

Relazione fog05 – Gozzi Marco

Introduzione

La continua e massiccia diffusione di dispositivi a basso costo capaci di generare dati e fornire servizi all'utenza sta rendendo impossibile la gestione di questo flusso di informazioni utilizzando classici datacenter per motivi di latenza, throughput, consumo di banda e scalabilità.

Il paradigma Fog Computing si pone come obiettivo la mitigazione di questi problemi avvicinando il processamento dei dati alla loro sorgente, sfruttando le risorse computazionali inutilizzate sui nodi vicini.

Questo approccio pone come principali problemi l'eterogeneità dei nodi e la loro federazione e interoperabilità: i vari nodi differiscono per architettura, risorse disponibili e ambiente d'esecuzione e spaziano da semplici attuatori/sensori a switch e router fino a completi proxy server e base station.

Alcuni nodi possono inoltre non essere dotati di uno stack TCP/IP completo ma parziale, supportando protocolli ad-hoc per nodi a risorse limitate come CoAP e MQTT.

Le applicazioni fog-aware possono quindi sfruttare la differenza di risorse tra i vari layer (fig. 1) per ottenere il miglior compromesso tra banda utilizzata, latenza e computazione necessaria; ad esempio, ipotizzando un'applicazione in esecuzione sui nodi terminali, è possibile utilizzare i nodi nel layer fog per ottenere una prima veloce ma approssimativa risposta e lasciare al layer superiore rappresentante il cloud la computazione di una risposta corretta e completa che arriverà al nodo terminale con maggior ritardo. È così possibile implementare applicazioni latency-aware senza rilassare vincoli di consistenza e correttezza in modo significativo.

Caratteristica importante del paradigma Fog Computing è la virtualizzazione poiché è opportuno che alcune funzioni siano rese indipendenti dal nodo su cui eseguono e che sia possibile migrarle con facilità dove necessarie.

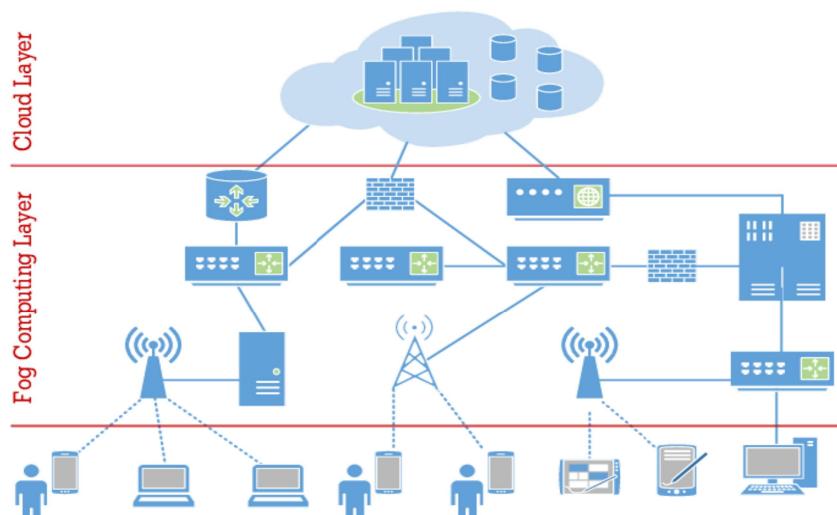


Figure 1: Architettura a layer - Fog Computing

fog05

fog05 nasce dal bisogno di una infrastruttura software che si occupi della federazione di risorse (computazionali, di memoria, di rete e di I/O) fornendo supporto a sistemi altamente eterogenei nell'ambito del Fog Computing.

L'obiettivo è quello di un sistema decentralizzato e modulare capace di funzionare sul maggior numero di nodi possibili, inclusi quelli con risorse estremamente limitate, connessi ad una stessa rete, indipendentemente dalla loro posizione.

fog05 fornisce una astrazione unificata di metodi di virtualizzazione differenti tra cui runtime, hypervisor e networking, permettendo la realizzazione di sistemi eterogenei e completamente virtualizzati: questo è reso possibile da un ricco modello di informazioni accessibile tramite API.

Architettura

fog05 è suddiviso in vari componenti indipendenti, ciascuno accompagnato da un file di configurazione json che contiene le dipendenze necessarie e alcuni parametri di configurazione tra cui "ylocator" che fornisce l'indirizzo del broker e "nodeid" che rappresenta l'identificativo del nodo.

Il componente Agent si occupa della gestione del nodo su cui esegue, effettua advertising delle risorse disponibili ed agisce come proxy per altri nodi resource-constrained; la sua esecuzione è necessaria affinché il nodo venga riconosciuto come partecipante ad un sistema fog05. Lo stato è mantenuto tramite un protocollo distribuito di pub/sub e storage/querying chiamato [zenoh](#).

Il layer di astrazione unificata è fornito da diversi plugin componibili per ottenere il supporto ai runtime desiderati:

- plugin-os-[linux, windows]: si occupa di gestire il filesystem in ambiente Linux/Windows ed estrarre le informazioni relative allo stato delle risorse;
- plugin-net-linuxbridge: permette di creare, gestire ed eliminare reti, interfacce e bridge di rete in ambiente Linux;
- plugin-fdu-[containerd, kvm, lxd, ros2, native]: ciascun plugin aggiunge il supporto per la singola tecnologia di virtualizzazione/containerizzazione; il plugin native permette di eseguire file binari presenti nel filesystem del nodo.

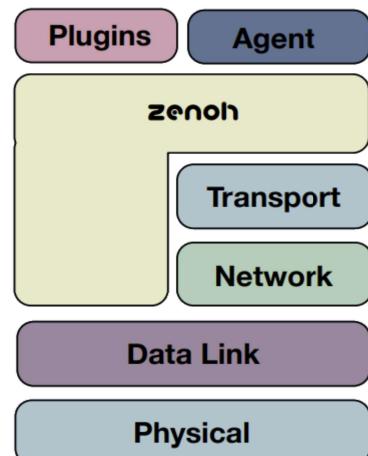


Figure 2: Architettura fog05

Sono disponibili interfacce in [Go](#), [OCaml](#) e [Python3](#) per creare nuovi plugin; è compito del plugin gestire il ciclo di vita delle FDU implementando le API dichiarate dalle interfacce.

Un sistema indipendente fog05 (fig. 3) è un sistema in cui:

- una istanza di zenoh è in esecuzione su un nodo detto broker e in ascolto su una certa porta;
- l'agent di ciascun nodo ha stabilito una connessione al broker.

L'interazione con fog05 avviene due set di API distinti, a seconda delle funzionalità e astrazioni necessarie:

- Fog Orchestration Engine (ForcE): fornisce funzionalità di orchestrazione e gestione di applicazioni e servizi (Entities), verifica i vincoli dichiarati all'interno dei descrittori di Entity (Manifest) e implementa con un algoritmo first-fit l'allocazione delle singole unità di lavoro che compongono ciascuna Entity;
- Fog Infrastructure Manager (FIM): virtualizza l'infrastruttura hardware sottostante fornendo primitive per la gestione e il monitoraggio; le astrazioni presenti sono:
 - Resource: qualunque risorsa assegnabile in modo esclusivo o condiviso, tra cui risorse computazionali (CPU, GPU, FPGA), storage (RAM, disco), rete e I/O (I2C, GPIO, COM);
 - Node: contenitore di risorse, fisico o virtuale, su cui è in esecuzione un'istanza di Agent fog05;
 - FDU: una Fog Deployment Unit rappresenta una unità di lavoro indivisibile (file binario, uni-kernel, container, VM); una FDU richiede un insieme di risorse per il proprio funzionamento, espresso mediante file json;
 - Network: rappresenta reti di livello 2 e 3 tra un insieme di FDU.

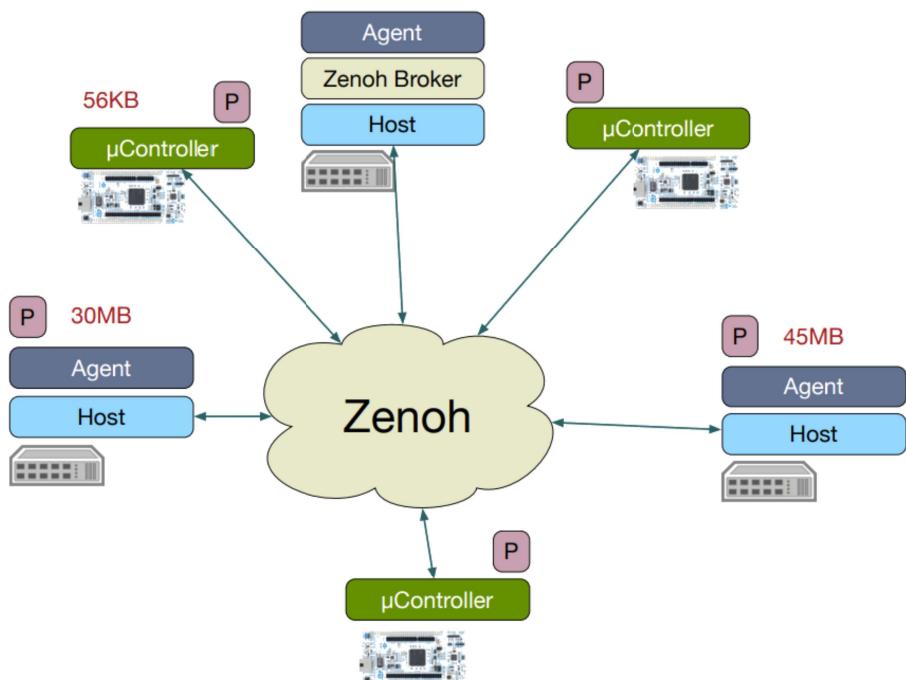


Figure 3: esempio sistema fog05

FDU descriptor

Per poter eseguire la propria applicazione è necessario definire un descrittore di tale applicazione; un descrittore prevede campi comuni, come ad esempio id, nome e requisiti computazionali, e altri specifici per i singoli plugin.

