UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura

Corso di Laurea in Ingegneria Informatica, Elettronica e delle Telecomunicazioni

Simulazione di Sistemi Fog N-Tier con Posizionamento Dinamico dei Servizi

Simulation of N-Tier Fog Systems with Dynamic Service Placement

Relatore:

Chiar.mo Prof. Michele Amoretti

Correlatore:

Dott. Ing. Gabriele Penzotti

Tesi di Laurea di:

Filippo Scaramuzza

ANNO ACCADEMICO 2020-2021

Ai miei genitori.

Ringraziamenti

Indice

Introduzione

Capitolo 1

Stato dell'Arte

In questo capitolo si introdurranno i principali concetti utili ad una comprensione generale degli aspetti fondanti del *Cloud Computing*, *Fog Computing* e degli altri principali paradigmi in quest'ambito di ricerca. Verranno poi citate alcune possibili applicazioni del *Fog Computing* e verrà fatta una breve introduzione a *YAFS*, il simulatore utilizzato per analizzare generalizzazioni dei suddetti scenari.

1.1 Cloud Computing nell'Era dei Big Data

Il NIST (*National Institute of Standards and Technology*) definisce il *Cloud Computing* come un modello che promuove l'accesso globale alle risorse informatiche condivise, tipicamente *on-demand* [1]. L'infrastruttura di questo paradigma, nella sua versione più semplice, è relegata principalmente in *data center*, ovvero dei raggruppamenti di risorse virtualizzate altamente accessibili che possono essere riconfigurate dinamicamente per garantire la scalabilità dei servizi. Questi fungono da nodi centrali e garantiscono agli utenti un'infrastruttura, una piattaforma oppure un servizio software utile per le loro applicazioni e i propri scopi (IaaS, Paas, SaaS).

Nonostante il *Cloud Computing* abbia indubbiamente un ruolo chiave nel rendere accessibile una potenza di calcolo altrimenti troppo difficile da poter

essere realizzata in proprio, nella moderna era dei Big Data il tempo richiesto per accedere ad alcune applicazioni *Cloud-based*, che concentrano l'intera elaborazione dei dati nei suddetti *data-center*, potrebbe essere troppo elevato e rendere questo paradigma impraticabile per applicazioni *real-time* o, in generale, ovunque la latenza debba essere ridotta al minimo. Inoltre l'ormai noto incremento dei dispositivi connessi in ambito IoT (*Internet of Things*) ed il relativo rapido aumento dei dati generati nell'*edge*[1] della rete richiedono che le risorse di calcolo siano geograficamente situate il più vicino possibile ai dispositivi stessi, così da diminuire al massimo la latenza, aumentando di conseguenza il *throughput* della rete.

Per affrontare queste problematiche è quindi necessario garantire il cosiddetto *Cloud-to-Thing Continuum*, ovvero la possibilità di rendere disponibili potenza computazionale, di storage e di networking ovunque nella rete, dal Cloud agli end-nodes. In questo ambito sono state avanzate numerose proposte, ad esempio il *Fog Computing*, sia in ambito industriale che accademico [2, 3].

1.2 Fog Computing ed altri paradigmi nel Cloud-to-Thing Continuum

Con *Fog Computing* si intende un'architettura a livello di sistema che distribuisce le funzioni di elaborazione, archiviazione, controllo e rete più vicine agli utenti lungo un *Cloud-to-Thing Continuum* [2].

Il *Fog Computing* dunque è innanzitutto caratterizzato da un approccio distribuito. Ciò deriva dal bisogno di superare i limiti dell'approccio centralizzato del Cloud Computing, come latenza, privacy e sovraccarico dei dati. In secondo luogo, i nodi Fog possono essere posizionati ovunque nella rete tra gli end-node e il Cloud [4]. Questa flessibilità che contraddistingue

[1] Con *edge* si intende la zona perimetrale della rete, in cui risiedono gli end-node. L'edge è caratterizzato da un punto di demarcazione (ad esempio un *gateway*) che lo separa dal *network core*, ovvero la parte centrale gestita dagli Internet Service Provider.

il Fog Computing sottolinea la visione di questo paradigma non come una sostituzione, bensì come un'estensione del Cloud Computing, con lo scopo di colmare il divario tra quest'ultimo e i dispositivi IoT, garantendo quindi il continuum *Cloud-to-Thing*.

Ad esempio, in un'applicazione che si occupa di analisi di Big Data prodotti da migliaia di dispositivi IoT, lo strato di Fog che si pone ad un livello "più basso", ovvero più vicino ai dispositivi rispetto al Cloud, potrebbe svolgere una funzione di filtraggio, pre-elaborazione e aggregazione del flusso dati, rendendo eventualmente disponibili alcuni risultati ai livelli inferiori e alleggerendo il carico nel Cloud, a cui rimarrebbero i task più complessi ma su una mole di dati molto ridotta e pre-elaborata.

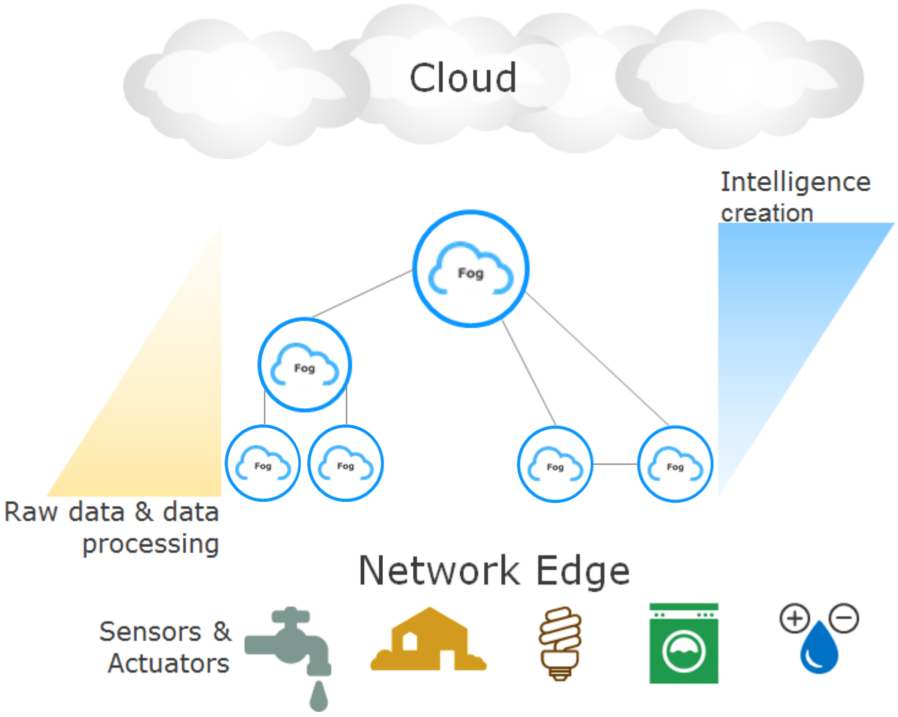

Figura 1.1: Architettura OpenFog N-Tier [2]

Negli anni sono state proposte alcune architetture di particolare rilievo per scenari di Fog Computing. Molte ricerche fanno riferimento ad un'architettura a 3 livelli, composta da Cloud, Fog e IoT [5]. Per questo lavoro di

Tesi l'architettura di riferimento è quella fornita dall'*OpenFog Consortium*. Quest'ultimo è stato fondato da ARM, Cisco, Dell, Intel, Microsoft e dall'Università di Princeton, nel 2015. Ad oggi OpenFog e i suoi membri fanno parte dell'*IIC* (*Industrial Internet Consortium*).

L'architettura di riferimento è detta *N-Tier Architecture*, il cui schema è mostrato in Figura 1.1. Questa mantiene comunque una struttura composta da tre macro-entità: il Cloud, il livello Fog e gli end-node/dispositivi IoT. Il livello Fog, però, è ulteriormente scomposto in sotto-livelli (*tier*) che più si allontanano dai dispositivi, più aumentano le loro capacità computazionali. Il numero di livelli Fog da adottare dipende dai requisiti dello scenario secondo diversi parametri, ad esempio il numero di dispositivi IoT, il carico di lavoro, le capacità dei nodi ad ogni livello, i requisiti di latenza minima e così via. Inoltre i nodi Fog in ogni *tier* possono essere collegati tra loro formando una maglia capace di fornire caratteristiche aggiuntive, come resilienza, tolleranza ai guasti, bilanciamento del carico e così via. Questo significa che i nodi sono in grado di comunicare sia orizzontalmente che verticalmente all'interno dell'architettura Fog.

I vantaggi fondamentali dell'architettura di riferimento definita dall'OpenFog Consortium sono riassunti con il termine *SCALE* [2], ovvero:

- **Security**. Sicurezza aggiuntiva per garantire transazioni sicure e affidabili. Dal *Security Pillar* dell'architettura di riferimento, si evince la necessità di uno o più "nodi fidati" (*Root of Trust*) quando si trattano dati sensibili.

