'asse y, invece, è rappresentato, in percentuale, quante richieste

per quella funzione siano state instradate ed eseguite su x86 (`amd64`) piuttosto che su ARM (`arm64`).

Per RoundRobin l'andamento è esattamente quello che ci si aspetta: per entrambe le funzioni le richieste sono distribuite equamente tra le due architetture. La policy Random ha un andamento più irregolare, con picchi prima su una e poi sull'altra architettura, come lecito attendersi. Tuttavia, essendo basato comunque su un generatore pseudo-randomico, la distribuzione delle richieste, nel complesso, non si discosta troppo da quella di RoundRobin.

Infine, questo grafico mostra il motivo per cui UCB1 ha completato un numero di richieste leggermente superiore rispetto a LinUCB, come già osservato nei grafici precedenti. La fase in cui UCB1 fa anche *exploration* risulta infatti più breve, il che gli consente di privilegiare prima l'*exploitation*. LinUCB, al contrario, adotta un approccio più prudente e mantiene una fase di esplorazione più prolungata. Questo effetto non è frutto solamente del tipo di MAB utilizzato, ma anche della configurazione dei suoi parametri. Utilizzando un valore di `c` più elevato per UCB1 e un valore di α più basso per LinUCB (si veda la Tabella 5 per i dettagli su questi parametri), il primo algoritmo diventerebbe più conservativo e il secondo più aggressivo, ottenendo con ogni probabilità l'effetto opposto: UCB1 che fa maggior esplorazione rispetto a LinUCB.

Inoltre, bisogna tenere conto che in questa situazione statica e ideale, il contesto di LinUCB non fornisce un'informazione aggiuntiva, ma comunque richiede più iterazioni per convergere con la stessa confidenza del modello context-free.

È comunque importante sottolineare che, nel lungo periodo, entrambi gli algoritmi convergono verso le architetture con cui le funzioni mostrano maggiore affinità, dimostrando la capacità di sfruttare efficacemente l'eterogeneità del cluster.

La Figura 22 illustra la distribuzione statistica dei tempi di esecuzione per le due funzioni dell'esperimento, categorizzati per ogni policy di scheduling analizzata. Per evidenziare con maggiore precisione la densità e l'andamento

dei dati, ai box plot è stato sovrapposto uno *stripplot*[24], fornendo così una visione ancora più precisa della variabilità delle latenze. Da questo grafico emerge chiaramente come le policy basate sui MAB riescano a garantire tempi di esecuzione più bassi per le richieste, rispetto a Random e RoundRobin, che invece mostrano una variabilità molto più alta, come testimoniato da box plot più estesi e da una maggiore dispersione dei tempi di esecuzione.

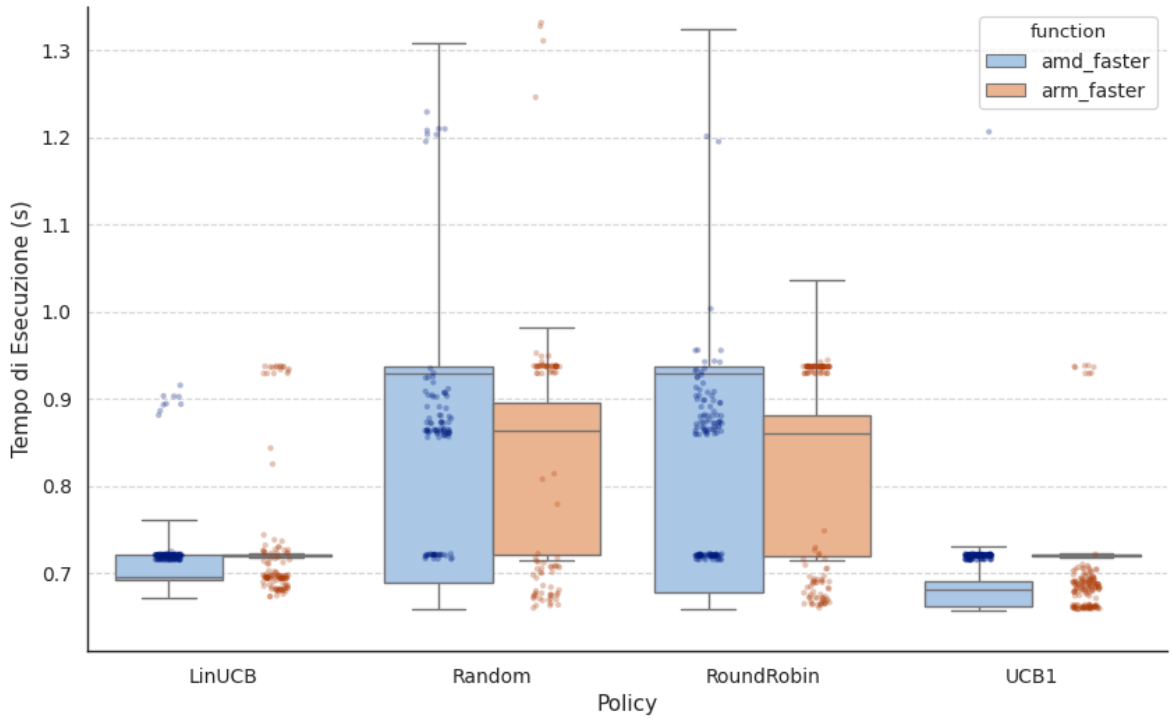

Figura 22: Boxplot e stripplot dei tempi di esecuzione per l'esperimento con scenario statico

Si riportano infine nelle tabelle sottostanti i tempi medi di esecuzione di ogni funzione per ogni policy, e il throughput totale ottenuto. Nella Tabella 7 è possibile notare come, con le policy dei MAB, entrambe le funzioni sperimentino tempi di esecuzione medi più bassi, potendo beneficiare di un maggior numero di esecuzioni sull'architettura a loro più affine.

[24]Uno stripplot è una rappresentazione grafica in cui i singoli punti dei dati vengono disegnati lungo un asse, permettendo di visualizzare direttamente la distribuzione e gli eventuali cluster di osservazioni.

Policy	Richieste Completate	Throughput (req/s)
LinUCB	4871	8.14
Random	4090	6.83
RoundRobin	4329	7.23
UCB1	5074	8.47

Tabella 6: Throughput e volume di richieste gestite dalle varie policy nello scenario statico

Policy	amd_faster (s)	arm_faster (s)
LinUCB	0.73	0.74
Random	0.88	0.87
RoundRobin	0.83	0.82
UCB1	0.69	0.72

Tabella 7: Tempi medi di esecuzione per ogni tipologia di funzione e policy analizzata nello scenario statico

Policy	Richieste	vs Random	vs RoundRobin
UCB1	5074	+24.06%	+17.21%
LinUCB	4871	+19.10%	+12.52%
RoundRobin	4329	+5.84%	–
Random	4090	–	−5.52%

Tabella 8: Analisi comparativa sul numero di richieste completate da MAB e baseline nello scenario statico

I dati riportati nella Tabella 8 sono il risultato concreto del lavoro di riprogettazione e sviluppo affrontato finora, confermando in modo evidente l'efficacia della soluzione proposta. Si è riusciti a ottenere un incremento del throughput

complessivo nell'ordine del 15–20% rispetto alle politiche tradizionali. Questi numeri dimostrano che l'integrazione dei Multi-Armed Bandit all'interno di Serverledge ha portato a un reale e tangibile vantaggio prestazionale.

Il load balancer, infatti, non si limita più a instradare le richieste ignorando l'eterogeneità del cluster, ma ottimizza attivamente l'allocazione. I risultati confermano che i MAB sono effettivamente in grado di imparare a runtime le affinità tra le singole funzioni e l'hardware sottostante, premiando le architetture più veloci. Riescono a raggiungere questi risultati senza conoscere a priori le architetture disponibili, la tipologia di funzioni o il loro codice, né eventuali affinità con le architetture. Tutto il lavoro infrastrutturale svolto inizialmente per rendere il framework compatibile con l'architettura ARM trova qui la sua validazione definitiva: l'eterogeneità del cluster non rappresenta più un ostacolo o un limite da gestire, ma è diventata a tutti gli effetti una risorsa attivamente sfruttata per minimizzare i tempi di risposta e massimizzare l'efficienza dell'intero sistema.

6.4 Esperimento 2: scenario con carico medio

In questo secondo esperimento, il carico complessivo viene incrementato portando il numero di utenti su Locust da 6 a 9, suddivisi equamente in tre categorie:

- 3 utenti per `amd_faster`: come nell'esperimento precedente, il tempo di attesa tra il completamento di una richiesta e l'invio della successiva è impostato a 0 secondi, in modo da massimizzare il carico di base.
- 3 utenti per `arm_faster`: configurati esattamente con le stesse modalità della categoria precedente, senza tempi di attesa.
- 3 utenti per `amd_only`: per generare un carico variabile, ma non eccessivo, questa funzione è configurata per un'esecuzione di 10 secondi a ogni invocazione, seguita da un tempo di attesa di 30 secondi. Si sono scelte tempistiche deterministiche al fine di garantire condizioni di test riproducibili e ottenere misurazioni rigorosamente confrontabili tra i diversi algoritmi. Se, infatti,

le richieste di questa specifica categoria di utenti non fossero state allineate temporalmente, non avrebbero generato un carico omogeneo *sull'intero pool* di nodi x86. In un simile scenario, i tempi di esecuzione rilevati sarebbero dipesi in gran parte dal *singolo* nodo selezionato dal meccanismo di consistent hashing. Questo avrebbe introdotto un'ulteriore variabile nel sistema, spostando il focus dell'esperimento dalla scelta dell'architettura (governata dal load balancer) all'efficienza del routing interno. Di conseguenza, il confronto tra le diverse policy di bilanciamento sarebbe risultato molto più imprevedibile e soggetto a un'eccessiva variabilità.

L'obiettivo di questo esperimento è analizzare il comportamento degli algoritmi in uno scenario in cui l'architettura x86 risulta periodicamente più satura rispetto a quella ARM. L'introduzione della funzione `amd_only`, infatti, genera picchi regolari di lavoro intensivo sui nodi x86. Questa dinamica aumenta la contesa sulle risorse rispetto allo scenario precedente, provocando potenziali rallentamenti temporanei anche per le altre funzioni in esecuzione sugli stessi nodi.

In questo contesto, ci si aspetta che le logiche basate sui MAB continuino a offrire prestazioni superiori alla baseline. Per migliorare la leggibilità dei grafici, si è scelto di mantenere come unico termine di paragone la policy RoundRobin, avendo già dimostrato di essere superiore alla policy Random nello scenario ideale. Il focus di questo test sta però nel confronto interno tra i MAB: LinUCB dovrebbe rivelarsi più efficace rispetto a UCB1. Integrando l'occupazione di memoria nel calcolo del reward, LinUCB ha infatti gli strumenti contestuali per percepire la saturazione periodica dell'architettura x86 e deviare temporaneamente il traffico compatibile verso l'anello ARM, aggirando così il collo di bottiglia.

6.4.1 Esperimento 2: Risultati

Come per l'esperimento precedente, si parte dall'analisi delle richieste completate.