Ad esempio, il plugin per il supporto a containerd e immagini OCI è scritto in linguaggio Go a partire dal relativo sdk: i campi ammessi da tale sdk sono definiti nel file fdu.go; nell'implementazione del plugin vengono utilizzati questi campi per l'interazione con il demone containerd, come l'attributo "uri" di "image" che indica l'indirizzo a cui è reperibile l'immagine OCI da eseguire.

```
{  
  "id": "test-app-docker",  
  "name": "test-app",  
  "computation_requirements": {  
    "cpu_arch": "x86_64",  
    "cpu_min_freq": 0,  
    "cpu_min_count": 1,  
    "ram_size_mb": 64.0,  
    "storage_size_gb": 0.1  
  },  
  "image": {  
    "uri": "docker.io/marcogoz",  
    "checksum": "",  
    "format": ""  
  },  
  "storage": [],  
  "hypervisor": "DOCKER",  
  "migration_kind": "COLD",  
  "interfaces": [],  
  "io_ports": [],  
  "connection_points": [],  
  "depends_on": []  
}
```

Figure 4: FDU descriptor

FDU lifecycle

Ciascuna FDU segue un ciclo di vita definito tramite la fsm di figura 5.

Prima di poter eseguire una FDU di cui si ha il descriptor, è necessario inserire tale descriptor all'interno del catalog del sistema fog05 tramite chiamata alla API *onboard*; ora è possibile passare dallo stato "UNDEFINED" a quello "DEFINED" tramite primitiva *define*, eventualmente indicando il nodeid del nodo su cui verrà preparata l'istanza della FDU.

A partire dallo stato "CONFIGURED" la FDU viene istanziata, ossia il plugin che fornisce il supporto al runtime necessario esegue alcune operazioni affinché al momento dell'esecuzione il supporto del runtime sia operativo.

Il passaggio allo stato "RUNNING" comporta l'esecuzione dell'istanza nel runtime sottostante corretto, cioè nell'hypervisor/container engine oppure in un processo nativo.

Gli stati "PAUSED", "SCALING" e "MIGRATING" sono supportati solamente per alcune FDU: ad esempio, un'applicazione nativa non supporta alcuno di questi stati mentre una FDU basata su VM può potenzialmente trovarsi in ciascuno di questi.

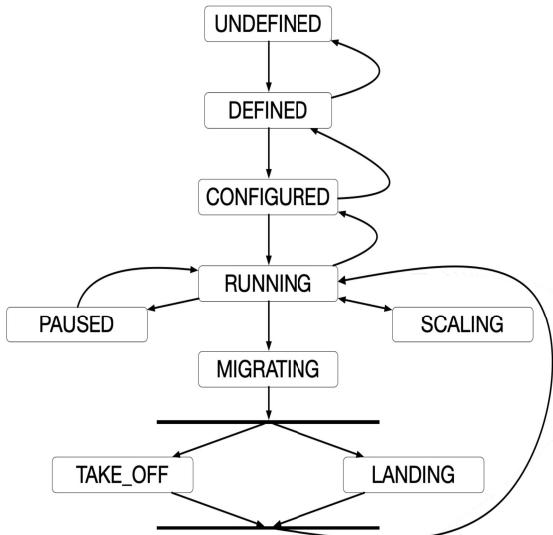


Figure 5: FSM del ciclo di vita di una FDU

Installazione

Sono disponibili [binari compilati](#) per sistemi Linux basati su Ubuntu.

Installare zenoh_{VERSION}.deb, fog05_{VERSION}.deb e libzenoh-{VERSION}.deb utilizzando apt, dpkg o altri strumenti.

Installare i moduli Python3 fog05-sdk==0.2.1, fog05==0.2.1 e zenoh==0.3.0 tramite pip3.

Installare i plugin desiderati, tra cui os-linux e net-linuxbridge, oltre a quelli necessari al supporto delle applicazioni che verranno eseguite.

Ogni plugin installato si collega al broker del sistema di cui fa parte secondo quanto indicando nel file di configurazione, locati in /etc/fos/agent.json per l'agent e in /etc/fos/plugins/<plugin-dir>/<plugin>.json per ciascun plugin.

FIM API

Al momento, le API per interagire direttamente con il FIM sono disponibili in tre linguaggi (Go, OCaml e Python3).

Principalmente, permettono di gestire il ciclo di vita delle FDU andando ad interagire con il broker per modificare lo stato del sistema mantenuto tramite zenoh; è possibile anche gestire reti virtuali, punti di connessione e indirizzi IP floating.

Prendendo in considerazione l'implementazione in Python3 le principali API sono:

- `onboard(self, descriptor)`: registra un FDU descriptor nel catalog del sistema;
- `define(self, fdUUID [,node_uuid=None] [, wait=True])`: definisce che un'istanza della FDU con UUID specificato sarà creabile sul nodo con UUID corrispondente a `node_uuid`; nel caso il parametro sia omesso, un nodo viene selezionato casualmente tra quelli presenti nel sistema;
- `start(self, instanceid [,env=""] [, wait=True])`: mette in esecuzione l'istanza di FDU nel nodo in cui era stata definita e configurata; `env` può contenere variabili d'ambiente che verranno passate al runtime al momento dell'avvio; il passaggio delle variabili segue il formato "VAR1=val1,VAR2=val2,VARn=valn";
- `instantiate(self, fdUUID, nodeID [,wait=True])`: shortcut per chiamare in successione `define`, `configure` e `start`; non permette di passare variabili d'ambiente all'esecuzione dell'istanza;
- `stop(self, instanceid [,wait=True])`: termina l'istanza;
- `undefine(self, instanceid)`: rimuove la FDU dal nodo;
- `terminate(self, instanceid [,wait=True])`: shortcut per chiamare in successione `stop`, `clean` e `undefine`.

Plugin-fdu-native

Questo plugin fornisce il supporto all'esecuzione di applicazioni binarie native nel nodo su cui è installato.

Il descrittore di una FDU per un'applicazione nativa deve contenere i seguenti campi:

```
"command": {  
    "binary": "/path/to/binary",  
    "args": ["some_argument_to_pass"]  
}
```

Inoltre il campo "hypervisor" deve contenere la keyword "BARE".

Il plugin effettuerà una chiamata al file binario indicato nel campo "binary" passando gli argomenti contenuti nel campo opzionale "args"; in ambiente Linux, la chiamata è effettuata dall'utente che ha eseguito il plugin: questo significa che, in caso di installazione effettuata tramite pacchetti precompilati (ad esempio .deb), l'utente è specificato nel file di servizio systemd fos_native.service; a default, in tale file l'utente specificato è "fos".

Con la chiamata alla API "configure" il plugin genera i file di log, predisponde la rete virtuale secondo quanto specificato nel FDU descriptor e regista se stesso tramite zenoh per gestire in callback i successivi passaggi di stato per la nuova istanza configurata.

Al momento del lancio dell'applicazione tramite API "start", viene effettuato il parsing delle variabili d'ambiente e generato uno script eseguibile bash se ambiente Linux o powershell se ambiente Windows; lo script è quindi eseguito in un sottoprocesso nativo tramite libreria psutil e il pid salvato in un apposito file.

Nel caso l'applicazione venga terminata tramite chiamata alla API "stop", si estraе il pid del processo in esecuzione dal file creato precedentemente e si effettuano chiamate dirette a pkill per tentare di terminare il processo; in caso di fallimento, si delega al plugin-os la terminazione del processo.

Il plugin non supporta le API per passare agli stati "PAUSED", "MIGRATION" e "SCALING": nel caso venissero invocate tali API si avrebbe un messaggio di errore, il lancio di un'eccezione e il passaggio dell'istanza di FDU in uno stato di errore.

Utilizzare questo plugin permette quindi di eseguire applicazioni di cui era stato precedentemente fatto il deploy su nodi remoti tramite qualunque altro nodo che abbia installate le FIM API e che possa effettuare una connessione con il broker del sistema.

Questo approccio permette di interagire con nodi remoti in modo indiretto tramite zenoh, cioè senza necessità di avere disponibile un metodo di comunicazione diretto tra i due nodi (ad esempio ssh).

Plugin-fdu-containererd

Questo plugin fornisce il supporto all'esecuzione di applicazioni containerizzate seguendo lo standard OCI nel nodo su cui è installato.

Il descrittore di una FDU per un'applicazione containerizzata contiene i seguenti campi (quelli marcati con * sono facoltativi):

```
"image": {  
    "uri": "uri/to/container/image",  
    "checksum": "image_checksum",  
    "format": ""  
    "uuid": "" *  
    "name": "" *  
}
```

Inoltre il campo "hypervisor" deve contenere la keyword "DOCKER".

Containererd

Il plugin è realizzato in linguaggio Go sfruttando come supporto alla containerizzazione containererd, un container runtime con uno scope ben definito che include:

- esecuzione di container tramite primitive tra cui create, start,stop, pause, resume, delete;
- supporto a filesystem di tipo copy-on-write tra cui overlayFS e auFS;
- supporto a gestione delle immagini con API dedicate per effettuare ad esempio operazioni di push e pull;
- monitoraggio a livello di container.

Sono fuori dallo scope di containererd e quindi da implementare a livelli superiori le seguenti funzionalità:

- networking, cioè creazione e gestione di interfacce di rete;
- building, cioè creazione di nuove immagini di container;
- volumi per la gestione di file esterni al filesystem dei container;
- logging, containererd passa lo STDIO al client che può gestirlo in autonomia.

