From Cloud to Edge: Seamless Software Migration at the Era of the Web of Things

Michele Beccari 856608

2023

1 Riassunto

Lo standard Web of Things (*WoT*) promosso di recente dal W3C costituisce un approccio promettente per progettare sistemi IoT interoperabili in grado di gestire l'etereogeneità delle piattaforme software e dei dispositivi.
L'architettura *WoT* prevede scenari di IoT caratterizzati da una moltitudine di Web Things (*WTs*) che comunicano secondo delle interfacce software ben definite.
Allo stesso tempo presume un'allocazione statica delle *WTs* agli host e non è in grado di gestire la dinamicità intrinseca degli ambienti IoT per quanto riguarda le variazioni del carico di rete e computazionale.
In queto paper vogliamo estendere il paradigma *WoT* al deployment nel continuo cloud-edge. In questo modo potremmo supportare un'orchestrazione dinamica e la mobilità delle *WTs* su tutte le risorse di calcolo disponibili.
A differenza degli approcci Mobile Edge Computing (*MEC*) allo stato dell'arte vogliamo sfruttare lo standard *WoT* e in particolare la sua capacità di standardizzare le interfacce software delle *WT* per propporre un concetto di Migratable WoT (*M-WoT*).
In un Migratable WoT le *WT* sono allocate senza soluzione di continuità agli host a seconda delle loro interazioni dinamiche.
In questo paper verranno proposte tre contribuzioni:

1. Architettura del framework *M-WoT*:
   Ci concentreremo sulla migrazione delle *WT* con il loro stato e sulla gestione della procedura di handoff della WT.

2. Formulazione rigorosa dell'allocazione delle *WTs* come problema di ottimizzazione multi obbiettivo.
   Proporremo anche un'euristica basata su grafi.

3. Descriviamo un'implementazione basata su container di M-Wot e una two fold validation.
   Utilizzeremo la two fold validation per verificare la performance della policy di migrazione proposta in una situazione con un setup di edge computing distribuito e in uno scenario di monitoraggio IoT del mondo reale.

2 Introduzione

L'impressionante crescita dell'Internet of Things (*IoT*) in termini di dispositivi connessi e di dati prodotti può essere spiegata dalla versatilità del suo paradigma, che può essere applicato in una grande varietà di casi d'uso, dal manifatturiero digitalizzato alle città smart e al monitoraggio dell'ambiente.

In questi domini la mobilità del servizio ha iniziato a rendersi molto interessante per vari scopi.

Da una parte molte applicazioni IoT operano in ambienti dinamici: di conseguenza alle soluzioni software è richiesto di adattarsi a cambiamenti rapidi

1. nell'utilizzo di banda
2. nell'utilizzo di risorse di calcolo
3. nel numero di dispositivi connessi
4. nei requisiti del servizio

Diverse piattaforme per l'IoT forniscono uno strato di adattamento supportando la mobilità senza soluzione di continuità lungo i nodi di un continuo cloud-edge.

Dall'altra parte i dispositivi IoT mobili che generano stream di dati mutevoli nello spazio e nel tempo spingono la ricerca verso delle architetture di calcolo flessibili in grado di auto configurarsi per soddisfare la qualità del servizio (QoS) per le applicazioni utente.

Questo è il caso dell'architettura mobile edge computing (MEC) (e concetti simili come il Cloudlet, il Fog Computing e il follow me cloud) che cerca di eseguire i processi in prossimità delle sorgenti dati. Una caratteristica fondamentale delle architetture MEC è la possibilità di delegare i servizi software ai server edge/fog il più vicino possibile alla posizione corrente dell'utente, utilizzando spesso:

- Tecniche di mobilità di container e/o virtual machine
- Politiche di migrazione guidate dalla mobilità fisica dei dispositivi IoT

La migrazione di servizi non è l'unica sfida aperta nel panorama IoT, che comprende un gran numero di procolli, stack tecnologici ed ecosistemi cloud. La maggior parte degli ambienti IoT sono caratterizzati dall'eterogeneità dei componenti hardware e software e dalla dinamicità delle loro interazioni.
I problemi di interperabilità secondo alcune stime possono ridurre i potenziali introiti fino al 40%. Allo stesso tempo, nuove opportunità di business possono nascere offrendo soluzioni che consentono a sistemi IoT diversi di comunicare fra loro. Anche se gli ecosistemi cloud possono mitigare alcuni dei problemi di interoperabilità con l'utilizzo di tecnologie web (es. api REST, JSON , web sockets ecc...) si basano spesso su architetture "a silo" con un vendor lock-in esplicito o implicito.
Inoltre le soluzioni basate su ecosistemi cloud utilizzando un approccio *sensor-to-cloud* dove i dispositivi sono gestiti utilizzando una connettività basata sul cloud, limitando nuovamente l'estensibilità della soluzione.

Lo standard Web of things del consorzio W3C rappresenza una soluzione recente e promettente per sbloccare il potenziale dell'IoT consentendo l'interoperabilità fra varie piattaforme Iot.
Il supporto per l'interoperabilità è gestito a livello applicativo definendo un'interfaccia standard per i componenti IoT (fisici o virtuali) nota come *thing description* che definisce formalmente le capacità e le potenzialità della web thing.
Nonostante la sua recente comparsa, alcune applicazioni interessanti dello standard WoT sono state proposte per diversi domini IoT. Allo stesso tempo però l'implementazione di riferimento del WoT non supporta la mobilità dei componenti software fra i nodi del cloud o dell'edge.
Questo perchè il runtime di una WT va deployato staticamente su un device.
In questo paper affronteremo le due problematiche dell'IoT menzionate in precedenza, ovvero:

- Migrazione dei servizi
- Interoperabilità dei servizi

da una prospettiva WoT: in particolare vogliamo estendere le funzionalità del WoT per gli ambienti IoT supportando l'orchestrazione dinamica e la mobilità delle WT's fra tutte le risorse computazionali disponibili di tutto lo spettro IoT (nodi dell'edge/fog/cloud)
La migrazione delle WT offre nuove opportunità rispetto ad approcci di mobilità software nella letteratura MEC.
Infatti visto che le interazione fra le WTs sono descritte attraverso interfacce software uniformi (ad esempio le TD, le things description) è possibile progettare delle policy di migrazione adattive e precise;
queste policy possono migrare gruppi di WT per soddisfarre i requisiti QoS di sistema considerando le condizioni della rete del mondo reale e del calcolo compuutaizonale, con molta meno complessità computazionale per il mnoitoraggio del servizio che altre soluzioni ad-hoc.
Allo stesso tempo però la mobilità di una WT da un nodo all'altro potrebbe avere un'impatto sulle operazioni di altri WT che la stavano usando.
Quindi le nuove soluzioni devono essere deployate per gestire l'handoff della WT e per garantire la consistenza di tutto il sistema.
Questo paper affronta domande di ricerca legate sia ai *meccanismi di migrazione* delle WT che alle *politiche di migrazione*, es.

- Come rendere possibile la migrazione senza soluzione di continuità di una WT fra due nodi?
- Come ottimizzare le performance di un deployment WoT orchestrando le allocazioni delle WT nel continuo cloud edge?

Per affrontare queste questioni proponiamo il Migratable Web of Things (M-Wot) un nuovo framework architetturale che supporta l'allocazione dinamica delle WT definite dal W3C su i nodi diclcolo disopnibili.
Specificatament, investighiamo come abilitare la migrazione delle WT preservandone lo steto, gestendo la procedura di handoff sui consumatore della WT.
Allo stesso tempo contempliamo la presenza di un servizio orchestratore per il WoT, che è in grado di monitorare le interazioni fra le Wt e di calcolare l'allocazione ottimale delle WT fra i nodi, basandosi su policy ad alto livello (es. si vuole massimizzare la località dei dati, minimizzare la latenza ecc...).
In maniera più dettagliata, questo studio porta sostanzialmente tre contribuzioni:

- Discutiamo dei vantaggi dei meccanismi di migrazione delle WT su due casi d'uso IoT, e poi i componenti dell'architettura M-Wot.
- Formuliamo il problema dell'allocazione delle WT come problema di ottimo multi-obbiettivo. Successivamente proporremo un'euristica centralizzata che mira a bilanciare la comunicazione fra gli host (generata dalle interazioni fra le WTs) e il carico computazionale di ogni host.
- Validiamo le operazioni del M-Wot attraverso due ambienti di test.

Prima valuteremo la performance di diverse policy di allocazione in scenari di edge-computing dove variamo il numero delle WT e il numero di interazioni fra di esse. Successivamente investigheremo l'efficacia del framework M-Wot in uno scenario di monitoraggio IoT dove i servizi di diagnostica in tempo reale sono migrati dinamicamente dal cloud ai nodi di edge in base alle condizioni attuali.

L'analisi della valutazione dimostrerà che l'euristica proposta può bilanciare la comunicazione fra gli host e il carico computazionale in maniera efficace se confrontato con policy di tipo greedy.
Inoltre, nel caso d'uso del monitoraggio Iot la soluzione M-Wot è in grado di ridurre in maniera efficace la latenza di diagnostica rispetto ad un'approccio senza migrazioni allo stato dell'arte.
Il resto del paper è strutturato come segue: la sezione 2 riassume gli approcci di migrazione dei servizi Iot e l'architettura Wot del W3C.

La sezione 3 evidenzia le novità del framework M-Wot e la sua adeguatezza nei casi d'uso IoT selezionati.
La sezione quattro descrive l'architettura M-Wot e i componenti che ne consentono il funzionamento.
La sezione 5 discute del deploymente del WoT come problema di ottimo multi ovviettivo e propone un'euristica basata su un grafo per allocare le WT ai nodi.
La sezione 6 abbozza l'effettiva implementazione del M-Wot.
I risultati sperimentali sono presentati nella sezione 7.
La sezione 8 trae le conclusioni e discute i futuri sviluppi

3 Lavori correlati

Per quanto ne sappiamo, il problema dell'allocazione dinamica e della migrazione live delle WT può essere considerato relativamente nuovo nella letteratura dei sistemi WoT.
Allo stesso tempo ci sono molti paper scientifici che affrontano la migrazioni di servizi software fra i nodi del cloud e i nodi edge per supportare la mobilità fisica dei dispositivi IoT.
Per questo motivo divideremo i lavori correlati in due sezioni.
Prima nella sezione 2-A rivedremo velocemente l'architettura WoT, motivati dalla novità dello standard e dalla necessità di notrudurre la terminoligia utilizzata nel paper; discuteremo anche i (pochi) strumenti e applicazioni sviluppati fin'ora.
Successivamente nella sezione 2-B rivedremo le architetture e le tecnologie che consentono la migrazione dei servizi IoT, concentrandoci principalmente su approcci Mobile Edge Computing (MEC).