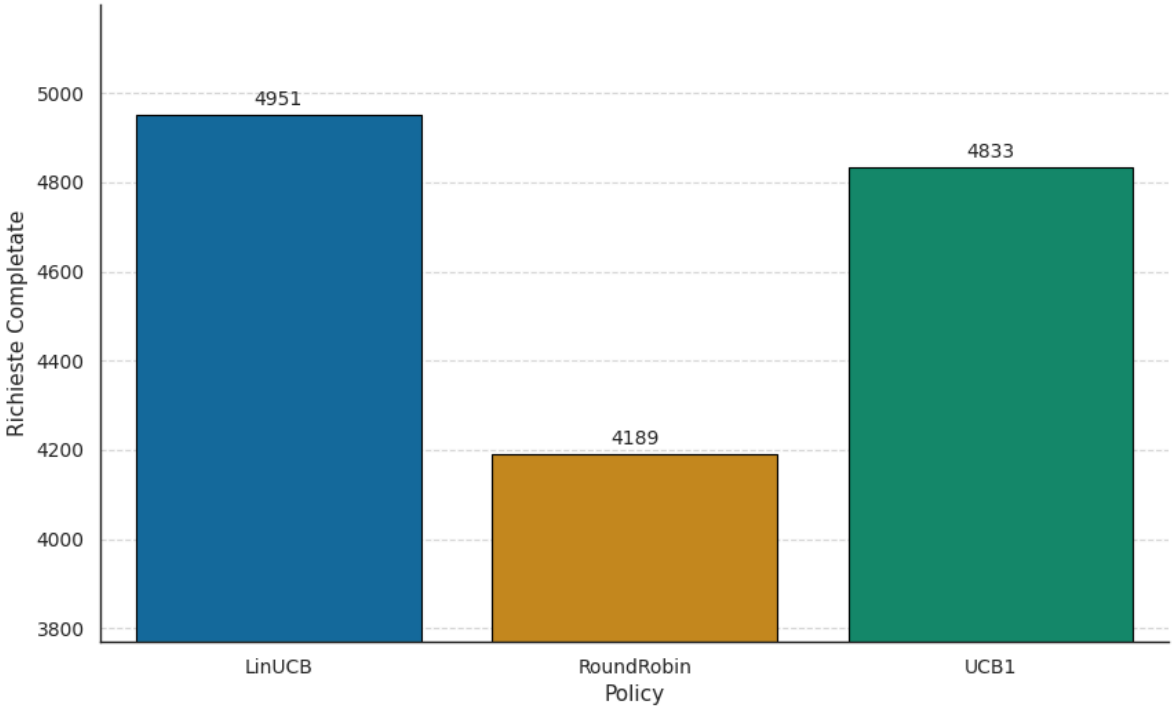

Figura 23: Grafico a barre delle richieste completate per l'esperimento con carico variabile

Osservando la Figura 23 si può vedere come, anche in questo scenario, i MAB offrano prestazioni *nettamente* migliori rispetto alla baseline. Inoltre, a differenza di quanto accaduto in precedenza, LinUCB si comporta meglio di UCB1. Questo risultato è dovuto al fatto che, come emergerà più chiaramente nel prosieguo dell'analisi, LinUCB è riuscito a gestire in modo più efficace il carico variabile sui nodi dell'architettura x86.

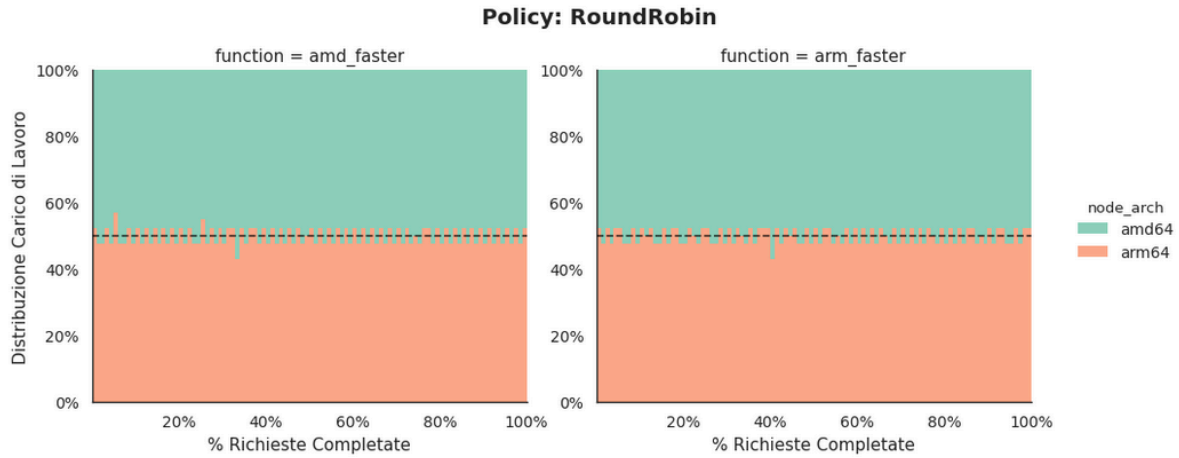

Figura 24: Distribuzione delle richieste sulle due architetture per la policy RoundRobin

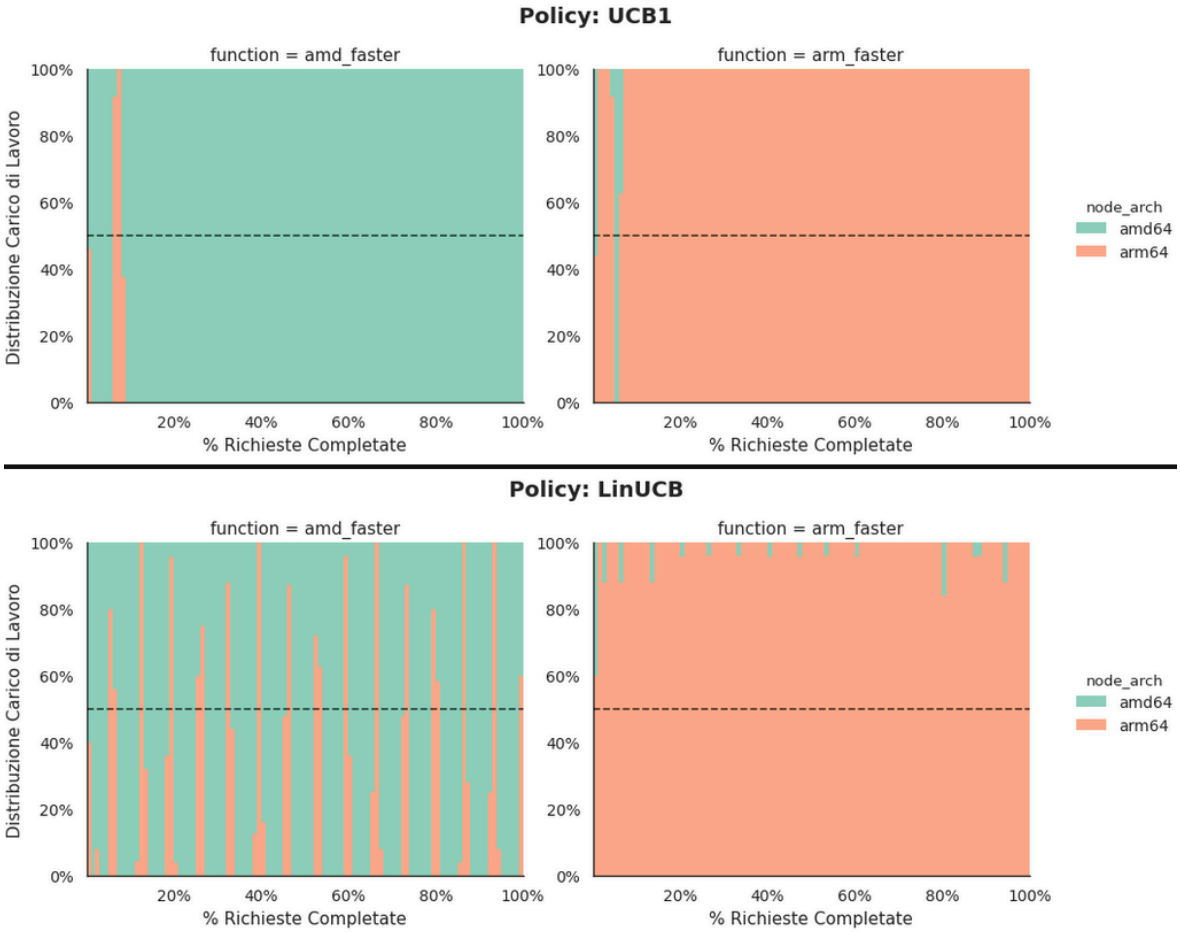

Figura 25: Distribuzione delle richieste sulle due architetture per le policy UCB1 e LinUCB

La Figura 24 presenta, come ci si sarebbe aspettato, uno scenario perfettamente comparabile con quello del primo esperimento. Infatti, RoundRobin continua a distribuire le richieste alternando le architetture scelte. Nella Figura 25, invece, emerge tutta la differenza tra i due MAB. Osservando il comportamento di UCB1, si nota come la policy prenda una decisione quasi definitiva durante le prime fasi di esecuzione: impara rapidamente quale sia l'architettura tendenzialmente migliore per ciascuna funzione e le assegna la totalità del traffico in modo statico. Tuttavia, questa mancanza di una visione globale dei cambiamenti del cluster spiega la maggior dispersione nello stripplot della Figura 28: quando i nodi x86 subiscono improvvisi sovraccarichi, UCB1 continua comunque a inviare richieste a tali nodi, creando maggiore contesa sulle risorse e rallentamenti. Sebbene infatti il reward, inevitabilmente, sarà peggiore di quello atteso, un cambiamento momentaneo non è sufficiente a provocare una nuova fase di exploration, portando il MAB a testare nuovamente l'altro braccio. Inoltre, poiché il peggioramento del reward è solo temporaneo e non correlabile allo stato del cluster, per UCB1 risulta più complesso reagire a questa variazione. Se il sovraccarico dei nodi x86 si protraesse a lungo nel tempo, UCB1 riconsidererebbe a sua volta l'altro braccio. Ma si starebbe parlando di uno scenario diverso da quello attuale.

Al contrario, il grafico di LinUCB mostra un andamento nettamente più dinamico, in particolar modo per la funzione `amd_faster`. Avendo accesso all'occupazione di memoria come variabile di contesto, LinUCB riesce a riconoscere i momenti in cui l'architettura x86 è sotto stress, reagendo tempestivamente e instradando parte del traffico verso l'anello di nodi ARM. È proprio questa continua e fluida redistribuzione del carico, evidente nei continui sbalzi della distribuzione percentuale, che permette a LinUCB di evitare i colli di bottiglia, assorbire i picchi e mantenere una latenza più contenuta.

Per evidenziare in maniera ancora più evidente la capacità di LinUCB di adattarsi dinamicamente al contesto rispetto a UCB1, si è prodotto il grafico della Figura 26.

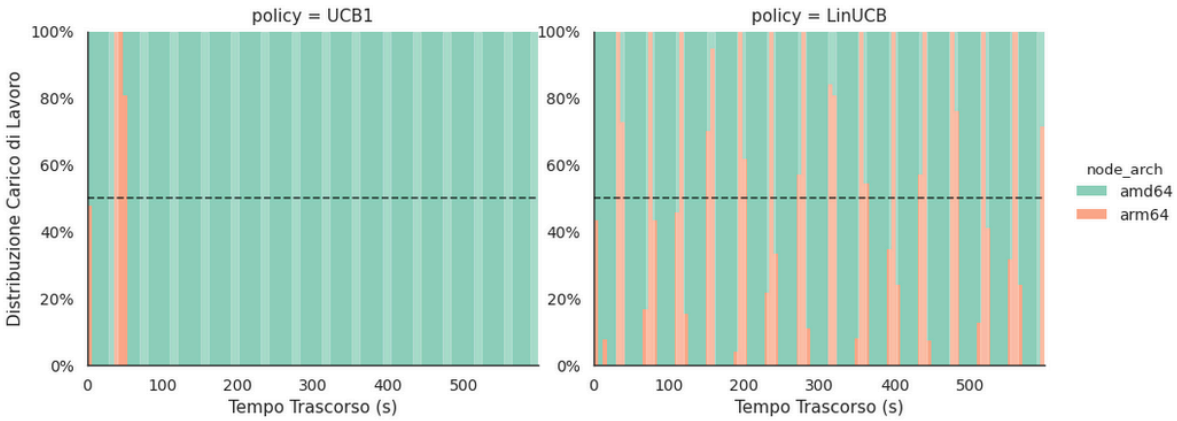

Figura 26: Distribuzione delle richieste per `amd_faster` sulle due architetture da parte di UCB1 e LinUCB. Le bande bianche indicano gli intervalli di tempo in cui il carico sull'architettura x86 era più elevato, per l'esecuzione di `amd_only`

Dal grafico si osserva come i cambiamenti nell'architettura scelta per l'esecuzione della funzione `amd_faster` siano per buona parte allineati alle bande bianche, che indicano gli intervalli temporali in cui l'architettura x86 risulta maggiormente sotto carico.