- **Cognition**. L'infrastruttura Fog è consapevole dei requisiti e degli obiettivi delle applicazioni, quindi distribuisce le capacità di elaborazione, comunicazione, controllo e archiviazione lungo il continuum Cloud-to-Things, creando applicazioni che soddisfano meglio le esigenze specifiche dello scenario.

- **Agility**. Lo sviluppo di un nuovo servizio è solitamente lento e costoso, a causa dei costi e dei tempi necessari ai grandi fornitori per avviare

o adottare l'innovazione. Il Fog Computing, invece, offre innovazione rapida e scalabilità conveniente, in cui individui e piccoli team possono utilizzare strumenti di sviluppo aperti (ad esempio API e SDK, secondo il pilastro *Openness* definito da OpenFog [2]) e la proliferazione di dispositivi IoT per offrire nuovi servizi.

- **Latency**. L'architettura Fog supporta l'elaborazione e l'archiviazione dei dati vicino all'utente, con conseguente bassa latenza. Pertanto, il Fog computing soddisfa perfettamente la richiesta di elaborazione in tempo reale, in particolare quindi per le applicazioni *real-time*.

- **Efficiency**. Riconoscimento e condivisione delle risorse inutilizzate dei dispositivi finali che partecipano al networking.

1.2.1 Possibili Applicazioni del Fog Computing

Smart Vehicles e Traffic Control

I veicoli smart (*Smart Vehicles*) sono in grado di produrre giornalmente svariati terabyte di dati grazie alla combinazione di sensori *LIDAR*[2], dei sensori GPS, delle videocamere intelligenti a bordo dei veicoli per il riconoscimento delle immagini e così via. Oltre ai dati prodotti dagli *Smart Vehicles* ci sono quelli generati dai sistemi di controllo intelligenti, ovvero l'infrastruttura del *Traffic Control* (semafori intelligenti, sensori di rilevamento del traffico e delle congestioni, videocamere, ecc.). È evidente come un modello *Cloud-only* non sia adatto, vista l'enorme mole di dati, a garantire un throughput sufficientemente elevato. In quest'ambito è inoltre estremamente importante avere latenze ridotte al minimo, vista la natura *real-time* di molti vincoli decisionali [6]. Un'architettura Fog potrebbe soddisfare le particolari esigenze di questo scenario. In quest'ultimo i nodi Fog a bordo degli *Smart Vehicles*

[2]LIDAR (*Light Detection and Ranging*) è un metodo per determinare la presenza di oggetti, la loro forma e la loro distanza dall'osservatore utilizzando un laser e misurando il tempo necessario alla luce riflessa per tornare al trasmettitore laser.

possono offrire i normali servizi di *infotainment*, ADAS[3], guida autonoma, sistemi anticollisione, navigazione, e così via, comunicando per mezzo di diverse tecnologie, come DSRC (*Dedicated Short Range Communications*) o le reti cellulari (3G, LTE, 5G, ecc.).

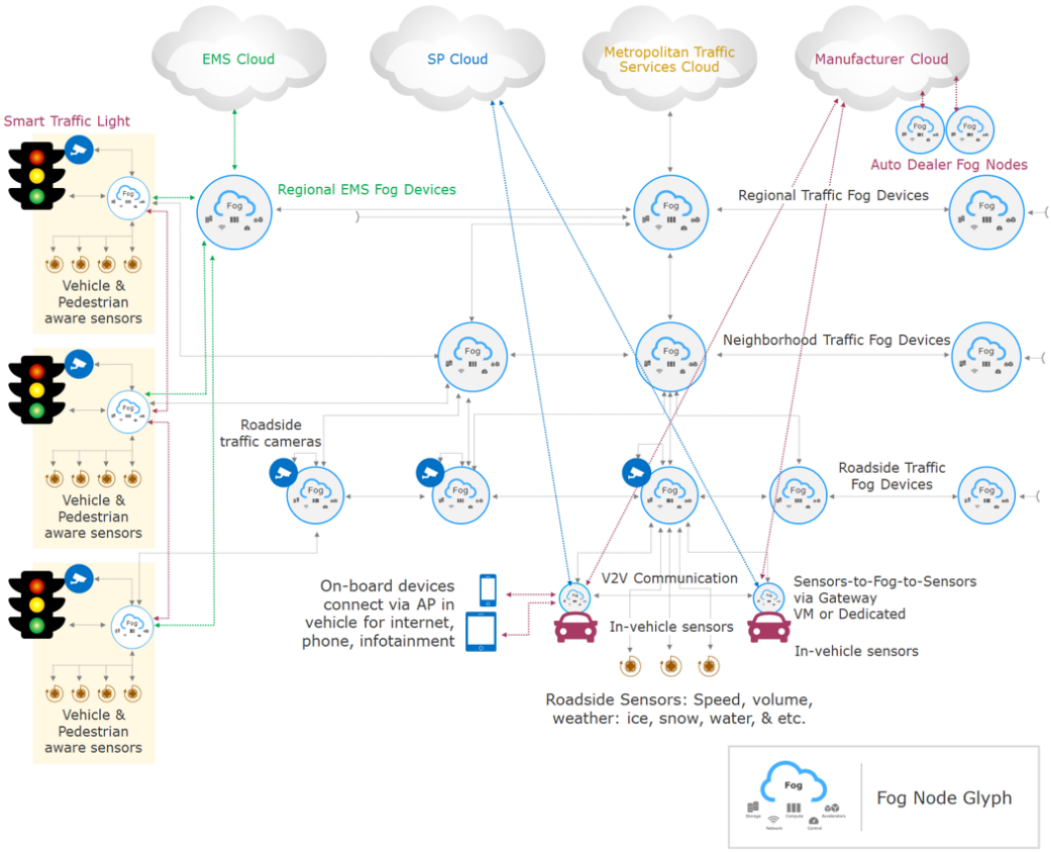

Figura 1.2: Architettura di uno scenario Fog in ambito Smart Vehicles e Traffic Control [2]

L'architettura di riferimento, mostrata in Figura 1.2, è strutturalmente gerarchica, con 3 livelli di nodi Fog. Il primo livello (*Roadside Fog Nodes*) si occupa di raccogliere i dati dai vari sensori e telecamere. I nodi Fog in questo livello eseguono alcune veloci analisi, utili ad esempio a comunicare ai veicoli in transito particolari condizioni del traffico o della strada. I dati

[3]ADAS (*Advanced Driver Assistance Systems*) è un insieme di tecnologie che assistono il guidatore nelle operazioni di guida e parcheggio in modo sicuro.

aggregati dal primo livello sono inviati al secondo e/o al terzo livello (non c'è necessariamente una gerarchia nelle comunicazioni, questo per definizione dell'architettura Fog e della sua flessibilità), ovvero i *Neighborhood Fog Nodes* e i *Regional Fog Nodes*. In genere, ogni livello Fog nella gerarchia fornisce ulteriori capacità di elaborazione, storage e rete. Ad esempio, livelli gerarchicamente più alti garantiscono un trattamento aggiuntivo per fornire analisi dei dati o capacità di archiviazione di grandi dimensioni, utili ad esempio per analisi sul lungo periodo o per inviare dati ad altre zone della rete stradale per particolari task.

Smart Cities e Smart Buildings

Un altro interessante impiego del Fog Computing è nell'ambito delle *Smart Cities* e degli *Smart Buildings*. Infatti sebbene la maggior parte delle città moderne disponga di una o più reti cellulari che forniscono una copertura dell'intera città, queste reti hanno spesso limiti di capacità e larghezza di banda che soddisfano appena le esigenze degli attuali abbonati. Ciò lascia poca larghezza di banda per i servizi più avanzati previsti in una città intelligente nell'era dei Big Data, come *Smart Parking*, *Traffic Control*, ospedali intelligenti, controllo sul consumo elettrico e così via.

Questo caso d'uso è già realtà, ad esempio, nella città di Barcellona, dove i risultati hanno dimostrato che il Fog Computing è la possibile chiave di volta per l'implementazione di servizi molto avanzati e complessi, in città che producono decine di milioni di Gigabyte di dati giornalmente [7].