3.1 W3C web of things

Il gruppo WoT del W3C ha iniziato le sue attività nel 2015 con l'obbiettivo di definire un insieme di standard di riferimento che consentisse l'interoperabilità fra diversi sistemi IoT.
Il cuore della proposta è la definizione della *Web thing* (WT), che rappresenta l'astrazione di un'entità fisica o virtuale. I metadati e l'interfaccia possono essere descritti formarlmente da un WoT thing description (TD)
. L'architettura di una WT comprende quattro blocchi di nostro interesse:

- Interaction affordances
- Security Configuration
- Protocol bindings
- Behaviour

come mostrato nella figura 1.
I primi tre blocchi sono inclusi nella TD: l'ultimo può essere descritto come una sequenza di di metadati standardizzati comprensibili dalla macchina che consentono ai consumatori di scoprire ed interpretare le capacità di una WT per poterci interagire.
Più nel dettaglio:

- L' *Interaction affordances* (semplicemente affordances da qui in avanti) fornisce un modello astratto di come i consumatori posso interagire ocn la WT,in termini di proprietà (es. le variabili di stato della WT), azioni (es. i comandi che possono essere invocati sulla WT) ed eventi (es. gli eventi inviati dall WT).
- I *protocol bindings* definiscono la mappatura fra le affordances astratte e le strategie di rete (es. i protocolli) hce possono essere utilizzati per interagire con la WT
- La *Security configuration* definisce i meccanismi di controllo dell'accesso alle afforances

La TD può essere codificata con i mezzi del linguaggio JSON-LD, includendo quindi una descrizione semantica.
Infine, il *behaviuour* è l'implementazione della WT,include le affordances (il codice delle azioni).
Tutti i blocchi di cui sopra sono eseguiti all'interno di un runtime software detto Servient che può operare sia come server che come client.
Nel primo il Servient operi come server il si dice che il servient hosta ed *espone* le cose, ovvero crea un oggetto a run-time che serve le richieste verso la WT che sta hostand (come accedere alle proprietà esposte, alle azioni e agli eventi).
Nel caso in cui il servient operi come client si dice che il servient *consuma* le cose. In questo caso il servient processa la TD, genera una rappresentazione a runtime chiamata la "consumed thing" e la rende disponibile ai client che stanno interagendo con la WT remota.
Data la recente apperizione dello standard, la letteratyra sul WoT è ancora scarsa e limitata a poche applicazioni e strumenti di supporto.
La mappatura di dispositivi IoT del mondo reale a Web Things 23C è stata esplorata in vari paper per i cellulari, l'industria automotive e i le reti di sensori wireless.
Specificatamente nell'ultimo caso gli autori hanno dimostrato come deployare delle applicazioni interoperabili basate si WoT in grado di gestire sensori eterogenei con tre diverse tecnologie di accesso wireless (Wi-Fi, BLE e Zigbee).
Riguardo gli strumenti, oltre a quelli che implementano lo standard W3C WoT in diversi linguaggi di programmazione (es. javascript e python) citiamo la piattaforma WoT store che supporta la gestione senza soluzione di continuità di WT e applicazioni miste in grado di consumare più WT eterogenee allo stesso tempo.

3.2 Migrazione servizi IOT

Una moltitudine di soluzioni sono state proposte per consentire la migrazione di servizi senza soluzione di continuità gra i nodi di un sistema distribuito.
Possiamo classificare gli approcci esistenti in due grandi categorie:

- migrazione statica
- migrazione dinamica

Nel primo caso la migrazione del software è usata come sinonimo di modernizzazione del software, ovvero il processo di adattare le capacità di un sistema esistente per poterlo deployare in un nuovo ambiente; il lettore può fare riferimento ad un certo paper per un sondaggio esaustivo sulla migrazione di sistemi legacy verso software cloud-based.
Nel secondo caso, ovvero quando si parla di migrazione dinamica, si fa riferimento al processo di delegare l'esecuzione a run-time di servizi software da un nodo all'altro.
Ci contreremo sulla migrazione dinamica perchè è più rilevante per lo scopo di questo paper.
Nel dominio IoT possiamo introdurre un ulteriore distizione fra approcci di migrazione indotti dall'utente e approcci di migrazione introdotti dalla mobilità.
Il primo caso include diversi studi su come consentire alle applicazioni mobili di migrare senza soluzione di continuità fra i nodi durante le normali operazioni.
L'obbiettivo finale è quello di offire la migliore quality of experience agli utenti mentre passano da un device all'altro.
Per questo scopo in un paper descrivono il middleware TRAMP per la mobilità precisa delle applicazioni multimediali; la decisione di migrazione è definita manualmente dagli utenti.
Negli approcci indotti dalla mobilità la migrazione software è ottenuta assicurandosi che la gestione e il processing dei dati sia sempre il più vicino possibile alla posizione corrente del dispositivo.
Un modello concettuale di questo tipo è denotato come Mobile Edge Cloud, anche se presenta divere sovrapposizioni con altre architetture allo stato dell'arte, come quella cloudlet, follow me cloud e fog computing.
Il fog computing ha diverse definizioni: in questo paper facciamo riferimento alla proposta in un paper, che definisce il fog clopud come "uno strato di risorse che è fra il dispositivi sull'edge e i data center

nel coud, con caratteristiche che possono somigiliare ad emntrambi.
Un'illustrazione dettagliata delle tecniche di migrazione dei servizi può essere trovata nel paper 5; qui, le sfide uniche del MEC comparate alla migrazione live per i datacenter e alle procedure di handover nelle reti cellulari vengono evidenziate
In maniera simile nel paper 28 gli autori propongono il concetto di Companion Foc Computing (CFC), un architettura software composta di strati distribuiti, uno in esecuzione sul dispositivo mobile e una sul fog server; quest'ultimo è allocato dinamicamente sui nodi di un infrastruttura fog per minimizzare la distanza dalla posizione corrente del dispositivo.
In maniera analoga, lo studio in 29 propone una architettura di rete basata su cloudlet includendo un'algoritmo cooperativo per la mobilità del carico di lavoro fra i nodi del cluster.
In generale le piattaforme correlate al MEC devono risolvere due problemi principali:

- Come definire la strategia di migrazione servizi, considerando lo stato corrente di utilizzo dei nodi oltre che la QoS dell'applicazione IOt
- Come implementare la mobilità software, gestendo anche la migrazione dello stato di esecuzione.

Rispetto al primo problema (la policy di migrazione) la maggior parte delle policy di migrazione che tengono in conto della qualità del servizio si concentrano sul ritardo come il principale indicatore di performance (30) e utilizzano dei modelli con processi decisionali di Markov (MDP) per descrivere l'evoluzione del sistema nel tempo (ovvero la mobilità del dispositivo e le conseguenti azioni di mobilità del servizio).
Visto che i pattern di mobilità possono essere difficili da raccogliere in anticipo, un numero crescente di studi sta investigando le applicazioni di tecnici di machine learning per il calcolo della policy di migrazione ottimal; un'esempio può essere trovato in [3], dove l'utilizzo di tecniche di deep reinforcement learning (DRL) è dimostrato che masssimizzi la reward dell'utente, definita come la differenza fra la QoS e il costo di migrazione.
Fra gli studi che non si concentrano sul ritardo, citiamo la piattaforma auto-organizzante per la gestione del servizio per le smart city proposta in [31] dove la metrica ETX (expected transmission count) è utilizzata per determinare l'allocazione ottimale dei servizi IoT nei fog nodes.
Per quanto riguarda il secondo problema (ovvero la mobilità del software) le macchine virtuali (VM) e i container sono le tecniche più investigate per implementare la migrazione dei servizi con o senza stato.
La migrazione delle VM secondo la mobilità dei dispositivi prevista è considerata in [9]; inoltre per ridurre l'overhead di rete introdotto dal trasferimento della VM viene applicata una tecnica di sintetizzazione di container consentendo ad un nodo fog di riprendere l'esecuzione di una VM applicando dei delta ad un'immagine base.
Le possibilità di effettuare migrazione orizzontali (roaming) o verticali (offloading) di funzioni IoT basate su container dovker è dimostrata in [10].
Da un punto di vista delle performance l'implementazione basata sui container è spesso considerata più adatata per la virtualizzazione ai bordi della rete rispetto all'approccio VM-Bases [32].
Questo è confermato da diversi studi sperimentali, includo lo studio [33] che investiga l'implementazione di meccanismi di virtualizzazione per la gestione dei dati IoT e dimostra che l'impatto energetico su computer single-board è trascurabile.
Un alternativa all'utilizzo delle macchine virtuali e dei container è costituita dalla migrazione del codice attivo, per questo scopo il framework ThingMigrate consente la migrazione di processi Javascript attivi fra diverse macchine utilizzando dei meccanismi di iniezione per tracciare lo stato locale di ogni funzione.
Rispetto agli studi citati fino ad'ora la migrazione delle WT affrontata in questo paper può essere considerata un'istanza speciale di una migrazione dinamica basata sugli agenti [35]; allo stesso tempo presenta nuove opportunità oltre che nuove sfide tecnichce che sono discusse in dettaglio nella sezione che segue.