Si nota inoltre che, nella fase iniziale, anche UCB1 tenta di reagire alla prima variazione di carico, seppur con un certo ritardo. Tuttavia, non riuscendo a correlare le fluttuazioni temporanee dei tempi di esecuzione sull'architettura x86 con lo stato del sistema, l'algoritmo tende rapidamente a privilegiare l'exploitation, continuando a selezionare x86 sulla base dei reward storicamente più elevati ottenuti su quel braccio.

È lecito supporre che, con un parametro `c` più alto per UCB1 e quindi con una exploitation meno aggressiva, anche UCB1 potrebbe adattarsi meglio a un carico variabile. Si è pertanto condotto un esperimento aggiuntivo, del medesimo scenario, utilizzando nuovamente UCB1 e aumentando il valore del suo parametro `c`. Anche utilizzando un valore `c = 1.5`, nettamente più alto del suo default in Serverledge (`c = 0.8`), UCB1 mostra maggiore propensione all'esplorazione, ma senza mai riuscire ad adattarsi realmente al carico variabile, come mostrato dalla Figura 27. L'aumento del parametro `c` non lascia comunque intravedere un miglioramento tangibile. Questo si può osservare

anche nella Figura 30, dove UCB1 con `c = 1.5` completa complessivamente meno richieste rispetto al modello UCB1 con `c = 0.8`. Questo perché rendere il modello meno propenso all'exploitation ne aumenta anche il regret.

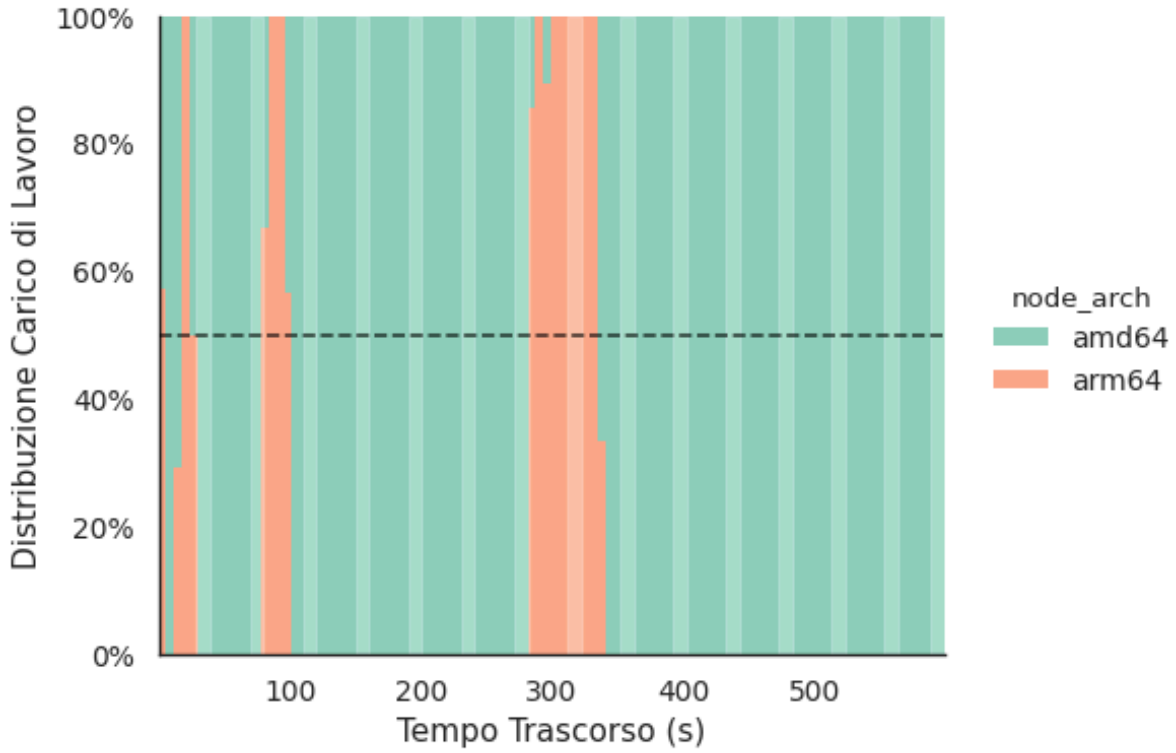

Figura 27: Distribuzione delle richieste per `amd_faster` sulle due architetture da parte di UCB1 con `c=1.5`. Le bande bianche indicano gli intervalli di tempo in cui il carico sull'architettura x86 era più elevato, per l'esecuzione di `amd_only`

Continuare ad alzare ulteriormente il valore di questo parametro, quindi, difficilmente porterebbe a una soluzione. Inoltre, aumentandolo troppo, si rischierebbe di ottenere un comportamento più simile a quello di RoundRobin che a quello di LinUCB, come discusso nella Sezione 5.3.1.

Infatti, il motivo per cui LinUCB riesce ad adattarsi così bene rispetto a UCB1 non è dato da un miglior bilanciamento tra exploration ed exploitation, ma dalla possibilità di utilizzare il contesto, e correlare una più alta occupazione di memoria sull'architettura x86 a esecuzioni rallentate per `amd_faster`.

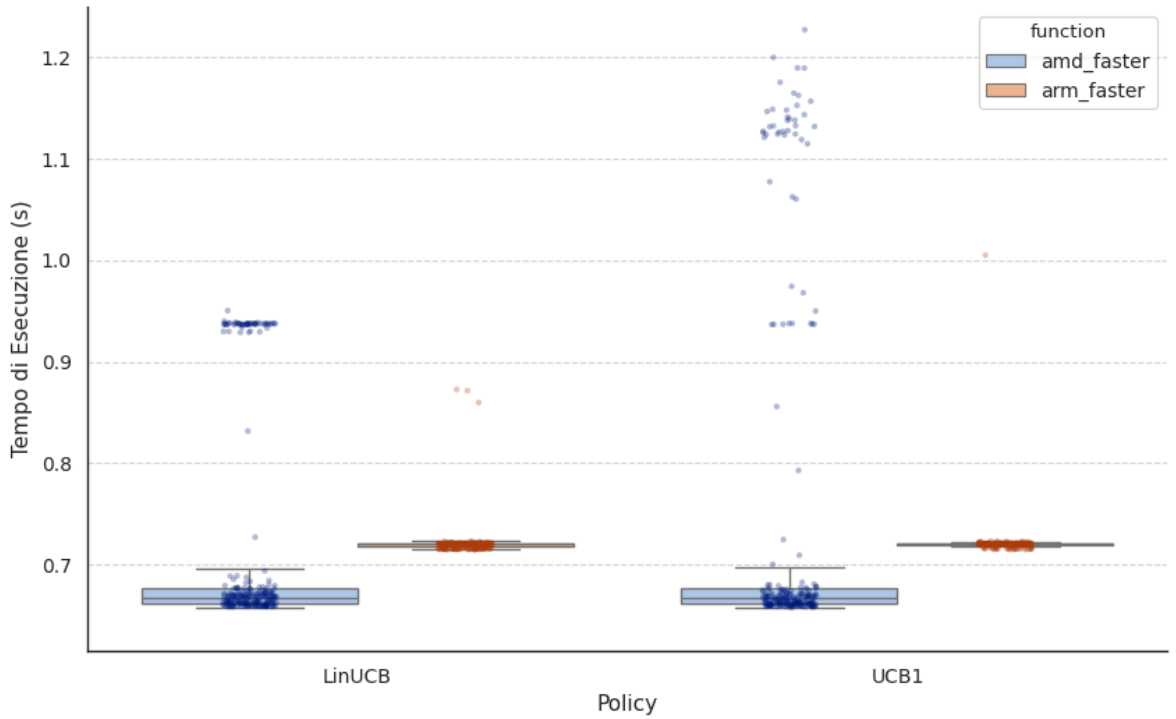

Figura 28: Boxplot e stripplot dei tempi di esecuzione con LinUCB e UCB1 (`c = 0.8`) per l'esperimento con carico variabile

Per migliorare la leggibilità della Figura 28, è stato omesso il boxplot della policy RoundRobin, che avendo un grafico molto più pronunciato avrebbe reso molto meno evidenti le differenze tra i due MAB. Per completezza, il suo grafico è riportato separatamente nella Figura 29. La Figura 28 ci mostra come per UCB1 la dispersione dei tempi di esecuzione di `amd_faster` sia più alta. Ciò è dovuto alle esecuzioni di `amd_faster` effettuate sull'architettura x86 quando questa è più satura. È inoltre interessante notare il fatto che LinUCB presenti una dispersione leggermente più alta per `arm_faster` rispetto a quanto si osserva per UCB1. È plausibile che questo comportamento sia legato al fatto che, durante i picchi di utilizzo, LinUCB sposti l'esecuzione di `amd_faster` verso l'architettura ARM. Questa scelta introduce una leggera contesa sulle risorse dei nodi ARM negli intervalli considerati.

Pur non essendo una condizione ottimale in senso assoluto, rappresenta comunque l'esito migliore in quel contesto: è preferibile accettare una moderata contesa sulle risorse dell'architettura meno congestionata piuttosto che conti-

nuare a instradare le richieste di `amd_faster` verso quella temporaneamente più satura, aggravandone ulteriormente il carico.

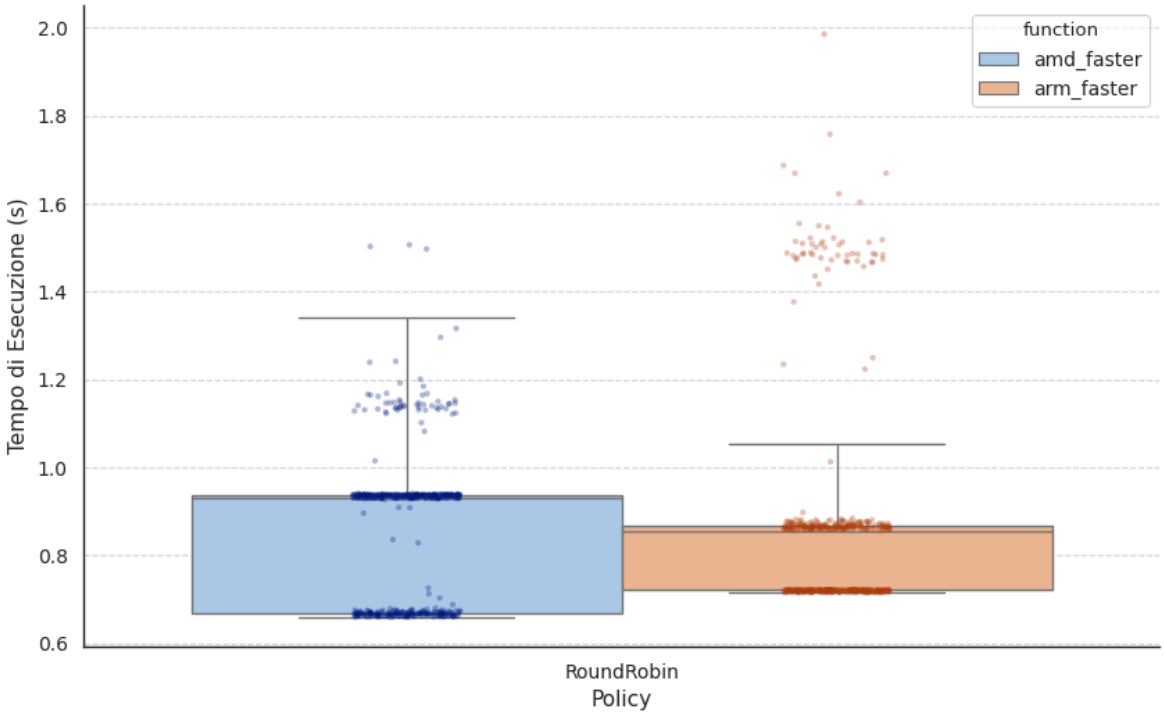

Figura 29: Boxplot e stripplot dei tempi di esecuzione con RoundRobin per l'esperimento con carico variabile

Nella Figura 30, si porta un confronto rispetto al numero di richieste completate durante l'esperimento da parte delle policy analizzate, includendo anche la policy UCB1 con parametro `c = 1.5`.