1.2.2 Altri Paradigmi

Per analizzare i vari altri paradigmi che possono essere considerati un'estensione, più che un'alternativa al Fog Computing, è utile introdurre il concetto di *Edge Computing*. Quest'ultimo è un un paradigma nato dall'esigenza di spostare la capacità di calcolo verso l'edge della rete. In [8] viene definito l'Edge Computing come *"tecnologie abilitanti che consentono di effettuare calcoli ai margini della rete sui dati, a valle per conto di servizi Cloud e a*

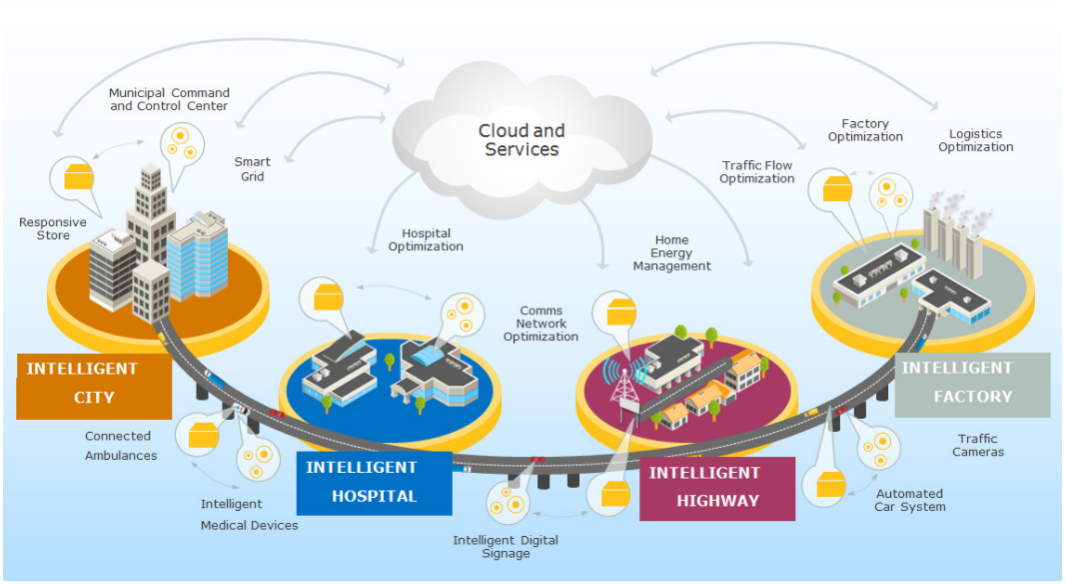

Figura 1.3: Opportunità per una *Smart City* [2]

monte per conto di servizi IoT". L'idea è quella di estendere le capacità dal Cloud all'edge della rete, con l'obiettivo di portare la potenza di calcolo il più vicino possibile ai generatori dei dati, ovvero ai dispositivi IoT. Nonostante sia il Fog Computing che l'Edge Computing muovano le capacità computazionali vicino agli end-node, OpenFog afferma che l'Edge Computing venga erroneamente chiamato Fog Computing (e viceversa): la fondamentale distinzione sta nel fatto che il Fog Computing è strutturalmente gerarchico e fornisce potenza computazionale, networking e storage ovunque nella rete, dal Cloud agli end-node (*Cloud-to-Thing Continuum*), mentre l'Edge Computing tende ad essere limitato all'edge [2].

I principi fondanti dell'Edge Computing possono essere messi in pratica in diversi modi, in termini di tipo di dispositivi utilizzati, protocolli di comunicazione adottati e così via.

Mobile Cloud Computing e Cloudlet Computing

Il *Mobile Cloud Computing* (MCC) è basato sul concetto del *mobile offloading*: l'idea alla base, per un dispositivo mobile, è quella di delegare, quando

possibile, storage e calcoli ad entità remote (ad esempio il Cloud) in modo da ridurre il carico di lavoro e ottimizzare il consumo di energia. In realtà oggi il concetto di MCC è stato esteso tenendo in considerazione i principi dell'Edge Computing. La nuova interpretazione del MCC è quella di delegare l'elaborazione e lo storage dei dati a dispositivi situati all'edge della rete, piuttosto che al Cloud. L'implementazione più comune di questa visione è il *Cloudlet Computing* (CC), che consiste nell'utilizzare dei *Cloudlet*[4] per eseguire elaborazione ed archiviazione dei dati vicino ai dispositivi finali.

Multi-access Edge Computing

Esattamente come il *Mobile Cloud Computing* è un'estensione del *Mobile Computing* attraverso il *Cloud Computing*, analogamente, il *Multi-access Edge Computing* (MEC) è un'estensione del *Mobile Computing* attraverso l'*Edge Computing*. In [9] il MEC viene definito come una piattaforma che fornisce funzionalità IT e di Cloud computing all'interno della *Radio Access Network* (RAN) in 4G e 5G, in prossimità dei dispositivi mobili. Il Multi-access Edge Computing è stato precedentemente definito come *"Mobile Edge Computing"*, ma il paradigma è stato ampliato per includere una più ampia gamma di applicazioni oltre alle attività specifiche per dispositivi mobili. Esempi di applicazioni di MEC includono analisi video, *Smart Vehicles*, monitoraggio della salute e realtà aumentata.

Mist Computing

Spesso chiamato anche *IoT Computing*, il *Mist Computing* viene impiegato per raggiungere l'edge più estremo dei dispositivi connessi (micro-controllori e sensori) [10]. Il Mist Computing può essere visto come la prima posizione di calcolo nel continuum IoT-Fog-Cloud. Infatti può aiutare a conservare la larghezza di banda e la carica della batteria poiché sono solo i dati es-

[4]Un *Cloudlet* è una piccola infrastruttura Cloud *affidabile*, situata nell'edge della rete disponibile per i dispostivi mobili vicini, che collabora con il Cloud per servire i dati in modo più efficiente.

senziali ad essere trasmessi al gateway, al server o al router. Inoltre il Mist Computing offre l'utilizzo di meccanismi di controllo dell'accesso ai dati che possono garantirne la riservatezza a livello locale. Di contro, questo paradigma ha che i micro-controllori e i sensori utilizzati nell'infrastruttura possono essere utilizzati solo per un'elaborazione leggera, pertanto il loro utilizzo è ristretto ad un numero piuttosto esiguo di applicazioni, a meno dell'implementazione dello stesso nel continuum Clout-to-Thing venendo quindi incluso in un'architettura più ampia, basata ad esempio sul Fog Computing [11].

1.3 Simulatori di Fog Computing

La progettazione e l'ottimizzazione di sistemi distribuiti su larga scala richiedono una descrizione realistica del flusso dei dati, delle richieste emesse dai nodi, dei servizi disponibili e ogni aspetto necessario alla comprensione del comportamento di sistemi che scambiano enormi moli di dati. La via più semplice per comprendere le potenzialità e allo stesso tempo i limiti di scenari così complessi è quella della *simulazione*.

La ricerca nell'ambito del Fog Computing vanta un numero consistente di strumenti di simulazione più o meno avanzati. Tra i più noti vi sono *iFogSim*[5] [12] ed *EdgeCloudSim*[6] [13], entrambi basati su *CloudSim*, un simulatore di architetture Cloud. Oltre alla simulazione un altro importate strumento che garantisce esperimenti ripetibili e controllabili è l'*emulazione*. Tra i software più rilevanti esistono *EmuFog*[7] [14] e *FogBed*[8] [15]. Entrambi consentono all'utente di progettare scenari Fog con applicazioni basati su Docker o Mininet.

[5]Disponibile su: `https://github.com/Cloudslab/ifogsim`
[6]Disponibile su: `https://github.com/CagataySonmez/EdgeCloudSim`
[7]Disponibile su: `https://github.com/emufog/emufog`
[8]Disponibile su: `https://github.com/fogbed/fogbed`

1.3.1 YAFS, *Yet Another Fog Simulator*

Per questo lavoro di Tesi è stato utilizzato, tra le altre cose, il simulatore *YAFS*[9] (*Yet Another Fog Simulator*), un simulatore ad eventi discreti sviluppato in Python e basato su *SimPy*, ovvero un framework DES (*Discrete Event Simulator*) basato sui processi, anch'esso sviluppato in Python [16]. YAFS è progettato per analizzare e progettare applicazioni in scenari Fog e incorpora le strategie per il service placement, lo scheduling e l'instradamento dei dati. Le ragioni che hanno portato alla scelta di YAFS sono esposte nel seguito.

- **Placement, Scheduling, Routing e processi personalizzati**. L'algoritmo di service placement viene invocato all'avvio e viene eseguito durante la simulazione secondo una distribuzione personalizzata. L'algoritmo di routing sceglie il percorso che collega il mittente e il destinatario e l'algoritmo di scheduling sceglie l'applicazione che deve eseguire il task associato alla richiesta. Oltre agli algoritmi appena esposti che possono essere definiti dall'utente, quest'ultimo può implementare funzioni personalizzate che possono essere invocate in fase di esecuzione per fornire implementazioni flessibili di eventi reali come il movimento delle fonti del carico di lavoro, la generazione di guasti della rete e la raccolta di dati specifici utilizzando anche applicazioni di terze parti.

- **Creazione dinamica delle sorgenti dei messaggi**. Ogni fonte di carico di lavoro rappresenta la connessione alla rete di un utente, di un sensore IoT o di un attuatore che richiede un servizio. Ogni sorgente è associata ad un'entità DES di rete che genera richieste secondo una distribuzione personalizzata. Le sorgenti del carico di lavoro possono essere create, modificate o rimosse dinamicamente, consentendo la modellazione degli spostamenti degli utenti in un generico ecosistema.

- **Topologia della rete**. YAFS basa la struttura della topologia sulla *Complex Network Theory*, grazie all'implementazione della libreria

[9]Disponibile su: `https://github.com/acsicuib/YAFS`

NetworkX, da cui deriva la possibilità di applicare tutti gli algoritmi che ne derivano, ottenendo quindi indicatori di maggior interesse sulle topologie adottate.

- **Risultati**. YAFS esegue la registrazione automatica basata su CSV di due tipi di eventi:
  - generazione del carico di lavoro e del calcolo eseguito su di esso;
  - trasmissioni dei messaggi sui collegamenti.