3.3 M-WOT definizioni preliminari e motivazioni

Consideriamo uno scenario distribuito composto da un insieme di nodi di calcolo distribuiti lungo tutto lo stack dello spettro Iot (dall'edge al cloud) come mostrato nella figura 2; ogni nodo è abilitato al WoT, ovvero puà hostare uno o più servient (l'ambiente di runtime dell'architettura WoT) e ogni servient contiene una singla WT in stato di esecuzione.
Definiamo la migrazione WT come la capacità di delegare dinamicamente un WT fra diversi nodi, stoppando l'esecuzione al nodo sorgente e riprendendola al nodo di destinazione.
Presumiamo che il processo di migrazione sia stateful, ovvero lo stato interno di una WT e la su TD doverebbero essere spostati e aggiornati insieme al codice.
In particolare tutti i valori delle Properties e le informazioni che descrivono il contesto computazione della WT doverebbero essere considerati parte dello stato quindi migrati.
Rispeto agli approci di migrazione classici (VM, container, agent based) visti in precedenza la migrazione della WT presenta dei vantaggi e delle nuove sfide di ricerca da affrontare:

- Challenge: thing handoff management. Il WoT consente le interazione senza soluzione di continuità fra software eterogenei durante le operazioni di consumo della WT; se una WT migra su un altro nodo, tutte le altre WTs che la stavano consumando devono essere notificare per poter aggiornare i loro oggetti consumati e puntare al nuovo indirizzo della TD.
  Questo caso d'uso è rappresentato nella figura 2, dove entrambe le WT A e B stanno consumando la WT C; La WT C verrà migrata dall'host 1 all'host 2 in un'istante futuro.
  Per questo motivo una procedura di signalling adeguata deve essere effetuata per informare A e B di quando l'attivazione della WT C sull'host 2 è stata completata, in modo tale che possano consumare nuovamente la TD della WT C.
  Inoltre il processo di migrazione introduce un intervallo di handoff durante il quale la WT C potrebbe non essere in grado di processare le invocazioni remote dalla WT A e B, la durata di questo handoff è chiaramente un parametro critico che determina la performance del sistema.

- Advantage: support to group migrations.
  Continuando rispetto al punto precedente, un framework per la migrazione WoT potrebbe supportare la mobilità di gruppi di componenti software (rispetto ad un singolo servizio come accade negli approcci MEC) come conseguenza delle dipendenze attive di dati (le interazioni) fra le WT, a fianco della mobilità fisica dei dispositivi IoT.
  Infatti ogni WT espone le proprie affordances attraverso la TD in maniera standardizzata; come risultato è possibile costruire un grafico delle dipendenze real-time fra tutte le WT di un sistema WoT distribuito e conseguentemente progettare policy di allocazione che determinano migrazioni di gruppo di sottoinsiemi di WTs che interagiscono fra di loro per massimizzare la località dei dati.
  Chiaramente le policy migrazioni basate su gruppi potrebbero anche essere deployate su altre architettura a micro servizi; tutta via, per il caso del M-WoT questa feature potrebbe essere supportata in un modo agnostico rispetto al proctcollo visto che le interazioni fra le WTs occorono secondo un'interfaccia standard e quindi potrebbero essere gestite attraverso lo strato di monitoring del M-WOT descritto nella sezione IV-C.

Le figure 3(a) e 3(b) mostrano due possibili casi d'uso della migrazione WoT, collegate a due modelli concettuali di WTs differenti: la migrazione del servizio di processamento dei dati e la migrazioni dei digital twin.
In maniera più specifica, la figura 3a rappresenta un'applicazione di structurual health monitoring basata su tecnologie IoT/WoT per come è stata proprosta insiene ad altri dal progetto MAC4PRO.
Assumiamo che il sistema di monitoraggio possa lavorare in due modalità diverse: normale e critica, dentando due requisiti QoS per il rilevamento dei rischi.
All'estremità dell'edge ci sono i sensori (es. degli accelerometri) che monitorano le vibrazioni deòlla costruzione nel coros del tempo.
I dati dei sensori sono resi disponibili attraverso la SWT (sensor web thing) che fornisce funzionalità di querying dei dati e querying e aggiornamento dello stato del dispositivo.

L'analisi dei dati dei sensori è gesista dalle WTs migrabili T_1 , T_2, T_3 e T_4 che implementano rispettivamente le funzionalità di data fusion, data cleaning, data alerting, data forecasting.
Nella modalità normale T_1 e T_2 sono in esecuzione su nodi fog in prossimità della struttura monitorata, mentre T_3 e T_4 sono hostati in remoto sul glod; questo introduce della latenza di rete nel rilevare anomalie o situazioni di pericolo (calcolate da T_3) ma allo stesso tempo minimizzano il carico sui nodi fog.
Ad un certo punto dell'esecuzione del sistema presumiamo che anomali dei dati consecutive sono notate all'interno dei dati (T_2) per qui i sistema di monitoraggio passa dalla modalità *normale* alla modalità *critica*; questa azioni potrebbe anche richiedere un grado di responsiveness più alto per il sistema di diagnostica.
Nell'ambiente M-WOT il cambio di modalità può essere gestito automaticamente migrando il servizio T_3 dal cloud ai nodi fog (o viceversa quando la modalità torna ad essere *normale*) segna nessun bisogno di una configurazione manuale e senza introddure alcun meccanismo di signaling esplicito fra le WTs coinvolte (ovvero T_2 e T_3).
La figura 3(b) rappresenta il secondo caso d'uso dove la migrazione coinvolge i digital twins del WoT. I digital twin sono definiti nello standard W3C come una rappresentazione virtuale di un dispositivo o di un gruppo di dispositivi che risiedono sul cloud o su un nodo dell'edge (...) possono modellare un singolo dispositivo o possono aggregare più dispositivi in una rappresentazione virtuale dei dispositivi combinati.
A questo proposito, consideriamo un applicazione WoT per l'industria automotive come quella proposta in [17]; una WT è associata ad ogni componente all'interno dei veicoli per consentire accesso senza soluzione di continuità e interazioni con i segnali dell'auto.
Similmente a [17] le sensor web things sono incaricate di recuperare i dati dall'hardware del veicolo.
Inoltre presumiamo la presenza si una Veichle web thing, definita come il digital twin del veicolo per intero; la VWT è l'unico punto di accesso ad un sottoinsieme di proprietà azione ed eventi delle SWT, ma espone anche nuove affordances derivate dall'analisi e dall'elaborazione dei dati di più sensori es. per la diagnostica in tempo reale del veicolo.
A causa dei requisiti enerfitivi la VWT è hostata al di fuori del veicolo, su un fog node di proprietà del comune
. Mentre il veicolo su muove nei limiti dello scenario, la sua VWT veiene generata automaticamante sul nodo fog più vicino, in maniera simile alle applicazioni MEC, anche se qui è la mobilità fisica del dispositivo ad indurre la mobilità del digital twin della WT.
In aggiunta immaginiamo uno scenario in un acittà con molti VMTs diversi fra loro, associati a diversi tipi di veicoli (es. auto, bici, autobus eccc...); le VMT sono successivamente consumate dalle City web things nel cloud per fornire servizi avanzati legati alla mobilità, come smart parking, monitoraggio del traffico, routing multi modale eccc...
Evidenziamo il fatto che il numero di VMT può essere molto dinamico nel tempo, ovvero nuove things potrebbero essere create o eliminate come effetto della mobiilità terrestre; in maniera simile il carico compuitazionale per l'esecuzione delle VMT e della CWT potrebbe variare nel tempo.
Nel nostro ambiente M-WoT le VMT sono allocate dinamicamente fra i nodi del cloud e del fog quando compaiono nel sistema; inoltre, policy di load balancing multi obbiettivo possono essere utilizzate, ad esseempio per minimizzare la distanza dall'orgine dei dati (es. il veicolo) mentre si massimizza l'utilizzo delle risorse di calcolo dei nodi fog e cloud.

4 Architettura M-WOT

L'architettura sotware del M-WOT è rappresentata nella figura 4. Presumiamo un'insieme di servients del WoT deployati su diversi nodi, ogni servient hosta esattamente una WT.
In maniera differente rispetto ad un deploymente WoT legacy, che si presume essere statico, il M-Wot consente la mobilità delle WT fra nodi diversi.
A questo scopo il M-WoT offre due nuovi componenti, l'orchestrator e il thing directory; questi moduli non migrano e possono essere deployati o sull'edge (se i requisiti computazionali sono soddisfatti) o

sui server nel cloud.
Inoltre, uno strato di monitoraggio è stato aggiunto allo stack dei servient.
In questo paragrado descriveremo in dettaglio la struttura interna dei tre moduli, mentre nella sezione IV-D chiarificheremo le operazioni dei modili quando avviene il processo di migrazione di una WT.

4.0.1 Thing directory

Il thing directory serve come registro delle risorse del M-WoT ovvero delle thing descriptors attive.
Più nel dettaglio, assumiamo due tipi di TD, una associata alle WT e una ai servient: queste ultime descrivono le capacità dell'ambiente di runtime e sono utilizzate per abilitare le funzionalità dello strato di monitoraggio descritto nella sezione IV-C.
Una volta attivato ogni servient registra la propria TD e la TD della WT che hosta.
Il thing directory gioca due ruoli principali.
Funziona come servizio di discovery, quando viene interrogato dai client ritorna la lista dei TD che rispettano i parametri di interrogazione, come risultato l'orchestrator può essere al corrente della lista dei servients disponibili al momento nello scenario WoT.
Come secondo ruolo supporta notifiche push verso le WTs o verso i servient, quando specifici eventi di sistema sono rilevati, ad esempio quando una WT comleta la procedura di handoff.
A questo scopo presumiamo che la WT T_1 sia stata consumata dalla T_2 che accede periodicamente ad una delle sue proprietà.
Se T_1 è migrata su un nodo diverso la pipeline dei dati si rompe a meno che T_2 sia notificata dell'evento di mobilità e della nuova posizione del servizio.
Il processo di notifica è illustrato nel diagramma di sequenza della figura 7, discussa successivamente nella sezione IV-D.
Alternativamente si potrebbe utilizzare un meccanismo di polling (coinvoltendo T_1 e TDir nel nostro esempio).
Tuttavia questo approcio potrebbe introdurre dell'overhead di rete significativo con conseguente spreco di banda, per cui non è stato considerato nella nostra soluzione.