L'insieme di queste funzionalità implica che immagini costruite attraverso altri strumenti costruiti on-top-of containererd non si comportino nello stesso modo se eseguiti tramite tali strumenti o tramite containererd.

Per questo motivo, un'immagine costruita tramite Docker che esponga porte e faccia uso di volumi e venga eseguita tramite containererd non sarà accessibile tramite tali porte e non utilizzerà il volume indicato, portando a comportamenti inaspettati.

Ad esempio, eseguire un'istanza di NGINX (uri "docker.io/library/nginx:latest") tramite Docker permette l'accesso ad un server web alla porta specificata mentre la stessa esecuzione tramite containererd non ne permette la visualizzazione in quanto non vi è supporto al networking.

Filesystem

Il filesystem utilizzato in containerd da fog05 è overlayFS: si tratta di un union mount filesystem che genera un nuovo sottoalbero per ogni layer definito all'atto della creazione dell'immagine del container: ad esempio, un'immagine creata tramite Dockerfile genera un [layer](#) per ciascun comando RUN, COPY, ADD presente.

I sottoalberi sono posizionati nel filesystem dell'host nel path `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/` e containerd genera per ciascun layer una directory numerata progressivamente.

La generazione avviene al momento del pull dell'immagine del container mentre all'atto dell'esecuzione viene creato un layer aggiuntivo in cui collocare eventuali nuovi file.

Nel caso una delle directory dovesse già esistere poichè generata tramite altri strumenti, nel momento dell'esecuzione si avrebbe un fallimento .

Siccome overlayFS è union mount, al momento dell'esecuzione dell'istanza il filesystem unificato è generato unendo i vari layer che costituiscono l'immagine e montato nel path `/run/containerd/io.containerd.runtime.v2.task/fos/<instance_UUID>/`.

File aggiunti successivamente nel filesystem a layer risultano inesistenti all'interno del container: per ottenere questo effetto, è necessario che questi file vengano creati all'interno del filesystem unificato.

Questi filesystem sono di proprietà dell'utente root: qualunque accesso, sia in lettura che in scrittura, richiede privilegi elevati.

Implementazione API

Oltre ad un'analisi del codice sorgente del plugin, il comportamento dei componenti è stato verificato con i seguenti strumenti:

- demone containerd: `# ctr -n fos event`
- plugin-fdu-containerd: `$ journalctl -f -u fos_ctd`
- containerd.service: `$ journalctl -f -u containerd.service`

define: se l'immagine specificata nel descriptor è un file locale (prefisso "file://") allora questa è importata nel catalogo di containerd; se l'URI si riferisce invece a un'immagine remota allora se ne effettua il pull dal registry indicato.

configure: vengono aggiunti i ConnectionPoint e le interfacce di rete indicati nel descriptor e i metodi di callback per le successive operazioni.

start: viene generato il layer d'esecuzione, si effettua il parsing delle variabili d'ambiente per poi generare il container vero e proprio ed eseguirlo.

pause/resume: rispettivamente, dovrebbero mettere in pausa e riprendere l'esecuzione del container; attualmente ciò non avviene.

migrate: non presente nella corrente implementazione del plugin.

stop: il container viene fermato, il filesystem unificato è deallocated e il layer d'esecuzione eliminato.

Supporto alla migrazione

Uno degli obiettivi del lavoro era verificare il supporto fornito da fog05 alla migrazione di container Docker. Siccome nella versione attuale del plugin questa funzionalità è assente, la scelta è stata quella di fornire tale funzionalità ponendo alcuni vincoli sulla tipologia di container: si è convenuto di implementare la migrazione per i container che persistano il proprio stato su filesystem.

Come scritto prima, la root del filesystem unificato è montata al path `/run/containerd/io.containerd.runtime.v2.task/fos/<instance_UUID>/rootfs`.

Dato il vincolo introdotto, lo stato è contenuto interamente in questa directory: pertanto, effettuare la migrazione di un container con questa caratteristica significa trasportare questa directory e il suo contenuto sul nodo destinazione.

Utilizzando un demone per la sola copia dei file in modo da limitare il codice eseguito con privilegi elevati, necessari per accedere ai filesystem unificati, una possibile implementazione è ottenuta con i successivi passi:

1. tramite FIM API, mettere in pausa il container da migrare, di cui si deve conoscere lo UUID (d'ora in poi `_uuid_old`), e avviare il container sul nodo destinazione per ottenerne il nuovo UUID (d'ora in poi `_uuid_new`) e generare il filesystem remoto;
2. creare sul nodo di partenza un file con nome `_uuid_old` contenente la destinazione e `_uuid_new`, per segnalare al demone il container da migrare: il compito del demone è trasferire il filesystem del container riducendo possibili overhead dovuti a elevato numero di file e/o dimensione, mantenendo gli attributi presenti; tramite rsync è possibile ottenere questo comportamento specificando le opzioni a (archivio, preserva attributi e sottocartelle), E (mantiene eseguibilità dei file), A (mantiene ACL), X (mantiene xattr), z (comprime i file trasferiti); la destinazione remota deve essere espressa nel formato [USER@]HOST; è inoltre necessario che i due nodi possano collegarsi tramite ssh in modo automatico (nel codice riportato si suppone che il nodo di partenza possa effettuare login passwordless come root sul nodo destinazione utilizzando la chiave privata specificata);
3. terminare container di partenza.

Il demone è implementato con il seguente codice bash:

```
#!/bin/bash
CTDFS='/run/containerd/io.containerd.runtime.v2.task/fos/'
inotifywait -m /path/fos/new -e create -e moved_to | 
while read _dir _action _uuid_old; do
    { IFS= read -r _uuid_new && IFS= read -r _dest; } <
'/path/fos/new/"$_uuid_old"
    rsync -e "ssh -i /path/.ssh/root_key -o
StrictHostKeyChecking=no" -aEAXz "$CTDFS""$_uuid_old"/rootfs/
"$_dest": "$CTDFS""$_uuid_new"/rootfs'
        unset -v _uuid_new _dest
done
```

Nel caso non fosse possibile utilizzare rsync, il secondo passo si suddivide in:

- creazione archivio

```
# tar -cpzf '/path/fos/tar/'"${_uuid_new}" -C  
'/run/containerd/io.containerd.runtime.v2.task/fos/'"${_uuid_old}"  
'rootfs'
```
- copia su nodo remoto dell'archivio;
- decompressione archivio ricevuto

```
# tar -xf '/path/to/received/archive/'"${_uuid_new}"  
--directory='/run/containerd/io.containerd.runtime.v2.task/fos/'"${_ui  
d_new}"'rootfs/'
```

I comandi tar devono essere eseguiti con privilegi elevati in quanto accedono a cartelle appartenenti all'utente root.

Utilizzando le FIM API Python, il codice per gestire le FDU è il seguente:

```
inst_info=api.fdu.define(fdu_uuid, start_node_uuid)  
api.fdu.configure(inst_info.get_uuid())  
api.fdu.start(inst_info.get_uuid(), env)  
time.sleep(20) # do some work  
  
# api.fdu.pause(inst_info.get_uuid()) # at the moment not implemented  
inst_info_remote=api.fdu.define(fdu_uuid, destination_node_uuid)  
api.fdu.configure(inst_info_remote.get_uuid())  
api.fdu.start(inst_info_remote.get_uuid(), env)  
  
# if old container is in local node notify daemon to move it  
with open('/path/fos/new/' + inst_info.get_uuid(), 'w') as uuid_old:  
    uuid_old.write(inst_info_remote.get_uuid() + "\n")  
    uuid_old.write(destination_address)
```

Si sfrutta fog05 per eseguire un'applicazione nativa sul nodo iniziale che genera correttamente il file leggendo dall'Environment i parametri necessari (vedere `fdu_native_mig.json` e `migrator_receiver.sh`).

```
# solution if old container is NOT in local node  
mig_desc = json.loads(read_file('fdu_native_mig.json'))  
mig_fdu_descriptor = FDU(mig_desc)  
fduD_mig = api.fdu.onboard(mig_fdu_descriptor)  
fdu_id_mig = fduD_mig.get_uuid()  
inst_info_mig = api.fdu.define(fdu_id_mig, inst_info.get_uuid())  
api.fdu.configure(inst_info_mig.get_uuid())  
api.fdu.start(inst_info_mig.get_uuid(),  
            "_new_uuid=" + inst_info_remote.get_uuid() +  
            ",_old_uuid=" + inst_info.get_uuid() +  
            ",_destination=" + destination_address)  
  
api.fdu.terminate(inst_info.get_uuid())  
time.sleep(20) # do some work on migrated container  
api.fdu.terminate(inst_info_remote.get_uuid())
```

Verifica della migrazione

Per verificare il corretto funzionamento della proposta presentata, è stata preparata un'immagine Docker basata su Ubuntu:20.04 con Python3 installato utilizzando il seguente Dockerfile:

```
FROM marcogozzi/upy:1
RUN mkdir -p /home/fos/config && mkdir /tmp/fos
COPY main-app-t.py /home/fos
COPY ./config /home/fos/config
VOLUME /tmp/fos
ENTRYPOINT ["python3"]
CMD ["/home/fos/main-app-t.py", "-c", "/home/fos/config/config.json"]
```

L'immagine esegue tramite Python3 l'applicazione main-app-t.py:

- All'avvio, se presente nell'Environment un parametro di nome "load_state", l'applicazione tenta di recuperare dal filesystem lo stato che deve essere presente in `/tmp/fos/state.json`; finché tale file non è stato trovato, l'applicazione effettua controlli a polling e non procede;
- una volta completato il caricamento dello stato, l'applicazione carica la configurazione specificata o nell'Environment o nel file di configurazione `/home/fos/config/config.json` oppure utilizza valori di default;
- a questo punto viene eseguita la business_logic utilizzando lo stato e la configurazione ottenuti precedentemente: per verificare la correttezza della migrazione, l'applicazione crea a intervalli regolari dei file contenenti il timestamp di creazione in modo da verificare che, una volta effettuata la migrazione sul nodo destinazione questi siano stati trasportati correttamente; inoltre, lo stato viene aggiornato ad ogni iterazione e salvato su filesystem, in modo che durante la migrazione, impostando il parametro "load_state" nell'Environment dell'istanza destinazione, venga trasferito l'ultimo aggiornamento dello stato.