  I risultati sono salvati in formato *raw* con un'impronta noSql, con l'idea che da dati in formati più semplici derivino analisi più veloci.

Capitolo 2

Architettura del Sistema Simulato

In questo capitolo verranno affrontati innanzitutto i dettagli implementativi del simulatore *YAFS*, la sua struttura e la modellazione delle simulazioni. Successivamente verranno approfondite le caratteristiche degli scenari simulati nel corrente lavoro di Tesi.

2.1 Struttura e Funzionamento di YAFS

Per le simulazioni realizzate nel corrente lavoro di Tesi è stato fatto uso del simulatore *YAFS* [1] (*Yet Another Fog Simulator*) [16]. Quest'ultimo utilizza una libreria per la generazione e la gestione degli eventi chiamata *SimPy* [2], ovvero un'implementazione di un simulatore ad eventi discreti (DES, *Discrete Event Simulator*), che garantisce un'interfaccia per la definizione dei processi (i componenti attivi della simulazione) e delle risorse (ad esempio i nodi ed i collegamenti della rete).

YAFS è definito principalmente da sei classi: *Core*, *Topology*, *Selection*, *Placement*, *Population* e *Application*. Le relazioni che intercorrono tra loro

[1] Disponibile su: `https://github.com/acsicuib/YAFS`

[2] Disponibile su: `https://simpy.readthedocs.io`

sono mostrate in Figura 2.1 ed esposte nel seguito.

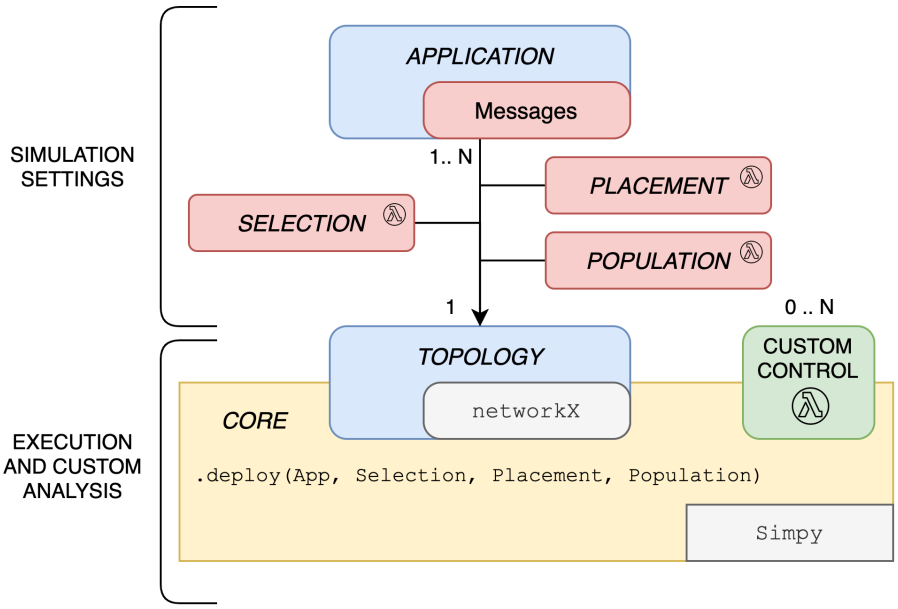

Figura 2.1: Architettura di YAFS.[16]

2.1.1 Topologia e Modellazione delle Entità

Le entità della topologia sono modellate come un insieme di *nodi* (ovvero i dispositivi della rete, come dispositivi IoT, nodi fog, server e cloudlet) interconnessi da *archi* (i collegamenti di rete). La modellazione della topologia, tramite un *grafo* permette l'applicabilità della *Complex Network Theory*. Grazie all'integrazione di *NetworkX* [17], una nota libreria scritta in Python, è possibile applicare algoritmi per eseguire misure e analisi sui grafi, come il calcolo di node degree, centrality, clustering, assortativity, communities e così via. NetworkX accetta inoltre la definizione dei grafi tramite JSON, linguaggio ampiamente utilizzato per la creazione dello scenario con YAFS e permette l'esportazione dei grafi in formato GEXF, utile ad esempio per l'analisi dei grafi tramite il software *Gephi* [3].

[3]Disponibile su: `https://gephi.org/`

Gli attributi obbligatori per la definizione di un nodo sono un identificativo univoco (ID), il numero di istruzioni eseguite dal nodo in un'unità di tempo (IPT) e la capacità della memoria (RAM). L'utente è libero di aggiungere attributi personalizzati, utili allo scenario specifico che si vuole studiare, come è stato fatto nel corrente lavoro di Tesi (maggiori informazioni sono al Capitolo 3). Un esempio di definizione dei nodi è mostrato nel Listato 2.1.

```
{
  "id": 120, "RAM": 1, "IPT": 530,
  "POWERmin": 574,
  "POWERmax": 646,
  "coordinate":
  {
    "lat": 39.30, "long": 3.34
  }
},
{
  "id": 12, "RAM": 10, "IPT": 100
}
```

Listato 2.1: Definizione di due nodi Fog utilizzando la rappresentazione JSON [16]

La definizione dei collegamenti è molto simile. Questi hanno i seguenti attributi obbligatori: la larghezza di banda (BW), la velocità di propagazione (PR), l'ID del nodo sorgente (s) e l'ID del nodo di destinazione (d). I valori BW e BR sono utili al calcolo della latenza secondo la formula:

$$\frac{Message.size.bytes}{BW} + PR$$

2.1.2 Modellazione delle Applicazioni

In YAFS le applicazioni (intese come raggruppamenti di servizi), sono strutturate come dei *Distributed Data Flow* (DDF) [18]. In particolare un'applicazione è definita da un insieme di moduli che si scambiano messaggi. Infatti, un DDF è rappresentato da un *grafo diretto aciclico*, dove i nodi sono i moduli che eseguono delle azioni sui messaggi in ingresso, provenienti da un certo percorso sugli archi del grafo. Questa rappresentazione è utile al fine

di garantire il partizionamento delle applicazioni e la scalabilità, ad esempio tramite l'implementazione di microservizi [19].

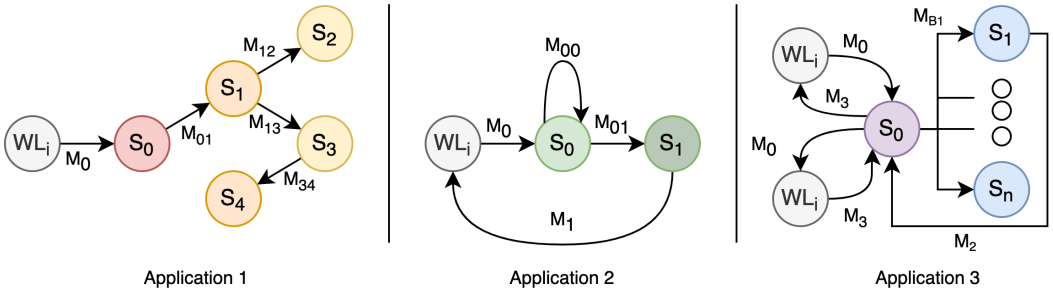

Figura 2.2: Tre tipologie di applicazioni realizzabili, con la loro rappresentazione tramite grafo.[16]

La definizione delle applicazioni è composta da quattro parti: *moduli* (o *servizi*), *messaggi*, *trasmissioni* e dati generali. I *moduli* possono avere diversi attributi, anche a seconda dello specifico scenario, ma quelli obbligatori sono solo un identificativo univoco (*ID*) e il nome (*name*). I *messaggi*, ovvero i dati scambiati, hanno principalmente due attributi obbligatori: il numero di istruzioni (*instructions*) e la grandezza in byte (*bytes*). Le *trasmissioni* definiscono le modalità in cui i servizi scambiano le informazioni e tramite le quali si inviano dati. Gli attributi obbligatori in questo caso sono: il modulo di afferenza (*module*) e il messaggio in ingresso. Tramite la definizione delle trasmissioni è possibile, con un particolare attributo detto *fractional*, definire la probabilità di propagare un determinato messaggio *message_out* sulla ricezione di un particolare messaggio *message_in*.

In Figura 2.2 sono mostrate tre tipologie di applicazioni che sono realizzabili. La prima, `Application 1`, è strutturata gerarchicamente, con la ricezione dei messaggi M_{ij} che scatena l'invio di altri messaggi. Nella seconda applicazione, `Application 2`, è possibile osservare un'interazione di un servizio con se stesso ed infine nell'ultima applicazione, `Application 3`, è mostrato il *broadcasting* di un messaggio M_{B1} che raggiunge tutti i moduli dell'applicazione. In tutte e tre le applicazioni WL_i indica il nodo che genera il carico di lavoro (ad esempio un dispositivo IoT).