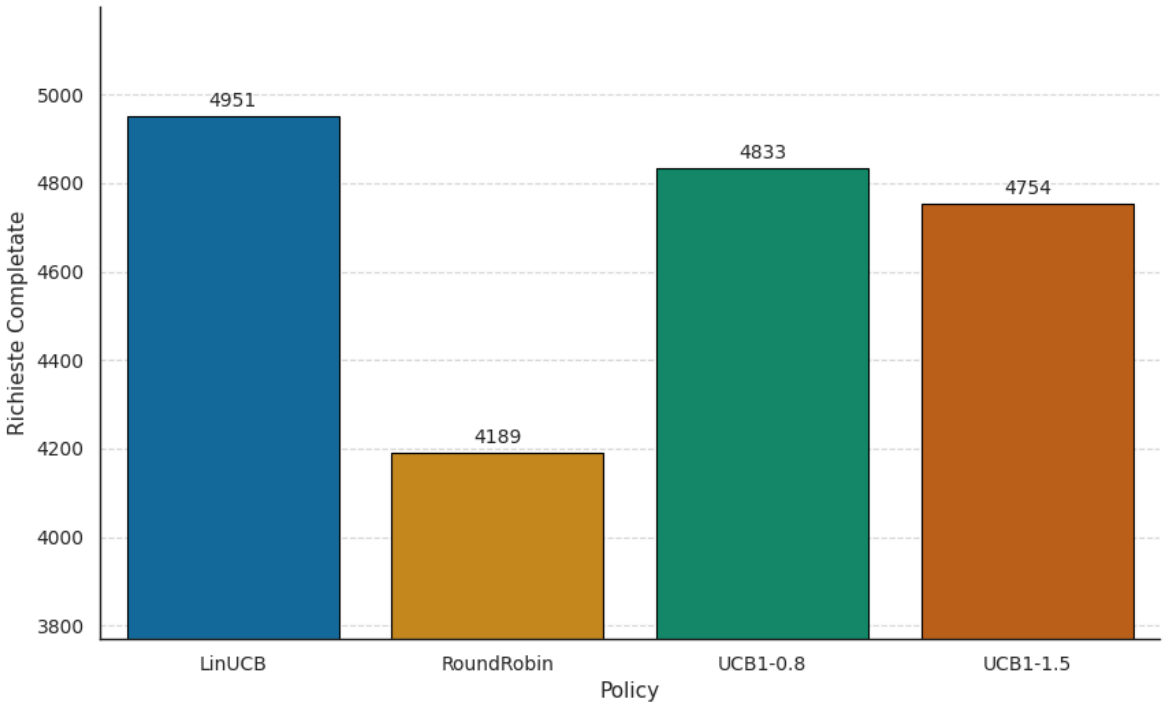

Figura 30: Grafico a barre delle richieste completate nell'esperimento con carico variabile medio, utilizzando UCB1 con due diversi valori del parametro `c`

Come fatto per l'esperimento precedente, si riportano le tabelle relative al throughput e ai tempi di esecuzione medi. Tuttavia, la Tabella 10 presenta dei dati che, letti individualmente, non possono rappresentare un quadro completo della situazione.

Sebbene si possa vedere anche qui come, con LinUCB, i tempi medi di esecuzione della funzione `amd_faster` siano leggermente più bassi, è guardando questi dati insieme a quelli della Figura 28 che si nota come anche la varianza di questi tempi di esecuzione sia più bassa con questa policy.

Policy	Richieste Completate	Throughput (req/s)
LinUCB	4951	8.27
RoundRobin	4189	7.00
UCB1-0.8	4833	8.08

Tabella 9: Throughput e volume di richieste gestite dalle varie policy nello scenario con carico variabile

Policy	amd_faster (s)	arm_faster (s)
LinUCB	0.72	0.72
RoundRobin	0.85	0.85
UCB1-0.8	0.75	0.73

Tabella 10: Tempi medi di esecuzione per ogni tipologia di funzione e policy analizzata nello scenario con carico variabile

Policy	Richieste	vs RoundRobin
LinUCB	4951	+18.19%
UCB1-0.8	4833	+15.37%
RoundRobin	4189	–

Tabella 11: Analisi comparativa sul numero di richieste completate da MAB e baseline nello scenario con carico variabile

La Tabella 11 presenta il dato sulle differenze percentuali rispetto al numero di richieste eseguite. Come già analizzato, i MAB si comportano meglio rispetto alla baseline, e in questo scenario con carico variabile il modello migliore è il MAB LinUCB.

Policy	Richieste	vs RoundRobin
LinUCB	2475	+18.03%
UCB1	2381	+13.54%
RoundRobin	2097	–

Tabella 12: Analisi comparativa sul numero di richieste completate specificamente per la funzione `amd_faster` nello scenario con carico variabile

Infine, nella Tabella 12, si ripete lo stesso confronto considerando solamente i dati relativi alla funzione `amd_faster`, che è anche quella gestita in modo diverso dai due MAB. Qui risulta ancora più evidente come il guadagno di LinUCB rispetto alla baseline sia maggiore di quello di UCB1, per tutte le ragioni discusse finora.

6.5 Esperimento 3: scenario con carico elevato

Questo esperimento è un'estensione dell'Esperimento 2: l'organizzazione delle funzioni e degli utenti Locust è infatti la medesima. Quello che cambia è il carico generato dalla funzione `amd_only`. In questo caso, per generare un carico maggiore, la durata dell'esecuzione è stata portata da 10 a 30 secondi, mentre il `wait_time` tra le richieste è stato abbassato da 30 a 20 secondi. In questo modo si vuole creare, sull'architettura x86, un carico comunque variabile, ma più pesante rispetto all'esperimento precedente. Si vuole analizzare anche in uno scenario di questo tipo come i MAB si comportino rispetto alla baseline e nel confronto diretto tra di essi. Si riportano anche in questo caso tutti i grafici e si commentano le differenze osservate.

6.6 Esperimento 3: Risultati

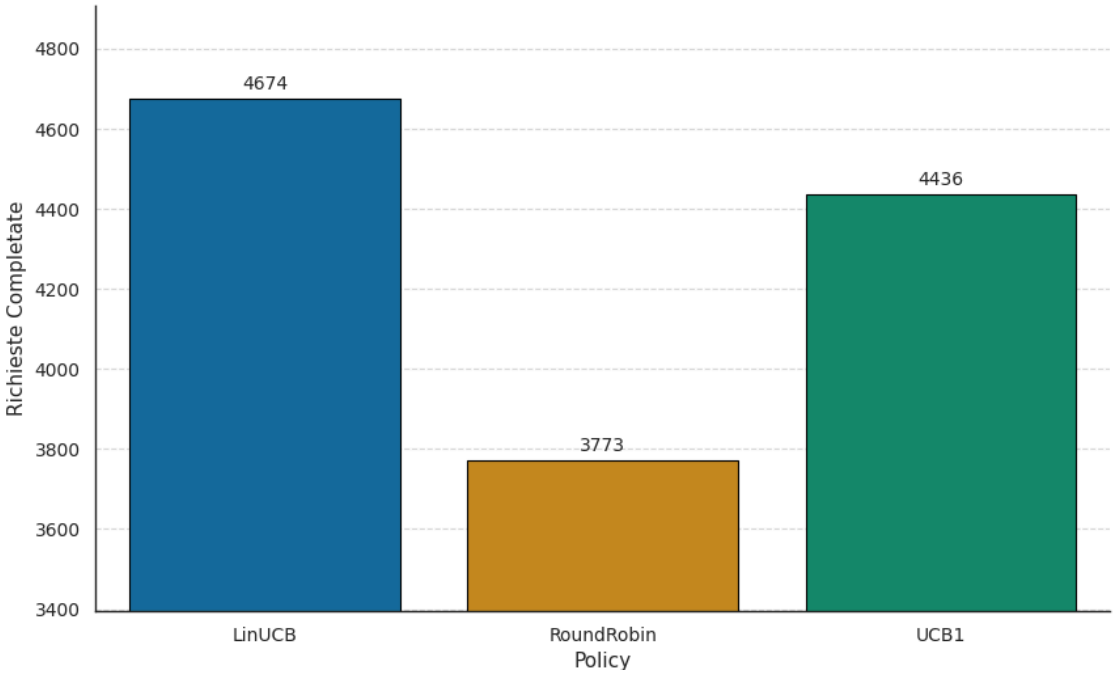

Figura 31: Grafico a barre delle richieste completate per l'esperimento con carico variabile elevato

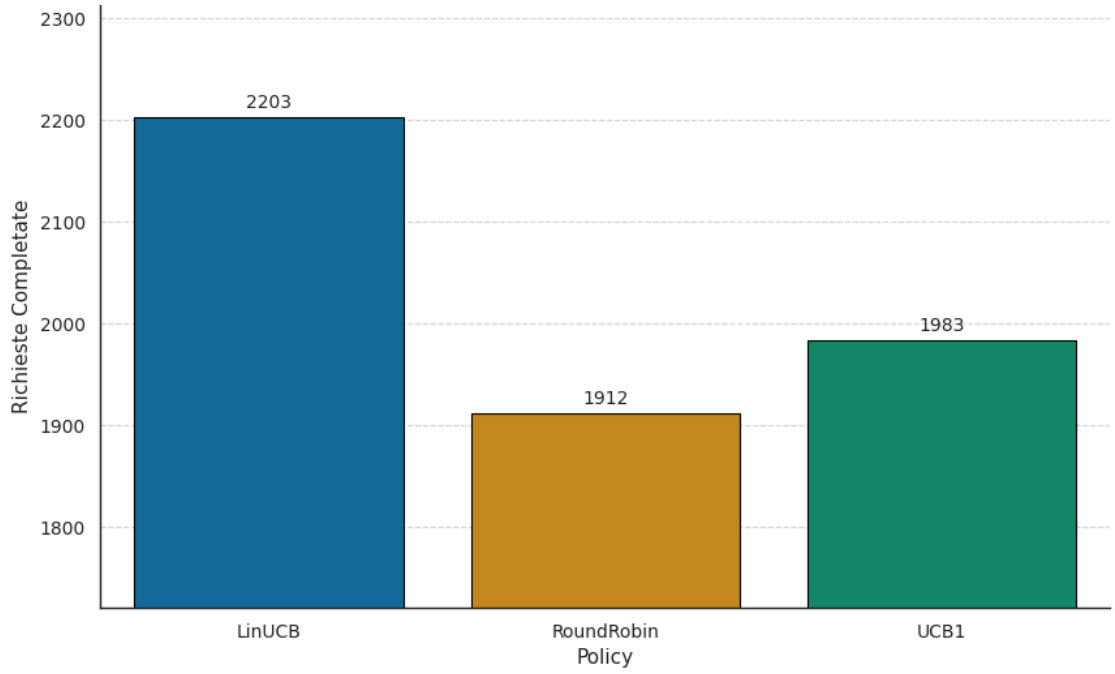

Figura 32: Grafico a barre delle richieste completate per la funzione `amd_faster` con carico variabile elevato

Nel grafico della Figura 31 si può notare come il numero di richieste completate dalle policy sia comunque minore rispetto a quanto osservato per l'esperimento precedente. Questo è dovuto alla presenza di un carico più pesante che porta a rallentamenti maggiori. Nella Figura 32 si riportano i dati relativi al numero di richieste completate per la funzione `amd_faster`. È interessante analizzare questo grafico poiché l'architettura più affine a questa funzione è quella soggetta al carico variabile elevato. Qui si può osservare come LinUCB performi decisamente meglio non solo della baseline RoundRobin, ma anche del MAB UCB1, che mostra prestazioni più vicine a RoundRobin che a LinUCB. Questo rappresenta una prima chiara evidenza di come l'utilizzo di un contextual Multi-Armed Bandit possa fare la differenza in condizioni di carico variabile più pesante.