Il codice usato per eseguire il test è `Migrazione/fos_dep_doc_mig-rel.py`.

I file di log prodotti sono in `Migrazione/log`.

I file utilizzati per generare l'immagine Docker sono in `Migrazione/Docker`.

Per replicare la prova, in `fos_dep_doc_mig-rel.py` modificare l'IP del broker zenoh, gli UUID dei nodi (ottenibile cat `/etc/machine-id`) e l'indirizzo IP del nodo destinazione, oltre ai vari path.

L'implementazione alternativa senza rsync è disponibile nei file `zipper.sh`, `mover.sh` e `receiver.sh` nella cartella `Migrazione`.

Progetto

La seconda parte del lavoro consiste nel progettare un sistema distribuito composto da sensori, attuatori e servizi di monitoraggio, con supporto a interazioni a polling e ad eventi; la gestione del sistema deve avvenire tramite fog05.

Analisi

Sensore:

- dispositivo fisico o virtuale di input che ricava informazioni riguardo un aspetto fisico dell'ambiente in cui è in esecuzione;
- è completamente passivo, cioè non deve prendere decisioni o controllare direttamente altri dispositivi;
- deve permettere di ottenere una nuova lettura quando necessario.

Supporto software per un sensore:

- mantenere l'ultimo valore letto dal sensore e l'unità di misura utilizzata;
- offrire un'interfaccia per effettuare una nuova lettura ed impostare una frequenza con cui effettuare le successive (e/o il ritardo).

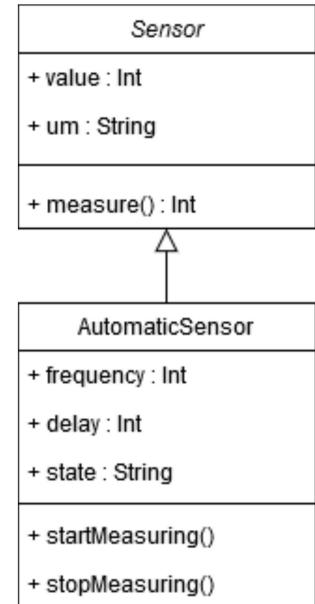


Figure 6: UML sensore

Attuatore:

- dispositivo fisico o virtuale che permette di eseguire azioni su di sé o su altri dispositivi, eventualmente producendo cambiamenti nell'ambiente in cui opera;
- è completamente passivo, cioè non deve prendere decisioni o controllare direttamente altri dispositivi;
- deve permettere di eseguire una azione quando necessario.

Supporto software per un attuatore:

- offrire un'interfaccia per eseguire l'azione dell'attuatore supportato; siccome le azioni hanno complessità variabile, il numero di attributi necessari è fortemente dipendente dal singolo attuatore.

Servizio di monitoraggio: poiché sensori ed attuatori sono elementi passivi, cioè eseguono solamente comandi che vengono loro impartiti, è opportuno che vi sia almeno un componente software che si occupi di monitorare l'output dei primi per azionare i secondi. Siccome nuovi sensori e nuovi attuatori possono essere introdotti durante il funzionamento del sistema, allora è opportuno che il servizio sia riprogrammabile oppure che possano coesistere più servizi contemporaneamente capaci di coordinarsi.

Supporto a polling: a causa delle risorse limitate, alcuni nodi potrebbero non eseguire ripetutamente letture dai sensori disponibili; inoltre, alcuni sensori potrebbero generare poco frequentemente nuovi dati mentre in un certo istante è necessaria una lettura recente; per questo motivi, è opportuno che sia presente un meccanismo che permetta di interrogare a polling qualunque elemento che possa fornire informazioni.

Supporto a eventi: alcuni nodi potrebbero avere risorse molto limitate oppure il costo di interazione con essi potrebbe essere elevato; per limitare la comunicazione è possibile limitare lo scambio di messaggi a solamente quelli che trasmettano novità di rilievo per un certo nodo/dispositivo.

Uso di fog05: poiché la gestione avviene tramite fog05 è possibile utilizzarne alcune caratteristiche come la generazione di UUID per distinguere nodi, eseguibili e istanze; queste informazioni possono essere trasmesse alle istanze per fornire loro maggior conoscenza sul sistema e per potersi identificare reciprocamente durante il processamento dei messaggi.

Ad esempio, se si volesse eseguire un'istanza A del software di supporto S su un nodo X e poi utilizzare un'istanza B di servizio di monitoraggio su un nodo Y, sarebbe sufficiente passare durante l'avvio di B passare gli UUID di X, di S e di A direttamente tramite Environment, senza bisogno di comunicazione preliminare tra i due.

Protocolli di comunicazione: data la natura distribuita su possibili nodi a risorse limitate, potrebbe essere necessario limitare i protocolli di comunicazione a quelli progettati per tale situazione;

- MQTT: protocollo pensato per IoT, fornisce supporto a comunicazione di tipo publish/subscribe basato su broker per dispositivi dotati di scarse risorse che operano in reti non affidabili, con poca banda e alta latenza; tra le features più importanti vi sono Quality of Service differenziabile, messaggi ritenuti per client non online (collegati precedentemente con connessione persistente a topic con QoS > 0), disaccoppiamento spaziale e asincronismo. Esiste una versione basata su UDP (MQTT-SN), la cui specifica non è stata evoluta negli ultimi anni.
- CoAP: protocollo leggero basato su modello REST, offre supporto a operazioni GET, PUT, POST, e DELETE su URL presso cui si sono registrate risorse; è stato progettato per funzionare su dispositivi a limitatissime risorse anche grazie a un header limitato a soli 4 byte, UDP on IP e un encoding compatto per le opzioni; fornisce supporto per la sicurezza.

Entrambi i protocolli sono standardizzati, il primo da OASIS e il secondo tramite RFC.

Sono disponibili implementazioni per i principali linguaggi tra cui Python, JavaScript, C, C++, C#, Java e Kotlin.

Descrizione del sistema demo

Con riferimento alla fig. 7, sono presenti tre nodi in cui tre sensori e tre attuatori eseguono in condizioni normali, con il flusso di informazioni da gestirsi come in fig. 8: in particolare un sistema di monitoraggio deve essere programmabile all'avvio e, in base alla lettura di ciascun sensore, comandare uno specifico attuatore.

Se si verifica una condizione anomala, ossia i sensori A e B segnalano

contemporaneamente un valore oltre a una soglia limite, allora il servizio che se ne occupa deve avviare una verifica su tali sensori eseguendo un certo software di controllo prestabilito sui nodi in cui sono presenti tali sensori.

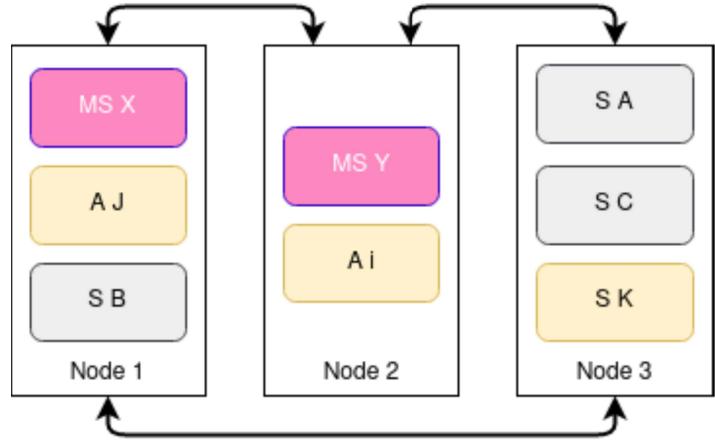


Figure 7: Deploy sistema

Flusso di messaggi: per quanto stabilito finora, la comunicazione tra sensori e servizi di monitoraggio è bidirezionale poiché in base ai dati letti dai sensori e ricevuti dai servizi potrebbe derivare un cambiamento nella configurazione dei primi; viceversa, gli attuatori sono solo esecutori di comandi per cui possono ricevere entrambi ma non hanno necessità di comunicare.

Non tutti i sensori comunicano con tutti i servizi di monitoraggio e non tutti i servizi hanno necessità di comunicare con tutti gli attuatori.

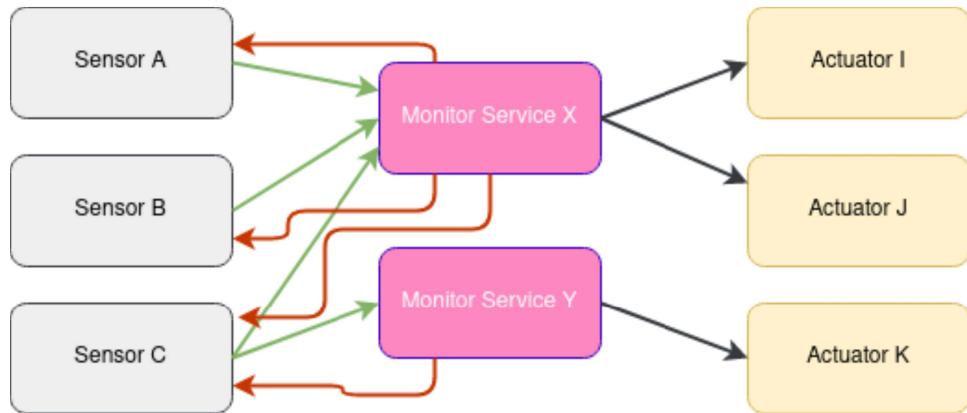


Figure 8: Messaggi contenenti *dati*, *comandi* o *entrambi*

Il sistema è distribuito su più nodi, di cui si conoscono a tempo di sviluppo i dispositivi presenti (sensori e attuatori) mentre il deploy dei servizi di monitoraggio può variare essendo componenti puramente software.