2.1.3 Politiche dinamiche

Le classi *Selection*, *Placement* e *Population* sono utili per la generazione dinamica degli eventi dello scenario. In particolare la prima definisce quali nodi devono eseguire un particolare servizio, di conseguenza indirizza il *workload*. La classe *Placement* sceglie i servizi che devono essere allocati nei vari nodi, mentre la class *Population* posiziona i generatori del workload nei nodi della rete. Queste tre classi possiedono principalmente due interfacce: una contenente l'*initialization function* (che prepara l'allocazione dei moduli e del workload nei nodi della rete) e una contenente una funzione che viene invocata secondo una specifica distribuzione temporale.

```
1  delayActivation = deterministicDistributionStartPoint(300, 300, name='
       ↪ Deterministic')
2  periodActivation = deterministicDistribution(name='Deterministic', time=100)
3
4  popA = Statical(name='StaticalPop')
5  popA.set_sink_control({'id': a_id_fog_device, 'number': 2, 'module': appA.
       ↪ get_sink_modules()})
6  popA.set_src_control({'number': 1, 'message': appA.get_message('M.Action'),
       ↪ 'distribution': periodicActivation})
7
8  top20Devices = [''array_ids_fog_devices'']
9  popB = Evolution(top20Devices, name='DynamicPop', activation_dist =
       ↪ delayActivation)
10 popB.set_sink_control({'model': 'actuator-device', 'number': 2, 'module':
       ↪ appB.get_sink_control()})
11 popB.set_src_control({'number': 1, 'message': appB.get_message('M.action'),
       ↪ 'distribution': periodicActivation})
```

Listato 2.2: Definizione di due Population policies: una statica (`popA`) ed una dinamica (`popB`) [16]

Nel Listato 2.2 è mostrato un esempio di definizione di politiche di *Population*: una statica ed una dinamica. Nelle righe 1 e 2 vengono definite due distribuzioni temporali: la prima che inizia a 3000 unità temporali e da quel punto in poi invoca un'attivazione ogni 300 unità temporali, la seconda invece che invoca un'attivazione ogni 10 unità temporali. Alla riga 4 viene generata un'istanza di una classe Population predefinita. Le applicazioni YAFS hanno due tipi di moduli: *workload sources* e *workload sinks* (rispettivamente "sen-

sori" ed "attuatori"). Le righe 5 e 6, tramite JSON, definiscono l'allocazione dei moduli sink e dei moduli source con una specifica distribuzione e il tipo di messaggio che questi devono trattare.

Nel caso della seconda politica di *Population*, viene utilizzato un modello più complesso (righe 8-11): alla riga 9 viene istanziato un oggetto Evolution (Listato 2.3). Questo segue la distribuzione di riga 1, dunque, il processo DES creato, inizia a produrre messaggi dopo un certo intervallo di tempo secondo una specifica scadenza. Ad ogni attivazione il processo creato genera workload con le caratteristiche definite alla riga 11.

```
1
2  class Evolution(population):
3          def __init__(self, listIDEntities, **kwargs):
4                  # initialization of internal variables
5                  super(Evolutionm self).__init__(**kwargs)
6
7          def initial_allocation(self, sim, app_name):
8                  # dealing assignments
9                  sim.deploy_sink(app_name, node=fog_device, module=module)
10
11         def run(self, sim):
12                 # dealing assignements: msg, distribution and app_name
13                 id = ... # listIDEntities.next
14                 idsrc = sim.deploy_source(app_name, id_node=id, msg=...,
       ↪ distribution=...)
```

Listato 2.3: Struttura di una classe Population [16]

Nel Listato 2.3 è mostrata una versione semplificata della classe Evolution. In questo tipo di classi è sempre presente una funzione obbligatoria chiamata `initial_allocation` e, facoltativamente, una funzione chiamata `run` che viene chiamata secondo l'eventuale distribuzione temporale specificata.

2.2 Descrizione dello scenario simulato

L'architettura di riferimento per la definizione dello scenario simulato è illustrata in Figura 2.3.

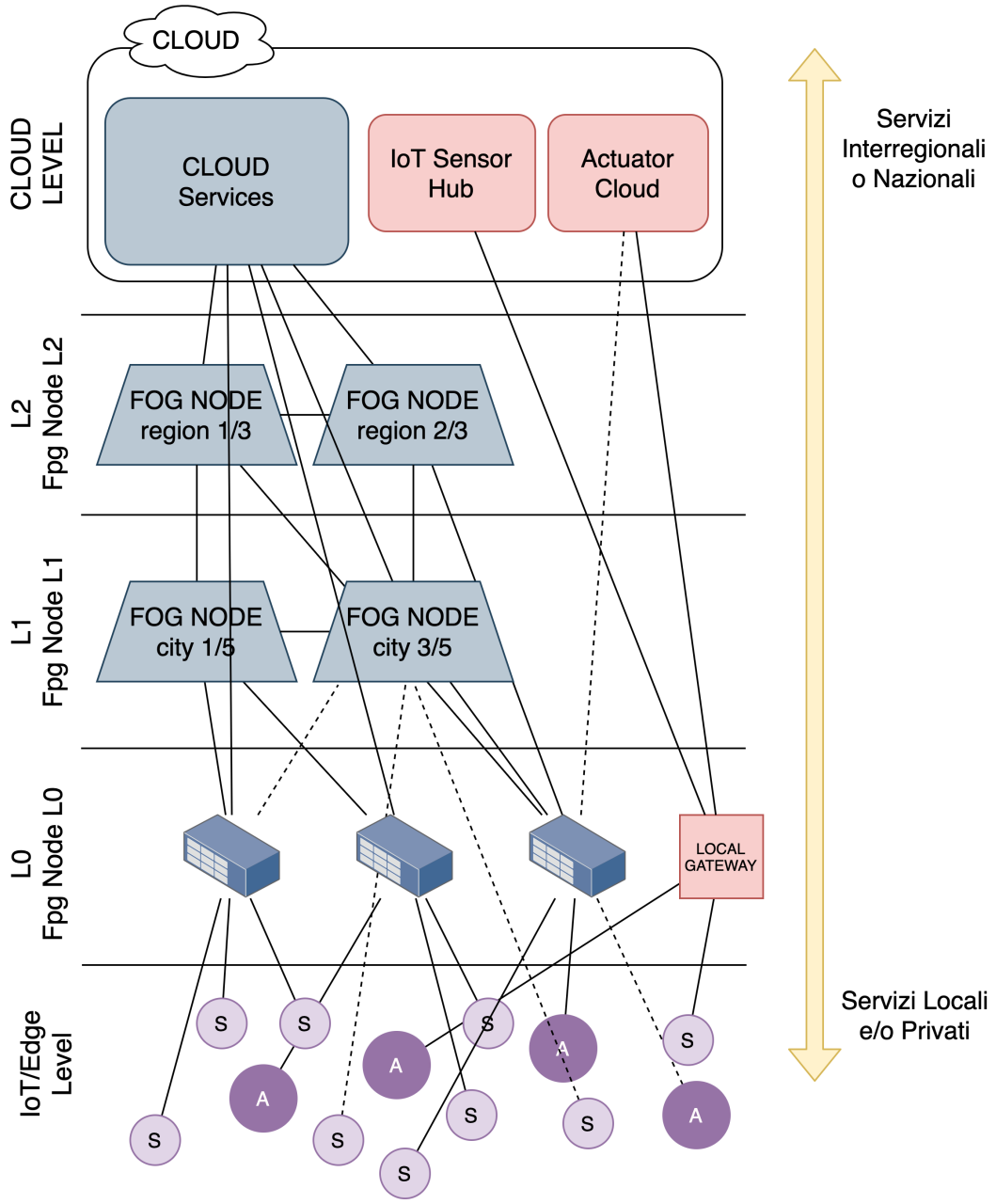

Figura 2.3: Architettura dello scenario simulato

Tra i vari aspetti considerati durante la definizione della topologia di rete si è voluto enfatizzare il concetto del Fog Computing come "architettura verticale". Dall'esempio in Figura 2.3 è infatti possibile evincere una strut-

tura della rete gerarchica ed a livelli. Ogni livello è caratterizzato da diverse capacità di elaborazione, in base ai servizi offerti alla rete: un nodo a livelli più bassi offre pochi servizi, ma generalmente produce molti dati (si pensi ad esempio ai dispositivi IoT), mentre un nodo a livelli più alti non genera dati *propri*, piuttosto offre una rielaborazione dei dati ricevuti dai livelli più bassi offrendo molti servizi alla rete.

Un fattore di particolare rilevanza è inoltre la *raggiungibilità*. Ricordando infatti la necessità di rendere i servizi della rete disponibili anche nelle zone più remote della stessa, ovvero vicino all'*edge* per ridurre al minimo la latenza, i nodi appartenenti a livelli più bassi sono più radicati nel territorio ed ognuno di essi è raggiungibile da aree via via più limitate e circoscritte.

2.2.1 Architettura a livelli

Il sistema simulato in questo lavoro di Tesi offre degli spunti per l'implementazione del Fog Computing in scenari anche molto differenti tra loro. Infatti viene proposta una architettura a livelli molto flessibile e adattabile a diversi tipi di implementazioni, che possono avere qualsiasi estensione geografica a seconda delle diverse esigenze.