4.0.2 WT orchestrator

L'orchestrator costituisce il componente core dell'architettura M-WoT.
Come spiegato in precedenza sfrutta la TDir per recuperare la lista dei servient attivi (ovvero delle loro TD).
Successivamente interroga periodicamente ogni servient attraverso al sua interfaccia WoT per raccogliere statistiche live, come l'utilizzo della CPU e il traffico di rete generato dalle interazioni WT.
Basandosi sui valori delle metriche ricevute e sulle politiche di ottimizzazioni in uso, l'orchestrator determina l'allocazione adatta delle Wts e/o dei servient sui nodi.
Il piano di allocazione è poi trasferito ad un layer sottostante (esterno al M-Wot) chiamato in genere qui il *Migration substrate* che è incaricato di implementare la mobilità fisica del software fra i nodi sorgenti e i nodi di destinazione.
I passi appena citati sono eseguiti continuamente dall'orchestrator durante la vita del sistema; come risultato la dinamicità dell'ambiente WoT/IoT riguardo alla creazione /distruzione delle WT, variazioni nell'utilizzo della banda e aggiornamenti a rutime delle policy sono supportati a pieno.
Inoltre, per favorire l'estensibilità della piattaforma la struttura dell'orchestrator è stata divisa in tre sottomoduli principali che riflettono la pipeline interna dei dati:

1. Thing manager: polla periodicamente i dati dalla TDir per gestire la lista dei servient/Wts attivi e le rispettive TD.
   La lista è utilizzata per raccogliere report periodici da ogni servient.

2. Optimizer: esegue la policy di allocazione per le WT/servient.
   Allo stato corrente dell'implementazione il modulo hosta l'algoritmo di ottimizzazione sbasato su grafi definito nella sezione V e altre policy greedy valutate nella sezione VII.

Tuttavia sottolineiamo il fatto che ogni policy che imlementa l'interfaccia verso lo lo strato superiore (ovvero il thing manager) e lo strato inferiore (ovvero la migrazione) può essere installata ed utilizzata

3. Migration: riceve il piano di deployment dell'optimizer e implementa le procedure di handoff per le WT.
   Prima stoppa l'esecuzione delle WT per migrarle verso i loro nodi, poi, attraverso connettori specifici comanda azioni al Migration Substrate per abilitare il trasferimento fisico dei servient (e delle wt hostate) dai nodi sorgenti ai nodi di destinazionie.

L'architettura M-Wot non dipende su nessuna tecnologia specifica di mobilità del software.
Abbiamo invece introdotto uno strato di astrazione - chiamato il Migration substrate - che può adottare una qualsiasi soluzione allo stato dell'arte (attraverso dei migration connectors adatti) come container docker, macchine virtuali o processi javascript.
Questo connettori attuerano il piano restituito dall'optimizer che utilizzeranno come input.
Concretamente l'implementazione corrente si affida a Docker Swarm come migration connector di default, come meglio dettagliato nella sezione VI.

4.0.3 Servient M-WoT

Infine il framework M-WoT introduce delle piccole modifiche al runtime servient per fornire all'optimizer dati in tempo reale sulle performance del sistema.
Più specificamente uno strato di API di monitoraggio è stato introdotti fra l'applicazione WT e il runtime di scripting come raffigurato nella figura 6.
Questo strato è incaricato di intercettare le invocazione all'api di scripting e di generare periodicamente dei *Thing Report* (Trs).
Questi ultimi possono essere considerati come uno snapshot dell'esecuzione corrente del servient/WT e contengono i valori delle metriche (sia del servient che della WT) richieste dall'optimizer;
Nell'appendice riporteremo un frammento della struttura della TR in uso.
Il monitoring layer esponde tutti dati raccolti attraverso un'azione apposita nelle afforfance che è stata aggiunta alla TD del serveint; invocandola l'Orchestrator può richiedere una nuova richiesta di generazione di TR.

4.1 Esempio di migrazione

Per riassumere le operazioni dei tre componenti presentati fino ad ora, vediamo un esempio di un processo di migrazione di una WT.
Consideriamo due WT/Servient, rispettivamente T_A/S_A e T_B/S_B (con T_A in esecuzione su S_A e T_B su S_B) hostati sui nodi N_1 ed N_2.
Assumiamo anche che T_B abbia consumato T_A e che stia leggendo periodicamente alcune delle sue proprietà.
All'istante t il thing manager interroga S_A e S_B per raccogliere le TR; questo è implementato consumando le TD dei servient e inviando un comando retrieveReport (dettagli nella sezione VI).
Successivamente l'optimizer viene eseguito: una nuova allocazione viene prodotta dove T_A deve essere spostato sul nodo N_2.
La sequenza di operazioni che effettuano la migrazione di T_A da N_1 ad N_2 è mostrata nella figura 7.
Per prima cosa l'esecuzione corrente di T_A viene stoppata: questo viene fatto dall'orchestrator (e più specificamente dal sottomodulo migration) invocando l'azione stop su S_A, che di conseguenza stoppa l'applicazione della WT, pulisce le risorse di sistema recupera il contesto dei dati applicativi (lo stato corrente) e lo ritorna all'orchestrator.
Di conseguenza il contesto applicativo di T_A è salvato come metadati all'intendo di TDir per essere utilizzato successivamente.
In sequito l'orchestrator (utilizzando un connector adatto) invia una richiesta al migration substrate (es docker swarm) per far muovere T_A/S_A al nodo di destinazione (N_2).

Dopo che S_A è stato rigenerato registra la sua nuova TD (con gli indirizzi di rete delle sue affordances aggiornate) nel TDir.
Di seguito interroga la TDir per recuperare il contesto della T_A, quest'ultimo viene deserializzato e iniettato come oggetto globale nello script applicativo della T_A
Infine T_A inizia il processo di inizializzazione e si espone facendo partire la registrazione della propria TD all'interno della TDir.
A questo punto T_A riprende nello stato in cui era prima di essere stoppata ed è considerata completamente migrata.
La TDir invia una notifica a T_B riguardo la procedura di handoff, T_B recupera la nuova TD di T_A dalla TDir e la consuma di nuovo per poter puntare alla posizione del servizio aggiornata.
Infine T_B ricomincia ad interagire con T_A e ad accedere alle sue affordances.

5 M-WOT migration policy

In questa sezione caratterizzeremo formalmente le operazioni dell'optimizer come problema di ottimo multi obbiettivo.
Per lo scopo di questo studio consideriamo un processo di ottimizzazione a due passaggi che considera il problema del load balancing (ovvero quanto carico ha ogni host) e l'overhead delle comunicazioni di rete (ovvero quanti dati vengono scambiati fra gli host).
Il problema di ottimizzazione è definito formalmente nella sezione V-A.
Successivamente un'euristica basata su grafi viene proposta nella sezione V-B, e la sua complessità computazionale è definita formalmente nella sezione V-C.
La tabella 1 riporta la lista delle variabili nella sezione V-A.

5.1 Formulazione del problema

A prescindere dal caso d'uso target, consideriamo un generico deployment WoT con N_{WT} WT attive.
Il sistema si evolve su una serie di time-slot $T = \{t_0, t_1...\}$; ogni slot ha una durata di t_{slot} secondi ed è uguale all'intervallo fra le esecuzioni consecutive della policy di migrazione.
Poniamo $WT = \{wt_1, wt_2, ...wt_{N_{WT}}\}$ come l'insieme delle WT, che possono essere eterogenee in termini di modello dei dati (es. le affordances).
Senza perdere in generalità poniamo $A_i = \{a_i^1, a_i^2, ...a_i^{N_{A_i}}\}$ le affordances esposte dalla wt_i nella sua TD, ogni affordance può rappresentare una proprietà, un'azione oppure un evento.
Presumiamo che A_i sia statico, ovvero wt_i non può aggiornare la propria TD a runtime (es. non può definire nuove proprietà).
Poniamo H l'insieme dei nodi di host, con $H = \{h_1, h_2, ...h_{N_H}\}$ e assumiamo che i nodi siano eterogenei in termini di hardware.
Ogni nodo potrebbe infatti avere una potenza di calcolo diversa, senza perdere in generalità modelliamo la diversa potenza di calcolo attraverso un'indice di potenza computazionale $\gamma(h_l), \forall h_l \in H$ che astrae i dettagli dell'hardware ed è definito come il numero massimo di things che possono essere eseguite sull'host.
L'allocazione dei servient agli host è definita dalla funzione policy $P : WT \times T \to H$;
per ogni WT wt_i il valore $P(wt_i, tk) = h_m$ specifica la macchina (ovvero h_m) che la sta hostando all'istante di tempo t_k.
Basandosi sull'output della policy di allocazione, l'insieme $PT_{m,k} \subseteq WT$ denota la lista delle WT che sono hostate dall'host h_m nel time slot t_k, ovvero $PT_{m,k} = \{wt_i \in WT | P(wt_i, t_k) = h_m\}$.
Secondo l'architettura WoT vista nella sezione due, ogni WT wt_i può interagire con un altra WT wt_j se prima la consuma.
Questo cosa è modellata con l'assunzione che, in ogni time slot t_k, wt_i può inviare una lista di richieste $R_{i,j,k} = \{r_{i,j,k}^1, r_{i,j,k}^2...\}$ alla wt_j consumata; ogni richiesta $r_{i,j,k}^y$ fa riferimento ad un affordance di wt_j, e consiste in:

- Lettura o scrittura di una proprietà

- Invocazione di un evento
- Elaborazione di un evento

Vale la pena evidenziare che la notazione sopra presume che la stessa affordance a_i^x potrebbe essere attivata più volte da wt_i durante lo stesso timeslot, ma vengono considerate due richieste diverse (ad esempio la WT wt_i legge due volte la stessa proprietà a_j^x sulla WT wt_i nel time slot t_k).
L'implementazione di ogni richiesta $r_{i,j,k}^y$ comporta dello scambio di dati fra le WT wt_i e wt_j; definiamo $B(r_{i,j,k}^y)$ come i dati scambiati (in byte) fra le due WT, includendo sia gli eventuali parametri passati da wt_i a wt_j sia eventuali valori di ritorno da wt_j a wt_i.
Il valore $B(r_{i,j,k}^y)$ è incluso nel messaggio TR, che è periodicamente mandato da ogni WT all'optimizer come descritto in precedenza nella sezione 4.
Denotiamo con $B(i,j,k) = \sum B(r_{i,j,k}^y)\ \forall r_{i,j,k}^y \in R_{i,j,k}$ il carico totale di comunicazione fra le WT wt_i e wt_j all'istante di tempo t_k.
Chiaramente, $B(i,j,k) = 0$ sia nel caso in cui wt_i non stia consumando wt_j all'istante t_k o nel caso in cui non ci siano interazioni fra loro (ovvero $R_{i,j,k} = \emptyset$).
L'obbiettivo dell'optimizer è determinare la policy che calcola - ad ogni istante di tempo t_k - il trade-off ottimale fra l'utilizzo di risorse computazionali (ovvero il load balancing fra gli host) e la località dei dati (ovvero quanti dati sono trasferiti fra gli host),
A questo scopo definiamo la metrica del network overhead (NO) come il carico di comunicazione fra gli host (in byte) che occorre a causa delle interazioni fra le WT hostate su nodi diversi. Più formalmente:

$$NO(t_k) = \sum_{wt_i \in WT, wt_j \in WT, P(wt_i,t_k) \neq P(wt_j,t_k)} B(i,j,k)$$

. Chiarifichiamo che la metrica $NO(t_k)$ sopra quantifica le comunicazioni *end-to-end* dello strato applicativo fra i nodi del cluster, generate dalle interazioni fra le diverse WT; non include l'overhead dello strato di rete (es quello casusato da forwarding multi hop di messaggi fra i router).
Questo perché il framework M-Wot è implementato nello strato di applicazione e la conoscenza della topologia della rete sottostante non è nota.
Introduciamo poi la metrica di host fairness (HF) definita come la differenza fra il nodo più carico ed il nodo meno carico del cluster, ovvero:

$$HF(t_k) = \max_{h_m \in H} L(h_m, t_k) - \min_{h_m \in H} L(h_m, t_k)$$

Qui $L(h_m, t_k)$ definisce il carico computazionale di h_m nel time slot t_k ed è collegato al numero di WT hostate diviso la potenza computazionale, ovvero:

$$L(h_m, t_k) = \frac{|PT_{m,k}|}{\gamma(h_m)}$$

Definiamo $p_{wt_i,h_m}^{t_k}$ come la variabile binaria che indica l'allocazione delle WT, definita $\forall t_k \in T$, $\forall wt_i \in WT$ e $\forall h_m \in H$ come segue:

$$p_{wt_i,h_m}^{t_k} = \begin{cases} 1 \text{ se } P(wt_i, t_k) = h_m \\ 0 \ \textit{altrimenti} \end{cases}$$

Attraverso le metriche NO ed HF introdotte sopra, il problema di migrazione può essere formalmente definito come segue:

$$\min_{p_{wt_i,h_m}^{t_k}} \ NO(t_k)$$

Tale che;

$$L(h_m, t_k) \leq 1 \ \forall h_m \in H$$

$$HF(t_k) \leq \Delta$$

Il primo dei due vincoli serve per assicurarsi che l'allocazione su ogni host non ecceda le capacità computazionali per quell'host ($\gamma(h_m)$).
Nel secondo vincolo Δ è un parametro definito dall'utente che quantifica il trade-off menzionato in precedenza.
È facile notare che le metriche HF ed NO sono legate fra loro: minimizzare il carico di rete si può raggiungere con una policy che alloca tutte le WT allo stesso host.
Tuttavia questo costituisce il caso peggiore per l'HF.
Per questo motivo abbiamo due scenari estremi:

1. L'obbiettivo del sistema è quello di minimizzare lo scambio di dati sulla rete, a prescindere dalla latenza del servizio.
   Questo potrebbe essere il caso di uno scenario Iot edge-cloud dove lo stakeholder ha interesse nel minimizzare la quantità di dati trasferita ad un'infrastruttura remota per ragioni di privacy.
   In questo caso $\Delta = \infty$

2. L'obbiettivo del sistema è quello di minimizzare la latenza del servizio, evitando la presenza di bottleneck di performance (es host sovvraccaricati) mitigando comunque le comunicazioni fra host.
   In questo caso $\Delta = 1$

Tutte le situazioni intermedie sono modellate personalizzando il parametro Δ, che presumiamo essere l'input del problema di ottimizzazione.

5.2 Euristica proposta

Proponiamo un'euristica basata su grafi che rispetta il vincolo 7 ($HF(t_k) \leq \Delta$) mentre rilassa il vincolo 6 ($L(h_m, t_k) \leq 1 \; \forall h_m \in H$) e utilizza un approccio greedy per la funzione obbiettivo.
La soluzione si basa sulla costruzione di un grafo delle dipendenze $G(V, E, W, L)$ che modella le interazioni fra le WT:

- V è l'insieme dei vertici, ogni vertice rappresenta una WT, quindi $V = WT$ e $v_i = wt_i, \forall wt_i \in WT$

- E è l'insieme degli archi, ogni arco $e_l(v_i, v_j)$ connette due vertici $v_i, v_j \in V$ e modella le interazioni fra le due WT.
  Più specificamente esiste l'arco $e_l(v_i, v_j)$ se $B(i, j, k) > 0$ oppure se $B(j, i, k) > 0$.

- $W : E \to \mathcal{R}$ è una funzione peso, che assegna un costo ad ogni arco $e_l(v_i, v_j) \in E$. Qui, il valore $W(e_l(v_i, v_j))$ quantifica la quantità totale di dati scambiati fra le WT, in caso wt_i stia consumando wt_j o viceversa, ovvero $W(e_l(v_i, v_j)) = B(i, j, k) + B(j, i, k)$.

- $L : V \to \mathcal{R}$ è una funzione di carico che assegna un costo ad ogni vertice $v \in V$.
  Se assumiamo di conoscere il carico della CPU ($C(r)$) indotto da ogni richiesta ricevuta da wt_j allora possiamo definire $L(v_j)$ in maniera molto precisa come

$$L(v_j) = \sum_{wt_i \in WT} \sum_{r \in R_{i,j,k}} C(r)$$

  In questo paper non presumiamo di avere una conoscenza così granulare, per cui generalmente settiamo $L(v_i) = 1 \forall v_i \in V$, ovvero presumiamo che tutte le WT producano lo stesso carico, mentre il carico totale dell'host h_m (denotato da qui in avanti con $L(h_m)$) è semplicemente il numero delle WT hostate.

Il grafo G è costruito e aggiornato continuamente dall'optimizer processando i messaggi di TR ricevuti da ogni servient.

All'inizio di ogni time-slot (es. t_k) l'optimizer visita il grafo e alloca le WT agli host a seconda dell'output della policy (ovvero i valori $PT(h_i, t_k)$).
Visto che la policy è calcolata un'unica volta per ogni timeslot, ommeteremo la notazione temporale (ovvero i vari t_k) nel resto della sezione.

Il ragionamento dietro la policy proposta è il seguente. Per prima cosa calcoliamo l'insieme di componenti connesse nel grafo delle dipendenze G.
Per costruzione, ogni componente contiene un'insieme chiuso di WT che interagiscono fra di loro; per questo motivo l'overhead di rete che c'è fra componenti del grafo diverse è 0.
Il carico di ogni componente è definito come la somma dei carichi delle sue WT; successivamente le componenti del grafo sono ordinate in base al loro valore di carico e gli host sono associati con un algoritmo round robin.
In caso il vincolo 7 sulla load fairness è rispettato ($HF(t_k) \leq \Delta$), l'algoritmo termina la sua esecuzione.
Altrimenti rompiamo le componenti calcolate fino ad ora (introducendo quindi dell'overhead di rete ad ogni iterazione) migrando iterativamente una WT dall'host più carico all'host meno utilizzato, fino a quando il vincolo 7 è rispettato.
La wt_i migrata è seleta con un approccio greedy come quella che minimizza l'overhead di rete, calcolato come la differenza fra:

- Il nuovo overhead generato quando stacchiamo wt_i dal suo host di partenza
- Il guardagno di performance sull'host di destinazione, causato dal fatto che wt_i è diventato un servizio locale per quell'host

L'algoritmo mostra lo pseudocodice dell'euristica proposta.
Per prima cosa costruiramo il grafo delle dipendenze $G(V, E, W, L)$ e calcoliamo il suo insieme di componenti connesse, indicato con GC alla riga 1.
Il calcolo di ogni componente G_i (ovvero $L(G_i)$) è stimato come la somma dei carichi dei suoi vertici (riga 3).
Dopo ordiniamo l'insieme GC basandoci sui valori di carico e l'insieme degli host H basandoci sulla loro potenza di calcolo, rappresentata dalla metrica γ. A questo scopo, alle righe 5-6 la funzione Sort (non riportata qui) ordina un'insieme passato come primo argomenti in ordine decrescente a seconda dei valori di metrica forniti dal secondo argomento.
Il loop alle righe 8-13 assegna i sotto-grafi agli host con un criterio round robin, aggiornando anche il carico per ogni host come il carico dei suoi vertici/WT (linea 11).
Successivamente verifichiamo che il vincolo di fariness sia rispettato utilizzando la funzione CheckBalanced (righe 30-39) che ritorna gli host associati ai valori di carico più alti e più bassi, rispettivamente h_{max} ed h_{min}.
Se la differenza di carico è più bassa della soglia utente Δ, allora viene ritornata l'allocazione corrente.
Altrimenti un meccanismo greedy è implmentato nel loop alle righe 15-27; qui, ad ogni iterazione, un WT candidata v_s viene migrata da h_max ad h_min (righe 22-25) (aggiornando di conseguenza le informazioni di carico per ogni host) e la condizione di load balancing è verificata nuovamente alla riga 26.
La WT/vertice da migrare (v_s) è scelta come quella che minimizza la funzione di overhead alla riga 21.
La funzione di overhead considera:

- Il numero totale di comunicazioni di rete (in byte) fra v_s e ogni altra WT hostata da h_max, che diventerà ora una comunicazione inter-host e costituirà quindi del network overhead dopo la migrazione della WT (il valore è salvato nella variabili loss alla riga 17)
- Il numero totale di comunicazione di rete (in byte) fra v_s ed ogni altra WT hostata da h_min, che ora avverà localmente (comunicazione intra host) e di conseguenze ridurrà l'overhead di rete (il valore è salvato nella variabile gain alla riga 18)
  Il calcolo dei valori di gain/loss è effettuato utilizzando la funzione helper TotInteractions (righe 41-46) che ritorna il numero totale di interazioni fra una WT/vertice target (v_s) e un insieme di vertici (S) fornito come input, nel grafo delle dipensenze G.