Ciascun servizio è responsabile di una lista di sensori da cui ricevere informazioni e di una di attuatori da comandare.

È possibile aggiungere nuovi servizi durante l'esecuzione del sistema, come nel caso dovessero rendersi necessarie altre gestioni di dati di dispositivi già operanti.

È compito della logica di business dei servizi assicurarsi di non comandare in modo equivoco uno stesso attuatore, ad esempio inviando contemporaneamente due comandi opposti: questo può essere fatto staticamente assegnando ogni attuatore ad un solo servizio oppure dinamicamente coordinando i vari servizi.

Nel caso trattato, sarà compito del servizio X gestire la condizione anomala riportata precedentemente.

Progettazione

Sensore

La classe Sensor rappresenta un generico sensore, di cui value è l'ultima lettura e um l'unità di misura; il metodo measure effettua una misurazione, salva il valore in value e lo ritorna.

AutomaticSensor estende Sensor introducendo il concetto di letture automatiche ripetute (dis-)attivabili; delay indica il ritardo tra due letture successive.

MockSensor estende a sua volta AutomaticSensor introducendo un valore soglia variabile, mantenuto in threshold, oltre il quale il sensore emette una allerta agli oggetti MockSub registrati presso di esso, a cui passa il nuovo valore invocando il metodo alert.

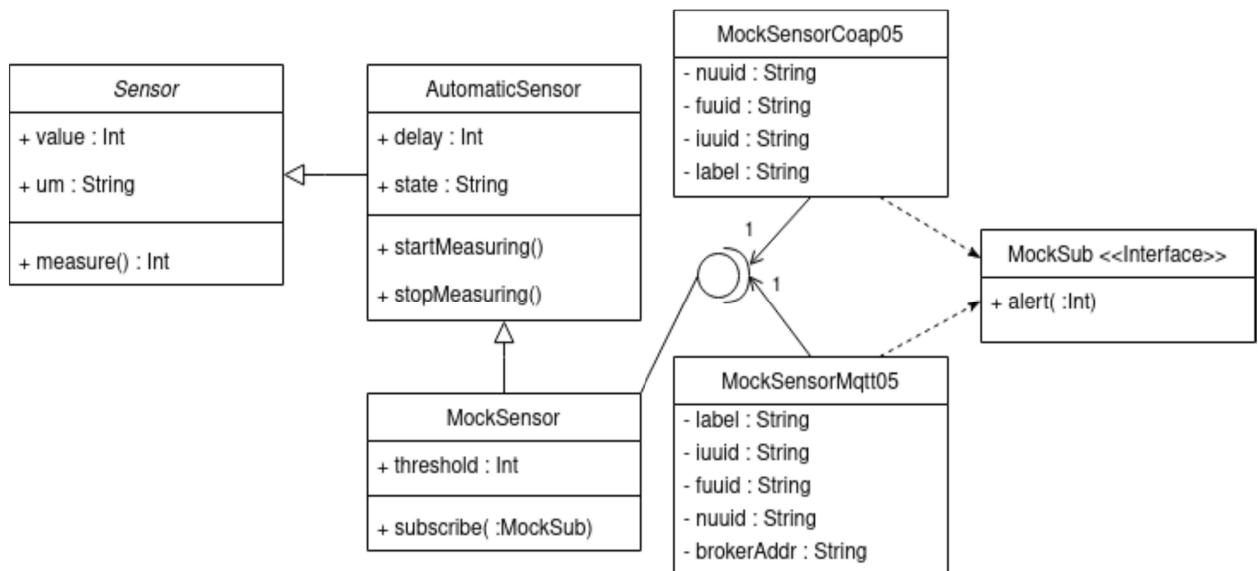


Figure 9: UML sensore

MockSensorCoap05 e **MockSensorMqtt05** agiscono da proxy per MockSensor verso l'esterno implementando rispettivamente i protocolli CoAP e MQTT; sono inoltre fog05-aware, per cui utilizzeranno gli UUID forniti per la comunicazione.

MockSensorCoap05: risorsa CoAP osservabile, espone gli attributi di MockSensor agli URL `coap://<host>:<port>/<iuuuid>` e `coap://<host>:<port>/<label>`.

```
root = resource.Site()
root.add_resource([self.iuuuid], self)
root.add_resource([self.label], self)
asyncio.Task(aiocoap.Context.create_server_context(root,
                                                    bind=('0:0:0:0:ffff:7f00:1',
                                                          self.port)))
self.loop = asyncio.get_event_loop()
asyncio.get_event_loop().run_forever()
```

A richieste GET risponde con una rappresentazione JSON codificata in UTF-8 dello stato del sensore;

```
async def render_get(self, request):
    return aiocoap.Message(payload=self.encode_state())
```

Se la richiesta GET include l'opzione Observe, allora ogni cambiamento dello stato provocherà l'invio di una notifica al client che ha fatto richiesta, invocando il metodo:

```
    self.updated_state()
```

A richieste PUT aggiorna i campi specificati, eventualmente utilizzando i metodi esistenti per state, cioè effettuando una chiamata a startMeasuring() o stopMeasuring() e restituisce la nuova rappresentazione.

```
async def render_put(self, request):
    msg_json = fos_utils.decode_state(request.payload)
    self.sensor.threshold = msg_json.get("threshold", self.sensor.threshold)
    self.sensor.delay = msg_json.get("delay", self.sensor.delay)
    self.sensor.state = msg_json.get("state", self.sensor.state)
    return aiocoap.Message(payload=self.encode_state())
```

MockSensorMqtt05: effettua una connessione al broker specificato da brokerAddr tramite cui comunicare con i servizi di monitoraggio.

Ogni istanza si sottoscrive alle topic <nuuid>/<fuuid>/<iuuuid>/in e <nuuid>/<fuuid>/<label>/in; in questo modo, un servizio che voglia comunicare con essa manderà un messaggio su tale topic specificando l'operazione desiderata ed eventuali parametri necessari (ad esempio GET o PUT delay 50); come nell'implementazione CoAP, ogni cambiamento dello stato dell'istanza causerà l'invio di un messaggio contenente la rappresentazione aggiornata dello stato stesso sulle topic <nuuid>/<fuuid>/<iuuuid> e <nuuid>/<fuuid>/<label>.

Nel costruttore: `def __init__(self, mock_sensor: MockSensor):`

```
# sensor init
self.sensor = mock_sensor
self.sensor.subscribe(self)
self.mqttc = mqtt.Client()

# Mqtt Specific
self.mqttc.message_callback_add(self.topic_uuid + "/in", self.on_message_in)
self.mqttc.message_callback_add(self.topic_label + "/in",
self.on_message_in)
self.mqttc.connect(self.broker)
self.mqttc.subscribe(self.topic_uuid + "/in", 0)
self.mqttc.subscribe(self.topic_label + "/in", 0)
self.mqttc.loop_forever()
```

Metodo di callback:

```
def on_message_in(self, client, userdata, message):
    msg_json = fos_utils.decode_state(message.payload)
    # dict.get(key, default) returns value associated to key if exists,
    # default if not
    self.sensor.threshold = msg_json.get("threshold", self.sensor.threshold)
    self.sensor.delay = msg_json.get("delay", self.sensor.delay)
    self.sensor.state = msg_json.get("state", self.sensor.state)
    self.advert()

def advert(self):
    self.mqttc.publish(self.topic_uuid, payload=self.encode_state())
    self.mqttc.publish(self.topic_label, payload=self.encode_state())
```

Attuatore

La classe Actuator rappresenta un generico attuatore, che espone un unico metodo execute che attiva l'attuatore per completare l'azione.

TimedActuator estende Actuator introducendo il concetto di azione prolungata nel tempo; executionTime indica per quanti secondi l'attuatore effettua la propria azione; è possibile ignorare tale attributo invocando execute con argomento.

MockTimedActuator estende a sua volta TimedActuator implementando il metodo execute con una azione simulata (ad esempio stampa a video o scrittura su un file); ad ogni cambiamento di stato, notifica gli oggetti che hanno effettuato subscribe.

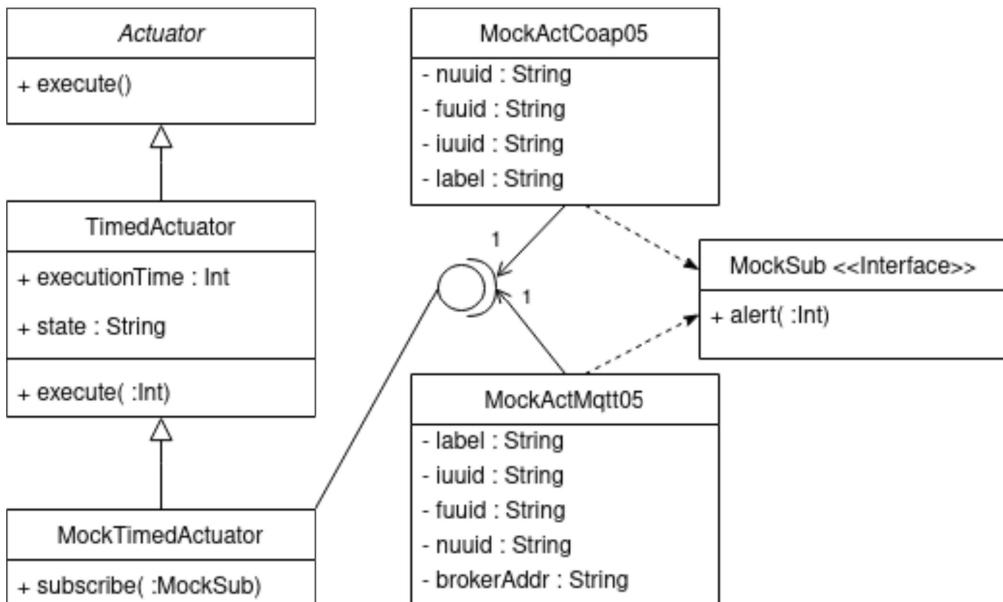


Figure 10: UML attuatore

MockActCoap05 e **MockActMqtt05** agiscono da proxy per MockTimedActuator verso l'esterno implementando rispettivamente i protocolli CoAP e MQTT; sono inoltre fog05-aware, per cui utilizzeranno gli UUID forniti per la comunicazione.