Livello IoT/Edge

In questo livello vengono raggruppati i sensori e gli attuatori che rientrano nell'ambito IoT, cioè dispositivi caratterizzati da una bassa capacità computazionale, scarsa alimentazione, discontinuità e mobilità. I dispositivi che operano in questo livello utilizzano protocolli di comunicazione a basso consumo, con un *payload* ridotto e tempi di *sleep* prolungati. La caratteristica fondamentale di questi dispositivi è infatti che, per ridurre i consumi, rimangono per tempi molto prolungati in una fase di *sleep*, nella quale il dispositivo resta senza alimentazione (tranne per la circuiteria fondamentale), per poi passare allo stato di *awake*, nel quale vengono eventualmente inviati o ricevuti i dati, tornando immediatamente nello stato di *sleep*.

In questo livello vengono inclusi anche i cosiddetti nodi *edge*, il cui scopo è quello di eseguire validazione o controllo dei dati inviati o ricevuti. In particolare questi dispositivi sono generalmente centraline per attuatori o stazioni per sensori più performanti che possono controllarne un numero più consistente.

Livello Fog $L0$

In questo livello sono raggruppati i nodi che forniscono i servizi che non rientrano strettamente nell'ambito IoT, ma che non ne offrono ai livelli superiori. Sono i cosiddetti *gateway*, spesso privati, che raggruppano sensori e attuatori di zone circoscritte per il loro monitoraggio.

Livello Fog $L1$

In questo livello si trovano i nodi che si occupano di raccogliere ed aggregare i dati provenienti da più zone. Questi nodi possono avere, ad esempio, una copertura provinciale o sub-provinciale. Questi offrono diversi servizi alla rete, sia ai i livelli superiori che ai quelli inferiori, ad esempio fornendo raccolta dati per il mantenimento dello storico per i livelli inferiori e fornendo pre-elaborazione degli stessi per i livelli superiori. Le capacità di questi nodi sono piuttosto limitate, ma hanno il grande vantaggio di essere replicabili al fine di garantire la scalabilità. Nel caso di sovraccarico è inoltre possibile sfruttare ad esempio l'operazione di *offloading*[4] su nodi vicini dello stesso livello.

[4]Con *offloading* si intende il trasferimento di attività computazionali ad alta intensità ad un processore separato, come un acceleratore hardware o ad una piattaforma esterna, come un cluster o il cloud. L'*offload* dell'elaborazione su un nodo esterno può fornire maggiore potenza di elaborazione e superare i limiti hardware di un dispositivo, come potenza di calcolo, archiviazione ed energia limitate.

Livello Fog $L2$

I nodi presenti in questo livello (e negli eventuali successivi Livelli Fog LN) hanno una copertura maggiore rispetto a quella dei livelli precedenti, ad esempio regionale o sub-regionale. Come anticipato in questo modello architetturale, i nodi ai livelli superiori hanno capacità computazionali via via sempre maggiori, in grado quindi di aggregare maggiormente i dati, effettuare veloci analisi statistiche o implementare algoritmi di Machine Learning distribuiti.

Livello Cloud

In questo livello vengono raggruppati i servizi controllati dai produttori dei sensori e degli attuatori (si pensi ad esempio ai servizi utili a raccogliere dati rilevati o ad inviare particolari segnali di controllo agli attuatori), nonché i servizi che richiedono risorse elevate per eseguire, ad esempio, algoritmi di Machine Learning o analisi statistiche avanzate. Nel Capitolo 3 verrà inoltre spiegato come il Cloud sia un tassello fondamentale per l'algoritmo di Placement, fungendo da "riserva" per i servizi che non trovano un nodo utile per il piazzamento.

2.2.2 Interconnessioni tra Livelli e Scambio di Messaggi

Una delle caratteristiche del Fog Computing è la flessibilità della sua implementazione. Infatti i nodi, seppur ordinati gerarchicamente, non sono necessariamente connessi con i nodi ai livelli subito successivi o precedenti, ma possono essere eventualmente connessi con qualsiasi altro livello superiore o inferiore.

Per quanto riguarda invece le interconnessioni interne ai livelli, queste seguono diverse modalità. Nel caso dei livelli IoT/Edge e Fog $L0$ generalmente

non sono previste connessioni, mentre nei livelli Fog da $L1$ ad LN viene utilizzato un grafo *small-world*[5] secondo il modello *Watts-Strogatz* [20].

Il sistema simulato è un'implementazione di un'architettura basata sullo scambio di messaggi tra i servizi. Questo avviene tramite un'operazione di *replay* dei messaggi che avviene secondo specifiche distribuzioni di probabilità, che diminuisce ai livelli superiori. Ad esempio un messaggio generato dal Livello IoT/Edge raggiunge il livello $L0$ con probabilità pari a 0.8, il livello $L1$ con probabilità pari a 0.4 e così via.

[5]Una rete *small-world*, è un tipo di grafo in cui la maggior parte dei nodi sono uno vicino (*neightbors*) dell'altro e dato un nodo, i suoi vicini sono molto probabilmente vicini tra loro, con il risultato che ogni nodo è raggiungibile dall'altro con un ridotto numero di *hop*.

Capitolo 3

Sviluppo ed Utilizzo

3.1 Sistema Realizzato per Simulazioni ed Analisi

Al fine di valutare le prestazioni degli scenari di Fog Computing, descritti al Capitolo 2, è stato implementato un sistema di simulazione che ne permette in una prima fase la definizione (topologia, applicazioni, servizi, richieste, ecc...) e, successivamente, l'analisi dei principali aspetti utili alla comprensione dello scenario, come il successo del *service placement* e delle richieste di servizi da parte dei nodi della rete.

La definizione e l'esecuzione della simulazione seguono il diagramma di flusso mostrato in Figura 3.1. Tramite il software realizzato è possibile eseguire due principali tipologie di analisi:

1. **analisi del service placement con cambio di parametri**: viene valutato il successo del service placement al variare di specifici parametri di definizione dello scenario, specificando il numero di iterazioni e quali di questi devono variare ad ogni esecuzione;

2. **simulazione di uno scenario specifico**: una volta definito e simulato uno specifico scenario, è possibile ottenere un'analisi sul soddisfacimen-

to delle richieste da parte dei nodi dei vari servizi offerti dalla rete, con e senza *failure control* dei nodi.

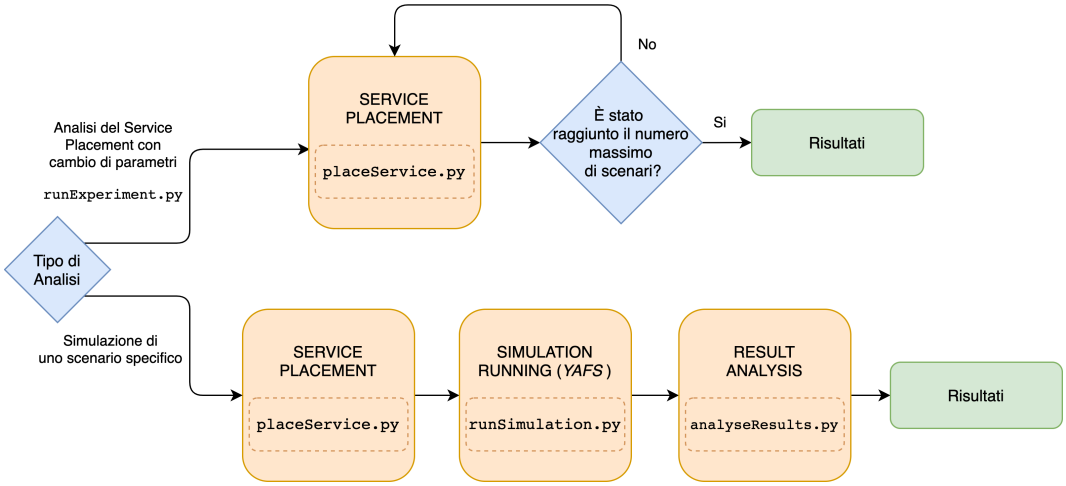

Figura 3.1: Diagramma di flusso del sistema di simulazione.

3.1.1 Utilizzo del Software Realizzato

Entrambe le tipologie di simulazione prevedono alcuni passaggi obbligati. Nella fase di inizializzazione è necessaria la configurazione dell'esperimento. Questa può essere eseguita tramite opportune modifiche al file `experimentConfiguration.py`. Le possibilità di configurazione sono esposte nel seguito.

Configurazione della Topologia

La topologia, generata dinamicamente in fase di inizializzazione, può essere configurata con i seguenti parametri:

- `IOT_DEVICES_NUM`: il numero di device IoT che la rete deve supportare.
- `NETWORK_LEVELS_NUM`: il numero di livelli dello scenario Fog. Questo valore comprende il livello IoT, il livello Gateway, i livelli Fog e il livello Cloud.

- `REDUCTION_FACTOR_1`: il fattore di riduzione del numero di nodi dal livello IoT al livello Gateway.
- `REDUCTION_FACTOR_2`: il fattore di riduzione del numero di nodi dal livello FOG (i) al livello FOG ($i+1$).
- `LINK_GENERATION_PROBABILITY_FOG0`: probabilità di generazione di collegamenti tra i nodi del livello FOG (0).
- `HUB_GENERATION_PROBABILITY`: probabilità di generazione di nodi "hub" (con un elevato *degree*) all'interno dei livelli FOG.
- `MIN_CONN_TO_UPPER_LEVEL`: numero minimo di connessioni da un livello FOG a quelli superiori.
- `MAX_CONN_TO_UPPER_LEVEL`: numero massimo di connessioni da un livello FOG a quelli superiori.