5.3 Complessita computazionale

La complessità computazionale è espressa in termini di N_W (numero di WT) e N_H (numero di nodi) per il caso peggiore.
Alla riga 1 dell'algoritmo vengono calcolate le componenti connesse del grafo G, questa operazione viene completata in tempo $O(N_W)$ con una visita DFS del grafo.
Dopodichè dalla riga 8 alla riga 13 le componenti connesse sono assegnate ai nodi di calcolo, anche questa operazione viene effettuata in $O(N_W)$.
La complessità del loop di bilanciamento (righe dalla 15 alla 26) dipende dal valore di Δ e da come è stata definita la funzione L.
Presumiamo che tutti gli host siano omogenei ($\gamma(h_m) = 1 \forall H$), per cui $L(h_m, t_k) = PT_{m,k}$.
Questa ssunzione è in linea con l'analisi sperimentale presentata nella sezione 7.
Considerando un'allocazione completametne sbilanciata delle WT ai nodi, il loop viene eseguito per $N_W - \Delta$ volte; il loop interno (righe 16-20) ha complessità $O(\frac{N_W}{N_H})^2$ perchè visitiamo ogni WT hostata dal nodo più utilizzato e per ogni WT calcoliamo il NO totale con le WT hostate sul nodo meno utilizzato.
Infine, la funzione checkbalance effettua un loop sull'insieme N_H, per cui avrà complessità $O(N_H)$.
Mettendo tutto insieme, la complessità dell'algoritmo è $O(N_W)$ nel caso in cui la procedura di load balancing non venga eseguita (ovvero $\Delta = \infty$).
In caso contrario la comlessità del dell'algoritmo è dominata dal loop alle righe 14-26, e ha complessità uguale a $O(\frac{N_W^2}{N_H^2}) + O(N_W) \cdot O(N_H)$.
Dal momento che presumiamo che $N_W \ll N_H$ la complessità totale dell'algoritmo 1 è $\approx O(N_W^2)$

6 Implementazione

In questa soluzione dettaglieremo dei componenti architetturali presentati nella sezione 4.
La nostra soluzione estende il framework thingweb node wot, l'implementazione di riferimento uggucuale del WOT, a cui abbuamo aggiunto delle primitive specifiche per supportare il processo di migrazione delle WT.

6.1 Thing directory ed orchestrator

La TDir è implementata come una WT dedicata e non migrabile, che è hostata da un servient WoT che espone un API specifica per gestire le TD e i contesti.
Fra le interazioni più importanti delle affordances citiamo:

- registerThing: azione che prende una TD come input e la rendere globalmente disponibile agli altri componenti M-Wot
- getThingById e listThings: azioni che ritornano rispettivamente una o più TD in base all'id o a un filtro semantico
- getContextById: azione che ritorna il contesto associatoad una WT.
- thingRegistered: evento invocato ogni volta che un WT si registra nella TDir, e fa mandare in broadcast la sua TD a tutti i suoi subscriber.

L'orchestrator è implementato come applicazione Node.js scritta in typescript utilizzando il framework nest con la modalità standalone applicazion.
L'orchestrator include diversi moduli che lavorano in sinergia, e corrispondono ai tre componenti presentati nella sezione 4.2:

- Thing manager: fornisce un TaskManager in grado di eseguire dei task generici ad intervalli prestabiliti. La funzionalità è implementata dal package @nestjs/schedule, che a sua volta utilizza il pacchetto node-cron.
  Fra gli altri task, citiamo il task collectReports che recupera periodicamente la lsita delle WT

attive dalla TDir e invoca l'azione retrieveReport su ognuna di esse per recuperare le relative TR.

- Optimizer: forisce le strutture dati che rappresentano lo stato corrente del deployment M-WoT. In particolare tiene traccia delle metriche live delle Wt (ovvero le interazioni con le altre Wt) e la lista degli host.
  Inoltre fornisce la classe astratta Policy, con un metodo getAllocation che ritorna l'allocazione pianificata delle WT ai nodi (ovvero gli insiemi $PT(h_m, t_k)$ dell'algoritmo 1).
  Ogni policy installata dall'optmizer deve implementare il metodo getAllocation

6.2 Servient WOT

Il tool *node-wot* di default è stato esteso in due modi:

1. Il run time dello script è stato dotato di un proxy per il monitoraggio L'implementazione del CLI di default è stata modificata per per poter gestire l'iniezione e il recupero dello stato della WT.

1. Api di monitoraggio
   Le api di monitoraggio sono una serie di classi e funzioni TypeScript che raccolgono i dati richiedti dall'optimizer.
   Più nello specifico l'api di monitoraggio intercetta tutte le invocazioni che la WT fa alle funzioni di scripting WoT e aggiorna il numero delle attivazioni di ogni proprietà/azione/evento oltre che al tempo necessario per eseguirle.
   Dopodichè salva questi i dati all'interno della TR, la cui struttura è riportata nella sezione 1.
   I campi principali della TR comprendono:

   - l'id della WT che viene monitorata
   - l'hostId del nodo che sta hostando la WT/servient
   - il serviceId utilizzato per mappare la WT al servizio corrispondente nel docker swarm
   - L'utilizzo medio di memoria e CPU del nodo
   - La lista delle interazioni

   La lista delle interazioni contiene statistiche relative alle interazioni con ogni WT consumata, e più nello specifico il numero specifico di volte che una certa afforfance è stata attivata, e la latenza dovuta alla richiesta e risposta.

2. Context migration Nel caso di WT attive il framework M-Wot supporta la migrazione del relativo context, ovvero di tutte quelle informazioni che caratterizzano lo stato interno, includendo

   - le variabili globali dell'applicazione in esecuzione sulla wt.
   - I valori delle proprietà delle WT.
   - Lo stato corrente di eventuali librerie esterne in uso.

   Prima che la migrazione possa iniziare, tutte le possibili operazioni in corso andrebbero interrotte e il contesto va salvato.
   Questo è stato implementato aggiungendo la funzione stop alla TD della WT, che disabilita tutte le affordances per evitare un possibile cambio di stato durante il salvataggio del contesto.
   Dopoodichè la funzione stop salva il contesto della WT e lo ritorna al servient, il contesto viene poi salvato nella Rdit come descritto nella sezione 4.
   Dopo che il nuovo servient è stato deployato, e prima di mandare in esecuzione la WT migrata, il servient fa una richiesta al TDir (utilizzando il thing id) per recuperare il contesto.
   Il contesto è poi passato alla WT per essere caricato, restorando così lo stato della WT prima della migrazione.
   Per semplificare e per facilitare i compiti del programmatore abbiamo automatizzato il processo

di aggiunta delel funzionalità ausiliare all'interno del behaviour dlla WT.
Più nel dettaglio, i metodi per stooppare le affordances della WT e ritornare il contesto sono automaticamente iniettati nel codice dell'applicazione WT dal servient prima di esporla.
Il servient cerca per un commento specifico nello script per capire se e quando il codice M-Wot vada inserito.
L'unica operazione richiesta al programmatore per consentire la migrazione delle WT è quella di aggiungere il commento al codice della propria applicazione.

7 Valutazione

In questa sezione testiamo la performance del framework M-Wot con un valutazione sperimentale twofold.
Per priam cosa nella sezione 7.1 confronteremo diverse poliy di migrazione, incluse diverse varianti dell'euristica basata su grafi presentata nella sezione 5 con casi limiti creati ad hoc.
Successivamente nella sezione 7.2 investigheremo sull'effiaccia del meccanismo di migrazione delel WT sul continuo cloud-edge.
Più nel dettagio valuteremo un'applicazione reale di monitoraggio IoT di strutture, ispirata da uno dei casi d'uso presentati nella sezione 3.
Le caratteristiche e i parametri di ogni scenario sono discusse separatemente nelle sezioni 7.1 e 7.2.