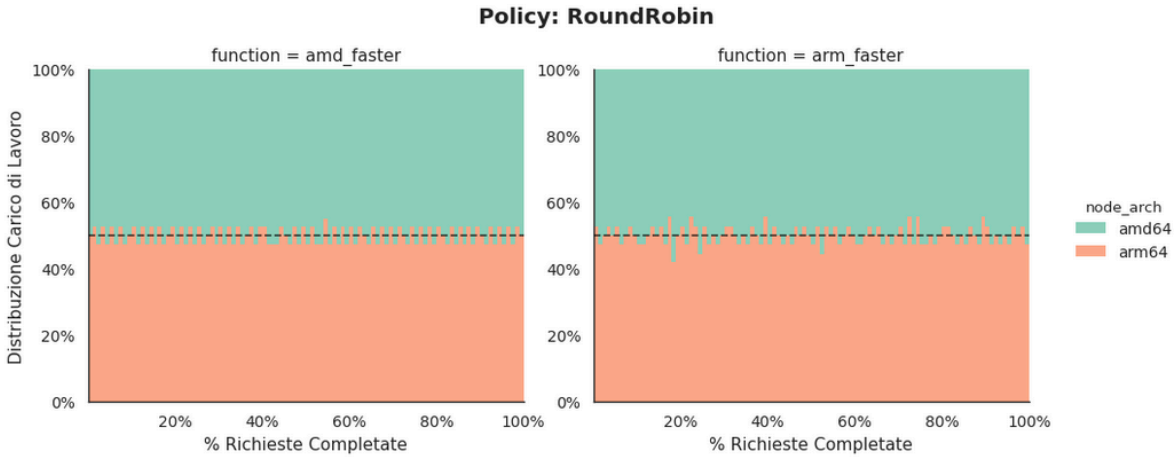

Figura 33: Distribuzione delle richieste sulle due architetture con la policy RoundRobin e carico variabile elevato

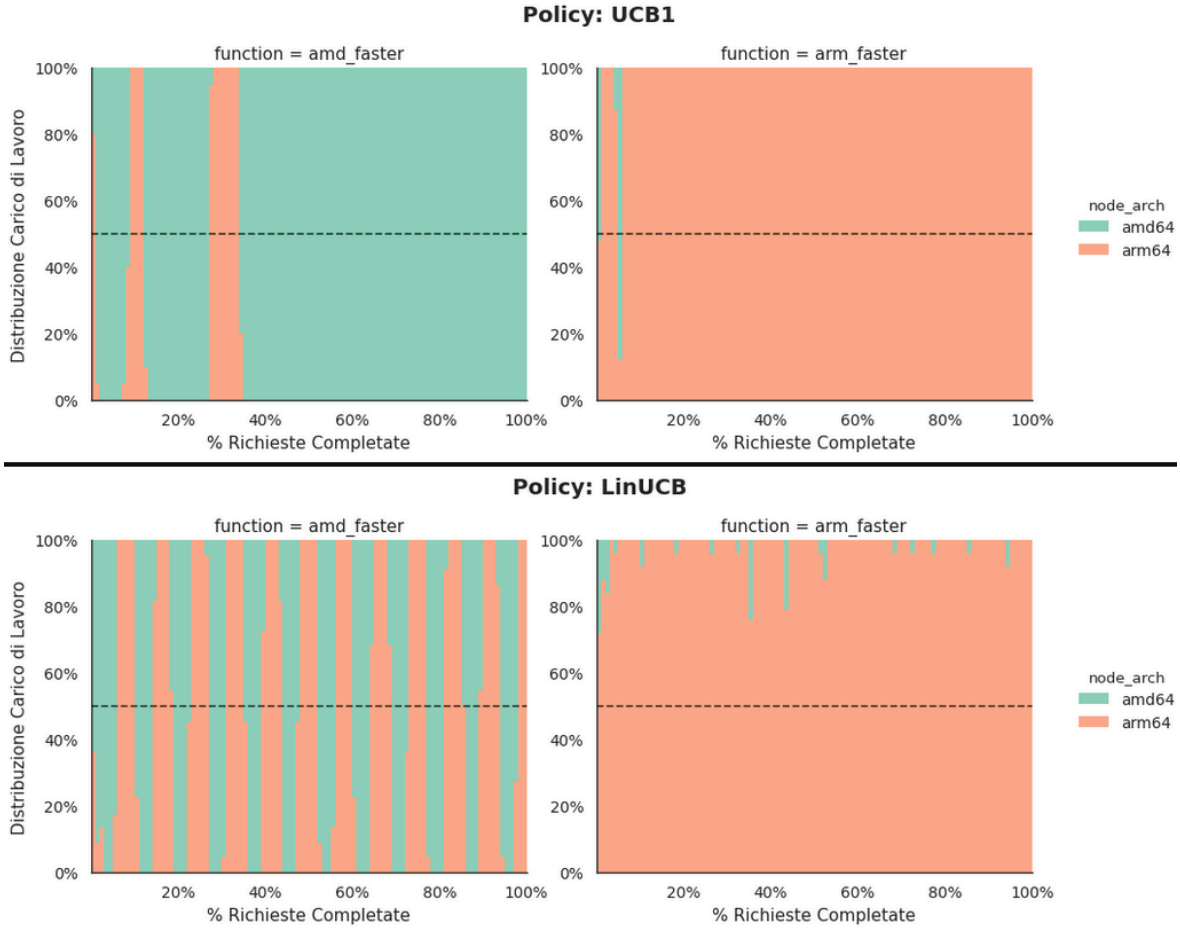

Figura 34: Distribuzione delle richieste sulle due architetture con le policy UCB1 e LinUCB nello scenario di carico variabile elevato

I grafici relativi alla distribuzione delle richieste in questo esperimento (Figura 33 e Figura 34) presentano proprio la situazione che si poteva intuire dai grafici precedenti: LinUCB si adatta al carico variabile, mentre UCB1 non riesce a farlo in maniera efficace. Il grafico di UCB1, rispetto a quello del secondo esperimento (Figura 25), mostra tuttavia una maggiore reattività ai tempi di esecuzione variabili: si può infatti osservare come, nella finestra tra il 10% e il 15% delle richieste completate, l'algoritmo sposti l'esecuzione di `amd_faster` su ARM. Lo stesso accade anche tra il 30% e il 40% circa. Tuttavia non riesce ad adattarsi in maniera efficiente e rapida come fa LinUCB nel lungo termine.

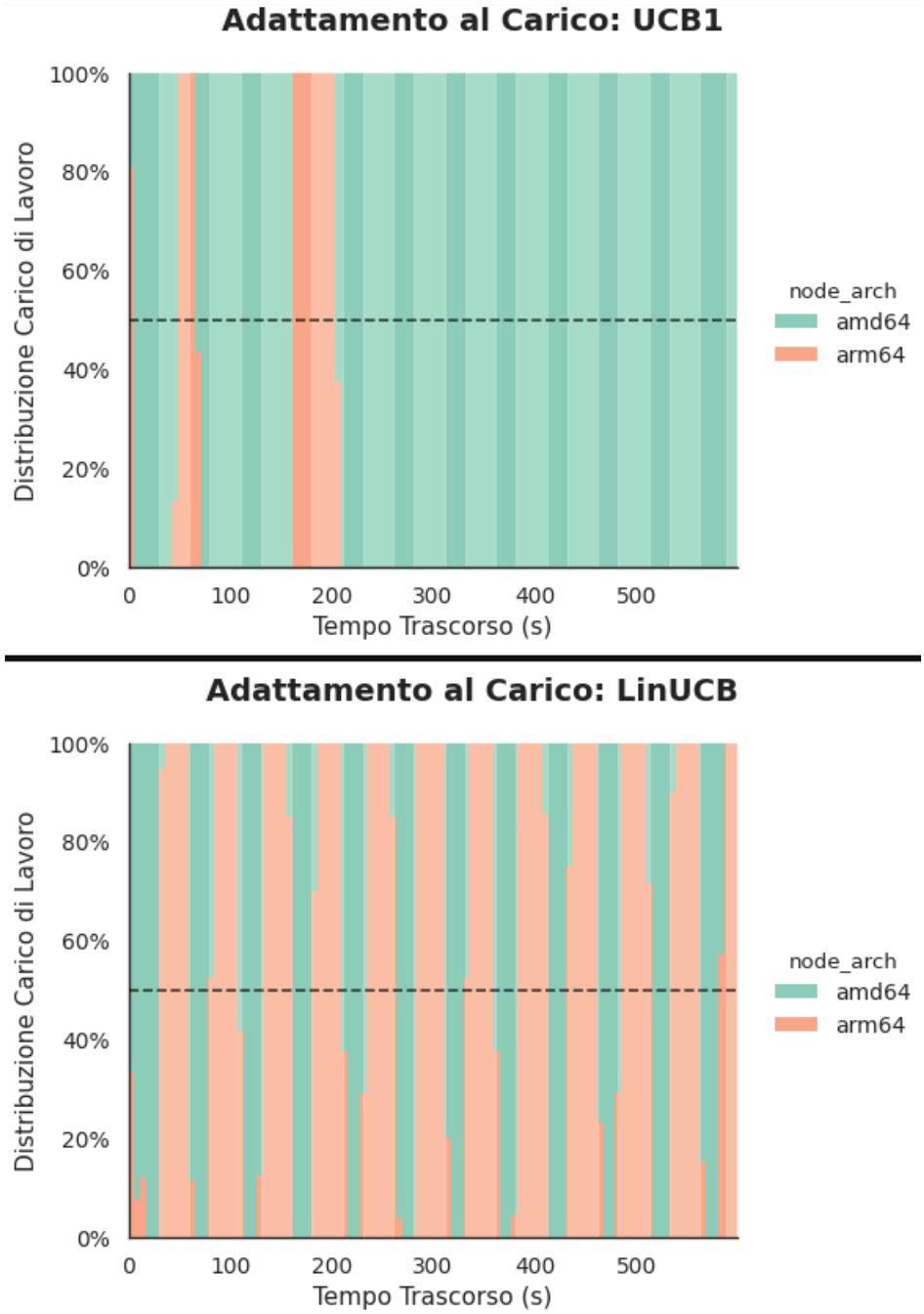

Figura 35: Distribuzione delle richieste per `amd_faster` sulle due architetture da parte di UCB1 e LinUCB con carico variabile elevato. Le bande bianche indicano gli intervalli di tempo in cui il carico sull'architettura x86 era più elevato, per l'esecuzione di `amd_only`

Anche in questo caso, LinUCB riesce ad adattarsi in maniera molto buona alle variazioni di carico (Figura 35), riuscendo a far operare tutte le funzioni

nella situazione migliore compatibilmente con lo stato del sistema. Nella Figura 36 è riportato lo stesso grafico per questo esperimento, ma eseguito anche con la policy UCB1 configurata con il parametro `c = 1.5`. Favorendo maggiormente l'esplorazione, l'algoritmo UCB1 migliora leggermente la propria adattività allo stato del cluster eterogeneo, ma resta comunque lontano dai risultati di LinUCB. Avendo osservato comunque un miglioramento del modello con l'incremento di questo parametro, si è deciso di ripetere nuovamente l'esperimento, aumentando il valore del parametro `c` di UCB1 prima a 2.75, e poi ancora a 4.0. La Figura 37 mostra l'andamento della distribuzione delle richieste nel tempo di queste due varianti del modello UCB1.