Configurazione delle Caratteristiche di Rete

È possibile, modificando gli opportuni parametri, agire sulle caratteristiche dei singoli nodi appartenenti alla topologia, come la loro memoria (RAM) o il numero di istruzioni per unità di tempo (IPT). La prima è configurabile tramite parametri nella forma `FUNC_NODE_RAM_LEVEL`, dove `LEVEL` può essere `FOGi` (con `i` pari al numero del livello Fog di riferimento) oppure `CLOUD`. Allo stesso modo è possibile configurare le istruzioni per unità di tempo, tramite i parametri nella forma `FUNC_NODE_IPT_LEVEL`. Questi parametri e tutti gli altri che hanno il prefisso `FUNC`, possono essere inizializzati con una stringa contenente una determinata distribuzione, ad esempio:

```
1 self.FUNC_NODE_IPT_FOG2 = "random.randrange(400, 900)"
```

Allo stesso modo è possibile agire sulle caratteristiche dei collegamenti di rete, ovvero la loro larghezza di banda (BW) e la velocità di propagazione del segnale (PR). Per entrambe i parametri di configurazione sono nella forma:

- `FUNC_EDGE_[PR o BW]_SAME_LEVEL`: il valore viene assegnato a collegamenti tra nodi che appartengono allo stesso livello.
- `FUNC_EDGE_[PR o BW]_ADJ_LEVEL`: il valore viene assegnato a collegamenti tra nodi che appartengono a livelli adiacenti.
- `FUNC_EDGE_[PR o BW]_NON_ADJ_LEVEL_1`: il valore viene assegnato a collegamenti tra nodi che sono separati da un solo altro livello oltre a quelli di appartenenza.
- `FUNC_EDGE_[PR o BW]_NON_ADJ_LEVEL_2`: il valore viene assegnato a collegamenti tra nodi che sono separati da più di un livello oltre a quelli di appartenenza.

Configurazione delle Applicazioni e delle Richieste

Tramite opportuni parametri, è possibile configurare la generazione delle applicazioni e dei servizi ad esse collegati. Inoltre è possibile specificare il numero di richieste che devono essere effettuate dagli utenti (i dispositivi IoT o i nodi Gateway).

- `FUNC_APP_GENERATION`. Le applicazioni generate sono dei *grafi diretti aciclici*. Un esempio di configurazione tramite questo parametro è esposto di seguito.

```
1 self.FUNC_APP_GENERATION = "nx.gn_graph(random.randint(2,7))"
```

  I valori 2 e 7, indicano il range del possibile numero di servizi per quella specifica applicazione.
- `FUNC_APP_DEADLINES`: distribuzione delle *deadline* delle applicazioni. Questo specifica la priorità di una determinata applicazione in fase di esecuzione.
- `FUNC_SERVICE_RESOURCES`: distribuzione delle risorse richieste dai servizi.

- `FUNC_SERVICEMESSAGESIZE`: distribuzione delle dimensioni dei messaggi scambiati tra i servizi.
- `FUNC_REQUESTPROB`: distribuzione di probabilità di richieste per le applicazioni.
- `FUNC_USERREQRAT`: distribuzione del numero di richieste emesse dagli utenti per le applicazioni durante la simulazione.

Analisi sull'Algoritmo di Service Placement

Il software è stato realizzato in modo che fosse possibile valutare l'algoritmo di placement utilizzato, operando sui parametri del file `runExperiment.py`. In modo automatico, vengono generati

$$\frac{\texttt{VALUE\_TO} - \texttt{VALUE\_FROM}}{\texttt{STEP}}$$

scenari al variare del parametro `PARAM_TO_CHANGE`.

Analisi dei Risultati

Nel caso in cui si intenda valutare le caratteristiche di un particolare scenario, dopo aver eseguito la simulazione, è possibile ottenere un sommario di quest'ultima. Questo è ottenibile dal file `generateSimulationSummary.py`. Il sommario generato è un file `pdf` contenente le seguenti informazioni:

- descrizione dello scenario simulato;
- elenco dei parametri di configurazione dello scenario;
- valutazione della distribuzione dell'utilizzo delle risorse globali;
- valutazione dell'utilizzo percentuale di risorse per ogni livello della topologia;
- valutazione del successo dell'allocazione dei servizi per ogni livello della topologia;

- valutazione del soddisfacimento delle richieste con e senza *failure control* dei nodi.

3.2 Implementazione del Simulatore

In questa sezione vengono approfonditi i principali aspetti dell'implementazione del software realizzato, come l'algoritmo di service placement, la generazione della topologia e delle simulazioni.

3.2.1 Algoritmo di Service Placement

Per ottenere il massimo beneficio dall'implementazione di un'architettura Fog, è necessario un efficace algoritmo di service placement. In generale questi algoritmi sono improntati a massimizzare il *Quality of Service*[1] (QoS) o il bilanciamento del carico, oppure a minimizzare il consumo di energia, la latenza o il costo della comunicazione.

In questo lavoro di Tesi, l'algoritmo implementato (in: `placeService.py`) è fondato su due aspetti principali: preservare la *privacy* dei dati scambiati tra i servizi e far si che le applicazioni siano disponibili per gli utenti il più velocemente possibile. L'algoritmo utilizza un approccio *greedy*[2] valutando i diversi aspetti dei servizi, come privacy (la sensibilità dei dati), risorse necessarie e *deadlines* delle applicazioni. La privacy dei servizi è relativa al loro livello di piazzamento: con livelli di privacy crescenti, i servizi potranno essere piazzati solo in livelli via via più bassi, ovvero più vicini al livello IoT. La tendenza dell'algoritmo è quella di allocare le applicazioni il più vicino possibile a questo livello, così da minimizzare la latenza.

[1] Con *Quality of Service* si intende l'insieme dei valori che indicano la qualità del servizio offerto dalla rete, in termini di throughput, gestione degli errori, gestione dei ritardi e utilizzo della banda.

[2] L'approccio *greedy*, come paradigma negli algoritmi, indica la costruzione della soluzione ottima scegliendo, ad ogni iterazione, la via che offre il maggiore vantaggio in quello specifico stadio.

Il primo passo che viene compiuto è l'ordinamento delle applicazioni secondo un ordine crescente sulla base del livello di privacy dei servizi che la compongono. In particolare viene utilizzato il seguente approccio:

$$\text{privacy}(APP_x) \leq \text{privacy}(APP_y)$$
$$\text{se e soltanto se}$$
$$\min_{S_u \in APP_x} \text{privacy}(S_u) \leq \min_{S_v \in APP_y} \text{privacy}(S_v)$$

dove S_u e S_v sono i servizi delle singole applicazioni. Nel caso particolare in cui due applicazioni abbiamo lo stesso valore di privacy, allora vegnono ordinate secondo la loro *deadline*, in ordine crescente.

Una volta eseguito l'ordinamento delle applicazioni, l'algoritmo di placement comincia ad allocare i servizi iniziando dalla prima applicazione della lista. L'algoritmo tende a posizionare i servizi nei primi nodi disponibili cominciando da quelli più vicini agli utenti (livello IoT). Nel caso di insufficienza di risorse e se il livello di privacy lo permette, i servizi vengono allocati nel Cloud. Una applicazione viene considerata "allocata" se e solo se lo sono tutti i suoi servizi, altrimenti viene scartata.

Le applicazioni che l'algoritmo di *service placement* tenta di allocare sono generate nella configurazione dell'esperimento.

3.2.2 Configurazione dell'Esperimento

La configurazione dell'esperimento (in: `config/experimentConfiguration.py`) gestisce la generazione della topologia di rete, delle applicazioni e degli utenti, da intendersi come i dispositivi che generano le richieste dei servizi.

Generazione della Topologia

Come accennato nel capitolo 2, il simulatore *YAFS* utilizza la libreria *NetworkX* per l'analisi e l'utilizzo dei grafi. La stessa libreria Python è stata dunque utilizzata anche per la generazione della rete.

Per semplicità è stato introdotto un livello di *gateway*, il cui scopo è quello di fungere da intermediario tra i dispositivi IoT e la rete. I nodi di questo livello sono generati come mostrato di seguito.

```
H.add_nodes_from([(i, {"level[z]": level}) for i in range(iot_nodes // 5)])
```

La divisione del numero di nodi IoT (`iot_nodes`) per 5, indica che il numero dei *gateway* accennati sopra saranno 1/5 del numero di nodi IoT.