7.1 Analisi delle policy

Consideriamo un setup distribuito composto da tre server sull'edge ($N_H = 3$) dislocati fisicamente nei data center DISI/ARCES all'università di Bologna e connessi con una LAN ethernet con una distanza di hop di uno fra di loro.
Specificatamente due server sono equipaggiati con cpu a 4 core da 2 GHz e 4 giga di ram, mentre il terzo server è quipaggiato con un intel xeon e5440 e 32 giga di ram.
Inoltre l'orchestrator e il TDir sono stati installati su un altro server nello stesso data center.
Per cui in totale il setup dell'esperimento è dato da 4 nodi, di cui tre costituiscono lo spazio di deployment del M-Wot e possono essere utilizzati per hostare le WT.
Su questo spazio abbiamo deployato N_{WT} servient, dove ognuno hosta esattamente una WT.
All'avvio i servient sono alloati asualmente su tutti i N_H nodi disponibili.
Le interazioni fra le WT sono modellate come segue.
Deviamo dal significato fisico delle WT e la sua corrispondenza con applicazioni del mondo reale perchè vogliamo concentrarci sullle operazioni di migrazione e sulla valutazione delle performance delle poliy.
Per questo motivo ogni WT espone esattamente un'azione denna sua TD (es. test) che calcola una sequenza di operazioni trigonometriche (tan e atan) per generare del carico sulla CPU.
Ogni WT (es. wt_i) consuma esattamente N_C WTm scekte a casi fra le N_{WT} disponibili.
Su ogni wt_j consumata wt_i manda una rihiesta ogni 1.5 secondi.
Per applicare automatiamente le configurazioni di test su ogni WT abbiamo implementato un'applicazione di mashup, ovvero un client WoT che è incaricato di consumare le WT e di passargli il giusto setup (ovvero le WT che devono consumare).
Ogni $t_f = 45$ secondi l'orchestrator raccogliere il thing report prodotto da ogni servient, ogni 190 secondi una nuova allocazione delle WT è calcolata dall'optimizer secondo la policy corrente eed implementata attraverso la migrazione delle WT appropriata sui server dell'edge.
Quest'ultima è anche la durata di un time slot ($t_{slot} = 190$ secondi), in accordo alla formulazione del problema presentata nella sezione 5.1.
Il settaggio dei parametri t_f e t_{slot} consente all'optimizer di raccogliere almeno tre report da ogni WT e quindi di stimare le interazioni fra WT prima di calcolare una nuova allocazione delle WT sui nodi.
L'analisi delle performance si basa sulle seguenti metriche:

- Network overhead (NO): indice di performance che quantifica la quantità di comunicazioni di rete fra host prodotte dalle interazioni remote delle WT.

A differenza del modello teorico, calcoliamo il NO in termini del numero di interazioni e non in bytes, dal momento che tutte le interazioni delle WT sono relative alla stessa affordance (ovvero l'azione test); questo è equivalente a settare $B(i,j,k) = 1, \forall wt_i, wt_j \in WT, t_k \in T$

- CPU fairness (CG): è l'indice di performance che quantifica la giustizia del bilanciamento del carico in termini della differenza del carico di CPU massimo e minimo fra i N_H nodi del cluster. Settiamo $\gamma(h_l) = 1, \forall h_l \in H$.

- Thing fairnes (TF): simile alla metrica CF, tuttavia in questo caso la giustizia del bilanciamento è espressa in termini di numero di WT hostate rispettivamente dal nodo più carico e dal nodo meno carico (in alnternativa al carico medio delle CPU)

- Interaction Lateny (IL): la latenza media necessaria per effettuare un'invocazione di un azione richiesta da una WT esterna; più esplicitamente questo è il tempo medio che passa da quando wt_i richiede un'azione test a wt_j a quando la risposta corrispondente viene ricevuta.
  Per questo motivo tiene in considerazione sia la la tenza dovuta all'elaborazione sia la latenza di rete nel caso in cui wt_i e wt_j siano in esecuzione su diversi nodi del cluster.

Abbiamo confrontato le seguenti policy:

- NoMigrate: questa è la solizione WoT allo stato dell'arte, ovvero le WT sono deployate staticamente sui nodi e non sono migrate durante tutta la vista del sistema.

- Greedy netload: questa è una policy greedy che puntata minimizzare la metrica NO.
  Ad ogni timeslot prende la WT che genera il NO più grande e la sposta nello stesso nodo della WT che la consuma.

- Greedy CPUload: questa è una policy greedy che punta a minimizzare la metrica CF.
  Ad ogni timelost seleziona il nodo dell'edge nel cluster he ha associato il più alto carico sulla CPU, preleva una WT e la muove verso il nodo che il carico di CPU più basso.

- Graph based, $\Delta = \infty$: questa è la policy basata su grafi presentanta nella sezioni 5; settiamo $\Delta = \infty$, per cui la policy punta esclusivamente a minimizzare la metrica NO, mentre nessun'azione viene effettuata per il load balancing (vengono saltate le righe 16-26 dell'algoritmo).

- Graph based, $\Delta = 5$, questa è ancora la policy della sezione 5, con però l'utilizzo del parametro di bilanciamento.
  Questa policy calcola una soluzione che minimizza il NO assicurandosi che la metrica TF non superi la soglia Δ uguale a 5.

- Grah based, $\Delta = 1$ questa policy è simile alla precedente, tuttavia settiamo il massimo bilanciamento delle allocazioni delle WT sui nodi del cluster.

Per ogni configurazione abbiamo eseguito 10 ripetizioni e abbiamo fatto una media dei valori delle metriche; per ogni ripetizione sono state considerate:

- un'allocazione casuale delle WT ai nodi

- Dipendenze casuali fra le WT

Le immagini 8(a), 8(b), 8(c) e 9(a) mostrano le metriche introdotte in precedenza al variare delle policy e le configurazioni di N_{WT} (ovvero il numero di WT coinvolte nello scenario).
Il valore N_C è fissato a 3, ovvero ogni WT consuma esattamente altre 3 WT, scelte a caso.
Dai valori di NO della figura 8(a) possiamo notare che la quantità di comunicazioni fra host aumenta con il numero di WT attive, come atteso.
Allo stesso tempo le poliy graph based e netload sono più efficaci della nomigrate e della cpuload dal

momento che entrambe mirano ad allocare le WT che interagiscono fra di loro sullo stesso nodo; Il miglioramento della metrica NO della policy graph based può essere regolato utilizzando il parametro Δ.
Con $\Delta = \infty$ il NO è sempre 0, dato che il grafo delle dipendenze risulterà essere probabilmente un grafo connesso (questo anche perchè $N_C = 3$); come risultato tutte le WT sono spostate sullo stesso nodo edge, come mostrato più nel dettaglio di seguito.
Per $\Delta = 1$ e $\Delta = 5$ le poliy basate su grade introduono del NO a causa dei vincoli di load balaning, ma il NO è comunque più basso del NoMigrate, per cui è comunque preferibile che ad un allocazione fissa.
La capacità di load balancing delle sei policy sono analizzate dalla figura 8b, che mostra la matrica TF in funzione del numero delle WT.
Con la policy graph based con $\Delta = \infty$ la TF è sempre uguale al numero delle WT dello scenario, perchè tutte le WT sono allocate allo stesso nodo.
Notiamo che se $\Delta = 1$ oppure $\Delta = 5$ il valore di TF è sempre più basso della soglia richiesta, dimostrando l'efficacia del meccanismo di load balancing.
La giustizia in termini di WT si traduce in un miglior utilizzo delle risorse computazionale, come mostrato nella figura 8(c),
Qui la metrica CF è mostrata per le 6 policy.
Notiaiamo che l'euristica graph-based con $\Delta = \infty$ e $\Delta = 1$ sono rispettivamente il caso peggiore e il caso migliore, ancora una volta dimostrando la versatilità del nostro approcio.
Comparando le figure 8(a) e 8(c) possiamo anche apprezzare che le policy graph-based (con $\delta \neq \infty$) sono in grado di ottenere un migliore trade-off fra NO e CF rispetto alle due policy greedy. In base ai requisiti del sistema (località dei dati o utilizzo delle risorse) l'amministratore può ottenere il trade off fra performance e utilizzo delle risorse aggiustando il parametro Δ, il cui settaggio ottimale dipende chiaramente dallo senario.
La figura 9(a) fornisce ulteriori informazioni sull'allocazione delle WT fornendo per le varie policy graph based e diversi valori di Δ l'utilizzo di CPU medio per ogni nodo del cluster (indicato dai colori di ogni barra); i valori di utilizzo della cpu sono normalizzati fra 0 e 100%.
È facile notare che utilizzando valori di Δ più bassi siha un utilizzo più bilanciato delle risorse computazionali del cluster, mentre con $\Delta = \infty$ viene utilizzato un solo nodo.
Infinie la figura 9(b) mostra la metrica IL per le sei policy: evidenziamo che la latenza non è presa in considerazione nelle ottimizzazioni della sezione 5, anche se policy orientate alla riduzione della latenza possono essere progettate e installate sullìoptmizier in futuro.
In ogni caso, la poliy graph based con $\Delta = \infty$ supera le altre poliy per tutte le onfigurazioni di WT, questo è dato dalla riduzione della latenza di comunicazione data dal fatto che tutte le interazioni fra WT avvengono localmnente sullo stesso nodo.
Nelle figure 10(a), 10(b) e 10(c) espandiamo le valutazioni considerando l'impatto delle diverse quantità di interazioni fra WT sulle performance del sistema.
Più specificamente consideriamo un numero fisso di WT ($N_{WT} = 15$) mentre sull'asse del x faciamo variare il grado delle WT (N_C) ovvero il numero di peers consumati da ogni WT, selezionate anche in questo caso in maniera casuale.
La figura 10(c) mostra la metrica NO per le sei policy; come atteso il numero di comunicazioni aumenta all'aumentare del valore N_C sull'asse delle x.
L'unica eccezione è la policy graph based con $\Delta = \infty$: similmente a quanto visto in precedenza il NO è 0 dal momento in cui le WTs che interagiscono fra loro sono allocate sullo stesso nodo, tuttavia più di una componente connessa è presente sul grafo delle dipendenze per $N_C = 1$ e $N_C = 2$.
Come risultato la metrica CF della policy graph based con $\Delta = \infty$ mostra il trend crescente della figura 10(a)m per $N_C = 1$ e $N_C = 2$ un'allocazione più bilanciata è ottenuta perché i componenti del grafo sono allocati su nodi diversi, mentre per $N_C = 3$ il grafo è completamente connesso per cui l'intero grafico è allocato sullo stesso nodo.
Confrontando 9(c) e 10(a) possiamo apprezzare anora ome le policy graph based (con $\Delta \neq \infty$) sono in grado di raggiungere un miglior compronesso NO-CF delle polocy nomigrate e greedy.
Questo si traduce in aumento rilevante delle performance per le poliy graph based per la metrica IL