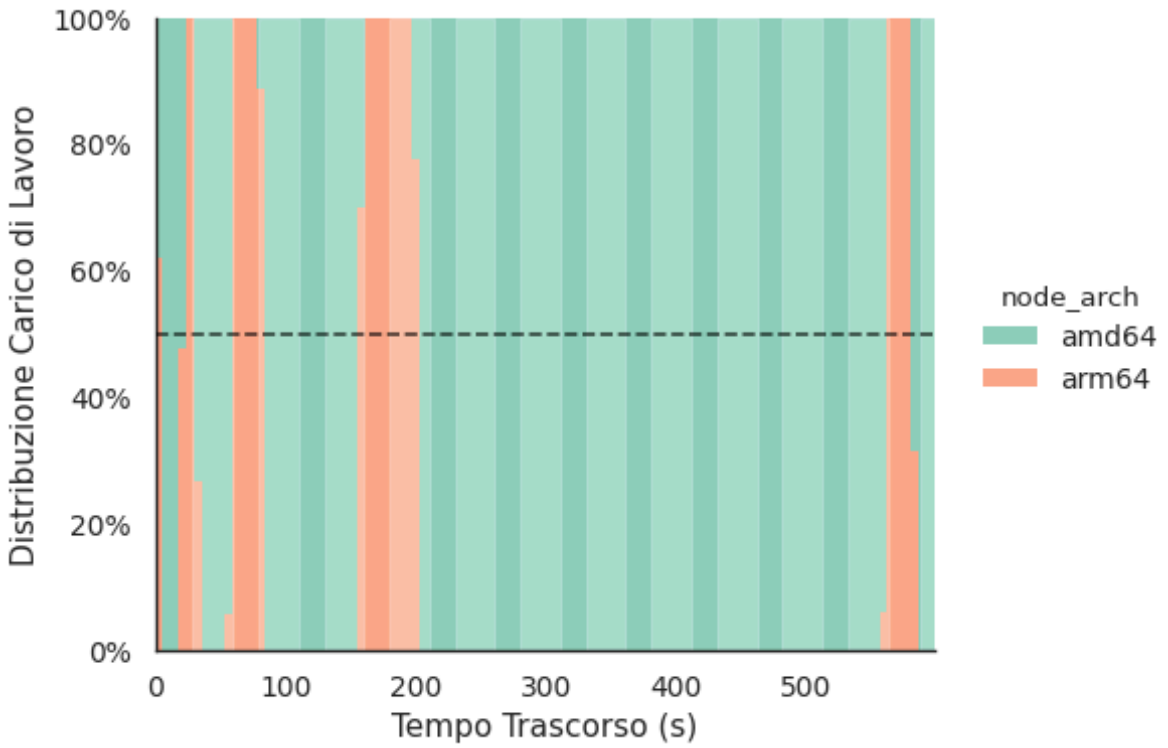

Figura 36: Distribuzione delle richieste per `amd_faster` sulle due architetture da parte di UCB1 con `c=1.5` e con carico variabile elevato. Le bande bianche indicano gli intervalli di tempo in cui il carico sull'architettura x86 era più elevato, per l'esecuzione di `amd_only`

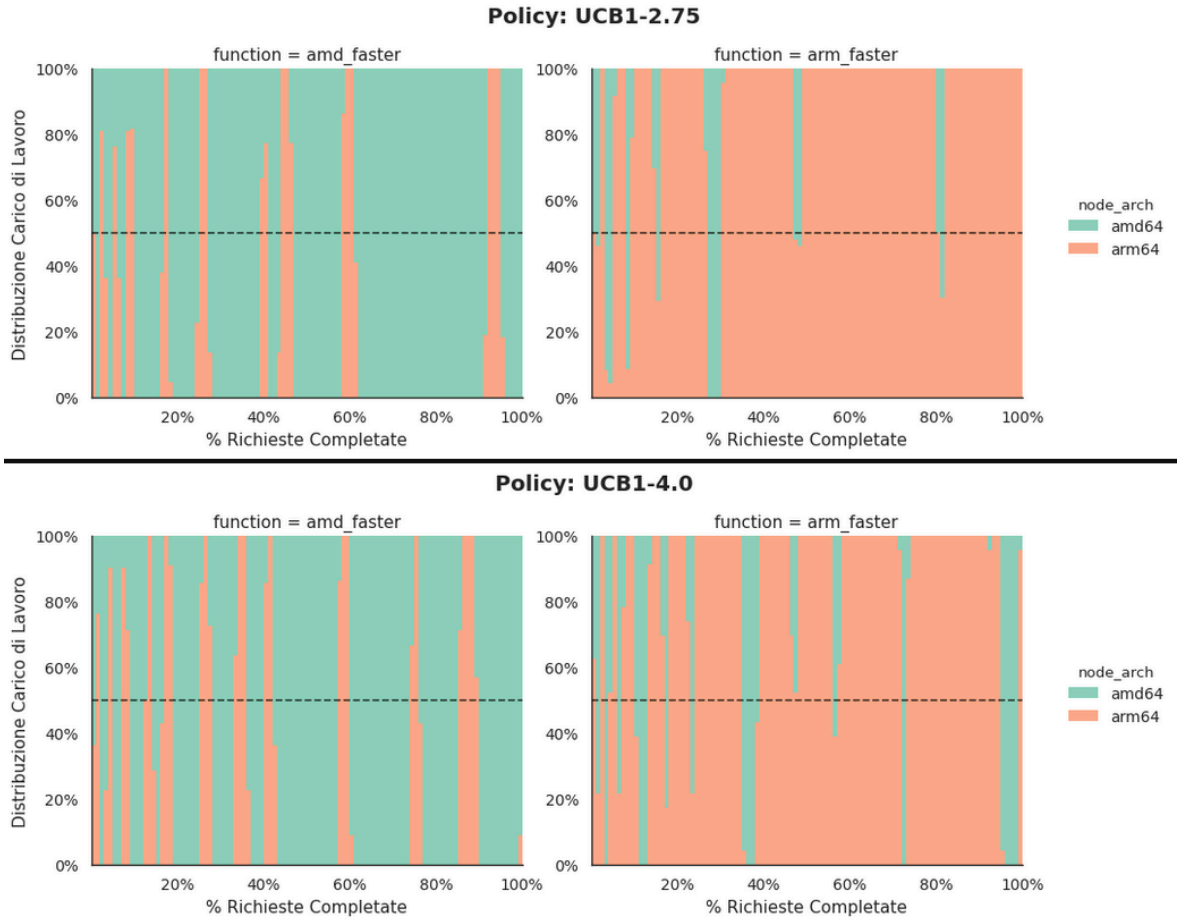

Figura 37: Distribuzione delle richieste per `amd_faster` sulle due architetture da parte di UCB1 con `c=2.75` e `c=4.0` con carico variabile elevato

Qui si può notare come l'aumento del peso del termine di exploration consenta al MAB UCB1 di cambiare l'architettura selezionata con più facilità. Tuttavia, a differenza di quello che succede con LinUCB, questo avviene con entrambe le funzioni `amd_faster` e `arm_faster`. Il costo da pagare per avere un modello UCB1 che si adatti meglio a un traffico variabile è quello di avere un modello che, in generale, fa meno exploitation. Questo significa che un modello di questo tipo (con `c = 2.75`, per esempio) performa meglio di UCB1 con `c = 0.8` in *questo* scenario, ma potrebbe comportarsi peggio nello scenario dell'Esperimento 1. Mentre nell'Esperimento 2 l'aumento di questo parametro in UCB1 non aveva prodotto miglioramenti, in questo terzo scenario si ottengono risultati più evidenti. La motivazione è legata alla durata del

sovraccarico: poiché la funzione `amd_only` viene eseguita per un intervallo temporale più lungo, il ring x86 restituisce tempi di esecuzione degradati per una finestra più prolungata. Di conseguenza, UCB1 accumula una serie più lunga di campionamenti consecutivi con reward subottimali. Questa sequenza prolungata decrementa la stima della media empirica storica Q_a del braccio x86 in modo più marcato rispetto a una congestione di breve durata. È proprio questo calo prolungato del termine di exploitation che permette all'algoritmo, in combinazione con un parametro `c` più elevato, di far prevalere il termine di exploration dell'architettura ARM, forzando la deviazione del traffico. A differenza di quanto succede con LinUCB, in grado di performare in maniera convincente in tutti gli esperimenti, per UCB1 è necessario un estensivo *fine-tuning* del parametro, dimostrando meno flessibilità del modello a causa della sua natura context-free. Il fine-tuning, inoltre, dipende anche dalla specifica situazione: come detto, un aumento di `c` nel secondo esperimento non ha prodotto nessun risultato, mentre in questo terzo scenario consente di ottenere dei leggeri miglioramenti. Alla luce di tutte queste considerazioni, quindi, il fine-tuning rimane un meccanismo poco utilizzabile nel caso generale.

Inoltre, come si può notare dalla Figura 38, in entrambi questi casi il MAB UCB1 non sempre reagisce alle variazioni di carico, e quando lo fa spesso c'è un ritardo. Questo è motivato dal fatto che UCB1 non può reagire al cambiamento dello stato del cluster, ma solamente *alle sue conseguenze*. Quando otterrà dei reward regolarmente più bassi del previsto per il braccio x86 (perché più saturo) per la funzione `amd_faster`, allora reagirà cambiando architettura. Quando il braccio x86 torna a essere meno saturo, UCB1 se ne potrà accorgere solo quando il suo termine di exploration per quella funzione tornerà a essere dominante, e a determinare una nuova esecuzione verso quell'architettura. Quindi, mentre LinUCB, grazie al contesto, può conoscere lo stato del cluster, UCB1 può solo reagire alle conseguenze dei cambiamenti che avvengono, senza mai avere una visione d'insieme.

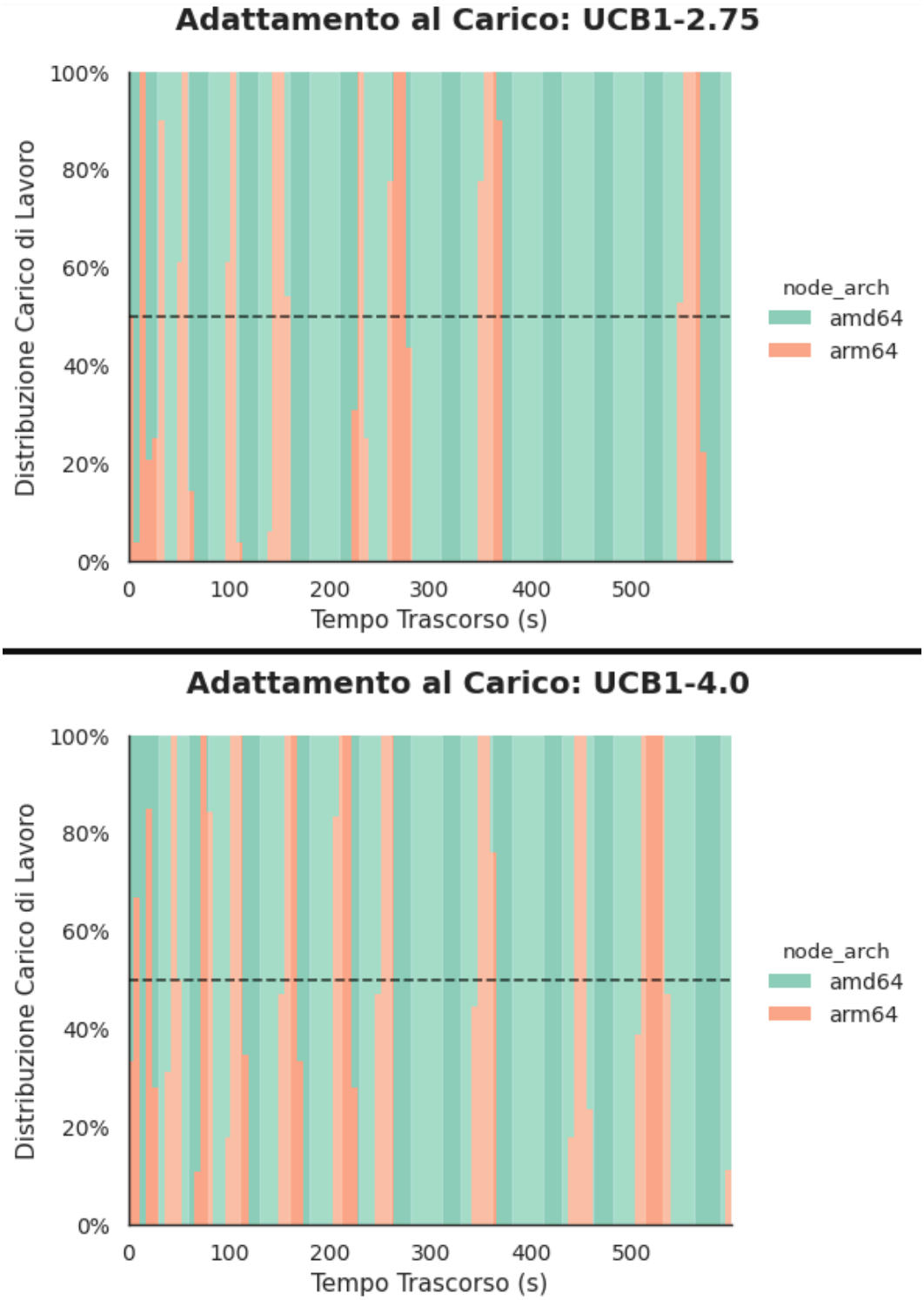

Figura 38: Distribuzione delle richieste per `amd_faster` sulle due architetture da parte di UCB1 (`c = 2.75` e `c = 4.0`) con carico variabile elevato. Le bande bianche indicano gli intervalli di tempo in cui il carico sull'architettura x86 era più elevato, per l'esecuzione di `amd_only`