MockActCoap05: risorsa CoAP osservabile, espone gli attributi di MockTimedActuator agli URL `coap://<host>:<port>/<iuuid>` e `coap://<host>:<port>/<label>`.

```
root = resource.Site()
root.add_resource([self.iuuuid], self)
root.add_resource([self.label], self)
asyncio.Task(aiocoap.Context.create_server_context(root,
                                                     bind=('0:0:0:0:0:ffff:7f00:1',
                                                           self.port)))
self.loop = asyncio.get_event_loop()
asyncio.get_event_loop().run_forever()
```

A richieste GET risponde con una rappresentazione JSON codificata in UTF-8; se la richiesta GET include l'opzione Observe, allora ogni cambiamento dello stato provocherà l'invio di una notifica al client che ha fatto richiesta.

```
async def render_get(self, request):
    return aiocoap.Message(payload=self.encode_state())
```

A richieste PUT aggiorna i campi specificati; se state diventa execute, allora viene effettuata una chiamata al relativo metodo; restituisce la nuova rappresentazione.

```
async def render_put(self, request):
    msg_json = fos_utils.decode_state(request.payload)
    self.actuator.execution_time = msg_json.get("execution_time",
                                                self.actuator.execution_time)
    self.actuator.state = msg_json.get("state", self.actuator.state)
    return aiocoap.Message(payload=self.encode_state())
```

MockActMqtt05: effettua una connessione al broker specificato da brokerAddr tramite cui comunicare con i servizi di monitoraggio. Adotta la stessa procedura per la comunicazione di MockSensorMqtt05 (sottoscrizione e pubblicazione sulla stessa struttura di topic, per input `<nuuid>/<fuuid>/<iuuuid>/in` e `<nuuid>/<fuuid>/<label>/in` mentre per output `<nuuid>/<fuuid>/<iuuuid>` e `<nuuid>/<fuuid>/<label>`).

Servizio di monitoraggio

Contiene la logica di business che utilizza le letture dei sensori per decidere quando e come attivare gli attuatori messi a disposizione.

Un servizio viene lanciato per gestire istanze note di sensori e attuatori, le cui label sono specificate in un file di configurazione, insieme alla politica da seguire.

Nel caso proposto, per ogni sensore specificato che emetta un allarme, il servizio deve attivare certi attuatori tra quelli a lui noti con parametri prestabiliti: questi sono inseriti in file di configurazione come ad esempio "service_x.json": ciascuna entry prevede una chiave che rappresenta la label del sensore di provenienza; il valore associato è una lista di dizionari, ciascuno contenente la label dell'attuatore da attivare e i parametri necessari

```
"mappings": {
```

```
    "A": [{"label": "J", "state": "execute", "execution_time": 10}] }
```

La comunicazione tra servizio e dispositivi avviene tramite più proxy, uno per ciascun dispositivo; ogni proxy implementa l'interfaccia DeviceCommunicator che espone i metodi i metodi:

- send(action, payload) per inviare un messaggio al corrispettivo dispositivo specificando l'azione da eseguire con una verbo HTTP e passando eventuali parametri tramite oggetto JSON codificato come string in UTF-8;
- stop() per interrompere la comunicazione chiudendo appropriatamente i canali.

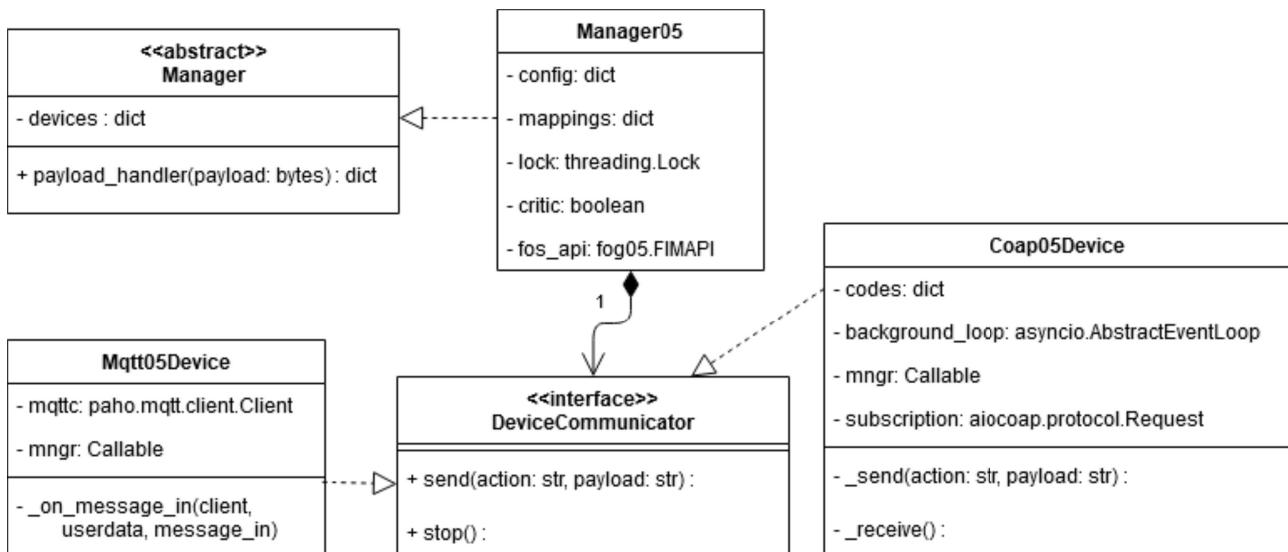


Figure 11: UML Servizi di monitoraggio

L'istanziazione dei proxy è realizzata da un metodo factory per nascondere la varie implementazioni.

```
def get_connector(config, handler):
    # (self, nuuid, fuuid, iuuuid, broker, payload_handler)
    if config["connector"]["name"] == "mqtt":
        return Mqtt05Device(handler, config["nuuid"], config["fuuid"],
                            config["iuuuid"], config["label"],
                            **config["connector"])
    # (self, ip, port, label, payload_handler)
    elif config["connector"]["name"] == "coap":
        return Coap05Device(handler, config["nuuid"], config["fuuid"],
                            config["iuuuid"], config["label"],
                            **config["connector"])
```

Per generare un proxy è necessario passare alla factory due argomenti:

- la configurazione del proxy, contenuta nel file di configurazione nel campo "devices"

```
"devices": {
    "A": {"nuuid": "", "label": "A", "desc": "fdु_sens_mqtt.json",
           "connector": {"name": "mqtt", "broker": "localhost"}},
    "B": {"nuuid": "", "label": "B", "desc": "fdु_sens_coap.json",
           "connector": {"name": "coap", "ip": "localhost", "port": "9500"}}
}
```

- una callback che il proxy invocherà alla ricezione di ciascun messaggio, passandone il payload; in particolare sarà passato il metodo payload_handler del servizio.

I proxy implementati sono due:

Mqtt05Device: proxy per dispositivo connesso tramite Mqtt e avviato tramite fog05; sfrutta la libreria **paho.mqtt** per gestire il protocollo Mqtt;

Effettua la connessione al broker specificato dall'argomento "broker" e si sottoscrive alla topic costruita come specificato nei paragrafi riguardanti sensore e attuatore (<nuuid>/<fuuid>/<iuuuid> oppure <nuuid>/<fuuid>/<label>), impostando come callback il metodo passando tramite l'argomento "payload_handler".

Nel costruttore:

```
def __init__(self, payload_handler, nuuid, fuuid, iuuuid, label, name, broker):
    self.mngr = payload_handler
    self.mqttc = mqtt.Client()
    self.label = label
    self.topic = nuuid + "/" + fuuid + "/" + iuuuid if iuuuid != "" else nuuid
+ "/" + fuuid + "/" + label
    self.mqttc.connect(broker)
    self.mqttc.message_callback_add(self.topic, self._on_message_in)
    self.mqttc.subscribe(self.topic, 0)
    self.mqttc.loop_start()
```

La send dell'interfaccia DeviceCommunicator prevede l'inserimento dell'azione nel payload e pubblica sulla topic corretta il messaggio.

```
def send(self, action, payload):
    msg_json = fos_utils.decode_state(payload)
    if "action" not in msg_json:
        msg_json["action"] = action
    self.mqttc.publish(self.topic + "/in",
                      payload=fos_utils.encode_state(msg_json))
```

Utilizzando il metodo Client.loop_start(), la libreria genera automaticamente un thread in background che si mette in attesa di nuovi messaggi, liberando il thread da cui è invocato il metodo; per terminare la comunicazione è presente il metodo Client.loop_stop() invocato nell'implementazione del metodo DeviceCommunicator.stop().

Coap05Device: proxy per dispositivo connesso tramite CoAP e avviato tramite fog05; sfrutta la libreria `aiocoap` per gestire il protocollo CoAP.