nella figura 10(b); per $N_C = 1$ la riduzione della latenza fornita dalla policy graph based rispetto alla policy no migrate sale del 37% con $\Delta = \infty$ e del 13% con $\Delta = 5$.
Nelle analisi presentate fino ad ora abbiamo considerato scenari WoT dove il numero delle WT è fissato all'avvio, per cui il processo di scoperta delle WT può essere considerato statico nel tempo.
Nelle figure 10(c) e 11(a) abbiamo analizzato le performance del M-Wot in un ambiente dinamico dove il numero delle WT attive (e quindi la quantità di traffico) varia nel tempo.
Più specificamente abbiamo inzializzato il sistema con $N_{WT} = 0$.
Ogni 360 secondi una nuova WT è creta e aggiunta allo scenario, ogni WT consuma esattamente un peer ($N_C = 1$).
La figura 10(c) mostra la metrica NO Dduante l'evoluzione del sistema, espressa in time-slot.
Ricordiamo che ogni un timeslot corrisponde ad un esecuzione della policy dell'optimzier, e questo evento occorre ogni 190 secondi.
È facile notare che la metrica NO aumenta significativamente nel tempo per la policy NoMigrate come conseguenza della creazione delle nuove WT, e quindi delle comunicazioni inter host aggiuntive introdotte nel sistema. Al contrario le policy graph based sono in grado di gestire le allocazioni delle nuove WT così da rispettare sempre l'obbiettivo di minimizzazione del NO .
L'adattabilità del framework m-wot alle condizioni del carico di rete è dimostrata ulteriormente dalla figura 11(a) che mostra la metrica TF allo scorrere dei time slot, per il caso della policy graph based con $\Delta = \infty$ la TF aumenta nel tempo come conseguenza del fatto che aggiungengo nuove WT nel sistema - componenti connesse del grafo più grandi potrebbero crearsi ed essere migrate allo stesso nodo.
Al ontrario, le policy Graph based con $\Delta = 5$ e $\Delta = 3$ allocano dinamicamente le WTs così che i vincoli di bilancimento (rappresentati dal valore Δ) siano sempre soddisfatti.
infine abbiamo calcolato la scalabilità delle soluzioni proposte monitorando il consumo di CPU e RAM sui nodi dell'orhestrator e del thing directory.
Le figure 12(a) e 12(b) mostrano quanto trovato.
I rusltato sono stato ottenutoi campionando le metriche dei container ogni secondo, facendo poi una media dei risultati per diversi numeri di WT deployate.
È possibile notare che il consumo cresce linearmente ma è trascurabile anche con 100 WT.
Inoltre, l'overhead introdotto dalla policy graph based è solo leggermente più alto di una policy nomifrate, anche se m-wot deve eseguire le procedure di allovazione delle WT e le procedure di handoff descritte nella sezione 4.4.
Chiaramente, nonostante questi risultati positivi, l'orchestrator centralizzatio potrebbe fare da bottleneck su deployement WoT su larga scala; per affrontare il problema possiamo pensare all'utilizzo di uan rete di orchestrator, dove ognuno controlla una specifica regione di nodi.
Framework m-wot di questo tipo richiederebbo meccanismi adeguati di replica dei dati, load balancing e di gossiping, che abbiamo in programma di investigare in lavori futuri.

7.2 Analisi di un caso d'uso

Consideriamo un'applicazione di monitoraggio IoT che replica le operazioni del caso d'uso del monitoraggio dinamico strutturale presentato nella sezione 3.
Specificamente, assumiamo he un sistema W3C WoT sia stato progettato per acquisire e processare i dati Iot di una smart building. Il sistema WoT comprende tre Wt:

- Una sensing WT che effettua acquisizione di dati da un sensore IoT (ad esempio un accelerometro) utilizzando una connessione seriale.
  Più specificamente, assumiamo che le WT sensing possano operare in due modi, che differiscono per la loro sensor query frequency (qf). Avremo quindi:
  - La modalità normale, con un campionamento ogni 5 secondi
  - La modalità warning, con un campionamento ogni secondo

  Il cambio di modalità avviene quando le ultime tre lettura onsecutive sono più alte o più basse di una soglia statia, in altre parole la granularità del sisemta di monitoraggio si aggiusta in base

al rilevamento di possibili anomlie nei dati.

- Una processing WT che riceve continuamente i dati in tempo reale dalla WT sensing e applica un metodo statistico per prevedere i prossimi valori del sensore.

- Una reporting WT che produce una notifica (es un allarme) basandosi sull'output della processing wt

Sorvoliamo sul significato fisico dei valori di sensing della WT, mentre ci concentriamo sulle capacità del sistema WoT di minimizzare la la tenza di elaborazione, specialmente nella modalità warning, ovvero vogliamo minimizzare il tempo che passa da quando il dato è acquisito a quando il dato previsto è prodotto in output.
Consideriamo un setup iniziale con due nodi ($N_C = 2$), rispettivamente un server di edge (connesso al sensore IoT) e un server cloud su unternet.
Due scenari sono configurati e confrontati nell'analisi:

- Migration OFF: questo rappresenta lo stato dell'arte degli ambienti WoT, dove la migrazione delle WT è disabilitata.
  Le WT sensing e reporting sono deployate sul nodo di edge, mentre la processing WT è deployata sul cloud per via della maggiroe potenza computazionale

- Migration ON: questo corrisponde all'ambiente M-Wot dove la WT procesing è configurata per essere mirabile, ovvero può essere mossa dinamicamente sull'edge o sul cloud a seconda della modalità dei sensori.
  A questo scopo, deployamo nell'optimizer una policy speifica per lo senario che controlla il numero di interazioni fra le WT sensing e processing in ogni time slot: in caso questo valore sia alto di una certa soglia , l'optimizer realizza che la WT sensing è in modalità warning per cui migra la WT processing sull nodo dell'edge, migrando quindi più vicino al punto di acquisizione per minimizzare la latenza. Altrimenti, la processing WT è allocata nel nodo nel cloud.

Nell'ambiente di test la sensing WT inizia in modalità normale per 5 seondi, poi passa alla modalità warning per 1 secondo, poi ancora ripete la stessa sequenza per altre due volte. La figura 11(b) mostra la metria NO con il passare dei time slot: per la configurazione migration OFF il valore NO in ogni slot è uguale al numero di messaggi scambiati dalle WT sensing e processing, visto che sono hostate su nodi diversi.
Il pico orrisponde agli intervalli in cui la WT sensing passana in modalità warning. È interessante notare che:

1. La configurazione Migration ON segue la stessa curva di migration OFF quando la comunicazione inter host è sotto una certa soglia

2. Il NO della migration ON è zero in corrispondenza dei periodi di warning, dato che la proessing WT è migrata all'edge node, per cui tutte le comunicazione avvengono localmente

Queste azioni impattano l'utilizzo delle risorse computazionali sui nodi di cloud/edge oltre che alla latenza di elaborazione: la latenza è riportata nella figura 11(c).
Possiamo notare l'efficacia del framework M-Wot in termini di riduzione della latenza per la configurazione Migration On che è più evidente durante i periodi di warning perchè la latenza fra edge e cloud è eliminata.

7.3 Conclusioni

Il WoT W3C costituisce un approccio recente e promettente per progettare sistemi IoT su larga scala composti da diversi componenti eterogenei che interagiscono fra di loro.
Lo standard attuale riguarda sia WT fisiche che virtuali, tuttavia presume un allocazione statica delle WT ai nodi computazionali, introducendo quindi dei bottleneck di performance in scenari Iot dinamici caratterizati da interazionni fra WT che cariano nel tempo.

In questo paper abbiamo puntato a superare questi problemi proponendo M-Wot, un nuovo framrwork software che supporta migrazioni live e allocazione dinamica delle Wt fra i nodi computazionali nel continuo cloud edge.
La soluzione proposta sfrutta la presenza delle interface uniformi e ben definite delle WT (ovvero le TF) per supportare meccanismi di mgirazione stateful delle WT oltre che ad abilitare allocazioni dinamiche di WT ai nodi del continuo cloud edge.
Abbiamo proposto ed implementato un architettura centralizzata M-Wot con nuovi componenti software per la mobilità software delle WT, la gestione dell'handoff delle WT, la gestione dell'handoff, la gestione del contesto, il monitoraggio dei servient e l'ottimizzazione dei deploymente Wot.
Riguardo all'ottimizzazione del deployment abbiamo affrontato il problma di massimizzare la località dei dati WOT bilanciando allo stesso tempo il carico sulle riosrse computazionali; a questo scopo, un euristica centralizzata che calcola un tradeoff fra località dei dati e giustizia del carico di lavoro a descrzione dell'utente è stata proposta.
Infine abbiamo verificato il guadagno di performance delle policy proposte e dell'efficacia del framework m-wot tramite due casi di test caratterizzati da densità di WT statiche e dinamiche e dversi volumi di carico.
Essendo uno studi pioneristico sulla mobilità dei servizi in ambienti WoT c'è spazio per diverse estensioni per quanto riguarda l'architettura software, la definizione delle policy e l'analisi.
L'architettura entralizzata potrebbe soffire si problemi di single point-of-failure e di scalablità su ambienti di larga sala; a questo scopo una soluzione immediata potrebbe essere quella di impiegare diversi orchestrator distribuiti, dove ognuno gestisce un sottinsieme delel WT disponibili e si ocupa di task colalborativi ona la scoperta dei servient e la replicazione del contesto.
Similmente, delle metriche aggiuntivi possono essere racolte dallo strato di API di monitoring dei servient (es. l'utilizzo della banda) e conseguentemente nuove policy multi obbiettivo potranno essere definite nel optimizer m-wot; dato il grande numero di parametri che potenzialmente potrebbero modificare le performance dei deployment WoT, siamo interessati all'applicazione di tecniche di machine learning (e principalmente di approcci deep reinforcement learning) per deployment adattivi senza soluzione di continuità in ambient Wot distribuiti.
Infine, abbiamo in programma di testare in maniera ulteriore l'efficacia dei meccanismi di migrazione in scenari SHm del mondo reale, come il monitoraggio di grandi strutture civili ed industriali (come ponti o contenitori pressurizzati); qui investigheremo la possibilità di ri configurare dinamicamente il carico di lavoro sui nodi del cloud/edge basandosi sui requisidi di qualità del servizio dei sistemi di monitoraggio e g sugli stream di dati in tempo reale fra le Wt.