Per i livelli successivi viene utilizzato, come descritto nel paragrafo 2.2.2, il modello *Watts-Strogatz*:

```
# Number of nodes of upper levels
new_nodes = iot_nodes // (fogi_reduction_factor ** (level-1)) if iot_nodes
    ↪ // (fogi_reduction_factor**(level-1)) >= 2 else 2

# Small-World graph generation for FOG levels greater than 0 (e.g provincial
    ↪  fog nodes)
H = nx.watts_strogatz_graph(int(new_nodes), 2, hub_prob)
```

Ognuno dei livelli della rete è generato come un grafo isolato `H`. Dopo che questo è stato generato, viene unito al grafo globale `G` (che viene restituito dalla funzione di generazione della topologia), come componente non connessa. Una volta che tutti i livelli sono stati generati, vengono creati i collegamenti tra i singoli nodi appartenenti a livelli differenti, come mostrato nel Listato 3.1, unendo le varie componenti non connesse.

```
# Connection between levels
for level in range(levels-1):

  connection_to_upper_level = 0 # counter of connection to upper levels

  for node_from in ranges[level]["nodes"]:

    # extralevel_threshold indicates the % of node in fog levels > 1 that
    ↪ are connected to upper levels
    extralevel_threshold = 100 #all the node can be conected to upper level
    if level > 1 and random.randrange(100) > extralevel_threshold:
      # If there aren't connection to upper levels a link is generated
      if node_from == ranges[level]["nodes"][-1:] and
    ↪ connection_to_upper_level == 0:
        G.add_edge(node_from, random.choice(ranges[level+1]["highest_degrees
    ↪ "]))

```

```
15        continue
16
17      connection_to_upper_level += 1
18
19      ## chance that nodes are connected to non-touching upper levels
20      for r in range(random.randrange(min_conn_to_up, max_conn_to_up)):
21        rnd_level_to = level+1
22        while random.randrange(100) < 20:
23          rnd_level_to += 1
24          if(rnd_level_to >= levels):
25            rnd_level_to = level+1
26            break
27
28        G.add_edge(node_from, random.choice(ranges[rnd_level_to]["
      ↪ highest_degrees" if level > 0 else "nodes"]))
```

Listato 3.1: Generazione dei collegamenti tra i nodi appartenenti a livelli diversi.

Impostazione della Simulazione

Dall'esecuzione delle funzioni `networkGeneration()`, `appGeneration()` e `userGeneration()` di `experimentConfiguration.py`, vengono generati tre file JSON, utilizzati in fase di inizializzazione della simulazione:

- `appDefinition.json`: contiene la definizione delle applicazioni, i servizi che la compongono, i messaggi e le relative trasmissioni;
- `networkDefinition.json`: contiene la definizione della rete, ovvero dei nodi e dei collegamenti tra essi;
- `usersDefinition.json`: contiene la definizione dei nodi che generano il *workload*.

Dall'esecuzione dell'algoritmo di placement, viene inoltre generato il file `allocDefinition.json`, utilizzato in fase di inizializzazione per definire la posizione dei servizi all'interno della rete.

Dai file JSON indicati sopra, la simulazione viene inizializzata in `runSimulation.py` come mostrato nel Listato 3.2.

```
# TOPOLOGY
t = Topology()
dataNetwork = json.load(open(path_json+'networkDefinition.json'))
t.load(dataNetwork)
t.write("network.gexf")

# APPLICATION
dataApp = json.load(open(path_json+'appDefinition.json'))
apps = create_applications_from_json(dataApp)
#for app in apps:
# print apps[app]

#PLACEMENT algorithm
placementJson = json.load(open(path_json+'allocDefinition%s.json'%
    ↪ specificSuffix))
placement = JSONPlacement(name="Placement",json=placementJson)


# POPULATION algorithm
dataPopulation = json.load(open(path_json+'usersDefinition.json'))
pop = JSONPopulation(name="Statical", json=dataPopulation, iteration=it)
```

Listato 3.2: Configurazione della simulazione da file JSON.

Una volta eseguita la prima fase di inizializzazione, il software avvia due simulazioni: la differenza tra le due è che la prima viene eseguita senza *failure control* dei nodi. La doppia esecuzione è necessaria per valutare l'andamento del numero di richieste soddisfatte per i servizi durante l'esecuzione della simulazione. Nel Listato 3.3 è mostrata la fase di creazione della simulazione con il *failure control* (la creazione dell'altra simulazione è pressoché identica):

```
stop_time = simulated_time
s_f = Sim(t, default_results_path=resultspath + "Results_RND_FAIL__%i_%i" %
    ↪ (stop_time,it))

dynamicFail = True
if dynamicFail:
  time_shift = 10000
  distribution = deterministicDistributionStartPoint(name="Deterministic",
    ↪ time=time_shift,start=10000)
  failurefilelog = open(path_json+"Failure_%s_%i.csv" % (specificSuffix,
    ↪ stop_time),"w")
  failurefilelog.write("node, module, time\n")
  random.seed(time.time())
  rng = np.random.default_rng()
```

```
12    randomValues = rng.choice(170, size=170, replace=False)
13
14    s_f.deploy_monitor("Failure Generation", failureControl, distribution,sim=
        ↪ s,filelog=failurefilelog,ids=randomValues)
15
16  #For each deployment the user - population have to contain only its specific
      ↪  sources
17  for aName in apps.keys():
18    print ("Deploying app: ",aName)
19    pop_app = JSONPopulation(name="Statical_%s" % aName, json={}, iteration=it
        ↪ )
20    data = []
21    for element in pop.data["sources"]:
22      if element['app'] == aName:
23        data.append(element)
24    pop_app.data["sources"]=data
25
26    s_f.deploy_app2(apps[aName], placement, pop_app, selectorPath)
27
28  print("Simulation WITH failures starting...")
29  s_f.run(stop_time, test_initial_deploy=False, show_progress_monitor=False)
```

Listato 3.3: Creazione della simulazione.

Come accennato nel Capitolo 2, il simulatore YAFS basa le simulazioni sulla libreria SimPy, il cui vantaggio è quello di poter avviare politiche dinamiche, come appunto il *failure control* dei nodi, tramite la funzione `deploy_monitor()`. Nel Listato 3.3, a riga 7, viene definita la distribuzione temporale con la quale viene richiamata la politica di *failure control*.

Capitolo 4

Risultati

4.1 Simulazioni

Qui elenco delle varie simulazioni con spiegazione dei risultati ed eventuali conclusioni che da essi si ottengono.

Conclusioni

Appendice A

Appendice

Bibliografia

[1] P. Mell and T. Grance. The nist definition of cloud computing. *NIST Special Pubblication 800-145*, 2011.

[2] OpenFog. IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing. *IEEE Std 1934-2018*, pages 1–176, 2018.

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[4] Pedro Neves, Bradley Schmerl, Javier Cámara, and Jorge Bernardino. Big Data in Cloud Computing: Features and Issues. pages 307–314, 01 2016.

[5] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal*, 4(5):1125–1142, 2017.

[6] Andrei Furda and Ljubo Vlacic. Real-Time Decision Making for Autonomous City Vehicles. *Journal of Robotics and Mechatronics*, 22:694, 08 2010.

[7] Jordi Garcia, Ester Simó, Xavier Masip-Bruin, Eva Marín-Tordera, and Sergi Sànchez-López. Do We Really Need Cloud? Estimating the Fog Computing Capacities in the City of Barcelona. In *IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 290–295, 2018.

[8] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[9] F. Giust. MEC deployments in 4G and evolution towards 5G. *MEC Deployments in 4G and Evolution Towards 5G*, 2018.

[10] Jürgo S. Preden, Kalle Tammemäe, Axel Jantsch, Mairo Leier, Andri Riid, and Emine Calis. The Benefits of Self-Awareness and Attention in Fog and Mist Computing. *Computer*, 48(7):37–45, 2015.

[11] Manas Kumar Yogi, K. Chandra sekhar, and G. Vijay Kumar. Mist Computing: Principles, Trends and Future Direction. *International Journal of Computer Science and Engineering*, 4(7):19–21, Jul 2017.

[12] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[13] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. EdgeCloudSim: An environment for performance evaluation of Edge Computing systems. In *Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 39–44, 2017.

[14] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. EmuFog: Extensible and scalable emulation of

large-scale fog computing infrastructures. In *IEEE Fog World Congress (FWC)*, pages 1–6, 2017.

[15] Antonio Coutinho, Fabiola Greve, Cassio Prazeres, and Joao Cardoso. Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing. In *IEEE International Conference on Communications (ICC)*, pages 1–7, 2018.

[16] Isaac Lera, Carlos Guerrero, and Carlos Juiz. YAFS: A Simulator for IoT Scenarios in Fog Computing. *IEEE Access*, 7:91745–91758, 2019.

[17] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx.

[18] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. In *5th International Conference on the Internet of Things (IOT)*, pages 155–162, 2015.

[19] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[20] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, Jun 1998.

[21] Qiang Wu, Jun Shen, Binbin Yong, Jianqing Wu, Fucun Li, Jinqiang Wang, and Qingguo Zhou. Smart fog based workflow for traffic control networks. *Future Generation Computer Systems*, 97:825–835, 2019.

[22] NIST. Mobile cloud computing. [online]. *Technical Report, National Institute of Standards and Technology.* `https://www.nist.gov/programs-projects/mobile-cloud-computing`.

[23] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile edge computing, Fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680–698, 2018.