Ogni proxy utilizza due `asyncio` event loop che possono essere usati per eseguire task asincroni e callback, effettuare operazioni di I/O su rete e lanciare sottoprocessi.

Un event loop è utilizzato per la sottoscizione alla risorsa CoAP per ricevere gli aggiornamenti dello stato di tale risorsa mentre il secondo loop ("self.background_loop") è eseguito in un thread in background ed è utilizzato per effettuare le operazioni di send.

Nel costruttore:

```
asyncio.get_event_loop().create_task(self._receive())
self.background_loop = asyncio.new_event_loop()
self.t = threading.Thread(target=start_background_loop,
                           args=(self.background_loop,), daemon=True)
self.t.start()
```

Callback di ricezione messaggi:

```
async def _receive(self):
    ctx = await Context.create_client_context()
    request = Message(code=codes.Code.GET, uri=self.uri, observe=0)
    self.subscription = ctx.request(request)
    self.subscription.observation.register_callback(self.msg_callback)
    async for msg in self.subscription.observation:
        # print("Coap05Device _receive " + str(msg.payload))
        pass
```

Operazioni di send:

```
def send(self, action, payload):
    asyncio.run_coroutine_threadsafe(self._send(action, payload),
                                    self.background_loop)

async def _send(self, action, payload):
    context = await Context.create_client_context()
    request = Message(code=self.codes[action], payload=payload,
                      uri=self.uri)
    response = await context.request(request).response
    self.mngr(response.payload)
```

Terminare il proxy comporta la cancellazione della sottoscizione e la chiusura dell'event loop in background

```
def stop(self):
    self.subscription.observation.cancel() # stop CoAP subscription
    self.background_loop.stop()
```

Manager

Classe base che rappresenta un servizio di monitoraggio; implementa il metodo payload_handler inserendo il payload decodificato del messaggio arrivato all'interno della variabile dizionario "devices", nel campo "state"; il metodo ritorna un oggetto JSON che rappresenta lo stato ricevuto.

Manager05

Estensione di Manager per gestire la comunicazione tramite istanze di DeviceConnector, con deploy effettuato tramite fog05.

Tramite variabile d'ambiente di nome "config" ottiene il nome del proprio file di configurazione in formato json mentre nella variabile d'ambiente "system" può essere passata la configurazione dell'istanza di sistema su cui agire (i.e. gli URI con cui comunicare costruire i DeviceConnector e relativi parametri).

Le configurazioni sono mantenute nelle variabili omonime "self.config" e "self.mappings"; per ogni dispositivo specificato nei mappings, si ottiene un'istanza di connettore tramite factory e la si conserva nel campo "connector" della entry relativa in "devices".

```
devices = set(self.mappings.keys())
for label in self.mappings.keys():
    for mapping in self.mappings[label]:
        devices.add(mapping["label"])
for device in devices:
    self.devices[device] = {}
    self.devices[device]["connector"] =
        get_connector(self.config["devices"][device],
                      self.payload_handler)
```

Se nella configurazione sono presenti entrambi i sensori "A" e "B" allora viene generata una istanza di FIM API ed eventualmente caricato in fog05 il descrittore del software di controllo, se non ne è già stato fornito il FDU UUID.

```
self.critic = False
if "A" in self.mappings.keys() and "B" in self.mappings.keys():
    from fog05 import FIMAPI
    from fog05_sdk.interfaces.FDU import FDU
    self.critic = True
    self.critic_nodes = {self.config["devices"]["A"]["nuuid"],
                         self.config["devices"]["B"]["nuuid"]}
    self.fos_api = FIMAPI(self.config.get("fimapi", "192.168.31.232"))
    if "json" in self.config["fdt_control"]:
        # if descriptor is a filename, load it in fog05
        fdu_descriptor = FDU(json.loads(
            fos_utils.read_file(self.config["fdt_control"])))
        fduD = self.fos_api.fdu.onboard(fdu_descriptor)
        self.config["fdt_control"] = fduD.get_uuid()
```

Successivamente, il servizio avvia la lettura per tutti i sensori che controlla.

```
for label in self.devices.keys():
    if "sens" in self.config["devices"][label]["desc"]:
        payload = {"action": "PUT", "state": "on"}
        self.devices[label]["connector"].send("PUT",
            fos_utils.encode_state(payload))
```

Ogni messaggio in arrivo tramite DeviceConnector causa l'esecuzione del metodo payload_handler che modifica lo stato del servizio: poiché i proxy possono eseguire su thread paralleli è opportuno un accesso alla risorsa in mutua esclusione, ad esempio utilizzando un oggetto threading.Lock.

```
content = {}
with self.lock:
    content = super().payload_handler(payload)
    if self.critic:
        self.devices[content["label"]]["last_update"] = int(time.time())
```

Una volta ottenuto accesso ed aggiornato lo stato dei dispositivi, per ogni attuatore specificato nei "mappings" di configurazione si invia un messaggio con i relativi parametri

```
mappings = self.mappings[content["label"]]
for mapping in mappings:
    self.devices[mapping["label"]]["connector"].send("PUT",
        fos_utils.encode_state(mapping))
```

Se il servizio si occupa dei sensori "A" e "B" allora si aggiorna il timestamp di ricezione del messaggio ed eventualmente si esegue il software di controllo sui nodi corretti.

```
if self.critic and (content["label"] == "A" or content["label"] == B):
    time_delta = self.devices["A"].get("last_update", 100) -
        self.devices["A"].get("last_update", 0)
    if -5 < time_delta < 5:
        for node in self.critic_nodes:
            self.fos_api.fdu.instantiate(self.config["fdi_control"],
                node, wait=False)
    # reset timestamps so that two more alarms
    # have to arrive to trigger another control
    self.devices["A"]["last_update"] = 100
    self.devices["B"]["last_update"] = 0
```

Ottimizzazioni topic MQTT

Siccome le topic hanno come radice `nuuid/fuuuid/iuuuid/` è possibile che alcune operazioni pub/sub siano gestite efficientemente dal broker poiché i messaggi destinati ad una certa istanza su di uno specifico nodo verranno inviati solamente a tale nodo, in quanto solo da esso sono state effettuate sottoscrizioni che iniziano con quello specifico identificatore.

Questa ottimizzazione non si può applicare quando il messaggio è inviato dall'istanza perché possono esistere più nodi destinatari se molteplici servizi sono sottoscritti alla particolare istanza.

Prendendo come riferimento la fig. 9, una publish effettuata da X verrà propagata solamente al nodo su cui esegue l'istanza specificata; al contrario, una publish effettuata da C potrebbe essere propagata su più nodi nel caso i due sistemi di monitoraggio siano entrambi in esecuzione.

Inoltre, adottare tale radice permette di utilizzare wildcard per sottoscriversi a più topic contemporaneamente: ad esempio sottoscrivendosi alla topic `+/<fuuuid>/+` si ricevono tutti gli aggiornamenti di stato di tutte le istanze di tale FDU, qualunque sia il nodo su cui sono in esecuzione, mentre sulla topic `<nuuid>/+/+` verranno pubblicati tutti gli aggiornamenti di stato di qualunque dispositivo.

Deployment

Come detto, il deploy del sistema avviene tramite fog05.

Ogni combinazione dispositivo-tecnologia è eseguibile utilizzando un apposito descrittore che crea una istanza di dispositivo e la passa ad una nuova istanza del relativo connettore per la specifica tecnologia; al momento quindi i descrittori sono quattro ("fdu_act_coap.json", "fdu_act_mqtt.json", "fdu_sens_coap.json" e "fdu_sens_mqtt.json").

Dato un file di configurazione del sistema `system.json`, il deploy avviene con il seguente codice:

```
descs = json.loads(read_file("/var/fos/demo/deploy/system.json"))
for label in descs["devices"]:

    # onboard descriptor given in config file if not given an fuuid
    if "desc" in descs["devices"][label]:
        desc = json.loads(read_file("/var/fos/demo/deploy/" +
                                     descs["devices"][label]["desc"]))
        fdu_descriptor = FDU(desc)
        fduD = api.fdu.onboard(fdu_descriptor)
        fdu_id = fduD.get_uuid()
    elif "fuuid" in descs["devices"][label]:
        fdu_id = descs["devices"][label]["fuuid"]
    else:
        continue # no FDU descriptor available, skip this device

    inst_info = api.fdu.define(fdu_id, descs["devices"][label]["nuuid"])
    descs["devices"][label]["nuuid"] = uuid_portatile
    descs["devices"][label]["fuuid"] = fdu_id
    descs["devices"][label]["iuuid"] = inst_info.get_uuid()

    time.sleep(1)
    api.fdu.configure(inst_info.get_uuid())

    env = "nuuid=" + descs["devices"][label]["nuuid"] + \
          ",fuuid=" + fdu_id + \
          ",iuuid=" + inst_info.get_uuid() + \
          ",label=" + label + \
          ",port=" + descs["devices"][label]["connector"].get("port",
                    str(9500)) + \
          ",ip=" + descs["devices"][label]["connector"].get("ip",
                    str(9500)) + \
          ",broker=" + descs["devices"][label]["connector"].get("broker",
                    str(9500))

    time.sleep(1)
    api.fdu.start(inst_info.get_uuid(), env)
```

L'uso delle `sleep` serve per evitare che una chiamata ad una API avvenga prima che il demone zenoh abbia completato la chiamata precedente.

Un servizio di monitoraggio è avviabile utilizzando il descrittore "fd_manager.json" che avvia un'istanza di Manager05; nello stesso path dell'eseguibile (indicato nel campo command.args) deve essere presente un file di configurazione contenente i mappings.

Il nome di tale file deve essere passato tramite Environment con la chiave "config", insieme alla descrizione dell'istanza del sistema tramite chiave "system".