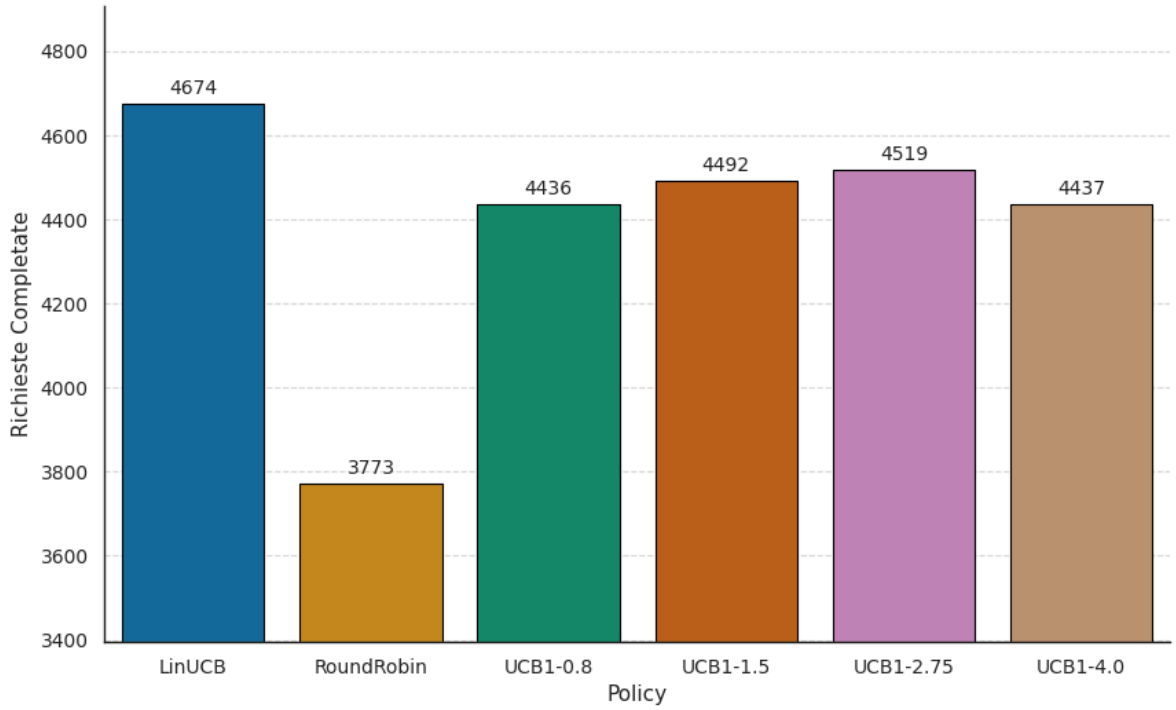

Figura 39: Grafico a barre delle richieste completate per l'esperimento con carico variabile elevato, con le configurazioni aggiuntive testate per UCB1

Il grafico della Figura 39 mostra il numero di richieste smaltite durante questo esperimento da tutte le policy testate. Si può notare chiaramente come, per UCB1, l'iniziale aumento del parametro `c` porti a un beneficio per il numero totale di richieste smaltite, ma un suo aumento troppo spinto porta il MAB a essere troppo "incerto", e non convergere su un'architettura anche nel caso in cui la funzione abbia un'affinità forte con essa. Questo è proprio quello che si è osservato nella Figura 37, dove anche la funzione `arm_faster` non convergeva del tutto all'esecuzione su architettura ARM.

Infine, la Tabella 13 e la tabella Tabella 14 mostrano un confronto del miglioramento in percentuale dei vari modelli rispetto alla baseline RoundRobin, considerando il numero di richieste completate. Nella prima tabella si analizza il totale delle richieste, mentre nella seconda solamente quelle relative alla funzione `amd_faster`, dato che il carico aggiuntivo generato da `amd_only` nell'esperimento è relativo all'architettura con cui questa funzione ha maggiore affinità.

LinUCB si conferma in conclusione il modello migliore, sia in questo esperimento, sia in generale: senza bisogno di nessun fine-tuning riesce a convergere correttamente nel primo esperimento, e riesce a gestire bene sia un carico più leggero, sia un carico, comunque variabile, ma più importante come quello di questo esperimento.

Il MAB UCB1, invece, riesce a performare in maniera ottimale in uno scenario ideale, ma non riesce ad adattarsi in maniera altrettanto efficace in scenari con carico variabile. L'algoritmo risulta comunque migliore della baseline RoundRobin. Tuttavia, alla luce dei risultati sperimentali, in un deployment reale di Serverledge la soluzione più adatta per lo scheduling del load balancer architecture-aware risulta essere LinUCB.

Policy	Richieste	vs RoundRobin
LinUCB	4674	+23.88%
`UCB1 (c=2.75)`	4519	+19.77%
`UCB1 (c=1.5)`	4492	+19.06%
`UCB1 (c=4.0)`	4437	+17.60%
`UCB1 (c=0.8)`	4436	+17.57%
RoundRobin	3773	–

Tabella 13: Analisi comparativa del throughput totale nello scenario con carico variabile elevato

Policy	Richieste	vs RoundRobin
LinUCB	2203	+15.22%
`UCB1 (c=2.75)`	2166	+13.28%
`UCB1 (c=4.0)`	2134	+11.61%
`UCB1 (c=1.5)`	2101	+9.88%
`UCB1 (c=0.8)`	1983	+3.71%
RoundRobin	1912	–

Tabella 14: Analisi comparativa delle richieste completate per la funzione `amd_faster` nello scenario con carico variabile elevato

Capitolo 7

Conclusioni e Sviluppi Futuri

Questo lavoro ha affrontato il problema di come i framework FaaS possano trarre vantaggio dall'eterogeneità dei cluster su cui operano, con particolare attenzione agli scenari Edge. Sebbene l'utilizzo di strategie di scheduling *architecture-aware* stia crescendo in diversi contesti, molti sistemi FaaS continuano a utilizzare algoritmi di instradamento statici. Questo perché, nel modello tradizionale, tali framework operano principalmente nel cloud centrale, dove l'eterogeneità dei nodi è un fattore meno rilevante rispetto all'Edge della rete.

Il punto di partenza concettuale di questo lavoro è stato l'introduzione del paradigma dell'Edge Computing, che non rappresenta semplicemente un'estensione del Cloud, ma un nuovo ecosistema con caratteristiche proprie. Spostare la computazione verso i margini della rete significa andare incontro a una maggiore eterogeneità a livello hardware, dove i tradizionali processori x86-64 si affiancano a dispositivi ARM64, scelti per la loro superiore efficienza energetica e termica. L'obiettivo centrale di questo lavoro di tesi è stato dimostrare che questa eterogeneità non rappresenta solo un problema da risolvere, ma può essere sfruttata attivamente nella fase di distribuzione delle richieste per ottenere un vantaggio prestazionale.

Utilizzando Serverledge come piattaforma FaaS di riferimento, il primo obiettivo è stato estenderlo per renderlo pienamente architecture-aware, integrando la consapevolezza dell'hardware a livello di metadati, protocolli di discovery

e policy di offloading. Questa fase ha richiesto l'adattamento dei runtime tramite immagini Docker multi-architettura. L'estensione ha coinvolto non solo linguaggi interpretati, ma anche linguaggi compilati come Go. Per quest'ultimo è stata necessaria una fase di progettazione dedicata, poiché la compilazione produce binari specifici per ogni architettura e non esiste un layer di astrazione hardware come per l'interprete Python o la Java Virtual Machine. Questa fase ha incluso anche un refactoring di Serverledge, volto a fornire una libreria che gli utenti possano importare per eseguire le proprie funzioni Go all'interno della piattaforma.

Dopo aver implementato i meccanismi necessari per garantire la compatibilità multi-architettura, l'attenzione si è spostata sul load balancer, il componente su cui integrare le logiche di ottimizzazione per la gestione efficiente di cluster eterogenei. I classici algoritmi come il Round-Robin, pur garantendo stabilità, si sono dimostrati subottimali per cluster ibridi. Sebbene riuscissero a gestire senza errori un cluster eterogeneo, non potevano sfruttarne appieno le caratteristiche. La soluzione implementata è partita da una riprogettazione strutturale basata sul consistent hashing, separando in base all'architettura i nodi in anelli di hash distinti. Questo ha permesso di stabilizzare il routing, minimizzare il problema del rehashing globale e favorire l'utilizzo di container warm, rendendo più rare le latenze legate ai cold start. Grazie al consistent hashing, ogni volta che il load balancer seleziona un'architettura per eseguire una funzione, i nodi corrispondenti vengono considerati nello stesso ordine per quella funzione, ma in ordine diverso per funzioni differenti. In questo modo il carico si distribuisce in modo uniforme, massimizzando al contempo i warm start.

Avere un'infrastruttura in grado di instradare correttamente le richieste su architetture diverse rappresentava soltanto la condizione necessaria per affrontare la seconda parte del lavoro di questa tesi. La sfida principale consisteva infatti nello stabilire *dove* eseguire le funzioni per minimizzarne i tempi di esecuzione.

Poiché l'affinità architetturale di una funzione non è prevedibile a priori, si è scelto di escludere approcci statici o modelli predittivi offline, che avrebbero avuto una capacità limitata di adattarsi alla continua evoluzione dei workload serverless, privilegiando invece un approccio basato su Reinforcement Learning.

L'integrazione dei modelli Multi-Armed Bandit all'interno del load balancer ha trasformato il componente in un gestore delle richieste intelligente. I test sperimentali, eseguiti su cluster ibridi con deployment automatizzato tramite Terraform e Ansible, mostrano chiaramente che le politiche di routing basate sull'apprendimento online ottengono risultati migliori rispetto agli approcci tradizionali.

Il confronto tra i diversi algoritmi dei MAB ha però evidenziato un aspetto importante. UCB1, a prescindere dal bilanciamento tra esplorazione e sfruttamento, valuta le prestazioni delle architetture assumendo implicitamente un ambiente stazionario. Non avendo una rappresentazione dello stato corrente del cluster, può solo cercare di convergere verso valori medi assoluti. LinUCB, invece, introducendo un contesto legato all'occupazione di memoria, tiene conto anche delle condizioni in cui si trova il cluster nel momento della decisione. Quando poi ottiene il reward, può correlare i risultati ottenuti da un determinato braccio con il contesto in cui la decisione è stata presa.

In questo modo l'algoritmo può adattare dinamicamente le scelte per l'esecuzione delle funzioni: quando un'architettura risulta momentaneamente più satura, il traffico può essere deviato verso alternative meno cariche, anche se storicamente leggermente meno efficienti per quella funzione. Nei risultati sperimentali questa capacità di adattarsi rapidamente al variare del carico ha portato alle prestazioni complessivamente migliori.

7.1 Sviluppi Futuri

I risultati di questo lavoro forniscono basi concrete per la distribuzione delle richieste in sistemi FaaS multi-architettura e aprono la possibilità di progettare e integrare ulteriori funzionalità e ottimizzazioni in questo contesto.