Poichè il plugin per il supporto ad applicazioni native effettua separazione delle variabili d'ambiente con il carattere "," allora questo è sostituito con "^".

Inoltre, la presenza di doppi apici risulta creare problemi con le variabili d'ambiente, quindi sono sostituiti con il simbolo "%".

```
desc = json.loads(read_file("/var/fos/demo/deploy/fd_manager.json"))
fd_manager_descriptor = FDU(desc)
fd_managerD = api.fdu.onboard(fd_manager_descriptor)
fd_manager_id = fd_managerD.get_uuid()

# replace item separator "," with "^" and double quotes with "%"
# to avoid string splitting during env parsing operation
sys_desc = json.dumps(descs, separators=("^", ":")) .replace("\\"", "%")

for manager in ["service_x", "service_y"]:
    inst_info = api.fdu.define(fd_manager_id, node_id)

    time.sleep(1)
    api.fdu.configure(inst_info.get_uuid())
    env = "nuuid=" + uuid_portatile + \
          ",fuuid=" + fd_manager_id + \
          ",iuuid=" + inst_info.get_uuid() + \
          ",label=" + manager + \
          ",config=" + "/var/fos/demo/" + manager + ".json" + \
          ",system=" + sys_desc

    time.sleep(1)
    api.fdu.start(inst_info.get_uuid(), env)
    time.sleep(1)
```

Il software di controllo dei sensori è descritto nel file "fd_control_software.json" ed in particolare avvia un'applicazione bash che notifica la propria avvenuta esecuzione scrivendo su standard output (rediretto su file di log di fo05, al path "/var/fos/logs/") e sul file "control_software.log" nel path in cui esegue.

Test

L'ambiente di test è costituito da 3 cloni VM su cui sono in esecuzione i soli componenti di Fog05. Le immagini di partenza sono Ubuntu 20.04 ed eseguono con risorse minime (1 CPU core e 500 MB di RAM). Hanno a disposizione una rete virtuale dedicata collegata anche ad internet per effettuare il pull delle immagini dei container.

Strumenti

psrecord

Utility che sfrutta la libreria python [psutil](#) per monitorare il consumo di CPU e memoria di un processo; permette il logging dei valori registrati ed eventualmente una rappresentazione finale in forma grafica.

Nethogs

Strumento di networking per monitorare il traffico di rete in tempo reale.

Iptraf

Strumento di networking che fornisce statistiche sull'utilizzo della rete quali banda media, massima e totale utilizzata (in entrata e/o in uscita).

wireshark

Strumenti per l'analisi dei protocolli di rete: permette l'ispezione di tutti i pacchetti in transito sulle interfacce di rete monitorate.

Casi

Il sistema è in idle e si ha ingresso o uscita di un nodo nel/dal sistema

I componenti agent, linux_native e networkbridge non hanno mostrato correttamente alcuna attività; solamente zenoh viene attivato per aggiornare lo stato del sistema: durante le registrazioni durate circa un minuto, si notano le interazioni tra il nodo che ospita il processo zenoh e il nuovo entrato avvenire in circa 8 secondi, con un consumo medio di banda pari a 10kbps e massimo di circa 100 kbps.

Lifecycle di una applicazione di prova Docker

Utilizzando l'immagine di Hello World resa disponibile all'URI docker.io/gabrik91/loop:latest, si effettuano a distanza di 8 secondi l'una dall'altra le seguenti azioni: onboard, define, configure, start; durante l'esecuzione si verificano due volte i log con la relativa chiamata a distanza di 2 secondi e dopo ulteriori 8 secondi si interrompe l'esecuzione con terminate.

I componenti che mostrano utilizzo di risorse sono zenoh e agent in quanto responsabili della gestione dello stato dei nodi: in particolare ad ogni chiamata di API si ha un uso istantaneo e momentaneo di CPU, mediamente al 50% per il processo di zenoh e 45% per agent. L'utilizzo di RAM rimane invece costante.

Per quanto riguarda il plugin containerd vi è un unico punto in cui si ha attività, corrispondente all'avvio del container.

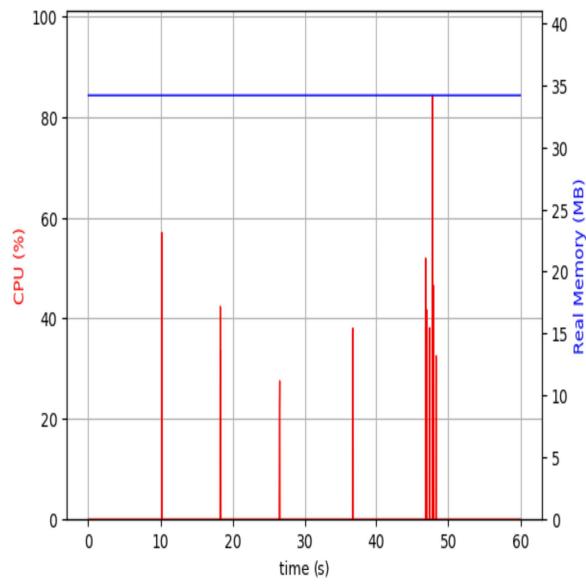


Figure 12: Esempio utilizzo risorse, Zenoh

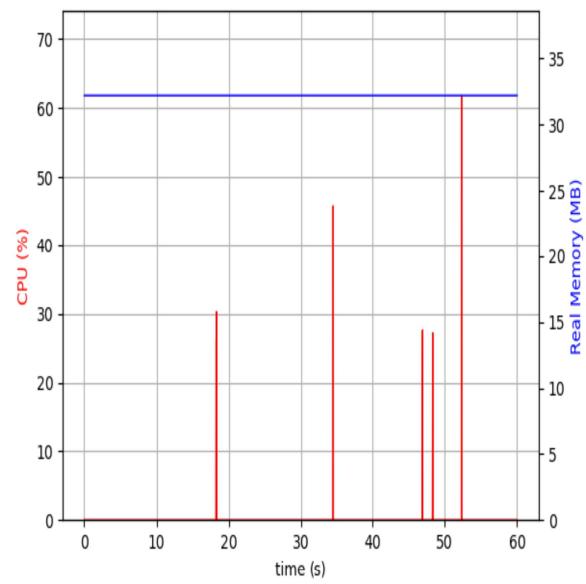


Figure 13: Esempio utilizzo risorse, Agent

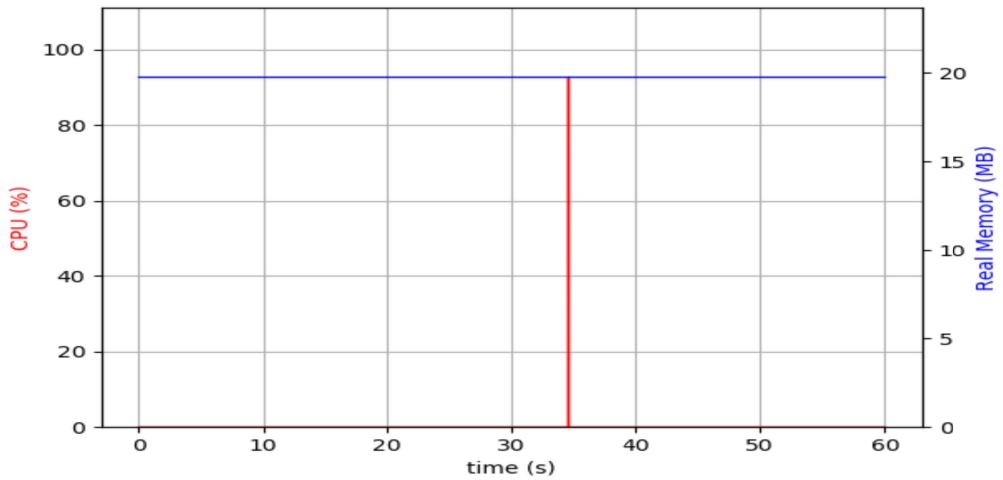


Figure 14: Esempio utilizzo risorse, plugin_containerd

Test di latenza in fase di avvio

Per misurare la latenza dovuta a Fog05 in fase di avvio di una applicazione è possibile costruire una applicazione di test apposita che all'avvio scriva la data corrente, espressa in millisecondi rispetto a epoch; stampando prima dell'avvio di tale applicazione la data espressa nello stesso formato, la latenza è calcolabile come differenza tra i due valori.

Per quanto riguarda un'applicazione nativa, è sufficiente lanciare tramite il plugin native un processo /bin/bash a cui si passa come argomento il path a un file shell eseguibile contenente l'unica istruzione "date +%s%3N" in modo che esso stampi la data con precisione al millisecondo.

Per effettuare la medesima prova con il plugin containerd è sufficiente scrivere un dockerfile che importi il file descritto precedentemente e che abbia come entry point l'esecuzione di tale file tramite shell, per poi costruire ed effettuare push dell'immagine costruita. La latenza tra l'invio del comando è l'esecuzione del container sarà calcolabile come nel caso di applicazione nativa.

I risultati dell'applicazione nativa eseguita al di fuori di Fog05 hanno latenza nulla mentre tramite il framework il ritardo medio è di 1.023 secondi.

Per quanto riguarda l'esecuzione del container, il ritardo medio aggiunto da Fog05 risulta essere mediamente di 0.4781 secondi.

Si deve però far presente che per permettere a Fog05 di registrare correttamente la richiesta di lancio di una applicazione tramite zenoh è necessario introdurre un delay tra le chiamate di define, configure e start: in caso contrario è possibile che una delle chiamate successive venga eseguita prima che la precedente sia stata propagata correttamente all'interno dello storage distribuito; questo fatto dilata di almeno un secondo la latenza quando l'esecuzione avviene tramite Fog05.