In primo luogo, l'approccio basato su contextual bandit come LinUCB potrebbe essere ulteriormente potenziato. Attualmente, la feature principale del contesto riguarda lo stato della memoria dei nodi, ma sarebbe interessante ampliare il vettore includendo metriche aggiuntive, come l'utilizzazione media della CPU. Altre metriche che potrebbero essere interessanti da valutare riguardano indicatori di rete come la latenza stimata. Questo permetterebbe al load balancer di prendere decisioni più precise e reattive, tenendo conto non solo delle risorse hardware disponibili, ma anche dello stato della rete. Tuttavia, l'ampliamento del vettore di feature va valutato con cautela, per evitare di introdurre un overhead eccessivo sia nella memoria occupata dal MAB sia nella quantità di dati da scambiare per mantenere tali metriche aggiornate.

Una seconda direzione per lo sviluppo di questo lavoro riguarda l'ampliamento del concetto di "eterogeneità" dei nodi. Ad esempio, oltre a distinguere tra architetture CPU diverse, si potrebbero avere hash ring dedicati a nodi con caratteristiche aggiuntive, come la presenza di acceleratori hardware (GPU o NPU) per workload di intelligenza artificiale. Questo si allinea con l'idea sempre più diffusa di Edge AI, in cui si tenta di eseguire la fase di inferenza il più vicino possibile alla fonte dei dati.

Infine, un'ulteriore evoluzione del framework potrebbe esplorare l'efficienza energetica come metrica da ottimizzare. Mentre in questa tesi si è avuto come obiettivo quello di massimizzare le prestazioni riducendo i tempi di esecuzione, nell'Edge Computing il consumo energetico può essere tanto importante quanto il tempo di esecuzione. Trasformare la funzione obiettivo dei MAB da una pura minimizzazione dei tempi di risposta a un'ottimizzazione multi-obiettivo

(tenendo conto di latenza e consumo energetico stimato) rappresenterebbe un ulteriore passo verso un Cloud-Edge Continuum performante e sostenibile.

Elenco delle Figure

Elenco delle Tabelle

Elenco dei Listati

Elenco degli Algoritmi

Bibliografia

[1] J. Gedeon, F. Brandherm, R. Egert, T. Grube, e M. Mühlhäuser, «What the Fog? Edge Computing Revisited: Promises, Applications and Future Challenges», *IEEE Access*, vol. 7, fasc. , pp. 152847–152878, 2019, doi: 10.1109/ACCESS.2019.2948399.

[2] E. Jonas *et al.*, «Cloud Programming Simplified: A Berkeley View on Serverless Computing», technical report EECS-2019–3, 2019. [Online]. Disponibile su: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

[3] I. Baldini *et al.*, «Serverless Computing: Current Trends and Open Problems», in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, e R. Buyya, A c. di, Singapore: Springer Singapore, 2017, pp. 1–20. doi: 10.1007/978-981-10-5026-8_1.

[4] A. Agache *et al.*, «Firecracker: Lightweight Virtualization for Serverless Applications », in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, feb. 2020, pp. 419–434. [Online]. Disponibile su: https://www.usenix.org/conference/nsdi20/presentation/agache

[5] Google Cloud, «Open-sourcing gVisor, a sandboxed container runtime». [Online]. Disponibile su: https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime

[6] L. Wang, M. Li, Y. Zhang, T. Ristenpart, e M. Swift, «Peeking Behind the Curtains of Serverless Platforms», in *2018 USENIX Annual Technical*

Conference (USENIX ATC 18), Boston, MA: USENIX Association, lug. 2018, pp. 133–146. [Online]. Disponibile su: https://www.usenix.org/conference/atc18/presentation/wang-liang

[7] K. Zhao, Z. Zhou, L. Jiao, S. Cai, F. Xu, e X. Chen, «Taming Serverless Cold Start of Cloud Model Inference With Edge Computing», *IEEE Transactions on Mobile Computing*, vol. 23, fasc. 8, pp. 8111–8128, 2024, doi: 10.1109/TMC.2023.3348165.

[8] E. van Eyk *et al.*, «The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms», *IEEE Internet Computing*, vol. 23, fasc. 6, pp. 7–18, 2019, doi: 10.1109/MIC.2019.2952061.

[9] S. Eismann *et al.*, «The State of Serverless Applications: Collection, Characterization, and Community Consensus», *IEEE Transactions on Software Engineering*, vol. 48, fasc. 10, pp. 4152–4166, 2022, doi: 10.1109/TSE.2021.3113940.

[10] W. Shi, J. Cao, Q. Zhang, Y. Li, e L. Xu, «Edge Computing: Vision and Challenges», *IEEE Internet of Things Journal*, vol. 3, fasc. 5, pp. 637–646, 2016, doi: 10.1109/JIOT.2016.2579198.

[11] M. Villari, M. Fazio, S. Dustdar, O. Rana, e R. Ranjan, «Osmotic Computing: A New Paradigm for Edge/Cloud Integration», *IEEE Cloud Computing*, vol. 3, fasc. 6, pp. 76–83, 2016, doi: 10.1109/MCC.2016.124.

[12] I. Cilic, P. Krivic, I. Zarko, e M. Kusek, «Performance Evaluation of Container Orchestration Tools in Edge Computing Environments», *Sensors*, vol. 23, p. 4008, 2023, doi: 10.3390/s23084008.

[13] X. Chen, L.-H. Hung, R. Cordingly, e W. Lloyd, «X86 vs. ARM64: An Investigation of Factors Influencing Serverless Performance», in *Proceedings of the 9th International Workshop on Serverless Computing*, in WoSC '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 7–12. doi: 10.1145/3631295.3631394.

[14] X. Chen, R. Cordingly, L.-H. Hung, e W. Lloyd, «Predicting ARM64 Serverless Functions Runtime: Leveraging function profiling for generalized performance models», in *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, 2024, pp. 63–72. doi: 10.1109/UCC63386.2024.00018.

[15] Docker, Inc., «What is a Container?». [Online]. Disponibile su: https://www.docker.com/resources/what-container/

[16] G. R. Russo, T. Mannucci, V. Cardellini, e F. L. Presti, «Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum», in *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2023, pp. 131–140. doi: 10.1109/PERCOM56429.2023.10099372.

[17] M. Anderson, «The Shift to ARM-Based Servers». [Online]. Disponibile su: https://medium.com/@mike.anderson007/the-shift-to-arm-based-servers-87aaa749ac19

[18] Amazon Web Services, «AWS Lambda Pricing». [Online]. Disponibile su: https://aws.amazon.com/lambda/pricing/

[19] J. Tharwani e A. Purkayastha, «Cost-Performance Evaluation of General Compute Instances: AWS, Azure, GCP, and OCI», *International Journal of Computer Trends and Technology*, vol. 72, fasc. 11, p. 127, nov. 2024, [Online]. Disponibile su: https://www.ijcttjournal.org/archives/ijctt-v72i11p127

[20] D. Lambion, R. Schmitz, R. Cordingly, N. Heydari, e W. Lloyd, «Characterizing X86 and ARM Serverless Performance Variation: A Natural Language Processing Case Study», in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, in ICPE '22. Bejing, China: Association for Computing Machinery, 2022, pp. 69–75. doi: 10.1145/3491204.3543506.

[21] C. Ebert, «Comparing AWS Lambda Arm64 vs x86_64 Performance Across Multiple Runtimes in Late 2025». [Online]. Disponibile su: https://chrisebert.net/comparing-aws-lambda-arm64-vs-x86_64-performance-across-multiple-runtimes-in-late-2025/

[22] Open Container Initiative, «opencontainers/image-spec: OCI Image Format». [Online]. Disponibile su: https://github.com/opencontainers/image-spec

[23] Docker, «Docker Buildx: Extended Build Capabilities with BuildKit». [Online]. Disponibile su: https://github.com/docker/buildx

[24] S. Arys, R. Carlier, e E. Rivière, «Energy-Aware Scheduling of a Serverless Workload in an ISA-Heterogeneous Cluster», in *Proceedings of the 10th International Workshop on Serverless Computing*, in WoSC10 '24. Hong Kong, Hong Kong: Association for Computing Machinery, 2024, pp. 25–30. doi: 10.1145/3702634.3702954.

[25] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, e H. Chen, «Serverless computing on heterogeneous computers», in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, in ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 797–813. doi: 10.1145/3503222.3507732.

[26] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, M. B. Chhetri, e M. A. Salehi, «Load balancing for heterogeneous serverless edge computing: A performance-driven and empirical approach», *Future Generation Computer Systems*, vol. 154, pp. 266–280, 2024, doi: https://doi.org/10.1016/j.future.2024.01.020.

[27] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, e V. Schiavoni, «Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling», in *2019 27th Euromicro International Conference on Parallel, Distributed*

and Network-Based Processing (PDP), 2019, pp. 400–405. doi: 10.1109/EMPDP.2019.8671554.

[28] S. Jayaram Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, e G. R. Ganger, «Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling», in *Proceedings of the 29th Symposium on Operating Systems Principles*, in SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 642–657. doi: 10.1145/3600006.3613175.

[29] P. Diwan, S. Toraskar, V. Venkitaraman, N. K. Boran, C. Chaudhary, e V. Singh, «MIST: Many-ISA Scheduling Technique for Heterogeneous-ISA Architectures», in *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*, 2024, pp. 348–353. doi: 10.1109/VLSID60093.2024.00064.

[30] R. Cox, F. Dabek, F. Kaashoek, J. Li, e R. Morris, «Practical, distributed network coordinates», *SIGCOMM Comput. Commun. Rev.*, vol. 34, fasc. 1, pp. 113–118, gen. 2004, doi: 10.1145/972374.972394.

[31] Docker Inc., «Core concepts: glibc and musl». Consultato: 24 febbraio 2026. [Online]. Disponibile su: https://docs.docker.com/dhi/core-concepts/glibc-musl/

[32] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, e D. Lewin, «Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web», in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, in STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 654–663. doi: 10.1145/258533.258660.

[33] G. DeCandia *et al.*, «Dynamo: amazon's highly available key-value store», in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, in SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220. doi: 10.1145/1294261.1294281.

[34] Apache Cassandra Documentation, «Multiple Tokens per Physical Node (vnodes)». 2026. [Online]. Disponibile su: https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html#multiple-tokens-per-physical-node-vnodes

[35] M. N. Katehakis e A. F. Veinott, «The Multi-Armed Bandit Problem: Decomposition and Computation», *Mathematics of Operations Research*, vol. 12, fasc. 2, pp. 262–268, 1987, Consultato: 27 febbraio 2026. [Online]. Disponibile su: http://www.jstor.org/stable/3689689

[36] A. Slivkins, «Introduction to Multi-Armed Bandits», *Found. Trends Mach. Learn.*, vol. 12, fasc. 1–2, pp. 1–286, nov. 2019, doi: 10.1561/2200000068.

[37] J. Kawale e E. Chow, «A Multi-Armed Bandit Framework for Recommendations at Netflix». [Online]. Disponibile su: https://aicouncil.com/talks/a-multi-armed-bandit-framework-for-recommendations-at-netflix

[38] G. R. Russo, E. D'Alessandro, V. Cardellini, e F. L. Presti, «Towards a Multi-Armed Bandit Approach for Adaptive Load Balancing in Function-as-a-Service Systems», in *2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, 2024, pp. 103–108. doi: 10.1109/ACSOS-C63493.2024.00039.

[39] I. Čilić, I. P. Žarko, P. Frangoudis, e S. Dustdar, «QoS-Aware Load Balancing in the Computing Continuum via Multi-Player Bandits». [Online]. Disponibile su: https://arxiv.org/abs/2512.18915

[40] L. Li, W. Chu, J. Langford, e R. E. Schapire, «A contextual-bandit approach to personalized news article recommendation», in *Proceedings of the 19th International Conference on World Wide Web*, in WWW '10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 661–670. doi: 10.1145/1772690.1772758.

[41] P. Auer, N. Cesa-Bianchi, e P. Fischer, «Finite-time Analysis of the Multiarmed Bandit Problem», *Mach. Learn.*, vol. 47, fasc. 2–3, pp. 235–256, mag. 2002, doi: 10.1023/A:1013689704352.

[42] J. Murel e E. Kavlakoglu, «Cos'è la regressione ridge?». [Online]. Disponibile su: https://www.ibm.com/it-it/think/topics/ridge-regression

[43] F. Muschera, «serverledge-tesi: A FaaS platform for Edge-Cloud environments». [Online]. Disponibile su: https://github.com/FilippoMuschera/serverledge-tesi